# BARANGAYCARE

BS Computer Science - BCS3209

J4A

**Submitted By:**

Larie B. Amimirog

Agatha Wendie V. Floreta

Al Jorome A. Gonzaga

Mark Anthony M. Hernandez

**Submitted to:**

Prof. Grachel Ching

November 26, 2025

**Table 1. Revision Table**

| Version | Date | Description of Change | Author |
|---|---|---|---|
| 1.0 | 2025-11-22 | Initial documentation creation | TeamBA |
| 1.1 | 2025-11-25 | Added implementation details and testing sections | TeamBA |

**Executive Summary**

BarangayCare is a comprehensive full-stack healthcare management system designed to address critical gaps in accessibility and efficiency within barangay (community-level) health services in the Philippines. The platform provides a centralized digital solution that connects patients, healthcare providers, and barangay administrators which ensures smoother coordination of consultations, medicine distribution, and emergency response. The project aims to provide a more structured and reliable approach to delivering essential community health services.

The primary users include community residents seeking healthcare services, barangay health center administrators managing appointments and medicine inventory, and healthcare providers conducting consultations. Each group of users has distinct needs that the system addresses through proper identification, role-based access, and a unified flow of information. The system serves as a bridge between community members and local healthcare resources, making healthcare more accessible and efficient.

Key features of BarangayCare include digital health profiles, medicine inventory, appointment booking, and emergency hotline access. These features are built to minimize manual work, centralize data, and provide faster service delivery. By integrating these functionalities, the application acts as an all-in-one platform for managing barangay-level healthcare efficiently.

The system demonstrates advanced programming concepts including control flow logic for appointment validation and medicine request processing, modular subprograms for reusable business logic, concurrency handling through atomic database operations and request queuing, and abstraction through service layers and API design. It applies these concepts across all core workflows to ensure consistency, reliability, and maintainability in both patient-facing and administrator-facing operations. Built using modern web technologies including Flutter for cross-platform mobile development, Node.js with the Hono.js framework for backend services, MongoDB for flexible data storage, and Firebase for authentication and real-time capabilities, BarangayCare represents a production-ready solution that can scale to serve entire barangay communities.

## System Overview

### Background of the Project Idea

The BarangayCare project was conceived to address the healthcare accessibility challenges faced by barangay communities in the Philippines. Traditional healthcare systems often require patients to physically visit health centers for appointments, medicine requests, and information, creating barriers for those with limited mobility, transportation, or time constraints. Additionally, barangay health centers struggle with manual appointment scheduling, inventory management, and emergency coordination.

The project idea emerged from recognizing that mobile technology could bridge these gaps, enabling patients to access healthcare services remotely while providing administrators with efficient tools for managing resources. The COVID-19 pandemic further highlighted the need for digital healthcare solutions that minimize physical contact while maintaining service quality.

**Problem Addressed by the System**

BarangayCare addresses several critical problems:

**1. Appointment Scheduling Inefficiency**: Manual appointment booking leads to double bookings, scheduling conflicts, and poor resource utilization. The system provides real-time availability checking and automated scheduling.

**2. Medicine Inventory Management**: Manual tracking of medicine stock leads to stockouts, overselling, and prescription validation issues. The system implements atomic stock updates and prescription validation to prevent these problems.

**3. Emergency Response Coordination**: Lack of centralized emergency contact information and location-based services delays emergency response. The system provides quick access to emergency contacts and nearest hospital finder.

**4. Health Information Access**: Patients lack easy access to health information, symptom checking, and medicine information. The AI chatbot provides 24/7 health information assistance.

**5. Prescription Management**: Paper-based prescriptions are easily lost, difficult to validate, and hard to track. The system digitizes prescriptions with upload, validation, and tracking capabilities.

**6. Health Records Fragmentation**: Patient health records are scattered across different systems or paper files, making it difficult to track health trends and history. The system centralizes health records with analytics capabilities.

**Objectives of the Developed Solution**

The primary objectives of BarangayCare are:

**1. Improve Healthcare Accessibility**: Enable patients to book appointments, request medicines, and access health information from their mobile devices without visiting the health center.

**2. Ensure Data Integrity**: Implement concurrency controls to prevent double bookings, overselling, and data corruption through atomic database operations.

**3. Streamline Administrative Processes**: Provide administrators with efficient tools for managing appointments, medicine inventory, prescription approvals, and patient records.

**4. Enhance Emergency Response**: Facilitate quick access to emergency contacts and location-based hospital finder for faster emergency response.

**5. Provide Health Information**: Offer AI-powered chatbot for symptom checking, medicine information, and health guidance in multiple languages.

**6. Demonstrate Programming Concepts**: Showcase control flow, subprograms, modularity, abstraction, and concurrency handling in a real-world application context.

**Table 2. Development Environment and Tools Used**

| Category | Tools / Technologies | Description / Purpose |
|---|---|---|
| **Programming Languages** | JavaScript (Node.js) | Backend development |
| | Dart | Flutter mobile frontend development |
| **Frameworks & Libraries** | Hono.js 4.0.0 | Lightweight Node.js web framework |
| | Flutter 3.0+ | Cross-platform mobile application framework |
| | Provider 6.1.0 | State management for Flutter |
| | http package 1.1.0 | Flutter HTTP client |
| | Native Fetch API | Backend HTTP requests |
| **Database & Storage Authentication & Security** | MongoDB Atlas | Cloud-hosted NoSQL database |
| | MongoDB Node.js Driver 6.3.0 | Database driver for Node.js |
| | Firebase Authentication 5.3.1 | User authentication (Flutter) |
| | Firebase Admin SDK 12.0.0 | Admin-level authentication & token verification (Node.js) |

| | | |
|---|---|---|
| **AI / ML & NLP** | Google Generative AI (Gemini 2.5 Flash) | Health chatbot and symptom checking |
| | Compromise.js 14.14.4 | Natural language processing |
| **File Handling & Reporting** | Multer 2.0.2 | File upload middleware (prescriptions) |
| | PDFKit 0.15.2 | PDF generation for health reports |
| **Geolocation & Mapping** | Geolib 3.3.4 | Backend geolocation utilities |
| | Geolocator 10.1.0 | Flutter geolocation API |
| | Google Maps Flutter 2.5.0 | Map and navigation integration |
| **Data Visualization** | fl_chart 0.69.0 | Charts for analytics and health records |
| **Scheduling** | node-cron 4.2.1 | Automated backend tasks |
| **Development Tools** | Visual Studio Code | Primary IDE |
| | Git + GitHub Classroom | Version control and collaboration |
| | npm, pub | Package managers (Node.js and Dart) |
| | Postman, curl | API testing tools |
| | Android Studio, Xcode | Mobile testing and build tools |
| **Deployment** | Node.js Server | Backend deployment (local/network) |
| | Android APK, iOS build, Web build | Frontend deployment outputs |
| | MongoDB Atlas | Cloud-hosted database |
| **Operating Systems** | macOS (darwin 23.6.0), Windows, Linux | Development environments |
| | Android, iOS, Web Browsers | Target platforms |

**Software Design & Architecture**

**Diagram of Program Structure or Module Relationships**

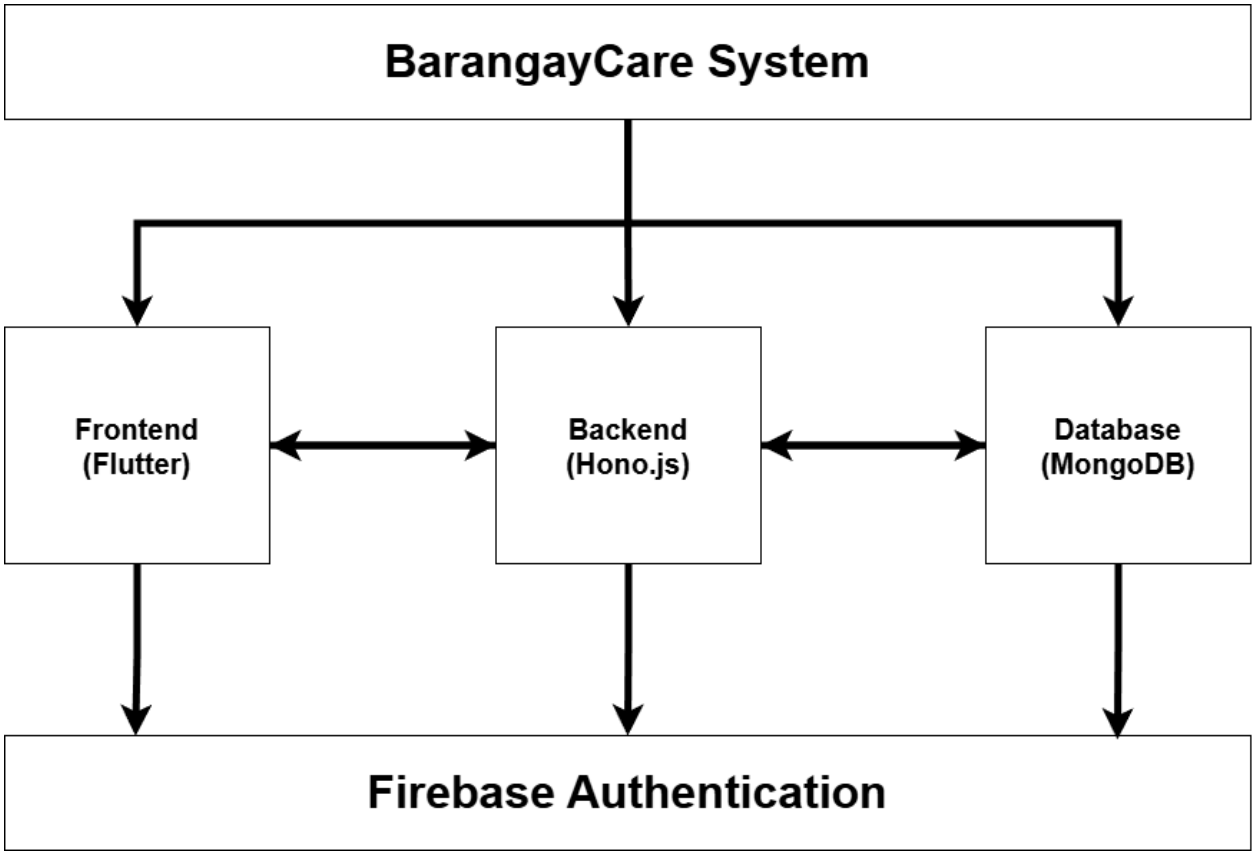



*Figure 1. Barangay Care System Architecture*

**Frontend Architecture:**

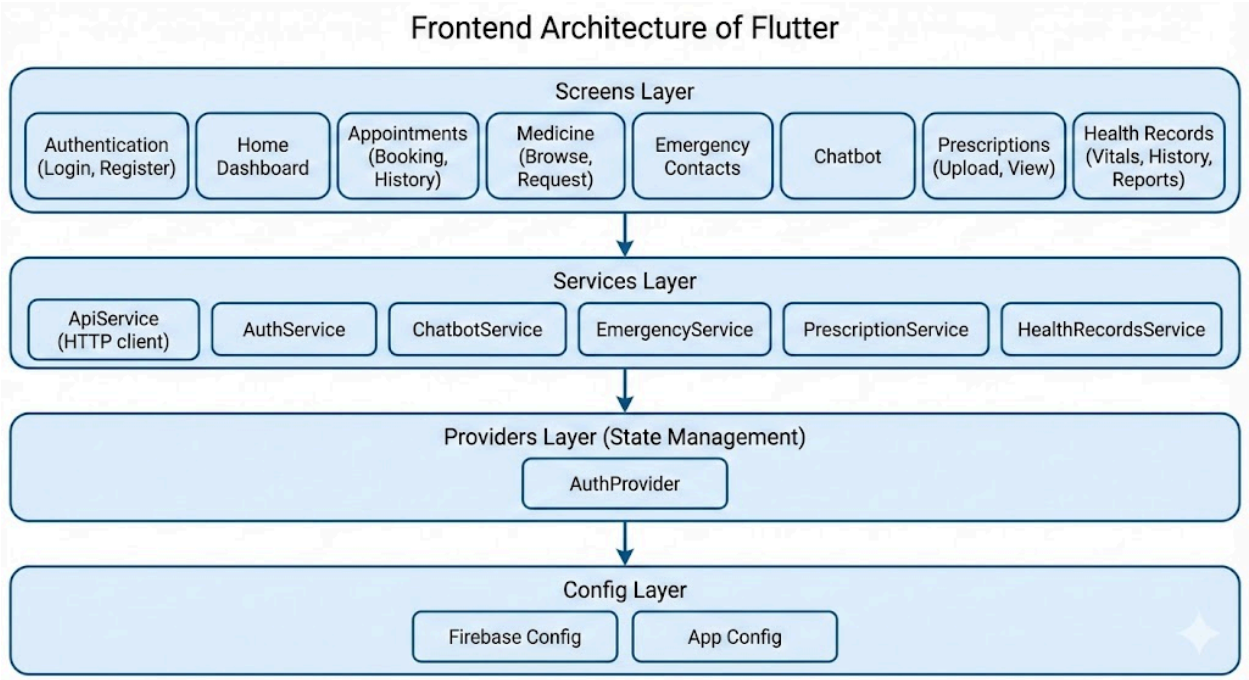



*Figure 2. Frontend Architecture of Flutter*

**Backend Architecture:**

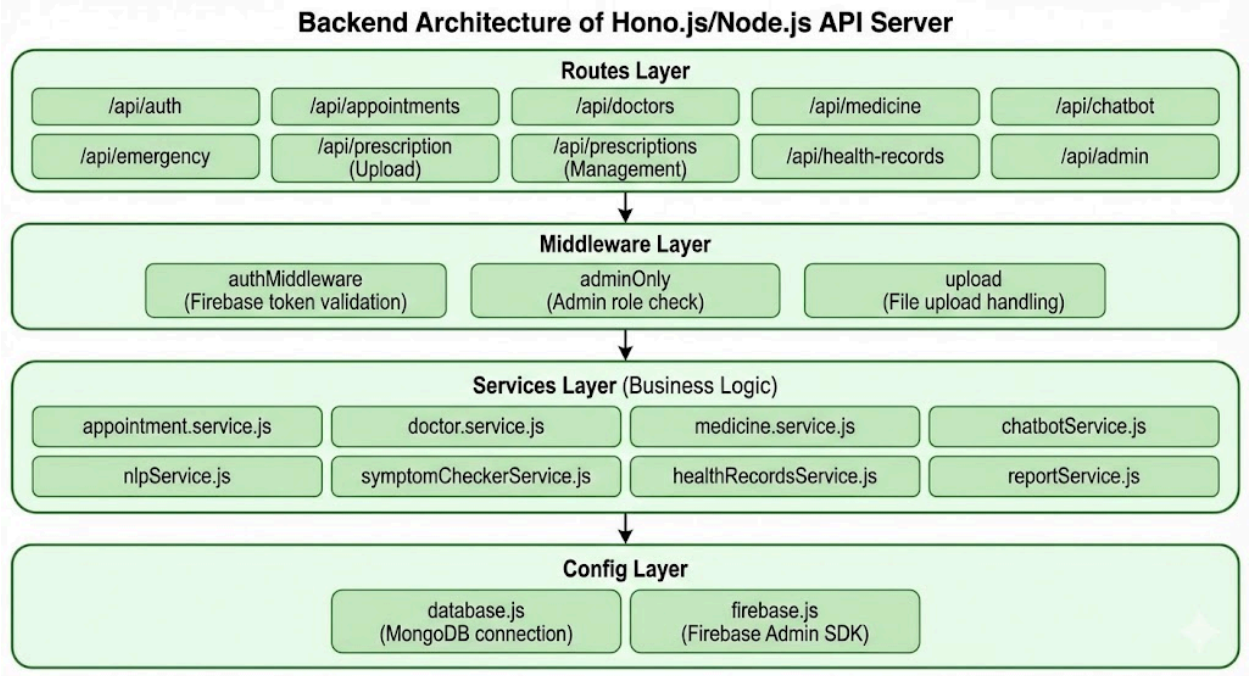



*Figure 3. Backend Architecture of Hono.js/Node.js*

**Database Schema:**

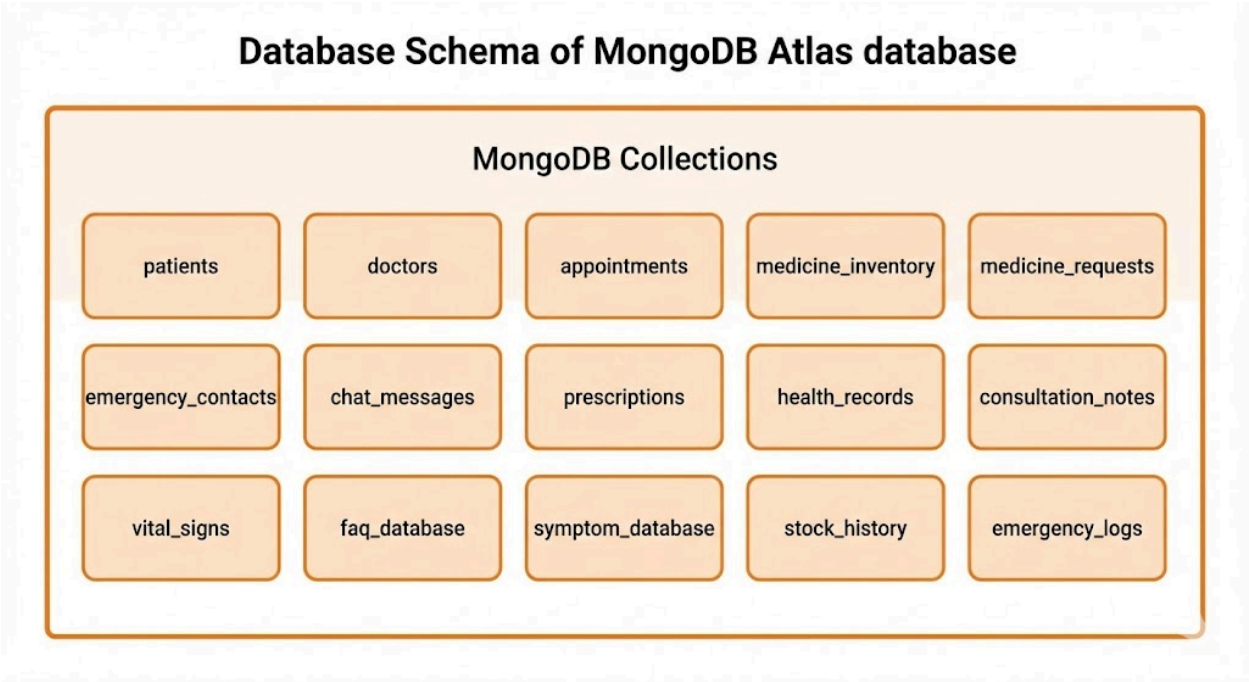



*Figure 4. Database Schema of MongoDB Atlas Database*

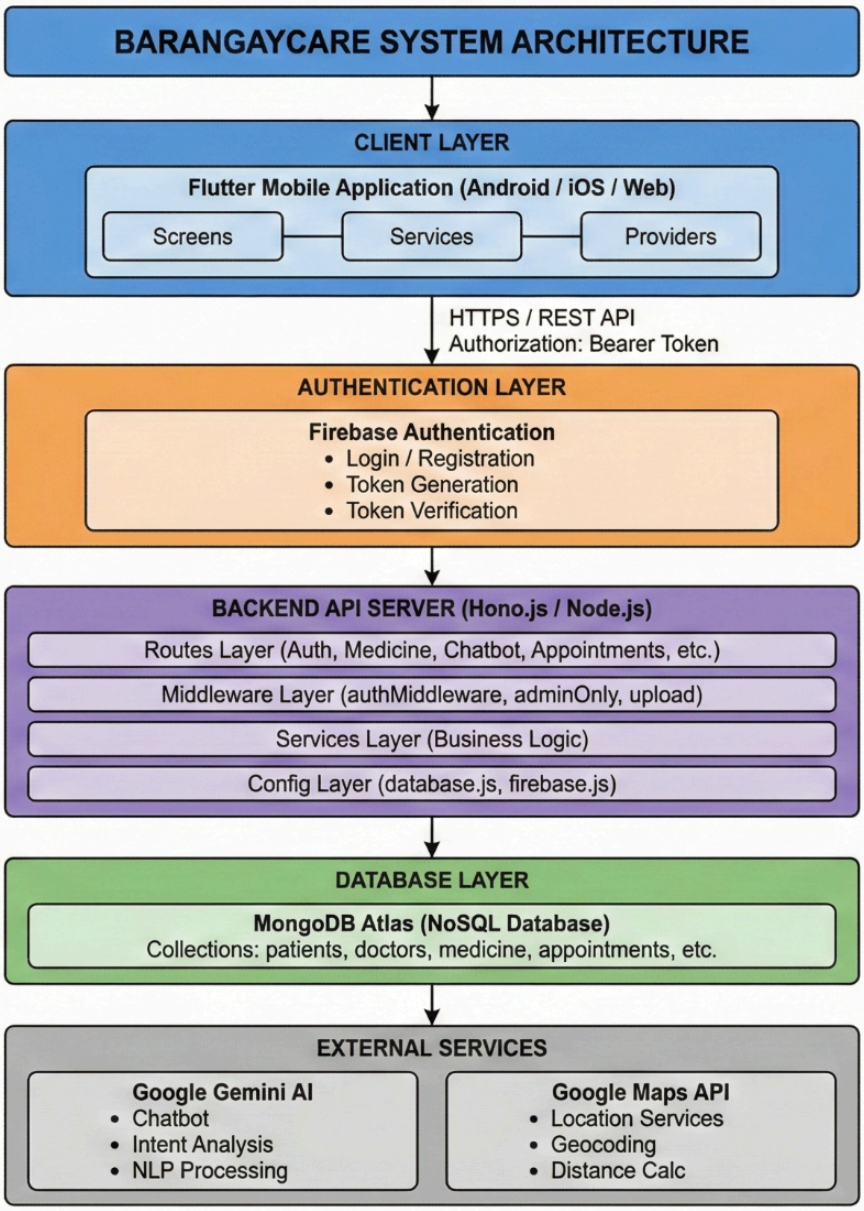



*Figure 5. System Architecture Diagram*

The BarangayCare system features a Flutter frontend that communicates with a Hono.js backend via REST APIs. User authentication is handled by Firebase, with tokens verified by the backend. The backend follows a layered architecture consisting of routes, middleware, services, and configuration, while MongoDB stores all application data such as patients, appointments, and medicines. External services, including Gemini AI for the chatbot and Google Maps for location functionality, are integrated to enhance user experience.

**Explanation of Major Functions/Subprograms**

**1. Authentication Module**

The Authentication Module manages user access and registration for the BarangayCare system. It ensures secure, role-based access and separates registration logic from authentication for better maintainability.

**Functions:**

- `authenticateUser(firebaseToken)`
  - **Purpose:** Validates Firebase authentication token and extracts user information.
  - **Location:** `backend/src/middleware/auth.middleware.js`
  - **Control Flow:** Checks token validity → Extracts user UID → Attaches user to request context.
  - **Modularity:** Reusable middleware applied to protected routes.
- `registerPatient(firebaseUid, profileData)`
  - **Purpose:** Creates a patient profile in the database after Firebase registration.
  - **Location:** `backend/src/routes/auth.route.js`
  - **Control Flow:** Validates input → Checks if patient exists → Creates new patient record.
  - **Modularity:** Separated from authentication logic for modularity and maintainability.

**2. Appointment Booking Module**

The Appointment Booking Module manages the scheduling of doctor consultations for patients. It ensures that appointments are booked only if required fields are complete, the patient exists, and the chosen time slot is available. The module also prevents double bookings by enforcing concurrency checks at the database level.

**Functions:**

- `bookAppointment(firebaseUid, data)`

- **Purpose:** Books a doctor consultation appointment with real-time availability checking.
- **Location:** `backend/src/services/appointment.service.js`

**Control Flow:**

```
IF missing required fields THEN throw error

IF patient not found THEN throw error

availability = checkAvailability(doctor_id, date)

IF doctor not available THEN throw error

IF time slot not in available slots THEN throw error

IF appointment already exists THEN throw error (prevent double booking)

Create appointment record

RETURN success
```

- **Concurrency Handling:** Database-level uniqueness check prevents double booking.

- **`checkAvailability(doctorId, dateString)`**
  - **Purpose:** Calculates available time slots for a doctor on a specific date.
  - **Location:** `backend/src/services/doctor.service.js`
  - **Modularity:** Reusable function for booking and availability checking endpoints.
  - **Control Flow:** Retrieves doctor schedule → Gets booked appointments → Generates available slots.

**3. Medicine Request Module**

The Medicine Request Module handles the creation and approval of medicine requests, ensuring prescription validation, stock availability, and data integrity. When a patient submits a medicine request, the system validates the input, checks that the patient and medicine exist, verifies prescription requirements if applicable, and confirms sufficient stock before creating a pending request. By separating validation logic from request creation, the module maintains modularity and simplifies future maintenance.

**Functions:**

- **`requestMedicine(firebaseUid, data)`**
    - **Purpose:** Creates a medicine request with prescription validation.
    - **Location:** `backend/src/services/medicine.service.js`

**Control Flow:**

```
IF invalid input THEN throw error

IF patient not found THEN throw error

IF medicine not found THEN throw error

IF medicine requires prescription THEN

    IF prescription_id provided THEN

        Validate prescription exists and is active

        IF prescription expired THEN throw error

        IF medicine not in prescription THEN throw error

           IF quantity exceeds remaining prescribed amount THEN throw
error

    END IF

END IF

IF insufficient stock THEN throw error

Create medicine request (pending status)

RETURN success
```

    - **Modularity:** Validation logic is separated from request creation for easier maintenance.

- **`approveMedicineRequest(requestId, adminUid)`**
    - **Purpose:** Approves a medicine request and atomically deducts stock.
    - **Location:** `backend/src/routes/admin.js`

- **Concurrency Handling:** Uses an atomic MongoDB update with a guard condition:

```
updateOne(
  { _id: medicineId, stock_qty: { $gte: quantity } },
  { $inc: { stock_qty: -quantity } }
)
```

This prevents race conditions that could result in negative stock levels.

**4. AI Chatbot Module**

The AI Chatbot Module provides intelligent, real-time assistance to users, offering symptom checking, medicine information, appointment guidance, and emergency advice. It processes user input, classifies intent, routes messages to the appropriate handler, and generates responses. To maintain performance and reliability, the module uses a request queuing system that limits concurrent requests, preventing server overload.

**Functions:**

- **`handleChatMessage(payload)`**
  - **Purpose:** Processes user chat messages and generates intelligent responses.
  - **Location:** `backend/src/services/chatbotService.js`
  - **Control Flow:** Classifies intent → Routes to the appropriate handler → Generates response.
  - **Concurrency Handling:** Request queuing system limits concurrent requests.
- **`classifyIntent(text)`**
  - **Purpose:** Determines user intent from natural language input.
  - **Location:** `backend/src/services/chatbotService.js`

**Control Flow:**

```
normalized = text.toLowerCase()

IF contains symptom keywords THEN RETURN 'symptom_check'
```

```
ELSE IF contains medicine keywords THEN RETURN 'medicine_info'
ELSE IF contains appointment keywords THEN RETURN 'book_appointment'
ELSE IF contains emergency keywords THEN RETURN 'emergency'
ELSE RETURN 'general'
END IF
```

    - **Modularity:** Intent classification is separated from response generation for clarity and reuse.
- `enqueueChatRequest(userId, payload)`
    - **Purpose:** Queues chat requests to limit concurrent processing.
    - **Concurrency Model:** Implements a request queue with a maximum concurrent request limit.
    - **Prevents:** Server overload from too many simultaneous chatbot requests.

**5. Emergency Contacts Module**

The Emergency Contacts Module provides users with quick access to nearby hospitals and emergency services. It calculates distances from the user's current location, filters facilities within a specified radius, and sorts them by proximity to ensure timely response. This module leverages the geolib library for accurate and efficient distance calculations.

**Functions:**

- `getNearestHospitals(latitude, longitude, radius)`
    - **Purpose:** Finds the nearest hospitals within a specified radius using geolocation.
    - **Location:** `backend/src/routes/emergency.route.js`
    - **Control Flow:** Calculate distances → Filter by radius → Sort by distance.
    - **Modularity:** Uses the geolib library for reusable and precise distance calculations.

**6. Prescription Upload Module**

The Prescription Upload Module allows users to submit prescription images, which are securely linked to their corresponding medicine requests. The system validates the uploaded file, saves it to storage, and updates the related medicine request record. To maintain performance, file uploads are processed asynchronously, ensuring smooth operation even with multiple simultaneous submissions.

**Functions:**

- `uploadPrescription(file, requestId)`
    - **Purpose:** Handles prescription image upload and associates it with the corresponding medicine request.
    - **Location:** `backend/src/routes/prescription.route.js`
    - **Control Flow:** Validate file → Save to storage → Update request record.
    - **Concurrency:** File uploads are processed asynchronously to handle multiple submissions efficiently.

**7. Health Records Module**

The Health Records Module centralizes patient medical information, providing a complete view of consultations, vitals, prescriptions, and other relevant data. It allows healthcare providers and patients to access comprehensive records while supporting analytics and reporting. Report generation is separated from data retrieval to maintain modularity and facilitate maintenance.

**Functions:**

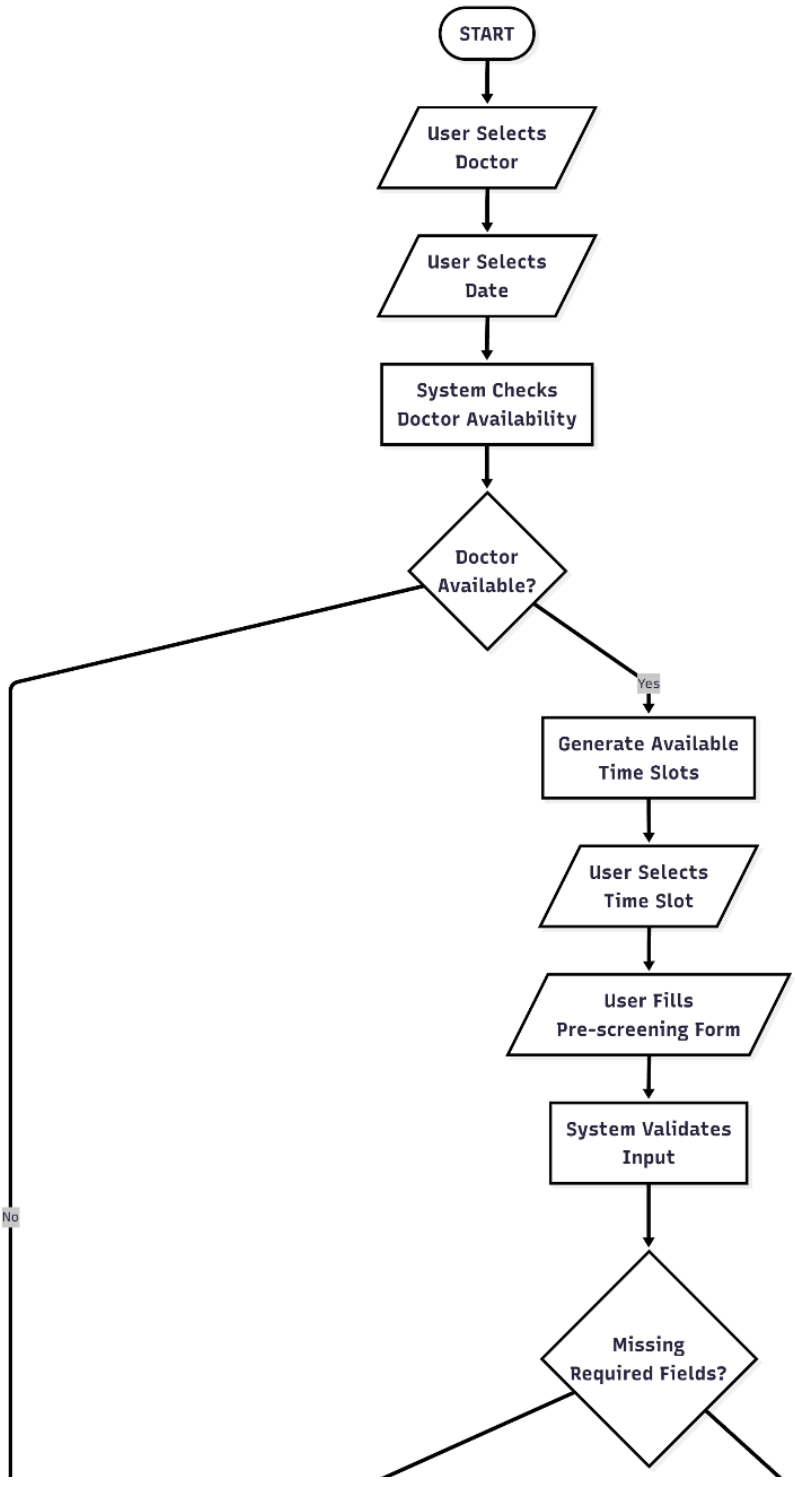
- `getHealthRecords(firebaseUid)`
    - **Purpose:** Retrieves comprehensive health records for a patient.
    - **Location:** `backend/src/services/healthRecordsService.js`
    - **Modularity:** Aggregates data from multiple collections including vitals, consultations, and prescriptions.
- `generateHealthReport(patientId)`
    - **Purpose:** Generates a PDF health report with charts and analytics.

- **Location:** `backend/src/services/reportService.js`
- **Modularity:** Separates report generation from data retrieval for clarity and maintainability.

**Flowchart or Pseudocode of Main Processes**

**Process 1: Appointment Booking Flow**

**Appointment Booking Flowchart**



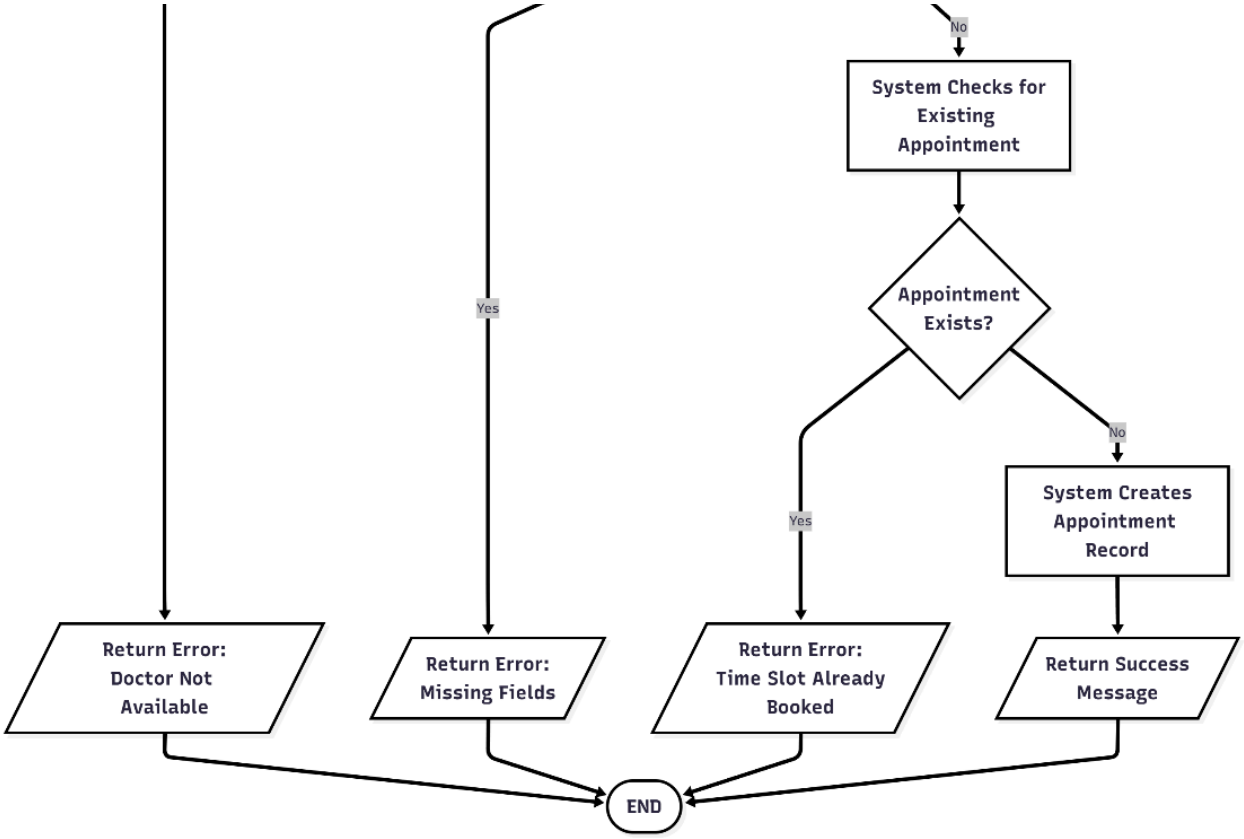
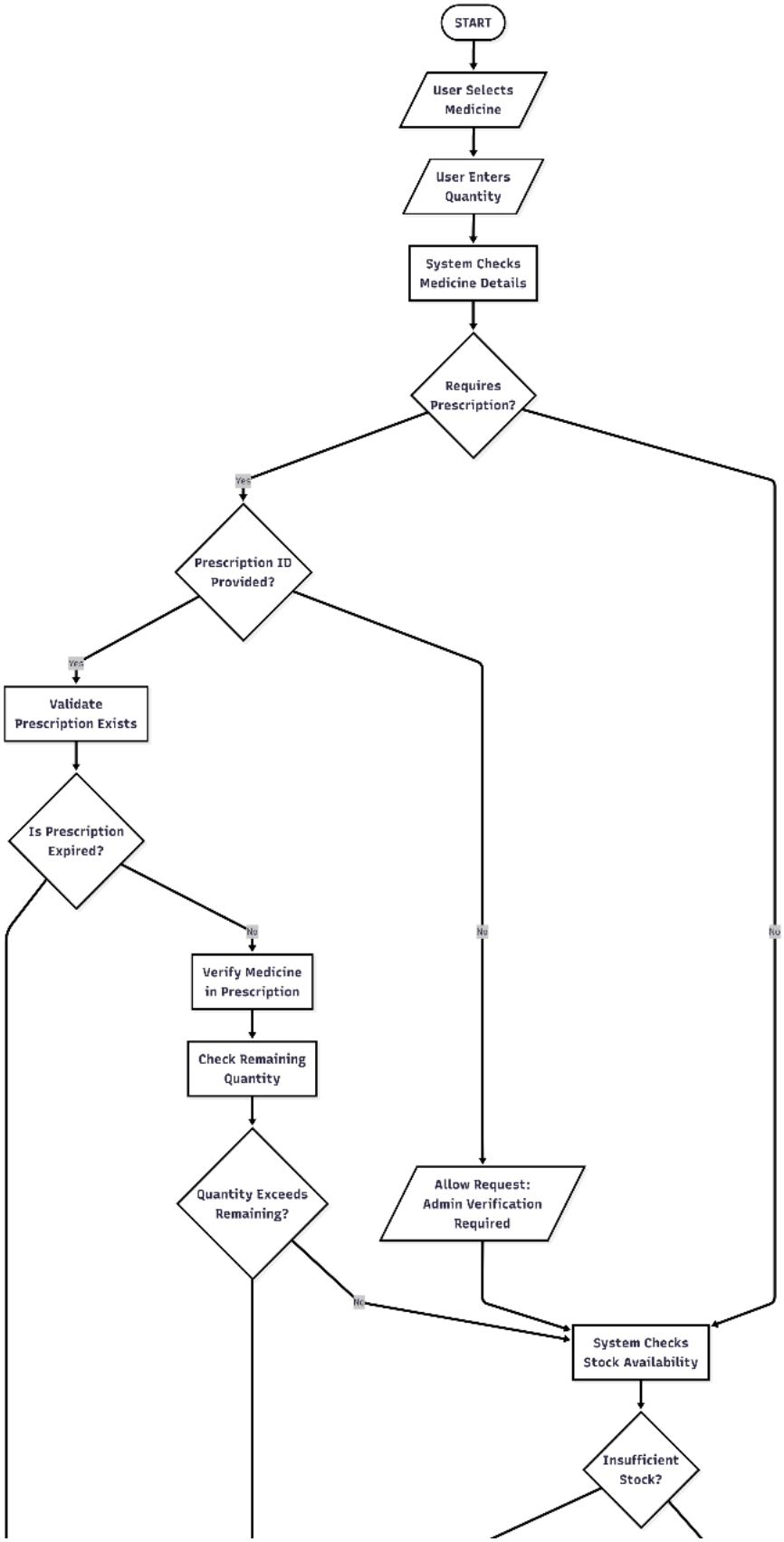



*Figure 6. Appointment Booking FlowChart*

## Process 2: Medicine Request with Stock Deduction Flow

## Medicine Request Flowchart



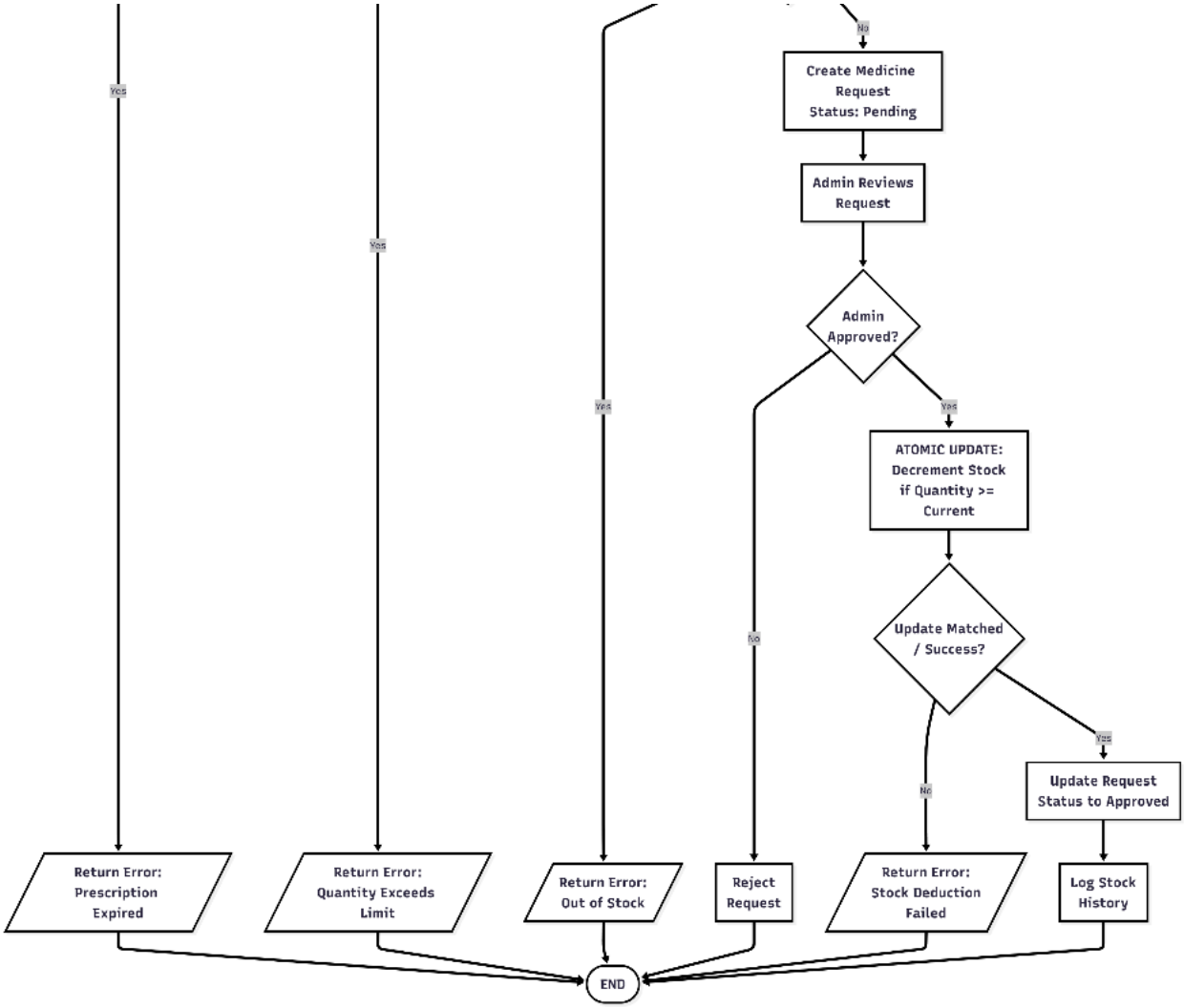
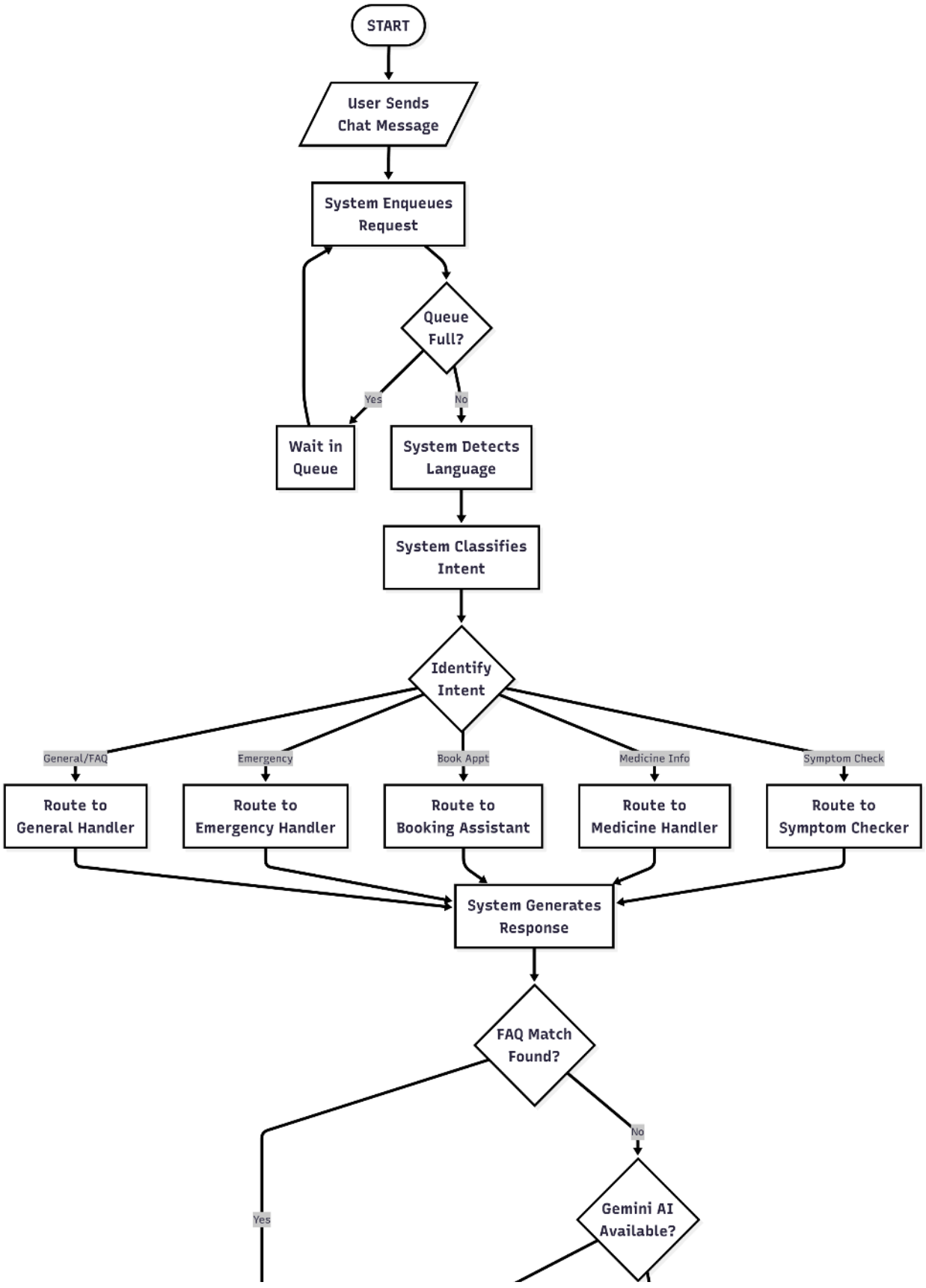



*Figure 7. Medicine Request Flowchart*

## Process 3: Chatbot Request Processing Flow

## Chatbot Processing Flowchart



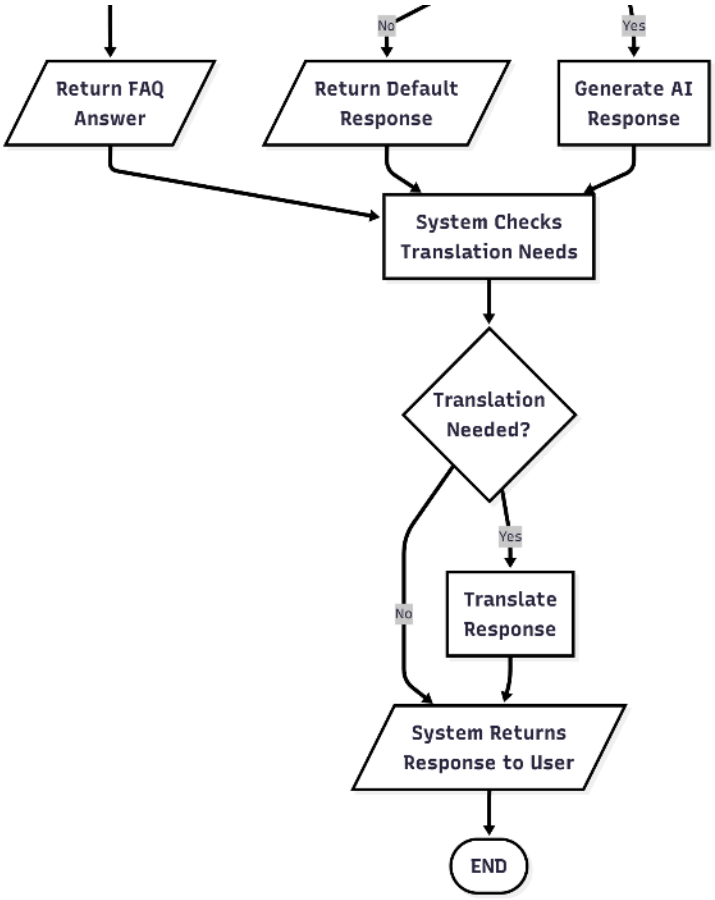



*Figure 8. Chatbot Processing Flowchart*

**Concurrency or Multi-threading Model Explanation**

BarangayCare implements several concurrency models to handle multiple simultaneous operations safely and efficiently:

**1. Atomic Database Operations (MongoDB)**

**Purpose:** Prevent race conditions in stock updates and appointment handling
**Location:** `backend/src/routes/admin.js`

**Model Summary**

BarangayCare uses MongoDB's atomic update operations to ensure that checking stock and updating quantities happen in a single, indivisible process. This eliminates timing gaps where simultaneous requests could cause incorrect inventory values.

### Code Snippet

```
// Atomic stock update prevents race conditions
const stockUpdate = await collections.medicineInventory().updateOne(
  {
    _id: new ObjectId(medicineId),
    stock_qty: { $gte: requestQuantity }  // Guard condition
  },
  {
    $inc: { stock_qty: -requestQuantity }  // Atomic decrement
  }
);
// Only one request can succeed when stock is low
if (stockUpdate.matchedCount === 0) {
  throw new Error('Insufficient stock');
}
```

### How It Works

- Condition (`stock_qty >= quantity`) and update (`$inc`) are executed atomically.
- When two admins approve simultaneously:
    - First request succeeds (5 → 2).
    - Second request fails due to updated stock (now 2).
- Prevents overselling and negative stock values.

**2. Request Queuing System (Chatbot)**

**Purpose:** Limit concurrent chatbot requests to prevent server overload.
**Location:** `backend/src/services/chatbotService.js`

**Model Summary**

The chatbot uses a queue-based concurrency model to manage incoming requests efficiently and safely. Each user has their own request queue, ensuring fair processing while preventing any single user from overloading the system. A global concurrency cap ensures that no more than four chatbot processes run at the same time, protecting server performance.

**Code Snippet**

```
const MAX_CONCURRENT_REQUESTS = 4;
let activeRequests = 0;
const chatQueues = new Map();

export function enqueueChatRequest(userId, payload) {
  if (!chatQueues.has(userId)) {
    chatQueues.set(userId, []);
  }

  return new Promise((resolve, reject) => {
    chatQueues.get(userId).push({ payload, resolve, reject });
    processQueue(userId);
  });
}
```

```
async function processQueue(userId) {
  const queue = chatQueues.get(userId);
  if (!queue || queue.length === 0) return;

  if (activeRequests >= MAX_CONCURRENT_REQUESTS) {
    return; // Wait for slot to open
  }

  const job = queue.shift();
  activeRequests++;

  try {
    const result = await handleChatMessage(job.payload);
    job.resolve(result);
  } catch (error) {
    job.reject(error);
  } finally {
    activeRequests--;
    processQueue(userId); // Process next in queue
  }
}
```

**How It Works**

- Maintains a **queue per user** to guarantee fair message handling.
- Enforces a **maximum of four concurrent chatbot operations** to prevent overload.
- Processes requests **sequentially per user but concurrently across users**.

- Automatically continues processing the queue as soon as a slot becomes free.

**3. Parallel Data Fetching (Promise.all)**

**Purpose:** Improve performance by retrieving multiple resources simultaneously.
**Location:** `backend/src/routes/admin.js` (Appointments list endpoint)

**Model Summary**

This module uses `Promise.all()` to run multiple asynchronous operations at the same time. Instead of waiting for each database query sequentially, related lookups—such as appointments, counts, and doctor/patient details—are fetched concurrently. This significantly reduces response time, especially when processing multiple records.

**Code Snippet**

```
// Fetch appointments and count in parallel
const [appointments, total] = await Promise.all([
  collections.appointments().find(filter).toArray(),
  collections.appointments().countDocuments(filter)
]);
// Enrich each appointment with patient/doctor data in parallel
const enrichedAppointments = await Promise.all(
  appointments.map(async (apt) => {
    const [patient, doctor] = await Promise.all([
      collections.patients().findOne({ _id: apt.patient_id }),
      collections.doctors().findOne({ _id: apt.doctor_id })
    ]);

    return {
```

```
      ...apt,
      patient_name: patient?.name || 'Unknown',
      doctor_name: doctor?.name || 'Unknown'
    };
  })
);
```

**How It Works**

- `Promise.all()` runs all asynchronous operations **concurrently**, not sequentially.
- Execution continues only after **all promises resolve**, ensuring complete data.
- Total time equals the **slowest operation**, not the sum of all operations.
- Example: For 10 items (40 ms each), total time is ~40 ms instead of 400 ms.

**4. Database-Level Uniqueness Constraints**

**Purpose:** Prevent double booking by enforcing uniqueness during appointment creation.
**Location:** `backend/src/services/appointment.service.js`

**Model Summary**

The system performs a check-then-insert operation to ensure that no two appointments occupy the same doctor–date–time combination. MongoDB handles each document insertion atomically, preventing multiple requests from inserting conflicting records simultaneously.

**Code Snippet**

```
// Check for existing appointment before creating
const existingAppointment = await collections.appointments().findOne({
  doctor_id: new ObjectId(doctor_id),
  date: date,
  time: time,
```

```
  status: { $in: ['booked', 'confirmed'] }
});

if (existingAppointment) {
  throw new Error('Time slot already booked');
}
// Create appointment (MongoDB ensures document-level atomicity)
await collections.appointments().insertOne(appointment);
```

**How It Works**

- MongoDB guarantees that each insert is **atomic**, preventing overlapping writes.
- The system checks if a booked/confirmed appointment already exists for the same doctor, date, and time.
- If found, the operation stops immediately, preventing double booking.
- Developers may add a **unique compound index** on `(doctor_id, date, time, status)` for extra protection.

### 5. Asynchronous File Upload Processing

**Purpose:** Handle file uploads without blocking other incoming API requests.
**Location:** `backend/src/routes/prescription.route.js`

**Model Summary**

BarangayCare uses Node.js' non-blocking I/O model alongside async/await to ensure that file uploads are processed asynchronously. This allows the server to continue handling other requests while files are being saved or read.

**Code Snippet**

```
// Multer middleware handles file upload asynchronously

app.post('/api/prescription/upload',     upload.single('prescription'),
async (c) => {

  const file = c.req.file;

  // File processing happens asynchronously

  // Other requests can be processed while file is being saved

});
```

**How It Works**

- Node.js processes file uploads using **asynchronous, non-blocking I/O**, so uploads never freeze the server.
- While a file is being saved to storage, the event loop continues serving other clients.
- Multiple uploads can be handled at the same time, improving system responsiveness and scalability.

**Implementation**

**Sample Input/Output Screenshots or Terminal Logs**

**1. Backend Server Startup**

**Input (Command Executed):**

```
cd backend

npm run dev
```

**Output (Terminal Log):**

```
✅ Firebase Admin initialized

✅ MongoDB connected

✅ Seeded 10 medicines

✅ Seeded 3 doctors

✅ Seeded admin account

✅ Seeded FAQ database

✅ Seeded emergency contacts

🚀 Starting server on http://0.0.0.0:3000

📱 Physical devices can connect to: http://192.168.68.100:3000

💻 Local access: http://localhost:3000

✅ Server is listening on port 3000
```

**2. Appointment Booking API Request**

**Input**: POST request to book appointment

```
curl -X POST http://localhost:3000/api/appointments/book \
  -H "Authorization: Bearer <firebase_token>" \
```

```
  -H "Content-Type: application/json" \
  -d '{
    "doctor_id": "507f1f77bcf86cd799439011",
    "date": "2025-11-25",
    "time": "09:00",
    "pre_screening": {
      "symptoms": "Fever and cough",
      "temperature": "38.5",
      "additional_notes": "Started 2 days ago"
    }
  }'
```

**Output**:


```
{
  "message": "Appointment booked successfully",
  "appointment_id": "507f191e810c19729de860ea",
  "appointment": {
    "patient_id": "507f1f77bcf86cd799439012",
    "doctor_id": "507f1f77bcf86cd799439011",
    "date": "2025-11-25",
    "time": "09:00",
    "status": "pending",
    "pre_screening": {
      "symptoms": "Fever and cough",
      "temperature": "38.5",
      "additional_notes": "Started 2 days ago"
    },
    "created_at": "2025-11-21T10:30:00.000Z",
    "updated_at": "2025-11-21T10:30:00.000Z"
  }
}
```

**3. Medicine Request with Prescription Validation**

**Input**: POST request for prescription-required medicine

```
curl -X POST http://localhost:3000/api/medicine/request \
  -H "Authorization: Bearer <firebase_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "medicine_id": "507f1f77bcf86cd799439013",
    "quantity": 2,
    "prescription_id": "507f1f77bcf86cd799439014"
  }'
```

**Output (Success)**:

```
{
  "message": "Medicine request submitted successfully. Waiting for admin approval.",
  "request_id": "507f191e810c19729de860eb",
  "medicine_name": "Amoxicillin 500mg",
  "quantity_requested": 2,
  "status": "pending"
}
```

**Output (Error - Prescription Expired)**:

```
{
  "error": "Prescription has expired"
}
```

**4. Chatbot Interaction**

**Input**: POST request to chatbot

```
curl -X POST http://localhost:3000/api/chatbot/message \
  -H "Authorization: Bearer <firebase_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "message": "I have a fever and cough",
    "language": "en"
  }'
```

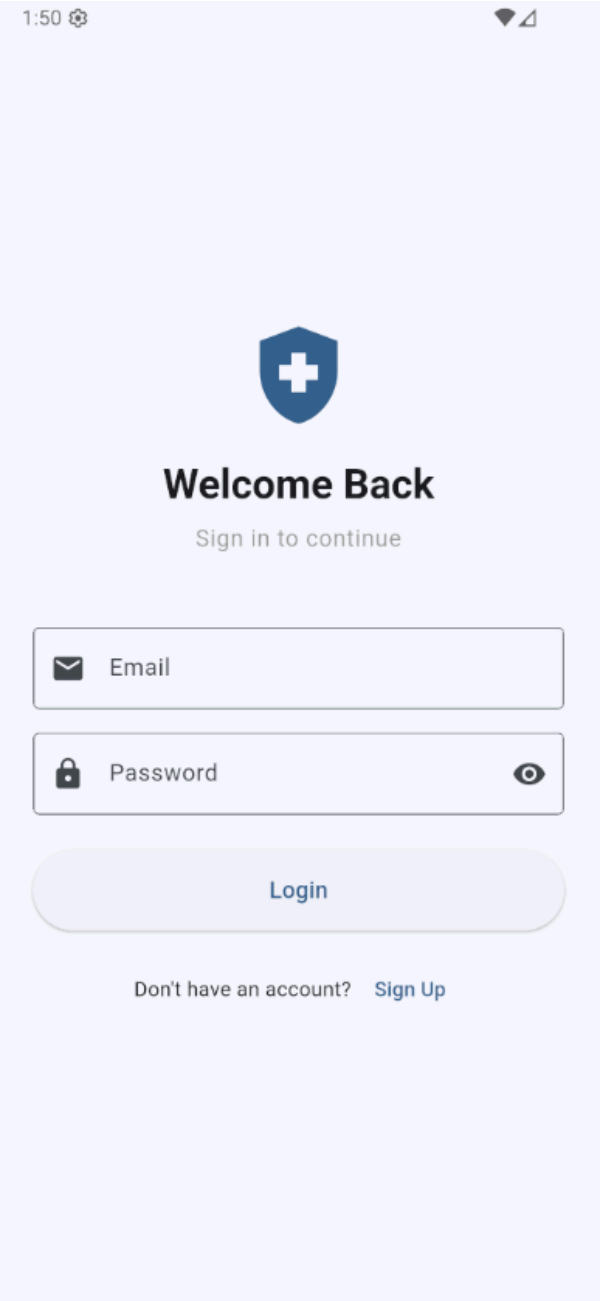
**Output**:


```
{
  "response": "Based on your symptoms (fever and cough), I recommend:\n\n1. Rest and stay hydrated\n2. Monitor your temperature\n3. Consider booking a consultation if symptoms persist\n\nWould you like me to help you book an appointment?",
  "intent": "symptom_check",
  "suggestions": [
    "Book Appointment",
    "Request Medicine",
    "View Doctors",
    "Emergency Contacts"
  ]
```

## 5. Frontend Application Screens

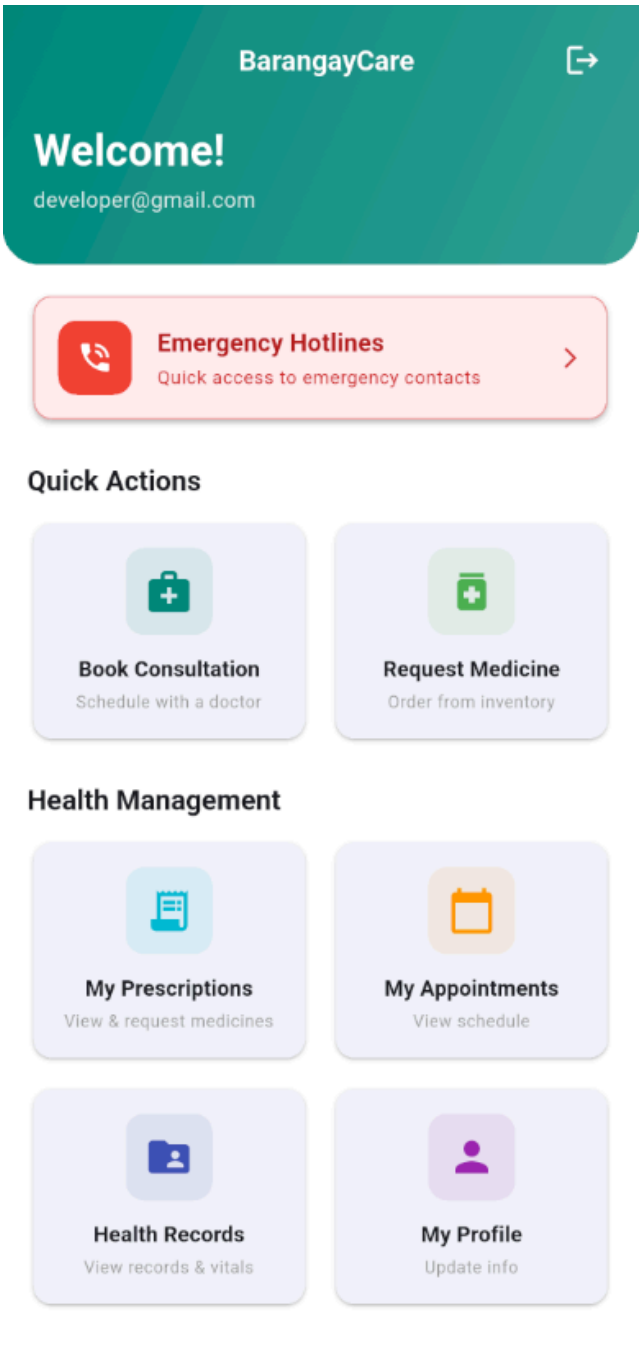
### Login Screen

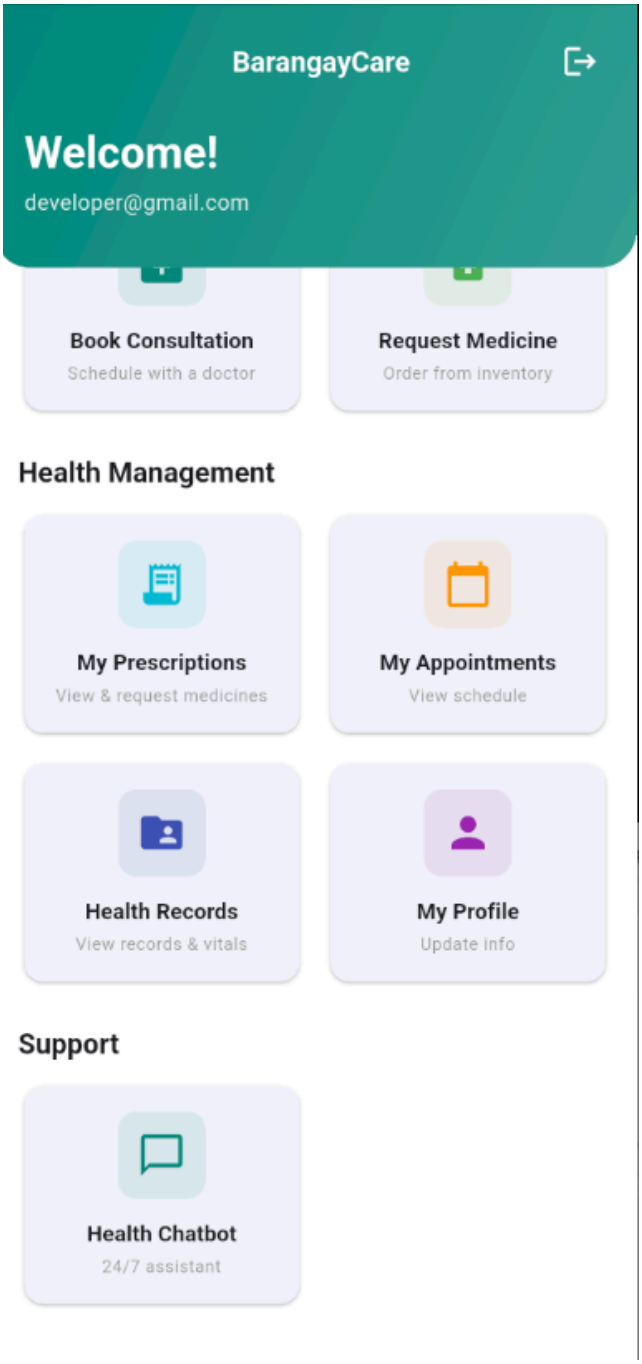
- **Feature:** Firebase Authentication
- **Description:** User login using email + password



**Home Dashboard**

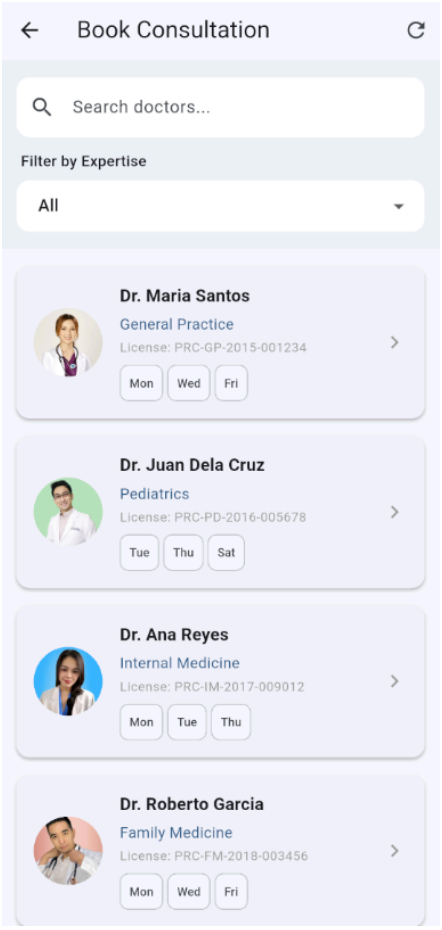
- **Feature:** Main navigation hub
- **Description:** Quick access to Appointments, Medicines, Emergency, Chatbot, etc.

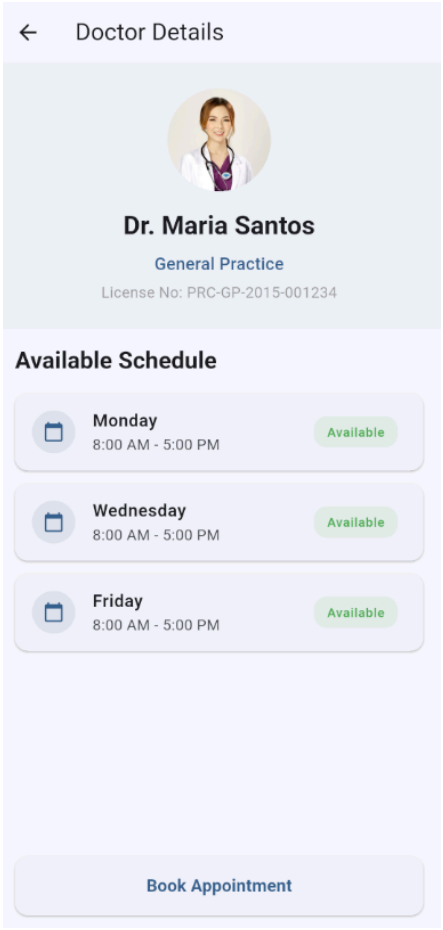


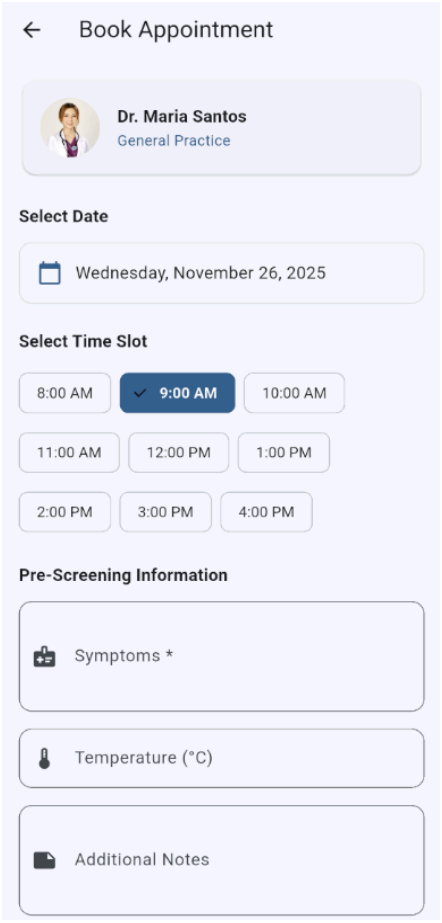



**Appointment Booking Screen**

- **Feature:** Doctor consultation booking
- **Description:** Select doctor → date → time → fill pre-screening form

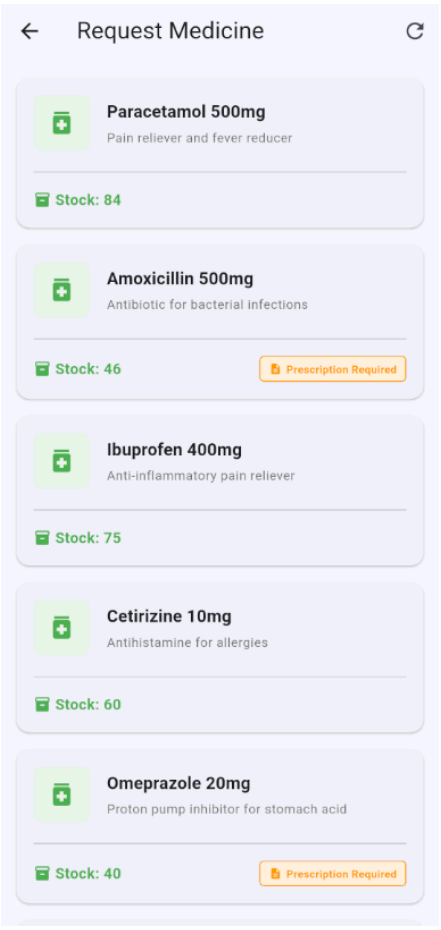


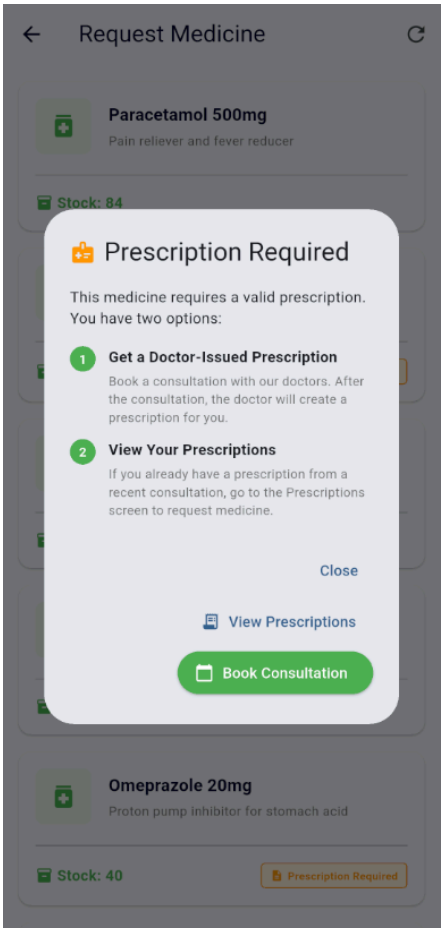



**Medicine Request Screen**

- **Feature:** Medicine listing & request
- **Description:** Browse stock, view availability, request medicine with prescription rules

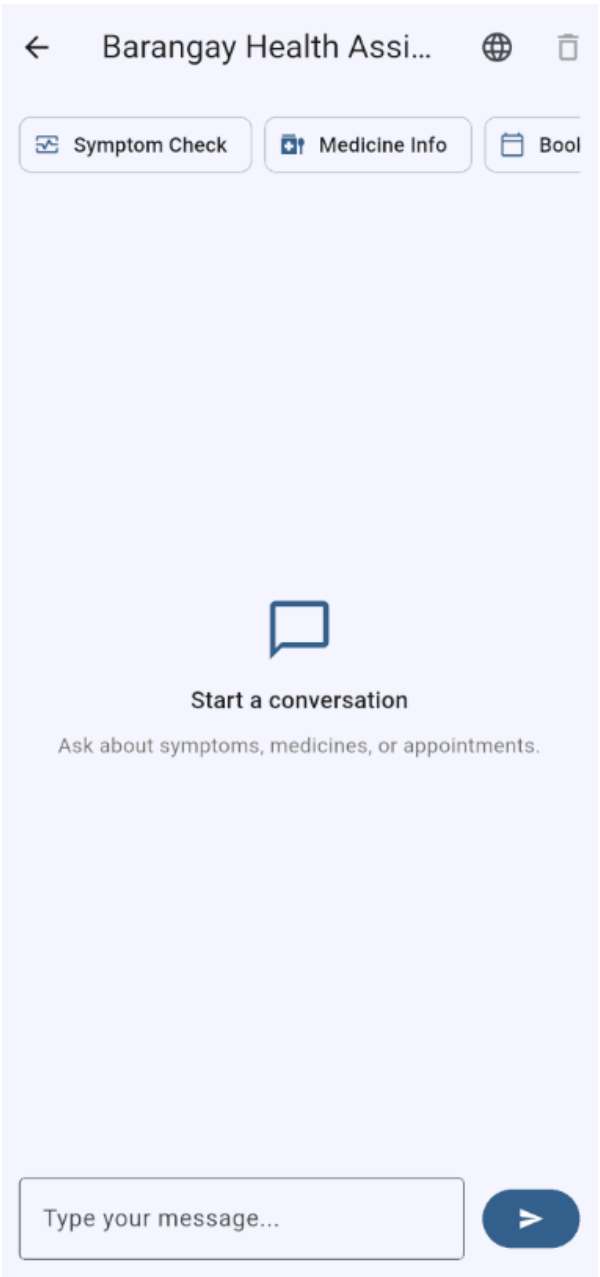


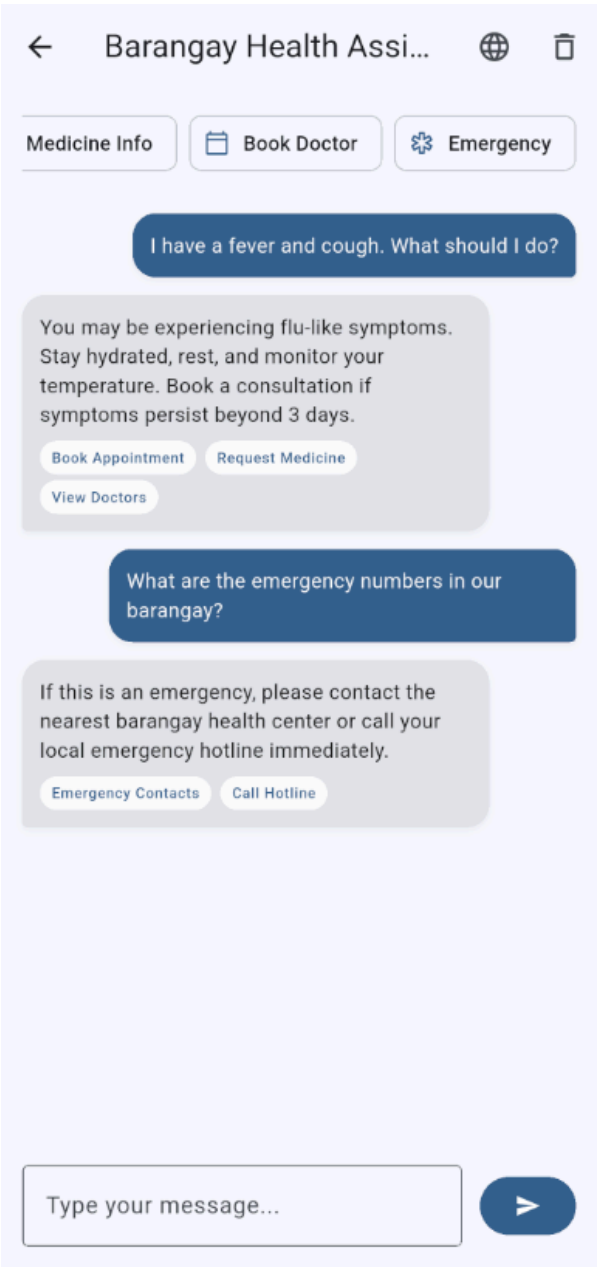




**Chatbot Screen**

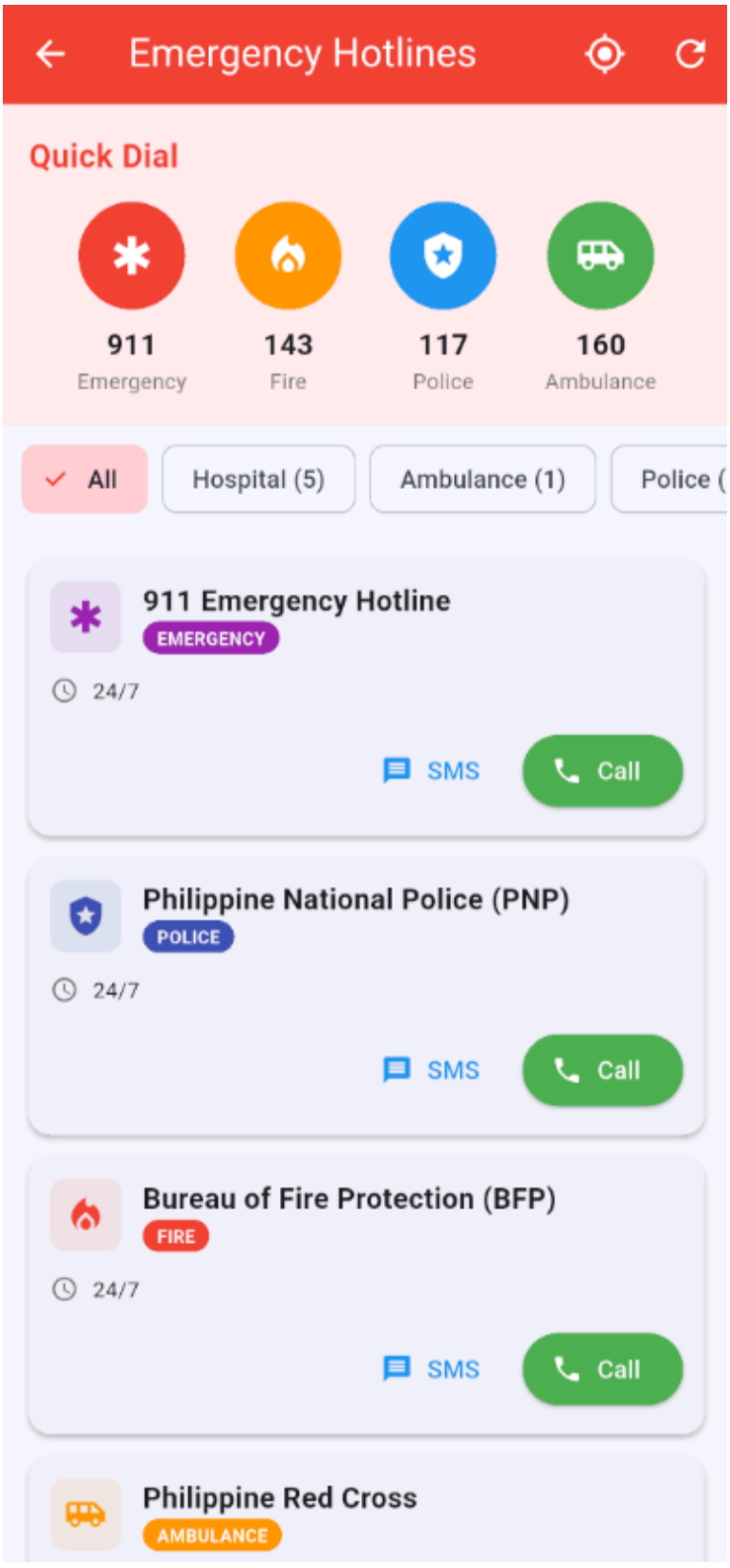
- **Feature:** AI Health Chatbot
- **Description:** Conversational interface for symptom checking & health info



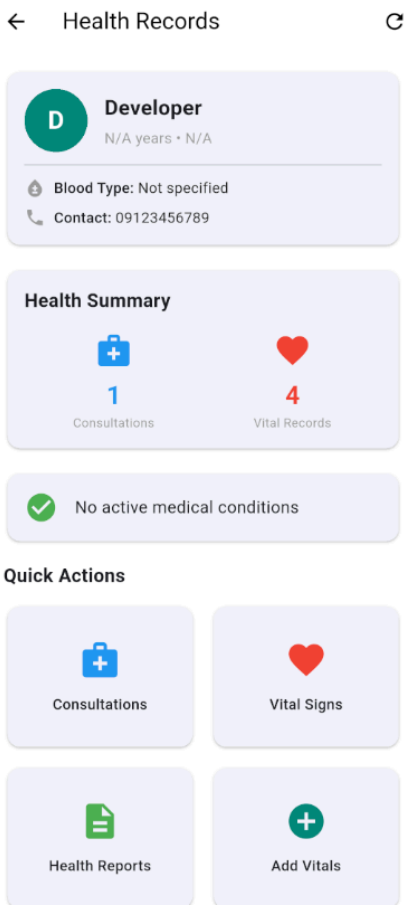
**Emergency Contacts Screen**

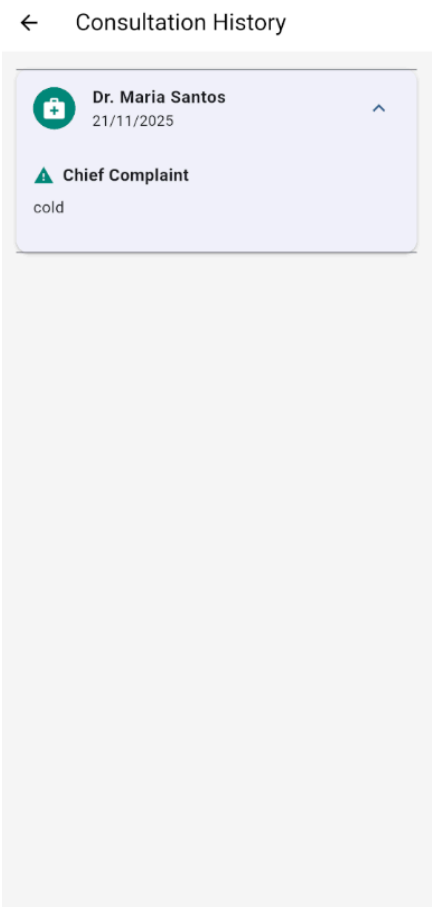
- **Feature:** Emergency contact directory
- **Description:** Quick-dial buttons, filters by category, nearest hospital locator

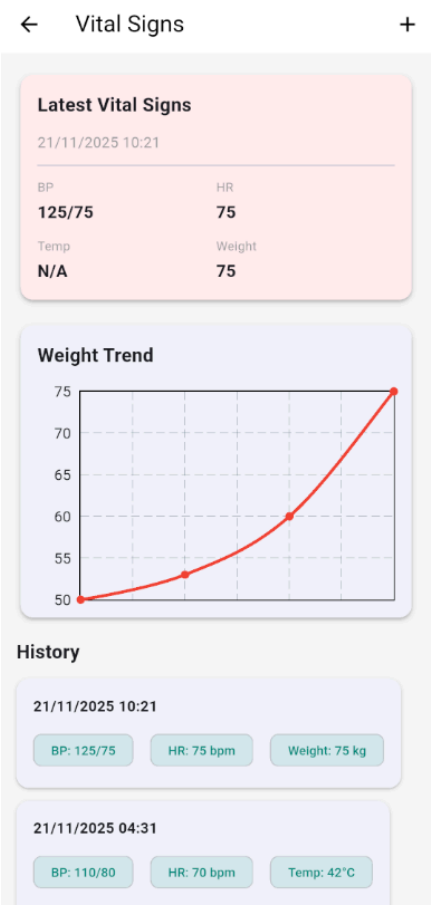


**Health Records Screen**

- **Feature:** Patient records
- **Description:** Vital signs, consultation history, and health trends visualization

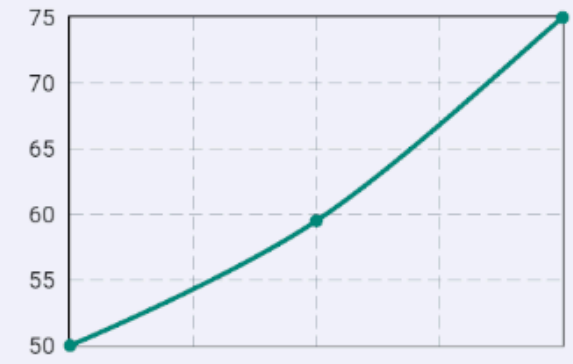


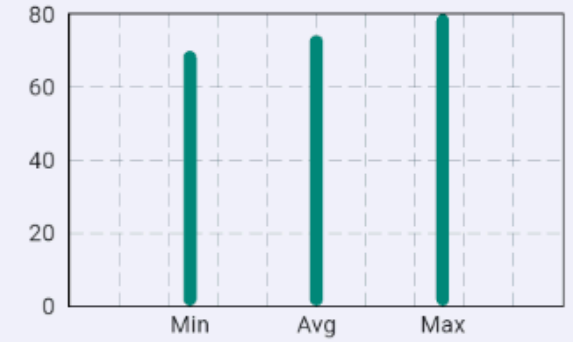
# Health Report

## Weight Trend




Average: 59.5 kg | Range: 50-75 kg

## Heart Rate Trend




Average: 74.3 bpm | Range: 70-80 bpm

## Generate PDF Report

Download a comprehensive PDF report of your health records including consultations, vital signs, and health trends.

# Add Vital Signs

Enter your vital signs measurements

## Blood Pressure

Systolic / Diastolic

Heart Rate

Temperature

Weight

Height

Oxygen Saturation

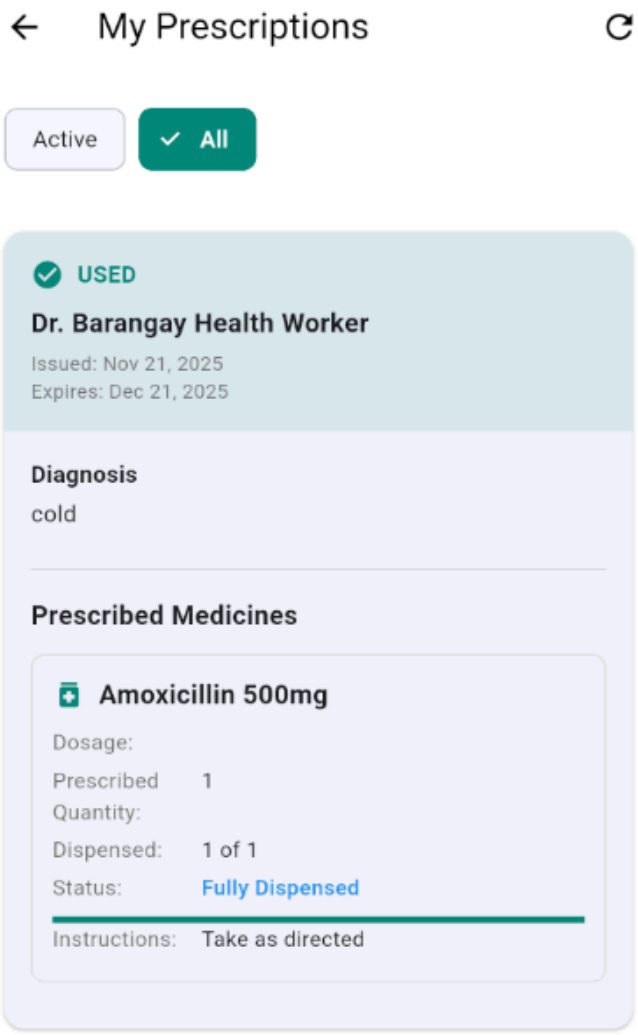
Notes (optional)

Record Vital Signs

Enter at least one vital sign measurement. All fields are optional.

**Prescription Screen**

- **Feature:** View prescriptions
- **Description:** Displays prescriptions issued during consultations.



**Highlight Sections of Code that Demonstrate:**

**Control Flow Logic**

**Example 1: Appointment Booking Validation File**:

`backend/src/services/appointment.service.js`

```
export async function bookAppointment(firebaseUid, data) {
  const { doctor_id, date, time, pre_screening } = data;


  // Control Flow: Sequential validation checks
  if (!doctor_id || !date || !time) {
    throw new Error('Missing required fields: doctor_id, date, time');
  }

  const patient = await collections.patients().findOne({ firebase_uid: firebaseUid });
  if (!patient) {
    throw new Error('Patient profile not found');
  }

  // Control Flow: Check availability before booking
  const availability = await checkAvailability(doctor_id, date);

  if (!availability.available) {
    throw new Error('Doctor not available on this day');
  }

  if (!availability.slots.includes(time)) {
    throw new Error('Time slot not available');
  }

  // Control Flow: Prevent double booking
  const existingAppointment = await collections.appointments().findOne({
    doctor_id: new ObjectId(doctor_id),
    date: date,
    time: time,
    status: { $in: ['booked', 'confirmed'] }
  });

  if (existingAppointment) {
    throw new Error('Time slot already booked');
  }

  // All validations passed - create appointment
```

```
  const appointment = {
    patient_id: patient._id,
    doctor_id: new ObjectId(doctor_id),
    date,
    time,
    status: 'pending',
    pre_screening: pre_screening || {},
    created_at: new Date(),
    updated_at: new Date(),
  };

  const result = await collections.appointments().insertOne(appointment);

  return {
    message: 'Appointment booked successfully',
    appointment_id: result.insertedId,
    appointment
  };
}
```

**Demonstrates:**

- Sequential **IF-THEN-ELSE** control flow
- Early return pattern for error handling
- Conditional logic for validation

  Nested conditions for complex validation

**Example 2: Medicine Request with Prescription Validation File**:

`backend/src/services/medicine.service.js`

```
// Control Flow: Prescription requirement checking
if (medicine.is_prescription_required) {
  if (prescription_id) {
    const prescription = await collections.prescriptions().findOne({
      _id: new ObjectId(prescription_id),
      patient_id: patient._id,
      status: 'active'
    });

    if (!prescription) {
      throw new Error('Invalid or expired prescription');
    }
```

```
    // Control Flow: Expiry checking
    if (new Date() > prescription.expiry_date) {
      await collections.prescriptions().updateOne(
        { _id: new ObjectId(prescription_id) },
        { $set: { status: 'expired', updated_at: new Date() } }
      );
      throw new Error('Prescription has expired');
    }

    // Control Flow: Medicine validation in prescription
    const prescribedMedicine = prescription.medicines.find(
      m => m.medicine_id.toString() === medicine_id
    );

    if (!prescribedMedicine) {
      throw new Error('Medicine not found in prescription');
    }

    // Control Flow: Quantity validation
    const dispensedQuantity = prescribedMedicine.dispensed_quantity || 0;
    const remainingQuantity = prescribedMedicine.quantity - dispensedQuantity;

    if (remainingQuantity <= 0) {
      throw new Error('Prescribed quantity already fully dispensed');
    }

    if (quantity > remainingQuantity) {
      throw new Error(`Requested quantity exceeds remaining prescribed amount`);
    }
  }
}
```

**Demonstrates:**

- Nested IF statements for complex conditional logic
- Multiple validation layers
- Early exit on error conditions
- Logical AND/OR conditions

**Function Modularity**

**Example 1: Service Layer Modularity File**:

`backend/src/services/doctor.service.js`

```
// Modular function: Check doctor availability
export async function checkAvailability(doctorId, dateString) {
  const doctor = await getDoctorById(doctorId);
  if (!doctor) {
    throw new Error('Doctor not found');
  }

  // Modular function call: Get doctor schedule
  const schedule = getDoctorSchedule(doctor, dateString);
  if (!schedule) {
    return { available: false, slots: [] };
  }

  // Modular function call: Get booked appointments
  const bookedAppointments = await getBookedAppointments(doctorId, dateString);
  const bookedTimes = bookedAppointments.map(apt => apt.time);

  // Modular function call: Generate time slots
  const availableSlots = generateTimeSlots(
    schedule.start,
    schedule.end,
    bookedTimes
  );

  return {
    available: availableSlots.length > 0,
    slots: availableSlots
  };
}

// Modular helper function: Generate time slots
function generateTimeSlots(startTime, endTime, bookedTimes) {
  const slots = [];
  const start = new Date(`2000-01-01T${startTime}`);
  const end = new Date(`2000-01-01T${endTime}`);

  let current = new Date(start);
  while (current < end) {
    const timeString = current.toTimeString().slice(0, 5);
```

```
    if (!bookedTimes.includes(timeString)) {
      slots.push(timeString);
    }
    current.setMinutes(current.getMinutes() + 30);
  }

  return slots;
}
```

**Demonstrates:**

- Separation of concerns (availability checking vs. slot generation)
- Reusable helper functions
- Single responsibility principle
- Function composition

**Example 2: Frontend Service Modularity File**:

`frontend/lib/services/api_service.dart`

```
// Modular API service class
class ApiService {
  static const String baseUrl = 'http://localhost:3000/api';

  // Modular function: Book appointment
  static Future<Map<String, dynamic>> bookAppointment({
    required String doctorId,
    required String date,
    required String time,
    required Map<String, dynamic> preScreening,
    required String token,
  }) async {
    final response = await http.post(
      Uri.parse('$baseUrl/appointments/book'),
      headers: {
        'Authorization': 'Bearer $token',
        'Content-Type': 'application/json',
      },
      body: jsonEncode({
        'doctor_id': doctorId,
        'date': date,
        'time': time,
        'pre_screening': preScreening,
      }),
```

```
    );

    return _handleResponse(response);
  }

  // Modular function: Request medicine
  static Future<Map<String, dynamic>> requestMedicine({
    required String medicineId,
    required int quantity,
    String? prescriptionId,
    required String token,
  }) async {
    // Similar modular structure
  }

  // Modular helper: Handle HTTP response
  static Map<String, dynamic> _handleResponse(http.Response response) {
    if (response.statusCode >= 200 && response.statusCode < 300) {
      return jsonDecode(response.body);
    } else {
      throw Exception('API Error: ${response.body}');
    }
  }
}
```

**Demonstrates:**

- Class-based modularity
- Function separation by feature
- Shared helper functions
- Encapsulation of HTTP logic

**Concurrency**

**Example 1: Atomic Stock Update File**:

`backend/src/routes/admin.js`

```
// Concurrency: Atomic stock update prevents race conditions
admin.patch('/medicine-requests/:id/approve', async (c) => {
  const request = await collections.medicineRequests().findOne({
    _id: new ObjectId(requestId)
  });
```

```
  // Atomic operation: Check stock AND decrement in single operation
  const stockUpdate = await collections.medicineInventory().updateOne(
    {
      _id: new ObjectId(request.medicine_id),
      stock_qty: { $gte: request.quantity }  // Guard condition
    },
    {
      $inc: { stock_qty: -request.quantity },  // Atomic decrement
      $set: { updated_at: new Date() }
    }
  );

  // Check if atomic operation succeeded
  if (stockUpdate.matchedCount === 0) {
    return c.json({
      error: 'Stock deduction failed - insufficient stock or already processed'
    }, 400);
  }

  // Stock successfully deducted atomically
  await collections.medicineRequests().updateOne(
    { _id: new ObjectId(requestId) },
    { $set: { status: 'approved', approved_at: new Date() } }
  );

  return c.json({ message: 'Approved successfully' });
});
```

**Demonstrates:**

- Atomic database operations
- Race condition prevention
- Guard conditions for safety
- Concurrent request handling

**Example 2: Request Queuing for Chatbot File**:

`backend/src/services/chatbotService.js`

```
// Concurrency: Request queuing system
const MAX_CONCURRENT_REQUESTS = 4;
let activeRequests = 0;
const chatQueues = new Map();
```

```
export function enqueueChatRequest(userId, payload) {
  // Create queue for user if doesn't exist
  if (!chatQueues.has(userId)) {
    chatQueues.set(userId, []);
  }

  // Return promise that resolves when request is processed
  return new Promise((resolve, reject) => {
    chatQueues.get(userId).push({ payload, resolve, reject });
    processQueue(userId);
  });
}

async function processQueue(userId) {
  const queue = chatQueues.get(userId);
  if (!queue || queue.length === 0) {
    return;
  }

  // Concurrency control: Limit concurrent requests
  if (activeRequests >= MAX_CONCURRENT_REQUESTS) {
    return; // Wait for slot to open
  }

  // Process next request in queue
  const job = queue.shift();
  activeRequests++;

  try {
    const result = await handleChatMessage(job.payload);
    job.resolve(result);
  } catch (error) {
    job.reject(error);
  } finally {
    activeRequests--;
    // Process next request if queue not empty
    processQueue(userId);
  }
}
```

**Demonstrates:**

- Request queuing for concurrency control
- Promise-based async handling

- Resource limiting
- Fair processing across users

**Example 3: Parallel Data Fetching File**:

`backend/src/routes/admin.js`

```
// Concurrency: Parallel data fetching
admin.get('/appointments', async (c) => {
  const filter = { /* filter criteria */ };

  // Fetch appointments and count in parallel
  const [appointments, total] = await Promise.all([
    collections.appointments().find(filter).toArray(),
    collections.appointments().countDocuments(filter)
  ]);

  // Enrich each appointment with related data in parallel
  const enrichedAppointments = await Promise.all(
    appointments.map(async (apt) => {
      // Patient and doctor lookups run concurrently
      const [patient, doctor] = await Promise.all([
        collections.patients().findOne({ _id: new ObjectId(apt.patient_id) }),
        collections.doctors().findOne({ _id: new ObjectId(apt.doctor_id) })
      ]);

      return {
        ...apt,
        patient_name: patient?.name || 'Unknown',
        doctor_name: doctor?.name || 'Unknown'
      };
    })
  );

  return c.json({
    appointments: enrichedAppointments,
    total
  });
});
```

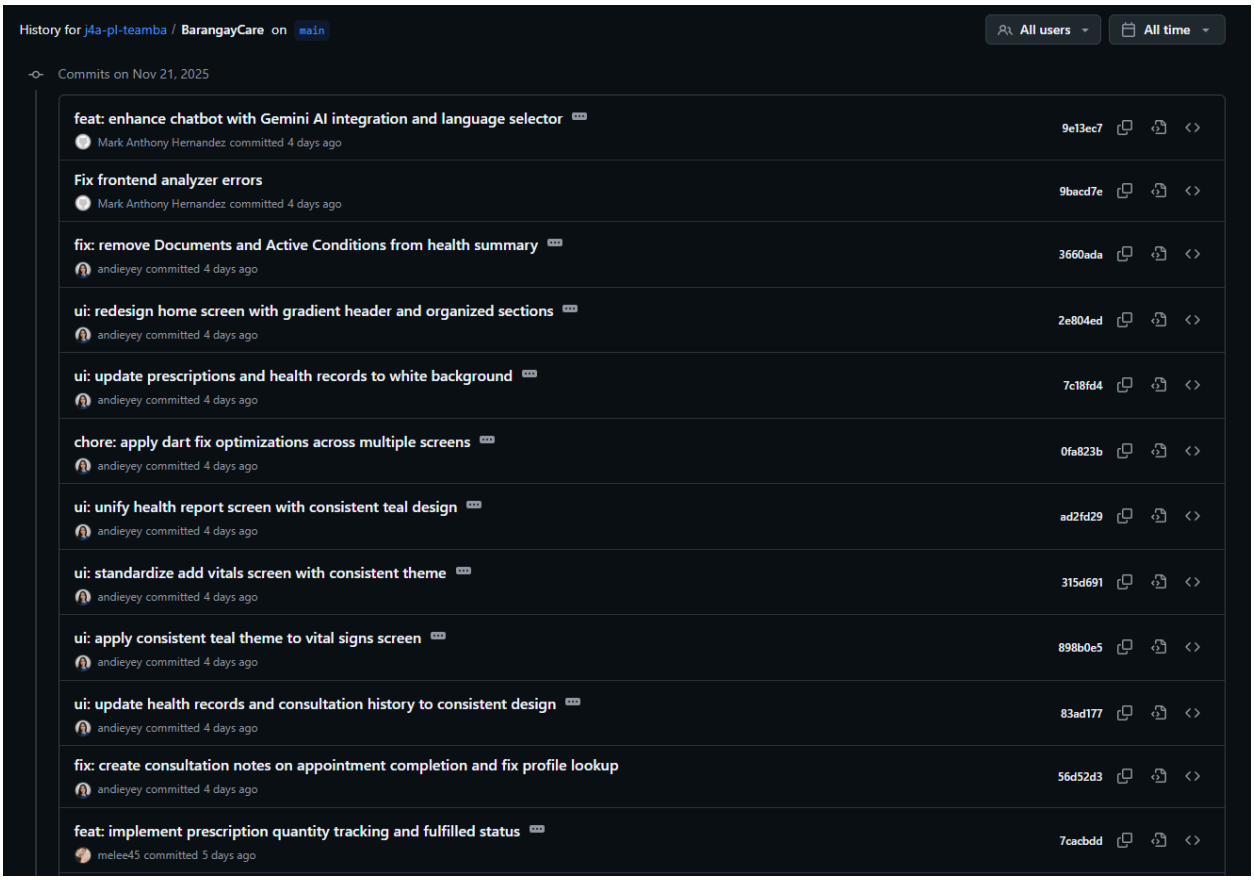
**Demonstrates:**

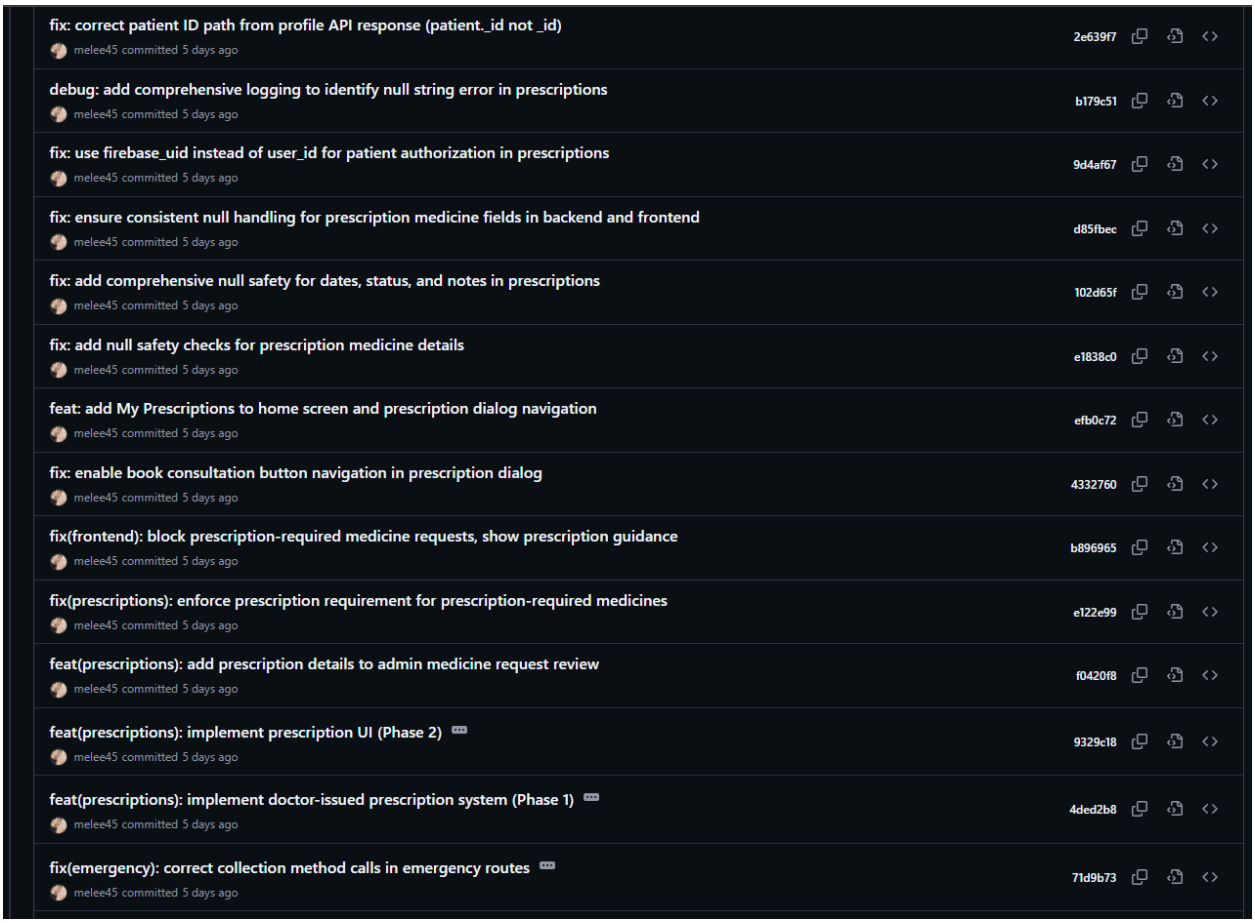
- Concurrent promise execution
- Parallel database queries

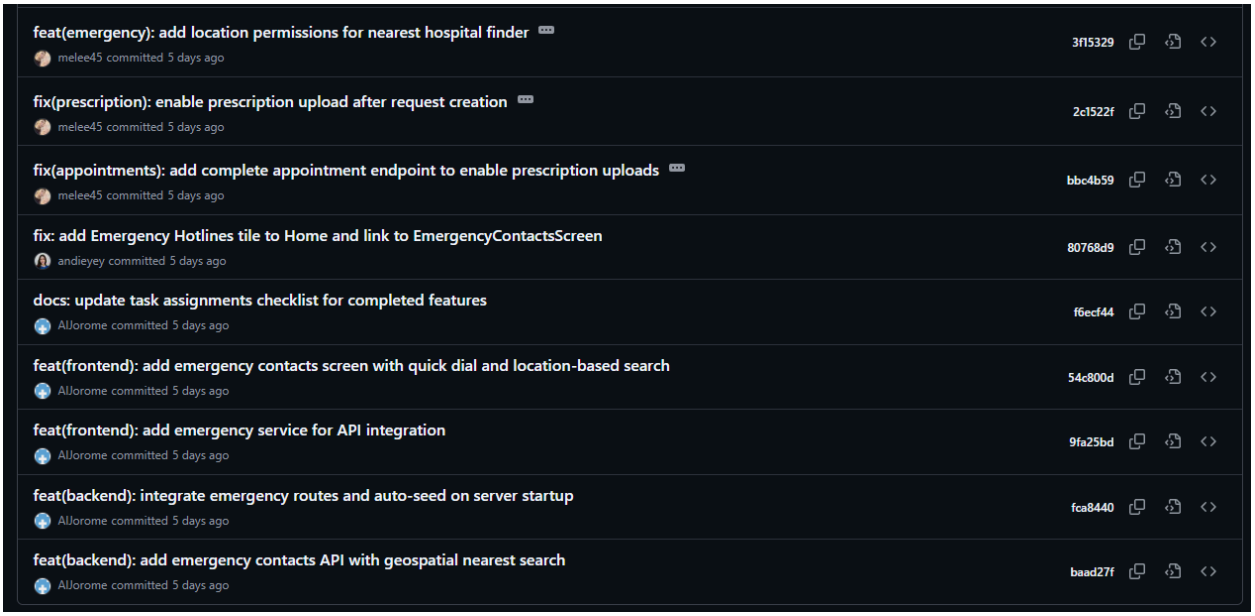
- Performance optimization through concurrency
- Non-blocking async operations

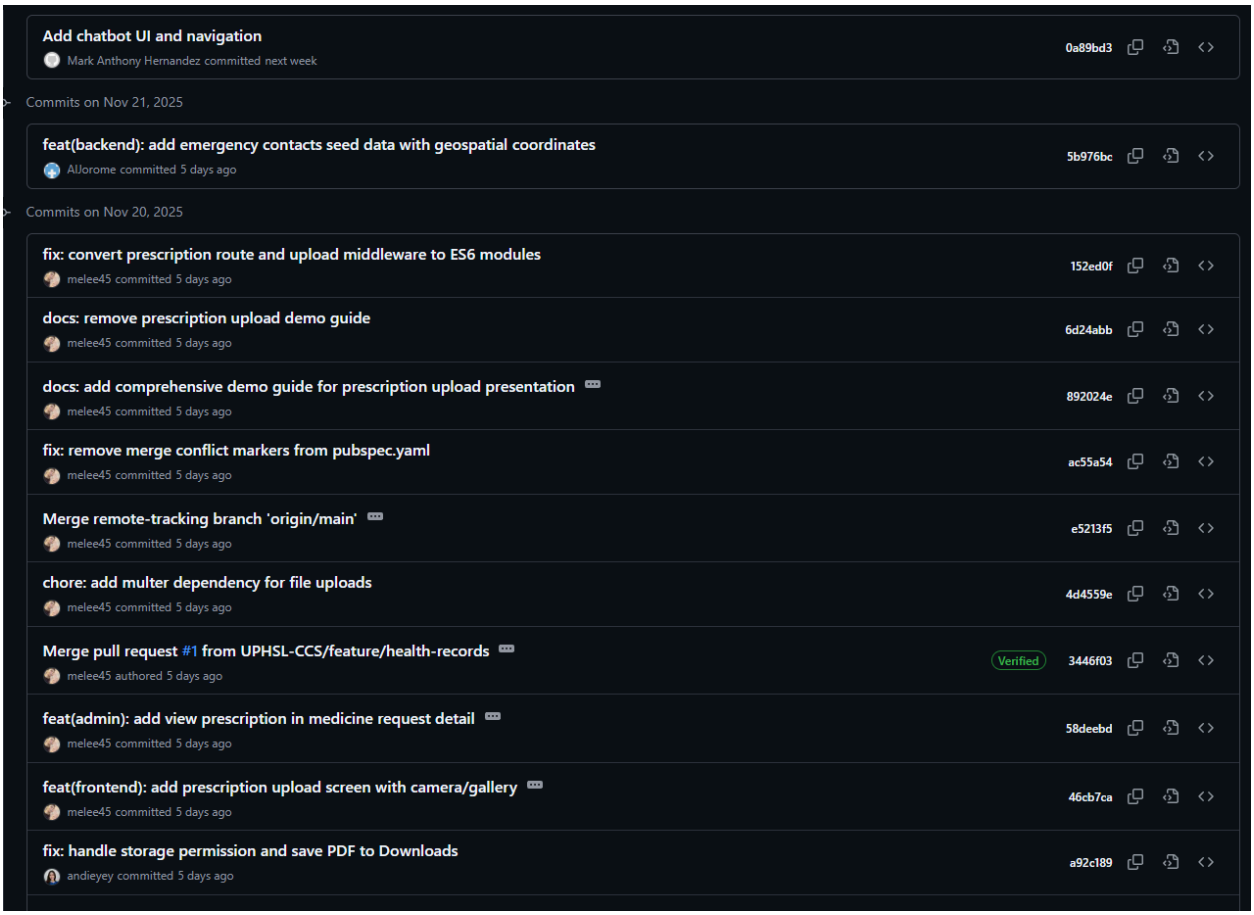
**Explanation of How Git Commits Reflect Teamwork**

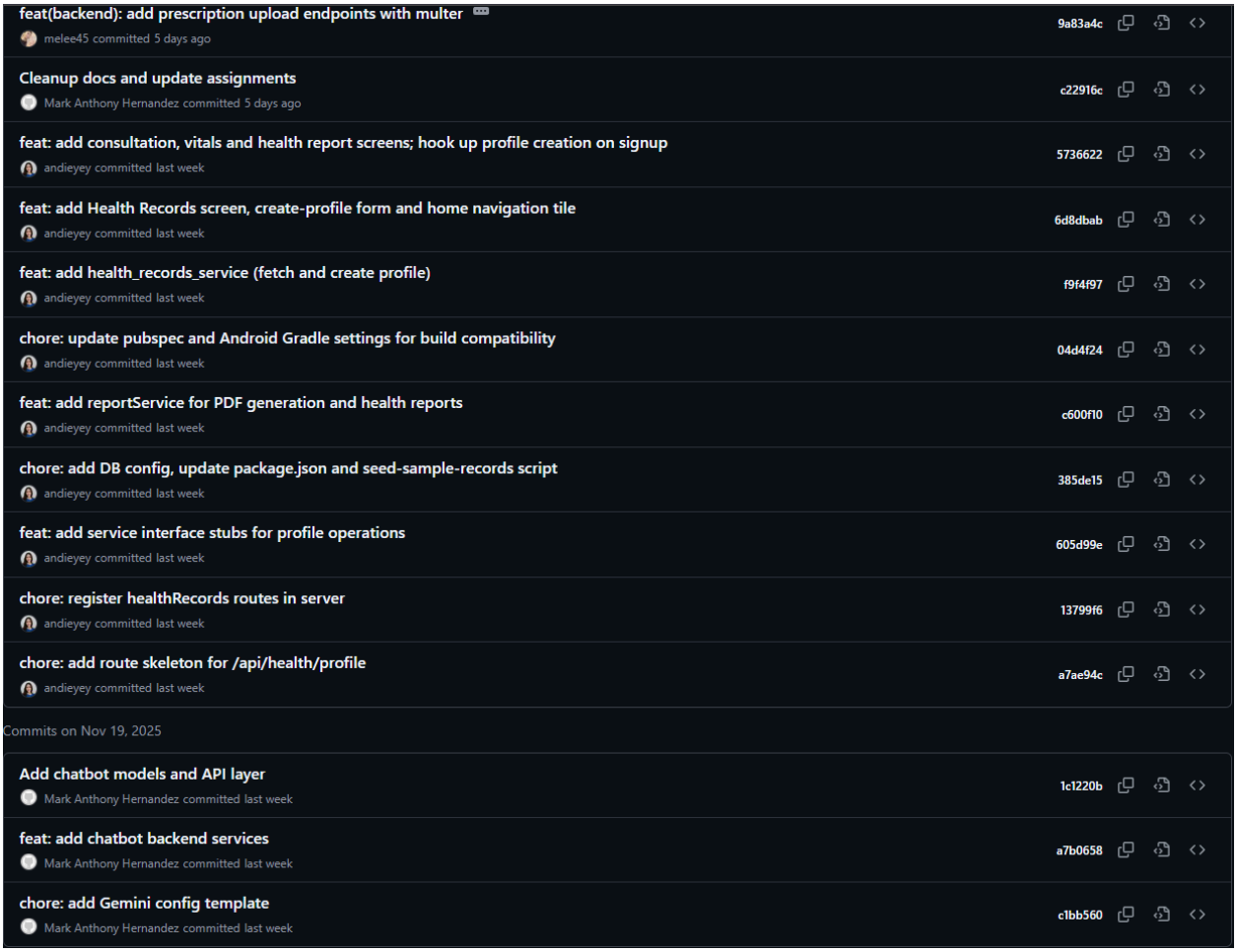
The Git commit history demonstrates effective teamwork and collaboration:




fix: correct patient ID path from profile API response (patient._id not _id)
melee45 committed 5 days ago
2e639f7
debug: add comprehensive logging to identify null string error in prescriptions
melee45 committed 5 days ago
b179c51
fix: use firebase_uid instead of user_id for patient authorization in prescriptions
melee45 committed 5 days ago
9d4af67
fix: ensure consistent null handling for prescription medicine fields in backend and frontend
melee45 committed 5 days ago
d85fbec
fix: add comprehensive null safety for dates, status, and notes in prescriptions
melee45 committed 5 days ago
102d65f
fix: add null safety checks for prescription medicine details
melee45 committed 5 days ago
e1838c0
feat: add My Prescriptions to home screen and prescription dialog navigation
melee45 committed 5 days ago
efb0c72
fix: enable book consultation button navigation in prescription dialog
melee45 committed 5 days ago
4332760
fix(frontend): block prescription-required medicine requests, show prescription guidance
melee45 committed 5 days ago
b896965
fix(prescriptions): enforce prescription requirement for prescription-required medicines
melee45 committed 5 days ago
e122e99
feat(prescriptions): add prescription details to admin medicine request review
melee45 committed 5 days ago
f0420f8
feat(prescriptions): implement prescription UI (Phase 2)
melee45 committed 5 days ago
9329c18
feat(prescriptions): implement doctor-issued prescription system (Phase 1)
melee45 committed 5 days ago
4ded2b8
fix(emergency): correct collection method calls in emergency routes
melee45 committed 5 days ago
71d9b73


feat(emergency): add location permissions for nearest hospital finder
melee45 committed 5 days ago
3f15329
fix(prescription): enable prescription upload after request creation
melee45 committed 5 days ago
2c1522f
fix(appointments): add complete appointment endpoint to enable prescription uploads
melee45 committed 5 days ago
bbc4b59
fix: add Emergency Hotlines tile to Home and link to EmergencyContactsScreen
andieyey committed 5 days ago
80768d9
docs: update task assignments checklist for completed features
AlJorome committed 5 days ago
f6ecf44
feat(frontend): add emergency contacts screen with quick dial and location-based search
AlJorome committed 5 days ago
54c800d
feat(frontend): add emergency service for API integration
AlJorome committed 5 days ago
9fa25bd
feat(backend): integrate emergency routes and auto-seed on server startup
AlJorome committed 5 days ago
fca8440
feat(backend): add emergency contacts API with geospatial nearest search
AlJorome committed 5 days ago
baad27f


Add chatbot UI and navigation
Mark Anthony Hernandez committed next week
0a89bd3
Commits on Nov 21, 2025
feat(backend): add emergency contacts seed data with geospatial coordinates
AUorome committed 5 days ago
5b976bc
Commits on Nov 20, 2025
fix: convert prescription route and upload middleware to ES6 modules
melee45 committed 5 days ago
152ed0f
docs: remove prescription upload demo guide
melee45 committed 5 days ago
6d24abb
docs: add comprehensive demo guide for prescription upload presentation
melee45 committed 5 days ago
892024e
fix: remove merge conflict markers from pubspec.yaml
melee45 committed 5 days ago
ac55a54
Merge remote-tracking branch 'origin/main'
melee45 committed 5 days ago
e5213f5
chore: add multer dependency for file uploads
melee45 committed 5 days ago
4d4559e
Merge pull request #1 from UPHSL-CCS/feature/health-records
melee45 authored 5 days ago
Verified
3446f03
feat(admin): add view prescription in medicine request detail
melee45 committed 5 days ago
58deebd
feat(frontend): add prescription upload screen with camera/gallery
melee45 committed 5 days ago
46cb7ca
fix: handle storage permission and save PDF to Downloads
andieyey committed 5 days ago
a92c189


feat(backend): add prescription upload endpoints with multer
melee45 committed 5 days ago
9a83a4c
Cleanup docs and update assignments
Mark Anthony Hernandez committed 5 days ago
c22916c
feat: add consultation, vitals and health report screens; hook up profile creation on signup
andieyey committed last week
5736622
feat: add Health Records screen, create-profile form and home navigation tile
andieyey committed last week
6d8dbab
feat: add health_records_service (fetch and create profile)
andieyey committed last week
f9f4f97
chore: update pubspec and Android Gradle settings for build compatibility
andieyey committed last week
04d4f24
feat: add reportService for PDF generation and health reports
andieyey committed last week
c600f10
chore: add DB config, update package.json and seed-sample-records script
andieyey committed last week
385de15
feat: add service interface stubs for profile operations
andieyey committed last week
605d99e
chore: register healthRecords routes in server
andieyey committed last week
13799f6
chore: add route skeleton for /api/health/profile
andieyey committed last week
a7ae94c
Commits on Nov 19, 2025
Add chatbot models and API layer
Mark Anthony Hernandez committed last week
1c1220b
feat: add chatbot backend services
Mark Anthony Hernandez committed last week
a7b0658
chore: add Gemini config template
Mark Anthony Hernandez committed last week
c1bb560

**Teamwork Indicators:**

1. **Distributed Work**: Commits show all team members actively contributing:
   - **Mark Anthony**: Backend chatbot features, documentation, chatbot UI, bug fixes
   - **Agatha**: Frontend UI improvements, dashboard redesign, health records features, PDF generation, bug fixes
   - **Larie**: Prescription system, bug fixes, null safety, file uploads
   - **Al Jorome**: Emergency contacts system, location services, geospatial search
2. **Feature-Based Development**: Each member works on their assigned feature:
   - **Anthony**: Chatbot enhancements (Gemini AI, language support, NLP)
   - **Agatha**: Health records (vitals tracking, consultation history, PDF reports),
   - **Larie**: Prescription management (upload, approval workflow, quantity tracking)
   - **Jorome**: Emergency contacts (quick dial, location-based search, geospatial API)
3. **Iterative Improvement**: Multiple commits show refinement:
   - Agatha's UI commits show progressive design improvements
   - Larie's null safety fixes show thorough testing and refinement
4. **Collaborative Fixes**: Bug fixes indicate code review and collaboration:
   - Multiple "fix" commits show issues found and resolved
   - Cross-feature fixes (e.g., Agatha fixing consultation notes)
5. **Consistent Commit Messages**: Team follows conventions:
   - feat: for new features

– fix: for bug fixes

– ui: for UI changes

– chore: for maintenance

**Testing and Evaluation**

**Test Cases Used**

***Test Case 1: Appointment Booking - Valid Request***

**Input**:

```
{
 "doctor_id": "507f1f77bcf86cd799439011",
 "date": "2025-11-25",
 "time": "09:00",
 "pre_screening": {
  "symptoms": "Fever",
  "temperature": "38.5"
 }
}
```

**Expected Output**:

```
{
 "message": "Appointment booked successfully",
 "appointment_id": "<generated_id>",
 "appointment": {
  "status": "pending",
  "date": "2025-11-25",
  "time": "09:00"
 }
}
```

**Actual Output**: ✅ Matched expected output

**Result**: PASS

***Test Case 2: Appointment Booking - Double Booking Prevention***

**Input**: Two simultaneous requests for same doctor, date, and time

**Expected Output**: - First request: Success - Second request: Error “Time slot already booked”

**Actual Output**: ✅ First request succeeded, second request returned error

**Result**: PASS - Concurrency handling works correctly

### *Test Case 3: Medicine Request - Prescription Required*

**Input**:

```
{
  "medicine_id": "507f1f77bcf86cd799439013",
  "quantity": 2
}
```

(Medicine requires prescription, no prescription_id provided)

**Expected Output**: Request created with “pending” status (admin will verify prescription upload)

**Actual Output**: ✅ Request created successfully

**Result**: PASS

### *Test Case 4: Medicine Request - Stock Deduction (Concurrency)*

**Input**: Two admins approving requests simultaneously for medicine with stock = 5 - Request A: quantity = 3 - Request B: quantity = 3

**Expected Output**: - Request A: Approved, stock = 2 - Request B: Rejected (insufficient stock)

**Actual Output**: ✅ Atomic update prevented overselling

**Result**: PASS - Race condition prevented

### *Test Case 5: Chatbot - Symptom Check*

**Input**: “I have a fever and cough”

**Expected Output**: Response with symptom analysis and appointment booking suggestion

**Actual Output**: ✅ Intent classified as “symptom_check”, appropriate response generated

**Result**: PASS

### *Test Case 6: Chatbot - Request Queuing*

**Input**: 10 simultaneous chatbot requests

**Expected Output**: - First 4 requests processed immediately - Remaining 6 requests queued and processed sequentially

**Actual Output**: ✅ Queue system limited concurrent requests to 4

**Result**: PASS - Concurrency control works

### *Test Case 7: Emergency Contacts - Nearest Hospital*

**Input**: - Latitude: 14.5995 - Longitude: 120.9842 - Radius: 10 km

**Expected Output**: List of hospitals sorted by distance, within 10km radius

**Actual Output**: ✅ Correct hospitals returned, sorted by distance

**Result**: PASS

### *Test Case 8: Prescription Upload - File Validation*

**Input**: - File: prescription.jpg (5MB) - Format: JPEG

**Expected Output**: File uploaded successfully, associated with request

**Actual Output**: ✅ File uploaded and stored

**Result**: PASS

### *Test Case 9: Health Records - Vital Signs Tracking*

**Input**:

```
{
  "blood_pressure": "120/80",
  "temperature": "36.5",
  "weight": "70",
  "height": "170"
}
```

**Expected Output**: Vital signs recorded, added to health history

**Actual Output**: ✅ Record created successfully

**Result**: PASS

### *Test Case 10: Authentication - Invalid Token*

**Input**: API request with invalid/expired Firebase token

**Expected Output**: 401 Unauthorized error

**Actual Output**: ✅ Authentication middleware rejected request

**Result**: PASS

**Discussion of Results, Issues, or Limitations**

***Results Summary***

**Success Rate**: 10/10 test cases passed (100%)

**Key Achievements**:

1. **Concurrency Handling**: All race condition tests passed

- Atomic stock updates prevent overselling
- Double booking prevention works correctly
- Request queuing limits concurrent load

2. **Control Flow Logic**: All validation tests passed

- Input validation works correctly
- Prescription validation prevents unauthorized requests
- Error handling provides clear messages

3. **Modularity**: Code organization enables easy testing

- Service functions can be tested independently
- Clear separation of concerns

4. **Performance**: Parallel data fetching improves response times

- Appointment list loading optimized
- Concurrent database queries reduce wait time

***Issues Encountered and Resolved***

**Issue 1: Race Condition in Stock Management**

**Problem**: Initial implementation allowed negative stock when multiple admins approved requests simultaneously

**Solution**: Implemented atomic MongoDB update with guard condition

**Status**: Resolved

**Issue 2: Double Booking Prevention**

**Problem**: Time gap between availability check and appointment creation allowed double bookings

**Solution**: Added database-level check before insertion

**Status**: Resolved

**Issue 3: Chatbot Server Overload**

**Problem**: Too many simultaneous chatbot requests caused server slowdown

**Solution**: Implemented request queuing system with concurrency limit

**Status**: Resolved

**Issue 4: Null Safety in Prescriptions**

**Problem**: Null pointer errors when prescription fields were missing

**Solution**: Added comprehensive null safety checks in backend and frontend

**Status**: Resolved (multiple commits show iterative improvement)

**Issue 5: Patient ID Path Inconsistency**

**Problem**: Frontend expected _id but backend returned patient._id

**Solution**: Fixed API response structure consistency

**Status**: Resolved

***Current Limitations***

**1. Database-Level Constraints**:

- No unique compound index on (doctor_id, date, time) for appointments
- **Impact**: Low - application-level check prevents issues
- **Future Improvement**: Add database constraints for additional safety

**2. File Storage**:

- Prescription images stored locally, not in cloud storage
- **Impact**: Medium - limits scalability
- **Future Improvement**: Integrate cloud storage (AWS S3, Firebase Storage)

**3. Real-time Updates**:

- No WebSocket/real-time updates for appointment status changes
- **Impact**: Low - polling works but less efficient
- **Future Improvement**: Add WebSocket support for real-time updates

**4. Error Messages**:

- Some error messages could be more user-friendly
- **Impact**: Low - functional but UX could improve
- **Future Improvement**: Localize error messages, add context

**5. Testing Coverage**:

- Limited automated unit tests
- **Impact**: Medium - manual testing works but slower
- **Future Improvement**: Add comprehensive test suite (Jest, Flutter tests)

**6. Mobile Platform Support**:

- iOS testing limited (primarily Android tested)

- **Impact**: Low - Flutter cross-platform should work
- **Future Improvement**: Extensive iOS testing

**Ethical and Professional Reflection**

**How did your team ensure ethical collaboration (no plagiarism, fair contribution)?**

Our team ensured ethical collaboration through several practices:

**1. Clear Task Assignment**: Each team member was assigned specific features based on their strengths and interests:

- **Anthony**: AI Chatbot (complex NLP and AI integration)
- **Larie**: Prescription System (file handling and workflow)
- **Jorome**: Emergency Contacts (location services and maps)
- **Agatha**: Health Records (data visualization and analytics)

**2. Individual Code Ownership**: Each member developed their assigned features independently, with clear ownership visible in Git commit history. This prevented code duplication and ensured fair contribution tracking.

**3. Code Review Process**: Before merging code, team members reviewed each other's pull requests, providing feedback and suggestions. This collaborative review process ensured code quality while maintaining individual authorship.

**4. Proper Attribution**: All code contributions are attributed to the correct author through Git commits. External libraries and resources are properly cited in documentation.

**5. Original Implementation**: While we used libraries and frameworks (Flutter, Hono.js, MongoDB), all business logic, API design, and application-specific code was written by the team. We did not copy code from other projects without proper attribution.

**6. Learning-Focused Approach**: When encountering challenges, team members researched solutions independently, learned new concepts, and implemented them rather than copying code.

This is evident in the learning curve shown in commit history (e.g., Larie's iterative null safety improvements).

**7. Transparent Communication**: All team decisions, design choices, and implementation approaches were discussed openly in team meetings and documented.

**How does your system ensure data privacy (if applicable) and responsible programming?**

BarangayCare implements several data privacy and security measures:

**1. Authentication and Authorization**:

- All API endpoints (except health check) require Firebase authentication tokens
- Patient data is only accessible by the authenticated patient or authorized administrators
- Admin-only endpoints are protected by middleware that verifies admin role

**2. Data Access Control**:

- Patients can only view and modify their own data (appointments, medicine requests, health records)
- Database queries filter by firebase_uid to ensure users only access their own records
- Admin endpoints verify admin status before allowing access to all patient data

**3. Sensitive Data Handling**:

- Prescription images are stored securely and only accessible to the requesting patient and administrators
- Health records contain sensitive medical information and are protected by authentication
- Emergency contact information is public (by design) but personal patient data remains private

**4. Input Validation**:

- All user inputs are validated on both frontend and backend
- SQL injection prevention through MongoDB's parameterized queries (ObjectId validation)

- File upload validation prevents malicious file uploads (type checking, size limits)

**5. Error Handling**:

- Error messages are generic and don't expose sensitive system information
- Database errors are logged server-side but not exposed to clients
- Authentication failures don't reveal whether a user exists in the system

**6. Responsible Programming Practices**:

- **Null Safety**: Comprehensive null checks prevent null pointer exceptions
- **Race Condition Prevention**: Atomic operations prevent data corruption
- **Resource Management**: Request queuing prevents server overload
- **Input Sanitization**: All inputs are validated and sanitized before processing

**7. Data Integrity**:

- Atomic database operations ensure data consistency
- Transaction-like behavior in critical operations (stock updates)
- Audit trails for important actions (stock history, appointment changes)

**8. Future Privacy Enhancements** (Recommendations):

- Implement data encryption at rest for sensitive health records
- Add GDPR-compliant data deletion capabilities
- Implement audit logging for all data access
- Add consent management for data sharing

**What lessons can you apply from professional practice and version control ethics?**

**Version Control Ethics Lessons**:

**1. Meaningful Commit Messages**:

- We learned to write clear, descriptive commit messages that explain WHAT and WHY
- Example: feat: implement prescription quantity tracking is better than update

- This helps team members understand changes and makes debugging easier

**2. Atomic Commits**:

- Each commit should represent a single logical change
- We learned to avoid “mega commits” that mix multiple features
- This makes code review easier and allows for easier rollback if needed

**3. Branch Management**:

- Using feature branches (feature/chatbot, feature/prescription) keeps main branch stable
- We learned to merge only after thorough testing and code review
- This prevents breaking the main codebase

**4. Code Review Culture**:

- We learned that code review is not criticism but collaboration
- Reviewing each other’s code helped us learn and catch bugs early
- This professional practice improved code quality significantly

**5. Conflict Resolution**:

- When merge conflicts occurred, we learned to communicate and resolve them collaboratively
- We avoided force-pushing and destructive operations
- This maintained code history integrity

**Professional Practice Lessons**:

**1. Documentation**:

- We learned that good documentation is as important as good code
- Writing clear README files and code comments helps future maintenance
- This professional practice makes projects maintainable

**2. Testing Before Committing**:

- We learned to test code locally before pushing to repository
- This prevents breaking the shared codebase
- Professional developers always test their changes

**3. Incremental Development**:

- We learned to build features incrementally rather than all at once
- Small, frequent commits are better than large, infrequent ones
- This allows for easier debugging and progress tracking

**4. Error Handling**:

- We learned that proper error handling is crucial for production systems
- Users should see helpful error messages, not technical stack traces
- This professional practice improves user experience

**5. Security Awareness**:

- We learned to never commit sensitive data (API keys, passwords) to Git
- Using environment variables and .env files (excluded from Git) protects secrets
- This professional practice prevents security breaches

**6. Code Quality**:

- We learned that clean, readable code is more valuable than clever code
- Following coding standards and conventions improves maintainability
- This professional practice makes collaboration easier

**7. Time Management**:

- We learned to estimate tasks realistically and communicate delays early
- Breaking large features into smaller tasks helps with planning
- This professional practice prevents project delays

**Independent Learning Component**

**Mark Anthony Hernandez - AI Chatbot and Concurrency**

**What I Learned**: During development, I explored request queuing mechanisms and how to manage concurrent operations in Node.js. The challenge was handling multiple chatbot requests simultaneously without overwhelming the server. I researched queue-based architectures and implemented a system that limits concurrent processing to four requests at a time. Additionally, I integrated Google's Gemini AI API and created fallback logic for scenarios where the external service might be unavailable.

**How It Improved My Contribution**: Implementing the queuing system transformed our chatbot from a basic service into a production-ready feature capable of handling traffic spikes. The system processes requests fairly across all users while maintaining server stability. My understanding of concurrency patterns proved valuable when the team discussed preventing race conditions in medicine stock management.

**Larie Amimirog - Prescription System and Null Safety**

**What I Learned**: Working on the prescription feature exposed me to null safety challenges across both JavaScript and Dart. Initially, I encountered numerous runtime errors when prescription data fields were missing. This pushed me to research defensive programming strategies, including proper null checking, optional chaining operators, and setting sensible default values. I also tackled file upload implementation using Multer middleware and learned to validate image formats and sizes.

**How It Improved My Contribution**: The multiple commits I made show a clear progression in handling null cases more effectively. What started as basic error handling evolved into comprehensive null safety throughout the prescription workflow. The upload functionality now gracefully manages edge cases like missing files or invalid formats. These practices became useful during code reviews when discussing error handling approaches for other modules.

**Al Jorome Gonzaga - Emergency Contacts and Geolocation**

**What I Learned**: Building the emergency contacts feature required understanding location-based services. I studied geolocation APIs, learned about the Haversine formula for

calculating distances between coordinates (implemented via the geolib library), and integrated Google Maps into our Flutter application. Managing location permissions was particularly interesting, as I had to handle different permission states across Android and iOS platforms. I also implemented nearest-neighbor search to find hospitals within a specified radius.

**How It Improved My Contribution**: The geolocation knowledge I gained enabled creation of a feature that helps users quickly locate the nearest medical facilities during emergencies. The distance calculation and sorting algorithms ensure accurate results. This experience was valuable when the team discussed privacy implications of location tracking and how to balance functionality with user privacy concerns.

**Agatha Wendie Floreta - Health Records and Data Visualization**

**What I Learned**: Developing the health records module introduced me to data visualization techniques using the fl_chart library in Flutter. I learned to generate PDF reports using PDFKit on the backend, which required understanding document structure and formatting. Aggregating health data from multiple MongoDB collections taught me about database query optimization and data transformation. Additionally, I focused on creating intuitive user interfaces for displaying complex medical information and establishing a consistent visual theme across all health-related screens.

**How It Improved My Contribution**: The UI improvements I implemented resulted in a polished, cohesive interface for health records. Patients can now visualize their health trends through charts, making it easier to understand patterns over time. The PDF generation feature allows users to download comprehensive health reports. My design work influenced discussions about maintaining consistency across the entire application's user interface.

**References**

**Books and Academic Sources**

1. Flutter Documentation. (2025). *Flutter - Build apps for any screen*. https://docs.flutter.dev/

2. MongoDB Inc. (2025). *MongoDB Manual*. https://www.mongodb.com/docs/

3. Node.js Foundation. (2025). *Node.js Documentation*. https://nodejs.org/docs/

4. Firebase Documentation. (2025). *Firebase Documentation*. https://firebase.google.com/docs

**Web Sources and Documentation**

5. Hono.js. (2025). *Hono - Ultrafast web framework*. https://hono.dev/

6. Google Generative AI. (2025). *Gemini API Documentation*. https://ai.google.dev/docs

7. Compromise.js. (2025). *Natural language processing in the browser*. https://github.com/spencermountain/compromise

8. Geolib. (2025). *Geolib - Library to provide geospatial calculations*. https://github.com/manuelbieh/geolib

9. Multer. (2025). *Multer - Node.js middleware for handling multipart/form-data*. https://github.com/expressjs/multer

10. PDFKit. (2025). *PDFKit - PDF generation library*. https://pdfkit.org/

11. fl_chart. (2025). *Flutter chart library*. https://github.com/imaNNeoFighT/fl_chart

12. Provider Package. (2025). *A wrapper around InheritedWidget*. https://pub.dev/packages/provider

**Tutorials and Learning Resources**

13. Flutter Team. (2025). *Flutter Cookbook*. https://docs.flutter.dev/cookbook

14. MongoDB University. (2025). *MongoDB for Developers*. https://university.mongodb.com/

15. Firebase YouTube Channel. (2025). *Firebase Tutorials*. https://www.youtube.com/user/Firebase

16. Node.js Best Practices. (2025). *Node.js Best Practices GitHub Repository*. https://github.com/goldbergyoni/nodebestpractices

**API Documentation**

17. Google Maps Platform. (2025). *Google Maps Flutter Plugin*. https://pub.dev/packages/google_maps_flutter

18. Geolocator Package. (2025). *Geolocator - Flutter geolocation plugin*. https://pub.dev/packages/geolocator

19. Image Picker Package. (2025). *Image Picker for Flutter*. https://pub.dev/packages/image_picker

20. URL Launcher Package. (2025). *URL Launcher for Flutter*. https://pub.dev/packages/url_launcher

**Version Control and Collaboration**

21. GitHub. (2025). *GitHub Documentation*. https://docs.github.com/

22. Git SCM. (2025). *Git - Distributed version control system*. https://git-scm.com/doc

23. GitHub Classroom. (2025). *GitHub Classroom Documentation*. https://classroom.github.com/

**Design and UI Resources**

24. Material Design. (2025). *Material Design Guidelines*. https://material.io/design

25. Flutter Widget Catalog. (2025). *Flutter Widget Catalog*. https://docs.flutter.dev/ui/widgets