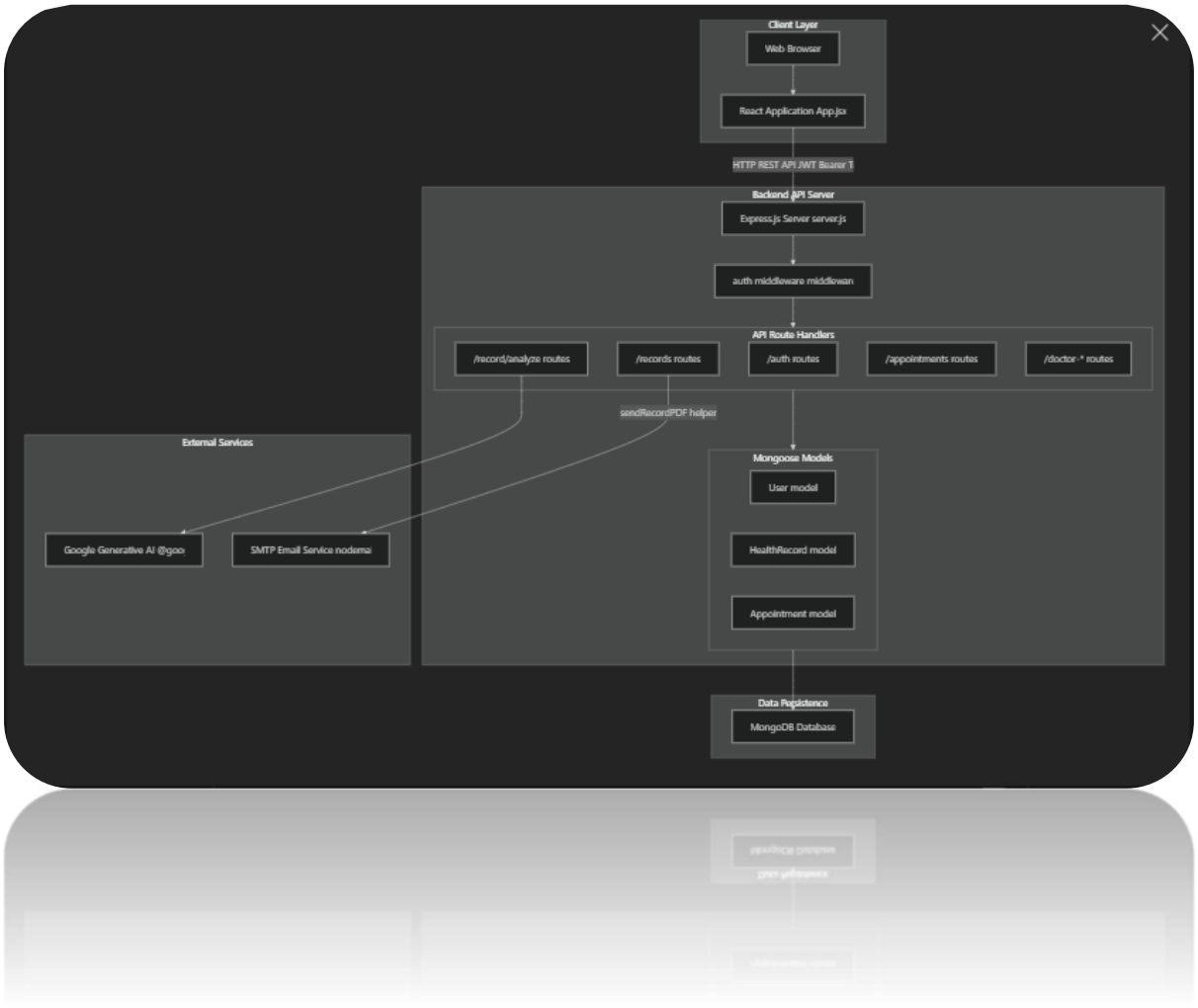
Sr No.	Name.
1	System Architecture
2	User Roles and Capabilities
3	Core Features Overview
4	Technology Stack
5	Backend Route Architecture
6	Frontend Component Architecture
7	Key Data Flow: Record Creation with AI Analysis
8	Authentication and Authorization Flow
9	Project Structure Overview
10	Feature Architecture Overview
11	Feature Integration Flow
12	Feature Interaction Matrix
13	Role-Based Feature Access
14	Core Features Summary
15	Feature Activation Sequence
16	Health Records Management
17	Access Control
18	CRUD Operations

19	Patient Records Interface
20	Doctor Records Management Interface
21	Automated PDF and Email Workflow
22	AI Integration
23	Health Record Analysis System
24	Clinical Communication Assistant
25	Strict Context Limitation
26	AI Model Configuration

1.System Architecture

Meditrack follows a three-tier architecture with a React frontend, Express.js backend API server, and MongoDB database. The system integrates with Google Generative AI for intelligent health analysis and uses SMTP services for email notifications.

High-Level Architecture Diagram



2.User Roles and Capabilities

Meditrack implements role-based access control with two primary user types stored in the User model:

Role	*Capabilities*	*Access Level*
Doctor	Create, read, update, delete health records View all patients and their records Approve/reject appointments Send emails to patients Access doctor-specific dashboard	Full CRUD on health records Protected routes require role === "doctor"
Patient	View own health records (read-only) Request AI analysis of records Chat with AI about health records Request appointments Receive email notifications	Read-only access to own records Protected by user ID matching

The auth middleware verifies JWT tokens and attaches user information (including role field) to req.user for all protected routes. Role checks are enforced at the route handler level.

3.Core Features Overview

3.1.Health Records Management

The records system centers around the HealthRecord model, which stores:

Patient notes and observations

Prescriptions (array of strings)

Vital signs (blood pressure, pulse, temperature)

Doctor and patient references

Key Routes:

POST /records - Create new record (doctor only)

GET /records/patient/:id - Retrieve patient records

PUT /records/:id - Update existing record (doctor only)

DELETE /records/:id - Delete record (doctor only)

Each record creation or update triggers asynchronous PDF generation via the sendRecordPDF helper function.

3.2.AI-Powered Analysis

The system integrates Google's Gemini AI (gemini-2.5-flash model) through the @google/generative-ai package. The /record/analyze endpoint implements safety constraints:

Analyze this health record safely.

Do not diagnose diseases.

Give general safe advice.

The frontend provides two AI interaction modes:

One-shot analysis - Single AI analysis of a complete record

Conversational chat - Multi-turn dialogue where questions are appended to record context

The formatAIText() function in HealthTipsPage.jsx parses raw AI output into structured HTML with headings, lists, and warning styles.

3.3.Appointment Scheduling

Appointments track doctor-patient scheduling with status transitions:

requested - Initial patient request

accepted - Doctor approval

rejected - Doctor denial

completed - Post-appointment

3.4.PDF & Email Notifications

The sendRecordPDF helper (called from the records routes) implements asynchronous PDF generation:

1. Fetches record from database
2. Generates PDF using PDFKit
3. Sends email via Nodemailer to patient's email address

This is a "fire-and-forget" pattern where API responses don't wait for email delivery.

4.Technology Stack

4.1.Backend Dependencies

Package	*Version*	*Purpose*
Express	^4.18.2	Web server framework
Mongoose	^7.0.0	MongoDB ODM
Jsonwebtoken	^9.0.0	JWT authentication
Bcryptjs	^2.4.3	Password hashing
@google/generative-ai	^0.24.1	Gemini AI integration
Nodemailer	^6.9.1	Email delivery
Pdfkit	^0.17.2	PDF generation
Cors	^2.8.5	Cross-origin requests
Dotenv	^16.0.0	Environment configuration

Frontend Stack

- **React** - Component-based UI framework
- **axios** - HTTP client wrapped in custom API module
- **localStorage** - Client-side token and user data persistence

5.Backend Route Architecture

The backend organizes API endpoints by functional domain, with authentication middleware protecting most routes:

5.1.Route-to-Code Mapping

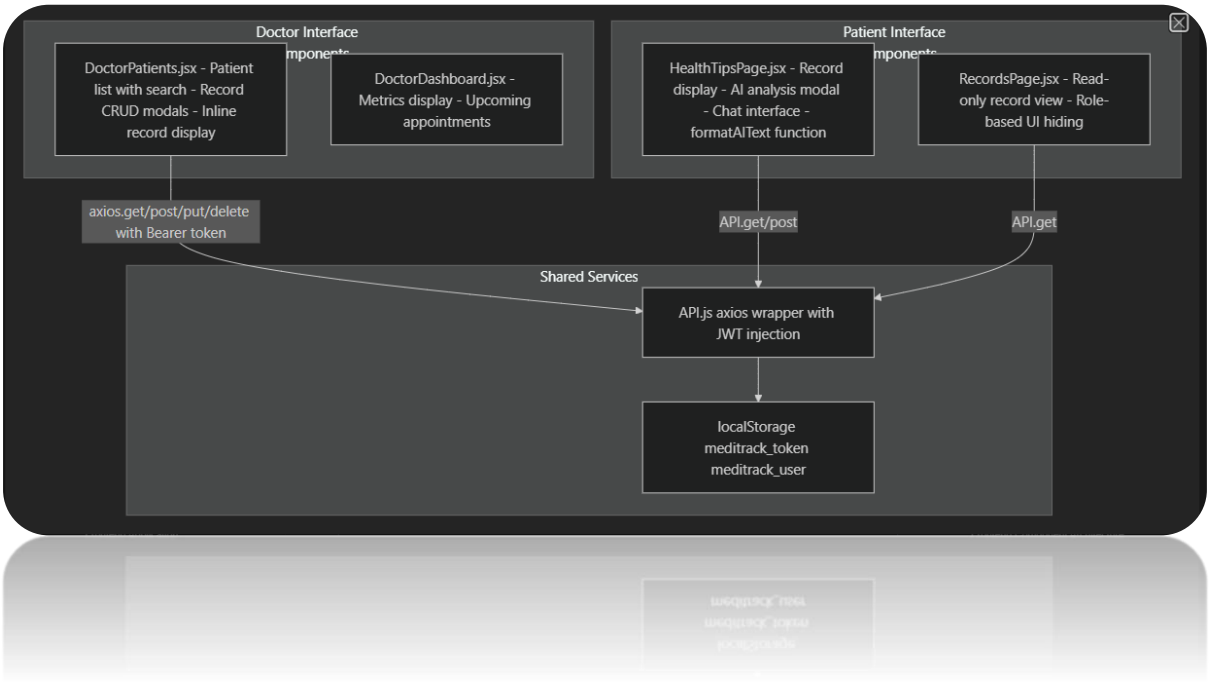
Key Implementation Details:

- auth middleware verifies JWT from Authorization: Bearer <token> header
- req.user object contains _id, role, and other user fields after authentication
- Role checks like req.user.role !== "doctor" enforce authorization
- Most routes return JSON responses with res.json()

6.Frontend Component Architecture

The React frontend implements role-based interfaces with distinct components for doctors and patients:

Component-to-File Mapping



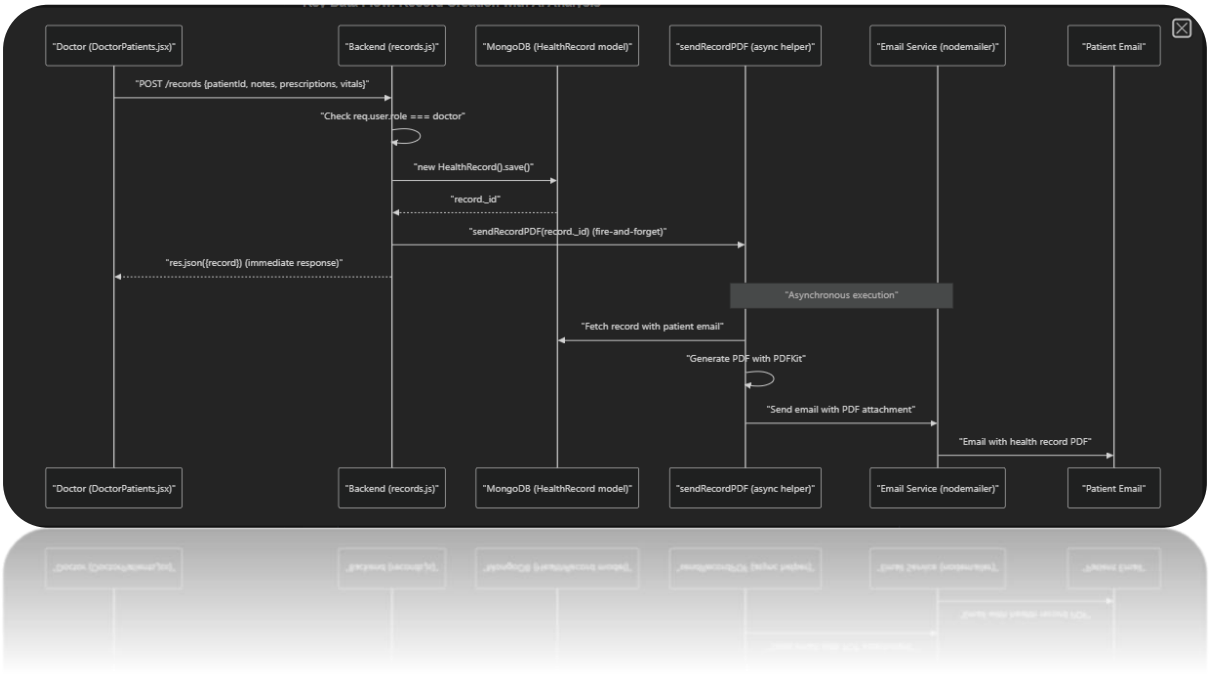
Key Implementation Patterns:

- Components check user.role === "patient" to conditionally render UI elements
- The API module automatically injects JWT token from localStorage
- Doctor components use direct axios calls with manual token headers
- Patient components use the centralized API wrapper
- State management via React hooks (useState, useEffect)

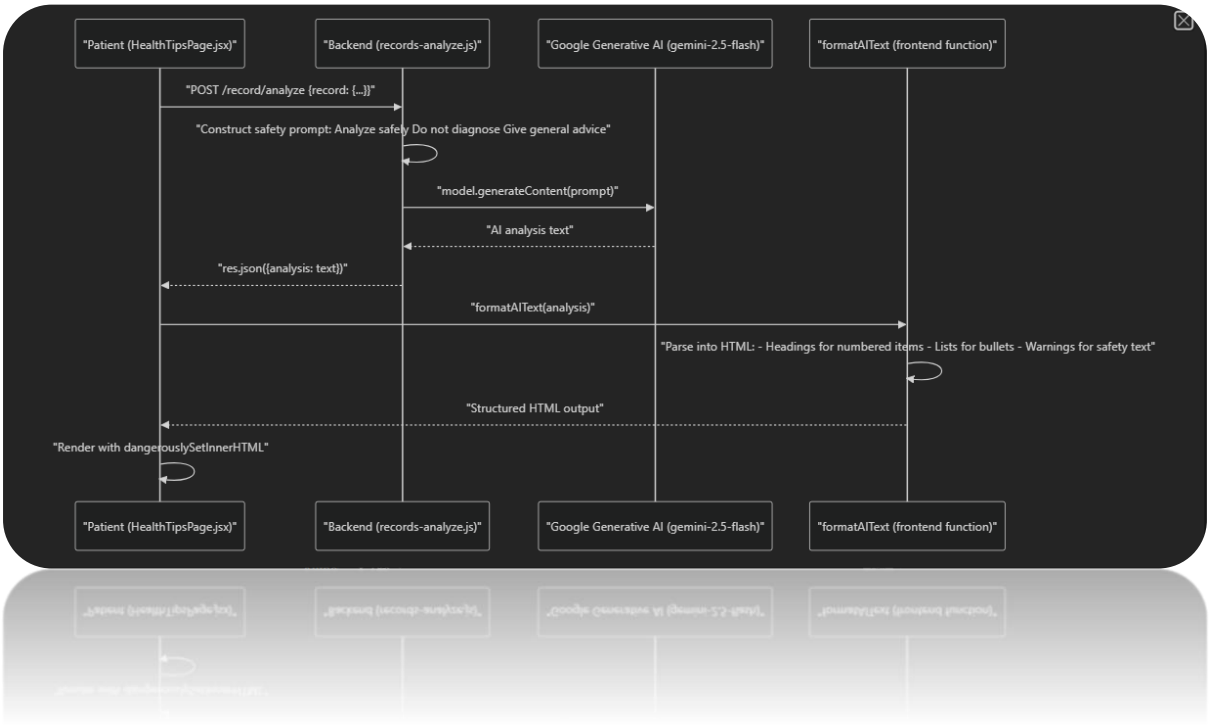
7.Key Data Flow: Record Creation with AI Analysis

The following sequence demonstrates the complete lifecycle of a health record from creation through AI analysis:

7.1.Record Creation and PDF Generation Flow



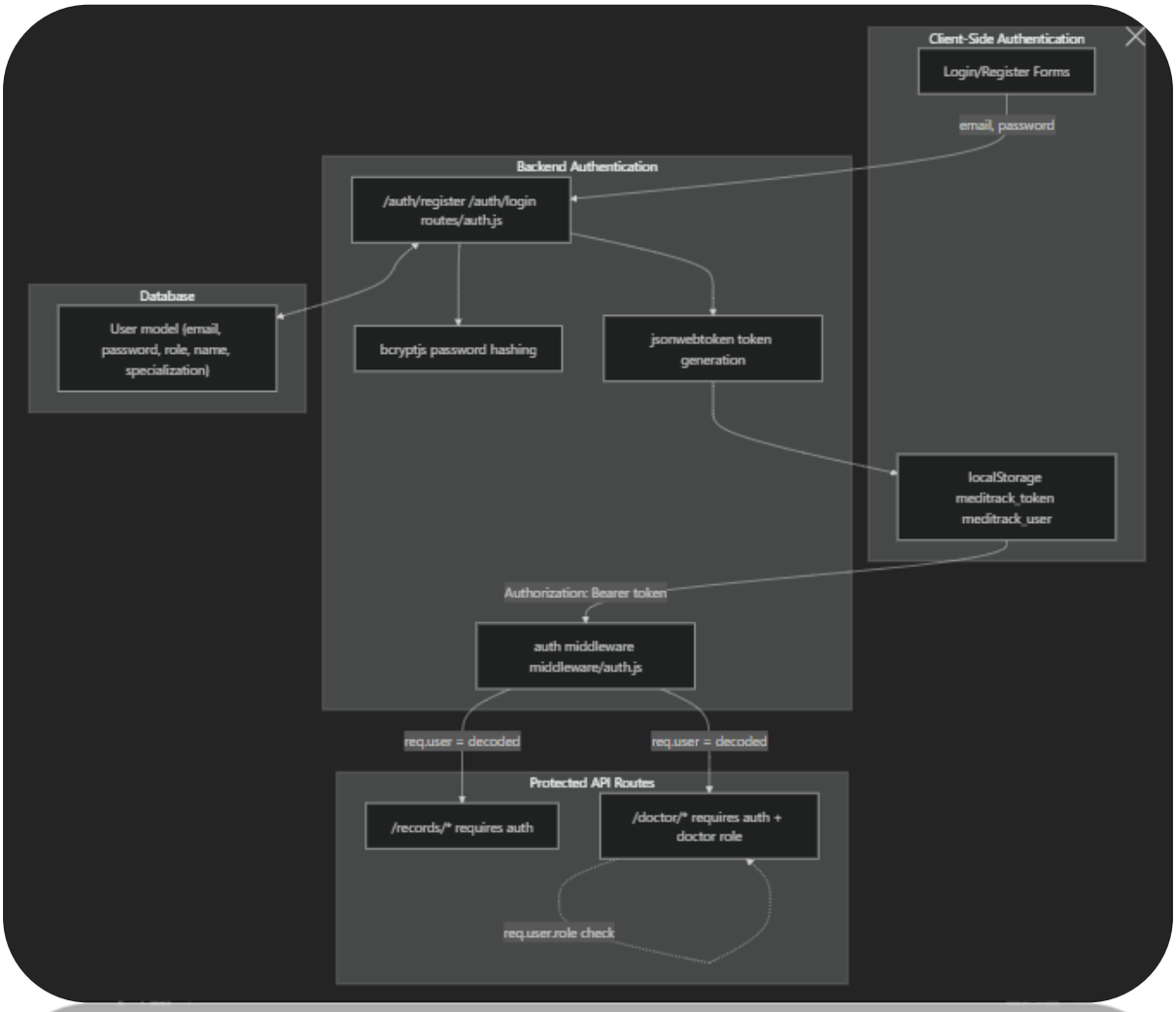
7.2.AI Analysis Request Flow



8.Authentication and Authorization Flow

The system implements JWT-based authentication with role-based access control:

8.1.Authentication Architecture



Authorization Checks:

- Line-level role verification: if (req.user.role !== "doctor") return res.status(403)
- Patient ID matching: if (req.user.role === "patient" && req.user._id.toString() !== id) return res.status(403)
- Frontend conditional rendering: {!isPatient && <button>...</button>}

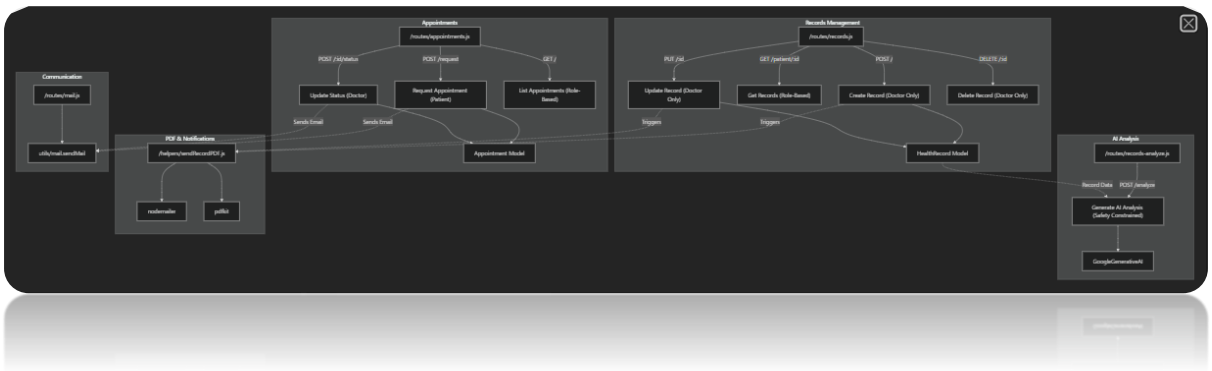
9.Project Structure Overview

```
Meditrack-React/
├── backend/
│   ├── server.js              # Express app entry point
│   ├── models/                # Mongoose schemas
│   │   ├── User.js            # User model (role, email, password)
│   │   ├── HealthRecord.js    # Health record model
│   │   └── Appointment.js     # Appointment model
│   ├── routes/                # API route handlers
│   │   ├── auth.js            # Authentication endpoints
│   │   ├── records.js         # Health records CRUD
│   │   ├── records-analyze.js # AI analysis endpoint
│   │   ├── appointments.js    # Appointment scheduling
│   │   └── doctor-*.js        # Doctor-specific routes
│   ├── middleware/
│   │   └── auth.js            # JWT verification middleware
│   ├── helpers/
│   │   └── sendRecordPDF.js   # PDF generation and email
│   └── package.json           # Backend dependencies
│
└── frontend/
    └── src/
        ├── App.jsx            # React app root
        ├── api.js             # Axios wrapper with auth
        ├── doctor/
        │   └── DoctorPatients.jsx  # Doctor interface
        └── patient/
            ├── RecordsPage.jsx     # Patient record view
            └── HealthTipsPage.jsx  # AI analysis interface
```

10.Feature Architecture Overview

Meditrack implements five core functional areas that work together to provide a complete healthcare records management platform. The system enforces role-based access control throughout, with doctors having administrative capabilities and patients having read/view access with AI-enhanced insights.

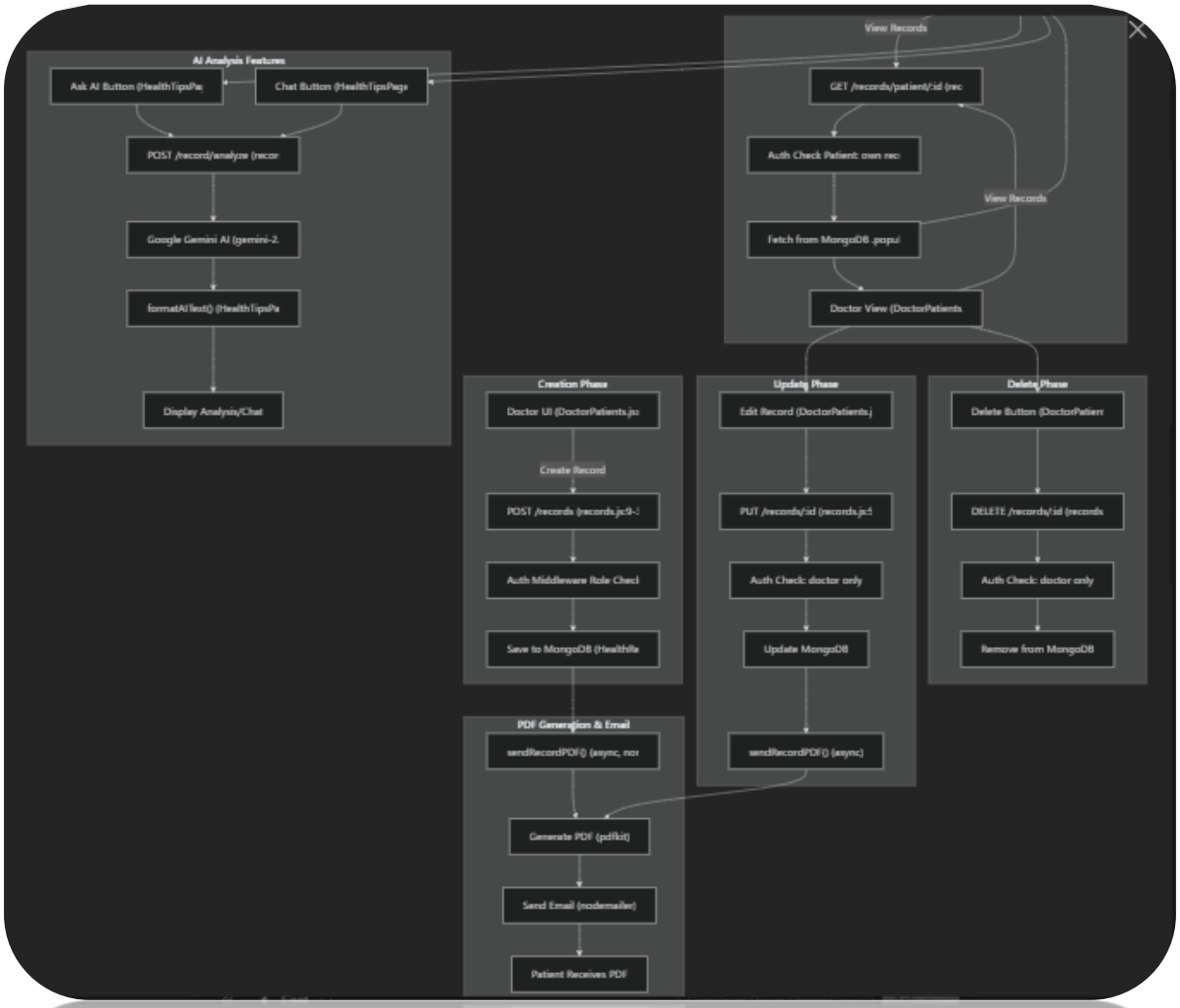
Core Feature Map



11.Feature Integration Flow

The core features integrate through shared data models and event-driven triggers. The primary integration pattern is the health record lifecycle, where record creation/updates automatically trigger downstream notifications while AI analysis and appointments operate on existing record data.

Health Record Lifecycle with Feature Interactions



12.Feature Interaction Matrix

The following table shows how each feature interacts with others and what triggers these interactions:

Source Feature	*Target Feature*	*Interaction Type*	*Trigger*	*Implementation*
Health Records	PDF/Email	Event-driven	Record create/update	records.js27 records.js68 call sendRecordPDF()
Health Records	AI Analysis	On-demand	Patient request	Patient sends record data to records-analyze.js8
Appointments	Email	Event-driven	Status change	appointments.js60-69 appointments.js111-123
AI Analysis	Health Records	Read-only	Analysis request	AI receives record via request body records-analyze.js10
Doctor-Patient Comm	Email	On-demand	User action	Mail routes call sendMail() utility

13.Role-Based Feature Access

Meditrack enforces role-based access control at both the route handler level and through the auth middleware. The following diagram shows which features are accessible to each role:

Role Enforcement Patterns

The system uses consistent patterns to enforce role-based access:

1. **Explicit Role Checks**: Route handlers check req.user.role before allowing operations
    - Example: backend/routes/records.js11-12 - if (req.user.role !== "doctor") return res.status(403)
2. **Data Ownership Checks**: Patients can only access their own data
    - Example: backend/routes/records.js41-42 - if (req.user.role === "patient" && req.user._id.toString() !== id)
3. **Automatic Filtering**: List endpoints automatically filter by role
    - Example: backend/routes/appointments.js140-146 - Appointments filtered by patient or doctor field based on role

14.Core Features Summary

14.1. Health Records Management

The primary feature of Meditrack, providing CRUD operations for patient health records. Doctors create, update, and delete records containing notes, prescriptions, and vitals. Patients view their records in read-only mode.

Key Implementation:

- Route: backend/routes/records.js
- Model: HealthRecord with
  fields patient, doctor, notes, prescriptions, vitals
- Access Control: Doctor-only for CUD operations, role-based for read
- Auto-triggers: PDF generation on create/update

14.2. AI-Powered Analysis & Chat

Integration with Google Gemini AI to provide health insights and conversational assistance. The system enforces safety constraints to prevent diagnosis and ensure responsible AI use.

Key Implementation:

- Route: backend/routes/records-analyze.js8-34
- API: GoogleGenerativeAI with model gemini-2.5-flash
- Safety Prompt: records-analyze.js16-23 includes "Do not diagnose diseases" and "Give general safe advice"
- Input: Health record JSON structure
- Output: Formatted AI analysis text

14.3. Appointment Scheduling

Workflow for patients to request appointments with doctors and doctors to approve/reject them. Includes conflict detection to prevent double-booking and automatic email notifications.

Key Implementation:

- Route: backend/routes/appointments.js
- Model: Appointment with fields patient, doctor, date, reason, status
- Statuses: requested, accepted, rejected, completed
- Conflict Detection: appointments.js37-47 checks for existing appointments at same time
- Date Validation: appointments.js28-32 blocks past dates

14.4. PDF Generation & Email Notifications

Automatic generation of PDF documents from health records and email delivery to patients. Operates asynchronously using a fire-and-forget pattern to avoid blocking API responses.

Key Implementation:

- Helper: backend/helpers/sendRecordPDF.js6-73
- Libraries: pdfkit for PDF generation, nodemailer for email
- Trigger Points: records.js27 (create), records.js68 (update)
- Email Service: Gmail SMTP configured via process.env.SMTP_USER and SMTP_PASS
- PDF Content: Patient name, doctor name, date, notes, prescriptions, vitals

14.5. Doctor-Patient Communication

Email messaging system enabling bidirectional communication between doctors and patients through the platform.

Key Implementation:

- Routes: /mail and /doctor-mail endpoints
- Utility: utils/mail.sendMail function
- Used by: Appointment notifications, record updates, direct messaging

15.Feature Activation Sequence

When a doctor creates a health record, multiple features activate in sequence:

Step	*Feature*	*Action*	*Code Reference*
1	Records Management	Validate doctor role	records.js11-12
2	Records Management	Create HealthRecord document	records.js16-24
3	Records Management	Call sendRecordPDF(record._id)	records.js27
4	PDF Generation	Fetch record with populated patient/doctor	sendRecordPDF.js7-9
5	PDF Generation	Generate PDF using pdfkit	sendRecordPDF.js16-70
6	Email Notifications	Send via nodemailer	sendRecordPDF.js32-43
7	Records Management	Return response to client	records.js29

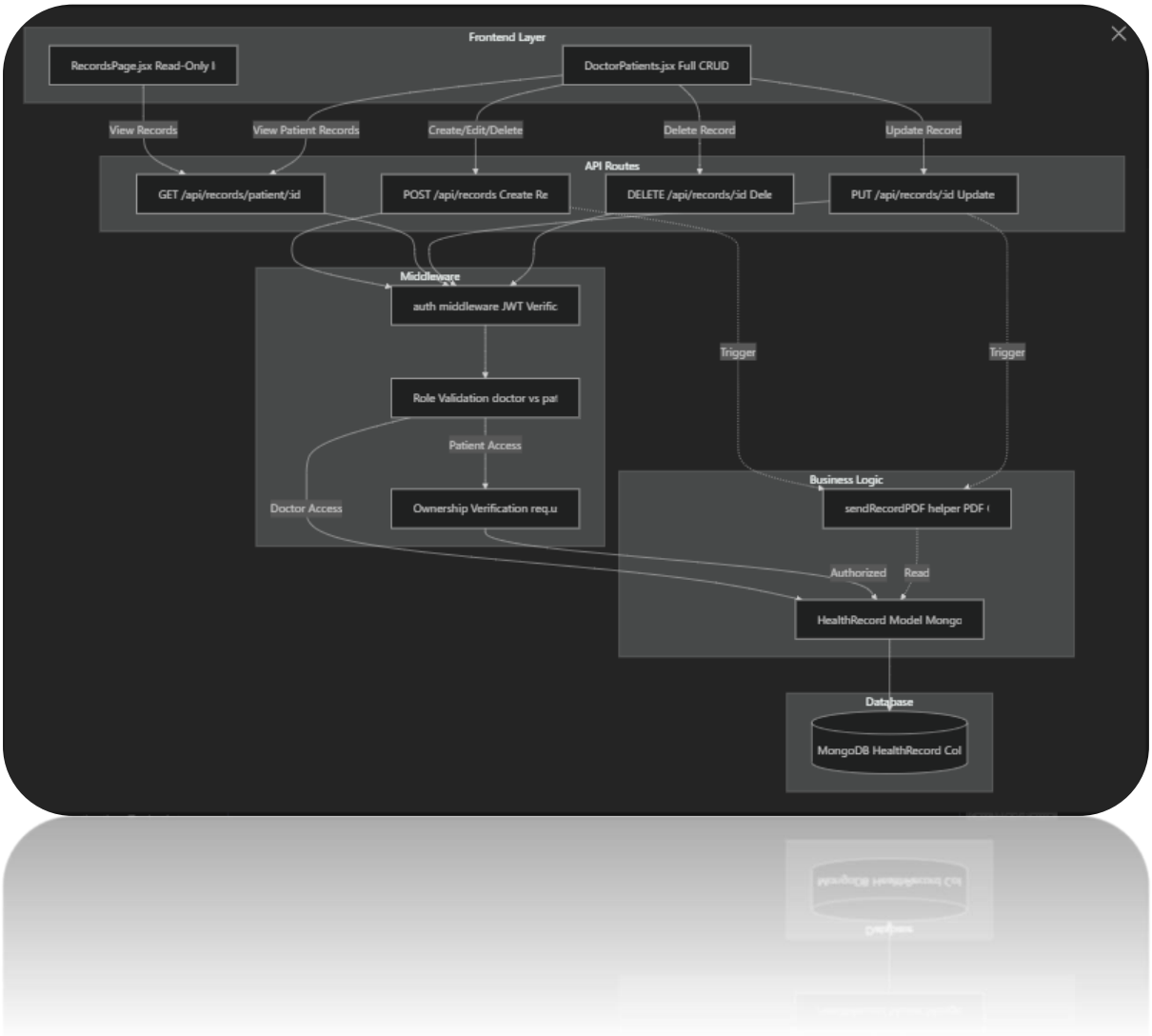
Note that steps 4-6 execute asynchronously after step 3 returns, ensuring API response speed is not blocked by PDF/email operations.

16. Health Records Management

16.1.System Overview

The health records system follows a strict role-based access model where doctors have full CRUD privileges and patients have read-only access to their own records. All record modifications trigger automated PDF generation and email delivery to the patient.

Health Records Architecture



16.2.Data Model

The HealthRecord model stores medical information with references to both the patient and the attending doctor. Each record contains clinical notes, prescriptions, and vital signs.

HealthRecord Schema

Field	*Type*	*Description*	*Required*
patient	ObjectId (ref: User)	Reference to patient user	Yes
doctor	ObjectId (ref: User)	Reference to doctor who created record	No
notes	String	Clinical notes and observations	No
prescriptions	Array of Strings	List of prescribed medications	No
vitals	Object	Patient vital signs (bp, pulse, temp)	No
vitals.bp	String	Blood pressure reading	No
vitals.pulse	String	Pulse rate	No
vitals.temp	String	Body temperature	No
createdAt	Date	Timestamp of record creation	Auto-generated

The schema enforces referential integrity through Mongoose's ref field, enabling population of doctor details when querying records.

17.Access Control

The system implements two-layer authorization: role-based access control (RBAC) and ownership verification.

17.1.Role-Based Permissions

Operation	*Patient*	*Doctor*
Create Record	❌	✅
View Own Records	✅	✅
View Other Patient Records	❌	✅ (via appointments)
Update Record	❌	✅
Delete Record	❌	✅

Patient Access Rules:

- Can only access records where patient field matches their user ID
- Enforced at backend/routes/records.js41-42
- Returns 403 error if attempting to access other patients' records

Doctor Access Rules:

- Can create records for any patient
- Can update any record (no ownership check)
- Can delete any record (no ownership check)

18.CRUD Operations

18.1.Create Record

Endpoint: POST /api/records

Authorization: Doctor only

Request Body:

```
{
 "patientId": "objectId",
 "notes": "Clinical observations",
 "prescriptions": ["Medication A", "Medication B"],
 "vitals": {
  "bp": "120/80",
  "pulse": "72",
  "temp": "98.6"
 }
}
```

Process:

1. Verify user has doctor role at backend/routes/records.js11-12
2. Create new HealthRecord with doctor field set to req.user._id at backend/routes/records.js16-22
3. Save to database at backend/routes/records.js24
4. Trigger PDF generation and email via sendRecordPDF(record._id) at backend/routes/records.js27
5. Return created record

18.2.Read Records

Endpoint: GET /api/records/patient/:id

Authorization:

- Patients: Can only access their own ID
- Doctors: Can access any patient's records

Response:


```
{
  "records": [
    {
      "_id": "recordId",
      "patient": "patientId",
      "doctor": {
        "_id": "doctorId",
        "name": "Dr. Smith",
        "email": "doctor@example.com"
      },
      "notes": "Clinical notes",
      "prescriptions": ["Med 1", "Med 2"],
      "vitals": {
        "bp": "120/80",
        "pulse": "72",
        "temp": "98.6"
      },
      "createdAt": "2024-01-15T10:30:00.000Z"
    }
  ]
}
```

Process:

1. Extract patient ID from URL parameters at backend/routes/records.js39
2. If user is patient,
   verify req.user._id matches :id at backend/routes/records.js41-42
3. Query records with .populate("doctor") to include doctor details
   at backend/routes/records.js44
4. Return records array

18.3.Update Record

Endpoint: PUT /api/records/:id

Authorization: Doctor only

Request Body:

```
{
 "notes": "Updated clinical notes",
 "prescriptions": ["Updated medication list"],
 "vitals": {
  "bp": "118/78",
  "pulse": "70",
  "temp": "98.4"
 }
}
```

Process:

1. Verify user has doctor role at backend/routes/records.js56-57
2. Update record using findByIdAndUpdate with {new: true} option at backend/routes/records.js61-65
3. Trigger PDF generation and email at backend/routes/records.js68
4. Return updated record

Note: No ownership verification - any doctor can update any record.

18.4.Delete Record

Endpoint: DELETE /api/records/:id

Authorization: Doctor only

Process:

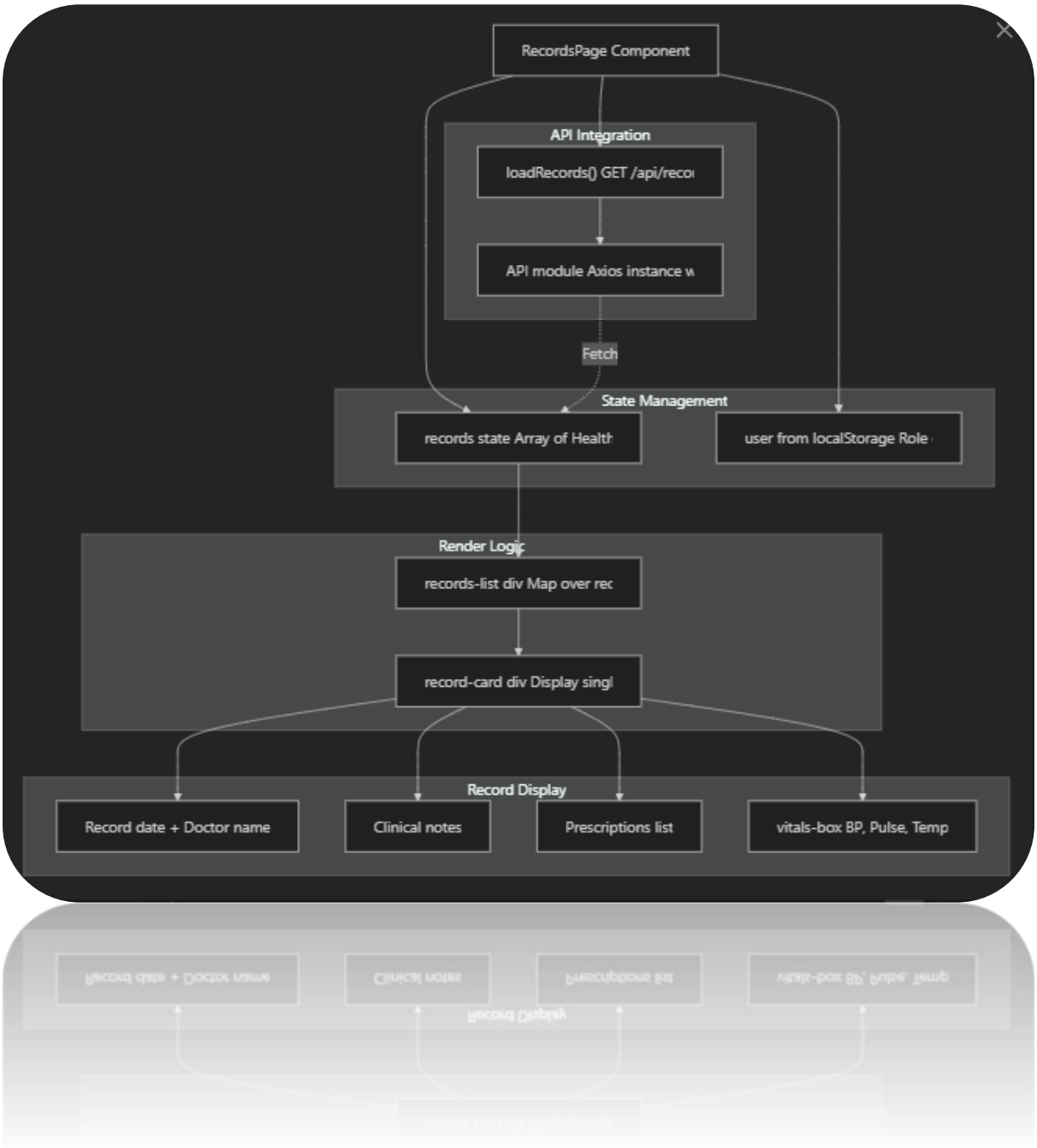
1. Verify user has doctor role at backend/routes/records.js81-82
2. Delete record using findByIdAndDelete at backend/routes/records.js84
3. Return success message

Note: Deletion does not trigger email notification. The record is permanently removed from the database.

19.Patient Records Interface

The patient-facing interface (RecordsPage.jsx) provides a read-only view of health records with minimal interaction capabilities.

19.1.UI Component Structure



19.2.Key Features

Read-Only Access:

- UI conditionally hides create/edit/delete buttons if user.role === "patient" at frontend/src/patient/RecordsPage.jsx12
- Functions openCreate(), openEdit(), saveRecord(), and deleteRecord() include early returns for patients at frontend/src/patient/RecordsPage.jsx32 frontend/src/patient/RecordsPage.jsx40 frontend/src/patient/RecordsPage.jsx62 frontend/src/patient/RecordsPage.jsx84

Data Loading:

- Records loaded on component mount via useEffect at frontend/src/patient/RecordsPage.jsx22-24
- API call uses patient's ID from localStorage at frontend/src/patient/RecordsPage.jsx27

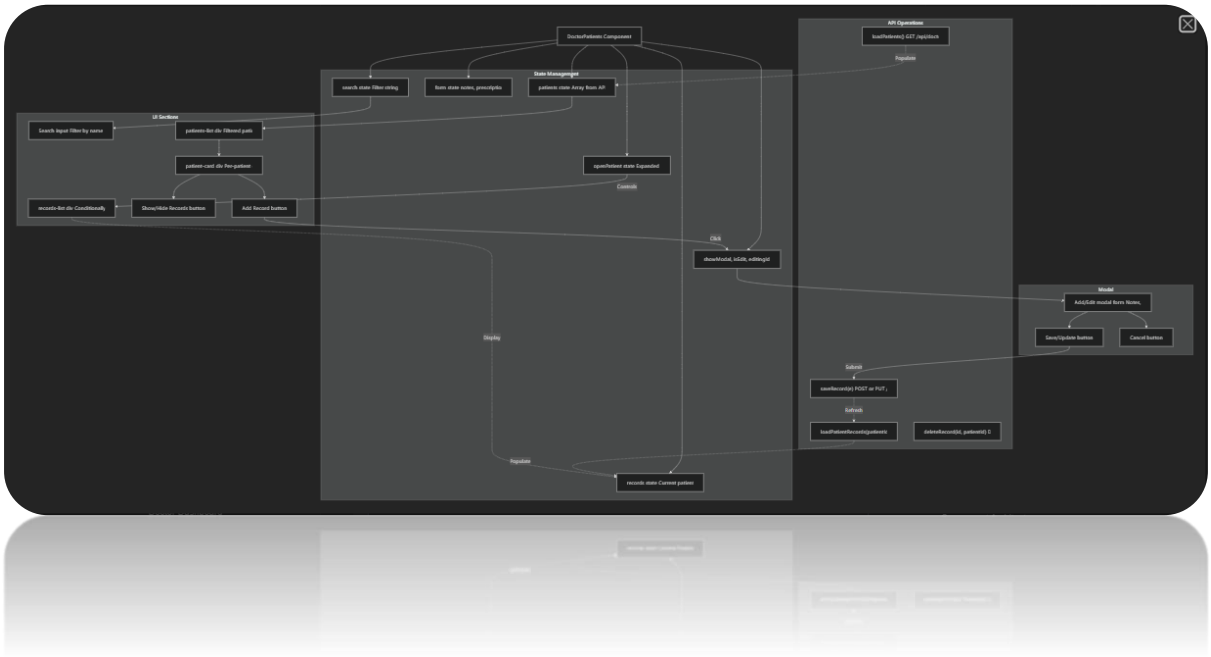
Record Display: Each record card shows:

- Creation date at frontend/src/patient/RecordsPage.jsx108
- Doctor's name (or "Self / Patient" if no doctor) at frontend/src/patient/RecordsPage.jsx118
- Clinical notes at frontend/src/patient/RecordsPage.jsx119
- Prescriptions as comma-separated list at frontend/src/patient/RecordsPage.jsx121-123
- Vitals in a dedicated box at frontend/src/patient/RecordsPage.jsx125-131

20.Doctor Records Management Interface

The doctor interface (DoctorPatients.jsx) provides full CRUD capabilities with a patient-centric workflow.

20.1.Component Architecture



20.2.Key Workflows

Patient List Management:

1. Load patients on mount at frontend/src/doctor/DoctorPatients.jsx157-159
2. Search filter applied at frontend/src/doctor/DoctorPatients.jsx162-164
3. Each patient rendered as expandable card at frontend/src/doctor/DoctorPatients.jsx179-252

Record Creation:

1. Click "Add Record" button triggers openAddRecord(patientId) at frontend/src/doctor/DoctorPatients.jsx57-71
2. Modal opens with empty form
3. Form submission calls saveRecord(e) at frontend/src/doctor/DoctorPatients.jsx91-127
4. Data formatted: prescriptions split by comma at frontend/src/doctor/DoctorPatients.jsx97
5. POST request to /api/records at frontend/src/doctor/DoctorPatients.jsx113-117
6. Records list refreshed at frontend/src/doctor/DoctorPatients.jsx121

Record Editing:

1. Click "Edit" button triggers openEditRecord(rec, patientId) at frontend/src/doctor/DoctorPatients.jsx74-88
2. Form pre-populated with record data at frontend/src/doctor/DoctorPatients.jsx79-85
3. Prescriptions array joined to comma-separated string at frontend/src/doctor/DoctorPatients.jsx81
4. PUT request to /api/records/:id at frontend/src/doctor/DoctorPatients.jsx107-111

Record Deletion:

1. Click "Delete" button triggers deleteRecord(id, patientId) at frontend/src/doctor/DoctorPatients.jsx130-144
2. Confirmation dialog at frontend/src/doctor/DoctorPatients.jsx131
3. DELETE request to /api/records/:id at frontend/src/doctor/DoctorPatients.jsx134-137
4. Records list refreshed at frontend/src/doctor/DoctorPatients.jsx139

Toggle Records View:

- Click "Show/Hide Records" toggles openPatient state at frontend/src/doctor/DoctorPatients.jsx147-155
- When opening, loads patient records via loadPatientRecords(p._id) at frontend/src/doctor/DoctorPatients.jsx153
- Records list conditionally rendered based on openPatient === p._id at frontend/src/doctor/DoctorPatients.jsx207-249

21.Automated PDF and Email Workflow

All record creation and update operations automatically trigger PDF generation and email delivery to the patient.

Workflow Trigger Points

Operation	*Trigger Location*	*Function Called*
Create Record	backend/routes/records.js27	sendRecordPDF(record._id)
Update Record	backend/routes/records.js68	sendRecordPDF(updated._id)
Delete Record	None	No email sent on deletion

Process Flow

The sendRecordPDF helper function orchestrates:

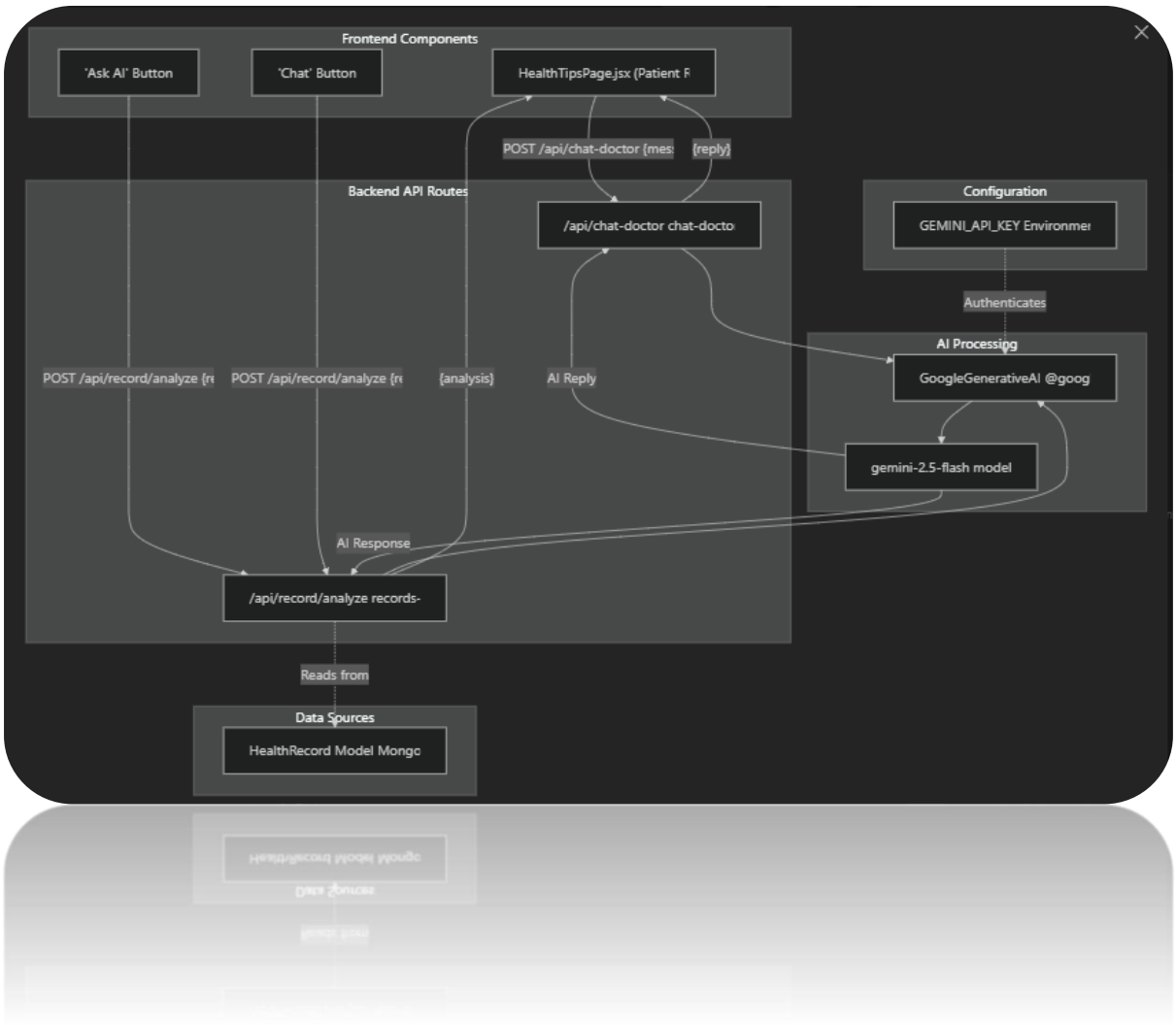
1. Loading the complete health record from database
2. Generating a PDF document with patient and doctor information
3. Sending the PDF as an email attachment to the patient

22.AI Integration

System Architecture

The AI integration consists of two independent API endpoints that share a common architecture pattern: prompt engineering with safety constraints, API calls to Google Generative AI, and response formatting.

AI Integration Points

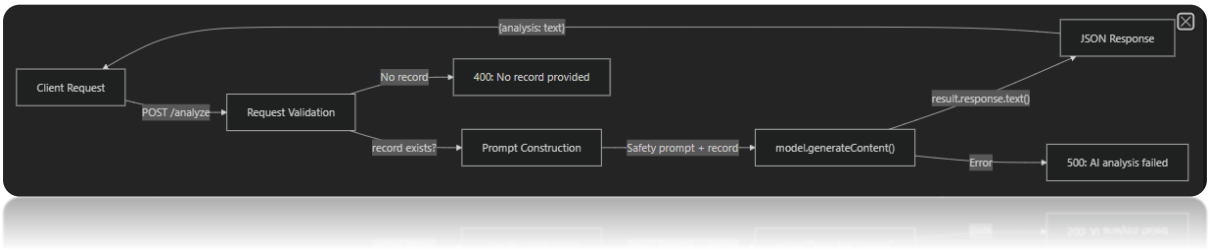


23.Health Record Analysis System

The health record analysis system provides AI-powered insights into patient health records through two interaction modes: one-shot analysis and conversational chat.

23.1.Backend Implementation

The /api/record/analyze endpoint in backend/routes/records-analyze.js handles both analysis modes using a single POST endpoint:



The endpoint accepts two request patterns:

Request Type	*Body Schema*	*Use Case*
Single Analysis	{record: Object}	One-time analysis via "Ask AI" button
Conversational Chat	{record: Object, question: String}	Multi-turn dialogue via "Chat" interface

The implementation at backend/routes/records-analyze.js8-34 instantiates the AI model at module load time:

const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY);
const model = genAI.getGenerativeModel({ model: "gemini-2.5-flash" });

23.2.Safety Prompt Engineering

The system constructs a safety-first prompt that appears at backend/routes/records-analyze.js16-23:

Analyze this health record safely.

Do not diagnose diseases.

Give general safe advice.

Record:

${JSON.stringify(record, null, 2)}

This prompt engineering enforces three critical constraints:

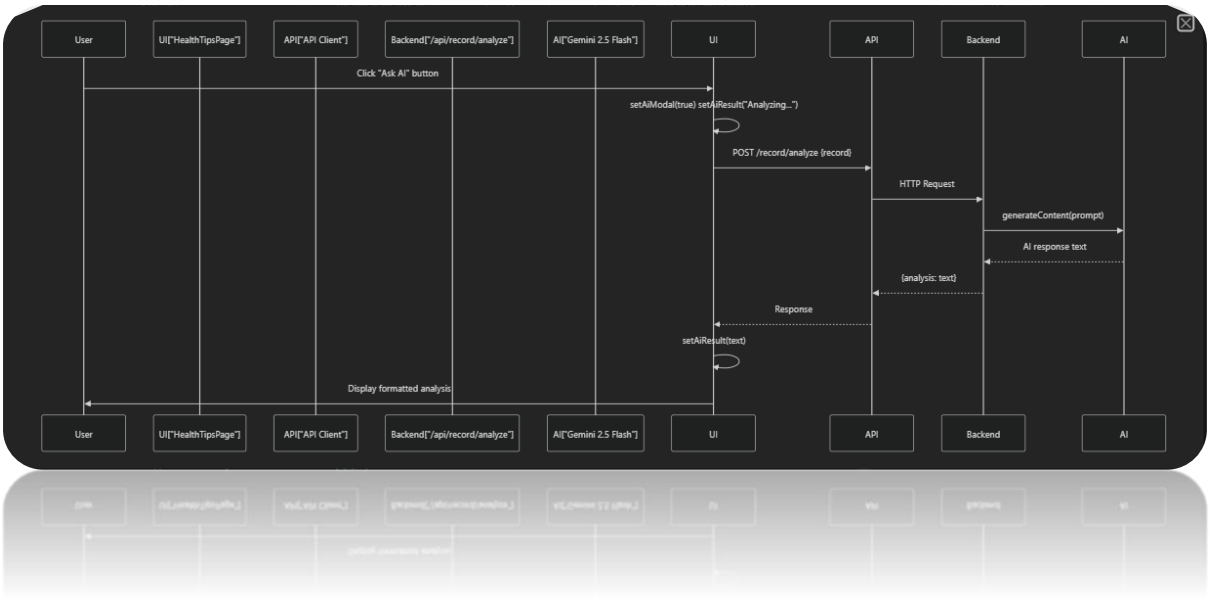
1. **No Diagnosis**: The AI explicitly cannot diagnose medical conditions
2. **General Advice Only**: Responses are limited to safe, general health guidance
3. **Contextual Analysis**: The full record object (including notes, prescriptions, vitals) is provided as JSON

The JSON.stringify() with formatting provides the AI with structured data that includes all fields from the HealthRecord schema.

23.3.Frontend Integration

The patient interface at frontend/src/patient/HealthTipsPage.jsx implements two distinct AI interaction patterns:

One-Shot Analysis ("Ask AI")



The implementation at frontend/src/patient/HealthTipsPage.jsx88-99 opens a modal and displays results:

- Immediate UI feedback: "Analyzing with AI..."
- Single API call to /record/analyze with the record object
- Error handling with fallback message: "AI analysis failed."

23.4.Conversational Chat

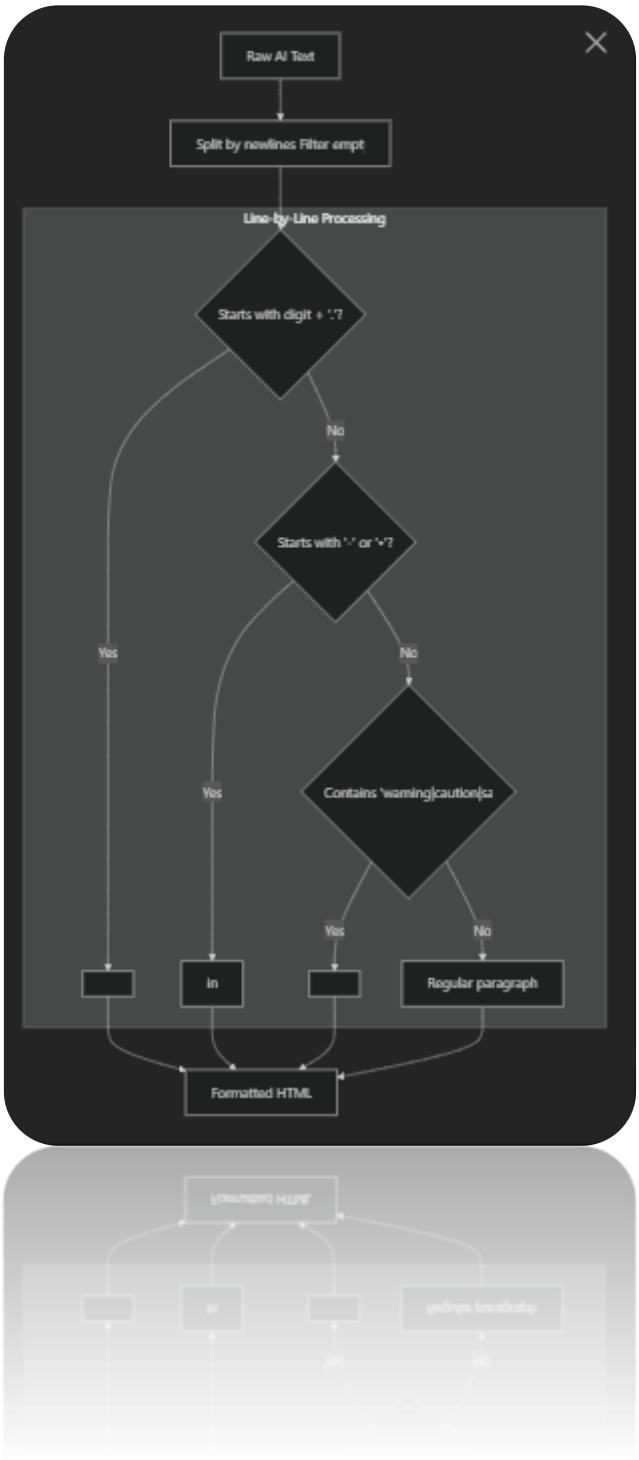
The chat interface at frontend/src/patient/HealthTipsPage.jsx104-127 enables multi-turn conversations about a specific health record:

Step	*Function*	*Location*
Initialize Chat	User clicks "Chat" button	frontend/src/patient/HealthTipsPage.jsx213-223
Display Chat Modal	setChatModal(true) with initial AI greeting	frontend/src/patient/HealthTipsPage.jsx216-219
Send Message	sendChatMessage() appends user message	frontend/src/patient/HealthTipsPage.jsx104-127
Format Response	formatAIText() converts plain text to HTML	frontend/src/patient/HealthTipsPage.jsx48-83

Each message in the chat calls the same /record/analyze endpoint with both the original record and the user's question, maintaining conversation context through the full record object.

23.5.Response Formatting

The formatAIText() function at frontend/src/patient/HealthTipsPage.jsx48-83 transforms AI text into structured HTML with semantic styling:



The formatter implements stateful list handling (inList flag) to properly open and close <ul> tags, ensuring valid HTML structure.

Text Pattern Matching:

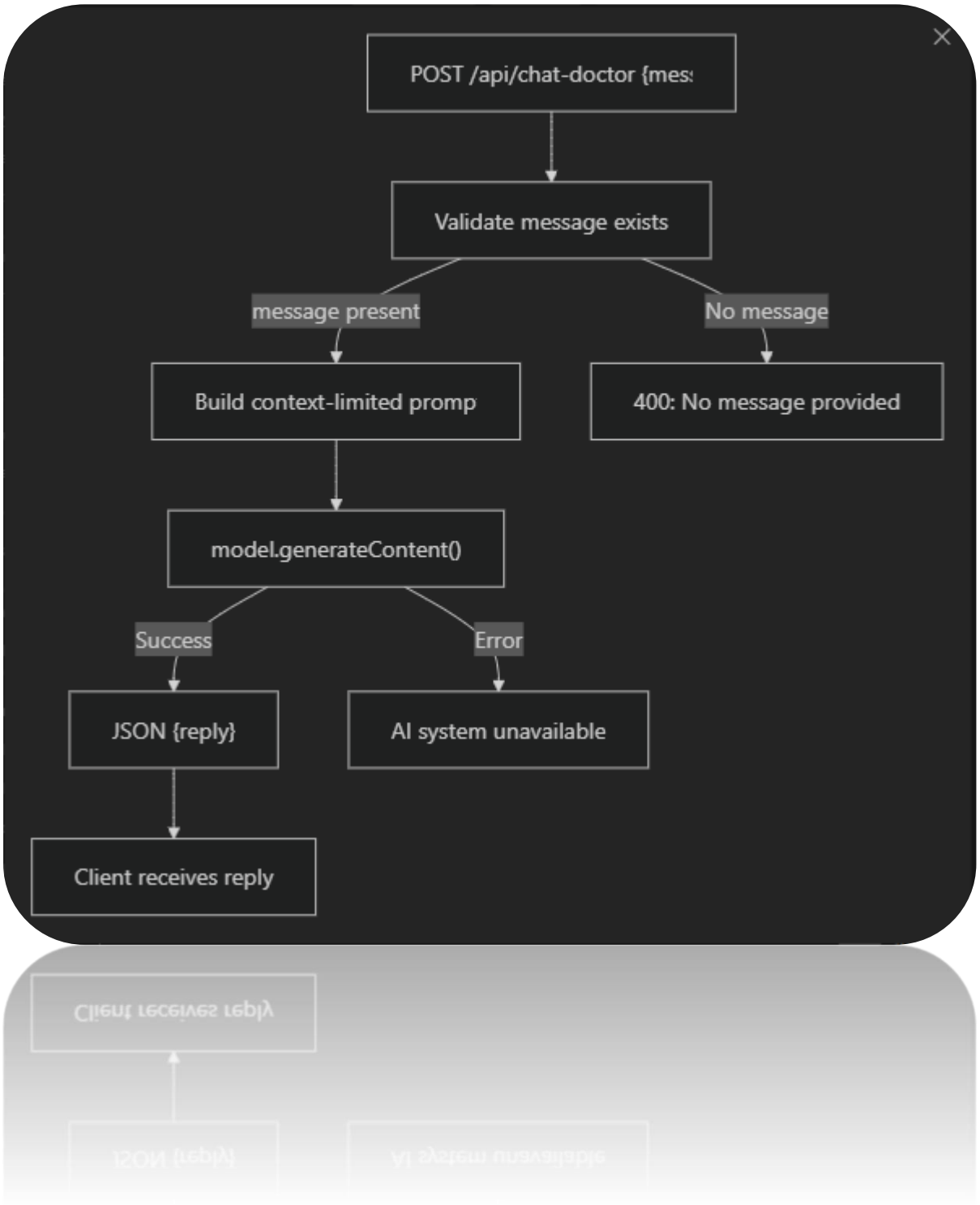
- Numbered headings: /^\d+\./ → <h3 class="chat-heading">
- Bullet points: /^[-•]/ → within <ul class='chat-list'>
- Safety warnings: /warning|caution|safety/i → <p class="chat-warning">
- Default text: <p>

24.Clinical Communication Assistant

The /api/chat-doctor endpoint provides a separate AI assistant focused on clinical communication and health-related queries.

24.1.Backend Implementation

The implementation at backend/routes/chat-doctor.js1-55 follows the same architectural pattern as record analysis but with stricter context limitations:



The model initialization at backend/routes/chat-doctor.js5-6 uses the same Gemini 2.5 Flash model:

const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY)
const model = genAI.getGenerativeModel({ model: "gemini-2.5-flash" })

25.Strict Context Limitation

The prompt at backend/routes/chat-doctor.js17-41 implements a whitelist-based context restriction:

Allowed Topics:

1. Doctor communication
2. Appointments
3. Symptoms
4. Vitals (BP, pulse, fever, etc.)
5. Basic health guidance
6. Medications/prescriptions
7. Medical reports
8. Patient-doctor messages

Refusal Protocol: If the user asks about unrelated topics (jokes, politics, math, stories, coding, movies, random topics), the AI must respond:

"I can only help with health-related communication and doctor assistance."

Additional Constraints:

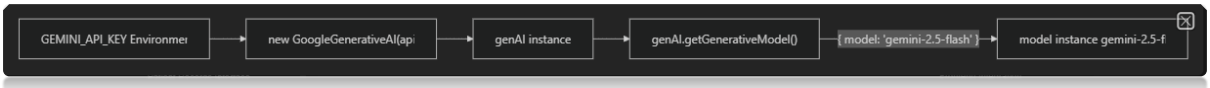
- DO NOT diagnose
- Keep responses short, medical, helpful, and professional

This differs from the record analysis endpoint in that it actively refuses off-topic queries rather than passively ignoring them.

26.AI Model Configuration

Both AI endpoints share a common configuration pattern:

Model Instantiation



Configuration Parameters

Parameter	*Value*	*Location*	*Purpose*
GEMINI_API_KEY	Environment variable	.env file	Authenticates with Google AI
Model name	"gemini-2.5-flash"	backend/routes/records-analyze.js6 backend/routes/chat-doctor.js6	Specifies Gemini model version

The model is instantiated once at module load time (not per-request), allowing Node.js to reuse the same AI client instance across multiple requests.