Software Design Specifications

for

Chikitsa
(Medical Data Encryption System)

Prepared by

Group Name: Team-5

Team-5

C Srikanth	SE22UCSE062	se22ucse062@mahindrauniversity.edu.in
Dev M Bandhiya	SE22UCSE078	se22ucse078@mahindrauniversity.edu.in
G Aditya Vardhan	SE22UCSE092	se22ucse098@mahindrauniversoty.edu.in
K Sai Bharat Varma	SE22UCSE125	se22ucse125@mahindrauniversoty.edu.in
K Harpith Rao	SE22UCSE141	se22ucse141@mahindrauniversity.edu.in
Punith Chavan	SE22UCSE310	se22ucse310@mahindrauniversoty.edu.in
Harsha B	SE22UCSE321	se22ucse321@mahindrauniversity.edu.in

Instructor: ***Vijay Rao***

Course: **Software Engineering**

Lab Section: ***CS-1 & CS-2***

Teaching Assistant: ***Sravanthi Acchugatla***

Date: **10-3-2025**

Table of Contents

1 Introduction

The Chikitsa project is a digital healthcare platform designed to simplify and streamline medical consultations through a unified web and mobile application. With an emphasis on accessibility, it allows patients to connect with certified doctors, schedule appointments, and maintain their health records online. The software is built with scalability and ease-of-use in mind, targeting both urban and rural populations with limited access to traditional healthcare systems.

This Software Design Specification (SDS) document outlines the core architectural structure and detailed design of the system. It translates the functional requirements into technical components and describes how the software modules interact, the technologies used, and how data is processed within the system. The SDS acts as a blueprint for developers, testers, and stakeholders, ensuring a common understanding of the software's framework and intended behavior.

1.1 Purpose

The purpose of this Software Design Specification (SDS) document is to outline the architectural and detailed design of *Chikitsa*, a mobile and web-based healthcare management system. This document serves as a bridge between the system requirements and the actual coding process by defining the components, data flow, and user interactions in a clear and structured manner. It is intended for developers, testers, and stakeholders involved in the development and maintenance of the software.

1.2 Scope

Chikitsa is designed to provide an accessible and centralized healthcare platform for users, allowing them to register as patients or doctors, book appointments, consult online, and maintain electronic health records (EHR). The system will include modules such as user authentication, appointment scheduling, doctor-patient communication, and report storage. It aims to enhance the efficiency and reach of medical services, particularly in remote or underserved regions.

1.3 Definitions, Acronyms, and Abbreviations

- **EHR** – Electronic Health Record
- **API** – Application Programming Interface
- **UI/UX** – User Interface / User Experience
- **DBMS** – Database Management System
- **OTP** – One-Time Password
- **SDS** – Software Design Specification
- **MVC** – Model-View-Controller (software design pattern)

1.4 References

- *IEEE Standard 1016-2009: IEEE Standard for Information Technology—System Design—Software Design Descriptions*
- *Project Requirement Specification Document*
- *Technology Documentation*
- *GitHub Repository* - https://github.com/harpith/CHIKITSA

2 Use Case View

2.1 Individual Dashboard Access

Purpose: To allow all users to log in and access dashboards securely, according to their access permissions.

Actors:

- Patient
- Doctor
- Admin

Preconditions - User must have a registered account on the portal.

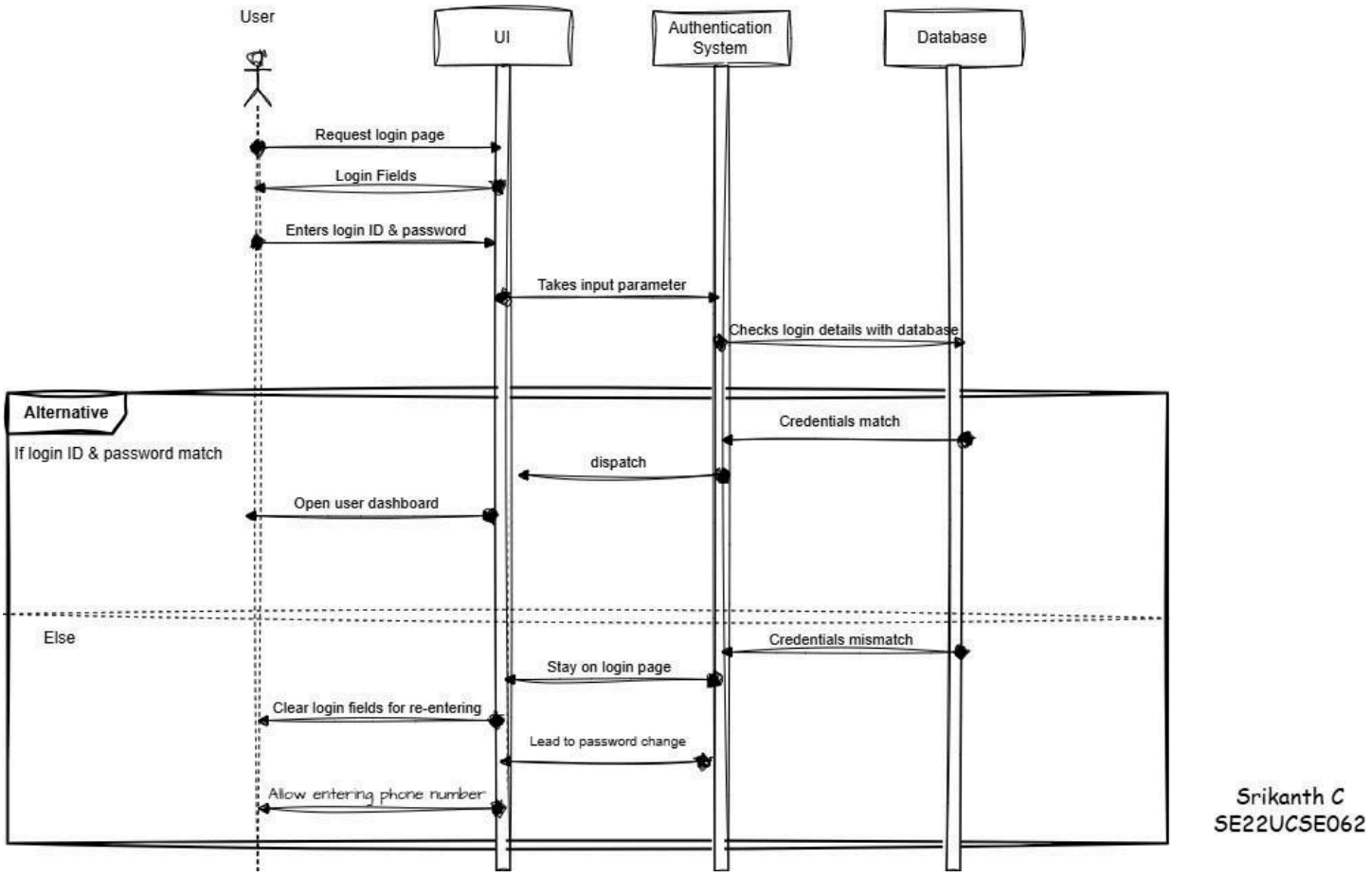
Post conditions - The user is authenticated and redirected to their respective dashboard.

Event Flow:

- Base Flow sequence
    1. User enters login credentials
    2. The system validates credentials against stored ones in the database.
    3. If valid, a JWT token is assigned to the user and they are redirected to the dashboard.

- Alternative Flow -
    1. If the user enters wrong credentials, an error message is shown and the page is reset.
    2. If users forget their passwords, they can reset them via OTP verification.

- Exceptions - System failure.



2.2 Secure Medical Data View

Purpose: To allow patients & doctors to view medical records securely

Actors:

- Patients
- Doctors

Preconditions - User must be authenticated, Medical record must exist in the database

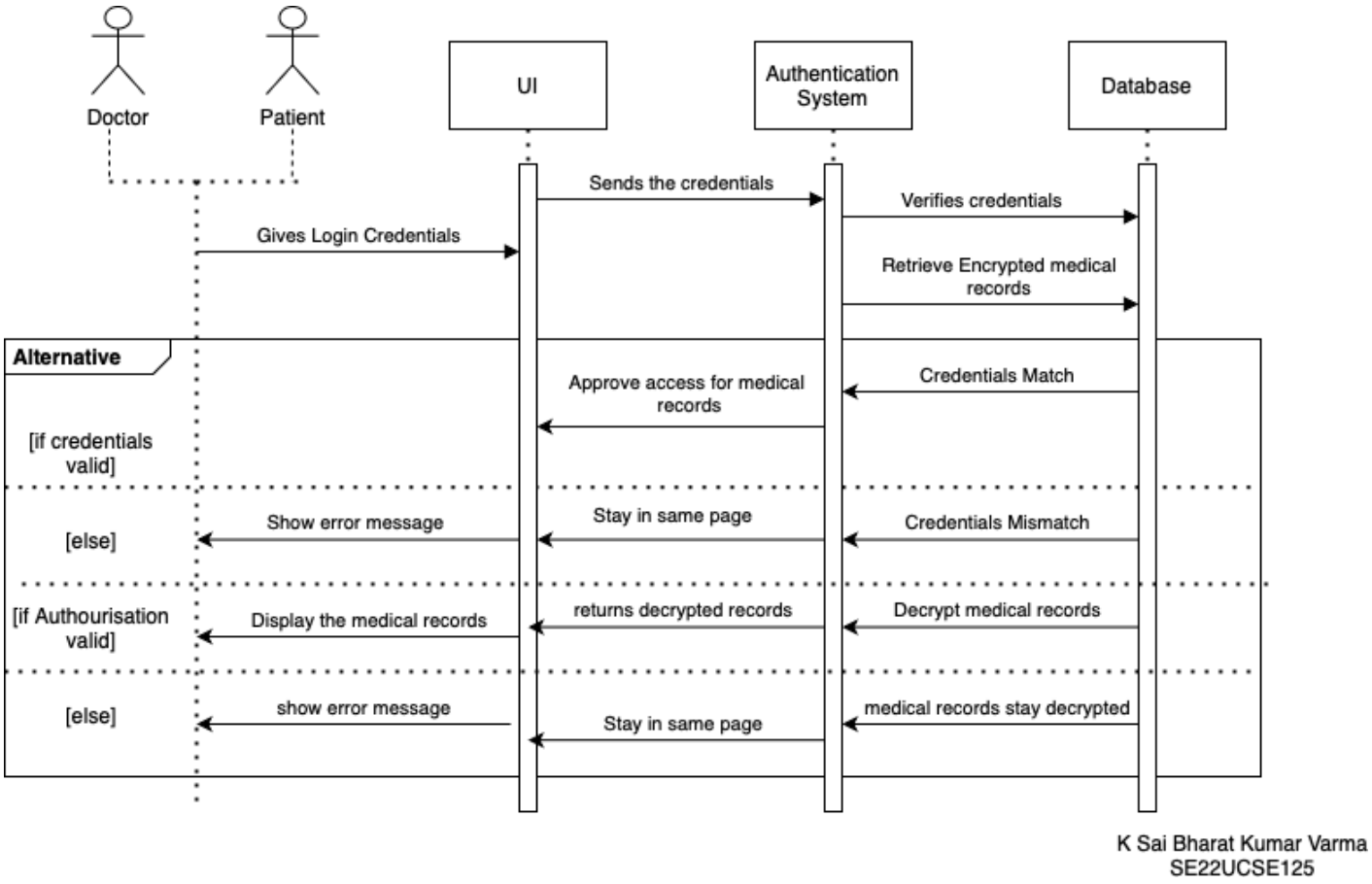
Postconditions - The requested medical records are displayed to the authorized user.

Event Flow

- **Basic Flow sequence:**
    1. The user selects the "View Medical Records" option.

2. The system checks user authorization.
3. If authorized, the system retrieves and decrypts the medical record.
4. The system displays the records securely.

- **Alternative Flow:**
  1. If no records exist, the system displays a "No records found" message.
- **Exceptions:**
  1. Unauthorized access attempt: Log the event and display an error message.



2.3 Secure Linking of Users for Data Exchange

Purpose: To allow an Admin to securely link a Patient and a Doctor so that the doctor can upload prescriptions and the patient can view them.

Actors:

- Admin
- Users

Preconditions:

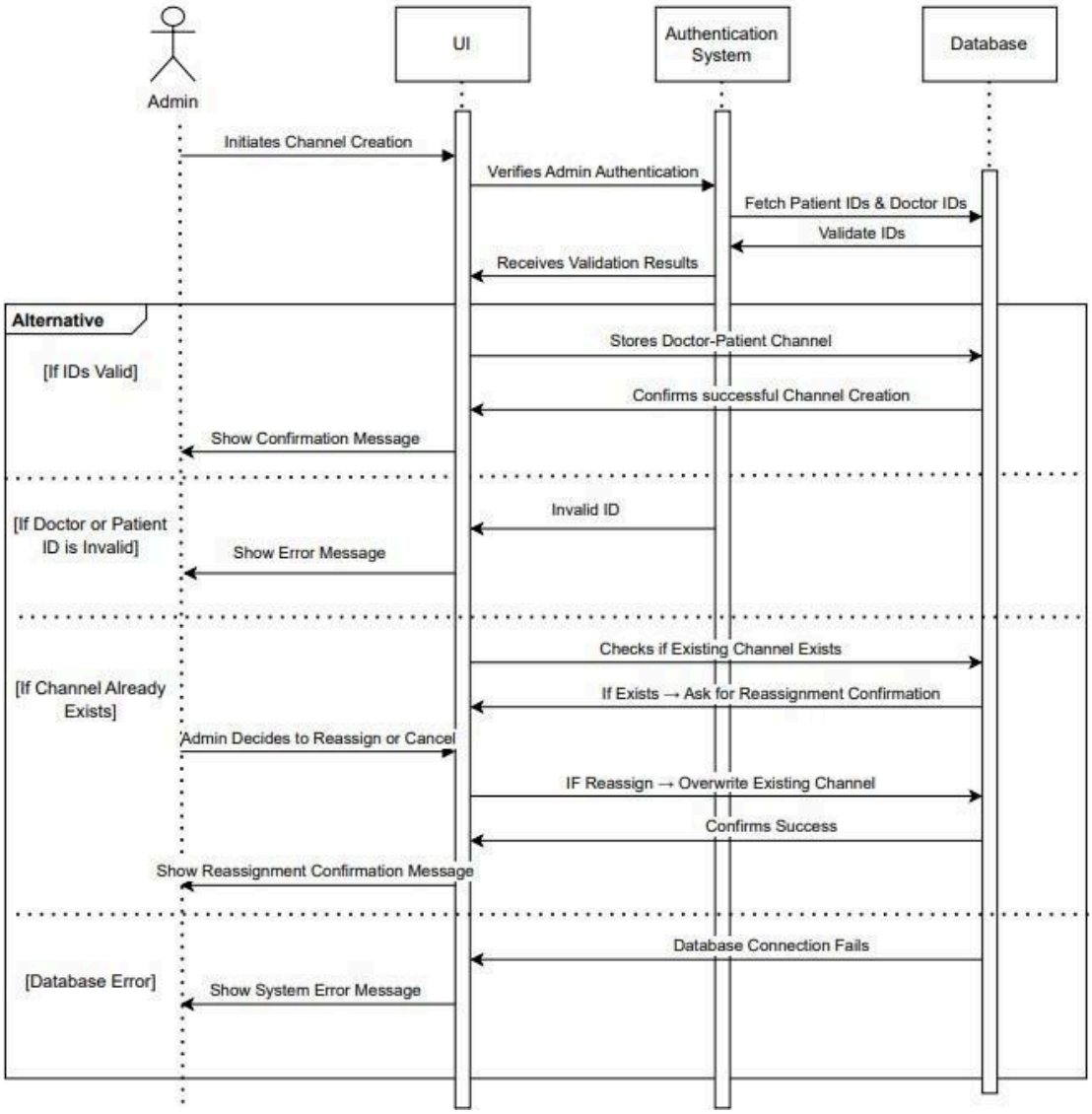
- The Admin must be authenticated.
- Both the Doctor and Patient accounts must exist in the system.

Postconditions:

- A secure channel is created between the doctor and the patient.
- The doctor can upload prescriptions securely.
- The patient can view prescriptions assigned to them.

Event Flow:

- **Basic Flow Sequence:**
    1. The Admin logs into the system.
    2. The Admin selects "Create Doctor-Patient Channel."
    3. The system prompts for the Patient account ID and the Doctor account ID.
    4. The Admin enters the Patient ID and Doctor ID.
    5. The system validates both IDs and verifies that a doctor-patient relationship exists or can be assigned.
    6. If valid, the system creates a secure channel linking the patient and doctor.
    7. A confirmation message is sent to both the doctor and the patient.
    8. The doctor can now upload prescriptions, and the patient can view them securely.
- **Alternative Flow**
    1. Unauthorized Admin Access: The system denies access and logs the event.
    2. Database Connection Failure: The system notifies the admin and retries the request.
    3. Patient/Doctor Deletion: If a linked account is deleted, the system removes the channel and notifies the admin.



2.4 Data Consistency

Purpose: To ensure that the health data stored in the database remains accurate, consistent, and validated to prevent errors, unauthorized modifications, or inconsistencies.

Actors:

- Admin
- Database System

Preconditions:

- Doctor must be authenticated.
- The health data must be valid, complete, and formatted correctly before being stored.
- The database connection must be active.

Postconditions:

- The database remains accurate and consistent with validated health data.
- Any invalid, duplicate, or inconsistent data is rejected before being stored.
- All modifications are logged for future audits.

Event Flow:

Basic Flow:

1. Doctor logs into the system.
2. The user selects the "Update Health Data" or "Add New Record" option.
3. The system validates the input data (e.g., format, completeness, duplicate check).
4. If the data is valid, the system commits it to the database and logs the transaction.
5. If the data is invalid or inconsistent, the system rejects it and prompts for correction.
6. The system maintains audit logs of all modifications for security and traceability.

Alternative Flow:

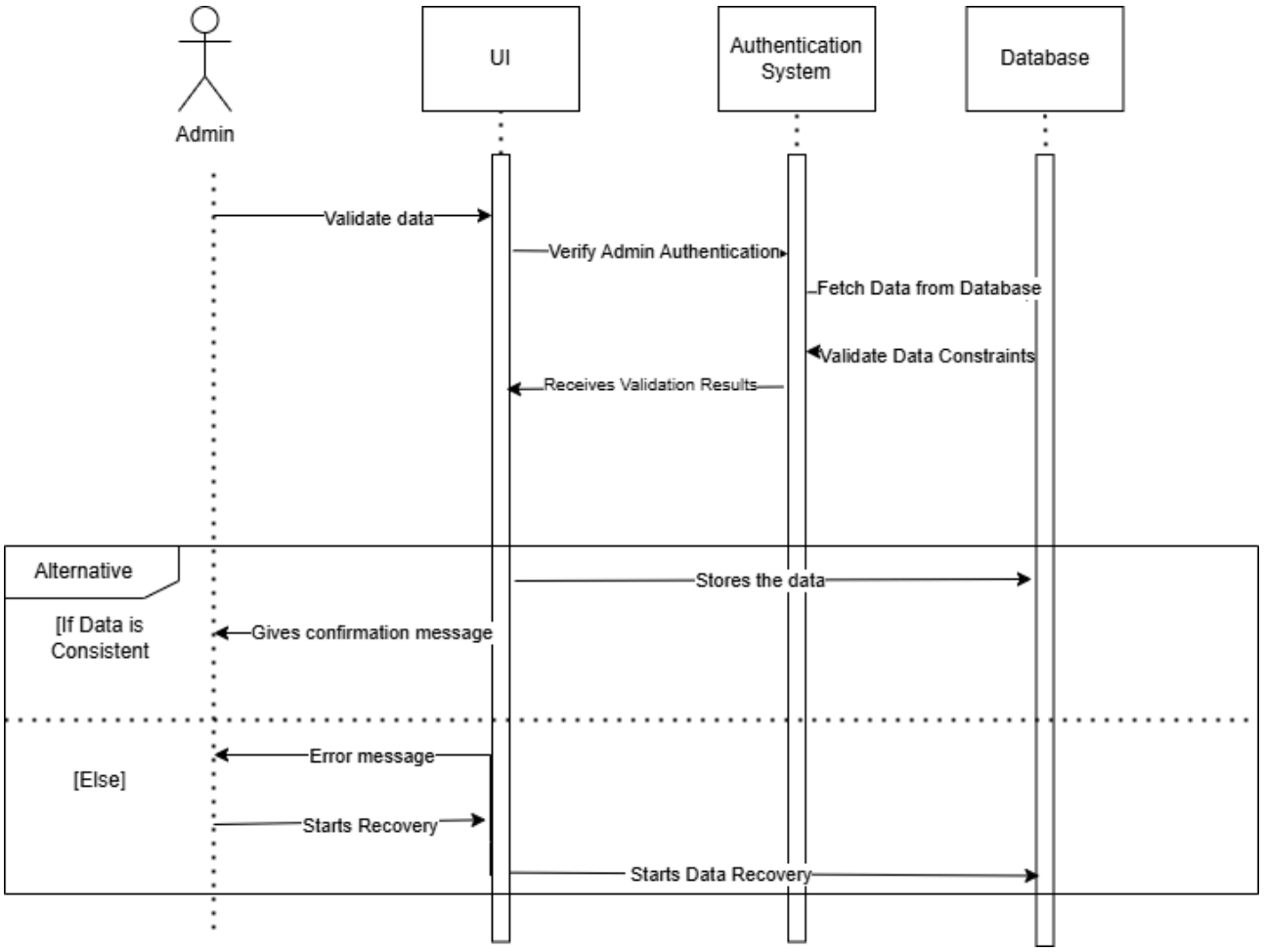
1. If invalid data is entered, the system displays an error message and requests corrections.
2. If a duplicate record is detected, the system notifies the user and suggests merging or updating the existing record.
3. If the database connection fails, the system retries the transaction and notifies the doctor if unsuccessful.

Exceptions:

Unauthorized Modification Attempt: The system blocks access, logs the event, and alerts the admin.

Data Corruption Detected: The system runs integrity checks and restores data from backups if necessary.

Incomplete or Missing Data: The system rejects incomplete records and requests required information.



3 Design Overview

3.1 Design Goals and Constraints

Software Constraints

Frontend

Framework: ReactJS
Styling: Tailwind CSS
"@tailwindcss/vite":
"^4.1.2", Build Tool: Vite

Key Dependencies:
"@tailwindcss/vite":
"^4.1.2",

"react": "^19.0.0"

"react-dom": "^19.0.0"

"react-icons": "^5.5.0"

"react-router-dom": "^7.4.1"

"tailwindcss": "^4.1.2"

Backend

Framework: Node.js with Express.js

Key Dependencies:

axios: 1.3.4

dotenv: 16.0.3

mongoose: 8.13.2

Database

Software - MongoDB Atlas 8.0

Access: Interacted with through the backend using Mongoose ODM for schema-based modeling and validation

Hardware Constraints

- Must be compatible with budget smartphones (e.g., 2–3GB RAM, Android 8+).
- Backend is deployed on limited-resource cloud infrastructure (1 vCPU, 1GB RAM).
- Performance should remain stable under low-bandwidth conditions (~512 kbps).

Objectives

- **Responsiveness**: Ensure optimal performance across mobile and tablet screens.
- **Scalability**: Backend services should be structured to support additional features or integrations with minimal rework.
- **Ease of Use**: UI should be intuitive for users with minimal technical experience.

- **Data Consistency**: Use schema validation (Mongoose) to ensure data integrity between frontend and database.
- **Fast Load Times**: Vite and code-splitting should be used to reduce initial load time.
- **Separation of Concerns**: Strict separation between frontend, backend, and database logic.

Design and Implementation Strategy Constraint

- **Frontend Architecture:**
    - Built using **ReactJS** with a component-based design pattern.
    - **Tailwind CSS** is used for utility-first styling, supporting a scalable and responsive UI.
- **Backend Architecture:**
    - API design using **Node.js** and **Express.js**.
    - Ensures stateless communication between client and server.
    - Encourages modular routing and middleware-based request handling.

3.2 Design Assumptions

Assumptions

- The app will primarily run on Android 8+ smartphones with 2GB RAM.
- MongoDB Atlas will be available 24/7 with <300ms average response
- Frontend uses Axios to call Express routes at API endpoints.

3.3 Significant Design Packages

Frontend

- **App Layer**: Manages routes using `react-router-dom`, and UI state.
- **Component Layer**: Modular, reusable UI components styled with Tailwind CSS.

- **Service Layer**: Interfaces with backend APIs using Axios.
- **Routing**: Handles page transitions using `react-router-dom`.

Backend

- **Server Schema**
  Routing, Middleware & Controller combined as a single unit.

Shared/External

- **Build Tool**: Vite handles frontend build and development server.
- **Database Integration**: Mongoose used as ODM for schema modeling with MongoDB Atlas.

3.4 Dependent External Interfaces

The table below lists the public interfaces this design requires from other modules or applications.

External Application	Module Using the Interface	Interface Name	Description and Interface Usage
Mongo DB Atlas	Data Persistence Module	Mongoose ODM	Used by the backend to define schemas and perform CRUD operations securely on encrypted medical data.
Environment Variables (.env)	Configuration Module	dotenv	Used to load sensitive keys, MongoDB credentials, and encryption parameters during runtime.
React Router	Frontend Navigation Module	react-router-dom	Used to define application routes for user-facing modules like login, dashboard, and admin panel.
Tailwind CSS	Frontend UI Module	tailwindcss	Provides a utility-first CSS framework to quickly style and layout the components.
Vite	Frontend Build Module	@tailwindcss/vite	Used for development and optimized production builds of the frontend app.

3.5 Implemented Application External Interfaces (and SOA web services)

The table below lists the implementation of public interfaces this design makes available for other applications.

Interface Name	Implementing the Interface	Functionality / Description
/api/encrypt	Encryption Controller (Backend)	Accepts raw medical data, encrypts it using the encryption engine, and stores it securely in MongoDB.
/api/decrypt/:id	Decryption Controller (Backend)	Retrieves encrypted data by ID, decrypts it with appropriate key, and returns it to the authorized user.
/api/auth/login	Auth Module (Backend)	Authenticates users and returns access tokens for session management.
/api/keys/rotate	Key Management Module (Backend)	Rotates encryption keys as per admin or system schedule to maintain security.
/api/data/upload	Data Handling Controller (Backend)	Accepts and processes uploaded medical files, encrypts them, and logs the transaction.

4 Logical View

The system follows a **Client-Server Architecture** that separates responsibilities between the client-side application and the backend server to enhance modularity, maintainability, and scalability.

The architecture is organized into the following components:

Components of the Client-Server Architecture:

- **Client (Frontend Application)**
  The client-side application is built using **ReactJS** and is responsible for presenting the user interface to patients, doctors, and administrators. It communicates with the backend server via secure HTTP requests (REST APIs) and handles user interactions like logins, data viewing, uploads, and appointments.

- **Server (Backend Application)**
  The server-side application is built using **Node.js** and **Express.js**. It manages business logic, data processing, authentication, encryption, and API request handling. The server acts as an intermediary between the client application and the database, processing requests and enforcing access control.

- **Database (MongoDB)**
  The system uses a **MongoDB** database to securely store encrypted medical records, user information, appointments, and system logs. The server handles all database interactions, ensuring data integrity and security through AES-GCM encryption before storage.

Interaction Flow:

- The **client application** sends API requests to the **server** for various actions like logging in, uploading records, making

appointments, and retrieving data.

- The **server** processes these requests, performs authentication (using JWT tokens), handles encryption/decryption where necessary, and interacts with the **MongoDB database**.
- The **server** sends back responses to the **client application**, which updates the user interface accordingly.

4.1 Design Model

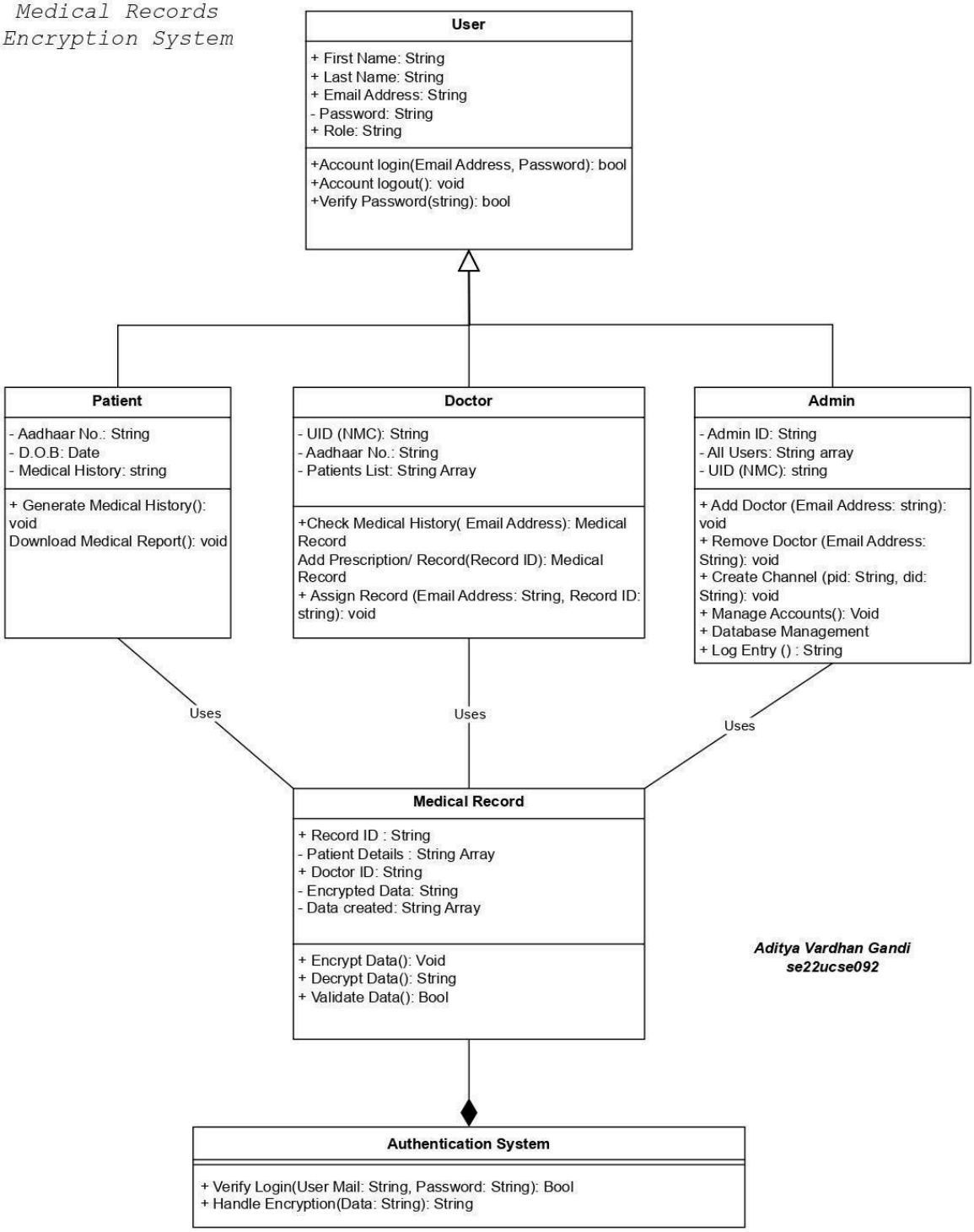
Module Decomposition:

The system is decomposed into the following modules:

Module	Description
User Management Module	Handles patient, doctor, and admin registration, login, and profile management.
Appointment Management Module	Manages appointment booking, cancellation, and rescheduling.
Medical Record Module	Stores and retrieves patient medical history, prescriptions, and reports.

Chikitsa

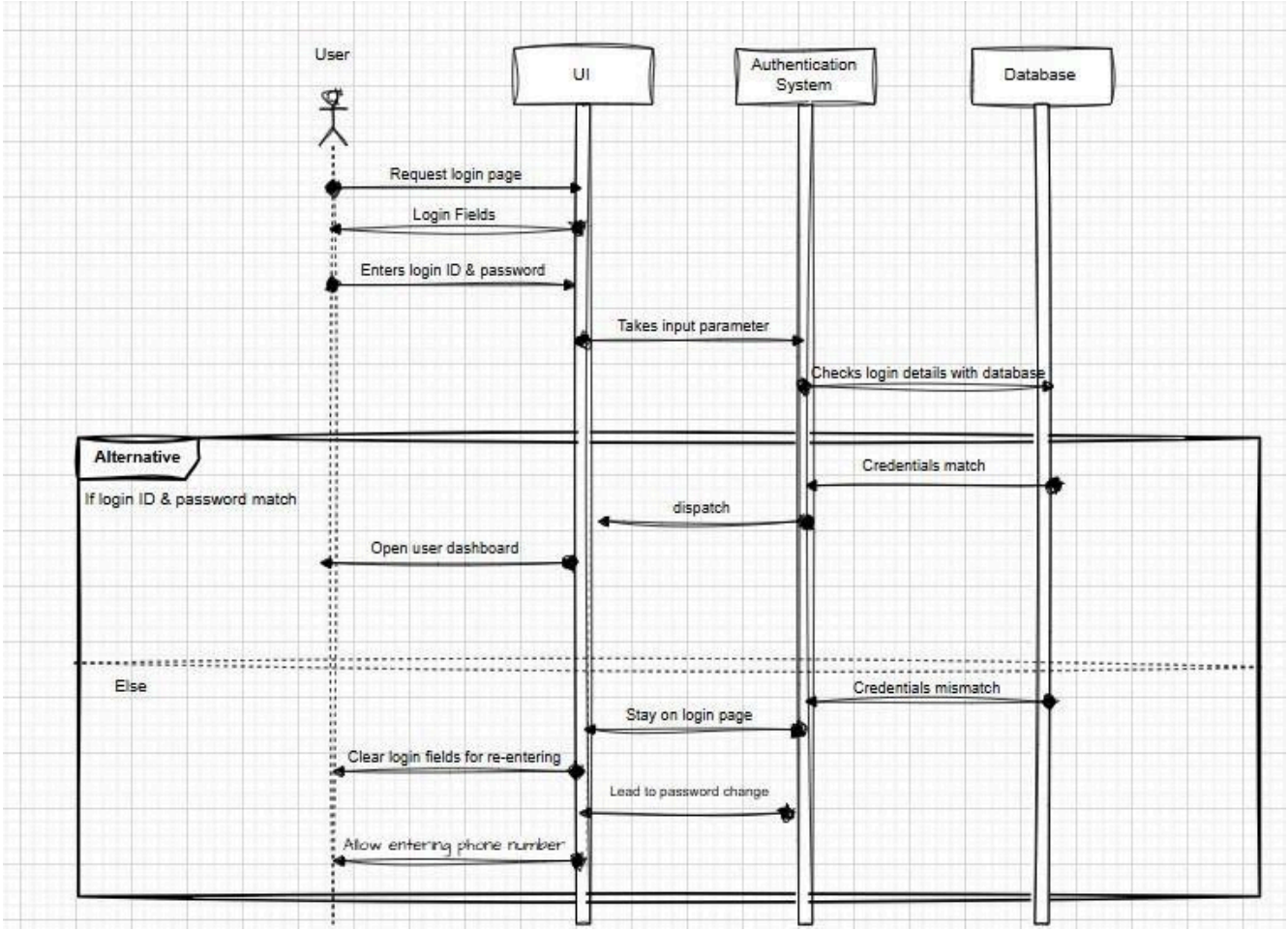
Medical Records Encryption System



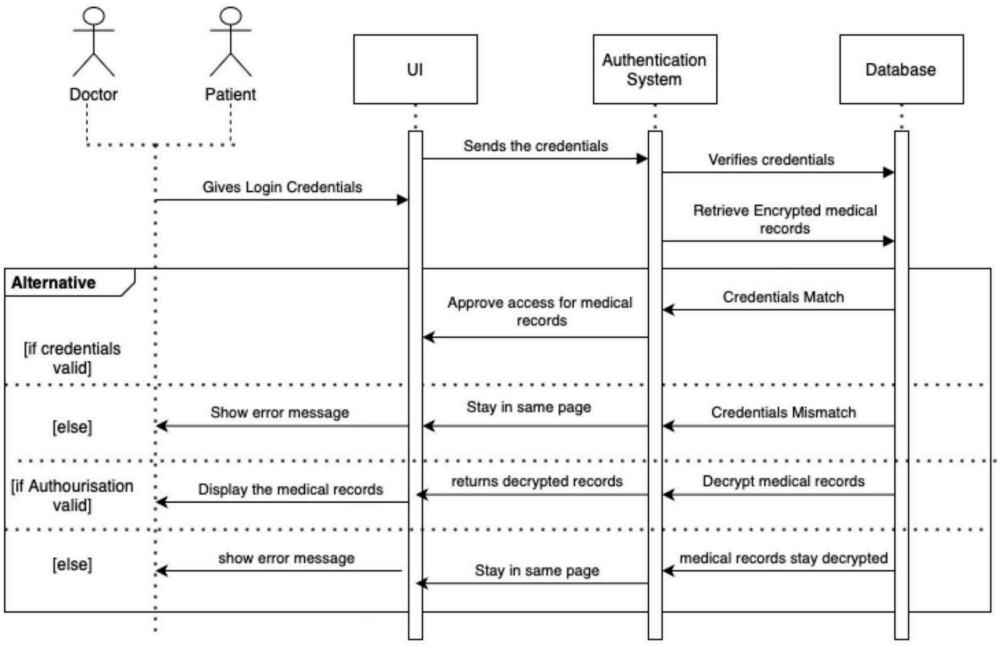
Aditya Vardhan Gandi
se22ucse092

4.2 Use Case Realization

Use Case 1: Login and Signin



Use Case 2: To allow patients & doctors to view medical records securely

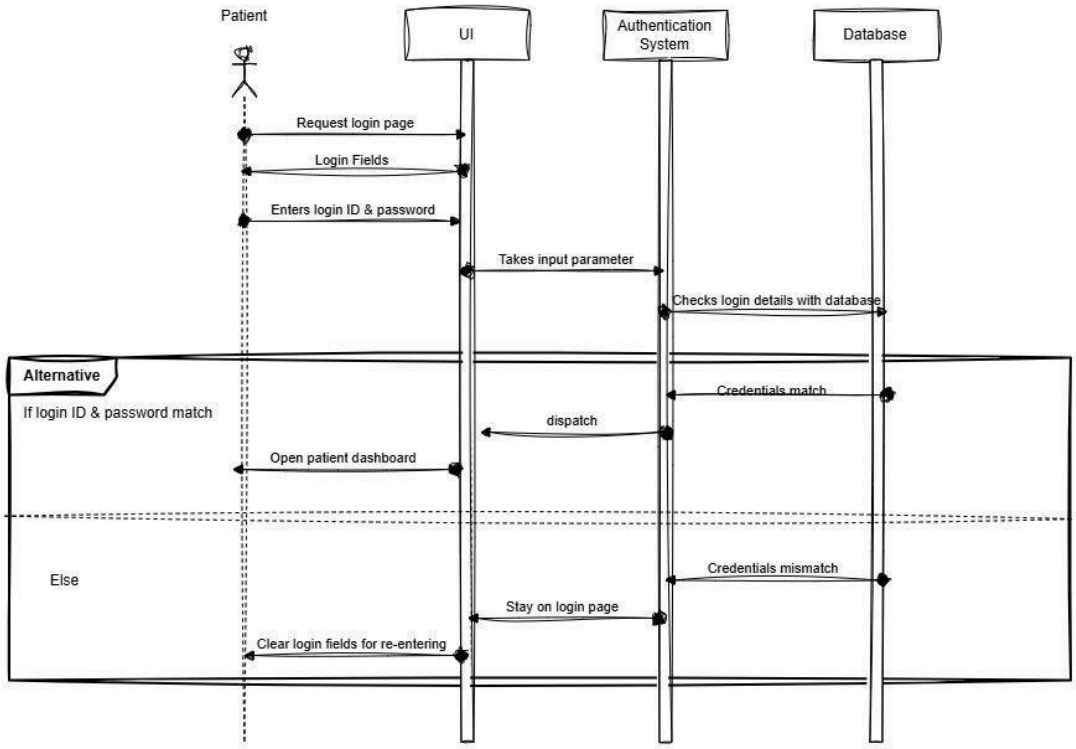


K Sai Bharat Kumar Varma
SE22UCSE125

Use Case 3:To log in as admin and monitor all overall access to the medical records

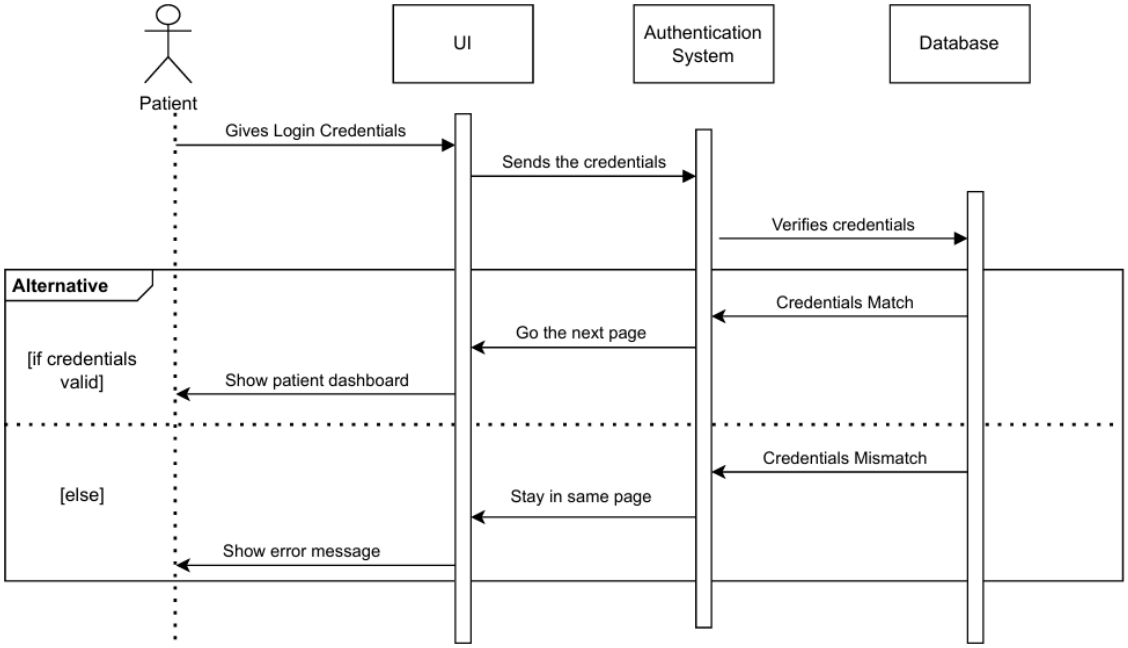
Sequence Diagram

Chikitsa
Medical Records Encryption System



Punith Chavan
SE22UCSE310

Use case 4:

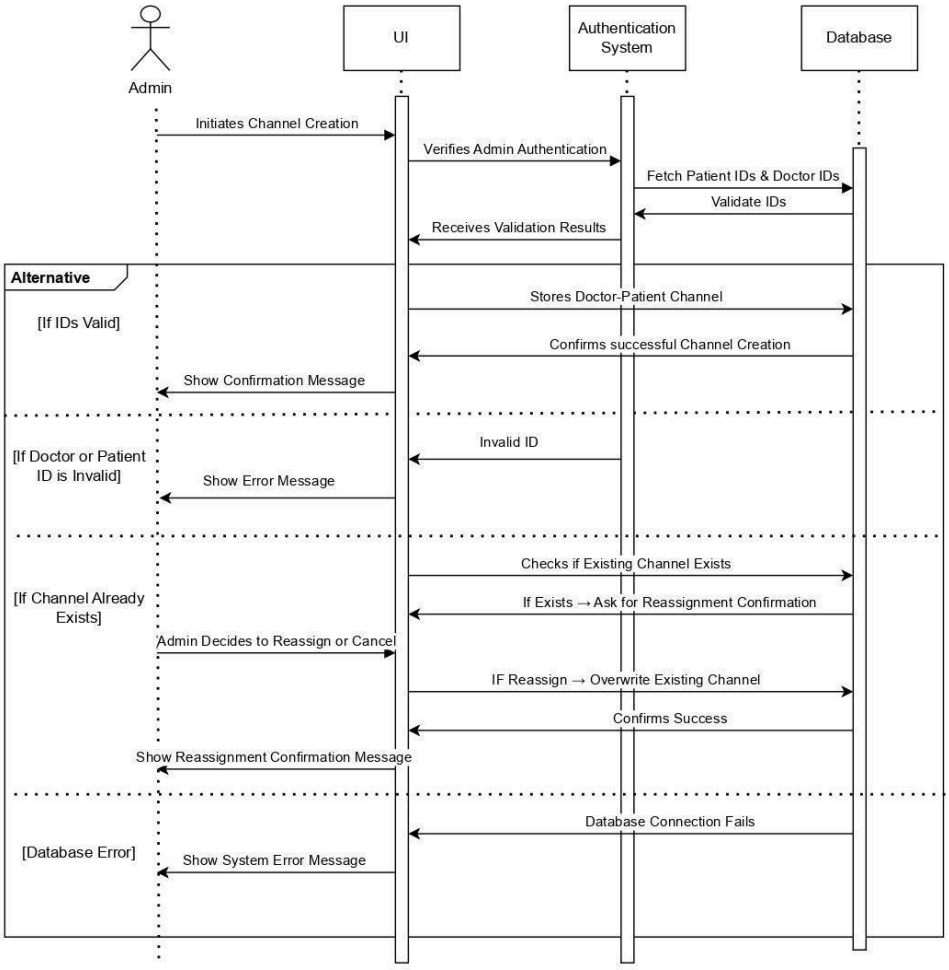


Name: Dev M. Bandhiya
Roll No.: SE22UCSE078

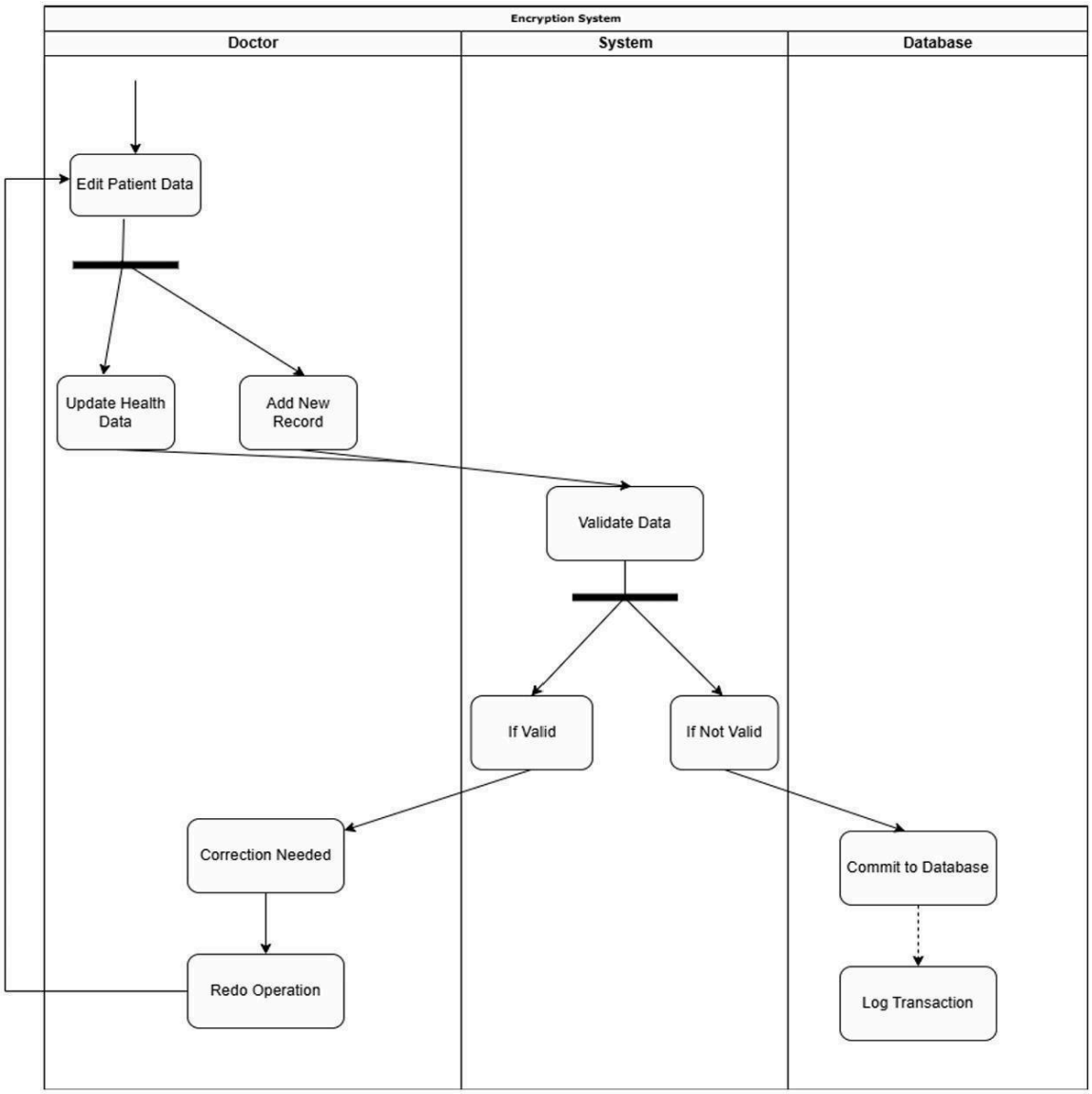
Use Case-5:Admin Doctor -Patient Channel Creation

Admin Doctor-Patient Channel Creation use case

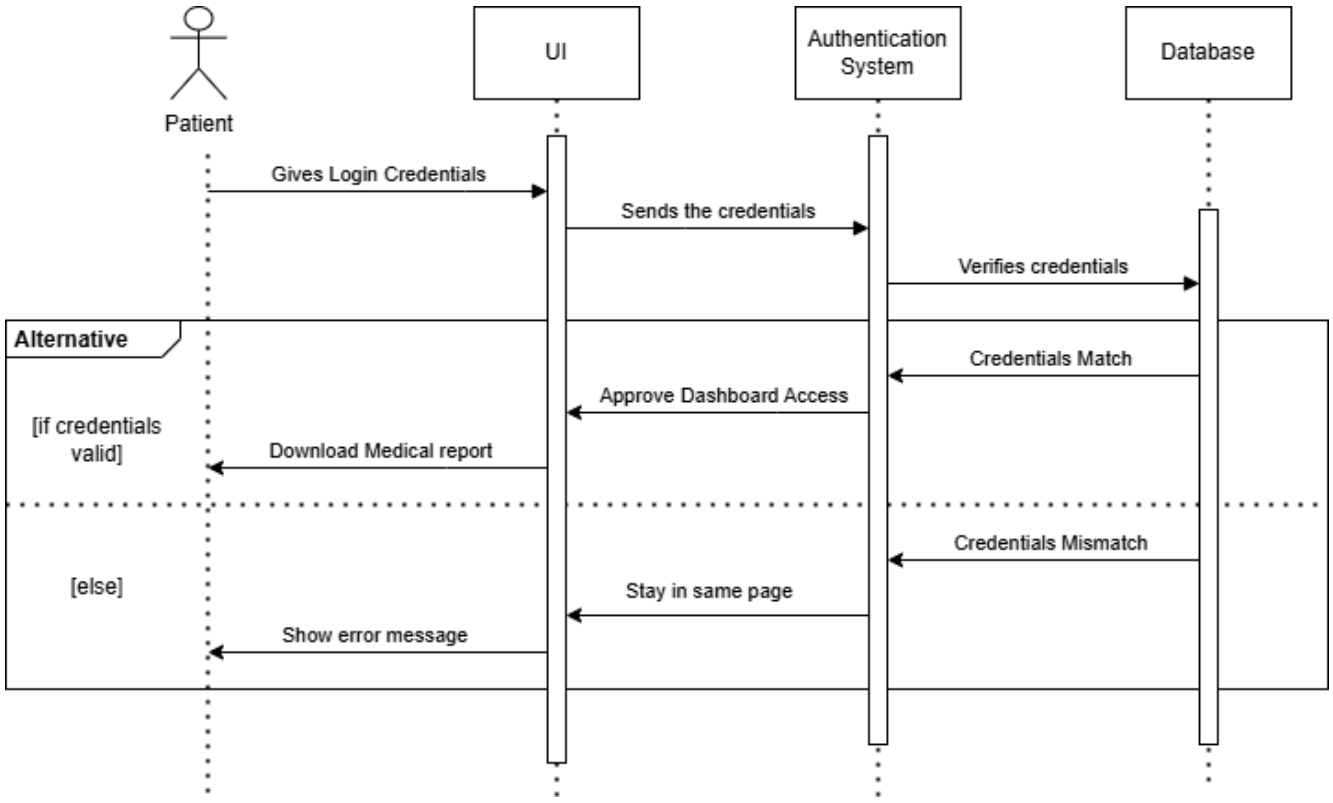
Name: G Aditya Vardhan
Roll No.: SE22UCSE092



Use Case 6: DataBase Consistency



Use Case 7: Download Medical Reports



5 Data View

In the Chikitsa Healthcare Management System, MongoDB is used as the primary database. MongoDB stores data in flexible, JSON-like documents which allows for dynamic and scalable data modeling. Collections in MongoDB replace traditional relational tables, and documents within collections represent entities.

5.1 Domain Model

1. Users Collection:-

{
"_id": ObjectId,
"username": "john_doe",
"password": "hashed_password",
"role": "Patient / Doctor / Admin", "contact": "9876543210", "email": "john@example.com",
"Aadhaar No":"9889 9888 9888 9888"
}

2. Appointments Collection:-

```
{
  "_id": ObjectId,
  "patient_id": ObjectId, // Reference to users
  "doctor_id": ObjectId, // Reference to users
  "date": "2025-04-07",
  "time": "10:30 AM",
  "status": "Booked / Cancelled / Completed"
}
```

3.Medical_Records Collection:-

```
{
  "_id": ObjectId,
  "patient_id": ObjectId, // Reference to users
  "doctor_id": ObjectId, // Reference to users
  "description": "Fever and headache",
  "prescription": "Paracetamol 500mg",
  "report": "URL or Embedded File"
}
```

5.2 Data Model (persistent data view)

Collection	**Relationships**	**Purpose**
users	Referenced by appointments, medical_records, feedback, notifications	Stores user details
appointments	References users (patient_id, doctor_id)	Stores appointment info
medical_records	References users (patient_id, doctor_id)	Stores medical history
feedback	References users (patient_id, doctor_id)	Stores user feedback
notifications	References users (user_id)	Stores system notifications for users

5.2.1 Data Dictionary

i) Users Collection

Field Name	Data Type	Description	Constraints
_id	ObjectId	Unique identifier for user	Primary Key
username	String	Username of the user	Unique, Required
password	String	Encrypted password	Required
role	String	Role of the user (Patient/Doctor/Admin)	Required

contact	String	Contact number of the user	Required, Unique
email	String	Email address	Required, Unique

---

ii) Appointments Collection

Field Name	Data Type	Description	Constraints
_id	ObjectId	Unique identifier for appointment	Primary Key
patient_id	ObjectId	Reference to user (Patient)	Foreign Key
doctor_id	ObjectId	Reference to user (Doctor)	Foreign Key
date	Date	Appointment date	Required
time	String	Appointment time	Required
status	String	Appointment status (Booked/Cancelled/Completed)	Required

iii) Medical_Records Collection

Field Name	**Data Type**	**Description**	**Constraints**
_id	ObjectId	Unique identifier for record	Primary Key
patient_id	ObjectId	Reference to user (Patient)	Foreign Key
doctor_id	ObjectId	Reference to user (Doctor)	Foreign Key
description	String	Health issue description	Required
prescription	String	Medicines prescribed	Optional

report	String	Medical report file URL or details	Optional

6 Exception Handling

In Chikitsa, exceptions are managed through a structured system to ensure smooth operation and meaningful user feedback. Below is a description of the key exceptions defined within the application, the circumstances under which they may be triggered, how they are logged, and the necessary follow-up actions:

1. AuthenticationFailedException

- Triggered When: A user (patient or doctor) enters an incorrect email or password during login.
- Handling: A generic message "Invalid email or password" is displayed to avoid revealing account existence.
- Logging: Email entered, IP address, and timestamp are stored in the authentication logs.

2. InvalidUserIDException

- Triggered When: The admin enters an invalid or non-existent PATxxx or DOCxxx ID while trying to link a patient and doctor.
- Handling: Error message such as "Invalid Patient or Doctor ID" is shown.
- Logging: Admin ID, entered ID, and timestamp are logged.
- Follow-up Action: Admin should re-verify the IDs. Frequent input errors may prompt a suggestion to use an autocomplete or ID picker tool.

3. UnauthorizedAccessException

- Triggered When: A user tries to access resources not assigned to them (e.g., a doctor accessing records of a patient not under their care).
- Handling: User is redirected to an "Access Denied" page.
- Logging: User ID, attempted resource, and timestamp are recorded.
- Follow-up Action: Admin is alerted if repeated attempts are made. The user may be temporarily restricted or monitored.

4. RecordNotFoundException

- Triggered When: Expected data like appointment details or medical history is not found in the database.
- Handling: A message such as "No records found" is displayed to the user.
- Logging: User ID and the attempted access (e.g., appointment ID) are logged.
- Follow-up Action: If this occurs frequently, admins may be notified to ensure data integrity and completeness.

5. DatabaseConnectionException

- Triggered When: The system fails to connect to the centralized database due to network or server issues.
- Handling: All users are shown a "Service temporarily unavailable" message.
- Logging: System logs include error stack trace and timestamps.
- Follow-up Action: An automated alert is sent to the technical team to investigate and restore connectivity.

7 Configurable Parameters

This section outlines the configurable parameters for the application. These parameters support flexibility in adapting to different environments and evolving system requirements. They include security settings, system limits, timeouts, role definitions, and more.

Configuration Parameter Name	Definition and Usage	Dynamic?
MAX_LOGIN_ATTEMPTS	Maximum number of consecutive failed login attempts before locking the user account. Helps prevent brute force attacks.	Yes
TOKEN_EXPIRY_DURATION	Defines how long an access token remains valid after login. Helps manage session security.	Yes
PRESCRIPTION_UPLOAD_LIMIT_MB	Maximum allowed size (in MB) for a prescription file uploaded by doctors. Prevents excessive resource consumption.	Yes
CHANNEL_TIMEOUT_HOURS	Duration after which an inactive doctor-patient channel is automatically marked as expired/archived.	Yes
ALLOWED_FILE_TYPES	Specifies the accepted formats (e.g., PDF, PNG, JPG) for medical records and prescriptions. Used during upload validation.	Yes
ENCRYPTION_ALGORITHM	Indicates the encryption method (e.g., AES-256) used for storing sensitive records and prescriptions.	No
DECRYPTION_KEY_TIMEOUT	Time period the decryption key is held in memory after a user accesses a medical record. Enhances security.	Yes
DOCTOR_ROLE_ID	Unique role ID used to identify and authorize doctor-specific features.	No
PATIENT_ROLE_ID	Unique role ID used to identify and authorize patient-specific features.	No
ADMIN_ROLE_ID	Unique role ID used for system-level administrative permissions, such as channel creation and role management.	No
DATA_RETENTION_DAYS	Duration (in days) that logs and records are retained before purging. Complies with data protection regulations.	Yes
EMAIL_NOTIFICATION_ENABLED	Toggles email notifications for actions like prescription uploads or channel creation.	Yes
AUTHENTICATION_METHOD	Defines the mode of authentication (e.g., OTP, Password, Biometric) the system supports.	No
MAX_CONCURRENT_SESSIONS	Controls how many active sessions a user can have simultaneously. Limits misuse.	Yes
DB_CONNECTION_RETRY_LIMIT	Number of retries for database connection failures before throwing a system error.	Yes

CHANNEL_CREATION_VALIDATION_TIMEOUT	Time window within which the admin must complete linking doctor and patient IDs, or the request expires.	Yes
SYSTEM_LOGGING_LEVEL	Determines the logging verbosity (e.g., DEBUG, INFO, WARN, ERROR). Helps in debugging and monitoring.	Yes

8 Quality of Service

The Chikitsa system has been designed with a strong focus on maintaining high-quality service standards to ensure reliability, user trust, and effective healthcare delivery. The following subsections address key areas such as availability, security, performance, and monitoring.

8.1 Availability

The *Chikitsa* healthcare application is designed to meet the business requirement of **99.9% system availability**, ensuring uninterrupted access to healthcare services for both patients and doctors. High availability is critical to enable users to book appointments, access medical records, and consult doctors at any time, especially in emergencies.

Design Features Supporting Availability

To meet this requirement, the system incorporates several architectural elements and design decisions:

- **Cloud-Based Infrastructure**: Deployed on reliable cloud platforms such as Firebase or AWS to leverage auto-scaling, redundancy, and geographic distribution.
- **Database Replication**: Ensures that backup copies of data are available to maintain continuity in case of primary database failure.
- **Load Balancing**: Distributes incoming traffic across multiple servers to avoid single points of failure and ensure consistent performance.
- **Modular Microservices Architecture**: Allows independent deployment and fault isolation, ensuring that a failure in one service (e.g., chat module) does not impact others (e.g., appointment system).
- **Graceful Degradation**: The app is designed to handle temporary service outages (e.g., limited access mode) without a full system shutdown.

Impacting Factors and Downtime Considerations

While *Chikitsa* is engineered for maximum uptime, certain operations may impact availability:

- **Mass Data Updates**: Large-scale updates, such as bulk import of patient data or doctor records, are scheduled during low-traffic hours and handled in batches to prevent server overload.
- **Periodic Maintenance**: Server and database maintenance is planned during late-night windows (e.g., 2 AM – 4 AM IST) with prior notification to users. The system may enter a "read-only" mode during this time.
- **Housekeeping Tasks**: Background processes like log rotation, report archival, and token expiration are performed asynchronously to minimize disruption.

Recovery and Failover

In the event of an unexpected outage, automated failover protocols ensure a switch to standby servers or backup databases. Regular backups are maintained to allow quick data restoration, and uptime monitoring tools are used to alert administrators instantly when issues occur.

8.2 Security and Authorization

Security and data privacy are fundamental business requirements for *Chikitsa*, given the sensitive nature of patient information and the need to prevent unauthorized access. The system is designed to comply with data protection standards and ensures that only authenticated and authorized users can access specific features and datasets.

Authorization Framework and Role-Based Access

Chikitsa enforces **role-based access control (RBAC)** to ensure that users only interact with the features and data relevant to their roles. The primary roles include:

- **Patient** – Can access and manage their own health records, view prescriptions, and book/cancel appointments.
- **Doctor** – Can view medical history of assigned patients, update diagnosis and prescriptions, and manage appointment slots.
- **Admin** – Manages user accounts, monitors system health, and has access to high-level data for analytics and compliance.

These roles are enforced throughout the application via authorization middleware in the backend API layer, ensuring secure access to both resources and functionality.

Authentication Mechanisms

To securely verify user identities, *Chikitsa* implements:

- **Password-based login** with encrypted password storage using industry-standard hashing algorithms.

Custom Authorization Design

Beyond the generic authorization framework, *Chikitsa* includes domain-specific rules, such as:

- **Doctor-Patient Binding**: Doctors can only access the records of patients who have booked an appointment with them, ensuring patient confidentiality.
- **Restricted Report Sharing**: Lab reports and prescriptions are accessible only by the doctor who created them and the patient involved.
- **Time-bound Access Tokens**: Certain features like prescription download or chat history are time-limited to avoid misuse.

User Administration and Setup

User account setup and access management are exposed through a secure admin dashboard, allowing system administrators to:

- Create, activate, or deactivate user accounts.
- Assign and modify user roles and permissions.

- Monitor login activity and receive alerts for suspicious access patterns.
- Reset user credentials or force logout in case of security incidents.

Additionally, access logs are maintained for auditing purposes, and periodic reviews ensure compliance with internal policies and regulatory standards.

8.3 Load and Performance Implications

The *Chikitsa* application is expected to support a high volume of concurrent users due to its nature as a healthcare platform that facilitates real-time appointments, messaging, and access to sensitive medical data. This section outlines the anticipated system load and its implications on the detailed design components, providing a foundation for load and performance testing strategies.

1. Expected Load and Transaction Rates

Based on projected usage:

- **Concurrent Users**: 1,000–5,000 users online simultaneously during peak hours.
- **Appointment Transactions**: Up to 500 appointment bookings per hour during high traffic.
- **Message Processing**: An average of 20–30 messages per second between patients and doctors via real-time chat or queries.
- **Document Access**: Approximately 200 medical reports or prescriptions accessed/downloaded per hour.

2. Performance Requirements

- **API Response Time**: Critical APIs (e.g., appointment booking, login, messaging) must respond within **<500ms**under peak load.
- **Page Load Time**: Initial load of the mobile or web app must occur within **2–3 seconds** on standard 4G and broadband networks.
- **Real-Time Messaging**: Message latency should not exceed **200ms** for a responsive user experience.
- **Database Query Latency**: Complex queries (e.g., patient history) must complete within **<1 second**.

3. Database Design and Growth Projections

To ensure scalability and responsiveness:

- **Database Optimization**: All frequently accessed tables (appointments, user profiles, messages) are indexed for fast lookup.
- **Expected Growth**:
    - **Appointments Table**: ~50,000 new records/month
    - **Messages Table**: ~500,000 messages/month
    - **Reports/Prescriptions Table**: ~25,000 documents/month
- **Archival Strategy**: Non-active records older than 6 months will be moved to archival storage to reduce load on the active database.

4. Design Implications

To meet these requirements:

- **Asynchronous Processing**: Non-blocking operations (e.g., report uploads, notifications) use background queues (e.g., Firebase Functions or similar task queues).
- **Caching Layer**: Frequently accessed data like doctor lists, FAQs, or past appointments are cached using in-memory storage (e.g., Redis) to reduce DB load.
- **Connection Pooling**: Backend services use connection pooling to optimize database access and avoid bottlenecks during high traffic.
- **Horizontal Scalability**: The system architecture supports horizontal scaling of application servers, database read replicas, and load balancers to handle increased user demand.

5. Load and Performance Test Planning Support

This design supports the development of detailed test plans with defined benchmarks:

- Simulate **5,000 concurrent users** accessing different modules.
- Emulate **peak hour booking spikes** to test system responsiveness under pressure.
- Execute **stress tests** on the chat system to validate real-time communication under heavy load.
- Monitor **database response time** under high write and read loads to ensure compliance with SLAs.

8.4 Monitoring and Control

To ensure reliability, availability, and performance, the *Chikitsa* application integrates several controllable processes and exposes key operational metrics for real-time monitoring and long-term analysis. These capabilities are crucial for identifying issues, ensuring service-level objectives (SLOs), and supporting system maintenance.

Controllable Processes

The following background processes and handlers are implemented within the application:

- **Message Handlers**:
  Real-time message processing between patients and doctors is managed through WebSocket-based handlers. These monitor active sessions, ensure message delivery acknowledgment, and handle retry logic in case of transient failures.
- **Scheduled Daemons / Cron Jobs**:
  - **Appointment Cleanup**: Automatically cancels expired or unattended appointments and notifies users.
  - **Session Expiry Handler**: Monitors token/session expiration and securely logs out inactive users.
  - **Report Upload Watcher**: Monitors and verifies successful upload and storage of medical documents (lab reports, prescriptions).
  - **Archival Process**: Moves inactive or historical data to archive tables or storage buckets during off-peak hours.

- **Notification Dispatcher**:
  Background service that manages push notifications and SMS alerts for reminders, appointment confirmations, and follow-ups.

Monitoring Metrics and Measurable Values

To maintain visibility into application behavior, *Chikitsa* will publish the following metrics through integrated monitoring tools (e.g., Firebase Analytics, Prometheus/Grafana, or custom dashboards):

- **User Activity Metrics**:
  - Number of active users (per minute/hour/day)
  - Login success/failure rate
  - Session duration
- **Performance Metrics**:
  - API response times (per endpoint)
  - Message delivery latency
  - Appointment booking throughput
- **System Health Metrics**:
  - Uptime and downtime logs
  - Memory and CPU usage of backend services
  - Error rate per service (categorized by type: 4xx, 5xx)
- **Database & Storage Metrics**:
  - Read/write query volume
  - Slow query logs
  - Storage utilization and growth patterns
- **Custom Application Logs**:
  - Report upload status (success/failure)
  - Notification delivery success rate
  - Payment gateway callback monitoring (if applicable)

These metrics are logged and visualized using a dashboard accessible to system administrators. Alerting mechanisms (email/SMS) are configured for anomalies, such as elevated error rates or service unresponsiveness.

Control Interfaces

Administrators and DevOps teams are provided with:

- A web-based dashboard for real-time metric viewing.
- Command-line scripts and secure APIs to manually restart or pause background processes.
- Access control for who can trigger or modify daemons and handlers.