Design and Development of a Smartphone App for Health Monitoring and Alerts for Farm Workers

Omer Oztoprak
z1942688@students.niu.edu

Follow this and additional works at: https://huskiecommons.lib.niu.edu/allgraduate-thesesdissertations

Part of the Robotics Commons

Recommended Citation
Oztoprak, Omer, "Design and Development of a Smartphone App for Health Monitoring and Alerts for Farm Workers" (2023). Graduate Research Theses & Dissertations. 7840.
https://huskiecommons.lib.niu.edu/allgraduate-thesesdissertations/7840

This Dissertation/Thesis is brought to you for free and open access by the Graduate Research & Artistry at Huskie Commons. It has been accepted for inclusion in Graduate Research Theses & Dissertations by an authorized administrator of Huskie Commons. For more information, please contact jschumacher@niu.edu.
ABSTRACT

DESIGN AND DEVELOPMENT OF A SMARTPHONE APP FOR HEALTH MONITORING AND ALERTS FOR FARM WORKERS

Omer Oztoprak, M.S.
Department of Mechanical Engineering
Northern Illinois University, 2023
Dr. Ji-Chul Ryu, Director

This thesis presents the design and development of a novel smartphone application for simultaneous health monitoring and alerts for agricultural workers using multiple wearable sensor devices via Bluetooth Low Energy (BLE). The application combines health data from a range of sensors such as an IMU that detects body movement, a pulse oximeter that captures oxygen saturation and heart rate, and a thermometer that measures body temperature. The software framework is built using Jetpack Compose in Android Studio, and it incorporates the MVVM (Model-View-ViewModel) model to provide optimized data flow and robust efficiency. By integrating the Dagger Hilt Dependency Injection framework, we enhance the modularity and maintainability of the developed application. Additionally, the utilization of Kotlin coroutines enables addressing the complexities arising from asynchronous operations, especially with simultaneous BLE sensor connections. A systematic data extraction method for sensors is also presented to integrate commercial BLE sensors into custom applications. Furthermore, the application provides an alert system for workers in agricultural settings. It instantly alerts users through unique auditory and vibrational cues when abnormal health readings are detected. The proposed functions of the application in detecting abnormal
health parameters and triggering timely alerts were verified through experimental studies. This research contributes to the field of occupational health by presenting a practical solution for developing an open framework smartphone application for agricultural workers.
DESIGN AND DEVELOPMENT OF A SMARTPHONE APP FOR HEALTH MONITORING AND ALERTS FOR FARM WORKERS

BY

OMER OZTOPRAK
© 2023 Omer Oztoprak

A THESIS SUBMITTED TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE MASTER OF SCIENCE

DEPARTMENT OF MECHANICAL ENGINEERING

Thesis Director:
Dr. Ji-Chul Ryu
ACKNOWLEDGEMENTS

Firstly, my deepest appreciation goes to my advisor, Dr. Ji-Chul RYU. Throughout my M.S. journey at NIU, especially during my thesis, his guidance, encouragement, and invaluable insights were pivotal. His unwavering patience and support ensured a smooth completion of my study. I’d also like to convey my sincere thanks to Dr. Sachit Butail for imparting knowledge through their outstanding courses. Lastly, the constant love and belief from my family and friends, paired with their relentless motivation, has been a guiding light for me over the years.
DEDICATION

To my mother and father
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nomenclature</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation and Objective</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Literature Review</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>6</td>
</tr>
<tr>
<td>2 Background</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Mobile Health</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Wireless Communication: Bluetooth Low Energy</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Mobile Application Development: Android</td>
<td>13</td>
</tr>
<tr>
<td>3 Methodology: Structure and Design</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Wearable Biosensors Selection</td>
<td>16</td>
</tr>
<tr>
<td>3.2 App Architecture Design</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Data Management</td>
<td>24</td>
</tr>
<tr>
<td>3.4 BLE Connection</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Alert Module Design</td>
<td>29</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>6.1</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>61</td>
</tr>
<tr>
<td>6.3</td>
<td>63</td>
</tr>
<tr>
<td>References</td>
<td>65</td>
</tr>
<tr>
<td>Appendix: BLE Manager Class</td>
<td>70</td>
</tr>
</tbody>
</table>
NOMENCLATURE

ADB  Android Debug Bridge
ADB  Light emitting diode
ATT  Attribute Protocol
BLE  Bluetooth Low Energy
bpm  beats per minute
DI   Dependency Injection
ECG  Electrocardiogram
GATT Generic Attribute Profile
GSM  Global System for Mobile Communications
HCI  Host Controller Interface
IDE  Integrated Development Environment
IMU  Inertial Measurement Unit
IOS  Iphone Operating System
IoT  Internet of Things
L2CAP Logical Link Control Adaptation Protocol
MEMS Microelectromechanical Systems
MTU  Maximum Transmission Unit

MVVM  Model-View-ViewModel

RF  Radio Frequency

SMP  Security Manager Protocol

SpO2  Saturation of peripheral oxygen

UI  User Interface

UUID  Universally Unique Identifier

WBAN  Wireless Body Area Network
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Packet format</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Verification of Oximeter readings</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Temperature data: Walnut and Koogeek sensors readings with error calculation</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Health Data Records: Retrieved from mobile device’s local storage</td>
<td>58</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Agricultural workers encounter physical challenges [5].</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Architecture of the first tier of Wireless Body Area Network</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>BLE Protocol Stack</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>GATT data hierarchy</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>MVVM Design pattern representation</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>a) Walnut thermometer GATT server discovery. b) Incoming byte arrays from 0xFFF1 characteristic</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>a) A fingertip oximeter by Healthtree [37] b) A smart infant thermometer by Walnut Cares [38]</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>A 9-Axis Inertial Measurement Unit (LPMS2 made by LP-Research Inc.) [39]</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>MVVM architecture in BLE operations: A visualization of data flow and component interactions</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>a) Depiction of the fixed sensor coordinate system utilized by the LPMS-B2 9-Axis IMU sensor. b) Placement of the IMU sensor on the upper-middle back region of a user.</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Trunk inclination and maximum holding time</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Presentation Layer</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Health monitoring and alert mobile application architecture</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Flowchart for scanning mechanism</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Flowchart for Ble connection and obtaining data</td>
<td>41</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.5</td>
<td>Flowchart for Ble connection and obtaining data.</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>The data storage operations focusing on local storage.</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>a) IMU sensor on a foam plate, secured to the upper back b) Sensor setup on user with IMU on back, thermometers on the chest, and oximeter on the finger.</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>a) The developed app’s home screen with a clickable image for initiating BLE connections. b) App’s data display with established BLE connections and data monitoring.</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Torso inclination exceeds the 60-degree threshold.</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>a) O2 saturation drops below 94%, pop-up notification. b) Body temp &gt; 100°F, pop-up notification. c) Heart rate &gt; 120, pop-up notification. d) Trunk angle &gt; 60°, pop-up notification. e) Trunk angle 20°- 60°, alert then pop-up notification.</td>
<td>57</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 Motivation and Objective

The increasing integration of technology into the field of health and safety has opened new ways for real-time monitoring and preventive measures, especially in occupational settings where risks are high. Agricultural workers, which make up an important part of the labor force, are often exposed to numerous physical and environmental dangers. Despite the importance of their roles, these workers are vulnerable to negative health results caused by extreme temperatures, humidity, solar radiation, and demanding physical tasks [1].

As climate change progresses, the hottest summer temperatures continue to surpass the previous records and affect a wider range of areas with severe heat waves. When workers are forced to enter exhausting outdoor activities with minimum control over heat exposure, the risk of heat-related disease increases and significantly affects their general health and productivity [2]. From 1992 to 2006, there were a total of 423 reported cases of heat-related deaths due to heat among agricultural workers in the United States [3]. In addition, these workers often suffer from musculoskeletal disorders due to repetitive movements, awkward postures and daily tasks that require heavy lifting (as shown in Figure 1.1). These conditions often result in chronic pain, disability, and decreased efficiency [4]. It is important to emphasize that chronic musculoskeletal disorders are not rare. It is rather a common issue among farm workers, which is closely tied to the common working positions they need
to maintain. Therefore, simultaneous monitoring of these physical exposures and instant feedback mechanisms is very critical to prevent potential injuries.

Figure 1.1: Agricultural workers encounter physical challenges [5].

To these challenges, this thesis presents research on a real-time health monitoring and warning system specifically designed for farm workers. The proposed system extends beyond the collection of data by adding a warning mechanism that triggers alarms when the following parameters exceed the preset threshold values. This feature could mitigate the increase in health risks by enabling farm workers to take quick action. Since the communication technology that connects multiple wearable biosensors plays an important role in the efficiency of such system, this thesis also explores the use of the Bluetooth Low Energy (BLE) communication, commonly applied in IoT applications. Effective management of multiple simultaneous connections is a critical aspect of such systems. However, the lack of extensive knowledge and well-organized resources leads to difficulties in fully leveraging the capabilities of the BLE technology.

The ultimate goal of this research is to contribute significantly to the prevention of heat-related diseases and musculoskeletal injuries among agricultural workers and to support their resilience to physical exposures. To this end, a real time health monitoring and alert system, as a smartphone app, is developed in this thesis. The development of a system will not only
serve as an immediate technological solution but will also form the basis for further research. It can also help establish potential regulations and guidelines to improve worker safety and health in high-risk occupations such as farming.

1.2 Literature Review

Kim et al.’s study involved creating a wearable sensor equipped with a single BLE module [6]. This sensor can transmit crucial health metrics heart rate, respiratory rate, and body temperature in real-time to a smartphone application. Their primary target was agricultural workers. The study aimed to address limitations found in existing devices, such as the likelihood of data loss, user discomfort, and low temporal resolution. By doing this, the research improved the practicality and efficiency of remote health monitoring, facilitating better health management and timely interventions for the user.

In another study, the authors designed an innovative application of Classic Bluetooth technology by incorporating it into an IMU [7]. Two IMU sensors with built-in Bluetooth modules are used to monitor movements that pose injury risks among construction workers. With real-time monitoring, the device could instantly identify abnormalities and communicate this information via Bluetooth Classic to the worker’s smartphone. This immediate feedback allowed the workers to adjust their movements, accordingly, helping to prevent potential injuries.

Singh and Ricke examined a wide range of applications for commercial BLE sensors, focusing on constructing an open messaging protocol using the host controller interface to obtain raw data from diverse wearable sensors, both commercial and prototypes [8]. In their research, they developed an Android application that can connect with several heart rate sensors at once, with the intent to explore how elements of the app might be
integrated into an open data framework. Their proposed framework’s potential to handle multiple BLE connections running simultaneously raised the possibility of more thorough health monitoring and better data collection. In another study, the researchers created a framework that facilitated multiple Classic Bluetooth connections with certain devices via a smartphone application called mHealthDroid [9]. Their work emphasized the importance of open data frames for the integration of various biosensors, fostering the development of more sophisticated health applications.

An innovative study by Yadav et al. used the lightweight PM6100 Palm Patient Monitoring wearable device to continuously monitor six critical health parameters [10]. Collected data was transmitted in real-time to a smartphone app via a single BLE connection, increasing the accessibility of crucial health data. Another research effort resulted in the creation of a Remote Health Monitoring and Alert System [11]. This system successfully integrated several prototype sensors along with an RF and a GSM module, enabling the monitoring of critical vital signs. Through a wearable device, it sent alerts concerning potentially hazardous situations, thus facilitating proactive healthcare measures.

A team of researchers expanded on these foundational studies by developing an IoT-enhanced ECG system [12]. Their system facilitated remote, real-time monitoring of cardiovascular health, with the collected data stored on the ThingSpeak cloud for accessibility via a mobile application. Moreover, Petropoulos, Sikeridis, and Antonakopoulos introduced a wearable device designed to monitor and correct the user’s sitting posture [13]. The device employed wireless (Zigbee) motion sensors (IMUs) attached to the user’s back and the collected data was processed through an IoT mobile application.

Another group of researchers created a comprehensive system for continuous monitoring of medical data such as pulse rate, body temperature, and oxygen level for the elderly [14]. The system integrated an Arduino microcontroller board with relevant sensors, and the
collected data was consistently transmitted to a master computer functioning as a cloud server. This enabled caregivers to access real-time medical data via their edge devices.

Additionally, another study provided a practical solution to remote health monitoring in pandemic conditions [15]. They presented a wireless body temperature and oxygen saturation monitoring system that was based on Android smartphones. This system utilized the MAX30205 thermometer sensor and the MAX30102 module to detect body temperature and blood oxygen saturation levels, respectively. The data was collected by an ESP32 microcontroller on the TTGO board and transmitted via Bluetooth to Android devices.

Some other studies have used wireless communications other than Bluetooth technology. Mahmud et al. designed a health monitoring system that leveraged Wi-Fi for communication between mobile phone accessories and the phone itself [16]. The system was able to provide instant alerts for various health metrics. Another study by Petropoulos et al. introduced a wearable device that used Zigbee technology and an IMU to monitor user posture via a smartphone [17]. Taylor and Serif also ventured into remote patient monitoring by using Lorawan for communication in a wearable ECG monitoring system [18].

In conclusion, recent studies emphasize the growing significance and utilization of wireless technologies, particularly BLE and IoT, in the healthcare industry for real-time tracking of various health metrics. From monitoring heart rate and body temperature to assessing motion and posture, these technologies offer a comprehensive view of health data to both users and healthcare professionals. The survey of the previous studies suggest that most rely on BLE technology, using prototype sensors to measure and transmit multiple health data via single connections. Also, one can observe the development of multiple BLE connections where data from up to four different heart rate sensors are monitored simultaneously, with the intention of exploring how aspects of the app can be integrated into an open data framework. Despite these advancements, existing resources are not sufficiently clear or accessible for the straightforward development of health monitoring applications capable of handling multiple
simultaneous connections from heterogeneous biosensors. Thus, the literature suggests a need for a novel framework that could expedite the development of health monitoring apps requiring multi-BLE communication.

1.3 Contributions

The primary contribution of this research is the development of a real-time health monitoring and alert system tailored to the unique needs of farm workers, a field where managing multiple simultaneous BLE connections from heterogeneous biosensors presents complex challenges. These challenges stem from potential sensor interference and data loss. This study introduces an innovative multi-sensor communication framework that successfully manages multiple BLE connections without these issues. Additionally, it includes the design and development of an easily modifiable and imitative Android application using the MVVM architectural techniques, making the source code understandable and adaptable for similar applications. Collectively, these contributions fill a significant gap in the literature, providing a clear framework for handling multiple BLE connections and offering an easily understandable and modifiable real-time health monitoring and alert system.

1.4 Outline

This thesis is organized into several distinct chapters, each focusing on a key aspect of the research. Chapter 2 provides the background necessary to understand the broader context of the study, including an in-depth exploration of sensor technologies, WBAN, BLE, and Android. Chapter 3 details the methodology employed, covering topics such as wearable biosensor selection, the architecture of the Android application, BLE connection, data man-
agement, and the alert system. Chapter 4 is mostly covering the technical implementation of Chapter 3. Chapter 5 presents the experimental setup and procedure, followed by detailed descriptions on data monitoring, alert mechanism and data storage results, and a comprehensive discussion of these findings. Finally, Chapter 6 concludes the thesis with reflections on the future work and conclusion, offering insight into possible directions and implications.
CHAPTER 2
BACKGROUND

2.1 Mobile Health

Mobile health (or mHealth for short) is a rapidly evolving field due to the relevance of devices such as smartphones and tablets. These small devices are particularly useful for communicating with healthcare providers and sharing important health-related data. Therefore, smartphones play a crucial role in the mHealth field, serving as the central hub for health monitoring and data exchange. Smartphones also have the capacity to collect, evaluate and transmit health-related data to both healthcare providers and users using a variety of sensors and applications [19]. Such connectivity allows for real-time monitoring of data and immediate intervention, improving the overall efficiency of healthcare.

Wireless Body Area Network (WBAN) refer to specialized sensor networks that employ wireless sensor nodes positioned within an individual’s body to gather crucial physiological information, including blood pressure, body temperature, heart rate, and blood glucose levels, etc. [20]. These networks facilitate the remote monitoring of health, so enabling healthcare providers to get a continuous flow of information regarding a patient’s health condition. WBAN possess the capability to be categorized as either wearable or implantable, thereby expanding the range of potential applications for these networks. Data transmission for WBANs can be classified into three primary tiers. The first tier involves communication within the body’s range, where a central hub is positioned in close proximity to the user, and other tiers mostly include the transmission of health data to healthcare providers or remote
servers [21]. In this study we focus on developing the first tier of the WBAN so other tiers won't included to our system. Figure 2.1 depicts a representative WBAN system and its first tier of communication. In this network, a variety of wearable sensors placed on or within the human body communicate with a central device such as a smartphone via a wireless communication method, enabling the user to observe real-time health data.

![Figure 2.1: Architecture of the first tier of Wireless Body Area Network](image)

Wearable sensors have become very popular and common in healthcare due to their ability to measure crucial health metrics such as blood pressure, glucose level, body temperature, oxygen saturation, and heart rate. These measurements can be transmitted either wirelessly or with a wired link to a more advanced device such as a smartphone or micro controller for real-time display and monitoring. These systems can be as practical as any other accessories we use in daily life such as wristwatches, and sunglasses, as they provide easy access to health metrics and a user-friendly interface for health monitoring [22]. Wearable sensors can improve their health and well-being in real-time as they provide real-time feedback to
the user to take precautions. The integration of wearable sensors with smartphones and other mHealth components like WBAN enhances the capabilities of health monitoring and represents a vital step in transforming healthcare delivery.

### 2.2 Wireless Communication: Bluetooth Low Energy

Bluetooth Low Energy is one of the most widely used protocols in WBAN due to its prevalence in smartphones. Introduced in 2010, BLE is a modified version of Bluetooth Classic and its main development idea focuses on low power consumption [23]. BLE saves power by maintaining a connection with minimal energy consumption, using short bursts of data transmission, and entering sleep mode between communications. Therefore, it enables power efficiency which is necessary to develop sensors that are small and compact with a long battery life. These features enable integration into wearable devices and other health-related products that require continuous operation and energy saving.

The BLE protocol stack is the architectural framework of the BLE communication. As it is inherently straightforward, It is possible to break the BLE protocol stack down into three primary parts: the Controller, the Host, and the Application Layer (See Figure 2.2). The Controller part consists of two main parts which are the Physical Layer and Link Layer is the lowest level of the stack and is responsible for radio operations. On the other hand, it is the responsibility of the Host to provide higher-layer functionality that is capable of supporting a variety of applications. The host layer consists of five components as shown in Figure 2.2. The HCI serves as a communication bridge, facilitating interaction between the Controller and the Host layers. The top application layer is responsible for user interaction [24].
The Attribute protocol is a fundamental component of BLE, responsible for the organization of data into attributes. These attributes consist of a UUID, permissions, a value, and a handle. The ATT system operates based on a server-client architecture, wherein the server is responsible for maintaining attributes and the client is responsible for requesting or altering these attributes. The standardization of data structure by ATT facilitates efficient and adaptable communication within the BLE protocol, hence fostering interoperability and uniformity among different devices [25].

Expanding upon the concept of ATT, the GATT introduces a hierarchical structure that categorizes data into entities referred to as Services and Characteristics. Services group attributes that are connected, whereas characteristics define specific data values and information. GATT facilitates the interaction between devices by establishing distinct roles,
namely the GATT server and client, and outlining specific protocols for tasks such as data reading, writing, and subscribing to data changes. For example, Figure 2.3 provides a visual representation of the GATT data hierarchy. It consists of three key components: services, characteristics, and descriptors. Services may include zero or more characteristics, and each characteristic may further contain zero or more descriptors. This strict hierarchy is a standard that all GATT-compatible BLE devices adhere to, ensuring efficient and organized attribute management. The standardized representation of the GATT facilitates the easy sharing of information among varied BLE devices, hence increasing communication within the BLE ecosystem [26].

The GAP governs how BLE devices discover, connect, and communicate with one another. GAP controls advertising and scanning, connection establishment, and security fea-
tures. By defining different roles like Broadcaster, Observer, Peripheral, and Central, GAP is responsible for how devices interact and function. As the foundation for Bluetooth interactions, GAP provides the structure needed for standardized and secure communication, making it a vital component in BLE’s operation, including in applications like health monitoring [27].

The fore mentioned concepts can be illustrated with an example. Imagine a wearable fitness band equipped with a heart rate sensor, intended to be connected to a smartphone app for real-time heart rate monitoring during exercise. Using the GAP, the sensor advertises its presence, and the smartphone establishes a secure connection once it discovers the sensor. The heart rate data is organized into attributes by the ATT, and these are grouped into a “Heart Rate Service” by the GATT. Within this service, various characteristics represent different aspects of heart rate, such as current and maximum rates. The smartphone, acting as the GATT Client, reads these values from the heart rate sensor (the GATT Server) and can subscribe to notifications for automatic updates. The coordination of GAP, ATT, and GATT ensures a seamless and efficient communication process, allowing the heart rate sensor and smartphone to interact in a structured and secure manner, providing the user with continuous, real-time feedback on their health metrics.

2.3 Mobile Application Development: Android

Nowadays, mobile phones have become a lifestyle rather than just a means of communication by providing a myriad of applications. Mobile application development is the process of creating software for mobile devices such as smartphones, tablets, and personal digital assistants [28]. Android and iOS are the two major operating systems for mobile devices and make mobile application development widespread by providing many functionalities. Glob-
ally, Android takes the biggest portion of the market as iOS is limited to certain devices and a brand.

Android is a software platform and operating system designed for mobile devices, which is built upon the Linux kernel and has been developed by Google in collaboration with the Open Handset Alliance [29]. Due to its open source platform nature, it is compatible with a wide variety of devices in the market. On the contrary, IOS is not open-source and exclusive to iPhones and iPads developed by Apple [30]. In addition, Android’s open-source structure provides greater flexibility and a broader selection of hardware options. For these reasons, Android OS was selected in this thesis.

An IDE is a software application that provides a comprehensive set of tools for software development. Android Studio is the official IDE for Android OS that provides robust tools to create test and build an app, including code editing, debugging, and testing. In addition, it uses Kotlin and Java programming languages [31]. Flutter, a toolkit developed by Google, enables cross-platform development, allowing a single code base to run on both Android and iOS, and utilizes Dart programming language [32]. MIT App Inventor provides a visual drag-and-drop interface, making the creation of simple Android apps accessible [33]. Among these Android Studio was selected, as the official Android IDE with native BLE API integration, rich debugging tools, and vast community support, offers the most streamlined and efficient environment for Bluetooth Low Energy app development.

Among the various tools available for Android app development, from the traditional XML-based UI design to third-party UI frameworks, each offers its own advantages. Therefore, a recently developed Android tool Jetpack [34] is utilized in this thesis to improve the user interface design process. As an Android-specific toolkit, it allows developers to create engaging and adaptable user interfaces. Jetpack is designed to work seamlessly with Kotlin, our chosen programming language in this study.
An architectural pattern is a reusable solution to a commonly occurring problem in software architecture, providing a structured approach to design scalable, maintainable, and consistent systems. The MVVM architectural pattern has emerged as a favored choice in Android development. MVVM ensures a clean separation of concerns: the Model manages data and core logic, the View handles user interface interactions, and the View Model acts as an intermediary, efficiently processing and relaying information between the Model and the View [35]. This separation not only promotes organized and maintainable code but also allows for modular development, where changes in one component minimally affect others. Figure 2.4, which shows the interconnections and roles of each component within the pattern.

Figure 2.4: MVVM Design pattern representation
CHAPTER 3
METHODOLOGY: STRUCTURE AND DESIGN

In this chapter, we introduce an overview of the proposed health monitoring and alert system for farm workers. This chapter focuses on the necessary hardware components for data collection, the methods and technologies employed, and the overall structure of the system. Firstly, we discuss the selection of biosensors and hardware components for the system. This includes the selection criteria and the general approach to finding the most appropriate biosensors. Subsequently, we delved into the software development environment selection, and Android Studio is selected as the optimal development environment. For creating a sophisticated app capable of managing multiple BLE connections, the UI toolkit Jetpack Compose, and the Model View View Model design pattern are employed. Later in this chapter, we introduce the methodologies used for determining the app’s overall structure, data management BLE connections, and alert system.

3.1 Wearable Biosensors Selection

In health monitoring applications, measurements such as heart rate and oxygen saturation provide vital information about a patient’s condition. These health measures are especially important for farm workers who often experience overexertion and pose respiratory risks. Additionally, farm workers often work in direct sunlight, lifting heavy objects and bending at wide angles. A set of biosensors such as an oximeter (for heart rate and oxygen), a body
temperature sensor, and an IMU sensor (for body orientation) are suitable for monitoring these factors.

The selection process of a particular device for these sensors is as follows. Given that BLE is the most common communication protocol used in these devices, the primary selection criterion was BLE compatibility. This study emphasized the use of built-in BLE sensors and aimed to create an open framework for their simultaneous connection.

The second criteria for selection were whether the devices allow open access to their data. While many sensors with built-in BLE modules are available, a significant number of them do not offer open access to their raw data, relying instead on proprietary platforms for data exchange. This study specifically sought to develop an open framework for sensors granting open access to their data. Although most commercially available IMUs met these criteria. However, few commercially available oximeters and thermometers provide raw data access.

To address this issue, a third-party app, LightBlue [36], was employed to evaluate the accessibility and processing capability of raw data from different biosensors by displaying the sensors’ GATT server. The GATT server provides a data channel that reports its identifying characteristics. The user can simply click and listen to the notification from the relevant data channel and determine the byte array in which data is stored. A screenshot from the LightBlue app, depicting the GATT server of a thermometer (Walnut Infant), is presented in Figure 3.1a. For example, to obtain display data from the sensor, one must select the characteristic channel marked by 0xFFF1. Figure 3.1b illustrates the incoming byte array from the thermometer. Therefore, this technique proved to be a reliable method for assessing a sensor’s appropriateness for our open data access requirement.
Figure 3.1: a) Walnut thermometer GATT server discovery. b) Incoming byte arrays from 0xFFF1 characteristic.

Through this selection procedure, a fingertip oximeter by HealthThree and a smart infant thermometer by Walnut Cares were selected as suitable biosensors for this study (See Figure 3.2). These devices provided access to the raw data for oxygen saturation, heart rate, and body temperature, respectively. Despite the lack of extensive documentation, the exploratory process with the third-party app enabled the integration of these sensors with our custom Android application. Consequently, this empowered our application to accurately monitor and analyze the key health parameters required for this study.
Lastly, an LPMS-B2 9-axis inertial measurement unit [39] was selected as the IMU sensor (See Figure 3.3.). This sensor was chosen due to its built-in BLE capability and its ability to provide processed orientation data, including Euler angles and quaternion data, in addition to the raw accelerometer, magnetometer, and gyroscope data. This IMU contains three different MEMS sensors (a 3-axis gyroscope, a 3-axis accelerometer, and a 3-axis magnetometer) for accurate, high-speed orientation data around all three axes. Furthermore, it has built-in temperature and barometric pressure sensors that enable accurate altitude determination. This sensor can transmit data at rates of up to 400Hz and is capable of running all calculations on-board in real-time [39].
Figure 3.3: A 9-Axis Inertial Measurement Unit (LPMS2 made by LP-Research Inc.) [39]

In contrast to the oximeter and the thermometer, the LPMS-B2 is an open-source sensor with extensive documentation. On their website, the details about this sensor required for BLE communication are readily available, including its specific service (0x180F) and characteristics (0x2A19). The thorough description about the LPMS-B2 format data, written in the user manual, that uses its own protocol called LP-Bus enables developing a straightforward method for extracting raw data.

In conclusion, the selection of wearable biosensors for this research was conducted through a rigorous process that took into account the open-access data feature of the biosensors and their compatibility with third-party apps. The selected biosensors, the Walnut Infant Thermometer, the HealthThree Pulse Tip Oximeter, and the LPMS-B2 9-axis IMU, provide the necessary data for real-time monitoring of the physical exposures of agricultural workers.
3.2 App Architecture Design

The Android application is developed using the Model View View Model architecture, a popular method for developing well-organized and simple-to-maintain apps. This architecture divides the structure of the application into three major components: the Model, the View, and the View Model. The app’s capability to connect with multiple BLE devices allows us to examine each component and how data flows between them. One of the strengths of MVVM is its distinct separation of each component. This split guarantees that any complexities involved with handling these connections are confined within the Model as the number of BLE connections rises. This makes the program more adaptable to modifications since changes to connection management have no effect on the presentation or user interface layers.

The View is essentially the user interface of the application, providing the visual canvas through which users interact and engage with its features. Within our application, there are two primary interfaces: the initial screen that welcomes users upon launching the app, and a main screen dedicated to real-time data monitoring. This latter screen prominently displays health metrics, showcasing heart rate in beats per minute, oximeter readings as percentages, and body temperatures in Fahrenheit. A standout feature is the inclusion of visual diagrams representing a human figure. This diagram aids users by illustrating the inclination of the torso, to help perform posture assessments. The entire of this user interface was created using Jetpack Compose – Google’s advanced UI toolkit. Its adoption ensures a user experience that’s both immersive and user-friendly.

The Model serves as the robust backbone of the proposed application, diligently managing a multitude of tasks that may not be immediately visible to the user. Through Android’s advanced toolset, the Model identifies nearby BLE devices and establishes reliable connec-
tions with them. Once these connections are secured, the Model embarks on the continuous process of gathering intricate data streams from each device. But its role doesn’t end with mere data collection. The Model also undertakes the responsibility of organizing this influx of data, sorting it, and interpreting it to ensure that it aligns with the application’s needs. Post this refinement process, the Model ensures the safe and structured storage of this data within our app’s local storage. In doing so, it guarantees that when the application needs to retrieve or analyze this information, it’s readily available in its most coherent and usable form.

Filling the gap between Model and View, the View Model acts as an intermediary, providing seamless communication and data transformation. Basically, the View Model notes when the Model is updated with new data or state changes from BLE device reads or other data sources. Acting as an observer, it identifies these data changes in the Model and then processes the raw data or converts it into a more digestible format. This processing step is crucial as raw data from the model may not always be directly viewable or in a user-friendly format. After the data has been refined, the View Model allows it to be passed to the View, allowing the UI to present the data to end users in an understandable and interactive way. Given the asynchronous nature of BLE connections, where data can flow occasionally or operations may not be executed instantly, View Model uses Kotlin’s coroutines and streams. These tools effectively manage asynchronous tasks, ensuring that data from the Model is seamlessly and quickly transferred to the View without delays or interruptions to the user experience.
Figure 3.4: MVVM architecture in BLE operations: A visualization of data flow and component interactions.

The three main components in our MVVM architectural pattern are; Model, View Model, and View are prominently represented by separate rectangles arranged in order from left to center-right. The leftmost “Model” section provides a miniature flowchart summarizing BLE operations chronologically. Starting with “Detect BLE Devices”, going through “Connect to Devices”, “Collect Data”, “Process Data”, and ending with “Store Data”. These stages are connected to each other by vertical arrows, providing the logical flow of operations. There are bidirectional arrows that connect the components. One between “Model” and “View Model” is the symbol for persistent data flow and intermittent commands or requests. At the same time, the arrow connecting the “View” and the “View Model” symbolizes two-way communication: the View is constantly updating based on data, while at the same time passing user interactions back into the View Model. Annotations next to these arrows explain their purpose, providing a clear understanding of the complex interaction between components and data flow in our application.
3.3 Data Management

There are four primary data sets that are utilized to monitor the health condition of a farm worker. These include oxygen saturation and heart rate data obtained from the oximeter, body temperature data acquired from the thermometer, and orientation data obtained from the IMU. In regard to the oximeter and thermometer, we have identified the relevant attributes and functionalities for data extraction. However, the extraction process necessitates a detailed examination of their byte array. The byte arrays were seen and analyzed using the LightBlue application, resulting in the identification and extraction of the necessary data. In contrast, the process of retrieving data from the IMU was rather uncomplicated due to the availability of the manufacturer’s communication format table. The process of extracting data from each sensor is elaborated on in this section.

The Walnut Infant Thermometer’s data was accessed through a specific service (0xFFF0) and characteristic (0xFFF1) and incoming byte arrays were observed. An example of data array is given by

\[0x690172014B00220BB065\]

Since it is in hexadecimal format, and the third most significant byte is 0x72, which is a particular byte related to the temperature data. By converting the byte to decimal, subtracting it from the number 104, dividing it by 10, and then adding 36, the body skin temperature \(T_c\) in Celsius can be obtained as

\[T_c = \frac{104 - \alpha}{10} + 36\]  \hspace{1cm} (3.1)

where \(\alpha\) denotes the third byte of the byte array in decimal.

The temperature can further be converted to Fahrenheit, \(T_f\), given by

\[T_f = \frac{9}{5}T_c + 32\]
\[ T_f = \frac{9}{5} \cdot T_f + 32 \]  

This correlation was tested and verified for temperatures between 90 and 110 degrees Fahrenheit, which is the typical human body temperature range. Similarly, for the Health Three Pulse Tip Oximeter, a specific service (0xFFE0) and characteristic (0xFFE1) were identified. Four byte Arrays of different sizes have been observed and one of them always starts with FF, and that one includes the oxygen saturation and heart rate data. With the byte array, the decimal value of the fifth byte represents the oxygen saturation, and the sixth byte represents the heart rate. An example of an oximeter incoming byte array is given by 0xFF440100624D7B09. With this byte array, the fifth byte is 0x62, which is 98 in decimal format, represents oxygen saturation, and the sixth byte of 0x4D represents the heart rate of 77 (bpm).

The LPMS-B2 IMU sensor utilizes the LP-BUS communication format. In order to extract raw data, such as orientation, from the sensor’s data channel, we must know its data packet structure. In addition, the manufacturer offers proprietary software that enables users to request specific data from the sensor. For instance, it enables the customization of the sensor to exclusively send gyroscope data. Table 3.1 displays the data packet format of the LPMS-B2 IMU sensor. It can be seen that the initial 6 bytes and the last two bytes remain constant regardless of the specific data being requested. To extract raw data from a device, the data packet must first be obtained using the initial and last bytes. Once the data packet is obtained, the specific orientation data can be found by looking up the corresponding byte in the data format table. For our application, we only utilized Euler angles as the preferred method for obtaining orientation data from the sensor.
Table 3.1: Packet format

<table>
<thead>
<tr>
<th>Packet byte no.</th>
<th>Content</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3Ah</td>
<td>Packet start</td>
</tr>
<tr>
<td>1</td>
<td>01h</td>
<td>OpenMAT ID LSB (ID = 1)</td>
</tr>
<tr>
<td>2</td>
<td>00h</td>
<td>OpenMAT ID MSB</td>
</tr>
<tr>
<td>3</td>
<td>09h</td>
<td>Command no. LSB</td>
</tr>
<tr>
<td>4</td>
<td>00h</td>
<td>Command no. MSB</td>
</tr>
<tr>
<td>5</td>
<td>34h</td>
<td>Data length LSB (56 bytes)</td>
</tr>
<tr>
<td>6</td>
<td>00h</td>
<td>Data length MSB</td>
</tr>
<tr>
<td>7-10</td>
<td>xxxxxxxxh</td>
<td>Timestamp</td>
</tr>
<tr>
<td>11-14</td>
<td>xxxxxxxxh</td>
<td>Euler data x-axis</td>
</tr>
<tr>
<td>15-18</td>
<td>xxxxxxxxh</td>
<td>Euler data y-axis</td>
</tr>
<tr>
<td>19-22</td>
<td>xxxxxxxxh</td>
<td>Euler data z-axis</td>
</tr>
<tr>
<td>...</td>
<td>xxxxxxxxh</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>xxxxxxxxh</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>xxh</td>
<td>Check sum LSB</td>
</tr>
<tr>
<td>64</td>
<td>xxh</td>
<td>Check sum MSB</td>
</tr>
<tr>
<td>65</td>
<td>0Ah</td>
<td>Message end byte 2</td>
</tr>
<tr>
<td>66</td>
<td>0Dh</td>
<td>Message end byte 1</td>
</tr>
</tbody>
</table>

The accurate interpretation of sensor data necessitates the careful consideration of sensor placement to ensure reliable data acquisition. The pulse oximeter is a finger-type device that should be affixed to the user’s index finger in order to accurately monitor oxygen saturation. For accurate measurement of body temperature with the thermometer of our choice, it is recommended to affix the thermometer to the user’s chest. However, the positioning of
the IMU sensor is a more complex task that requires a comprehensive understanding of its coordinates. The IMU was strategically placed on the middle-upper back of the user (See Figure 3.5a). This location was chosen as it provided an ideal spot for measuring trunk inclination, a critical factor in evaluating the risk of musculoskeletal disorders among workers. The local and global reference coordinate systems employed in this study are characterized as right-handed Cartesian coordinate systems. In these systems, the X-axis is considered positive when oriented towards the magnetic west, the Y-axis is considered positive when oriented towards the magnetic south, and the Z-axis is considered positive when pointing upwards (with gravity being vertically downward at a magnitude of -1g) [39]. Figure 3.5b illustrates the axis orientation of IMU, which is rigidly attached to the sensors. By aligning the sensor such that the Euler angles around the X-axis represented the trunk inclination, the system could effectively track and analyze the postural inclination of the workers.

Figure 3.5: a) Depiction of the fixed sensor coordinate system utilized by the LPMS-B2 9-Axis IMU sensor. b) Placement of the IMU sensor on the upper-middle back region of a user.
3.4 BLE Connection

Our application is particularly intended to simplify the integration of various BLE devices, resulting in an environment in which these devices may connect with smartphones with ease. The `BLEManager` class is the central component of our app and it is part of the Model Architecture. This class is in charge of the initial discovery phase, which involves continually scanning for accessible BLE devices. When one of our target devices, which includes an IMU called “LPMSB2-6F36F6”, an oximeter called “OXIMETER”, and a device called “Walnut”, is identified by its unique name, it is added to a filtered list of detected devices. This functionality guarantees that our application only focuses on the necessary devices, improving efficiency and responsiveness.

The application initiates a BLE scan, searching for peripheral devices nearby. As each device is discovered, it is assessed to determine if it matches the predefined criteria, in this case by its name (e.g., “LPMSB2-6F36F6”, “OXIMETER”, “Walnut”). Once identified, these devices are added to a list, ensuring no duplicates. The aim is to create a queue of devices ready for connection. Instead of attempting to connect to all discovered devices simultaneously, the application connects to them one by one. When a connection with a device is successfully established, the next device in the queue is targeted. By handling one connection at a time, the application flow remains straightforward. This makes it easier to troubleshoot, reducing complexity in both the code and connection logic. Every BLE connection attempt involves some level of radio frequency activity. By connecting sequentially, the application minimizes simultaneous RF communications, potentially reducing interference and increasing the success rate of connections.

After successful connections are established with devices, the application initiates the data exchange with them by listening to notifications of the relevant data channels. This
starts data stream whenever new data is ready by a peripheral device. At this point, all the incoming data are captured by our application, only particular bytes are related to health data. These data are extracted using the method described in the 3.3. To simplify the process of managing these diverse data streams, the data from each device is encapsulated in a unique data class. The ImuData class wraps the data from the IMU, the OxiData class holds the oximeter’s data, and the TempData class is for the thermometer. By creating these data classes, each set of data is grouped in a structured manner, allowing for more efficient data processing and management. The View Model layer then interacts with these classes, observing changes in data and passing it on to the View for display.

### 3.5 Alert Module Design

The instant alert mechanism is crucial for agricultural workers to take immediate precautions to protect their health from potential physical exposures. There is an acceptable range for each of these health metrics to be considered normal and safe. As readings outside of this range are considered abnormal, the alert mechanism is designed to detect these abnormal readings by comparing readings to threshold values. The oximeter is used to gauge blood oxygen saturation, with a normal range typically falling between 94 percent and 100 percent [40]. A saturation level below this range could indicate insufficient oxygen supply to the body’s tissues. In addition, the skin temperature for a healthy individual should be between 97°F and 100.0°F. Any temperature exceeding 100.0°F may be indicative of fever, potentially suggesting an ongoing bodily response to infection or illness [41]. The normal range of heart rate for adults is between 60 to 100 bpm. Heart rates exceeding 100 bpm could suggest tachycardia, while rates below 60 bpm could imply bradycardia, both of which may be potential indicators of heart-related concerns [42]. Therefore, these sensors’ readings
collectively provide a comprehensive overview of an agricultural worker’s real-time health status.

According to ISO standards [43], a proper standing posture is defined as 0 degrees, with a permissible trunk inclination range of 0 to 20 degrees, signifying an ergonomically safe torso angle. The range of 20 to 60 degrees is tolerable for short periods, with a sustainable limit of 4 minutes at 20 degrees and 1 minute at 60 degrees, beyond which there’s a significant risk of discomfort or musculoskeletal injury. Inclination beyond 60 degrees is deemed unsafe due to the increased likelihood of musculoskeletal disorders. As illustrated in Figure 3.6, the green area signifies an acceptable range, yellow depends on duration, and red is a non-recommended zone. Thus, the precise location and alignment of the IMU sensor are critical for accurately evaluating these postural shifts.

![Figure 3.6: Trunk inclination and maximum holding time](image)

The alert mechanism compares real-time user interface data to predetermined threshold values. The user is notified to take precautions if a reading falls outside the range. A pop-up notification bar is used to notify the user of the abnormal health metric. This is the ideal method because it doesn’t take up any UI space. Additionally, it is anticipated that workers will place their phones in their pockets, so a text-based alert may not be sufficient. To address this issue, the notification will play a custom sound, such as “Abnormal temperature value
detected”, and vibrate the phone in case of noisy environments to alert users. Lastly, to prevent continuous alerts from happening, a delay of five seconds is employed, which is sufficient to alert users without overwhelming them. These features ensure that users are effectively informed.
CHAPTER 4

TECHNICAL IMPLEMENTATION

4.1 Overview

This chapter provides a detailed description of how the design methodologies proposed in Chapter 3 can be implemented on the development of an Android app. This chapter also explains how the essential features work. Therefore, this chapter can serve as a guide for creating a health monitoring and alert system. In that regard, Chapter 4.2 describes the entire procedure from establishing the environment to constructing the BLE service including adding necessary dependencies to the gradle.file and configuring dependency injection, navigation activity, and permission management. The procedure also includes adding Bluetooth permissions to AndroidManifest.xml, introducing user interface components, and implementing Dagger Hilt dependency injection. These are the initial stages in developing the presentation layer of our application. In Chapter 4.3, the architecture of the app is described in detail. The Chapter explains the functionality of each component and how data flows from BLE devices to the app screen as it is crucial for understanding the app’s overall structure. To that end, specific components of the Model, Model View and View design patterns, which are discussed in Chapter 3 will be introduced. After setting up the environment, component introduction, and data transmission between components, a BLE service can be further constructed. The BLEReceiveManager class introduced and explained in Chapter 4.4. This chapter explains how the scanning mechanism operates, how connections are established, and how data is extracted to a specific data class. In this section, it is
briefly explained how general BLE functions operate. Chapter 4.4 addresses an explanation of how multiple BLE devices are simultaneously connected, and how data flows to the Model View and associated data class. Finally, the alert logic, notification compact library, and app implementation are described in Chapter 4.5.

4.2 Setting Up Environments

Android Compose Activities are utilized in the development of the app. Although the XML file approach can also be used, Jetpack Compose was selected since it’s more concise and intuitive. We began with an Empty Compose Activity, a template specifically designed for creating user interfaces using Jetpack Compose. In essence, all the visual elements and user interactions on the screen or within a composable are managed by Jetpack Compose. We created the Navigation, Start Screen, and Data Monitoring Screen using composable functions.

A modular architecture of presentation layer is illustrated in Figure 4.1, depicting component connections. These components primarily relate to our app’s presentation layer, which displays data to users and manages their interactions.

Upon selecting the template, we integrated the necessary external libraries. These dependencies are added to the gradle.file to manage the libraries and frameworks the project requires. For instance, we added Dagger Hilt for dependency injection, Navigation Compose for navigation, and the Accompanist Permissions library to handle permissions. As we aimed to develop an app which uses Bluetooth, we added permissions and required configurations to the AndroidManifest.xml. For BLE operations, location permissions like are crucial, ensuring the app can access device location.
Dependency Injection (DI) simplifies dependency management and control flow in applications. Our application uses Dagger Hilt, a DI framework tailored for Android. The App module is installed in the SingletonComponent, ensuring that dependencies provided by this module are available throughout the app’s lifetime. Within this module, Dagger Hilt injects dependencies for the BluetoothAdapter classes, representing the local Bluetooth adapter on an Android device. Additionally, it provides an instance for the BLEReceiveManager interface. Both components, essential for Bluetooth operations, are readily available in the app through Hilt. Using Dagger Hilt simplifies the structure since it reduces the need to manually add dependencies.

Having set up the environment for User Interface (UI) components, we first tackled navigation component to enable navigation between screens. This composable function takes two parameters: onBluetoothStateChanged and navController. The onBluetoothStateChanged lambda is invoked when the state of the Bluetooth adapter changes. The Navigation func-
tion first establishes a NavHost object responsible for app navigation. This NavHost object, created using the navController parameter, defines two routes, the start screen and data monitoring screen, each with a dedicated composable function for its UI.

A critical aspect of BLE applications is responding to system-wide Bluetooth events. This is achieved through the SystemBroadcastReceiver composable, which monitors specific system broadcasts and triggers related actions. This mechanism ensures app-device Bluetooth state synchronization. The onBluetoothStateChanged lambda in the Navigation function corresponds to the SystemBroadcastReceiver composable.

The MainActivity class serves as the app’s entry point, initiating the app and presenting its UI. This class, upon checking the Bluetooth adapter’s state, will display a dialog if the adapter is disabled. The onCreate method, invoked when the activity first loads, handles initial setups, such as view and data initialization. This method subsequently calls the setContent method, assigning the activity’s content. Here, the Navigation composable function is set as the content. Initially, the Navigation presents the start screen, but based on the Bluetooth state, it may navigate to the data monitoring screen.

4.3 App Architecture

The View component serves as the primary interface layer of the proposed application, housing both the start screen, designed for user commands like initiating BLE connections, and the data monitoring screen which displays data from our three BLE sensors. This screen also integrates an alert mechanism explained in Chapter 4.5 and captures user interactions, prompting the View Model for BLE connection tasks. An illustrative representation of our application’s architecture is depicted in Figure 4.2. This visual guide offers a modular breakdown of communication, starting from user interaction right down to the output which are
alert notifications. It’s designed to simplify the comprehension of the application’s workflow and serves as a point of reference for this section. Specifically, within the data monitoring screen, a *Composable* function represents the UI. This function responds to Bluetooth state changes: on app launch, it checks permissions and, if conditions permit, triggers the `startReceiving()` function from the View Model.

![Diagram](image)

**Figure 4.2:** Health monitoring and alert mobile application architecture

The View Model, next to the View, serves as the core of our application’s presentation logic. It bridges the UI layer with the underlying data management layer. Within the View Model, commands from the user are processed, with each request swiftly identified by the command processor. Depending on the nature of the request, specialized data requesters tailored for IMU, oximeter, and temperature data retrieve relevant information from the Model layer. To ensure real-time data reflection in the UI, the View Model employs state-backed properties using Jetpack Compose’s `mutableStateOf()` function. These properties mirror data from BLE devices and are observed by the UI components. By subscribing to
the MutableSharedFlow streams from the BLE Manager using a custom collect function, the View Model ensures immediate capture and representation of data from the BLE devices. Notably, it also has the capability to store sensor data on local storage through a custom function.

At the heart of our system is the Model, the main architectural component responsible for data management and BLE connections. Integral to this model is the BLE Manager, which scans and establishes stable connections to devices. Once a device is successfully connected, the data flows into the View Model through the Model’s connection system. Aiding this flow is the Data Management Layer, dedicated to the processing of real-time data from connected BLE devices. Furthermore, it’s equipped with three MutableSharedFlow to manage specific streams of data for IMU, oximeter, and the thermometer. The data models play a crucial role in this architecture. The IMU Data model captures properties associated with an IMU device, detailing orientation data presented in terms of Euler angle. The Oxi Data model provides a snapshot of oximeter readings, encapsulating SpO2 levels and pulse rate. Temp Data model focuses on thermometer’s temperature data.

After being processed, these data models return the refined data to the View Model. The LiveData or state attributes are then updated by the View Model. The View continuously monitors LiveData attributes. This ensures that users are presented with accurate data in real time. Then these data are compared to preset threshold values for each health metric. If the data exceeds the threshold, the alert function triggers a sound, vibration, and text message to the user. This completes one cycle of the app, which begins with user input and ends with the data being displayed, alerted to the user, and saved.
4.4 BLE Connections

This sub-chapter provides BLE operations for the functionality of our application, more specifically how our app interacts with BLE devices from detection to data extraction in more detail. First the scanning mechanism of the application is outlined with the flowchart illustrated in Figure 4.3. It details the sequence of actions performed from the initiation of BLE connections to the discovery of devices.

The scan mechanism initiates with the “Start” phase, symbolized by the red oval. Upon activation, the system initializes the `BLEReceiveManager`, a critical component that facilitates direct communication with BLE devices.

Subsequently, certain constants essential for the operation, such as device name, device’s services and characteristic UUID, are set up. This ensures that the mechanism knows precisely what devices or services it should be interacting with. Following this, shared data flows like `IMUdata`, `OxiData`, and `TempData` are initialized. These flows represent the data channels that would collect and manage information from the connected devices.

The BLE scanner is then set up. This scanner’s responsibility is to actively search for nearby BLE devices. Next, the system checks whether it’s currently scanning. If not, the mechanism is routed back to set up the scanner again. If scanning is active, the system proceeds to discover devices using a specific `scanCallback` function.
Start
Initialize BLEReceiveManager
Initialize constants
Initialize shared flows
Setup bleScanner

Is Scanning?

Discover devices using scanCallback
No

All 3 devices discovered?
Yes

Stop Scanning

Yes

Copy scanResults to devicesToConnect

No

Devices left to connect?

Yes

Connect to IMU_DEVICE_NAME

Device name

Connect to Oximeter DEVICE_NAME

Connect to Temperature DEVICE_NAME

No

End

Figure 4.3: Flowchart for scanning mechanism
The next decision-making point is to determine whether all three target devices have been discovered. If not, the system loops back and continues the discovery process. Once all devices are identified, the scanning process stops.

The discovered devices are then stored to the `scanResults` list. The `scanResults` is copied over to `devicesToConnect`, a list that represents devices pending connection. The mechanism then checks if there are any devices left to connect in the `devicesToConnect` list. If no devices remain, the system routes towards the “End”. If there are devices left, it inspects the device name to determine which specific device it’s dealing with.

Based on the identified device names, the system could connect to the IMU, the oximeter, or the thermometer. If none of these names match the current device’s name, the system handles this as an unknown device name, ensuring that unrecognized or unexpected devices do not interfere with the mechanism. Finally, after processing all devices and managing any unexpected scenarios, the scan mechanism concludes its operation, symbolized by the “End” phase.

Next, the process of establishing a direct BLE connection with the devices requires a series of steps as outlined in the flowchart in Figure 4.4. Once the scanning process is complete and the connection process has started, the main logic for data extraction is visualized.

After initializing the connection, we start the critical component called `gattCallback`. This callback controls a large number of BLE interactions and performs many operations related to data communication. It initiates data communication by reaching the Gatt server of the other device to be connected. In particular, it handles changes in connection states and helps discover services on the connected BLE device.
Figure 4.4: Flowchart for Ble connection and obtaining data
Upon initialization of the `gattCallback`, the system subsequently assesses the status of the initiated connection. This step is crucial in ensuring the integrity of the subsequent communication process. A status confirmation, termed `GATT SUCCESS`, signals a successful connection initiation. This, however, is succeeded by the inspection of the connection’s specific state, captured in the `newState` parameter.

Having established a secure connection, the system undertakes the procedure of service discovery. This phase involves enumerating the services provided by the IMU device, thus facilitating an understanding of the device’s capabilities and functionalities.

Then, the system validates the identity of the connected device by referencing its name. This is a safeguard to ensure intended communication with the correct IMU device. If the device’s name aligns with the predefined device name, the system proceeds to adjust the MTU, optimizing the data transmission process.

When a change in the device’s characteristics is detected, the system receives a byte sequence containing the sensor readings. This raw data goes through an extensive processing step that includes decoding and transforming the readings to obtain the required health data.

After the steps from connection initialization to data processing, it has the extracted data from the BLE device, such as oxygen saturation, and temperature. This data is sent to the View Model for monitoring and storing.

### 4.5 Alert Module

The Alert Module of our health monitoring application hinges on the adept integration of Android’s Kotlin-based Jetpack Compose UI toolkit and several critical Android functionalities, such as Notification Compact Library.
At the outset, the module defines a set of constants representing acceptable physiological thresholds for body temperature, oxygen saturation, and pulse rate. The values, such as Min Normal Temperature and Max Normal Temperature, serve as vital sign reference ranges against which real-time readings are compared to trigger alerts.

Once connected, the UI displays data from three devices: the oximeter, the thermometer and the IMU. In the event of a lack of data or a connection issue, pertinent messages guide the user. Additionally, the PostureGauge composable offers a visual representation of a user’s posture based on the Euler angles in degrees, with distinct visual cues for varying posture quality.

Critical to this module is its capability to alert users based on data deviations from acceptable ranges. The custom main function showAlertIfValueNotInRange() encapsulates this logic. They compare real-time data against predefined constants and, if required, invoke the custom showAlarmNotification function. The logic for this function is shown in Figure 4.5. Furthermore, to avoid continuous alerts, a delay mechanism has been incorporated to provide a delay between successive notifications.

The showAlarmNotification function encapsulates the notification logic. Utilizing the Android NotificationManager, it creates channels for distinct alert types, establishes vibration patterns, and leverages custom audio notifications. Notably, if the Android version is Oreo or higher, the function establishes notification channels with specified audio attributes.

For posture analysis, the module uses a orientation data based on the euler angle. Depending on the torso angle, different actions are taken. If the user maintains a certain posture beyond a given time threshold, an alert is dispatched, guiding them to adjust their posture.
4.6 Data Storage

The system is designed to efficiently archive sensor readings, leveraging the Android platform’s capability to save data directly to the device’s local storage. When data is received from the `BLEReceiveManager`, it’s processed and saved as structured text files on the phone’s local storage.

The `DirectConnectionViewModel` is at the core of the data handling process. It processes the sensor readings and saves them to the local storage. Through Kotlin’s Coroutines, these operations are efficiently managed in the background, ensuring the main user interface remains responsive and uninterrupted.
In essence, the approach to data storage is streamlined and focused. It ensures that sensor readings are safely archived on the device, allowing for easy retrieval and sharing while not burdening the primary functionalities of the application.
The principal objective of this study is to devise a health monitoring system with an alert mechanism for farm workers. To demonstrate that this thesis work can serve as exploratory research aiming to enhance the wellbeing of agricultural. The proposed system underwent testing by an individual performing basic motions, simulating the agricultural work conditions.

5.1 Experimental Setup and Procedure

As detailed in Chapter 3, three wearable sensors were used: an IMU, an oximeter, and an thermometer. The IMU data rate can be changed using the manufacturer’s software, and its data rate set at 10 Hz. The data rates for the thermometer and oximeter come preconfigured. However, the data are displayed on UI with the 2Hz data rate.

Sensor reading verification is necessary to check our raw data extraction method is implemented without an issue. The following method are used to verify sensor readings:

The oximeter features an LED display that showcases heart rate and oxygen saturation levels. By comparing the extracted raw data with the LED readings on the device, we can validate its accuracy. A precise match indicates correct data extraction.

We lack the convenience of an LED display as on an oximeter to verify thermometer readings. A workable approach is to use an additional thermometer and compare the two thermometer readings. Although an exact match is not expected, only minor differences
should emerge between the two reading groups. The readings from the additional wearable thermometer are obtained using an application specifically designed for this sensor, and we treat this as the reference value to compare with the actual thermometer data that our application reads.

To validate the orientation data used for posture tracking, a real-time comparison is made by observing the actual posture of the user and the changeable posture in the user interface.

The necessary subjects, equipment, and software were prepared: One observer, one smartphone camera to record experiment (iPhone 14 Pro; Apple, USA), two Android smartphones to validate the app in different hardware (10L; TCL, China; Galaxy S22; Samsung, South Korea), two wearable temperature sensors for body temperature verification (Walnut Thermal; Kickstarter, USA; Koogeek Wearable Smart Baby Thermometer; Koogeek, China), one pulse tip oximeter to measure oxygen saturation (Finger Tip Pulse Oximeter; Health Tree, China), one inertial measurement unit to measure body orientation (LPMS-B2; LP-Research, Japan), one laptop to download the application to smartphones (S510UN; Lenovo, China) running Android Studio, WiFi.

To assess our app’s functionality, we undertook the following experimental preparation and procedure:

**Environment & Setup:** The experiment was performed in a garden during summer, emulating the typical conditions faced by agricultural workers. The IMU was strategically secured to a foam white plate, ensuring alignment of the local x-axes of both systems, as depicted in Figure 5.1a. Preparing the user involved fitting the marker plate, with the affixed IMU, to their upper middle back using a belt, guaranteeing the IMU’s local y-axis pointed towards the user’s head. The thermometers adhered to the chest following their user manuals’ guidelines, and the oximeter was comfortably attached to a finger (Refer to Figure 5.1b.).
Procedure:

1. The participant adorns the belt, thermometers, and oximeter.

2. Our application, operational on Android Studio, is launched on a TCL 10L Android smartphone via the Android Debug Bridge, a command-line tool facilitating direct communication and operations between a computer and an Android device.

3. With all devices activated, they commence their respective advertising processes.

4. Initiating the app involves selecting an image of a farmworker on its start screen, triggering the establishment of all Bluetooth connections and subsequent data monitoring.

5. The accuracy of sensor readings is ascertained using the methodologies discussed earlier.

6. The participant undertakes actions like increasing heart rate or bending the torso, to validate the alert feature.

7. The alert mechanism’s efficacy—spanning vibration, sound, and pop-up notifications—is evaluated.

8. Device reconnection reliability is gauged by either switching off a device or taking it out of the connection range.

9. The phone’s local storage is inspected to ensure accurate data storage.

10. The entire procedure is replicated on another Android smartphone, specifically the Galaxy S22, to gauge consistency across devices.
5.2 Result - Data Monitoring

The implementation of the Android application was a crucial stage in this research, primarily focusing on enabling simultaneous connections to multiple BLE devices and ensuring real-time health monitoring of agricultural workers. The application was designed to connect to three separate BLE devices.

Testing the application involved running it on different Android devices. This approach allowed the evaluation of its performance across a variety of hardware and software configurations. The Logcat, a diagnostic output tool used in the Android operating system was used to monitor the application’s operations in detail, particularly focusing on the simultaneous BLE connections and real-time data streaming.

The application’s real-time data extraction and processing capabilities were extensively tested, focusing on the consistent and accurate retrieval of information from the three se-
lected sensors (IMU, oximeter, and thermometer). The extraction of IMU data was direct and uncomplicated. To retrieve raw data, we adhered to the manufacturer’s guide and concurrently observed the real-time data packets in Logcat. By comparing these data packets with the data format table and examining the first and last bytes, we ensured accurate data packet extraction. This validation gave us confidence that the data monitored on our custom app was accurate. The oximeter and thermometer posed more significant challenges, necessitating careful byte array conversions as explained in Chapter 3. Despite these complexities, the application demonstrated consistent performance in extracting and displaying real-time data from all sensors. During the experiment, temperature and oximeter readings were recorded every 10 seconds and subsequently cross-verified. The oximeter readings were carefully analyzed and found to match exactly with its LED readings, thereby confirming the accuracy of the data extraction process for this device.

Table presents the oximeter data recordings and verification, providing a detailed overview of the experiment’s findings. The tables illustrate the successful extraction and match of real-time data, from the oximeter. Every 10 seconds, 15 samples were recorded to verify the oximeter data.

As for the thermometer, it was compared with another smart wearable thermometer (Koogeck). While an exact match between these two devices was not anticipated since they are from different manufacturers, the differences observed were minimal and within an acceptable range. Every 10 seconds, 15 samples were recorded to verify the thermometer data. This process allowed for a thorough validation of the data extraction techniques and the consistency of the readings across different devices.
<table>
<thead>
<tr>
<th>Time (s)</th>
<th>App Readings</th>
<th>Device (LED) Readings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Oxygen</td>
<td>Pulse</td>
</tr>
<tr>
<td></td>
<td>Saturation (%)</td>
<td>Rate (bpm)</td>
</tr>
<tr>
<td>0</td>
<td>95</td>
<td>85</td>
</tr>
<tr>
<td>10</td>
<td>96</td>
<td>86</td>
</tr>
<tr>
<td>20</td>
<td>97</td>
<td>87</td>
</tr>
<tr>
<td>30</td>
<td>95</td>
<td>88</td>
</tr>
<tr>
<td>40</td>
<td>96</td>
<td>88</td>
</tr>
<tr>
<td>50</td>
<td>97</td>
<td>87</td>
</tr>
<tr>
<td>60</td>
<td>95</td>
<td>86</td>
</tr>
<tr>
<td>70</td>
<td>96</td>
<td>85</td>
</tr>
<tr>
<td>80</td>
<td>97</td>
<td>85</td>
</tr>
<tr>
<td>90</td>
<td>95</td>
<td>86</td>
</tr>
<tr>
<td>100</td>
<td>96</td>
<td>87</td>
</tr>
<tr>
<td>110</td>
<td>97</td>
<td>88</td>
</tr>
<tr>
<td>120</td>
<td>95</td>
<td>89</td>
</tr>
<tr>
<td>130</td>
<td>96</td>
<td>90</td>
</tr>
<tr>
<td>140</td>
<td>97</td>
<td>90</td>
</tr>
</tbody>
</table>

Table provides a detailed overview of the experiment findings by presenting the temperature data records and validation. The tables highlight the minimal discrepancies in temperature readings between the Walnut and Koogeek thermometers, demonstrating the successful extraction of real-time data.
Table 5.2: Temperature data: Walnut and Koogeek sensors readings with error calculation

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Walnut Temperature (F)</th>
<th>Koogeek Temperature (F)</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>97.52</td>
<td>97.7</td>
<td>0.18</td>
</tr>
<tr>
<td>10</td>
<td>97.52</td>
<td>97.7</td>
<td>0.18</td>
</tr>
<tr>
<td>20</td>
<td>97.52</td>
<td>97.7</td>
<td>0.18</td>
</tr>
<tr>
<td>30</td>
<td>97.52</td>
<td>97.7</td>
<td>0.18</td>
</tr>
<tr>
<td>40</td>
<td>97.7</td>
<td>97.7</td>
<td>0.00</td>
</tr>
<tr>
<td>50</td>
<td>97.7</td>
<td>97.7</td>
<td>0.00</td>
</tr>
<tr>
<td>60</td>
<td>97.7</td>
<td>97.7</td>
<td>0.00</td>
</tr>
<tr>
<td>70</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>80</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>90</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>100</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>110</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>120</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>130</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
<tr>
<td>140</td>
<td>97.7</td>
<td>97.88</td>
<td>0.18</td>
</tr>
</tbody>
</table>

The UI design was tested to ensure that it could be used to monitor data effectively. Upon starting the application, the user is greeted with a clickable image that, when pressed, initiates BLE connections with the devices as shown in Figure 5.2a. These connections are established in the background one after another to ensure a smooth user experience.

Once all the connections are established, the UI displays data from all three sensors. As can be seen in Figure 5.2b the numerical values of oxygen saturation, heart rate, and body temperature are presented clearly, aided by both text and images to help the user distinguish between each health metric. The interface also presents a gauge that visualizes the readings from the IMU. This gauge is central to the interface, with an image in its center representing the current posture of farm workers. This visual representation allows users to understand the data more intuitively, making it easier for them to recognize potential health
risks and take necessary actions. Overall, the application’s UI emphasizes user-friendliness and real-time data representation to ensure effective monitoring of agricultural workers.

![Image](a) The developed app’s home screen with a clickable image for initiating BLE connections. b) App’s data display with established BLE connections and data monitoring.

Figure 5.2: a) The developed app’s home screen with a clickable image for initiating BLE connections. b) App’s data display with established BLE connections and data monitoring.

### 5.3 Result - Alert Mechanism and Data Storage

The alert mechanism is a critical feature of the developed app, providing real-time notifications for agricultural workers whenever any significant deviations from normal health parameters or posture angles occur. Particularly for posture angles, the alert system considers both the angular value and duration for which the posture is maintained, adding a layer of sensitivity, and ensuring comprehensive coverage of potential health risks. Unique audio
cues for different alerts improve usability in diverse work environments and a delay function is incorporated to prevent alert fatigue, ensuring a minimum 5-second gap between similar alerts.

During testing, while some alerts may not be observable in a healthy individual, the alerts related to the IMU and heart rate data were to confirmed that the app functions as expected. In a separate test, alert features for oxygen saturation and temperature were also evaluated. To test these alerts under regular, normal settings, the preset thresholds for oxygen saturation and temperature were adjusted. All alerts, including sound, vibration, and pop-up notifications, functioned fully and as anticipated. Figure 5.4 displays screenshots captured when alerts were triggered for different health metrics.

In order to evaluate the performance of the IMU alert mechanism following the alert mechanism design explained in Chapter 3.5, the posture orientation was initially analyzed. Firstly, the user maintained their posture within the range of 0-20 degrees, which is considered the safest range for posture. The user interface effectively reflected this position, and as anticipated, no warning was activated. Subsequently, the user proceeded to bend his torso at an estimated angle of 50 degrees and maintained this position. Approximately one minute later, an alarm was activated, prompting the user to modify his posture. Finally, as shown in Figure 5.3 when the user inclines the trunk beyond an angle of 60 degrees, an alarm is triggered immediately upon beyond this threshold. The manufacturer’s software was used to verify the trunk angle. It was confirmed that a straight up position corresponds to 0 degrees, while inclinations of 20, 50, and 60 degrees have been determined. For example, in order to figure out the inclination of the posture at an angle of 50 degrees, the user proceeded to bend his torso by 50 degrees, relying on the readings provided by the manufacturer’s software utilizing Euler angles. Subsequently, the user held their torso in the previous position, and established a connection between the sensor and our application. As anticipated, an alarm
was generated after around one minute. The same logical approach was utilized for the other trunk angles, and the alarm mechanism functioned as intended.

![Figure 5.3: Torso inclination exceeds the 60-degree threshold.](image)

The oximeter is capable of measuring heart rate and oxygen saturation levels. Additionally, it prompts the user to engage in physical exercise in order to elevate their heart rate. The predetermined upper limit for heart rate was set at 120 bpm, and whenever the user’s heart rate is above this number, an alarm is triggered for the user. The initial preset setting for oximeter saturation was set at 94. During the monitoring process, the user’s oxygen saturation level fell below this predetermined threshold, resulting in the activation of an alarm for low oxygen saturation. During the testing process, it was seen that the user’s body temperature measurement exceeded the critical upper limit of 100°F. As a result, an
alert was generated to notify the user and advise them to take necessary measures, such as resting and staying hydrated by drinking water.

The predetermined low thresholds for body temperature and heart rate were established at 97°F and 60 bpm, respectively. Nevertheless, it is challenging to identify values below these thresholds in a physiologically healthy individual. In order to activate the alarm mechanism within the application, these thresholds have been adjusted. For instance, a non-critical temperature value of 98°F has been designated as the threshold for low criticality. Similarly, a heart rate of 80 beats per minute has been established as the critical low value. The occurrence of data falling below the specified thresholds prompts the activation of the alarm system, followed by the execution of comprehensive alert testing procedures.

Lastly, the application data storage features were tested. Upon running the app it starts to store data automatically to the phone’s local storage. Table 5.3 presents the data recorded between 10 to 12 seconds after the BLE Connection was established. With the IMU having the fastest data rate, the trunk inclination angle updates for each line of the table, while other health metrics update at a slower rate. As a result, data storage tends to align with the fastest data rate, which is that of the IMU. Furthermore, the observations indicate that approximately 1 kB of data is stored in local storage every second. In practical terms, recording 8 hours (a typical worker shift) of continuous data will consume around 30 MB of space. This amount does not constitute a significant data payload, and nearly any phone’s storage capacity can accommodate this requirement.
Figure 5.4:  a) O₂ saturation drops below 94%, pop-up notification. b) Body temp > 100°F, pop-up notification. c) Heart rate > 120, pop-up notification. d) Trunk angle > 60°, pop-up notification. e) Trunk angle 20°- 60°, alert then pop-up notification.

The integration of real-time alerts and data storage facilitates a comprehensive health monitoring system. The immediate identification and communication of potential health risks to the user are ensured by the alert mechanism, while the data storage maintains a record of these events for future analysis and preventative measures. This combination pro-
Table 5.3: Health Data Records: Retrieved from mobile device’s local storage.

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Body Temperature (F)</th>
<th>Oxygen Saturation (%)</th>
<th>Pulse Rate (bpm)</th>
<th>Trunk Inclination (rad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.043</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8018</td>
</tr>
<tr>
<td>10.175</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8020</td>
</tr>
<tr>
<td>10.268</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8414</td>
</tr>
<tr>
<td>10.354</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8435</td>
</tr>
<tr>
<td>10.446</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8340</td>
</tr>
<tr>
<td>10.583</td>
<td>97.52</td>
<td>97</td>
<td>94</td>
<td>0.8140</td>
</tr>
<tr>
<td>10.661</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.8068</td>
</tr>
<tr>
<td>10.759</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.8151</td>
</tr>
<tr>
<td>10.849</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.8379</td>
</tr>
<tr>
<td>10.941</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.8757</td>
</tr>
<tr>
<td>11.124</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.8967</td>
</tr>
<tr>
<td>11.155</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.9458</td>
</tr>
<tr>
<td>11.257</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.9540</td>
</tr>
<tr>
<td>11.331</td>
<td>97.52</td>
<td>96</td>
<td>95</td>
<td>0.9423</td>
</tr>
</tbody>
</table>

Provides immediate, as well as long-term insights into the health of agricultural workers, which can be instrumental for in-depth analysis, diagnosis, and formulation of intervention strategies by occupational health professionals. The integrated alert and data storage systems underscore the effectiveness of the application as a real-time health and posture monitoring tool.
CHAPTER 6
DISCUSSIONS AND CONCLUSIONS

6.1 Discussions

The findings from this research indicate the feasibility and effectiveness of using an Android smartphone application with wearable biosensors for real-time health and posture monitoring for agricultural workers. However, the selection of suitable biosensors proved to be a challenging task, particularly with commercially available sensors. These sensors, while enabling access to raw data, often lack comprehensive documentation. This requires a rigorous process of testing using third-party applications to ascertain if the raw data from these sensors could be accessed and processed. It should be also noted that a ring-type oximeters would be more practical for agricultural workers, given the nature of their work. A ring-type oximeter was not used in this study, because there was no ring-type oximeter commercially available that provides open access to the raw data.

While the IMU and oximeter maintained a continuous BLE connection, the skin thermometer’s BLE connection was observed to drop connections periodically. This issue was identified as a hardware-related setting per-configured by the manufacturer, and it was not allowed to change the settings. The same behavior of periodic connection drops was observed with another BLE thermometer from a different manufacturers, such as Koogeek. It is presumed that this behavior may be associated with a power-saving strategy implemented by the manufacturers. To address this issue, a re-connection mechanism was developed within
the developed app. Whenever a device’s connection drops, re-connection procedures are now initiated, ensuring continuous monitoring.

The connection instability with the thermometer led to inaccurate readings from both the IMU and the oximeter. Upon further analysis, it became evident that the shared buffer used for these devices was causing interference with sensor readings. Specifically, when the thermometer experienced connection drops, the subsequent reconnection process was either prolonged or stalled. It, in turn, hindered other ongoing operations, including the processing of the IMU data. To address this challenge, we allocated distinct resource files for storing the data of each sensor, backed by separate buffers. Leveraging Kotlin coroutines, we then streamed data to these resource files asynchronously, ensuring that the readings from one sensor wouldn’t interfere with another. This strategy guarantees continuous and precise monitoring.

Integrating an additional sensor into the current system is a streamlined process. Initially, the new sensor must be added to the connection queue. This ensures that once established connections with existing devices are secured, the new sensor can connect smoothly. For each new device, a distinct GattCallback object needs to be instantiated to handle its unique data exchanges. To extract the raw data from the newly added device, it’s crucial to reference its specific data format table.

In essence, adapting our system to accommodate multiple BLE connections can be summarized down to two main steps. First, extend the architecture, which is originally designed for a single connection, by adding analogous files for each new device. Second, effectively manage potential issues arising from multiple connections and multi-threading scenarios.

The developed Android application demonstrated a powerful and user-friendly interface that can process and display data while effectively managing multiple BLE connections. The app also successfully demonstrated that it could alert users when the data indicates potential
issues. Despite facing the instability of temperature sensor connections, the application was able to handle such situations and ensure the smooth functioning of the system.

6.2 Conclusions

The starting point of this thesis was to develop a system to improve the health and safety of agricultural workers. A comprehensive literature review was first conducted to understand existing practices. To develop such a system, it was identified that various health metrics need to be monitored in real time. Therefore, the goal of this work was set to develop a smartphone application that can connect to multiple sensors at the same time and also to develop an open framework where new sensors can be easily connected to increase the value of our application.

The development of a health monitoring and alert system for agricultural workers. It involved identifying suitable biosensors, mastering the intricacies of BLE technology in Android, building on our Android development expertise, creating an asynchronous application and improving troubleshooting skills.

Extensive research was conducted to find suitable biosensors that provide open access to their raw data. Using a third-party application helped categorize these sensors based on their open access data characteristics. We then chose the best system and programming environment to develop the proposed application. A smartphone application was chosen as the best option to develop such a system due to its prevalence. The Android operating system was then selected, followed by Android Studio and Jetpack compose as they are regarded as the most suitable IDE and tools for developing asynchronous programming and easy and efficient user interface design.
The hardware and software selections were followed by the application development process was initiated. First, an MVVM architectural design pattern was chosen to clarify the separation between user interface and logic to improve the sustainability and efficiency of the app. Next, the presentation layer of our app was created, which includes a start screen that can interact with the user to initiate Bluetooth scanning and a screen to monitor health metrics and alert notifications. This layer also includes a navigation system to switch between these screens.

At the core of this effort, we developed the `BLEManager` class, which oversees all BLE operations and data processing. This class forms the foundation of our application by seamlessly establishing BLE connections with three different sensors to extract health data such as oxygen saturation and heart rate from the oximeter, temperature readings from the thermometer, and orientation details from the IMU. Once this data is acquired via BLE, it is passed to the `ViewModel` via `mutableSharedFlow` class. By leveraging `MutableStateOf()` function, the data is then displayed by the user interface.

The logic of the alert mechanism is based on thresholds set by health authorities. Basically, an abnormal value outside the threshold range would trigger the alert mechanism. The Notification Compact Library was used to create the alert mechanism and user alert features such as sound, vibration, pop-up notification were added. This ensured a timely response alert. A data storage mechanism was then developed for further analysis, with data stored asynchronously in the phone storage.

Testing this app involved validating the monitoring data, checking the alert feature and data storage. Once app launches the start screen and a button on the screen is displayed. Then clicking on the button starts the BLE scanning process and then the three sensor connections are observed one by one. The LED display of the oximeter and the displayed data are compared to verify the data tracking. An additional thermometer was used for thermometer verification and the body temperature reading was verified. Finally, the posture
tilt for the IMU was observed and verified with the posture tracking in the app. To test the alert mechanism, the user performed a series of actions that could change the health conditions and trigger the alert mechanism. Finally, the phone local storage observed to verify the data storage functionality.

To this end, we have successfully built an Android app specifically designed for real-time health and posture monitoring of agricultural workers, leveraging the capabilities of multiple BLE connections to effortlessly connect with various sensors and collect key health metrics. Our app’s unique alert system, equipped with different sound, vibration and pop-up notifications, ensures that users are immediately notified of critical health issues. This innovation is not limited to agriculture; its potential extends to countless professions that require instant health and posture monitoring. There is also a wide horizon of improvements, from the embedding of additional health metrics to advanced data analytics. In conclusion, this thesis contributes to advance in digital assistants for occupational health surveillance, standing as a testament to the important role of technology in supporting worker safety.

6.3 Future Works

The following potential enhancements are suggested as future work.

- Extended Sensor Integration: New sensors can be integrated into the application to monitor additional health metrics. These could include hydration levels, environmental factors like UV exposure, or even stress levels.

- Data Analytics Module: Introducing advanced data analytics can offer users predictive insights into potential health risks based on historical data. This can provide proactive measures rather than just reactive alerts.
• Cloud Integration: Storing data on the cloud can offer more substantial storage solutions and easier access for occupational health professionals to analyze and offer timely interventions.

• Collaboration with Health Professionals: Partnering with occupational health experts can refine the thresholds and alerts, ensuring they are in line with the latest research and best practices.
REFERENCES


[14] K. Umapathy, S. Chandramohan, D. Muthukumaran, M. Sivakumar, M. Vinoth, and S. Selvakumar, “Edge and android application based health monitor,” in *2023 5th Inter-


APPENDIX

BLE MANAGER CLASS
Below is the BLE Manager class integrated into the appendix.

```kotlin
package com.example.bledirectconnection.data.ble
import android.annotation.SuppressLint
import android.annotation.SuppressLint
import android.bluetooth.*
import android.bluetooth.le.ScanCallback
import android.bluetooth.le.ScanResult
import android.bluetooth.le.ScanSettings
import android.content.ContentValues.TAG
import android.content.Context
import android.os.Handler
import android.os.Looper
import android.os.Log
import com.example.bledirectconnection.data.*
import com.example.bledirectconnection.util.OximeterResource
import com.example.bledirectconnection.util.Resource
import com.example.bledirectconnection.util.WalnutResource
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.util.*
import javax.inject.Inject
@Suppress("DEPRECATION")
@SuppressLint("MissingPermission")
class DirectConnectionBLEReceiveManager @Inject constructor(
    private val bluetoothAdapter: BluetoothAdapter,
    @ApplicationContext private val context: Context
) : DirectConnectionReceiveManager {

    private val DEVICE_NAME = "LPMSB2-6F36F6"
    private val IMU_SERVICE_UUID = "0000180f-0000-1000-8000-00805f9b34fb"
    private val IMU_CHARACTERISTICS_UUID = "00002a19-0000-1000-8000-00805f9b34fb"
    private val OXIMETER_DEVICE_NAME = "OXIMETER"
    private val OXIMETER_SERVICE_UUID = "0000ffe0-0000-1000-8000-00805f9b34fb"
    private val OXIMETER_CHARACTERISTICS_UUID = "0000ffe1-0000-1000-8000-00805f9b34fb"
    private val WALNUT_DEVICE_NAME = "Walnut"
    private val WALNUT_SERVICE_UUID = "0000fff0-0000-1000-8000-00805f9b34fb"
    private val WALNUT_CHARACTERISTIC_UUID = "0000fff1-0000-1000-8000-00805f9b34fb"

    override val data:MutableSharedFlow<Resource<ImuData>> = MutableSharedFlow()
    override val oximeterData:MutableSharedFlow<OximeterResource<OximeterData>> = MutableSharedFlow()
```
override val walnutData: MutableSharedFlow<WalnutResource<WalnutData>> = 
    MutableSharedFlow()

private val bleScanner by lazy {
    bluetoothAdapter.bluetoothLeScanner
}

private var gattimu: BluetoothGatt? = null
private var gattoxi: BluetoothGatt? = null
private var gattwalnut: BluetoothGatt? = null
private var isScanning = false // tracks ble scanner
private val coroutineScope = CoroutineScope(Dispatchers.Default)
private val handler = Handler(Looper.getMainLooper())
private val scanResults = mutableListOf<ScanResult>()
private val devicesToConnect = mutableListOf<ScanResult>()
private val scanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)
        val deviceName = result.device.name ?: return
        if (deviceName == DEVICE_NAME || deviceName == OXIMETER_DEVICE_NAME || deviceName == WALNUTDEVICE_NAME) {
            if (!scanResults.any { it.device.address == result.device.address }) {
                scanResults.add(result)
                Log.d("BLEReceiveManager", "Discovered \$\{result.device.name\}, Address: \$\{result.device.address\}"
            }
        }
        if (scanResults.size == 3) { // All devices found
            isScanning = false
            bleScanner.stopScan(this)
            Log.d("BLEReceiveManager", "All devices discovered")
            // Copy scanResults to devicesToConnect
            devicesToConnect.addAll(scanResults)
            // Start connection with the first device
            connectNextDevice()
        }
    }
}

private val connectionAttempts = mutableMapOf<String, Int>()
private var MAXIMUM_CONNECTION_ATTEMPTS = 1000
private val gattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                coroutineScope.launch {
                    data.emit(Resource.Loading(message = "Discovering IMU Services...")
                    connectNextDevice()
                }
                gatt.discoverServices()
            }
        }
    }
}

private fun connectNextDevice() {
    this@DirectConnectionBLEReceiveManager.gattimu = gatt
} else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
    coroutineScope.launch {
        data.emit(
            Resource.Success(
                data = ImuData(
                    0f,
                    0f,
                    0f,
                    0f,
                    0f,
                    0f,
                    0f,
                    0f,
                    ConnectionState.Disconnected
                )
            )
        )
    }
    gatt.close()
}
else {
    gatt.close()
    val deviceAddress = gatt.device.address
    val currentAttempt = connectionAttempts[deviceAddress] ?: 0
    connectionAttempts[deviceAddress] = currentAttempt + 1
    if (currentAttempt <= MAXIMUM_CONNECTION_ATTEMPTS) {
        reconnectDevice(deviceAddress)
    } else {
        coroutineScope.launch {
            data.emit(Resource.Error(errorMessage = "Could not connect to $deviceAddress"))
        }
    }
}
override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
    with(gatt) {
        val deviceName = gatt.device.name
        when (deviceName) {
            DEVICE_NAME -> {
                coroutineScope.launch {
                    data.emit(Resource.Loading(message = "Adjusting IMU MTU space..."))
                }
                requestMtu(55)
            }
        }
    }
}
override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
    val characteristic = findCharacteristics(IMU_SERVICE_UUID,
        IMU_CHARACTERISTICS_UUID, gattimu)
    if (characteristic == null) {
        coroutineScope.launch {
            data.emit(Resource.Error(errorMessage = "Could not find imu publisher"))
        }
        return
    }
    enableNotifications(characteristic, gattimu)
}//@Suppress("DEPRECATION")

var buffer = mutableListOf<Byte>()
private var iterationCounter = 0
@Deprecated("Deprecated in Java")
override fun onCharacteristicChanged( gatt: BluetoothGatt,
    characteristic: BluetoothGattCharacteristic
) {
    with(characteristic) {
        when (uuid) {
            UUID.fromString(IMU_CHARACTERISTICS_UUID) -> {
                @Suppress("DEPRECATION")
                var value = getValue()
                synchronized(buffer) {
                    buffer.addAll(value.toList())
                    while (buffer.size > 55) {
                        buffer.clear()
                        //buffer.removeAt(0)
                    }
                }
            }
            // Convert the byte array to a readable hexadecimal string and log it
            val startingSequence = listOf(0x3A, 0x01, 0x00, 0x09, 0x00).map { it.toByte() }
            val endingSequence = listOf(0x0D, 0x0A).map { it.toByte() }
            // Find the data packet and its end index
            synchronized(buffer) {
                var (dataPacket, endIndex) = findDataPacket( buffer,
                    startingSequence, endingSequence
                )
        }
    }
}
while (dataPacket != null && endIndex >= 0 && dataPacket.size > 50) {
    iterationCounter++

    // Process the data packet
    Log.d("ProcessedByteArray", "Processed\_byte\_array
    
    " + dataPacket.joinToString(" ") +
    
    String.format("%02X", byte)
    
    )
    Log.d("Buffer", "Buffer\_array\_array\_array\_array\_array:\$
    
    buffer.joinToString(" ") +
    
    String.format("%02X", byte)
    
    )

    val q0 =
        ByteBuffer.wrap(dataPacket, 11, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val q1 =
        ByteBuffer.wrap(dataPacket, 15, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val q2 =
        ByteBuffer.wrap(dataPacket, 19, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val q3 =
        ByteBuffer.wrap(dataPacket, 23, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val euAngX =
        ByteBuffer.wrap(dataPacket, 27, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val euAngY =
        ByteBuffer.wrap(dataPacket, 31, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val euAngZ =
        ByteBuffer.wrap(dataPacket, 35, 4)
        .order(ByteOrder.LITTLE_ENDIAN)
        .float
    val Pressure =
        ByteBuffer.wrap(dataPacket, 39, 4)
val Altitude = ByteBuffer.wrap(dataPacket, 43, 4)
    .order(ByteOrder.LITTLE_ENDIAN)
    .float
val temp = ByteBuffer.wrap(dataPacket, 47, 4)
    .order(ByteOrder.LITTLE_ENDIAN)
    .float - 10f

val connectionResult = ImuData(
    q0,
    q1,
    q2,
    q3,
    euAngX,
    euAngY,
    euAngZ,
    Pressure,
    Altitude,
    temp,
    ConnectionState.Connected
)
coroutineScope.launch {
    data.emit(
        Resource.Success(data = connectionResult)
    )
}

if (iterationCounter == 10) {
    buffer.clear()
    iterationCounter = 0
} else {
    synchronized(buffer) {
        println("buffer size: \$\{buffer.size\}, endIndex: \$\{endIndex\}")
        if (endIndex <= buffer.size) {
            buffer =
                buffer.subList(endIndex, buffer.size)
            .toMutableList()
        } else {
            // Handle error or invalid situation
            println("Error: endIndex ($endIndex) is greater than or equal to the buffer size ($\{buffer.size\})")
        }
    }
}
// Try to find the next data packet
val (nextDataPacket, nextEndIndex) = findDataPacket(
    buffer,
    startingSequence,
    endingSequence
) 
dataPacket = nextDataPacket
d endIndex = nextEndIndex

private val oximeterGattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                coroutineScope.launch {
                    oximeterData.emit(OximeterResource.Loading(message = "Discovering Oximeter Services..."))
                    connectNextDevice()
                    gatt.discoverServices()
                }
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                coroutineScope.launch {
                    oximeterData.emit(OximeterResource.Success(
                        oximeterdata = OximeterData(0, 0, 0f, ConnectionState.Disconnected))
                    )
                    gattoxi?.close()
                }
            } else {
                gattoxi?.close()
                val deviceAddress = gatt.device.address
                val currentAttempt = connectionAttempts[deviceAddress] ?: 0
                connectionAttempts[deviceAddress] = currentAttempt + 1
                if (currentAttempt <= MAXIMUM_CONNECTION_ATTEMPTS) {
                    reconnectDevice(deviceAddress)
                } else {
                    coroutineScope.launch {
                        data.emit(Resource.Error(errorMessage = "Could not connect to device $deviceAddress"))
                    }
                }
            }
        } else {
            gattoxi?.close()
            val deviceAddress = gatt.device.address
            val currentAttempt = connectionAttempts[deviceAddress] ?: 0
            connectionAttempts[deviceAddress] = currentAttempt + 1
            if (currentAttempt <= MAXIMUM_CONNECTION_ATTEMPTS) {
                reconnectDevice(deviceAddress)
            } else {
                coroutineScope.launch {
                    data.emit(Resource.Error(errorMessage = "Could not connect to device $deviceAddress"))
                }
            }
            // Handle unsuccessful connection attempts if necessary
override fun onServicesDiscovered(gatt: BluetoothGatt?, status: Int) {
    super.onServicesDiscovered(gatt, status)
    Log.d("BLEReceiveManager", "onServicesDiscovered")
    if (status == BluetoothGatt.GATT_SUCCESS) {
        gatt?.let { gatt ->
            this@DirectConnectionBLEReceiveManager.gattoxi = gatt
            val service = gatt.getService(UUID.fromString("OXIMETER_SERVICE_UUID"))

            if (service != null) {
                val characteristic = service.getCharacteristic(UUID.fromString("OXIMETER_CHARACTERISTICS_UUID"))
                enableNotifications(characteristic, gattoxi)
            } else {
                Log.d("BLEReceiveManager", "Oximeter service not found")
            }
        } ?: run {
            Log.d("BLEReceiveManager", "Gatt object is null")
        }
    } else {
        Log.d("BLEReceiveManager", "onServicesDiscovered received: $status")
    }
}

override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
    val characteristic = findCharacteristics(OXIMETER_SERVICE_UUID, OXIMETER_CHARACTERISTICS_UUID, gattoxi)
    if (characteristic == null) {
        coroutineScope.launch {
            data.emit(Resource.Error(errorMessage = "Could not find oximeter publisher"))
        }
        return
    }
    enableNotifications(characteristic, gattoxi)
}

// override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
//    // If the Oximeter device requires an MTU change, implement it here.
//    // }

@Deprecated("Deprecated in Java")
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
}
if (characteristic.uuid == UUID.fromString(OXIMETER_CHARACTERISTICS_UUID)) {
    val receivedData = characteristic.value
    if (receivedData != null) {
        if (receivedData.size > 1) {
            val oximeterReading = parseOximeterData(receivedData)
            oximeterReading?.let {
                coroutineScope.launch {
                    oximeterData.emit(OximeterResource.Success(
                        oximeterData = it))
                }
            } else {
                Log.e(TAG, "Unexpected data length received: \${receivedData.size}"")
            }
        }
    }
}

private val walnutGattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                coroutineScope.launch {
                    walnutData.emit(WalnutResource.Loading(message = "Discovering Walnut Services..."))
                    connectNextDevice()
                }
                gatt.discoverServices()
                this@DirectConnectionBLEReceiveManager.gattwalnut = gatt
                Log.d("BLEReceiveManager", "Walnut Device connected")
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                coroutineScope.launch {
                    walnutData.emit(WalnutResource.Success(
                        walnutData = WalnutData(
                            0f, ConnectionState.Disconnected
                        )))
                }
                gattwalnut?.close()
            } else {
                gattwalnut?.close()
            }
        }
    }
    
    override fun onServicesDiscovered(gatt: BluetoothGatt, services: BluetoothGatt可以更好 understood as a BluetoothGattCallback object that handles the connection state change events. If the connection status is success and the new state is CONNECTED, it launches a coroutine to emit a loading message and initiates the next device connection. It also discovers the services of the connected device. If the state is DISCONNECTED, it emits a success message with the connection state as Disconnected and closes the gattwalnut connection. If the state is any other, it also closes the gattwalnut connection.

```java
private val walnutGattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                coroutineScope.launch {
                    walnutData.emit(WalnutResource.Loading(message = "Discovering Walnut Services..."))
                    connectNextDevice()
                }
                gatt.discoverServices()
                this@DirectConnectionBLEReceiveManager.gattwalnut = gatt
                Log.d("BLEReceiveManager", "Walnut Device connected")
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                coroutineScope.launch {
                    walnutData.emit(WalnutResource.Success(
                        walnutData = WalnutData(
                            0f, ConnectionState.Disconnected
                        )))
                }
                gattwalnut?.close()
            } else {
                gattwalnut?.close()
            }
        }
    }
    
    override fun onServicesDiscovered(gatt: BluetoothGatt, services: BluetoothGattCharacteristic) {
        // Handle services discovered
    }
    
    override fun onCharacteristicRead(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic, value: ByteArray) {
        // Handle characteristic read
    }
    
    override fun onConnectionLost(gatt: BluetoothGatt, cause: Int) {
        // Handle connection lost
    }
    
    override fun onServiceAdded(gatt: BluetoothGatt, service: BluetoothGattService) {
        // Handle service added
    }
    
    override fun onServiceRemoved(gatt: BluetoothGatt, service: BluetoothGattService) {
        // Handle service removed
    }
    
    override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
        // Handle characteristic changed
    }
    
    override fun onDescriptorChanged(gatt: BluetoothGatt, descriptor: BluetoothGattDescriptor, value: ByteArray) {
        // Handle descriptor changed
    }
    
    override fun onDescriptorWrite(gatt: BluetoothGatt, descriptor: BluetoothGattDescriptor, value: ByteArray, success: Boolean) {
        // Handle descriptor write
    }
    
    override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int) {
        // Handle MTU changes
    }
    
    override fun onRssiUpdated(gatt: BluetoothGatt, rssi: Int) {
        // Handle RSSI updates
    }
    
    override fun onGattCallbackError(gatt: BluetoothGatt, error: BluetoothGattError) {
        // Handle GATT callback errors
    }
}
```
val currentAttempt = connectionAttempts[deviceAddress] ?: 0
connectionAttempts[deviceAddress] = currentAttempt + 1
if (currentAttempt <= MAXIMUM_CONNECTION_ATTEMPTS) {
    reconnectDevice(deviceAddress)
} else {
    coroutineScope.launch {
        data.emit(Resource.Error(errorMessage = "Could not connect to device $deviceAddress"))
    }
} // Handle unsuccessful connection attempts if necessary

override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int)
{
    Log.d("BLEReceiveManager", "Walnut Services discovered")
    if (status == BluetoothGatt.GATT_SUCCESS) {
        gatt.services.forEach { service ->
            service.characteristics.forEach { characteristic ->
                if (characteristic.isNotifiable() || characteristic.isIndicatable()) {
                    if (enableNotification(characteristic, gattwalnut)) {
                        Log.d("BLEReceiveManager", "Notification enabled for Walnut characteristic: $characteristic")
                    } else {
                        Log.d("BLEReceiveManager", "Failed to enable notification for characteristic: $characteristic")
                    }
                }
            }
        } // Add a delay after enabling notification
        handler.postDelayed({
            val characteristicToRead =
                gattwalnut?.getService(UUID.fromString("0000fff0-0000-1000-8000-00805f9b34fb"))
                ?.getCharacteristic(UUID.fromString("0000fff1-0000-1000-8000-00805f9b34fb"))
                gattwalnut?.readCharacteristic(characteristicToRead), 500)
        } else {
            Log.e("BLEReceiveManager", "Failed to discover services")
            disconnect()
        }
    } else {
        Log.e("BLEReceiveManager", "Failed to discover services")
        disconnect()
    }
}
override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
    val characteristic =
        findCharacteristics(WALNUT_SERVICE_UUID,
            WALNUT_CHARACTERISTIC_UUID,gattwalnut)
    if (characteristic == null) {
        coroutineScope.launch {
            walnutData.emit(WalnutResource.Error(errorMessage = "Could not find WALNUT publisher"))
        }
        return
    }
    while (enableNotification(characteristic,gattwalnut)) {
        //gatt.readCharacteristic(characteristic)
    }
}

@Deprecated("Deprecated in Java")
override fun onCharacteristicRead(gatt: BluetoothGatt,
    characteristic: BluetoothGattCharacteristic,
    status: Int) {
    // burada kaldık
    //Log.d("BLEReceiveManager", "onCharacteristicRead: Received data")
    gattwalnut?.readCharacteristic(characteristic)
    handleCharacteristicData(characteristic)
}

@Deprecated("Deprecated in Java")
override fun onCharacteristicChanged(gatt: BluetoothGatt,
    characteristic: BluetoothGattCharacteristic) {
    Log.d("BLEReceiveManager", "onCharacteristicChanged: Received Walnut data")
    handleCharacteristicData(characteristic)
    gattwalnut?.readCharacteristic(characteristic)
}

private fun handleCharacteristicData(characteristic: BluetoothGattCharacteristic) {
    with(characteristic) {
        when (uuid) {
            UUID.fromString(WALNUT_CHARACTERISTIC_UUID) -> {
                // Log.i("IncomingData", "Incoming byte array: 
                // JoinToString("", ", prefix = ", postfix = ") { "0x%02X
                // format(it) }")
                val temperature = (180*(36+(value[2]-104)/10f)+3200)/100
                val walntData = WalnutData(
                    temperature,
ConnectionState.Connected
}
coroutineScope.launch {
    walnutData.emit(
        WalnutResource.Success(walnutdata = walntData)
    )
}
else -> Unit
}
}
// override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
// If the Oximeter device requires an MTU change, implement it here.

private fun connectNextDevice() {
    if (devicesToConnect.isEmpty()) {
        return
    }
    val nextDeviceResult = devicesToConnect.removeAt(0)
    when (nextDeviceResult.device.name) {
        DEVICE_NAME -> gattimu = nextDeviceResult.device.connectGatt(
            context, false, gattCallback, BluetoothDevice.TRANSPORT_LE)
        OXIMETER_DEVICE_NAME -> gattool = nextDeviceResult.device.
            connectGatt(context, true, oximeterGattCallback)
        WALNUT_DEVICE_NAME -> gattwalnut = nextDeviceResult.device.
            connectGatt(context, true, walnutGattCallback)
        else -> {
            // Handle unknown device name
        }
    }
}

private fun reconnectDevice(deviceAddress: String) {
    val deviceToReconnect = scanResults.find { it.device.address == deviceAddress }
    deviceToReconnect?.let {
        when (it.device.name) {
            DEVICE_NAME -> gattimu = it.device.connectGatt(context, false,
                gattCallback, BluetoothDevice.TRANSPORT_LE)
            OXIMETER_DEVICE_NAME -> gattool = it.device.connectGatt(
                context, true, oximeterGattCallback)
            WALNUT_DEVICE_NAME -> gattwalnut = it.device.connectGatt(
                context, true, walnutGattCallback)
            else -> {
                // Handle unknown device name
            }
        }
    }
}
private fun parseOximeterData(data: ByteArray): OximeterData? {
    // Check if the first byte is 0xff
    if (data[0] != 0xff.toByte()) {
        return null
    }

    // Extract oxygen saturation and heart rate from the 5th and 6th bytes
    val spo2 = data[4].toInt() and 0xff
    val pulseRate = data[5].toInt() and 0xff

    return OximeterData(
        spo2 = spo2,
        pulseRate = pulseRate,
        pi = 0f, // You can update this value if required
        connectionState = ConnectionState.Connected
    )
}

private fun enableNotifications(characteristic: BluetoothGattCharacteristic, gatt: BluetoothGatt?) {
    val cccdUuid = UUID.fromString(CCCD_DESCRIPTOR_UUID)
    val payload = when {
        characteristic.isIndicatable() -> BluetoothGattDescriptor.ENABLE_INDICATION_VALUE
        characteristic.isNotifiable() -> BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
        else -> {
            Log.d("BLEReceiveManager", "Characteristic is neither indicatable nor notifiable")
            return
        }
    }

    characteristic.getDescriptor(cccdUuid)?.let { cccdDescriptor ->
        gatt?.let {
            if (it.setCharacteristicNotification(characteristic, true) == false) {
                Log.d("BLEReceiveManager", "setCharacteristicNotification failed")
                return
            }
            writeDescription(cccdDescriptor, payload, it)
        } ?: run {
            Log.d("BLEReceiveManager", "Not connected to a BLE device!")
        } ?: Log.d("BLEReceiveManager", "CCCD descriptor not found")
    }
}

private fun enableNotification(characteristic: BluetoothGattCharacteristic, gatt: BluetoothGatt?): Boolean {
    val cccdUuid = UUID.fromString(CCCD_DESCRIPTOR_UUID)
val indicatePayload = BluetoothGattDescriptor.
    ENABLE_INDICATION_VALUE
val notifyPayload = BluetoothGattDescriptor.
    ENABLE_NOTIFICATION_VALUE

characteristic.getDescriptor(cccdUuid)?.let { cccdDescriptor ->
    gatt?.let {
        if (it.setCharacteristicNotification(characteristic, true) ==
          false) {
            Log.d("BLEReceiveManager", "setCharacteristicNotification" failed)
            return false
        }
        // Attempt to enable both indications and notifications
        writeDescription(cccdDescriptor, indicatePayload, it)
        writeDescription(cccdDescriptor, notifyPayload, it)
        return true
    } ?: return false
}
Log.d("BLEReceiveManager", "CCCD descriptor not found")
return false

private fun findCharacteristics(serviceUUID: String, characteristicsUUID:
    String, gatt: BluetoothGatt?): BluetoothGattCharacteristic? {
    return gatt?.services?.find { service ->
      service.uuid == UUID.fromString(serviceUUID)
    }?.characteristics?.find { characteristic ->
      characteristic.uuid == UUID.fromString(characteristicsUUID)
    }
}

private fun writeDescription(descriptor: BluetoothGattDescriptor,
    payload: ByteArray, gatt: BluetoothGatt) {
    descriptor.value = payload
gatt.writeDescriptor(descriptor)
}

override fun startReceiving() {
    coroutineScope.launch {
        data.emit(Resource.Loading(message = "Scanning Ble devices..."))
    } isScanning = true
    val settings = ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
    .build()
    bleScanner.startScan(null, settings, scanCallback)
}

override fun reconnect() {
    gattimu?.connect()
}

override fun disconnect() {
    gattimu?.disconnect()
gattoxi?.disconnect()
gattwalnut?.disconnect()
}
override fun closeConnection() {
    bleScanner.stopScan(scanCallback)
    val characteristic = findCharacteristics(IMU_SERVICE_UUID,
        IMU_CHARACTERISTICS_UUID,gattimu)
    val characteristicOximeter = findCharacteristics(
        OXIMETER_SERVICE_UUID, OXIMETER_CHARACTERISTICS_UUID, gattoxi)
    val characteristicWalnut = findCharacteristics(WALNUT_SERVICE_UUID,
        WALNUT_CHARACTERISTIC_UUID, gattwalnut)
    if(characteristic != null){
        disconnectCharacteristic(characteristic, gattimu)
    }
    if(characteristicOximeter != null){
        disconnectCharacteristic(characteristicOximeter, gattoxi)
    }
    if(characteristicWalnut != null){
        disconnectCharacteristic(characteristicWalnut, gattwalnut)
    }
    gattimu?.close()
    gattoxi?.close()
    gattwalnut?.close()
}
private fun disconnectCharacteristic(characteristic: BluetoothGattCharacteristic, gatt: BluetoothGatt?) {
    val cccdUuid = UUID.fromString(CCCD_DESCRIPTOR_UUID)
    characteristic.getDescriptor(cccdUuid)?.let { cccdDescriptor ->
        gatt?.let {
            if (!it.setCharacteristicNotification(characteristic,false)){
                Log.d("ReceiveManager","set␣characteristics␣notification␣failed")
                return
            }
            writeDescription(cccdDescriptor, BluetoothGattDescriptor.
                DISABLE_NOTIFICATION_VALUE, it)
        } ?: Log.d("ReceiveManager", "Not␣connected␣to␣a␣BLE␣device!")
    }
}
private fun findDataPacket(buffer: MutableList<Byte>, startingSequence: List<Byte>, endingSequence: List<Byte>): Pair<ByteArray?, Int> {
    if(buffer.size >= startingSequence.size) {
        val startingIndex = buffer.windowed(startingSequence.size).
            indexOfFirst { it == startingSequence }
        if (startingIndex >= 0 && buffer.size > startingIndex +
            endingSequence.size) {
            val endingIndex = buffer.subList(startingIndex +
                startingSequence.size, buffer.size).windowed(
                    endingSequence.size).indexOfFirst { it == endingSequence }
            if (endingIndex >= 0) {
                ...
val actualEndingIndex = startingIndex + startingSequence.size + endingIndex
val byteArray = buffer.subList(startingIndex, actualEndingIndex + endingSequence.size).toByteArray()
return Pair(byteArray, actualEndingIndex + endingSequence.size)

return Pair(null, -1)