**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY



**CAPSTONE PROJECT REPORT:**

**CLINIC MANAGEMENT WEB APP**

**Supervised by:**

M.S. Nguyen Duy Hiep

|  |  |
| --- | --- |
| **Name** | **ID** |
| Bach Nhat Minh | 20225509 |

TABLE OF CONTENTS

[Chapter 1: Project Introduction 4](#_Toc202766720)

[1.1 Problem Statement 4](#_Toc202766721)

[1.2 Objectives and Scope 4](#_Toc202766722)

[1.3 Solution Approach 4](#_Toc202766723)

[1.4 Report Outline 4](#_Toc202766724)

[Chapter 3: Technologies Used 5](#_Toc202766725)

[3.1 FastAPI 5](#_Toc202766726)

[3.2 React 5](#_Toc202766727)

[3.3 PostgreSQL 6](#_Toc202766728)

[3.4 SQLAlchemy 6](#_Toc202766729)

[3.5 Pydantic 7](#_Toc202766730)

[3.6 Uvicorn 7](#_Toc202766731)

[3.7 Node.js/npm 7](#_Toc202766732)

[3.8 TypeScript 7](#_Toc202766733)

[3.9 Vercel 8](#_Toc202766734)

[3.10 Render 8](#_Toc202766735)

[3.11 Design Patterns Used 8](#_Toc202766736)

[3.11.1 Repository Pattern 9](#_Toc202766737)

[3.11.2 Dependency Injection 9](#_Toc202766738)

[3.12.3 Model-View-Controller (MVC) / Model-View-ViewModel (MVVM) like (Frontend) 9](#_Toc202766739)

[Chapter 4: Application Development and Deployment 10](#_Toc202766740)

[4.1 Architectural Design 10](#_Toc202766741)

[4.1.1 Software Architecture Selection 10](#_Toc202766742)

[4.1.2 Overview Design 12](#_Toc202766743)

[4.1.3 Detailed Design of Functions / Package-Level Design 12](#_Toc202766744)

[4.2 Detailed Design 14](#_Toc202766745)

[4.2.1 UI Design 14](#_Toc202766746)

[4.2.2 Class Design 15](#_Toc202766747)

[4.3 Implementation 16](#_Toc202766748)

[4.3.1 Libraries and Tools Used 16](#_Toc202766749)

[4.3.2 Demonstration of Main Functions 18](#_Toc202766750)

[4.4 Deployment 18](#_Toc202766751)

[4.4.1 Frontend Deployment (React Application on Vercel) 19](#_Toc202766752)

[4.4.2 Backend Deployment (FastAPI Application on Render) 19](#_Toc202766753)

[4.4.3 Database Deployment (Google Cloud SQL for PostgreSQL) 21](#_Toc202766754)

[4.4.4 Post-Deployment Configuration & Connection 22](#_Toc202766755)

[4.4.5 Auto-Updates from GitHub (CI/CD) 22](#_Toc202766756)

[Chapter 5: Conclusion and Future Development 22](#_Toc202766757)

[5.1 Conclusion 22](#_Toc202766758)

[5.2 Future Development Directions 24](#_Toc202766759)

# Chapter 1: Project Introduction

## 1.1 Problem Statement

Vietnam is facing a growing public health challenge regarding skin cancer. Fsuch as climate change and a large population of outdoor workers contribute to an increasing incidence rate. A critical issue is that skin cancer is often detected at late stages, significantly reducing survival rates. This is compounded by a lack of access to specialized dermatological care, especially in rural areas. There is a clear need for a scalable, accessible, and affordable solution to facilitate early detection and connect patients with medical professionals efficiently.

## 1.2 Objectives and Scope

The primary objective of the **SeekWell** project is to develop a web-based platform that democratizes the early detection of skin cancer. It aims to empower community health workers (referred to as "Cadres"), patients, and doctors with modern technology.The scope of the completed project encompasses the following core functionalities:

Key Objectives:

* AI-Powered Screening: To provide an AI-driven tool that can analyze images of skin lesions and provide a preliminary risk assessment to assist healthcare workers.
* Bridging Care Gaps: To create a streamlined workflow for referring potential cases from community health workers to specialized doctors.
* Centralized Health Records: To manage patient information and skin lesion history in a secure and centralized system.
* Community and Support: To foster a community space where users can share experiences and find support.
* Efficient Consultation: To facilitate appointment booking and communication between patients and doctors.

**Scope:** The project encompasses the entire backend infrastructure for a multi-user platform with four distinct roles: **Patient**, **Doctor**, **Cadre**, and **Admin**. The system's scope includes user authentication, profile management, AI-assisted lesion analysis, appointment scheduling, a community forum, and direct messaging capabilities.

## 1.3 Solution Approach

The solution is a robust web application built with a modern, decoupled architecture. A **FastAPI** backend provides a high-performance RESTful API, while a **React** frontend (to be developed) will offer a responsive and intuitive user interface.

The core of the solution is the integration of an AI service. Community Cadres can upload images of a patient's skin lesion through the app. The backend processes this image, sends it to a fine-tuned Vision Transformer (ViT) model hosted on Hugging Face Spaces for analysis, and returns a risk assessment. This data, along with the patient's case file, is then made available to doctors for review, diagnosis, and follow-up, creating an end-to-end chain of care.

## 1.4 Report Outline

**Chapter 2** outlines the functional and non-functional requirements.

**Chapter 3** describes the technology stack and design patterns.

**Chapter 4** provides a deep dive into the system's architecture, database schema, backend API, and AI model development.

**Chapter 5** concludes the report and suggests directions for future development.

# Chapter 2: Requirements Analysis

#### 2.1 Functional Overview

The SeekWell platform is designed to serve four main user roles with specific functionalities tailored to their needs.

* **Admin:** Manages the entire platform, including user accounts and system settings.
* **Cadre (Community Health Worker):** Acts as the frontline user, conducting initial screenings in the community.
* **Doctor:** A registered medical professional who reviews cases, provides diagnoses, and consults with patients.
* **Patient:** The end-user whose health is being monitored.

#### System Actors and Use Cases

### 2.2.1 Cadre Use Cases:

* Register and log into the system.
* Create and manage patient profiles under their care.
* Initiate a new skin lesion case by uploading an image through the AISkinAnalysisDashboard.
* Receive a preliminary analysis from the integrated AI service.
* View the history of lesions for their patients in the AnalysisHistory tab.
* Refer a patient's case to a registered doctor for further review.

### 2.2.2 Patient Use Cases:

* Register and log into the system.
* View their personal profile and medical history, including all lesion analyses.
* Book appointments with doctors, with an urgent pathway for high-risk AI results.
* Communicate with their assigned doctor via a secure chat.
* Participate in the community forum by creating posts and commenting.
* Reset their password.

### 2.2.3 Doctor Use Cases:

* Register and log into the system.
* View a dashboard of patients referred to them.
* Review patient lesion images, AI analysis results (AnalysisResults), and history.
* Submit a formal diagnosis for a lesion.
* Manage their appointment schedule.
* Communicate with their patients via a secure chat.

### 2.2.4 Admin Use Cases:

* Manage all user accounts (activate, deactivate, assign roles).
* View system-wide reports and analytics.
* Oversee all data within the platform.

# Chapter 3: Technologies Used

This chapter details the key technologies, frameworks, libraries, and design patterns that were instrumental in the successful development and completion of the Clinic Management System.

## 3.1 FastAPI

**Description:** FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+, leveraging standard Python type hints.

**Reasons for Choosing & Advantages:** Selected for its exceptional speed, ease of development, automatic data validation and serialization (via Pydantic integration), automatic interactive API documentation (Swagger UI and ReDoc), and robust dependency injection system. Its asynchronous capabilities (async/await) were crucial for building a responsive and scalable backend.

**Community Support:** Benefits from a large, active community and comprehensive documentation, ensuring ample resources and support.

## 3.2 React

**Description:** React is a declarative, efficient, and flexible JavaScript library for building user interfaces, particularly single-page applications.

**Reasons for Choosing & Advantages:** Chosen for its component-based architecture, which promotes code reusability, modularity, and easier maintenance of the frontend. The virtual DOM ensures efficient UI updates, leading to a smooth and responsive user experience. The vast ecosystem of libraries (e.g., React Router, Axios) and strong community support were also key factors.

**Relevant History:** Developed and actively maintained by Meta (formerly Facebook) and a large community of individual developers and companies.

## 3.3 PostgreSQL

**Description:** PostgreSQL is a powerful, open-source object-relational database system known for its reliability, feature robustness, and performance.

**Reasons for Choosing & Advantages:** Selected for its ACID compliance, strong support for complex queries and transactions, extensibility, and a wide array of data types suitable for healthcare data. Its proven stability and scalability make it an excellent choice for managing sensitive patient information.

**Drawbacks:** While powerful, initial setup and advanced administration can be more complex compared to simpler database systems, though this was managed effectively for the project's needs.

## 3.4 SQLAlchemy

**Description:** SQLAlchemy is a comprehensive Python SQL toolkit and Object Relational Mapper (ORM).

**Reasons for Choosing & Advantages:** Utilized to provide an abstraction layer over SQL, allowing developers to interact with the PostgreSQL database using Python objects and methods. This simplified data access logic (CRUD operations), improved code readability, and helped prevent SQL injection vulnerabilities. Its flexibility supports both ORM patterns and raw SQL execution when needed.

**Community Support:** A mature and widely-used library with extensive documentation and a strong community.

## 3.5 Pydantic

**Description:** Pydantic is a data validation and settings management library that uses Python type hints to validate data.

**Reasons for Choosing & Advantages:** Integral to FastAPI, Pydantic was used for defining clear data schemas for API request and response bodies. This enabled automatic data validation, serialization, and documentation, significantly reducing boilerplate code and improving data integrity at the API boundaries.

**Relevant History:** Has become a standard for data validation in modern Python web frameworks.

## 3.6 Uvicorn

**Description:** Uvicorn is an ASGI (Asynchronous Server Gateway Interface) server, built using uvloop and httptools for high performance.

**Reasons for Choosing & Advantages:** Chosen as the ASGI server to run the FastAPI application, enabling its asynchronous capabilities and ensuring efficient handling of concurrent requests.

## 3.7 Node.js/npm

**Description:** Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. npm (Node Package Manager) is the default package manager for Node.js.

**Reasons for Choosing & Advantages:** Essential for the frontend development workflow. Used to manage React project dependencies, run the development server (react-scripts), build the production version of the frontend application, and utilize a vast ecosystem of JavaScript libraries.

## 3.8 TypeScript

**Description:** TypeScript is an open-source language which builds on JavaScript, one of the world’s most used tools, by adding static type definitions.

**Reasons for Choosing & Advantages:** Adopted for the frontend React application to enhance code quality, maintainability, and developer productivity. Static typing helps catch errors early in the development process, improves code readability, and provides better autocompletion and refactoring capabilities in IDEs.

## 3.9 Vercel

**Description:** Vercel is a cloud platform for static sites and Serverless Functions that enables developers to host web projects with high performance and easy deployment.

**Reasons for Choosing & Advantages:** Chosen for deploying the frontend React application due to its seamless integration with GitHub for continuous deployment (CI/CD). Vercel automatically detects Create React App projects, simplifying configuration. It provides features like SSL, a global CDN, and easy environment variable management, making it ideal for hosting modern frontend applications like the one developed for this project.

## 3.10 Render

**Description:** Render is a unified cloud platform to build and run applications and websites. It supports deploying web services, static sites, databases, and more.

**Reasons for Choosing & Advantages:** Selected for deploying the backend FastAPI application. Render offers straightforward GitHub integration for CI/CD, simplifying the deployment process for Python applications. It supports specifying root directories, runtimes, build commands (e.g., pip install -r requirements.txt), and start commands (e.g., for Uvicorn). Render also provides easy management of environment variables, crucial for database connections and API keys.

## 3.11 Design Patterns Used

Design patterns are general, reusable solutions to commonly occurring problems within a given context in software design. The following patterns were employed in this project:

### 3.11.1 Repository Pattern

**Explanation:** This pattern mediates between the domain logic and data mapping layers (database). It provides a collection-like interface for accessing domain objects, abstracting the underlying data storage and retrieval mechanisms.

**How and why it's used in the project:** Implemented in the backend/app/crud.py module. Functions like create\_user, get\_patient\_by\_user\_id, update\_appointment, etc., encapsulate the SQLAlchemy database interactions for their respective entities. This decouples the API endpoint logic (in routers/) from the specifics of database operations, making the business logic cleaner, more testable, and easier to modify if the data source changes. It centralizes data access logic, improving maintainability.

### 3.11.2 Dependency Injection

**Explanation:** A design pattern where an object or function receives other objects or functions (dependencies) that it needs, rather than creating them internally.

**How and why it's used in the project:** FastAPI heavily utilizes dependency injection. This is evident in API route handlers, such as db: Session = Depends(get\_db) and current\_user: models.User = Depends(get\_current\_active\_user). FastAPI manages the creation and provision of these dependencies (like database sessions and authenticated user objects), making the route functions more focused on their specific tasks, easier to test (by mocking dependencies), and promoting cleaner code.

### 3.12.3 Model-View-Controller (MVC) / Model-View-ViewModel (MVVM) like (Frontend)

**Explanation:** Architectural patterns that separate application concerns. The View is responsible for the UI, the Model manages the data and business logic, and the Controller/ViewModel acts as an intermediary.

**How and why it's used in the project:** The React frontend application (frontend/src/) adheres to principles similar to MVC/MVVM:

**Model:** Represents the application data, often fetched from the backend API and managed within React component state (e.g., using useState, useEffect hooks to manage patient lists, appointment details, chat messages in components like DashboardPage.tsx).

**View:** Comprises the React components that render the user interface based on the current state and props (e.g., LoginPage.tsx, Sidebar.tsx, PatientDashboard.tsx).

**Controller/ViewModel Logic:** Resides within the React components themselves (or custom hooks/services). This logic handles user interactions (e.g., button clicks, form submissions), makes API calls (using Axios) to fetch or update data, and updates the component's state, triggering UI re-renders. This separation improves code organization, testability, and reusability of UI components.

# Chapter 4: Application Development and Deployment

This chapter outlines the architectural design, detailed design specifications, implementation process, and deployment strategy for the completed Clinic Management System.

## 4.1 Architectural Design

### 4.1.1 Software Architecture Selection

The project successfully implemented a Client-Server Architecture. This architecture distinctly separates the user interface (client-side) from the business logic and data management (server-side).

* **Client (Frontend):** A Single Page Application (SPA) developed using React and TypeScript. It is responsible for rendering the user interface, handling user interactions, and communicating with the backend API.
* **Server (Backend):** A RESTful API developed using Python with the FastAPI framework. It handles business logic, data processing, database interactions, and AI service integration.

Within the backend, a **Layered Architecture** was adopted to further promote modularity and separation of concerns:

* **Presentation Layer (API Routers):** Located in backend/app/routers/. This layer is responsible for handling incoming HTTP requests, validating request data using Pydantic schemas, invoking appropriate business logic, and formatting HTTP responses.
* **Service/Business Logic Layer:** Primarily encapsulated within backend/app/crud.py and specific logic within router files or dedicated service modules. This layer contains the core application logic, orchestrates data operations, and implements business rules.
* **Data Access Layer (DAL):** Implemented using SQLAlchemy ORM (backend/app/models.py, backend/app/database.py). This layer abstracts the database interactions, allowing the application to work with Python objects instead of raw SQL queries, and manages database connections and sessions.
* **Database Layer:** PostgreSQL serves as the persistent data store for all application data.

**Advantages of this architecture:**

* **Separation of Concerns:** Frontend and backend concerns are clearly separated, allowing for independent development, testing, and deployment.
* **Scalability:** Both frontend and backend can be scaled independently based on demand.
* **Technology Flexibility:** Different technologies can be used for frontend and backend.
* **Maintainability:** The layered approach within the backend makes the codebase easier to understand, modify, and maintain.

### 4.1.2 Overview Design

The system is composed of two primary deployable units: the frontend React application and the backend FastAPI application.

**Backend Component:** The backend consists of several key modules:

* **API Routers (auth, users, patients, chat):** Define API endpoints.
* **CRUD Module:** Centralizes data access operations.
* **Models Module:** Defines SQLAlchemy ORM classes.
* **Schemas Module:** Defines Pydantic data validation models.
* **Database Module:** Manages database connections and sessions.
* **Config Module:** Handles application settings.
* **Dependencies Module:** Provides reusable authentication/authorization logic.
* **AI Integration (within chat router):** Connects to Google Gemini for chatbot functionality.

**Frontend Package:** The frontend is structured with:

* **Components (components/):** Reusable UI elements (e.g., forms, layout elements, specific feature components).
* **Pages (pages/):** Top-level components representing different views/screens of the application.
* **Services/API Layer (implicit):** Axios instances and functions for making API calls to the backend.
* **Routing:** Managed by react-router-dom for navigation within the SPA.
* **State Management:** Primarily using React's built-in state and context, or potentially a dedicated state management library if complexity grew.
* **Types (types/):** TypeScript definitions for data structures.

### 4.1.3 Detailed Design of Functions / Package-Level Design

**User Authentication & Authorization (backend/app/routers/auth.py, dependencies.py):**

* Handles user registration, password hashing, JWT token generation upon successful login, and token validation for protected routes.
* Dependencies define access control based on user roles (ADMIN, DOCTOR, PATIENT, CLINIC\_STAFF).

**Patient Management (backend/app/routers/patients.py, crud.py, models.py, schemas.py):**

* Defines API endpoints for creating, reading, updating, and deleting patient records.
* Includes logic for assigning doctors to patients and updating EMR summaries.
* SQLAlchemy models define the Patient table and its relationships. Pydantic schemas validate patient data.

**Appointment Management (Conceptual, based on proposal):**

* API endpoints (likely in a dedicated appointments.py router) allow for creating, viewing (by patient, doctor, or all for staff/admin), updating, and canceling appointments.
* The Appointment model in models.py stores appointment details and links to Patient and Doctor models.

**EMR Management (Integrated into patients.py, chat.py, crud.py):**

* Doctors can create/update EMRs via specific API endpoints.
* The MedicalReport model likely stores detailed consultation notes, diagnoses, and links to prescriptions.
* The chat.py router also interacts with EMR by logging chat notes.

**Chatbot Functionality (backend/app/routers/chat.py):**

* Receives messages from the frontend.
* Constructs prompts for the Google Gemini API, incorporating patient EMR data (if available and authorized) and the user's query.
* Sends requests to Gemini and returns the AI's response to the frontend.
* Logs relevant chat interactions or notes into the patient's EMR.

**Frontend UI Rendering and Interaction (frontend/src/):**

* React components are responsible for rendering different views based on application state and user role.
* User interactions trigger state updates and API calls to the backend.
* DashboardPage.tsx is a key component for staff, integrating patient listing, EMR viewing, and the AI chat interface.

## 4.2 Detailed Design

### 4.2.1 UI Design

The user interface was designed to be clean, intuitive, and role-specific, ensuring ease of use for all user types.

**Screen Dimensions & Responsiveness:** The application is designed as a responsive web application, primarily targeting desktop and tablet use, ensuring usability across various screen sizes.

**Color Palette and Typography:** A professional and calming color palette was used, primarily featuring blues, grays, and whites, with accent colors (green for success, red for errors) for important actions and feedback. Typography prioritizes readability with clear sans-serif fonts (system defaults like Segoe UI, Roboto, Helvetica Neue).

**Common UI Element Styling:**

* **Forms:** Consistently styled input fields, labels, and buttons with clear visual hierarchy.
* **Navigation:** A persistent sidebar for authenticated users (Sidebar.tsx), providing role-based navigation links. A top header (BaseDashboard.tsx) includes branding and user account options.
* **Data Display:** Tables and lists are used for displaying patient lists, appointment schedules, and medical records, with clear formatting.
* **Modals/Dialogs:** Used for confirmations, detailed views, or form inputs where appropriate.

**Feedback Mechanisms:** Visual feedback is provided through toast notifications for actions like successful login/registration or data saving. Loading indicators (spinners or text) are used during data fetching or processing. Form validation messages are displayed inline.

**Key UI Screens (Screenshots would typically be included here):**

**Login Page:** A clean interface with fields for email and password, and a link to the registration page.

**Registration Page:** Form for new patients to create an account.

* **Patient Dashboard:** (For Patients) View upcoming appointments, access limited medical records (prescriptions, appointment history), manage profile, and interact with the chatbot.
* **Doctor Dashboard:** (For Doctors) View appointment schedule, access patient EMRs, tools for creating/updating consultation notes, diagnoses, and prescriptions.
* **Staff Dashboard:** (For Clinic Staff) View and manage patient lists, patient details including EMR summaries, manage clinic appointment schedule, and use the AI symptom checker chat.
* **Admin Dashboard:** (For Administrators) User management interface, system settings, and potentially reporting tools.
* **Chat Interface:** A dedicated modal for interacting with the AI chatbot, displaying conversation history.

### 4.2.2 Class Design

The primary classes in the system are represented by the SQLAlchemy ORM models (backend/app/models.py) and Pydantic schemas (backend/app/schemas.py).

**Backend Data Models (SQLAlchemy - models.py):**

* User: Represents all users with attributes like user\_id, username, email, hashed\_password, role, full\_name.
* Patient: Linked to User, stores patient-specific details like date\_of\_birth, gender, address, phone\_number, assigned\_doctor\_id, and EMR summary (though detailed EMRs might be in MedicalReport).
* Doctor: Linked to User, stores doctor-specific details like major, hospital\_id.
* Hospital: Stores hospital information.
* Appointment: Stores appointment details, linking Patient and Doctor, including appointment\_day, appointment\_time, reason.
* MedicalReport: Stores detailed consultation records, diagnoses, treatment plans, linked to Patient and Doctor.
* ChatMessage: Stores messages for the chat functionality, linked to User.

**Backend Data Schemas (Pydantic - schemas.py):** These define the structure for API request and response data, ensuring validation. Examples:

* UserCreate, UserSchema, UserUpdate
* PatientCreate, PatientSchema, PatientUpdate, PatientEMRUpdate
* AppointmentCreate, AppointmentSchema, AppointmentUpdate
* Token, TokenData
* ChatMessageCreate, ChatResponse
* MedicalReportCreate, MedicalReportSchema, MedicalReportUpdate

**Frontend Component Structure (React - frontend/src/):**

* Container components (in pages/) manage state and logic for specific views.
* Presentational components (in components/) are responsible for rendering UI elements and receiving data via props.
* Service/utility functions handle API calls and data transformations.

4.2.3 Database Design

## 4.3 Implementation

### 4.3.1 Libraries and Tools Used

**Backend (requirements.txt):**

|  |  |
| --- | --- |
| **Purpose** | **Tool Name/Version** |
| Web Framework | FastAPI (version 0.115.12) |
| ORM | SQLAlchemy (version 2.0.41) |
| Database Driver (PostgreSQL) | psycopg2-binary (version 2.9.10) |
| Data Validation | Pydantic (version 2.11.5) |
| ASGI Server | Uvicorn (version 0.34.2) |
| Password Hashing | passlib, bcrypt |
| JWT Handling | python-jose |
| Environment Variables | python-dotenv |
| AI SDK | google-generativeai(version 0.8.5) |

**Frontend (frontend/package.json):**

|  |  |
| --- | --- |
| **Purpose** | **Tool Name/Version** |
| UI Library | React (version 19.1.0) |
| HTTP Client | Axios (version 1.9.0) |
| Routing | react-router-dom (version 7.6.0) |
| Language | TypeScript (version 4.9.5) |
| Build/Dev Tool | react-scripts (version 5.0.1) |
| Notification Library | react-toastify (version 11.0.5) |
| Markdown Rendering | react-markdown (version 10.1.0) |

**Development Tools:**

* **IDE:** Visual Studio Code (assumed common choice).
* **Version Control:** Git & GitHub (repository bnmbanhmi/clinic-management).

**Database Management:** PostgreSQL server, pgAdmin or DataGrip.

### 4.3.2 Demonstration of Main Functions

**User Registration & Login:**

**Role-Based Dashboards:**

## 4.4 Deployment

The Clinic Management System has been successfully deployed, with the frontend hosted on Vercel and the backend on Render, utilizing a Google Cloud SQL for PostgreSQL instance for data persistence. Both services are configured for auto-deployment from the GitHub repository.

**Application Structure:**

* **Frontend:** React application located in the frontend directory.

**Live URL:** clinic-management-nine-lime.vercel.app

* **Backend:** FastAPI (Python) application located in the backend directory.

**Live URL:** clinic-management-be3h.onrender.com

* **Database:** Google Cloud SQL for PostgreSQL.
* **Source Code:** github.com/bnmbanhmi/clinic-management

### 4.4.1 Frontend Deployment (React Application on Vercel)

The React-based frontend application is deployed and hosted on Vercel, leveraging its seamless integration with GitHub for continuous deployment.

1. **Vercel Project Setup:**

A Vercel project was created and connected to the GitHub repository: <https://github.com/bnmbanhmi/clinic-management>. This connection enables automatic builds and deployments upon code changes to the specified branch.

1. **Project Configuration (via Vercel UI):**

* **Framework Preset:** Vercel auto-detected the project as a "Create React App."
* **Root Directory:** Configured to frontend, ensuring Vercel uses the correct directory for build and deployment processes.
* **Build Command & Output Directory:** Standard Create React App settings were used (npm run build command, and frontend/build as the output directory).

1. **Environment Variables (Frontend Project on Vercel UI):**

REACT\_APP\_BACKEND\_URL: Set to the live backend URL (clinic-management-be3h.onrender.com) to enable communication between the frontend and backend services.

1. **Deployment & URL:**

Vercel automatically deploys changes pushed to the main branch (or the designated production branch) of the connected GitHub repository.

1. **Live Frontend URL:** The deployed frontend is accessible at clinic-management-nine-lime.vercel.app.

### 4.4.2 Backend Deployment (FastAPI Application on Render)

The FastAPI backend application is deployed as a web service on Render, which also supports continuous deployment from GitHub.

1. **Render Project Setup:**

A "Web Service" was created on Render and linked to the same GitHub repository: github.com/bnmbanhmi/clinic-management.

1. **Service Configuration (via Render UI):**

* **Root Directory:** Set to backend to specify the location of the backend application code.
* **Runtime:** Python, automatically detected by Render due to the presence of backend/requirements.txt.
* **Build Command:** pip install -r requirements.txt (executed relative to the backend directory) to install all necessary Python dependencies.
* **Start Command:** uvicorn app.main:app --host 0.0.0.0 --port $PORT --workers 1 to run the FastAPI application using Uvicorn, making it accessible on the port assigned by Render.

1. **Backend Dependencies (backend/requirements.txt):**

A comprehensive requirements.txt file located at backend/requirements.txt lists all Python dependencies for the backend, critically including psycopg2-binary for PostgreSQL database connectivity.

1. **Environment Variables (Backend Service on Render UI):**

* DATABASE\_URL: The complete PostgreSQL connection string for the Google Cloud SQL database. Format: [postgresql://db\_user:db\_password@34.67.156.97:5432/db\_name](mailto:postgresql://db_user:db_password@34.67.156.97:5432/db_name).
* SECRET\_KEY, ALGORITHM, ACCESS\_TOKEN\_EXPIRE\_MINUTES: Variables critical for JWT authentication and security.
* FRONTEND\_URL: Set to the live frontend URL (clinic-management-nine-lime.vercel.app) for CORS configuration and potentially other backend-to-frontend communications.
* MAIL\_USERNAME, MAIL\_PASSWORD, MAIL\_FROM, etc.: Configuration for email services (if implemented).
* GEMINI\_API\_KEY: API key for accessing the Google Gemini service for the chatbot functionality.
* PYTHON\_VERSION: Specified if a particular Python version is necessary for the application.

1. **Deployment & URL:**

Render automatically deploys new versions of the backend when changes are pushed to the main branch of the GitHub repository.

1. **Live Backend URL:** The deployed backend API is accessible at clinic-management-be3h.onrender.com.

### 4.4.3 Database Deployment (Google Cloud SQL for PostgreSQL)

The application relies on a Google Cloud SQL for PostgreSQL instance for persistent data storage.

**Instance Configuration:**

* **Public IP Address:** 34.67.156.97 (used in the DATABASE\_URL for the backend).
* **Authorized Networks:** Currently configured to allow access from any IP address (0.0.0.0/0) for ease of development and connection from Render. *For enhanced security in a production environment, this should be restricted to Render's outbound IP addresses or specific necessary ranges.*
* **Hardware:** 1 vCPU, 614.4 MB Memory, 10 GB HDD Storage (Auto storage increase enabled).
* **Edition & Version:** Enterprise edition, PostgreSQL 17.5.
* **Backup & Recovery:**

Automated backups are enabled.

Point-in-time recovery is disabled.

Instance deletion prevention is enabled.

Backup retention after deletion is disabled.

* **Location & Availability:** The instance is located in us-central1-f and is configured as a zonal (not highly available) instance.

**Network Configuration:** The Render backend service connects to this Cloud SQL instance using its public IP address.

**SSL/TLS:** Google Cloud SQL provides SSL certificates. The backend application (via psycopg2) should be configured to use secure connections, potentially by appending ?sslmode=require or similar parameters to the DATABASE\_URL if not handled by default by the driver or Cloud SQL proxy.

### 4.4.4 Post-Deployment Configuration & Connection

1. **Frontend to Backend Link:** The REACT\_APP\_BACKEND\_URL environment variable on Vercel correctly points the frontend to the live backend URL (clinic-management-be3h.onrender.com).
2. **Backend to Frontend Link:** The FRONTEND\_URL environment variable on Render ensures the backend is aware of the frontend's live URL (clinic-management-nine-lime.vercel.app), primarily for Cross-Origin Resource Sharing (CORS) configuration.
3. **CORS Configuration:** The FastAPI application's CORS middleware is configured to allow requests from the frontend. For development, the origins list was set to allow all origins.  
   *In a production setting, this list should be restricted to the specific frontend URL (clinic-management-nine-lime.vercel.app) for security.*

### 4.4.5 Auto-Updates from GitHub (CI/CD)

Both Vercel (monitoring the frontend directory) and Render (monitoring the backend directory) are configured for Continuous Integration/Continuous Deployment (CI/CD).

Any push to the main branch of the <https://github.com/bnmbanhmi/clinic-management> repository that includes changes within these respective directories will automatically trigger new builds and deployments on Vercel and Render. This ensures that the live applications are always up-to-date with the latest stable code.

# Chapter 5: Conclusion and Future Development

## 5.1 Conclusion

The Clinic Management System project has culminated in the successful delivery of a comprehensive and modern web application, addressing all core requirements initially proposed. This system effectively modernizes the client's previous operations by introducing robust functionalities, including patient management, online appointment booking, electronic medical record (EMR) management, and an innovative AI-powered chatbot for user support.

The chosen client-server architecture, featuring a React frontend and a FastAPI backend, has proven highly effective. This structure ensures a clear separation of concerns, allows for independent scalability of components, and promotes a maintainable codebase. The system's robustness and reliability are further enhanced by the utilization of technologies such as SQLAlchemy for object-relational mapping, Pydantic for data validation, and JSON Web Tokens (JWT) for secure authentication. A significant value addition is the integration of Google Gemini for the chatbot, which provides immediate assistance to users and aids clinic staff with preliminary symptom assessment.

The application successfully implements role-based access control, guaranteeing that Patients, Doctors, Clinic Staff, and Administrators have appropriate and secure access to system features and data. The user interface has been designed with a focus on intuitiveness and user-friendliness, contributing to an enhanced overall user experience. Ultimately, the project has achieved its primary objectives of streamlining clinic operations, improving data management capabilities, and enhancing the quality of service delivery.

The system has been successfully deployed and is fully operational. The frontend React application is hosted on Vercel and is accessible at clinic-management-nine-lime.vercel.app. The backend FastAPI application is deployed on Render, available at clinic-management-be3h.onrender.com. Data persistence is managed by a Google Cloud SQL for PostgreSQL instance. Continuous integration and continuous deployment (CI/CD) pipelines are established through GitHub, ensuring that both Vercel and Render automatically deploy updates from the bnmbanhmi/clinic-management repository, keeping the live applications current with the latest stable code. Environment variables on Vercel link the frontend to the backend, and CORS policies on the backend are configured to permit requests from the live frontend URL, ensuring seamless communication between the services. This successful deployment marks the project's readiness for operational use.

## 5.2 Future Development Directions

While the current system fulfills its core objectives, the platform is well-positioned for future enhancements and expansions. Potential future development directions include:

**Advanced EMR Features:**

* Integration with standardized medical coding systems (e.g., ICD-10, SNOMED CT) for diagnoses.
* Support for image attachments (X-rays, lab reports) within the EMR.
* Customizable EMR templates for different specialties or consultation types.
* Drug interaction checking and allergy alerts within the prescription module.

**Enhanced Patient Portal:**

* Secure messaging between patients and doctors/clinic.
* Online payment integration for appointments or services.
* Patient education resources linked to their conditions or treatments.
* Ability for patients to fill out pre-appointment questionnaires.

**Telemedicine Capabilities:**

* Integration of video conferencing tools for remote consultations.

**Inventory Management:**

* Module for managing clinic supplies and pharmacy stock.

**Mobile Application:**

* Develop native mobile applications (iOS and Android) for patients and doctors to provide on-the-go access.

**Enhanced AI Chatbot:**

* Expand the chatbot's knowledge base and conversational capabilities.
* Enable more complex interactions, such as triaging patients based on symptoms or providing personalized health tips (with appropriate disclaimers).

The successful completion of this project provides a strong foundation for these future enhancements, ensuring the Clinic Management System can continue to evolve and meet the growing needs of the clinic and its patients.