# Project Synopsis

## TITLE OF THE PROJECT:

### End-to-End Encrypted Chat Room Web Application

## CONTENTS OF SYNOPSIS

### Catalog

# 1. Title of the Project

**End-to-End Encrypted Chat Room Web Application**

---

# 2. Introduction and Objectives of the Project

## 2.1 Introduction

Today's digital communication landscape is dominated by concerns about privacy and data security. Most popular messaging platforms store our conversations on their servers, making them vulnerable to data breaches, government surveillance, and unauthorized access. This creates a real need for truly private communication tools.

My project addresses this problem by developing a web-based chat application that prioritizes user privacy through three key features:

- **End-to-End Encryption (E2EE):** Messages are encrypted on the sender's device and can only be decrypted by the intended recipients. Even the server hosting the application cannot read the message content - it simply acts as a relay for encrypted data..

- **Ephemerality (Temporary Nature):** Unlike traditional chat apps that store your conversation history forever, this application is designed to be ephemeral. Messages aren't saved anywhere - not on servers, not in databases. When you close the chat room, everything disappears.

- **Room-Based, Code-Driven Access:** Instead of requiring user accounts or friend lists, people can create temporary chat rooms with unique access codes. Share the code with whoever you want to chat with, and they can join instantly.

The technical foundation includes Next.js and Tailwind CSS for a modern, responsive user interface, Socket.io for real-time messaging capabilities, and MongoDB for managing temporary room information (but never storing actual messages). The goal is creating a platform where people can have genuinely private conversations without worrying about their data being stored or accessed by others.

## 2.2 Objectives

The principal objectives for this project are:

1. **Develop a Secure Chat Platform:** Create a web application where users can make temporary chat rooms using unique access codes, without needing to create accounts or provide personal information.

2. **Implement Strong E2EE:** Use proven cryptographic methods to ensure that only the people in a conversation can read the messages. This includes secure key generation and exchange between users.

3. **Guarantee Message Ephemerality:** To design the system so that chat messages are not stored persistently. Message history will be volatile, existing only for the duration of an active chat room session on the client-side.

4. **Enable Real-Time Interaction:** Use WebSocket technology through Socket.io to make sure messages are delivered instantly between users in the same chat room.

5. **Secure Room Access Control:** To implement a system for generating unique, non-guessable room codes and to allow controlled entry based on these codes, including options for setting user limits per room.

6. **Create an Intuitive User Interface:** To build a user-friendly, responsive frontend that simplifies the process of room creation, joining, and secure messaging.

7. **Minimize Data Retention:** To ensure the backend system only manages essential, transient session data for active rooms (e.g., room IDs, participant session identifiers for routing), without storing E2EE private keys or any decrypted message content.

8. **Implement Effective Chat Destruction:** When a chat room ends or everyone leaves, automatically delete all related temporary data from the server and clear message displays from users' browsers.

---

## 3. Project Category

This project spans multiple areas of computer science and software development:

- **Networking Application:** It fundamentally relies on network protocols for real-time, multi-user communication, primarily leveraging WebSockets through Socket.io.

- **Web Application:** The user interface and core functionality will be delivered as a web-based application, accessible via standard browsers, using Next.js for the frontend and Node.js for the backend.

- **Security Application:** Privacy and security are central to the project, involving cryptographic techniques like end-to-end encryption and principles of data minimization and temporary storage.

- **Real-Time Systems:** The chat functionality requires immediate message processing and delivery to create a smooth conversation experience for users.

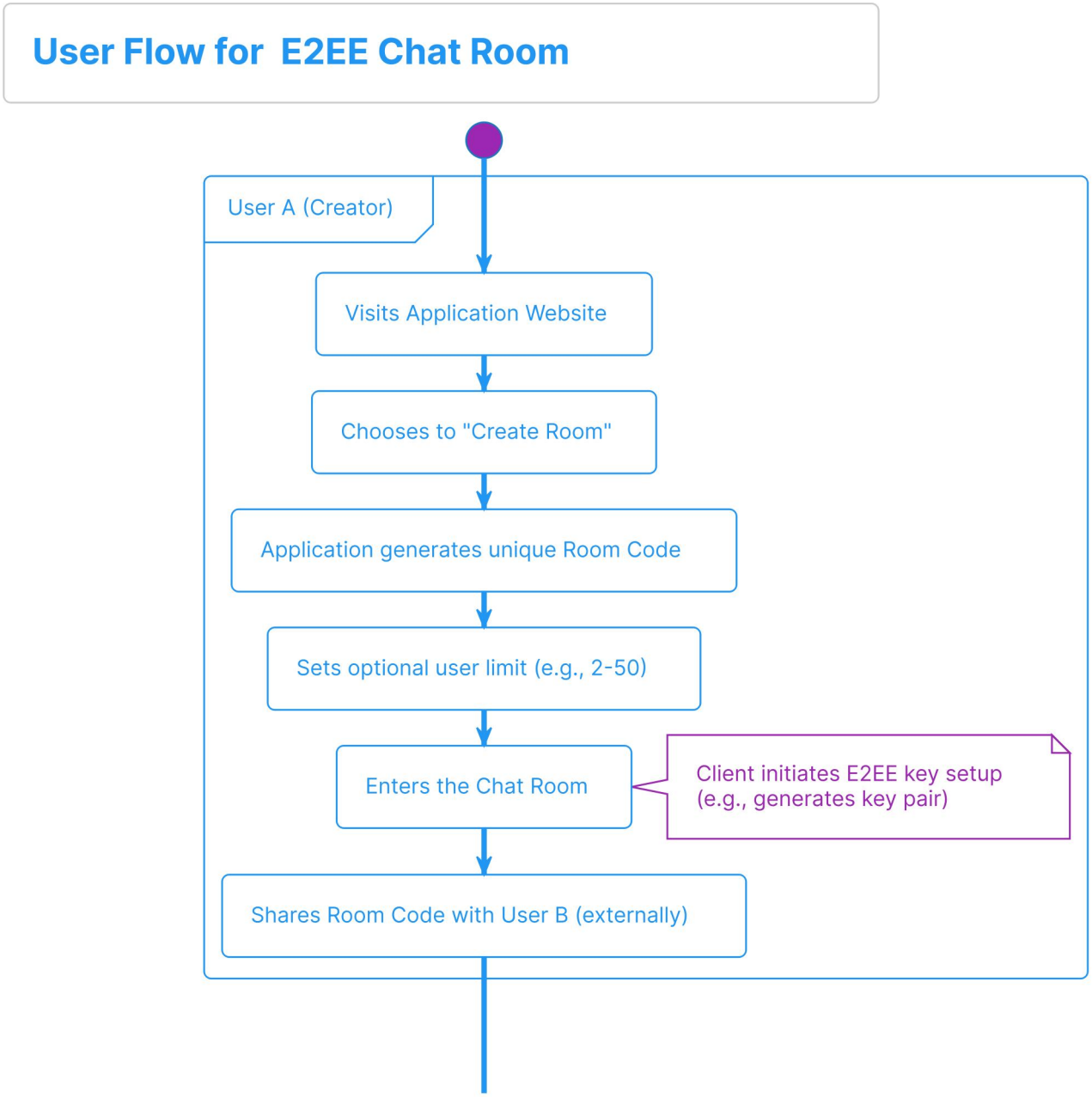
While the project uses MongoDB as a database, it's specifically for temporary session management rather than permanent data storage, which distinguishes it from typical RDBMS-centric or data-intensive applications.

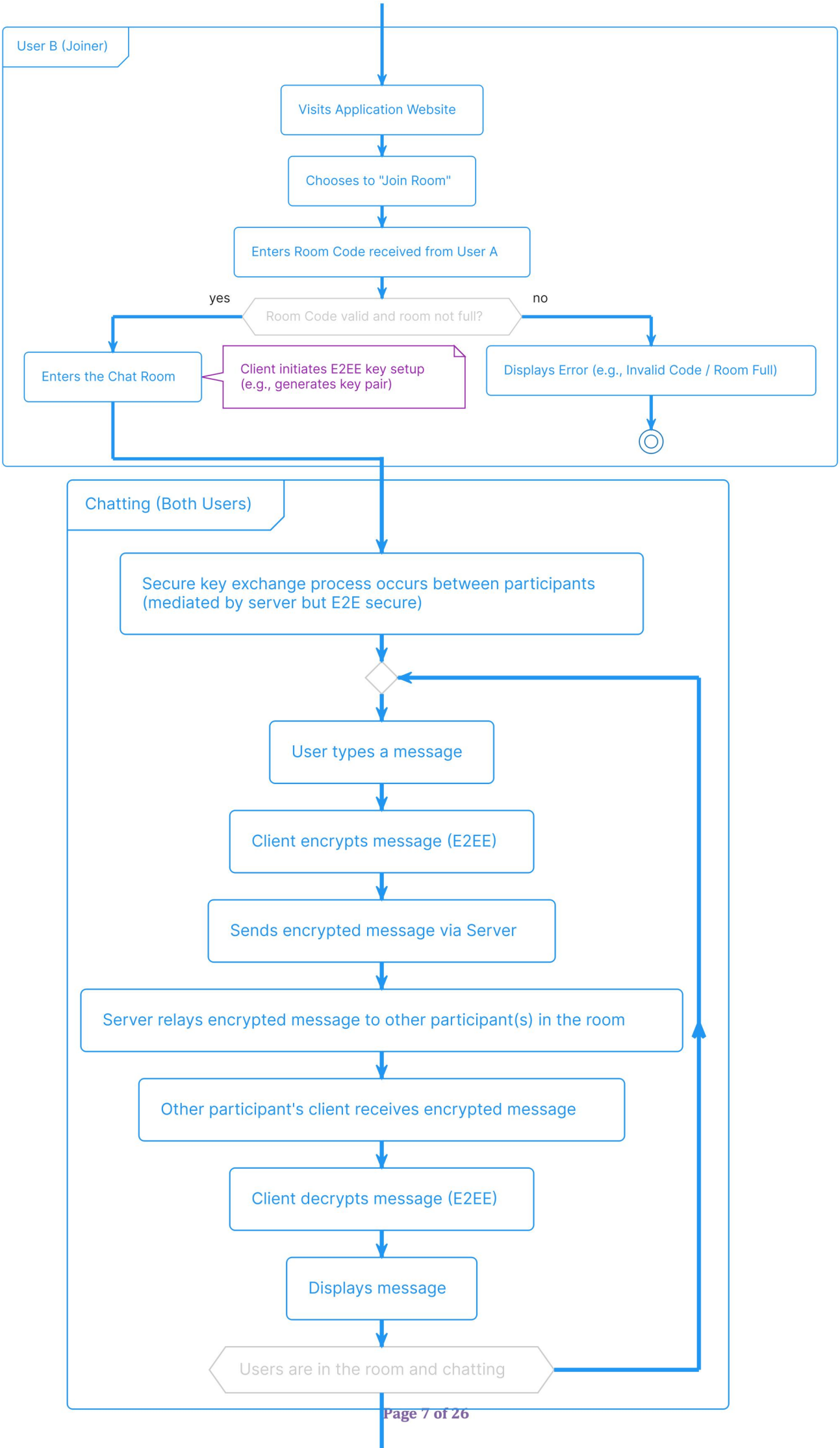
## 4. Analysis

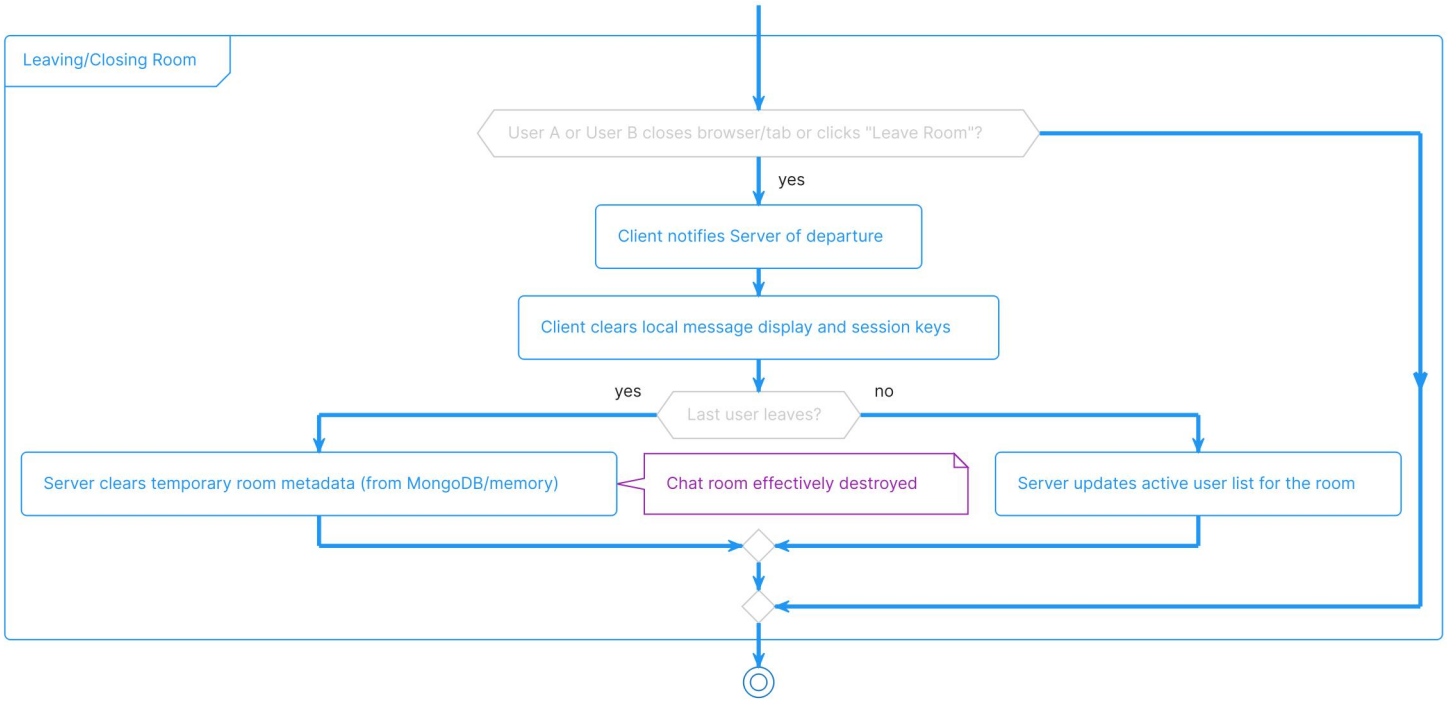
This section outlines the system's operational flow, data interactions, and structural design through various diagrams.

### 4.1 User Flow Diagram (Activity Diagram)

This diagram depicts the typical sequence of actions and decisions a user makes while interacting with the application, from creating/joining a room to participating in a chat and exiting.



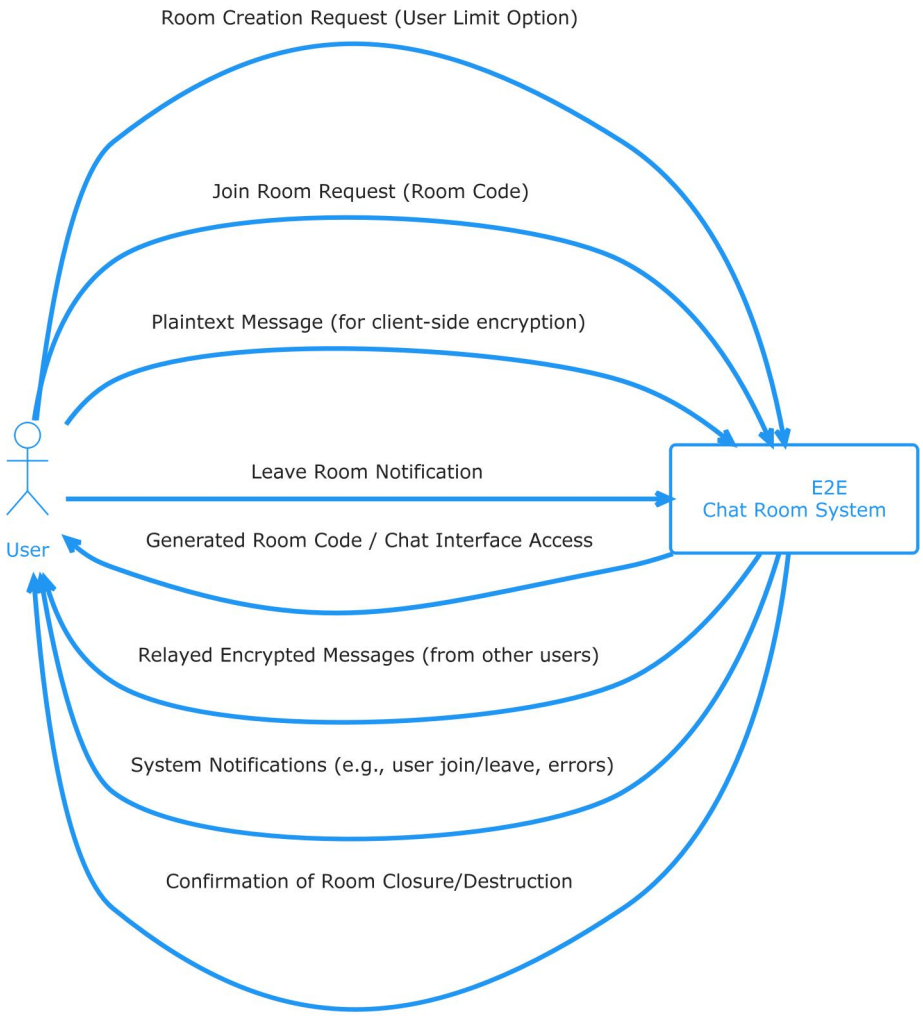

User B (Joiner)
Visits Application Website
Chooses to "Join Room"
Enters Room Code received from User A
yes
Room Code valid and room not full?
no
Enters the Chat Room
Client initiates E2EE key setup
(e.g., generates key pair)
Displays Error (e.g., Invalid Code / Room Full)
Chatting (Both Users)
Secure key exchange process occurs between participants
(mediated by server but E2E secure)
User types a message
Client encrypts message (E2EE)
Sends encrypted message via Server
Server relays encrypted message to other participant(s) in the room
Other participant's client receives encrypted message
Client decrypts message (E2EE)
Displays message
Users are in the room and chatting




**Description of User Flow Diagram:** The diagram illustrates the user journey: one user (Creator) initiates a room, obtains a unique code, and shares it. Another user (Joiner) uses this code to enter. Inside the room, a secure key exchange establishes E2EE. Encrypted messages are then exchanged, relayed by the server but only readable by clients. Upon exit, local data is cleared, and the server eventually purges room metadata if all users depart, ensuring ephemerality.

## 4.2 Context Level DFD (Level 0)

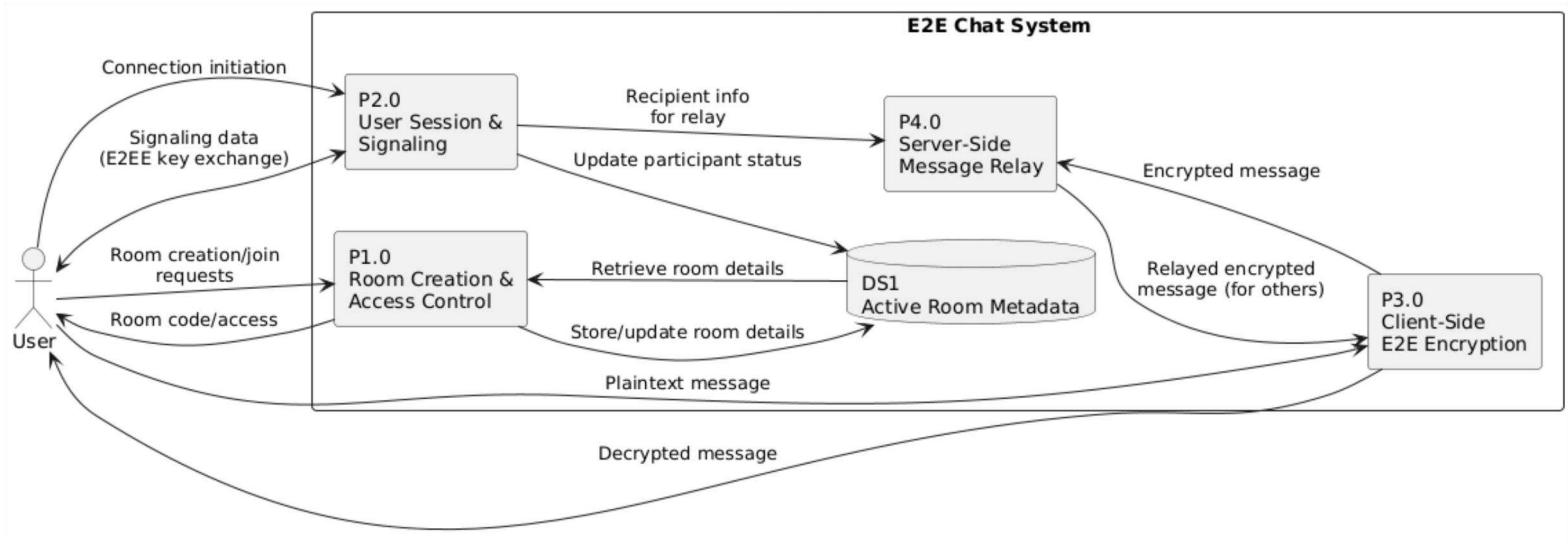
This diagram provides a high-level overview of the system as a single process, showing its inputs from and outputs to the external entity (User).




**Description of Context Level DFD:** Users interact with the " E2E Chat Room System" by providing inputs like room creation requests, room joining requests, and message content (which gets encrypted before leaving their device). The system responds with outputs including room access confirmation, encrypted messages from other users, and status notifications. The main purpose is facilitating secure, temporary communication between users.

## 4.3 Level 1 DFD

This diagram decomposes the system into its major functional processes, data stores, and the data flows between them.



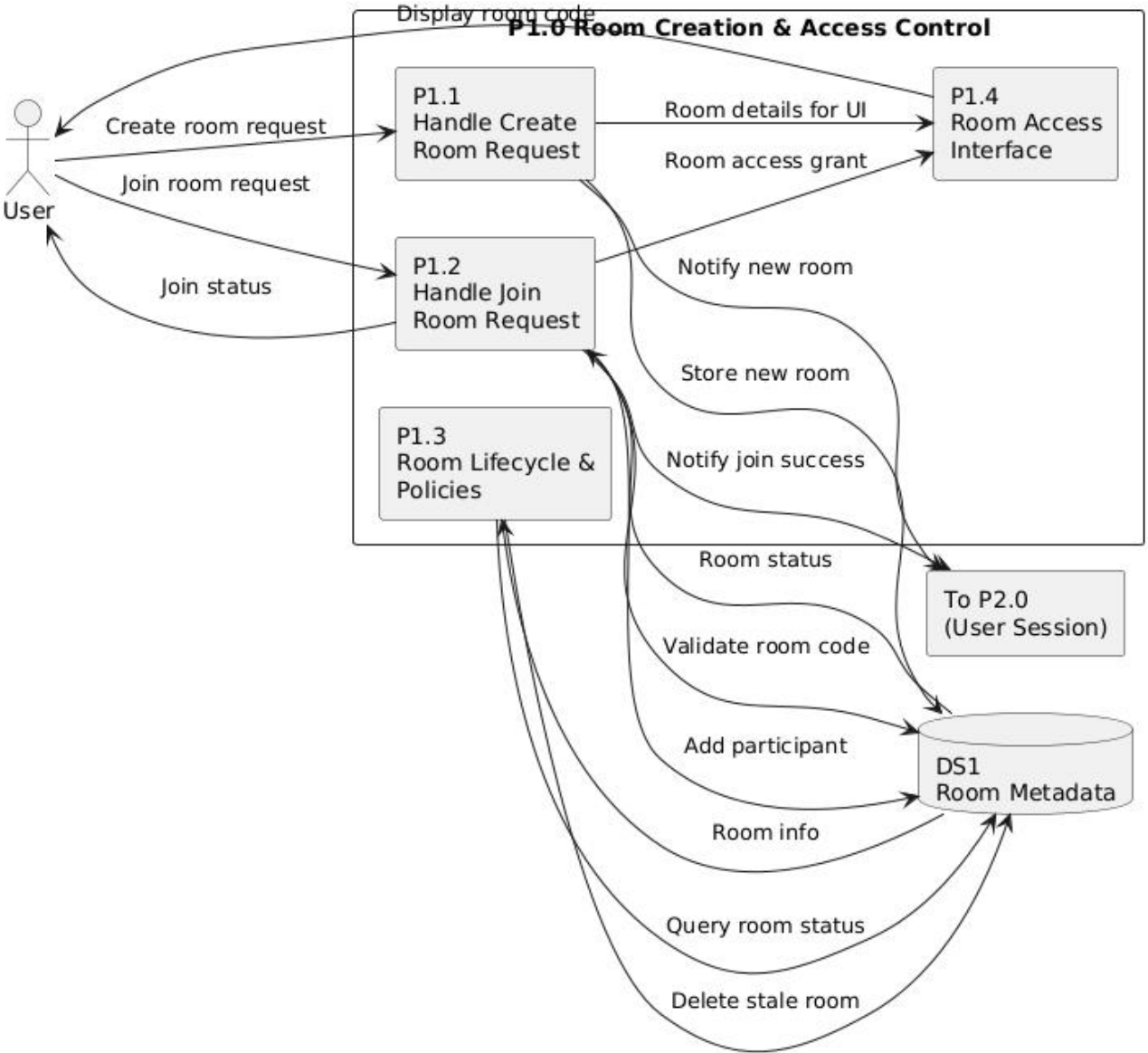
**Description of Level 1 DFD:**

- **P1.0 Room Creation & Access Control:** Manages requests to create new rooms, validates attempts to join existing rooms, and controls access based on room codes and user limits. Interacts with D1 to store and retrieve room status.
- **P2.0 User Session Mgt. & Signaling:** Handles individual user connections to rooms, manages their session lifecycle (join/leave), and crucially, facilitates the exchange of signaling messages required for clients to perform E2EE key establishment.
- **P3.0 Client-Side E2E Encryption/Decryption:** This process resides entirely on the client's device. It encrypts outgoing messages before transmission and decrypts incoming messages after reception, using keys unknown to the server.
- **P4.0 Server-Side Real-time Encrypted Message Relay:** The backend component that receives encrypted messages from one user and forwards them to other authenticated users in the same room. It can't read the message content.
- **D1: Temporary Active Room Metadata Store:** A MongoDB database that holds temporary information about active rooms (like room codes, user limits, and lists of current participants). This data gets deleted when rooms become inactive.

## 4.4 Level 2 DFDs

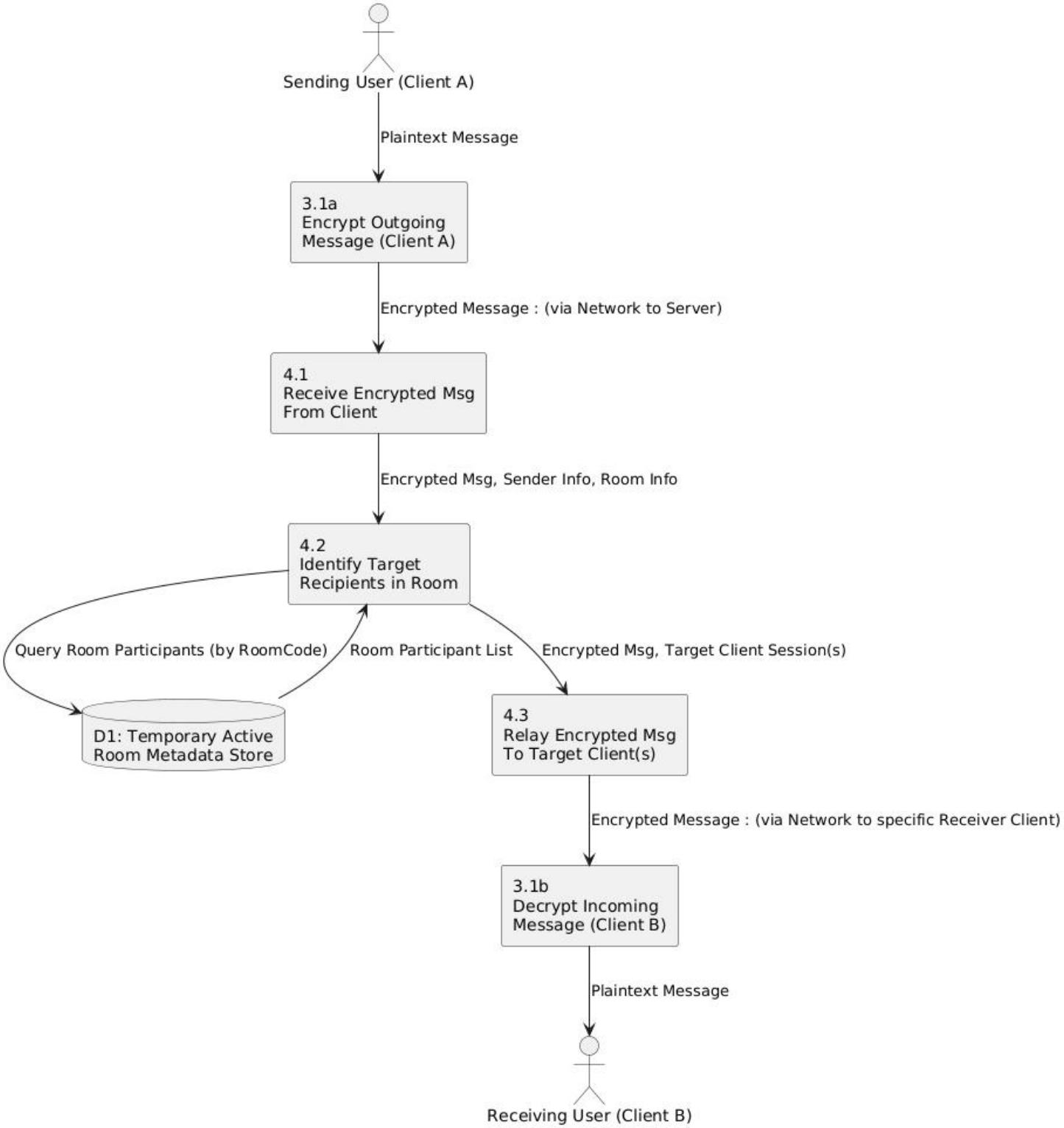
These diagrams provide a more detailed breakdown of selected processes from the Level 1 DFD.

### 4.4.1 Level 2 DFD for Process 1.0: Room Creation & Access Control



### 4.4.2 Level 2 DFD for Message Exchange (Involving P3.0 and P4.0)

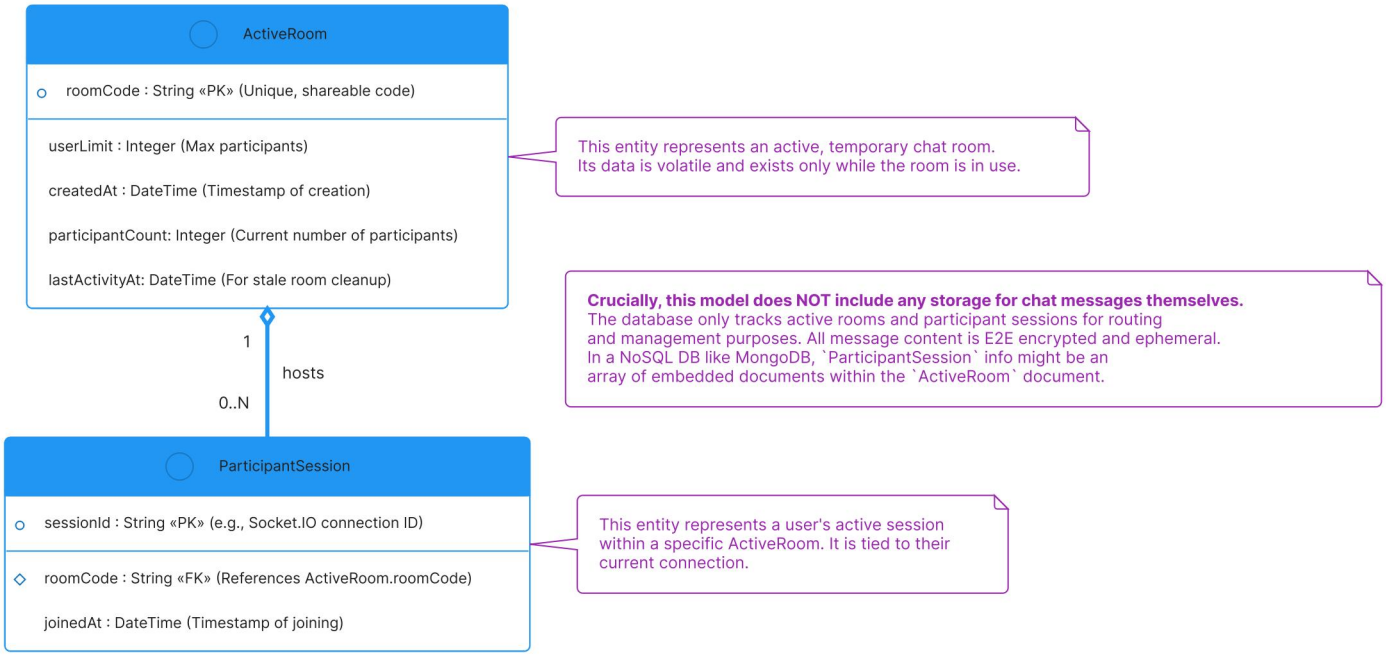
This diagram details the flow of a message from sender to receiver, highlighting client-side encryption/decryption and server-side relay.



## 4.5 Entity Relationship Diagram (ERD) / Database Design (for Temporary Room Data)

This ERD conceptualizes the entities and relationships for the temporary metadata stored by the server (e.g., in MongoDB) to manage active chat room sessions. **No message content is stored.**

**ERD for Temporary Active Room Metadata (Conceptual for MongoDB)**




**Conceptual MongoDB Document Structure (for an `ActiveRooms` collection):**

This example illustrates how an active room's metadata might be structured in a MongoDB document.

```
{
  "_id": "<ObjectId>", // Auto-generated by MongoDB
  "roomCode": "A7B3C9", // Unique, application-generated room identifier
  "userLimit": 25,
  "participantSessions": [ // Array of active participant session details
    { "sessionId": "socket_io_session_id_1", "joinedAt": "ISODate(...)" },
    { "sessionId": "socket_io_session_id_2", "joinedAt": "ISODate(...)" }
  ],
  "createdAt": "ISODate(...)",
  "lastActivityAt": "ISODate(...)" // Updated on new message or join/leave
}
```

This structure would be queried and updated by the backend (Process 1.0 and 2.0) to manage room access and message routing.

# 5. Complete System Structure

## 5.1 Number of Modules and their Description

The application's architecture is modular to promote clarity, maintainability, and testability. Key modules include:

1. **Frontend User Interface (UI) Module (Next.js, Tailwind CSS):**
    - **Description:** This module handles everything users see and interact with. It renders all the pages (home page, room creation forms, chat interface), manages user interactions, displays decrypted messages, and maintains the visual state of the application.

2. **Client-Side End-to-End Encryption (E2EE) Module (Web Crypto API):**
    - **Description:** Operating entirely within each user's browser, this is the security heart of the system. It generates encryption keys, handles secure key exchange with other users in the room, encrypts outgoing messages, and decrypts incoming messages. Importantly, all private keys stay on the user's device and never get sent to the server.

3. **Client-Side Real-Time Communication Module (Socket.io Client):**
    - **Description:** This module manages the persistent connection between the user's browser and the server. It handles sending encrypted messages to the server and receiving relayed encrypted messages and setup signals from the server.

4. **Backend Room Management & Signaling Server Module (Node.js, Socket.io):**
    - **Description:** The server-side component that coordinates chat room operations. It processes room creation requests, validates room joining attempts, manages active room lifecycles, helps facilitate encrypted key exchange between clients, and relays encrypted messages. Crucially, it never has access to actual message content or users' private keys.

5. **Temporary Room Metadata Storage Module (MongoDB Driver & Logic):**
    - **Description:** This backend module provides the connection and operations for interacting with MongoDB, which stores temporary information about active chat rooms (room codes, user limits, participant lists) but never stores messages or encryption keys. CRUD operations (Create, Read, Update, Delete) on temporary room metadata, ensuring data is available for room management and cleared upon room inactivity.

## 5.2 Data Structures

The application will utilize various data structures, both client-side and server-side, to manage state and information flow:

- **Client-Side Structures:**
  - `currentRoomDetails`: An object holding details of the room the client is currently in (e.g., `{ roomCode: string, userLimit: number, participantNicknames: string[] }`).
  - `displayedMessages`: An array of message objects shown in the chat interface, including sender name, decrypted content, and timestamp. This gets cleared when leaving the room.
  - `sessionEncryptionContext`: Securely holds the cryptographic keys for the current chat session. This exists only in memory and gets cleared when the session ends.
  - `roomParticipantsInfo`: A Map or array storing temporary information about other participants in the room, potentially including their public key fragments if needed during the key exchange phase.
- **Server-Side Structures (Node.js/Socket.io - In-Memory for active sessions, often synced/backed by DB):**
  - `activeRoomsData`: A JavaScript Map where keys are room codes and values contain room details like user limits, sets of connected user IDs, and activity timestamps.
- **MongoDB Document Structure (for `ActiveRooms` collection):**
  - Fields include `roomCode`, `userLimit`, an array of `participantSessions` (each with `sessionId`, `joinedAt`), `createdAt`, `lastActivityAt`.
- **Network Message Formats (Conceptual):**
  - **Encrypted Chat Message (Client <-> Server):** A object like `{ roomCode: string, encryptedPayload: string (Base64 encoded ciphertext + IV/nonce), senderSessionId?: string }`.
  - **Signaling Message (Client <-> Server, for E2EE key exchange):** A object like `{ roomCode: string, signalType: string (e.g., 'offer', 'answer', 'candidate'), signalPayload: object, targetSessionId?: string }`. The `signalPayload` structure depends on the chosen key exchange protocol.

## 5.3 Process Logic of Each Module

This section describes the core logic flow for each module.

1. **Frontend UI Module:**
    - **Room Creation:** User clicks create → UI prompts for optional user limit → sends request to communication module → receives room code from server → displays code and navigates to chat interface.
    - **Room Joining:** User inputs `roomCode` -> UI triggers "join room" action with code to Communication Client. On success/failure from server, UI navigates to chat or displays error.
    - **Message Display:** Receives decrypted message object (from E2EE Module) -> appends to chat view with appropriate styling (sender, timestamp).
    - **Sending Message:** User types message -> UI captures input -> passes plaintext to E2EE Module for encryption.
    - **Leaving Room:** User clicks "leave" or closes tab -> UI triggers "leave room" action to Communication Client -> clears local message display and any session-specific E2EE keys.
2. **Client-Side E2EE Module:**
    - **Initialization (on room entry):** Generates necessary cryptographic material.
    - **Key Exchange:** Sends public key information to other users through the server, receives their public keys, computes shared secrets, derives symmetric session keys.
    - **Message Encryption:** Takes plaintext from UI → uses session key to encrypt with unique nonce/IV → returns ciphertext to communication module.
    - **Message Decryption:** Takes ciphertext from communication module → uses session key to decrypt → returns plaintext to UI module.
3. **Client-Side Real-Time Communication Module:**
    - **Connection Management:** Establishes and maintains WebSocket connection to Socket.io server upon entering a room context. Handles reconnect logic if necessary.
    - **Event Emission:** Sends structured events to server:
        - create_room_request (with user limit)
        - join_room_request (with roomCode)
        - encrypted_message_to_server (with roomCode, encryptedPayload)

- key_exchange_signal_to_server (with roomCode, signalType, signalPayload, targetSessionId if applicable)
- leave_room_notification

- **Event Listening:** Handles events from server:
  - room_created_success (with roomCode)
  - join_room_status (success or error message)
  - new_encrypted_message_from_server (passes encryptedPayload to E2EE Module)
  - key_exchange_signal_from_server (passes signalPayload to E2E Module)
  - user_joined_room_notification, user_left_room_notification (for UI updates)

4. **Backend Room Management & Signaling Server Module:**

   - **On `create_room_request`:** Generates unique room code → creates database entry → adds creator to Socket.io room → confirms creation to user.

   - **On `join_room_request`:** Validates room code and capacity → adds user to Socket.io room and database → notifies user and others in room.

   - **On `encrypted_message_to_server`:** Receives encrypted message → broadcasts to other users in same room without decrypting.

   - **On `key_exchange_signal_to_server`**Receives setup signals → forwards to appropriate recipients without interpreting content.

   - **On client `disconnect` or `leave_room_notification`:** Removes user from room → updates database → notifies remaining users → deletes room if empty.

5. **Temporary Room Metadata Storage Module:**

   - **`createRoom(details)`:** Inserts a new document into the `ActiveRooms` MongoDB collection.
   - **`findRoomByCode(roomCode)`:** Retrieves a room document.
   - **`addParticipantToRoom(roomCode, sessionId)`:** Updates the specified room document to add a participant.
   - **`removeParticipantFromRoom(roomCode, sessionId)`:** Updates room document to remove a participant.
   - **`deleteRoom(roomCode)`:** Deletes a room document.
   - **`getParticipantCount(roomCode)`:** Returns current number of participants.

## 5.4 Testing Process to be Used

A multi-layered testing strategy will be implemented to ensure application quality, security, and reliability:

1. **Unit Testing:**
    - Focus on individual functions and components in isolation.
    - **Client-Side:** Test encryption/decryption functions with known test vectors, test React components for proper rendering and state management.
    - **Server-Side:** Test individual Socket.io event handlers and helper functions with mocked dependencies.

2. **Integration Testing:**
    - Verify interactions between different modules.
    - **Client-Server:** Test the complete flow of Socket.io events between client and server for room creation, joining, message relay, and signaling.
    - **Module Interactions:** Test Frontend UI <-> E2EE Module, Backend Server <-> MongoDB Storage Module.

3. **End-to-End (E2E) Testing:**
    - Simulate real user scenarios from start to finish using browser automation tools (e.g., Cypress, Playwright).
    - **Key Scenarios:** User A creates a room, User B joins; both exchange multiple encrypted messages; one user leaves, then the other; attempts to join full/invalid rooms. Verify message display and ephemerality.

4. **Security Testing:**
    - **E2EE Verification:** Manually inspect network traffic using browser developer tools to confirm all transmitted data is properly encrypted.
    - **Vulnerability Assessment:** Check for common web security issues and assess room code generation strength.
    - **Logical Flaw Detection:** Review logic for key exchange and session management for potential weaknesses.

5. **Usability Testing:**
    - Gather qualitative feedback from a small group of test users regarding the application's ease of use, clarity of instructions, and overall user experience.

**Primary Testing Tools:**

- ► Jest and React Testing Library (for frontend unit/integration)
- ► Jest or Mocha/Chai (for backend unit/integration)
- ► Cypress or Playwright (for E2E tests)
- ► Browser Developer Tools

## 5.5 Reports Generation

Given the application's core principles of ephemerality and privacy, traditional report generation is intentionally minimal and avoids storing sensitive user data:

1. **Client-Side Debugging Logs (Developer-Enabled):**
    - **Content:** Timestamps of significant client-side events (e.g., “Room joined: XYZ”, “Key exchange step 1 complete”, “Message encrypted”, “Error: Decryption failed”). **Strictly no message content or cryptographic key material will be logged.**
    - **Purpose:** For developers or advanced users to diagnose local issues related to connectivity, E2EE setup, or UI errors.
    - **Generation:** Implemented via `console.log` or a lightweight client-side logging utility, typically enabled via a browser console command or a debug flag in development builds.

2. **Server-Side Operational Logs (Anonymized ):**
    - **Content:** Event timestamps, server operations (room created, user joined, message relayed), anonymized room identifiers, error codes and stack traces, aggregate metrics (active connections, active rooms)**.**
    - **Purpose:** For system administrators/developers to monitor server health, performance, identify bottlenecks, track error rates, and debug server-side operational issues.
    - **Generation:** Using a robust logging library (e.g., Winston, Pino) in the Node.js backend, with configurable log levels.

3. **Ephemeral Session “Report” (User Interface):**
    - **Content:** The dynamically rendered chat messages displayed within the user’s active browser session.
    - **Purpose:** This is the primary “report” visible to the user – their live conversation.
    - **Generation:** This “report” is the application’s core user interface. It is ephemeral by design; when the user leaves the room or closes the browser tab/window, this displayed information is cleared from their client.

**No persistent reports containing chat message content, user identities (beyond temporary session identifiers), or detailed user activity logs will be generated or stored by the server.** The system prioritizes not collecting data that doesn’t absolutely need to be collected.

# 6. Tools / Platform, Hardware and Software Requirement Specifications

## 6.1 Software Requirements

- **Frontend Development:**
  - Programming Languages: JavaScript (ES6+), TypeScript
  - Core Framework/Library: Next.js (React-based framework)
  - Styling: Tailwind CSS
  - Client-Side Cryptography: Web Crypto API (built into modern browsers)
  - Real-Time Client Communication: Socket.io Client library
  - Package Management: bun (Node Package Manager)
  - Version Control System: Git

- **Backend Development:**
  - Runtime Environment: Node.js
  - Real-Time Server Framework: Socket.io
  - Programming Languages: JavaScript (ES6+), TypeScript
  - Database: MongoDB (for temporary room metadata)
  - Package Management: bun

- **Development Environment:**
  - Operating System: Windows 11
  - Code Editor/IDE: Visual Studio Code
  - Web Browsers (for development & testing): Latest stable versions of Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.
  - Terminal/Command Line Interface: For running scripts, Git commands, etc.

- **Deployment Environment (Server-Side):**
  - Operating System: Linux-based OS (e.g., Ubuntu Server) is standard for Node.js.
  - Process Manager (for Node.js application): PM2, or systemd.
  - Database Server: MongoDB instance.
  - Cloud Platform: Vercel (ideal for Next.js).

- **User Environment (Client-Side):**
  - Operating System: Any modern OS capable of running current web browsers (Windows, macOS, Linux, Android, iOS).
  - Web Browser: Latest stable versions of Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, with full support for WebSockets and the Web Crypto API.

## 6.2 Hardware Requirements

► **Development Machine:**
- Processor: Multi-core processor AMD Ryzen 5
- RAM: Minimum 8 GB .
- Storage: 100 GB free disk space SSD.
- Network: Broadband internet connection.

► **Server Machine (for Deployment - indicative for a small to moderate load):**
- Processor: 1-2+ vCPUs (AWS t3.small/medium equivalent).
- RAM: 2-8 GB (depends on concurrent user load - Socket.io can be memory-intensive).
- Storage: 20 GB - 50 GB+ SSD (for OS, application, logs, and MongoDB data if co-hosted).
- Network: Reliable connection with sufficient bandwidth for real-time WebSocket traffic.

► **User Machine (Client-Side):**

- **Standard Requirements:** Any desktop, laptop, tablet, or smartphone from the last 5-7 years
- **Processor:** Any modern CPU capable of handling JavaScript execution for encryption without significant delay
- **RAM:** 2 GB minimum (more recommended for better browser performance)
- **Network:** Stable internet connection (WiFi, Ethernet, or reliable mobile data)

---

## 7. Industry/Client Affiliation

**No.**

This " End-to-End Encrypted Chat Room Application" is being developed purely as an academic project. It is intended to fulfill educational requirements and explore concepts in secure web application development. It is not commissioned by, associated with, or undertaken for any specific industry, client, or commercial organization.

## 8. Future Scope and Further Enhancement of the Project

While the initial version focuses on core secure chat functionality, several enhancements could be added in future iterations:

1. **Advanced Room Controls:**

    - **Room Passwords:** Add optional password protection in addition to room codes.
    - **Moderation Tools:** Give room creators basic moderation tools (mute or remove disruptive users).
    - **Customizable User Limits:** Allow room creators to adjust user limits during active sessions

    .

2. **Rich Media & Interaction (E2EE):**

    - **Encrypted File Sharing:** Allow secure file transfers within chat rooms.
    - **Markdown/Rich Text Formatting:** Support for basic message formatting to improve readability and expression.
    - **Emoji Reactions:** Allow users to react to messages with emojis.
    - **E2EE Voice/Video Calls:** Integration of WebRTC for encrypted peer-to-peer voice and video calls

    .

3. **User Experience Improvements:**

    - **Typing Indicators:** Securely implement indicators to show when other users are typing.
    - **Read Receipts (Optional & E2EE):** A privacy-conscious implementation of message read receipts.
    - **UI Themes & Personalization:** Allow users to choose different visual themes.
    - **Improved Notification System:** More refined in-app notifications.

4. **Advanced Security Features:**

    - **Key Verification Mechanisms:** Allow users to verify each other's identities through safety numbers or QR code scanning.
    - **Formal Security Audit:** Engage professional security reviewers to assess the cryptographic implementation.

5. **Scalability and Performance:**

    - **Horizontal Scaling for Socket.io:** mplement Redis adapter for Socket.io to handle more concurrent users across multiple server instances.
    - **Optimized Message Broadcasting:** More efficient message delivery mechanisms for very large rooms (if user limits are increased).

These potential enhancements would progressively build upon the foundational secure and ephemeral chat system, adding value and utility.

---

## 9. Limitations of the Project (Initial Version)

The initial development phase will focus on delivering core functionality, and as such, certain limitations will exist:

1. **No User Accounts:** The application operates without traditional registration or login systems. Users remain anonymous within each chat session.
2. **No Message History:** All conversations are temporary. Messages aren't saved anywhere and disappear when rooms close or users leave.
3. **Text Messages Only:** Initial version supports only text-based communication. File sharing, voice messages, or video calls aren't included yet.
4. **Basic Key Exchange:** While end-to-end encryption is implemented, advanced features like Perfect Forward Secrecy or explicit key fingerprint verification aren't included initially.
5. **Room Code Security:** Room security relies primarily on keeping the generated codes secret. While codes are designed to be hard to guess, additional security measures against code compromise aren't a primary focus initially.
6. **Single Server Focus:** The backend architecture is optimized for single server deployment. Horizontal scaling strategies are considered future enhancements.
7. **No Offline Support:** Users must be actively connected to send or receive messages. There's no message queuing for offline users.
8. **Trust in Client Code:** The effectiveness of encryption depends on the integrity of JavaScript code running in users' browsers. Users must trust that this code correctly implements encryption and doesn't compromise security.
9. **Limited Room Management:** Initial version doesn't include features for room creators to manage participants (like kicking or banning users).
10. **Modern Browser Dependency:** Requires recent browser versions with WebSocket and Web Crypto API support, which may limit compatibility with very old browsers.

These limitations help keep the project scope manageable while ensuring robust implementation of core security and communication features.

---

## 10. Security and Validation Checks

Security is a foundational requirement of this application. The following checks, principles, and validations will be integral to its design and implementation:

1. **End-to-End Encryption (E2EE) Implementation:**
    - **Core:** All user-to-user message content will be encrypted on the sender's client device and decrypted only on the recipient(s)' client device(s).
    - **Algorithms:**ndustry-standard cryptography using AES-256-GCM for message encryption and secure key exchange protocols like Diffie-Hellman.
    - **Key Management:** ll private keys and session keys are generated and stored only on client devices - never transmitted to or stored by the server.
2. **Message and Data Ephemerality:**
    - **No Server-Side Message Storage:** The server will not store any chat message content, either in plaintext or ciphertext form, in any database or persistent logs.
    - **Client-Side Data Clearing:** When users leave rooms, their browsers automatically clear displayed messages and encryption keys from active Java memory.
    - **Temporary Room Metadata Purging:** Server-side metadata related to active chat rooms (e.g., room ID, list of active participant sessions) will be actively deleted from the temporary store (MongoDB/memory) once a room becomes empty or after a defined period of inactivity.
3. **Secure Room Access and Control:**
    - **Unique and Complex Room Codes:** Chat rooms will be accessed via unique, randomly generated codes of sufficient complexity to make guessing impractical.
    - **Server-Side Validation:** The backend server will rigorously validate room codes and enforce any defined user limits before granting a user access to a chat room.
    - **Rate Limiting (Consideration):** Basic rate limiting on attempts to join rooms may be implemented on the server-side to mitigate brute-force attacks on room codes.
4. **Input Validation (Client-Side and Server-Side):**

- **Data Sanitization:** All user input is validated and sanitized on both client and server sides to prevent injection attacks.

- **Socket.io Event Payload Validation:** Socket.io event payloads are validated to ensure they conform to expected formats.

5. **Transport Layer Security (TLS/SSL):**

    - All communication between the client's browser and the web server (serving the Next.js application) and the Socket.io server will be enforced over HTTPS and WSS (Secure WebSockets) respectively. This protects the already E2E-encrypted message payloads and critical signaling messages while they are in transit to/from the server.

6. **Protection Against Common Web Vulnerabilities:**

    - **Cross-Site Scripting (XSS):** Although message content is E2E encrypted, any user-generated input that might be displayed directly by the UI (e.g., user-chosen temporary nicknames, if implemented) will be properly escaped/sanitized by the frontend framework (Next.js/React) to prevent XSS.
    - **Secure Headers:** Implement appropriate HTTP security headers (e.g., Content Security Policy, X-Content-Type-Options).

7. **Secure Code Practices:**

    - **Dependency Management:** Regularly update all third-party libraries and dependencies (both frontend and backend) to patch known vulnerabilities, using tools like `npm audit`.
    - **Principle of Least Privilege:** Server-side processes will operate with the minimum necessary permissions.

8. **Signaling Channel Security:**

    - Ensure that signaling messages (used for E2EE key exchange setup) are relayed correctly only to the intended participants within a specific room and are protected in transit by WSS.

**Validation Approach:**

- ► Conduct regular code reviews with a focus on security implementation details.
- ► Perform manual security testing, including attempts to bypass E2EE and inspecting network traffic.
- ► Utilize browser developer tools for examining client-side data handling and storage.
- ► Write automated tests specifically targeting the encryption/decryption logic and key exchange process.

---

## 11. Bibliography (References)


**Cryptography and Security:**

1. OWASP Foundation. (2021). *OWASP Top Ten Web Application Security Risks*. Retrieved from owasp.org/www-project-top-ten/
2. Mozilla Developer Network (MDN). *Web Crypto API*. Retrieved from developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

**Web Technologies and Development:**

3. Next.js Official Documentation. Vercel. Retrieved from nextjs.org/docs
4. Socket.IO Official Documentation. Retrieved from socket.io/docs/
5. Tailwind CSS Official Documentation. Tailwind Labs. Retrieved from tailwindcss.com/docs


---

# Index of comments