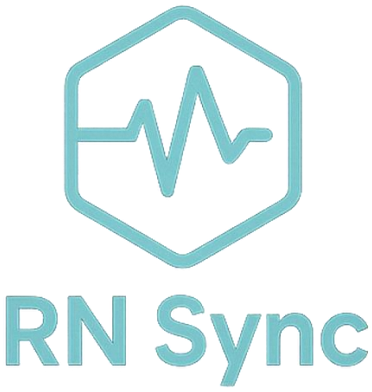
ELEC 490/498 Project Blueprint

iOS-Based Centralized Monitoring Hub



Submitted by Group 22 on November 3rd, 2025

Group members:

1. Benjamin Nguyen | 20dan2 | 20289253
2. Jack Fergusson | 20jtf1 | 20292332
3. Khalid Tahir | 20kht1 | 20267024
4. Ziyad Soultan | 20zs55 | 20285358

Faculty Supervisor: Nasim Montazeri

Executive Summary

RN Sync is an iOS-based centralized monitoring hub that ingests multi-sensor patient data (ECG, SpO_2, motion), synchronizes it and streams through a Node.js backend, and displays real-time views on iOS clients for clinicians and researchers. The Hub focuses on ensuring a reliable real-time streaming connection from patient devices to the cloud and onto iOS. Some features of the Hub include accurate cross-device time alignment on a Coordinated Universal Time (UTC) clock with drift correction, simple reconnections after brief outages, OAuth 2.0 and TLS 1.3 for authenticity and secure connections, encryption at rest, and a performance target of end-to-end p95 latency under 500 ms with synchronized playback across at least four concurrent streams per subject. Enhanced goals include higher advanced alerting, two-factor authentication, audit logging, language toggling, and memory and CPU resource management. Development follows an iterative plan across four prototypes: 1. core data pipeline proving single-source streaming from test script, to backend, and basic iOS chart under 500 ms; 2. synchronization across multiple streams and dashboard foundation rendering; 3. user features and historical data, including authentication, device management, and live/historical views; 4. polished UI, security, and deployment, native notifications, complete security hardening, "Clinical Simulation" test, and TestFlight distribution. Execution uses two-week Scrum sprints, going from planning, weekly stand-ups, review, and retrospective.

A modular design enables parallel work: backend modules for WebSocket ingestion, data validation/UTC normalization, authentication, and an API gateway; frontend modules for an API service, reusable visualization components, and notifications. The approach relies on React Native/iOS with native background, a Node.js backend with WebSocket ingestion plus time-series and object storage, iPhones, iPads, and Xcode used for device-level testing, and a PaaS deployment (Firebase/AWS Amplify/Heroku selected early) to minimize operations burden.

The expected contribution is a practical, reusable template for low-latency, time-synchronized, multi-sensor monitoring on the iOS operating system, functional to students, research labs, and possible healthcare professionals, demonstrating a clean, testable pipeline from live captured data to cloud synchronizations and mobile rendering with clear performance.

Table of Contents

List of figures

List of tables

1. Introduction

RN Sync, an iOS-Based Centralized Monitoring Hub, aims to solve the problems of inefficiency, inaccuracy, and inaccessibility of physiological monitoring systems for healthcare, fitness, and research purposes. The primary end users for this product are clinical researchers, who would benefit significantly from an inexpensive, easy-to-use system for comprehensive patient data monitoring. Secondary users such as healthcare professionals and fitness coaches are also included in the target market.

The app's primary function is to combine real-time data from multiple sources into a comprehensive dashboard and to store this data for later analysis. This function can dramatically increase efficiency in assessing physiological metrics, as all data for a patient is displayed in one place, and the data can be viewed from anywhere at a glance. The system will also increase the user's ability to make insights into the data, as data is shown in a time-synchronized fashion, meaning one could analyze multiple metrics at a precise point in time. It could also analyze the historical data of a patient to gain insights into their typical numbers in comparison with current readings.

The scope of this project contains the following main achievements:

1. The app is published to the iOS App Store.
2. The app can receive signals from multiple computers simultaneously.
3. Data is organized into a single dashboard.
4. All data is stored and can be viewed at a later time.

As an extended scope, the app should be able to receive signals from multiple patients and categorize them accordingly, making the app more practical for a healthcare practitioner with many patients at once.

To achieve the aspects of the scope listed above, the needed inputs are the test signals from several laptops (or historical data from the database), and the output would be the visualization of the data. The main components of the system are the server, the laptops which transmit

data to the server, the app's frontend, which displays the data from the server, and a database to store the data. All these components will be connected using Node.js with an Express.js server.

1.1. Design Problem

Many healthcare professionals from different industries, as mentioned in the Introduction, are seeking a solution that is affordable and reliable. To achieve this, a User Interface built with React Native, coupled with a Node.js Cloud backend, will be the pillar of the application. Within the application, there are historical medical views of metrics such as heart rate, blood pressure, body temperature, respiration rate, electrocardiogram, etc. Data used in the app originates from patients' sensors and gets uploaded to a central local Hub to be processed and synchronized to the Cloud for viewing from remote devices. Most data handled by the Hub is stored and processed for viewing in a time-series database, except for clinical notes/documents and medical images stored in object storage. After being processed, the data will synchronize across all authorized devices for real-time viewing. A general diagram, Figure 1, shows a high-level system of the connections between the patient, server, and users and will be revised for the blueprint.

1.2. System Specifications

Interface requirements	**Target value**	**Tolerance**
Responsiveness	95%	± 5%
TLS and Authentication	≤ 15 minutes	0
Performance requirements		
Uptime availability	99%	± 0.5%
API latency	≤ 200 milliseconds	± 50 ms
Streaming latency	≤ 500 milliseconds	± 100 ms
Account Authentication with email	≤ 5 seconds	± 2 s
Data ingestion for batches up to 2MB	10k samples/minute	≤ 0.1%
Real time updates	≤ 500 milliseconds	± 100 ms
Data strip frequency	≥ 256 hz	±

Table 1: Required System Specifications

Functional requirements	**Target value**	**Tolerance**
Account authentication with 2FA	≤ 5 seconds	± 2 s
Audit logging	< 10s	± 5 s
Interface requirements		
Responsiveness	95%	± 5%
Language toggling	100%	
Alerts & notifications emissions	< 2 seconds	± 500 ms
Performance requirements		
Resource usage (memory)	≤ 250 MB	± 50 MB
Resource usage (CPU)	≤ 25%	± 5%

Table 2: Optional (Enhanced) System Specifications

1.2.1 Interface Specifications

RN Sync's interface will be constructed with simplicity and performance in mind to ensure it is accessible and fast. Dimensions of the application will include only the fully supported iPhone and iPad lineups: iPhone 11, iPad 8th generation or newer iPhone/iPad models, ensuring that users will not face any responsiveness issues. All Cloud communications by the Hub will use the REST architecture with a WebSocket endpoint to synchronize real-time data with real time updates with a target value of less than 500ms for improved data accuracy. TLS version 1.3 and Open Authentication 2.0 will be used for secure connections to new devices [1]. Connection will prompt warning when the 15 minutes mark is reaching, and users are required to reauthenticate at least every 12 hours and inactivity after 30 minutes [2]. Since medical data are highly sensitive, a hierarchy following the CIA triad will be created to provide access to only authorized users. Within the database, the timestamps for the Hub will follow ISO-8601 UTC protocol; the time zone used for this project will be Eastern Standard Time, but timestamps will be stored in (UTC -5) [3].

1.2.2 Performance Requirements

RN Sync must ensure the end-to-end latency for live views is less than 500 ms within the 95% (p95); the latency is from the Cloud to the user interface [4]. According to a journal from Computers in Radiology, medical data types vary; therefore, each type will have a specific

storage size catered to its usage. Although images such as digital/computed radiography and mammography are heavy, they are not frequently updated, so their lifespan will remain in the database until the patient's information is archived. On the other hand, time-based data such as ECGs will have a lifespan of 6 months as they are constantly being updated to the database [5]. After an ECG strip reaches 6 months, new data will override the oldest data strips, and the process repeats. Data strips size, calculated by taking the product of sample's frequency and sample's bytes, using higher frequency (> 256 hz) and larger sample's byte (> 2 bytes), equate to a maximum of 500KBps. Therefore, a safe date transfer rate for live data strips will have a minimum of 4bps.

2. Methodology/Work Breakdown Structure

2.1. Approach

2.1.1 Iterative Prototyping: Building and Refining in Steps

Iterative prototyping means we build, test, analyze, and improve our product in a cycle. Instead of trying to build the whole system at once, we will develop RN Sync in a series of versions, or iterations. Each iteration will result in a working piece of software that adds new features and improves on the last one based on feedback. This approach helps us find and fix problems early, which is much easier and cheaper than finding them at the end.

Our Four-Iteration Plan for RN Sync

We will break down the development of RN Sync into four clear iterations, each with a specific goal. This plan is designed to tackle the biggest technical risks first.

- **Iteration 1: The Core Data Pipeline.** The first prototype will focus only on the most critical technical challenge: getting data to flow from the backend to the app with low latency.
    - **Goal:** Prove that we can stream a single data source from a test script, through our backend, and display it on a basic chart in the app.
    - **Success Metric:** Data appears on the iOS screen with a measured latency under our 500 ms target. We are focused on function, not looks at this stage.

- **Iteration 2: Multi-Stream Sync and UI Foundation.** Now that the basic pipeline works, this version will handle multiple data streams and build the main user interface.
    - **Goal:** Display four or more data streams on the dashboard at the same time, making sure they are perfectly synchronized. We will also build the main dashboard UI components.
    - **Success Metric:** All four streams are displayed correctly and in sync without slowing down the app. The main dashboard UI is in place.
- **Iteration 3: User Features and Past Data.** This iteration adds key features for users, like logging in and viewing historical data.
    - **Goal:** Implement the full user login system, features for managing devices, and the ability to view past data from the database on the dashboard.
    - **Success Metric:** A user can log in, see a list of data sources, and switch between viewing live and historical data.
- **Iteration 4: Polish, Security, and Deployment.** The final iteration is about making the app ready for release.
    - **Goal:** Improve the user interface based on feedback, add native notifications for alerts, finish all security features, and conduct the final "Clinical Simulation" test.
    - **Success Metric:** The app passes all our tests and is successfully deployed to the cloud and distributed via TestFlight.

This iterative plan gives us a clear roadmap, breaking a complex project into smaller, manageable steps.

2.1.2 Agile Development: A Flexible and Structured Process

Agile development is a way of managing projects that focus on flexibility and delivering working software quickly in short cycles called sprints. A sprint is a fixed period, usually one to four weeks, where the team works to complete a set amount of work. This helps the team adapt to new challenges and feedback. We will use the Scrum framework, which is a popular way to implement Agile.

How We'll Use Scrum for RN Sync

Our team already has a "highly collaborative" approach, which is a great fit for Scrum.

Sprint Length: The team will work in two-week sprints. This is short enough for frequent feedback but long enough to get meaningful work done.

Agile Meetings (Events): Our project will follow the rhythm of the five formal Scrum events. These are regular meetings to check our progress and adjust our plan.
Sprint Planning: At the start of each two-week sprint, the whole team will meet to plan. The Product Owner will present the most important tasks. The Developers will then decide how much work they can complete in the sprint, creating a Sprint Backlog.

Weekly Stand-up: Every week, we will have a quick 15-minute meeting to sync up. Each person will answer three questions: What did I do this week? What will I do this week? Are there any problems blocking me? This helps us stay aligned and solve problems quickly.
Sprint Review: At the end of the sprint, we will show the working software we built to our faculty supervisor. This is how we get feedback on the product.
Sprint Retrospective: After the review, the team will meet privately to discuss how we can improve our process. We'll talk about what went well, what didn't, and what we'll change for the next sprint.

2.1.3 Modular Design: Building a Clean and Scalable System

Modular design is a technique where you break a program's functions into independent, interchangeable parts called modules. The idea is to create modules that each have one specific job and don't depend too much on other modules. This approach makes the whole system easier to understand, test, maintain, and build in parallel.

How We'll Break Down RN Sync

We will break down our system into logical modules for both the backend and the frontend. This structure is what allows us to develop the two parts in parallel (as planned in Phase 1) and ensures that a change in one area won't break another.

- Backend Modules (Node.js): The server will be built from a set of services where each module has one clear job.
- Ingestion Service: Its only job is to handle the live WebSocket connections and receive raw data from the sensors.

- Data Processing Module: This module takes the raw data and turns it into a structured format. It will parse the data, check that it's valid, and make sure all timestamps are in a standard format (UTC).
- Authentication Service: This module manages everything related to users and security, like logins and session management.
- API Gateway: This is the single-entry point for all REST API requests. It routes requests to the correct internal service.
- Frontend Modules (React Native): The app will also be broken down into modules to keep the code clean and reusable.
- API Service: This module handles all communication with the backend. No other part of the app will make network requests directly.
- Authentication Store: This module manages the user's login status on the app, while securely storing the authentication token.
- Visualization Component Library: This is a set of reusable React components for displaying data to users. These components just take data as input and don't care where it comes from.
- Notification Module: This module will handle sending native iOS notifications for important alerts.

2.2. Design tools, hardware, instrumentation

2.2.1 The Apple Development Ecosystem

The project's explicit goal is to create an "iOS-Based Centralized Monitoring Hub" for deployment on the Apple App Store via TestFlight. This choice irrevocably anchors the client-side development and testing process within Apple's proprietary ecosystem.

- **Hardware:** The team's access to Apple MacBooks and iPhones is a critical and non-negotiable prerequisite for this project.
    - **Apple MacBooks:** These machines serve as the primary development workstations. Their necessity stems from the requirement to run macOS, the exclusive operating system for Xcode. All compilation, debugging, and packaging of the iOS application must be performed within this environment.

- **Apple iPhones:** While the iOS Simulator within Xcode is a valuable tool for rapid UI development and iteration, it is insufficient for comprehensive testing and validation. Physical iPhone devices are essential for several critical validation tasks. These include accurately measuring real-world application performance, assessing the impact of background processes on battery life, testing the reliability of push notifications, and validating the user experience with native touch gestures.

- **Operating Systems:**
  - **macOS:** This is the host operating system for the entire client-side development toolchain. It provides the foundation for running Xcode, the iOS Simulator, and the command-line tools (CLI) required by the React Native framework for building and managing the application.
  - **iOS:** This is the target deployment environment for the RN Sync application. Development and testing must be conducted against recent and stable versions of iOS to ensure compatibility and leverage the latest platform features. The project's scope, which specifies support for the iPhone 11 and newer, requires testing across a range of iOS versions corresponding to these devices to guarantee a consistent and reliable user experience.

2.2.2 The Cloud Deployment Environment: Platform-as-a-Service (PaaS)

The RN Sync architecture features a central cloud server that ingests, processes, and stores sensor data before streaming it to iOS clients. For a project of this scale and nature, a Platform-as-a-Service (PaaS) model is the most strategic choice for deploying the Node.js backend.
The proposal identifies several suitable PaaS providers, including Google Firebase, AWS Amplify, and Heroku, noting their generous free tiers which align with the project's budgetary constraints. The primary advantage of a PaaS model over an Infrastructure-as-a-Service (IaaS) model (e.g., renting a raw virtual machine) is the abstraction of underlying infrastructure management. A PaaS provider handles server provisioning, operating system maintenance, networking, and scaling, allowing the development team to focus exclusively on writing and deploying the application code. This significantly reduces the operational overhead and

cognitive load on the team. The selection among these platforms should be finalized during Milestone 1, with a focus on ease of deployment for Node.js applications that require persistent WebSocket connections.

2.2.3 The Simulation Environment: Local Data Generation

A core component of the project's testing and validation strategy involves simulating physiological data streams to test the system's performance and reliability under various conditions. The team's plan to use laptops to "run the scripts that generate the mock physiological data streams" establishes these machines as the "Patient Side" data sources depicted in the system architecture diagram.

While any modern laptop is sufficient in terms of hardware, its role is defined by the software it runs. These machines will be configured with a dedicated simulation toolchain (detailed in Section 3.1) capable of generating multiple, concurrent, and realistic data streams (e.g., ECG) and transmitting them to the backend server via WebSocket. This simulation environment is critical for executing the test plans outlined in the proposal, including load testing with four or more concurrent streams, injecting malformed data to test parsing robustness, and varying data rates to stress the system's integrity.

2.2.4 Backend Implementation Stack

The selection of a cohesive and appropriate software stack is paramount to achieving the project's functional and non-functional requirements. The following technologies are recommended based on their alignment with the system's real-time data handling needs, the development team's collaborative model, and industry best practices for building scalable mobile applications.

- **Node.js Runtime:** The central challenge of the RN Sync backend is to efficiently handle "multiple, concurrent, long-running connections for real-time data streaming". Node.js is exceptionally well-suited for this I/O-bound workload. Its architecture is built upon a single-threaded event loop and non-blocking I/O model. When a request that involves I/O (like receiving data from a WebSocket or querying a database) is made, Node.js does not block the main thread. Instead, it offloads the operation to the system's kernel and registers a callback function. The event loop is then free to service other incoming

requests. When the I/O operation completes, the kernel places the corresponding callback into the event queue, which the event loop processes in turn. This mechanism allows a single Node.js process to handle thousands of simultaneous connections with minimal memory and CPU overhead, making it an ideal choice for scalable, real-time applications like RN Sync. This stands in contrast to traditional multi-threaded models or systems like standard Python, which is constrained by the Global Interpreter Lock (GIL) and is less efficient at handling many concurrent I/O-bound tasks.

- **Full-Stack Synergy:** The use of JavaScript across the entire stack streamlines the development process and enhances team velocity. This "single-language paradigm" reduces the cognitive load and context-switching required for developers moving between frontend and backend tasks, which directly supports the team's plan to collaborate evenly throughout the project. This synergy extends to practical benefits, such as the ability to share common logic (e.g., data validation schemas for sensor readings) between the client and server, utilize a unified set of development tools (e.g., package managers, linters, formatters), and foster a more flexible and resilient team where any member can contribute to any part of the codebase.

While the core language and runtime provide the foundation, specific frameworks and libraries are needed to accelerate development and provide robust functionality.

The backend requires a framework to handle HTTP requests for its REST API and a library to manage the core WebSocket functionality.[1]

- **Express.js:** As proposed, Express.js is the de facto standard for building web servers and APIs in Node.js.[1] Its minimalist and unopinionated nature is ideal for this project, providing powerful routing capabilities for the REST API (used for functions like user authentication, device management, and fetching historical data metadata) without imposing a rigid structure that might conflict with the real-time WebSocket architecture.
- **`ws` Library:** For a system where performance is critical to meeting the sub-500ms latency target, a dedicated, a high-performance WebSocket library such as `ws` is strongly recommended. It is a lightweight, thoroughly tested library that provides low-level

control over the WebSocket server, ensuring minimal overhead and maximum throughput for the real-time data ingestion path.

2.2.5 Frontend Implementation Stack

The choice of React Native is well-justified for building the iOS application. It allows the team to leverage existing web development knowledge to build a truly native application. This approach significantly accelerates UI development and iteration cycles. A key advantage of React Native is its "bridge" architecture, which allows JavaScript code to interact with native platform APIs. This is essential for implementing features such as native modules for performing background tasks and delivering time-sensitive notifications to users.

To build a robust and maintainable frontend, the following categories of libraries are essential additions to the stack. The proposal's UI build milestone can be significantly de-risked by selecting these tools early.

- **State Management (e.g., Zustand, Redux Toolkit):** The RN Sync dashboard will manage a complex state, including multiple real-time data streams, user authentication status, device connection states, and UI configuration settings. A dedicated state management library provides a predictable and centralized way to manage this complexity, preventing bugs and making the application easier to debug and scale.
- **Navigation (e.g., React Navigation):** A multi-screen application requires a robust navigation solution. React Navigation is the community standard for managing transitions between screens, handling navigation stacks, and passing data between different parts of the UI.
- **Data Visualization (e.g., Victory Native, Recharts):** The core function of the app is to render data streams smoothly and efficiently. The choice of a charting library is critical. While some libraries are easy to implement, they may suffer from performance issues when rendering high-frequency data updates. The team must evaluate and select a library specifically designed for performance in a React Native environment. This choice directly impacts the ability to meet the latency targets and is a primary consideration for the "Real-time Rendering Performance" contingency plan.

2.2.6 Development Toolchain and IDEs

The team's productivity will be greatly enhanced by a standardized and powerful development environment.

- **Visual Studio Code (VS Code):** As proposed, VS Code is the recommended primary code editor for all JavaScript development. Its lightweight nature, extensive extension ecosystem, and integrated terminal make it a highly efficient environment. To enforce code quality and consistency across the team, the following extensions should be considered mandatory:
    - **ESLint:** Statically analyzes code to quickly find problems and enforce coding standards.
    - **Prettier:** An opinionated code formatter that ensures a consistent code style across the entire project.
    - **GitLens:** Supercharges the built-in Git capabilities, making it easier to visualize code authorship and history.
- **Xcode:** While coding will primarily occur in VS Code, Xcode remains an indispensable tool.[1] Its role is not just as a secondary IDE but as the essential build system and deployment manager for the iOS application. It is used for configuring native project settings, managing signing certificates and provisioning profiles, running the application on physical devices, and archiving and uploading the final build to App Store Connect for TestFlight distribution.[1]
- **Node Package Manager (npm):** npm is the default package manager for the Node.js and JavaScript ecosystem. It will be used to manage all project dependencies, including frameworks, libraries, and development tools.
- **React Native CLI:** The command-line interface is the primary tool for initializing, running, and bundling the React Native application.

2.2.7 Data Stream Simulation

The testing plan requires simulating "four concurrent sensor data streams," injecting "malformed and large data packets," and increasing "update frequency" for stress tests. This cannot be achieved reliably by hand.

- Python with asyncio and websockets libraries: Python is the ideal language for creating sophisticated data stream simulators. Its extensive scientific computing libraries can be used to generate realistic physiological waveforms (e.g., ECG signals with P, QRS, and T waves, plus added noise) rather than simple, predictable patterns. The asyncio library allows for the creation of highly concurrent clients that can simulate as many devices as we need connecting simultaneously, while the websockets library provides the means to transmit this data to the RN Sync backend. This combination allows for the scripting of complex test scenarios, such as connection drops, latency variations, and the injection of malformed packets as required by the test plan.

2.2.8 API & Network Testing

The system interfaces, both REST and WebSocket must be thoroughly tested during development.

- Postman: The team will create a shared Postman Collection that documents every endpoint of the REST API. Each request in this collection should have associated automated tests (written in JavaScript within Postman) that verify response codes, data schemas, and business logic. This collection serves as living documentation and can be run automatically in a CI/CD pipeline to prevent regressions.
- WebSocket-specific clients: For low-level debugging of the real-time data path, developers will need a tool to manually connect to the WebSocket server, send arbitrary messages, and inspect the responses. Postman now includes a WebSocket client, and various browser extensions like "Simple WebSocket Client" also serve this purpose well. These tools are invaluable for debugging connection handshakes and data framing issues during initial development.

2.2.9 Version Control & CI/CD

The team will setup a GitHub to manage development using a branching strategy involving the following:

- A main branch that always represents the stable, deployable state of the project.
- A develop branch where all work is integrated.
- Feature branches (e.g., feature/real-time-dashboard) created from develop for each new piece of work. When a feature is complete, it is merged back into develop via a pull request, which allows for code review.
- Continuous Integration/Continuous Deployment (CI/CD): To automate the workflow and improve quality, the team should leverage GitHub Actions. A CI/CD pipeline can be configured to automatically:
    - On every commit to a feature branch: Run code quality checks and execute the entire unit test suite. This provides immediate feedback to developers if their changes have introduced any issues.
    - On every merge to the develop branch: Deploy the Node.js backend to a staging environment on the chosen PaaS.
    - On every merge to the main branch: Automatically build the iOS application using Xcode, sign it, and upload it to TestFlight for distribution to testers. This also triggers a deployment of the backend to the production environment. This level of automation minimizes manual error, accelerates the development-test-feedback loop, and embodies professional software engineering practice.

2.2.10 Team communication and File Sharing

Discord / Microsoft Teams are for synchronous and asynchronous communication. They are the virtual space for team huddles, quick questions, pair programming sessions, and general status updates. They are for discussion about the work, not for storing the work itself.

2.3. Validation and Testing

A robust, multi-phase testing strategy will be implemented for this project to ensure that RN Sync is reliable, scalable and meets the high demands expected in clinical settings. This approach will be systematic, with testing occurring throughout the development process to ensure that results are both credible and reproducible, and that time constraints are met with ease.

2.3.1 Testing Methodology and Sequence

Testing activities are distributed across the five technical milestones to ensure the verification process goes together with implementation. Each test stage supports the corresponding deliverable for that phase, allowing issues to be resolved/discovered as early as possible. Testing will proceed through the followed structured hierarchy:

1. Unit Testing and Verification validates correctness of individual modules
    a. This will be conducted during Milestone 2 (Backend Build)
2. Integration Testing: evaluates module interoperability and flow of system
    a. This will be conducted during Milestone 3 (iOS UI Build)
3. Validation Testing: simulates real-world environments to confirm usability and performance
    a. This will be conducted during Milestone 4 (Pilot testing)
4. Evaluation and Acceptance quantify system success against predefined specifications
    a. This will be conducted during Milestone 5 (Final Integration & Deployment)

Each stage feeds forward to the next and any issues identified in lower-level testing will be resolved before advancing to the next stage.

2.3.2 Unit Testing & Verification

The implementation of unit testing for this project will employ a systematic approach that will be executed immediately after the completion of core functions. This phase intends to validate the low-level processing capabilities, such as unit conversions, timestamp normalization, and binary data deserialization, that are required by data-centric applications. All target values in

the table below are set based on realistic benchmarks from medical data and mobile streaming standards (i.e. IEEE 11073 for data accuracy, <500 ms clinical latency tolerance, and typical authentication reliability expectations).

Module	Description	Test Method	Measurement	Target Value
Data Parsing	Converts binary/string values into structured JSON packets	Send simulated binary payload with malformed data packets (random noise, missing bytes, etc.) through ingestion script. Compare parsed results to expected output.	Packet Recognition rate & crash frequency	≥ 99.9 accuracy, 0 critical crashes.
Time Synchronization	Aligns time stamps across multiple device streams	Inject two simultaneous simulated sensor streams while intentionally using different time references	Time drift (ms) after normalization	≤50 ms deviation from UTC baseline
Data Caching & Recovery	Buffers data during network disconnects and restores on reconnect	Manually simulate network interruptions; observe replay behaviour	Data retention rate	100% recovery after reconnect
Authentication	Validates secure connection and OAuth 2.0 handshake	Simulate valid/invalid login tokens; attempt 3+ failed logins	Unauthorized access attempts blocked	100% rejection of invalid attempts

Table 3: Module testing and verifying implemented features

All unit tests outputs will be logged automatically for reproducibility and versioned in logged actions.

2.3.3 Integration Testing

This phase will focus on combining the modules to test inter-component communication and synchronization. Most importantly, this phase will test the system's ability to handle the operational flow required in RN Sync.

A. Connection Dynamics and Edge Case Testing
    a. Goal
        i. The team will test the connection and disconnection ability of devices. This must happen quickly and with no manual intervention after initially setting the command.
    b. Experimental Setup
        i. The team will connect the Hub to 2+ active test streams. We will then test the connectivity by adding additional data streams while disconnecting inactive ones at variable intervals.
    c. Success Metric
        i. Pairing and unpairing must be established and terminated within 2 seconds, with 0 dropped messages post-reconnect.
B. Multi-Source Reliability
    a. Goal
        i. Evaluate the Hub's stability when connected to multiple simultaneous connections.
    b. Experimental Setup
        i. Set up a dedicated simulation board to transmit four concurrent sensor data streams via WebSocket.
    c. Success Metric
        i. The app must successfully connect to all the devices and maintain display of the sensor data without a crash or dropped connection.

C. Data Flow Integration
    a. Goal
        i. Ensure synchronization between backend data processing and frontend visualization pipeline.
    b. Experimental Setup
        i. Record timestamps at data ingestion, database commit, and UI render frame.
    c. Success Metric
        i. End to end latency ≤ 500 ms.

2.3.4 Validation Testing

This final phase of testing will confirm the product's readiness for use in the target environment, prioritizing responsiveness and user efficiency in a mock clinical setting.

A. Clinical Simulation
    a. Applicable Test
        i. Simulate the target environment (i.e. ICU) to evaluate the effectiveness of the centralized Hub.
    b. Experimental Setup
        i. The team will act as medical personnel using the Hub to monitor patients. At random times, critical event data will be injected into the patient streams.
    c. Success Metric
        i. The user must be able to identify the critical alert sent by the Hub.
        ii. Time to alert ≤ 1s.

B. Data Visualization
    a. Applicable Test
        i. Validate the display's ability to render data streams smoothly.
    b. Experimental Setup

i. Connect the Hub to multiple devices and measure the time difference between when the Hub receives the data and when data points are rendered on the iOS device's screen.

c. Success Metric

i. The display latency must be less than 500ms.

C. Data Integrity

a. Applicable Test

i. A stress test to ensure the Hub's data integrity under high loads.

b. Experimental Setup

i. Increase the simulated data to > 4 streams while increasing the update frequency.

c. Success Metric

i. Data from all streams is successfully received and written to the log with no data loss for a duration of 30 minutes.

ii. ≤ 0.01 data loss sustained for ≥ 30 minutes

2.3.5 Evaluation

The success of the project will be quantitatively evaluated based on performance in the preceding sections. This project will be considered a success if the following criteria are met:

1. Deploy a working iOS application to a device that successfully collects and visualizes mock sensor data.
2. All core logic functions achieve 99.9% accuracy and render real-time data with latency ≤ 500 ms across > 4 concurrent streams.
3. 100% uptime during simulated 30-minute operation.
4. ≥ 99.9% accurate data parsing and timestamp alignment.
5. Secure authentication and session management under all tested conditions.

2.3.6 Validation Table

Requirement / Spec	Test Method	Measurement / Metric	Target Value	Responsible Member(s)
End-to-end latency	Log ingestion, storage, and UI render timestamps	Latency (ms)	≤ 500 ms	Ben Nguyen
Reliable parsing of sensor data	Binary Data Simulation	Recognition rate (%)	≥ 99.9%	Khalid Tahir
Synchronization Accuracy	Inject time-offset data streams	Time Deviation (s)	≤ 50 ms drift	Jack Fergusson
Connection Stability	Rapid connect/disconnect simulation	Recovery Time (s)	≤ 2 s	Ziyad Soultan
Visualization Smoothness	Continuous Rendering with 4+ feeds	Frame rate (FPS)	≥ 30 FPS	Ben Nguyen
Data Integrity	30 min stress test with 6 streams	Packet loss (%)	≤ 0.01%	Khalid Tahir
Security & Access Control	Verify that user authentication tokens are validated correctly; unauthorized access attempts rejected	Unauthorized login rate (%)	0% Unauthorized Login Success	Jack & Ziyad

Table 4: Validation Table

2.4. Potential Problems and Mitigation Strategies

No.	Problem	Mitigation	Recovery Strategy
1	End-to-end latency	Performance improvements	Look into other third-party charting libraries
2	Backend Scalability and Cost	Select backend with generous free tier	Move to cheap paid tier
3	Team member unavailability	Prevent knowledge silos	Other teammates take over
4	Socket disconnection	Continuous pings	Socket reconnection routine using timing data
5	App store denial	Assure compliance, submit early	Resubmit with detailed notes
6	Insecure patient data	Encrypt sensitive data	Reconfigure backend design

Table 5: Potential problems and possible solutions

1 - End-to-end Latency: One of the most difficult technical challenges of this project is achieving a low latency for the communication between devices, and the delivery of this data to the front-end. It is critical for this product that the time between an event happening to a patient and this event being reported is as minimal as possible. To prevent wait times from becoming too long, the team will constantly look for performance improvements that can be made either in the communication between devices or the front-end data displaying process. If these performance improvements prove to be insufficient, then changes can be made to the tech stack for the project. For example, a switch could be made between WebSocket libraries that may allow for faster, more reliable communication.

2 - Backend Scalability and Cost: In the unlikely event that the application's resource usage during load testing exceeds the limits of the chosen free-tier cloud service, the team has a pre-approved plan to mitigate this. The project will transition to a low-cost paid tier, which is well within the faculty-provided $600 project budget. While this would slightly increase the overall cost of the project, this problem should not add much of a delay to development.

3 - Team Member Unavailability: The risk of project delays due to the unexpected unavailability of a team member is mitigated by the project's technology choices and management practices. The full-stack JavaScript environment ensures that all team members have a foundational understanding of the entire codebase, preventing knowledge silos. Regular code reviews, comprehensive documentation, and strict adherence to version control practices will ensure that any team member can assume critical tasks if required.

4 - Socket Disconnection: A frequent issue with data transmission using WebSockets is socket disconnection. A temporary disconnection could de-synchronize the patient data or leave gaps in historical data. Therefore, mitigating these disconnections will play a critical role in ensuring this project’s technical success. One common mitigation strategy for this is to continuously ping each transmitter and receiver in the WebSocket connection. This keeps each node awake and ready for new connections or transmissions. In the event of a disconnection despite these efforts, a comprehensive socket reconnection routine will need to be in place. A basic version of this routine would be as follows: Upon disconnection, transmitter keeps track of all data points that should have been sent, along with their timestamp. Upon reconnection, receiver gets all this data and inserts it into the historical data using the timestamps.

5 - App Store Denial: The app being denied from the app store would be disastrous for the project. One of the success metrics outlined for the final product is that it is widely accessible, and the Apple App Store is the greatest way to achieve this. To mitigate this outcome, the requirements for the App Store will be referenced at every step of the design and development process. In addition, the app will be submitted greatly ahead of the true due date, such that if the app is denied, the team has enough time to fix any issues and resubmit with detailed review notes, as suggested by Apple.

6 - Insecure Patient Data: As this project deals with sensitive data constantly, one necessary success metric is that this data is never lost, stolen, or tampered with. The team will use the guidelines of confidentiality, integrity, and availability in handling this data. For example, all sensitive data will be encrypted both in transit and at rest. If data security continues to be an

issue, the team may have to redesign the structure of the backend including the technologies being used. This would represent a significant delay in development.

3. Milestones/Division of Labor

The project schedule is organized around five key technical milestones. These milestones represent significant stages of development and are distinct from administrative deadlines such as reports and presentations, as stipulated in the project guidelines. The timeline is designed to be both ambitious and realistic, ensuring the project is completed in time for the final showcase and demonstration.

No.	**Milestone**	**Due date**	**Responsible member(s)**
1	Defined System Requirements & Architecture	Week 8	Everyone
2	Backend Build	Week 21	Khalid, Ziyad
3	iOS UI Build	Week 21	Jack
4	Pilot Testing	Week 23	Ben
5	Final Integration & Deployment	Week 25	Everyone

Milestone 1: This initial phase focuses on finalizing the detailed technical specifications, system architecture, data models, and the definitive technology stack. The output will be a comprehensive design document that serves as the blueprint for development.
Assigning this milestone to everyone on the team ensures that all members are on-board with the design decisions that are made, before progressing to further design and development. All members will make proposals for the design and architecture, then all options will be discussed, and a final decision will be made communally.

Milestone 2: This phase involves server-side development. Key deliverables include the implementation of the Node.js application, the WebSocket endpoint for real-time data ingestion, the REST API for auxiliary functions, and the integration with the chosen database solution.
Khalid and Ziyad will be primarily responsible for directing this milestone. These 2 members have greater experience with backend development, and they will leverage this experience towards building a backend that meets all the project's success criteria.

Milestone 3: This phase runs in parallel with the backend build and focuses on the client-side application. The team will develop the React Native application, including the main data visualization dashboard, the user interface for device pairing and management, and the user authentication flow.
Jack is responsible for directing the client-side development. Jack has experience in design and in development for iOS apps. This milestone is well-suited for one person, as the overall visual design choices must be cohesive.

Milestone 4: This critical phase involves the integration of the frontend and backend components into a cohesive system. The integrated application will undergo rigorous testing against the performance specifications defined in Section 4, including load testing with four concurrent streams and latency measurements. The measurements will be compared against those of the validation table to ensure that the project's goals are being reached.
Ben has been placed as the sole leader of this milestone, as it is critical that every part of the application is tested. Having one person in charge of this task allows them to focus on potential bugs and problems throughout every part of the design and development.

Milestone 5: In the final phase, any issues identified during pilot testing, finalize the application, prepare comprehensive user and technical documentation, and execute the deployment strategy will be addressed. This includes deploying the backend service to a cloud platform and distributing the iOS application via Apple's TestFlight for final evaluation.
All members of the team will be responsible for ensuring that this is completed.

A detailed project schedule, visualized as a Gantt chart, is provided in the appendix to outline the specific timelines, dependencies, and critical path for each task within these milestones.

4. Budget

4.1. Hardware BOM

All necessary hardware for development and testing is already available to the project team.

- **Apple MacBook:** Provided by team members. These are required for developing and building iOS applications using Xcode and React Native. Cost: $0.00.
- **Apple iPhone:** Provided by team members. These are essential for on-device testing, debugging, and final demonstration of the native application. Cost: $0.00.
- **Laptops:** Provided by team members. These will be used to run the scripts that generate the mock physiological data streams for testing. Cost: $0.00.

4.2. Software BOM

The selection of technologies for this project is based on a thorough analysis of the system's core requirements, particularly its need for high-performance, real-time data handling.

- **Backend - Node.js:** The project's central challenge is handling multiple, concurrent, long-running connections for real-time data streaming. While both Python and Node.js were initially considered, Node.js is the superior choice for this specific application. Node.js is built on a single-threaded, non-blocking I/O architecture, which is inherently built for managing I/O-bound tasks such as network communication with exceptional efficiency. This architecture allows a Node.js server to handle thousands of simultaneous connections with minimal resource overhead, making it ideal for scalable, real-time applications like RN Sync. In contrast, Python's standard implementation is limited by the Global Interpreter Lock (GIL), which prevents true parallelism across multiple CPU cores for a single process, making it less suitable for the team’s intentions.
- **Frontend - React Native:** The decision to use React Native for the iOS application allows the development team to leverage modern web development paradigms and a single codebase to build a truly native application. This approach accelerates the UI

development process and facilitates rapid iteration. The team's access to MacBooks and iPhones for development and testing ensures a seamless workflow within the Apple ecosystem as React Native integrates directly with Xcode for building, testing, and deployment.

- **Full-Stack Synergy:** A significant benefit of selecting Node.js for the backend is the creation of a unified, full-stack JavaScript ecosystem. With JavaScript used for both the server-side logic (Node.js) and the client-side application (React Native), the development process is streamlined. This single-language paradigm reduces the cognitive load and context-switching for developers, simplifies code sharing for common logic (e.g., data validation rules), and enhances the team's overall velocity and collaborative efficiency.

The entire software development stack consists of free and open-source tools.

- **Xcode:** Apple's integrated development environment (IDE) for all Apple platforms. It is available for free download from the Mac App Store. Cost: $0.00.
- **Node.js:** The JavaScript runtime environment for the backend server. It is open-source and free to use. Cost: $0.00.
- **React Native CLI:** The command-line interface for creating and managing React Native projects. It is open-source and free. Cost: $0.00.
- **Git & GitHub:** The version control system and hosting service that will be used for source code management. Free plans are available for academic and small team use. Cost: $0.00.
- **Visual Studio Code:** A lightweight but powerful source code editor that will be used for development. It is free. Cost: $0.00.

4.3. Infrastructure Budget

The deployment strategy is designed to be cost-effective, efficient, and suitable for an academic project, leveraging industry-standard platforms for both the frontend and backend components.

- **iOS Application Deployment:** The iOS application will be distributed to team members, the faculty supervisor, and teaching assistants for testing and final demonstration using

Apple's TestFlight platform. This is the standard method for pre-release app distribution. The process involves using Xcode to configure code signing, creating a release archive of the application, and uploading the build to App Store Connect for management and distribution. This requires enrollment in the Apple Developer Program.

- **Backend Service Deployment:** The Node.js backend application and its associated database will be deployed to a cloud-based Platform-as-a-Service (PaaS) provider. The selection will prioritize platforms that offer a generous free tier capable of supporting the project's development, testing, and demonstration needs without incurring costs. Several leading platforms are suitable:
    - **Google Firebase:** An integrated suite of backend services, including Authentication, the Firestore NoSQL database, and Cloud Functions for serverless computing. Its "Spark" free plan provides ample resources for a project of this scale.
    - **AWS Amplify:** A comprehensive platform from Amazon Web Services for building and deploying full-stack applications. Its free tier includes sufficient compute hours, data storage, and data transfer to cover the project's lifecycle.
    - **Heroku:** A developer-friendly platform known for its simple deployment workflow. Its "Eco" or "Basic" dynos offer a low-cost ($5-7 per month) hosting solution that would fall well within the project's budget should the free tier prove insufficient.

These are services necessary for collaboration and data transmission

- **Apple Developer Program License:** The team will use the free provisioning profile on Xcode for testing, but a license will be required for final deployment to the App Store as part of our deliverables. Cost: $155.00
- **Cloud Infrastructure:** As an additional measure (outside of scope) to enable data viewing outside the physical range of the sensors, the Hub will require a broker service (AWS, Firebase, etc.) to transmit sensor readings. This will be covered under the academic tiers of the cloud services. Cost: $0.00

The final platform will be selected during Milestone 1, but all identified options provide a viable path to deployment while adhering to the project's budgetary constraints.

5. Risk Mitigation, Environmental Impact, and Legal Considerations

For the early stages of the development of RN Sync, there are no environmental impact that needs considerations. However, if the app grows in popularity and becomes a widely used software, it will require a concrete scalable solution to accommodate for the large amount of data transfer per second. It will also require a robust and secure storage solution which may have environmental impacts that can contribute to global warming such as heat generation from datacenters. Since Canada is located towards the colder side of the Earth during the Winter and medical data are highly sensitive, a possible solution is to create a local datacenter at every hospital. Information will be securely protected and the energy used for the datacenter can also serve as a consistent heating solution

The development of this software ensures a consistent viewing of current and existing medical sensor data, and it should not be used for self-diagnosis. A possible health risks include the misinterpretation of missing data due to WebSocket delay or sensitive information being exposed to unauthorized users due to existing connections. As a measure against unauthorized access, authentication will disconnect the existing connection after 3+ incorrect attempts. Users should also not be entirely dependent on notifications assigned to the Hub to alert them to patients' conditions. When data is incorrect or delayed, users should contact the personnel in charge of the patients to confirm data integrity.

6. Progress to Date

Most of the progress made so far has been in making decisions for the scope, direction, and design of the project. In addition to these decisions, the team has also determined a list of specifications that will act as a set of end goals to keep the project on track.

After extensive research into industry-standard technology, as well as what will work best for the team’s specific task, a comprehensive technology stack has been finalized, covering front-end, back-end, CI/CD, and various tools.

The scope of the project and the success metrics have been determined. Having these both firmly established as the project moves forward will provide the team with a unified, well-defined goal. Each phase in the upcoming development can be evaluated as it relates to the goals set out for the final product.

Having completed the first phase of this project, the next two phases are set to begin simultaneously. These phases are front-end and back-end development. The team is prepared to take on these tasks, as all planning and decision making has been properly completed.

7. Conclusion

RN Sync delivers an iOS-based centralized monitoring hub that unifies the fragmented sensor feeds in the healthcare industry, with coherent, time-aligned, low-latency visualization on a mobile device. The design centers on time-based alignment of data, resilient reconnections, and secure transport/auth (TLS 1.3 + OAuth 2.0) backed by a modular Node.js/WebSocket pipeline and PaaS deployment, enabling sub-500 ms p95 live-view latency across at least four concurrent streams. These choices align directly with the project's interface and performance targets, including streaming latency and responsiveness, and enforce clear security and session practices for clinical-adjacent use. Execution follows a staged plan validated by measurable criteria. Upon completing the five milestones, the team will demonstrate: (1) stable real-time visualization under realistic network conditions with end-to-end latency ≤ 500 ms, (2) accurate cross-stream alignment with ≤ 50 ms drift from a UTC baseline, (3) recoverable data integrity after disconnects with rapid pairing/unpairing (≤ 2 s) and ≤ 0.01% packet loss in stress tests, and (4) a deployable TestFlight build. Collectively, these outcomes prove user value for researchers and clinicians.

Next steps are to: finalize the cloud PaaS choice and deployment shape for persistent WebSockets; lock the latency/retention thresholds tied to the ≤ 500 ms p95 and historical data plan; and complete the pilot protocol and success metrics (≥ 30 FPS rendering, ≥ 99.9% parsing/alignment accuracy, secure auth under all tested conditions). With these decisions in place, the team can confidently complete the milestones of system integration, "Clinical Simulation," and distribution, all while measuring progress against objective, auditable metrics.

References

[1] B. Dowling, M. Fischlin, F. Gunther and D. Stebila, "A Cryptographic Analysis of the," *Journal of Cryptography,* 2021.

[2] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr and J. P. Richer, "NIST," 25th October 2025. [Online]. Available: https://pages.nist.gov/800-63-3/sp800-63b.html#:~:text=4.2.&text=Periodic%20reauthentication%20of%20subscriber%20sessions,just%20before%20the%20inactivity%20timeout.. [Accessed 25th October 2025].

[3] H. L. 7, "Coordinated Universal Time: An overview," Health Level 7, [Online]. Available: https://hl7-definition.caristix.com/v2/HL7v2.4/DataTypes/TS. [Accessed 2 October 2025].

[4] F. Inc., "Single-Lead ECG Specifications Explained: Sampling Rate, Resolution, and Storage," Fibion Inc., [Online]. Available: https://web.fibion.com/articles/ecg-sampling-resolution-storage-guide/#:~:text=125%20Hz%20%C3%97%202%20bytes,sleep%20monitoring%2C%20or%20ambulatory%20studies.. [Accessed 3 October 2025].

[5] R. V. Dandu, "Storage media for computers in radiology," *COMPUTERS IN RADIOLOGY,* vol. 18, no. 4, pp. 287-289, 2008.

Appendices

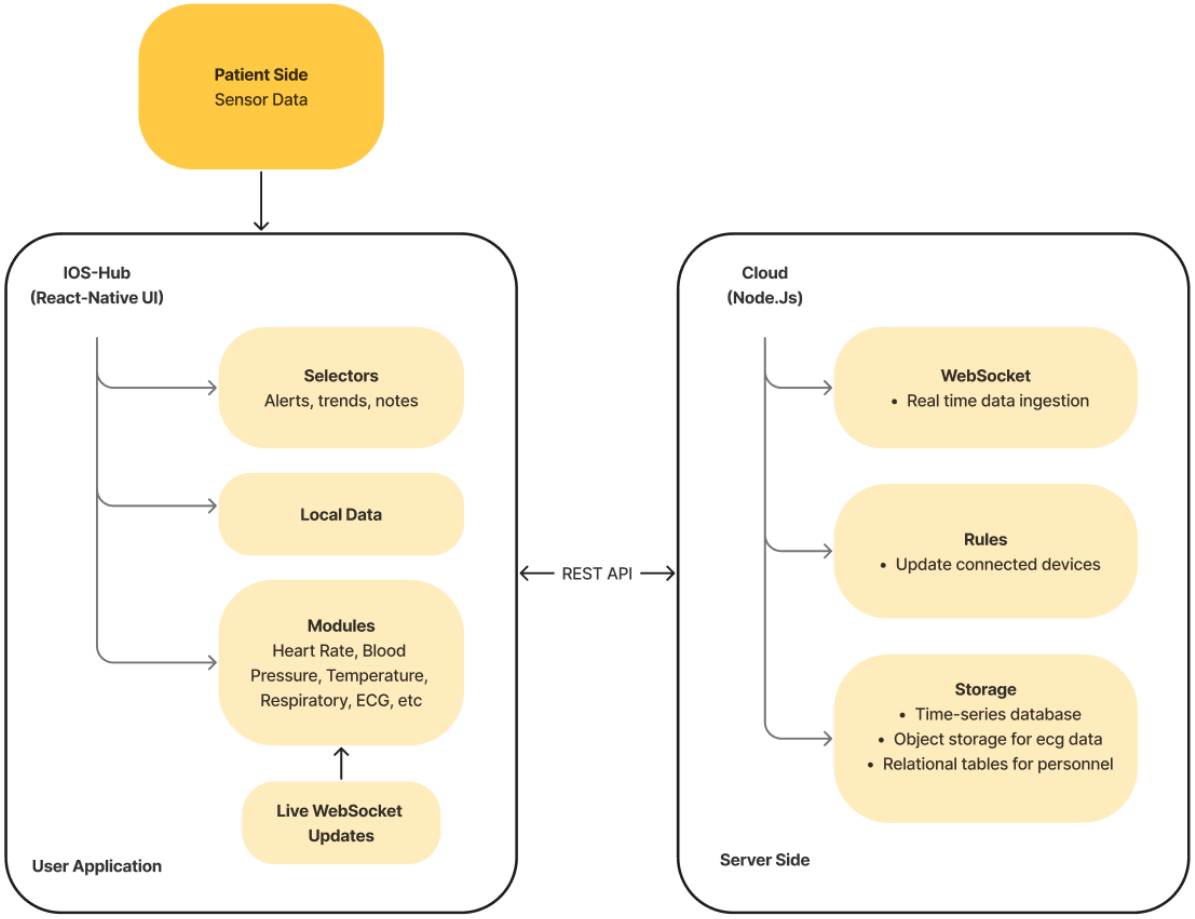


Figure 1: Data flow diagram displaying how signals move between patients, users, and the Cloud

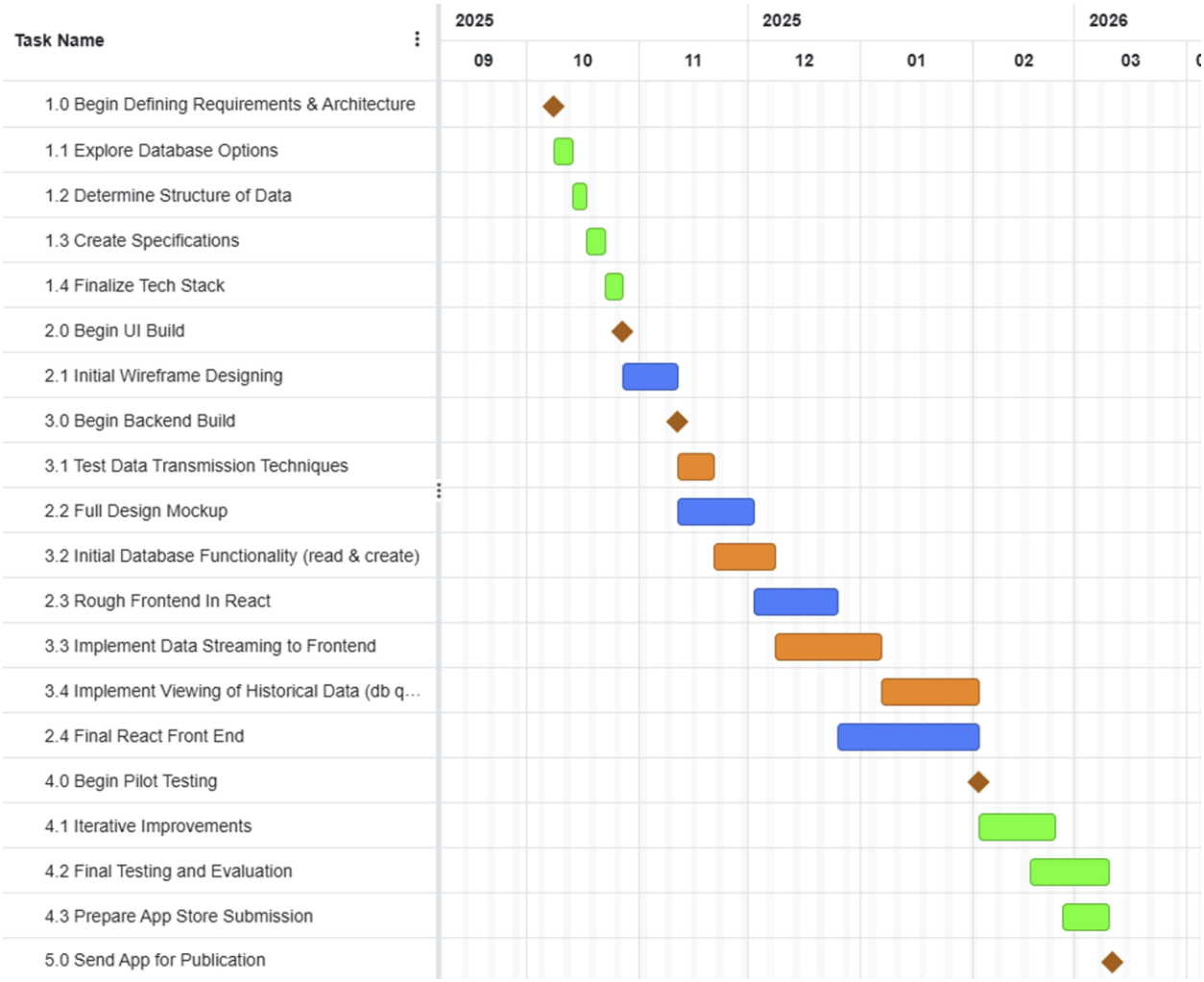


Figure 2: Project Timeline & Major Milestones

Milestone	Resources Required
Defined System Requirements & Architecture	**Personnel:** Full Team, Faculty Supervisor **Hardware:** Team Laptops **Software:** Diagramming tools (e.g., Lucidchart, draw.io, Figma), Word Processor, Git/GitHub
2. Backend Build	**Personnel:** Full Team **Hardware:** Team Laptops **Software:** Node.js, VS Code/WebStorm, Postman, Git/GitHub **Services:** Selected Cloud Hosting Platform Account (e.g., Firebase, AWS)
3. iOS UI Build	**Personnel:** Full team **Hardware:** Team MacBooks, Team iPhones **Software:** Xcode, React Native CLI, Node.js/npm, VS Code, Git/GitHub
4. Pilot Testing	**Personnel:** Full Team **Hardware:** Team MacBooks, Team iPhones, Team Laptops **Software:** Xcode, iOS Simulator, Deployed Backend & Frontend applications
5. Final Integration & Deployment	**Personnel:** Full Team **Hardware:** Team MacBooks **Software:** Xcode, Git/GitHub **Services:** Apple Developer Program Account, Finalized Cloud Hosting Platform

Table 6: Scheduling Access to Resources