Name: Jinze Huang
Date: May 16, 2025
Section: CSCI-GA.3033-026

Course Project

Weekly Update Date: May 16, 2025
Team Members: Jinze Huang

Answers to Individual Questions: points / **100 points**

Professor's Comments:

1 Team Members and Links

- Jinze Huang @jh9108
- Github: https://github.com/RogerHuangPKX/cloudcomputing25

2 Introduction

The contemporary digital communication landscape is predominantly characterized by centralized platforms. While offering convenience, these architectures often introduce concerns related to data sovereignty, censorship, and susceptibility to single points of failure. Peer-to-Peer (P2P) paradigms present a compelling alternative, fostering direct user-to-user communication, thereby enhancing resilience and user privacy. This project undertakes the design, implementation, and analysis of a sophisticated P2P text chat application, aiming to bridge the gap between the robustness of P2P networks and the feature set expected of modern communication tools.

The primary objective is the creation of a resilient and scalable chat system that synergizes the intrinsic benefits of P2P networking (e.g., fault tolerance, reduced server load for message relay) with essential functionalities. These include dynamic multi-room capabilities with granular access control (public vs. password-protected rooms), persistent message history leveraging cloud database solutions, and an intelligent, on-demand AI assistant for individual users. A critical addition to the typical P2P model is a dedicated administrative panel, providing crucial oversight and data management capabilities—a feature often absent in purely decentralized systems but vital for operational viability and content moderation in many contexts.

3 Cloud Services

3.1 Database

The database is hosted on Aliyun RDS postgreSQL instance, which is a reliable and scalable cloud database service.

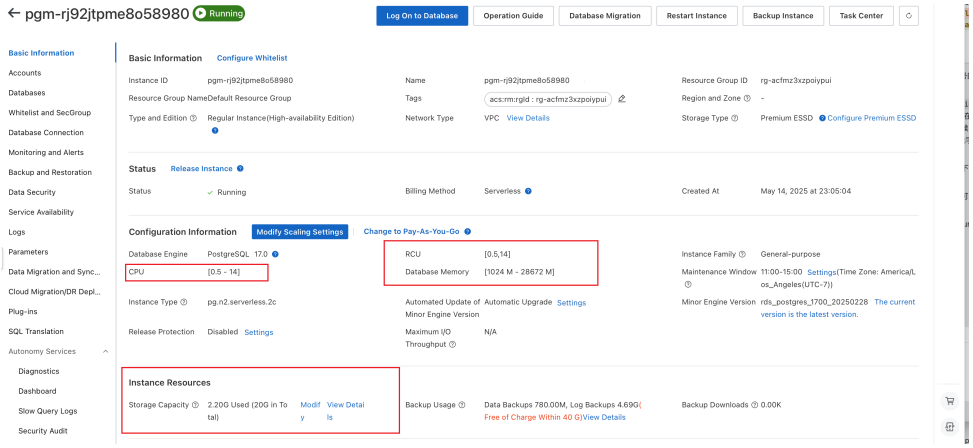
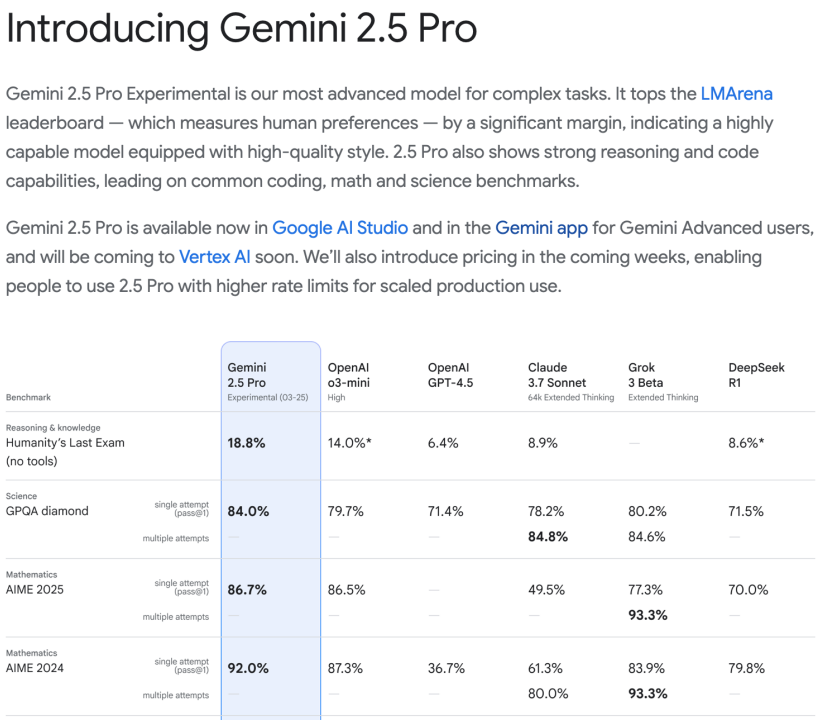


Figure 1: Database

3.2 AI Service

The AI service is hosted on Google Cloud AI Platform, which is a reliable and scalable AI service. I used the Gemini 2.5 Pro model for this project.



Introducing Gemini 2.5 Pro

Gemini 2.5 Pro Experimental is our most advanced model for complex tasks. It tops the LMArena leaderboard — which measures human preferences — by a significant margin, indicating a highly capable model equipped with high-quality style. 2.5 Pro also shows strong reasoning and code capabilities, leading on common coding, math and science benchmarks.

Gemini 2.5 Pro is available now in Google AI Studio and in the Gemini app for Gemini Advanced users, and will be coming to Vertex AI soon. We'll also introduce pricing in the coming weeks, enabling people to use 2.5 Pro with higher rate limits for scaled production use.

Benchmark		Gemini 2.5 Pro Experimental (03-25)	OpenAI o3-mini High	OpenAI GPT-4.5	Claude 3.7 Sonnet 64k Extended Thinking	Grok 3 Beta Extended Thinking	DeepSeek R1
Reasoning & knowledge Humanity's Last Exam (no tools)		**18.8%**	14.0%*	6.4%	8.9%	—	8.6%*
Science GPQA diamond	single attempt (pass@1)	**84.0%**	79.7%	71.4%	78.2%	80.2%	71.5%
	multiple attempts	—	—	—	**84.8%**	84.6%	—
Mathematics AIME 2025	single attempt (pass@1)	**86.7%**	86.5%	—	49.5%	77.3%	70.0%
	multiple attempts	—	—	—	—	**93.3%**	—
Mathematics AIME 2024	single attempt (pass@1)	**92.0%**	87.3%	36.7%	61.3%	83.9%	79.8%
	multiple attempts	—	—	—	80.0%	**93.3%**	—

Figure 2: AI Service

3.3 Bootstrap Server

The bootstrap server is hosted on Azure, which is a reliable and scalable cloud server.

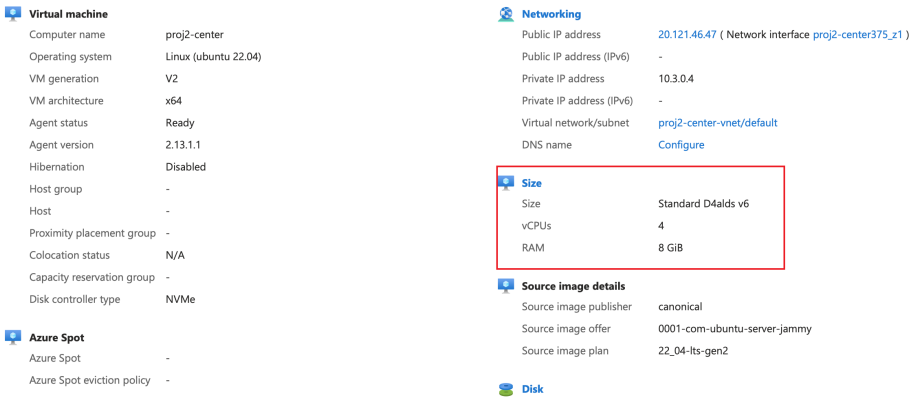


Figure 3: Bootstrap Server

4 System Architecture

The whole system architecture is shown in the following diagram:

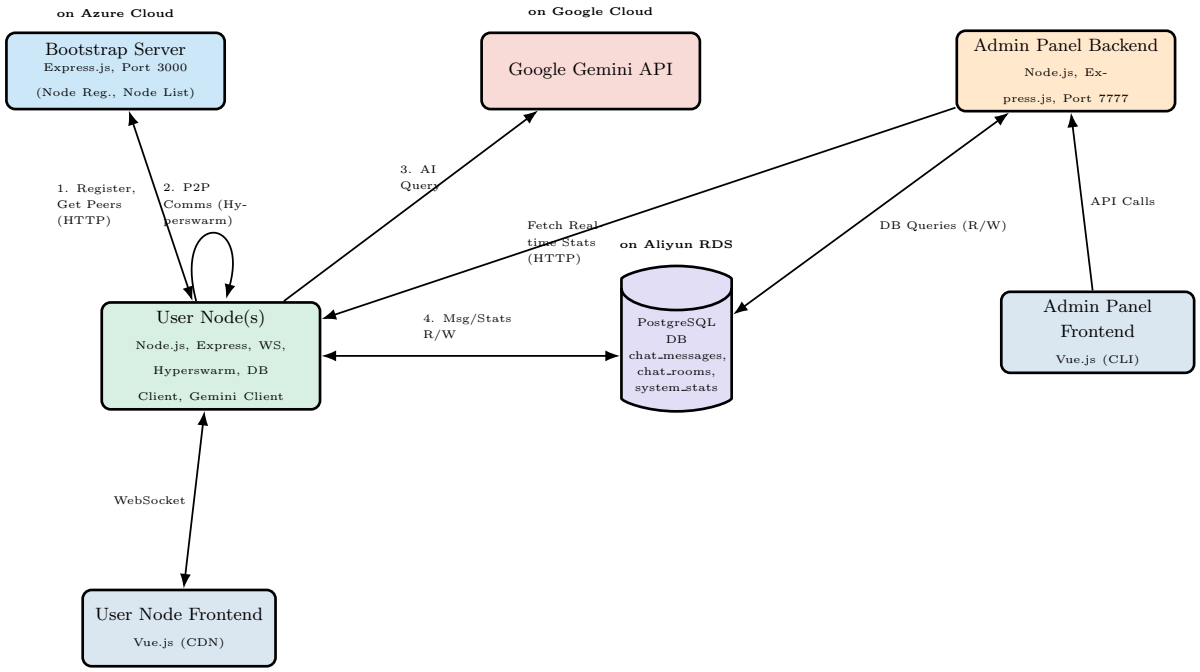


Figure 4: System Architecture Diagram

4.1 High-Level Overview

The system is architected as a distributed network of User Nodes, which form the core of the **P2P** chat functionality. These nodes engage in direct peer-to-peer communication for exchanging chat messages within specific, dynamically generated "Topics" (representing chat rooms).

A centralized **Bootstrap Server** plays a crucial role in the initial peer discovery phase, helping nodes find each other within the P2P network.

User Nodes also interface with a cloud-hosted **PostgreSQL database** for persistent storage of message history and retrieval of historical conversations. Each User Node is enhanced with a private AI assistant feature, achieved through direct API calls to the Google Gemini API.

Complementing the user-facing components, a distinct **Admin Panel**—itself composed of a backend API and a frontend web application—provides administrators with tools to manage chat messages, oversee room activities, and monitor system-wide statistics. This panel interacts with both the PostgreSQL database and a dedicated statistics API exposed by the User Nodes.

4.2 Design Philosophy and Trade-offs

The system's design is guided by several core principles, with an acknowledgment of the inherent trade-offs:

- **Decentralization for Core Chat**: Prioritizing P2P communication for chat messages using Hyperswarm aims to enhance user privacy for message content and reduce reliance on central servers for message relay. This improves resilience against single points of failure for the chat functionality itself. The trade-off is increased complexity in areas like NAT traversal and global message ordering.
- **Strategic Centralization**: Employing centralized services for specific functions addresses common P2P challenges:

  - **Bootstrap Server**: Simplifies initial peer discovery, especially for nodes behind restrictive NATs. The trade-off is a single point of failure for discovery if the bootstrap server is down, though connected peers can continue communicating.
  - **Cloud Database for Persistence**: Ensures reliable, long-term storage and retrieval of message history, which is difficult to guarantee in a purely P2P, transient network. This centralizes message data, which is a trade-off against complete data decentralization but offers practical benefits for history and moderation.

- **Modularity and Encapsulation**: Designing components (User Node, P2P module, Database module, AI service integration, Admin Panel) with well-defined responsibilities and APIs (WebSocket, HTTP, internal events) promotes maintainability, testability, and independent development.
- **Scalability and Maintainability**: Utilizing Node.js for its asynchronous, event-driven architecture, along with established libraries and design patterns, aims for a codebase that can be scaled and maintained effectively. Scalability of individual User Nodes will depend on instance resources, while the P2P nature distributes chat load.

4.3 Core Technologies Employed

Table 1: Core Technologies Employed

Category	**Technologies**
Backend Services (User Node, Bootstrap Server, Admin Backend)	Node.js (v18.x), Express.js (for HTTP API routing and serving static files)
Real-time Communication (User Node Frontend-Backend)	WebSockets, implemented using the ‘ws‘ library in Node.js
P2P Networking Layer	‘hyperswarm‘ library (utilizes DHT, attempts NAT traversal)
Database System	PostgreSQL (Cloud-hosted, e.g., Aliyun RDS, AWS RDS), ‘pg‘ client library for Node.js
AI Assistance	Google Gemini API (‘gemini-2.0-flash‘ model), ‘@google/genai‘ Node.js SDK
User Node Frontend	Vue.js (v3.x via CDN), TailwindCSS (CDN), Marked.js (CDN for Markdown)
Admin Panel Frontend	Vue.js 3 (via Vue CLI), Vue Router
Process Management and Monitoring	‘pm2‘
Security Utilities (Admin Panel)	‘bcryptjs‘ (password hashing), ‘jsonwebtoken‘ (simplified usage)

4.4 Detailed P2P Network Architecture

The P2P network relies on the **‘hyperswarm‘** library, utilizing a Kademlia-like DHT (Distributed Hash Table) for peer discovery. Nodes announce and discover topics (chat rooms) on this DHT.

Key P2P Aspects:

- **Peer Discovery & NAT Traversal**: Hyperswarm uses a DHT (Mainline DHT from BitTorrent) for discovering peers interested in the same topic. It attempts NAT traversal (e.g., hole punching) for direct TCP/UTP connections. If direct connection fails, it might use relaying, though our Bootstrap Server primarily aids initial peer list gathering.

- **Connection Encryption**: Connections are end-to-end encrypted using the Noise Protocol Framework (XX handshake). Let K_A be the keypair of Alice and K_B be the keypair of Bob. The encrypted session S_{AB} is established as:

$$S_{AB} = \text{Noise}_{XX}(K_A, K_B)$$

  This ensures confidentiality between peers.

- **Topic Generation & Verification**: Each chat room is a unique topic. The topic identifier is derived from the room name and an optional password. Let R_{name} be the room name and $P_{optional}$ be the optional password. The topic seed string S_{topic} is:

$$S_{topic} = R_{name} \oplus P_{optional}$$

  where $\oplus$ denotes string concatenation. The 32-byte topic buffer T_{buffer} used in the DHT is:

$$T_{buffer} = \text{SHA256}(S_{topic})$$

  This implicitly verifies access: only users knowing the correct R_{name} (and $P_{optional}$ if set) can compute the same T_{buffer} to join the room's swarm. Private room security depends on the strength of this combination.

- **Peer Lifecycle in a Topic**:
  1. *Announcement*: Node announces T_{buffer} to DHT.
  2. *Discovery*: Node queries DHT for peers on T_{buffer}.
  3. *Connection*: Hyperswarm establishes secure stream S_{AB}.
  4. *Data Exchange*: Application messages (chat, presence) are sent over S_{AB}.
  5. *Disconnection*: Peers leave or connections drop.

- **Presence Management**: Nodes broadcast 'presence_update' (with 'nodeId', 'nickname') upon joining/leaving a topic. Each node maintains a local list of peers per topic.

- **Message Propagation**: Messages are broadcast directly from the sender to all connected peers in the room's specific swarm (identified by T_{buffer}). No central relay is used for chat messages.

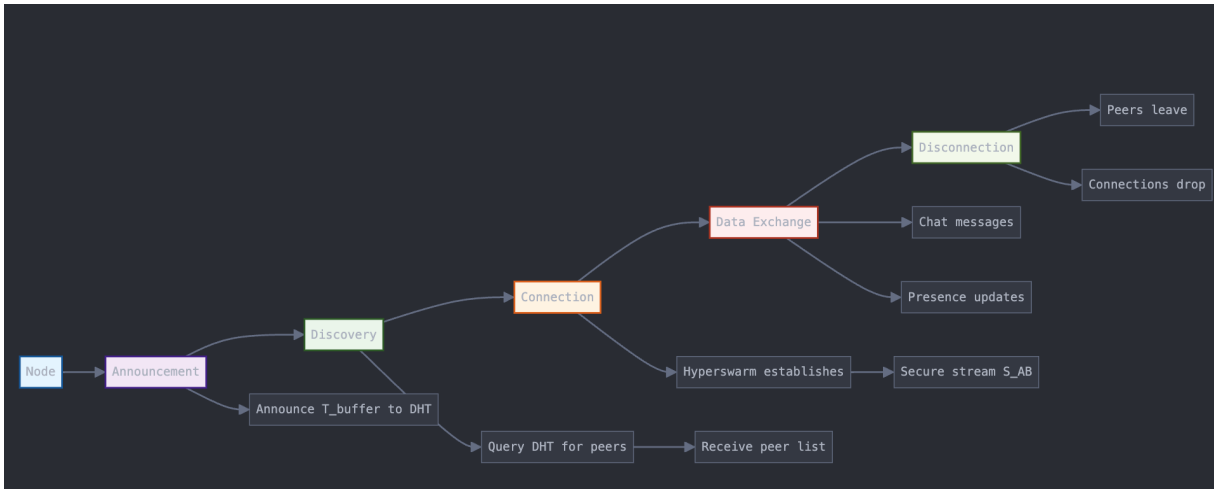


Figure 5: Node DHT Communication Flow

4.5 Architectural Diagram

The following diagram illustrates the high-level architecture and interactions between the primary components of the system.

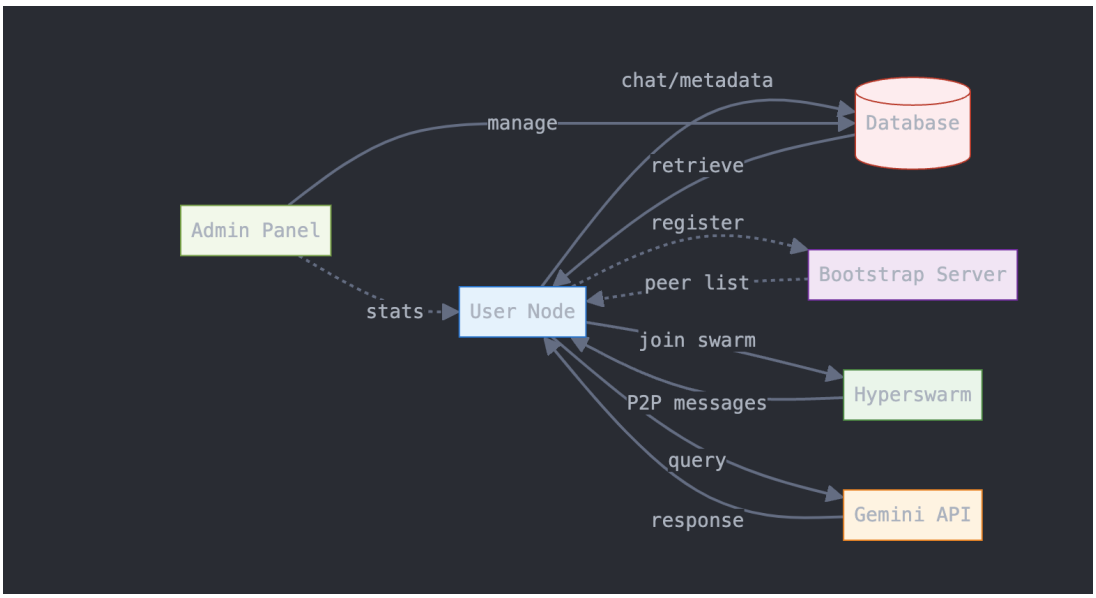


Figure 6: System Architecture: Key Interactions Between Components

5 Database Design

5.1 chat_message

```
CREATE TABLE chat_messages (
    message_id SERIAL PRIMARY KEY,
    sender_id VARCHAR(255) NOT NULL,
    sender_nickname VARCHAR(255),
    message_content TEXT NOT NULL,
    room_name VARCHAR(255) NOT NULL,
    timestamp TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    message_type VARCHAR(50) DEFAULT 'user_message'
);

CREATE INDEX idx_chat_messages_timestamp ON chat_messages (timestamp);
CREATE INDEX idx_chat_messages_sender_id ON chat_messages (sender_id);
CREATE INDEX idx_chat_messages_room_name ON chat_messages (room_name);
```

5.2 chat_rooms

```

CREATE TABLE chat_rooms (
    room_id SERIAL PRIMARY KEY,
    room_name VARCHAR(255) UNIQUE NOT NULL,
    creator_node_id VARCHAR(255),
    description TEXT,
    has_password BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_chat_rooms_room_name ON chat_rooms (room_name);
```

5.3 system_stats

```

CREATE TABLE system_stats (
    stat_name VARCHAR(255) PRIMARY KEY,
    stat_value INTEGER,
    updated_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);
```

6 Core Components

6.1 User Node

The User Node, a Node.js server application, is central to user interaction and P2P network participation.

Core Functions: It manages WebSocket connections for its Vue.js frontend client, facilitates P2P communication via Hyperswarm (for chat rooms/topics), interacts with PostgreSQL for message history and room metadata, provides a private AI assistant (Google Gemini API), and exposes HTTP API endpoints for system statistics (queryable by the Admin Panel).

Technology Stack (Backend): Node.js (v18.x), Express.js, 'ws' (WebSocket library), 'hyperswarm' (P2P networking), 'pg' (PostgreSQL client), '@google/genai' (Gemini API SDK).

Backend Logic ('user_node/server.js'): The server initializes Express for static file serving and APIs. A WebSocket server handles real-time client communication (joining rooms, messages, AI queries). It integrates P2P ('P2PNode') and database ('db/index.js') modules, manages in-memory state for online users, tracks concurrent user statistics, and provides the API endpoints detailed in Table 2.

Table 2: User Node API Endpoints

Method	Endpoint	Description
GET	/api/health	Basic health check, including P2P status.
GET	/api/rooms/list	Fetches and returns a list of public chat rooms from the database.
GET	/api/stats/online-users	Returns current total online users, per-room breakdown, and historical maximum.

User Interface (UI - 'user_node/public/'): A Vue.js (CDN) single-page application styled with TailwindCSS (CDN). Key features include room joining (public/private), real-time chat display (with Markdown via Marked.js), AI command input, and an online user list.

6.1.1 P2P Communication Module ('user_node/p2p/index.js')

This module, encapsulated as a 'P2PNode' class (extending Node.js 'EventEmitter'), handles all P2P network interactions using 'hyperswarm'. **Key Functions**: Manages the Hyperswarm instance and node identity (cryptographic key pair). It handles joining/leaving topics (chat rooms derived from room name + optional password), peer discovery, establishing encrypted P2P connections, and broadcasting/receiving application-level messages (chat, presence updates). Incoming P2P data is parsed, and relevant internal events ('message', 'peer_joined_room', 'peer_left_room') are emitted for the main server logic.

6.1.2 Database Interaction Module ('user_node/db/index.js')

This module centralizes PostgreSQL database operations for the User Node. **Key Functions**: Initializes and manages a connection pool ('pg.Pool'). Provides functions to save chat messages ('chat_messages' table), retrieve recent messages, get or create room records ('chat_rooms' table, handling public/private rooms), list public rooms, and initialize/manage system-wide statistics like 'max_concurrent_users' in the 'system_stats' table.

6.2 AI Capability Integration (Google Gemini API)

Each User Node provides a private AI assistant to its connected user via the Google Gemini API. **Invocation**: Users send messages prefixed with '/ai ' (e.g., '/ai Explain P2P'). **Process**: The User Node's 'callGeminiService' function parses the query, initializes the '@google/genai' client (using 'GEMINI_API_KEY'), and sends the request to a Gemini model (e.g., 'gemini-2.0-flash'). The AI's response

is relayed privately back to the originating user's WebSocket as a 'personal_ai_response'. **Privacy**: AI interactions are private, not broadcast P2P, and not logged in the shared chat database.

6.3 Bootstrap Server

A lightweight, centralized Node.js/Express.js application designed to facilitate the initial discovery of peers in the P2P network. This is particularly helpful for nodes operating behind NATs that might struggle with purely DHT-based discovery initially.

- **Core Responsibilities**: Maintains a volatile, in-memory list of currently active User Nodes, including their network address (IP, port) and unique identifier. It does not participate in chat message relay or P2P topic management.
- **Deployment Example**: Hosted on a publicly accessible virtual machine (e.g., Azure, IP '20.121.46.47', Port '3000'), with the Node.js process managed by 'pm2'.
- **Key API Endpoints**:
  - 'POST /register': User Nodes send their connection information (ID, address, port, type) to this endpoint upon startup to be added to the active list.
  - 'POST /unregister': User Nodes send a request to this endpoint before shutting down to be removed from the list.
  - 'GET /nodes': Returns a JSON array of all currently registered User Node information, which new nodes can fetch to attempt direct connections.

```
jh9108@proj2-center:~$ pm2 list
```

id	name	mode	↺	status	cpu	memory
0	bootstrap-server	fork	0	online	0%	44.9mb

```
jh9108@proj2-center:~$
```

Figure 7: PM2

```
jh9108@proj2-center:~/project-2/bootstrap_server$ npm start

> bootstrap_server@1.0.0 start
> node server.js

Bootstrap server script loaded. Starting...
Bootstrap server listening on port 3000
Registered endpoints:
  POST /register { nodeId, ip, port, type }
  GET  /nodes
  POST /unregister { nodeId }
```

Figure 8: API Endpoints

6.4 Cloud Database (PostgreSQL)

Serves as the persistent storage backend for critical application data, ensuring durability and accessibility beyond the lifecycle of individual P2P sessions.

6.5 Admin Panel

A web application for administrative control, comprising a backend API and a frontend UI.

Table 3: Admin Backend API Functionalities

Category	**API Endpoint(s) & Description**
Authentication	‘POST /api/auth/login‘: Administrator login.
Message Management	‘GET /api/messages‘: Retrieve messages with filtering. ‘DELETE /api/messages/:messageId‘: Delete a specific message.
Room Management	‘GET /api/rooms‘: Retrieve list of rooms. ‘DELETE /api/rooms/:roomId‘: Delete a room and its associated messages.
Statistics	‘GET /api/stats/overview‘: Aggregate system statistics (real-time user data from User Nodes, historical data from DB).

Authentication: Admin backend handles authentication by comparing submitted credentials (username/password) against configured, hashed values. The frontend manages login state and protects routes.

7 Component Interaction

This section details the sequence of operations for key user scenarios.

7.1 User Joining a Room

Initialization: User Node starts, connects to DB, initializes P2P module.

```
jh9108@proj2-center:~/project-2/user_node$ PORT=4000 ANNOUNCED_IP=127.0.0
.1 node server.js
[P2P] P2PNode instance created. Application Node ID: 6a22500aa897f01c12cd
5e69901abc92b6bf0f62ff17c23966114eb6aa835c7c
[DB] Connected to PostgreSQL database!
[DB] Successfully acquired a client from the pool.
Database connected successfully.
[DB] system_stats table checked/created.
[P2P] Initializing P2P communication system for Node ID: 6a22500aa897f01c
12cd5e69901abc92b6bf0f62ff17c23966114eb6aa835c7c
P2P Node initialized with ID: 6a22500aa897f01c12cd5e69901abc92b6bf0f62ff1
7c23966114eb6aa835c7c
```

Figure 9: User Joining a Room

Client Request: User inputs room details (name, optional password, nickname) on frontend and initiates join.

```
[BootstrapClient] Registering node with payload: {
  nodeId: '6a22500aa897f01c12cd5e69901abc92b6bf0f62ff17c23966114eb6aa835c
7c',
  ip: '127.0.0.1',
  port: '4000',
  type: 'user'
}
[BootstrapClient] Node registered successfully: { message: 'Node register
ed successfully', totalNodes: 1 }
[Bootstrap] Successfully registered with http://localhost:3000
Client connected. Assigned ID (server's Node ID): 6a22500aa897f01c12cd5e6
```

Figure 10: Client Request

WebSocket Transmission: Frontend sends ‘join_room‘ message (with room name, password, nickname) to User Node backend. **User Node Backend Processing**:

```
[P2P] Broadcasting message to 0 peers (no active connections on current t
opic 6b86b273ff34...). {
  type: 'presence_update',
  sender_id: '6a22500aa897f01c12cd5e69901abc92b6bf0f62ff17c23966114eb6aa8
35c7c',
  nickname: 'abc#1',
  topicHex: '6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb787
5b4b'
}
[DB] Room '1' found. DB record has_password: false
[DB] Fetched 12 recent messages for room '1'.
Client abc#1 joined room: 1
```

Figure 11: User Node Backend Processing

7.2 Message Sending and Receiving in a Room

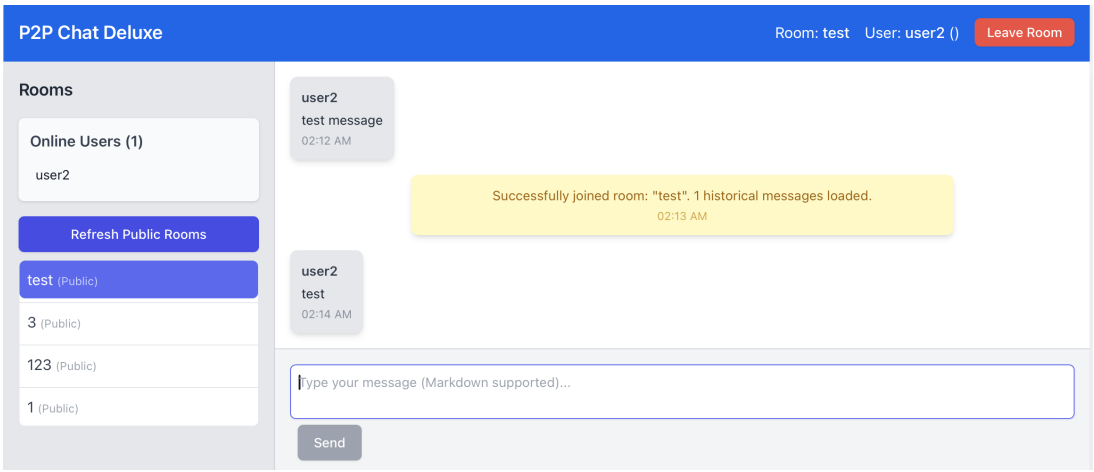


Figure 12: Message Sending and Receiving in a Room

7.3 Private AI Assistant Interaction

1. **User Command**: User types ‘/ai ¡query¿‘ in frontend.
2. **Client to Server (WebSocket)**: Frontend sends raw input as ‘client_chat_message‘ to User Node.
3. **User Node Backend - Parsing**: Backend identifies ‘/ai‘ prefix, extracts ‘query‘.
4. **AI Service Call**: ‘callGeminiService(query)‘ is invoked:
   - Initializes ‘GoogleGenAI‘ client (using ‘GEMINI_API_KEY‘).
   - Sends ‘query‘ to Gemini model (e.g., ‘gemini-2.0-flash‘).
5. **Response Relay**: Backend receives textual response from Gemini API.
6. **Server to Client (WebSocket - Private)**: Backend sends ‘personal_ai_response‘ (with original query and AI’s response) only to the initiating client.
7. **Frontend Display**: UI displays AI interaction privately to the user.

```
[DB] Room 'test' found. DB record has_password: false
[DB] Fetched 2 recent messages for room 'test'.
Client user1 joined room: test
Received message from client (user1): { type: 'client_chat_message', cont
ent: '/ai NYU' }
[AI Service] Sending query to Gemini: "NYU..."
[AI Service] Received response from Gemini.
```

Figure 13: Private AI Assistant Interaction

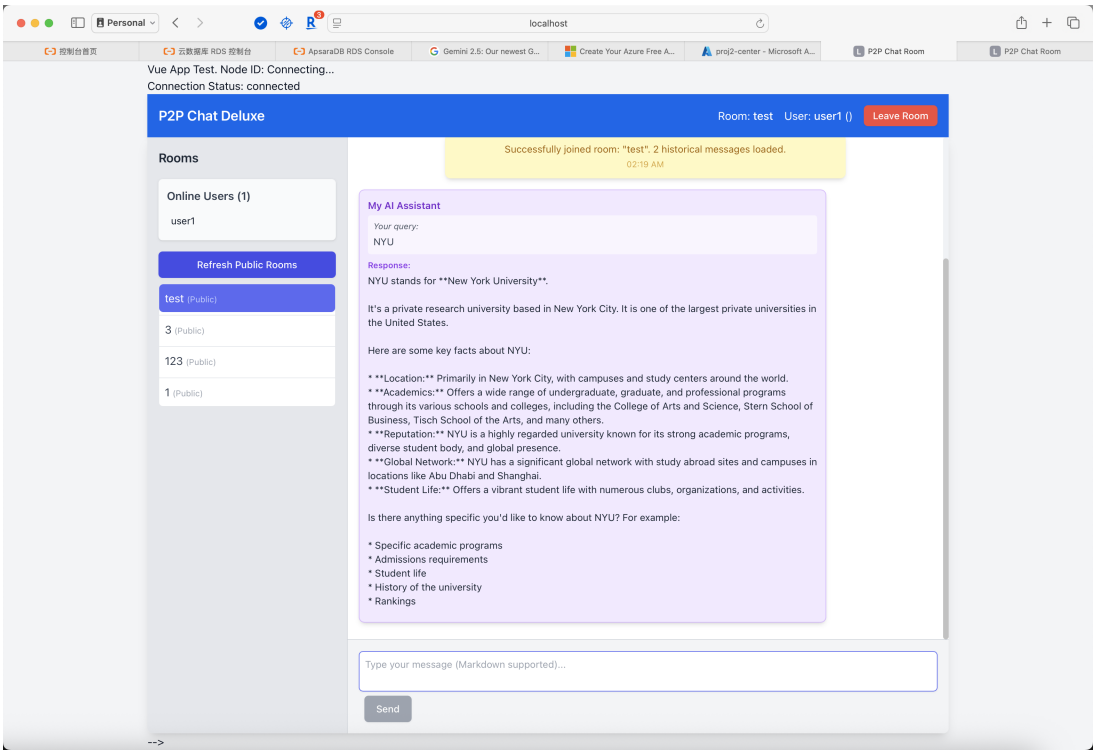


Figure 14: Private AI Assistant Interaction

7.4 Admin Panel Statistics Retrieval

1. **Admin Login**: Administrator logs into Admin Panel frontend.
2. **Dashboard Load & API Call**: Dashboard component mounts and calls ‘getStatsOverview()‘ from its API service module to fetch system statistics.
3. **Frontend to Admin Backend (HTTP)**: API service sends GET request to Admin Backend’s ‘/api/stats/overview‘.
4. **Admin Panel Backend Processing (‘/api/stats/overview‘)**:
   (a) *Fetch Real-time Stats*: GET request to User Node’s ‘/api/stats/online-users‘ (via ‘USER_NODE_STATS_URL‘) for current online user counts.
   (b) *Fetch Historical Stats*: Queries PostgreSQL for ‘max_concurrent_users‘ from ‘system_stats‘ table.
   (c) *Data Aggregation*: Combines real-time and historical stats into a JSON response.
   (d) *Response to Admin Frontend*: Sends aggregated JSON back to Admin Panel frontend.
5. **Admin Frontend - Data Rendering**: Dashboard component receives data and updates UI to display total online users, per-room breakdown, and historical maximum.

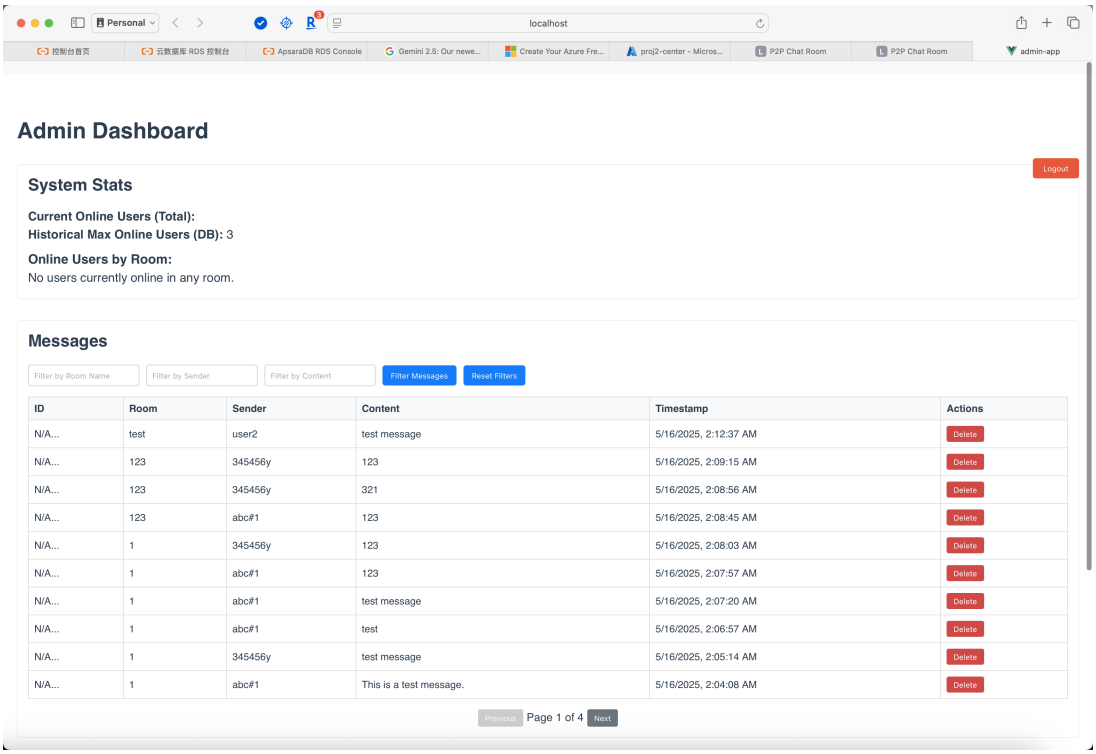


Figure 15: Admin Panel Statistics Retrieval

```
jh9108@proj2-center:~/project-2/admin_panel/backend$ node admin_api.js
Debug: apiRoutes.js fully parsed. Typeof router at export: function
Admin Panel API server listening on port 7777
Admin Panel: Successfully connected to PostgreSQL database.
```

Figure 16: Admin Panel Statistics Retrieval

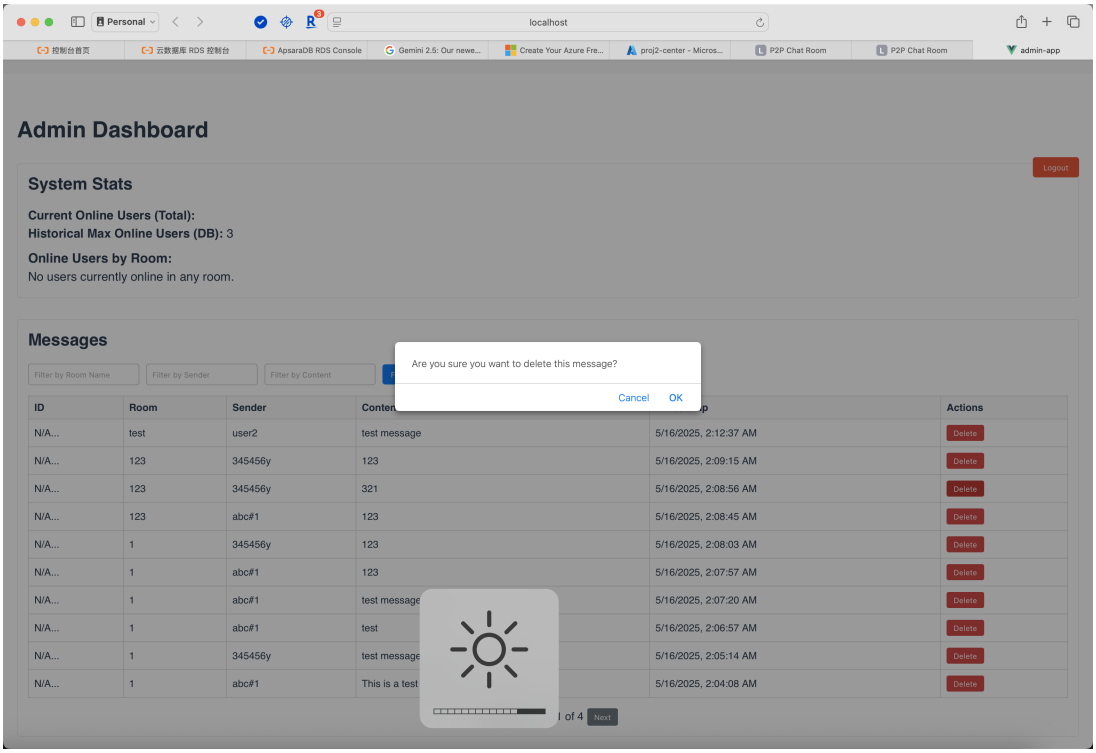


Figure 17: Admin Panel Statistics Retrieval

8 Detailed Setup and Configuration Guide

This section provides a more granular guide for setting up and configuring the project components. A prerequisite is a working knowledge of Node.js, npm, Git, and basic PostgreSQL administration.

8.1 Global Prerequisites

- **Node.js**: Version 18.x or higher (Project tested with v18.20.8). Install from https://nodejs.org/ or use a version manager like ‘nvm‘. Verify with ‘node -v‘ and ‘npm -v‘.
- **Git**: For cloning the project repository. Verify with ‘git –version‘.
- **pm2**: Node.js process manager. Install globally: ‘sudo npm install pm2 -g‘. Verify with ‘pm2 –version‘.
- **PostgreSQL Server**: A running instance of PostgreSQL (v12+ recommended). Note down connection details: host, port, database name, user, and password. Ensure the user has DDL (CREATE TABLE) and DML (INSERT, SELECT, UPDATE, DELETE) privileges on the target database.
- **Google Gemini API Key**: Obtain from Google AI Studio.
- **Project Code**: Clone the project repository: ‘git clone ¡your_repository_url¿ && cd ¡project_root_folder¿‘.

8.2 Component-Specific Setup

8.2.1 Bootstrap Server (‘bootstrap_server/‘)

1. **Navigate**: ‘cd bootstrap_server‘
2. **Install Dependencies**: ‘npm install‘. Key dependencies from ‘package.json‘:
   - ‘express‘: For the web server framework. (e.g., ‘4.17.1‘)
3. **Configuration**:
   - Port: Defined in ‘server.js‘, defaults to ‘3000‘. Modify if necessary. No ‘.env‘ file is typically used for this simple component, but could be added for port configurability.
4. **Start with pm2**: ‘pm2 start server.js –name bootstrap-server‘
5. **Verification**: ‘pm2 list‘ to check status. ‘pm2 logs bootstrap-server‘ for logs. Access ‘http://¡bootstrap_server_ip¿:30 in a browser; expect ‘[]‘ initially.

8.2.2 User Node (‘project-2/user_node/‘)

1. **Navigate**: ‘cd project-2/user_node‘ (adjust path if already inside the project root)
2. **Install Dependencies**: ‘npm install‘. Key dependencies from ‘package.json‘:
3. **Start with pm2**: ‘pm2 start server.js –name user-node‘
4. **Verification**: ‘pm2 list‘. Check logs: ‘pm2 logs user-node‘. Critical logs include successful DB connection, P2P initialization, and registration with Bootstrap Server. Access ‘http://¡user_node_ip¿:4000‘ in a browser to see the chat UI.

8.2.3 Admin Panel - Backend (‘project-2/admi_panel/backend/‘)

1. **Navigate**: ‘cd project-2/admin_panel/backend‘
2. **Install Dependencies**: ‘npm install‘. Key dependencies:
3. **Environment Configuration (‘.env‘ file)**: Create ‘project-2/admin_panel/backend/.env‘ with the following, replacing placeholders:
4. **Start with pm2**: ‘pm2 start admin_api.js –name admin-backend‘
5. **Verification**: ‘pm2 list‘. Check logs: ‘pm2 logs admin-backend‘. Test login API (e.g., with Postman/curl): ‘POST http://localhost:7777/api/auth/login‘ with JSON body ‘”username”: ”admin”, ”password”: ”adminpassword”‘. Expect a success response.

8.2.4 Admin Panel - Frontend (‘project-2/admin_panel/frontend/admin-app/‘)

This is a Vue CLI project.

1. **Navigate**: ‘cd project-2/admin_panel/frontend/admin-app‘
2. **Install Dependencies**: ‘npm install‘. Key dependencies (from ‘package.json‘):

API Base URL Configuration: The frontend needs to know the Admin Backend API URL.

- **Recommended method**: Create a ‘.env.local‘ file in the ‘admin-app‘ root directory:
- Vue CLI automatically picks up ‘VUE_APP_‘ prefixed variables from ‘.env.[mode].local‘ files.

Build for Production: ‘npm run build‘. This compiles and minifies assets into the ‘dist/‘ directory. **Deployment of Frontend Static Files**: The contents of the ‘dist/‘ folder must be served by a web server.

- **Option A (Integrated with User Node - Recommended for simplicity)**:
  1. Copy the entire ‘dist/‘ directory into ‘project-2/user_node/public/‘ and rename it to ‘admin‘ (so path becomes ‘project-2/user_node/public/admin/‘).
  2. In ‘project-2/user_node/server.js‘, add a route to serve these static files. After ‘app.use(express.static(path.join(’public’)));‘, add:
  3. Restart the User Node: ‘pm2 restart user-node‘. Access via ‘http://¡user_node_ip¿:4000/admin/‘.
- **Option B (Standalone Web Server)**: Configure Nginx, Apache, or another web server to serve the contents of the ‘dist/‘ directory. Ensure proper configuration for SPA routing (history mode).

8.3 Recommended Startup Order and System Verification

1. **PostgreSQL Database**: Ensure the service is running and accessible.
2. **Bootstrap Server (‘bootstrap-server‘)**: Start first. Verify by checking its logs and accessing its ‘/nodes‘ endpoint.
3. **User Node (‘user-node‘)**: Start after Bootstrap. Verify logs for DB connection, P2P init, and Bootstrap registration. Access its web UI.
4. **Admin Panel Backend (‘admin-backend‘)**: Start after User Node and DB. Verify logs for DB connection. Test an API endpoint (e.g., login).
5. **Access Frontends**:

- User Chat: ‘http://¡user_node_ip_or_domain¿:¡user_node_port¿‘ (e.g., ‘http://localhost:4000‘)
- Admin Panel: Depends on deployment (e.g., ‘http://¡user_node_ip_or_domain¿:¡user_node_port¿/admin/‘ or its standalone URL).

6. **Overall System Test**: Create users, join rooms, send messages. Check Admin Panel for updated stats and data.

9 Challenges and Future Work

9.1 Encountered Challenges and Solutions

- **P2P NAT Traversal**: Reliable peer discovery and connection establishment across diverse network topologies (especially symmetric NATs) remains a significant hurdle for P2P systems. Hyperswarm provides good out-of-the-box capabilities but is not infallible. The current bootstrap server assists primarily by providing an initial list of known peers rather than full-fledged STUN/TURN relaying. *Solution Approach*: Partial mitigation through a centralized bootstrap server. Full solution would involve STUN/TURN integration.
- **State Management in Distributed Environment**: Synchronizing online user lists and ensuring consistent views across peers connected via P2P and WebSocket clients to different User Node instances (if scaled horizontally) is complex. The current model relies on P2P presence broadcasts and User Node-level aggregation for its connected WebSocket clients.
- **Scalability of Centralized Components**: The Bootstrap Server, with its in-memory store, and a single User Node instance acting as a stats provider for the Admin Panel, can become bottlenecks under high load. *Consideration*: The Bootstrap server's load is relatively light (registration/listing). User Node stats API could be made more resilient.
- **Security Considerations**:
  - *P2P Message Security*: Messages over Hyperswarm connections are typically encrypted at the transport layer provided by the library, but application-level end-to-end encryption (E2EE) for chat content was not implemented, making messages readable by intermediate P2P nodes if they could intercept the stream (though direct peer connections make this harder).
  - *Admin Panel Authentication*: Current admin authentication is basic (hashed password). While ‘jsonwebtoken‘ is a dependency, a full JWT-based session management with refresh tokens and secure API protection for all admin routes was simplified for this iteration.
  - *Input Validation and Sanitization*: Essential to prevent XSS, SQL injection, and other common web vulnerabilities. Implemented to a degree, but requires continuous attention.
- **Data Consistency and Ordering**: In a P2P network, messages may arrive out of order or be dropped. While TCP-like connections from Hyperswarm help, application-level handling for strict ordering or message delivery confirmation in a multi-peer scenario was not a primary focus, relying on timestamps for ordering at the presentation layer and best-effort P2P delivery.

10 Conclusion

This project has successfully culminated in the development and demonstration of a hybrid Peer-to-Peer text chat system that integrates a robust set of modern communication features. By judiciously combining the decentralized communication strengths of Hyperswarm with the practical necessities of centralized support services—such as a Bootstrap Server for peer discovery and a cloud-hosted PostgreSQL database for message persistence and administrative oversight—the system offers a functional

and extensible platform. The incorporation of a private AI assistant via the Google Gemini API and a comprehensive administrative panel further distinguishes the application.