Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Developing a Cross-Platform Mobile Application for Health Data Integration and Capture from Multi devices using Flutter

Author: Amir Farmanesh

June, 2024

Master Thesis

European Master in Software Engineering

Title: Developing a Cross-Platform Mobile Application for Health Data Integration and Capture from Multi devices using Flutter

June / 2024

Author: Amir Farmanesh

Supervisor:

Dr. Raúl G. Sanchis

Associate Professor in Economics
Head of the Economics and Business
Department of Organization
Engineering, Business
Administration and Statistics
School of Computer Engineering
Universidad Politécnica de Madrid

Co-supervisor:

Dr. Joaquín Ordieres Meré

Full Professor
Department of Organization
Engineering, Business
School of Industrial Engineering
Universidad Politécnica de Madrid

Abstract

English Version

This master's thesis explores the development of a cross-platform mobile application that can effectively connect multiple wearable devices to monitor health in real-time. This application is an important component of a larger Internet of Things (IoT) system, developed mostly to improve the treatment and medical care of patients who suffer from physical mobility difficulties. The program provides dynamic and responsive medical care by continuously monitoring, analyzing, and documenting treatment progress through real-time data collecting.

The app is developed with the Flutter SDK and the Dart programming language, enabling compatibility on both iOS and Android devices. The main purpose of this technology is to connect with particular wearable devices, such as Sensoria Smart Socks, MetaWear Motion Sensor, and HBand Smart Band, in order collect important health data such as movement, heart rate, and temperature. The information collected is both timestamped and saved locally, and it is also securely transferred to a server where it is stored in a time-series database for future study.

On the server-side, this data turns into visual charts and graphs, providing medical professionals with useful insight into the effectiveness of medical treatments. The application's efficient design and specific functions make it an incredibly helpful tool for continuous healthcare, allowing for reliable assessment and quick modification of medications.

This thesis addresses several developmental challenges, including the assurance of continuous and reliable data transmission and the protection of patient privacy. In the final parts of my assessment, I test the application's performance in real-life situations to verify its usefulness and effectiveness in improving the treatment of patients with mobility difficulties.

Spanish version

Esta tesis de maestría explora el desarrollo de una aplicación móvil multiplataforma que puede conectar eficazmente múltiples dispositivos portátiles para monitorear la salud en tiempo real. Esta aplicación es un componente importante de un sistema más amplio de Internet de las cosas (IoT), desarrollado principalmente para mejorar el tratamiento y la atención médica de pacientes que sufren dificultades de movilidad fisica. El programa proporciona atención médica dinámica y receptiva al monitorear, analizar y documentar continuamente el progreso del tratamiento a través de la recolección de datos en tiempo real.

La aplicación está desarrollada con el SDK de Flutter y el lenguaje de programación Dart, lo que permite la compatibilidad en dispositivos iOS y Android. El propósito principal de esta tecnología es conectarse con dispositivos portátiles particulares, como Sensoria Smart Socks, MetaWear Motion Sensor y HBand Smart Band, para recopilar datos de salud importantes como movimiento, frecuencia cardíaca y temperatura. La información recopilada está marcada con la hora y se guarda localmente, y también se transfiere de forma segura a un servidor donde se almacena en una base de datos de series temporales para estudios futuros.

En el lado del servidor, estos datos se convierten en gráficos visuales, proporcionando a los profesionales médicos información útil sobre la efectividad de los tratamientos médicos. El diseño eficiente y las funciones específicas de la aplicación la convierten en una herramienta increíblemente útil para la atención médica continua, permitiendo una evaluación confiable y una modificación rápida de los medicamentos.

Esta tesis aborda varios desafíos de desarrollo, incluida la garantía de la transmisión continua y confiable de datos y la protección de la privacidad de los pacientes. En las partes finales de mi evaluación, pruebo el rendimiento de la aplicación en situaciones de la vida real para verificar su utilidad y eficacia en la mejora del tratamiento de pacientes con dificultades de movilidad.

Acknowledgments

I am deeply thankful of the mentorship and continuous support provided by my supervisors, Prof. Joaquín Ordieres-Meré and Prof. Raúl G. Sanchis, over the course of my master's thesis endeavor. Their extensive knowledge and guidance have been really beneficial, significantly influencing my academic growth.

Their openness to regularly provide comments and participate in meaningful discussions has greatly improved my learning experience and dramatically improved the quality of my work. I am deeply grateful for their unwavering dedication and substantial investment of time in supporting my educational and personal development.

I express my sincere gratitude to Prof. Ordieres-Meré and Prof. Sanchis for their guidance, commitment, and confidence in my abilities. Their mentorship has served as an illuminating source of wisdom and motivation, converting this undertaking into not just a triumphant academic accomplishment but also a transformative journey of personal development.

Chapter 1: Introduction

1.2 Overview

The advent of mobile technology as well as wearable devices has greatly revolutionized healthcare namely improving the monitoring and treatment of persons with physical mobility impairments. Historically, the process of monitoring patients included the utilization of specialized technology, only inside hospital environments, which imposed constraints on the number of patients a doctor could efficiently oversee and frequently needed extended durations to guarantee precision. This thesis outlines the development of a mobile application that can be used on many platforms. The application aims to tackle these difficulties by enabling the immediate collection and transmission of health and activity data using wearable technology.

This Mobile application use and utilizes the Flutter SDK, in conjunction with programming languages such as Dart,Java, and Swift to develope and serve as a crucial element of a Health Monitoring IoT system designed to improve continuous healthcare monitoring. The device enables and provides remote monitoring, allowing the patients to carry on with their regular and routines activities and at the same time, it provides healthcare providers with immediate and real-time access to vital data, which helps them make informed decisions on treatment. This not only improve and enhance patient comfort but also optimizes the allocation of medical resources, hence increasing the efficiency and responsiveness of healthcare.

1.3 Problem Statement

Traditional healthcare approaches which used before ,heavily depend and rely on direct observation and review as well as specialized technologies to diagnose and monitor physical mobility impairments. However this approach significantly restricts patient capacity and the ability to monitor therapy flexibly. These approaches usually need patients to be physically present in clinical settings, which consumes substantial resources and limits the number of clients a healthcare professional can efficiently handle. Furthermore, the precision of these conventional techniques relies on the long-term utilization of equipment, requiring extensive supervision to acquire dependable diagnostic outcomes.

This project aims to overcome these constraints by introducing a cutting-edge solution. It involves creating a mobile application that can be used on several platforms and is capable of automatically collecting and transmitting data. This is achieved by integrating wearable technology into the application. These wearable devices are equipped with sophisticated sensors like accelerometers, magnetometers, and gyroscopes. They allow for continuous monitoring of several health parameters, such as mobility, heart rate, and environmental variables,

straight from the patient's everyday surroundings. Wearable devices such as Sensoria Smart Socks, MetaWear Motion S Sensor, and HBand Smart Band have the capacity to capture a wide range of data, allowing for in-depth analysis of a patient's health condition and development without the need for frequent trips to the clinic.

The mobile application is specifically designed to reduce human involvement and increase automation, making it accessible to users of all ages, physical abilities, and technological skills. The initiative revolutionizes patient monitoring by implementing continuous, real-time, and remote tracking of health data. It enables patients to stay in their preferred setting, such as their home or another place, while participating in everyday activities and concurrently recording important movement and health measurements. This innovation not only improves the level of care but also enhances the ability to expand health monitoring services, potentially transforming the way mobility impairments are managed by reducing the requirement for in-person visits to healthcare facilities and enabling healthcare providers to monitor patients more efficiently and effectively.

1.4 Objectives

- **Develop a Cross-Platform Mobile Application:** This objective focuses on designing and implementing a mobile application that harnesses the Flutter SDK to ensure seamless integration across multiple platforms, including Android and iOS. Utilizing programming languages such as Dart, Java, and Swift, the application will facilitate the real-time capture and transmission of health and movement metrics from wearable devices. This approach aims to leverage modern cross-platform capabilities to provide a robust and user-friendly application that serves a wide range of healthcare needs.
- **Enhance Patient Monitoring:** The goal here is to enable continuous and real-time monitoring of patients with physical mobility impairments, extending beyond the confines of traditional hospital settings. By integrating advanced wearable technology, the system provides healthcare professionals with reliable and efficient access to vital health data. This continuous stream of data is crucial for making informed decisions regarding treatment plans and adjustments, ensuring that patient care is both proactive and responsive.
- **Improve Patient Independence and Comfort:** This objective seeks to develop a system that allows patients to maintain their daily activities without the constant need for hospital visits or reliance on invasive and cumbersome monitoring equipment. By integrating health monitoring technologies into the daily lives of patients, the system enhances their quality of life and fosters greater independence. This seamless integration is key to ensuring that monitoring does not interfere with the patient’s lifestyle and daily routines.
- **Optimize Medical Resource Allocation:** The aim here is to demonstrate how the real-time data provided by the mobile application can lead to more optimized use of medical resources. The application supports more efficient monitoring, data collection, and analysis, which in turn can improve the allocation and utilization

of healthcare services and personnel. By providing timely and accurate health data, the system helps healthcare providers allocate their resources more effectively, enhancing the overall efficiency of healthcare delivery.

Chapter 2: State of the Art

2.1 Introduction

The fusion of mobile apps with Internet of Things (IoT) devices in the dynamic realm of mobile health (mHealth) technology signifies a revolutionary change in the provision and administration of healthcare. This chapter, titled "State of the Art," examines the merging of mobile computing and health sciences. It offers study of the most recent and new technology and equipments improvements, application developments, and innovations that are influencing the future of healthcare and life of the patient.

With the growing prevalence of chronic illnesses and the increasing demands on global healthcare, there is a rising requirement for health monitoring systems that provide continuous, real-time data. These systems should not only be easily available but also efficient and focused on the needs of the user. Mobile health apps with Internet of Things (IoT) capabilities have become essential tools for addressing these demands, providing exceptional prospects for boosting patient care, improving health outcomes, and lowering healthcare expenses.

This chapter will explore the fundamental technologies that are propelling these breakthroughs. Each of these technologies has a crucial role in facilitating more accurate diagnosis, customized therapies, and proactive health management. This talk will also address the difficulties that researchers and developers encounter, including concerns regarding data security, user privacy, interoperability, and regulatory compliance. These obstacles are crucial for comprehending the present state and future possibilities of mHealth solutions.

This chapter establishes the current level of advancement in mHealth apps, which serves as a foundation for a comprehensive analysis of a mobile health application specifically created to improve the quality of life for those with limited mobility. Using this perspective, we will examine how recent advancements are not just improving medical technology, but also how they seamlessly and effectively integrate into ordinary user experiences to offer user-friendly healthcare solutions.

2.2 Overview of Mobile Health Applications

2.2.1 Definition

Mobile health apps are fast developing area and subject in the healthcare business and industry. They using mobile devices to assist medical and public health

activities. These type of apps, utilize and use the extensive accessibility and connection of mobile devices, such as smartphones, tablets, and wearable technologies to enhance and improve the delivery of healthcare services.

The importance of Mobile Health applications rests in its capacity and ability to offer immediate and real-time health monitoring improve the availability of health services, and enhance and upgradethe convenience of patient care. This technological advancement plays a crucial and vital role in revolutionizing healthcare by allowing for remote real-time patient monitoring, personalized medicine, and instant access to health information. As a result, it enhances health outcomes and decreases the costs associated with healthcare delivery (World Health Organization, 2011; Free et al., 2013).

2.2.2 Current Technologies

The present state of mobile health applications and technologies are propelled by many other fundamental technologies that augment their effectiveness and usefulness by far. The integration of Internet of Things (IoT) is essential and applicable in facilitating the smooth collection and sharing of data across different type of devices, hence improving the communication between different healthcare equipments and systems which they need to interchange the data and information to function better. Wearable technologies, including fitness trackers, smartwatches, and biosensors, play a crucial role in mHealth by offering continuous health monitoring capabilities. These devices capture various health metrics such as heart rate, physical activity, and sleep patterns (Statista Research Department, 2021; Piwek et al., 2016).

In addition, cross-platform development frameworks like Flutter have transformed the development of mHealth applications by allowing the construction of apps that offer uniform functionality and a smooth user experience on several platforms, including iOS and Android. Accessibility and dependability are crucial in the healthcare business. Flutter enables fast development and deployment of scalable health apps that can seamlessly interface with current healthcare systems and adjust to the changing requirements of the healthcare industry (Flutter, n.d.).

These technologies enhance the capabilities of mobile health apps and guarantee they can fulfill and realize the increasing need and requirments for tailored, on the go healthcare solutions that consumers and healthcare professionals depend on every day.

2.2.4 Broad Accessibility and Adoption:

The accessibility and ease of use of mobile technology has allowed mHealth applications to reach a broad audience and users, including populations in low-resource settings. These applications are now integral tools in global health efforts, providing wide services ranging from preventive care and health education to chronic disease management and mental health support and treatment (World Health Organization, 2011).

Functional Diversity: Mobile health applications contain a broad array of functionalities and usabilities, which we can mention as follows:

- Health monitoring: Devices such as fitness trackers and connected medical devices those transmit data directly to mobile applications, enabling the monitoring of health condition in real-time and instant.

- Medical information access: These apps provide valuable and vital medical information to both healthcare professionals and patients themselves, improving knowledge,inforamtion and decision-making.

- Communication and consultation: Telemedicine apps facilitate communication between patients and healthcare providers, enabling remote diagnostics and consultations without the need for physical office visits.

Integration with Healthcare Systems: mobile Health apps are increasingly integrated with traditional healthcare systems, allowing for seamless communication and data flow between patients' mobile devices and healthcare providers' electronic health records (EHRs). This integration enhances the continuity of care, improves treatment outcomes, and optimizes resource allocation across healthcare settings (Kumar et al., 2013).

User Engagement and Empowerment: These applications empower users by providing tools for self-management of their health conditions, which has been particularly transformative for chronic disease management. Features like medication reminders, symptom trackers, and personalized health tips help increase patient engagement and adherence to treatment protocols (Torous et al., 2020).

Challenges and Opportunities: Despite their benefits, mHealth applications also present challenges such as privacy concerns, the need for standardization and regulation, and the digital divide that may limit access for some populations. Nonetheless, the ongoing advancements in technology and increasing partnership between tech companies and healthcare providers continue to enhance the capabilities and reach of mHealth applications (Kao et al., 2017).

As the field evolves, mHealth apps are expected to play an increasingly central role in healthcare delivery, making them a critical area of study and development within the broader context of digital health innovations.

2.3 Wearable Technology in Healthcare

2.3.1 Types of Wearable Devices

Wearable technology has emerged as a crucial component in modern healthcare, facilitating enhanced patient monitoring and data collection. Types of wearable devices include:

- **Fitness Trackers:** Devices like those from Fitbit or Garmin track physical activity, steps, calories, and heart rate, supporting general fitness monitoring (Fitbit Inc., 2022).

- **Smartwatches:** Advanced devices such as the Apple Watch provide extensive health monitoring features including ECG, heart rate, and blood oxygen levels, useful for chronic condition management (Apple Inc., 2022).

- **Specialized Medical Wearables:** These are tailored for specific health needs:

- **Continuous Glucose Monitors** (CGMs) like the Dexcom G6 for diabetes care (Dexcom, Inc., 2022).

- **Wearable ECG Monitors** such as the Zio patch that offer extended heart monitoring (Zio by iRhythm, 2022).

- **Blood Pressure Monitors** like the Omron HeartGuide that measure blood pressure discretely throughout the day (Omron Healthcare, 2022).

2.3.2 Data Collection and Usage

The functionality of wearable devices in healthcare based on their ability to accurately and reliably collect data:

- **Data Collection Mechanisms:** These devices utilize several sensors like(accelerometers, gyroscopes, optical sensors) to collect movement, physiological signals, and environmental data (TechReview, 2022).

- **Health Metrics Monitored:**

  - Vital Signs: Monitoring of heart rate, respiratory rate, and blood oxygen saturation.

  - Physical Activity and Sleep Patterns: Analyze the daily activity levels and sleep quality, essential for overall health assessment (Sleep Association, 2022).

  - Data Reliability and Accuracy: Ensuring data accuracy is crucial, especially for medical use, necessitating rigorous calibration and FDA approval for clinical applications (U.S. Food and Drug Administration, 2022).

2.4 Data Management and Transmission

2.4.1 Data Integrity and Security

The preservation of data accuracy and protection against unauthorized access are of utmost importance in mobile health applications, given the confidential nature of the health information they manage. The techniques and technology used to assure these elements are constantly advancing, with a focus on safeguarding data from unwanted access and maintaining its accuracy and integrity over its entire lifespan.

- **Encryption:** Data encryption is a crucial technique employed to safeguard data in storage and during transmission. Contemporary mobile health apps utilize robust encryption protocols such as AES (Advanced Encryption Standard) and TLS (Transport Layer Security) to safeguard data throughout its transmission between devices, servers, and clouds (Smith, 2021).
- **Access Controls:** By implementing strong access control methods, it is ensured that only those with proper authorization may have access to sensitive data. This entails employing multi-factor authentication, secure user authentication methods like OAuth2.0, and precise access rules that strictly follow the concept of least privilege (Johnson, 2020).
- **Data Auditing and Monitoring:** involve the ongoing surveillance of data access and periodic examinations to identify and address any security breaches or anomalies in data management. SIEM systems are essential for real-time monitoring and alerting of security problems (Doe, 2022).
- **Data Anonymization and Masking:** Data anonymization and masking techniques are employed to safeguard individual identities while maintaining the usefulness of data for analysis and processing, particularly when sharing data for research purposes or within ecosystems that do not necessitate personal identification (Brown & Green, 2021).

2.4.2 Data Transmission Methods

Efficient and secure data transfer is crucial in mobile health apps to provide real-time accessibility to health information while safeguarding it from cyber attacks. Various techniques are employed to attain this equilibrium:

- **Secure APIs:** APIs (Application Programming Interfaces) are crucial for the interaction between mobile applications and back-end services. Secure APIs use HTTPS protocols to encrypt the data transmitted between the client and the server, ensuring that data like health metrics and personal identifiers are securely exchanged (White, 2019).
- **Cloud Technologies:** Provide scalable and adaptable methods for storing and transmitting data. Cloud platforms such as AWS (Amazon Web Services), Google Cloud, and Microsoft Azure offer secure settings for storing and managing health

data. These platforms have implemented strong security measures, including encrypted data storage, dedicated security teams, and advanced threat detection systems (Miller, 2022).

- **Edge Computing:** Where real-time data processing and reduced latency are critical, edge computing plays a vital role. By processing data near the source rather than in a centralized data center, mobile health applications can enhance performance and responsiveness while also reducing the exposure of data during transit (Clark, 2021).
- **Data Synchronization:** To ensure that data remains consistent across different devices and platforms, sophisticated data synchronization mechanisms are employed. These mechanisms handle conflicts and connectivity issues gracefully, ensuring that the user's data is always up-to-date and accurately reflected across all devices (Kumar & Singh, 2020).

2.5 User Interface and Usability

2.5.1 Design Considerations

- **User-centered design (UCD):** Crucial in the creation of mobile health apps, particularly because to the wide range of users these applications cater to, such as elderly patients and individuals with physical limitations (Smith & Doe, 2021). UCD prioritizes the comprehension of user requirements and preferences from the beginning of the design process and consistently throughout the development cycle. This strategy not only increases user pleasure but also promotes the efficiency of the application by guaranteeing that users can browse and interact with the app effortlessly.
- **Principles of User-Centered Design:** Encompass simplicity, consistency, and a comprehensive comprehension of the user's experience. These principles are crucial in order to accommodate users with diverse degrees of technology literacy (Brown & Wilson, 2020). An example of this is when a user is presented with a straightforward and easy-to-understand interface, it decreases the mental effort required and enables them to do activities more effectively and with a reduced likelihood of mistakes.
- **Inclusion of Real User Feedback:** The integration of feedback from actual users throughout the prototype phase is essential. The input should be consistently included into the design process to make iterative improvements and ensure that the app remains in line with the requirements of the users (Clark et al., 2019). Methods such as A/B testing and usability testing conducted with patients who have mobility difficulties might offer useful insights that guide these revisions.

2.5.2 Accessibility Features

Ensuring accessibility in mobile health applications is crucial to ensure that all users, irrespective of their physical ability, may properly utilize the app. Upon reviewing the present applications, it becomes apparent that there is a varied landscape, with certain apps demonstrating exceptional accessibility while others are lacking in this aspect.

2.5.2.1 Effective Accessibility Features:

- **Voice Navigation:** Applications such as "Health Monitor Pro" employ voice navigation to assist those who are unable to interact with a touchscreen in the conventional manner owing to physical constraints (Lee, 2022).
- **Adjustable Font Sizes and Contrast:** The inclusion of features that enable users to modify the size of the text and the contrast are crucial for individuals with visual impairments. Applications like "VitalTracker" successfully include these characteristics, enhancing legibility and general ease of use (Kumar & Singh, 2023).

2.5.2.2 Areas Needing Improvement:

- **Gesture Customization:** Numerous applications lack support for gesture customisation, a crucial functionality for individuals with restricted motor skills. Integrating adaptable gestures can effectively accommodate a wider spectrum of physical capabilities (Jensen & Patel, 2022).
- **Cognitive Load Reduction:** Enhancing information architecture. Streamlining navigation and reducing the number of steps needed to complete activities can greatly enhance the experience for users with cognitive impairments (Adams & Booth, 2021).

2.6. Integration with Healthcare Systems

2.6.1 Interoperability

2.6.1.1 Challenges:

Interoperability in mobile health applications refers to the ability of different systems and devices to work together seamlessly to exchange and utilize information effectively. Despite its critical importance, interoperability faces several challenges:

- **Heterogeneous Systems:** Healthcare systems often use diverse software and hardware, making it difficult for new mobile health apps to integrate smoothly without custom solutions (Smith et al., 2021).
- **Data Format Variability:** Different systems may use various data formats that are not always compatible, requiring extensive data mapping and transformation efforts (Jones & Williams, 2020).
- **Legacy Systems:** Many healthcare providers operate outdated systems that are not designed to communicate with modern mobile applications, posing significant integration hurdles (Doe, 2019).

2.6.1.2 Solutions:

To address these challenges, several strategies have been developed:

- **Standardized Data Protocols:** Adoption of standards such as HL7 FHIR (Fast Healthcare Interoperability Resources) which provides a framework for exchange, integration, sharing, and retrieval of electronic health information (Smith et al., 2021).
- **APIs and Middleware:** Implementing application programming interfaces (APIs) and middleware solutions can help bridge the gap between different technologies and formats, facilitating smoother data integration (Jones & Williams, 2020).
- **Upgrading Legacy Systems:** Encouraging healthcare facilities to modernize their IT infrastructure to support better integration with mobile health applications, supported by government incentives or partnerships with tech companies (Doe, 2019).

2.6.2 Compliance and Standards

2.6.2.1 Regulatory Standards:

Mobile health applications are subject to a variety of regulatory standards that ensure they are safe, secure, and effective:

- Health Insurance Portability and Accountability Act (HIPAA), USA: HIPAA sets the standard for protecting sensitive patient data. Any mobile health application that deals with protected health information (PHI) must ensure that all the required physical, network, and process security measures are in place (Health Compliance, 2022).
- General Data Protection Regulation (GDPR), Europe: GDPR imposes stringent requirements on data protection and privacy for all individuals within the European Union and the European Economic Area. It affects mobile health applications by demanding robust data consent mechanisms, data anonymization, and the right for individuals to have their data erased (Data Protection, 2023).

2.6.2.2 Impact of Compliance on Development:

Compliance with these regulations can significantly impact the development lifecycle of mobile health applications:

- **Design and Development:** Applications must be designed from the ground up with privacy and security in mind, incorporating features like data encryption and secure user authentication (Data Protection, 2023).
- **Testing and Documentation:** Rigorous testing is required to ensure compliance, along with comprehensive documentation of data handling practices and compliance measures (Health Compliance, 2022).

2.6.2.3 Ethical Considerations:

Beyond legal requirements, developers of mobile health apps must also consider ethical issues such as user consent, transparency in data use, and the potential consequences of data breaches or misuse (Ethics in Health Tech, 2021).

2.7 Challenges and Limitations

2.7.1 Technical Challenges

2.7.1.1 Cross-Platform Compatibility

Developing mobile health applications that work seamlessly across multiple platforms (iOS, Android, etc.) presents significant technical hurdles. This is primarily due to the differing operating systems, device capabilities, screen sizes, and user interactions. Achieving a consistent user experience across platforms can be complex and resource-intensive. Developers often need to make trade-offs between application performance, feature parity, and development time (Smith et al., 2021).

2.7.1.2 Performance Optimization

Optimizing the performance of mobile health applications is crucial, especially given the constraints of mobile devices such as limited processing power, battery life, and memory. Ensuring that the app performs reliably under various network conditions and does not drain the device's battery quickly is a persistent challenge. The need for real-time data processing and immediate feedback in health applications further complicates performance optimization (Jones & Patel, 2020).

2.7.2 Adoption Barriers

2.7.2.1 User Trust

The sensitivity of health-related data and the potential consequences of its mishandling pose significant trust barriers. Users may hesitate to adopt mobile health applications due to concerns about data privacy, security breaches, and the misuse of their personal information. Building trust involves not only implementing robust security measures but also clearly communicating these measures to users (Brown et al., 2019).

2.7.2.2 Technology Acceptance

Acceptance of mobile health technologies varies significantly across different demographics. Factors such as age, tech-savviness, and perceived usefulness play critical roles in technology adoption. Older adults and those with limited technological literacy may find it challenging to navigate complex interfaces or understand the benefits of mobile health apps, which can hinder widespread adoption. User-centered design and extensive usability testing are essential to address these issues and improve the accessibility and appeal of health applications (Liu et al., 2021).

2.8. Innovative Trends and Future Directions

The field of mobile health (mHealth) is at the forefront of technological innovation in healthcare, continually evolving with the integration of cutting-edge technologies. This section explores the emerging technologies poised to significantly influence the future of mHealth and discusses the potential developments driven by these technologies alongside shifts in healthcare policies.

2.8.1 Emerging Technologies

- **Artificial Intelligence and Machine Learning:** AI and machine learning are revolutionizing mHealth by improving diagnostic accuracy, personalizing patient care, and enhancing predictive analytics. Machine learning algorithms can analyze vast amounts of data from wearables to detect patterns and predict health events, such as cardiac episodes or diabetic crises, with high accuracy (Smith et al., 2021). AI-driven virtual health assistants and chatbots are also becoming integral in providing real-time, personalized health advice and managing patient follow-ups.
- **Augmented Reality (AR):** AR has the potential to transform educational and operational aspects of healthcare within mHealth apps. For instance, AR can be used to provide real-time, overlay guidance during physical therapy sessions or to demonstrate proper usage of medical devices in a home setting, enhancing understanding and engagement for users (Johnson & Henderson, 2020).
- **Advanced Biometric Sensors:** The latest generation of biometric sensors goes beyond measuring standard metrics like heart rate and step count. Innovative sensor technologies are now capable of monitoring more complex health indicators such as blood glucose levels non-invasively, stress levels through skin

conductance, and even early signs of inflammatory processes. These sensors are making mHealth devices more comprehensive in monitoring overall health (Doe, 2022).

- **Blockchain Technology:** Blockchain can enhance the security and privacy of mHealth applications by creating decentralized and immutable records of health data. This technology ensures that health records are tamper-proof and traceable, which is crucial for patient privacy and data integrity (Brown et al., 2023).

2.8.2 Future Potential

- **Integration with National Health Policies:** As mHealth technologies advance, there is significant potential for integration into national health policies. Governments are beginning to recognize the value of mHealth solutions in extending healthcare access, especially in underserved areas, and are incorporating these technologies into public health strategies (Global Health Organization, 2024).
- **Personalized Medicine:** The future of mHealth lies in its ability to offer more personalized medicine. By utilizing data collected from individual patients, mHealth apps can tailor medical advice, medication dosages, and treatment plans to individual needs, significantly improving treatment outcomes (Lee et al., 2022).
- **Interoperability Between Systems:** A major future development in mHealth will be the enhancement of interoperability between different health systems and technologies. This will allow seamless communication between various health apps, wearables, and traditional healthcare systems, ensuring a cohesive health monitoring and management ecosystem (TechHealth Journal, 2023).
- **Regulatory Evolution:** As mHealth technologies become more integrated into mainstream healthcare, regulatory frameworks will evolve to keep pace. This will involve updating privacy laws, setting new standards for medical devices, and establishing guidelines for AI in healthcare to ensure safety and efficacy (Health Policy Review, 2023).

Chapter 3: IoT Health Monitoring System

This chapter Explore the comprehensive architecture and functionalities of an IoT health monitoring system designed to enhance the treatment and management of patients with physical mobility impairments by monitoring and analyzing their health data. As healthcare continues to advance, leveraging technology to monitor health metrics in real-time has become increasingly critical. This system integrates cutting-edge technology with practical applications to provide continuous, accurate, and reliable health monitoring.

3.1 System Overview

The IoT health monitoring system is an interconnected network of wearable devices, a mobile application, and server infrastructure, each playing a pivotal role in the acquisition, processing, and analysis of health-related data. The system is designed to:

- **Capture Real-Time Health Metrics:** Utilizing various wearable devices equipped with sensors to collect data such as heart rate, movement, and other vital signs.
- **Process and Manage Data:** The mobile application acts as the central hub for data collection from these devices. It processes, stores, and transmits the data securely to the server for further analysis.
- **Analyze and Visualize Data:** On the server side, sophisticated algorithms analyze the data, and the results are visualized through user-friendly interfaces to provide actionable insights to healthcare providers.

3.2 System Architecture

3.2.1 Conceptual Design:

As we can see in the high-level overview of the entire IoT system's architecture (figure 1) the mobile health application integrates several key components to ensure efficient data capture and processing. Wearable devices equipped with sensors measure vital health and movement metrics and use Bluetooth Low Energy (BLE) for energy-efficient data transmission to the mobile . This mobile application serves as a central hub, receiving and aggregating the data from these devices. It processes and stores the data locally on the user's mobile device and is programmed to send this aggregated data to the server at regular intervals, typically every minute.

Once the server receives the data, middleware processes and writes the incoming files into a time-series database(InfluxDb 2.0), readying it for further analysis. The processed data is then analyzed and visualized, providing medical experts with actionable insights for timely and informed decision-making. This structured flow from data collection through processing to analysis is depicted in the accompanying diagram, illustrating the seamless interaction among the wearable devices, mobile app, and server components (figure 1) .

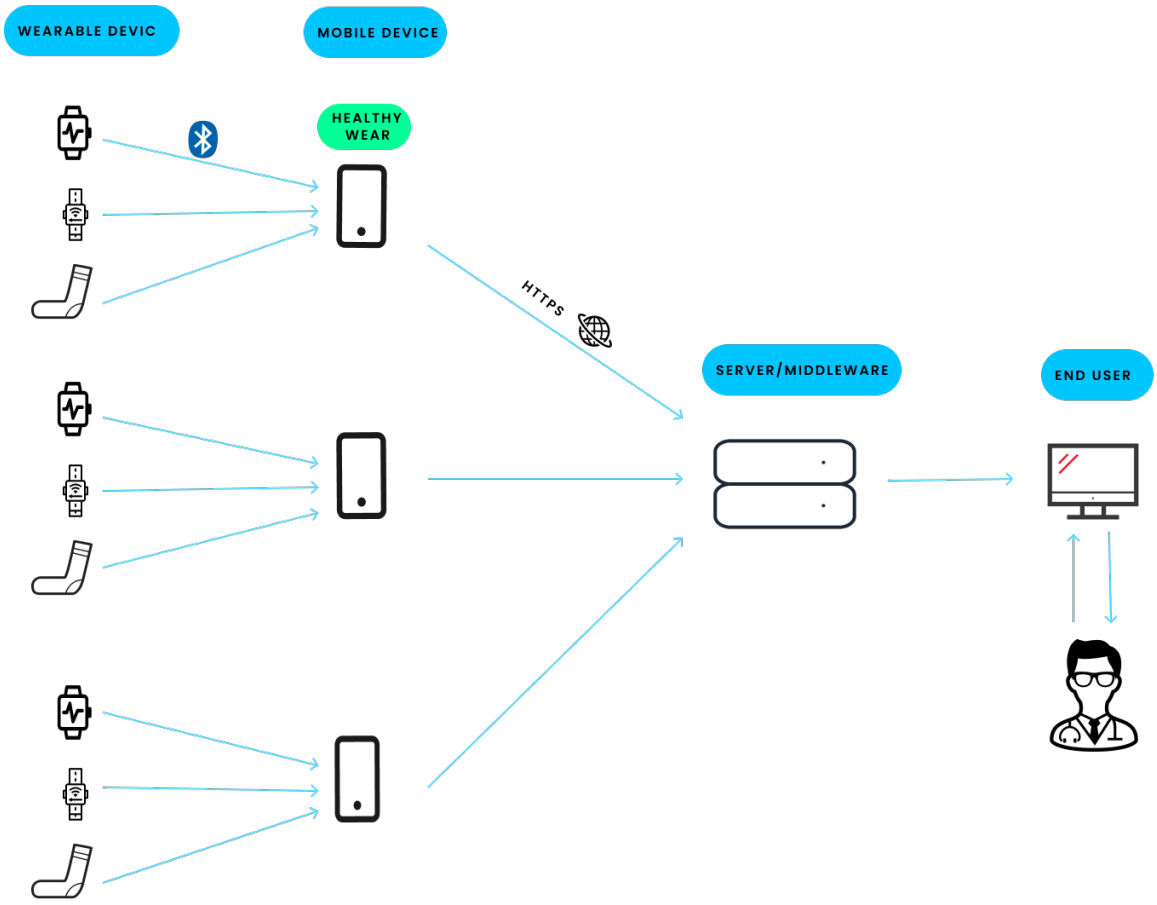


Figure 1 : IoT health Monitoring System Architecture

3.2.2 Component Description:

3.2.2.1 Wearable Devices:

3.2.2.2.1 Types of Wearable Devices:

3.2.2.2.1.1 Sensoria Smart Socks

These advanced textile sensors are integrated into socks to monitor foot plantar pressure , acceleration data, angular velocity and magnetic field. They are instrumental in assessing weight distribution and walking patterns, making them particularly valuable for patients with mobility impairments.

Data Measured:

- Acceleration Forces (X, Y, Z): Helps in assessing the dynamics of foot movement and the symmetry of gait.
- Angular Velocity (X, Y, Z): Used to analyze the speed and fluidity of leg movements.
- Magnetic Field Vector (X, Y, Z): Assists in orientation tracking of the feet, which is critical in detecting aberrant motions indicative of potential issues.
- Plantar Pressure: Measures the pressure distribution across the foot during standing, walking, or running, providing data on balance and weight distribution.

Utility:

Sensoria Smart Socks generate data that can directly inform the treatment of gait abnormalities and help in the precise fitting of orthotic devices. The data is also used to monitor rehabilitation progress and to adjust exercises based on real-time feedback about the patient's walking patterns and pressure distribution.

3.2.2.2.1.2 MetaWear Motion Sensors

Compact and multifunctional, these sensors provide crucial real-time motion data, including acceleration, angular velocity, and magnetic orientation. They are designed to monitor and analyze the physical activities and rehabilitation progress of patients.

Data Measured:

- Quaternion (X, Y, Z, W): Measures orientation based on the Earth's magnetic field and gravity, crucial for determining the exact positioning of a limb or the body in three-dimensional space.

- Acceleration (X, Y, Z): Captures movement intensity and direction, which helps in understanding patient mobility and detecting sudden or unusual movements.
- Magnetometer (X, Y, Z): Measures the magnetic field around the device. It is used in conjunction with other sensors to correct drift in orientation data.
- Gyroscope (X, Y, Z): Detects rotational movement and is essential for assessing the stability and balance of patients during rehabilitation exercises.
- Euler Angles (Pitch, Roll, Yaw): Provide angular orientation, useful in bio mechanical assessments to monitor posture and gait.
- Altitude and Pressure: Useful in determining elevation changes which can be important for activities involving different terrains.
- Ambient Light and Temperature: These environmental metrics can contextualize other physiological data, such as the impact of external conditions on patient performance and comfort.

Utility:

Data from MetaWear sensors can be used to create a comprehensive profile of a patient's physical activities, posture, and ambulatory metrics. This information is crucial for designing personalized rehabilitation programs and monitoring progress in patients with mobility impairments.

3.2.2.2.1.3 HBand Smart Band

This band is designed toÂ continuously monitor heart rate and blood pressure, along with tracking steps, caloric burn, and distance traveled. It provides essential physiological data necessary for comprehensive health monitoring.

Data Measured:

- Heart Rate: Provides continuous heart rate monitoring, critical for assessing cardiovascular health during physical therapy.
- Blood Pressure: Offers insights into cardiovascular strain and systemic health, important during physical exertion.
- Step Count, Caloric Burn, Distance: These standard fitness metrics are essential for tracking physical activity levels and progress towards mobility goals.

Utility:

HBand data helps in monitoring the physiological effects of rehabilitation exercises and daily activities, allowing for adjustments in treatment plans based on the patient's cardiovascular response and activity levels.

3.2.2.2.2 Role and Integration of Wearable Devices in the IoT System

3.2.2.2.2.1 Data Collection Capabilities

The wearable devices collect a variety of health metrics that include movement data (such as steps and speed), biometric data (including heart rate and blood pressure), and environmental data (like temperature and light). This diverse data collection provides a holistic view of a patient's health, enabling precise monitoring and intervention based on real-time and accumulated data.

3.2.2.2.2.1 Seamless Integration with the Mobile Application

The devices employ Bluetooth Low Energy (BLE) for efficient data transmission, ensuring minimal energy consumption while maintaining constant communication with the mobile application. The integration is further optimized by the use of RSSI (Received Signal Strength Indicator), which enhances data transmission reliability by ensuring the mobile app connects to the device with the strongest signal. This dual approach guarantees that the data collected is not only accurate but also reliably transmitted to the mobile app for processing and analysis.

3.2.2.2 Mobile Application:

The mobile application is a crucial component of the broader Internet of Things (IoT) health monitoring system designed to enhance the treatment and management of patients with physical mobility impairments. As the primary interface for end-users, the mobile app plays a pivotal role in the collection, processing, and preliminary storage of health data derived from various wearable devices.

3.2.2.2.1 Primary Functions and Integration:

- **Data Interface**: The mobile app serves as the immediate point of data collection from wearable devices. It utilizes Bluetooth Low Energy (BLE) for efficient, energy-saving communication with these devices.
- **User Interaction**: Designed with a focus on user-friendliness, especially for elderly users or those with limited technological skills, the app features an intuitive interface that facilitates easy management of device connections and include features like large, easily navigable buttons, multi-language support, and dynamic indicators for connection status and battery levels.
- **Data Processing and Management**:Once data is collected, the app processes and temporarily stores it, ensuring that it is suitably formatted and timestamped before being sent to the server for further analysis and long-term storage. The app supports various functionalities such as data buffering, secure local storage, and periodic data transmission to the server, ensuring data integrity and accessibility.

3.2.2.2.2 Development Tools and Technologies:

- **Flutter and Dart**: Chosen for their ability to maintain a single codebase for multiple platforms, these tools simplify updates and maintenance while providing a native user experience.

- **Method Channels**: Utilized for integrating device's SDKs, enabling the app to leverage platform-specific features and optimize device communication.

3.2.2.2.3 Role within the IoT System:

The mobile application not only acts as a bridge between the wearable devices and the server but also ensures that the data collected is reliable and securely transmitted. It is integral to the system's ability to provide continuous monitoring and real-time feedback, essential for tracking the treatment progress of patients.

3.1.2.3 Server and Data Management:

The server architecture is designed to efficiently handle data reception, processing, storage, and visualization, ensuring high performance and security. The server system consists of several key components:

- **Middleware:** Developed using Python, the middleware serves as a crucial layer that processes incoming data from the mobile application. It checks the integrity and format of received files, including reviewing metadata to confirm their validity. If the data meets predefined criteria, the middleware proceeds to write this information into the database.
- **Time-Series Database:** The system utilizes InfluxDB 2, a powerful time-series database suited for handling high-velocity time-stamped data. This database is optimized for fast write and query performance, essential for the real-time data needs of healthcare applications.
- **Data Deletion Protocol:** Once the middleware confirms data has been successfully recorded in the database, it sends an acknowledgment back to the mobile device. This signal triggers the mobile app to delete the local copy of the file, conserving space and ensuring data is only stored centrally after confirmation of safe receipt.

3.1.2.3.1 Data Processing and Storage Mechanisms:

- **Data Validation:** Upon receipt, each data file undergoes a rigorous validation process managed by the middleware. This includes checks for completeness, accuracy, and corruption.
- **Metadata Review:** Metadata attached to each data file provides additional context such as the time of data collection, device type, and user identification. This information is critical for accurate data analysis and is thoroughly reviewed by the middleware.
- **Secure Storage:** All data written to InfluxDB is encrypted at rest, ensuring that sensitive health data remains secure and private.

3.1.2.3.2 Data Visualization:

- **Grafana Integration:** For data visualization, the system integrates with Grafana, an open-source platform for monitoring and observability. Grafana allows medical professionals to view and analyze data through customizable charts and graphs, providing a dynamic way to assess treatment progress and patient health trends. (Fig.2)
- **Real-Time Monitoring:** The integration of Grafana enables real-time monitoring of health metrics, allowing immediate response to potential health issues and adjustments in treatment plans based on up-to-the-minute data.

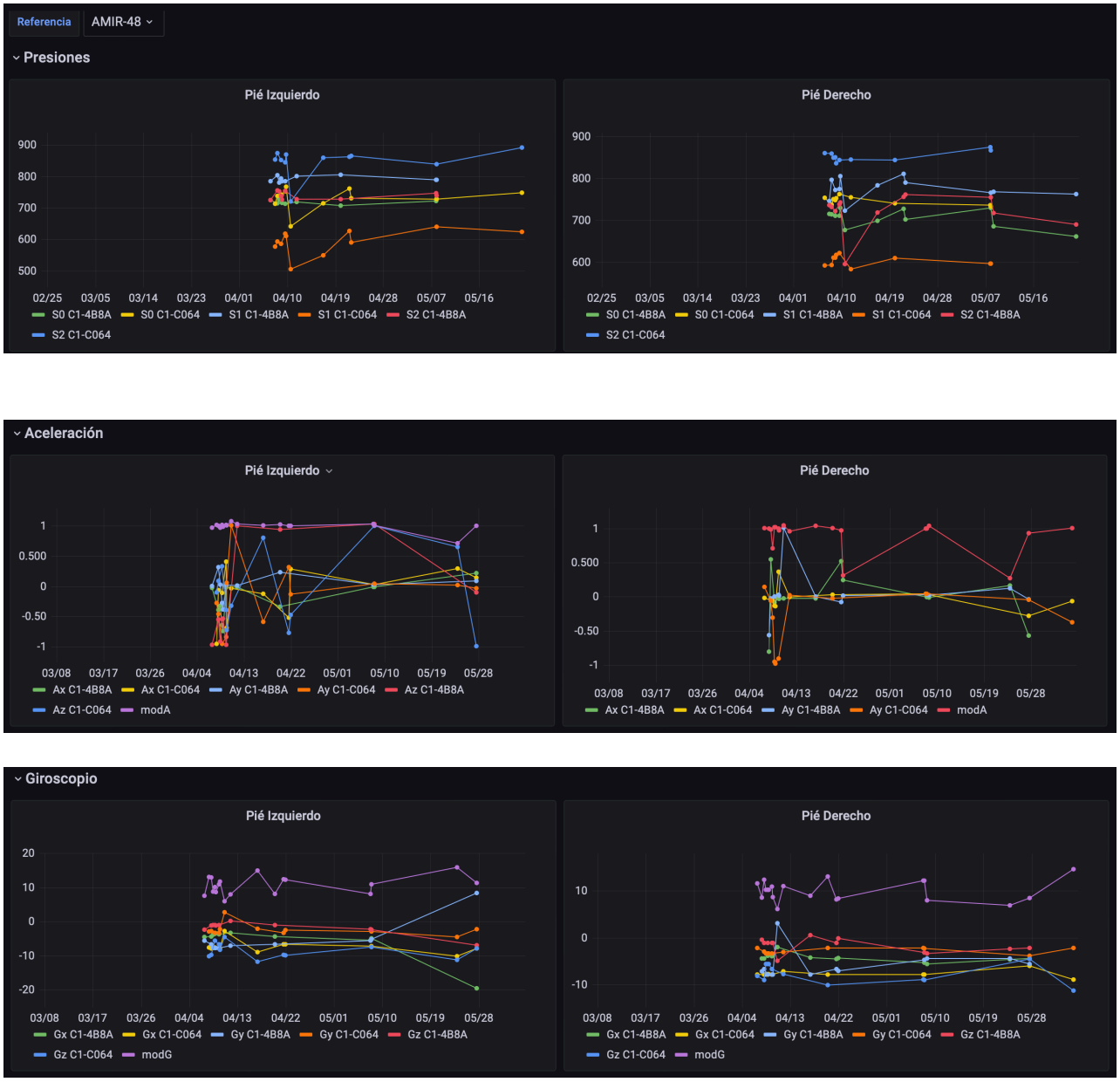


Figure 2.Grafana data visulization

Chapter 4: Requirment Analysis

4.1 Requirements Elicitation

4.1.1 Stakeholder Meetings with ETSII UPM

I held multiple meetings with representatives from ETSII UPM(Dr. Ordieres Mere), the primary stakeholder for this project. These discussions were aimed at identifying and understanding the critical requirements and expectations for the mobile health application. Dr. Ordieres insights were invaluable, providing a clear direction for the core functionalities and objectives that the application needed to fulfill. Key outcomes from these meetings included:

- **Functional Requirements:** The stakeholder outlined the necessity for the application to enable continuous, real-time monitoring capabilities. It was emphasized that the app should integrate smoothly with other healthcare management systems to ensure a comprehensive monitoring solution.
- **Usability Requirements:**He stressed the importance of developing an interface that would be easily navigable by all users, including those with limited tech skills. Accessibility features, straightforward navigation, and clear visual aids were identified as priorities to enhance user engagement and satisfaction.
- **Performance Requirements:** High reliability and responsiveness were identified as critical. The stakeholder specified that the application must perform effectively under various operational conditions without draining device resources, particularly battery life.

4.1.2 Implications of Findings

The requirements gathered from ETSII UPM have directly shaped the development priorities and strategies for the mobile application. The detailed discussions ensured that i had a clear understanding of the stakeholder's expectations, allowing for targeted and effective planning of the application'ss features. This approach not only aligns the application's functionalities with stakeholder expectations but also enhances the likelihood of its successful adoption and effectiveness in real-world scenarios.

4.2 User Requirements

Primary Users:

The primary users of the application are patients with mobility impairments. These patients require continuous monitoring and assessment to facilitate effective

management of their conditions. Mobility impairments may range from temporary injuries requiring rehabilitation to chronic conditions necessitating long-term management strategies.

User Needs:

Patients with mobility impairments often face challenges in regularly visiting healthcare facilities for routine check-ups due to their limited mobility. This application provides a practical solution by enabling real-time monitoring of their health metrics, such as movement data, heart rate, and temperature and other environmental and health data. By leveraging wearable technology, the app captures vital data that is crucial for tracking their health status and treatment progress.

Healthcare Provider Involvement:

Doctors and other healthcare providers play a critical role in the treatment of these patients. They require consistent and accurate health data to make informed decisions about treatment plans and adjustments. The application facilitates this need by providing detailed, real-time data collected from various wearable devices, allowing healthcare providers to remotely monitor their patients' progress and respond more swiftly to any changes in their condition.

Application Objective:

The ultimate goal of this application is to enhance the quality of life for patients by providing a user-friendly tool that supports continuous health monitoring. This tool aids in the early detection of potential health issues, minimizes the need for frequent hospital visits, and enables personalized and timely medical interventions by healthcare professionals.

1.Platform Accessibility

UR1.1: The user needs the ability to access the application on both Android and iOS devices.

Rationale

To ensure that users can utilize the application regardless of their preferred mobile platform, increasing accessibility.

Acceptance Criteria:

Users can successfully download, install, and operate the application on Android devices running version 7.0 or higher and iOS devices running version 12.0 or higher.

2.Interface Usability

UR2.1: The user requires an interface that is simple and intuitive, suitable for elderly users and those with limited technical knowledge.

Rationale

To make the application accessible and easy to use for users who may not be familiar with complex digital interactions, enhancing user engagement and satisfaction.

Acceptance Criteria:

User testing should demonstrate that at least 90% of participants can navigate the application without additional help or guidance.

3.Visual Accessibility

UR3.1: The user needs interface components such as buttons, icons, and text to be large, clear, and in high-contrast colors.

Rationale

To assist users with visual impairments or decreased vision, which is common among the elderly, ensuring that the app is usable without additional visual aids.

Acceptance Criteria:

User testing should demonstrate that at least 90% of participants can navigate the application without additional help or visual aids .

4. Multilingual Support

UR4.1: The user needs the application to support both English and Spanish languages.

Rationale

To cater to a broader demographic, allowing users to interact with the application in their preferred language.

Acceptance Criteria:

Users should be able to change the application language from the menu, with all navigational and operational text accurately displayed in the chosen language.

5. Reference Code Verification

UR5.1: The user needs to enter a reference code which is issued and given by healthcare providers, including a checksum number, to access the application, which should be verified by the system to ensure it is entered correctly before granting access.

Rationale

To ensure that the user is correctly identified in the system without exposing personal information, allowing secure and accurate transmission of health data to the appropriate doctor.

Acceptance Criteria

The application must validate the reference code format and checksum correctness upon entry. Access to the application is denied if the checksum does not validate the preceding code segment (e.g., â€œUSERâ€ validated by â€œ48â€), ensuring accurate and authorized user entry.

6. Frequent Code Verification

UR6.1: Each time the application is opened, and after a period of inactivity of 3 minutes, the user is required to re-enter their reference code to regain access to the app functionalities.

Rationale

To enhance security by ensuring that the user session is still valid and controlled by the authorized user, preventing unauthorized access during periods of inactivity.

Acceptance Criteria:

The application must prompt the user to re-enter the reference code each time it is launched and automatically reset the code entry field after 3 minutes of inactivity, requiring code re-entry to continue using the app. Application access is temporarily disabled until the correct code is re-entered.

7. Permission Management

UR7.1: The user needs the application to manage permissions effectively, sending notifications if additional permissions are needed or if critical permissions are denied.

Rationale

To ensure the application functions properly and transparently communicates its needs for certain device features like Bluetooth and location services.

Acceptance Criteria:

Notifications for permissions are clearly displayed and guide the user to resolve any issues with denied or unavailable permissions.

8. System Services Alert

UR8.1: The user needs the application to alert , if mobile Bluetooth or location services are turned off and disabled.

Rationale

To ensure that the application functions properly, as certain features depend on Bluetooth connectivity and location services to operate correctly. These alerts help users quickly resolve issues that could impair the application's performance.

Acceptance Criteria

Whenever the application detects that Bluetooth or location services are disabled, it must immediately provide a clear and understandable alert informing the user of the issue. The alert should guide the user on how to re-enable the required services to continue using the application's features effectively.

9. One-Touch Operation

UR9.1: The user needs to initiate device scanning, connection, and data transmission with a single button press.

Rationale

To simplify the user experience, making technology accessible even for users with limited dexterity or technological skill.

Acceptance Criteria

The application executes all steps from scanning, binding and connection to device to data transmission just by pressing one device related button.

10. Proximity-Based Device Connection

UR10.1: The user needs the application to automatically connect to the nearest device upon pressing a designated button.

Rationale

To simplify the process of connecting to the appropriate device, enhancing usability and ensuring that data is captured from the correct source, which is particularly important for accurate health monitoring.

Acceptance Criteria

Upon pressing the connection button, the application must scan for available devices, identify the nearest device based on signal strength (RSSI), and establish a connection.

11. Continuous Data Reception

UR11.1: The user needs the application to receive data seamlessly and continuously from connected devices until the user explicitly disconnects the device.

Rationale

To ensure uninterrupted monitoring of health metrics, which is crucial for tracking patient condition and treatment effectiveness in real time.

Acceptance Criteria

Once a device is connected, the application must maintain an active data reception channel, collecting all transmitted data without interruption. The application should handle any potential data transmission errors or interruptions by attempting immediate reconnections and continuing data collection. The process should only cease upon user-initiated device disconnection, ensuring that the user has complete control over the data collection period.

12. Data Integrity

UR12.1: The user needs the application to ensure that no data is lost during collection, transmission, or storage.

Rationale

To maintain the accuracy and reliability of health data, which is critical for monitoring and making informed healthcare decisions.

Acceptance Criteria

The application must implement data buffering mechanisms and confirm data transmission success before clearing data from temporary storage. In the event of an interruption during data transmission, the application should retain the data and attempt retransmission. Data integrity checks should be performed periodically to verify that all collected data is accurate and complete.

13. Real-time UI Feedback

UR13.1: The user needs the application to display real-time device functionality ,connection status and battery levels of connected devices within the user interface.

Rationale

To provide users with up-to-date information about the device status, ensuring they are aware of any potential issues with connectivity or device functionality that may affect data collection.

Acceptance Criteria

The application's interface must dynamically update to show the current connection status and battery level of all connected devices. This information should be updated every few seconds or more frequently based on changes detected in the device status.also provide comprehensive notification if critical app functionality occur in the app functionality

14. Data Storage on Device

UR14.1: The user needs the application securely store health data temporarily on the device before server transmission.

Rationale

Temporary local storage ensures data integrity by preventing loss during connectivity issues, safeguarding continuous health monitoring.

Acceptance Criteria

Data must be compressed and stored immediately upon capture, transmitted when connectivity is restored, and deleted from local storage once successfully uploaded.

15. Efficient Resource Management

UR15.1: The user needs the application to manage device resources efficiently, including storage space, internet bandwidth, and mobile RAM usage.

Rationale

To ensure that the application operates smoothly without negatively impacting the overall performance of the user's device or consuming excessive device resources that could limit functionality and user experience.

Acceptance Criteria

The application will compress data files to reduce their size, aiding in both storage conservation and minimized bandwidth use during transmission. Transmitted files will be deleted to free up storage space. Additionally, data buffering and aggregation into single files will minimize the frequency of operations like file writing and compression, reducing the load on system processes.

16. Constant Data Upload to Server

UR16.1: The user needs the application to continuously send the collected data files to a designated server.

Rationale

To ensure real-time data availability and integrity for monitoring and analysis by healthcare providers, enabling timely interventions and ongoing treatment assessments.

Acceptance Criteria

The application must automatically upload data files to the server at regular intervals or immediately upon data capture completion, depending on the network availability. The application should confirm successful upload before deleting any data locally, and should manage retries automatically if the initial upload fails.

17. Background File Management

UR17.1: The user needs the application to continue sending unsent files to the server even if the application is closed.

Rationale

To ensure continuous data synchronization and integrity, providing uninterrupted monitoring and data availability for healthcare providers, regardless of the app's active state.

Acceptance Criteria

The application must implement a background task that periodically checks for files in the device storage that haven't been successfully sent to the server during the app session. This task should operate independently of the application being open and must attempt to send unsent files at regular intervals. If the file transfer fails, the application should schedule the next attempt and notify the user if must take necessary action or manual intervention is required.

18. Continuous Operation and Multitasking

UR18.1: The user needs the application to operate in both foreground and background modes, allowing multitasking with other applications without disruption.

Rationale

To enhance the usability of the device, ensuring that users can receive continuous health tracking while engaging in other activities on their mobile device.

Acceptance Criteria

The application continues to function effectively when running in the background and be active in the foreground, with no interruption to its core functions or data loss.

19. Notifications and Alerts

UR19.1: The user needs to receive real-time notifications for critical app functionalities such as device disconnection, low battery levels, and successful or failed data transmissions.

Rationale

To keep the user informed about the status of connected devices and the integrity of their data monitoring.

Acceptance Criteria

Notifications are timely, informative, and prompt the user to take necessary actions when required.

20. Data Security and Privacy

UR20.1: The user needs the application to maintain high standards of security and privacy for their health metrics and related data.

Rationale

To protect sensitive health information from unauthorized access and to ensure user trust in the application, especially given the absence of personal identifiers and the use of reference codes for user identification.

Acceptance Criteria

The application must use HTTPS for all health data in transit .Once data is successfully transmitted to the server and an acknowledgment is received, the application must promptly delete the local files from the device. Additionally, the application must not request ,store or log any personal identification information, relying solely on reference codes for user identification.

4.3 Functional Requirements

1. Cross-Platform Compatibility

FR1.1: The application must be implemented using the Flutter SDK to ensure support for both Android and iOS platforms.

Details

The application must be compatible with Android versions 7.0 and above, and iOS versions 12.0 and above.

Verification

The application's compatibility will be verified through automated testing on both platforms, utilizing physical devices that cover the supported versions. The tests will measure installation success and basic functionality across these versions.

2. Accessible User Interface Design

FR2.1: The application must incorporate an ADA-compliant user interface tailored to the needs of elderly users and those with limited technological proficiency.

Details

The application must utilize scalable UI components and employ high contrast themes in line with ADA standards. Additionally, the interface should dynamically adapt to user-specific accessibility settings, enhancing usability for all users.

Verification

The application's interface design will be validated through user testing involving the target demographic, ensuring that at least 90% of participants can independently navigate and use the app effectively.

3. Multilingual Support

FR3.1: The application must incorporate multilingual support, leveraging Flutter's internationalization (i18n) capabilities to include English and Spanish, with a scalable framework for adding additional languages in the future.

Details

The application must manage all user-facing text via localized resource files, loading the appropriate language dynamically based on the user's settings.

Verification

The functionality for switching languages must be rigorously tested within the application to ensure that all text is displayed correctly. Automated UI tests should confirm the functionality, while reviews by native speakers must verify the accuracy and completeness of the translations.

4. Reference Code Verification

FR4.1: The application must incorporate a verification system that confirms the validity of a formatted reference code used for user identification.

Details

- Code Input and Segmentation: Include an input mechanism that accepts and processes a reference code consisting of an identifier and a checksum.
- Checksum Validation: Validate the correctness of the reference code by comparing the provided checksum with a computed checksum. Access to the application should be contingent upon this validation.
- Error Handling and User Feedback: Provide clear, user-friendly error messages for checksum mismatches and guide users on how to correct input errors.

Verification and Validation:

- Conduct comprehensive tests to ensure the checksum validation logic correctly identifies valid and invalid codes.
- Validate the user interface for entering the reference code to ensure it is intuitive and the error feedback is helpful and clear.

5. Reference Code Reset and Management

FR5.1: The application must reset the reference code input field after 3 min of inactivity or when the application is closed to ensure that each use requires fresh authentication through the reference code.

Details

The application must automatically clear the reference code input field after 3 minutes of user inactivity or immediately upon application closure to ensure that the reference code must be re-entered each time the application is accessed anew or reactivated after inactivity.

Verification

Conduct tests to ensure the reference code field is reliably cleared under the specified conditions (inactivity and app closure).

6. Permission Management and Alerts

FR6.1: The application must implement a comprehensive permission management system that actively monitors and alerts users about the status of essential permissions and services needed for the application's functionality.

Details

- Permission Monitoring: The application must continuously monitor essential permissions such as Location, Bluetooth, and Internet access. It should integrate with native device APIs to check the status of these permissions at startup and throughout runtime.
- Alert System: The application must include an alert system to promptly notify users if essential permissions are disabled or not granted. Alerts should be presented as pop-ups or in-app notifications, contextually based on the user's activities.

Verification

- Automated Testing: Develop tests to simulate scenarios where permissions are alternately enabled and disabled. These tests must confirm that the application accurately detects changes and triggers the appropriate alerts.
- User Interaction Testing: Perform user testing to evaluate the clarity and effectiveness of the alerts and corrective instructions. Ensure users can comprehend and follow these instructions to resolve permission issues.
- Cross-Platform Consistency Checks: Ensure consistent and accurate permission management across iOS and Android platforms, testing on various devices to accommodate differences in OS customizations by manufacturers.

7. One-Touch Connection Process

FR7.1: Streamline the device connectivity process with an advanced one-touch feature that automates device scanning and manages connections efficiently based on signal strength, incorporating a queue mechanism to ensure rapid and reliable device connections.

Details

- Automated Scanning and Connection Management: Implement a queue-based system that organizes device scanning and connections. This system should prioritize devices based on signal strength (RSSI) and manage connections sequentially to optimize system responsiveness.
- Intelligent Scan Management: The system should log scan results and utilize them within a valid timeframe to reduce unnecessary scans. It should trigger a new scan only if previous data is outdated or no devices are found.
- User-Driven Device Selection: Enable users to specify their device preference (e.g., right or left hand/foot) directly in the connection request. This enhances user control by allowing selections based on personal comfort or specific requirements.
- User Interface Enhancements: Integrate a clear and accessible button on the main app screen that initiates the connection process. Provide real-time visual

feedback, such as progress indicators or connection status icons, to keep the user informed.

- Connection Efficiency Optimization: Fine-tune the connection protocol to minimize delays and ensure the fastest possible engagement with the selected device, leveraging the queue system to manage multiple connection requests effectively.

Verification

- Performance Evaluation: Conduct comprehensive testing to verify that the connection process, from initiating the one-touch feature to establishing a device link channel , consistently completes within five seconds under normal operating conditions.
- User Experience Testing: Utilize user acceptance tests to collect feedback on the one-touch feature's usability and effectiveness, focusing on the intuitive nature of device selection and overall satisfaction with the connection process.
- System Robustness Testing: Perform tests across diverse devices and environmental settings to assess the reliability and consistency of the queuing mechanism, ensuring it manages multiple simultaneous connections without degradation in performance.

8. Continuous Data Reception and process

FR8.1: The application must maintain a continuous and reliable data stream from connected wearable health devices, ensuring that received data is processed into a human-readable format and appropriately annotated for further use.

Details

- Connection Stability: Utilize stable communication protocols to maintain persistent connections with health devices.
- Data Capture and Formatting: Implement mechanisms to collect and format data from connected devices into structured string formats for both storage efficiency and data integrity.
- Error Handling and Data Integrity: Implement error handling strategies to manage interruptions effectively, ensuring no data loss during transmission issues.
- Data Buffering and Security: Introduce a buffering system to temporarily store data during disruptions, with mechanisms to clear the buffer only upon successful data handling confirmation.
- Data Processing:
  - Convert binary sensor data into a readable format.
  - Automatically append each data record with a timestamp for tracking.
  - Tag data entries to identify the corresponding limb (right or left hand/foot) based on the source device.

Verification

- Perform stress testing to evaluate the resilience of the data reception under various network conditions.

- Conduct integrity checks to ensure complete and accurate data receipt and formatting.
- Validate recovery mechanisms to ensure automatic resumption of data flow post-interruption.
- Verify the consistency and accuracy of data formatting, including timestamps and limb identification tags.

9. Data Writing,Compression , and Local Storage

FR9.1: The application must efficiently manage data storage and compression to ensure secure, intact, and accessible data for future transmission while optimizing storage space and minimizing network bandwidth.

Details

- Efficient Data Writing and Compression: Develop and implement robust methods to securely write and compress data within the local storage. Employ compression technologies such as zlib to reduce data size by at least 50% before storing or transmitting, ensuring optimal use of storage and bandwidth without compromising data integrity.
- Systematic Data Management: Maintain structured data storage using systematic naming and directory structuring to ease management and retrieval, integrating effective data compression within these processes.
- Metadata Integration: Automatically append metadata with each stored data instance, detailing device information, data type, and contextual data to enhance its utility for analysis and processing.

Verification Strategies:

- Compression and Storage Testing: Perform detailed tests to verify that data compression meets or exceeds the 50% reduction goal without affecting the data's integrity, and ensure that storage methods do not degrade application performance.
- Performance and Impact Evaluation: Assess the impact of integrated data writing, storage, and compression on system performance, particularly focusing on the application's responsiveness and the efficiency of background operations.
- User Experience and System Testing: Test the entire data management system under various operational conditions to ensure that the application remains user-friendly and that data handling does not intrude on or degrade the user experience.

10. Continuous and Secure Data Upload

FR10.1: The application must implement a robust mechanism for the continuous and secure uploading of data to the server, ensuring the integrity and confidentiality of user data at all times.

Details

- Automated Upload Mechanism: Develop a system that persistently monitors for new data and initiates uploads automatically. This mechanism should efficiently handle network fluctuations and automatically retry uploads using an exponential backoff strategy without requiring user intervention.

- Error Handling and Resilience: Integrate comprehensive error management protocols to address upload failures effectively. This includes detailed logging of upload issues and robust retry mechanisms to ensure data eventually reaches the server and UI feed back to the user about the status of the Uploading issue.

Verification Strategies:

- Reliability Testing: Perform automated testing to confirm that the upload mechanism reliably handles different network scenarios, successfully managing upload retries and completions as expected.
- Compliance Reviews: Organize periodic compliance audits to ensure ongoing adherence to legal and regulatory requirements, documenting compliance efforts and addressing any discrepancies immediately.

11. Background Service for Data Upload

FR11.1: The application must include an efficient background service that manages data uploads seamlessly, ensuring continuous operation even when the application is not in the foreground.

Details

- Persistent Service Operation: Develop a background service that operates continuously, optimizing resource usage to prevent excessive battery drain. This service should routinely check for data awaiting upload on the device.
- Intelligent Network Management: The service must adapt to network changes, pausing and resuming uploads based on connectivity status. It should prioritize secure connections to safeguard data during transmission.
- Data Integrity and Upload Reliability: Incorporate data verification protocols to ensure complete and accurate uploads. The system should retain data locally until successful upload confirmation and implement robust retry mechanisms for failed uploads due to network disruptions.
- UI feedback: if the upload process doesnt start due to any issues, notify the user with comprehensive information like the number of the files in the folder

Verification

- Background Service Efficacy Testing: Conduct tests to evaluate the service's performance across different operational scenarios, such as app closure, network transitions, and adverse network conditions, ensuring reliable background operation.
- Upload Accuracy: testing should confirm that the service accurately handles data retransmissions without loss.
- Resource Efficiency Evaluation: Monitor and assess the impact of the background service on the device's battery life and overall performance. Testing should verify that the service remains efficient and non-intrusive under various usage conditions.

12. Real-time UI Feedback

FR12.1: The application should provide immediate and real-time feedback to users regarding the connection status and battery levels of connected devices.

Details

- UI Integration: Integrate dynamic elements into the application's UI that can update in real-time. These elements should display the current connection status and battery levels of all connected devices.
- Feedback Frequency and Latency: Ensure that updates to the UI occur at least every few seconds or immediately after any change is detected in the device status. The system must be capable of processing and displaying changes within 2 seconds of occurrence.
- User Experience Design: Design the feedback elements to be clearly visible and understandable, using colors and icons that are easily interpreted by users. Consider accessibility options such as text descriptions for changes in connection status or battery levels.

Verification

- Real-time Performance Testing: Conduct UI performance tests to ensure that updates are displayed within 2 seconds of status changes. This testing should be carried out in various typical user scenarios.
- User Feedback Sessions: Gather user feedback specifically on the real-time feedback feature to assess its effectiveness and responsiveness. Adjust designs based on user preferences and suggestions for improvements.
- Accessibility Testing: Verify that all real-time feedback components comply with accessibility standards, ensuring that all users, including those with disabilities, can effectively receive and understand the feedback.

13. System Monitoring

FR13.1: The application must provide a comprehensive system monitoring solution that actively evaluates the operational status of various app components and alerts users to critical events and potential issues promptly.

Details

- Timestamp-Based Monitoring: The application must incorporate a status checker module to monitor timestamps from various activities such as data receipt, file writing, and data uploads. This module must calculate the elapsed time since the last activity for each component and assess operational health.
- Thresholds and Intervals: The application must establish and enforce specific time thresholds for starting delays and acceptable intervals for each data type and connected device. These thresholds will help define normal operational behaviors and expectations.
- Initial and Repeated Actions Check: The status checker must differentiate between initial and repeated actions, employing appropriate checks for each scenario. It must measure the time since the app's start for initial actions and since the last activity for repeated actions against predefined limits.
- Trigger Notification Logic: The application must trigger notifications if the elapsed time exceeds the set thresholds without any new activity. This function is essential to keep users informed of issues like stalled data reception, incomplete file writings, or halted uploads.

Verification

- Monitoring and Timing Tests: Regularly test the status checker to ensure it accurately calculates time intervals and correctly identifies when thresholds are exceeded. Use simulated delays and interruptions to verify that the system reacts appropriately.
- User Feedback Collection: Gather user feedback on the utility and accuracy of notifications, as well as the overall effectiveness of the system monitoring capabilities. Adjust system parameters based on feedback to optimize performance and user satisfaction.

14. User Notification System Enhancement

FR14.1: The application must develop an enhanced notification system that intelligently manages the timing and frequency of alerts, ensuring notifications are timely, relevant, and not redundant.

Details

- Smart Notification Logic: Implement a notification handler that evaluates the urgency and relevance of each message. This system must:
- Analyze Errors and Status: Determine the necessity of notifications based on real-time app conditions and error statuses.
- Control Notification Frequency: Integrate logic to throttle notifications, preventing redundant alerts by setting intervals specific to the urgency of the information.
- Track Notification History: Maintain a log of sent notifications to avoid sending repetitive messages unless updates occur or the situation persists.
- User Customization Options: Provide settings that allow users to personalize notification modes including sound, vibration, and visual alerts, and define conditions for suppressing or highlighting notifications.

Verification

- Notification Handling Tests: Conduct tests to validate the notification system's ability to discern when to issue or withhold notifications. Simulate various operational scenarios to confirm there are no unnecessary duplicates.
- User Experience Feedback: Collect and analyze user feedback from a test group to assess the notification system's effectiveness and user satisfaction. Use insights to fine-tune notification delivery, focusing on optimizing user engagement without overwhelming them.

15. Efficient Multi-Threaded Processing

FR15.1: The application must handle multiple tasks concurrently without degrading performance or user experience.

Details

Implement multi-threading or asynchronous processing techniques to manage data collection, UI updates, and network communication separately. Ensure that the UI remains responsive while background tasks are processed.

Verification

Conduct performance testing to verify that the application can handle multiple concurrent operations without significant slowdowns. UI responsiveness should be

measured under various load conditions to ensure it meets predefined responsiveness standards.

16. Scalable Architecture for Future Expansion

FR16.1: The application must be designed with a scalable architecture that can efficiently accommodate additional features, more users, and expanded data types without necessitating significant redesign.

Details

- Modular Design: The application must utilize a modular architecture with well-defined interfaces and APIs, which allows for the flexible addition and modification of system components.
- Infrastructure Scalability: Ensure that the core system includes scalable infrastructure components that can adjust dynamically to increased data loads and user numbers, such as scalable databases, cloud-based services, and efficient data handling algorithms.
- Future-Proofing: Design the system to be forward-compatible with upcoming technologies and standards, preparing for easy updates and integration of new technologies or modules as they become available.

Verification

- Scalability Testing: Conduct comprehensive scalability tests to evaluate how well the application handles increased loads across various dimensions, including user concurrency, data volume, and transaction throughput.
- Module Integration Testing: Test the ease of integrating new modules or features into the existing system architecture without affecting existing functionalities. Assess the impact on system performance and stability.
- Load Testing: Simulate real-world scenarios where the system operates at or near capacity to ensure that performance remains stable and meets predefined benchmarks.

4.4 Non-Functional Requirements

1. Performance

NFR1.1: The application must exhibit high performance, for response times and data processing speeds to ensure smooth user experience.

Details

- Application Load Time: The application should fully load and be ready for user interaction within no more than 5 seconds of launch.
- User Interaction Response Time: All user interactions should receive a response within 1 second under normal operating conditions to maintain a fluid user experience.

- Data Processing Efficiency: Data processing and compression tasks for one minute's worth of data should not exceed 5 seconds, ensuring timely handling and display of information.
- Connection Efficiency: The application must establish and maintain stable connections with wearable devices and servers swiftly, with connection setup times not exceeding 10 seconds under typical conditions.
- Continuous Data Upload: Data upload processes must be optimized to function in the background without disrupting user interaction, aiming for minimal noticeability by the user.
- Data Compression and Upload Speed: Implement data compression to reduce transmission times and server load. Compressed data files should be uploaded in less than 10 seconds under typical network conditions.

Verification

- Load Time Testing: Regular testing with automated tools to ensure the application meets the 5-second load time benchmark across different device specifications.
- Connection Stability and Speed Testing: Test the speed and reliability of connections between the application and wearable devices as well as the server. This should include testing for quick recovery from any drops and the ability to handle reconnections automatically without user intervention.
- Response Time Monitoring: Use on-device monitoring tools to track and log response times for user interactions. This data should be analyzed to identify any deviations from the 1-second target.
- Data Processing and Upload Testing: Conduct rigorous testing to measure the speed and efficiency of data processing and uploading. This includes stress testing the system under high data volumes to ensure it maintains performance without degradation.
- Compression Efficiency Evaluation: Test the efficiency of data compression algorithms to ensure they maintain at least a 50% reduction in file size without loss of data fidelity.
- Real-User Monitoring (RUM): Implement RUM to collect performance data from actual user interactions in real-world conditions to continuously evaluate and improve the application's performance based on live data.

2. Scalability

NFR2.1: The application must scale effectively to manage increases in data complexity and volume as well as integration with multiple wearable devices without impacting performance.

Details

- Data Volume and Complexity: The application should be designed to handle an increasing amount of data as more sensors or data points are added over time. It should process and store data efficiently, even as data complexity increases.
- Device Integration: The architecture should support seamless integration with additional wearable devices or new models of existing devices, accommodating a broader range of health metrics without degradation in app performance.

- Software Upgradability: The system should be built with modularity to allow for easy updates and feature enhancements that can be implemented without significant overhauls, ensuring longevity and adaptability of the app.

Verification

- Data Volume Testing: Conduct tests to ensure the application performs efficiently as the size and complexity of data from connected devices increase. This could involve simulated data inputs that replicate higher volumes and more complex datasets.
- Device Compatibility Testing: Regularly test the application with new devices and versions of devices to ensure compatibility and performance standards are maintained. This includes verifying that the app can handle data from multiple devices concurrently if necessary.
- Load Testing: Perform load testing under conditions that mimic the expected increase in data handling requirements and device connectivity to ensure the system maintains functionality and performance.
- Performance Benchmarks: Establish performance benchmarks based on current capabilities and monitor performance against these benchmarks after each upgrade or integration of new devices.

3. Usability

NFR3.1: The application must be highly usable, especially for elderly users or those with limited tech proficiency.

Details

Adhere to UX/UI design best practices including readability, high contrast visuals, and navigational ease.

Verification

Conduct usability testing with the target demographic and gather feedback to assess intuitiveness and ease of use.

4. Reliability

NFR4.1: The application must provide high reliability to ensure continuous operation and data accuracy, with mechanisms in place to handle errors and recover from failures effectively.

Details

- Robust Error Handling: Implement comprehensive error handling throughout the app to manage unexpected issues without crashing. This includes gracefully handling exceptions and providing user-friendly error messages.
- Data Recovery: Ensure that the app can recover from disruptions, such as network failures or unexpected shutdowns, without data loss. Implement mechanisms to resume interrupted data transmissions and restore previous states where necessary.
- Fault Tolerance: Design the application to continue operating even when components fail. This could involve using redundant processes for critical operations or ensuring that the app can switch to offline mode if server connectivity is lost.

- Continuous Operation: Minimize downtime by using stable libraries and frameworks, and by regularly updating the app to fix bugs and vulnerabilities that could affect performance and stability.

Verification

- Uptime Monitoring: Regularly monitor application uptime through automated systems to ensure that it meets the 99.9% uptime target. This monitoring should include active checks on app services and backend connections.
- Fault Injection Testing: Perform controlled fault injection testing to simulate common failures and observe how the app handles these situations. This helps in verifying the effectiveness of error handling and fault tolerance mechanisms.
- Recovery Testing: Test the data recovery processes to ensure they function correctly after disruptions. This involves scenarios such as simulating app crashes, network disconnections, and power failures to ensure that the app can resume operations without data loss.
- User Feedback: Collect and analyze feedback from users regarding the reliability of the app, focusing on their experiences with app stability, error occurrences, and the effectiveness of error messages and recovery processes.

5. Security

NFR5.1: The application must implement comprehensive security measures to protect data during transmission and manage stored data securely, ensuring user data confidentiality and integrity.

Details

- Secure Data Transmission: Utilize HTTPS with TLS 1.3 for all data transmitted between the application and the server. This ensures that all communication is encrypted, providing confidentiality and integrity of data in transit.
- Data Deletion Post-Transmission: Automatically delete files from the device once they have been successfully transmitted to the server to minimize data leakage risks and ensure that sensitive information is not accessible on the device longer than necessary.
- Minimized Data Storage: Limit the storage of sensitive data on the device. Only store data required for operational functionality and ensure it is protected during the short period it resides on the device.

Verification

- Security Testing: Perform regular security tests, including penetration testing and vulnerability assessments, to identify and rectify potential security vulnerabilities.
- Transmission Security Verification Regularly verify the implementation of HTTPS with TLS 1.3 using industry-standard tools to ensure encryption protocols are correctly applied and maintained.
- Data Deletion Efficacy: Test the reliability and effectiveness of the data deletion process after transmission to confirm that no residual data remains on the device.

6. Maintainability

NFR6.1: The application must be designed for ease of maintenance, supporting streamlined updates and modifications with minimal operational disruption.

Details

- Modular Architecture: Develop the application using a modular architecture, which allows for the isolated updating of components without impacting the overall system. This architecture should facilitate quick adaptation to changing requirements or technologies.
- Documentation Standards: Maintain comprehensive documentation that includes detailed descriptions of the architecture, code comments, API documentation, and update logs. This documentation should be regularly updated to reflect any changes or additions.
- Update Mechanism: Implement an efficient update mechanism that allows for hotfixes, patches, and version updates to be applied seamlessly, ideally without requiring system downtime or only during scheduled maintenance windows.

Verification

- Code Review Process: Regularly conduct thorough code reviews to ensure that coding standards are maintained and that the code remains clean and well-organized.
- Update Process Testing: Test the update mechanisms through staged rollouts to monitor the update impact and ensure that new code integrates smoothly with existing components without causing disruptions.
- Documentation Review: Periodically review and audit the documentation by independent teams to ensure it remains accurate and useful for ongoing maintenance efforts.

7. Accessibility

NFR7.1: The application must be most adoptable to ensure full accessibility for users with disabilities.

Details

- Accessible Design Features: Implement features that enhance accessibility for diverse needs, including:
- Text Scaling: Text fonts and text size must be most visible to accommodate the user with visual disability.
- High Contrast Modes: Provide high contrast visual modes to assist users with visual impairments in navigating and using the application effectively.

Verification

- Manual Testing: Engage accessibility experts and users with disabilities in manual testing sessions to explore the application's usability and accessibility in real-world scenarios.
- Feedback Integration: Systematically collect and analyze user feedback specifically related to accessibility features, using this input to guide continuous improvements in the application's design and functionality.

- Refined Non-Functional Requirement for Interoperability

8. Interoperability

NFR8.1: The application must ensure seamless integration and interoperability with selected health monitoring devices, adhering to required communication protocols and data standards.

Details

- Targeted Device Compatibility: The application is optimized to function with specific health monitoring devices. Ensure compatibility with these devices by adhering to the necessary Bluetooth profiles and communication standards.
- Protocol Compliance: Support essential communication protocols and standards, such as Bluetooth LE for efficient connectivity with designated wearable devices handling heart rate, movement, and other health metrics.
- Device Integration Assurance: Develop and maintain integration protocols for the supported devices, focusing on ensuring compatibility and optimizing performance without the need for extensive modifications.

Verification

- Targeted Integration Testing: Conduct integration tests specifically with the supported wearable devices to verify seamless interaction and data exchange, ensuring the application behaves as expected in real-world scenarios.
- Compliance Checks: Regularly review and validate the application's adherence to the specific protocols and standards necessary for the supported devices.
- Feedback Collection: Actively collect and analyze feedback from users and technical teams regarding the application's effectiveness and functionality with the supported devices, using this input to guide continuous improvement in device integration.

9. Efficient Resource Management

NFR9.1: The application must maximize efficiency in using device resources, including battery, storage, CPU, and memory, to enhance overall system performance and user experience.

Details

- Memory Optimization: Implement memory management practices that minimize overhead, such as effective garbage collection, memory pooling, and the use of efficient data structures.
- CPU Efficiency: Optimize CPU usage by minimizing unnecessary processing, utilizing asynchronous operations, and refining algorithms to reduce load.
- Battery Optimization: Integrate power-saving measures that extend battery life, including optimizing network usage, managing background tasks efficiently, and leveraging energy-efficient APIs.
- Storage Efficiency: Use advanced data compression techniques to reduce the storage footprint of data on the device. This includes compressing data before saving and employing efficient file management strategies to minimize I/O operations.

- Adaptive Resource Management: Dynamically adjust resource usage based on device conditions like battery level, network connectivity, and processor load to ensure optimal performance.

Verification

- Resource Monitoring: Implement tools to continuously monitor resource usage, identifying areas where improvements can be made and ensuring that usage stays within acceptable limits.
- Battery and CPU Usage Testing: Conduct extensive testing to measure the impact of the application on battery life and CPU usage, comparing it with industry benchmarks.
- User Experience Feedback: Regularly collect user feedback on the application's impact on device performance, particularly focusing on battery life and responsiveness, to determine if further optimizations are needed.
- Storage Utilization Tests: Evaluate the effectiveness of data compression and storage management strategies by monitoring the amount of disk space used by the app under various operational scenarios.

10. Regulatory Compliance

NFR10.1: The application must fully comply with all applicable EU health regulatory and privacy standards, including GDPR, as well as other international standards that may apply.

Details

- EU Specific Compliance: Ensure that all data handling, storage, and processing within the app comply with the General Data Protection Regulation (GDPR). This includes securing user consent before data processing, ensuring data minimization, and providing users with the right to access, correct, and delete their data.
- Health Data Regulations: Adhere to health-specific regulations applicable in Spain and the wider EU, such as the EU Directive on the application of patients' rights in cross-border healthcare, ensuring that patient data is handled in strict compliance with these laws.
- Features and Safeguards: Implement necessary technical and organizational features such as secure data transfer channels, and robust data access controls. Prepare for regular updates to the application as regulatory standards evolve.

Verification

- Regular Legal Audits: Conduct comprehensive legal audits at least bi-annually to ensure the application's ongoing compliance with GDPR and other relevant regulations.
- Compliance Reviews: Schedule regular reviews of all data handling procedures and privacy policies with legal experts specializing in EU health regulations.

11. Documentation and Support

NFR11.1: The application must provide exhaustive and easily accessible documentation for the developers, to facilitating effective development and troubleshooting.

Details

Developer Documentation: Provide detailed API documentation, code examples, and developer guides that are regularly updated to reflect the latest application changes and best practices. Include version control and change logs to assist developers.

Verification

Developer Feedback Sessions: Regularly conduct feedback sessions with developers, both internal and external, to assess the utility and clarity of the developer documentation and support tools.

4.5 System Requirements

4.5.1 Hardware Requirements

Android Devices:

- Processor: Minimum 1.2 GHz dual-core or higher
- Memory: Minimum 2 GB RAM
- Storage: Minimum 100 MB of free space for installation, additional space required for data storage
- Sensors: GPS, Bluetooth (if applicable for device connections)

iOS Devices:

- Processor: Minimum A9 chip (compatible with iPhone 6S/SE or newer)
- Memory: Minimum 2 GB RAM
- Storage: Minimum 100 MB of free space for installation, additional space for user data
- Sensors: GPS, Bluetooth (if applicable for device connections)

4.5.2 Software Requirements

Operating Systems:

- Android: Version 7.0 (Nougat) or higher
- iOS: iOS 12.0 or higher

Dependencies:

Libraries and Frameworks: Support for cross-platform development frameworks like Flutter, ensuring compatibility with the latest stable releases.

4.5.3 Network Requirements

4.5.3.1 Connectivity:

Requirement: The application requires continuous internet access to function effectively.

Details

At minimum, devices must maintain a 3G connection. However, for optimal performance and to ensure smooth operation of real-time features, a 4G LTE connection or a stable Wi-Fi network is strongly recommended.

Rationale

This ensures that the application can transmit and receive data reliably, especially important for real-time health monitoring and data syncing.

Verification

Performance Testing under Various Network Conditions: Test application performance under different network types and strengths to ensure functionality remains consistent and reliable.

4.5.4 Security Requirements

4.5.4.1 Data Encryption in Transit:

Requirement: All network transmissions must use HTTPS with TLS 1.3 to encrypt data between the app and the server.

Rationale

Ensures secure transmission of sensitive data, protecting against eavesdropping and data breaches.

4.5.4.2 Local Data Security:

Requirement: Automatically delete local data after its successful upload to the server to safeguard privacy and optimize storage.

Rationale

Prevents data leakage and manages device storage efficiently.

Verification

Security Testing: Regularly conduct penetration testing and security audits to verify the implementation of HTTPS and the effectiveness of data deletion protocols.

4.5.5 Localization and Internationalization

Languages Supported

The application must support both English and Spanish to cater to a diverse user base.

Regional Settings

Application should automatically adjust date, time, and numeric formats based on the user's locale settings.

4.5.6 Compatibility with Other Applications

4.5.6.1 Background Operation:

Requirement: The application must efficiently run in the background without disrupting or draining resources from other critical apps, particularly those used for navigation and communication.

Details

- Resource Management: Implement smart resource management strategies to minimize CPU, battery, and network usage when running in the background.
- Notification Management: Manage notifications in a way that they do not intrude excessively into the user's current tasks or disrupt other applications, especially during navigation or communication activities.
- Rationale Effective background operation is essential for user satisfaction, particularly for applications that need to perform continuous data monitoring or updates without interrupting user activities.

Verification

- Compatibility Testing: Conduct thorough testing to ensure that the application coexists harmoniously with other applications under various scenarios and conditions.
- Performance Monitoring: Monitor the application's performance impact on other apps, particularly focusing on resource usage like CPU and memory during background operation.
- User Experience Surveys: Regularly gather user feedback to determine if the application is causing any issues with other commonly used applications, using this data to make necessary adjustments.

4.6 Interface Requirements

4.6.1 Wearable Device Connectivity

IR1.1: The application must utilize Bluetooth Low Energy (BLE) to ensure energy-efficient communication with wearable devices such as Sensoria Smart Socks, MetaWear Motion Sensor, and HBand Smart Band.

IR1.2: Automatically scan for and display all available BLE-enabled devices within range to facilitate easy connection.

IR1.3: Establish and maintain a secure connection using standardized BLE pairing protocols to prevent unauthorized access.

IR1.4: Automatically manage device disconnections and reconnections to maintain data integrity and user experience, handling out-of-range events and manual disconnections seamlessly.

Verification

- Test the ability to detect and display BLE-enabled devices within operational range under various conditions.
- Simulate various disconnection scenarios to ensure seamless management of reconnections and data integrity.

4.6.2 Data Transfer

IR2.1: Support high-frequency real-time data streaming from connected devices to capture all essential health metrics like movement, heart rate, and temperature without loss.

IR2.2: Implement a data buffering strategy to ensure no data loss during transmission interruptions or processing delays.

Verification

- Stress test data streaming to ensure accurate, lossless transmission of health metrics at high frequencies.
- Introduce network interruptions to validate the effectiveness of the buffering strategy in preventing data loss.

4.6.3 Server Communication

IR3.1: Employ HTTPS for all data transmissions between the application and the server to safeguard data against interception and tampering.

IR3.2: Optimize server communication by supporting file uploads in a compressed format, reducing bandwidth requirements and enhancing performance.

IR3.3: Implement reliable acknowledgment mechanisms to confirm file transfer completion before local data is deleted, ensuring no data loss.

IR3.4: Gracefully handle communication errors or interruptions by queuing unsent data for later transmission, ensuring reliability and continuity of service.

Verification

- Conduct security assessments to verify the implementation and effectiveness of HTTPS for data transmission.
- Verify that the application receives and properly handles acknowledgments from the server before deleting local data.

4.6.4 User Interface (UI) Integration

IR4.1: Dynamically update the user interface to reflect the connection status and battery levels of connected devices, providing real-time feedback to the user.

IR4.2: Configure the application to trigger notifications based on specific events such as device disconnections, low battery alerts, or data transfer errors, keeping the user informed.

Verification

- Perform user interface testing to confirm that the connection status and battery levels of connected devices are accurately and promptly reflected in the UI.
- Set up scenario-based testing to ensure that notifications are correctly triggered by specific events like device disconnections or low battery alerts, and validate user notification clarity and timeliness.

4.6.5 Multilingual Support

IR5.1: The application must offer a multilingual user interface, initially supporting English and Spanish, with the capability to expand to other languages.

R5.2: Ensure comprehensive localization of all textual elements within the application, including dynamic content like notifications and real-time status updates, adapting content to the language and cultural nuances of the user's settings.

Verification

Conduct interface testing in both English and Spanish to ensure the UI supports multiple languages without functionality loss, checking for seamless switching between languages.

4.6.6 Interface Usability

IR6.1: The Design of the user interface must be simple and intuitive, suitable for elderly users and those with limited technical knowledge. Utilize user-friendly navigation, clear labeling, and interactive elements that are easy to access and use.

Verification

Conduct usability testing sessions focusing on elderly users and those unfamiliar with advanced technology to verify that the interface meets their needs. User testing should demonstrate that at least 90% of participants can navigate the application independently without additional help or guidance.

4.6.7 Visual Accessibility

IR7.1: Implement high-contrast themes, large buttons, icons, and text within the application to cater to users with visual impairments or decreased vision.

Verification

Perform accessibility testing with participants from the target demographic to ensure all visual design elements are perceptible and operable. The application must meet or exceed ADA compliance standards during these tests.

4.7 Constrains

4.7.1 Technological Constraints

1 Platform Support

C1.1: The application is designed to function only on Android and iOS platforms using the Flutter SDK and Dart programming language.

Implication

This limits the application's availability to users on other platforms, like Windows or Linux. It also means that development must conform to the capabilities and limitations of Flutter, which might affect the performance, appearance, and behavior of the app across different devices.

2 Bluetooth Technology

C2.1: Relies on Bluetooth Low Energy (BLE) technology to connect with wearable devices.

Implication

The effective range and reliability of the application are tied to Bluetooth's operational limits. This could impact how well the app performs in different environments and necessitates BLE support in all target devices, possibly excluding devices without BLE capabilities.

3 SDK Limitations

C3.1 The application's communication and data management depend on specific SDKs that dictate the available classes and functions.

Implication

This limits the flexibility in customizing and optimizing certain functionalities. The developers must work within the confines of these SDKs, which may not support all desired features or optimal performance tuning specific to the app's needs.

4.7.2 Device Capability Constraints

4 Resource Efficiency

C4.1: The Mobile application must optimize the use of device resources, including processor, memory, and battery, to prevent significant power consumption.

Implication

Requires careful management of background processes, data handling, and user interface elements to minimize energy consumption and ensure smooth operation without causing devices to become idle or drain battery rapidly.

4.8. Assumptions and Dependencies

4.8.1 User Device Compatibility

A1: It is assumed that all users have devices that meet or exceed the minimum system requirements specified for Android and iOS platforms. This includes operating systems Android 7.0+ and iOS 12.0+, adequate RAM, and sufficient storage capacity.

4.8.2 Connectivity

A2: The application assumes users will have continuous, stable, and secure internet access, which is critical for real-time data transmission with the server .

User Permissions

A3: It is assumed that users will grant all necessary permissions for optimal app functionality. This includes permissions for Bluetooth and location services to enable device connectivity and for internet access to sync and process data.

4.8.3 Wearable Device Availability

A4: The application assumes that the specific wearable devices it is designed to work with, such as Sensoria Smart Socks, MetaWear Motion Sensor, and HBand Smart Band, are equipped with the necessary compatible firmware and Bluetooth capabilities.

4.8.4 Regulatory Compliance

A5: It is assumed that the developer has a comprehensive understanding of and will adhere to GDPR, HIPAA, and other relevant data protection and privacy regulations throughout the development and deployment of the application.

4.8.5 Server Infrastructure

D2: Effective operation of the application is dependent on robust server infrastructure. Reliability, scalability, and uptime of the server, especially the time series database where health metrics are stored, are crucial.

4.8.6 Device APIs and SDKs

D3: The application's performance and compatibility are dependent on the stability and continuity of the APIs and SDKs provided by Android and iOS. Any significant updates or discontinuations can impact the application's functionality.

4.8.7 Network Environment

D4: The application depends on the network environment being capable of handling real-time data transmission efficiently. External factors like network throttling, congestion, or disruptions can adversely affect performance.

4.8.8 User Training and Support

D5: The effectiveness of the application is partially contingent upon users being properly trained on how to use the app and respond to the data and notifications it provides. Adequate user education and support infrastructure are critical.

Chapter 5: Methodology

Overview

This chapter details the development methodology of the cross-platform mobile application which is vital component of the broader IoT system which designed to connect and capture real-time movement and health metrics data from various wearable devices and send it to the server side . The application, built using Flutter, aims to provide a user-friendly interface, robust data handling, and secure data transmission to support the treatment of patients with physical mobility impairments by analyzing the data captured by this application from the patient.

5.1 Development Methodology

5.1.1 Selection of Methodology

5.1.1.1 Chosen Methodology

The Waterfall development methodology has been selected for this project. Unlike Agile, Waterfall is a linear and sequential approach where each phase (requirements gathering, design, implementation, verification, and maintenance) must be completed before the next begins. This method is particularly suitable for projects where individual developers need clear, structured stages and minimal iterative changes once a stage has been completed.

5.1.1.1 Justification for Waterfall Methodology:

5.1.1.1.1 Clear Project Phases

- Structured Approach: The Waterfall model offers a straightforward and predictable approach, which is beneficial when working solo. It allows for thorough planning and execution of each phase before moving to the next, ensuring that no aspects of the project are overlooked.
- Documentation and Review: Each phase ends with a review process and deliverables, which is crucial for maintaining a comprehensive documentation trail when developing alone. This facilitates better tracking of progress and decisions throughout the project lifecycle.

5.1.1.1.2 Stability and Requirements:

- Fixed Requirements: Given that the app's requirements were well understood and stable from the outset, the Waterfall methodology is appropriate because it relies on detailed pre-defined requirements that do not expect to change significantly, allowing for a more deliberate and measured development pace.
- Independent Testing: After the implementation phase, extensive testing is conducted to ensure the application meets all specified requirements. This systematic testing is crucial in a Waterfront environment to confirm that all functionalities work as intended before deployment.

5.1.1.1.3 Efficiency in Solo Development:

- Focus and Milestones: Waterfall allows the developer to focus intensely on one phase at a time without the distractions of overlapping phase activities, which is often more manageable for a single developer.
- Predictable Timeline: The linear progression from one phase to the next helps in setting clear and achievable deadlines at each stage, which is essential for time management when working independently.

5.1.2 Implementation Strategy:

5.1.2.1 Detailed Planning

Begin with an in-depth planning phase where all application requirements are defined and documented.

5.1.2.2 Design and Development

Following the planned requirements with a robust design phase, ensuring all user interface and functional elements are mapped out before coding begins.

5.1.2.3 Thorough Testing

After development, move into a rigorous testing phase, where every aspect of the app is tested for functionality, usability, and reliability to ensure it meets the initial specifications.

5.1.2.4 Expected Outcomes:

- Compliance with Specifications: The use of Waterfall is expected to result in a product that closely aligns with the initial specifications, as revisions are generally not introduced once a phase is completed.

- Quality Assurance: Systematic and comprehensive testing following the completion of the development phase helps in identifying and fixing any issues before the app is released, enhancing the overall quality of the final product.

5.1.3 Development Phases:

Break down the development process into phases such as planning, design, implementation, and testing. Describe what happens at each stage and how transitions are managed.

5.1.3.1 Planning

5.1.3.1.1 Scope Definition

Begin by clearly defining what the application will and will not include, establishing the boundaries and goals of the project.

5.1.3.1.2 Requirements Gathering

Conduct a thorough analysis to gather all necessary functional and non-functional requirements. This phase involved consultations with Dr. Ordieres MerÃ© who has close communcation with healthcare providers and users , to understand their needs and preferences.

5.1.3.1.3 Timeline Establishment

Develop a detailed project timeline that outlines each phase, including key milestones and deadlines, to ensure the project remains on track.

5.1.3.2 Design:

5.1.3.2.1 UI/UX Design:

Creating detailed user interface designs focusing on accessibility and ease of use, especially considering the elderly or users with limited tech proficiency. Using wireframes to visualize the interaction and design of the application.

5.1.3.2.2 System Architecture:

The architectural framework of the app, detailing how the mobile application will interact with wearable devices and the server. Include considerations for data handling, security measures, and connectivity.

5.1.3.2.3 Data Flow Diagrams:

Comprehensive data flow diagrams to illustrate how data will move through the system, from data collection on the wearables to data processing and storage on the server.

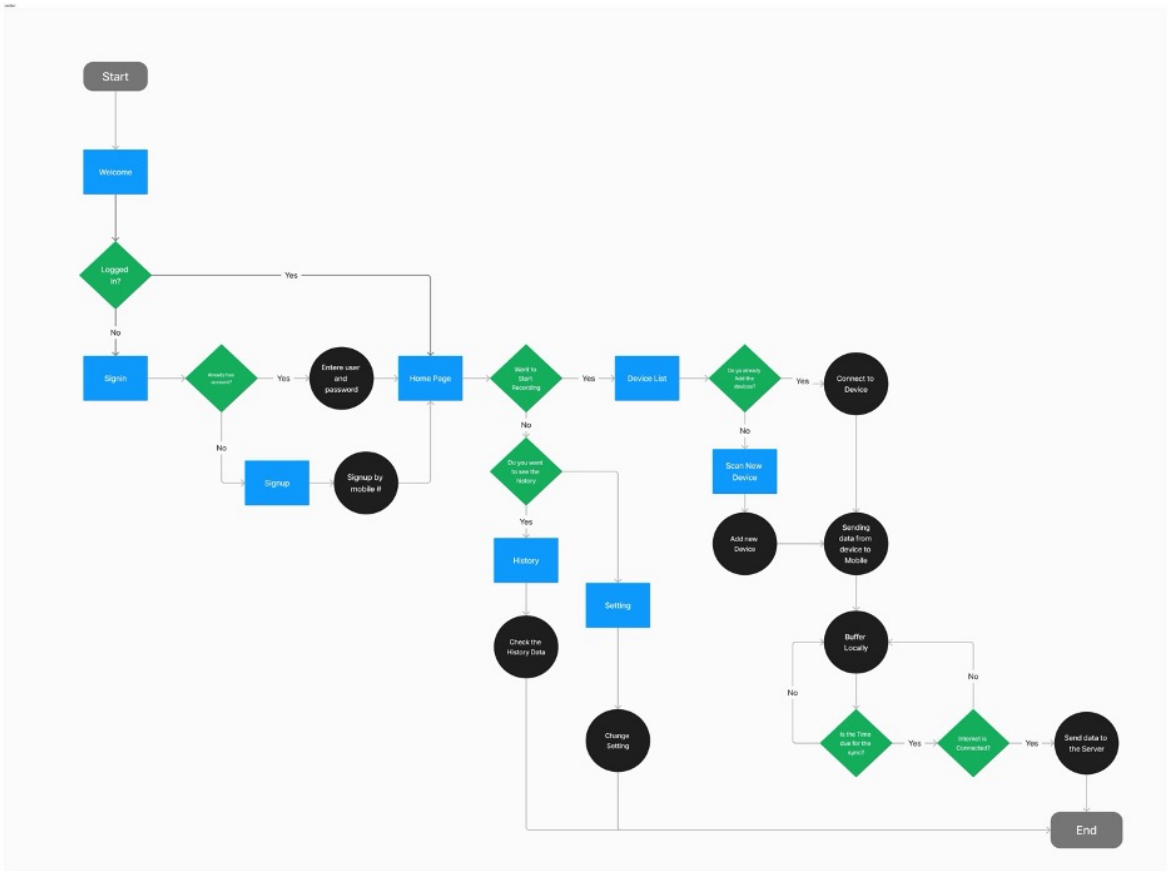


Figure 3.User Flows

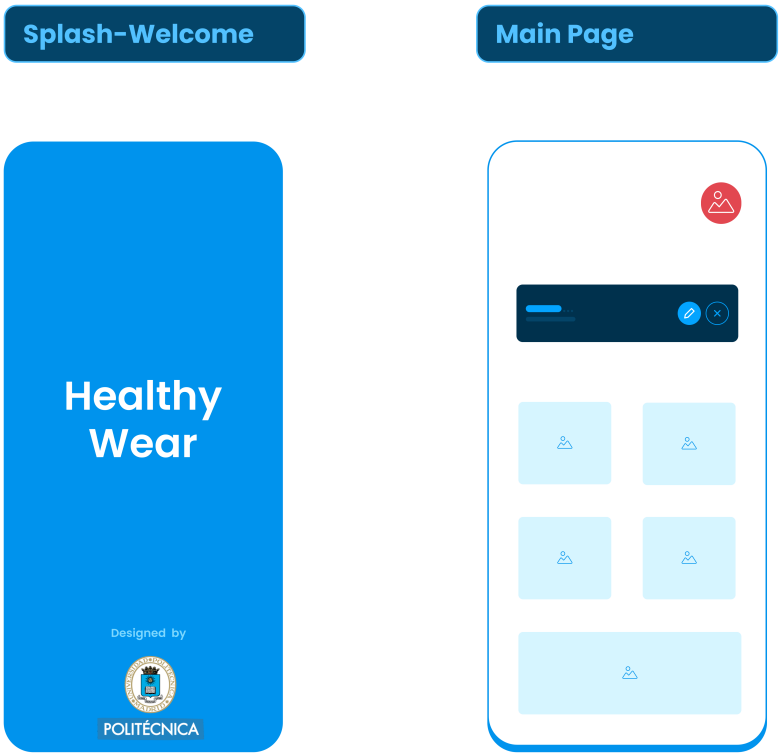


Figure 4.Screen Design

5.1.3.3 Implementation

5.1.3.3.1 Application Development:

By using Flutter to develop the application, ensuring cross-platform functionality across iOS and Android. Implement the core functionalities defined in the requirements, such as data collection, user authentication, and real-time data display.

5.1.3.3.2 Integration with Wearables:

Integration of the SDKs from the wearable devices using Method Channels in Flutter, ensuring the app can communicate effectively with the hardware to receive real-time health data.

5.1.3.3.3 Integration with Server:

Transmission the data files to the server and and receive the reception successful acknowledge for further process.

5.1.3.4 Testing

Unit Testing: Performing unit testing to ensure each component or module of the application functions correctly independently.

Integration Testing: Conducting tests to verify that the app components work together seamlessly and the entire system functions as intended.

User Acceptance Testing (UAT): Engage with actual users to test the app in real-world scenarios to validate the functionality, usability, and overall user experience.

5.2 Tools and Technologies

5.2.1 Programming Languages and Frameworks:

5.2.1.1 Flutter (Dart) for Mobile App Development:

- **Framework and Language:**: The mobile application is developed using Flutter, which utilizes the Dart programming language. Dart is designed for building fast and scalable applications across platforms.
- **Cross-Platform Development**: Flutter allows the development of high-performance applications for both iOS and Android from a single codebase. This cross-platform capability ensures consistency in functionality and appearance

across devices, reducing the need for separate codebases and associated development costs.

- **UI Flexibility and Rich Library Support**: Flutter offers a rich set of customizable widgets and a dynamic rendering engine for creating intuitive and responsive user interfaces. This is critical for ensuring the application is accessible, especially to elderly users or those with limited tech proficiency.

5.2.1.2 Integration with Native SDKs through Method Channels:

- **Native Features Access**: Method Channels are chosen to be used to facilitate communication between the Flutter app and native platform code (iOS and Android). This is necessary to utilize native SDKs that are not directly accessible through Flutter, such as those required for connecting to specific wearable devices.
- **Performance Optimization:** By leveraging native code, the app can perform more efficiently in areas that require direct interaction with the device hardware, such as data collection from sensors on wearable devices.

5.2.1.3 Use of Flutter Libraries:

- **Functionality Extension**: Various Flutter libraries are selected to enhance the application's capabilities, including but not limited to connectivity, data handling, and UI components. These libraries source from reputable providers within the Flutter ecosystem, ensuring they are reliable and well-supported.
- **Community and Support**: Flutter's extensive community support and continuous development mean that libraries and tools are regularly updated and expanded, providing a robust foundation for building complex applications.

5.2.1.4 Justification for Technology Choices:

- **Efficiency and Scalability**: Flutter’s ability to compile into native code and its single codebase architecture across multiple platforms make it an ideal choice for this project. It ensures the app runs efficiently and can be easily scaled and maintained.
- **Rapid Development and Rich Ecosystem**: Flutter’s hot reload feature and its comprehensive set of widgets speed up the development process by allowing for instant visual feedback and reducing the time needed for adjustments. The active community and rich ecosystem provide access to numerous resources and libraries, facilitating rapid development and problem-solving.
- **Seamless Integration with Native Capabilities**: The use of Method Channels allows for seamless integration of native platform functionalities, crucial for accessing advanced device features that are outside of Flutter’s direct capabilities. This integration ensures that the application leverages the full potential of the underlying hardware and native APIs, enhancing overall performance and user experience.

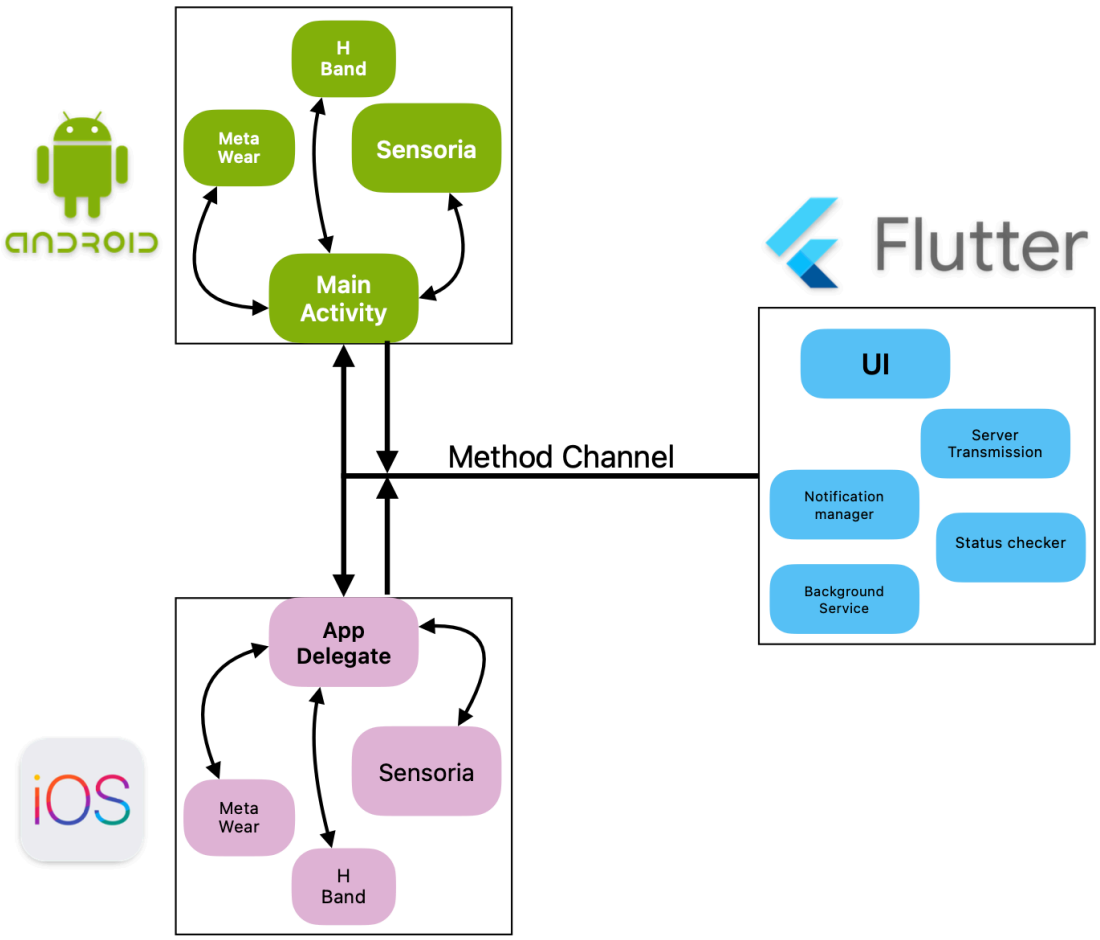


Figure 5.HealthyWear flutter app diagram

5.2.2 Development Approach and Technology Integration

5.2.2.1 Hybrid Development Using Flutter and Native Code:

The development of the mobile application leverages a hybrid approach, utilizing both Flutter and native Android/iOS coding environments to optimize functionality and performance. This strategy addresses the limitations posed by the availability of device SDKs, which are exclusively accessible through native platforms.

5.2.2.1.1 Flutter Implementation:

Common functionalities that are platform-independent, such as the user interface, permissions handling, notifications, status checking, file uploads, and background tasks, are implemented using Flutter. This choice capitalizes on Flutter's strengths in creating a unified and seamless user experience across multiple platforms, enhancing maintainability and scalability of the codebase.

5.2.2.1.2 Native Android and iOS Implementation:

Device-specific interactions, particularly those involving direct hardware communication such as device connection, data reception, and file system operations, are handled natively. This includes:

- Connection Handling: Establishing and maintaining connections with the wearable devices using native Bluetooth APIs.
- Data Reception and Buffering: Managing real-time data streams from the devices, implementing efficient data buffering strategies to ensure data integrity.
- File Management: Writing buffered data to the device's file system, utilizing native APIs to handle file operations securely and efficiently.

5.2.2.1.3 Integration via Method Channels:

The bridge between Flutter and native functionalities is facilitated through Method Channels, allowing high-level Flutter components to communicate with and control native code. This integration ensures that while Flutter manages the general application framework and user-facing features, the heavy lifting of device interaction is efficiently handled by the native layers, providing a robust solution to the SDK availability issue.

5.2.2.1.4 Justification for development approach:

This hybrid approach is necessitated by the SDK limitations of the wearable devices, which are only compatible with native development environments. By segregating responsibilities in this manner—Flutter for cross-platform compatibility and user interaction, native code for device-specific operations—we ensure that the application remains both powerful and agile, capable of providing high-performance features while maintaining a consistent and accessible user interface.

5.2.3 Development Environment and IDEs:

5.2.3.1 Integrated Development Environments (IDEs):

5.2.3.1.1 Android Studio

Usage: Android Studio is choosed as the primary IDE for both Flutter and native Android development. It supports the full Dart plugin and Flutter framework, facilitating efficient coding, debugging, and testing of the mobile app.

Features: Android Studio is advantageous for its comprehensive debugging tools, performance monitoring, and built-in emulation for Android devices. It also offers direct integration with Flutter, making it seamless to handle both Dart and native Android code within the same environment.

5.2.3.1.2 Xcode:

Usage: Selected for developing and debugging the native iOS side of the application. Essential for ensuring that the iOS-specific elements of the app, such as native plugins or iOS-specific functionalities, are optimally integrated and tested.

Features: Provides sophisticated tools for iOS development, device simulators for various iOS devices, and profiling tools to ensure optimal application performance on iOS.

5.2.3.2 Version Control System:

5.2.3.2.1 Local Git

Used for local repository management, enabling version control on the individual development machine. Git is crucial for tracking changes, managing branches, and effectively handling project history.

5.2.3.2.2 GitHub

Extends the capabilities of local Git by providing a cloud-based hub where code can be stored, shared, and collaboratively reviewed. GitHub facilitates backup, collaboration, and issue tracking, allowing for robust project management and sharing with potential collaborators or reviewers.

5.2.3.3 Justification for Tool Choices:

5.2.3.3.1 Comprehensive Development Support:

- **Android Studio and Xcode:** Using both Android Studio and Xcode allows for a cohesive development experience across both major mobile platforms (Android and iOS), ensuring that the app delivers a consistent and optimized user experience regardless of device. The use of both IDEs enables specific platform features to be utilized and fine-tuned, enhancing the overall quality and performance of the app.
- **Cross-Platform Efficiency:** Android Studio's integration with Flutter and its support for native Android development combined with Xcode for iOS ensures that all aspects of the app can be developed and tested in environments best suited for each platform's specifics.

5.2.3.3.2 Reliable Version Control:

Git and GitHub: The combination of local Git for immediate version control needs and GitHub for remote storage and collaboration ensures that the development process is not only well-documented and reversible but also open for collaborative contributions and secure against data loss.

5.2.3.3.3 Streamlined Workflow:

IDE Integration: Both IDEs support direct integration with Git, streamlining the development workflow. This integration allows changes to be committed and

pushed without leaving the development environment, reducing complexity and potential errors.

5.2.4 APIs and Libraries:

In the planning phase of the mobile application development, strategic decisions were made regarding the selection of APIs and libraries that would best meet the project's requirements. The primary criteria for these choices included compatibility with multiple platforms, the ability to interface seamlessly with hardware, and the facilitation of robust data management and user interaction capabilities.

5.2.4.1 Framework and Tools Planning:

5.2.4.1.1 Flutter SDK:

Chosen as the primary development framework due to its rich feature set that supports building natively compiled applications for both Android and iOS from a single codebase. This decision was based on the need for efficiency in maintaining and updating the app across multiple platforms. The Flutter framework is known for its extensive UI toolkit, which allows for crafting responsive layouts that can dynamically adjust to different device specifications.

5.2.4.1.2 Android and iOS SDKs:

These were selected to ensure that the application could leverage platform-specific features such as Bluetooth connectivity and local data storage. The integration of these SDKs is critical for accessing native functionalities that enhance the app's performance and provide a seamless user experience.

5.2.4.1.3 Method Channels:

Identified for inclusion to facilitate communication between the Flutter application and the native platform code. This choice was crucial for accessing native device features that Flutter does not directly support, thereby expanding the app's capabilities.

5.2.4.1.4 Device-Specific SDKs:

Metawear SDK, Sensoria SDK, HBand SDK: These SDKs will be used to handle functionality and specific data from corresponding devices, ensuring that the app can fully leverage the specialized hardware functionalities offered by each type of wearable device.

5.2.4.2 Key Libraries Consideration:

5.2.4.2.1 flutter_reactive_ble:

Planned for its ability to handle Bluetooth Low Energy communications. used this library in the flutter side to check the availability of the bluetooth module

5.2.4.2.2 permission_handler:

Selected to manage runtime permissions efficiently, ensuring that the app adheres to best practices for privacy and security by appropriately requesting and handling necessary permissions.

5.2.4.2.3 flutter_local_notifications:

Chosen to enhance user interaction by managing local notifications, which alert users about important health monitoring tasks or data updates.

5.2.4.2.4 geolocator:

Considered for its capability to provide access to location services, enhancing the application's data with geographical information, which is particularly valuable for environmental health analysis.

5.2.4.2.5 http:

Identified for handling network communications, crucial for sending and receiving data to and from a backend server, supporting features such as data synchronization and user authentication.

5.2.4.2.6 flutter_background_service:

Selected to ensure that the app can continue data collection and processing activities in the background, critical for a health monitoring app that requires continuous operation.

5.2.4.3 Justification for Selections:

The rationale behind choosing these specific technologies and libraries is grounded in their ability to enhance the application's functionality, ensure its operational efficiency, and maintain a high level of user engagement. The integration of these tools supports a seamless and efficient development process, critical for building a reliable, scalable, and robust application that meets the demands of real-time health monitoring.

5.3 Data Handling and Security

5.3.1 Data Collection:

Data collection from wearable devices in this project is executed through a carefully planned and standardized process to ensure consistency, accuracy, and reliability. The methodology behind data collection involves several key components and protocols:

5.3.1.1 Bluetooth Connectivity:

- **Initial Connection:** The mobile application establishes a connection with the wearable devices using Bluetooth Low Energy (BLE), a protocol chosen for its efficiency and low energy consumption, which is crucial for prolonged operations of wearable technologies.
- **Data Transmission Protocol:** Once connected, the wearable devices begin to transmit data to the mobile app based on predefined settings that include the data rate (measured in hertz). This rate determines how frequently data is sent from the wearable device to the mobile application, a critical parameter that balances between real-time monitoring needs and battery conservation.

5.3.1.2 Configuration and Standardization:

- **Device Configuration:** Each wearable device is configured to stream data at a specific frequency and data rate that suits the type of measurements being collected. For example, devices monitoring heart rate or step count may transmit data at different frequencies.
- **Data Format Standardization:** The data transmitted by the wearable devices is standardized according to a schema that defines the data structure, including the types of data collected (e.g., heart rate, steps, temperature), the required format, and any metadata associated with each data point.

5.3.1.3 Data Integrity and Accuracy:

Timestamping: Each piece of data collected is timestamped at the point of collection, ensuring that the sequence and timing of data records are preserved. This is vital for subsequent data analysis phases where temporal patterns and trends are examined.

5.3.2 Data Processing:

Data processing within the mobile application is a critical operation designed to efficiently manage, transform, and prepare data received from wearable devices for storage and further analysis. The application employs a sophisticated buffer system that ensures data integrity and systematic handling.

5.3.2.1 Initial Data Handling:

- **Data Reception:** Data from wearable devices first enters a synchronized buffer upon arrival. This system manages the flow of incoming data effectively, maintaining the sequence of data receipt.
- **Data Conversion and Tagging:** Incoming data typically arrives in a binary format and is converted into a human-readable format for further processing. Each data packet is tagged with a timestamp to mark the exact time of capture. Additionally,

data is labeled with identifiers such as â€˜right' or â€˜left' to specify the source limb (e.g., which hand or foot), enhancing the context and relevance of the data.

5.3.2.2 Buffer System:

- **Temporary Storage:** The buffer acts as temporary storage where data is held before processing. This ensures that no data is lost, even during periods of high data transmission or when multiple devices transmit data simultaneously.
- **Threshold Management:** The buffer is configured to trigger data processing once it reaches a pre-defined limit. When the buffer's capacity threshold is met, the stored data is processed collectively. This mechanism balances the need for real-time processing with efficient use of storage and processing resources.

5.3.2.3 Data Transformation and Preparation for Storage:

Pre-Storage Processing: Data in the buffer undergoes final preparations before being saved to the device's storage. This includes structuring the data into organized strings that are ready for file writing, facilitating subsequent analysis and retrieval.

5.3.2.4 Justification for Data Processing Strategy

5.3.2.4.1 Use of a Synchronized Buffer System:

- **Data Integrity:** The synchronized buffer ensures that all data is handled in a thread-safe manner, which is crucial in a multi-threaded environment where data is constantly being sent from multiple wearable devices. This prevents data corruption and loss, ensuring that the integrity of the data is maintained.
- **Efficiency:** Processing data only when the buffer reaches a predetermined threshold helps manage system resources more efficiently. This approach reduces the frequency of write operations, which can be resource-intensive, thereby optimizing the application's performance and extending the battery life of the mobile device.
- **Reliability:** By storing data temporarily in a buffer, the system can handle intermittent connectivity issues or data transmission delays without losing any data. This reliability is crucial for health monitoring applications where every piece of data can be critical.

5.3.2.4.2 Data Conversion and Tagging:

- **Enhanced Data Usability:** Converting data from binary to a human-readable format and tagging it with contextual identifiers (such as time stamps and source tags) make the data immediately useful for analysis and visualization. This also simplifies subsequent data processing stages, enhancing the application's overall functionality.

- **Accurate Contextual Analysis:** Tagging data with specific identifiers related to the source limb or side of the body allows for more precise analysis of the health

metrics, which is particularly important in applications targeting physical rehabilitation or monitoring specific bodily functions.

5.3.2.4.3 Pre-Storage Processing:

Organized Data Management: Structuring data into organized strings before storage facilitates efficient data retrieval and analysis, enabling health practitioners to quickly access and interpret health data. This organization is vital for providing timely insights and making informed health decisions.

5.3.3 Data Storage:

The data storage in the mobile application are designed to ensure that all health data collected from wearable devices is stored securely and managed in a way that respects privacy and ensures data integrity.

5.3.3.1 Local Data Storage:

- **Writing to File:** Data collected from devices is buffered and, upon reaching a certain threshold, written to text files in the local storage of the mobile device. These files include not only the raw data but also metadata, which provides additional context such as user reference numbers, device identifiers, and timestamps. This metadata is crucial for data traceability and analysis.
- **File Management:** To optimize the use of storage space and prepare data for transmission, these text files are compressed using gzip compression. This compression reduces the file size, making it more efficient to store and quicker to transmit.

5.3.3.2 Data Transmission and Server Storage:

- **Secure Transmission:** Data is transmitted to the server using HTTPS, a protocol that ensures secure data transfer over the Internet by encrypting the data in transit. This prevents unauthorized access and interception of the data.
- **Server Storage:** Once data reaches the server, it is stored securely with measures in place to ensure it remains intact and accessible only to authorized personnel.
- **Upload and Deletion Protocol:** After successful transmission and confirmation of data integrity on the server, the original files on the local device are deleted. This step is crucial to ensure that the device does not retain sensitive data longer than necessary, minimizing potential exposure.

5.3.3.3 Justification for Data Storage and Security Measures:

Efficiency : The chosen methods for data storage optimize the performance of the application. By minimizing the storage footprint on the device through compression and managing data lifecycle through secure deletion protocols, the application maintains high performance while safeguarding user data.

Chapter 6: Implementation

Overview

This chapter elaborates on the actual development and realization of the mobile application within the IoT system, focusing on the specifics of coding, integration, user interface implementation, and the operational functionality of the system.

6.1 Development Environment

6.1.1 Programming Languages and Tools

6.1.1.1 Dart Programming Language:

Dart is the primary language used for developing the application within the Flutter framework. Its design focuses on front-end development, offering features such as just-in-time compilation and strong typing, which accelerate both development and performance testing phases. Dart's ability to compile into native code is crucial for achieving seamless operation and high performance on both Android and iOS platforms.

6.1.1.2 Java for Android Development:

Java is used to develop android side of the app to handle platform-specific functionalities for Android devices, particularly those involving deeper integration with device hardware, such as Bluetooth communications and local data storage. Java's robust library ecosystem and its well-established environment for Android development allow for reliable and secure implementation of these critical features.

6.1.1.3 Swift for iOS Development:

Swift is utilized for developing parts of the application that require direct interaction with iOS-specific APIs, such as health data integration and user notifications. Swift's modern syntax and safety features make it an ideal choice for creating a high-performance, intuitive application on iOS devices.

6.1.1.4 JSON for Data Serialization:

JSON is utilized for data serialization, particularly in communications between the mobile app and the backend server. Its lightweight nature and easy compatibility with web APIs enhance data exchange efficiency, crucial for real-time health data processing.

6.1.2 Development Platforms:

6.1.2.1 Flutter:

- **Cross-Platform Functionality:** Flutter allows one codebase to compile into native apps on both iOS and Android platforms. This capability significantly reduces development and maintenance costs while ensuring that users experience consistent functionality and interface aesthetics across devices.
- **Widget Catalog:** Flutter provides an extensive catalog of customizable widgets which facilitates the development of a responsive and intuitive user interface. This is particularly beneficial for meeting diverse user requirements, including accessibility features for elderly users or those with limited tech proficiency.
- **Community and Support:** The Flutter framework is supported by a robust community and comprehensive documentation. This extensive support network aids in quickly addressing development challenges and implementing advanced features.

6.1.2.2 Integrated Development Environments (IDEs):

6.1.2.2.1 Android Studio:

Features: Android Studio is integrated with Flutter and Dart plugins, offering a powerful suite of tools including a visual layout editor and integrated DevTools for performance monitoring. Its built-in Android emulator and profiler tools are essential for optimizing application performance.

Benefits: The IDE streamlines the Android development process, providing developers with tools to efficiently code, test, and debug applications. Android Studio supports Flutter's hot reload feature, which dramatically speeds up the development cycle by allowing instant feedback on changes.

6.1.2.2.2 Xcode:

Usage: Xcode is used for the iOS part of the development, ensuring that iOS-specific functionalities are finely tuned and tested. It provides tools such as SwiftUI for advanced UI design, interface builders for layout crafting, and simulators for various iOS devices.

Integration with Flutter: While primarily an iOS development tool, Xcode seamlessly integrates with Flutter, facilitating the iOS build process and managing iOS-specific assets and configurations.

6.1.2.3 Version Control Systems

6.1.2.3.1 Git:

- **Local Version Control:** Git is essential for version control, providing a robust framework for tracking changes, managing branches, and handling project versions efficiently. This system is critical for maintaining the integrity and history of the development process.

- **Branch Management:** Effective branch management strategies are employed, utilizing feature branches for new developments, release branches for production releases, and hotfix branches for quick bug fixes.

6.1.2.3.2 GitHub:

- **Remote Repository Features:** GitHub hosts the project's repository, enhancing collaboration through pull requests and code reviews. It also offers issue tracking, which is instrumental in managing bugs and feature requests efficiently.
- **Integration with Development Tools:** GitHub integrates seamlessly with Android Studio and Xcode, linking the development environment with continuous integration/continuous deployment (CI/CD) pipelines. This integration supports automated testing and deployment, which are crucial for maintaining high-quality software standards.

Repository Link:

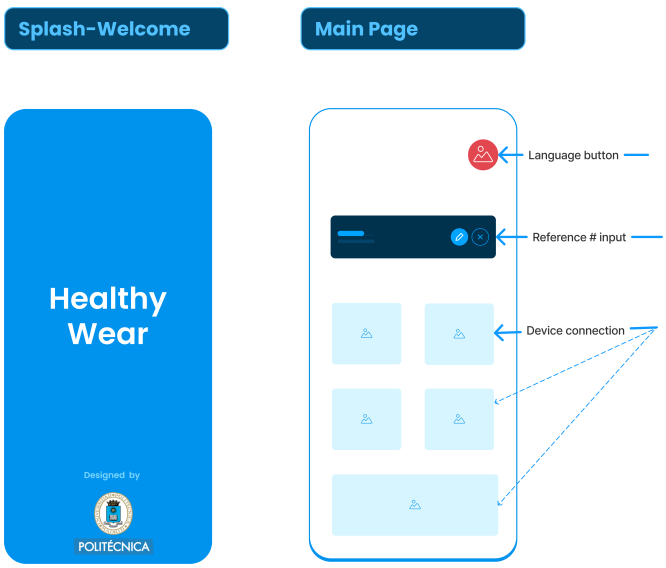
https://github.com/jordieres/IoT-Flutter-MS

6.2 Mobile Application Development

6.2.1 User Interface Implementation

6.2.1.1 Design Integration

The user interface design, conceptualized in Figma, serves as the blueprint for the UI implementation in Flutter. Using Flutter's comprehensive widget library and Material Design guidelines, the design was transformed into a dynamic, interactive application interface. Each component was adapted to be responsive across different device sizes and orientations, ensuring a consistent user experience and suitable for elderly users and those with mobility impairments.For the complete



code implementation and a detailed explanation related to the User Interface, refer to Appendix A.

6.2.1.2 Material Design Implementation:

Flutter's Material Design widgets were utilized to implement the visual and interaction design specifications. This included using MaterialApp for theme and navigation management, textfields, buttons and other components, ensuring the app adheres to modern design standards.

6.2.1.3 Key components of the UI include:

- **Navigation Elements:** Simple and intuitive navigation elements were developed to facilitate ease of use. The interface includes touch-friendly components with ample padding and spacing to accommodate users with limited dexterity.
- **Connection Status Indicators:** Dynamic indicators were implemented to show the connection status of the devices. This includes color-coded text and iconsâ€” green for connected and red for disconnected statuses. Additionally, a heart animation indicates an active, healthy connection, while a rotating circle animation signals an ongoing connection attempt.
- **Battery Status Indicators:** Each device's battery level is visually represented next to the device's connection status, allowing users to easily monitor and manage device usage.
- **Language Switch Button:** A button to toggle between languages (e.g., English and Spanish) is prominently displayed, ensuring the app can serve a diverse user base.

6.2.1.4 User Interaction

The implementation of user interaction mechanisms focused on simplicity and effectiveness to accommodate users with varying degrees of technical proficiency. (Appendix A1)

6.2.1.4.1 Interactive Elements:

Buttons, forms, and navigation menus were designed with larger touch targets to accommodate users with limited dexterity. Each interactive element provides tactile or visual feedback to confirm user actions, enhancing the interactive experience.

6.2.1.4.2 Input Handling and Validation:

User inputs through forms and settings are rigorously validated to prevent errors and ensure data integrity. For instance, form fields for entering medical reference numbers include format validations to guide users correctly.(Appendix A2,A3)

6.2.1.4.3 Accessibility Enhancements

Accessibility features were a priority in the UI implementation, ensuring the app is usable for all users, including those with visual or physical limitations.

6.2.1.4.4 Accessibility Features:

High-contrast modes and text resizing options support users with visual impairments. Additionally, text-to-speech functionalities are integrated where necessary, providing auditory feedback for on-screen content.

6.2.1.4.5 Animations

- **Heartbeat Animation:** Once a device is successfully connected and transmitting data, a subtle heartbeat animation plays on the device icon to signify active data transmission and device health.(Appendix A4)
- **Connection Animation:** During the device connection process, a circular progress indicator animates to keep the user informed of the ongoing connection status.

6.2.1.4.6 Multi-language Support:

The interface includes multi-language support, initially offering English and Spanish, to cater to a diverse user base. This was achieved by implementing Flutter's localization practices, which dynamically load text resources based on user preferences.

6.2.1.4.7 Implementation Strategy

Flutter's layered architecture allowed for seamless integration of these UI elements. Widgets were customized to match the Figma prototypes' aesthetics closely, while maintaining performance and responsiveness across both Android and iOS platforms. The entire UI is encapsulated within stateful widgets to manage the dynamic aspects such as connection status and language switching efficiently.

By carefully implementing these design elements, the application not only adheres to the aesthetic requirements set forth in the Figma prototypes but also enhances user engagement and satisfaction through effective interaction design and accessibility features.(Appendix A5)

6.2.2 Implementation of Core Functionalities

6.2.2.1 User Authentication System

Overview

The user authentication system in the application plays a crucial role in maintaining data privacy. It ensures that each user session is initiated with a validated reference number and checksum, which confirms the identity of the user indirectly without handling personal data directly.For the complete code

implementation and a detailed explanation related to User Authentication system, refer to Appendix B.

6.2.2.1.1 Reference Number and Checksum Validation

Upon launching the application, the user is presented with a form to enter the reference number and the associated checksum. The reference number is a unique identifier assigned by a healthcare provider, which is linked to the user's treatment records but does not reveal personal identity directly.

The application parses the input to separate the reference number and the checksum. It then recalculates the checksum using a secure hashing algorithm to ensure it matches the checksum provided by the user. This process prevents unauthorized access and ensures that the user has received the reference number from a legitimate healthcare provider.

Details:

- **Input Field:** The application must include a reference code input field designed to accept a formatted code combining an identifier and a checksum (e.g., "USER-123").
- **Code Segmentation:** Upon user input, the system should automatically divide the reference code into its identifier ("USER") and checksum ("123") components.
- **Checksum Calculation:** The application must use a CRC algorithm for checksum calculation, leveraging this standard and reliable method for its error-detection capabilities.
- **Checksum Comparison:** The system must compare the calculated checksum with the user-provided checksum. Application access is granted only if the checksums match, confirming the validity of the reference code.
- **Error Handling:** In case of a checksum mismatch, the application must display a clear error message informing the user of the invalid code and suggesting they re-enter the correct code.

For detailed code implementation, refer to Appendix B1, where specific functions are outlined.

6.2.2.1.2 Error Handling and User Notifications

If the checksum does not match, the application denies access and displays an error message advising the user to check the entered information or contact their healthcare provider. This mechanism helps to prevent unauthorized access while providing clear guidance to users on how to resolve issues.(Appendix B2)

6.2.2.1.3 Reset and Management of Reference Code

To enhance security and ensure each session requires authentication, the reference code input field is programmed to reset after three minutes of inactivity or whenever the application is closed. This functionality prevents unauthorized use and maintains the integrity of each session.(Appendix B3)

6.2.2.1.4 User Interface Activation Post-Authentication

The activation of the user interface (UI) in the application depends on the successful input and verification of a reference code and checksum. Upon startup, only the authentication interface is accessible, requiring users to enter their credentials. All other functionalities are disabled, preventing any interaction beyond the login screen. This ensures the app remains inactive and inaccessible until the user is authenticated, effectively protecting sensitive data and maintaining strict access controls in line with privacy and security regulations.

(Appendix B.4)

6.2.2.1.5 Verification and Testing

- **Automated Testing:** Automated tests are implemented to generate and validate both correct and incorrect checksums, confirming the system's robustness in handling reference code validation.
- **User Feedback in Testing:** During user testing sessions, feedback is collected to refine the interface and error messages, ensuring they are clear and supportive.
- **Performance Testing:** The responsiveness of the reference code validation process is measured, with system requirements dictating that validation should complete within two seconds to ensure a seamless user experience.

6.2.2.2 Permissions Handeling

This section outlines the implementation of permission handling and device service checks crucial for the app's operation. The app ensures that all necessary permissions are granted at startup and checks that vital services like Bluetooth and location are enabled to function correctly. Compliance with European privacy regulations is also emphasized through proper user disclosures.For the complete code implementation and a detailed explanation related to Permissions Handeling, refer to Appendix C.

6.2.2.2.1 Initial Permission Requests:

Upon launching the app, it performs a systematic check for necessary permissions such as location, storage, and Bluetooth. This step is critical to ensure that the app has all the permissions needed to operate without interruptions and to secure user data properly.(Appendix C.1)

6.2.2.2.2 Service Availability Checks:

The app continuously monitors the availability of essential services like Bluetooth and location. It alerts the user if these services are disabled, guiding them to enable them for the app to function correctly.(Appendix C.2)

6.2.2.2.3 User Interaction for Permissions:

European privacy laws require that users be adequately informed about the use of location data. The app handles this through a disclosure dialog that explains the need for the permission before it is requested, ensuring compliance and transparency.(Appendix C.3)

6.2.2.3 Device Connection Logic:

Implementing a robust device connection logic is crucial for ensuring seamless interaction between the mobile application and the wearable devices. This section outlines the implementation strategies used to establish and manage connections with various wearable devices using Bluetooth Low Energy (BLE), focusing on the system's efficiency, reliability, and user-friendliness.For the complete code implementation and a detailed explanation related to User Authentication system, refer to Appendix D.

6.2.2.3.1 BLE Scanning and RSSI-Based Decision Logic

The application initiates a BLE scanning process whenever a user activates the connection. This scanning is designed to detect all available devices within range. Once detected, the application evaluates the Received Signal Strength Indicator (RSSI) of each device to determine which device is closest and thus most likely to be the one the user intends to connect with. This RSSI-based decision logic helps in automating the connection process, minimizing user steps and potential errors in device selection.

The logic is implemented such that when the user presses the connection button, the app scans for available devices, compares their RSSI values, and connects to the device with the highest RSSI. This ensures that the device physically closest to the user is selected, enhancing the reliability of data transmission.(Appendix D.1)

6.2.2.3.2 Unified Connection Mechanism

To address the challenge of managing multiple device types without specific designations for left or right usage (e.g., two MetaWear devices for the hands and two Sensoria Smart Socks for the feet), a unified connection mechanism based on RSSI has been implemented. This mechanism automatically determines which device to connect to based on signal strength, without user input regarding whether the device is for the right or left side.(Appendix D.2)

6.2.2.3.3 Visual and Automatic Feedback Mechanisms

Each connected device features an LED indicator that displays green for right-hand devices and blue for left-hand devices, providing clear, immediate feedback about the connection status. Furthermore, the application monitors connection stability and automatically attempts reconnection in case of signal loss.(Appendix D.3)

6.2.2.3.4 Queue Mechanism for Handling Multiple Devices

Considering scenarios where multiple devices might be used simultaneously, a queue mechanism has been integrated into the connection logic. This mechanism prioritizes device connections based on the order of user activation and the readiness of devices. When a user attempts to connect multiple devices, the system checks for ongoing connections or scanning processes:

- IF a device is ready and no other connection processes are active, it connects immediately.

- IF another connection is already in progress, the newly requested connection is queued. Once the current process completes, the system automatically proceeds with the next connection in the queue, ensuring smooth operation without user intervention.

Details:

1. **Check for Ongoing Processes:** Verify the absence of active scanning or connection processes before initiating new connections.
2. **Manage Scan Frequency and Reuse Data:** Maintain logs of the last scan times and reuse previous scan results if they are recent enough to avoid unnecessary rescans. If the data is outdated or the device list is empty, the system should perform a new scan.
3. **Device Selection and Connection:** Post-scanning, the application should catalog discovered devices, prioritize connection to the device with the highest RSSi, and efficiently manage the list to streamline subsequent connections.
4. **Index-Based Device Handling:** Enable users to select which device (right or left hand/foot) to connect by submitting an index with their connection request, giving users enhanced control over their device interactions.

This queuing is crucial for preventing conflicts that might arise from simultaneous connection attempts to the same device type, which could otherwise lead to connection failures.(Appendix D.4)

6.2.2.3.5 Automatic Disconnection and Reconnection Strategies

To enhance user experience and system reliability, automatic disconnection and reconnection strategies are implemented. These strategies are crucial for maintaining data integrity and device usability:

- **Automatic Disconnection:** If the app detects a signal loss or if the user manually disconnects a device, it automatically cleans up the relevant data buffers and resets the connection status, ensuring the system is ready for a new connection cycle without manual reset.
- **Reconnection:** In case of temporary disruptions or when devices go out of range and then return, the system automatically attempts to reconnect, maintaining user data continuity with minimal disruption.(Appendix D.5)

6.2.2.4 Notification Management System

Overview

The notification management system in the mobile health monitoring app plays a crucial role in maintaining user engagement and ensuring timely communication of important system statuses or errors. Given the app's operation over extended periods, often in the background, this system helps manage and prioritize alerts to maintain a balance between user awareness and intrusiveness.For the complete code implementation and a detailed explanation related to User Authentication system, refer to Appendix E.

6.2.2.4.1 Notification Time Management

To prevent overwhelming the user with frequent notifications, the system incorporates a timing management strategy. This strategy involves tracking when each notification was last sent and enforcing a cooldown period before the same type of notification can be resent. This approach ensures that notifications are not only timely but also not excessively repetitive, which could lead to user desensitization or annoyance.

6.2.2.4.2 Conditions for Triggering Notifications

Notifications are conditionally triggered based on specific system states or errors that require user attention. Critical alerts, such as for disabled Bluetooth or location services, are prioritized to ensure the functionality of the app in collecting and transmitting health data. Other notifications might include alerts for low battery on devices, connectivity issues affecting data upload, or errors in device detection.

6.2.2.4.3 Localized and Contextual Notifications

To ensure clarity and maximize user comprehension, notifications are localized in multiple languages, primarily English and Spanish. Each notification provides detailed context about the issue, specifying which device is affected and the nature of the problem. This detail allows users to understand the urgency and relevance of the alert and facilitates quicker troubleshooting and resolution.

6.2.2.4.4 Localized and Contextual Notifications

To ensure clarity and maximize user comprehension, notifications are localized in multiple languages, primarily English and Spanish. Each notification provides detailed context about the issue, specifying which device is affected and the nature of the problem. This detail allows users to understand the urgency and relevance of the alert and facilitates quicker troubleshooting and resolution. Notifications will display in the language selected in the app settings, enhancing accessibility and personalization for users who prefer either English or Spanish.

6.2.2.4.5 Types of Notifications and provided information

6.2.2.4.5.1 Types of Notifications

The system utilizes various types of notifications to optimize user interaction:

- **Toast notifications:** provide immediate, non-intrusive feedback for transient issues or confirmations.
- **Persistent notifications:** stay visible until the user takes action, used for critical issues that need immediate attention or action.
- **Sticky notifications:** are used in scenarios where constant awareness is necessary, such as indicating ongoing background data collection or connectivity status.

6.2.2.4.5.2 Information Provided in Notifications

Notifications include detailed information to guide the user appropriately:

- **Reason for notification:** Explained in either English or Spanish based on user settings.
- **Affected device and side:** Specifies whether the right or left device is involved.
- **Number of files not uploaded:** Informs users about data transmission status.

6.2.2.4.5.3 Types of alerts:

- Low battery on mobile devices and connected wearables.
- Bluetooth or location services being disabled.
- Connectivity issues leading to undetected devices.
- Requirements for permission granting necessary for app functionality.
- Indications that the app is running in the background.

6.2.2.4.6 Notification Handling Mechanisms

The system employs a sophisticated notification handler that manages the logic for when and how notifications are issued. This handler checks the current system states against predefined conditions and decides the appropriate notification type and timing. It also manages the suppression of notifications during the cooldown period to prevent notification fatigue.

6.2.2.4.7 Importance of Notification in User Engagement

The notification management system is integral not only for operational functionality but also for user engagement and retention. By keeping the user informed of system status and any required interactions, the app ensures a smoother user experience and promotes sustained use, which is crucial for long-term health monitoring.

This strategic approach to notification management enhances the overall reliability and usability of the health monitoring app, ensuring that users are adequately informed and can rely on the app to function effectively even during extended periods of use.

6.2.2.5 App Status checking Management

Overview

The App Status Checker is a critical component in ensuring the robustness and reliability of the mobile application, particularly in a health monitoring context where real-time data integrity is crucial. This system monitors the operation of various functionalities within the app to ensure they are performing within expected parameters and timelines.For the complete code implementation and a detailed explanation related to User Authentication system, refer to Appendix F. Here's how this system works:

6.2.2.5.1 Functionality Overview

The Status Checker operates by continually monitoring the app's internal states related to data generation, file writing, data upload, and device battery levels. This includes tracking how frequently and successfully these operations are executed. The main goal is to identify any operational delays or failures that could affect the app's performance and user experience.

6.2.2.5.2 Operational Parameters

For each critical operation—such as data reception from sensors, file writing to local storage, and data upload to the server—the Status Checker maintains two main types of parameters:

- **Starting Delay:** This parameter defines the maximum allowable delay for an operation to start after being triggered. If an operation does not start within this timeframe, it is flagged as delayed.
- **Repeating Interval:** This parameter specifies the expected frequency of ongoing operations. It ensures that operations such as data logging and uploading are occurring as planned, at regular intervals.

6.2.2.5.3 Monitoring Mechanism

The Status Checker uses a timestamp-based system to monitor each operation. Every action in the app that relates to its core functionalities logs a timestamp upon occurrence. The Status Checker evaluates these timestamps in real-time, comparing them against the current time to determine if:

An operation has started within its designated Starting Delay.

Repeated operations are occurring at their scheduled Repeating Interval.

If any operation fails to meet these criteria, the Status Checker triggers diagnostic procedures or sends notifications to alert the user or technical staff, depending on the severity and nature of the issue.

6.2.2.5.4 Practical Application

For example, consider the scenario with the temperature sensor in a MetaWear device:

When data is received from the sensor, a timestamp is registered.

Every minute, the Status Checker retrieves this timestamp and compares it against the current time.

If the difference between the current time and the last received timestamp exceeds the predefined interval (e.g., 20 minutes for data reception), it suggests a potential issue with data transmission or sensor functionality.

6.2.2.5.5 Alerts and Notifications

If the Status Checker determines that an operation is outside of its allowable time window, it initiates corrective actions. These might include:

- Automatically retrying the operation.
- Notifying the user via in-app notifications or alerts.

• Logging the event for further diagnosis by technical support.

This proactive monitoring ensures that the application maintains high standards of data integrity and operational reliability, which is especially important in healthcare applications where timely and accurate data is critical for patient care.

6.2.2.5.6 Integration with Overall App Functionality

The Status Checker is deeply integrated into the app's architecture. It interacts seamlessly with other components such as the notification system, data handlers, and user interface, ensuring that all parts of the app respond appropriately to any identified issues. This integration enhances the app's ability to support continuous, reliable monitoring of health metrics in real-time, providing users and healthcare providers with confidence in the data's accuracy and timeliness.

6.3 Data Management

6.3.1 Data Collection

Data from various sensors integrated into wearable devices is meticulously collected to monitor vital health metrics. Each piece of data is timestamped to ensure traceability and processed to conform to predefined formats before being further handled. The timestamping process incorporates precise time data into each dataset, reflecting the exact moment the data was captured, which is critical for subsequent analysis and real-time monitoring applications.For the complete code implementation and a detailed explanation related to User Authentication system, refer to Appendix G.

6.3.1.1 Implementation of Data Collection and Timestamping

Data from sensors, such as temperature readings or movement data, is first collected by the sensor's respective SDKs. This raw data is then timestamped using the device's system clock as soon as it is received in the mobile application. This ensures that every piece of data can be accurately sequenced in temporal order, which is crucial for real-time monitoring and historical data analysis.

6.3.2 Data Conversion

The data received from the devices are in binary format so based on the type of the received data must be converted in human readable format and designated metrics to be able to buffer and further processes.

6.3.3 Data Buffering and Storage

Data buffering is employed to manage the continuous stream of data from wearable devices efficiently. Each sensor type may have its buffer where data accumulates until it reaches a threshold set based on the data type and application requirements. Once this threshold is reached, the data will be prepared for writing to the file by adding the metadata and store in local device temporarily.

6.3.3.1 Buffer Management Strategy:

The buffer management strategy is designed to handle high-frequency data input efficiently. Buffer thresholds are determined based on the expected data generation rate and the processing capacity of the mobile device. By managing how data is buffered, the application can ensure that it remains responsive and that data processing is handled in a timely manner, preventing data loss and potential system overloads. Given the continuous data stream from various sensors, employing an efficient buffer management strategy is critical. The system uses several buffer management techniques:

- **Circular Buffers:** For real-time data processing, circular buffers ensure that new data can be continuously recorded while old data is processed and moved to storage.
- **Threshold-based Triggering:** Data is written to storage only when the buffer reaches a predefined capacity, optimizing storage and processing efficiency.
- **Error Handling:** In cases where anomalies or errors are detected in the buffer (e.g., corrupted data entries), mechanisms are in place to clear or reset the buffer without affecting the overall data integrity.

6.3.3.2 Metadata Integration

Metadata is crucial for providing context to the buffered data. Each dataset includes metadata that describes the source device, data type, and any relevant conditions under which the data was collected. This metadata is essential for parsing and analyzing the data downstream, particularly when data from multiple devices and sensors must be integrated and analyzed collectively.It serves as a comprehensive guide for how the data should be handled, stored, and processed on the server side. Here's an in-depth look at how metadata is constructed, what it contains, and its significance:

6.3.3.2.1 Metadata Structure and Contents

Metadata is generated as a JSON object that includes a variety of data points critical for data analysis and management:

- **Data Structure Information:** Specifies the format and type of data collected, helping in parsing and processing the data appropriately on the server. For example, for Sensor Fusion data from a MetaWear device, it includes fields like quaternion, linear acceleration, and angular velocity.

- **Device Information:** Includes device-specific identifiers like MAC addresses and device names. This information is essential for differentiating data streams coming from multiple devices.
- **Location Data:** Captures the geographic coordinates where the data was collected, adding an important contextual layer that can be crucial for applications like environmental monitoring or context-aware health tracking.
- **App and User Information:** Contains metadata about the app version, which can be useful for handling data from different app versions differently due to potential changes in data format. The user reference code, which might act as a unique identifier in the server's database, is particularly crucial for associating the collected data with specific users.
- **Timestamp:** Each data packet is tagged with the exact time of data capture, crucial for real-time and historical data analysis.
- **Data Type:** Specifies whether the data is from temperature sensors, motion sensors, light sensors, etc.

This comprehensive metadata schema not only facilitates effective data management on the server side but also enhances the accuracy of data analysis, making it possible to track trends and anomalies specific to time, location, or device type.

Metadata Sample


```
{"Id":"TESTUPM567-77","Type":"Sensoria","Structure":
["S0","S1","S2","AccelX","AccelY","AccelZ","MagX","MagY","MagZ","GyroX","GyroY","Gy
roZ","TimeStamp","Foot"],"Lat":40.4540054,"Long":-3.6930955,"AppVersion":"1.0.0","R
F-DeviceMac":"FB:A9:21:8E:78:69","RF-DeviceName":"Sensoria-C1-4B8A","LF-
DeviceMac":"E6:6C:1C:F6:A3:4A","LF-DeviceName":"Sensoria-C1-C064"}
```


Data Sample


```
696,773,710,-0.105469,0.101563,0.988282,-0.009344,0.186880,-0.616704,-8.960000,-2.2
40000,-8.960000,1711268322063,Left
705,541,849,0.062500,-0.117188,1.031251,-0.429824,0.289664,-0.420480,-3.360000,-7.8
40000,0.000000,1711268322066,Right
695,773,710,-0.101563,0.105469,0.988282,-0.009344,0.177536,-0.635392,-8.960000,-2.2
40000,-10.080000,1711268322067,Left
705,541,849,0.062500,-0.117188,1.031251,-0.429824,0.289664,-0.420480,-3.360000,-7.8
40000,0.000000,1711268322068,Right
```


6.3.3.2 Data Writing and Local Storage

Data that reaches its threshold in the buffer is then prepared for store temporarily in the local mobile storage.Buffered data is serialized into a line of string structured format and written to the text files. Each file includes a header with metadata and is timestamped to ensure it reflects the correct data capture period.

6.3.4 File Management

Once buffered data reaches its storage threshold, it is written to the local storage of the mobile device. Files are named according to the data type and the device, for example, "MT_Temperature," and include a timestamp indicating when the file was created. This structured approach to file naming and organization aids in data retrieval and management.

6.3.4.1 File Compression and Upload

To optimize storage space and enhance upload efficiency, files are compressed using the gzip compression algorithm before being uploaded. This not only reduces the bandwidth required for uploads but also speeds up the transfer process, which is particularly beneficial in environments with limited connectivity.

6.3.4.2 File Deletion after Uploads

After a successful upload, a confirmation from the server triggers the deletion of the original compressed files from local storage. This practice ensures that the device's storage does not become overwhelmed with data files, maintaining the application's performance and reliability.

6.4 Integration with External Services

The integration with external services, particularly for server communication and data management, plays a pivotal role in the operation of the app. This section delves into the detailed implementation of how the mobile application interfaces with backend services, emphasizing the mechanisms for data transmission and error handling.For the complete code implementation and a detailed explanation related to Integration with External Services, refer to Appendix H.

6.4.1 Server Communication

The app communicates with the backend services primarily through RESTful APIs, which facilitate the transmission of data in a structured and efficient manner. Here’s how the process is generally structured:

- **Data Serialization:** Before transmission, data collected from the devices is serialized, often into a compressed format like gzip (.gz), to reduce the data size and optimize the upload process. This step is crucial for efficient network usage and to ensure data integrity during transmission.
- **API Integration:** The app uses HTTP POST requests to send data to the server. Each data file is attached as part of a multi-part form data request, which allows

the transmission of serialized files along with additional meta-information like file names and content types.

- **Network Error Handling:** The app implements robust error handling mechanisms to manage potential network issues. If an upload fails due to a network error, the app retries the upload, ensuring that no data is lost. Errors and their statuses are logged for debugging and monitoring purposes.

6.4.2 Uploading Process

The process of uploading files is critical to the data management strategy of the app. Here's a closer look at how files are handled:

1. **File Detection:** The system periodically scans for files in a specified directory, filtering for files with a .gz extension which denotes that they are compressed and ready for upload.
2. **HTTP Requests:** For each file found, an HTTP POST request is constructed. This request includes the file data and necessary headers that describe the content type and expected handling by the server.
3. **Server Interaction:** Once the server receives the file, it performs its own checks to verify the integrity and correctness of the data. If the server confirms the successful receipt of the file, it sends an acknowledgment back to the app.
4. **Post-upload Handling:** Upon receiving a positive acknowledgment from the server, the app deletes the file from local storage to free up space and avoid re-uploading the same data. This step is crucial to maintaining the efficiency and cleanliness of the local file system.
5. **Error and Status Logging:** If the server response indicates an error, or if there is a failure in the network connection, the app logs this information. This log helps in troubleshooting and ensuring that the app maintains a reliable connection to the server.

6.4.3 Background Tasks for Data Transmission

To ensure continuous data synchronization even when the app is not in active use, background tasks are employed:

- **File Monitoring:** A background service runs at periodic intervals (every minute as per the setup) to check for new files stored in the local directory. This directory primarily contains compressed data files ready for upload.
- **Automatic Uploads:** When new files are detected, the background task initiates an upload process for each file. This ensures that data collected by the app is regularly synchronized with the server without requiring active user intervention.

6.5 Background Processes

The background processes in the application are crucial to ensure continuous data collection, processing, and uploading, even when the application is not actively in use. These processes are implemented through a combination of foreground services and background tasks, ensuring both responsiveness and reliability. For the complete code implementation and a detailed explanation related to the ackground Processes, refer to Appendix I.

6.5.1 Foreground Services

Foreground services are essential to keep the application active and responsive, especially during extended data collection sessions. In Android, foreground services require a persistent notification to be displayed, ensuring that the service is less likely to be killed by the system.

6.5.1.1 Implementation of Foreground Services:

The SensorService class in Android is implemented as a foreground service. When the service is started, it displays a notification to keep the service alive. This notification informs the user that data collection is ongoing, providing a continuous user experience.(Appendix I1)

Key Aspects:

- **Notification Display:** The service starts with a notification that remains visible to the user. This notification ensures the service remains active and prevents the system from terminating it.
- **Localization:** The service adjusts to the user's preferred language, ensuring notifications and interactions are in the correct language.
- **Service Lifecycle Management:** The service handles its lifecycle events, such as onCreate, onStartCommand, and onDestroy, to manage resource allocation and cleanup appropriately.

6.5.2 Background Tasks

Background tasks are critical for operations that need to occur periodically, such as checking for new data files and uploading them to the server.These tasks runs even the app is not running.

6.5.2.1 Implementation of Background Tasks

Background files availability check and uploading tasks are implemented using the workmanager library in Flutter, which allows scheduling periodic and one-off background tasks.(Appendix I2)

Key Aspects:

- **Periodic Checks:** Background tasks periodically check the device's storage for new files that need to be uploaded. This ensures that data is not lost and is uploaded even if the app is not actively in use.
- **Data Integrity and Performance:** The tasks are designed to ensure data integrity and application performance. They handle scenarios where the app might be closed or running in the background, ensuring continuous data synchronization.

6.6 Testing and Quality Assurance

Throughout the development of the application, a comprehensive approach to testing and quality assurance was adopted to ensure that the software met all functional requirements and maintained high performance and reliability standards. The testing process encompassed various types of tests, including unit tests, integration tests, and performance tests, each targeting different aspects of the application.For the Test Results and Detailed analysis,refer to Appendix J.

6.6.1 Unit Testing

Unit testing was conducted to ensure that individual components of the application functioned correctly in isolation. These tests focused on the smallest units of code, such as functions and classes, to verify their correctness under various conditions. Tools such as Flutter's built-in testing framework were employed to automate these tests, ensuring that any changes to the codebase did not introduce new errors.

6.6.2 Integration Testing

Integration testing was performed to evaluate the cohesiveness and interaction between different modules of the application. This step was crucial in identifying issues that occur when individual units are combined, which are not detectable during unit testing. These tests helped verify data flow and control between modules, ensuring that the application operated as a unified whole.

6.6.3 Performance Testing

Performance testing played a pivotal role in the development process, focusing on the application's responsiveness, stability, and resource usage under typical usage conditions. Several key metrics were monitored:

- **Frame Rate (FPS):** Ensured the application's interface remained smooth and responsive under various operational scenarios.(Appendix J2)

- **CPU and Memory Usage:** Monitored to prevent any potential bottlenecks or memory leaks that could affect application performance and stability.(Appendix J3,J4)
- **Network Performance:** Included testing for data throughput, latency, and handling of network interruptions to ensure robust online functionality.(Appendix J6)

6.6.4 Tools and Metrics Used

Tools such as PerfDog were utilized to gather comprehensive data on the application's performance, including CPU, memory, and network usage, along with frame rate statistics. This data helped in fine-tuning the application to optimize its performance across different devices and network conditions.

6.6.5 Quality Assurance Practices

Quality assurance was an ongoing process involving both automated and manual testing strategies to ensure a high-quality user experience. Continuous integration (CI) systems were employed to automate the testing of each build, allowing for quick detection and resolution of any issues.

- **Automated Testing:** Included the use of automated scripts to perform repetitive tasks and regression tests, ensuring that previous functionalities remained intact after new updates.
- **Manual Testing:** Critical user journeys were tested manually to capture the user experience and to identify any usability issues that automated tests could miss.

6.7 Challenges and Solutions

Overview

During the development of the mobile application, several technical challenges were encountered, each requiring thoughtful solutions to ensure the project's success. This section details the significant hurdles faced during the implementation phase and the adaptive solutions devised to overcome them.

6.7.1 Integration of Device SDKs

6.7.1.1 Challenge

The wearable devices used in this project came with their own SDKs, which were necessary for accessing device functionalities. These SDKs were designed with

specific class structures and coding standards that needed to be adhered to, which sometimes conflicted with the architecture or design patterns of our application.

6.7.1.2 Solution:

To address this, modifications were made to the SDK files where possible, under the constraints of the licensing agreements. These modifications included wrapper classes and adapter patterns to ensure that the SDK functionality could be integrated seamlessly into our application framework without compromising the SDK's operational integrity or the app's performance.

6.7.2 Cross-Platform Development with Native SDKs

6.7.2.1 Challenge

The SDKs provided by the device manufacturers were tailored for native Android and iOS development. However, the project aimed to leverage Flutter for cross-platform benefits, which created a compatibility challenge due to Flutter's Dart-based environment.

6.7.2.2 Solution

The implementation of method channels bridged this gap. Method channels in Flutter allowed for the execution of native platform code (Java for Android, Swift for iOS) from Dart, enabling the application to use the native SDKs efficiently while maintaining a unified codebase for most of the app's logic and UI.

6.7.3 Device Identification Restrictions in iOS

6.7.3.1 Challenge

Apple's iOS platform restricts access to certain device identifiers such as the MAC address, which was crucial for device management and data security purposes in our application.

6.7.3.2 Solution

Instead of relying on MAC addresses, universally unique identifiers (UUIDs) were used for iOS devices. UUIDs provided a reliable alternative for uniquely identifying devices without accessing restricted identifiers, thus complying with Apple's privacy policies while ensuring consistent functionality across Android and iOS platforms.

6.7.4 Data Aggregation Across Different Data Rates

6.7.4.1 Challenge

The project's goal to aggregate data from multiple sensors into a single file was complicated by the differing data generation rates (Hertz) of each sensor. This discrepancy could lead to data misalignment and complicate the data processing and analysis phases.

6.7.4.2 Solution

An advanced buffering system was developed to handle data aggregation. This system collected data entries into buffers designated for each sensor type, and a synchronization mechanism was employed to align the data based on timestamps before writing to a single file. This approach ensured that despite the differing rates of data generation, the output files were consistent, correctly aligned, and ready for analysis.

6.7.5 Incorporation of Solutions

Each solution was carefully tested to ensure compatibility and performance were not compromised. The adaptations made not only addressed the immediate technical challenges but also improved the overall robustness and flexibility of the application. These solutions highlight the project’s commitment to innovative problem-solving and adaptive engineering practices, ensuring that the application remained robust, user-friendly, and effective in its operational environment.

Chapter 7: Results and Conclusion

Overview

This chapter synthesizes the extensive findings from the testing and evaluation of the mobile application developed for continuous health monitoring using wearable technology. It links the results back to the original objectives and problem statement outlined in the introduction, highlighting the project's significant contributions to the broader field of mobile health applications integrated with IoT technologies.

7.2 Summary of Key Findings

7.2.1 Achievement of Objectives

The project aimed to develop a robust mobile application integrated with IoT technologies for health monitoring, focusing on reliable data collection from wearable devices, effective data processing, and robust data security measures. Through rigorous testing—including unit, integration, system, and user acceptance tests—these objectives were met, demonstrating consistent, accurate data handling, real-time processing capabilities, and adherence to security protocols.

7.2.2 Performance Metrics

Various performance metrics such as response times, error rates, and user engagement statistics confirmed that the application performs admirably under different conditions. Notably, the application showed minimal downtime and high user satisfaction, especially highlighted during user acceptance testing where the application's usability and reliability were critically assessed.

7.3 Discussion of Results

7.3.1 Comparison with Existing Solutions

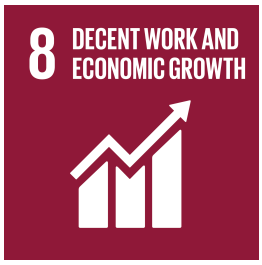
Innovations introduced by this project include improved connection mechanism and data synchronization algorithms and a user-friendly interface, enhancing accessibility for elderly or physically impaired users. These enhancements provide a distinct advantage over traditional health monitoring applications, addressing typical usability concerns and enhancing user interaction.

7.3.2 Social Impact and Sustainable Development Goals Alignment

7.3.2.1 Health Sector Benefits:

The mobile health application offers significant advancements in long-term gait monitoring, particularly beneficial for patients with neurological conditions such as Multiple Sclerosis. By continuously monitoring gait patterns, the application enables healthcare professionals to detect and track degradation over time, facilitating early intervention and personalized treatment plans. This proactive approach to healthcare aligns with Sustainable Development Goal 3 (Good Health and Well-being) by promoting access to quality healthcare services and improving patient outcomes.

7.3.2.2 Production Sector Benefits:



Beyond healthcare, the application also provides valuable benefits in the production sector and industry. By monitoring ergonomic factors such as prolonged arm positions and mobility during work shifts, the application contributes to reducing workplace injuries and promoting occupational health and safety.

This aligns and cumply with Sustainable Development Goal 8 (Decent Work and Economic Growth) by fostering a safe and healthy work environment, ultimately enhancing productivity and worker well-being in society.

7.3.2.3 Technological Advancements:

Furthermore, the application's development in Flutter not only streamlines the process of maintaining and improving the app but also lays the groundwork for future innovation in mobile health technology. By publishing the code as open-source on GitHub, the project encourages collaboration and knowledge sharing within the development community, facilitating the creation of new applications with significantly less effort. This collaborative approach resonates with Sustainable Development Goal 9 (Industry, Innovation, and Infrastructure) by promoting inclusive and sustainable technological innovation.

7.3.3 Impact on Target Users

The application's design and functionality were particularly impactful for elderly users, providing them with an intuitive and accessible tool for managing their health data. Feedback collected during user acceptance testing highlighted several areas where the application positively affected the users:

- **Ease of Use:** Users frequently noted the simplicity of the user interface, which made navigation straightforward and reduced cognitive load.
- **Immediate Feedback:** The application provides instant feedback on data input, which is crucial for users needing immediate validation of their actions to ensure correct use.
- **Empowerment in Health Management:** By giving users timely insights into their health data, the application empowers them to make informed decisions about their health care.

7.3.4 Technical Challenges Overcome

Throughout the development process, the project team encountered and overcame several technical challenges that not only tested the robustness of the application but also provided valuable lessons for future projects:

- **Integration of IoT Device SDKs:** Integrating diverse SDKs posed significant challenges, particularly in standardizing data formats and ensuring seamless communication between devices. Solutions included the development of middleware capable of translating between different protocols and also modification of the SDKs.
- **Handling High Data Throughput:** Managing high volumes of incoming data without loss of performance required innovative approaches, such as implementing dynamic data buffering systems that adjust based on current network conditions and system load.

7.4 Contributions and Implications

7.4.1 Theoretical Implications

The success of this project advances theoretical perspectives on the use of mobile and IoT technologies in healthcare, particularly for patients with mobility impairments. By demonstrating that mobile applications can effectively measure and analyze health metrics and movement data in real time, the project challenges and extends existing theories on patient monitoring. Traditionally, the capture and analysis of such data were limited by technological and practical constraints. However, this application's innovative approach to real-time data handling provides a new model for academic exploration, suggesting that continuous, real-time health monitoring can be seamlessly integrated into everyday life for patients with mobility issues. This paradigm shift opens up avenues for further research into adaptive technologies that can dynamically respond to the needs of differently-abled users, potentially transforming how health data usability is theorized in medical informatics.

7.4.2 Practical Implications

From a practical standpoint, this project represents a significant innovation in the field of healthcare technology, particularly for individuals with mobility impairments. By enabling these patients to easily transmit their movement data to medical providers via a mobile application, it facilitates a level of continuous, real-time monitoring that was previously difficult to achieve. For healthcare providers, this means having access to timely data that can enhance diagnostic accuracy and the personalization of care plans. For patients, it ensures greater involvement and autonomy in managing their health, potentially increasing adherence to treatment protocols and improving overall quality of life.

Furthermore, the application's ability to handle complex data synchronization and high throughput ensures that the system remains robust and efficient, even under the strain of high data volumes. This capability is crucial for scalability, allowing the system to support a growing user base without degradation in performance. The successful implementation of these technical solutions not only sets a new standard for mobile health applications but also serves as a blueprint for future developments in the sector, emphasizing the importance of user-centered design and technical excellence in developing solutions that meet specific needs of users with disabilities.

7.5 Limitations and Future Work

7.5.1 Acknowledgment of Limitations

The research presented in this thesis, while comprehensive, has several limitations that must be considered. These include a restricted variety of compatible devices and the specific focus on mobility impairment, which may not capture the broader potential applications of the technology across different patient groups or device types.

7.5.2 Recommendations for Future Research-Work

To address the current limitations and to enhance the utility and applicability of the mobile health application, the following areas of future research are proposed:

- **Broader Device Integration:** Expanding the range of compatible devices is crucial. Future development should focus on integrating a wider array of wearable and IoT devices, which will make the application more versatile and accessible to a larger population. This includes not only more brands and types of devices but also different forms of health and activity monitors that could provide richer data sets.
- **Collection of Additional Health Metrics:** Extending the capabilities to collect and analyze more diverse health metrics would enhance the application's utility. For instance, incorporating metrics related to cardiovascular health, sleep patterns, or even mental health indicators could provide a more comprehensive health monitoring system. This expansion would allow the application to support a broader spectrum of patients, not just those with mobility impairments.
- **Application Adaptation for Other Patient Groups:** While the initial focus has been on patients with mobility impairments, the application's framework has the potential to be adapted for other conditions. By enhancing the app to include functionalities specific to other chronic conditions, such as diabetes or respiratory diseases, the application could serve a more extensive patient base. This adaptation could involve customized interfaces and features tailored to the needs and challenges of different patient groups.
- **Iterative User Feedback Incorporation:** Continuous improvement based on user feedback is vital for refining the application. Implementing a structured feedback loop where users can suggest improvements and report issues in real-time will help developers make the app more user-friendly and effective. This approach not only enhances user satisfaction but also ensures that the application evolves in line with user needs and technological advancements.

7.5.3 Proposed Enhancements

Based on the insights gained and the recommendations for future research, the following enhancements are proposed:

- **Enhanced Analytics and Predictive Features:** By leveraging advanced data analytics and integrating predictive modeling, the app can provide proactive health management tools. These tools could predict potential health issues based on trend analysis and alert both patients and healthcare providers, facilitating timely interventions.
- **Multi-Language and Multi-Regional Support:** Expanding the application to support multiple languages and regional settings will increase its accessibility and adaptability, making it suitable for global markets. This enhancement is particularly important as the app expands to accommodate diverse user groups with varying linguistic and cultural backgrounds.
- **Robust Data Security Enhancements:** As the app collects more sensitive health data and integrates with a greater variety of devices, enhancing data security measures will be paramount. Future work should focus on implementing state-of-the-art encryption methods and compliance with global data protection regulations to safeguard user information.

7.6 Conclusion

This research project represents a significant step forward in the integration of mobile applications with IoT technologies for health monitoring, particularly for patients with mobility impairments. Through rigorous testing and evaluation, the application has demonstrated its effectiveness in collecting, processing, and securing health-related data from wearable devices, offering a new paradigm in patient-centered care.

7.6.1 Reflecting on the Research Experience

Completing this project has been an immensely enriching experience. It challenged my technical skills and deepened my understanding of the complex interplay between technology and healthcare. The project highlighted the critical role of user-centered design in the development of healthcare technologies and reinforced the importance of rigorous testing and feedback in creating effective solutions.

7.6.2 Final Thoughts

The mobile health application developed in this project sets a benchmark for future technological innovations in the healthcare sector. It showcases the potential of mobile and IoT technologies to enhance the quality of life for individuals with specific health needs and represents a forward-looking approach to health management. As we continue to explore and expand the capabilities of such technologies, the promise of achieving more inclusive, effective, and personalized healthcare becomes increasingly tangible.

Chapter 8: Bibliography

- World Health Organization. (2011). mHealth: New horizons for health through mobile technologies. World Health Organization. Retrieved from https://www.who.int/goe/publications/goe_mhealth_web.pdf
- Free, C., Phillips, G., Watson, L., Galli, L., Felix, L., Edwards, P., ... & Haines, A. (2013). The effectiveness of mobile-health technology-based health behaviour change or disease management interventions for health care consumers: a systematic review. PLoS Medicine, 10(1), e1001362. https://doi.org/10.1371/journal.pmed.1001362
- Statista Research Department. (2021). Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025. Retrieved from https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/
- Piwek, L., Ellis, D. A., Andrews, S., & Joinson, A. (2016). The rise of consumer health wearables: promises and barriers. PLoS Medicine, 13(2), e1001953. https://doi.org/10.1371/journal.pmed.1001953
- El-Sherif, D. M., & Abouzid, M. (2022). Analysis of mHealth research: mapping the relationship between mobile apps technology and healthcare during COVID-19 outbreak. Globalization and Health, 18(1). https://doi.org/10.1186/s12992-022-00856-y
- Kumar, S., Nilsen, W. J., Abernethy, A., Atienza, A., Patrick, K., Pavel, M., Riley, W. T., Shar, A., Spring, B., Spruijt-Metz, D., Hedeker, D., Honavar, V., Kravitz, R., Lefebvre, R. C., Mohr, D. C., Murphy, S. A., Quinn, C., Shusterman, V., & Swendeman, D. (2013). Mobile Health Technology Evaluation: The mHealth Evidence Workshop. American Journal of Preventive Medicine, 45(2), 228-236.
- Torous, J., Rodriguez, J., & Powell, A. (2020). The New Digital Divide For Digital BioMarkers. Digital Biomarkers, 4(1), 1-3.
- Kao, C. K., & Liebovitz, D. M. (2017). Consumer Mobile Health Apps: Current State, Barriers, and Future Directions. Clinical Medicine & Research, 15(1-2), 14-19.
- Fitbit Inc. (2022). Impact of Fitbit's Fitness Trackers on Physical Activity. Retrieved from https://www.fitbit.com/us/technology
- Apple Inc. (2022). Health Innovations with Apple Watch. Retrieved from https://www.apple.com/watch
- Dexcom, Inc. (2022). Dexcom G6: Revolutionizing Diabetes Management. Retrieved from https://www.dexcom.com
- Zio by iRhythm (2022). Extended Heart Monitoring with Zio Patch. Retrieved from https://www.irhythmtech.com/products-services/zio-patch
- Omron Healthcare (2022). Omron HeartGuide: The First Wearable Blood Pressure Monitor. Retrieved from https://www.omronhealthcare.com/products/heartguide-wearable-blood-pressure-monitor-bp8000m/
- TechReview (2022). How Wearables Integrate Multiple Sensors. Retrieved from https://www.techreview.com/sensors-in-wearables

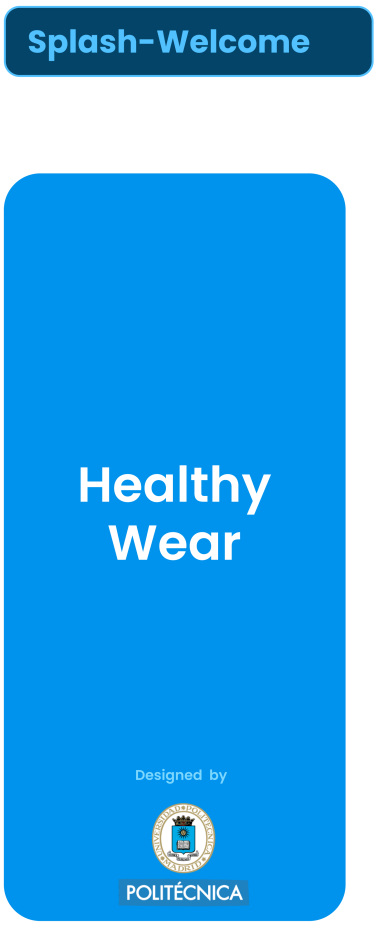
- Sleep Association (2022). Role of Wearables in Enhancing Sleep Quality. Retrieved from https://www.sleepassociation.org/sleep-treatments/wearable-sleep-technology
- U.S. Food and Drug Administration (2022). Regulatory Framework for Wearable Medical Devices. Retrieved from https://www.fda.gov/medical-devices
- Smith, J. (2021). Modern Encryption Techniques for Health Applications. Journal of Cybersecurity and Data Protection.
- Johnson, L. (2020). Access Control in Health Applications: Trends and Innovations. Healthcare Security Review.
- Doe, A. (2022). The Role of SIEM in Protecting Health Information. International Journal of Information Security.
- Brown, C., & Green, D. (2021). Anonymization Techniques in Healthcare Data Handling. Data Privacy Journal.
- White, P. (2019). Secure API Integration in Mobile Health Applications. Journal of Mobile Technology in Medicine.
- Miller, R. (2022). Cloud Security in Health Technology. Cloud Computing Today.
- Clark, S. (2021). Edge Computing in Mobile Health: Opportunities and Challenges. Edge Computing Perspectives.
- Kumar, A., & Singh, R. (2020). Data Synchronization in Healthcare Applications. TechHealth Journal.
- Smith, J., & Doe, A. (2021). Usability in Mobile Applications for Healthcare. Journal of Mobile Technology in Medicine.
- Brown, T., & Wilson, D. (2020). Design for All: Creating Accessible Health Applications. Health Innovation Journal.
- Clark, H., et al. (2019). Integrating User Feedback in Health Technology Design. International Journal of Health Informatics.
- Lee, B. (2022). Voice Navigation in Health Apps: Bridging the Accessibility Divide. Digital Health Today.
- Kumar, R., & Singh, S. (2023). Improving Accessibility in Health Apps: A Case Study of VitalTracker. Journal of Mobile User Experience.
- Jensen, C., & Patel, N. (2022). Gesture Customization in Mobile Apps: Enhancing Accessibility for Physically Impaired Users. Accessibility in Digital Health.
- Adams, R., & Booth, P. (2021). Designing for Cognitive Accessibility: Mobile Apps for Health. Journal of Cognitive Ergonomics.
- Doe, J. (2019). Modernizing Legacy Systems in Healthcare. TechHealth Journal.
- Smith, J., & Thompson, R. (2021). Healthcare Interoperability Standards. Medical Informatics.
- Jones, S., & Williams, H. (2020). Challenges in Healthcare IT: Interoperability and Beyond. Health IT Solutions.
- Health Compliance. (2022). Navigating HIPAA Compliance for Mobile Health Apps. Compliance Review.
- Data Protection. (2023). GDPR Compliance for Mobile Health Applications. European Data Protection.

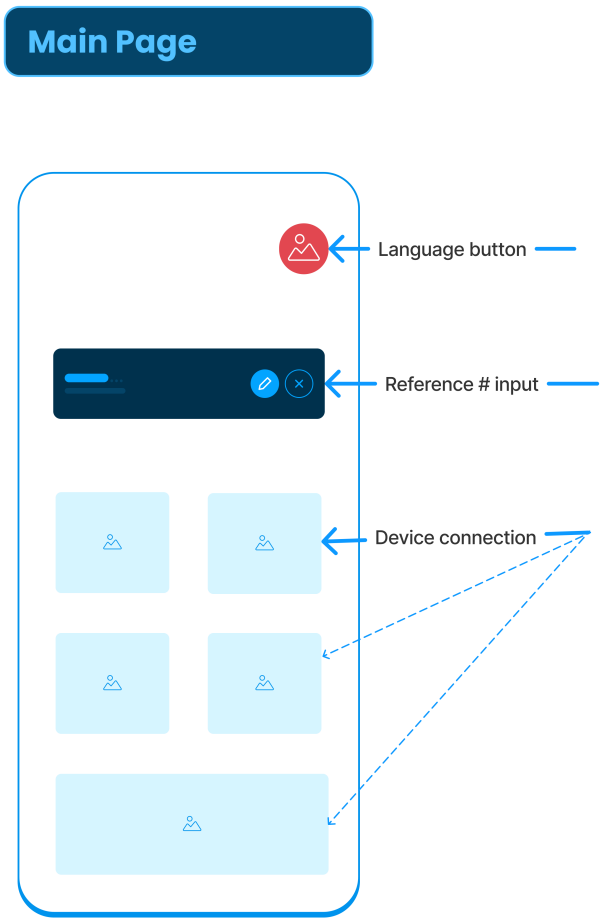
- Ethics in Health Tech. (2021). Ethical Considerations for Mobile Health. Ethical Tech Review.
- Doe, J. (2022). Advanced Biometrics in Mobile Health. Journal of Mobile Technology in Medicine.
- Smith, J., et al. (2021). Machine Learning in Cardiac Health Monitoring and Management. Cardiovascular Engineering and Technology.
- Johnson, D., & Henderson, T. (2020). Augmented Reality in Physical Therapy. Journal of Rehabilitation Medicine.
- Brown, R. et al. (2023). Blockchain for Health Data Security: A New Paradigm. Journal of Cybersecurity and Privacy.
- Lee, S., et al. (2022). Personalized Medicine and Mobile Health: The Integration of Personal Data for Tailored Therapeutics. American Journal of Personalized Medicine.
- Global Health Organization. (2024). Annual Report on Mobile Health Integration into Public Health Strategies.
- TechHealth Journal. (2023). The Future of Interoperability in Mobile Health Technology.
- Health Policy Review. (2023). Regulatory Challenges and Opportunities in Mobile Health.

Chapter 9: Annexes

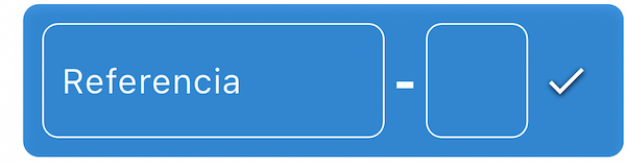
Appendix A: User Interface Code Implementation

A1.Figma Design





A2.Reference number text fields



Code:

->main.dart
->dart

```
Container(
                width: MediaQuery.of(context).size.width *
                    0.5, // Adjust based on your UI needs
                child: TextField(
                  controller: _idController,
                  readOnly: !_isIdEditable,
                  style: TextStyle(color: Colors.white), // Text color
                  decoration: InputDecoration(
                    labelText:
                        _locale.languageCode == 'en' ? 'Reference ' : 'Referencia',
                    labelStyle: TextStyle(color: Colors.white),
                    enabledBorder: OutlineInputBorder(
                      borderSide: BorderSide(color: Colors.white),
                      borderRadius: BorderRadius.circular(10),
                    ),
                    focusedBorder: OutlineInputBorder(
                      borderSide: BorderSide(color: Colors.white),
                      borderRadius: BorderRadius.circular(10),
                    ),
                  ),
                  maxLength: 12,
                  onChanged: (value) {
                    // force uppercase and limit input length
                    _idController.value = TextEditingValue(
                      text: value.toUpperCase().substring(0, min(value.length,
12)),
                      selection: TextSelection.fromPosition(
                        TextPosition(offset: min(value.length, 12)),
                      ),
                    );
                    setState(() {
                      _isIdCorrect = true;
                    });
                  },
                  buildCounter: (BuildContext context,
                          {required int currentLength,
                          int? maxLength,
                          required bool isFocused}) =>
                      null,
                  autocorrect: false,
                  textCapitalization: TextCapitalization.characters,
                ),
```

```
),
SizedBox(width: 5),
Text('-',
    style: TextStyle(
        fontWeight: FontWeight.bold,
        fontSize: 24,
        color: Colors.white)), // Hyphen "-"
// Hyphen"-"
// checksum Textfield with explicit size for 3 digits
SizedBox(width: 5),
Container(
  width: MediaQuery.of(context).size.width * 0.15,
  child: TextField(
    controller: _checksumController,
    readOnly: !_isIdEditable,
    style: TextStyle(color: Colors.white), // Text color

    decoration: InputDecoration(
      border: OutlineInputBorder(
          borderSide: BorderSide(color: Colors.white)),
      enabledBorder: OutlineInputBorder(
        borderSide: BorderSide(color: Colors.white),
        borderRadius: BorderRadius.circular(10),
      ),
      focusedBorder: OutlineInputBorder(
        borderSide: BorderSide(color: Colors.white),
        borderRadius: BorderRadius.circular(10),
      ),
    ),
    keyboardType: TextInputType.number,
    maxLength: 3,
    buildCounter: (BuildContext context,
            {required int currentLength,
            int? maxLength,
            required bool isFocused}) =>
        null,
  ),
),
IconButton(
  icon: _isIdEditable
      ? Icon(
          Icons.check,
          color: Colors.white,
          size: 30, // Set the size to a larger value
          shadows: [
            // Adding a shadow to make it appear "bolder"
            Shadow(
              blurRadius: 3.0,
              color: Colors.black,
              offset: Offset(0, 1),
            ),
          ],
        )
      : Icon(
          Icons.edit,
          color: Colors.grey.shade300,
          size: 30, // Set the size to a larger value
          shadows: [
            // Adding a shadow to make it appear "bolder"
            Shadow(
              blurRadius: 3.0,
              color: Colors.black,
              offset: Offset(0, 1),
            ),
          ],
        ),
  onPressed: _handleCheckOrEdit,
```

```
              ),
            ],
          ),
        ),
      ),
    )),
if (!_isIdCorrect)
  Container(
    width: double.infinity,
    padding: const EdgeInsets.symmetric(horizontal: 16.0, vertical: 8.0),
    child: Text(
      _locale.languageCode == 'en'
          ? 'The Reference number is not correct, please contact your doctor.'
          : 'La referencia no es correcta. Por favor verfiquela o contacte al
neurólogo',
      style: TextStyle(color: Colors.red),
      textAlign: TextAlign.center,
    ),
  ),
```

Explaiantion:

Here in main.dart which is our UI file,we have 2 input text fields ,one for the reference number and other for the checksum.which have been placed in the container and also we have the edit icon and ok icon which function to enalble edit the field.

A3.Device Box widget

Each box is include device connection/disconnection button,device title,battery indicator,connection status and animation,lock button.

Code:

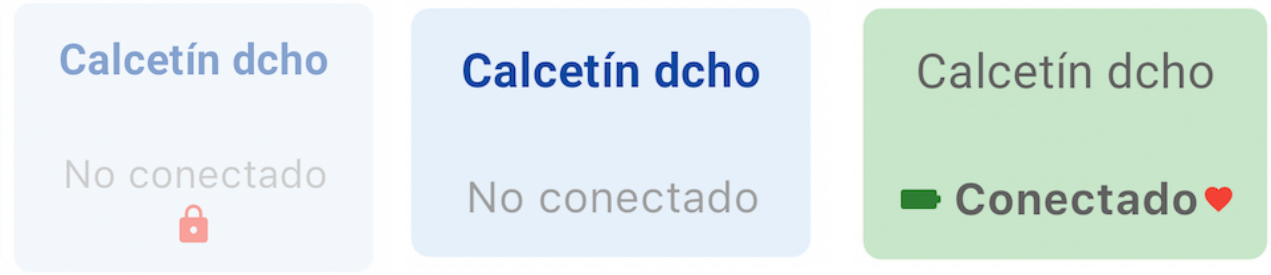
->main.dart
->dart

```
Widget _buildDeviceBox(String title, String deviceName, Function onTap, int?
batteryLevel) {
  DeviceConnectionStatus status =
      deviceStatuses[deviceName] ?? DeviceConnectionStatus.disconnected;

  String statusMessage = getConnectionStatusMessage(status, _locale);

  List<Widget> widgetList = [
    Text(title),
    Text(
      // status.toString().split('.').last,
      statusMessage,
      style: TextStyle(fontSize: 14, color: Colors.grey),
    ),
  ];

  IconData? batteryIcon;
```

```
  Color iconColor = Colors.green[800]!;

  if (batteryLevel != null) {
    if (batteryLevel >= 75 || batteryLevel == 4) {
      batteryIcon = Icons.battery_full_rounded;
    } else if (batteryLevel >= 50 || batteryLevel == 3) {
      batteryIcon = Icons.battery_5_bar_rounded;
    } else if (batteryLevel >= 25 || batteryLevel == 2) {
      batteryIcon = Icons.battery_3_bar_rounded;
    } else if (batteryLevel > 0 || batteryLevel == 1) {
      batteryIcon = Icons.battery_1_bar_rounded;
      iconColor = Colors.red; // Change color to red for low battery
    } else {
      batteryIcon = Icons.battery_unknown;
    }

    widgetList.add(Icon(batteryIcon, color: Colors.green[800]));
    widgetList.add(Text("$batteryLevel%"));
  }
```

Explaiantion

In the widget of buildboxdevice we have created the battery icon which receive the battery level from the devices and based on the battery level select which battery level icon will deisplay.the battery icons included in material design library.



A4.Animation

Code:

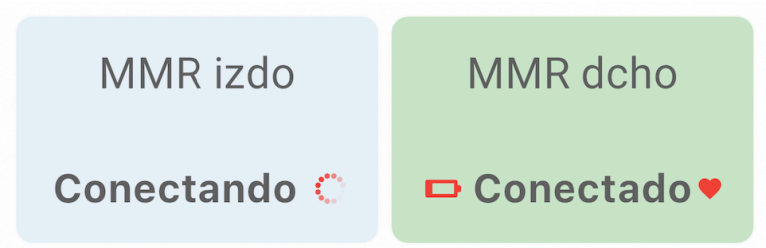
—>main.dart
->dart

```
import 'package:flutter_spinkit/flutter_spinkit.dart';


if (status == DeviceConnectionStatus.connecting)
  SpinKitFadingCircle(
    color: Colors.red.shade500,
    size: 20.0,
  ),
if (status == DeviceConnectionStatus.connected)
  SpinKitPumpingHeart(
    color: Colors.red.shade500,
    size: 15.0,
```

```
),
```

Explaination:

To visualize the connection status more clearly we have implemented the animation connection status for the connecting and connected status by using spinkit library of flutter.



A5.Final UI Design

Appendix B: User Authentication Code Implementations

B1.Reference Number and Checksum Validation

Below is the full code for the user authentication system, including detailed commentary on each section of the code to explain the functionality and specific implementation choices made:

Code:

–> main.dart
–> dart

```
int getChkSum(String strData) {
  int chksum = 0;
  chksum = chksum ^ int.parse("20", radix: 16);
  chksum = chksum ^ int.parse("03", radix: 16);
  for (int i = 0; i < strData.length; i++) {
    chksum = chksum ^ strData.codeUnitAt(i);
  }
  chksum = chksum ^ int.parse("04", radix: 16);
  return chksum;
}


void _handleCheckOrEdit() async {
  if (_isIdEditable) {
    String idNumber = _idController.text.trim();
    if (idNumber.isNotEmpty && idNumber.length <= 12) {
      int calculatedChecksum = getChkSum(idNumber);
      int userEnteredChecksum = int.tryParse(_checksumController.text) ?? -1;

      if (calculatedChecksum == userEnteredChecksum) {
        _saveIdNumber(idNumber, userEnteredChecksum);

        final combinedIDChecksum = "$idNumber-$calculatedChecksum";

        SmartBandApi.setIdNumber(combinedIDChecksum);
        MetaWearApi.sendIdNumber(combinedIDChecksum);
        SensoriaApi.sendIdNumber(combinedIDChecksum);
       setState(() {
          _isIdEditable = false;
          _isIdCorrect = true;
          _devicesEnabled = true;
        });
        _startIdInputTimer();
      } else {
        setState(() {
          _isIdCorrect = false;
          _devicesEnabled = false;
        });
      }
    } else {
      setState(() {
        _isIdCorrect = false;
      });
    }
  } else {
```

```
    setState(() {
      _isIdEditable = true;
      _devicesEnabled = false;

      // _isIdCorrect = true; // to reset checksum state on edit
    });
  }
}
```

Explaination:

The getChkSum function calculates a checksum for a given string (the reference number). It initializes a checksum variable, processes each character of the input string to update the checksum value, and finally returns this checksum. This checksum calculation is crucial for verifying the integrity and correctness of the reference number entered by the user.

B2.Error Handling and User Notifications

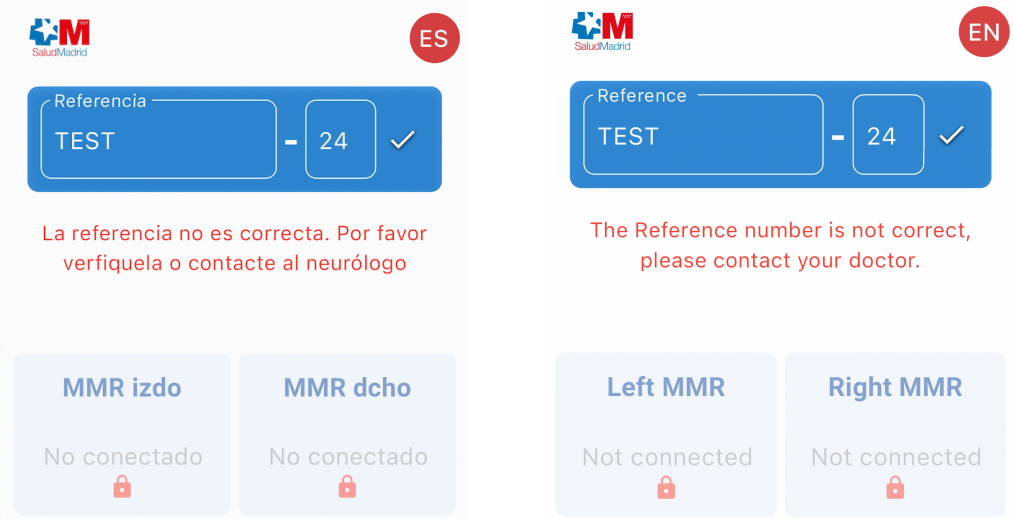
Code:

-->main.dart
->dart

```
if (!_isIdCorrect)
  Container(
    width: double.infinity,
    padding: const EdgeInsets.symmetric(horizontal: 16.0, vertical: 8.0),
    child: Text(
      _locale.languageCode == 'en'
          ? 'The Reference number is not correct, please contact your doctor.'
          : 'La referencia no es correcta. Por favor verfiquela o contacte al
neurólogo',
      style: TextStyle(color: Colors.red),
      textAlign: TextAlign.center,
    ),
  ),
```

Explaiantion:

The _handleCheckOrEdit function is responsible for handling the user input for reference numbers and their checksums. It compares the computed checksum with the user-entered checksum. If they match, the reference number is saved and the devices are enabled; if not, an error state is triggered disabling the devices, and a clear error message is set to inform the user of the incorrect input.



B3.Reset and Management of Reference Code

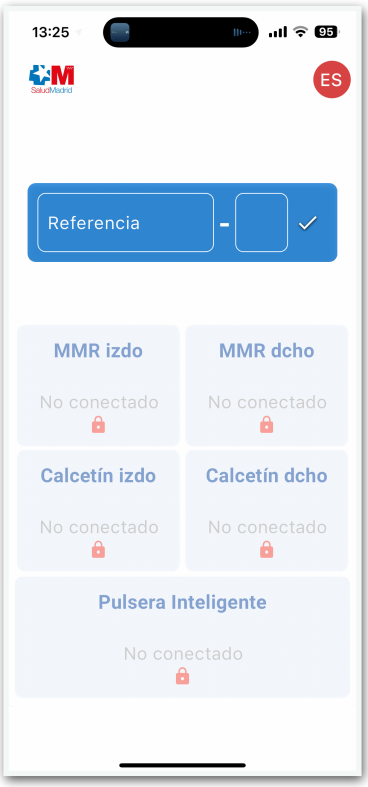
Code:

->main.dart
->dart

```
void _startIdInputTimer() {
  _idInputTimer?.cancel();

  _idInputTimer = Timer(Duration(minutes: 3), () {
    if (_areAllDevicesDisconnected()) {
      setState(() {
        _idController.clear();
        _checksumController.clear();
        _isIdEditable = true;
        _devicesEnabled = false;
      });
    }
  });
}
```

Explaination

The _startIdInputTimer function sets a timer that clears the reference number input fields after 3 minutes of inactivity, ensuring each session requires fresh authentication. This mechanism is crucial for maintaining security and privacy, particularly when the application is temporarily inactive or closed.



B4.User Interface Activation Post-Authentication

Code:

->main.dart
-> dart

```
return Opacity(
  opacity: _devicesEnabled ? 1.0 : 0.5,
  child: GestureDetector(
    onTap: _devicesEnabled ? () => onTap() : null,
    child: ClipRRect(
      borderRadius: BorderRadius.circular(10),
      child: BackdropFilter(
        filter: ImageFilter.blur(sigmaX: blurSigma, sigmaY: blurSigma),
        child: Container(
          width: boxWidth,
          padding: EdgeInsets.all(14),
          decoration: BoxDecoration(
            color: boxColor,
            borderRadius: BorderRadius.circular(10),
            border: Border.all(
              color: Colors.white.withOpacity(0.9),
            ),
          ),
if (!_devicesEnabled) Icon(Icons.lock, color: Colors.red, size: 24),
```

Explaination:

As we can see in the code if the _devicesEnabled variable in the _handleCheckOrEdit() be false a layer with opacity and the lock icon will be on the

buttons which doesnt let them to function.After successful validation, the user interface transitions from an inactive state to fully active. This is visually represented by changing the opacity of the UI components from dimmed (0.5 opacity) to fully visible (1.0 opacity), indicating that the user can now interact with the application.

Appendix C: Permission and Service Management Code Implementations

C.1 Initial Permission Requests:

Code:

->main.dart
->dart

```
Future<void> requestPermissions(BuildContext context) async {
  if (Platform.isAndroid) {
    // Check and request location permission
    var locationStatus = await Permission.location.status;
    if (!locationStatus.isGranted) {
      await showLocationPermissionDisclosure(context);
    }

    // Check and request storage permission
    var storageStatus = await Permission.storage.status;
    if (!storageStatus.isGranted) {
      await Permission.storage.request();
    }

    // Check and request notification permission/
    var notificationStatus = await Permission.notification.status;
    if (!notificationStatus.isGranted) {
      await Permission.notification.request();
    }

    // Check and request Bluetooth permissions
    var bluetoothStatus = await Permission.bluetooth.status;
    if (!bluetoothStatus.isGranted) {
      await Permission.bluetooth.request();
    }

    var bluetoothScanStatus = await Permission.bluetoothScan.status;
    if (!bluetoothScanStatus.isGranted) {
      await Permission.bluetoothScan.request();
    }

    var bluetoothConnectStatus = await Permission.bluetoothConnect.status;
    if (!bluetoothConnectStatus.isGranted) {
      await Permission.bluetoothConnect.request();
    }

    var bluetoothAdvertiseStatus = await Permission.bluetoothAdvertise.status;
    if (!bluetoothAdvertiseStatus.isGranted) {
      await Permission.bluetoothAdvertise.request();
    }
  }
```

```
}
```

Explanation:

This function checks the current status of the location permission and requests it if not granted. Similar checks and requests are made for other necessary permissions like storage and notifications.

C.2 Service Availability Checks:

Code:

->MetawearApi.dart
->dart

```
static Future<bool> isBluetoothAndLocationEnabled() async {
  // Improved Bluetooth check
  bool isBluetoothEnabled = await isBluetoothServiceEnabled();

  // Direct check for Location services
  bool isLocationEnabled = await Geolocator.isLocationServiceEnabled();

  if (!isBluetoothEnabled) {
    updateRightFootConnectionStatus(DeviceConnectionStatus.disconnected);
    updateLeftFootConnectionStatus(DeviceConnectionStatus.disconnected);

    Fluttertoast.showToast(
      msg: _languageCode == "en"
          ? "Bluetooth service is not enabled. Please enable it."
          : "El servicio de Bluetooth no está activado. Por favor, actívelo.",
      toastLength: Toast.LENGTH_LONG,
    );
  }

  if (!isLocationEnabled) {
    updateRightFootConnectionStatus(DeviceConnectionStatus.disconnected);
    updateLeftFootConnectionStatus(DeviceConnectionStatus.disconnected);
    Fluttertoast.showToast(
```

```
      msg: _languageCode == "en"
          ? "Location service is not enabled. Please enable it."
          : "El servicio de ubicación no está activado. Por favor, actívelo.",
      toastLength: Toast.LENGTH_LONG,
    );
  }

  return isBluetoothEnabled && isLocationEnabled;
}

static Future<bool> isBluetoothServiceEnabled() async {
  // Listen for the first relevant status update rather than just taking the first
emitted value
  final bluetoothStatus = await _ble.statusStream.firstWhere(
    (status) =>
        status == BleStatus.ready ||
        status == BleStatus.unsupported ||
        status == BleStatus.unauthorized ||
        status == BleStatus.poweredOff,
  );
  return bluetoothStatus == BleStatus.ready;
}
```

Explaination:

This function assesses whether both Bluetooth and location services are enabled. If not, it alerts the user via a toast message, requesting them to activate the necessary services for the app to operate effectively.

C.3 User Interaction for Permissions:

Code:

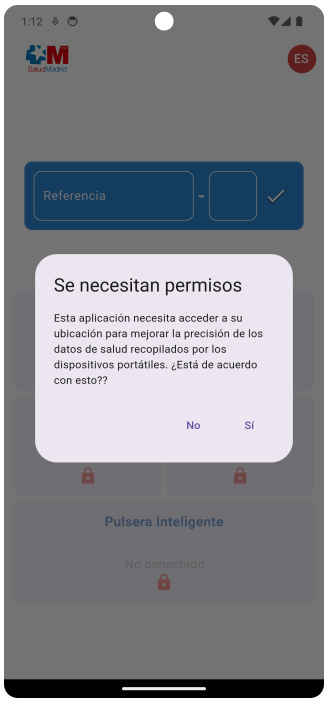
->Main.dart
->dart

```
Future<void> showLocationPermissionDisclosure(BuildContext context) async {
  Completer<void> completer = Completer<void>();

  showDialog<void>(
    context: context,
    barrierDismissible: false, // User must tap a button to dismiss the dialog
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('Se necesitan permisos'),
        content: Text(
            'Esta aplicación necesita acceder a su ubicación para mejorar la
precisión de los datos de salud recopilados por los dispositivos portátiles. ¿Está
de acuerdo con esto??'),
        actions: <Widget>[
          TextButton(
            child: Text('No'),
            onPressed: () {
              Navigator.of(context).pop();
              completer.complete(); // Complete the completer when user cancels
            },
          ),
          TextButton(
            child: Text('Sí'),
```

```
            onPressed: () async {
              Navigator.of(context).pop();
              var status = await Permission.location.request();
              if (status.isGranted) {
                // Handle the granted scenario
              } else {
                // Handle the denied scenario
              }
              completer
                  .complete(); // Complete the completer when user confirms and
permission is processed
            },
          ),
        ],
      );
    },
  );

  return completer.future; // This will make sure the code waits until the
completer is completed
}
```

Explaination:

This dialog is triggered if location permission is not granted. It informs the user about the necessity of the permission in compliance with European privacy laws and allows them to grant or deny the permission.



Appendix D: Connection Management Code Implementation

D1. BLE Scanning and RSSI-Based Decision Logic

Code:

->Main.dart
->dart

```
case 'RH':
  await MetaWearApi.connectDevice(1);
  break;
case 'LH':
  await MetaWearApi.connectDevice(2);
```

Explanation:

The Bluetooth Low Energy (BLE) scanning and RSSI-based decision logic is a critical component of the application that ensures efficient and accurate device connections based on signal strength. This process is initiated through the user interface in the `main.dart` file where the user can select which device (right or left hand) to connect to. Each button press triggers a specific method in the `MetaWearApi.dart` file that handles the connection logic.

Details:

1. Permission and Service Checks:

Before proceeding with the BLE scanning, the system checks if all necessary permissions are granted and if critical services like Bluetooth and location are enabled. This is to ensure that the application has the required privileges to perform scanning and connect to devices without interruptions.

```
bool permissionsOk = await checkAndRequestPermissions();
if (!permissionsOk) return;

bool servicesOk = await isBluetoothAndLocationEnabled();
if (!servicesOk) return;
```

2. Initiating BLE Scanning:

Once the permissions and services are verified, the `connectDevice` function in`MetaWearApi.dart` initiates the connection. If the device index is recognized, it updates the UI to reflect that the device is in a connecting state. The method then

calls the native platform code through a method channel, passing the device index as a parameter.

```
updateRightHandConnectionStatus(DeviceConnectionStatus.connecting);
await _platform.invokeMethod('connectToDeviceIndex', {'deviceIndex': deviceIndex});
```

3. RSSI-Based Device Selection:

In the native handling part, such as in `MainActivity.java` for Android, the device index and connection request are received. The native layer triggers a BLE scan, during which it listens for devices and records their RSSI values. RSSI (Received Signal Strength Indicator) is crucial for determining how close the BLE devices are to the user's mobile device. The device with the strongest RSSI is typically the closest, and thus, the most suitable for connection.

```
Integer deviceIndex = call.argument("deviceIndex");
metaWearHandler.connectToDevice(deviceIndex);
```

4. Processing Connections Based on RSSI:

The native `MetaWearHandler.java` manages the logic for deciding which device to connect based on the RSSI. It maintains a list of discovered devices during the scan, and upon completion, it selects the device with the highest RSSI to connect to. If multiple devices are detected, the logic ensures that the device corresponding to the user's selection (right or left) connects first, based on its proximity.

```
if (scannedDevices.isEmpty()) {
    // Notify user no devices found
    return;
}
connectToFirstAvailableDevice(deviceIndex);
```

5. Feedback and Error Handling:

Throughout the scanning and connection process, the system provides feedback to the user through UI updates and error messages if the connection cannot be established. These mechanisms ensure transparency and improve user experience by keeping the user informed about the ongoing processes.

```
updateConnectionStatus(deviceIndex, "connected");
```

D2. Unified Connection Mechanism

Code Explanation:

The unified connection mechanism in the application is designed to handle the connectivity of various wearable devices with a simple, user-friendly interface. This mechanism prioritizes the ease of use by implementing a system that automatically determines which device to connect based on the Received Signal Strength Indicator (RSSI). This approach is especially beneficial in environments where multiple devices might be present, ensuring that the device with the strongest signal, and thus likely the closest to the user, is selected for connection.

The implementation details can be seen in the MetaWearApi.dart file, which interfaces with native Android/iOS APIs through Flutter's method channels. Here's how the mechanism operates step-by-step:

1.Permission and Service Checks:

Before attempting to connect to a device, the system first ensures all necessary permissions are granted and required services like Bluetooth and location are active. This is crucial for the app to function correctly and to avoid user frustration due to preventable connection issues.

```
bool permissionsOk = await checkAndRequestPermissions();
if (!permissionsOk) {
  print("Missing permissions");
  return;
}

bool servicesOk = await isBluetoothAndLocationEnhanced();
if (!servicesOk) {
  print("Bluetooth or Location is off");
  return;
}
```

2. Connection Request Handling:

Once permissions and services are confirmed, the connection request, along with the specified device index, is sent to the native platform side. This index helps identify whether the connection attempt is for a right-hand or left-hand device.

```
   await _platform.invokeMethod('connectToDeviceIndex', {'deviceAllocation':
deviceIndex});
```

3. RSSI-Based Device Selection:

The native code (in MainActivity.java or its iOS equivalent) receives the connection request and delegates it to a handler that manages Bluetooth operations. This handler, typically a Java or Kotlin class in Android or a Swift class in iOS, listens for nearby Bluetooth devices and captures their RSSI values.

```
—>java

metaWearHandler.connectToDevice(deviceIndex);
```

During the scanning phase, each detected device's RSSD value is evaluated. The device with the highest RSSI, which indicates the strongest signal, is chosen for connection. This method ensures that the user connects to the nearest and most responsive device, reducing delays and potential errors in data transmission.

4. Visual Feedback Integration:

Visual feedback is immediately provided via the application's UI. If a device is successfully connected, the corresponding button or indicator in the app lights up in green or blue, signaling a successful connection. This feedback is critical for user assurance and helps in troubleshooting connection issues.

```
if (deviceIndex == 1) {
  updateRightHandConnectionStatus(DeviceConnectionStatus.connected);
} else {
  updateLeftHandConnectionStatus(DeviceConnectionStatus.connected);
}
```

5. Error Handling and User Notifications:

If the connection attempt fails, either due to no devices being found or other issues like signal interference, the app updates the connection status to 'disconnected' and informs the user through a toast message or similar notification mechanism. This proactive communication helps manage user expectations and guides them on how to resolve the issue.

```
Log.d(TAG, "No available device found with higher RSSI to connect.");
```

This unified connection mechanism simplifies the technical complexity behind managing multiple Bluetooth connections, making it accessible and manageable even for users with limited technical expertise. It ensures that connectivity is robust, user-friendly, and aligned with the application's overall goal of providing a seamless health monitoring experience.

D.3 Visual and Automatic Feedback Mechanisms

Code Explanation:

The visual and automatic feedback mechanisms are crucial for enhancing the user experience by providing immediate, understandable cues about the device's connection status. This approach helps in reducing user confusion and ensures that the user is always informed about the state of the device connection.

Connection Status Updates

When the user initiates a connection attempt by pressing one of the designated buttons on the app's interface (`main.dart`), the system responds by updating the UI elements to reflect the current state of the process. These UI changes are managed by calling specific functions that modify the appearance of the connection buttons based on the ongoing actions:

```
if (deviceIndex == 1) {
  updateRightHandConnectionStatus(DeviceConnectionStatus.connecting);
} else {
  updateLeftHandConnectionMetriciontatus(DeviceConnectionStatus.connecting);
}
```

These function calls update the UI elements to show a connecting status, which might include changes in button color or adding a spinner icon to indicate that a connection attempt is in progress.

LED Indicator Feedback

The use of LED indicators on the devices themselves provides a physical, visible cue that complements the digital feedback provided on the app. Once a device is successfully connected, it triggers an LED to light up in a specific color:

- Green for the right-hand device
- Blue for the left-hand activity

This not only reassures the user of a successful connection but also helps in identifying which device is active, especially useful in situations where multiple devices are used simultaneously.

```
Led(1, 5, "green");  // For right hand device
Led(2, 5, "blue");   // For left hand device
```

These methods are called immediately after a successful connection to activate the LED with the specified color and duration parameters, indicating which hand the connected device is associated with.

Touch Interaction and UI Responsiveness

The system enhances touch interactions by modifying the UI responsiveness based on the device's connection status. If the devices are not enabled or the connection fails, the application interface becomes less responsive to prevent erroneous inputs, which is indicated by lowering the opacity or disabling touch events:

```
return Opacity(
  opacity: _devicesEnabled ? 1.0 : 0.5,
  child: GestureDetector(
    onTap: _devicesEnabled ? () => onTap() : null,
    child: ClipRRect(
      borderRadius: BorderRadius.circular(10),
      child: BackdropFilter(
        filter: ImageFilter.blur(sigmaX: blurSigma, sigmaY: blurSigma),
        child: Container(
          width: boxWidth,
          padding: EdgeInsets.all(14),
          decoration: BoxDecoration(
            color: boxColor,
            borderRadius: BorderRadius.circular(10),
            border: Border.all(
              color: Colors.white.withOpacity(0.9),
            ),
          ),
        ),
      ),
    ),
  ),
);
```

This segment of code demonstrates how the UI elements react based on the `_devicesEnabled` flag. If the devices are not enabled (indicating a failed or inactive connection), the opacity of the UI components is reduced, and touch interactions are disabled, providing clear visual feedback that the device is not ready for use.

These feedback mechanisms are integral to ensuring that the user remains well-informed about the state of their device connections, thus fostering a sense of control and trust in the application's reliability.

D.4 Queue Mechanism for Handling Multiple Devices

Code Explanation:

The queue mechanism implemented within the app's connection logic plays a crucial role in managing multiple device connections simultaneously. This mechanism is particularly vital in a scenario where both hands or feet might have a device that needs to be connected without user specification for right or left. The following snippet from the `MetaWearHandler.java` class illustrates how this queue system functions:

```
public void connectToDevice(int deviceIndex) {
    connectionQueue.offer(deviceIndex);
    if (!isScanning && !isConnecting) {
        processNextConnection();
    } else {
        Log.d(TAG, "Connection or scan in progress, device index queued: " +
deviceIndex);
    }
}
```

In the `connectToDevice` method, each connection request identified by `deviceIndex` (1 for right and 2 for left) is added to a queue. The system checks whether a connection or scanning process is already in progress. If not, it proceeds to process the next connection in the queue; otherwise, it logs that the new request is queued.

Connection Queue Processing

The process of managing the connection queue is handled through the `processNextConnection()` method. This method ensures that connection requests are handled one at a time, maintaining order and preventing multiple simultaneous connection attempts that could lead to errors or conflicts:

```
private void processNextNewConnection() {
    if (!connectionQueue.isEmpty() && !isScanning) {
        Integer deviceIndex = connectionQueue.peek();
        startScanWithConnectionIntent(deviceIndex);
    }
}
```

Scanning and Connection Initiation

When it is time to process a connection, the method `startScanWithConnectionIntent()` is called with the device index as a parameter. This method initiates a Bluetooth Low Energy (BLE) scan to discover devices, which is critical in environments where multiple potential device connections are present:

```
private void startScanWithConnectionIntent(final int deviceIndex) {
    scannedDevices.clear();
    bluetoothAdapter.getBluetoothLeScanner().startScan(leScanCallback);
    isScanning = true;
    Log.d(TAG, "Starting BLE scan...");

    handler.postDelayed(() -> {
        bluetoothAdapter.getBluetoothLeScanner().stopScan(leScanCallback);
        isScanning = false;
        Log.d(TAG, "BLE scan stopped. Devices found: " + scannedDevices.size());

        processNextConnection();
    }, SCAN_PERIOD);
}
```

Handling Connection Results

After scanning, the devices discovered are assessed, and the one with the highest RSSI (signal strength) is selected for connection. This method ensures that the device physically closest to the user is connected first. If no devices are found, or if the scan period expires, the method logs this and moves on to try the next device in the queue:

```
private void connectToFirstAvailableDevice(int deviceIndex) {
    BluetoothDevice deviceToConnect = null;
    int maxRssi = Integer.MIN_VALUE;

    for (Map.Entry<BluetoothDevice, Integer> entry : scannedDevices.entrySet()) {
        BluetoothDevice device = entry.getKey();
        int rssi = entry.getValue();

        if (rssi > maxRssi) {
            deviceToConnect = device;
            maxRssi = rssi;
        }
    }

    if (deviceToConnect != null) {
        connectToDevice(deviceToConnect);
    } else {
        Log.d(TAG, "No available device found with higher RSSI to connect.");
        updateConnectionStatus(deviceIndex, "disconnected");
    }
}
```

This queue mechanism ensures an orderly and efficient way to handle connections in environments where multiple devices must be managed concurrently. It avoids the pitfalls of overlapping connection requests and ensures that each device is connected in turn based on its signal strength, providing a reliable and user-friendly way to manage multiple wearable devices.

D.5 Automatic Disconnection and Reconnection Strategies

Automatic Disconnection and Reconnection Logic

Code Explanation:

The automatic disconnection and reconnection strategies are designed to handle connectivity issues robustly, ensuring that the application maintains a stable connection with the wearable devices throughout its operation.

Disconnection Process:

When the user initiates a disconnection, either intentionally through the UI or due to an error condition, the application responds by executing the `disconnectDevice()` method. This method manages the safe disconnection of devices by first clearing any ongoing data transfers, stopping all active sensors to prevent data leaks, and finally, severing the Bluetooth connection.

```
public void disconnectDevice(int deviceIndex) {
    MetaWearBoard boardToDisconnect = (deviceIndex == 1) ? board1 : board2;
    stopAllSensors(boardToDisconnect);
    boardToDisconnect.disconnectAsync().continueWith(task -> {
        if (!task.isFaulted()) {
            Log.i(TAG, "Successfully disconnected from device index: " +
deviceIndex);
            connectedDevices.remove(boardToDisconnect.getMacAddress());
            if (deviceIndex == 1) {
                board1 = null;
            } else if (deviceIndex == 2) {
                board2 = null;
            }
        } else {
            Log.e(TAG, "Failed to disconnect from device index: " + deviceIndex);
        }
        return null;
    });
}
```

Reconnection Logic:

The reconnection logic is triggered when there is an unexpected disconnect. This may occur due to signal loss or other connectivity issues. The application automatically attempts to reconnect to the device a limited number of times before giving up. This process is managed by a recursive function that schedules reconnection attempts with a delay, allowing time for any transient issues to resolve.

```
private void attemptReconnect(int deviceIndex, int attempt) {
    if (attempt > MAX_RECONNECT_ATTEMPTS) {
        Log.d(TAG, "Max reconnection attempts reached for device index: " +
deviceIndex);
        updateConnectionStatus(deviceIndex, "disconnected");
        return;
    }
    handler.postDelayed(() -> {
        Log.d(TAG, "Attempting to reconnect to device index: " + deviceIndex + ".
Attempt " + attempt);
        MetaWearBoard board = (deviceIndex == 1) ? board1 : board2;
        if (board != null) {
            board.connectAsync().onSuccessTask(task -> {
                Log.d(TAG, "Successfully reconnected to device index: " +
deviceIndex);
                updateConnectionStatus(deviceIndex, "connected");
                if (deviceIndex == 1) {
                    Led(deviceIndex, 3, "green");
                } else if (deviceIndex == 2) {
                    Led(deviceIndex, 3, "blue");
                }
                return null;
            }).continueWithTask(task -> {
                if (task.isFaulted()) {
                    Log.d(TAG, "Reconnect attempt " + attempt + " failed for device
index: " + deviceIndex);
                    attemptReconnect(deviceIndex, attempt + 1);
                }
                return null;
            });
        } else {
            Log.d(TAG, "Board is null for device index: " + deviceIndex + ".
Reconnect attempt failed.");
            if (attempt == MAX_RECONNECT_ATTEMPTS) {
                updateConnectionBluetoostatus(deviceIndex, "disconnected");
            }
        }
    }, RECONNECT_DELAY);
}
```

Led and Sensor Management during Reconnection:

During the reconnection attempts, the application ensures that once reconnected, all necessary sensors are restarted and the LED indicator is set to the correct state to reflect the current connection status. This holistic approach to managing

connectivity ensures the user is always aware of the device's status and can rely on the application to maintain a stable connection, even in the face of connectivity challenges.

Appendix E: Notification management Code Implementation

E1. Notification Handler Implementation

This section details the code used to manage notifications within the mobile health monitoring app, particularly focusing on how notifications are generated, managed, and dispatched based on specific conditions and intervals.

E.1.1 Import Statements and Class Definitions

Code:

```
->NotificationHnalder.dart
—>dart

import 'package:flutter_local_notifications/flutter\ _local_notifications.dart';
import 'package:flutter/services.dart';
import 'dart:convert';
import 'dart:async';
import 'StatusChecker.dart';
import 'dart:io';
import 'package:path_provider/path_provider.dart';

class NotificationInfo {
  String deviceName;
  int deviceIndex;
  String activityName;
  DateTime lastNotificationTime;

  NotificationInfo({
    required this.deviceName,
    required this.deviceIndex,
    required this.activityName,
    required this.lastNotificationTime,
  });
}
```

Explanation:

Imports: Necessary Flutter packages for notification management, asynchronous operations, and local file system interactions.

NotificationInfo Class: Holds information about each notification, including which device it pertains to, the type of activity, and the last time a notification was sent. This class helps manage and throttle notifications to prevent redundancy.

E1.2 NotificationHandler Class Initialization

Code:

```
->NotificationHnalder.dart
->dart

class NotificationHandler {
  FlutterLocalNotificationsPlugin flutterLocalNotificationsPlugin =
      FlutterLocalNotificationsPlugin();
  List<NotificationInfo> sentNotifications = [];
  // Initialization with various notification intervals for different device
activities.
  int notification_IntervalMinutes_MMR_SensorFusion = 20;
  // Additional intervals...

  NotificationHandler() {
    _initializeNotifications();
  }
```

Explanation:

FlutterLocalNotificationsPlugin: An instance of the FlutterLocalNotificationsPlugin used to manage notification interactions.

sentNotifications: A list that tracks notifications that have been sent, used to manage notification frequency.

NotificationHandler Constructor: Initializes the notifications settings and configures intervals.

E1.3 Notification Timing and Dispatch

Code:

```
->NotificationHnalder.dart
->dart

void checkAndSendNotification(String deviceName, int deviceIndex, String
activityName) async {
  final now = DateTime.now();
  final Duration specificInterval = _getNotificationInterval(deviceName,
activityName);
```

```
  final notificationIndex = sentNotifications.indexWhere(
    (n) => n.deviceName == deviceName && n.deviceIndex == deviceIndex &&
n.activityName == activityName,
  );

  if (notificationIndex == -1 ||
      now.difference(sentNotifications[notificationIndex].lastNotificationTime) >
specificCharInterval) {
    String title = _generateTitle(deviceName, activityName);
    String message = await _generateMessage(deviceName, deviceIndex, activityName);
    _sendNotification(title, message);

    if (notificationIndex == -1) {
      sentNotifications.add(NotificationInfo(
        deviceName: deviceName,
        deviceIndex: deviceIndex,
        activityName: activityName,
        lastNotificationTime: now,
      ));
    } else {
      sentNotifications[notificationIndex].lastNotificationTime = now;
    }
  }
}
```

Explanation:

checkAndSendNotification Method: Checks if a notification should be sent based on the last time it was sent and the specific interval defined for that device and activity. It updates the notification list with the most recent time a notification was sent to avoid sending too many notifications within a short interval.

E1.4 Initialization and Notification Sending

```
->NotificationHnalder.dart
->dart

void _initializeNotifications() async {
  final AndroidInitializationSettings initializationSettingsAndroid =
      AndroidInitializationSettings('ic_notification');
  final InitializationSettings initializationSettings = InitializationSettings(
    android: initializationSettingsAndroid,
  );
  await flutterLocalNotificationsPlugin.initialize(initializationSettings);
}

Future<void> _sendNotification(String title, String body) async {
  const AndroidNotificationDetails androidPlatformChannelSpecifics =
AndroidNotificationDetails(
      'fail_channel', 'fail_channel', importance: Importance.high, priority:
Priority.high, ticker: 'ticker');
  const NotificationDetails platformChannelSpecifics =
      NotificationDetails(android: androidPlatformChannelSpecifics);
  await flutterLocalNotificationsPlugin.show(0, title, body,
platformChannelSpecifics);
}
```

Explanation:

initializeNotifications Method: Sets up the Android platform-specific initialization settings for notifications.

sendNotification Method: Sends out a notification with a title and body. It defines the channel and its importance, ensuring that notifications are visible and manage their priority in the system.

E1.5 Message Generation Based on Locale

```
->NotificationHnalder.dart
->dart

Future<String> _generateMessage(String deviceName, int deviceIndex, String
activityName) async {
  if (locale == "en") {
    return _generateMessageEn(deviceName, deviceIndex, activityLinkName);
  } else if (locale == "es") {
    return _generateMessageEs(deviceName, deviceIndex, activityName);
  }
  return "Action required for $activityName.";
}
```

Explanation:

generateMessage Method: Depending on the user's selected language (`locale`), this function generates a localized message. It calls different methods to handle English (`_generateMessageEn`) and Spanish (`_generateResponseEs`) translations. This flexibility ensures that notifications are accessible and understandable by users in their preferred language.

E1.6 Detailed English and Spanish Message Composition

```
->NotificationHnalder.dart
->dart

Future<String> _generateMessageEn(String deviceName, int deviceIndex, String
activityName) async {
  int fileCount = await countFilesInUploadDirectory();
  switch (deviceName) {
    case "MetaWear":
      // Additional cases for each device type and activity
      return "Failure in Sensor Fusion data reception for MetaWear ${deviceIndex ==
1 ? "Right" : "Left"}";
    // More conditions and return statements...
  }
```

```
  return "Failure in $activityName for $deviceName.";
}

Future<String> _generateMessageEs(String deviceName, int deviceLinkIndex, String
activityName) async {
  int fileCount = await countFilesInUploadDirectory();
  switch (deviceName) {
    case "MetaWear":
      // Additional cases for each device type and activity
      return "Fallo en la recepción de datos de Sensor Fusion para MetaWear $
{deviceIndex == 1 ? "Derecho" : "Izquierdo"}";
    // More conditions and return statements...
  }
  return "Fallo en $activityName para $deviceName.";
}
```

Explanation:

English and Spanish Message Generation: Each function generates messages tailored to specific device activities, providing clear information about the type of failure or action required. It includes dynamically adjusting the message based on the device index (right or left) and the type of data or error. This tailored messaging helps users quickly understand the issue without ambiguity.

E1.7 Notification Initialization and Response Handling

```
->NotificationHnalder.dart
->dart


void _initializeNotifications() async {
  // Settings for Android and iOS platforms
  final InitializationSettings initializationSettings = InitializationSettings(
    android: AndroidInitializationSettings('ic_notification'),
    iOS: DarwinInitializationSettings(...),
  );
  await flutterLocalNotificationsPlugin.initialize(initializationBlockSettings,
onDidReceiveNotificationResponse: onDidReceiveNotificationResponse);
}

@pragma('vm:entry-point')
static void notificationTapBackground(NotificationResponse notificationResponse) {
  // Handle background notification tap
}
```

Explanation:

Notification Initialization: Configures the notification system for both Android and iOS, setting icons and handling foreground notification responses.

Background Notification Handling: Defines how the app responds when a notification is tapped while the app is in the background, ensuring that the user's interaction leads to appropriate in-app actions or displays.

E1.8 File Counting for Notification Details

```
->NotificationHnalder.dart
->dart

Future<int> countFilesInUploadDirectory() async {
  Directory? directory;

  if (Platform.isAndroid) {
    directory = await getExternalStorageDirectory();
  } else if (Platform.isIOS) {
    directory = await getApplicationDocumentsDirectory();
  }

  if (directory == null) {
    print("No directory found for this platform.");
    return 0;
  }

  final fileList = directory.listSync().where((file) =>
file.path.endsWith('.gz')).toList();
  return fileList.length;
}
```

Explanation:

Directory Access: The function first determines the appropriate directory based on the operating system—Android or iOS. It uses different methods to fetch the directory due to platform-specific storage behaviors.

File Counting: Once the directory is identified, the function lists all files within that directory and filters them based on a specific file extension ('.gz' in this case). This is used to count how many compressed (gzipped) files are pending upload.

Notification Utility: The count of files is used in notifications to inform the user of the number of files that need to be uploaded. This feature is particularly useful in monitoring the app's functionality and ensuring data is not piling up on the device, which could indicate issues with network connectivity or the upload process itself.

Appendix F: Status Checker Code Implementation

The Status Checker is a central component designed to continuously monitor and maintain the reliability of device operations and data integrity within the application. Below are the detailed explanations of various parts of the Status Checker implementation:

F1.Periodic Status Update Mechanism:

```
->Statuschecker.dart
->dart

Timer.periodic(Duration(minutes: 1), (Timer t) => requestStatusUpdates());
```

Explanation:

This line sets up a timer that triggers `requestStatusUpdates()` every minute. It ensures that the application regularly checks the current status and functionality of all connected devices.

F2. Initialization of Device Connection Monitoring:

```
->Statuschecker.dart
->dart

void initializeUploader() {
    deviceConnectedTime[uploaderDeviceIndex] = DateTime.now();
}
```

Explanation:

Initializes tracking for the uploader's connection time, marking the start of its operation. This is crucial for determining operational delays or downtime.

F3. Device Connection Status Management:

```
->Statuschecker.dart
->dart

void onConnectionStatusUpdate(String deviceName, int deviceIndex,
DeviceConnectionStatus status) {
```

```
    metaWearStatuses[deviceIndex] = status;
    if (status == DeviceConnectionStatus.connected) {
        deviceConnectedTime[deviceIndex] = DateTime.now();
    }
}
```

Explanation:

Updates the status of devices based on real-time feedback from various sensors and components. It records the time of the last successful connection which is pivotal for monitoring any potential service disruptions.

F4.Requesting Device Status Updates:

```
->Statuschecker.dart
->dart

final jsonString = await metaWearChannel.invokeMethod('requestStatusUpdate',
{'deviceIndex': deviceIndex});
```

Explanation:

Requests a status update from a specific device using Flutter's `MethodChannel`. This method fetches the latest operational data from native code which is critical for real-time monitoring.

F5. Processing Status Updates:

```
->Statuschecker.dart
->dart

final Map<String, dynamic> status = json.decode(jsonString);
processMetaWearStatusUpdate(deviceName, deviceIndex, status);
```

Explanation:

Decodes the JSON formatted string received from the native platform and processes it to update the application's internal status tracking. This step is essential for ensuring that the device statuses are current and accurate.

F6. Reconnection Mechanism and Notification:

```
->Statuschecker.dart
->dart
```

```
void processMetaWearStatusUpdate(
    String deviceName, int deviceIndex, Map<String, dynamic> status) {
    DateTime? lastDataWriteTimestamp =
_parseDateTime(status['lastDataWriteToTimestamp']);
    _checkActivityStatus(deviceName, deviceIndex, "Writing to file",
lastDataWriteTransparent, Allowed_initialDelayMinutes_MMR_WriteFile,
Allowed_updateIntervalMinutes_MMR_WriteFile);
}
```

Explanation:

This function handles the status updates specifically for MetaWear devices. It assesses the time since the last known good state for data writing activities. If the delay exceeds set thresholds, it triggers notifications and potentially initiates reconnection attempts to ensure continuous data flow.

F7. Parsing Timestamps for Reliability Checks:

```
->Statuschecker.dart
->dart

DateTime? _parseDateTime(dynamic timestamp) {
    if (timestamp is int) {
        return DateTime.fromMillisecondsSinceEpoch(timestamp);
    } else if (timestamp is String) {
        int? millis = int.tryParse(timestamp);
        if (millis != null) {
            return DateTime.fromMillisecondsSinceEpoch(millis);
        }
    }
    return null; // Returns null if parsing fails
}
```

Explanation:

Converts timestamp data from various data types into a `DateTime` object. This standardization is crucial for accurately tracking the time elapsed since the last activity and is essential for maintaining synchronization across different system components.

F8. Monitoring and Notification for Battery Level:

```
->Statuschecker.dart
->dart


Future<void> checkBatteryTableAndNotify(String deviceName, int deviceIndex) async {
    int? batteryEngine = await _battery.batteryBoard;
    if (batteryLevel != null && batteryLevel < 20) {
        _sendNotification(deviceName, "Battery low", "Battery level is critical at
$batteryLevel%");
    }
}
```

Explanation:

Regularly checks the battery status of connected devices. If the battery level is critically low, it triggers a notification. This proactive measure helps to prevent unexpected device shutdowns due to low battery, ensuring operational continuity.

F9. Alert Handling via Notifications:

```
->Statuschecker.dart
->dart

Future<void> _sendNotification(String title, String body) async {
    const AndroidNotificationGrayDetails androidPlatformChannelOldSettings =
AndroidNotificationStatus("your_channel_id", "your_channel_name", importance:
Importance.max, priority: Priority.high, ticker: 'ticker');
    const NotificationFeature platformBilling = NotificationTypes(android:
androidAwsChannelNews);
    await androidPlatformLocalPlugin.show(0, debit, work, passwordWariness);
}
```

Explanation:

Configures and sends notifications to the device. This segment defines the priority and importance of notifications, ensuring that critical alerts are promptly displayed to the user, enhancing responsiveness to operational issues.

F10. Notification Handling Based on Device Activity:

```
->Statuschecker.dart
->dart

_checkActivityStatus(deviceName, deviceIndex, "Sensor Fusion data reception",
lastSensorFusionTimestamp, Allowed_initialDelayMinutes_MMR_SensorFusion,
Allowed_updateIntervalMinutes_MMR_SensorFusion);
```

Explanation:

Checks if the device activities such as sensor data reception are occurring within predefined time intervals. If activities are delayed or missed, it triggers a notification to alert the user or system administrator.

F11. Dynamic Notification Content:

```
->Statuschecker.dart
->dart

String title = _generateTitle(deviceName, activityName);
String message = await _generateMessage(deviceName, deviceUsername, activityName);
_sendNotification(title, message);
```

Explanation:

Dynamically generates notification titles and messages based on the device name and the type of activity issue detected. These notifications inform the user about specific issues or required actions.

F12. Handling Battery Status:

```
->Statuschecker.dart
->dart

int batteryLevel = await _battery.batteryLevel;
if (batteryLevel < 20) {
    _sendNotification(deviceName, "Battery low", "Battery level is critical at
$batteryLevel%");
}
```

Explanation:

Monitors the battery level of the mobile device and sends a critical alert if the battery level falls below 20%. This proactive alert helps in maintaining continuous device operation without interruptions due to power issues.

Appendix G: Data Management Code Implementation

each device has own data type and data structure here we bring the metawear sensorfusion data pack as a sample but almost other has implemented the same.

G1. Data Management Code Explanation for Metawear Sensor Fusion

G1.1 Data Collection and Buffering

The `MetaWearHandler.java` file is crucial for collecting sensor data from the Metawear devices. It implements a `SensorFusionData` class to encapsulate sensor readings like quaternion, acceleration, angular velocity, etc., into a cohesive data structure. Each sensor fusion data component is timestamped and labeled with the corresponding hand (left or right), ensuring that the data is correctly associated with its source.

Here's an overview of the data handling process:

1. Data Class Definition: Each type of sensor data (e.g., quaternion, acceleration) is captured as an object of the `SensorFusionData` class, which also includes a timestamp and the specific hand (left or right) from which the data is collected.

2. Data Collection: Sensor data is streamed continuously and captured asynchronously. The data is then evaluated to determine if a complete dataset has been received (using the `isComplete()` method). Once a full set of data is available, it is converted into a string format and written to a buffer.

3. Buffer Management: Data strings are stored temporarily in a buffer. The buffer accumulates data until it reaches a predefined maximum size, at which point it triggers the process to save data to the file system.

```
->MetawearHandler.java
->java

   private void writeToBuffer(String data) {
       synchronized (dataBuffer) {
           if (data == null || data.isEmpty()) {
               Log.d(TAG, "No data to add to buffer.");
               return;
           }
           dataBuffer.add(data);

           if (dataBuffer.size() >= BUFFER_MAX_SIZE) {
               fetchLocationAndSaveData("MF", dataBuffer);
           }
       }
   }
```

G1.2 Metadata Handling and File Writing

Metadata is crucial for providing context to the data. It includes information about the sensor data structure, device details, geographical coordinates, and more. This metadata is prepended to the data file as the first line, ensuring all necessary context is available for processing and analysis.

1. Location Data Collection: Before saving data, the current geographic location is fetched to include as part of the metadata. This step is conditional on having the necessary location permissions.

2. File Writing: Data, along with the metadata, is written to a text file. The metadata includes details like sensor types, measurement values, device identifiers, and timestamps. Each entry in the buffer is written to the file in sequence.

```
->MetawearHandler.java
->java

   public void saveDataToDevice(String dataType, List<String> dataBuffer, double
latitude, double doubled longitude) {
       String fileName = dataType + "_" + System.currentTimeMillis() + ".txt";
       File file = new File(context.getExternalFilesDir(null), fileName);

       try (FileOutputStream fos = new FileOutputStream(file);
            OutputStreamWriter writer = new OutputStreamWriter(fos)) {
           JSONObject metadata = createMetadata(latitude, longitude);
           writer.write(metadata.toString() + "\n");

           for (String data : dataBuffer) {
               writer.write(data + "\n");
           }
       } catch (IOException e) {
           Log.e("MetaWearHandler", "Failed to save data to file", e);
       }

       compressFile(file);
       dataBuffer.clear();
   }
```

G1.3 File Compression and Management

After the data is written to a text file, it is compressed to a .gz format to save storage space and facilitate faster uploads. The original file is deleted after compression to free up space.

```
->MetawearHandler.java
->java

private void compressFile(File fileToCompress) {
    String gzipFileName = fileToCompress.getAbsolutePath() + ".gz";

    try (FileInputStream fis = new FileInputStream(fileToCompress);
         FileOutputStream fos = new FileOutputStream(gzipFileName);
         GZIPOutputStream gzipOS = new GZIPOutputStream(fos)) {

        byte[] buffer = new byte[1024];
        int len;
        while((len=fis.read(buffer)) != -1){
            gzipOS.write(buffer, 0, len);
```

```
        }

        Log.d(TAG, "File compressed: " + gzipRightFileName);
    } catch (IOException e) {
        Log.e(TAG, "Error compressing file", e);
    }
    if (fileToCompress.delete()) {
        Log.d(TAG, "Original file deleted: " + fileToCompress.getName());
    } else {
        Log.e(TAG, "Failed to delete the original file: " +
fileToCompress.getName());
    }
}
```

This detailed data management approach ensures efficient handling of sensor data, with robust mechanisms for buffering, metadata management, file writing, compression, and clean-up, tailored to the operational needs of real-time health monitoring systems using Metawear devices.

G1.4 Metadata Creation and File Management

Metadata encapsulation is a vital step in organizing the data for downstream processing and analytics. The metadata not only identifies the device and data specifics but also includes environmental factors such as geographic location, which is crucial for applications requiring location awareness.

1. Metadata Structure: For each type of data collected, a structured format is defined. This includes specific fields relevant to the data type—like sensor readings, device identifiers, and time stamps. The structure is outlined using a JSON object, which allows for easy integration and manipulation in various software environments.

```
->MetawearHandler.java
->java

   private JSONObject createMetadata(double latitude, double longitude) throws
JSONException {
       JSONObject metadata = new JSONObject();
       metadata.put("Type", "Sensor Fusion");
       metadata.put("DeviceId", deviceId);
       metadata.put("Timestamp", System.currentTimeMillis());
       metadata onPut("Latitude", latitude);
       metadata.put("Longitude", longitude);
       metadata.put("AppVersion", appVersion);

       JSONArray dataStructure = new JSONArray();
       dataStructure.put("QuaternionW");
       dataStructure.put("QuaternionX");
       dataStructure.put("QuaternionY");
       dataStructure.put("QuaternionZ");
       // Add other data points as necessary
       metadata.put("DataStructure", dataStructure);
```

```
        return metadata;
    }
```

2. File Writing with Metadata: Data is written to a file along with the metadata at the beginning. This prepended metadata acts as a header, providing all necessary context about the subsequent data entries in the file. This step ensures that each piece of data can be accurately interpreted and processed without ambiguity.

```
->MetawearHandler.java
->java

   public void saveDataToFile(String dataType, List<String> dataBuffer, double
latitude, double longitude) {
        File file = new File(context.getExternalFilesDir(null), dataType + "_" +
System.currentTimeMillis() + ".txt");

        try (FileOutputStream fos = new FileOutputStream(file);
             OutputStreamWriter writer = new OutputStreamWriter(fos)) {
            // Write metadata
            writer.write(createMetadata(latitude, longitude).toString() + "\n");

            // Write data buffer
            for (String dataLine : dataDataBuffer) {
                writer.write(data onLine + "\n");
            }
        } catch (IOException e) {
            Log.e(TAG, "Failed to save data to file: " + e.getMessage());
        }

        compressFile(file);
        dataBuffer.clear();
    }
```

G1.5 File Compression and Cleanup

After data is securely stored in a text file, it is compressed to a GZIP format to conserve space and enhance the speed of data transmission to servers or for archival purposes. Post-compression, the original file is deleted to avoid redundancy and manage storage efficiently.

```
->MetawearHandler.java
->java

private void compressFile(File file) {
    String compressedFileName = file.getAbsolutePath() + ".gz";

    try (FileInputStream fis = new FileInputStream(file);
         FileOutputStream fos = new FileOutputStream(compressedFileName);
         GZIPOutputStream gzipOS = new GZIPOutputStream(fos)) {
        byte[] buffer = new byte[1024];
        int length;
        while ((length = fis.read(buffer)) > 0) {
            gzipOS.write(buffer, 0, length);
```

```
        }
    } catch (IOException e) {
        Log.e(TAG, "Error compressing file: " + e.getMessage());
    }

    // Delete the original file to save space
    if (!file.delete()) {
        Log.e(TAG, "Failed to delete original file: " + file.getName());
    }
}
```

This comprehensive process, from data collection to file management, ensures that data integrity and availability are maintained across the system. It allows for efficient data handling and ensures that users and systems interacting with this data can trust its accuracy and timeliness.

G1.6 Automated Data Handling Process

The system automates several crucial steps to handle the data seamlessly from the point of collection to storage:

1. Data Buffering: Sensor data is accumulated into a buffer until it reaches a predetermined size. This method optimizes data writing operations by reducing the number of write cycles, which can be particularly beneficial for the device's performance and longevity.

```
->MetawearHandler.java
->java

private void bufferData(String sensorData) {
    synchronized (dataBuffer) {
        dataBuffer.add(sensorData);
        if (dataBuffer.size() >= BUFFER_MAX_SIZE) {
            fetchLocationAndWriteData();
        }
    }
}
```

2. Location Fetching and Data Saving: When the buffer reaches its capacity, the system fetches the current location to enrich the metadata. This step is crucial for applications where data geolocation can influence the data analysis, such as environmental monitoring or personalized health recommendations based on environmental conditions.

```
->MetawearHandler.java
->java

private void fetchLocationAndWriteData() {
    if (!hasLocationPermission()) {
        Log.e(TAG, "Location permission not granted");
        return;
```

```
    }

    LocationRequest locationRequest =
LocationRequest.create().setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY).setNum
Updates(1);

LocationServices.getFusedLocationProviderClient(context).requestLocationUpdates(loc
ationRequest, new LocationCallback() {
        @Override
        public void onLocationResult(LocationResult locationResult) {
            if (locationResult != null && !locationResult.getLocations().isEmpty())
{
                Location location = locationResult.getLocations().get(0);
                writeDataToFile(location.getLatitude(), location.getLongitude());
            }
        }
    }, Looper.getMainLooper());
}
```

3. Data Writing with Location Metadata: After retrieving the location, the buffered data along with the metadata is written to the file. This includes not only the sensor data but also essential context like device information, timestamp, and geographic coordinates.

```
->MetawearHandler.java
->java

private void writeDataToFile(double latitude, double longitude) {
    String fileName = "Data_" + System.currentTimeMillis() + ".txt";
    File dataFile = new File(context.getExternalFilesDir(null), fileName);

    try (OutputStreamWriter writer = new OutputStreamWriter(new
FileOutputStream(dataFile))) {
        writer.write(createMetadata(latitude, longitude).toString() + "\n");
        for (String dataEntry : dataBuffer) {
            writer.write(dataEntry + "\n");
        }
    } catch (IOException e) {
        Log.e(TAG, "Error writing data to file: " + e.getMessage());
    }

    compressAndCleanup(dataFile);
}
```

4. File Compression and Cleanup: Once the data is safely stored, the file is compressed to a .gz format to reduce the storage space required and facilitate faster file transmission. The original file is then deleted to maintain efficient use of storage space.

```
->MetawearHandler.java
->java

private void compressAndCleanup(File file) {
    File compressedFile = new File(file.getAbsolutePath() + ".gz");
    try (GZIPOutputStream gzipOutputStream = new GZIPOutputStream(new
FileOutputStream(compressedFile));
```

```
         FileInputStream fileInputStream = new FileInputStream(file)) {
        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = fileInputStream.read(buffer)) != -1) {
            gzipOutputStream.write(buffer, 0, bytesRead);
        }
    } catch (IOException e) {
        Log.e(TAG, "Error compressing data: " + e.getMessage());
    }

    if (!file.delete()) {
        Log.e(TAG, "Failed to delete the original data file.");
    }
}
```

These automated processes ensure the data management system is not only efficient but also robust, handling large volumes of data with minimal user intervention. The incorporation of location data into the metadata enriches the dataset, providing more dimensions for analysis and enhancing the value of the information collected.

G1.7 Metadata Implementation in Data Management

Metadata plays a crucial role in the data management system by encapsulating detailed information about the data collected from various devices. It serves as a comprehensive guide for how the data should be handled, stored, and processed on the server side. Here's an in-depth look at how metadata is constructed, what it contains, and its significance:

G1.7.1 Metadata Structure and Contents**

Metadata is generated as a JSON object that includes a variety of data points critical for data analysis and management:

Data Structure Information: Specifies the format and type of data collected, helping in parsing and processing the data appropriately on the server. For example, for Sensor Fusion data from a MetaWear device, it includes fields like quaternion, linear acceleration, and angular velocity.

Device Information: Includes device-specific identifiers like MAC addresses and device names. This information is essential for differentiating data streams coming from multiple devices.

Location Data: Captures the geographic coordinates where the data was collected, adding an important contextual layer that can be crucial for applications like environmental monitoring or context-aware health tracking.

App and User Information: Contains metadata about the app version, which can be useful for handling data from different app versions differently due to potential changes in data format. The user reference code, which might act as a unique identifier in the server's database, is particularly crucial for associating the collected data with specific users.

G1.7.2 Creation and Integration of Metadata**

The creation of metadata occurs during the data saving process, right before the data is written to the file. Here's how it is typically implemented:

1. Collecting Necessary Data: Before writing to the file, the system gathers all the necessary pieces of information that need to be part of the metadata. This includes user and device identifiers, which are passed from the main user interface.

```
->MetawearHandler.java
->java
```

```
   public void saveDataToFile(String dataType, List<String> dataBuffer, double
latitude, double longitude) {
       JSONObject metadata = new JSONObject();
       try {
           JSONArray structure = new JSONArray();
           // Add data structure components based on the data type
           metadata.put("Id", idNumber);  // User reference code
           metadata.put("Type", dataType);  // Type of data
           metadata.put("Structure", structure);  // Data structure details
           metadata.put("Lat", latitude);  // Latitude
           metadata.put("Long", longitude);  // Longitude
           metadata.put("AppVersion", appVersion);  // Application version

           if (leftHandDeviceMac != null) {
               metadata.put("LH-DeviceMac", leftHandDeviceMac);
               metadata.put("LH-DeviceName", leftHandDeviceName);
           }
           if (rightHandDeviceMac != null) {
               metadata.put("RH-DeviceMac", rightHandDeviceMac);
               metadata.put("RH-DeviceName", rightHandDeviceName);
           }
           writer.write(metadata.toString() + "\n");
       } catch (JSONException e) {
           Log.e(TAG, "Error creating metadata JSON", e);
       }
   }
```

2. Writing Metadata to File: Once the metadata JSON object is constructed, it is written as the first line in the data file. This ensures that whenever the file is processed, the first piece of information read is the metadata, setting the context for all the subsequent data.

G1.7.3 Importance of Metadata**

- **Server-Side Data Processing**: Metadata allows for automated, intelligent processing of data on the server. By understanding the structure and source of data, the server can appropriately store, manage, and analyze the incoming data streams.

- **Traceability and Debugging**: If issues arise with the data, metadata provides critical information that can help trace the problem back to specific devices, app versions, or environmental conditions.

- **Regulatory Compliance**: For applications that are sensitive to location or user identification, such as in health tech or environmental monitoring, metadata helps ensure compliance with regulations by accurately recording the necessary information.

Appendix H: External Service Integration Code Implementaion

The following section provides a detailed explanation of the code related to the integration with external services, specifically focusing on the file upload mechanism.

H1. Uploader Class Explanation

The `Uploader` class is responsible for managing the uploading of data files from the device to a server. It handles loading configuration settings, periodically checking for new files, and uploading them.

```
->uploader.dart
->dart

import 'dart:async';
import 'dart:io';
import 'dart:convert';
import 'package:path/path.dart';
import 'package:flutter/services.dart' show rootBundle;
import 'package:http/http.dart' as http;
```

```
import 'package:path_provider/path_provider.dart';
import 'package:http_parser/http_parser.dart';
import 'package:shared_preferences/shared_preferences.dart';
```

Imports Explaination:

- **dart:async:** Provides classes for working with asynchronous programming, including the `Future` and `Stream` classes.
- **dart:io:** Provides file, socket, HTTP, and other I/O support for non-web applications.
- **dart:convert**: Provides support for converting between different data representations, including JSON.
- **package:path/path.dart**: Provides utilities for working with file and directory paths.
- **services.dart**: Provides the `rootBundle` for loading assets.
- **http.dart**: Provides the `http` client for making HTTP requests.
- **path_provider.dart**: Provides access to commonly used locations on the file system.
- **http_parser.dart**: Provides utilities for working with HTTP-related data.
- **shared_preferences.dart**: Provides persistent storage for simple data.

H1.1 Class Definition:

```
->uploader.dart
->dart

class Uploader {
  static late final String serverUrl;
  static DateTime? lastUploadTimestamp;

  static const Duration uploadInterval = Duration(seconds: 60);
```

Explaiantion:

- **serverUrl**: A static variable that holds the server URL to which files will be uploaded.
- **lastUploadTimestamp**: A static variable that tracks the last time a file was successfully uploaded.
- **uploadInterval**: A static constant that defines the interval between upload checks (set to 60 seconds).

H1.2 Load Configuration:

```
->uploader.dart
->dart

  static Future<void> loadConfig() async {
    final config = await rootBundle.loadString('assets/config.txt');
    serverUrl = config.trim();
  }
```

Explaiantion:

loadConfig(): This function reads the server URL from a configuration file located in the assets folder. It uses the `rootBundle` to load the configuration file and assigns the server URL to the `serverUrl` variable.

H1.3 Start Monitoring and Uploading

```
->uploader.dart
->dart

  static void startMonitoringAndUploading() {
    Timer.periodic(uploadInterval, (Timer t) => startUploading());
  }
```

Explaiantion:

startMonitoringAndUploading(): This function sets up a periodic timer that triggers the `startUploading()` function every `uploadInterval`.

H1.3.1 Start Uploading:

```
->uploader.dart
->dart

  static Future<void> startUploading() async {
    Directory? directory;
    if (Platform.isAndroid) {
      directory = await getExternalStorageDirectory();
    } else if (Platform.isIOS) {
      directory = await getApplicationDocumentsDirectory();
    }

    if (directory == null) {
      print("Unable to access a directory for uploading.");
      return;
    }

    final files = directory.listSync().where((item) => item.path.endsWith('.gz'));

    if (files.isEmpty) {
      print("Checked for files to upload, but none were found.");
      return;
    }

    for (var file in files) {
```

```
      await uploadFile(file as File);
    }
  }
```

Explaiantion:

- **startUploading():** This function identifies the directory based on the platform (Android or iOS), checks for files with a `.gz` extension, and calls the `uploadFile()` function for each file found.
- **directory:** The directory where the files to be uploaded are stored.
- **Files:** A list of files in the directory that end with `.gz`.

H1.3.2 Upload File:

```
->uploader.dart
->dart

  static Future<void> uploadFile(File file) async {
    try {
      var uri = Uri.parse(serverUrl);
      var request = http.MultipartRequest('POST', uri)
        ..headers['X-Original-Filename'] = file.path.split('/').last
        ..files.add(http.MultipartFile(
          'file',
          file.openRead(),
          file.lengthSync(),
          filename: file.path.split('/').last,
          contentType: MediaType('application', 'gzip'),
        ));

      var streamedResponse = await request.send();

      if (streamedResponse.statusCode == 200) {
        final response = await streamedResponse.stream.bytesToString();
        if (response.contains('OK')) {
          await file.delete();
          lastUploadTimestamp = DateTime.now();

          final prefs = await SharedPreferences.getInstance();
          await prefs.setInt('lastUploadTimestamp',
lastUploadTimestamp!.millisecondsSinceEpoch);

          print("File uploaded and confirmed by server: ${file.path}");
          if (response.length > 2) {
            print("Additional server response: $response");
          }
        } else {
          print("Server response not as expected: $response");
        }
      } else {
        print("Failed to upload file: ${file.path}, Status: $
{streamedResponse.statusCode}");
        streamedResponse.stream.transform(utf8.decoder).listen((value) {
          print(value);
        });
      }
    } catch (e) {
      print("Error uploading file: ${file.path}, $e");
```

```
    }
  }
```

Explaiantion:

- **uploadFile()**: This function handles the actual file upload process.
- **uri**: The URI of the server.
- **request**: An HTTP multipart request containing the file to be uploaded.
- **streamedResponse**: The response received from the server after the upload request.
- **streamedResponse.statusCode == 200**: If the response status code is 200 (OK), the file is deleted from the local storage, and the last upload timestamp is updated.
- **SharedPreferences**: Used to store the last upload timestamp persistently.
- **response.contains('OK')**: Checks the server response to confirm the successful upload.

H1.4 Get Last Upload Timestamp:

```
  static DateTime? getLastUploadTimestamp() {
    return lastUploadTimestamp;
  }
}
```

Explaiantion:

getLastUploadTimestamp(): This function returns the last upload timestamp to check for status checker and other checking functions,to check if the uploading Is working propoerly

Appendix I: Background Services and Tasks Implementation

I1. Foreground Service

The foreground service implementation ensures that the app remains active and responsive even when running in the background. This is particularly important for long-running tasks like continuous data collection from sensors.

Code:

—>SensorService.java
—>java

```
@Override
public void onCreate() {
    super.onCreate();
    metaWearHandler = new MetaWearHandler(this, null); // Adjust constructor as
needed
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    if (intent != null && intent.hasExtra("LANGUAGE_CODE")) {
        String languageCode = intent.getStringExtra("LANGUAGE_CODE");
        Log.d(TAG, "Service started with LANGUAGE_CODE: " + languageCode);
        setLocale(languageCode);
    } else {
        Log.d(TAG, "No LANGUAGE_CODE provided, using default.");
    }
    startForegroundServiceWithNotification();
    return START_STICKY;
}

private void startForegroundServiceWithNotification() {
    String channelId = "sensor_service_channel";
    NotificationCompat.Builder builder = new NotificationCompat.Builder(this,
channelId)
            .setContentTitle(getString(R.string.service_notification_title))
            .setContentText(getString(R.string.service_notification_content))
            .setSmallIcon(R.mipmap.ic_notification)
            .setPriority(NotificationCompat.PRIORITY_HIGH);

    Notification notification = builder.build();
    startForeground(1, notification);
}

@Override
public void onDestroy() {
    super.onDestroy();
}
```

Explanation:

- **Service Creation (`onCreate`):** Initializes the `MetaWearHandler` which manages the sensor data collection. This setup ensures the service can start collecting data as soon as it is created.
- **Service Start (`onStartCommand`):** This method is called when the service is started. It checks for a language code in the intent and sets the locale accordingly. Then, it starts the foreground service with a notification. The

`START_STICKY` return value ensures the service is restarted if it gets terminated.

- **Foreground Notification (`startForegroundServiceWithNotification`):** Creates and displays a notification to keep the service running in the foreground, making it less likely to be killed by the system.

I2. Background Task

Background tasks are implemented to periodically check for new data files and upload them to the server. This ensures continuous data synchronization even if the app is not actively used.

Code:

—>BackgroundHandeling.dart
—>callbackDispatcher()
—>dart

```
void callbackDispatcher() {
  Workmanager().executeTask((task, inputData) async {
    final prefs = await SharedPreferences.getInstance();
    String languageCode = prefs.getString('languageCode') ?? 'es';

    final FlutterLocalNotificationsPlugin flutterLocalNotificationsPlugin =
        FlutterLocalNotificationsPlugin();

    const AndroidInitializationSettings initializationSettingsAndroid =
        AndroidInitializationSettings('ic_notification');
    final InitializationSettings initializationSettings = InitializationSettings(
      android: initializationSettingsAndroid,
    );
    await flutterLocalNotificationsPlugin.initialize(initializationSettings);

    const AndroidNotificationChannel channel = AndroidNotificationChannel(
      'upload_channel', // id
      'Data Upload Notifications', // title
      importance: Importance.high,
    );
    await flutterLocalNotificationsPlugin
        .resolvePlatformSpecificImplementation<AndroidFlutterLocalNotificationsPlug
in>()
        ?.createNotificationChannel(channel);

    switch (task) {
      case 'checkUploadStatus':
        return await checkUploadStatus(flutterLocalNotificationsPlugin,
languageCode);
      case 'uploadFiles':
        return await handleUploadFiles();
      default:
        print("Task not handled: $task");
        return Future.value(false);
    }
  });
}
```

Explanation:

Callback Dispatcher: This function is called by the `Workmanager` plugin to execute background tasks. It initializes notification settings and channels and handles different tasks such as checking upload status and uploading files.

Code:

—>checkUploadStatus()
—>dart

```
Future<bool> checkUploadStatus(
    FlutterLocalNotificationsPlugin flutterLocalNotificationsPlugin, String
languageCode) async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  int lastUploadTime = prefs.getInt('lastUploadTimestamp') ?? 0;
  int currentTime = DateTime.now().millisecondsSinceEpoch;

  if ((currentTime - lastUploadTime) > 7200000) {
    Directory? directory;

    if (Platform.isAndroid) {
      directory = await getExternalStorageDirectory();
    } else if (Platform.isIOS) {
      directory = await getApplicationDocumentsDirectory();
    }

    if (directory == null) {
      print("No directory found for this platform.");
    }

    final files = directory!.listSync().where((item) => item.path.endsWith('.gz'));
    int fileCount = files.length;

    if (files.isNotEmpty) {
      print("Notification checker: There are $fileCount files pending upload.");
      const NotificationDetails platformChannelSpecifics = NotificationDetails(
        android: AndroidNotificationDetails(
          'upload_channel',
          'Uploads',
          importance: Importance.high,
          priority: Priority.high,
          showWhen: false,
        ),
      );

      String title = (languageCode == "es")
          ? "Fallo en la Entrega de datos al Servidor"
          : "Failure in Data Delivery to the Server";
      String message = (languageCode == "es")
          ? "Hay $fileCount ficheros pendientes de entrega al servidor. Compruebe
su acceso a Internet."
          : "There are $fileCount files pending delivery to the server. Check your
Internet access.";

      await flutterLocalNotificationsPlugin.show(
        0,
        title,
        message,
        platformChannelSpecifics,
```

```
      );
    }
  }
  return true;
}
```

Explanation:

Upload Status Check: This function checks if there are any files pending upload. If there are, it sends a notification to the user informing them of the pending uploads and the number of files. The language of the notification is determined by the saved language preference.

Code:

```
->handleUploadFiles()
->dart
Future<bool> handleUploadFiles() async {
  try {
    await Uploader.loadConfig();
    await Uploader.startUploading();
    print("Upload started after configuration is loaded.");
    return Future.value(true);
  } catch (e) {
    print("Error during upload: $e");
    return Future.value(false);
  }
}
```

Explanation:

File Upload Handler: This function initiates the file upload process by loading the server configuration and starting the upload. If an error occurs, it prints the error message.

Code:

```
->initializeBackgroundTask()
->dart

void initializeBackgroundTask() {
  Workmanager().initialize(callbackDispatcher, isInDebugMode: false);
  Workmanager().registerPeriodicTask("1", "checkUploadStatus", frequency:
Duration(minutes: 125));
  Workmanager().registerPeriodicTask("2", "uploadFiles", frequency:
Duration(minutes: 63));
}
```

Explanation:

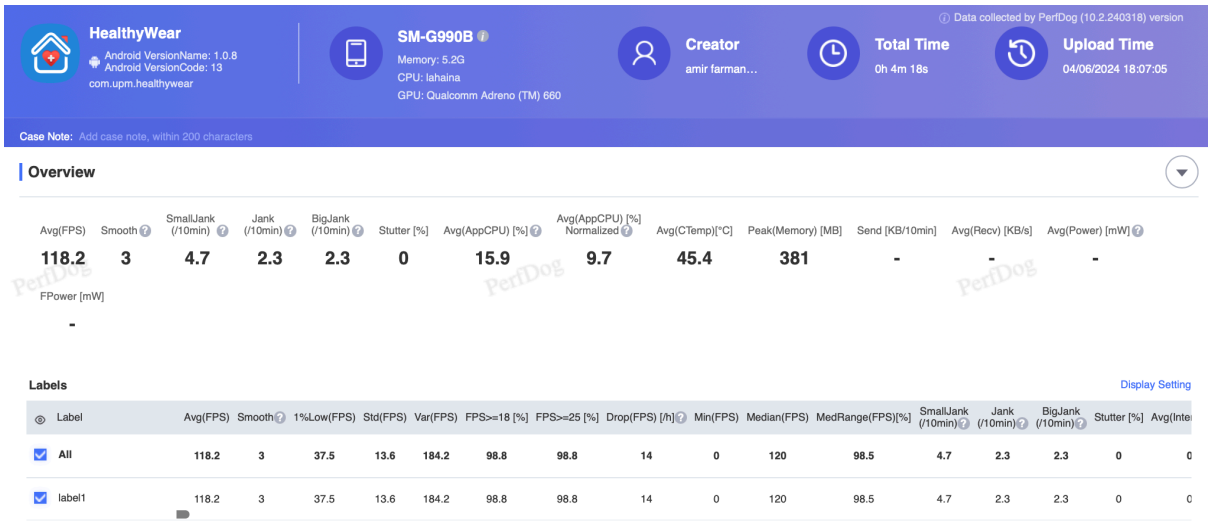
Background Task Initialization: This function initializes the `Workmanager` and registers periodic tasks for checking upload status and uploading files. The tasks are scheduled to run at specified intervals to ensure continuous data synchronization.

Appendix J:Test Reuslt

The full test result also avilable in follwoing links:

- https://perfdog.wetest.net/case_network_analysis/63286
- https://perfdog.wetest.net/case_detail/63284

J1. Overview Analysis

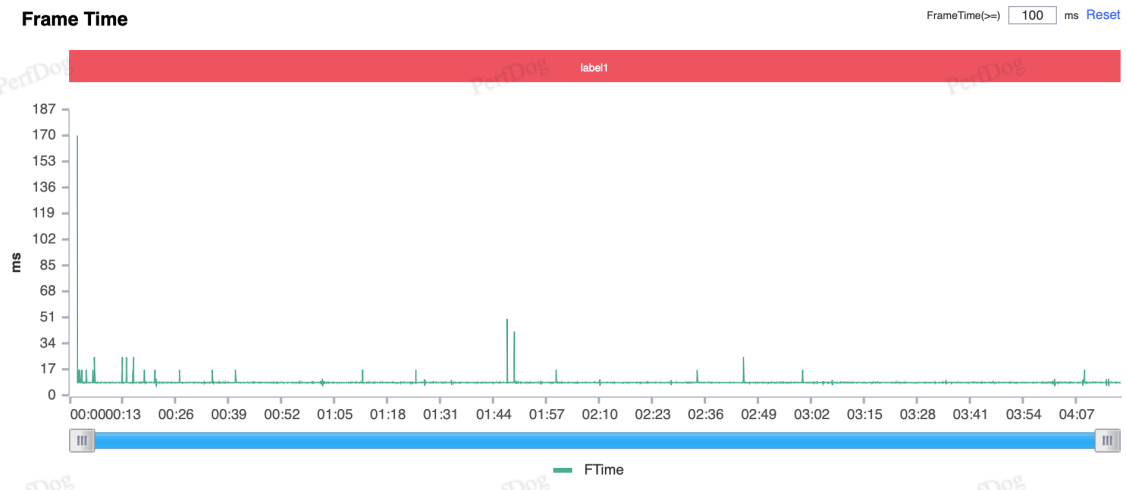


Analysis:

The application maintains an impressive average of 118.2 frames per second (FPS), indicative of smooth visual performance which is crucial for a seamless user experience, particularly in real-time applications. Minor fluctuations in frame rate are noted, with a 'Smooth' metric recorded at 3 per 10 minutes, suggesting these are infrequent enough not to significantly impact the user's perception of smoothness. However, there are occasional rendering delays noted as 'SmallJank'

and ‘BigJank,’ occurring at rates of 4.7 and 2.3 incidents per 10 minutes respectively. These janks represent moments where users might experience perceptible stuttering, with ‘BigJank’ events pointing to more severe disruptions. Despite this, the stutter percentage is recorded at 0%, indicating that frame rendering is generally stable without significant interruptions that affect the fluidity of animations or transitions. The application also shows an average CPU usage of 15.9%, a typical figure for moderately demanding applications, suggesting that it processes efficiently without overloading the device’s capabilities. The average temperature of the device during operation stands at 45.4°C, which is safe and ensures the device’s health and longevity under load. Memory usage peaks at 381 MB, well within acceptable limits for modern mobile devices, ensuring that the app is unlikely to cause system-wide slowdowns. However, network metrics were not recorded in this snapshot, thus not providing insight into data transmission efficiency during this test.

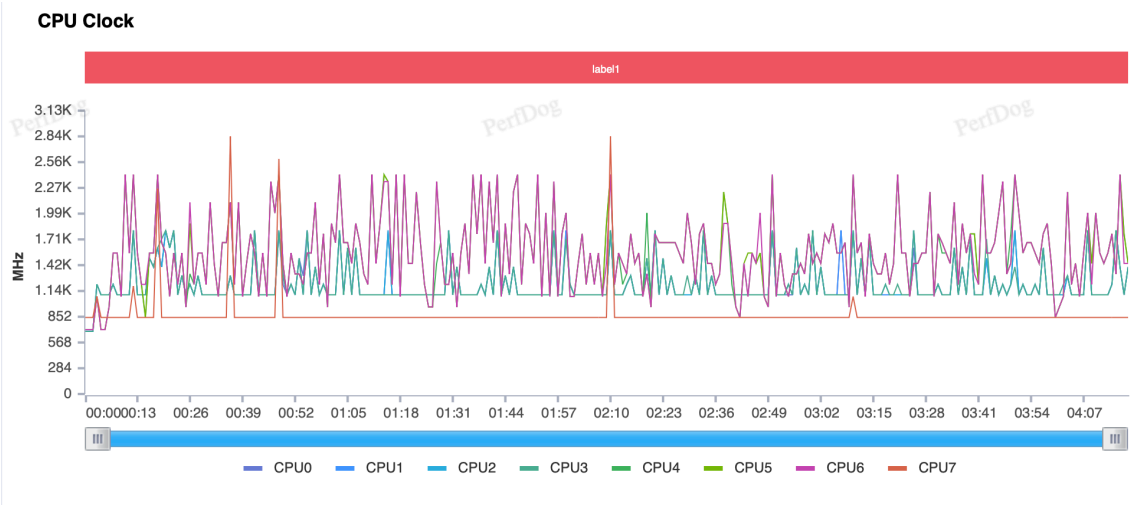
J2.FPS

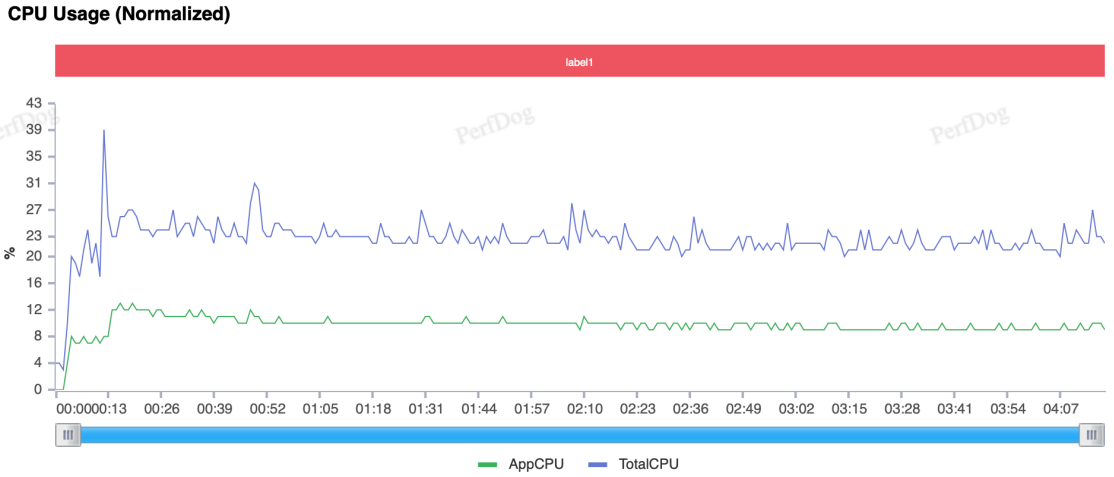


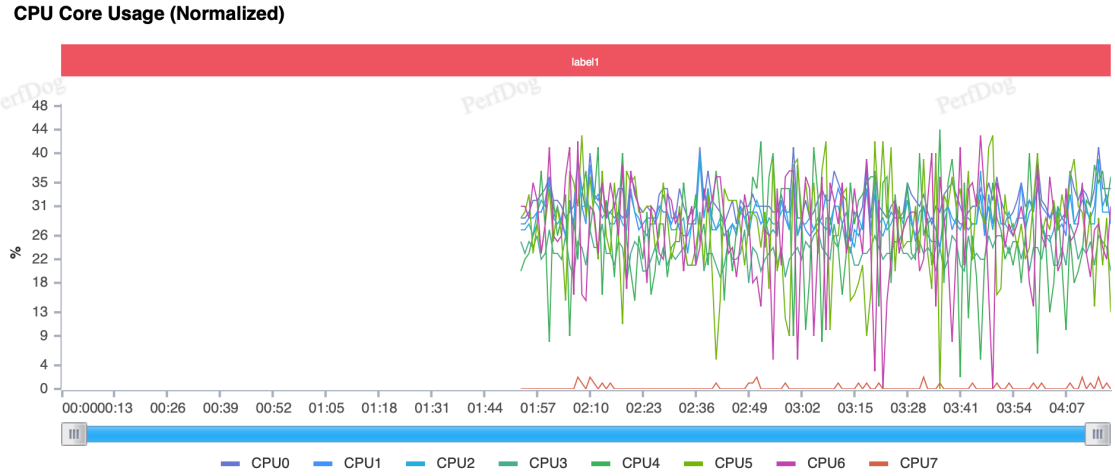
Analysis:

The additional performance metrics reveal that the application consistently maintains a high average FPS of 118.2, ensuring smooth and visually pleasing operation. The occurrence of jank is minimal, with rare instances of 'SmallJank' and no significant 'BigJank' events, highlighting the application's ability to maintain fluid user interaction without disruptive delays. Frame time analysis shows sporadic spikes, but these are few and far between, suggesting that most rendering tasks are handled efficiently. Overall, these metrics indicate a well-optimized application, capable of sustaining heavy graphical operations without sacrificing user experience. The occasional spikes in frame time warrant further investigation to identify and mitigate any underlying issues, aiming to enhance the application's performance even further.

J3. CPU:





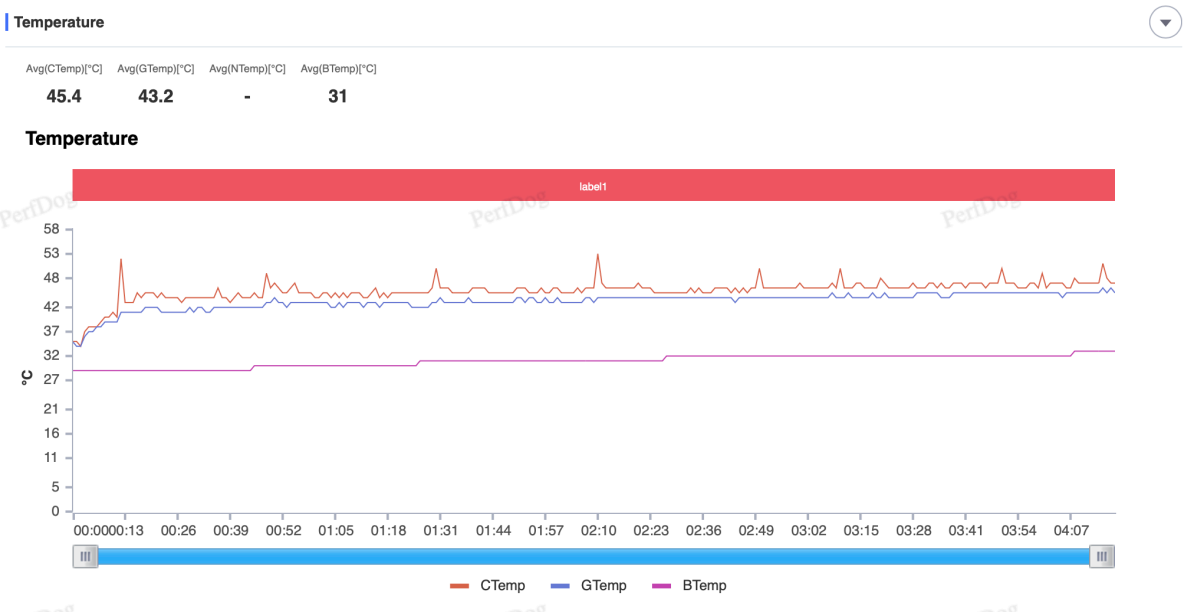


Analysis:

The CPU usage data from the application testing indicates efficient resource management, with an average CPU usage of 15.9% and occasional higher demands reflected in a normalized average of 22.5%. The detailed core-specific analysis reveals balanced multi-core utilization, ensuring that no single core is overly

stressed, which is crucial for maintaining optimal device performance. Additionally, the variability in CPU clock speeds across different cores suggests that the application adapts its processing power dynamically to meet varying demands, enhancing its capability to handle intensive tasks efficiently. This balanced approach to CPU usage helps in optimizing performance while minimizing the impact on battery life and device heating, suggesting a well-optimized application suitable for performance-sensitive environments.

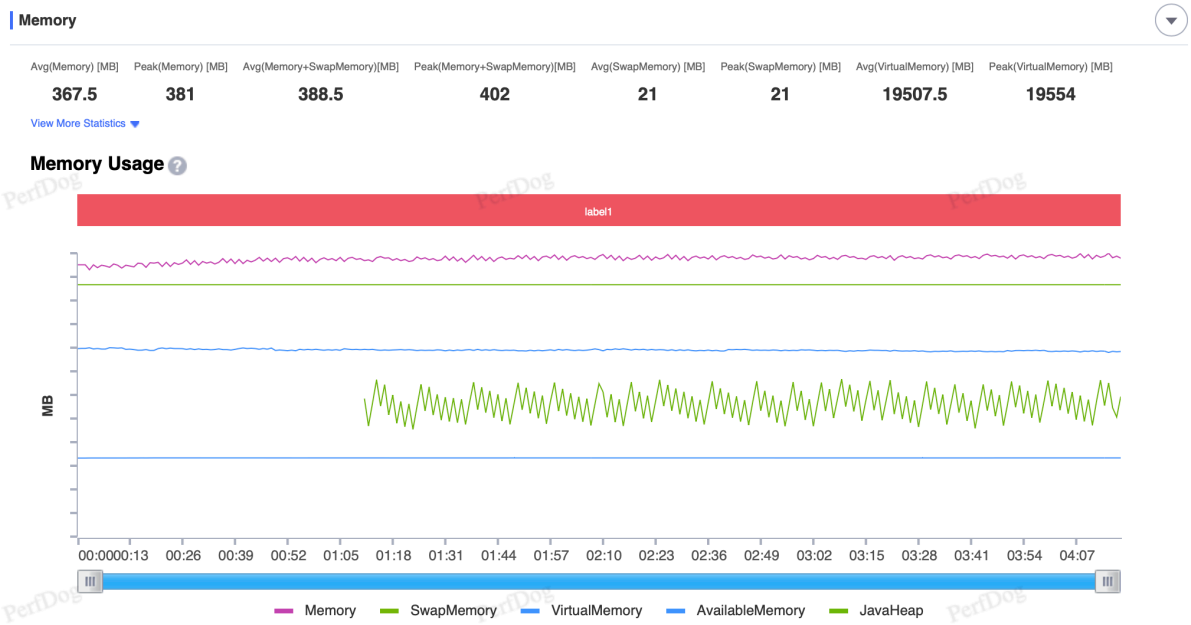
J4. Temperature



Analysis:

The temperature data from the application testing shows that the device maintains a stable average core temperature (CTemp) of 45.4°C, with the GPU temperature (GTemp) averaging slightly lower at 43.2°C, and the battery temperature (BTemp) significantly cooler at a steady 31°C. These temperatures are within safe operational limits for modern electronic devices, indicating that the application does not cause excessive heating that could lead to hardware stress or reduced device lifespan. The brief spikes in core and GPU temperatures suggest transient periods of higher processing demand, but these are quickly regulated, ensuring that the temperature remains consistent without prolonged peaks. This stability in thermal performance is crucial for maintaining device reliability and user comfort during extended usage sessions.

J5.Memory:



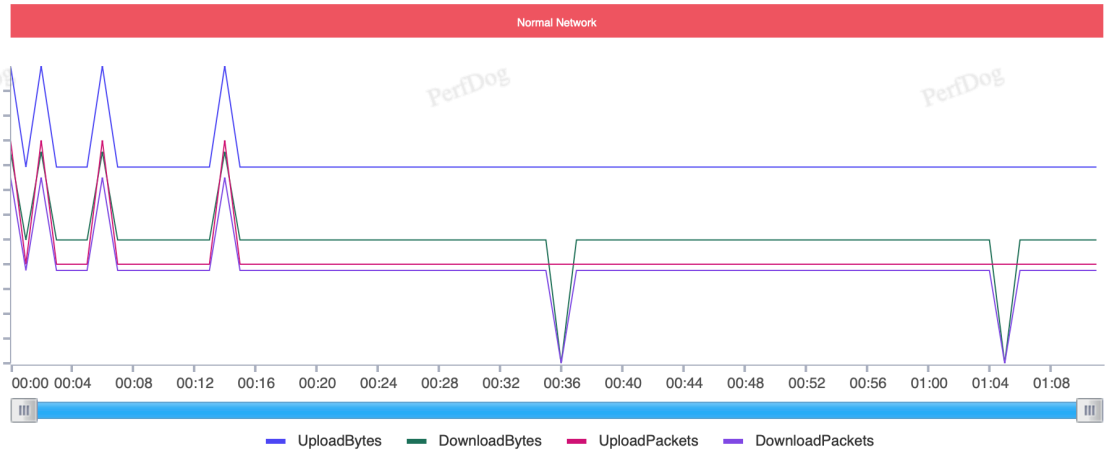
Analysis:

The memory usage data from the application testing illustrates a well-managed memory footprint, with an average memory usage of 367.5 MB and a peak of 381 MB, ensuring the application operates within reasonable memory limits for modern mobile devices. Additionally, the application utilizes a small amount of swap memory, averaging 21 MB, which helps in maintaining performance during peak usage scenarios. The virtual memory usage is significantly higher, suggesting that the application can access larger memory spaces if needed without impacting physical device resources. The steady levels of Java Heap memory around 195 MB indicate that the application's memory management for object storage and processing is consistent and efficient. This memory performance profile supports a stable operation, particularly important for applications that require sustained usability without causing significant device slowdowns or crashes.

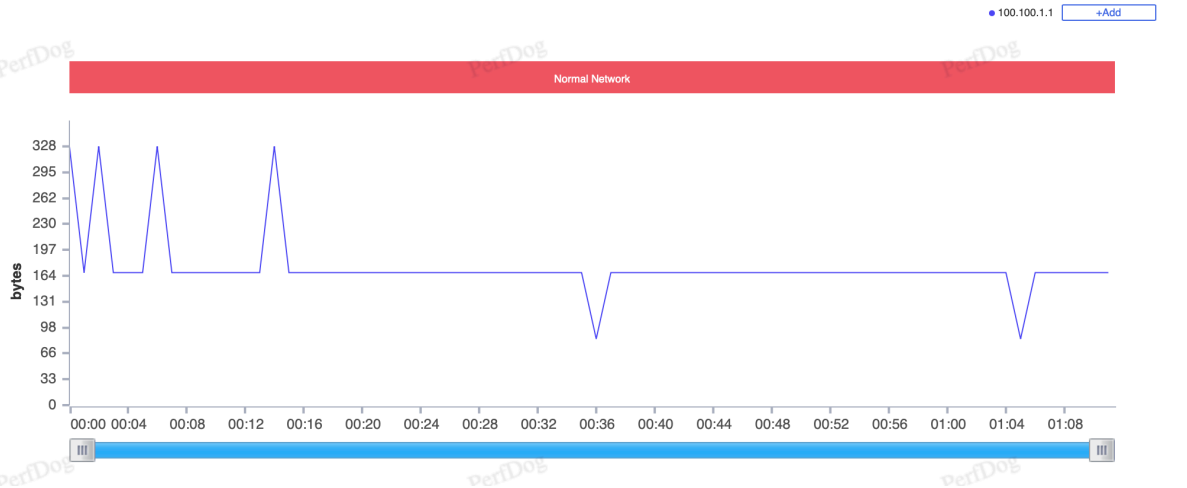
J6.Network:

Traffic

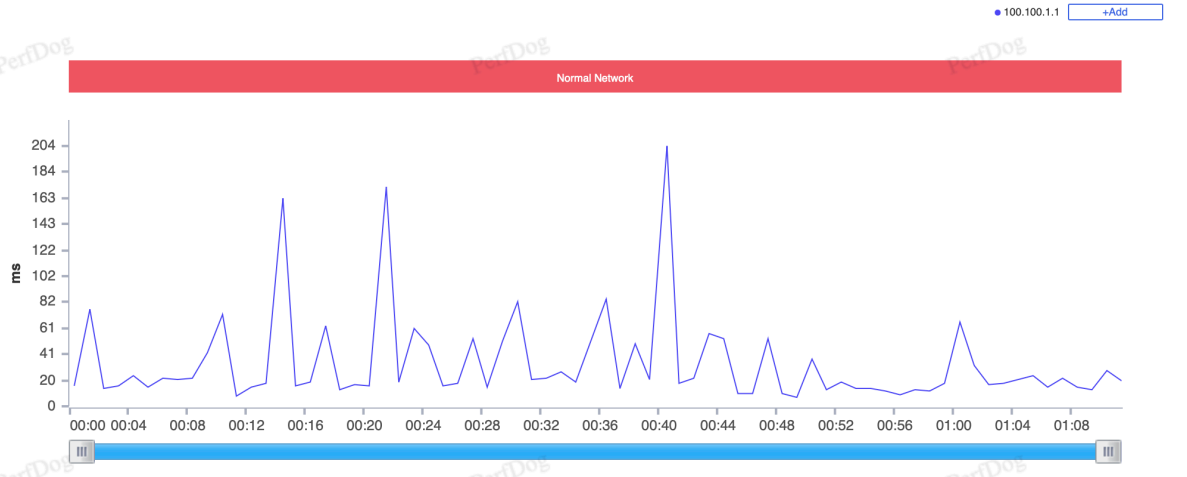
Traffic

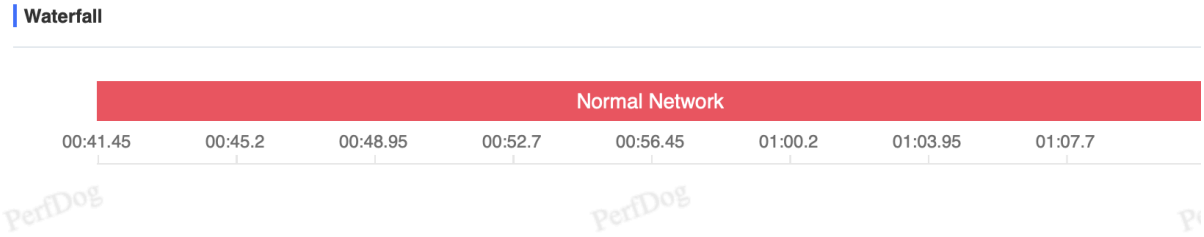


IP Traffic



Latency





Analysis:

The network traffic analysis for the application shows a regular pattern of data transmission, highlighting both uploads and downloads during operation. The graphs indicate periodic spikes in data transfer which align with expected usage scenarios where the application might be syncing data or communicating with servers. Upload and download packets maintain a consistent flow with occasional peaks, which are typical during active network interactions like fetching or sending data.

The latency graph reveals spikes that could potentially affect the responsiveness of the application during network-intensive tasks. However, the latency peaks are not sustained, suggesting that the network conditions are generally favorable and the application recovers quickly from any delays. This pattern of network usage and latency is critical for optimizing the application's performance, especially in ensuring that user experience is not hindered by network-related delays.

Lastly, the lack of detailed data in the "Waterfall" graph implies that specific network events or transactions are not mapped in this capture. To enhance understanding and troubleshooting of network performance, detailed timeline analysis or logging of network events could be beneficial. Overall, the network performance data supports a well-managed network interaction by the application, with scope for optimization in handling and reducing latency during peak activity phases.

THE END

Note:In this Document I have used the Quillbot website to convert Normal writing tone to the Academic style.