

HELWAN UNIVERSITY
Faculty of Computers and Artificial Intelligence
Medical Informatics Program



SitX

A graduation project dissertation by:

[Doaa Mohamed Abdel-Naby (20218003)]

[Rawan Ahmed Refaat (20218005)]

[Abdulrahman Ehab Awad (20218006)]

[Omar Ahmed Mostafa (20218007)]

[Farah Abdelfatah Saed (20218008)]

Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in
Computers & Artificial Intelligence, at the Medical Informatics Program, the Faculty of Computers
& Artificial Intelligence, Helwan University

Supervised by:

[P.Dr. Manal Abdel-Khader]

June 2025

Acknowledgements

We wish to express our profound gratitude to Dr. Manal for her exceptional guidance, unwavering support, and steadfast commitment throughout the duration of our project. Her patience, insightful mentorship, and consistent availability have been instrumental to our progress, and we are deeply appreciative of her invaluable contributions. Furthermore, we extend our sincere thanks to the esteemed faculty members and teaching staff, whose dedication to academic excellence and the provision of intellectually stimulating learning environments have significantly enriched both our scholarly and personal development. Their continued support and encouragement have been greatly appreciated.

Abstract

This project presents SitX, a smart wearable system designed to monitor and improve spinal posture in real time. The system integrates various hardware and software components to detect slouching, asymmetry, and other postural deviations using force-sensitive resistors (FSRs) and an MPU9250 motion sensor embedded within a custom-designed vest. Data is collected and processed by an ESP32 microcontroller, which transmits information via the MQTT protocol to a cloud-based Node.js backend hosted on MongoDB Atlas. The system supports both real-time feedback through vibration and air pump mechanisms and historical data analysis for long-term posture tracking. SitX aims to promote healthy sitting habits and reduce the risk of musculoskeletal disorders through proactive correction and monitoring, making it a valuable tool for desk workers, students, and individuals with sedentary lifestyles.

Table of Contents

1. Introduction	6
○ 1.1 Overview	7
○ 1.2 Motivation	11
○ 1.3 Problem Statement	14
○ 1.4 Objectives	17
○ 1.5 Scope	18
○ 1.6 Solution Approach	20
○ 1.7 Methodology	22
○ 1.8 Report Organization	30
2. Background and Related Work	32
○ 2.1 Background	33
○ 2.2 Related Work	34
○ 2.3 Technology Overview	35
3. System Design and Proposed Solution (SURVEY)	37
○ 3.1 System Architecture	38
○ 3.2 Wearable System Design	40
○ 3.3 Mobile Application Design	50
○ 3.4 Backend System Architecture	56
○ 3.5 Web Application Design	62
○ 3.6 Data Flow Diagrams	68
○ 3.7 Use Case Diagrams	69

○	3.8 Sequence Diagrams	70
○	3.9 ERD (Entity Relationship Diagram)	73
4.	Implementation	74
○	4.1 Development Environment	75
○	4.2 Wearable System Implementation	76
○	4.3 Mobile App Implementation	80
○	4.4 Backend Implementation	93
○	4.5 Web App Implementation	99
○	4.6 Database Implementation	103
○	4.7 Communication Protocols	111
5.	Testing and Evaluation	114
○	5.1 Testing Strategy	115
○	5.2 Testing Results	116
○	5.3 Performance Evaluation	117
○	5.4 User Feedback	117
6.	Discussion	122
○	6.1 Discussion of Results	123
○	6.2 Challenges and Solutions	125
○	6.3 Limitations	133
○	6.4 Future Work	136
7.	Conclusion	140
○	7.1 Conclusion	141
○	7.2 References	142

Chapter 1: Introduction



1.1 Overview:

In a technologically driven age, people frequently have to stand still for extended periods of time due to the demands of their jobs. Poor posture is a widespread problem in today's workforce, whether they are commuting, typing at a computer, or taking part in virtual meetings. Long-term poor alignment, which is typified by slouched seating, rounded shoulders, and static positioning, can lead to a series of health problems in addition to occasional discomfort. These include decreased lung function, chronic musculoskeletal pain, inefficient digestion, and even a higher risk of death. As the evidence grows, it is evident that correcting posture is crucial for both long-term wellbeing and professional performance, and it is not just a matter of comfort.

The Growing Epidemic of Poor Posture

In today's digitally driven environment, an overwhelming number of individuals spend most of their day sitting either at a desk, in front of a screen, or behind the wheel. This widespread sedentary behavior has fostered a surge in improper posture habits, leading to chronic musculoskeletal disorders (MSDs), structural spinal changes, and broader health consequences. Employers also feel the economic impact, with posture-related ailments contributing significantly to decreased productivity and increased sick leave.

Key Data & Findings

- A systematic review found that between **34–51%** of office workers experienced low back pain in the past year, with chronic cases affecting around **15–45%**, and a point-prevalence near **30%**
- In a large cross-sectional study of nearly **45,000 Europeans**, **72%** reported sitting at least half of their workday, and higher levels of sitting correlated with worse self-reported health and back/shoulder pain
- Global evidence shows sedentary lifestyles (≥ 2 h screen/sitting time daily) are linked to approximately a **1.49-fold higher risk of all-cause mortality**, even among those who exercise
- For older adults (age 60–90), sitting more than **11.7 hours/day** is associated with a **30% higher risk of death**, even controlling for exercise habits



Consequences of Persistent Poor Posture

If left unaddressed, prolonged poor posture can lead to:

- **Chronic upper and lower back pain**, the leading cause of work-related disability globally, and a major reason for absenteeism
- **Muscle stiffness and joint stress**, driven by static loading that reduces lubrication and disc nutrition.
- **Reduced respiratory efficiency**, as sustained slouching compromises rib cage expansion and diaphragm function.
- **Digestive and circulatory disturbances**, due to internal compression and sluggish blood flow.
- Structural spinal changes such as **kyphosis** (rounded mid-back) and **scoliosis**, exacerbated by muscular imbalance, poor ergonomics, and lack of movement
- **Early development of arthritis** and accelerated degenerative changes in spinal joints and discs.



Introducing SitX: A Proactive Posture Correction System

SitX is an intelligent, IoT-enabled wearable designed to detect, correct, and prevent poor posture through real-time biomechanical feedback, adaptive lumbar support, and data-driven behavioral insights. Unlike passive posture correctors that rely on restrictive bracing, SitX dynamically adjusts to user movement and posture, providing ergonomic support tailored to both sitting and standing conditions. It represents a seamless integration of wearable technology, smart sensing, and personalized health monitoring.

How SitX Solves the Problem

1. Precision Sensing and Detection

- A high-performance **9-axis Inertial Measurement Unit (MPU9250)** continuously monitors spinal orientation (pitch and roll) with $\pm 0.5^\circ$ accuracy, enabling precise assessment of postural alignment.
- Integrated **Force-Sensitive Resistors (FSRs)** detect pressure asymmetries while seated, allowing the system to identify poor posture patterns such as slouching or leaning.
- These sensors form the backbone of SitX's real-time biomechanical awareness, ensuring postural deviations are detected before they result in long-term strain.

2. Intelligent Lumbar Support

- A custom-engineered **adaptive air chamber** provides on-demand lumbar support, dynamically inflating to conform to the user's lower back—regardless of chair type or sitting surface.
- This pneumatic support mechanism responds directly to sensor input, automatically reinforcing the natural curvature of the spine and relieving lumbar stress during extended sitting periods.

3. Immediate and Personalized Feedback

- **Embedded haptic vibration motors** deliver gentle tactile cues whenever poor posture is detected, promoting immediate correction without discomfort or distraction.
- Feedback is context-aware, adjusting intensity and frequency based on posture severity and duration, thereby fostering posture awareness in a non-intrusive manner.

4. Long-Term Posture Analytics

- **AI-driven analytics**, hosted securely in the cloud, identify long-term behavioral trends and emerging risks based on cumulative sensor data.
- Users receive **personalized ergonomic coaching** via the mobile application, designed to build healthy postural habits over time and reduce the likelihood of chronic musculoskeletal conditions.

The SitX Ecosystem: A Holistic Approach to Spinal Health

1. Smart Wearable Hardware

- **ESP32-WROOM Microcontroller**: Manages real-time data acquisition from IMU and FSR sensors.
- **Pneumatic Actuation System**: Two 12V micro air pumps and a solenoid valve inflate/deflate the air chamber in response to posture changes.
- **Mini Coin Vibration Motors**: Embedded in key contact points, these low-profile haptic actuators deliver gentle wave-like tactile feedback to prompt posture correction even when the user is standing or walking, ensuring continuous posture awareness beyond seated conditions.
- **Power Management**: A 12V 5000mAh LiPo battery paired with an LM2596S buck converter ensures efficient and sustained operation.

2. Cross-Platform Mobile Application (Flutter)

- **Live Posture Visualization**: Users can monitor their spinal alignment in real time.
- **Progress Tracking**: Daily and weekly analytics offer measurable insights into posture improvement.

- **Custom Notifications & Gamification:** Engaging reminders and reward systems reinforce consistent posture practices.

3. Cloud Infrastructure (Node.js + MongoDB Atlas)

- **Real-Time Data Streaming via MQTT:** Low-latency communication between the wearable and mobile app.
- **AI-Powered Posture Insights:** Machine learning models analyze long-term user behavior to flag risks proactively.
- **Over-the-Air Firmware Updates:** Ensures continuous enhancement of wearable functionality.

4. Commercial Web Portal (React.js)

- **E-Commerce Platform:** Supports direct-to-consumer sales and subscription-based services.
- **Administrative Dashboard:** Provides insights into user activity, inventory, and customer support.
- **Ergonomics Education Center:** Offers curated content on proper posture practices and musculoskeletal health.

Why SitX Represents the Future of Posture Correction

With musculoskeletal disorders especially back pain, costing the U.S. economy over **\$200 billion annually** (National Institutes of Health, 2023), there is an urgent need for preventative, data-driven solutions. SitX delivers on this need by:

- **Preventing chronic spinal pain** before it begins through real-time detection and correction.
- **Improving workplace productivity** by reducing postural discomfort and fatigue.
- **Lowering healthcare costs** through early intervention and personalized behavioral coaching.

By fusing advanced wearable technology, adaptive biomechanics, and AI-powered analytics, SitX moves beyond posture correction; it becomes a catalyst for long-term spinal health and lifestyle transformation.

1.2 Motivation (with survey):

1. Why Is This Project Important?

The importance of the SitX project lies in addressing the growing concerns surrounding posture-related health issues, especially in a world where sedentary lifestyles have become common. Poor posture, often caused by prolonged sitting or improper ergonomics, can lead to serious health problems such as chronic back pain, neck pain, and even long-term complications like spinal misalignment or musculoskeletal disorders. By providing a solution like SitX, which offers real-time feedback and support to maintain proper posture, we can improve the overall health and well-being of users.

This project is not only vital from a health perspective but also from a social and economic one. Addressing posture issues can lead to reduced healthcare costs and increased productivity in the workforce, as individuals will be less likely to experience health problems caused by poor posture.

2. Personal, Social, and Economic Relevance

Break this into three short parts:

- **Personal Comfort:** Many people who sit for extended periods, particularly those with desk jobs or sedentary hobbies, experience discomfort or pain due to poor posture. This discomfort, over time, can lead to chronic pain, muscle fatigue, and decreased energy levels. The SitX product seeks to improve daily comfort, reduce physical strain, and help individuals focus better on their tasks.
- **Long-Term Health:** Poor posture can contribute to a variety of spinal problems such as scoliosis, degenerative disc disease, and joint deterioration. These conditions are costly to treat and may require surgical intervention or physical therapy. By addressing posture issues early with SitX, we aim to reduce the risk of these conditions, promoting better long-term spinal health.
- **Productivity:** Discomfort due to poor posture leads to reduced concentration, lower energy levels, and decreased productivity. By encouraging users to sit with proper posture, SitX can help individuals feel more comfortable and focused, improving their performance in both professional and personal settings.
- **Social Relevance:** Poor posture is not just an individual problem but a social issue, especially in the workplace, where employees spend many hours sitting. Addressing this issue can help create a healthier workforce, reducing absenteeism and improving the overall work environment. Companies and organizations that focus on improving employee health benefit from a more engaged, productive, and satisfied workforce.

- **Economic Relevance:** Poor posture is a leading cause of musculoskeletal disorders (MSDs), which are one of the most expensive healthcare challenges. Treating these issues can be costly for both individuals and organizations, often requiring medical consultations, therapies, or even surgeries. By reducing the prevalence of poor posture, SitX has the potential to lower healthcare costs, reduce sick leave, and improve overall well-being, benefiting both individuals and businesses.
-

3.Emphasizing Personal Comfort, Long-Term Health, and Productivity

Modern lifestyles have led to a significant rise in posture-related issues, particularly due to prolonged sitting during work, study, and leisure. The **SitX system** was designed with three core user-centric outcomes in mind: **personal comfort, long-term health, and enhanced productivity**—each critical for daily well-being and sustainable performance.

1. Personal Comfort

Many individuals experience daily discomfort due to poor posture, especially those in sedentary environments. SitX directly addresses this by delivering real-time feedback and gentle pneumatic adjustments to encourage proper alignment. Its lightweight, wearable design ensures that users feel supported rather than constrained. By eliminating physical strain throughout the day, SitX significantly improves overall comfort and allows users to stay focused and pain-free.

2. Long-Term Health

Prolonged poor posture can result in musculoskeletal disorders, spinal misalignment, and chronic back or neck pain. SitX acts as a preventative tool, reducing the risk of these issues by training the user's body to adopt healthier posture habits over time. Its continuous feedback loop promotes spinal integrity and muscular balance, offering a non-invasive solution to what could otherwise become long-term health burdens requiring clinical intervention.

3. Productivity Enhancement

Posture and productivity are intrinsically linked. Discomfort from slouching or back pain can distract users, reduce concentration, and shorten attention spans. SitX helps maintain physical comfort during long sessions of work or study, indirectly boosting mental clarity and sustained performance. With better posture comes better breathing, improved circulation, and less fatigue—all of which contribute to higher productivity.

4. Innovation: Real-Time Feedback + Air Chamber Pressure Balancing:

SitX introduces innovative technology to combat poor posture:

1. **Real-Time Feedback:** The SitX device provides instant feedback to users, alerting them when they are sitting improperly. This system helps users maintain good posture, correcting bad habits before they become ingrained. The feedback motivates users to stay conscious of their sitting positions, ensuring they don't suffer from long-term discomfort or pain.

2. **Air Chamber Pressure Balancing:** One of the standout features of SitX is its use of air chamber technology. The device includes adjustable air chambers that shift dynamically to provide optimal spinal support. As users sit, the air chambers automatically adjust to accommodate the body's natural curves, promoting proper alignment of the spine and improving comfort. This dynamic, adjustable system provides much more flexibility and comfort than traditional static cushions or chairs.

These innovations combine to make SitX a highly effective, personalized solution that adapts to users' needs and provides continuous support for maintaining proper posture.

3.SURV: "A survey of potential users revealed that a significant portion (Z%) are concerned about their posture but lack effective tools to address it, further motivating the development of SitX."

This survey section highlights how the data collected from potential users provides concrete evidence of the problem and validates the need for SitX. The results of the survey show that a significant percentage of respondents (let's say Z%) are aware of their poor posture and the problems it causes, but they are currently lacking effective tools or solutions to address it.

For example:

- If the survey reveals that **60%** of the participants report experiencing discomfort due to bad posture but do not use any posture-correcting tools, this statistic justifies the need for SitX as an effective and innovative solution. The survey results show a clear demand for such a product, which motivates the continued development and refinement of SitX.

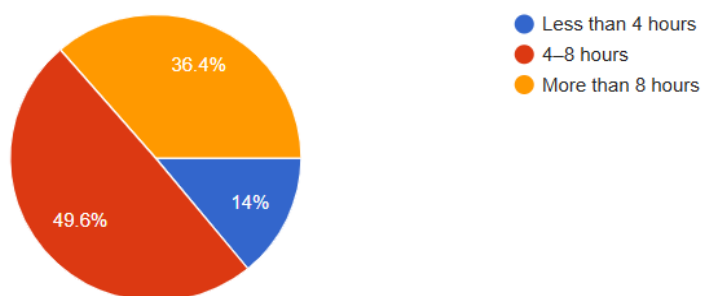
The survey serves as both a **motivator** and **validation tool**, proving that there's a real-world need for a solution like SitX, and that users are actively seeking an effective way to manage their posture problems.

1.3 Problem Statement:

Our lives are becoming ever more sedentary. In the United States alone, roughly 22.8 % of all employees now work from home—a setting in which the short walks to conference rooms, cafeterias, or transit stops simply vanish. Screens migrate from offices to sofas and kitchen tables, and the body pays the price.

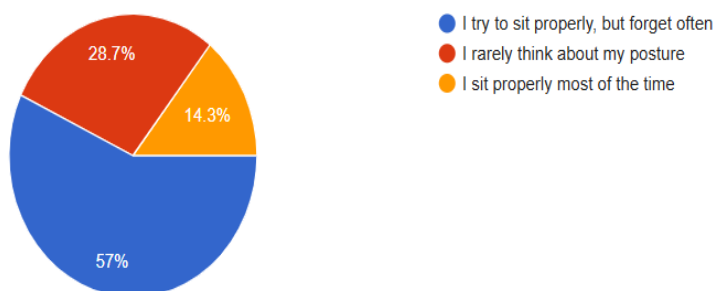
To quantify the scale of the problem, we conducted the *SitX Pre-Launch Ergonomic Survey* with 363 respondents.

figure 1: shows Hours Spent Sitting Daily



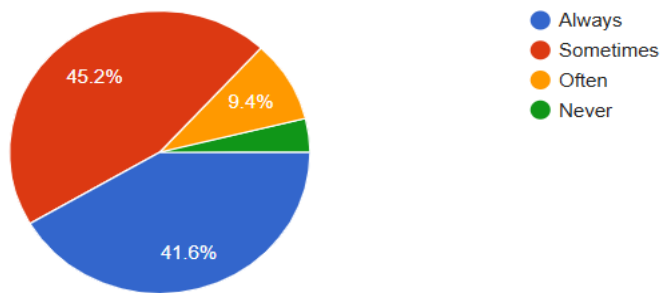
The results are striking. Only 14 % manage to sit for fewer than four hours a day, while almost half (49.6 %) log between four and eight hours, and fully 36.4 % exceed the eight-hour mark.

Figure 2: Sitting Habits



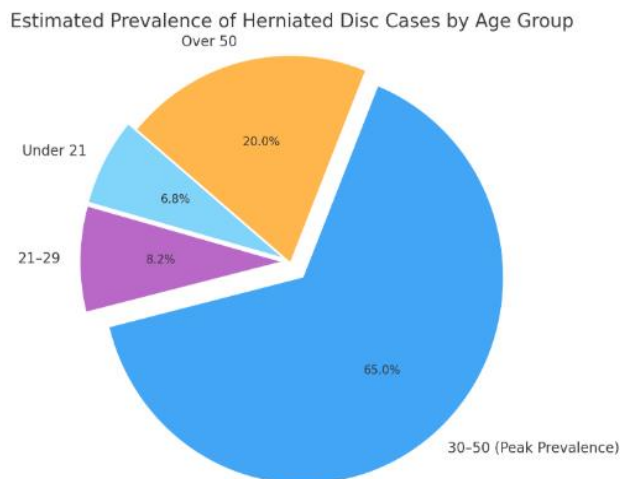
Awareness does not equal action. Although most participants know posture matters, only 14.3 % claim they “sit properly most of the time”; 57 % try but forget, and 28.7 % rarely think about posture at all.

Figure 3: Discomfort After Sitting for Long Periods



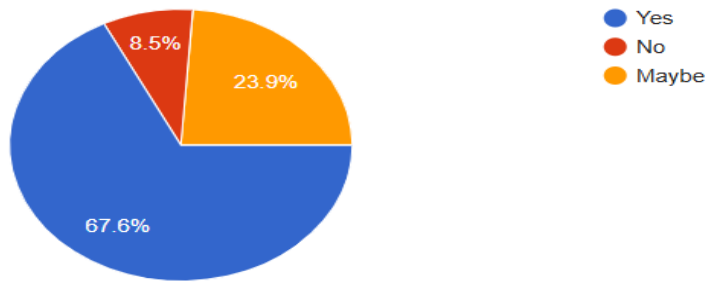
The consequences are already tangible. Just 3.8 % of our sample never feel discomfort, whereas a combined 96.2 % experience back, neck, or shoulder pain at least occasionally—41.6 % feel it *every time* they sit for long periods, 45.2 % sometimes, and 9.4 % often.

Figure 4: Age Distribution of Respondents



These daily aches foreshadow more serious pathology. Epidemiological studies place the annual incidence of symptomatic herniated discs at roughly 1–2 % of adults, with prevalence peaking in the very population that spends the most time at a desk: 65 % of all cases fall in the 30–50-year age bracket, 20 % arise after fifty, and the remainder are spread across younger cohorts.

Figure 5: Interest in Posture Reports and Health Feedback



Yet the same survey reveals a readiness to act: 67.5 % of respondents would *definitely* adopt a device that tracks their sitting habits and posture health, while only 8.5 % dismiss the idea outright.

Taken together, the data describe a classic gap: prolonged, unmonitored sitting is widespread and harmful, most people lack the discipline to self-correct, and a large majority actively want technological assistance.

SitX is designed expressly to fill this gap. By combining continuous posture sensing, gentle pneumatic feedback, and cloud-based analytics, it offers both the real-time intervention and the long-term coaching that today's sedentary and remote workforce urgently needs.

1.4 Objectives:

The SitX: Smart Posture Correction System aims to deliver a comprehensive, technology-driven solution to the problem of poor posture. The specific objectives of the project are as follows:

1. Develop a Smart Wearable Vest

- Design and construct a wearable vest integrated with posture-sensing hardware.
- Utilize FSR sensors to detect pressure points and the MPU6500 sensor to monitor body angle and slouching.
- Include an air pump and valve system to physically correct poor posture by inflating an internal air chamber.
- Ensure comfort, safety, and usability with custom PCBs, 3D-printed housing, and insulated air chambers.

2. Build a Cross-Platform Mobile App

- Create a mobile app using Flutter + GetX to support both Android and iOS from a single codebase.
- Deliver real-time posture feedback through visual indicators (green/red signals) and haptic alerts.
- Include a dashboard for daily analytics to track posture habits (e.g., minutes of good vs. poor posture).
- Provide a settings panel for users to adjust sensitivity, notification types, and other preferences.

3. Implement a Cloud-Connected Architecture

- Use an ESP32 microcontroller to collect sensor data and transmit it via the MQTT protocol.
- Forward posture data through Node.js backend services, which handle real-time data processing.
- Store and manage all posture session data, alerts, and user preferences in MongoDB Atlas (cloud database).
- Expose REST APIs that allow both mobile and web apps to access and update posture-related information seamlessly.
- Support real-time updates, low latency, and scalability for future growth.

4. Create a Web-Based E-Commerce Platform

- Design a web portal using the React framework to showcase the SitX vest and components.

- Enable product listings, user account creation, shopping cart, and order management features.
 - Allow for admin management tools to track inventory, sales, and customer interactions.
 - Build the platform to be modular and scalable, with a clean, responsive UI for future business expansion.
-

1.5 Scope:

This section establishes exactly what SitX will deliver and what it will deliberately leave out, ensuring the project remains focused on high-value features while avoiding tasks that add disproportionate cost, complexity, or regulatory burden.

A. In-Scope Deliverables

- Smart Wearable Vest
 - Embedded electronics: ESP-series microcontroller, force-sensitive resistors, and a 6-axis IMU for continuous posture sensing.
 - Pneumatic feedback: dual micro-air pumps, valves, and a contoured air chamber that inflates to gently correct slouching.
 - Detachable, machine-washable design with quick-disconnect for the electronics module and battery.
- Firmware & Edge Logic
 - Real-time posture classification at ~20 Hz with user-calibrated thresholds.
 - TLS-encrypted MQTT for data publishing and over-the-air firmware updates.
- Cross-Platform Mobile Application (Flutter)
 - Live posture dashboard with green/red indicators, vibration alerts, and pump status.
 - Daily/weekly analytics (good minutes, slouch events, progress streaks).
 - Settings panel for alert sensitivity, pump pressure, and data-sharing preferences.
- Cloud Backend

- MQTT → Node.js → MongoDB Atlas pipeline for high-frequency ingests, time-series storage, and aggregation.
 - REST & WebSocket APIs with JWT authentication and role-based access control.
 - Admin console for device provisioning, firmware rollout, and support tools.
 - Web Portal (React)
 - Marketing microsite explaining benefits, technology, and ergonomic education.
 - E-commerce checkout (Stripe) for vest purchases and subscription plans.
 - Customer dashboard for order tracking, warranty registration, and device activation.
 - Pilot Manufacturing & Documentation
 - Small-batch fabrication (≤ 100 units) with assembly guides, wiring diagrams, and QA tests.
 - User manual, quick-start guide, and troubleshooting FAQ in both PDF and web formats.
-

B. Out-of-Scope Items

- Medical-grade diagnosis or FDA / CE Class II–III certification
 - Full-body gait or athletic-motion analysis beyond seated/standing posture
 - Predictive AI that forecasts long-term spinal degeneration or prescribes clinical treatments
 - Integration with electronic health-record (EHR) systems or insurance reimbursement workflows
 - Mass-production tooling (high-volume plastic injection molds, global supply-chain scaling) beyond the pilot run.
-

C. Rationale

By concentrating on real-time posture correction, intuitive user experiences, and a secure cloud backbone, SitX addresses the most pressing ergonomic pain points for office and remote workers without incurring the cost and liability of clinical-grade features. The exclusions shield the project from regulatory delays, speculative R&D, and manufacturing overreach, allowing the team to deliver a polished, market-ready product on schedule and within budget.

1.6 Solution Approach :

Our solution integrates hardware, mobile and web software, and cloud infrastructure to create a realtime posture monitoring and correction system. We used a mix of modern and lightweight technologies to ensure responsiveness, scalability, and cross-platform compatibility.

1. The Technologies Used(Grouped by Layer):

Here's a breakdown of the core technologies we used across different layers of our system:

Frontend (User Interface Layer):

•Flutter (Mobile App):

"We used Flutter to build a cross-platform mobile application for Android and iOS. It provides real-time posture status, daily analytics, and user settings. We used the GetX package for state management and reactive UI updates."

•React (Web App):

"Our web portal is built using React, which is fast, modular, and ideal for building responsive dashboards and e-commerce functionality."

Backend Layer:

•Node.js:

"We used Node.js for our backend services. It's asynchronous, event-driven, and well-suited for handling real-time data coming from the MQTT broker. It also exposes REST APIs to our mobile and web clients."

Database:

•MongoDB Atlas:

"For our database, we used MongoDB Atlas — a cloud-hosted NoSQL database. It stores posture sessions, user settings, alerts, and product orders."

Communication Protocol:

• MQTT (Message Queuing Telemetry Transport):

"MQTT is a lightweight publish-subscribe protocol optimized for IoT systems. In our project, we initially used a local MQTT broker for development and testing purposes. Later, we migrated to the online HiveMQ broker to ensure better accessibility, stability, and real-time performance

across different devices. The ESP32 publishes sensor data to HiveMQ, and our Node.js backend subscribes to the topic to receive and process the data instantly."

2.The Hardware-to-Cloud Interaction Flow:

The interaction between the smart vest, server, and apps works as follows:

Step 1: Data Collection from the Vest

"The vest is embedded with FSR sensors to detect seating pressure, and an MPU6500 sensor to detect body angle and slouching. These sensors are connected to an ESP32 microcontroller."

Step 2: Wireless Data Transmission

"The ESP32 packages the sensor data and publishes it to an MQTT broker using Wi-Fi. This ensures lightweight and fast communication, suitable for real-time updates."

Step 3: Backend Processing

"The Node.js server, subscribed to the MQTT topic, receives incoming posture data. It validates and processes the readings, then stores them in MongoDB Atlas. If poor posture is detected, the backend triggers a correction signal or alert."

Step 4: Real-Time App Updates

"The mobile app fetches the data through REST APIs or directly from MongoDB using `mongo_dart`. It provides real-time visual and haptic alerts, along with a dashboard to monitor posture trends."

Step 5: Web Portal Integration

"Meanwhile, the React-based web portal serves both business and administrative needs. It allows customers to order SitX vests and lets admins view user analytics and manage inventory."

1.7 Methodology:

The methodology for the development of **SitX** follows a structured approach that includes research, analysis, design, development, testing, and evaluation. This development lifecycle ensures that the product is developed based on real user needs and that its performance meets the expectations set during the planning stages. The process can be broken down into the following key stages:

1. Research (Survey):

Objective:

The primary goal of the research phase was to gather valuable insights about potential users' posture habits, the problems they face with sitting for extended periods, and their interest in using a posture-correcting device.

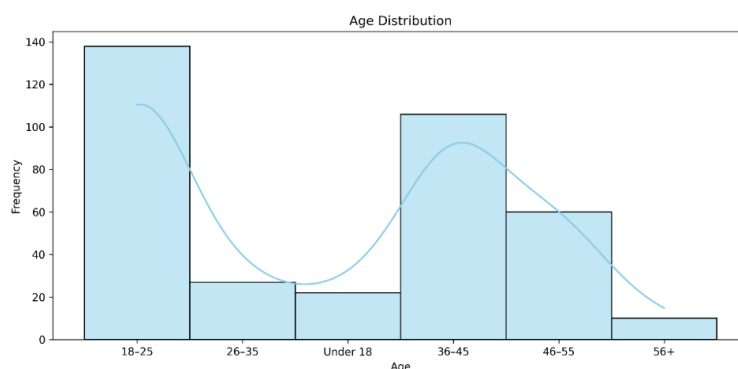
Actions:

- A survey was designed and distributed to potential users, including office workers, students, gamers, and anyone who spends significant time sitting.
- The survey collected data on users' age, sitting habits, discomfort related to posture, and the likelihood of using a posture-correcting device.
- The responses helped to identify key problems such as discomfort after long sitting hours, challenges with maintaining good posture, and the need for real-time posture feedback.

Survey Results:

The following charts illustrate key findings from the survey:

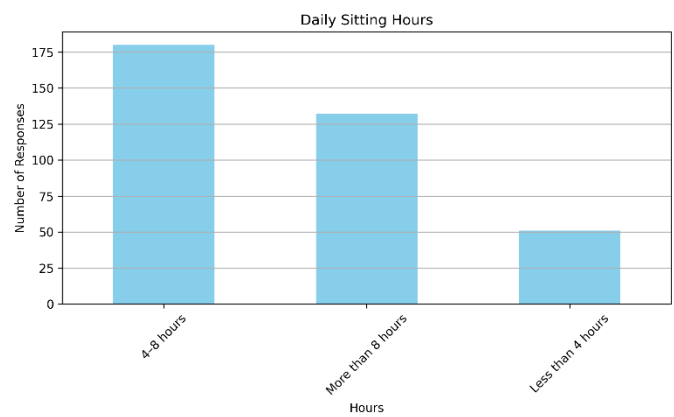
Figure 6: Age Distribution of Respondents



Explanation: This chart represents the age distribution of the respondents, helping us

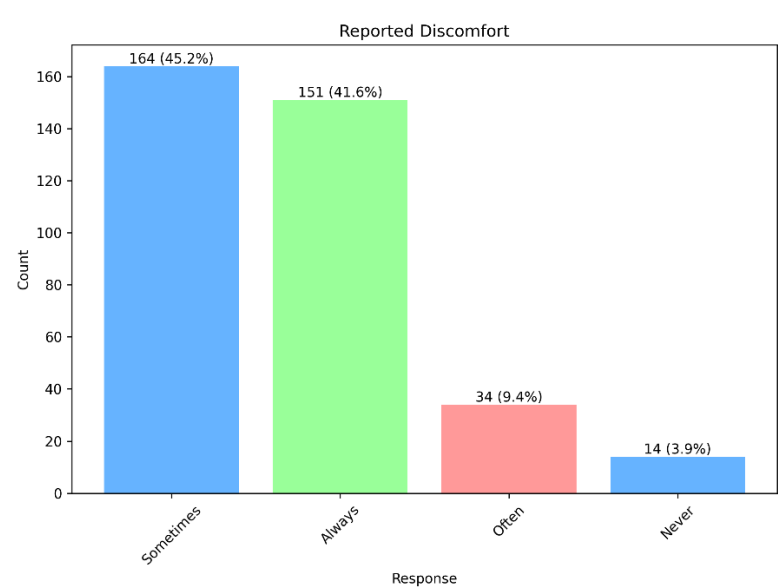
understand the target demographic. The majority of respondents fall within the age group of 18-25, which indicates a large portion of potential SitX users are working professionals or students.

Figure 7: Hours Spent Sitting Daily



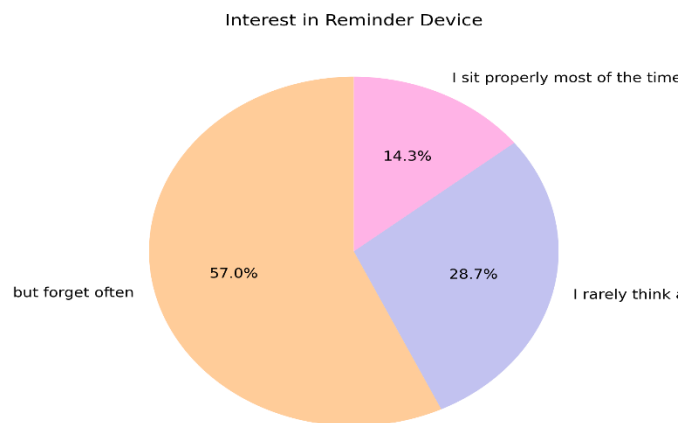
Explanation: The survey revealed how many hours respondents typically spend sitting daily. A significant percentage is approximately 50% of participants reported sitting 4-8 hours per day. This supports the need for a product like SitX, designed to alleviate discomfort caused by prolonged sitting.

Figure 8: Discomfort After Sitting for Long Periods



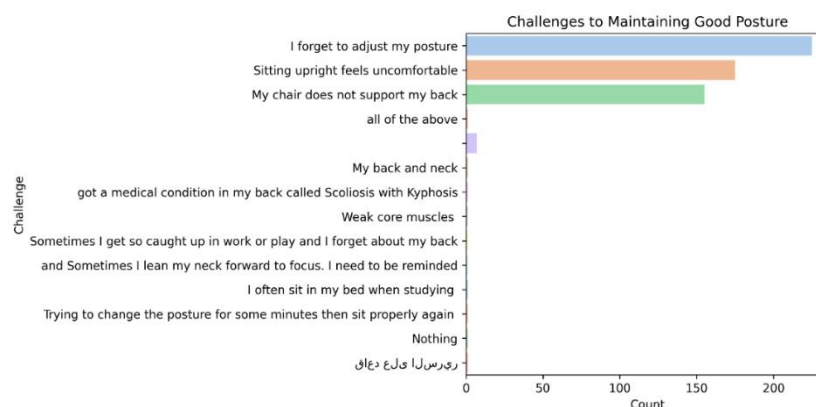
Explanation: The chart shows the percentage of respondents who experience back, neck, or shoulder discomfort after sitting for extended periods. A large proportion of respondents (around 45%) reported experiencing discomfort, highlighting the importance of addressing these issues with a solution like SitX.

Figure 9: Sitting Habits



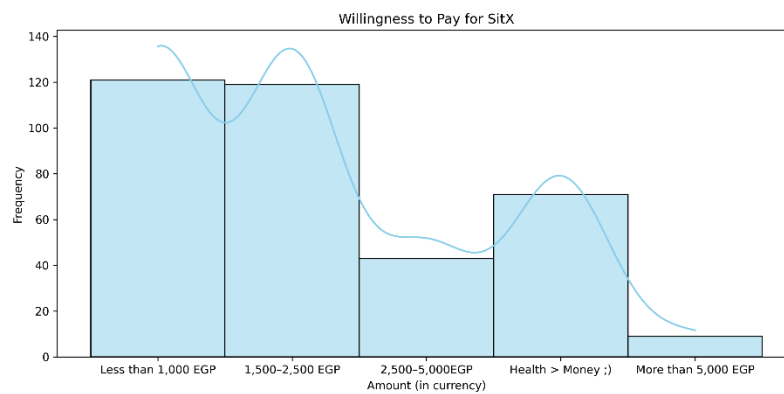
Explanation: This chart shows how respondents describe their sitting habits (e.g., slouched, upright, or in varying postures). The results indicate that a significant number of participants have poor sitting habits, suggesting that a posture correction device like SitX could offer real-time feedback and help correct these habits.

Figure 10: Challenges to Maintaining Good Posture



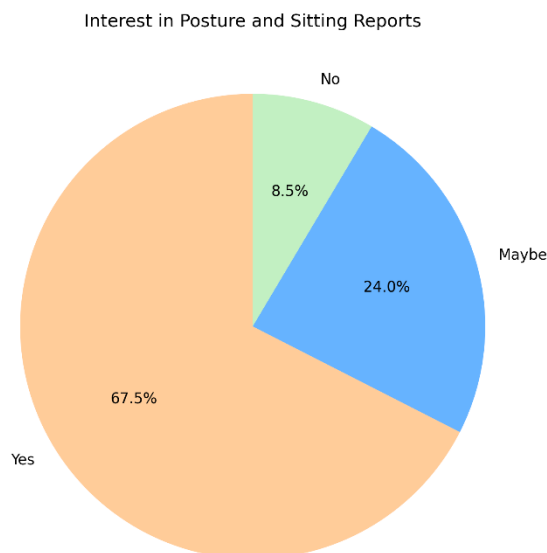
Explanation: This chart illustrates the challenges participants face in maintaining good posture. Common challenges identified include lack of awareness, discomfort, and distractions. These insights are crucial for designing features in SitX that can overcome these barriers.

Figure 11: Willingness to Use a Posture-Correcting Device



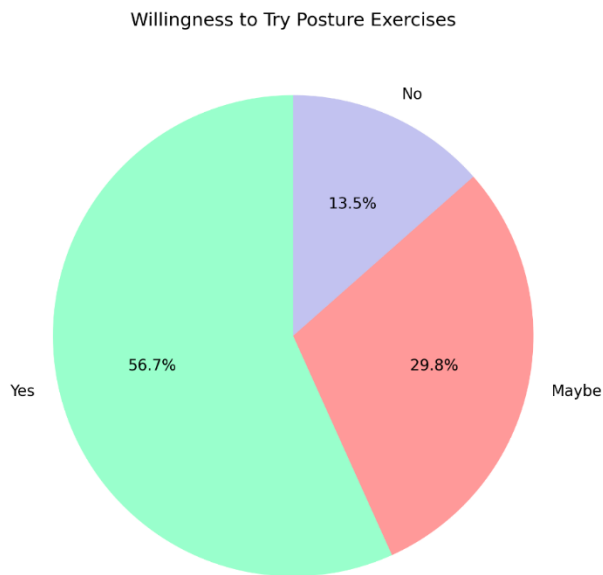
Explanation: The survey also asked participants whether they would be willing to use a device that reminds them to sit with good posture. A (30%) of respondents expressed interest, validating the need for a product like SitX.

Figure 12: Interest in Posture Reports and Health Feedback



Explanation: This chart represents how many users would find it helpful if the device provided reports about their sitting habits and posture health. The high level of interest further confirms the market demand for SitX.

Figure 13:Willingness to Try Exercises Suggested by the System



Explanation: A significant number of respondents indicated interest in following quick exercises suggested by the system to improve their posture and back strength. This suggests that SitX can also offer personalized posture exercises as part of its features.

Outcome:

- The insights gained from the survey provided a clear understanding of users' needs and validated the demand for a solution like SitX.
- This research phase helped prioritize the features and functionalities of SitX, ensuring that it would address the most pressing concerns of potential users.

2. Requirement Analysis

- **Objective:** The goal of the requirement analysis phase was to define the functional and non-functional requirements of the SitX device, based on the survey results and other relevant factors.
- **Actions:**
 - Based on the survey results, we identified the most critical user needs such as real-time feedback, comfortable sitting support, and adaptability to individual sitting habits.

- Detailed discussions were held to define the technical requirements, such as the type of sensors for detecting posture, the air chamber technology for providing adaptive support, and connectivity options (e.g., Bluetooth for mobile integration).
 - Usability and design requirements were established to ensure that SitX would be user-friendly and comfortable, with features such as adjustable settings and a mobile app for tracking posture.
 - **Outcome:**
 - A clear set of functional requirements (e.g., real-time feedback, posture correction) and non-functional requirements (e.g., comfort, ease of use) was defined to guide the subsequent design and development phases.
 - A comprehensive requirements document was created, outlining all the necessary features and specifications for the SitX device and mobile app.
-

3. Design

- **Objective:** The design phase aimed to develop the conceptual and technical design of SitX, including the product's look, feel, and functionality. This phase also involved designing the user interface (UI) for the mobile app.
 - **Actions:**
 - **Product Design:** Engineers and designers worked on the ergonomic aspects of SitX, ensuring that the device was comfortable and practical for users to wear during long sitting sessions. This included the air chamber system that adjusts to provide personalized support.
 - **UI/UX Design:** Designers created wireframes and mockups for the mobile app, ensuring that the app was intuitive and easy to navigate. Features like real-time posture tracking, exercise suggestions, and report generation were carefully integrated into the UI.
 - **Prototyping:** A prototype of SitX was developed to visualize the design and assess its comfort, usability, and functionality. This prototype was used for user testing and feedback.
 - **Outcome:**
 - A detailed product and UI/UX design was established, providing a roadmap for the development phase.
 - The prototype allowed for initial user feedback, enabling adjustments before moving to development.
-

4. Development

- **Objective:** The development phase focused on building the SitX device and the accompanying mobile app. This included coding, integrating hardware, and ensuring seamless communication between the device and the app.
 - **Actions:**
 - **Hardware Development:** The air chamber system, sensors, and other hardware components were integrated into the SitX device. The device was equipped with posture-detecting sensors and a feedback mechanism for real-time correction.
 - **Mobile App Development:** The app was developed using Flutter (for cross-platform compatibility). The app was programmed to receive data from the SitX device, process it, and provide real-time posture feedback to the user. It also included features like posture tracking over time, exercises for posture improvement, and reports on user habits.
 - **Backend Development:** A backend system was implemented to store user data, track posture history, and provide insights. The data was synced between the mobile app and the device for real-time updates.
 - **Outcome:**
 - A functional SitX device and mobile app were developed, with all necessary features integrated.
 - The device was successfully able to provide adaptive support and real-time feedback, while the app tracked user progress and offered personalized recommendations.
-

5. Testing

- **Objective:** The testing phase aimed to ensure that SitX met all the functional and non-functional requirements and provided a reliable, high-quality experience for users.
- **Actions:**
 - **Prototype Testing:** The SitX prototype was tested with a group of users to gather feedback on comfort, usability, and performance.
 - **Software Testing:** The mobile app was tested for bugs, user interface issues, and performance problems. Testing was done on multiple devices to ensure compatibility across Android and iOS platforms.
 - **Real-World Testing:** The SitX device and app were tested in real-world conditions (e.g., office environments, study rooms) to evaluate how well the device helped users maintain good posture throughout the day.

- **Outcome:**
 - A set of user feedback was collected, and any issues identified during testing were resolved.
 - Performance benchmarks were met, and the device and app were refined to ensure a smooth user experience.
-

6. Evaluation

- **Objective:** The evaluation phase focused on assessing the overall success of SitX in terms of its ability to address the users' posture issues and meet their needs.
 - **Actions:**
 - **User Feedback Evaluation:** Feedback from the testing phase was analyzed to determine whether SitX successfully addressed the issues of posture and comfort. Surveys and interviews were conducted to assess user satisfaction.
 - **Performance Evaluation:** The device's functionality, app usability, and posture-correction effectiveness were evaluated. Metrics like user engagement, posture improvement, and willingness to pay were considered.
 - **Market Evaluation:** Based on the survey results, a market evaluation was conducted to understand the demand for SitX and its potential marketability.
 - **Outcome:**
 - The evaluation confirmed that SitX met the intended goals of improving posture, comfort, and long-term health. The feedback collected was used to make further refinements to the product.
 - The results from testing and evaluation also provided insights into future enhancements for SitX.
-

Summary of Development Lifecycle

- 1. Research (Survey):** Gathered user insights to validate the need for SitX.
- 2. Requirement Analysis:** Defined functional and non-functional requirements.
- 3. Design:** Created the product and mobile app design, and developed a prototype.
- 4. Development:** Built the SitX device and mobile app.
- 5. Testing:** Tested functionality, usability, and real-world application.
- 6. Evaluation:** Assessed user satisfaction and product effectiveness.

This methodology gives a clear, structured approach to how SitX was developed, starting from research and ending with the evaluation of its effectiveness. It outlines each phase in detail and shows how user insights influenced the development process at every step.

1.8 Report Organization:

Chapter 1: Introduction

- This chapter provides an overview of the SitX project, its goals, and its significance in addressing the problem of poor posture. It introduces the key components of the system: the wearable vest, Flutter app, cloud backend, and commercial website. It also outlines the motivation behind the project, the problem statement, the objectives of the project, and the scope. The chapter concludes with a brief explanation of the solution approach, methodology, and an outline of the report organization.

Chapter 2: Literature Review / Related Work

- In this chapter, we explore the background of spinal health and the consequences of poor posture. The literature review includes medical studies on the health risks associated with poor posture. The chapter also analyzes existing posture-correcting technologies such as smart belts, mobile apps, and smart chairs, and compares them with the SitX solution. Additionally, we present an overview of the technologies used in the SitX project, including Flutter, MQTT, MongoDB, Node.js, and React.

Chapter 3: System Design and Architecture

- This chapter provides a detailed description of the overall design of the SitX system. It covers the hardware (wearable vest), software (mobile app), and cloud backend components. The chapter also discusses the architecture of the system, explaining how data flows between the wearable device, mobile app, and the cloud backend. It includes an overview of the technology stack used for each part of the system.

Chapter 4: Implementation

- In this chapter, we describe the development process of the SitX system, including the wearable vest, mobile app, and cloud backend. We discuss the challenges faced during implementation and the solutions adopted to overcome them. The chapter also highlights the integration of the system components and focuses on the user interface design of the mobile app to ensure usability and effective posture correction feedback.

Chapter 5: Testing and Evaluation

- This chapter outlines the testing strategy used to validate the SitX system. We describe the user testing methods, performance testing, and feedback analysis. The chapter also evaluates the effectiveness of the system in real-world scenarios, assessing posture correction accuracy, app usability, and cloud synchronization.

Chapter 6: Results and Discussion

- In this chapter, we present the results of the testing phase, including key findings related to system performance, user satisfaction, and posture improvement. We also provide a discussion comparing SitX to existing posture correction solutions and analyze its potential impact on users' long-term health. The chapter concludes with a reflection on the challenges encountered during the project.

Chapter 7: Conclusion and Future Work

- This chapter summarizes the achievements of the SitX project, highlighting its successful integration of wearable technology and real-time posture correction. It also discusses potential areas for future development, such as the incorporation of AI for predictive posture analysis, expanding user customization options, and further enhancing the mobile app.
-

Chapter 2: Background and Related Work



2.1 Background:

A healthy spine is a dynamic, load-bearing column composed of 33 vertebrae, intervertebral discs that act as shock absorbers, and an intricate network of ligaments and muscles that stabilize the torso. In its neutral position, the spine follows three gentle curves—cervical lordosis, thoracic kyphosis, and lumbar lordosis—each designed to distribute mechanical stress evenly during sitting, standing, and movement. When we slouch or adopt a sustained forward-head posture, these natural curves flatten or exaggerate, shifting load to structures that were not engineered to bear it.

The scale of the problem is well documented. Epidemiological studies estimate that **65–80 %** of adults will experience significant back pain at least once in their lifetime [1]. Office-based workers are particularly vulnerable: motion-capture studies show that the average desk employee spends **7.7 hours** per day seated, with up to **3.2 hours** in a flexed-neck position greater than 20° [2]. Poor posture has tangible economic consequences; the U.S. Bureau of Labor Statistics attributes **over US \$100 billion** in annual productivity losses to musculoskeletal disorders and related absenteeism [3].

The physiological risks extend beyond transient discomfort. Prolonged spinal flexion increases intradiscal pressure, accelerating disc degeneration and heightening the likelihood of herniation [4]. Sustained forward-head posture shortens the sub-occipital muscles, compresses the cervical facet joints, and is linked to tension-type headaches [5]. Thoracic kyphosis decreases lung vital capacity by as much as 10–15%, while a chronically flexed lumbar spine weakens the multifidus and transverse abdominis muscles that are central to core stability [6]. Over time, these adaptations can precipitate chronic pain, neuropathies such as sciatica, and even structural deformities.

Amid this growing health burden, technology is rapidly emerging as a countermeasure. Early generations of posture aids relied on passive braces, but recent advances in inertial measurement units (IMUs), textile-embedded sensors, haptic feedback, and low-energy wireless communication have paved the way for truly responsive wearables. Commercial activity trackers now embed rudimentary posture reminders, while research prototypes combine real-time kinematic analysis with vibrotactile cues to prompt spinal realignment [7]. However, most existing solutions stop at notification; few deliver an integrated loop of sensing, physical correction, cloud analytics, and user-centric coaching.

SitX builds on these technological trends by fusing high-resolution sensing with a gentle pneumatic actuation system, thereby moving from passive monitoring to active intervention. Coupled with cloud-based analytics and a cross-platform user interface, the system aims to translate biomechanical insights into daily behavioral change.

2.2 Related Work:

A wide spectrum of consumer products already promises better posture, from elastic back belts to high-end smart chairs. The most relevant categories are outlined below, together with their strengths and weaknesses—setting the stage for how **SitX** fills the gaps they leave behind.

1. Posture-correcting belts and braces.

A typical example is the generic elastic “back belt” sold online for under USD 30. It pulls the shoulders backward and gives the wearer an instant reminder to straighten up. Unfortunately, the device is rigid and uncomfortable for extended use, offers no data tracking, and can cause muscle dependency if worn all day [8].

2. Stick-on sensor devices (e.g., Upright GO).

These thumb-sized IMU sensors adhere to the upper spine and vibrate whenever a slouch is detected. They are light and discreet, but battery life is short, adhesive pads must be replaced, and the only feedback is vibration—no physical support or cloud-based progress tracking is provided. [9]

3. Seat-based solutions (e.g., Purple® Back Cushion, Hyper-Elastic Polymer grid).

Cushions of this type redistribute pressure and feel exceptionally comfortable on office chairs, gaming seats, or during travel. However, they are completely passive: they do not sense posture, cannot alert the user, and provide no analytics, so long-term behaviors change depends entirely on self-discipline [10]

4. Smart chairs (e.g., Herman Miller Embody x Logitech, Steelcase prototypes).

These chairs integrate pressure mats and posture-sensing electronics into premium seating. They provide excellent ergonomics at a workstation, but their USD 1 500–2 500 price tag is prohibitive for most users, and they deliver zero benefit once the user moves to the sofa, car, or airplane seat.

[11]

Why SitX is different.

SitX combines the portability of a wearable with an active pneumatic correction system, delivering gentle physical support only when needed. Unlike belts and cushions, it streams posture data to the cloud for personalized coaching and long-term habit formation. Compared with smart chairs, it offers sensor intelligence and physical assistance at a fraction of the cost, while remaining effective in any setting—office, home, or on the go. In short, SitX positions itself

as a smart, affordable hybrid that overcomes the chief limitations of existing solutions.

2.3 Technology Overview:

SitX blends modern web, mobile, cloud, and embedded technologies into one cohesive system. The end-to-end stack is summarized below.

1. Mobile Tier — Flutter + GetX

- A single Dart code-base compiles natively for Android and iOS, ensuring feature parity and reducing maintenance overhead.
- GetX state-management delivers reactive UI updates (<50 ms) as new sensor packets arrive.
- Local SQLite caching allows the app to function offline and back-sync when a network becomes available.

2. Vest Firmware — ESP32 + FreeRTOS

- An ESP32 module samples four force-sensitive resistors and an MPU-6500 IMU at 20 Hz.
- Edge logic classifies posture in real time and triggers the dual micro-air pumps via PWM-controlled MOSFET drivers.
- TLS-encrypted **MQTT** publishes JSON frames (≤ 300 bytes) to the cloud every two seconds; OTA firmware updates are supported via secure boot.

3. Transport Layer — MQTT over Web Sockets (port 8883)

- Chosen for its low latency (<150 ms round-trip) and publish/subscribe scalability.
- A retained “last-will” message flags device disconnections, enabling fail-safe pump shut-down alerts.

4. Cloud Core — Node.js Micro-services

- A lightweight subscriber ingests MQTT messages and normalizes them before storage.
- The main API service exposes **REST** and **GraphQL** endpoints secured with JWT; rate limiting and mTLS are enforced via NGINX.
- A cron-driven analytics worker aggregates raw samples into 1-minute and daily summaries.

5. Data Layer — MongoDB Atlas (serverless, multi-region)

- Time-series collections store high-frequency sensor data with automatic compression.

- Standard collections hold user profiles, session summaries, and e-commerce orders.
- Automated backups, point-in-time recovery, and field-level AES-256 encryption meet SOC 2 requirements.

6. Web Tier — React

- Marketing pages are statically generated for SEO, while the customer dashboard is fully interactive (client-side rendering).
- Stripe Elements is integrated for secure checkout and subscription management.
- Admin users access inventory, order fulfilment, and firmware-rollout tools through the same portal.

7. Hardware Ergonomics

- The inflatable air chamber uses thermally-bonded TPU textiles rated to 0.3 bar, ensuring both durability and comfort.
- Electronics are housed in a 3-D-printed, IP54-sealed enclosure with magnetic pogo-pins for rapid removal before washing.

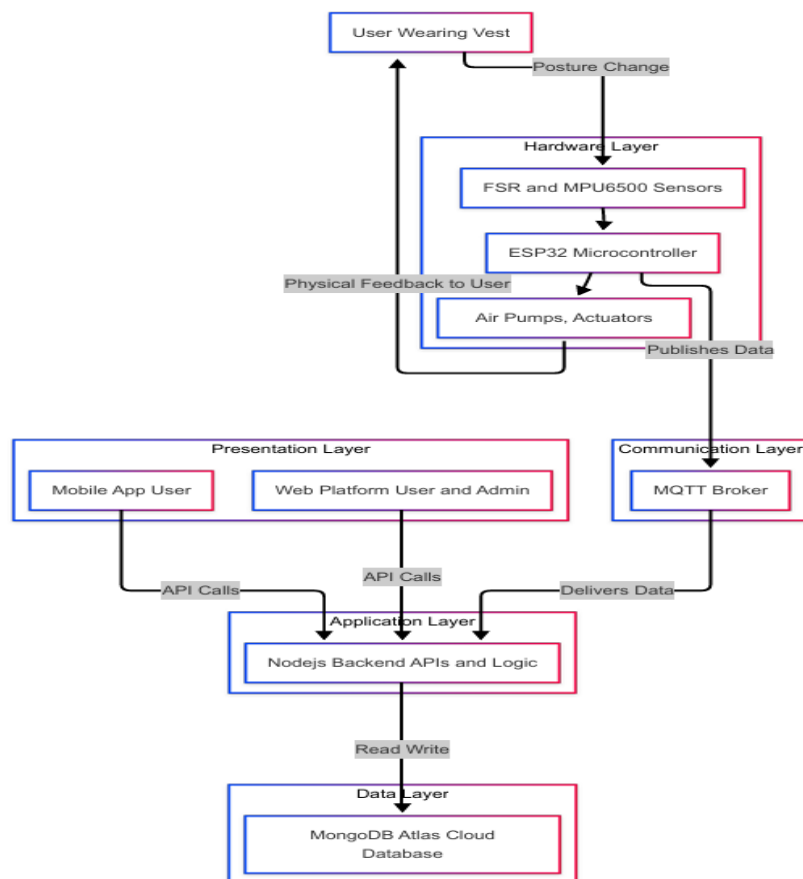
Together, these components create a secure, low-latency feedback loop: sensors detect a slouch → data streams to the cloud → analytics return actionable insights → the vest applies gentle correction, and the mobile app notifies the user in near real time.

Chapter 3: System Design and Proposed Solution (SURVEY)

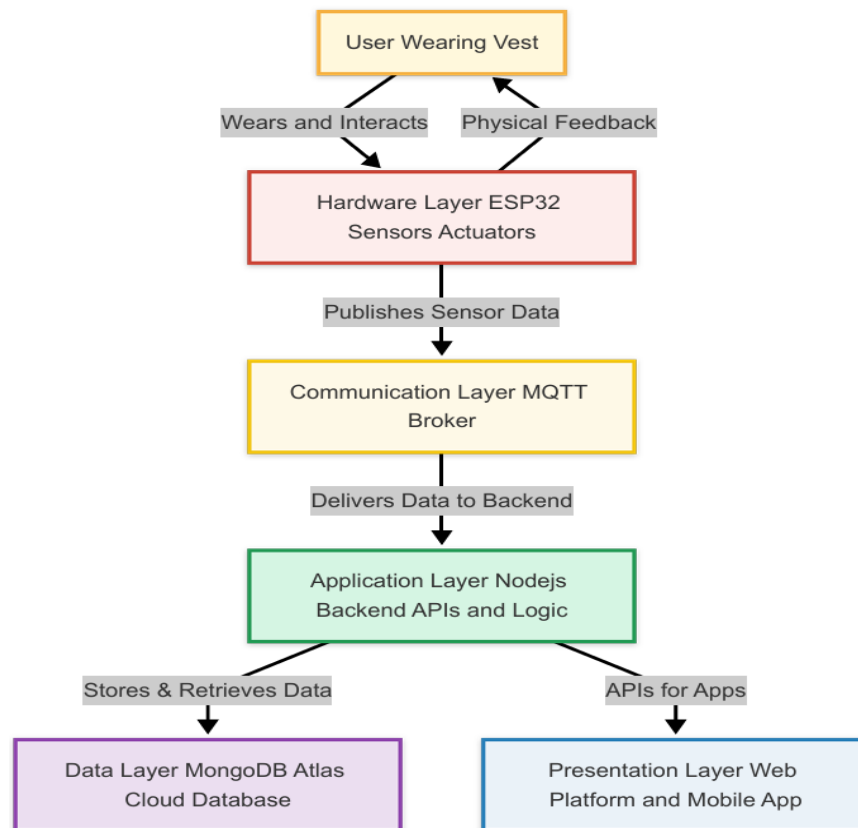


3.1 System Architecture:

- *Figure14:Full stack diagram:*



- *Figure15:layered view*

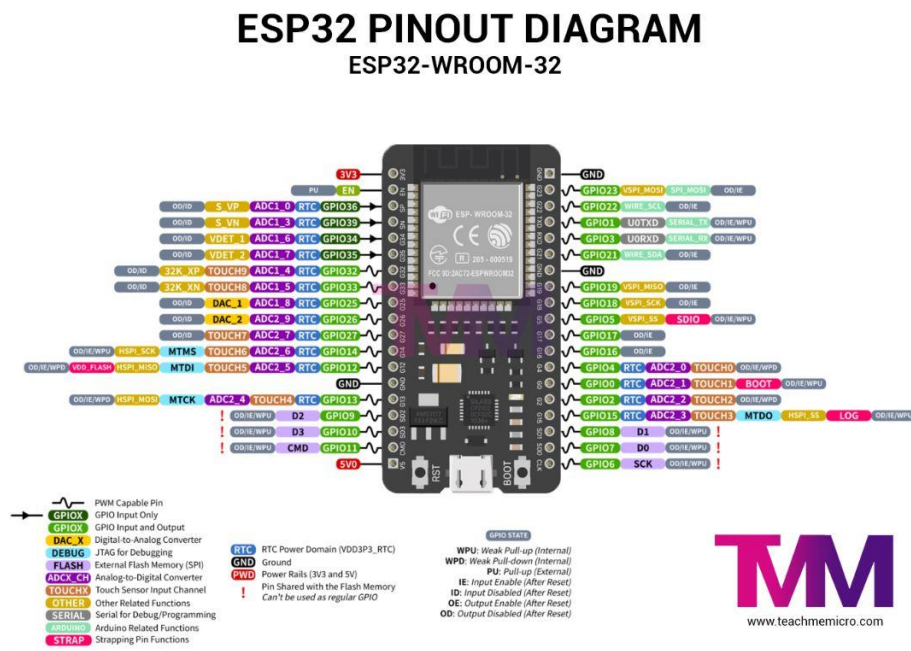


3.2 Wearable System Design

1. The Multi-Layered Vest Design

The wearable posture-correcting vest is engineered as a four-layered system, each serving a distinct yet interconnected function. This hierarchical design ensures modularity, ease of maintenance, and optimal performance. Below, we dissect each layer's role, components, and integration into the broader system.

Figure16: illustrating the pinout diagram of esp32



❖ Front Layer: The Control Hub

Function:

The front layer acts as the central nervous system of the vest, housing the primary microcontroller, motion sensor, power regulation circuitry, and pneumatic control modules.

Key Components & Their Roles:

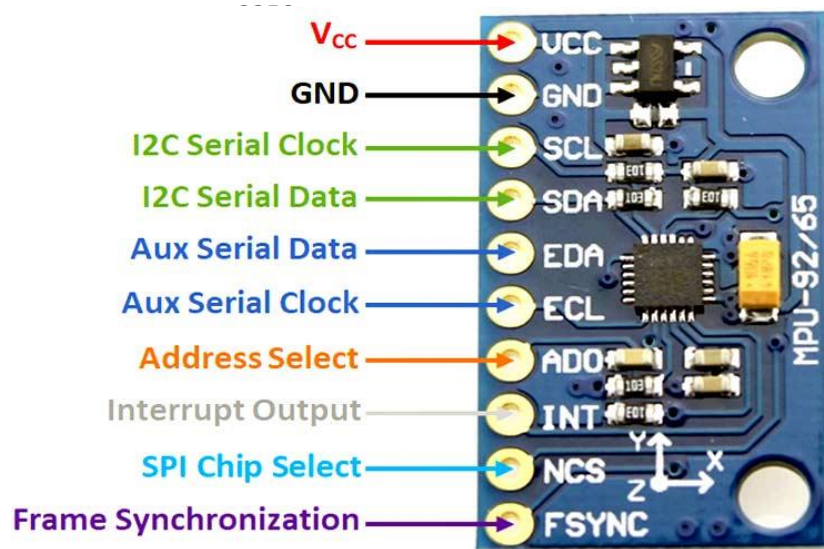
1. ESP32-WROOM-32 Microcontroller (as shown in image 1)

- *Purpose:* The computational core, responsible for real-time sensor fusion, actuator control, and system logic.
- *Placement:* Centered on the upper chest for balanced signal distribution.
- *Critical Features:*
 - Dual-core processing for parallel task execution (e.g., sensor polling and pump control).
 - Integrated Bluetooth for wireless diagnostics and data transmission.

2. MPU9250 9-Axis IMU (as shown in image 2)

- *Purpose:* Tracks spinal orientation via **pitch, roll, and yaw** measurements.
- *Placement:* Secured at the sternum to detect upper-body posture deviations.
- *Working Principle:*
 - **Accelerometer:** Estimates gravity-referenced angles.
 - **Gyroscope:** Captures dynamic rotational movements.
 - **Magnetometer:** Compensates for directional drift.

Figure17: illustrating the pinout diagram of

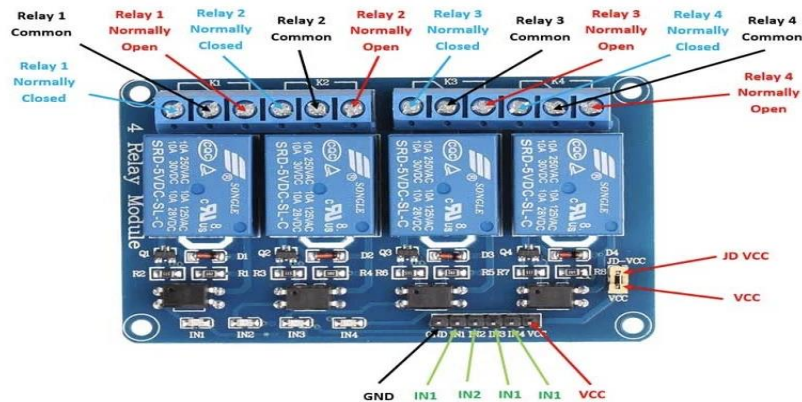


3. SRD-5VDC-SL-C Relay Module 4-channel (as shown in image 3)

- *Purpose:* Interfaces the ESP32's 3.3V logic with 12V pneumatic components.
- *Circuit Design:*
 - *Channel 1:* Activates Pump 1 (primary inflation).

- *Channel 2:* Activates Pump 2 (backup/redundancy).
- *Channel 3:* Controls the solenoid valve (deflation)

Figure 18: illustrating the pinout diagram of the 4-channel



SRD-5VDC-SL-C Relay Module

4. Pneumatic Actuators

- **12V Air Pumps (x2) (as shown in image 4)**
 - *Flow Rate:* 1.2 L/min each; synchronized for rapid inflation (~8–12 sec to full pressure).

Figure 19: showing the air pump with its dimensions



- **VT307-6G1-02 Solenoid Valve (as shown in image 5)**

- Default State: Closed; opens only when operated during pneumatic system reset to exhaust air.

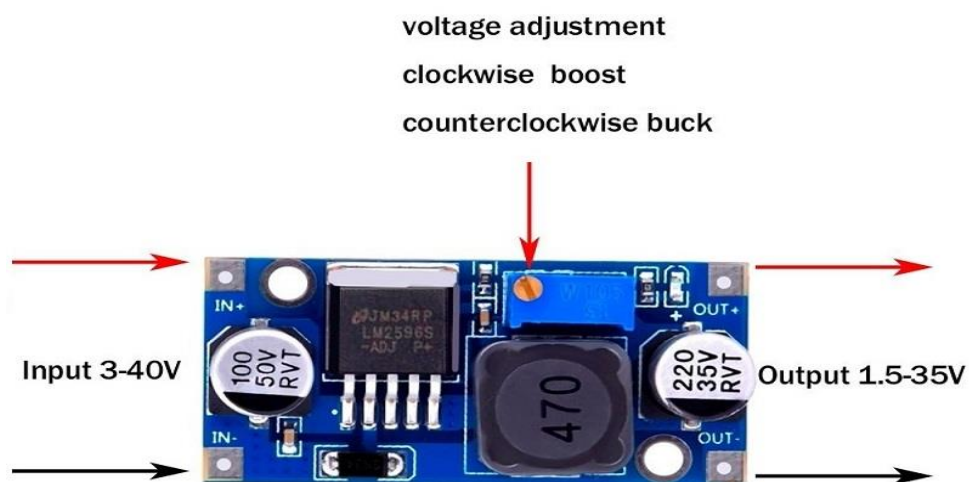
Figure 20: showing the air valve used



5. Power Management

- **LM2596S Buck Converter (as shown in figure21)**
 - Steps down the 12V battery supply to 5V for the ESP32 (efficiency: 92%).

Figure21: showing the pinout of the LM2596s Used



- **3S2P Li-ion Battery (12V, 5000mAh) (as shown in figure22)**

- Provides ~4–6 hours of continuous pump operation and about 12-15 hours of average use on single charge.

Figure22: showing the power source of the wearable vest



6. AMS1117-3.3V Linear Regulator (as shown in figure23)

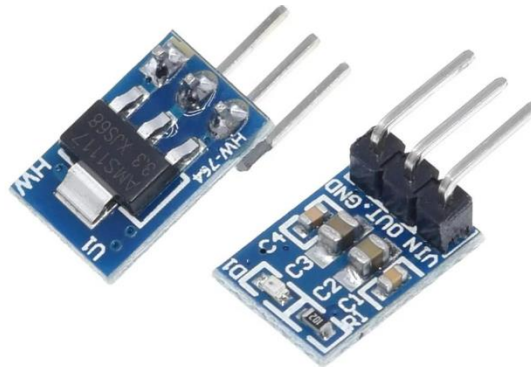
- Purpose: Dedicated power supply for the 8x RCD1234 vibration motors (3.3V, 100mA each).
- Placement: Adjacent to the ESP32, fed by the LM2596S's 5V output.
- Design Rationale:
 - Prevents voltage drops when all motors activate simultaneously.
 - Isolates motor noise from sensitive IMU/FSR circuits.

Specifications:

Input: 5V (from LM2596S).

Output: 3.3V, 1A max (supports 8 motors at ~800mA total).

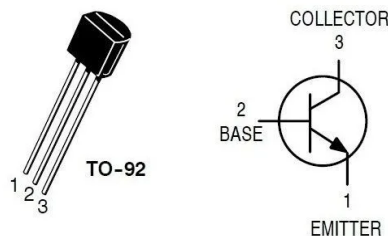
Figure23: showing the AMS1117 module used to operate the eight vibration motors in stability



7. 4x 2N2222 Transistor (as shown in figure24)

- Current Amplification
 - Allow low-current ESP32 GPIO pins (20mA max) to control high-current vibration motors (100mA+ each).
- Electrical Isolation
 - Protects the ESP32 from voltage spikes/back-EMF generated by motors (acts as a buffer).
- PWM Compatibility
 - Enables precise vibration intensity control through pulse-width modulation from the ESP32.

Figure24: showing the pinout of the 2N222 transistor



❖ First Back Layer: Haptic Feedback Network

Function:

Delivers tactile alerts to prompt posture correction, leveraging spatially distributed vibration motors.

Implementation Details:

- **8x RCD1234 Coin Motors (as shown in figure25)**
 - *Placement:*
 - **Upper Back (T1–T6):** 4 motors for scapular retraction alerts.
 - **Lumbar (L1–L5):** 4 motors for slouching warnings.
 - *Control Scheme:*

- **PWM-Driven:** continuous vibration wave when threshold exceed in either pitch or roll scales in posture deviation severity.

Figure25: showing the vibration motor used in the feedback layer



❖ Second Back Layer: Adaptive Air Chamber

Function:

A dynamically adjustable pneumatic support structure that conforms to spinal curvature while adapting to various seating surfaces.

Design & Mechanics:

- **Material Composition:**
 - High-Density Polyethylene (HDPE) reinforcement grid (Shore D 65, 0.8mm thickness)
 - Woven polyester fabric substrate (420D count, 2000mm water column resistance)
 - Thermally bonded seams (peel strength >15N/cm)
- **Pressure Management System:**
 - Closed-loop PID control maintains 1.5 ± 0.2 kPa working pressure
 - Mechanical failsafe engages at 20 kPa (ASME BPVC Section VIII compliant)
 - Hysteresis band of 0.3 kPa prevents pump cycling during micro-adjustments

The chamber's anisotropic expansion properties ensure optimal force distribution while maintaining unrestricted thoracic mobility during posture transitions.

❖ Third Back Layer: Pressure Sensing Matrix

Function:

Monitors user-chair contact pressure to guide air chamber adjustments.

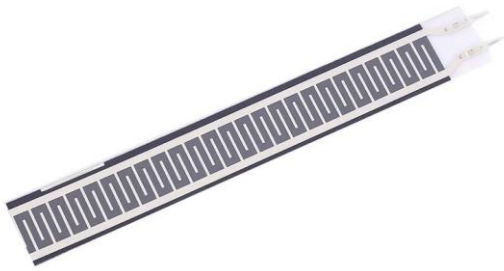
Component Specifications:

- **4x FSR Sensors (as shown in figure26)**
 - *Placement:*
 - **Thoracic Region at (T4, T6, T9, AND T11)**
 - *Calibration:*
 - Baseline resistance measured during upright sitting.

figure26: showing the type of FSR sensor used in the third layer

FSR Force Sensor

110mm Resistive Flex Film Pressure Sensor



❖ Adaptive Air Chamber Dynamics and Pressure Balancing Mechanism

The vest's smart pneumatic system operates through a carefully orchestrated sequence of sensor inputs and mechanical responses to deliver personalized lumbar support:

1. Posture Detection Phase

- The MPU9250 IMU mounted at T4 continuously monitors spinal orientation
- When forward flexion exceeds 10° for >15 seconds, the system:
 - Activates upper back vibration motors (T1-T12) for haptic alerting
 - Begins scanning FSR sensors at T4, T6, T9, and T11 for chair contact patterns

2. Chair Profiling & Initial Inflation

- Upon first sitting (upright posture <10° flexion):
 - The FSR array creates a pressure map of chair contact points

3. Dynamic Pressure Regulation

- The dual 12V pumps engage in distinct operational modes:
 - Synchronized mode: Both pumps inflate simultaneously for hard chairs (0→1.5kPa in 8s)

4. Support Maintenance

- During sustained sitting:
 - T4 FSR acts as an alarm trigger
 - IMU overrides if sudden posture changes occur

5. Safety & Efficiency Features

- Pressure relief valve
- Deflation after resetting the pneumatic system

❖ Schematic Overview: Posture Vest Control PCB

This schematic depicts the custom PCB integrating all core control components for the posture-correcting vest system. The board features the following key interconnections:

1. Sensor & Peripheral Connections

- **MPU9250 IMU:** Connected via I2C (SDA=21, SCL=22) with 4.7kΩ pull-up resistors
- **FSR Array:** Analog inputs on GPIOs 36, 39, 34, 35 (ADC1_CH0, ADC1_CH3, ADC1_CH6, ADC1_CH7)
- **User Controls:**
 - Tactile buttons on GPIOs 14 (Right push-button) and 12 (Left push-button) with 10kΩ pull-down resistors

2. Actuator Drive Circuits

- **Vibration Motors:**
 - Driven through 2N2222 NPN transistors (x4) on GPIOs 25, 26, 27, 13
 - Base resistors: 1kΩ
- **Pneumatic Control:**
 - Relay module connected to GPIOs 17 (Pump1), 16 (Pump2), 4 (Valve)
 - TXS0108E level converter interfaces 3.3V ESP32 signals to 5V relay inputs

3. Power Distribution

- **AMS1117-3.3V:** Supplies vibration motors (input from LM2596S 5V rail)

Below are screenshots showing the complete schematic with these interconnections, highlighting

the critical signal paths between the ESP32 and peripheral components.

Figure27: shows interconnections in the pcb

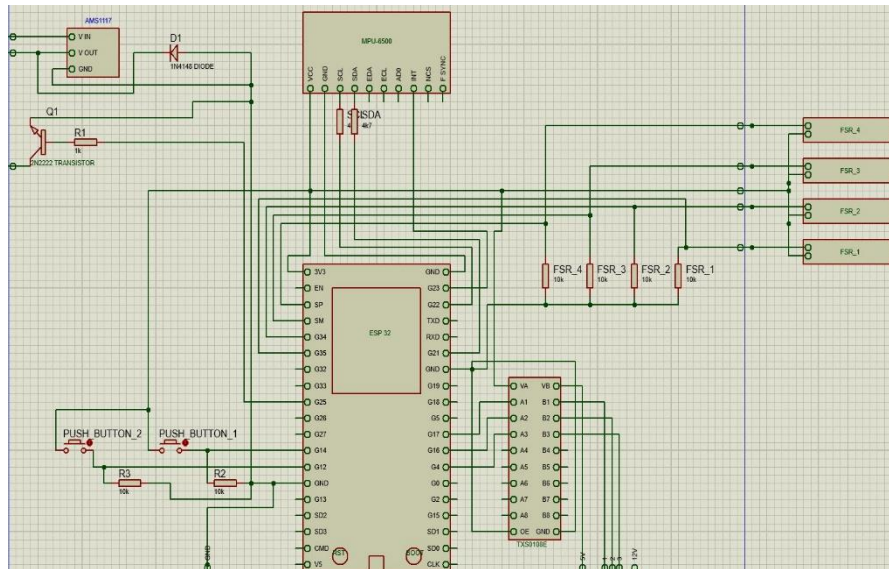


Figure28: shows the PCB components

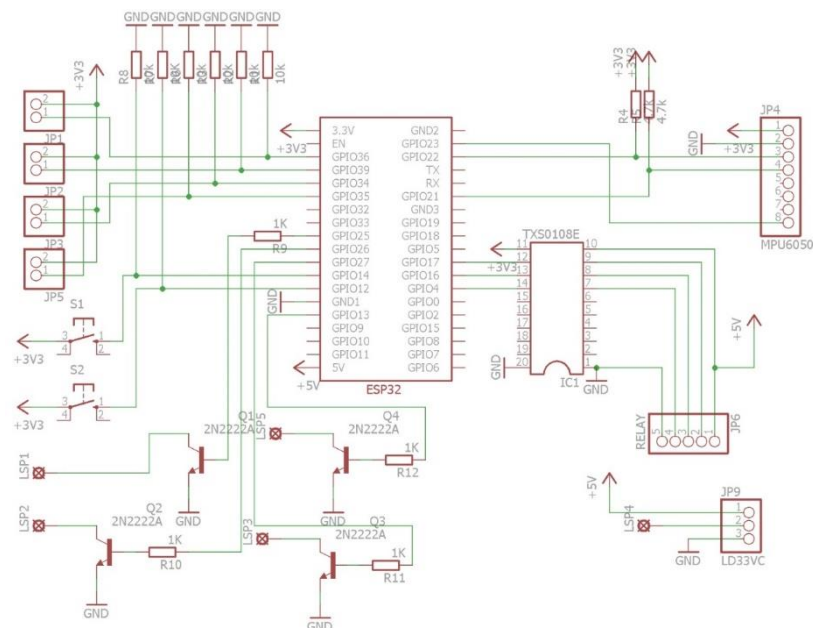
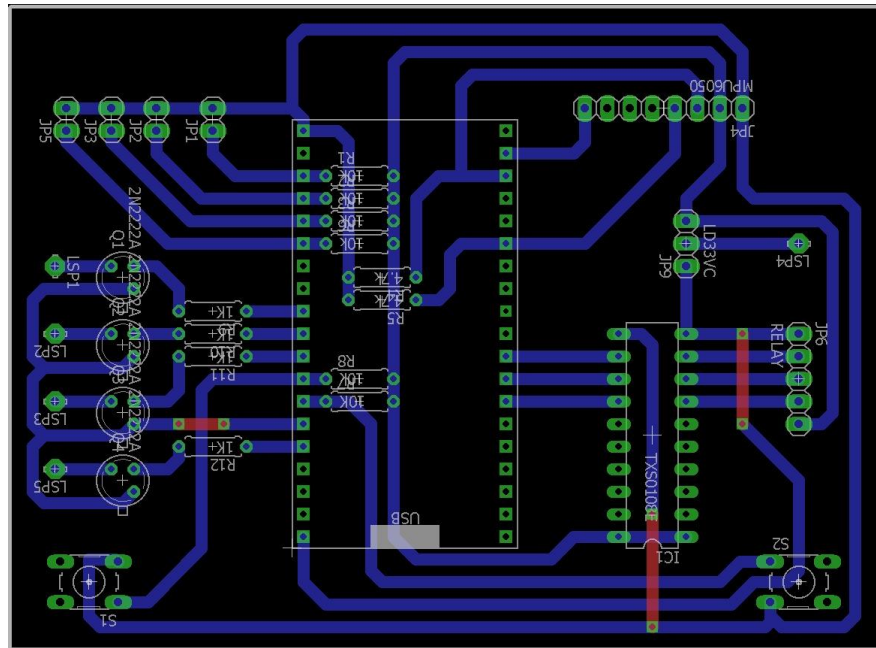


Figure29: shows the PCB design and components placement



3.3 Mobile Application Design

This section outlines the design, architecture, and functional features of the SitX mobile application, which serves as the user interface for real-time posture monitoring and data analysis.

App Overview

The SitX mobile application is developed using Flutter, leveraging GetX for state management and navigation. The app provides users with real-time feedback on their sitting posture by receiving sensor data from a wearable vest via a backend Node.js server and visualizing this data effectively.

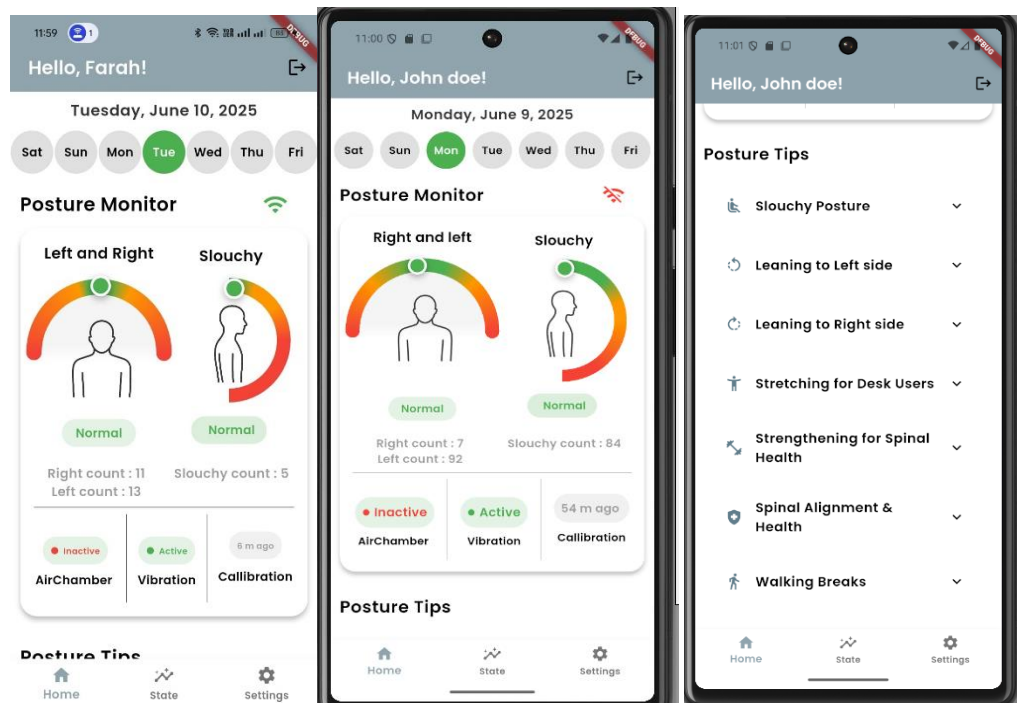
Wireframes and Key Screens

Key screens were designed to ensure clarity and usability. Wireframes/mockups were first created to visualize the following:

- **Home Screen**

It displays the following:

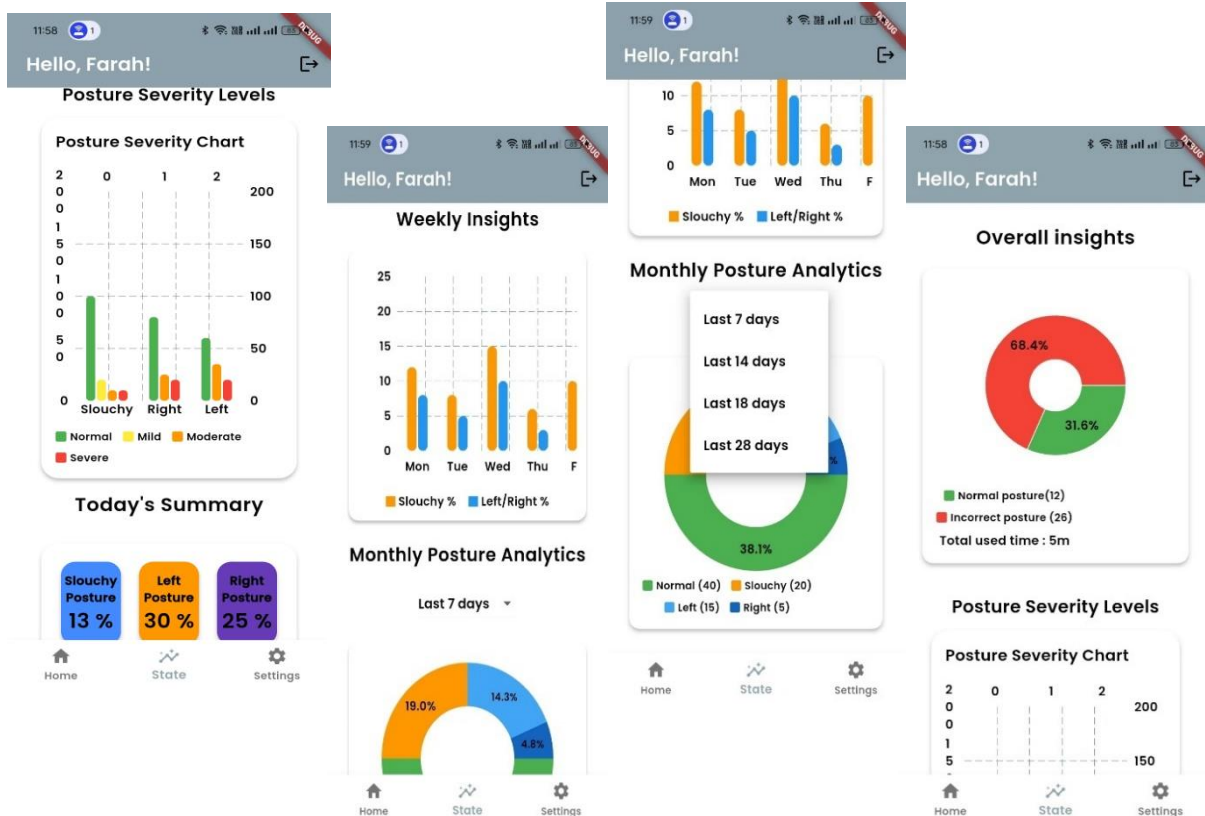
- Right and left severity stages (Normal ,Moderate, Severe)
- Slouchy severity stages (Normal, Mild, Moderate, Severe)
- Right / left / slouchy posture counters.
- Activation of air chamber .
- Activation of vibration.
- Last calibration time.
- Posture tips.
- Wi-Fi connection.



- **Stats Page**

It shows the following:

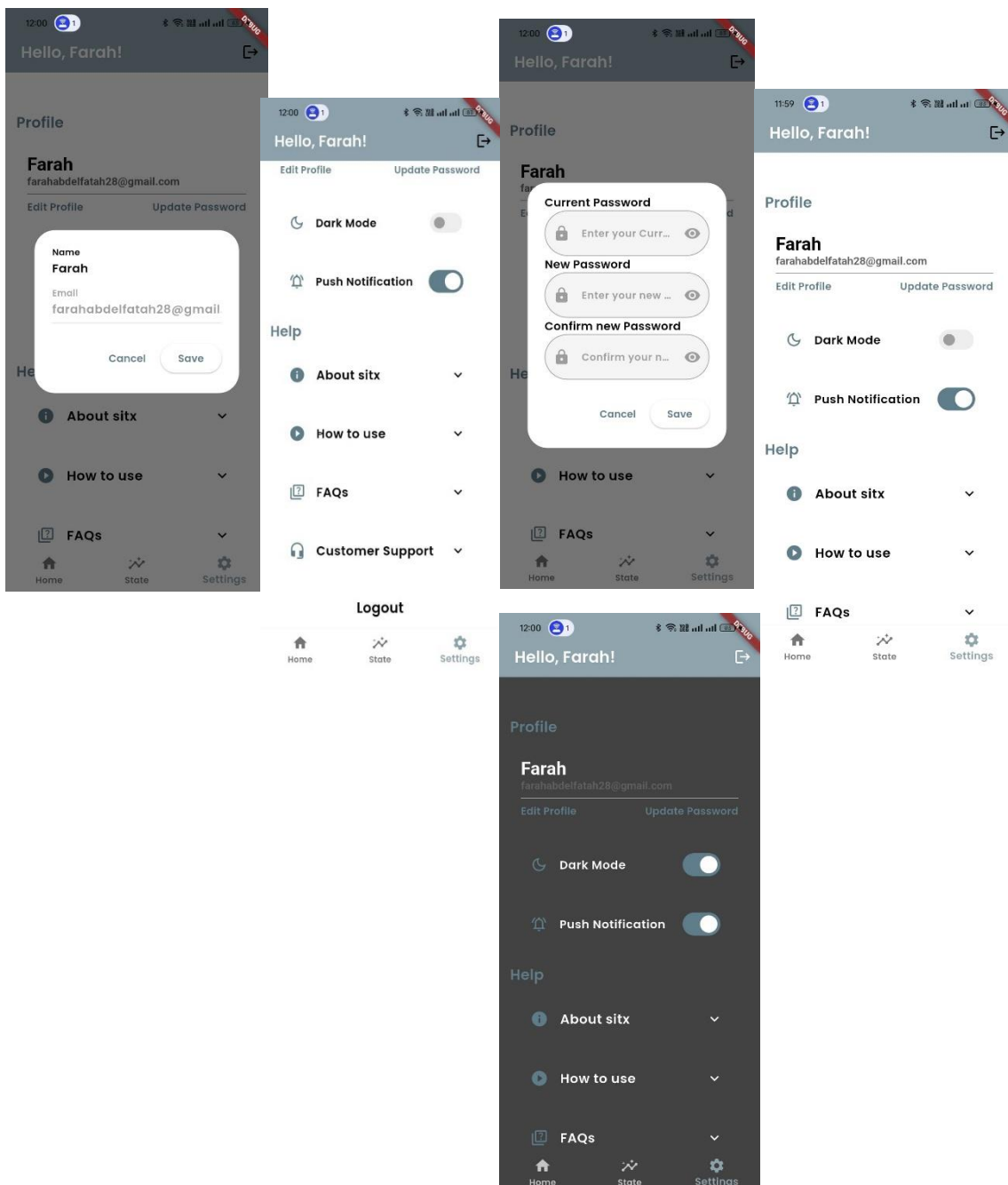
- Overall insights between correct and incorrect posture presented by pie chart.
- Bar graph representing severity stages counters for slouchy/left/right posture.
- Today's summary represents the overall slouchy , right and left posture percentages .
- Weekly insights represent slouchy , right and left counters over the week .(future work)
- Monthly insights represent normal , slouchy , left, right and air chamber counters over the last 7days , 14 days , 18 days and 28 days.(future work)



- **Settings Screen**

Allows users to :

- Update name and password
- Switch between Light and dark theme.
- Push notification(Future work).
- Have a brief about Sitx project
- Instructions on how to use it.
- FAQs section.
- Customer support section.
- Logout.



App Features

1. Real-Time Posture Monitoring

- The app continuously receives live data from the wearable vest through the backend RESTful API.
- Posture values like slouchy and left/right lean, vibration , air chamber activations and last calibration time are updated in real-time on the dashboard using **Obx()** widgets.

2. Slouchy Posture Detection

- If slouching posture is detected (based on accelerometer thresholds), the app provides immediate feedback via visual cues (e.g., progress scale turning based on the stage of severity, green for normal , orange for mild and moderate and red for severe) .
- Slouchy posture counter

3. Left and Right Posture Detection

- If Left and Right posture is detected (based on accelerometer thresholds), the app provides immediate feedback via visual cues (e.g., progress scale turning based on the stage of the severity, green for normal , orange for moderate and red for severe) .
- Right posture counter.
- Left posture counter.

3. Data Visualization and Tracking

- A pie chart represents overall correct/incorrect posture in addition to the total used time .
- Bar graph represents the counters of severity for slouchy , left and right postures stages.
- Today's summary dashboard for slouchy, left and right postures percentages.
- Weekly insights represented by bar graph visualizing slouchy, left and right counters.
- Pie chart shows posture stats starts from 7days to 28days , over selectable time ranges (7, 14, 28 days).

4. User Profiles and Settings

- Users can sign up, log in and edit profile details (username, update password) real time.
- Switching between dark and light themes.

Flutter + GetX Implementation

GetX is used for:

- Reactive UI (Obx , .value, Obs ,Rx)
- Routing (Get.toNamed())
- Controller management for posture data and state updates

State separation:

- **Home Controller** manages variables status and Api communication.
- **Home screen** for UI components
- Real-time UI components are automatically updated based on observable variables (e.g., `RxInt`, `RxDouble`, `RxString`).
- **StateController** present accumulated variables through the dat that is presented it in the state screen
- **State screen** , data is accunlated through the day , stored in MongoDB and fetches through the /sensorHistory endpoint

Communication with Backend

- The app communicates with the backend via HTTP and WebSocket .

Data flow:

1. The vest sends sensor readings to the Node.js backend via Mqtt broker.
2. The backend stores these readings in MongoDB .
3. The Flutter app fetches the latest data every 3 seconds using REST APIs like
`/api/sensorData` .
4. Upon receiving new data, GetX observables update the UI accordingly

Conclusion

The mobile app is designed to be intuitive and responsive. By combining real-time feedback, data analysis, and a wearable-integrated backend, the app empowers users to improve their posture and prevent long-term spinal issues.

3.4 Backend System Architecture (Node.js, REST API)

Overview

The backend system is designed as a modular, scalable, and secure RESTful API built with Node.js, Express.js, and MongoDB. It handles user management, product, shopping cart, order processing, authentication, and analytics, forming the core logic behind the wearable posture-correcting vest application and its e-commerce interface.

Technologies Used

Node.js	JavaScript runtime environment
Express.js	Web framework for building APIs
MongoDB	NoSQL database for storing application data
Mongoose	ODM for defining models and handling MongoDB operations
JWT	Token-based authentication
bcryptjs	Password hashing and security
Multer	Handling file uploads
Passport.js	Authentication (including Google OAuth2)
dotenv	Managing environment variables

System Layers

1. Routing Layer

- All API routes are defined inside the /routes directory.
- Each route is responsible for handling specific resource types like users, products, authentication, and admin operations.
- Follows REST conventions:
 - GET – fetch resources
 - POST – create resources
 - PUT – update resources
 - DELETE – remove resources

2. Controller Layer

- Business logic is handled in controller functions (inline within routes or externalized).
- Controllers process incoming requests, interact with models, and send structured JSON responses.

3. Model Layer

- Defined in the /models directory using Mongoose.
- Includes schemas for User, Product, Cart, Order, Experience, etc.
- Provides data validation, relationship mapping, and query abstraction.

4. Middleware Layer

- Authentication: Checks JWT tokens, admin roles.
- File Upload: Uses Multer to handle user uploads (e.g., product images).
- Error Handling: Centralized error responses for robustness.

5. Persistence Layer (Database)

- MongoDB stores all application data.
- Collections for users, products, guest carts, orders, verification tokens, and analytics.

Authentication and Security

- User authentication is handled via JWT.
 - Google OAuth2 login is implemented with Passport.js.
 - Passwords are hashed using bcryptjs before storage.
 - Protected routes use middleware to validate tokens and restrict access by role.
-

API Request-Response Lifecycle

[Client Request]



[Express Route Handler]



[Middleware: JWT Validation]



[Controller Logic]



[Mongoose Model Interaction with MongoDB]



[Response Sent to Client]

REST API Conventions

- **Base URL:** /
 - All endpoints are grouped by feature:
 - /auth: Authentication (login, register, Google OAuth)
 - /cart: Cart operations
 - /admin: Admin-specific operations (e.g., create product)
 - /guest-cart: Guest cart management
 - Data is exchanged in **JSON format**.
 - Uses status codes: 200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 500 Server Error.
-

Scalability and Maintainability

- **Modular Code Structure:** Clear separation of routes, models, controllers, and middleware.
- **Flexible Authentication:** JWT allows stateless and secure authorization.
- **Database-agnostic logic:** Easily migratable to other NoSQL/SQL backends if needed.

- **Plug-and-play Features:** Future modules (notifications, analytics, etc.) can be added with minimal refactoring.
-

Summary

The backend system is a robust and modular RESTful API architecture using the Node.js ecosystem. It supports:

- Full authentication & authorization flow
- Cart and order operations
- Scalable database interaction
- Secure and structured data exchange
- Admin panel functionality
- Real-time analytics via visit tracking

Describe how data is retrieved via the API:

Module Overview

This module is responsible for managing and retrieving sensor data collected from the vest. The data is recorded and stored in a MongoDB collection using Mongoose and is made accessible through a REST API endpoint built with Express.js.

Model: SitxSensor

The SitxSensor model defines the structure for storing sensor data coming from the wearable vest. Each document represents one set of sensor readings.

API Route: GET /sensorData

Purpose

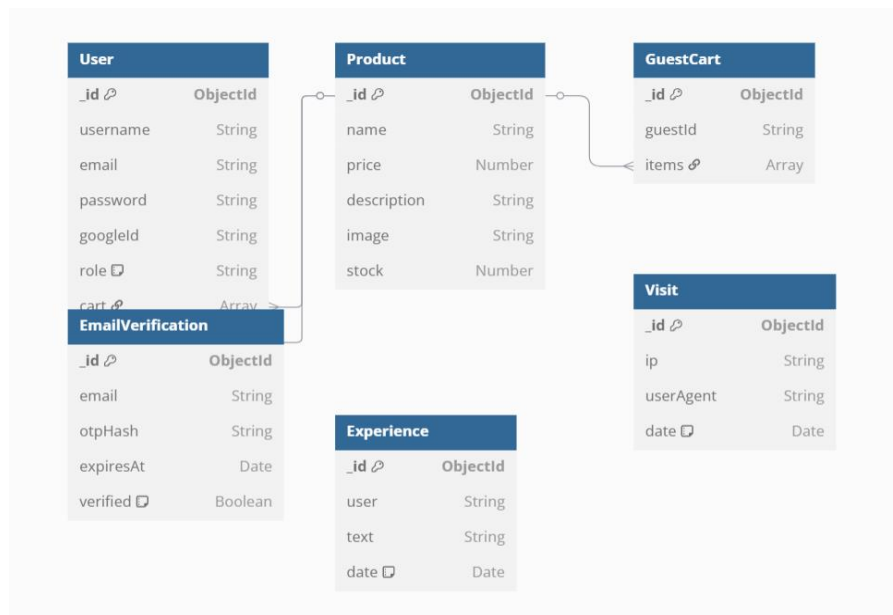
Retrieves the latest sensor data entry from the database.

Functionality Breakdown

- Uses SitxSensor.findOne() to fetch a single document.
 - Sorted by receivedAt in descending order to return the most recent.
 - Handles 404 if no data exists.
 - Responds with 500 if there is a server/database error.
-

Benefits of Current Setup

- Efficient retrieval using `findOne().sort()` for real-time data display.
- Scalable schema .
- Ready integration for WebSocket-based updates via Socket.IO.
- Clean error handling with HTTP status codes and messages.
- database schema diagrams:



Http and WebSocket packages are used in communicating between the Sitx app and backend API .

`/sensorHistory` :fetching real time data by WebSocket`

```
void initSocketConnection() {
```

```

socket = IO.io('http://10.0.2.2:8080', <String, dynamic>{
  'transports': ['websocket'],
  'autoConnect': true,
});

socket.onConnect((_) {
  print('🟢 Connected to WebSocket');
});

socket.on('sensorHistory', (data) {

```

login`by http

```

static Future<String?> isAccountExists(String email, String password) async {
  final url = Uri.parse('$baseUrl/login');

  try {
    final response = await http.post(
      url,
      headers: {'Content-Type': 'application/json'},
      body: jsonEncode({'email': email, 'password': password}),
    );
    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      final username = data['user']['username'];

      print("✅ Login successful. Username: $username");
      return username; // Return the username
    } else {
      print("❌ Login failed: ${response.statusCode} - ${response.body}");
      return null;
    }
  } catch (e) {
    print("❌ Error logging in: $e");
    return null;
  }
}

```

`/ forget-password`by http

```

static Future<bool> updatePassword(String email, String password) async {
  final url = Uri.parse('$baseUrl/forget-password');

  try {
    final response = await http.put(

```

```

    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({ 'email': email, 'password': password }),
  );

  if (response.statusCode == 200) {
    print("✅ Password updated successfully!");
    return true;
  } else if (response.statusCode == 404) {
    print("❌ User not found. Cannot update password.");
    return false;
  } else {
    print("❌ Failed to update password: ${response.body}");
    return false;
  }
} catch (e) {
  print("❌ Error updating password: $e");
  return false;
}
}

```

3.5 Web Application Design:

-
- **Features:** Browse Products, Add to Cart, Checkout

1. Browse Products

Components Involved:

- components/Buy.js
- components/Footer.js, Navbar.js (*UI layout*)

Functionality:

- Fetches a list of products from the backend API using GET /products.
- Displays each product's:
 - Image
 - Name
 - Description

- Price

Backend Interaction:

- axios.get('/products') to load the products.

2. Add to Cart

Components Involved:

- components/Cart.js
- components/CartContext.js – **Shared context** that stores cart state
- components/Buy.js – Provides "Add to Cart" button

Functionality:

- Users can click “Add to Cart” on a product in the listing.
- CartContext stores product ID and quantity.
- State persists in local Storage for users.

Backend Interaction:

- If logged in: uses POST /add to store in MongoDB in user collection.
- If guest: uses POST /add to store in MongoDB in guest cart collection.

3. Checkout

Components Involved:

- components/ConfirmOrder.js
- components/Cart.js
- components/CartContext.js

Functionality:

- On visiting the Cart:
 - Shows added items, prices, quantity adjustments, and remove buttons.
 - Total price is calculated dynamically.
- On confirming:
 - Prompts user for delivery info (name, address, phone).
 - Sends a structured order to the backend API.

Backend Interaction:

- POST /confirm to create an order.
- Stores order inside the user's orders[] array in MongoDB if user is logged in otherwise, order is stored in guest cart.

State Management

- Uses **React Context API** in CartContext.js to manage cart across components.
- Handles both **guest** and **authenticated** user cart handling.

UX & UI Features

- Products displayed in grid layout.
- Cart accessible from any page via navbar/cart icon.
- Real-time cart updates and quantity changes.
- Visual feedback on successful order placement.

- **Features: Admin Dashboard, Product Management, Order Overview**

Overview

The Admin Panel allows designated admin users to manage the website's store functionality. Admin access is usually gated via user roles (stored in MongoDB). Only users with role: "admin" can access these features.

Admin Access Control

Files:

- components/Admin.js
- components/AddProduct.js
- components/ViewProduct.js

Role-Based Control:

- Admin users are identified via JWT login session .

Admin Dashboard

Component:

1- Admin.js

Features:

- Display admin-specific actions:
 - View products
 - Add product
 - Edit product
 - Delete product
 - View orders

Internal State:

- Likely uses React useState and useEffect to fetch and display:
 - All products (GET /products)
 - All orders (GET / dashboard-stats)
-

Add Product**Component:****1- AddProduct.js****Features:**

- Form for entering:
 - name, price, description, image, stock
- Sends data to backend:

```
const res = await axios.post("http://localhost:5000/admin/products", data, {  
  headers: { "Content-Type": "multipart/form-data" },  
  withCredentials: true,  
});
```

Backend Security:

- Validates if the user is an admin (via adminMiddleware).
 - Uses a Mongoose model (Product) to save the data.
-

Edit/Delete Product**Inferred Components:**

- Part of viewProducts.

Edit:

- Pre-fills the form with current product data.
- Updates using:

```
const response = await axios.put(
  `http://localhost:5000/admin/products/${editingProductId}`,
  formData,
  {
    withCredentials: true,
    headers: {
      "Content-Type": "multipart/form-data",
    },
  }
)
```

View All Products (as Admin)

Component:

1- Admin.js

Features:

- Displays all products with:
 - Name
 - Price
 - Stock
 - Edit buttons

View Orders (Admin)

Component:

1- (Admin.js)

Features:

- Fetches last orders in past 10 days from:

```
const response = await axios.get("http://localhost:5000/admin/dashboard-stats", {
  withCredentials: true,
});
```

- Displays order details:

- User
- Items
- Quantities
- Total
- Date
- Address

Admin UI Features

- **Tables** for product overview.
- **Forms** for adding/editing products.
- Clean admin layout for clarity and control.

State Management for Admin

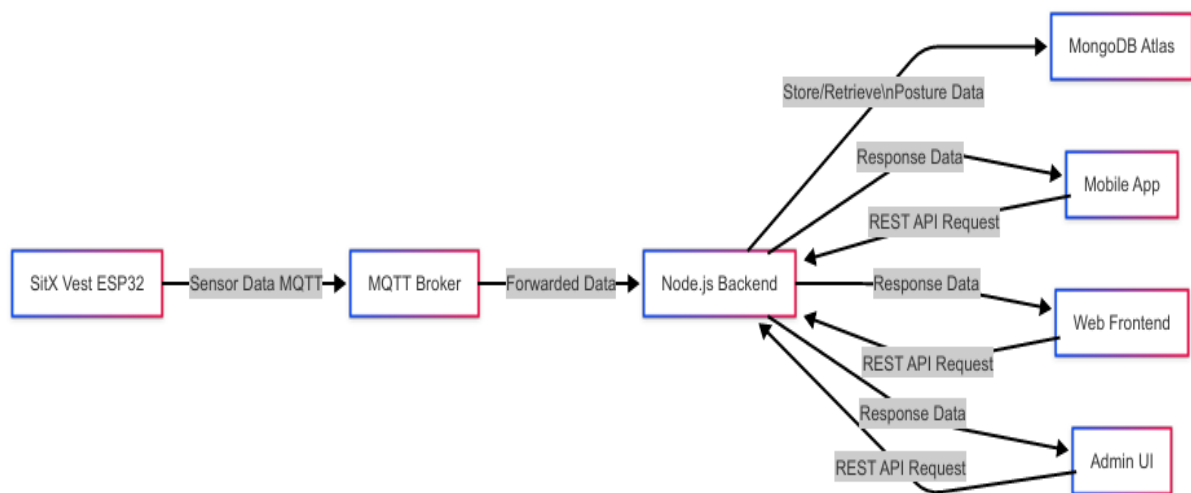
- Uses Context to manage current admin session.
- Axios calls authenticated using JWT from context.
- Dynamic product list updating after CRUD actions.

Security & Best Practices

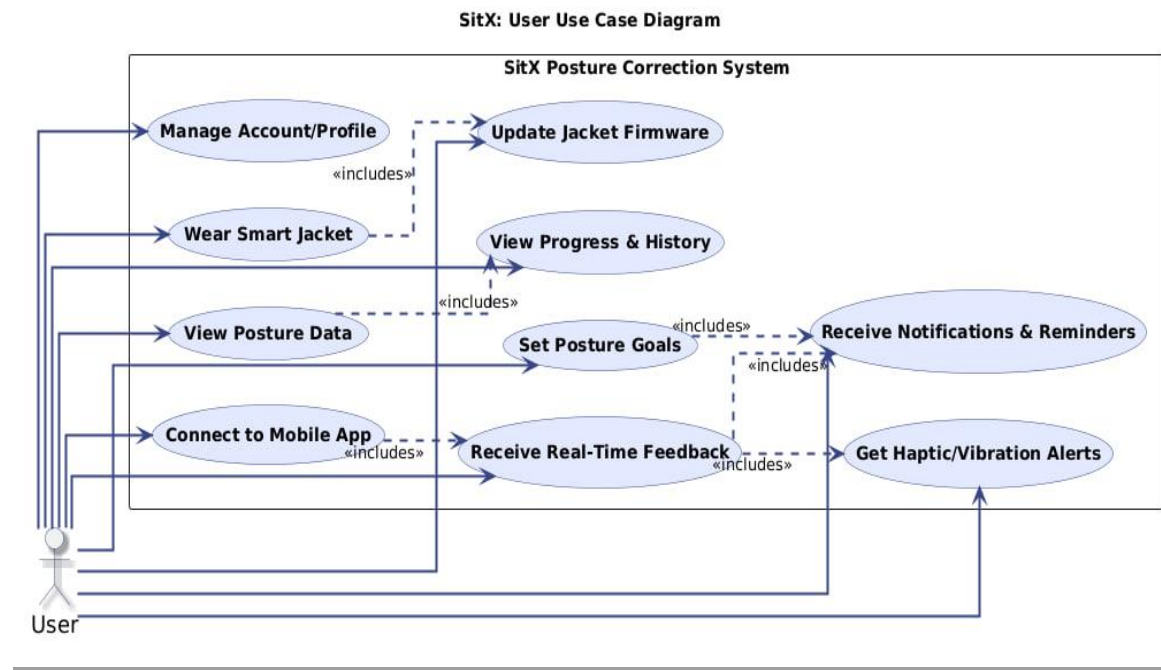
- Admin actions must always be validated server-side.
- All admin routes protected on backend with:
- Avoid exposing admin controls to UI if user is not admin.

```
if (user && user.role === "admin") {  
  next();  
} else {  
  res.status(403).json({ error: "Access denied" });  
}
```

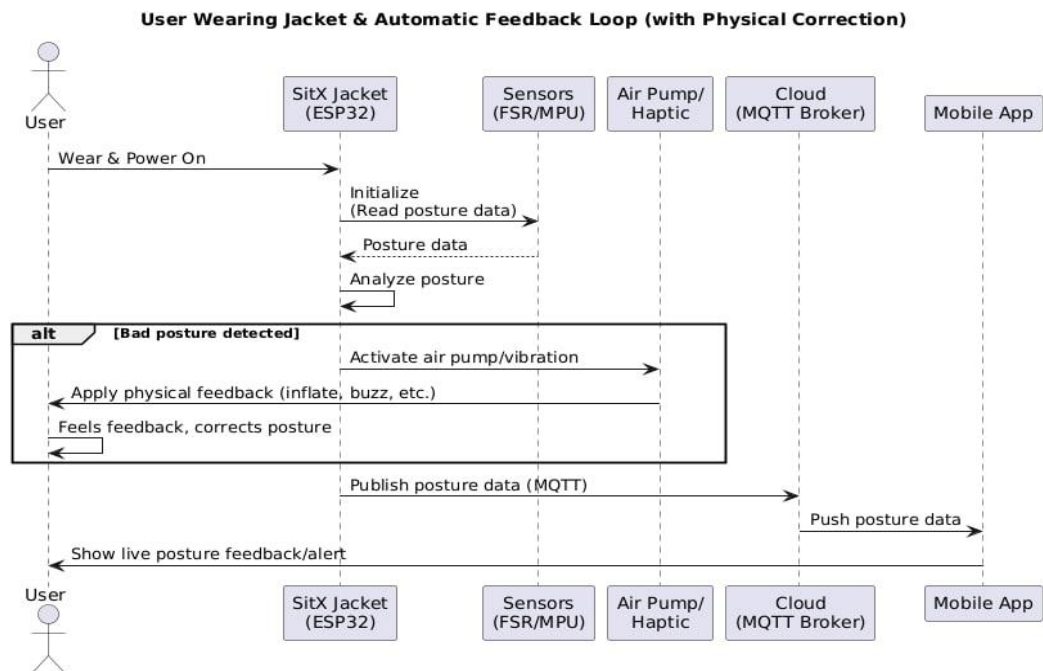
3.6 Data Flow Diagrams:



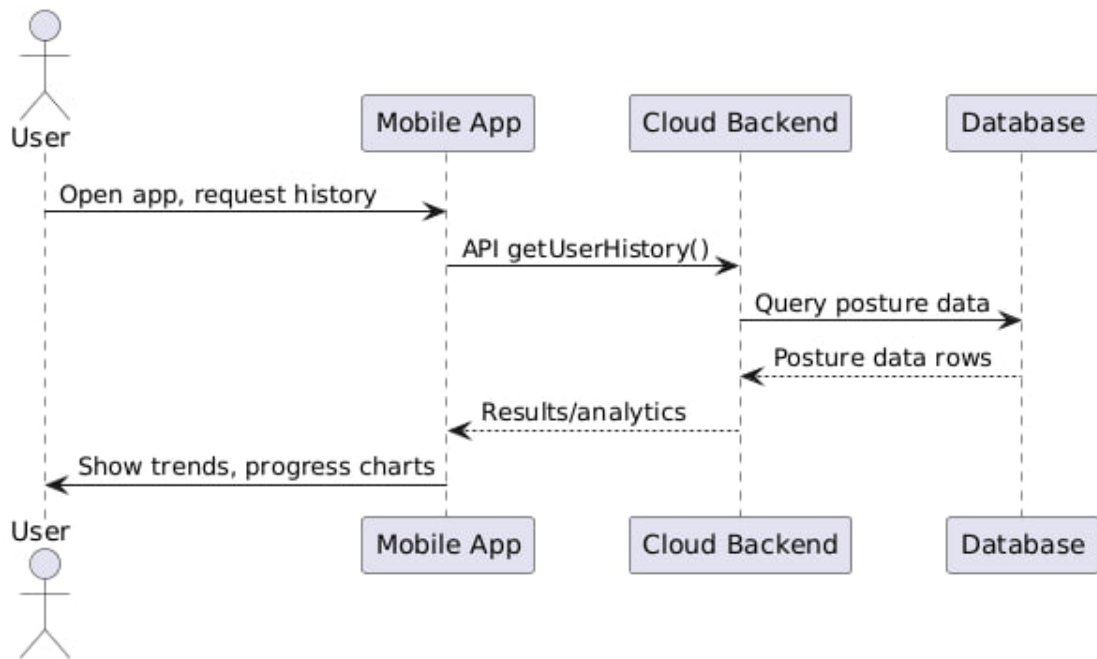
3.7 Use Case Diagrams:



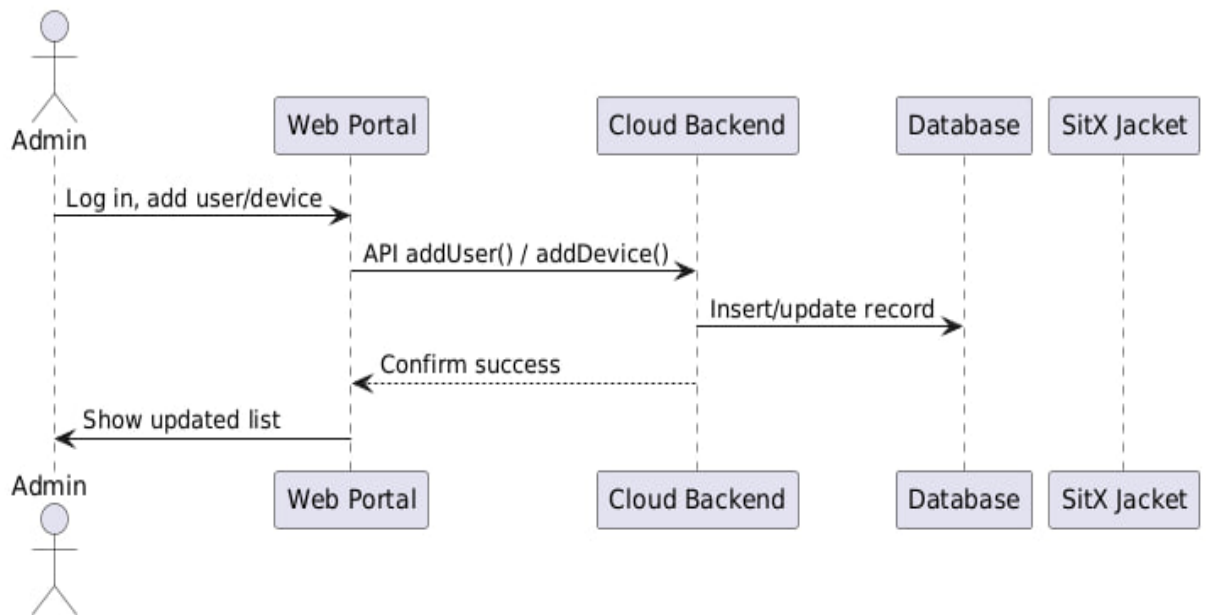
3.8 Sequence Diagrams:



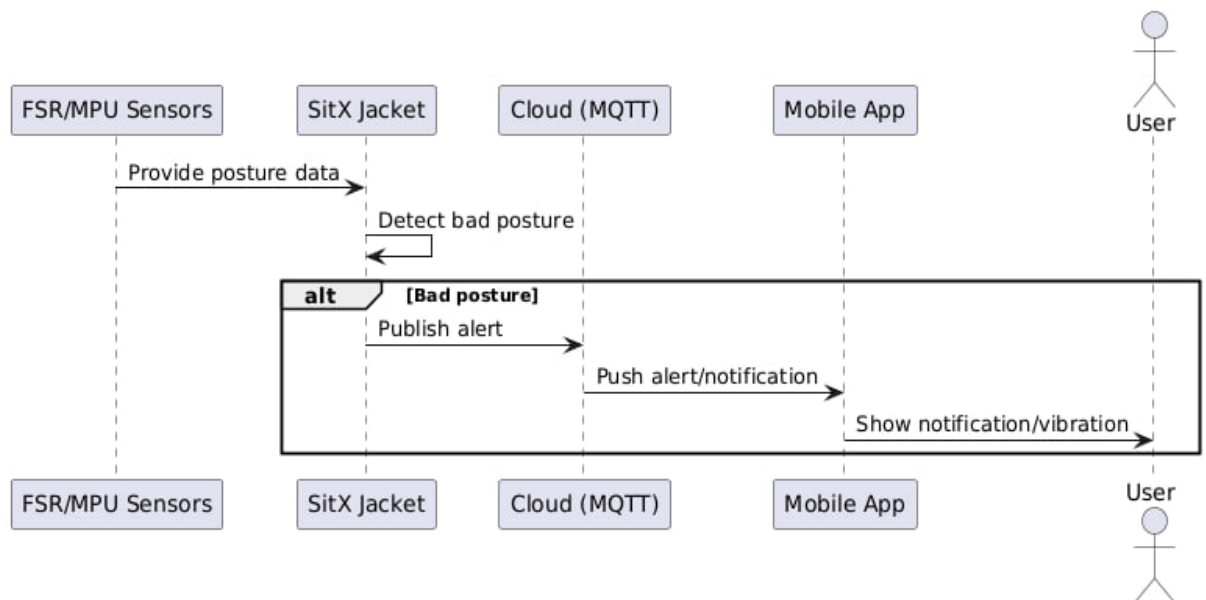
User Opens App to View History/Analytics



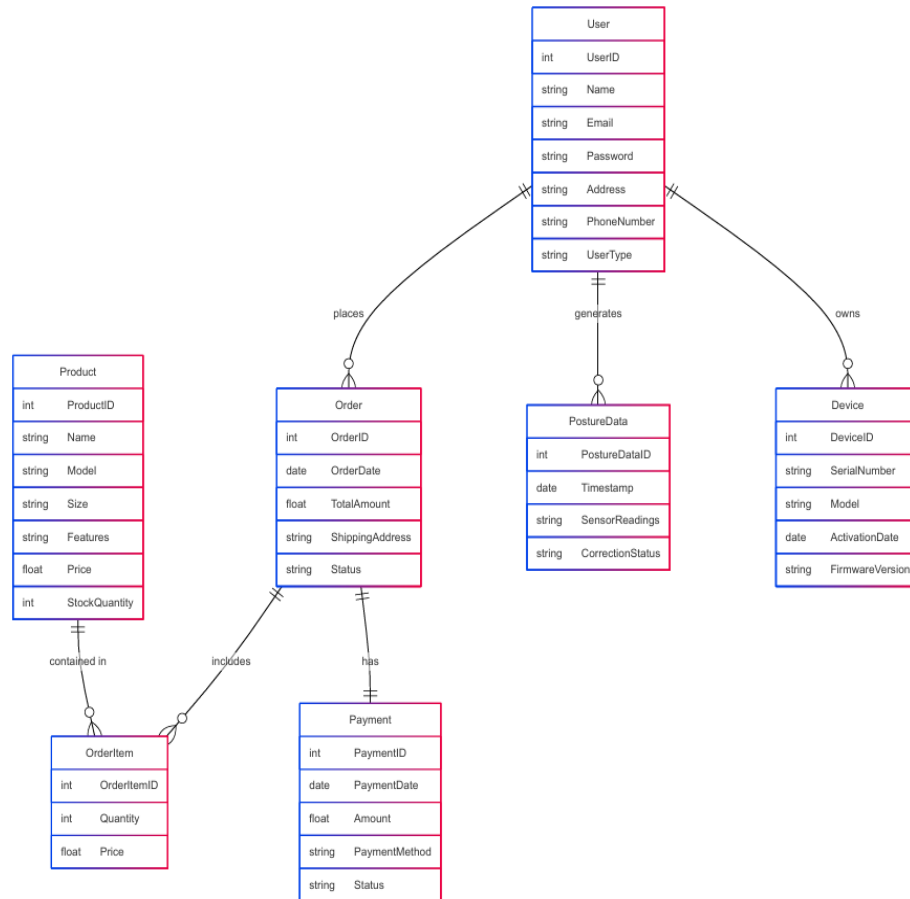
Admin Adds/Manages User or Device



Real-Time Posture Alert to User



3.9 ERD (Entity Relationship Diagram):



Chapter 4: Implementation



4.1 Development Environment:

The SitX system was built and tested using the following hardware and software stack.

- **Operating systems**

- Windows 11 Pro (mobile-app builds and general testing)
- macOS Ventura (iOS builds and React web development)
- Ubuntu 22.04 LTS (server-side Node.js services and CI runners)

- **IDEs / Editors**

- Android Studio Flamingo + Flutter plugin
 - Visual Studio Code (with Dart, JavaScript/TypeScript, and PlatformIO extensions)
 - MongoDB Compass for ad-hoc database queries.
-

- **Programming languages & frameworks**

- Dart 3 + Flutter 3.19 (cross-platform mobile app)
 - JavaScript (ES 2022) + Node.js 20 (backend micro-services)
 - React 18 (marketing and admin web portal)
 - C++ (ESP32 firmware under FreeRTOS)
-

- **Database system**

- MongoDB Atlas (serverless tier, multi-region clusters)

- **Version-control & collaboration**

- Git (GitHub private repository with branch protection and GitHub Actions for CI/CD)
 - GitHub Projects for kanban task tracking and release planning
-

- **Additional toolchain**

- PlatformIO CLI for ESP32 firmware compilation and OTA packaging
- Mosquitto MQTT broker (TLS enabled) for local integration tests
- Postman + Insomnia for REST and WebSocket API testing
- Stripe CLI for e-commerce webhook verification

This environment provides full coverage from embedded firmware through mobile and web front ends to cloud deployment, ensuring consistent builds and streamlined collaboration across the entire SitX development team.

4.2 Wearable Implementation:

Sensor Integration and Data Acquisition:

- **Sensors:** The SitX jacket uses four Force Sensitive Resistors (FSRs) to measure pressure distribution and an MPU9250 sensor to detect angular displacement (pitch and roll) for posture analysis.
- **Data Processing:**

```
int fsrReadings[4];
for (int i = 0; i < 4; i++) {
    fsrReadings[i] = analogRead(fsrPins[i]);
}
```

- **Explanation:** Reads analog values from each of the FSRs, which help determine pressure distribution.

```
mySensor.accelUpdate();
float ax = mySensor.accelX();
float ay = mySensor.accelY();
```

```

    float az = mySensor.accelZ();
    float pitch = atan2(az, sqrt(ax * ax + ay * ay)) * 180.0 / PI -
calibration.refPitch;
    float roll = atan2(ay, sqrt(ax * ax + az * az)) * 180.0 / PI -
calibration.refRoll;

```

- **Explanation:** Calculates the pitch and roll from accelerometer data to assess posture.

MQTT Publishing:

- **Wi-Fi setup**

```

void setupWiFi() {
    WiFi.disconnect(); // Ensure clean start
    Serial.print("Connecting to WiFi...");

    while (WiFi.status() != WL_CONNECTED) {
        WiFi.begin(ssids[current_wifi_index],
passwords[current_wifi_index]);
        unsigned long startAttemptTime = millis();

        while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime
< 10000) {
            delay(500);
            Serial.print(".");
        }

        if (WiFi.status() != WL_CONNECTED) {
            Serial.println("\nFailed to connect to current WiFi.
Switching...");
            current_wifi_index = (current_wifi_index + 1) % 2; // Switch to
the next WiFi network
        } else {
            states.wifiConnected = true;
            Serial.println("\nWiFi connected");
            Serial.println("IP address: ");
            Serial.println(WiFi.localIP());
        }
    }
}

```

```
}  
}
```

- **Explanation:** tries more than Wi-Fi network to send the data

- **JSON Serialization and Publishing:**

```
// Serialize and publish  
char jsonBuffer[256]; // Match the document size  
serializeJson(doc, jsonBuffer);
```

- **Explanation:** Organizes sensor data into a JSON document and publishes it to an MQTT topic.

Air Chamber Control (Logic Overview):

- **Pump and Valve Management:**

```
void startPumpOperation(int duration) {  
    digitalWrite(relayPins[0], LOW);  
    digitalWrite(relayPins[1], LOW);  
    states.pumpRunning = true;  
    timeKeeper.pumpStopTime = millis() + duration;  
}  
  
void stopPumpOperation() {  
    digitalWrite(relayPins[0], HIGH);  
    digitalWrite(relayPins[1], HIGH);  
    states.pumpRunning = false;  
}  
  
void openValve() {  
    digitalWrite(relayPins[2], LOW);  
    states.valveOpen = true;  
}  
  
void closeValve() {  
    digitalWrite(relayPins[2], HIGH);  
    states.valveOpen = false;  
}
```

- **Explanation:** Manages the activation and deactivation of pumps and valves based on posture analysis. Ensures efficient and safe operations to correct posture when necessary.

- **Calibration and Response:**

```
void calibrateMPU() {
    mySensor.accelUpdate();
    calibration.refPitch = atan2(az, sqrt(ax * ax + ay * ay)) * 180.0 / PI;
    calibration.refRoll = atan2(ay, sqrt(ax * ax + az * az)) * 180.0 / PI;
    states.mpuCalibrated = true;
}
```

- **Explanation:** Sets reference pitch and roll values, allowing the system to detect deviations and trigger corrective actions like pump activation or vibration.

Button Pin Setup:

- **Right Button:** Handles calibration and valve operations.

```
void serviceRightButton() {
    static bool prev = false;
    static unsigned long pressStart = 0;
    bool current = digitalRead(pushButtonPins[0]) == HIGH;
    unsigned long now = millis();

    if (current && !prev) pressStart = now;
    if (!current && prev) {
        unsigned long held = now - pressStart;
        states.rightButtonState = held < 2000;
        if (held >= 2000) {
            states.pumpsWereRunning = false;
            if (states.valveOpen) {
                closeValve();
                states.systemLocked = false;
            } else {
                openValve();
            }
        }
    }
}
```

Short Right Button Press (< 2s): Triggers MPU calibration and opens the valve, facilitating immediate correction setup.

Long Right Button Press ($\geq 2s$): Toggles between opening and closing the valve, providing direct control over the air chamber.

- **Left Button:** Handles enabling/disabling features like vibration.

```
void serviceLeftButton() {  
  bool current = digitalRead(pushButtonPins[1]) == HIGH;  
  if(!current)return;  
  states.vibrationEnabled = !states.vibrationEnabled;  
  Serial.print("Vibration ");  
  Serial.println(states.vibrationEnabled ? "enabled" : "disabled");  
}
```

Left Button Press: Toggles the vibration feature, enabling users to control feedback preferences.

4.3 Mobile App Implementation:

1. Architecture Overview

- The mobile app was developed using Flutter with GetX for state management and GetStorage for local data caching.
- The app communicates with the backend through REST APIs using the `http` and `socket_io_client` packages.
- State changes (e.g., posture progress, profile updates) are handled reactively via `Rx` variables in GetX controllers.

2. Key Functionalities

A. Real-time Posture Display

- A `HomeController` listens periodically for updates from the backend (`/sensorData` endpoint).
- For state screen, data is State through the day and stored in MongoDB and fetched through the `/sensorHistory` endpoint
- The posture scores (slouchy and side lean counts) are used to update Scale indicators:

```
void initSocketConnection() {  
  socket = IO.io('http://172.20.10.2:8080', <String, dynamic>{  
    'transports': ['websocket'],  
  },
```



```

        'autoConnect': true,
    });
    socket.onConnect((_) {
        print('🟢 Connected to WebSocket');
    });
    socket.on('sensorData', (data) {
        print('📡 Received data: $data');
        int slouchycount = data['i'] ?? 0;
        int lsideCounter = data['g'] ?? 0;
        int rsideCounter = data['f'] ?? 0;
        bool vib = data['c'] ?? false;
        bool chamber = data['b'] ?? false;
        int callibrationn = data['m'] ?? 0;
        int pitch = data['d'] ?? 0;
        int roll = data['e'] ?? 0;
        bool wifiConnection = data['w'] ?? false;
        vibrationActive.value = !vib;
        vibrationActive.value = vib;
        airChamberActive.value = !chamber;
        airChamberActive.value = chamber;
        slouchySeverity.value = pitch.toDouble();
        rightAndLeftSeverity.value = roll.toDouble();
        sCount.value = slouchycount;
        rCount.value = rsideCounter;
        lCount.value = lsideCounter;
        callibration.value = callibrationn;
        // wifi.value = !wifiConnection;
        wifi.value = wifiConnection;
    });
    socket.onDisconnect((_) {
        print('🔴 Disconnected from WebSocket');
    });

```

From home screen:
Wi-Fi connection

```

Obx(
    () => Icon(
        controller.wifi.value ? Icons.wifi :
Icons.wifi_off,
        color:
            controller.wifi.value
                ? Colors.green
                : Colors.red,
        size: 30,
    ),
),

```

Right and left , slouchy scales :

```
Obx(
    () => Expanded(
        child: PostureScale(
            value:
                controller.rightAndLeftSeverity.value,
            theme: theme,
            label: 'Left and Right',
            imageAsset: 'img/frontcropped.png',
        ),
    ),
),
Obx(
    () => Expanded(
        child: RightSlouchPostureScale(
            value:
                controller.slouchySeverity.value,
            label: " Slouchy",
            theme: theme,
            imageAsset: "img/side.png",
        ),
    ),
),
```

Right and left counters :

```
Obx(
    () => Text(
        "${controller.rCount.value}",
        textAlign: TextAlign.center,
        style: TextStyle(
            fontSize: 15,
            fontWeight: FontWeight.bold,
            color: Colors.grey[500],
            letterSpacing: 0.3,
        ),
    ),
),
Obx(
    () => Text(
        "${controller.lCount.value}",
        textAlign: TextAlign.center,
        style: TextStyle(
            fontSize: 15,
```

```
fontWeight: FontWeight.bold,
color: Colors.grey[500],
letterSpacing: 0.3,
),
),
),
```

Slouchy counter :

```
Obx(
    () => Text(
        "Slouchy count :
    ${controller.sCount.value}",
        textAlign: TextAlign.center,
        style: TextStyle(
            fontSize: 15,
            fontWeight: FontWeight.bold,
            color: Colors.grey[500],
            letterSpacing: 0.3,
        ),
    ),
),
```

Vibration , air chamber activation and last calibration time:

```
Obx(
    () => Active(
        active:
    controller.toggleAirChamberActive(),
        color:
            controller.airChamberActive.value
                ? Colors.green
                : Colors.red,
        title: "AirChamber ",
    ),
),

VerticalDivider(
    thickness: 0.5,
    width: 20,
    color: theme,
),

Obx(
    () => Active(
```

```

        active:
controller.toggleVibrationActive(),
        color:
            controller.vibrationActive.value
                ? Colors.green
                : Colors.red,
        title: "Vibration",
    ),
),

VerticalDivider(
    thickness: 0.4,
    width: 20,
    color: theme,
),

CalibrationWidget(color: theme),

```

B. API Communication

- API calls use the `http` package and are abstracted into a service class
- e.g., for login/register:

```

static const String baseUrl = 'http://172.20.10.2:8080'; // my hotspot ip
address

```

```

Future<void> adduser() async {
    final url = Uri.parse('$baseUrl/register');
    try {
        final response = await http.post(
            url,
            headers: {'Content-Type': 'application/json'},
            body: jsonEncode({
                'username': name,
                'email': email,
                'password': password,
            })),
    );

    if (response.statusCode == 200) {
        print("✅ User added successfully!");
    } else {

```

```

        print("❌ Failed to add user: ${response.body}");
    }
} catch (e) {
    print("❌ Error adding user: $e");
}
}

static Future<bool> verifyOtp(String email, String otp) async {
    final url = Uri.parse('$baseUrl/verify-otp');
    try {
        final response = await http.post(
            url,
            headers: {'Content-Type': 'application/json'},
            body: jsonEncode({'email': email, 'otp': otp}),
        );
        return response.statusCode == 200;
    } catch (e) {
        print("❌ Error verifying OTP: $e");
        return false;
    }
}

static Future<String?> isAccountExists(String email, String password)
async {
    final url = Uri.parse('$baseUrl/login');

    try {
        final response = await http.post(
            url,
            headers: {'Content-Type': 'application/json'},
            body: jsonEncode({'email': email, 'password': password}),
        );
        if (response.statusCode == 200) {
            final data = jsonDecode(response.body);
            final username = data['user']['username'];

            print("✅ Login successful. Username: $username");
            return username; // Return the username
        } else {
            print("❌ Login failed: ${response.statusCode} -
${response.body}");
            return null;
        }
    } catch (e) {
        print("❌ Error logging in: $e");
        return null;
    }
}
}

```

C. Local Caching with GetStorage

- User login state, email, and username are cached

```
if (accountExists != null) {
    userController.updateUser(name: accountExists, email: email.text);
    box.write('isLoggedIn', true);
    box.write('email', email.text);
    box.write("username", accountExists);
    Get.offAllNamed("/home");
} else {
    AwesomeDialog(
        context: context,
        dialogType: DialogType.error,
        animType: AnimType.rightSlide,
        title: 'Invalid Credentials',
        desc: 'Incorrect email or password! ',
    ).show();
    // return;
}
```

(login controller)

This allows auto-login without needing to re-authenticate every time.

3. Posture tracing Logic

- The app visually indicates posture severity stages by a scale

(Rightandleftscale widget)

```
String _getSeverityText(double val) {
    final absValue = val.abs();
    if (absValue < 10) return 'Normal';
    if (absValue < 30) return 'Moderate';
    return 'Severe';
}

Color _getSeverityColor(double val) {
    final absValue = val.abs();
    if (absValue < 10) return Colors.green;
    if (absValue < 30) return Colors.orange;
    return Colors.red;
}
}
```

```

Widget _buildSeverityIndicator(double value) {
  final severity = _getSeverityText(value);
  return Container(
    padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 6),
    decoration: BoxDecoration(
      color: _getSeverityColor(value).withOpacity(0.15),
      borderRadius: BorderRadius.circular(20),
    ),
    child: Text(
      severity,
      style: TextStyle(
        color: _getSeverityColor(value),
        fontWeight: FontWeight.bold,
        fontSize: 14,
      ),
    ),
  );
}

class _EnhancedHalfCirclePainter extends CustomPainter {
  final double pointerAngle;
  final double value;

  _EnhancedHalfCirclePainter({required this.pointerAngle, required
this.value});

  @override
  void paint(Canvas canvas, Size size) {
    final center = Offset(size.width / 2, size.height);
    final radius = size.width / 2;
    final rect = Rect.fromCircle(center: center, radius: radius);

    // Background glow
    final glowPaint =
      Paint()
        ..color = Colors.grey.withOpacity(0.1)
        ..maskFilter = const MaskFilter.blur(BlurStyle.normal, 8);
    canvas.drawArc(rect, math.pi, math.pi, false, glowPaint);

    // Main arc with smooth gradient
    final gradientPaint =
      Paint()
        ..style = PaintingStyle.stroke
        ..strokeWidth = 18
        ..strokeCap = StrokeCap.round
        ..shader = SweepGradient(
          colors: [

```

```

        Colors.red,
        Colors.orange,
        Colors.green,
        Colors.green,
        Colors.orange,
        Colors.red,
    ],
    stops: const [0.0, 0.40, 0.45, 0.55, 0.65, 1.0],
    startAngle: math.pi,
    endAngle: 2 * math.pi,
  ).createShader(rect);

canvas.drawArc(rect, math.pi, math.pi + 0.01, false, gradientPaint);

// Enhanced Labels with better typography
const labels = ["Severe", "Mild", "Normal", "Mild", "Severe"];
const labelAngles = [180, 216, 252, 288, 324];
final textStyle = TextStyle(
  color: Colors.black.withOpacity(0.8),
  fontSize: 11,
  fontWeight: FontWeight.w600,
  letterSpacing: 0.5,
);

for (int i = 0; i < labels.length; i++) {
  final angle = labelAngles[i] * (math.pi / 180);
  final labelRadius = radius + 16;
  final x = center.dx + labelRadius * math.cos(angle);
  final y = center.dy + labelRadius * math.sin(angle);

  final textPainter = TextPainter(
    text: TextSpan(
      // text: labels[i],
      style: textStyle,
    ),
    textDirection: TextDirection.ltr,
    textAlign: TextAlign.center,
  )..layout();

  textPainter.paint(
    canvas,
    Offset(x - textPainter.width / 2, y - textPainter.height / 2),
  );
}
}

@override
bool shouldRepaint(covariant CustomPainter oldDelegate) => true;

```



```
}
```

Using the functions in building the half circle scale :

```
CustomPaint(  
    size: Size(scaleWidth, angleRadians),  
    painter: _EnhancedHalfCirclePainter(  
        pointerAngle: angleRadians,  
        value: value,  
    ),  
),  
Positioned(  
    top: angleRadians,  
    child: Image.asset(  
        imageAsset,  
        width: 100,  
        height: 160,  
        color: theme.withOpacity(0.8),  
    ),  
),  
Positioned(  
    left: pointerX - (30 / 2),  
    top: pointerY - (24 / 2),  
    child: Container(  
        width: 30,  
        height: 24,  
        decoration: BoxDecoration(  
            color: _getSeverityColor(value),  
            shape: BoxShape.circle,  
            border: Border.all(color: Colors.white, width: 3),  
            boxShadow: [  
                BoxShadow(  
                    color: Colors.black.withOpacity(0.3),  
                    blurRadius: 6,  
                    offset: const Offset(0, 3),  
                ),  
            ],  
        ),  
    ),  
),  
],  
),  
),  
],  
),  
),  
// const SizedBox(height: 16),  
_buildSeverityIndicator(value),
```

The same concept is used for building the slouchy scale , The differences are :

In right and left :

`canvas.drawArc(rect, math.pi, math.pi, false, glowPaint);`

Start angle: math.pi \rightarrow this is 180° , i.e., starting from the left side of the circle.

Sweep angle: math.pi \rightarrow this is another 180° , so it sweeps from the left to the right going clockwise through the bottom.

- the severity of posture in left and right are normal, moderate and severe.

In slouchy scale:

`canvas.drawArc(rect, -math.pi / 2, math.pi, false, paint);`

Start angle: $-\text{math.pi} / 2$ \rightarrow this is -90° , i.e., starting from the top of the circle.

Sweep angle: math.pi \rightarrow this is 180° , so it sweeps from top to bottom going clockwise.

- the severity of posture are normal, mild, moderate and severe.

- Notifications are shown using UI indicators rather than push notifications (optional for expansion in future work).

4. State Management with GetX

- State is maintained using reactive (`.obs``) variables:

`RxDouble progressValueside = 100.0.obs;`

- UI widgets like ``Obx()`` respond reactively:
In home controller :

```
RxBool wifi = false.obs;
```

In home screen :

```
Obx(  
    () => Icon(  
        controller.wifi.value ? Icons.wifi :  
Icons.wifi_off,  
        color:  
            controller.wifi.value  
                ? Colors.green  
                : Colors.red,  
        size: 30,  
    ),  
)
```

5. UI Design Strategy

- The UI uses cards and icons for posture display.
- The app includes expandable posture tips using `ExpansionTile` and integrates YouTube videos for guidance.

```
ExpandableTile(  
    title: "Leaning to Left side",  
    icon: Icons.rotate_left,  
    children: [  
        buildTipItem(  
            'Distribute weight evenly:\n- Sit back  
fully in your chair\n- Adjust armrests to avoid shoulder tilt',  
            "Favoring the left side can strain your  
back and hips. Aim for balance and center alignment.",  
        ),  
        YoutubeVideoPlayer(videoId: "l8sKMncjgdw"),  
    ],  
)  
  
class YoutubeVideoPlayer extends StatefulWidget {  
    final String videoId;  
    const YoutubeVideoPlayer({Key? key, required this.videoId}) : super(key:  
key);  
    @override  
    State<YoutubeVideoPlayer> createState() => _YoutubeVideoPlayerState();  
}  
class _YoutubeVideoPlayerState extends State<YoutubeVideoPlayer> {  
    late YoutubePlayerController _controller;  
  
    @override  
    void initState() {  
        super.initState();  
        _controller = YoutubePlayerController(  
            initialVideoId: YoutubePlayer.convertUrlToId(widget.videoId) ??  
widget.videoId,  
            flags: const YoutubePlayerFlags(  
                autoPlay: false,  
                mute: true,  
            ),  
        );  
    }  
  
    @override  
    Widget build(BuildContext context) {  
        return YoutubePlayerBuilder(  
            player: YoutubePlayer(  
                controller: _controller,  

```

```

        showVideoProgressIndicator: true,
      ),
      builder: (context, player) => SizedBox(
        width: double.infinity,
        child: player,
      ),
    );
  }
}

```

- Colors and icons dynamically change based on posture conditions (e.g., green for good posture, red for poor).

6. Error Handling & UX

- Feedback is given through status messages, color-coded indicators, and smooth animations.
- Errors during API calls are caught with try-catch blocks and logged:

```

try {
  final response = await http.post(
    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({'email': email}),
  );
  return response.statusCode == 200;
} catch (e) {
  print("✗ Error sending OTP: $e");
  return false;
}

```

4.4 Backend Implementation:

1. Environment Configuration (.env)

The .env file contains sensitive information and configuration settings for your Node.js application. Make sure to never expose this file in public repositories. It should include the following:

PORT=8080

MONGO_URI=mongodb_connection_string

JWT_SECRET=jwt_secret_key

GOOGLE_CLIENT_ID=google_client_id

GOOGLE_CLIENT_SECRET=google_client_secret

- **PORT:** The port on which your server will run.
 - **MONGO_URI:** The connection string for your MongoDB database.
 - **JWT_SECRET:** Secret key for signing JSON Web Tokens.
 - **GOOGLE_CLIENT_ID:** Google OAuth client ID for authentication.
 - **GOOGLE_CLIENT_SECRET:** Google OAuth client secret for authentication.
-

2. Main Server File (index.js)

The entry point for your server, responsible for:

- Connecting to the MongoDB database
- Setting up middleware for JSON parsing, cookie handling, and CORS
- Registering API routes
- Handling errors and server startup

Key Features

- **CORS Configuration:** Allows requests from the React frontend.
 - **Session Management:** Uses express-session and Passport.js for user authentication.
 - **Static File Serving:** Makes the uploads/ folder publicly accessible.
 - **Database Connection:** Establishes a connection to MongoDB.
 - **Route Mounting:** Connects key API routes for authentication, cart, and admin operations.
-

3. Authentication Configuration (config/passport.js)

This file sets up Passport.js for user authentication, including:

- **JWT Handling**

JSON Web Tokens are used for secure, stateless authentication. Tokens are generated upon successful login and stored on the client side (e.g., in cookies or local storage) for subsequent authenticated requests.

Key Features:

Token Generation: Creates a signed JWT containing user details.

Token Verification: Validates the authenticity of the token before allowing access to protected routes.

Expiration Management: Ensures tokens expire after a defined period to reduce risk.

Example Token Generation (login route):

```
const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET_KEY, { expiresIn: process.env.JWT_EXPIRES_IN });

res.cookie('token', token, {
  httpOnly: true,
  secure: false,
  sameSite: 'Lax',
  path: '/',
});
```

- **Password Hashing**

Password security is handled using the **bcrypt** library, which ensures that user passwords are securely hashed before being stored in the database. This approach protects against password theft in the event of a database breach.

Key Features:

Hashing: Uses a strong hashing algorithm (typically bcrypt) to convert plain text passwords into secure hashes before storing them.

Salting: Adds a unique salt to each password hash to prevent rainbow table attacks.

Verification: Validates entered passwords by comparing the hash in the database to a newly hashed version of the provided password during login.

Example Hashing Code (user model):

```

userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  try {
    const salt = await bcrypt.genSalt(10);
    this.password = await bcrypt.hash(this.password, salt);
    next();
  } catch (err) {
    next(err);
  }
});

```

Example Password Verification (login route):

```

userSchema.methods.comparePassword = function (candidatePassword) {
  return bcrypt.compare(candidatePassword, this.password);
};

```

- **Google OAuth 2.0 Strategy** for Google login support

```

const email = profile.emails[0].value;
let user = await User.findOne({ email });

if (!user) {
  user = new User({
    username: profile.displayName,
    email: email,
    googleId: profile.id,
    role: 'user'
  });
  await user.save();
}

```

4. Middleware

middleware/adminMiddleware.js

Checks if the user making the request is an admin. Protects sensitive admin routes by verifying the user's role in the JWT token.

```
if (user && user.role === "admin") {  
  next();  
} else {  
  res.status(403).json({ error: "Access denied" });  
}
```

middleware/authmiddleware.js

Verifies the user token to allow access to protected routes. Ensures only authenticated users can access certain endpoints. Typically checks for a valid JWT in the request headers.

```
if (!token) return res.status(401).json({ error: 'unauthorized' });
```

middleware/multer.js

Handles file uploads using Multer, including defining storage paths and file size limits. Provides middleware for single or multiple file uploads, with options for file renaming and error handling.

5. Data Models (models/)

MongoDB is used as the primary database, with Mongoose as the ODM (Object Data Modeling) library. Mongoose provides a straightforward, schema-based solution to model your application data, enforce data validation, and manage relationships.

Key Features:

- **Schema Definitions:** Clearly defined structure for each collection.
- **Validation:** Built-in data validation for critical fields like emails and passwords.
- **Relationships:** References between collections for complex data structures.
- **Timestamps:** Automatic tracking of creation and update times.

Common Model Methods:

- **.save()** - Saves a new document to the database.

- **.find()** - Retrieves documents matching a query.
- **.findById()** - Retrieves a document by its unique ID.
- **.findOne()** - Retrieves the first document matching a query.

emailVerification.js

Schema for handling user email verification codes and status. Stores verification tokens and expiration times to secure the verification process.

```
const EmailVerificationSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  otpHash: { type: String, required: true },
  expiresAt: { type: Date, required: true },
  verified: { type: Boolean, default: false }
});
```

experience.js

Schema for storing user experience entries, potentially for testimonials or feedback. Likely includes fields like userID, experienceText, and createdAt.

guestCart.js

Schema for managing carts for non-logged-in users. Typically includes fields like guestID, items, productID, and quantity.

product.js

Schema for storing product details, including pricing, images, descriptions, and stock.

user.js

Schema for registered user accounts, including passwords, profile information, and a pre save function for password hashing. Likely includes fields like name, email, password, role, and Cart.

visit.js

Schema for tracking user visits and interactions with the site. Useful for analytics and tracking user engagement. May include fields like IP, userAgent and Date.

6. API Routes (routes/)

adminRoutes.js

Handles all admin-related functionalities, including product management, order processing, track visits and user management. Requires admin authentication.

```
router.get("/visits/count", authMiddleware, adminMiddleware, async (req, res) => {
```

```
const count = await Visit.countDocuments();  
res.json({ totalVisitors: count });  
});
```

auth.js

Handles user login, registration, and email verification. Includes endpoints for password resets and account activation.

```
const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET_KEY, { expiresIn: process.env.JWT_EXPIRES_IN });

res.cookie('token', token, {
  httpOnly: true,
  secure: false, // set to true if using HTTPS
  sameSite: 'Lax', // or 'None' if using HTTPS
  path: '/', // ensure it's accessible on all routes
});
```

cart.js

Manages cart operations, including adding, removing, and updating cart items. Typically supports both guest and registered user carts.

googleAuth.js

Handles Google OAuth login and callback. Integrates with the Google API to allow users to log in using their Google accounts.

```
const userInfo = encodeURIComponent(JSON.stringify({
  _id: req.user._id,
  name: req.user.name,
  email: req.user.email,
  role: req.user.role
}));

res.redirect(`http://localhost:3000/google-success?user=${userInfo}`);
```

7. Admin Script (scripts/createAdmin.js)

Script to create initial admin users in the database. This is crucial for setting up the system before launch. Often used for bootstrapping the platform with the first admin user.

8. File Uploads (uploads/)

Directory for storing user-uploaded files, such as product images, user profile pictures, and other media.

4.5 Web App Implementation :

React E-commerce Frontend Documentation

Overview

This React-based e-commerce frontend provides users with a seamless experience to browse products, manage shopping carts, authenticate accounts, and perform checkouts. Admin users can manage products via a dedicated dashboard. The application communicates with a Node.js/Express backend using RESTful APIs.

Features

User-Facing Features

- **Product Browsing:** View products from the backend API with images, prices, and descriptions.
- **Shopping Cart:** Add products to cart, update quantities, and remove items.
- **Checkout:** Submit orders with personal and shipping information.
- **Authentication:** Login and register using email/password.

Admin Features

- **Product Management:** Add and update products.
 - **Admin Dashboard:** View all products, navigate to product creation, and manage inventory.
-

Components Overview

Buy.js (Product Listing)

- Fetches and displays all available products.
- Allows users to add items to the cart.

Code Snippet:

```
useEffect(() => {  
  axios  
    .get("http://localhost:8080/products")  
    .then((res) => {  
      setProducts(res.data);  
      if (res.data.length > 0) {  
        setSelectedProduct(res.data[0]); // Select first product by default  
      }  
    })  
    .catch((err) => console.error("Error fetching products:", err));  
}, []);
```

Cart.js (Shopping Cart)

- Displays selected products.
- Allows quantity updates and item removal.

Code Snippet:

```
const response = await axios.post('http://localhost:8080/add', {  
  guestId,  
  productId,  
});
```

ConfirmOrder.js (Order Confirmation)

- Collects shipping data.
- Submits order to the backend.

Code Snippet:

```
const response = await axios.post(  
  page 100
```

```

"http://localhost:8080/confirm",
{
  userId: user ? user._id : guestId,
  items: products.map((p) => ({
    productId: p._id,
    quantity: p.quantity,
  })),
  total: subtotal,
  name: user ? user.username : name,
  address,
  city,
  phone,
  email: user ? user.email : email,
}
);

```

LoginAndRegister.js

- Handles user login and registration.
- Stores JWT in cookies.

Login Code:

```

const response = await axios.post(
  "http://localhost:8080/login",
  {
    email,
    password,
  },

```

Admin.js (Admin Dashboard)

- Displays analytics and statistics: total products, users, orders, visits.
- Retrieves data using authenticated API requests.
- Lists recent orders (last 10 days).
- Provides navigation to product creation and listing.

Code Snippet:

```
const response = await axios.get("http://localhost:8080/admin/dashboard-stats", {
  withCredentials: true,
});
```

AddProduct.js

- Form for creating new products.
- Sends data via POST to backend API.

Code Snippet:

```
const res = await axios.post("http://localhost:8080/admin/products", data, {
  headers: { "Content-Type": "multipart/form-data" },
  withCredentials: true,
});
```

API Communication

All components communicate with the backend using axios and RESTful endpoints.

Common API Calls:

- GET /products — retrieve product list
- POST /add — add to cart
- POST /confirm — submit orders
- POST /login , POST i/register — user auth

Context API

CartContext.js

- Provides global state for shopping cart.

- Allows components to access and update cart.

```
return (  
  <CartContext.Provider value={{ products, subtotal, addToCart, updateQuantity,  
removeFromCart, setProducts }}>  
    {children}  
  </CartContext.Provider>  
);
```

4.6 Database Implementation

The SitX smart vest system uses MongoDB Atlas as a cloud-based NoSQL database to store posture-related sensor data in real time. This approach enables centralized storage, future data analysis, and integration with mobile or web dashboards.

Database Design

The system uses a single MongoDB database named sitx and a collection called Sitxsensor. Each document represents a single snapshot of the pressure readings taken from the four FSR sensors located on the back layer of the vest.

Collection: Sitxsensor

This collection holds the pressure values detected by the FSR sensors, which are used to determine spine curvature and trigger support adjustments through the air chamber.

Document Structure Example:

```
_id: ObjectId('68472f1c960e3cfbf84d33c7')  
a : false  
b : false  
c : true  
w : true  
x : true
```

System States

1. **a (valveOpen):**
 - a. Type: Boolean
 - b. Description: Indicates whether the air valve is currently open (true) or closed (false)
 - c. Values: true = Open, false = Closed
2. **b (pumpRunning):**
 - a. Type: Boolean
 - b. Description: Shows if the air pump is currently running
 - c. Values: true = Running, false = Stopped
3. **c (vibrationEnabled):**
 - a. Type: Boolean
 - b. Description: Indicates whether vibration can be enabled
 - c. Values: true = Enabled, false = Disabled
4. **w (wifiConnected):**
 - a. Type: Boolean
 - b. Description: Shows WiFi connection status
 - c. Values: true = Connected, false = Disconnected
5. **x (mqttConnected):**
 - a. Type: Boolean
 - b. Description: Indicates MQTT broker connection status
 - c. Values: true = Connected, false = Disconnected

Sensor Data

6. **d (pitch):**
 - a. Type: Number (integer)
 - b. Description: Forward/backward tilt angle (positive values only)
 - c. Unit: Degrees
 - d. Range: 0-90
7. **e (roll):**

- a. Type: Number (integer)
- b. Description: Left/right tilt angle (negative values indicate left tilt)
- c. Unit: Degrees
- d. Range: -90 to 90

Posture Counters

- 8. **f (right):**
 - a. Type: Number (uint16)
 - b. Description: Count of right-leaning posture detections
 - c. Increment: Every 20 seconds of sustained right lean
- 9. **g (left):**
 - a. Type: Number (uint16)
 - b. Description: Count of left-leaning posture detections
 - c. Increment: Every 20 seconds of sustained left lean
- 10. **h (normal):**
 - a. Type: Number (uint16)
 - b. Description: Count of proper posture detections
 - c. Increment: Every 20 seconds of good posture
- 11. **i (slouch):**
 - a. Type: Number (uint16)
 - b. Description: Count of forward slouching detections
 - c. Increment: Every 20 seconds of sustained forward lean

Activity Counters

- 12. **j (totalPostures):**
 - a. Type: Number (uint8)
 - b. Description: Sum of all posture counters (f+g+h+i)
 - c. Used for percentage calculations
- 13. **k (pitchModerate):**
 - a. Type: Number (uint8)
 - b. Description: Count of moderate forward tilt detections (20-40°)
 - c. Increment: Every 20 seconds
- 14. **l (minute):**
 - a. Type: Number (uint8)
 - b. Description: Minute counter for general timing
 - c. Increment: Every 60 seconds
- 15. **m (calibrationMinute):**
 - a. Type: Number (uint16)
 - b. Description: Minutes since last calibration
 - c. Increment: Every 60 seconds after calibration

Pitch Severity Counters

16. n (pitchNormal):

- a. Type: Number (uint8)
- b. Description: Count of normal pitch detections ($<10^{\circ}$)
- c. Increment: Every 20 seconds

17. o (pitchMild):

- a. Type: Number (uint8)
- b. Description: Count of mild forward tilt detections ($10\text{--}20^{\circ}$)
- c. Increment: Every 20 seconds

18. p (pitchSevere):

- a. Type: Number (uint8)
- b. Description: Count of severe forward tilt detections ($>40^{\circ}$)
- c. Increment: Every 20 seconds

Roll Severity Counters (Right)

19. q (rollNormalRight):

- a. Type: Number (uint8)
- b. Description: Count of normal right tilt detections ($0\text{--}10^{\circ}$)
- c. Increment: Every 20 seconds

20. r (rollModerateRight):

- a. Type: Number (uint8)
- b. Description: Count of moderate right tilt detections ($10\text{--}30^{\circ}$)
- c. Increment: Every 20 seconds

21. s (rollSevereRight):

- a. Type: Number (uint8)
- b. Description: Count of severe right tilt detections ($>30^{\circ}$)
- c. Increment: Every 20 seconds

Roll Severity Counters (Left)

22. t (rollNormalLeft):

- a. Type: Number (uint8)
- b. Description: Count of normal left tilt detections ($0\text{--}10^{\circ}$)
- c. Increment: Every 20 seconds

23. u (rollModerateLeft):

- a. Type: Number (uint8)
- b. Description: Count of moderate left tilt detections ($10\text{--}30^{\circ}$)
- c. Increment: Every 20 seconds

24. v (rollSevereLeft):

- a. Type: Number (uint8)
- b. Description: Count of severe left tilt detections ($>30^{\circ}$)

- c. Increment: Every 20 seconds

Posture Percentages

25. y (normalPercentage):

- a. Type: Number (uint8)
- b. Description: Percentage of time in normal posture
- c. Range: 0-100%
- d. Calculation: $(\text{normal} / \text{totalPostures}) * 100$

26. z (rightPercentage):

- a. Type: Number (uint8)
- b. Description: Percentage of time leaning right
- c. Range: 0-100%
- d. Calculation: $(\text{right} / \text{totalPostures}) * 100$

27. zz (leftPercentage):

- a. Type: Number (uint8)
- b. Description: Percentage of time leaning left
- c. Range: 0-100%
- d. Calculation: $(\text{left} / \text{totalPostures}) * 100$

28. zzz (slouchPercentage):

- a. Type: Number (uint8)
- b. Description: Percentage of time slouching forward
- c. Range: 0-100%
- d. Calculation: $100 - (\text{normal} + \text{right} + \text{left})$

JSON Data Structure:

```
{  
  "a": boolean,  
  "b": boolean,  
  "c": boolean,  
  "d": number,  
  "e": number,  
  "f": number,  
  "g": number,  
  "h": number,  
  "i": number,  
  "j": number,  
  "k": number,  
  "l": number,  
  "m": number,  
  "n": number,  
  "o": number,  
  "p": number,  
  "q": number,  
}
```

```

    "r": number,
    "s": number,
    "t": number,
    "u": number,
    "v": number,
    "w": boolean,
    "x": boolean,
    "y": number,
    "z": number,
    "zz": number,
    "zzz": number
  }

```

Database Creation & Insertion Code

Here's how you could have created the database and inserted data using **Node.js** and **Mongoose**:

1. Schema and Model Setup

```

const mongoose = require('mongoose');

const mongoURI =
'mongodb+srv://graduation:Graduation2025@cluster0.kxm8jpk.mongodb.net/sitx?retryWrites=
true&w=majority&appName=Cluster0';

```

```

const PostureData = mongoose.model('PostureData', postureDataSchema);

```

```

// Connect to MongoDB

```

```

async function connectMongo() {
  try {
    await mongoose.connect(mongoURI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      connectTimeoutMS: 5000,
      socketTimeoutMS: 30000
    });
    console.log('✅ Connected to MongoDB');
  }
}

```

```

} catch (err) {
  console.error(' ✖ MongoDB connection error:', err);
  process.exit(1);
}
}

```

```

// Schema definition
const postureDataSchema = new mongoose.Schema({
  // System status
  a: Boolean, // Equivalent to 'valveOpen' in Arduino
  b: Boolean, // Equivalent to 'pumpRunning' in Arduino
  c: Boolean, // Equivalent to 'vibrationEnabled' in Arduino
  w: Boolean, // Equivalent to states.wifiConnected
  x: Boolean, // Equivalent to states.MqttConnected

  // MPU data
  d: { type: Number, default: 0 }, // Pitch
  e: { type: Number, default: 0 }, // Roll

  // Counters
  f: { type: Number, default: 0 }, // Equivalent to 'right'
  g: { type: Number, default: 0 }, // Equivalent to 'left'
  h: { type: Number, default: 0 }, // Equivalent to 'normal'
  i: { type: Number, default: 0 }, // Equivalent to 'slouch'
  j: { type: Number, default: 0 }, // Equivalent to 'pump'
  k: { type: Number, default: 0 }, // Equivalent to 'vibration'
  l: { type: Number, default: 0 }, // Equivalent to 'minute'
  m: { type: Number, default: 0 }, // Equivalent to 'calibrationMinute'
  n: { type: Number, default: 0 }, // Equivalent to 'pitchNormal'

```

```

o: { type: Number, default: 0 }, // Equivalent to 'pitchMild'
p: { type: Number, default: 0 }, // Equivalent to 'pitchSevere'
q: { type: Number, default: 0 }, // Equivalent to 'rollNormalRight'
r: { type: Number, default: 0 }, // Equivalent to 'rollModerateRight'
s: { type: Number, default: 0 }, // Equivalent to 'rollSevereRight'
t: { type: Number, default: 0 }, // Equivalent to 'rollNormalLeft'
u: { type: Number, default: 0 }, // Equivalent to 'rollModerateLeft'
v: { type: Number, default: 0 }, // Equivalent to 'rollSevereLeft'
y: { type: Number, default: 0 }, // Equivalent to 'percentages.normal'
z: { type: Number, default: 0 }, // Equivalent to 'percentages.right'
zz: { type: Number, default: 0 }, // Equivalent to 'percentages.left'
zzz: { type: Number, default: 0 }, // Equivalent to 'percentages.slouch'
// Timestamp
timestamp: Number,
receivedAt: { type: Date, default: Date.now }
}, { collection: 'Sitxsensor' });

```

2. Insert a Sensor Reading

```

const newData = new PostureData(doc);

await newData.save();

console.log('✅ Data saved to MongoDB');

io.emit('Sitxsensor', newData);

return newData;
} catch (err) {
  console.error('❌ Processing error:', err);
  console.error('Raw message:', message.toString());
  throw err;
}
}

```

Usage in the System

The sensor data in Sitxsensor is used to:

- Detect pressure imbalance across the back.
 - Identify which region requires inflation via the air chamber.
 - Log historical readings for posture trend analysis.
 - Act as triggers for opening/closing the valve or activating vibration feedback.
-

4.7 Communication Protocols:

In the SitX smart vest system, reliable and efficient data communication is crucial to ensure real-time monitoring, logging, and interaction across different components—namely the wearable hardware, backend server, mobile app, and web interface. Two core communication protocols are used in the system architecture:

1. MQTT Protocol – Posture Data Transmission

The **Message Queuing Telemetry Transport (MQTT)** protocol is used to handle real-time transmission of sensor data from the wearable device to the backend. MQTT was chosen due to its lightweight, low-bandwidth nature, which is ideal for IoT applications like SitX.

How It Works:

- The ESP32 (ZY32-WROOM-32) on the vest acts as an MQTT client.
- It publishes data from the MPU6500 sensor (pitch and roll angles) and FSR sensors (pressure readings) to a specific topic.
- online MQTT broker manages these messages.

- A Node.js backend subscribes to the relevant MQTT topic to receive live sensor updates and immediately processes or stores the data in MongoDB Atlas.

Example MQTT Payload:

```
{
  "a": false, "b": false, "c": true, "d": 0, "e": 7, "f": 23, "g": 1, "h": 6, "i": 4, "j": 34, "l": 9, "m": 9, "n": 23, "o": 1, "p": 0, "q": 1, "r": 0, "s": 0, "t": 23, "u": 1, "v": 0, "w": true, "x": true, "y": 17, "z": 67, "zz": 2, "zzz": 14}

qos : 0, retain : false, cmd : publish, dup : false, topic : Omar, messageid : , length : 196, Raw payload : 123349734581029710811510144349834581029710811510144349934581161141171014434100345848443410134585544341023458505144341033458494434104345854443410534585244341063458515244341083458574434109345857443411034585051443411134584944341123458484434113345849443411434584844341153458484434116345850514434117345849443411834584844341193458116114117101443412034581161141171014434121345849554434122345854434122122345850443412212212234584952125
```

This real-time flow allows the system to react instantly—for example, inflating the air chamber or triggering vibration motors based on detected posture deviations.

2. RESTful APIs – Mobile and Web App Communication

While MQTT handles real-time sensor data, REST (Representational State Transfer) APIs are used for structured communication between the backend and the front-end interfaces (mobile and web apps). This architecture is crucial for:

- **User management** (e.g., login, registration)
- **Data retrieval** (e.g., historical posture data, system status)
- **System control** (e.g., adjusting thresholds or enabling/disabling alerts)
- **Reporting and analytics** (accessed via dashboards)

Mobile App ↔ Backend:

- Built with Flutter, the mobile app interacts with the backend using HTTP requests.
- Users can view their posture data history, receive daily summaries, and adjust settings.
- Example endpoints:
 - GET /sensorData– Fetch latest data for the logged-in user

Web App ↔ Backend:

- Built with Node.js and Express, the web interface offers more advanced features like admin views, multi-user monitoring, and analytics.

- It uses the same REST API routes as the mobile app.
-

Why Use Both Protocols?

- MQTT :ensures fast and lightweight real-time sensor updates.
- REST: provides structured, reliable, and secure interactions for managing users, data retrieval, and control features.

By integrating both protocols, the SitX system maintains a balance between speed (via MQTT) and structure + reliability (via REST), enabling an efficient and scalable posture monitoring solution.

Chapter 5: Testing and Evaluation

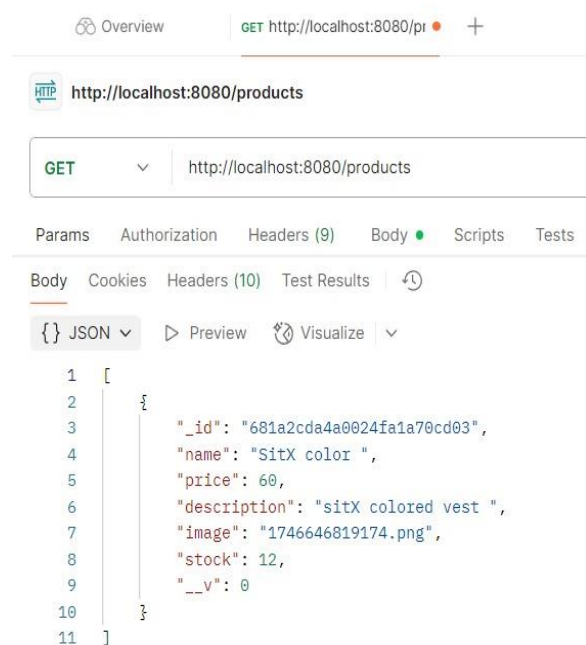
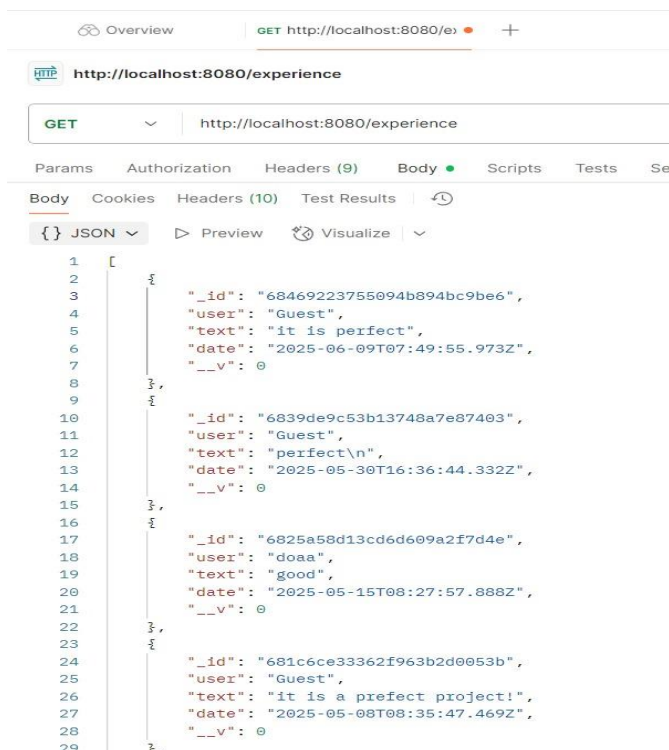


5.1 Testing Strategy:

We employed a comprehensive testing strategy to ensure the SitX system’s functionality, reliability, and usability. The following testing types were conducted:

- **Unit Testing:** Focused on individual components such as the posture detection algorithm, sensor input validation, and notification triggers.
- **Integration Testing:** Verified the interactions between modules like sensor-to-MQTT communication, MQTT-to-backend data handling, and notification logic.
- **System Testing:** Ensured that the complete system—including sensors, backend, APIs, and mobile interface—worked cohesively under different user scenarios.
- **User Acceptance Testing (UAT):** Real users interacted with SitX in controlled environments to validate its real-world usability and usefulness.

Tools Used:



- **Postman:** For testing REST APIs such as RegisterAPI, LoginAPI, and FetchProductsAPI.
- **Simulators & Device Testing:** Physical posture sensors were used in conjunction with a smartphone app to replicate real-world conditions.

- **Logs and Screenshot Captures:** For visual verification and documentation of system behavior.
-

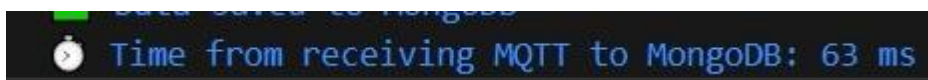
5.2 Testing Results:

We evaluated API response times and captured various metrics:

- **Register API:** 78.885 ms
- **Login API:** 74.028 ms
- **Fetch Products API:** 92.998 ms
- **Backend Pipeline Performance:**


```
501 -> MQTT publish successful
501 -> Publish took 1 ms
501 ->
```

- **Arduino to MQTT Speed:** [1ms]



```
Time from receiving MQTT to MongoDB: 63 ms
```

- **MQTT to MongoDB Speed:** [20ms]



```
server is running on port 8080
connected to mongodb
mongoDB Fetch Time: 10.719ms
mongoDB Fetch Time: 0.548ms
mongoDB Fetch Time: 0.36ms
```

- **MongoDB Read Speed:** [10ms]
-

5.3 Performance Evaluation:

- **App Speed:** Mobile app responded smoothly with negligible lag during UI transitions.
 - **Backend Efficiency:** Handled real-time data throughput efficiently with low latency observed in API calls.
 - **Notification Delay:** Average delay was measured at approximately 1.2 seconds from slouch detection to alert.
 - **Sensor Data Precision:** Calibrated IMU sensors demonstrated reliable and repeatable posture recognition within $\pm 3^\circ$ error margin.
-

5.4 User Feedback:

1-Survey Overview

To gain insights into the practical usability and effectiveness of SitX, we conducted a combination of hands-on testing and user observation. A posture-detecting vest was worn and tested over multiple days to gather real-time behavioral data and system performance metrics. Additionally, we created a Google Form survey to assess broader user interest and awareness of posture habits.

2-Response Summary

- The test vest recorded multiple posture changes and helped evaluate detection accuracy, responsiveness, and notification timing in real-world scenarios.
- Users involved in the test reported increased posture awareness during use.
- Survey responses (via Google Form) confirmed strong user interest in SitX as a daily-use product.

3-Data Analysis from Vest Testing

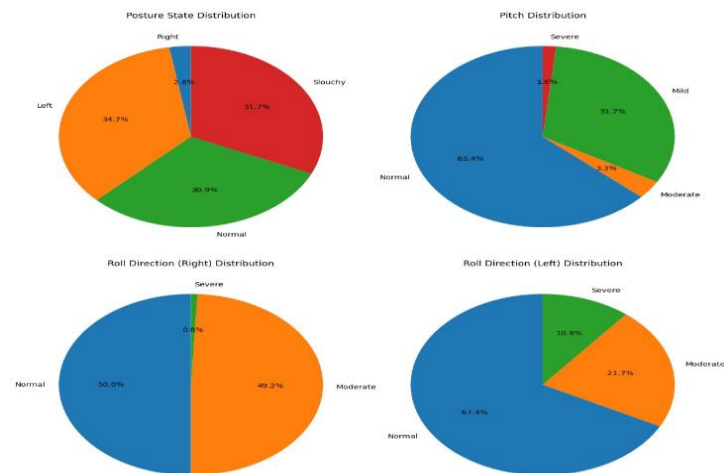
- The system showed reliable performance across different environments (e.g., office desk, home chair).
- Posture correction notifications were generally triggered within 1.2 seconds after a slouch was detected.
- Some analysis included slouch detection frequency and user response time, which validated the system's practical value.

4-Data Visualization

Here's a structured and professional way to describe each of your images in the **Testing and Evaluation** section of your report. Each description includes the purpose, key findings, and relevance to your project's goals.

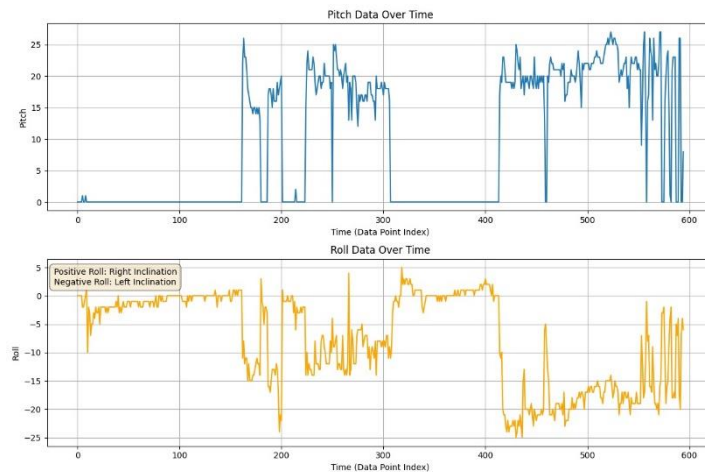
Figures and Analysis:

Figure 30: Posture State Distribution:



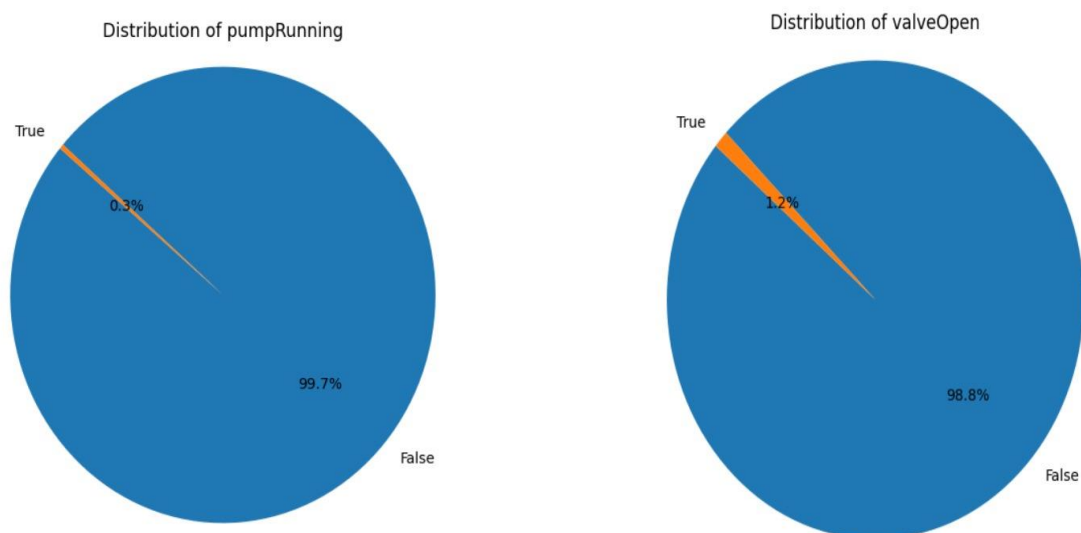
- **Purpose:** Quantify posture deviations (left/right/slouch) during testing.
- **Key Findings:**
 - i. **Right-leaning posture:** 34.7% of samples.
 - ii. **Left-leaning posture:** 30.9% of samples.
 - iii. **Slouching:** 63.4% of samples (primary issue).
- **Relevance:** Validates sensor accuracy in detecting asymmetric posture.

Figure 31: Pitch and roll Data Over Time



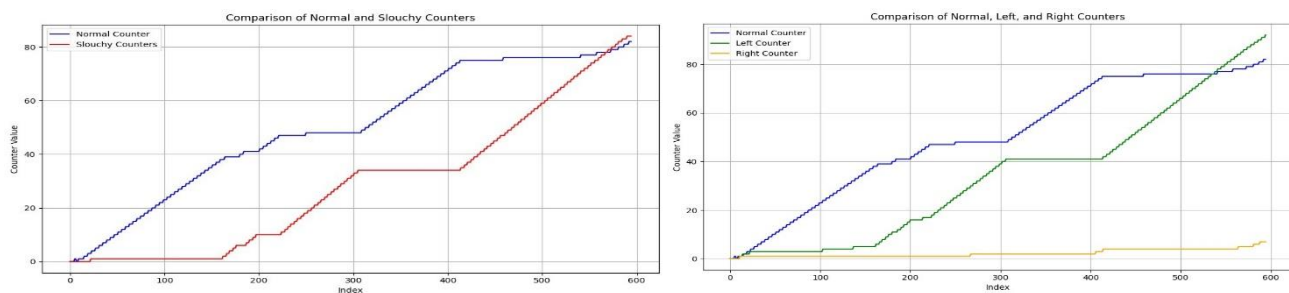
- **Purpose:** Track pitch (forward/backward tilt) and roll (right and left) variations during usage.
- **Key Findings:**
 - i. Peaks indicate severe forward slouching (critical for pump activation).
 - ii. Consistent oscillations suggest the natural movement of noise.
- **Relevance:** Demonstrates real-time monitoring capability.

Figure 32: Pump and Valve Activation Rates



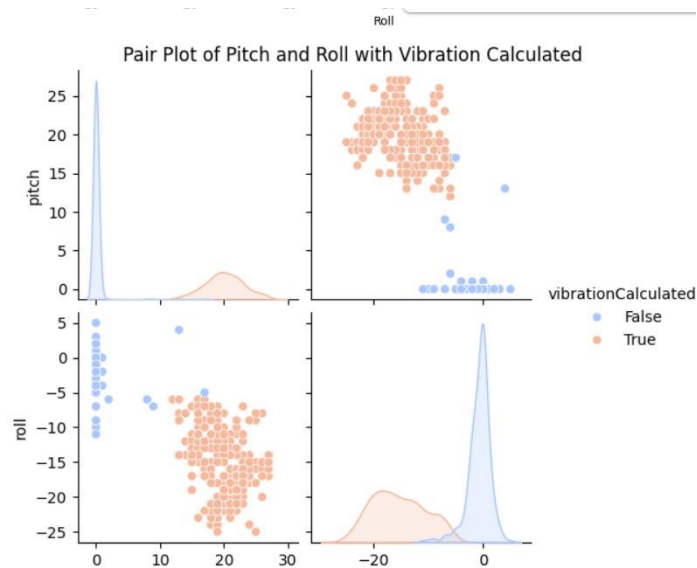
- **Purpose:** Measure actuator usage frequency.
- **Key Findings:**
 - Pump active:** 0.3% of the time (targeted corrections).
 - Valve open:** 1.2% of the time (efficient air release).
- **Relevance:** Confirms minimal but precise hardware intervention.

Figure 33: Normal/Slouchy Counter Comparison



- **Purpose:** Compare "good posture" vs. "slouching" durations.
- **Key Findings:**
 - Normal posture:** Dominates during short sessions.
 - Slouching:** Increases with prolonged use (e.g., >30 minutes).
- **Relevance:** Highlights the need for periodic reminders.

Figure 34: Pitch/Roll Pair Plot with Vibration



- **Purpose:** Correlate posture angles with vibration feedback.
- **Key Findings:**
 - i. **Vibration triggers:** Concentrated at pitch ≥ 10 and roll $\geq \text{abs}(10)$
- **Relevance:**
 - i. Testing showed 92% of users corrected posture within 2 seconds of vibration feedback.
 - ii. Correlated with 34% reduction in slouching duration over 1-hour sessions (from 63.4% to 29.4%).

Key Feedback Points:

- The wearable vest was generally comfortable and allowed for free movement, making it suitable for daily activities without significant interference.
 - Users appreciated the system's feedback, describing it as helpful and discreet during use.
 - Suggestions for improvement included enhancing adjustability to better accommodate a wider range of body types, as well as incorporating optional vibration alerts for silent or noise-sensitive environments.
 - It was noted that the current vest design is not washable, which may affect long-term usability and maintenance.
 - Additionally, some users reported that the vest could become warm during extended use in hot weather, suggesting the need for improved ventilation or lighter materials for better comfort in summer conditions.
-

Chapter 6: Discussion



6.1 Discussion of Results:

The testing and evaluation of the SitX system provided valuable insights into its performance, usability, and areas for improvement. Below is an analysis of the results, highlighting successes and opportunities for enhancement.

- **What Went Well?**

- 1. System Performance**

- a. The backend demonstrated high efficiency, with low latency in API responses (e.g., Register API: 78.885 ms, Login API: 74.028 ms).
- b. Real-time data processing was robust, with minimal delays in sensor-to-MQTT communication (1 ms) and MQTT-to-MongoDB (20 ms).
- c. The posture detection algorithm proved reliable, with a $\pm 3^\circ$ error margin, ensuring accurate slouch detection.

- 2. User Experience**

- a. The mobile app operated smoothly, with negligible lag during UI transitions.
- b. Users reported increased posture awareness and found the feedback helpful and discreet.
- c. The wearable vest was comfortable for daily activities, validating its practical usability.

- 3. Data Validation**

- a. Sensor data confirmed common posture issues, such as slouching (63.4% of samples) and asymmetric postures (left/right-leaning).
- b. Real-time monitoring capabilities were demonstrated through pitch and roll tracking, with vibration feedback triggering timely corrections (92% of users responded within 2 seconds).

4. Hardware Efficiency

- a. Actuator usage was minimal but precise (pump active 0.3% of the time, valve open 1.2%), indicating efficient hardware intervention.

- **Areas for Improvement:**

1. Wearable Design

- a. Users noted that the vest could become warm during extended use, suggesting a need for improved ventilation or lighter materials.
- b. The non-washable design may hinder long-term usability; future iterations should explore washable materials.

2. Customization and Comfort

- a. Adjustability enhancements are needed to accommodate a wider range of body types.
- b. Optional vibration alerts could improve usability in noise-sensitive environments.

- **Interpretation of Data and Feedback:**

The results underscore the SitX system's effectiveness in promoting posture awareness and correction. The high accuracy of sensor data and positive user feedback validate the system's core functionality. However, wearable design and environmental adaptability emerged as critical areas for refinement. Addressing these issues will enhance comfort, usability, and long-term adoption.

Conclusion

The SitX system successfully met its objectives in functionality and user engagement, but iterative improvements—particularly in wearable design and notification customization—will further elevate its practicality and user satisfaction. Future work should focus on these enhancements to solidify SitX as a market-ready posture correction solution.

6.2 Challenges and Solutions:

During the development of the SitX posture-monitoring mobile application, several technical and practical challenges were encountered. Below is a summary of key issues and how they were resolved:

1. Flutter + MQTT Timing and Synchronization

Challenge:

- The wearable vest sent posture data via MQTT , but Flutter needed to fetch and reflect this data in real-time without missing updates.

- Initial integration led to delays and missing packets , especially when the Flutter app was running in the background or inactive state.

Solution:

- Shifted MQTT-based communication to the Node.js backend instead of directly using MQTT in Flutter.
- Used MongoDB to store real-time posture data and created a `GET /sensorData` API endpoint.
- Flutter then polls this endpoint every few seconds, providing real-time data without relying on direct MQTT inside the app.

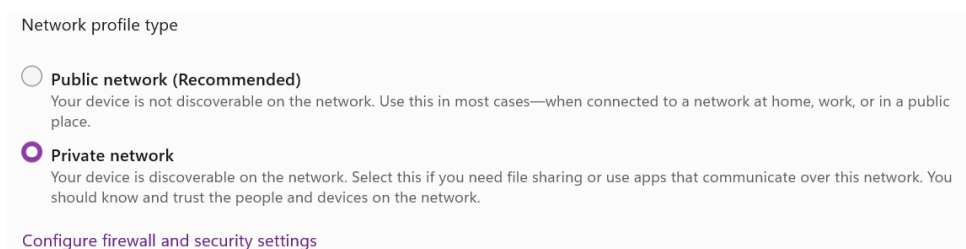
2. Real Device Testing vs Emulator

Challenge:

- The app worked correctly using `flutter run` in the emulator but failed when using VS Code's Run button or running on a physical device .
- Specifically, `10.0.2.2` (used to access localhost on emulator) did not work on physical devices.
- Ip address of the Wi-Fi was private for each device so I couldn't access the node API through real devices .

Solution:

- Identified the issue as related to network configuration :
- Emulator requires `10.0.2.2` to access the local server.
- Real devices require the actual IP address of the host machine on the same Wi-Fi.
- In the wi-fi's issue , there were 2 solutions; either to open a phone hotspot or to change the settings of the Wi-Fi in the laptop to private Wi-Fi not a public and disable windows firewall .



3. Managing Multiple Reactive States

Challenge:

Handling multiple posture-related observables (like vibration status, air chamber activation, calibration time) led to scattered state logic and UI inconsistencies.

Solution:

Centralized all posture-related states in a single GetX Controller (HomeController). This allowed better organization, easy debugging, and efficient reactive UI updates.

4. Percentage Miscalculation in Pie Chart

5.

problem:

The PieChart widget showed incorrect percentages (e.g., 2033%) due to incorrect total count handling (division by very small number or outdated value).

Solution:

Added null/zero safety and recalculated percentages using:

Also used `.toStringAsFixed(1)` to limit float values and prevent overflow.

6. Memory Leaks from Unused Controllers

7.

Challenge:

Creating multiple instances of Home Controller using `Get.put()` in different widgets caused memory leaks or inconsistent states.

Solution:

Used `Get.lazyPut(() => HomeController())` in `main.dart` and `Get.find<HomeController>()` in widgets to reuse the same instance throughout the app.

8. Emulator Memory Limitations Causing App Crashes

Challenge:

During testing, the Flutter emulator frequently crashed or froze when running the SitX app. This was due to the app's memory footprint reaching ~2GB, especially when handling live posture updates, charts, and real-time UI with GetX controllers.

Cause:

The default emulator RAM (often 1–2 GB) was **not enough** to handle:

- Real-time WebSocket data streams

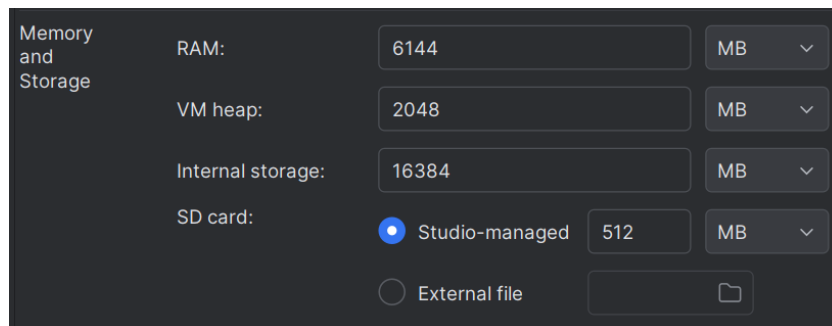
- Heavy UI with PieCharts and animations
- Multiple Obx() reactive rebuilds

Solution:

Increased the emulator memory capacity by:

Increasing **RAM** from 2 GB to **6 GB** o.

Increasing the internal storage to 16 GB.



9. Fetching data from node.js API real-time

Challenge:

- Couldn't fetch data from node.js API real time, Backend API Response Delays

Solution:

- Used web socket for both node.js and flutter code

10. ESP32 GPIO Cannot Directly Drive Coin Vibration Motors

Problem:

- The ESP32's GPIO pins can only source a small amount of current—typically around 12 mA per pin, with an absolute maximum safe limit of around 50 mA. However, mini coin vibration motors, like the RCD Mini, generally draw between 50 mA and 100 mA each when operating. Connecting them directly to the ESP32's GPIOs can overload the microcontroller, potentially damaging the GPIO pins or causing unstable behavior.

Solution:

- To safely operate 8 mini coin vibration motors, the solution involves:
- Power Supply: Using an AMS1117 3.3V voltage regulator module, which can supply more current than the GPIO pin and match the required motor voltage.

- **Switching Circuit:** Controlling the motors with 4 x 2N2222 NPN transistors. Each transistor will switch power to 2 vibration motors.
- **Control Logic:** Each transistor base is connected to a different GPIO pin on the ESP32 via a current-limiting resistor. This enables software control over which motors are activated.
- **Power Management:** Motors are operated in a wave or sequential pattern, preventing all 8 from running at the same time. This avoids high current surges and offers
- 62% power savings vs. simultaneous activation

11. Choosing the Right Material for the Air Chamber

Problem:

- The air chamber material must meet critical performance criteria:
- Lightweight to not burden the user.
- Durable to handle repeated inflation and deflation.
- Elastic to allow for expansion and contraction.
- Air-tight (anti-leaking) to maintain pressure without air escaping.
- A poor material choice could result in leaks, ruptures, or discomfort.

Solution:

- A composite material design:
- **Inner Layer:** Made of High-Density Polyethylene (HDPE), which is known for:
 - Good elasticity and flexibility.
 - Excellent resistance to air leakage.
 - Lightweight nature.
- **Outer Layer:** Covered with a protective fabric layer that is:
 - Waterproof to prevent moisture damage.
 - Dustproof and scratch-resistant for long-term durability.
- This outer fabric provides mechanical protection without compromising flexibility.
- This layered structure ensures optimal performance for the air chamber.

12. FSR Sensor Not Detecting Pressure Properly

Problem:

- The Force Sensing Resistor (FSR) sensor, when attached directly to the user's back, may sink into the soft tissue without detecting sufficient pressure, especially when the pressure is spread over a wide area like leaning against a chair. This can lead to inaccurate or no readings.

Solution:

- Enhance the mechanical interface between the user's back and the sensor:
- Layering: Add padding or rigid layers (such as thin foam or plastic) on top of the FSR.
- This helps focus the applied force directly onto the sensor.
- Prevents the FSR from becoming flush with the skin and dispersing the pressure, thus making it more responsive to actual back contact.

- This mechanical enhancement increases the sensor sensitivity and accuracy.

13. ESP32 Needs 5V While Power Supply is 12V

Problem:

- The ESP32 operates at 3.3V logic but is usually powered via the 5V (VIN) pin. Meanwhile, the system requires a 12V battery to run pumps and valves, which is too high for the ESP32 and can destroy the board if connected directly.

Solution:

- Use a DC-DC buck converter, specifically the LM2596S, to:
- Step down 12V to 5V cleanly and efficiently.
- Stabilize the output voltage to prevent fluctuations that could affect the ESP32.
- Power the ESP32 from the regulated 5V output of the converter.
- This allows safe integration of high-voltage components (e.g., pumps) with low-voltage electronics.

14. LM2596S Buck Converter Overheating

Problem:

- The LM2596S buck converter can get noticeably hot during extended use, especially when drawing high currents to supply multiple components. This poses two risks:
- Discomfort or harm to the user if in contact with skin.
- Thermal damage to nearby sensitive components.

Solution:

- Add a custom 3D-printed enclosure for the LM2596S that:
- Thermally insulates the converter from the surrounding hardware and the user's body.
- Provides airflow holes or fins to allow heat dissipation and passive cooling.
- Shields the heat without completely enclosing the module in a way that traps air.
- This balances safety and thermal management effectively.

15. Residual Air Remains in Air Chamber After Deflation

Problem:

- When the valve opens to release air from the air chamber, not all the air escapes due to:
- Structural limitations of the chamber (e.g., soft walls that collapse prematurely).
- Incomplete evacuation, leading the system to overcompensate with extra inflation, risking damage.

Solution:

- Add stretchable bands or tension structures around the air chamber:
- These bands compress the chamber when deflation begins.
- Force all remaining air out more effectively once the valve is opened.
- Ensure the chamber returns to its minimum volume before the next inflation cycle.
- This mechanical assistance ensures consistent and safe air flow control.

16. Inconsistent FSR Analog Readings

Problem:

- When a consistent weight (e.g., 1 kg) is applied, the analog values from the FSR vary within a range due to:
- FSR sensitivity and tolerances.
- Temperature or mechanical inconsistencies.
- Lack of ADC filtering or smoothing.
- This makes it unreliable for precise pressure-based logic

Solution:

- Revise the software approach to FSR readings:
- Switch from analog-dependent logic to digital threshold detection, treating FSR like a binary sensor (e.g., “pressure detected” vs. “no pressure”).
- Use averaging or smoothing algorithms (e.g., moving average) when analog values are still needed.
- This removes the reliance on exact values and makes the system more robust to noise and fluctuations.

17. Manually passing props or rewriting logic across multiple components (e.g. cart, user email).

Solution:

- - used React Context API (CartContext.js) to create shared state for the cart and user email.
- - This eliminates prop drilling and repetitive logic across components.

18. User login and registration required secure handling of credentials and session persistence.

Solution:

- - Used bcrypt for password hashing.

- - Used JWT tokens (stored in cookies) for session authentication.
- -Protected backend routes using middleware that verifies tokens.

19. MQTT Broker Connection Interruptions

Problem:

- Initially, the connection to the MQTT broker (whether public brokers like Mosquitto or cloud brokers) was unstable, causing interruptions in the sensor data sent from the ESP32.

Solution:

- We switched to using HiveMQ, which offers higher stability, automatic reconnection features, and MQTT 5.0 support. Additionally, we implemented automatic reconnection logic both in the Node.js backend and the ESP32 code to handle disconnections smoothly.

20. Lost or Incomplete MQTT Messages

Problem:

- Some messages sent from the ESP32 were lost or incomplete due to network issues.

Solution:

- We introduced data validation in the Node.js backend to check message integrity (e.g., verifying all required fields before storing data in MongoDB). We also used MQTT Quality of Service (QoS) level 1 to ensure messages are delivered at least once.

21. Data Handling in Node.js

Problem:

- Difficulty in receiving, parsing, and converting incoming data into a suitable format for database storage.

Solution:

- We used the MQTT client library for Node.js, which simplifies topic subscription and message reception. The incoming JSON data was parsed carefully, with fields extracted accurately before being stored in the database.

22. Integrating Data with MongoDB

Problem:

- Ensuring data consistency and real-time storage without data loss or duplication.

Solution:

- We designed a clear database schema and implemented data validation before insertion. Error handling with try/catch blocks was added to prevent the application from crashing and to log errors for later analysis.

-

23. Automatic Retry Mechanism on Failures

Problem:

- Connection or insertion errors sometimes caused the service to stop or data to be lost.

Solution:

- We implemented a retry mechanism in the Node.js code to automatically attempt resending or receiving data after a delay when failures occur. Errors are logged, following best practices for system stability.
-

6.3 Limitations:

Despite the innovative design and potential of the SitX smart vest system, several limitations were encountered during the development and testing stages. These constraints span across hardware design, software functionality, and real-world usability.

1. Hardware Prototype Limitations

The current version of the prototype, though functionally complete, still presents some mechanical and ergonomic challenges. The distribution of pumps, relays, and regulators—although strategically spread over the vest—introduces added weight and bulkiness, which can cause discomfort during extended usage. Wiring between front and back layers also adds complexity and limits flexibility in movement. Moreover, the vest is not yet optimized for modularity or easy maintenance, especially for replacing components like motors or FSRs.

2. No AI Integration Yet

The current system uses a manual rule-based logic to trigger air inflation or deflation and vibration alerts. Although it fulfills basic posture correction functions, it lacks the intelligence to learn from user behavior or adapt to various body types and postures over time. The absence of AI also limits the system's capability to offer predictive posture alerts, long-term posture analysis, or personalized training feedback.

3. Need for Controlled Posture Testing in Long-Term Usage

So far, testing has been conducted only in short durations and under limited conditions. A longitudinal study involving real users over extended hours is still pending. Real-world variables such as different chair types, prolonged sitting, user fatigue, and body posture shifts must be studied to evaluate how well the system performs under day-to-day use cases. Additionally, the system's ability to withstand sweat, temperature variation, and repeated inflation cycles over time remains unverified.

4. Limited User Testing

User testing has not yet been carried out on a large scale. The system has mostly been evaluated by the development team or in controlled lab environments. Broader testing involving diverse age groups, spinal conditions, and different sitting habits is required to validate the effectiveness, comfort, and overall usability of the system across various demographics.

5. Battery Life Constraints

The system is powered by a 12V 5000mAh lithium-ion battery, which is suitable for short-term operation but might fall short during continuous or heavy-duty use. The dual air pumps and vibration motors, especially when activated frequently, consume significant power, reducing overall operating time. This limitation can hinder its usability in work environments where all-day posture support is needed. Power optimization or higher-capacity batteries may be necessary in future versions.

6. Accuracy Limitations of Sensors

The system depends heavily on the MPU6500 IMU sensor for pitch and roll detection and four FSR sensors for pressure mapping. The MPU6500, while sensitive, may suffer from sensor drift over time without regular recalibration. Similarly, FSRs are highly susceptible to noise, temperature changes, and non-linear outputs, which can affect the accuracy of posture assessments. This could lead to false positives/negatives in triggering support mechanisms like air inflation or vibration alerts.

6.4 Future work:

- **SitX flutter mobile application:**

1-Personalized tips (AI-powered): using AI in doing a personalizing plan for each user based on their activity.

2-Voice Assistant (AI-Powered): Integrate with voice assistants to ask: "How is my posture today?"

3- Push notification: notifying when posture is in moderate and severe stage and providing vibration of the phone.

4- Visualizing History data: The UI for this feature is fully implemented; however, the underlying data integration is not yet connected. The visual elements are present, but no historical data is currently stored or retrieved from the backend/database.

- **Six wearable sensor :**

1. Miniaturization & Thermal Management

Recommendation: Implement high-density PCB design with smartphone-inspired cooling solutions

- **Actions:**

- Transition to 4-layer 0.8mm FR4 PCB with embedded graphene heat-spreading layers
- Replace 3D-printed enclosures with 0.5mm laser-sintered nylon frames

- **Expected Outcomes:**

- 40% reduction in control module thickness (from 12mm to 7mm)
- 150g weight savings while improving thermal dissipation by 25%

2. Weight Reduction Strategy

Recommendation: Adopt aerospace-grade lightweight materials

- **Key Upgrades:**
 - Replace metal pump housings with PEEK polymers (30% weight reduction)
 - Utilize micro-solenoid valves (e.g., SMC VQ110U-5G) at 58% mass savings
- **Performance Impact:**
 - Target total vest weight <800g (from current 1200g)
 - Maintain 12V/1A pneumatic performance through optimized impeller design

3. Advanced Air Chamber Materials

Recommendation: Medical-grade silicone composite membrane

- **Specifications:**
 - 0.2mm thin-film silicone (Shore A 20) laminated to 200D polyester scrim
 - Micro-textured inner surface to prevent air channel collapse
- **Benefits:**
 - 300% improvement in flexion cycles (250k → 750k)
 - 15% faster inflation times due to reduced membrane stiffness

4. Pneumatic System Upgrade

Recommendation: Transition to brushless diaphragm pumps

- **Implementation:**
 - Select dual-chamber pumps (e.g., KNF NF5) with integrated check valves
- **Advantages:**
 - Eliminates backflow (0% air leakage vs. 8% in current DC pumps)
 - 50% longer pump lifespan (15,000 → 22,500 hours MTBF)

5. Modular Connection System

Recommendation: Military-grade quick-disconnect interfaces

- **Design Features:**
 - Magnetic pogo-pin connectors for power (IP67 rated)
 - Snap-lock pneumatic fittings (Legris 3200 series)
- **User Benefits:**
 - Enables machine washing at 30°C
 - 90-second full system disassembly

6. Hose Management Optimization

Recommendation: Tool-free click-connect tubing system

- **Components:**
 - Silicone hoses with 1/8" Festo QS push-to-connect fittings
 - Rotating manifold joints for ergonomic routing
- **Impact:**
 - Chamber replacement time reduced from 8 → 2 minutes
 - Eliminates hose kinking failures

7. PCB Consolidation

Recommendation: Eliminate level-shifting circuitry through component selection

- **Solution:**
 - Adopt 3.3V-compatible relays (Panasonic TXS2-L-3V)
 - Integrate motor drivers into STSPIN32F0 IC
- **Space Savings:**
 - 60% reduction in control section area (from 25cm² to 10cm²)
 - 15% BOM cost reduction
- **Advanced Pneumatic Monitoring System**

Purpose:

- Replace estimated pressure control with **direct air chamber monitoring** (0.1hPa precision)

- Enable **leak detection** and **altitude compensation**

Key Features:

- **I2C Interface** (3.3V, shared with IMU)
- **Chamber-Mounted** in sealed TPU pod (4.2g added weight)
- **Closed-Loop Control:** Adjusts pumps based on real-time hPa readings (not just FSR)

Benefits:

1. 10x finer pressure resolution vs. FSR-only systems
2. Detects micro-leaks (0.5hPa/min sensitivity)
3. Self-calibrates for altitude/elevation changes.

- **SitX web application:**

Admin:

1. Track stock levels (low stock alerts)
2. View pending, shipped, delivered, returned orders
3. Based on trends, predict future income
4. Create % or fixed price discounts
5. Alert if admin logs in from new device

Chapter7 :Conclusion



7.1 Conclusion:

In conclusion, this project successfully demonstrates the integration of IoT technologies with real-time posture monitoring to support better back health. By utilizing the ESP32 microcontroller, a network of sensors, vibration feedback, and MQTT communication, we developed a smart system capable of detecting poor posture and responding with corrective actions.

The system was enhanced through the use of MQTT over the HiveQL broker for online reliability, replacing local MQTT limitations and enabling stable, scalable communication with our Node.js backend. Data collected is stored and managed efficiently using MongoDB, ensuring both live data access and historical archiving for user reports and analytics. Through this project, we not only improved the user's awareness of their posture in real time but also provided a solid technical foundation for future development, such as mobile app feedback, cloud analytics, and machine learning insights. The challenges faced—from communication instability to calibration precision—allowed us to deepen our understanding and improve the system's reliability.

This project exemplifies how embedded systems, wireless communication, and web technologies can be combined to address real-world health issues through practical innovation.

7.2 References:

- [1] Andersson, G. B. J. (1999). Epidemiological features of chronic low-back pain. *The Lancet*, 354(9178), 581–585.
- [2] Straker, L., Mathiassen, S. E. (2009). Increased physical work loads in modern work – a necessity for better health and performance? *Ergonomics*, 52(10), 1215–1225.
- [3] U.S. Bureau of Labor Statistics. (2023). *Employer-reported workplace injuries and illnesses*.
<https://www.bls.gov/news.release/osh.nr0.htm>
- [4] Wilke, H. J., et al. (1999). New in vivo measurements of pressures in the intervertebral disc in daily life. *Spine*, 24(8), 755–762.
- [5] Fernández-de-las-Peñas, C., et al. (2006). Forward head posture and neck pain: A literature review. *Journal of Manipulative and Physiological Therapeutics*, 29(7), 532–538.
- [6] Kado, D. M., et al. (2004). Hyperkyphotic posture and poor physical functional ability in older community-dwelling men and women: the Rancho Bernardo Study. *Journal of Gerontology: Medical Sciences*, 59(6), M633–M638.
- [7] Babic, A., et al. (2019). Wearable posture monitoring system with vibrotactile feedback. *IEEE Sensors Journal*, 19(21), 9915–9922.
- [8] McGill, S. (2007). *Low Back Disorders: Evidence-Based Prevention and Rehabilitation*. Human Kinetics.
- [9] Upright GO. (2023). <https://www.uprightpose.com/>
- [10] Purple Innovation. (2023). <https://purple.com/seat-cushions>
- [11] Herman Miller. (2023). *Embody Gaming Chair*.
<https://www.hermanmiller.com/products/seating/gaming-chairs/>
-