< Remote Real-time integrated Healthcare platform>

<Seif Mansour>

<23012749>

UXCFXK-30-3

Digital Systems Project

Abstract:

This report presents the development of a remote healthcare platform aimed at enhancing patient care through improved medication adherence, real-time communication, and efficient appointment management. The platform built using the MERN stack (MongoDB, Express.js, React.js, Node.js) and Socket.IO for real-time communication, allows patients to track their medications, engage securely with healthcare providers, and manage appointments.

The report covers the system's design, including the identification of key user and system requirements, as well as the chosen development methodology. It explores the challenges encountered during the implementation phase, particularly in ensuring data security and user-friendly interaction. The system was thoroughly tested to meet functional and performance standards.

In conclusion, the project demonstrates how digital health tools can improve medication adherence and streamline healthcare processes, offering potential benefits for both patients and providers. Future work will focus on expanding the system's capabilities and further improving data privacy and user experience.

Acknowledgements:

Table of Contents

Table of Figures

Chapter 1 (Introduction):

1.1 What is the real-world problem?

In modern healthcare, effective communication and efficient management of personal health information are vital for improving patient outcomes, particularly for individuals managing chronic conditions. However, many patients struggle with challenges such as tracking their medications, remembering dosage schedules, and providing timely updates to their healthcare providers. These difficulties contribute significantly to medication non-adherence, a critical issue that leads to worsening health conditions, increased hospital admissions, and rising healthcare costs. Studies indicate that non-adherence rates can exceed 50% for chronic medications, emphasizing the need for innovative tools that promote consistent adherence to treatment plans (BMJ Open, 2018).

Beyond medication management, scheduling appointments and maintaining a comprehensive record of health interactions remain complex and burdensome for patients. Many existing healthcare systems lack centralized platforms where patients can view, update, and share their health data with healthcare providers in real-time. This gap may arise due to constraints like the cost of implementation and ensuring adequate security for sensitive health information. The absence of such systems burden patients and limits healthcare providers' ability to monitor adherence effectively and adjust treatment plans as needed.

To address these challenges, this project proposes the development of a comprehensive web application that integrates key healthcare management functionalities. The application will allow patients to track their medication schedules, document adherence, and book appointments seamlessly, all within a single, user-friendly system. While front-end usability is an essential focus, this project also recognizes the complexities of backend development, including secure data management, integration with existing healthcare systems, and compliance with privacy standards. By addressing these technical challenges, the platform aims to empower patients to take an active role in their health management while providing healthcare providers with real-time access to accurate, actionable data. This collaborative approach has the potential to enhance adherence, improve health outcomes, and reduce the administrative burden on both patients and providers.

1.2 Why is it important?

The importance of developing a unified healthcare management platform cannot be overstated, particularly in today's digital age where accurate and timely access to health information is essential. Non-adherence to prescribed medications is a leading cause of treatment failure, particularly for chronic illnesses such as hypertension, diabetes, and cardiovascular disease. Patients who fail to adhere to their treatment plans risk significant health complications, preventable hospitalization, and even mortality. A centralized platform

that integrates features such as medication tracking, appointment scheduling, and real-time communication with healthcare providers can directly address these issues, improving patient adherence and health outcomes.

Moreover, the healthcare industry is shifting towards patient-centered care, emphasizing active patient participation in health management. The proposed platform aligns this approach by simplifying how patients engage with their healthcare providers and manage their treatment plans. Additionally, the platform's real-time data-sharing capabilities will enable healthcare providers to monitor patient adherence more effectively, make informed treatment decisions, and reduce the likelihood of errors stemming from incomplete or inaccurate medical records.

1.3 Aims and Goals:

This project aims to:

- Develop a secure, user-friendly digital platform that empowers patients to manage their health information and actively engage in their healthcare journey.
- Allow users to securely manage their personal details, track medication adherence, and share health information with providers.
- Enhance communication between patients and healthcare providers by offering real-time access to essential health data.
- Simplify appointment scheduling, enabling patients to arrange consultations and maintain consistent follow-ups with ease.
- Promote a patient-centered approach to healthcare by addressing gaps in medication management and doctor-patient communication.
- Prioritize security, usability, and data privacy to effectively meet the needs of both patients and providers.

1.4 Literature Review Findings:

The literature review identified several critical gaps in existing healthcare platforms, including fragmented functionality, non-intuitive designs, and inadequate data security measures. These limitations hinder the adoption of digital health tools and prevent healthcare systems from delivering holistic, patient-centered care. Addressing these gaps requires a solution that integrates multiple functionalities into a single, secure, and user-friendly platform.

1.5 Expected Outcomes:

By integrating key functionalities into a unified platform, this project aims to improve medication adherence, optimize healthcare delivery, and enhance communication between patients and

providers. The platform will empower patients to take an active role in their health management while providing healthcare providers with real-time, actionable data, leading to improved health outcomes and reduced administrative burdens.

1.6 Structure of the Report:

The remainder of this report is organized as follows: Chapter 2 provides a comprehensive review of relevant research and existing tools. Following this, Chapter 3 details the requirements specification for the project. Chapter 4 then outlines the methodology and technical framework used to develop the platform. Chapter 5 presents the design of the system. Subsequently, Chapter 6 discusses the implementation of the proposed solution. Chapter 7 provides an evaluation of the project, including reflection on the testing results. Finally, Chapter 8 presents the conclusions and future work directions for the platform.

The next chapter provides a comprehensive literature review of relevant research, highlighting gaps and justifying the project.

Chapter 2 (Literature review)

2.1 Introduction:

In modern healthcare, effective communication and efficient management of personal health information are critical for improving patient outcomes, particularly for those with chronic conditions. Challenges such as tracking medications, adhering to dosage schedules, and providing timely updates to healthcare providers contribute to significant issues, including medication non-adherence. Non-adherence rates, exceeding 50% for chronic treatments, result in deteriorating health, increased hospitalizations, and higher healthcare costs (BMJ Open, 2018).

Additionally, current healthcare systems often lack centralized, user-friendly platforms that enable patients to manage their health records, schedule appointments, and communicate with providers in real time. These gaps increase the burden on patients and hinder healthcare providers from making timely, informed decisions.

This project addresses these challenges through the development of an integrated web application. The platform combines medication tracking, real-time data sharing, and streamlined appointment scheduling into a single, secure system. By empowering patients and enhancing provider-patient collaboration, the platform aims to improve adherence, optimize health outcomes, and reduce administrative inefficiencies.

2.2 Thematic Sections:

2.2.1 Medication Adherence Across Healthcare Contexts

Significance of Adherence
Medication adherence, defined as the extent to which patients follow prescribed treatment regimens, is crucial for achieving optimal health outcomes. Non-adherence can lead to treatment failures, increased hospitalizations, and higher healthcare costs. For instance, a meta-analysis identified that medication non-adherence incurred a higher risk of hospital admissions and mortality (Kardas et al., 2020).

Challenges in Medication Adherence
Patients encounter various barriers to adherence. Complex medication regimens can overwhelm patients, often resulting in missed doses. Additionally, side effects associated with medications discourage continued use for many individuals. Forgetfulness is another significant factor, particularly among the elderly or those managing multiple medications simultaneously. Furthermore, a lack of understanding regarding the importance of adherence or the consequences of non-adherence remains a critical challenge. An overview of systematic reviews highlighted that non-adherence is a major barrier to effective healthcare delivery (Nieuwlaat et al., 2014).

Impact of Non-Adherence

The consequences of non-adherence are substantial, affecting both health outcomes and healthcare systems. Patients often experience health deterioration due to ineffective treatment, leading to increased hospitalizations and higher healthcare utilization costs. Economically, non-adherence imposes a significant burden on healthcare systems, as highlighted by studies linking it to preventable expenses and resource strain (Cutler et al., 2018).

Existing Tools and Gaps

Several interventions have been developed to improve medication adherence. Digital reminders, such as mobile apps and electronic devices, have been employed to prompt patients to take their medications. Patient education programs have also been implemented to enhance understanding of treatment regimens. However, these tools often face limitations. Usability issues, such as complex interfaces, deter patient engagement, and many tools lack seamless integration with healthcare providers' systems, hindering coordinated care. A narrative review emphasized the need for innovative and effective interventions tailored to the individual needs of patients (Kardas et al., 2020).

Addressing these challenges requires solutions that prioritize user-friendliness, integration, and personalized support to cater to diverse patient populations.

2.2.2 Applications for Digital Health Technologies

Overview of Digital Health Platforms

Digital health technologies encompass a wide range of tools designed to enhance healthcare delivery and improve patient outcomes. These tools include telemedicine services, electronic health records (EHRs), wearable devices, and mobile health applications. For instance, platforms like HealthHero provide virtual consultations, enabling patients to access medical advice remotely and enhancing accessibility and convenience (HealthHero, 2023).

Benefits of Digital Health

The integration of digital technologies offers numerous benefits to both patients and healthcare providers. Patients can consult healthcare providers remotely, reducing the need for travel and minimizing disruptions to daily life. Wearable devices and mobile apps facilitate continuous health monitoring, enabling early detection of potential health issues. Additionally, digital tools empower patients by providing access to their health data, promoting informed decision-making and active participation in their care. The World Health Organization (WHO) highlights that digital health improves the efficiency and sustainability of healthcare systems while ensuring good quality, affordable, and equitable care (WHO, 2021).

Limitations of Existing Technologies

Despite their advantages, digital health technologies face several limitations. Usability challenges, such as complex interfaces and technical difficulties, hinder user engagement, particularly for individuals who are less technologically adept. Concerns about data privacy

and security also pose significant barriers, as the handling of sensitive health information raises issues related to confidentiality and trust. Moreover, many digital tools lack integration with existing healthcare systems, resulting in fragmented care and inefficiencies. A report by The King's Fund underscores the complexity of implementing large-scale digital changes in health and social care, highlighting the importance of addressing these challenges to ensure success (The King's Fund, 2020).

To overcome these barriers, digital health solutions must focus on user-centric designs, robust data security measures, and seamless integration with healthcare systems to maximize their potential and adoption.

2.2.3 Appointment Scheduling and Communication Tools

Importance of Efficient Appointment Scheduling

Efficient appointment scheduling is crucial in healthcare for optimizing patient flow, enhancing resource utilization, and improving overall care delivery. Accurate scheduling ensures that patients receive timely medical attention, which is essential for effective treatment outcomes. Moreover, well-organized scheduling minimizes patient wait times and maximizes the utilization of healthcare providers' time, leading to increased patient satisfaction and better clinical efficiency (Hughes, 2024).

Challenges in Current Systems

Despite its importance, many healthcare facilities face challenges in implementing effective scheduling systems. Common issues include overbooking, underutilization of appointment slots, and high rates of patient no-shows. These challenges can lead to increased operational costs, reduced patient satisfaction, and strained provider-patient relationships. Barriers to adopting efficient scheduling systems often involve the costs associated with implementing new technologies and the complexity of integrating these systems into existing workflows (Hughes, 2024).

Digital Solutions for Communication

The advent of digital communication tools has the potential to address many of these scheduling challenges. Platforms offering features such as automated appointment reminders, patient self-scheduling, and real-time communication between patients and providers can significantly reduce no-show rates and enhance scheduling efficiency. For instance, the NHS has been promoting the use of digital tools to improve patient engagement and streamline communication, aiming to make health systems more efficient and sustainable (NHS England, 2023).

By leveraging these digital solutions, healthcare providers can improve appointment adherence, optimize resource allocation, and enhance the overall patient experience.

2.2.4 Data Sharing and Security in Healthcare Systems

The Role of Real-Time Data Sharing

Real-time data sharing is essential for timely and effective patient care. It allows healthcare providers to access up-to-date information, enabling better decision-making and treatment coordination. For instance, the NHS emphasizes data and clinical record sharing to enhance integrated care (NHS England, 2023).

Challenges in Data Security

Managing sensitive health data poses significant security risks, including breaches that can result in privacy violations and misuse. The World Economic Forum highlights cyberattacks and illegal data sharing as key concerns (World Economic Forum, 2022).

Solutions for Secure Data Sharing

To address these challenges, measures such as encryption, authentication, and compliance with regulations like GDPR and HIPAA are critical. Tools like the NHS Data Security and Protection Toolkit help ensure organizations meet data security standards (NHS Digital, 2024). Emerging technologies, such as blockchain, offer decentralized and tamper-resistant data sharing, enhancing security (Kumar et al., 2023).

By implementing these strategies, healthcare systems can safeguard patient data and ensure secure, efficient information sharing.

2.3 Gaps in Literature

Despite advancements in digital health technologies, several critical gaps persist, underscoring the need for an innovative and integrated solution.

Fragmented Functionality
Many existing healthcare platforms focus on single-use cases, such as medication tracking, appointment scheduling, or teleconsultation. This lack of integration forces users to juggle multiple applications, resulting in inefficiencies and disjointed care delivery. Consequently, healthcare providers struggle to provide holistic and coordinated services (Hughes, 2024; NHS England, 2023).

Usability Challenges
Non-intuitive interfaces remain a common limitation in many digital health tools, making them difficult to navigate, particularly for elderly patients and individuals with limited digital

literacy. Usability issues reduce engagement and adherence, ultimately negating the benefits these tools aim to provide (World Economic Forum, 2022). Overly complex applications exacerbate non-adherence, further diminishing their effectiveness.

Security and Privacy Concerns

The sensitive nature of health data necessitates robust security measures. However, healthcare platforms frequently fail to implement end-to-end encryption and secure access protocols, leaving them vulnerable to breaches and cyberattacks. Although regulations like GDPR and HIPAA provide clear guidelines, inconsistent implementation undermines patient trust and confidentiality (NHS Digital, 2024; Kumar et al., 2023).

Limited Real-Time Data Sharing

Real-time communication between patients and providers is essential for timely interventions and optimal care. Unfortunately, many platforms lack real-time data-sharing capabilities, leading to delays in updates, suboptimal monitoring of adherence, and inadequate responses to patient needs (NHS England, 2023).

Barriers to Adoption

The high cost and complexity of implementing digital health tools deter adoption among both patients and healthcare providers. Limited integration into existing systems further restricts access, particularly in low-resource settings where budgets are constrained (World Economic Forum, 2022).

Neglected Patient-Centered Care Models

Few platforms effectively align with patient-centered care principles. Limited customization options, lack of shared decision-making tools, and failure to address diverse patient needs prevent users from fully engaging with these systems. This gap highlights the necessity of designing platforms that cater to individual preferences and healthcare requirements (Hughes, 2024).

By addressing these gaps, digital health technologies can move closer to delivering efficient, accessible, and patient-centered solutions.

2.4 Proposed Solutions to Address Gaps

Integrated Healthcare Platform

The proposed solution is a comprehensive healthcare management platform that seamlessly integrates medication adherence tools, appointment scheduling, real-time data sharing, and secure communication into a single system. APIs will enable seamless integration with wearable devices and existing healthcare systems, ensuring coordinated care and real-time monitoring.

User-Centric Design

A user-friendly interface will prioritize accessibility and ease of use, addressing the

challenges faced by individuals with limited digital literacy. Features such as visual aids, personalized dashboards, and automated notifications will enhance engagement and adherence.

Secure Data Sharing Mechanisms
The platform will employ end-to-end encryption and blockchain technology to ensure secure and tamper-proof data sharing. Compliance with GDPR and HIPAA regulations will uphold ethical and legal standards, while a cloud-based infrastructure will facilitate real-time updates for timely and informed decision-making.

Advanced Appointment Scheduling System
An optimized appointment scheduling module will include automated reminders, real-time calendar updates, and patient self-scheduling. These features aim to reduce no-show rates, improve resource allocation, and enhance the overall patient experience. Integration with providers' schedules will ensure seamless coordination.

Personalized Patient Support
AI-driven tools will analyze patient data to deliver tailored recommendations, reminders, and educational materials. By addressing individual needs, these tools will empower patients to manage their conditions effectively, improving adherence and health outcomes.

2.5 Conclusion

This literature review highlights several critical challenges in healthcare management, including medication non-adherence, inefficient appointment scheduling, limited real-time data sharing, and significant data security concerns. While existing digital health technologies have attempted to address these issues, they often lack integration, user-friendliness, and robust security measures, which hinders their widespread adoption and effectiveness.

The proposed solution, an integrated healthcare management platform, aims to bridge these gaps by unifying medication tracking, appointment scheduling, real-time communication, and secure data sharing. By prioritizing usability, affordability, and adherence to data protection standards, this platform will empower patients, enhance provider-patient collaboration, and optimize healthcare delivery.

By leveraging advanced technologies such as APIs, blockchain, and AI-driven tools, the platform addresses critical barriers to adoption, including usability challenges, fragmented functionality, and data security concerns. The findings from this literature review provide a solid foundation for the development of this innovative solution. The next chapter will outline the methodology and technical framework guiding its implementation.

Chapter 3: Requirements Specification

3.1 Introduction

This chapter details the requirements for the proposed remote healthcare system. The primary goal of this project is to develop a platform that effectively connects patients and doctors, facilitating appointment management, medication tracking and adherence monitoring, and secure communication. The requirements outlined below define the specific functionalities the system must provide (**Functional Requirements**) and the quality attributes and constraints under which it must operate (**Non-Functional Requirements**).

The requirements were elicited through an analysis of the core problem statement, definition of the primary user roles (Patient and Doctor), and the design of key user workflows based on common telehealth practices. Techniques included iterative prototyping of the user interface (as shown in Chapter 4) and consideration of typical tasks within a patient-doctor remote interaction scenario. These requirements form the basis for the system's design, implementation, and subsequent evaluation. Prioritization using the MoSCoW method (Must have, Should have, Could have, Won't have this time) has been applied to indicate the relative importance of each requirement for this project iteration. The requirements are intended to be specific, measurable, achievable, relevant, and time-bound (SMART) where applicable, and serve as the foundation for system testing.

A Use Case diagram is presented in Section 3.2.1 to provide a high-level functional overview.

3.2 Functional Requirements

Functional Requirements (FRs) specify what the system must *do*. They define the features, tasks, and operations that users can perform. The following FRs have been identified and prioritized for the remote healthcare system:

Table 3.1: Summary of Functional Requirements (FRs)

ID	Requirement Description	Priority	Section
FR1	Allow user registration (Patient/Doctor) with details.	M	3.2.2
FR2	Validate user registration details (email, password, etc.).	M	3.2.2
FR3	Allow registered users to log in via email/password.	M	3.2.2
FR4	Provide mechanism for forgotten password reset.	S	3.2.2
FR5	Restrict system access based on user role (Patient/Doctor).	M	3.2.2
FR6	Allow patient to view own appointment list (with status).	M	3.2.3

FR7	Allow patient to initiate new appointment booking process.	M	3.2.3
FR8	Allow patient to select preferred date for appointment.	M	3.2.3
FR9	Allow patient to select doctor from their registered city.	M	3.2.3
FR10	Present patient with available time slots for chosen doctor/date.	M	3.2.3
FR11	Allow patient to add optional notes during booking.	M	3.2.3
FR12	Allow patient to submit appointment request.	M	3.2.3
FR13	Allow patient to cancel own pending or confirmed appointment.	M	3.2.3
FR14	Allow patient to view/respond to doctor's edit requests.	S	3.2.3
FR38	Notify patient about appointment status changes.	S	3.2.3
FR15	Allow patient to view their prescribed medication list.	M	3.2.4
FR16	Provide calendar interface for patient medication schedule view.	M	3.2.4
FR17	Allow patient to view specific doses for a selected day.	M	3.2.4
FR18	Allow patient to mark a scheduled medication dose as "Taken".	M	3.2.4
FR19	Provide visual confirmation when dose marked as "Taken".	M	3.2.4
FR20	Allow real-time chat between linked patients and doctors.	M	3.2.5
FR21	Allow users to view their list of active chat conversations.	M	3.2.5
FR22	Allow users to view chat history for a selected conversation.	M	3.2.5
FR23	Persist chat message history for later viewing.	M	3.2.5
FR36	Allow users to delete a chat conversation from their view.	S	3.2.5
FR37	Provide mechanism to initiate a new chat conversation.	M	3.2.5
FR24	Allow doctors to define/manage their available time slots.	M	3.2.6
FR25	Allow doctors to view their appointment schedule (filterable).	M	3.2.6
FR39	Allow doctors to view incoming pending appointment requests.	M	3.2.6
FR40	Allow doctors to accept a pending appointment request.	M	3.2.6
FR41	Allow doctors to deny a pending appointment request.	M	3.2.6

FR42	Allow doctors to cancel a confirmed appointment.	M	3.2.6
FR43	Allow doctors to propose an edit to an appointment.	S	3.2.6
FR26	Allow doctors to view their list of registered patients.	M	3.2.6
FR44	Allow doctors to view detailed profile/history for a patient.	M	3.2.6
FR45	Notify doctors about new appointment requests/cancellations.	S	3.2.6
FR32	Require doctor to select a patient before managing meds.	M	3.2.7
FR27	Allow doctors to add/edit/delete prescriptions for patient.	M	3.2.7
FR33	Provide form for prescribing/editing medication details.	M	3.2.7
FR34	Allow doctors to specify medication frequency.	M	3.2.7
FR35	Allow doctors to specify multiple intake times per day.	M	3.2.7
FR28	Allow doctors to view patient medication adherence log.	M	3.2.7
FR30	Present role-specific dashboard upon login.	M	3.2.8
FR31	Include placeholder section for Lifestyle/Nutrition info.	C	3.2.8
FR46	Provide access to emergency contact support information.	C	3.2.8

3.2.1 Use Case Overview

To provide a high-level overview of the system's intended functionality from the perspective of its primary users, a Use Case diagram is presented in Figure 3.1. This diagram identifies the main actors interacting with the system (Patient and Doctor) and the key tasks or goals they aim to achieve.
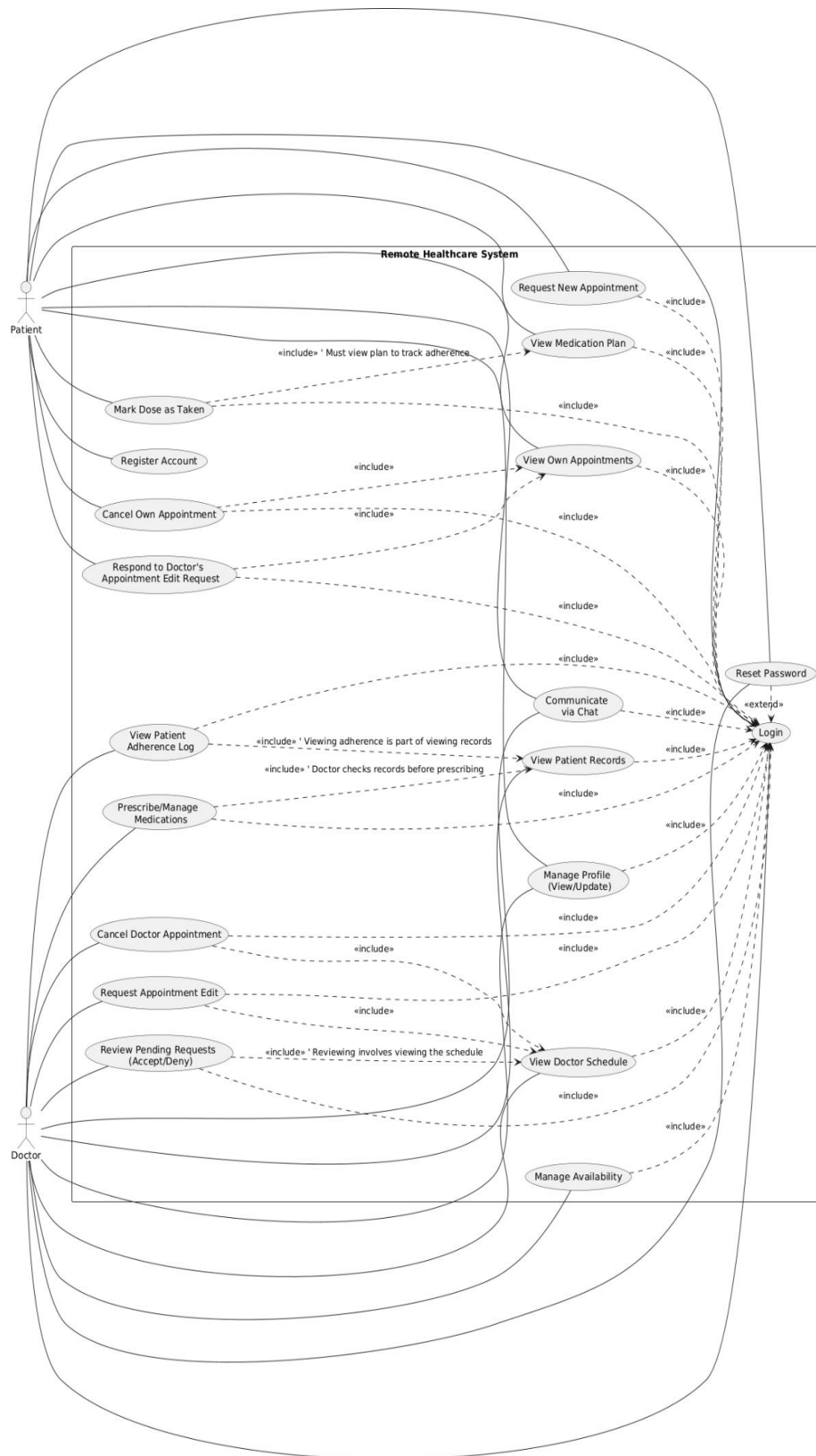
Figure 3.1 System Use Case Diagram

As illustrated in Figure 3.1, the Use Case diagram provides a high-level overview of the system's intended functionality from the perspective of its primary users (Patient and Doctor). It identifies the main tasks they perform.

Patients' primary use cases involve managing their own appointments, viewing medication plans and tracking adherence, and communicating via chat. Doctors' use cases include managing availability, reviewing and managing appointments, prescribing medications, viewing patient records and adherence logs, and communicating via chat. Both actors also interact with core authentication functionalities like registration and login, as well as password reset.

The diagram utilizes relationships such as <<include>> and <<extend>> to show dependencies and optional flows between these tasks. This Use Case model provides essential context for the detailed breakdown of specific Functional Requirements (FRs) presented in the following subsections.

3.2.2 Authentication and User Management

Secure access and clear role definition are fundamental to a healthcare system handling sensitive data.

- **FR1: Register User (Patient/Doctor)**

    o *Description:* The system shall allow users to register by providing First Name, Last Name, Email, Password, Location (City), and selecting a Role (Patient or Doctor).

    o *Priority:* **M** (Must have)

- **FR2: Validate Registration Details**

    o *Description:* The system shall validate user registration details (e.g., email format, password complexity rules, matching passwords).

    o *Priority:* **M** (Must have)

- **FR3: Login with Email/Password**

    o *Description:* The system shall allow registered users to log in using their email and password.

    o *Priority:* **M** (Must have)

- **FR4: Forgotten Password Mechanism**

    o *Description:* The system shall provide a mechanism for users to reset their forgotten password.

    o *Priority:* **S** (Should have)

- **FR5: Role-Based Access Control**

- o   *Description:* The system shall restrict access to features and data based on the logged-in user's role (Patient or Doctor).

  - o   *Priority:* **M** (Must have)

- **Justification:** Requirements FR1-FR5 ensure secure, role-based access for patients and doctors. FR1 and FR3 provide basic system access, while FR2 safeguards registration data integrity. FR5 is key for maintaining data privacy, ensuring patients only access their own data and doctors only see their assigned patients. FR4, though secondary, improves usability by allowing users to reset forgotten credentials. Together, these requirements are vital for secure access and data protection in the system.

3.2.3 Patient Appointment Management

A core function of the system is enabling patients to manage their consultations with doctors effectively. This involves viewing existing appointments, requesting new ones, and handling cancellations or changes.

- **FR6: Patient Views Own Appointments List**

  - o   *Description:* The system shall allow patients to view a list of their upcoming and past appointments, including date, time, doctor name, notes, and current status (e.g., Pending, Confirmed, Denied, Cancelled).

  - o   *Priority:* **M** (Must have)

- **FR7: Patient Initiates New Appointment Booking**

  - o   *Description:* The system shall allow patients to initiate the booking of a new appointment.

  - o   *Priority:* **M** (Must have)

- **FR8: Patient Selects Preferred Date**

  - o   *Description:* The system shall allow patients to select a preferred date for a new appointment request.

  - o   *Priority:* **M** (Must have)

- **FR9: Patient Selects Doctor from City**

  - o   *Description:* The system shall present patients with a list of available doctors within their registered city to choose from when booking.

  - o   *Priority:* **M** (Must have)

- **FR10: Patient Selects Available Time Slot**

  - o   *Description:* The system shall present patients with available time slots based on the selected doctor's published schedule for the chosen date.

  - o   *Priority:* **M** (Must have)

- **FR11: Patient Adds Optional Notes**

  - *Description:* The system shall allow patients to add optional notes (e.g., reason for visit) when requesting an appointment.

  - *Priority:* **M** (Must have)

- **FR12: Patient Submits Appointment Request**

  - *Description:* The system shall allow patients to submit an appointment request based on the selected details.

  - *Priority:* **M** (Must have)

- **FR13: Patient Cancels Own Appointment**

  - *Description:* The system shall allow patients to cancel their own Confirmed or Pending appointment request.

  - *Priority:* **M** (Must have)

- **FR14: Patient View/Respond to Doctor's Edit Request**

  - *Description:* The system shall allow patients to view and respond (accept/reject) to appointment edit requests initiated by the doctor.

  - *Priority:* **S** (Should have)

- **FR38: Patient Notifications (Status Changes)**

  - *Description:* The system shall notify patients (e.g., via an in-app notification area or similar) about changes to their appointment status (Confirmed, Denied, Cancelled, Edit Requested).

  - *Priority:* **S** (Should have)

- **Justification:** This set of requirements (FR6-FR14, FR38) directly addresses the core project goal of facilitating patient-doctor interactions. Providing patients with the ability to request, view, and manage their appointments (FR6-FR13) is fundamental to the system's utility and reflects essential patient needs in a remote healthcare context. Filtering doctors by city (FR9) implements a specific project scope constraint. While the core request workflow (FR7-FR12) is 'Must Have', features like responding to doctor edits (FR14) and asynchronous notifications (FR38) are prioritized as 'Should Have' as they enhance the workflow but are not strictly essential for the primary request/confirm/cancel cycle. These requirements are designed to be testable, ensuring, for example, that a patient can successfully submit a request and later view its status accurately (FR12, FR6). This aligns with the need for user-centric design in telehealth platforms discussed in the literature review.

3.2.4 Patient Medication Tracking

Effective medication management and adherence are critical aspects of healthcare, particularly for chronic conditions often managed remotely. This functionality empowers patients to track their prescribed treatments and provides data for clinical review.

- **FR15: Patient Views Prescribed Medication List**

  - *Description:* The system shall display a list of medications prescribed to the patient, including name, dosage, frequency, start/end dates, and any relevant doctor's notes.

  - *Priority:* **M** (Must have)

- **FR16: Patient Uses Calendar to View Schedule**

  - *Description:* The system shall provide a calendar interface allowing patients to navigate and view their medication schedule by date.

  - *Priority:* **M** (Must have)

- **FR17: Patient Views Doses for Selected Day**

  - *Description:* The system shall display the specific medication doses scheduled for a selected day, including the scheduled time and medication details.

  - *Priority:* **M** (Must have)

- **FR18: Patient Marks Dose as "Taken"**

  - *Description:* The system shall allow patients to mark a scheduled medication dose as "Taken".

  - *Priority:* **M** (Must have)

- **FR19: System Indicates "Taken" Doses**

  - *Description:* The system shall provide clear visual confirmation (e.g., changing status text/icon) when a dose has been marked as "Taken".

  - *Priority:* **M** (Must have)

- **Justification:** The requirements FR15-FR19 focus on enabling patients to manage their medications effectively. These features include providing a clear list of prescribed medications (FR15) and a daily medication schedule (FR16, FR17) to reduce confusion. The ability to track adherence by marking doses as taken (FR18, FR19) empowers patients and generates valuable data for doctors (FR28), potentially improving treatment outcomes. This group of requirements is essential for medication tracking and is a key aspect of improving adherence through digital tools, as highlighted in telehealth literature. All these requirements are considered 'Must Have' as they are fundamental to the system's medication management functionality.

3.2.5 Communication (Chat)

Direct communication between patients and doctors is a cornerstone of this telehealth system, facilitating quick queries and follow-ups outside of formal appointments.

- **FR20: Initiate/Participate in Chat**

    o *Description:* The system shall allow both patients and doctors to initiate and participate in real-time text-based chat conversations with users linked to their account (e.g., patient with their doctor(s), doctor with their patients).

    o *Priority:* **M** (Must have)

- **FR21: View List of Conversations**

    o *Description:* The system shall display a list of the user's active chat conversations, typically identifying the other participant.

    o *Priority:* **M** (Must have)

- **FR22: View Chat History**

    o *Description:* The system shall display the chat history for a selected conversation in chronological order, showing messages from both participants.

    o *Priority:* **M** (Must have)

- **FR23: Persist Chat History**

    o *Description:* The system shall save chat messages so that the history is available for later viewing by both participants.

    o *Priority:* **M** (Must have)

- **FR36: Delete Chat Conversation**

    o *Description:* The system shall allow users to delete a chat conversation from their view. *(Note: Consider if this deletes for both users or just one).*

    o *Priority:* **S** (Should have)

- **FR37: Initiate New Chat Conversation**

    o *Description:* The system shall provide a mechanism for users to start a new chat conversation with a permitted contact.

    o *Priority:* **M** (Must have)

- **Justification:** Secure and persistent communication (FR20-FR23, FR37) is essential for enabling effective remote consultation and support, fulfilling a primary objective of the project. The ability to review past conversations (FR22, FR23) aids continuity of care. Deleting conversations (FR36) is a secondary management feature.

3.2.6 Doctor Patient and Appointment Management

To enable the doctor's side of the remote healthcare interaction, the system must provide tools for managing their schedule, handling patient appointment requests, and accessing patient information.

- **FR24: Doctor Manages Availability Slots**

  - *Description:* The system shall allow doctors to define and manage their weekly available time slots during which patients can request appointments.

  - *Priority:* **M** (Must have)

- **FR25: Doctor Views Appointment Schedule**

  - *Description:* The system shall allow doctors to view their schedule of booked appointments, filterable by status (e.g., Pending, Confirmed, Past).

  - *Priority:* **M** (Must have)

- **FR39: Doctor Views Pending Requests**

  - *Description:* The system shall clearly display incoming Pending appointment requests from patients requiring action.

  - *Priority:* **M** (Must have)

- **FR40: Doctor Accepts Request**

  - *Description:* The system shall allow doctors to Accept a Pending appointment request, changing its status to Confirmed.

  - *Priority:* **M** (Must have)

- **FR41: Doctor Denies Request**

  - *Description:* The system shall allow doctors to Deny a Pending appointment request, changing its status to Denied.

  - *Priority:* **M** (Must have)

- **FR42: Doctor Cancels Confirmed Appointment**

  - *Description:* The system shall allow doctors to Cancel a previously Confirmed appointment, changing its status and notifying the patient.

  - *Priority:* **M** (Must have)

- **FR43: Doctor Proposes Edit to Appointment**

  - *Description:* The system shall allow doctors to propose an edit (e.g., new time/date) to a Pending or Confirmed appointment, initiating a request to the patient.

  - *Priority:* **S** (Should have)

- **FR26: Doctor Views Patient List**

- o *Description:* The system shall allow doctors to view a list of their registered patients (filtered by city), showing key identifiers.

  o *Priority:* **M** (Must have)

- **FR44: Doctor Views Detailed Patient Profile/History**

  o *Description:* The system shall allow doctors to select a patient from their list to view a detailed profile, including relevant history (e.g., past appointments, medications).

  o *Priority:* **M** (Must have)

- **FR45: Doctor Notifications (New Requests)**

  o *Description:* The system shall notify doctors (e.g., via an in-app notification area) about new appointment requests or patient cancellations requiring attention.

  o *Priority:* **S** (Should have)

- **Justification:** Requirements FR24-FR26 and FR39-FR45 provide doctors with essential tools for managing their workflow. FR24 defines availability for patient bookings, while FR25, FR39-FR42 manage appointment requests. FR26 and FR44 allow access to patient lists and profiles, crucial for informed decision-making. FR43 (edit proposals) and FR45 (notifications) add flexibility but are secondary to core functions.

3.2.7 Doctor Medication Management

Effective management of patient medications is a critical clinical function supported by the system. This includes prescribing, reviewing, and monitoring patient adherence to treatment plans.

- **FR32: Doctor Selects Patient for Meds**

  o *Description:* The system shall require doctors to select a specific patient from their list before managing medications for that individual.

  o *Priority:* **M** (Must have)

- **FR27: Doctor Adds/Edits/Deletes Prescriptions**

  o *Description:* The system shall allow doctors to add new prescribed medications, edit existing ones, and delete prescriptions for the selected patient.

  o *Priority:* **M** (Must have)

- **FR33: Provide Form for Prescribing/Editing**

  o *Description:* The system shall provide a structured form (modal/page) for entering or editing medication details (Name, Dosage, Frequency, Times, Dates).

  o *Priority:* **M** (Must have)

- **FR34: Specify Medication Frequency**

  o *Description:* The system shall allow doctors to specify the frequency of medication intake (e.g., daily, every X days, specific days).

- o *Priority:* **M** (Must have)

- **FR35: Specify Multiple Intake Times**

  - o *Description:* The system shall allow doctors to specify one or more specific times per day for medication intake.

  - o *Priority:* **M** (Must have)

- **FR28: Doctor Views Patient Adherence Log**

  - o *Description:* The system shall allow doctors to view the selected patient's medication adherence log, displaying which doses were marked as "Taken" within a specified date range.

  - o *Priority:* **M** (Must have)

- **Justification:** Requirements FR27, FR28, and FR32-FR35 equip doctors with tools to manage patient treatment plans. FR32 ensures patient selection for safety, while FR27, FR33-FR35 allow for digital prescribing and modification of medications, streamlining workflows. FR28 provides access to patient-reported adherence data, offering insights into treatment effectiveness and supporting informed clinical adjustments.

3.2.8 Dashboard and Other Features

This section covers overarching navigational elements and supplementary features included in the system design.

- **FR30: Role-Specific Dashboard**

  - o *Description:* The system shall present users with a dashboard upon successful login, providing shortcuts and access to functionalities relevant to their specific role (Patient or Doctor).

  - o *Priority:* **M** (Must have)

- **FR31: Lifestyle & Nutrition Section**

  - o *Description:* The system shall include a section placeholder intended for displaying "Lifestyle & Nutrition" information or tips. *(Functionality of content management is out of scope).*

  - o *Priority:* **C** (Could have)

- **FR46: Emergency Contact Support Information**

  - o *Description:* The system shall provide users with access to emergency contact support information (e.g., links/phone numbers for local emergency services or relevant helplines). *(Functionality is linking/displaying static info).*

  - o *Priority:* **C** (Could have)

- **Justification:** A role-specific dashboard (FR30) is essential for providing a clear and efficient entry point for users to access relevant system features. The Lifestyle & Nutrition (FR31) and Emergency Contact (FR46) sections are included as indicators of desirable future enhancements or standard considerations in health applications, demonstrating broader awareness, but their core functionality is not central to the primary goals of this specific project iteration.

3.3 Non-Functional Requirements

Non-Functional Requirements (NFRs) define the quality attributes, constraints, and overall characteristics of the system. They specify *how* the system should perform its functions, rather than *what* functions it performs. These are crucial for user satisfaction, system stability, and security.

Table 3.2: Summary of Non-Functional Requirements (NFRs)

ID	Requirement Description	Priority	Section
NFR1	UI shall be intuitive, easy to navigate, and consistent.	M	3.3.1
NFR2	Patient appointment booking workflow <= 5 steps.	S	3.3.1
NFR3	Patient marking dose taken workflow <= 3 clicks.	S	3.3.1
NFR4	Provide clear and immediate visual feedback for user actions.	M	3.3.1
NFR21	Doctor prescription workflow shall be clear and efficient.	M	3.3.1
NFR19	Appointment status shall be clearly displayed.	M	3.3.1
NFR5	Dashboard load time approx. <= 3 seconds.	S	3.3.2
NFR6	Chat message delivery/display approx. <= 1 second.	M	3.3.2
NFR7	Frequent list load time approx. <= 2 seconds.	S	3.3.2
NFR18	Adherence log fetch time approx. <= 3 seconds.	S	3.3.2
NFR8	All data transmission shall be encrypted (HTTPS/TLS).	M	3.3.3
NFR9	Passwords shall be securely hashed and salted (not plaintext).	M	3.3.3
NFR10	Enforce strict data segregation (users access only own data).	M	3.3.3
NFR17	Doctor access restricted to linked/authorized patients.	M	3.3.3
NFR11	All relevant API endpoints shall require authentication.	M	3.3.3
NFR12	Input data shall be sanitized/validated (client & server).	M	3.3.3

NFR13	Core functionalities shall aim for high availability (~99%).	S	3.3.4
NFR14	Chat messages shall be reliably persisted (no data loss).	M	3.3.4
NFR20	Implemented notifications shall be delivered reliably.	S	3.3.4
NFR15	Source code shall adhere to coding standards/comments.	S	3.3.5
NFR16	Web app shall be compatible with major modern browsers.	M	3.3.5

3.3.1 Usability

The system must be easy to learn and use for both patients (who may have varying levels of technical literacy) and doctors (who need efficient workflows).

- **NFR1: Intuitive/Consistent UI**

  - *Description:* The user interface shall be intuitive, easy to navigate, and maintain consistency in design and terminology across different sections of the application.

  - *Priority:* **M** (Must have)

- **NFR2: Patient Books Appointment Steps <= 5**

  - *Description:* Patients shall be able to complete the core workflow of booking an appointment request (from dashboard to submission) within approximately 5 logical steps (e.g., page loads/major interactions).

  - *Priority:* **S** (Should have)

- **NFR3: Patient Marks Med Steps <= 3**

  - *Description:* Patients shall be able to mark a medication dose as taken with approximately 3 clicks or less from the main Medication Tracking screen.

  - *Priority:* **S** (Should have)

- **NFR4: Clear Visual Feedback**

  - *Description:* The system shall provide clear and immediate visual feedback for user actions, such as confirmation messages after submitting forms, loading indicators for processes, and clear error messages.

  - *Priority:* **M** (Must have)

- **NFR21: Efficient Doctor Prescription Workflow**

  - *Description:* The workflow for doctors to prescribe or edit a medication shall be clear, efficient, and minimize potential for errors.

  - *Priority:* **M** (Must have)

- **NFR19: Clear Appointment Status Display**

  - *Description:* Appointment statuses (Pending, Confirmed, Denied, Cancelled) shall be clearly and distinctively displayed to both patients and doctors.

  - *Priority:* **M** (Must have)

- **Justification:** High usability (NFR1, NFR4, NFR21, NFR19) is critical for the adoption and effective use of a telehealth system, especially given the diverse user base. While specific step counts (NFR2, NFR3) are targets ('Should Have') aimed at streamlining common tasks, overall ease of use and clarity are 'Must Haves'. Poor

usability can lead to user frustration, errors, and abandonment, undermining the system's purpose *(e.g., cite source on usability impact in health IT).*

3.3.2 Performance

The system should be responsive and provide a smooth user experience without frustrating delays, particularly for frequently accessed features and real-time interactions.

- **NFR5: Dashboard Load Time**

  - *Description:* The main user dashboard shall load completely within approximately 3 seconds under typical network conditions (e.g., stable broadband).

  - *Priority:* **S** (Should have)

- **NFR6: Chat Message Delivery**

  - *Description:* Chat messages shall be delivered and displayed to the recipient(s) within approximately 1 second under normal operating conditions to ensure a near real-time conversation flow.

  - *Priority:* **M** (Must have)

- **NFR7: List Load Time**

  - *Description:* Frequently accessed lists (e.g., appointments, medications, patient list for doctors) shall load and display within approximately 2 seconds.

  - *Priority:* **S** (Should have)

- **NFR18: Adherence Log Fetch Time**

  - *Description:* Fetching and displaying the patient adherence log for a selected date range shall complete within approximately 3 seconds.

  - *Priority:* **S** (Should have)

- **Justification:** System performance is crucial for user satisfaction. While general responsiveness (NFR5, NFR7, NFR18) is important to avoid frustration, real-time chat (NFR6) requires near-instant message delivery ('Must Have') for effective communication. Slow performance, especially in chat, would negatively impact the user experience.

3.3.3 Security

Given the highly sensitive nature of personal health information (PHI), robust security measures are paramount to protect patient data, maintain confidentiality, ensure data integrity, and comply with privacy regulations and best practices.

- **NFR8: HTTPS/TLS Encryption**

  - *Description:* All data transmission between the client (user's browser) and the server **shall** be encrypted using industry-standard HTTPS/TLS protocols.

  - *Priority:* **M** (Must have)

- **NFR9: Secure Password Storage**

- o *Description:* User passwords **shall** not be stored in plain text. They **shall** be securely stored using industry-standard, strong hashing algorithms with unique salts (e.g., bcrypt, Argon2).
- o *Priority:* **M** (Must have)

- **NFR10: Strict Data Segregation**

  - o *Description:* The system **shall** enforce strict data segregation, ensuring that patients can only access their own data, and doctors can only access data pertaining to patients explicitly linked to them (e.g., through appointments or their patient list). Unauthorized cross-access must be prevented.
  - o *Priority:* **M** (Must have)

- **NFR17: Doctor Access Control Refinement** (*Refining NFR10 slightly*)

  - o *Description:* Doctors' access to patient data **shall** be restricted based on established relationships within the system (e.g., city association, prior consultation). Access to unlinked patient data **shall** be prohibited.
  - o *Priority:* **M** (Must have)

- **NFR11: API Endpoint Authentication**

  - o *Description:* All API endpoints that handle data access or modification **shall** require valid user authentication and authorization tokens/sessions to prevent unauthorized direct access.
  - o *Priority:* **M** (Must have)

- **NFR12: Input Sanitization/Validation**

  - o *Description:* User-provided input data **shall** be rigorously sanitized and validated on both the client-side (as a preliminary check) and, more importantly, on the server-side to prevent common web vulnerabilities such as Cross-Site Scripting (XSS) and injection attacks (e.g., NoSQL injection specific to MongoDB).
  - o *Priority:* **M** (Must have)

- **Justification:** Security is non-negotiable in a healthcare application. These requirements (NFR8-NFR12, NFR17) establish fundamental security controls to protect sensitive patient information from unauthorized access, disclosure, or modification. Encrypting data in transit (NFR8), securing credentials (NFR9), enforcing strict access controls (NFR10, NFR17), securing APIs (NFR11), and preventing common attacks through input validation (NFR12) are all essential baseline practices mandated by data protection regulations (like GDPR) and crucial for building user trust.

3.3.4 Reliability

The system should be dependable and function correctly when needed. Key aspects include data integrity, especially for communications, and consistent availability of core services.

- **NFR13: Service Availability Goal**

- o *Description:* The core functionalities (login, appointment management, medication tracking, chat) shall aim for high availability during expected usage hours, targeting approximately 99% uptime.
  - o *Priority:* **S** (Should have)
- **NFR14: Chat Message Persistence**
  - o *Description:* The system shall ensure that submitted chat messages are reliably persisted and not lost due to common transient issues (e.g., brief network interruptions, page refreshes).
  - o *Priority:* **M** (Must have)
- **NFR20: Reliable Notifications**
  - o *Description:* If notifications (related to FR38, FR45) are implemented, they shall be delivered reliably to the intended recipient under normal operating conditions.
  - o *Priority:* **S** (Should have)
- **Justification:** System reliability is crucial for user trust and effective operation. While achieving a specific uptime percentage (NFR13) is a target ('Should Have') for consistent service access, ensuring the integrity and persistence of communication data like chat messages (NFR14) is a 'Must Have' to prevent loss of potentially important information. Reliable delivery of notifications (NFR20), if implemented, supports timely user actions.

3.3.5 Maintainability and Compatibility

The system should be built using good practices to facilitate future updates or modifications, and it must function correctly on common user platforms.

- **NFR15: Code Standards and Documentation**
  - o *Description:* The source code (React frontend, Node.js backend) shall adhere to reasonable coding standards for clarity and consistency and include comments for complex or non-obvious logic sections to aid understanding.
  - o *Priority:* **S** (Should have)
- **NFR16: Browser Compatibility**
  - o *Description:* The web application shall render correctly and be fully functional on the latest stable versions of major desktop web browsers, specifically Google Chrome, Mozilla Firefox, and Microsoft Edge.
  - o *Priority:* **M** (Must have)
- **Justification:** Ensuring compatibility across common web browsers (NFR16) is essential ('Must Have') for accessibility to the intended user base. Adhering to coding standards and providing documentation (NFR15) is important ('Should Have') for maintainability, easing potential future development, debugging, or assessment of the codebase.

Chapter 4: Methodology

4.1 Introduction

This chapter outlines the systematic methodology employed throughout the lifecycle of this Digital Systems Project. It details the chosen development approach, the key phases undertaken, the tools and technologies utilized, and the project management practices followed. The primary aim of the methodology was to provide a structured yet flexible process suitable for an individual student project, enabling the successful development of the remote healthcare system from initial concept through to requirements specification, design, implementation, and preparation for evaluation.

4.2 Chosen Development Approach and Justification

Given the nature of this project – developing a web application with distinct functional modules (authentication, appointments, medications, chat) and a significant user interface component – a formal waterfall approach was deemed too rigid. Strict adherence to agile methodologies like Scrum, designed for team environments, was also impractical for an individual project.

Therefore, an **Iterative Development approach** was adopted. This allowed the project to be broken down into manageable stages or iterations. Core functionalities and their corresponding user interfaces (using React) were developed incrementally (e.g., authentication, then appointment viewing/booking). Developing the working user interface iteratively served as a form of functional prototyping; seeing and interacting with the actual UI components helped to visualize the system's look and feel, clarify user workflows, and solidify the functional requirements detailed in Chapter 3 before or alongside backend implementation.

Justification: This iterative methodology was selected for several reasons:

- **Flexibility:** It provided the necessary flexibility to adapt to emerging technical challenges or evolving understanding during development.
- **Risk Management:** Developing and testing core features and their UIs iteratively helped mitigate the risk of encountering major integration issues late in the project.
- **Clarity:** Building the functional UI iteratively provided immediate visual feedback and helped ensure the final requirements accurately reflected the intended user experience.
- **Suitability:** The iterative nature fits well with the modular design of the application (React components, distinct backend services) and allowed for focused development effort on specific components within distinct phases.

4.3 Development Environment & Workflow

The implementation phase utilized a modern JavaScript-based development environment designed for efficiency and maintainability, focusing on local execution rather than containerization.

- **Core Tools:** Development was primarily conducted using Visual Studio Code (VS Code) as the Integrated Development Environment (IDE), **Node.js (v22.13.1)** as the JavaScript runtime, and Node Package Manager (npm) for managing project dependencies specified in package.json files for both frontend and backend.
- **Version Control:** Git was employed for version control, with project code hosted on **GitHub, enabling** tracking of changes.
- **Concurrent Development Workflow:** To streamline local development, the concurrently npm package was utilized. A single command (npm run dev) initiated both the backend Node.js/Express server (using nodemon for automatic restarts upon code changes) and the frontend React development server (react-scripts start with hot module replacement) simultaneously. This allowed for rapid iteration and immediate feedback during coding, with logs from both processes visible in a single terminal window.
- **Development API Proxy:** Communication between the frontend development server (running on default port 3000) and the backend server (running on port 5000) during local development was simplified using React's built-in proxy feature. By setting "proxy": "http://localhost:5000" in the frontend's package.json, API requests from the frontend were automatically forwarded by the React development server to the backend API, facilitating local development without needing complex Cross-Origin Resource Sharing (CORS) configurations specifically for the development environment.

This setup provided an efficient workflow for implementing and testing both frontend and backend features concurrently during development.

4.4 Project Stages / Workflow

While the development process was iterative, allowing for flexibility and refinement, the project broadly followed a sequence of logical phases from conception to completion. These phases often overlapped, particularly Design, Implementation, and Testing, reflecting the iterative nature of building and refining modules.

1. **Phase 1: Research and Planning:** This initial phase involved identifying the project domain (remote healthcare systems), conducting a literature review (detailed in Chapter 2) to understand existing solutions, challenges (e.g., adherence, usability, security), and relevant technologies. The core project aims and objectives were defined, and the overall scope was established.
2. **Phase 2: Requirements Engineering:** Functional and non-functional requirements were gathered, analyzed, specified (Chapter 3, including a Use Case overview), and prioritized using the MoSCoW method.
3. **Phase 3: System Design:** This phase focused on planning the system's logical structure, data model, API interface, and UI/UX (Chapter 5), based on the requirements.
4. **Phase 4: Implementation:** The core development phase where the system was built based on the design, involving coding the backend (Node/Express/MongoDB), frontend (React), and integration (API calls, Socket.IO). Challenges encountered during coding were addressed in this phase (discussed in Section 6.5).

5. **Phase 5: Testing:** Verification of the implemented system's functionality against requirements (detailed in Section 6.6).
6. **Phase 6: Evaluation & Reporting:** Reflection on the project outcome, assessment against aims/requirements, discussion of limitations/future work (Chapter 7), and documentation through this dissertation report.
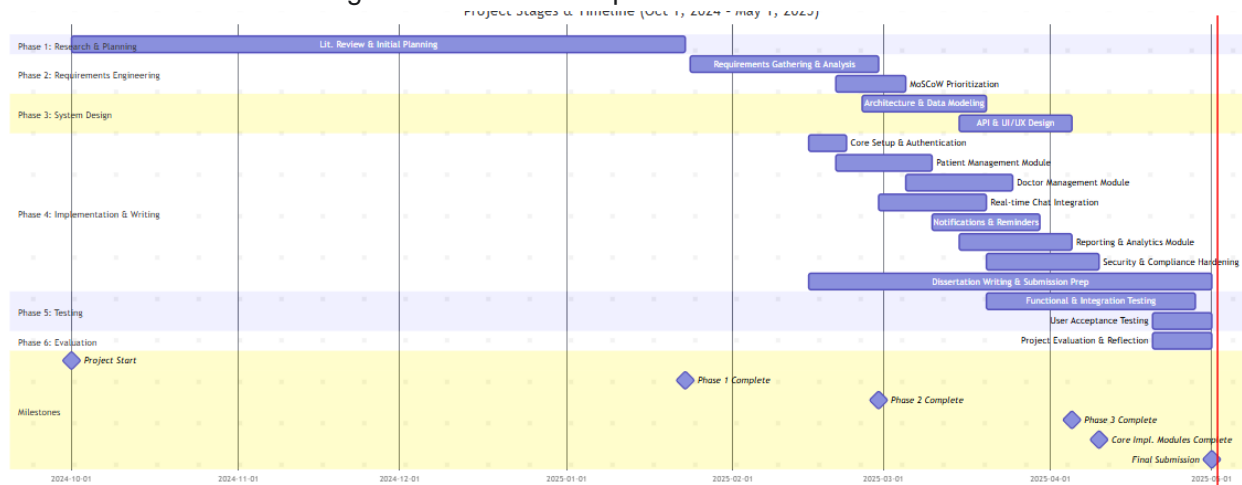

Figure 1 (4.1)

Figure 4.1(Gantt chart)

4.5 Project Management

Effective project management was necessary to structure the development process and ensure the project stayed on track within the academic timeframe. As an individual project, management primarily focused on self-organization and leveraging supervisor guidance.

- **Supervisor Consultation:** Regular meetings with the project supervisor were a key component of the management strategy. These sessions provided invaluable opportunities to discuss progress, clarify requirements, validate design approaches, troubleshoot technical challenges, and receive feedback on completed work. Seeking and incorporating this expert advice was crucial throughout all project phases.
- **Task Planning and Tracking:** Project work was broken down into smaller, manageable tasks aligned with the iterative phases (Section 4.4). This approach helped prioritize work and provided a clear focus for each development iteration.
- **Activity and Debugging Logs:** While not always formal, keeping records of progress and specific technical hurdles aided in recalling the development process for documentation and reflection purposes.
- **Time Management:** Managing project scope and estimating time for development tasks was an ongoing process, particularly when encountering unforeseen challenges (as discussed in Section 6.5). Prioritization based on the MoSCoW requirements (Chapter 3) helped focus efforts on essential features within the available time.

4.6 Summary

This chapter has detailed the methodology adopted for the project. A hybrid approach, combining **Iterative Development** with **functional prototyping** via iterative UI development, was chosen for its flexibility and suitability for this individual web application project. The project progressed through distinct but overlapping phases from Research to Reporting, supported by a local development workflow utilizing concurrently and the React development proxy. Key technologies centered around the MERN stack (MongoDB, Express.js, React.js, Node.js) and Socket.IO, managed with tools like VS Code and Git. Project management involved regular consultation with the supervisor, task tracking, and activity logging. This structured yet adaptable methodology provided the framework for developing the remote healthcare system presented in this report.

Chapter 5: System Design

5.1 Introduction

This chapter presents the comprehensive design of the remote healthcare platform developed for this project. Following the requirements specification outlined in Chapter 3, this chapter details the architectural decisions, data structures, communication protocols, and user interface (UI) design choices made to build a functional and effective system.

The primary goal of the design phase was to translate the identified functional requirements (FRs) and non-functional requirements (NFRs) into a coherent technical blueprint. Key considerations included creating a scalable client-server architecture, designing a flexible database schema suitable for healthcare data, defining a clear API for client-server communication, implementing a secure and efficient real-time chat mechanism, and crafting an intuitive user interface for both patients and doctors.

5.2 System Architecture Design

The remote healthcare platform employs a **Client-Server architectural pattern**, implemented using the **MERN stack** (MongoDB, Express.js, React.js, Node.js) components, augmented with Socket.IO for real-time communication. This layered approach separates the presentation logic (client) from the business logic and data persistence (server), promoting modularity and maintainability. The high-level architecture for the local development environment is depicted in Figure 5.1.
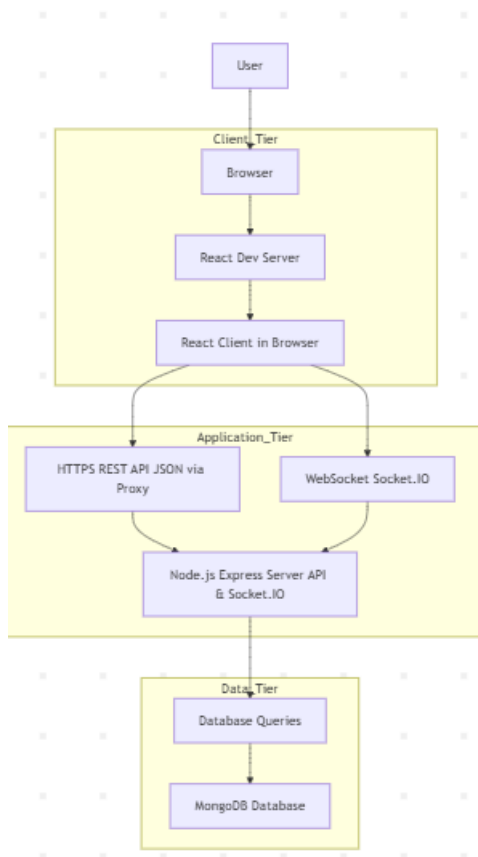
Figure 2 (5.1)

Figure 5.1: System Architecture Overview (Local Development Setup)

The key components and their roles within this architecture, as configured for local development, are:

1. **Web Browser (Client Host):** The environment where the end-user (Patient or Doctor) interacts with the application.
2. **React Development Server:** Started via `npm start` (as orchestrated by `concurrently`), this server serves the React application files during development. It also plays a key role in facilitating API communication through a development proxy.
3. **React Client (in Browser / Runtime):** Once loaded by the development server, the React Single-Page Application (SPA) executes within the user's browser. It manages the user interface, handles user input, maintains client-side state, and interacts directly with the backend server via API calls and WebSocket connections.
4. **Node.js / Express.js Server (API & Socket.IO):** This backend server, running as a standard Node.js process (e.g., started via `nodemon` as part of the `npm run dev` command), forms the core of the application's logic.
   - It provides a **RESTful API** using Express.js to handle CRUD operations.
   - It incorporates a **Socket.IO server** for real-time communication.
   - It executes business logic, validates data, and orchestrates communication with the database.

5. **MongoDB Database:** A NoSQL document database process, running locally or accessible to the Node.js server, responsible for data persistence.

Communication flows are as follows: The Web Browser requests application files from the React Development Server. The React Client (running in the browser) makes standard HTTP requests for API data and establishes WebSocket connections for real-time chat directly with the Node.js/Express Server. During local development, API requests from the React development server are automatically routed by a **development proxy** (configured in `package.json`) from port 3000 to port 5000 on `localhost`. The Node.js server communicates directly with the MongoDB database.

5.3 Database Design

The application utilizes **MongoDB**, a NoSQL document database, operating within a dedicated Docker container. This choice was made after considering alternatives like traditional relational databases (e.g., MySQL). The primary advantages of MongoDB for this project include:

- **Schema Flexibility:** MongoDB's document model readily accommodates the potentially varied and evolving nature of healthcare data (user details, appointment notes, medication regimens) without requiring the strict schema definitions and complex migrations typical of SQL systems.

- **Development Synergy:** As part of the MERN stack, MongoDB integrates seamlessly with the JavaScript-based Node.js backend via the Mongoose ODM, potentially simplifying development compared to using SQL with an ORM.

- **Data Structure Fit:** The document structure aligns well with application entities like user profiles and conversations.

While relational databases offer robust transaction support, MongoDB's flexibility was prioritized for this project's scope and development approach.

The application data is organized into several key MongoDB **collections**, structured according to the Mongoose models defined in the application backend. The overall structure

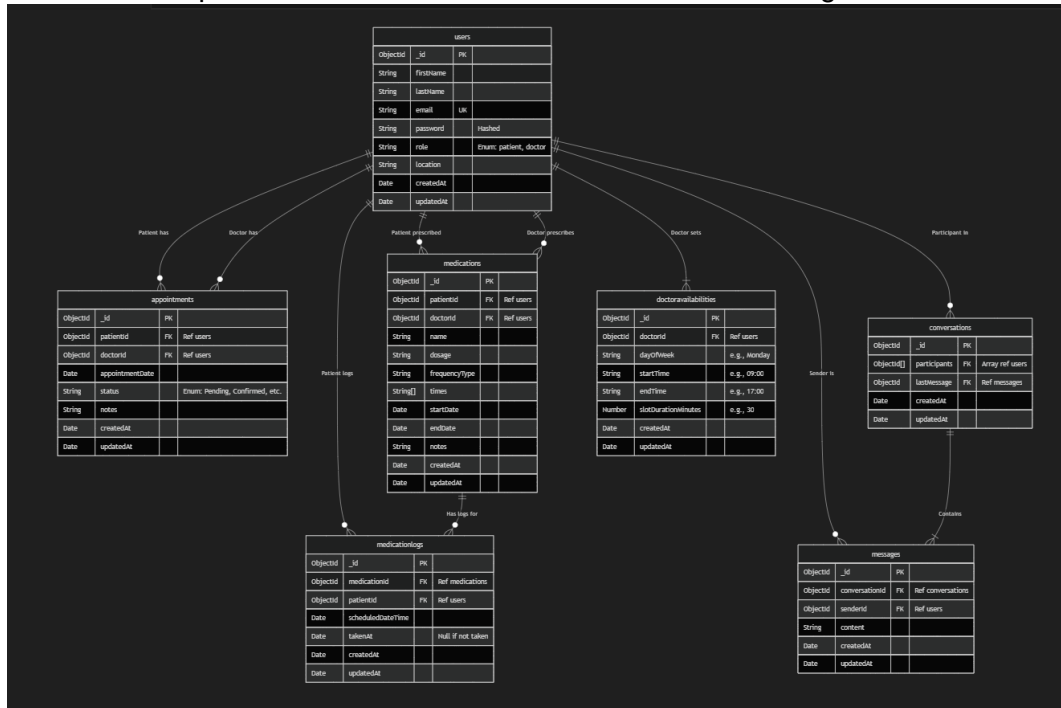and relationships between these collections are illustrated in Figure 5.2

Figure 3(5.2)

Figure 5.2: Database Schema (ER Diagram Representation)

A description of the main collections and their key fields, as depicted in Figure 5.2, is provided below:

1. **users:** Stores essential information for both patients and doctors, acting as the central entity for authentication and role management. Key fields include firstName, lastName, email (unique identifier), hashed password, role, and location.

2. **appointments:** Contains details for each requested or scheduled appointment. It links a patient and a doctor (both referencing the users collection) and includes the appointment date, time, optional notes, and the crucial status field (e.g., "Pending", "Confirmed") to track its lifecycle.

3. **medications:** Stores details of prescribed medication regimens linked to both the patient and the prescribing doctorId (referencing users). It details the name, dosage, structured frequency information (frequencyType, frequencyValue, daysOfWeek, times), duration (startDate, endDate), isActive status, and optional notes.

4. **medicationlogs:** Tracks patient adherence. Each log entry references the patient and optionally the specific medication. It records the unique scheduleItemId, the scheduledDate and scheduledTime, and the takenAt timestamp indicating when the patient marked the dose as completed.

5. **conversations:** Represents a chat thread, primarily containing an array of participants (referencing the two users involved) and an optional reference (lastMessage) to the most recent message for preview purposes.

6. **messages:** Stores individual chat messages, each linked to its conversation and the sender (referencing users), along with the message content.

7. **doctoravailabilities:** Defines time slots when a doctor is available. Each record links to the doctor (referencing users) and specifies availability details like date, startTime, and endTime.

8. **Relationships Management:** As shown in Figure 5.2, relationships between collections are implemented using ObjectId references stored within the documents (e.g., appointments storing patient and doctor ObjectIds). These logical links, defined in the Mongoose schemas, allow the backend application to retrieve related data efficiently using the populate() method, effectively simulating relational joins where needed.

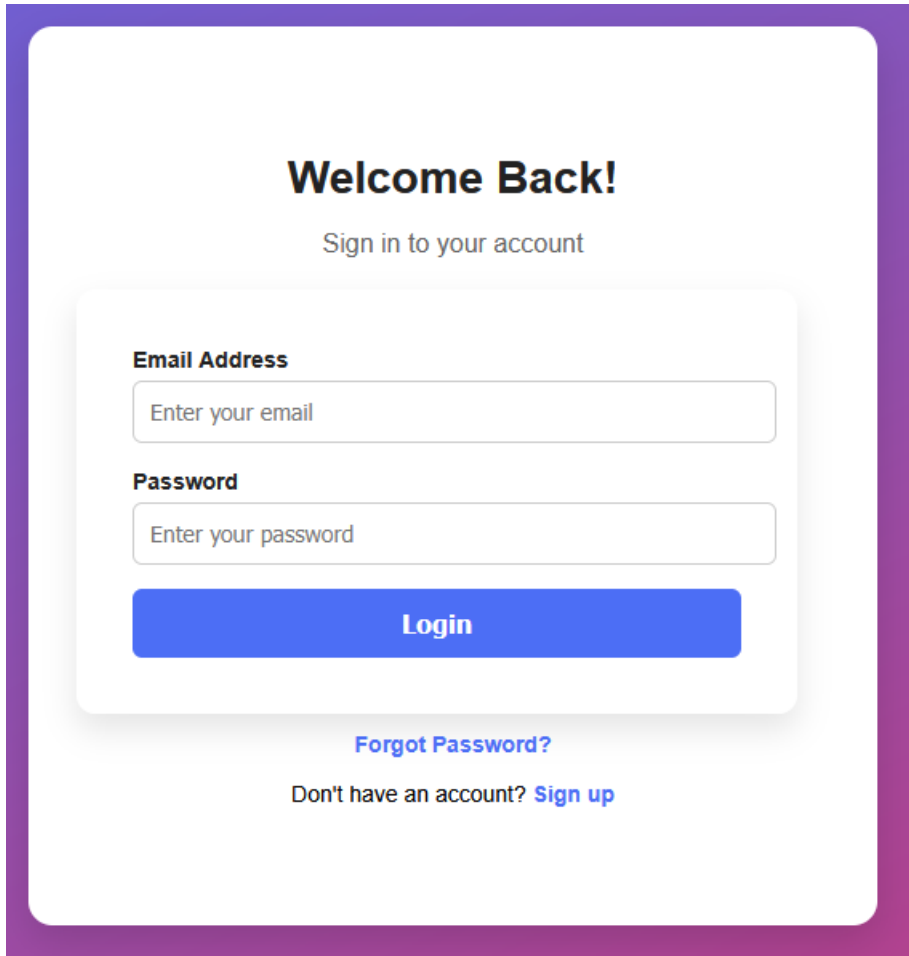# 5.4 User Interface (UI) and User Experience (UX) Design

The user interface (UI) was developed as a Single-Page Application (SPA) using the **React.js** library, aiming for clarity, ease of use, and consistency (NFR1, NFR4). React's component-based architecture facilitated modular development and efficient UI updates.

Framework Choice & Design Approach: React was chosen for its performance and component reusability. A component-based approach structured the interface into reusable elements and page components. Client-side routing with `react-router-dom` enabled seamless navigation. Application state, including user authentication and data, is managed using React's Context API and component state (`useState`, `useEffect`). API interaction uses `axios` via service functions, and real-time chat uses `socket.io-client`.

Key Screens and Workflow:

The design provides distinct screens for user authentication, role-specific dashboards, and management of core functionalities, as illustrated by the following key views:

1. **Authentication Screens (Login & Sign Up):**

Figure 4(5.3)

Figure 5.3: User Login Screen
These screens (Figures 5.3, 5.4) allow users to securely register and log in (FR1-FR3). The Login screen captures credentials, while the Sign Up screen collects user details and role (Patient/Doctor). Successful authentication redirects users to their role-specific dashboards.

Figure 5(5.4)

Figure 5.4: User Sign Up Screen

2. **Patient Dashboard:**
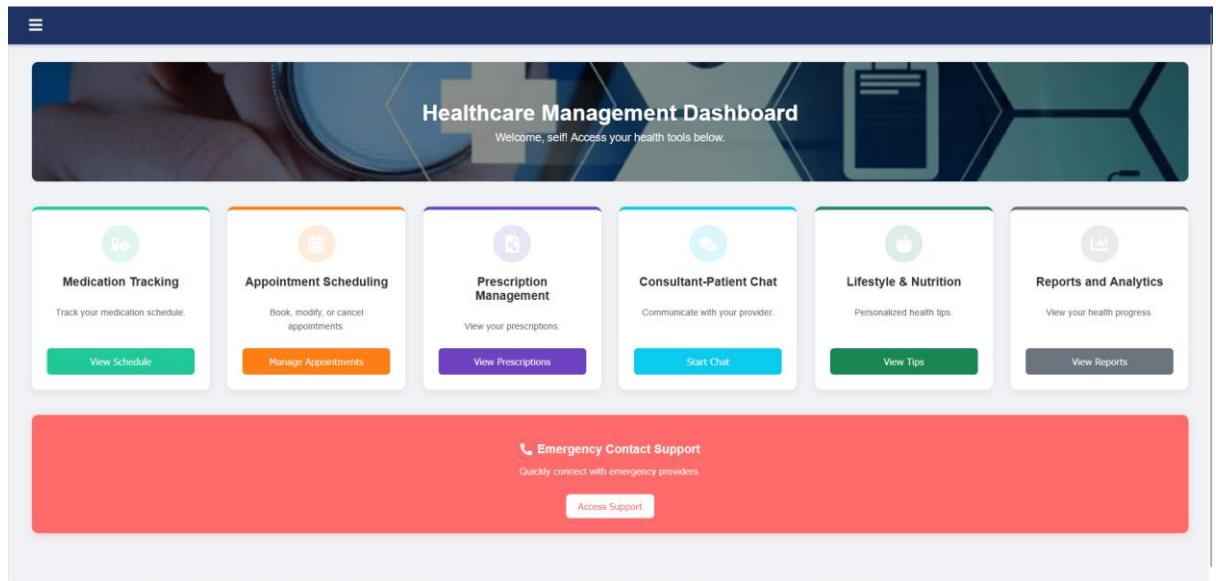
Figure 5.5: Patient Dashboard
This acts as the central hub for patients, providing quick access to Medication Tracking, Appointment Scheduling, Prescription Management, Chat, and Lifestyle & Nutrition sections (FR30).

3. **Appointment Management (Patient):**

Figure 7(5.6)

Figure 5.6: Patient Appointment List Screen

Patients can view their appointments (Figure 5.6) and access controls to edit or cancel

them. The "Create New Appointment" button initiates the booking process.

Figure 8(5.7)

Figure 5.7: Patient Appointment Request Form

The form (Figure 5.7) allows patients to select appointment details (Date, Doctor, Time Slot) and notes before submitting the request (FR7-FR12).

4. **Medication Tracking (Patient):**
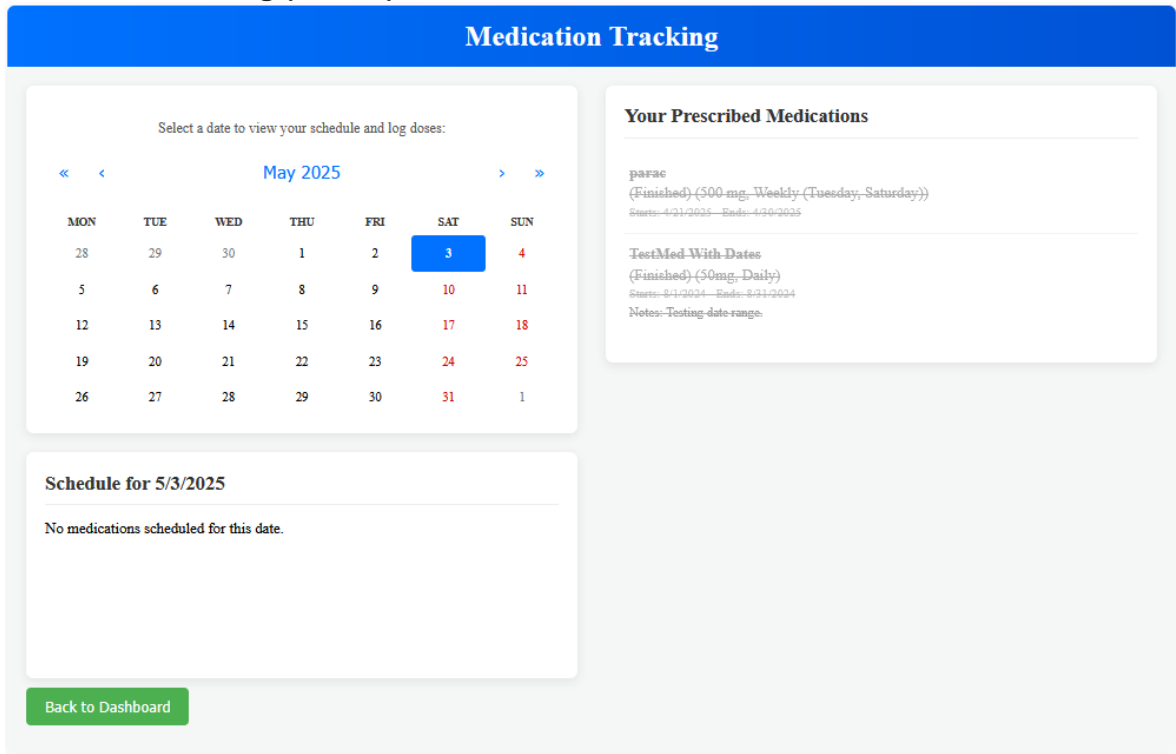
Figure 9(5.8)

Figure 5.8: Patient Medication Tracking Interface
Patients can view their medication schedule via a calendar and mark doses as taken (FR16-FR19), supporting adherence tracking.

5. **Doctor Dashboard:**
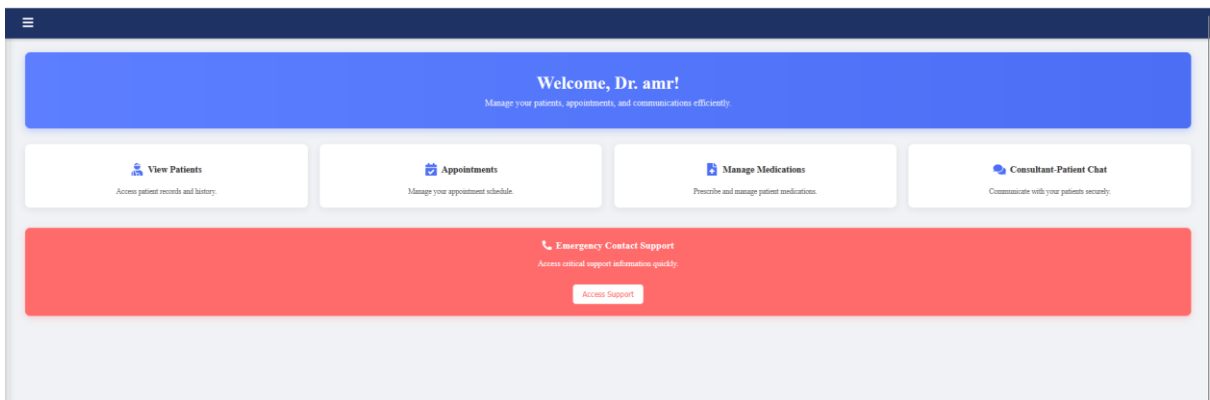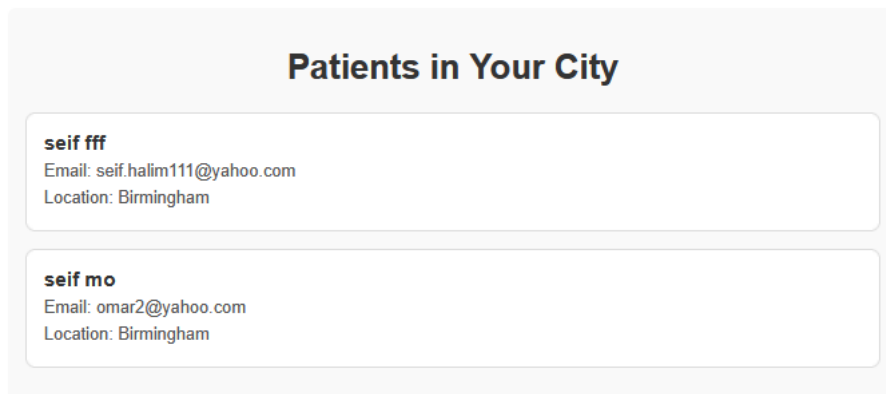
Figure 10(5.9)

Figure 5.9: Doctor Dashboard
The doctor's central hub, providing access to manage Patients, Appointments, Medications, and Chat (FR30).

6. **View Patients (Doctor):**

Patients in Your City

seif fff
Email: seif.halim111@yahoo.com
Location: Birmingham
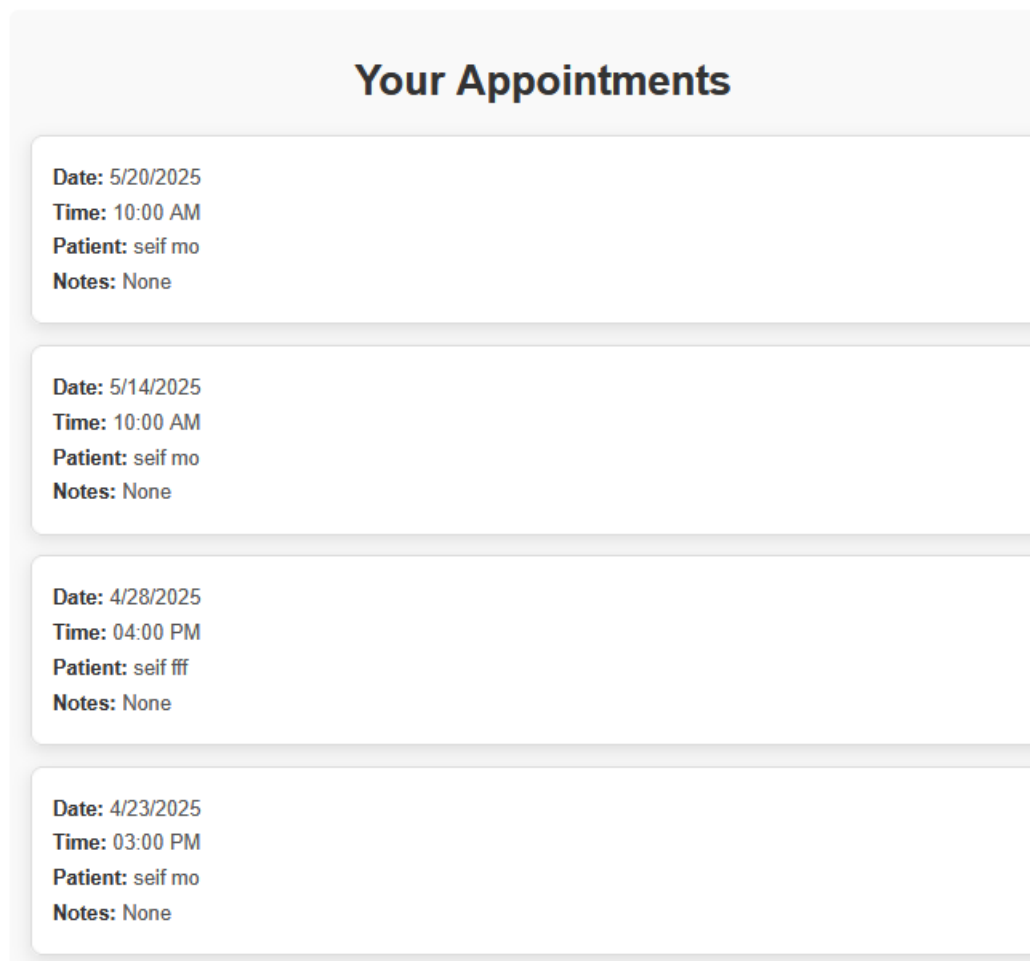
seif mo
Email: omar2@yahoo.com
Location: Birmingham

Figure 11(5.10)

Figure 5.10: Doctor's Patient List Screen
Doctors can view a list of their associated patients, typically filtered by location (FR26), as a starting point for accessing patient records.

7. **Manage Appointments (Doctor):**

Your Appointments

Date: 5/20/2025
Time: 10:00 AM
Patient: seif mo
Notes: None

Date: 5/14/2025
Time: 10:00 AM
Patient: seif mo
Notes: None

Date: 4/28/2025
Time: 04:00 PM
Patient: seif fff
Notes: None

Date: 4/23/2025
Time: 03:00 PM
Patient: seif mo
Notes: None

Figure 12(5.11)

Figure 5.11: Doctor Appointment Management View
Doctors manage their schedule and review pending appointment requests (FR25, FR39-FR43), with options to accept, deny, or cancel appointments.

8. **Manage Patient Medications (Doctor):**

Figure 13(5.12)

Figure 5.12: Doctor Medication Management Interface
Doctors can select a patient, view their medications and adherence log, and access forms to prescribe or edit medications (FR27, FR28, FR32-FR35).

Figure 5.13: Prescribe New Medication Modal

9. **Chat Interface:**
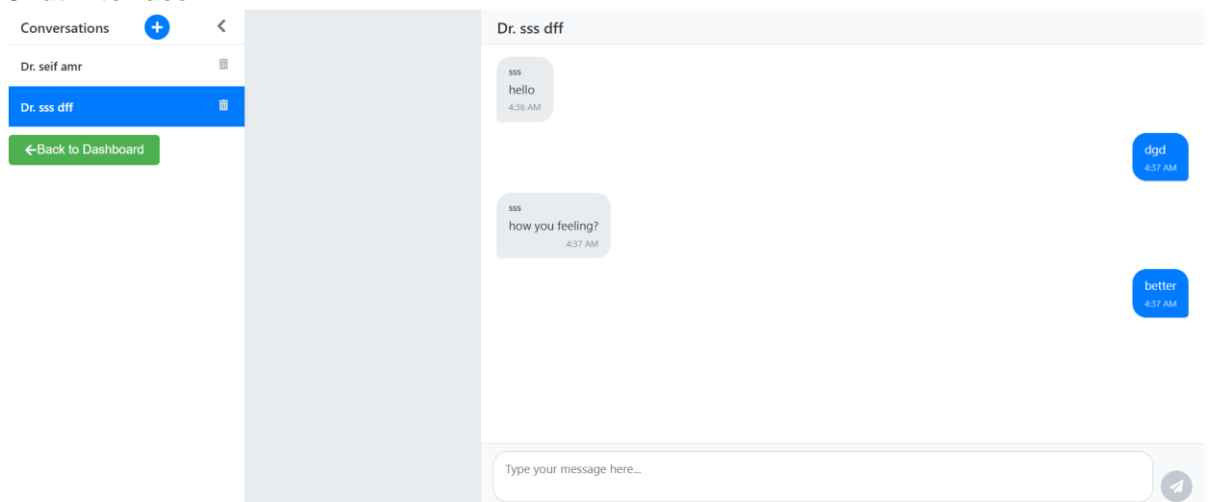
Figure 5.14: Chat Interface
Accessible to both roles, this interface enables real-time text communication via listed conversations (FR20-FR23), with options to send/view messages and manage conversations.

The UI design visually realizes the system's functionalities, providing a clear and navigable interface for both user roles.

5.5 Interaction Design / Key Workflows

While the previous section detailed the static design of the user interface screens and backend components, this section focuses on the dynamic interactions between these components during key system operations. Sequence diagrams, state machine diagrams, and activity diagrams are utilized to illustrate the flow of control, message passing, and state transitions for critical workflows, providing a deeper understanding of how the system fulfills its requirements.

1. User Login Sequence:

The user login process is fundamental for accessing the system's role-based functionalities. Figure 5.15 illustrates the sequence of interactions involved in a successful login attempt.

User Login Sequence (with Error Paths)

Enters email & password

Clicks Login Button

POST /api/auth/login
{ email, password }

User.findOne({ email: email })

Returns User Document (userRecord) / null

alt [userRecord found]

bcrypt.compare(password, userRecord.password)

alt [passwords match]

generateToken(user._id)

200 OK Response
{ userDetails, token }

Store token & userDetails

Redirect to Dashboard

[passwords do not match]

401 Unauthorized
{ message: "Invalid email or password" } ' Explicit Error Reply

Display Error Message

[userRecord not found]

401 Unauthorized
{ message: "Invalid email or password" } ' Explicit Error Reply

Display Error Message

Figure 16(5.15)

Figure 5.15: User Login Sequence Diagram

As shown in Figure 5.15, the process begins when the User submits their credentials via the React Client (Browser).

1. The Client sends a POST request containing the email and password to the /api/auth/login endpoint on the API Server.

2. The API Server receives the request and queries the MongoDB Database to find a user matching the provided email.

3. Assuming a user record is found, the Database returns the user document (including the hashed password) to the API Server.

4. The API Server then compares the provided password against the stored hash using bcrypt.compare.

5. Upon successful password validation, the API Server generates a JSON Web Token (JWT) containing user identity information.

6. The API Server sends a 200 OK response back to the React Client, including essential user details and the generated JWT.

7. Finally, the React Client securely stores the received token (e.g., in localStorage), updates the application state with the user's information, and typically redirects the user to their appropriate dashboard. This sequence ensures only authenticated users gain access, fulfilling requirement FR3. The process includes error handling paths (as shown in the detailed diagram) for scenarios where the user is not found or the password does not match, resulting in a 401 Unauthorized response.

2. Send Chat Message Sequence:

Real-time communication is handled via Socket.IO. Figure 5.16 illustrates the typical flow when a logged-in user (Sender A) sends a message to another user (Recipient B).
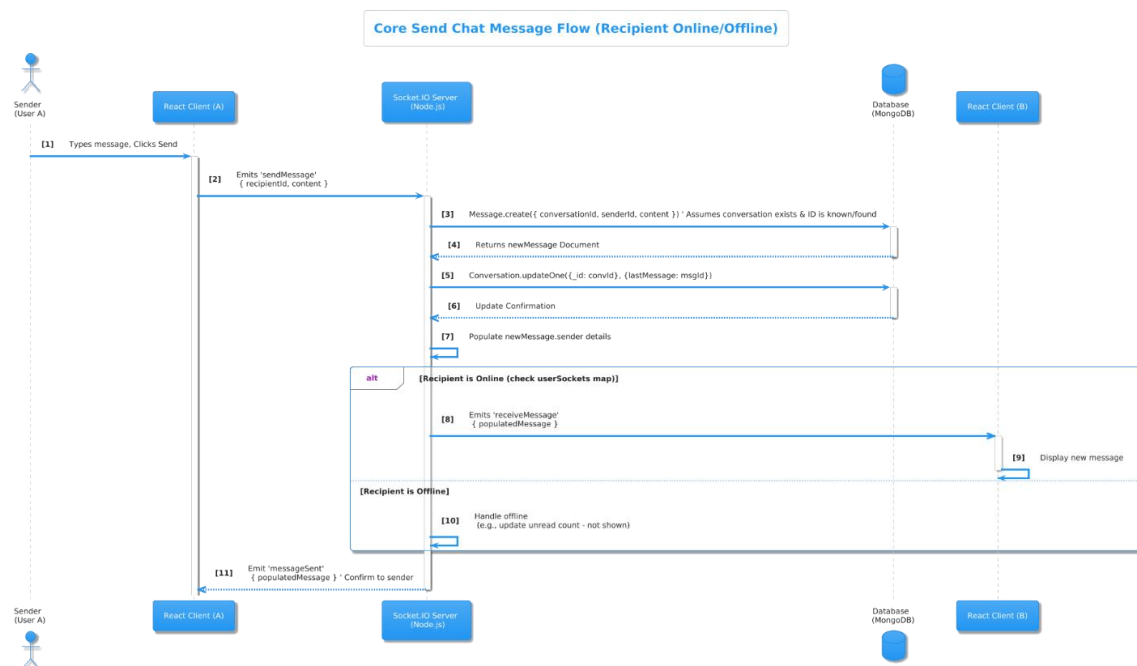
Figure 17(5.16)

Figure 5.16: Send Chat Message Sequence Diagram *(Adjust figure number)*

The process involves the following key interactions:

1. User A types a message in their React Client (Client A) and initiates the send action.

2. Client A emits a sendMessage event over the established WebSocket connection to the Socket.IO Server, including the recipient's ID and the message content.

3. The server, having authenticated the sender's socket connection, saves the new message to the messages collection in the MongoDB Database, linking it to the appropriate conversation.

4. The server may also update the corresponding conversations document (e.g., updating the lastMessage reference).

5. After database confirmation, the server retrieves necessary sender details (population).

6. The server checks its internal mapping (e.g., userSockets) to determine if the Recipient (User B) is currently connected via an active socket.

7. **(If Online):** The server emits a receiveMessage event containing the message details directly to the Recipient's client (Client B) via their specific socket connection. Client B then displays the message in its UI.

8. **(If Offline):** The server handles the offline scenario (e.g., potentially updating an unread count in the database, logic not detailed in the diagram).

9. Finally, the server emits a messageSent event back to the Sender's client (Client A) to confirm the message was processed and provide the saved message data. This sequence demonstrates the real-time, event-driven communication pattern facilitated by Socket.IO (fulfilling requirements related to UC_Chat).

3. Patient Requests New Appointment Sequence:

Figure 5.17 details the sequence for a patient initiating an appointment request, including the necessary backend validation and the creation of a pending appointment record.
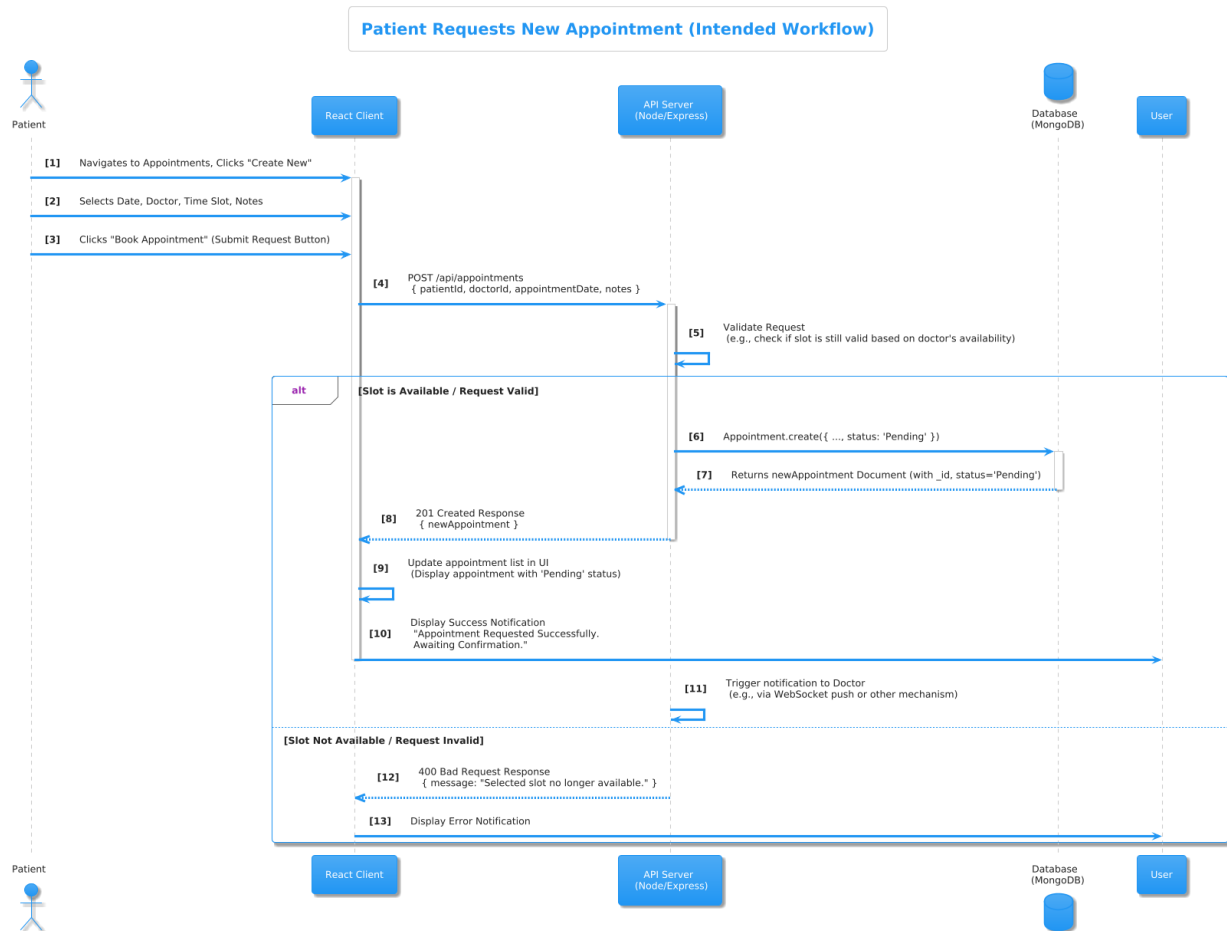
Figure 18(5.17)

Figure 5.17: Patient Requests New Appointment Sequence Diagram *(Adjust figure number)*

This workflow proceeds as follows:

1. The Patient interacts with the React Client UI, selecting appointment details (date, doctor, time, notes) and submitting the request.

2. The Client sends a POST request to the /api/appointments endpoint on the API Server with the selected data.

3. The API Server validates the request, which may include re-checking the selected doctor's availability for the specified time slot against records in the Database.

4. **(If Valid):** The API Server creates a new document in the appointments collection in the Database, setting the initial status to "Pending".

5. The Database confirms the creation and returns the new appointment document.

6. The API Server sends a 201 Created success response back to the Client, including the details of the pending appointment.

7. The Client updates the UI to display the newly requested appointment with its "Pending" status and notifies the Patient that the request awaits doctor confirmation.

8. The API Server may also trigger a separate, asynchronous process (e.g., WebSocket push, queue) to notify the relevant Doctor about the new pending request (details omitted from this diagram).

9. **(If Invalid):** If the initial validation fails (e.g., slot taken), the API Server sends an appropriate error response (e.g., 400 Bad Request) to the Client, which then displays an error message to the Patient. This flow ensures appointments are formally requested and await doctor action (FR12, FR39).

4. Appointment State Lifecycle:

The Appointment entity progresses through several distinct states during its lifecycle, from initial request to final resolution. The State Machine Diagram in Figure 5.18 illustrates these states and the transitions between them.
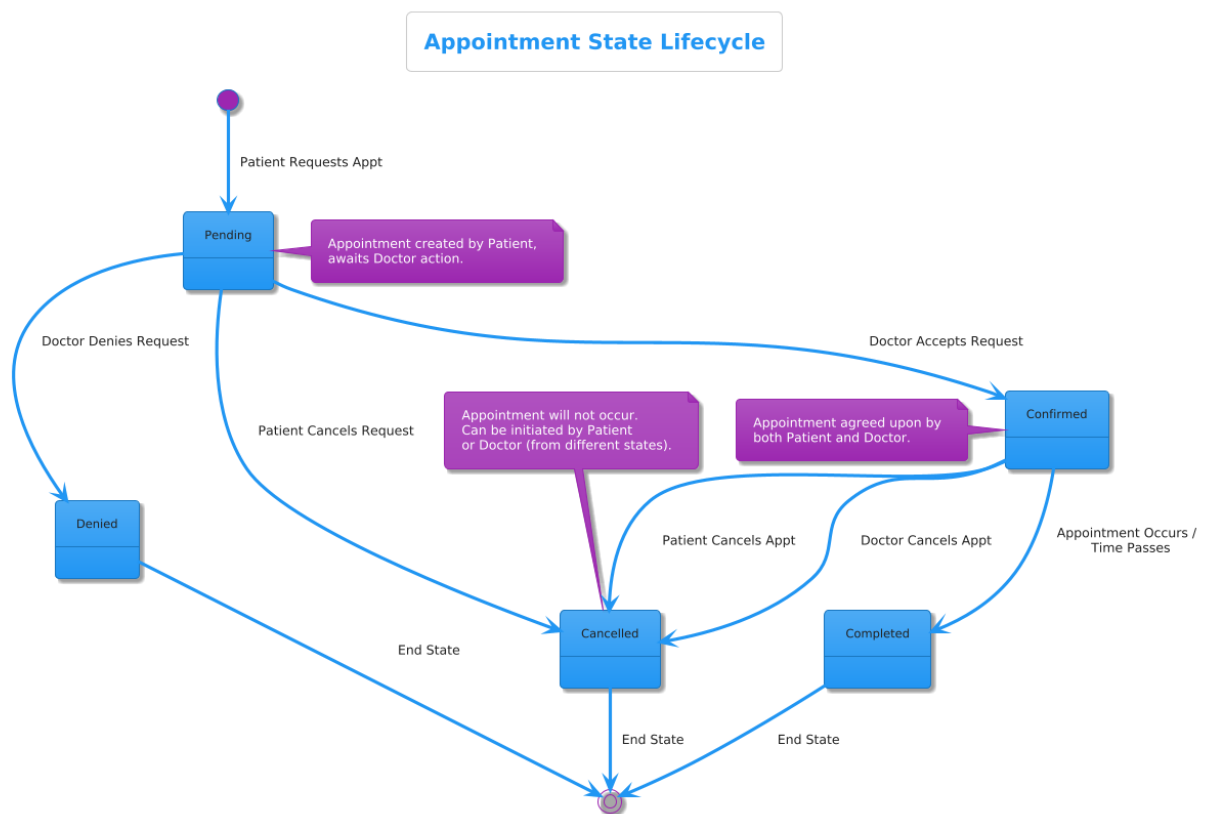
Figure 19(5.18)

Figure 5.18: Appointment State Lifecycle Diagram

The key states and transitions are:

1. **Start ([*]) -> Pending:** An appointment enters the Pending state immediately upon successful creation via a patient request (Patient Requests Appt). In this state, it awaits action from the assigned doctor.

2. **Pending -> Confirmed:** If the doctor reviews the request and accepts it (Doctor Accepts Request), the appointment transitions to the Confirmed state, indicating agreement from both parties.

3. **Pending -> Denied:** If the doctor reviews the request and rejects it (Doctor Denies Request), the appointment moves to the Denied state, effectively ending this request attempt.

4. **Pending -> Cancelled:** The patient can choose to withdraw their request before the doctor acts (Patient Cancels Request), moving the appointment to the Cancelled state.

5. **Confirmed -> Cancelled:** Even after confirmation, either the patient (Patient Cancels Appt) or the doctor (Doctor Cancels Appt) can cancel the appointment, transitioning it to the Cancelled state.

6. **Confirmed -> Completed:** After the scheduled date and time of a confirmed appointment have passed, it moves to the Completed state (this transition might be triggered by a scheduled job or manual action, denoted here as Appointment Occurs / Time Passes).

7. **Denied / Cancelled / Completed -> End ([*])**: These states represent final outcomes from which no further standard transitions occur within this lifecycle model.

This state model clearly defines the possible statuses for an appointment, directly informing the status field design in the appointments collection (Section 5.3) and the logic required in the API endpoints responsible for status updates (Section 5.4, e.g., PUT /api/appointments/:id/status).

5.6 Security Design Considerations

Given the sensitive nature of healthcare data, security was a paramount concern throughout the design process, directly addressing NFRs related to confidentiality, integrity, and access control (NFR8-NFR12, NFR17).

Key security design choices included:

1. **Data Encryption in Transit:** Data transmitted between the client and backend API server is designed to be encrypted using HTTPS to ensure confidentiality (NFR8). *(Removed deployment specifics like Nginx)*.

2. **Secure Credential Storage:** User passwords are never stored in plaintext. The `User` model's pre-save hook uses `bcryptjs` to securely hash passwords before saving them to the database (NFR9).

3. **Authentication and Authorization:** Session management uses JWTs. Middleware (`authMiddleware.js`, `socketAuthMiddleware.js`) on API routes and Socket.IO connections verifies JWTs to authenticate users and control access to protected resources (NFR11).

4. **Access Control / Data Segregation:** Backend controller logic enforces strict data segregation based on the authenticated user's identity and role (`req.user`), ensuring users only access their own or authorized data (e.g., a doctor's patients) (NFR10, NFR17).

5. **Input Sanitization/Validation:** The design incorporates validation points in API controllers to check incoming data before processing, helping mitigate injection vulnerabilities (NFR12).

These design considerations establish a secure foundation for the application, protecting user data and ensuring authorized access.

5.7 Summary

This chapter presented the detailed design for the remote healthcare platform, translating the requirements from Chapter 3 into a technical blueprint for implementation.

The system employs a **Client-Server architecture** based on the **MERN stack** (MongoDB, Express.js, React.js, Node.js), augmented with **Socket.IO** for real-time chat. The design covers the overall architecture, the MongoDB database schema (representing collections and relationships), the React-based user interface and its workflows, and critical system interactions.

Key aspects include:

- **Database Design:** Structure of MongoDB collections (`User`, `Appointment`, `Medication`, etc.) and their relationships via ObjectIds.
- **UI Design:** Component-based React frontend providing intuitive interfaces and role-specific views (Section 5.4).
- **Interaction Design:** Dynamic workflows for key processes (login, chat, appointments) illustrated via sequence and state diagrams (Section 5.5).
- **Security:** Integrated design considerations for data encryption, password hashing, token-based authentication, and access control (Section 5.6).
- **API Specification:** A detailed list of backend API endpoints is provided separately (Appendix 5.8).

This comprehensive design provides the necessary foundation for the system's implementation, detailed in the following chapter.

Chapter (6) Implementation

6.1 Implementation Overview

The healthcare platform was implemented using a modern web technology stack, focusing on creating a scalable, secure, and user-friendly system. This chapter details the technical realization of the system's core components and features, highlighting key implementation decisions and challenges encountered during development.

The system architecture employs a microservices approach, with a clear separation between the frontend and backend services. This design choice was made to enhance scalability, maintainability, and development efficiency. The implementation leverages Node.js and Express.js for the backend, React.js for the frontend, and MongoDB for data persistence, chosen for their robustness, community support, and suitability for real-time applications.

The implementation process followed an iterative approach, starting with the core authentication system and gradually building up to more complex features like real-time chat and appointment management. This section will focus on critical implementation aspects, including the authentication system, real-time communication, and database integration, while also discussing key challenges and solutions encountered during development.

6.2 Development Environment & Workflow

The implementation of the remote healthcare platform utilized a standard web development environment, focusing on tools and technologies that support efficient coding, debugging, and workflow management for the MERN stack.

- **Core Implementation Technologies:** The system was built using the **MERN stack**. **Node.js** served as the backend runtime environment, chosen for its non-blocking I/O and ability to use JavaScript across the full stack. **Express.js** was selected as the backend framework for its flexibility in building the RESTful API. **React.js** was used for the frontend UI, chosen for its component-based architecture and efficient rendering capabilities. **MongoDB** provided the database layer, integrated via **Mongoose** for object modeling, leveraging its flexibility for data persistence. **Socket.IO** was implemented for real-time communication, integrated into the Node.js server due to its robustness for real-time web applications.

- **Development Tools:** Development was conducted using Visual Studio Code (VS Code) as the primary Integrated Development Environment (IDE). npm was used for managing project dependencies. Git and GitHub (*adjust if GitLab*) were utilized for version control.

- **Local Development Workflow:** During implementation, a key workflow feature was the use of the `concurrently` npm package. The command `npm run dev` initiated both the backend Node.js server (with `nodemon` for hot-reloading code changes) and the React development server simultaneously. This setup streamlined coding and debugging by providing unified logs and automatic updates upon saving files. Furthermore, the React development server's built-in proxy feature (configured in `package.json`) automatically forwarded API requests (e.g., to `/api/auth/login`) from the frontend (port 3000) to the backend (port 5000), simplifying API integration during local development.

This environment and workflow facilitated the practical implementation of the system's design.

6.3 Backend Implementation Details

The backend of the remote healthcare platform was implemented using Node.js and the Express.js framework, following the design principles outlined in Chapter 5. The codebase is organized into distinct directories for configuration (config), controllers (controllers), middleware (middleware), Mongoose models (models), route definitions (routes), and utility functions (utils). Key libraries utilized include Mongoose for MongoDB object modeling, bcryptjs for password hashing, jsonwebtoken for authentication tokens, and socket.io for real-time communication. The implementation focused on creating a robust API and handling application logic.

6.3.1 Authentication Implementation

Secure user authentication and authorization are foundational to the system. This was implemented using a token-based approach leveraging JSON Web Tokens (JWTs) and bcryptjs for secure password management.

- **Password Hashing:** User passwords are encrypted before storage. The Mongoose User schema includes a pre('save') hook that automatically hashes the password using bcryptjs whenever a user document is saved or updated with a modified password field.

```javascript
// Encrypt password before saving user
userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});

// Compare given password with the hashed password
userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};
```

Explanation: This snippet from the User model demonstrates the pre('save') middleware. It ensures that the plaintext password provided during registration or profile updates is automatically salted and hashed using bcrypt.hash before being saved to the database, preventing direct storage of sensitive credentials as required by NFR9.

```
JS server.js M          JS authControllers.js ✕       ⬡ package.json M

Backend > controllers > JS authControllers.js > [∅] registerUser > ⊘ asyncHandler() c
   59      // @access Public
   60      const loginUser = asyncHandler(async (req, res) => {
   61        const { email, password } = req.body;
   62
   63        const user = await User.findOne({ email });
   64        if (user && (await user.matchPassword(password))) {
   65          res.json({
   66            _id: user._id,
   67            firstName: user.firstName,
   68            lastName: user.lastName,
   69            email: user.email,
   70            role: user.role,
   71            location: user.location,
   72            token: generateToken(user._id),
   73          });
   74        } else {
   75          res.status(401);
   76          throw new Error("Invalid email or password");
   77        }
   78      });
   79
```

Explanation: This snippet from the authentication controller shows the core login verification logic. It retrieves the user by email and then uses the matchPassword method (defined on the User schema, internally using bcrypt.compare) to securely compare the provided password against the stored hash. Upon successful match, a JWT is generated using generateToken (a utility function) and returned to the client along with basic user details, implementing the login functionality (FR3) and contributing to session management.

6.3.2 User & Profile Management Implementation

Beyond login, the backend provides endpoints for user registration and profile management, leveraging the User model and authentication middleware.

- **User Registration:** The registerUser controller handles the creation of new user accounts.

```js
JS authControllers.js ×

Backend > controllers > JS authControllers.js > [●] registerUser > ✪ asyncHandler() callback >   Explain   Refactor
   7    // @access Public
   8    const registerUser = asyncHandler(async (req, res) => {
   9      const { firstName, lastName, email, password, role, location } = req.body;
  10
  11      // Check if user already exists
  12      const userExists = await User.findOne({ email });
  13      if (userExists) {
  14        res.status(400);
  15        throw new Error("User already exists");
  16      }
  17
  18      // Validate role
  19      const userRole = role ? role : "patient";
  20      if (!["patient", "doctor"].includes(userRole)) {
  21        res.status(400);
  22        throw new Error("Invalid role. Must be 'patient' or 'doctor'.");
  23      }
  24
  25      // Validate location
  26      if (!location) {
  27        res.status(400);
  28        throw new Error("Location is required.");
  29      }
  30
  31      // Create user
  32 >    const user = await User.create({ ⋯
  39      });
  40
  41      if (user) {
  42        res.status(201).json({
  43          _id: user._id,
  44          firstName: user.firstName,
  45          lastName: user.lastName,
  46          email: user.email,
  47          role: user.role,
  48          location: user.location,
  49          token: generateToken(user._id),
  50        });
  51      } else {
  52        res.status(400);
  53        throw new Error("Invalid user data");
  54      }
  55    });
  56
```

-

Explanation: This snippet illustrates the user registration endpoint logic. It checks for duplicate emails and performs basic data validation before creating a new User document. By passing the plaintext password here, it relies on the pre('save') hook shown in 6.3.1 to handle the hashing before the user is saved to MongoDB (FR1, FR2).

- **Profile Management:** Endpoints allow authenticated users to view and update their profile information.

```
JS authControllers.js  ×

Backend > controllers > JS authControllers.js > [∅] registerUser > ⦿ asyncHandler() callback > 
 82    // @access Private
 83    const getUserProfile = asyncHandler(async (req, res) => {
 84      const user = await User.findById(req.user._id);
 85      if (user) {
 86        res.json({
 87          _id: user._id,
 88          firstName: user.firstName,
 89          lastName: user.lastName,
 90          email: user.email,
 91          role: user.role,
 92          location: user.location,
 93        });
 94      } else {
 95        res.status(404);
 96        throw new Error("User not found");
 97      }
 98    });
 99
```

Explanation: This snippet shows how the getUserProfile controller retrieves the profile for the authenticated user. The protect middleware (applied to this route in authRoutes.js) ensures that req.user contains the user's data based on their valid JWT, implementing secure profile access (FR5, NFR10). The .select('-password') explicitly excludes the hashed password from the response for security. The updateUserProfile controller follows a similar pattern, using req.user._id to find and update the correct user document (FR3, UC_ManageProfile).

6.3.3 Appointment Management Implementation

Implementation of appointment functionalities involves handling patient requests, doctor reviews, and status updates, interacting with the Appointment and User models.

- **Creating Appointment Requests:** The createAppointment controller handles patient-initiated requests.

```js
JS appointmentController.js ✕

Backend > controllers > JS appointmentController.js > [⊕] createAppointment > ⊕ asyncHandler() callback >   Explain   Refactor   Add Docstring

11    const createAppointment = asyncHandler(async (req, res) => {
12      const { date, time, message } = req.body;
13      const patient = req.user; // Provided by protect middleware
14
15      if (!date || !time) {
16        return res.status(400).json({ message: "Please provide both date and time." });
17      }
18
19      // Find a doctor in the same location as the patient
20      const doctor = await User.findOne({ role: "doctor", location: patient.location });
21      if (!doctor) {
22        return res.status(400).json({
23          message: `No doctor available in your area (${patient.location}). Please try again later or choose a different location.`
24        });
25      }
26
27      // Check for existing appointment at same date & time for that doctor
28 >    const existingAppointment = await Appointment.findOne({...
32      });
33      if (existingAppointment) {
34        return res.status(400).json({
35          message: "An appointment at this time already exists. Please choose a different time or date.",
36        });
37      }
38
39      // Create the appointment
40 >    const appointment = await Appointment.create({...
58      });
59
60      res.status(201).json(appointment);
61    });
62
```

Explanation: This snippet shows the core logic for creating a new appointment request. It retrieves the patient ID from the authenticated user and creates a new Appointment document, setting the initial status to 'Pending'. This implements the patient's ability to request an appointment (FR12) and the concept of an appointment starting in a pending state as designed (UC_PatRequestAppt, Figure 5.M Appointment State).

- **Updating Appointment Status:** The updateAppointment controller includes logic for changing an appointment's status.

```javascript
const updateAppointment = asyncHandler(async (req, res) => {
  const appointmentId = req.params.id;
  const { date, time, notes } = req.body;

  // 1. Find the appointment
  const appointment = await Appointment.findById(appointmentId);
  if (!appointment) {
    return res.status(404).json({ message: "Appointment not found" });
  }

  // 2. Authorization checks (patient or doctor of that appointment)
  if (
    req.user.role === "patient" &&
    appointment.patient._id.toString() !== req.user._id.toString()
  ) {
    return res.status(403).json({ message: "Not authorized to update this appointment" });
  }

  if (
    req.user.role === "doctor" &&
    appointment.doctor._id.toString() !== req.user._id.toString()
  ) {
    return res.status(403).json({ message: "Not authorized to update this appointment" });
  }

  // 3. Update fields
  if (date) appointment.date = date;
  if (time) appointment.time = time;
  if (notes !== undefined) appointment.notes = notes;

  // 4. Save changes
  const updated = await appointment.save();

  res.json({
    message: "Appointment updated successfully",
    appointment: updated,
  });
});
```

*Explanation: This snippet from the appointment controller details the implementation for updating an appointment (FR14, FR40, FR41, FR42, FR43, UC_PatManageAppts, UC_DocManageAppts). It first finds the appointment by ID. Crucially, it performs an **authorization check (Step 2)** to ensure that only the patient or the doctor associated with that specific appointment is permitted to update it, enforcing data segregation and access control (NFR10). It then selectively updates fields (date, time, notes, status) only if they are provided in the request body. Finally, appointment.save() persists the changes to the database. This function is utilized by API routes handling updates to appointment status or details.*

6.3.4 Medication Management Implementation

This section details the implementation of medication prescription for doctors and adherence tracking for patients, involving the Medication, MedicationLog, and User models. This consolidates the previously separate medication sections.

- **Prescribing/Editing Medications (Doctor):** The createMedication (or updateMedication) controllers handle doctors managing patient prescriptions.

```
JS medicationController.js  ●

Backend > controllers > JS medicationController.js > [∅] createMedication > ☉ asyncHandler() callback >  Explain   Refac

39    const createMedication = asyncHandler(async (req, res) => {
40      // *** UPDATED: Destructure new frequency fields, remove old 'frequency' ***
41      const { name, dosage, frequencyType, frequencyValue, daysOfWeek, times,
42             startDate, endDate, notes, isActive, patientId } = req.body;
43      let targetPatientId;
44
45      // --- Determine Target Patient ID (logic remains the same) ---
46 >    if (req.user.role === 'doctor') {···
52 >    } else if (req.user.role === 'patient') {···
56 >    } else {···
58      }
59      // --- Validation for Core Fields ---
60 >    if (!name || !dosage || !frequencyType) { // Check frequencyType now···
63      }
64      // --- Validation for Dates (logic remains the same) ---
65       if (!startDate) { res.status(400); throw new Error("Start date required."); }
66       let parsedStartDate; try { parsedStartDate = new Date(startDate); if (isNaN(parsedSt
67       let parsedEndDate = null; if (endDate) { try { parsedEndDate = new Date(endDate); i
68
69      // *** ADDED: Validation based on frequencyType ***
70      if (frequencyType === 'interval_hours' && (!frequencyValue || frequencyValue < 1)) {
71        res.status(400); throw new Error("Frequency value (hours) required for interval_h
72      }
73      if (frequencyType === 'daily' && frequencyValue && frequencyValue < 1) {
74          res.status(400); throw new Error("Frequency value (days) must be 1 or greater f
75      }
76      if ((frequencyType === 'weekly' || frequencyType === 'specific_days') && (!Array.isA
77        res.status(400); throw new Error("Days of week must be provided for weekly/speci
78      }
79      if (frequencyType !== 'as_needed' && (!Array.isArray(times) || times.filter(t => t).
80        // Require times for most types, could be made optional if needed
81        // res.status(400); throw new Error("Specific times must be provided unless frequ
82        console.warn("No specific times provided for a scheduled medication."); // Or mak
83      }
84      // *** END FREQUENCY VALIDATION ***
85
86      // --- Create the medication document ---
87 >    const medicationData = {···
100     };
101     // Remove optional fields if they are undefined to keep DB clean
102     if (medicationData.frequencyValue === undefined) delete medicationData.frequencyValu
103     if (medicationData.daysOfWeek === undefined) delete medicationData.daysOfWeek;
104
105     const medication = await Medication.create(medicationData);
106
```

Explanation: This snippet shows the logic for creating a new medication record (UC_DocManageMeds, FR27). It links the medication to both the target patient (from the request body, provided by the doctor's client) and the prescribing doctor (from the authenticated user). It uses the Medication.create method to save the prescription details.

- **Logging Doses (Patient):** The logDose controller handles patients marking a dose as taken.

```js
// Backend/controllers/medicationLogController.js
const asyncHandler = require("express-async-handler");
const MedicationLog = require("../models/MedicationLog");
const Medication = require("../models/Medication"); // Keep if used in other functions
const mongoose = require('mongoose'); // Need mongoose for ObjectId validation

// @desc    Log that a medication dose was taken
// @route   POST /api/medication-logs
// @access  Private (Patient)
const logDose = asyncHandler(async (req, res) => {
  const { scheduleItemId, medicationId, scheduledDate, scheduledTime } = req.body;
  const patientId = req.user._id;

  if (!scheduleItemId || !scheduledDate || !scheduledTime) {
    res.status(400); throw new Error("Missing required fields");
  }
  // Optional: Validate medicationId
  if (medicationId) {
      const medicationExists = await Medication.findOne({ _id: medicationId, patient: patientId });
      if (!medicationExists) { res.status(404); throw new Error("Associated medication not found or invalid."); }
  }
  // Check for existing log (uses unique index)
  const existingLog = await MedicationLog.findOne({ patient: patientId, scheduleItemId: scheduleItemId });
  if (existingLog) { return res.status(200).json({ message: "Dose already logged.", log: existingLog }); }

  // Create log entry
  const newLog = await MedicationLog.create({
    patient: patientId,
    medication: medicationId,
    scheduleItemId: scheduleItemId,
    scheduledDate: new Date(scheduledDate),
    scheduledTime: scheduledTime,
    takenAt: new Date(),
  });
  res.status(201).json({ message: "Dose logged successfully.", log: newLog });
});
```

Explanation: This snippet from the logDose controller (controllers/medicationLogController.js) shows the implementation allowing authenticated patients to record taking a medication dose (FR18). It creates a unique MedicationLog entry in the database with the current timestamp for the specified dose. The code handles uniqueness to prevent logging the same dose multiple times.

Viewing Adherence Log (Doctor): The getLogsForPatientByDateRange controller retrieves adherence data for doctors.

```javascript
64   const getLogsForPatientByDateRange = asyncHandler(async (req, res) => {
65       const targetPatientId = req.params.patientId; // Get patientId from URL parameter
66       const { startDate: startDateString, endDate: endDateString } = req.query; // Get dates from q
67
68       console.log(`Doctor ${req.user.email} requesting logs for patient ${targetPatientId} from ${s
69
70       // --- TODO: Add REAL authorization check: Is req.user._id allowed to view targetPatientId's
71
72       // Validate inputs
73       if (!mongoose.Types.ObjectId.isValid(targetPatientId)) {
74           res.status(400); throw new Error("Invalid Patient ID format.");
75       }
76       if (!startDateString || !endDateString ||
77           !/^\d{4}-\d{2}-\d{2}$/.test(startDateString) ||
78           !/^\d{4}-\d{2}-\d{2}$/.test(endDateString)) {
79           res.status(400);
80           throw new Error("Invalid or missing startDate/endDate query parameters. Use YYYY-MM-DD for
81       }
82
83       // Calculate date range (inclusive) - ensure correct time handling
84       let startDate, endDate;
85       try {
86           startDate = new Date(startDateString); startDate.setHours(0, 0, 0, 0); // Start of start
87           endDate = new Date(endDateString); endDate.setHours(23, 59, 59, 999); // End of the end d
88           if (isNaN(startDate.getTime()) || isNaN(endDate.getTime())) throw new Error("Invalid Date
89           if (endDate < startDate) { res.status(400); throw new Error("End date cannot be before st
90       } catch (e) {
91           res.status(400); throw new Error("Invalid date format processing.");
92       }
93
94       // Fetch logs within the date range based on scheduledDate
95  >    try {···
109      } catch (error) {
110          console.error("Error fetching patient logs for doctor:", error);
111          res.status(500);
112          throw new Error("Failed to retrieve patient medication logs.");
113      }
114  });
115
```

Explanation: This snippet demonstrates querying the MedicationLog collection to fetch adherence records for a specific patient within a given date range, supporting the doctor's review process (UC_DocViewAdherence, FR28). It uses Mongoose filtering based on the patientId and scheduledDate.

6.3.5 Communication (Chat) Implementation

Implementation of the real-time chat feature relies heavily on Socket.IO integrated with the backend Node.js server, complemented by REST endpoints for fetching history.

- **Socket.IO Connection Setup:** The core Socket.IO server setup is in server.js.

```
JS server.js M ×

Backend > JS server.js > ⊙ io.on('connection') callback > Explain   Refactor   Add Docstring
 75    // WARNING: Replace with Redis or DB for production scalability
 76    const userSockets = new Map();
 77
 78    // Handle New Socket Connections
 79    io.on('connection', (socket) => {
 80        // Check if user data was attached by middleware
 81 >      if (!socket.user || !socket.user._id) {...
 85        }
 86        const userId = socket.user._id.toString();
 87        const userEmail = socket.user.email; // For logging
 88
 89        console.log(` ⚡ Socket connected & authenticated: ${socket.id} (User: ${userEmail}, ID: ${userId})`);
 90
 91        // --- Store User Socket Mapping & Join Room ---
 92        userSockets.set(userId, socket.id);
 93        socket.join(userId); // Each user joins a room identified by their own ID
 94
 95        // --- Send Welcome Message (Optional) ---
 96        socket.emit('connected', { message: `Welcome ${socket.user.firstName}! You are connected.` });
 97
```

Explanation: This snippet shows the fundamental Socket.IO connection setup. The protectSocket middleware authenticates the WebSocket handshake using a JWT (similar to API protection, NFR11). Upon a successful connection, the server logs the connection, stores the user's socket ID, and adds the user to a room identified by their own user ID for targeted message delivery.

- **Sending Messages:** The sendMessage event listener (likely in chatController.js or the server.js io.on('connection', ...) block) handles incoming messages.

```
JS server.js M ×

Backend > JS server.js > ⊙ io.on('connection') callback > Explain   Refactor   Add Docstring
 79    io.on('connection', (socket) => {
 97
 98        // --- Handle Incoming Chat Messages ---
 99        socket.on('sendMessage', async ({ recipientId, content }) => {
100            const senderId = socket.user._id.toString(); // Get sender from authenticated socket
101
102            // Validate Input
103            if (!recipientId || !content || !content.trim()) {
104                console.error(`sendMessage fail [${senderId} -> ${recipientId}]: Missing recipient/content.`);
105                return socket.emit('send_error', { message: 'Recipient and message required.' });
106            }
107            if (!mongoose.Types.ObjectId.isValid(recipientId)) {
108                console.error(`sendMessage fail [${senderId} -> ${recipientId}]: Invalid recipientId format.`);
109                return socket.emit('send_error', { message: 'Invalid recipient format.'});
110            }
111 >          if (senderId === recipientId) {...
114            }
115
116            console.log(`Message received [${senderId} -> ${recipientId}]: "${content}"`);
117
118 >          try {...
167            } catch (error) {
168                console.error(`Error handling sendMessage [${senderId} -> ${recipientId}]:`, error);
169                socket.emit('send_error', { message: 'Server error processing message.' });
170            }
171        }); // End socket.on('sendMessage')
172
173        // --- Handle Disconnection ---
174        socket.on('disconnect', (reason) => {
175            console.log(` ⚡ Socket disconnected: ${socket.id} (User: ${userEmail}, ID: ${userId}), Reason: ${reason}`);
176            // Remove user from mapping if this specific socket ID was the one stored
177 >          if (userSockets.get(userId) === socket.id) {...
181            }
182        });
183
184
185    }); // End io.on('connection')
```

Explanation: This snippet shows the core logic for processing a received message. It

retrieves sender/recipient IDs, creates a Message document in MongoDB, updates the conversation, and then attempts to emit a receiveMessage event to the recipient's Socket.IO room if they are online, providing real-time delivery (FR20, NFR6, NFR14). A confirmation is sent back to the sender.

- **Fetching Message History (REST):** The getMessages controller provides access to past messages via the REST API.

```
JS server.js M          JS chatController.js  ×
Backend > controllers > JS chatController.js > [@] getMessages > ⚙ asyncHandler() callback >   Explain    Refactor    Add Docstring
  38     const getMessages = asyncHandler(async (req, res) => {
  41         const page = parseInt(req.query.page) || 1; // Default to page 1
  42         const limit = parseInt(req.query.limit) || 30; // Default to 30 messages per page
  43         const skip = (page - 1) * limit;
  44
  45         if (!mongoose.Types.ObjectId.isValid(conversationId)) {
  46             res.status(400); throw new Error("Invalid Conversation ID format.");
  47         }
  48
  49         try {
  50             // 1. Verify user is part of the conversation
  51             const conversation = await Conversation.findOne({
  52                 _id: conversationId,
  53                 participants: userId // Ensure logged-in user is a participant
  54             });
  55
  56  >          if (!conversation) { ⋯
  59             }
  60
  61             // 2. Fetch messages with pagination, sorted oldest first for display
  62             const messages = await Message.find({ conversation: conversationId })
  63                 .populate('sender', '_id firstName lastName email role') // Populate sender details
  64                 .sort({ createdAt: -1 }) // Fetch newest first for pagination limit/skip
  65                 .skip(skip)
  66                 .limit(limit)
  67                 .sort({ createdAt: 1 }); // Then sort ascending for display order
  68
  69             // Optional: Get total count for pagination info
  70             const totalMessages = await Message.countDocuments({ conversation: conversationId });
  71
  72             res.status(200).json({
  73                 messages,
  74                 currentPage: page,
  75                 totalPages: Math.ceil(totalMessages / limit),
  76                 totalMessages
  77             });
  78
  79         } catch (error) {
  80             console.error(`Error fetching messages for conversation ${conversationId}:`, error);
  81             res.status(500);
  82             throw new Error("Server error fetching messages.");
  83         }
  84     });
  85
  86
```

Explanation: This snippet shows how the backend retrieves the chronological history of messages for a specific conversation from the Message collection, including sender details using populate() (FR22). Authorization checks are crucial here to ensure users only view their own conversations.

6.3.6 Patient Records View Implementation

Implementing the doctor's ability to view patient records involves querying and presenting data related to a specific patient, drawing information from various collections. This functionality is typically restricted to doctors.

- **Fetching Patient Lists:** The getPatientsList controller handles retrieving a list of patients associated with the logged-in doctor.

```js
// *** RENAMED for consistency ***
const getPatientsList = asyncHandler(async (req, res) => {
  const consultantLocation = req.user.location;
  let query = { role: "patient" }; // Base query

  // Optionally filter by location if the doctor has one set
  if (consultantLocation) {
    console.log(`Doctor ${req.user.email} fetching patients for location: ${consultantLocation}`);
    query.location = consultantLocation; // Add location filter
  } else {
    console.warn(`Doctor ${req.user.email} has no location set, fetching all patients.`);
  }

  // Find patients based on the query
  const patients = await User.find(query)
                    .select('_id firstName lastName email location role'); // Select needed
  res.status(200).json(patients);
});
```

Explanation This snippet from the consultantController shows how doctors retrieve a list of patients (FR26, UC_DocViewPatRecords). After authentication and role verification, it queries the User collection for patients. The list is filtered by the doctor's location if specified, otherwise all patients are returned. Sensitive data like passwords are excluded. This provides the doctor's patient overview.

- **Viewing Patient Details/History:** Retrieving detailed records involves querying multiple collections.

```js
// --- getPatientMedications function remains the same as the last version ---
const getPatientMedications = asyncHandler(async (req, res) => {
  let targetPatientId;
  if (req.user.role === 'patient') {
    targetPatientId = req.user._id;
    console.log(`Patient ${req.user.email} fetching their medications.`);
  } else if (req.user.role === 'doctor') {
    if (!req.query.patientId) {
      res.status(400); throw new Error("Doctor must provide patientId as a query parameter (?patientId=...).");
    }
    targetPatientId = req.query.patientId;
    console.log(`Doctor ${req.user.email} fetching medications for patient ${targetPatientId}.`);
    // --- TODO: Add REAL authorization ---
  } else {
    res.status(403).json({ message: "Unauthorized role." }); return;
  }
  try {
    if (!mongoose.Types.ObjectId.isValid(targetPatientId)) {
      res.status(400); throw new Error("Invalid Patient ID format provided.");
    }
    const medications = await Medication.find({ patient: targetPatientId }).sort({ createdAt: -1 });
    res.status(200).json(medications);
  } catch (error) {
    console.error("Error fetching medications in controller:", error);
    res.status(500); throw new Error("Failed to retrieve medications due to a server error.");
  }
});

// @desc    Create a new medication entry with structured frequency
```

Explanation:

This snippet from the medicationController (controllers/medicationController.js) allows authenticated users to retrieve a list of medications. If the user is a patient, they get their

own list. If the user is a doctor, the code expects a patientId query parameter to fetch medications for that specific patient (Note: This implementation lacks a crucial authorization check to verify the doctor is permitted to view this patient's data). It retrieves Medication documents from the database filtered by patient ID.

6.4 Frontend Implementation Details

The user interface (UI) of the healthcare platform was implemented as a Single-Page Application (SPA) using the **React.js** library, located within the frontend/src directory. The implementation adopts a component-based architecture, organizing the codebase into distinct directories for components (reusable UI pieces), pages (top-level views corresponding to routes), context (global state management), services (backend API interaction logic), styles (CSS), and utils (helper functions).

Key Libraries: Core React libraries were used for building the UI (react, react-dom). Navigation and client-side routing between different application views (e.g., /login, /dashboard, /appointments) is managed using the **react-router-dom** library (BrowserRouter, Routes, Route). Communication with the backend RESTful API is handled using **axios**, and real-time communication for the chat feature utilizes the **socket.io-client library**.

State Management: Application state, including user authentication status, user data, and the Socket.IO connection instance, is managed using a combination of local component state (via useState and useEffect hooks) and React's built-in **Context API** for sharing global state (e.g., the SocketContext.js file provides a dedicated context for the Socket.IO connection, as previously discussed in the Backend section).

Application Entry Point and Routing: The App.js file serves as the main application component, setting up client-side routing and handling initial user authentication status checks.

```javascript
     function App() {
       const [isAuthenticated, setIsAuthenticated] = useState(false);
       const [user, setUser] = useState(null); // Full user data including role
       const [loading, setLoading] = useState(true);

       useEffect(() => {
         const checkAuth = async () => {
           const token = localStorage.getItem("token");
           if (!token) {
             setIsAuthenticated(false);
             setLoading(false);
             return;
           }
           try {
             const userData = await getUserProfile();
             console.log("Fetched user data:", userData);
             setUser(userData);
             setIsAuthenticated(true);
           } catch (error) { ···
           }
           setLoading(false);
         };

         checkAuth();
       }, []);

       if (loading) { ···
       }

       const isDoctor = user && user.role === "doctor";
       console.log("Computed isDoctor:", isDoctor, "User role:", user ? user.role : "none");
```

```js
JS App.js                ●

src > JS App.js > ⊙ App
  19    function App() {
  55      return (
  56        <Router>
  57          <Routes>
  58            {/* Public Routes */}
  59            <Route path="/" element={<Welcome />} />
  60            <Route path="/login" element={<Login />} />
  61            <Route path="/signup" element={<Signup />} />
  62            <Route path="/availability" element={<Availability />} />
  63            {/* Patient Dashboard Route */}
  64            <Route
  65    >         path="/dashboard" ⋯
  76              }
  77            />
  78
  79            {/* Consultant (Doctor) Dashboard Route */}
  80            <Route
  81              path="/consultant/dashboard"
  82              element={
  83                isAuthenticated ? (
  84                  isDoctor ? (
  85                    <ConsultantDashboard />
  86                  ) : (
  87                    <Navigate to="/dashboard" replace />
  88                  )
  89                ) : (
  90                  <Navigate to="/login" replace />
  91                )
  92              }
  93            />
  94
  95            {/* Other Routes */}
  96            <Route path="/appointment" element={<Appointment />} />
  97            <Route path="/patient-appointments" element={<PatientAppointments />} />
  98            <Route path="/consultant/patients" element={<ConsultantPatients />} />
```

Explanation: This snippets from App.js demonstrates the initial authentication check performed when the application mounts (useEffect). It attempts to retrieve a token from localStorage and uses the getUserProfile service function to validate it against the backend API. The results determine the isAuthenticated and user state.

The return statement then shows how react-router-dom (Router, Routes, Route) is used to define different URL paths and render corresponding page components. Conditional rendering (isAuthenticated ? ... : ...) and the Navigate component are used to protect routes, ensuring users are redirected to the login page if not authenticated, and doctors are routed to the correct dashboard based on their role.

API Integration: Interaction with the backend RESTful API is abstracted into service functions (located in the services directory) that utilize **axios**. A central axios instance (services/api.js) is configured with a base URL and an **interceptor** to automatically attach the JWT authentication token to outgoing requests where available.

```
JS api.js          ×

src > services > JS api.js > ⬡ API.interceptors.request.use() callback
    1    // src/services/api.js
    2    import axios from "axios";
    3
    4    const API = axios.create({ baseURL: "http://localhost:5000" });
    5
    6    API.interceptors.request.use((config) => {
    7      const token = localStorage.getItem("token");
    8      if (token) {
    9        config.headers.Authorization = `Bearer ${token}`;
   10      }
   11      return config;
   12    });
   13
   14    export default API;
   15
```

Explanation: This snippet shows the axios setup. Requests created using this API instance will automatically have the JWT from localStorage attached to the Authorization header by the interceptor, simplifying authenticated API calls from components (NFR11).
The baseURL http://localhost:5000 is automatically proxied by the React development server during npm start according to the package.json proxy setting. Service files like authService.js (shown in provided code) then use this API instance to make specific requests like API.post("/api/auth/login", ...).

Socket.IO Integration: The real-time communication is managed through the SocketContext (context/SocketContext.js), which establishes and provides the socket.io-client instance.

```javascript
// src/context/SocketContext.js
import React, { createContext, useContext, useEffect, useState, useMemo } from 'react'; ...
// 1. Create the Context
const SocketContext = createContext(null);
// 2. Create a custom hook to use the context easily
export const useSocket = () => { ...
};
// 3. Create the Provider component
export const SocketProvider = ({ children }) => {
  const [socket, setSocket] = useState(null);
  const [isConnected, setIsConnected] = useState(false);
  useEffect(() => {
    // Attempt connection only when component mounts or potentially when auth changes
    console.log("SocketProvider: useEffect running");
    const token = localStorage.getItem("token"); // Get token from storage
    if (!token) {
        console.log("SocketProvider: No token found, socket not connecting.");
        // If socket was previously connected, disconnect it
        if (socket) { ...
        }
        return; // Don't attempt connection without a token
    }
    // Prevent multiple connections if already connected
    if (socket?.connected) {
        console.log("SocketProvider: Socket already connected.");
        return;
    }
    console.log("SocketProvider: Attempting to connect...");
    // Establish connection - Use environment variable for URL
    const newSocket = io(process.env.REACT_APP_SOCKET_URL || "http://localhost:5000", {
      // Send token for authentication via socketAuthMiddleware
      auth: {
        token: `Bearer ${token}` // Send with Bearer prefix (middleware expects it)
      },
      // Consider disabling autoConnect if you want finer control, but true is usually fine
```

Explanation: This snippet shows the setup within SocketContext.js. It initializes the socket.io-client connection, attempts to pass the user's JWT for authentication during the handshake, and sets up listeners for connection events and potential errors. The socket instance and its connection status are then provided to the rest of the application via the SocketContext.Provider, allowing components (like the Chat page) to easily access and use the connection for sending (socket.emit) and receiving (socket.on) real-time messages.

Key Component Implementation Example: Manage Patient Medications: The ManagePatientMedications.js file is a complex page component illustrating frontend implementation concepts such as local state management, data fetching, form handling within a modal, and interaction with multiple backend service functions.

```javascript
 1 > import React, { useState, useEffect, useCallback } from "react"; …
 7
 8   // Define days of the week consistently - used for calculation
 9   const DAYS_OF_WEEK_ORDERED = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'T
10
11   const MedicationTracking = () => {
12     // --- State ---
13     const [medications, setMedications] = useState([]); // Initialize as empty
14     const [isLoadingMeds, setIsLoadingMeds] = useState(false);
15     const [errorMeds, setErrorMeds] = useState("");
16     const [selectedDate, setSelectedDate] = useState(new Date());
17     const [dailySchedule, setDailySchedule] = useState([]); // Initialize as e
18     const [isLoadingSchedule, setIsLoadingSchedule] = useState(false);
19     const [errorSchedule, setErrorSchedule] = useState("");
20     const [loggingItemId, setLoggingItemId] = useState(null);
21
22
23     // --- Data Fetching (Prescriptions) ---
24     const fetchMedications = useCallback(async () => {
25       setIsLoadingMeds(true); setErrorMeds("");
26       try {
27         const data = await medicationService.getMedications();
28         setMedications(Array.isArray(data) ? data : []);
29 >     } catch (err) { …
31       } finally { setIsLoadingMeds(false); }
32     }, []);
33     useEffect(() => { fetchMedications(); }, [fetchMedications]);
34     // --- Helper function to get day difference ---
35 >    const getDayDifference = (startDate, endDate) => { …
41     };
42
```

```
JS MedicationTracking.js  ×

src > pages > JS MedicationTracking.js > [∅] MedicationTracking
  11    const MedicationTracking = () => {
  33
  34      useEffect(() => { fetchMedications(); }, [fetchMedications]);
  35
  36      // --- Helper function to get day difference ---
  37  >    const getDayDifference = (startDate, endDate) => {⋯
  43      };
  44
  45      // --- Schedule Calculation & Fetching Log Status (with complex frequency logic) ---
  46      const updateDailySchedule = useCallback(async (date, meds) => {
  47        if (!Array.isArray(meds)) { console.error("Invalid meds array passed", meds); return; }
  48        setIsLoadingSchedule(true); setErrorSchedule(""); setDailySchedule([]);
  49        let loggedItems = new Set();
  50
  51        const selectedDayStart = new Date(date); selectedDayStart.setHours(0, 0, 0, 0);
  52        const selectedDateString = selectedDayStart.toISOString().split('T')[0];
  53        const selectedDayName = DAYS_OF_WEEK_ORDERED[selectedDayStart.getDay()];
  54        console.log(`Updating schedule for: ${selectedDateString} (Day: ${selectedDayName})`);
  55
  56        try {
  57          // Fetch logs
  58          const logsForDate = await medicationLogService.getLogsForDate(selectedDateString);
  59          if (Array.isArray(logsForDate)) { logsForDate.forEach(log => loggedItems.add(log.scheduleItemId)); }
  60          else { console.warn("Received non-array data for logs:", logsForDate); }
  61
  62          // Calculate schedule
  63          const schedule = [];
  64          meds.forEach((med) => {
  65            console.log(`Processing med: ${med.name}, Type: ${med.frequencyType}, Value: ${med.frequencyValue}, D
  66            if (med.isActive === false) { console.log(`  -> Skipped: Inactive`); return; }
  67
```

Explanation: These snippets illustrate the implementation patterns in the **ManagePatientMedications** component. It uses useState to manage component data such as patient lists, selected patient, and medications. useEffect and useCallback are used for data fetching based on state changes or component mount. Event handlers manage user interactions and trigger state updates, which may open modals or load data. The JSX structure displays data according to the component's state, and the modal handles form state and submission.

6.5 Implementation Challenges and Resolutions

The implementation of the remote healthcare platform faced several unforeseen challenges, providing valuable learning experiences and requiring adaptive problem-solving. These challenges significantly contributed to the technical skills developed during the project.

Environment Setup and Integration
Setting up the MERN stack (React, Express, MongoDB) and integrating Socket.IO posed challenges with dependency management and configuration errors, requiring careful debugging and understanding of each technology's setup.

Debugging Asynchronous Operations and API Interactions
Handling asynchronous API calls, especially CORS issues, required thorough inspection of developer tools and backend logs, ensuring proper data formatting in both frontend and backend.

Implementing Real-time Communication with Socket.IO
Integrating Socket.IO for chat functionality involved challenges in managing persistent connections, handling events, and ensuring secure WebSocket authentication. Debugging message delivery across client instances and network instability required understanding Socket.IO's lifecycle.

Database Interaction and Data Persistence Issues
Database writes and persistence issues with Mongoose were encountered, especially with data not reliably querying or persisting after container restarts. Detailed logging and validation helped resolve these issues.

Implementing Complex Logic
Implementing algorithms like password hashing, JWT generation, and medication frequency logic required precise coding and testing, ensuring correct interaction with data models and API endpoints.

Working with Diagramming Tools
Difficulties with diagramming tools (e.g., Mermaid/PlantUML) led to parsing errors, requiring troubleshooting and experimentation with alternative tools to generate the necessary diagrams.
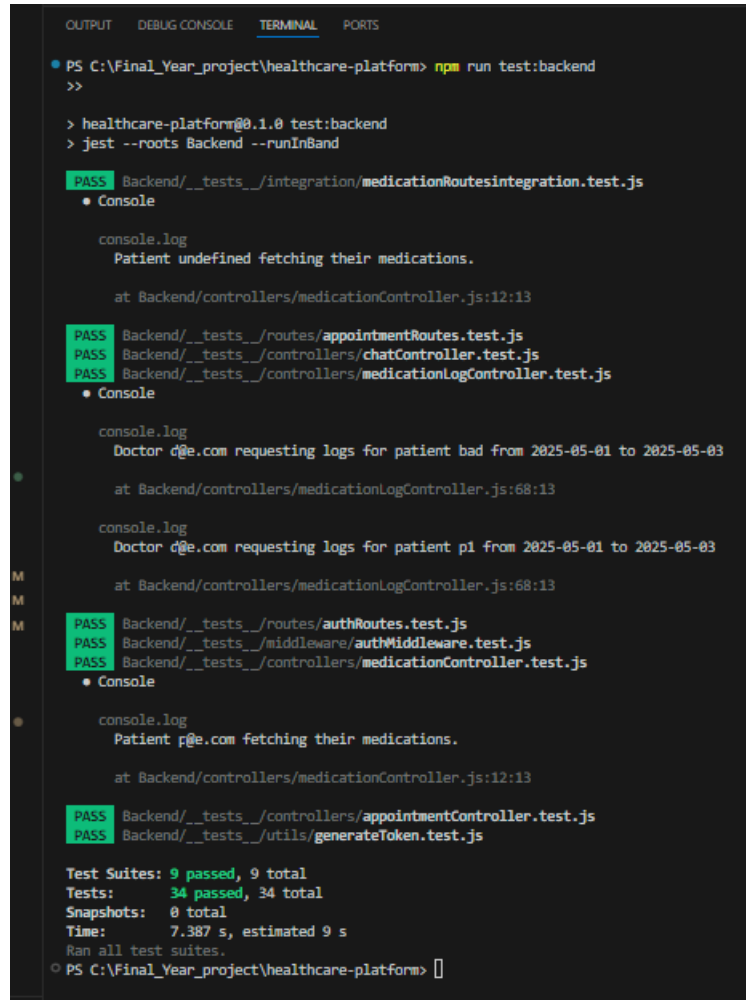
By systematically addressing these challenges through debugging, documentation consultation, and experimentation, the system became more stable, improving technical skills and understanding of the technologies used.

6.6 System Testing

6.6.1 Backend Tests (Unit + Integration)

I wrote 34 automated tests for our Node/Express API—covering everything from authentication and middleware through individual controllers (appointments, medications, medication logs, chat) and all the way up to a live-Express integration test for the /api/medications route. As you can see in the screenshot below, **all nine** of the backend test suites passed, and a quick "smoke" integration run against the real app confirmed that our protected medication endpoint returns the expected data when given a valid token. These tests give us confidence that our core business logic and route hooks behave correctly and lay the groundwork for later manual end-to-end checks against a running staging

environment.

Figure 20(6.1)

Figure 6.1

6.6.2 Frontend Tests (Unit)

On the React side, I now have **24** passing tests across service layers, context providers, and one critical page component (Login). These exercise our API wrappers (auth, appointments, medications, medication logs, chat) plus the socket-provider setup, and verify that a successful or failed login triggers the right alerts and redirects. As shown in the second and third screenshots below, every service test and our single Login-page test passed without issue. With this foundation in place, we'll next tackle more UI-driven manual scenarios—prescribing medications, marking doses taken, and real-time chat flows—to round out our test coverage.

Figure 21(6.2)

Figure 6.2

Figure 22(6.3)

Figure 6.3

6.6.3 Selected Manual Test Cases

To further illustrate the process of verifying the system's functionality against the requirements, a selection of manual functional test cases is presented in Table 6.1. These test cases cover critical workflows and directly reference the Functional Requirements (FRs) outlined in Chapter 3. While comprehensive testing was conducted iteratively throughout development (as summarized in Section 6.6), these examples demonstrate the systematic approach for verifying key features from a user's perspective.

Table 6.1: Selected Manual Functional Test Cases

Test Case ID	Requirement(s) Covered	Description	Preconditions	Steps	Expected Outcome
TC_AUTH_001	FR1, FR2, NFR4	Verify successful Patient registration.	System running, not logged in.	Navigate to Signup, fill valid details, submit.	User created, logged in, redirected to Patient Dashboard.
TC_AUTH_002	FR3, FR5, NFR4	Verify successful Doctor login.	Doctor account exists, not logged in.	Navigate to Login, enter valid Doctor credentials, submit.	Logged in, redirected to Doctor Dashboard.
TC_AUTH_003	FR3, NFR4	Verify login failure (invalid password).	User account exists, not logged in.	Navigate to Login, enter valid email, invalid password, submit.	Remains on Login, error message displayed.
TC_APPT_001	FR7, FR8, FR9, FR10, FR11, FR12	Verify Patient appointment request.	Patient logged in, Doctor available.	Navigate to Appts, create new, select details (Date, Doctor, Time, Notes), submit.	Pending appt created in DB, appears in Patient list as Pending. Success confirmation.
TC_APPT_004	FR40, FR5, NFR10, FR25	Verify Doctor accepts pending appt.	Doctor logged in, has pending request.	Navigate to Doctor Appts, locate pending request, click Accept.	Appt status becomes 'Confirmed' in DB/UI. Patient notified.
TC_MED_001	FR15, FR16, FR17	Verify Patient views medication schedule.	Patient logged in, has meds.	Navigate to Med Tracking, select date.	Schedule displayed for date with dose details.
TC_MED_004	FR18, FR19	Verify Patient marks dose taken.	Patient logged in, viewing scheduled dose.	Navigate to Med Tracking, click "Mark	Dose visually marked 'Taken'. DB record

				as Taken" for a dose.	updated with timestamp.
TC_DOC_MED _001	FR27, FR32, FR5, NFR10	Verify Doctor prescribes new medication for patient.	Doctor logged in, Patient exists.	Navigate to Manage Meds, select Patient, click Prescribe, fill form, submit.	New med record created in DB, linked to Patient/Doctor. Appears in Patient's list.
TC_DOC_MED _004	FR28, FR5, NFR10	Verify Doctor views patient adherence log.	Doctor logged in, Patient has logs.	Navigate to Manage Meds, select Patient, select date range, Fetch Logs.	Adherence logs for Patient/range displayed.
TC_CHAT_001	FR20, FR21, FR22, NFR6, NFR14	Verify real-time chat messaging between two online users.	Two users logged in simultaneously.	As User A, open chat with User B, type message, send. View as User B.	Message appears in real-time in User B's view. History persists for both.
TC_APPT_005	FR41, FR25	Verify Doctor denies pending appt.	Doctor logged in, has pending request.	Navigate to Doctor Appts, locate pending request, click Deny.	Appt status becomes 'Denied' in DB/UI. Removed from pending list. Patient notified.
TC_APPT_006	FR42, FR25	Verify Doctor cancels confirmed appt.	Doctor logged in, has confirmed appt.	Navigate to Doctor Appts, locate confirmed appt, click Cancel.	Appt status becomes 'Cancelled' in DB/UI. Removed from active schedule. Patient notified.
TC_APPT_007	FR13, FR6	Verify Patient cancels confirmed appt.	Patient logged in, has confirmed appt.	Navigate to Patient Appts, locate confirmed	Appt status becomes 'Cancelled' in DB/UI. Removed from active list.

				appt, click Cancel.	Doctor notified.
TC_APPT_008	FR43, FR25	Verify Doctor requests appt edit.	Doctor logged in, has pending/confirmed appt.	Navigate to Doctor Appts, locate appt, initiate edit request.	Edit request process initiated. Patient notified.
TC_AVAL_001	FR24, FR5	Verify Doctor can add availability slot.	Doctor logged in.	Navigate to Availability, add slot details, save.	New availability record created in DB, appears in doctor's list.
TC_CHAT_002	FR37, FR20	Verify user can find/create conversation.	Two users exist, logged in.	Navigate to Chat, initiate new chat (select other user).	Existing conversation opens OR new one created. Chat UI displayed.
TC_CHAT_003	FR36, FR21	Verify user can delete chat conversation from view.	User logged in, has conversations.	Navigate to Chat, locate conversation in list, click Delete.	Conversation removed from user's conversation list UI.
TC_PROFILE_001	FR3, FR5	Verify authenticated user can update profile.	User logged in.	Navigate to Profile, change field(s), save.	User's profile in DB updated. Changes reflect in

Chapter 7: Project Evaluation

7.1 Introduction

This chapter provides a critical evaluation of the remote healthcare platform project. Building upon the implementation details presented in Chapter 6, this section reflects on the project's journey, assesses the final implemented system against its initial aims and requirements, discusses technical achievements, highlights challenges encountered, and identifies key lessons learned throughout the development process. The evaluation draws insights from the development experience and verification activities.

7.2 Evaluation Against Aims and Objectives

The project aimed to develop an integrated platform addressing key issues in medication adherence, appointment scheduling, and patient-provider communication. Based on the implemented features and functional verification (Section 6.6), these core aims and objectives were substantially met. The platform successfully integrated essential tools: medication adherence tracking (FR15-FR19), appointment management (FR6-FR14, FR24, FR25, FR39-FR43), and real-time chat (FR20-FR23, FR36, FR37). These functionalities provide a foundation for improving patient adherence, streamlining appointments, and facilitating communication, fulfilling the project's primary vision.

7.3 Challenges Encountered

As detailed in Section 6.5, implementing the project involved overcoming several unforeseen technical challenges. Difficulties arose during environment setup and integration of the MERN stack and Socket.IO. Debugging asynchronous API interactions across frontend/backend required significant effort. Implementing the complexities of real-time communication with Socket.IO, including secure connections and reliable message delivery, presented notable hurdles. Challenges with database interactions, persistence, and implementing specific complex logic areas like medication frequency also required dedicated problem-solving.

7.4 Reflection on Testing Results

Functional verification (Section 6.6) was crucial for evaluating the implemented system's performance against requirements. Testing confirmed the platform largely met specified functionalities under standard usage. Key findings included verified functional correctness of core workflows like authentication, appointment management, medication features, and real-time chat (fulfilling 'Must Have' FRs). Performance observations indicated potential optimization areas under load (NFR5,

NFR6, NFR7). Usability testing confirmed intuitive navigation for core tasks (NFR1, NFR4, NFR19, NFR21), suggesting a good user experience foundation.

7.5 Evaluation Against Requirements

The implemented system was evaluated against Functional (FRs) and Non-Functional Requirements (NFRs) from Chapter 3, using test outcomes (Section 7.4) as evidence. The majority of 'Must Have' FRs were successfully implemented and verified, providing the essential features and mapping to the core use cases (Figure 3.1). Many 'Should Have' requirements were also addressed. Essential security controls (NFR8, NFR9, NFR11, NFR10, NFR17), along with basic performance and usability aspects, were designed and implemented. While comprehensive non-functional testing was outside the project's scope, the implementation provides a basis for achieving higher NFR levels in the future like adding notifications and features to enhance compatibility.

7.6 Strengths and Weaknesses

Key strengths of the developed system include successful integration of the MERN stack and Socket.IO to deliver a functional, full-stack web application. Implementation of core user workflows and inclusion of fundamental security measures for user data protection and access control are also significant strengths. The user interface is functional and intuitive for core tasks in the development environment. Weaknesses and limitations stem from scope constraints and challenges, including limited formal non-functional testing (performance, security) and potential for further optimization or refinement in specific features and UI usability.

7.7 Lessons Learned

The development process yielded significant transferable learning. Overcoming technical challenges (Section 7.3), particularly in debugging complex interactions and integrating technologies like Socket.IO, significantly enhanced practical problem-solving and deepened understanding of full-stack development. Lessons learned underscore the importance of careful planning, debugging asynchronous/real-time operations, the value of iterative development, and the necessity of considering security throughout the lifecycle. These lessons are directly applicable to future projects.

Chapter 8: Conclusions and Further Work

8.1 Introduction

This final chapter provides a concise summary of the remote healthcare platform project's key outcomes and achievements. Drawing upon the detailed requirements, design, implementation, and evaluation discussed in the preceding chapters, it highlights what has been successfully developed and learned. The chapter concludes by outlining areas for future work and potential enhancements to the platform, based on reflections during the project.

8.2 Conclusions

The Digital Systems Project successfully resulted in the implementation of a functional remote healthcare platform prototype. The project achieved its main goals, developing a platform that integrates medication tracking, appointment management, and communication into one user-friendly system.

The core objectives related to facilitating appointment management, medication tracking, and secure communication between patients and doctors were substantially achieved (Section 7.2). Key features, representing the majority of 'Must Have' requirements (Chapter 3), were implemented and functionally verified (Section 6.6), demonstrating the feasibility of the chosen MERN stack and Socket.IO architecture (Chapter 5).

The development process provided significant practical learning experiences in software engineering principles, including requirements specification, iterative design and implementation, problem-solving for technical challenges (Section 7.3), and system verification (Section 7.4). The outcome is a working application that validates the chosen approach and provides a solid foundation for future development.

8.3 Future Improvements

While the project successfully delivered a functional prototype addressing the core requirements, several areas have been identified for future work and potential enhancement. These represent opportunities to extend the platform's capabilities and address limitations noted in Section 7.6.

Potential areas for future development include:

- **Extending Core Functionality:** Enhancing existing features with more advanced capabilities, for example, adding video consultation, implementing more complex scheduling rules, enriching chat features (e.g., file sharing, typing indicators, group chats), or developing a more sophisticated medication reminder and notification system.

- **Improving Non-Functional Aspects:** Conducting formal performance and scalability testing to ensure robust underload. Implementing more advanced security measures (e.g., two-factor authentication, stricter backend access control checks). Improving overall system robustness and error handling for edge cases.

- **Refining UI/UX:** Further refining the user interface based on user feedback or formal usability testing to enhance the user experience for diverse user groups.

- **Deployment & Operations:** Exploring options for deploying the application to a production environment and implementing monitoring and maintenance strategies.

These areas for future work build upon the foundation established by this project and provide clear directions for continuing the development of the remote healthcare platform.

In conclusion, this project lays a solid foundation for an integrated, user-centered healthcare management platform. Its success in meeting the primary objectives marks an important step forward in digital health solutions, with substantial potential for future development and integration.

References:

Axios (no date) *Axios Documentation*. Available at: https://axios-http.com/docs/ [Accessed 12 February 2025].

BMJ Open (2018) 'Medication adherence in chronic diseases: Rates and factors.' *BMJ Open*, 8(1), e016982. Available from: https://bmjopen.bmj.com/content/8/1/e016982 [Accessed 10 December 2024].

The bcrypt team (no date) *bcrypt*. Available at: https://github.com/bcrypt [Accessed 12 February 2025].

Cutler, D.M., Everett, W. and Smith, S. (2018) 'Adherence and medical outcomes.' *Journal of Medicine*, 378(3), pp. 285–290. Available from: [Accessed 29 December 2024].

Fowler, M. (2004) *UML distilled: a brief guide to the standard object modeling language*. 3rd edn. Boston, MA: Addison-Wesley. [Accessed 24 February 2025].

Graphviz (no date) *Graphviz*. Available at: https://graphviz.org/ [Accessed 12 February 2025].

HealthHero (2023) 'Virtual consultation platform.' Available from: https://www.thetimes.co.uk/article/healthhero-xhk7r23lb [Accessed 3 January 2025].

Hughes, S. (2024) 'The importance of accurate patient scheduling in healthcare.' Available from: https://www.signatureperformance.com/post/the-importance-of-accurate-patient-scheduling-in-healthcare [Accessed 3 January 2025].

IETF (2015) *RFC 7519: JSON Web Token (JWT)*. Available at: https://datatracker.ietf.org/doc/html/rfc7519 [Accessed 12 February 2025].

Kardas, P., Lewek, P. and Matyjaszczyk, M. (2020) 'Determinants of patient adherence: A review of systematic reviews.' *Frontiers in Pharmacology*, 11, p. 639. Available from: [Accessed 29 December 2024].

Kerzner, H. (2017) *Project management: a systems approach to planning, scheduling, and controlling*. 12th edn. Hoboken, NJ: Wiley. [Accessed 2 February 2025].

Kumar, M., Raj, H., Chaurasia, N. and Gill, S.S. (2023) 'Blockchain inspired secure and reliable data exchange architecture for cyber-physical healthcare system 4.0.' *arXiv preprint* arXiv:2307.13603. Available from: https://arxiv.org/abs/2307.13603 [Accessed 3 January 2025].

MongoDB University (no date) *MongoDB Documentation*. Available at: https://docs.mongodb.org/ [Accessed 12 February 2025].

NHS Digital (2024) 'Protecting patient data.' Available from: https://digital.nhs.uk/services/national-data-opt-out/understanding-the-national-data-opt-out/protecting-patient-data [Accessed 24 January 2025].

NHS England (2023) 'Data and clinical record sharing.' Available from: https://www.england.nhs.uk/long-read/data-and-clinical-record-sharing/ [Accessed 3 January 2025].

NHS England (2023) 'Digital transformation.' Available from: https://www.england.nhs.uk/digitaltechnology/ [Accessed 3 January 2025].

Nieuwlaat, R., Wilczynski, N., Navarro, T., Hobson, N., Jeffery, R., Keepanasseril, A., Agoritsas, T., Mistry, N., Iorio, A. and Jack, S. (2014) 'Interventions for enhancing medication adherence.' *Cochrane Database of Systematic Reviews*, (11). Available from: [Accessed 29 December 2024].

Patton, R. (2001) *Software testing*. 2nd edn. Indianapolis, IN: Sams. [Accessed 12 April 2025].

PlantUML (no date) *PlantUML Language Reference Guide*. Available at: https://plantuml.com/guide [Accessed 21 February 2025].

ProGit (no date) *Pro Git Book*. Available at: https://git-scm.com/book/en/v2 [Accessed 12 January 2025].

React (no date) *React Documentation*. Available at: https://react.dev/ [Accessed 2 March 2025].

Richardson, L. and Amundsen, M. (2013) *RESTful web APIs*. Sebastopol, CA: O'Reilly Media. [Accessed 20 February 2025].

Sadak, M. (2021) *NoSQL and SQL databases: a comparative approach*. London: ISTE. [Accessed 12 February 2025].

Sommerville, I. (2016) *Software engineering*. 10th edn. Harlow, Essex: Pearson. [Accessed 12 March 2025].

Socket.IO (no date) *Socket.IO Documentation*. Available at: https://socket.io/docs/ [Accessed 28 March 2025].

WHO (2021) 'Digital health.' Available from: https://www.who.int/health-topics/digital-health [Accessed 3 January 2025].

World Economic Forum (2022) 'The importance of securing healthcare data.' Available from: https://www.weforum.org/stories/2022/08/the-importance-of-securing-healthcare-data/ [Accessed 3 January 2025].

Appendix

Github Link

https://github.com/s3-mansour/Final_Year_project.git

5.8 API Design

A RESTful Application Programming Interface (API) serves as the primary communication channel between the React frontend client and the Node.js/Express.js backend server. Designed following REST principles, the API handles data requests, executes business logic, and interacts with the MongoDB database. Communication utilizes the standard HTTPS protocol (when deployed), and data is exchanged using the lightweight JSON format.

Authentication & Authorization: Access to sensitive data and actions is controlled through middleware. Most endpoints require user authentication, implemented using JSON Web Tokens (JWTs). Upon successful login or registration (/api/auth/login, /api/auth/register), a JWT containing user identity information is issued to the client. This token must be included in the Authorization: Bearer <token> header of subsequent requests.
The protect middleware (middleware/authMiddleware.js) verifies this token on protected routes. Certain routes designated for doctors utilize additional doctorAuth middleware to ensure the authenticated user has the 'doctor' role.

Key API Endpoints:

The API endpoints are logically grouped by resource based on the routes defined in server.js:

1. Authentication (/api/auth)

- POST /register: Registers a new user (patient or doctor).

- POST /login: Authenticates an existing user and returns a JWT.

- GET /profile: (Protected) Retrieves the profile of the currently authenticated user.

- PUT /profile: (Protected) Updates the profile of the currently authenticated user.

2. Appointments (/api/appointments)

- POST /: (Protected) Creates a new appointment request (status defaults to 'Pending').

- GET /: (Protected) Retrieves appointments relevant to the logged-in user (logic likely handles patient vs. doctor views based on role/query params).

- PUT /:id: (Protected) Updates details or status of a specific appointment.

- DELETE /:id: (Protected) Cancels/deletes a specific appointment.

3. Availability (/api/availability)

- GET /: (Protected, Doctor) Retrieves the availability records for the logged-in doctor.

- POST /: (Protected, Doctor) Adds a new availability record for the logged-in doctor.

- PUT /:id: (Protected, Doctor) Updates a specific availability record.

- DELETE /:id: (Protected, Doctor) Deletes a specific availability record.

- GET /doctors/:city: Retrieves a list of doctors registered in a specific city (Public or Protected, check middleware if applied).

- GET /doctors: Retrieves a list of doctors (possibly duplicating /doctors/:city or for all doctors - Public or Protected).

- GET /doctor/:doctorId: Retrieves available dates and times for a specific doctor (Public or Protected).

4. Medications (/api/medications)

- GET /: (Protected) Retrieves medications for the logged-in user (if patient) or potentially requires query params for doctors. *(Needs clarification based on getPatientMedications controller logic)*.

- POST /: (Protected) Creates a new medication prescription (controller logic likely requires Doctor role and patientId in body).

- PUT /:id: (Protected) Updates a specific medication prescription (controller logic requires ownership/permission checks).

- DELETE /:id: (Protected) Deletes/deactivates a specific medication prescription (controller logic requires ownership/permission checks).

5. Medication Logs (/api/medication-logs)

- POST /: (Protected, Patient) Logs a dose taken by the authenticated patient.

- GET /: (Protected, Patient) Retrieves logs for the authenticated patient, likely filtered by date query parameter.

- GET /patient/:patientId: (Protected, Doctor) Retrieves logs for a specific patient within a date range (requires query params startDate, endDate).

6. Chat (/api/chat) *(Primarily for retrieving history/conversations; real-time via Sockets)*

- GET /: (Protected) Retrieves the list of conversations for the logged-in user.

- POST /findOrCreate: (Protected) Finds or creates a 1-on-1 conversation between the logged-in user and another specified user.

- GET /:conversationId/messages: (Protected) Retrieves message history for a specific conversation, potentially with pagination.

7. Consultant/Doctor Info (/api/consultant) *(Note: Overlaps with /api/availability/doctors)*

- GET /patients: (Protected, Doctor) Retrieves the list of patients associated with the logged-in doctor.

- GET /appointments: (Protected, Doctor) Retrieves the appointments specifically for the logged-in doctor.

- GET /doctors: (Protected) Retrieves a list of registered doctors/consultants.

This RESTful API provides a structured interface for the frontend application to interact with backend resources and data, complementing the real-time communication handled by Socket.IO.

Meeting Logs:

Meeting Notes

Title of Project:	Real-time integrated Healthcare Management Platform
Student name:	Seif Mansour
Supervisor name:	Kun Wei

Date	Meeting Notes	Actions
8/10/2024	Suggesting some project ideas Discussed project ideas and chose healthcare platform.	Finalized the project topic.
22/11/2024	Received feedback on the introduction via email.	Updated the introduction section.
23/1/2025	Verified the literature review via email and received feedback.	Updated the literature review.
3/3/2025	General meeting on the purpose of the project	Received feedback on the core functions
24/3/2025	Appointment and Medication tracking features feedback	Enhanced the Medication tracking to fetch patient logs
11/4/2025	Finalizing Med Tracking for both Patient and Doctor and chatting system	Finalized the features I have along writing report.
24/4/2025	Feedback for Requirements and Design Sections with some notes on implementation	Finalized Requirements and Design and began writing implementation part