ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΗΣ ΑΝΤΑΛΛΑΓΗΣ ΜΗΝΥΜΑΤΩΝ ΜΕ ΧΡΗΣΗ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ

της

ΑΝΝΑ ΖΩΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

υποβληθείσα σε μερική εκπλήρωση των απαιτήσεων για το πτυχίο

ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ



DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2025

Εγκεκριμένη από:

Δρ. Παπαδάκης Νικόλαος

DEVELOPMENT OF A MESSAGING APPLICATION USING ARTIFICIAL INTELLIGENCE

by

ANNA ZOI

A THESIS

submitted in partial fulfillment of the requirements for the degree

IN ELECTRICAL AND COMPUTER ENGINEERING



DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2025

Approved by:

Dr. Papadakis Nikolaos

Περίληψη

Η πρόοδος των συστημάτων επικοινωνίας έχει επηρεάσει σημαντικά το τρέχον ψηφιακό οικοσύστημα. Από τις πρώτες τεχνολογίες ανταλλαγής μηνυμάτων έως τα σύγχρονα συστήματα επικοινωνίας σε πραγματικό χρόνο, οι προσπάθειες έρευνας και ανάπτυξης στοχεύουν στη βελτίωση της ταχύτητας, της ασφάλειας και της "νοημοσύνης" της αλληλεπίδρασης των χρηστών. Ταυτόχρονα, η εκθετική ανάπτυξη της Τεχνητής Νοημοσύνης (AI), και πιο συγκεκριμένα της Επεξεργασίας Φυσικής Γλώσσας (NLP), έχει δημιουργήσει ευκαιρίες για την κατανόηση και ερμηνεία της ανθρώπινης ομιλίας εντός των σύγχρονων δικτύων επικοινωνίας. Αυτή είναι μια εργασία για τη δημιουργία μιας ολοκληρωμένης, πολυλειτουργικής εφαρμογής μηνυμάτων με δυνατότητες τεχνητής νοημοσύνης για την ανάλυση του περιεχομένου των συνομιλιών και τη βελτίωση της εμπειρίας του χρήστη. Η εφαρμογή υποστηρίζει τόσο ατομικές όσο και ομαδικές συνομιλίες, βιντεοκλήσεις και μεταφορά αρχείων και εικόνων μέσω μηνυμάτων. Περιλαμβάνει επίσης χαρακτηριστικά τεχνητής νοημοσύνης που μπορούν να συνοψίζουν ομαδικές συνομιλίες, καθώς και να αναγνωρίζουν τον συναισθηματικό ή επικοινωνιακό τόνο (π.χ., φιλικός, επιθετικός, ουδέτερος) της αλληλεπίδρασης. Το έργο σχετίζεται με τον αρχιτεκτονικό σχεδιασμό και την ανάπτυξη της εφαρμογής, την επικοινωνία μεταξύ του frontend και του backend, καθώς και τη χρήση μοντέλων μηχανικής μάθησης για την επεξεργασία και ανάλυση φυσικής γλώσσας. Στόχος είναι η δημιουργία ενός έξυπνου περιβάλλοντος επικοινωνίας που ενσωματώνει τόσο τις παραδοσιακές αρχές της ανταλλαγής μηνυμάτων όσο και τις δυνατότητες αυτόματης ανάλυσης, σύνοψης και κατηγοριοποίησης που παρέχονται από την τεχνητή νοημοσύνη.

Abstract

The evolution of communication systems has significantly influenced the current digital ecosystem. From early message exchange technologies to modern real-time communication systems, research and development efforts have aimed at improving the speed, security, and "intelligence" of user interaction. At the same time, the exponential growth of Artificial Intelligence (AI), and more specifically Natural Language Processing (NLP), has created opportunities for understanding and interpreting human speech within modern communication networks. This is a project for creating a comprehensive, multifunctional messaging application with AI capabilities for analyzing conversation content and enhancing the user experience. The application supports both individual and group chats, video calls, and file and image transfer via messages. It also includes AI features that can summarize group conversations, as well as recognize the emotional or communicative tone (e.g., friendly, aggressive, neutral) of interactions. The project relates to the architectural design and development of the application, the communication between the frontend and backend, as well as the use of machine learning models for processing and analyzing natural language. The goal is to create an intelligent communication environment that integrates both traditional messaging principles and the capabilities for automatic analysis, summarization, and categorization provided by artificial intelligence.

Table of Contents

Περίληψη **3**
Abstract **4**
Table of Contents **1**
Chapter 1 - Introduction **3**
1.1 3
1.2 Objectives of the Thesis 3
1.3 Methodology and Approach 4
Chapter 2 - Background and Related Work **4**
2.1 Evolution of Communication Systems 4
2.2 Real-Time Communication Architectures 5
2.3 Real-Time WebRTC Systems 5
2.4 Artificial Intelligence in Communication Applications 6
2.5 Sentiment Analysis 6
2.6 Chat Summarization 7
2.7 Summary of Related Works 8
Chapter 3 - System Design and Analysis **8**
3.1 Problem Description 8
3.2 Functional and Non-Functional Requirements 9
3.3 Usage Scenarios 9
3.4 System Architecture Overview 10
3.5 Data Flow and Communication Models 10
3.6 AI Module Overview 11
Chapter 4 - Technologies and Tools **12**
4.1 Frontend Framework: React 12
4.2 Backend Framework: Express.js 12
4.3 Real-Time Communication: Socket.io 13
4.4 Mobile Device Integration: Ionic and Capacitor 14
4.5 Database Management: MongoDB 15
4.6 Artificial Intelligence 16
4.7 Development Environment and Hosting Platforms 17
Chapter 5 - System Implementation **18**
5.1 Application Architecture 18
5.2 Frontend Implementation 20
5.2.1 State Management and React Query 20
5.2.2 WebSocket Architecture (Socket.IO) 22
5.2.3 Chat Interface 24
5.2.4 Authentication and Form Validation 27
5.2.5 Centralized Authentication Management with Zustand 29

5.2.6 UI Components and Ionic Integration 31
5.2.7 Real-Time Voice and Video Calls with WebRTC and Socket.IO 33
5.3 Backend Implementation 36
5.3.1 Authentication and Access Control 36
5.3.2 User Management 38
5.3.2 Conversation Creation and Chat Management 39
5.3.4 Message Management and Storage 42
5.3.2 Real-Time Communication with Socket.io 43
5.3.6 Real-Time Audio and Video Communication with WebRTC 44
5.4 Artificial Intelligence Integration 47
5.4.1 Integration of AI Tools on the Frontend 47
5.4.2 Artificial Intelligence Integration in the Backend 50
Chapter 6 - Results and Evaluation 53
6.1 Application Interface and User Experience 53
6.2 Functional Evaluation of the System 57
6.3 Usability and User Experience Evaluation 58
6.4 Evaluation of Artificial Intelligence Functions 58
Chapter 7 - Conclusions and Future ProspectsΑνάπτυξης 59
7.1 Conclusions 59
7.2 Future Extensions 59
References 59

Chapter 1 - Introduction

1.1

Now communication platforms have become very important. They can be both used for personal conversation as well as for business communication since they can give people the ability to communicate directly with one another, in real-time, while having access to information quickly; therefore, people can get quick replies. The evolution of communication systems has come a long way; communication systems today include a variety of very useful features that help facilitate faster, easier and more organized ways to communicate. Developers face a new challenge in their industry, which leads them to work harder to find new and better ways to use existing systems with the main goal of adding more smart features to them. In today's world, real-time communication plays a major role when it comes to not only the stability of communication between devices and users but also the quality of the features provided to the user. As such, developers focus on meeting these requirements by designing new features to satisfy user demands for maximum functionality. Due to the progress of artificial intelligence, there are applications that can do much more than just read messages. They can summarize a long conversation and see how someone interacts during it. In the context of this work, this system was implemented, utilizing React for the client side and Socket.IO for communication between end users. In the same way, the system also adopts an external artificial intelligence api service for summarizing conversations and understanding the tone of each conversation. Its main goal is to establish and develop a messaging system that not only provides instant messages, but also has information about the flow of communication and its essence.

1.2 Objectives of the Thesis

The objective of this project is to create an artificial intelligence (AI) based real-time messaging system to enhance user communication through improved quality and efficiency. The existing messaging systems usually just provide a means for users to send and receive messages; therefore, there is little intelligence built into the system to allow users to effectively comprehend and process the conversations they participate in. This work addresses this limitation by implementing conversation summarization and tone analysis to provide users with useful tools for understanding the flow and emotional character of conversations. At the same time, its purpose is to deliver a functional, user-friendly, and extensible system

that can adapt to various devices by combining modern web technologies with artificial intelligence technologies.

1.3 Methodology and Approach

The methodology utilized for this project is rooted in software engineering and artificial intelligence principles/methods. Initially, a review was conducted of the literature related to other messaging applications that would be comparable in order to obtain as much architectural information about them as possible, as well as to analyze the advantages and disadvantages associated with each.messaging application. The results of this analysis were then utilized to develop an adaptable, web-based architecture/system for real time communication within the application software. The front-end (built using React) of the system was developed as an interactive environment while the back-end (built utilizing Sock.io) was created in order to collect user data, at any given time, providing both users with the ability to send and receive messages in real time. The use of Sock.io was determined to be the most appropriate choice as it provides for a persistent connection between the client and the server allowing for the ability for both users to send/receive messages in real time. The back-end was built using Node.js as an appropriate platform for managing multiple concurrent connections, while utilizing an event-based architecture that supports real-time communication. The system also accesses an external AI service that supports the summarization of conversation data, as well as sentiment analysis of messages sent between users. These functions allow the system to effectively generate concise summaries of past conversations and assess emotional states of the message senders.

Chapter 2 - Background and Related Work

2.1 Evolution of Communication Systems

One of the basic ingredients of human interplay is communication, which has changed a lot throughout history. The development of digital media and the Internet, as well as other technologies, have had a huge impact on how people communicate with each other. Digital form of communications are much faster than traditional methods of communicating. People can now use messaging applications, which are another form of digital media, to send and receive their messages. In addition to these

increasing speed and efficiency, the increasing volume of data that is transferred creates a large amount of request for "smarter" systems for sending or receiving messages, as well as for analysing, organising and understanding the content of those messages.

2.2 Real-Time Communication Architectures

A real-time messaging system allows users and/or devices to send and receive data in real-time with minimal or no delays. A real-time messaging system can transmit data between users and/or devices via the Internet using WebSockets or event-driven servers (event-driven servers maintain a persistent connection to send messages, rather than making multiple HTTP requests). Therefore, users can receive push notifications without excessive network resource usage costs. To accommodate the requirements of each application, there are different types of architectures available. The most common are client-server architectures, where a single server receives messages, routes them, and handles any session synchronisation; whereas peer-to-peer (P2P), where clients communicate directly to one another reduces latency, however it may be problematic in terms of security, managing user states and scale to meet demands (this is why modern systems of this type care about their availability, scalability and fault tolerance). These tools include ways to improve load balances, distributed databases (such as Redis and MongoDB), and message queues. There are also library packages like Socket.IO, which help you build and function these types of applications through their features like reconnection; support for multiplexing; broadcast capabilities, and namespaces. Additionally, most apps now utilize an event-driven microservices architecture, allowing the components supporting each user service (like users' accounts and notifications) to be independently created and used (including AI-based analysis of all messages). This method improves fault isolation and provides autonomy in scaling independent services, which can enable the system to develop future capabilities, such as voice calls, video, or smart message summarization.

2.3 Real-Time WebRTC Systems

WebRTC-based WebSocket and Socket.IO systems offer real-time messaging with excellent efficiency. As the dominant standard providing low-latency direct communication between devices or browsers, WebRTC's ability to support direct audio, video and/or data from the device means that intermediary software or systems are not required. With peer-to-peer capabilities and complete end-to-end

encryption, WebRTC also allows for a dynamic negotiation of communication channels in NAT and firewall-ruled environments through the use of Session Description Protocols (SDP) and Interactive Connectivity Establishment (ICE) protocols. Currently, the most popular use of WebRTC is providing video conferencing; however, WebRTC has many other uses such as real-time collaboration tools, e-learning systems, and AI chatbots. Supporting all forms of data transfer (e.g., collaborative file transfer, real-time gaming), as well as supporting AIs (for moderation or summarisation purposes), when combined with backend tools like Express and MongoDB, WebRTC can deliver a unified low-latency cloud communications solution. [1]

2.4 Artificial Intelligence in Communication Applications

Artificial Intelligence technology has rapidly evolved into an integral part of communication applications today by providing analysis capability of all content within same application (i.e., how a message is interpreted by the recipient). Therefore, not only do these applications allow transmission of messages, but they include several new functions, including: Tracking tone, summarizing messages, extracting key words from messages, suggesting responses, and moderating harmful language and/or content, thus creating more meaningful connections between users. The application's backend will send messages and chats from users to an external AI service provider (such as OpenAI, etc.) and then receive appropriately formatted responses based on the user's request. AI will not be used to model internally but will only be made available as a service through APIs.

2.5 Sentiment Analysis

Sentiment analysis is the analysis of a piece of text in order to determine the emotional content of that text. In terms of how sentiment analysis is performed in more complicated systems, a detection method typically involves the development of a one-of-a-kind model that is trained using a large amount of data using an approach like a neural network because this type of architecture is capable of preserving the meaning of the text and other critical features of the text such as context, qualifiers, negating words, etc. In this particular project, the system is designed to be less difficult to implement; therefore, the approach to doing the analysis is going to be accomplished by using an external application programming interface (API). The API would allow for messages that have recently occurred or for the total context of a conversation to be sent to the API to provide a more accurate assessment of the total sentiments

contained in a conversation. By implementing a flexible layer for the analysis of sentiment that is adaptable to conversations but does not add any additional pressure to the overall chat system, this approach allows performing sentiment analysis to be accomplished using a simpler layer than more elaborate sentiment analysis systems using deep learning may have been able to accomplish in the research environment (and that, as an additional benefit, is easier to maintain) if they are analysing text containing subtleties of the language such as idiomatic expressions, sarcasm, and complex semantic structures. The objective of this project is to present the real-time conversation of an individual with meaningful emotional insights regardless of how advanced the existing sentiment model may be.

2.6 Chat Summarization

As more and more employees have switched over to working remotely, there is an increased need for group interactions in the workplace, schools, and communities as well. Because the rate of conversation and frequency of messages in a group chat can be quick and plentiful, participants who are either not present or who join later will have difficulty identifying key team messages. The use of artificial intelligence to summarize a group discussion's content generates a summary of the topics discussed, conclusions that were reached, and any new information resulting from the discussion. These AI systems summarize by interpreting and analyzing conversation content linguistically, contextually, and semantically - and produce a summary that captures both the meaning and emotional value of the messages exchanged during the exchange. Since the AI generates this summary based on input received in real-time, there is no need for individuals to manually review chat history or search for specific keywords. As a result, organizations can improve their ability to effectively communicate team information by providing timely access to summaries and better managing their team members' physical distance and time constraints. This research project demonstrated summarization using external artificial intelligence services such as OpenAI (via the OpenAI API). By utilizing the Summarization API, the artificial intelligence identifies which topics were discussed, what conclusions were drawn, and any updates made during the course of the conversation. Once an employee presses a button within a chat application, an automated machine generates and returns a summary of the conversation in a chat message format to the employee who initiated the request. With API, the quality of the summary is quite good without overloading the system compared to large local models that require a full training process. It is also scalable, as third-party services process large-scale messages without burdening the backend, and consistent, as only one model generates summaries of the same quality for each conversation. In terms of summarization, its main benefit is translating the conversation into a narrative, rather than individual

messages sent one after another. Therefore, summaries become understandable and useful so that anyone who has fallen behind in the conversation flow can be informed about agreements, any pending issues, and a plan for the next steps. Additionally, this solution will help maintain productivity by spending less time reading past messages and likely not missing important information. By combining instant messaging with AI-based summarization, chat systems are transformed from simple message delivery mediums into more proactive assistants in teamwork, enabling users to focus, act precisely, and make knowledge-based decisions.

2.7 Summary of Related Works

On the contrary, for more interactive systems, a growing number of researchers and developers have integrated messaging architectures with AI-based innovations. Previous works report the implementation of end-to-end NLP models for sentiment analysis and content summarization, and their direct integration with chat infrastructures. Some studies compare architectural choices – for instance, whether the AI logic should be embedded directly in the chat server or placed in an external microservice. At the same time, AI-related tasks are outsourced, while the core chat system remains dedicated purely to delivering instant messages with minimal possible latency. In this way, the benefits of traditional technologies are utilized – React for the frontend, Express and Socket.IO on the backend, and MongoDB for data storage – together with intelligent power via API integration (internal models are not needed). This trade-off between performance, flexibility, and intelligence informs the design of our existing system, making it both a modern communication platform and a gateway for interacting with AI-driven tools.

Chapter 3 - System Design and Analysis

3.1 Problem Description

In recent years, many more people have turned to online communication. From personal use, telework, and online education applications to messaging apps, messaging applications have also been adopted as one of the most widespread real-time communication platforms. These apps and their interfaces have been created to make communication faster and easier than email, where the goal is now

more about how to receive or interpret information quickly instead of how to send information. User fatigue is a common experience when you take into account long conversations and active group conversations, especially when you're continually trying to follow up on current conversation threads. In addition, the ability to identify your emotional state is limited, and this leads to increased likelihood of misunderstandings – particularly as people tend to struggle with reading and determining the appropriate emotional tone or meaning of the responses they receive. As a result, our effort is to find a way to overcome this barrier and build a real time system that facilitates fast and instantaneous forms of communication through the use of Artificial Intelligence and also enhances the communication aspect by making the experience of chatting more interesting and purposeful. As an important component of this system, we have implemented 2 additional forms of smart technology: Summary – to summarize the content of a conversation and Emotional Tone Reading – used to categorize a message by its sentiment based on the associated emotional tone. Together, these two technologies provide users with efficient, clear communication, while also leveraging Emotional Intelligence, keeping users informed and engaged in fast moving group communication.

3.2 Functional and Non-Functional Requirements

Functional requirements specify the system's expected behavior. Users must register and log in before they can securely access the chat page. They must send or receive real-time messaging capabilities via both group/private chat modes. Messages must have timestamps and users must receive notification types for incoming messages. Users can use the AI functionality to request summaries of previous conversations while also learning about the tone of those conversations. All messages are stored in a database which allows users to look back at any of their prior conversations based on their own individual references. Security is heightened through the use of encryption for all user credentials, as well as secure mechanisms for communication using Transfer Data Protocol (TDP).

3.3 Usage Scenarios

When users successfully log into the disappearing message feature, they can access a chat window to begin chatting with someone (or multiple persons) and see the same message at the same moment when the corresponding coding function has executed for both users. For chats with multiple participants, all participants will see their messages simultaneously, regardless of individual message

timings. If a user wishes to create an overview of one message in a group discussion, the user simply clicks “Get Summary,” and will receive an overview of the most important communication aspects and an overview of the topics discussed in that particular message’s discussion thread. This functionality will assist users who wish to review a long conversation; either they have logged out of the system prior to the new messages coming into existence or because they have linked to that conversation after it has been created by joining an ongoing chat. Another use case of chat is the emotes that are sent via a button; users can use the button to qualify the tone of the individual conversation where they would receive feedback about how that particular conversation has made them feel (the emotions that are a part of that chat). This type of information could be registered as a part of the chat system and could assist other individuals in how to experience the emotional tone of that particular chat.

3.4 System Architecture Overview

The software was developed with a client-server architecture. The client (frontend) is responsible for the user interface, while the server (backend) is responsible for the functionality of processing data and messages through the client and backend applications. React has been employed in the development of the frontend; hence, the user interface can enable real-time interaction with the user. Functions in the client and server have been implemented so that each one is independent from others. As new messages arrive, the user interface is updated with no need to reload the page – thus allowing for continuous live access to the internet. The Express/Node back end on the server handles all requests, routing and connecting to the database and AI services. Node is an event-driven environment, thus providing high-scale concurrent users. Express helps create RESTful APIs for such activities as user authentication, retrieving messages from the database, and sending requests to the AI system. Furthermore, the server uses Socket.IO to establish bidirectional communication capable of real-time user (client) to server direction, thus creating a persistent connection between the client and the server that sends/receives messages in real time. The MongoDB database contains data such as user messages, chat histories, and metadata in this open-source, document-based database that is flexible with data. The schema-less definition of MongoDB means the system can be flexible both when adding new features and when scaling the system.

3.5 Data Flow and Communication Models

The flow of information starts when a user logs into the system. First an API call is made to MongoDB to verify the user's account credentials (the user account is stored securely). After the account is verified either via the Socket.IO connection or API call, the connection is established. When a user sends a message to the server via the Socket.IO connection, that message data is sent as a socket event (through the established socket connection) to the server, written to MongoDB and forwarded to everyone in the current conversation. Each group chat is equal to a separate namespace on the server so when the messages are sent to recipients, they only get sent to members of that specific group. This same flow of data can be used when performing actions against the AI model. The server is responsible for retrieving messages relevant to the processing, sending those messages through the AI API for processing, and returning that information back to the client via the established connection which could either be a summary or tone detection (for example). Once completed, the results of the previous queries are visually represented in the user's interface and are included as part of the ongoing conversation stream of events. The process is effectively a continuous data flow cycle that is both complete and 'smart'.

3.6 AI Module Overview

The AI module that has the most focus on user-friendliness and becoming more aware while in conversation is the sentiment analysis module. The purpose of this module is to analyse the message text and determine if the conversation has more positive, neutral or negative sentiment. This function improves understanding of the sender's attitude toward different sections of the message where tone and expression are often hard to determine like when the conversation is being conducted professionally or in a group of people. The summarisation tool will then determine the number of sections that the long conversation is broken into and will point out the key points of the conversation to make it easier and shorter to present. The utility is very helpful when a user who has been away from a particular conversation wants to re-enter it. Rather than having to scroll through a lot of messages to see what has been discussed, they can take a look at the important points that have been highlighted in the summary. Both of these uses of artificial intelligence are made available through an API request from the backend to the external AI service.

Chapter 4 - Technologies and Tools

4.1 Frontend Framework: React

React is the library used to create the front-end. As there will be a high volume of messages being received and displayed in real time (within seconds). Therefore, it is imperative the interface can refresh the content of the display without having to reload the page. The library creates the functionality needed for users to see updated screen content almost instantly (at least as fast as possible) due to the creation and use of components. A component is a re-useable unit of functionality; thus, each component will maintain its own data and behaviour (the buttons used for sending messages, the input box for entering new message text, and the list of contacts). Adding new capabilities to the front-end can be accomplished quickly by creating and integrating new components into the existing system in a modular manner. A key feature of React is its virtual DOM. Rather than updating the browser's DOM with every change that gets made, React stores in memory a representation of the interface. When the state changes, except for the parts of the interface that actually have changed, React compares the updated model to the previous one and updates only the parts that need to be updated. This approach has significantly increased the performance and responsiveness of applications, particularly for use cases with many real-time updates. With this in mind, it can be said that React allows for fast switching between conversations by using declarative programming where UI state is defined by state; in turn, code is much simpler. When a message arrives via Socket.IO, the interface will automatically refresh itself, and all of the various add-ons available to React (Redux, React Router) add more functionality to usage including theme switching, grouping messages together and support for both web and mobile application creation. The architecture of React makes testing and debugging easier due to the availability of tools such as React Developer Tools that allow the user visualization and performance monitoring of their applications. [3]

4.2 Backend Framework: Express.js

For the backend, Express.js was chosen utilizing the flexibility of the Node.js framework to organize routes and middleware which is essential for integrating authentication message exchange and AI processing. The asynchronous nature of Express allows managing concurrent connections critical for high network throughput in messaging systems while support for promises and asyncawait ensures the smooth operation of the event loop. Middleware chaining facilitates adding updating application

connection authentication and error management which enhances code composability especially in realtime applications where a chat history request goes through multiple stages token validation logging data sanitization etc before reaching the handler. Additionally backend frameworks in more complex cases offer introspection and automatically generated interfaces allowing for example connecting Express routes for dynamic creation or validation of API schemas ensuring code documentation consistency though full automatic schema extraction is not implemented here. The flexibility of Express supports such extension as well as integration with Socket.IO where the same Express process handles HTTP endpoints user registration AI requests and realtime socket events messages notifications. With this resource context and middleware sharing is achieved such as the authenticated user context from Express middleware being reused in socket handlers for security and state management. Furthermore combining Express with clustering and load balancing multiple instances behind a load balancer with a shared database and socket coordination allows horizontal scaling for environments with increasing user load. [4]

4.3 Real-Time Communication: Socket.io

Modern data-driven web applications leverage real-time messaging allowing immediate response to state changes and information distribution to users. In this system Socket.IO facilitates bidirectional client-server communication providing WebSocket abstraction and fallback mechanisms where required. Additionally the reliability and flexibility of Socket.IO make real-time message exchange feasible even in non-ideal environments. Socket.IO provides a data-driven framework for a messaging system requiring rapid and reliable exchange of messages. The use of persistent socket connections allows applications to utilize persistent sockets to directly transmit events to users rather than relying on the client making repeat requests for information to the server. This enables functions such as real-time message exchange user presence indicators typing notifications and delivery confirmations offering a user experience similar to modern chat applications. Particularly important are the namespace and room capabilities provided by Socket.IO which allow logical organization of users and messages. Each chat room operates as an independent communication space whether it is a private conversation or a group chat. Thus messages are transmitted only to users participating in that specific conversation limiting unnecessary data transfer and improving the overall system performance. Furthermore since security is a key factor in designing such applications Socket.IO integrates with user authentication mechanisms where the user identity is verified via token before establishing the connection. The authenticated context is maintained throughout the socket communication ensuring that the same access rights apply to both REST requests and socket events. Regarding scaling Socket.IO supports operation in distributed environments through adapters such

as Redis enabling collaboration among multiple server instances. This capability is essential in messaging systems with a large number of concurrent users. [6]

4.4 Mobile Device Integration: Ionic and Capacitor

Capacitor is used to bridge the gap between the web environment and the device's native functionalities allowing access to native APIs and features such as camera GPS and push notifications. The same code used for the web frontend can be packaged and distributed as a mobile application. This is particularly important for such an application where interface consistency and responsiveness are critical. Ionic provides a set of UI components such as Buttons Tabs etc. which automatically adapt to the native look of each platform. The chat environment contact lists and user profile screens appear native on iOS and Android while Ionic facilitates the reuse of existing business logic and React components simplifying the development and maintenance of multiple versions. Responsive layout adaptable to various screen dimensions and orientations serves multi-device messaging environments while routing and state management maintain chat sessions and history. Ionic provides the UI with Capacitor bridging web applications and device native capabilities through a unified API critical for messaging functions such as camera storage notifications and background processes. Features such as capturing and sharing images file management and notifications are implemented via Capacitor. Its plugin system with ready-to-use plugins for common functions also allows the development of custom plugins in Swift or Kotlin for specialized needs. Furthermore the extensibility and future sustainability of the system are achieved by integrating device-level functions or SDKs e.g. AI inference voice communication in later development stages. The use of Capacitor facilitates the interaction of the web frontend with the device hardware without complicating the core code. Architecturally the combination of Ionic and Capacitor integrates into the described structure with the backend Express.js providing REST and real-time endpoints Socket.IO accessible by the mobile client as well as the web simplifying authentication state synchronization and message delivery. At the same time the delivery of AI-generated content e.g. tone analysis or summaries is now feasible on mobile clients ensuring the availability of intelligent functions regardless of platform. While hybrid applications used to face performance limitations newer Ionic and Capacitor have mitigated these drawbacks supported by techniques such as lazy loading caching and optimized event handling ensuring smooth operation under pressure and in low connectivity. The development workflow with Ionic and Capacitor supports efficient testing deployment and maintenance. You can build one application targeting both iOS and Android using separate builds per platform, updating the web layer without requiring resubmission to the respective app stores. This approach speeds up development times and

allows for new functionality and improvements to AI to reach users faster than with traditional methods of building separate applications for different platforms. The architecture allows for the preservation of both platform native project structures, enabling optimizations in performance to occur without negatively impacting the hybrid architecture of the two applications. Ionic ensures an efficient mobile interface while Capacitor provides the native functions required by a modern messaging application. Together they enable the system to operate as a cross-platform communication tool that is fast, stable and ready for future expansions in AI interaction and advanced notification systems. In this way it is ensured that the mobile experience maintains the same levels of usability and responsiveness that have been defined in the system's architecture.

4.5 Database Management: MongoDB

Given the requirements of reliability performance and scalability at the database level MongoDB was chosen a document-oriented structure whose dynamic schema responds to the evolving nature of data models in messaging systems which handle diverse data (messages metadata attachments AI information). Storage in documents allows flexible extension with new features avoiding the complexities of schema migrations. MongoDB's architecture supports the high read/write speed requirements essential in concurrent messaging environments. It provides efficient indexing and horizontal scaling with each message (text or multimedia content) stored as a document with timestamps sender information delivery status and AI annotations while its query model facilitates the retrieval of recent messages with optimized filters and sorting. Furthermore MongoDB's aggregation pipeline enables more complex searches and data processing such as extracting statistics from conversations or preparing data for processing by AI summarization modules. The core advantage of MongoDB in this specific system is that it naturally complements the Node.js environment and the backend with Express.js presented in previous chapters. Since both the server and the database operate with JSON-like data structures the application does not require costly transformations between different formats. This leads to cleaner code reduced overhead and smoother data flow from the client to the server and then to storage. MongoDB's asynchronous operations allow Node.js's non-blocking model to function efficiently serving concurrent requests such as sending and retrieving messages loading chat history and updating user status without blocking the event loop. Scalability was also a key reason for choosing MongoDB. As the number of platform users increases along with the volume of conversations and data storage grows at a fast pace. MongoDB resolves this issue via native support for replication & sharding. Specifically, replication guarantees that there are multiple synchronized copies of your database at any point in time (providing fault tolerance and high

availability). Horizontal scaling via sharding distributes data across multiple servers supporting the growing demands of the platform especially for user communities with rich multimedia content. Choosing MongoDB requires a balance between performance and consistency; despite document flexibility lack of organization can lead to chaotic development. Consequently discrete organization of collections (messages users conversations AI) prevents dysfunctional documents while indexing strategies focus on conversation identifiers timestamps and user IDs balancing query speed and cost. Although multi-document transactions are supported most operations remain atomic at the document level simplifying management and aligning with the event-driven model of the chat system. Furthermore MongoDB proves effective for real-time behavior where messages are written and retrieved instantly via Socket.IO events. Sharding splits data into chunks which are distributed across multiple servers allowing horizontal scaling for increased storage and performance requirements. This model fully aligns with the platform's long-term goals as future support is anticipated for large user communities with high traffic and rich multimedia content. At the same time adopting MongoDB requires careful design decisions to maintain a balance between performance and consistency. Although MongoDB's documents are flexible this flexibility can lead to chaotic and unstructured development without proper organization. For this reason collection management is done by separation (e.g. messages users conversations AI data) to avoid dysfunctional documents while indexing strategy balances query speed and storage cost (conversation IDs timestamps user IDs). Despite MongoDB's support for multi-document transactions the application targets atomic operations for simplicity and efficiency matching the event-driven model of a chat system. Ultimately MongoDB proved effective for real-time behavior (instant message writing/retrieval via Socket.IO) serving as a persistent layer for loading history AI summaries and an active presence system. Therefore it provides a robust scalable and flexible foundation for the data layer with its document-based model harmonizing with messages and sessions meeting real-time requirements and integrated into the Node.js ecosystem for future expansions as well as the long-term evolution of the platform.

4.6 Artificial Intelligence

The integration of Artificial Intelligence into the system enables more meaningful analysis and processing of conversations beyond simple message exchange. The AI modules analyze, interpret, and summarize conversations, supporting the management of message flow and emotional tone. Each incoming message is evaluated for its emotional polarity (positive, neutral, or negative), providing information that can be used for further functions such as mood-based summaries, conversation quality monitoring, or notifications in cases of intense or negative interaction. The ability to analyze the

sentiment of messages in relation to each other, via the directed sentiment processing section, allows the results of the analysis to be returned to the user almost immediately after a message is sent. Also part of the system is a summary module that helps manage large volumes of chat messages. Group chat messages are often made up of many different messages, including images and multimedia, as well as side conversations between group members, making it difficult to keep track of a conversation if you didn't see it in real-time. The summary module provides a condensed version of the most important conversation details, including the main events of the conversation, any important decisions made during the course of the conversation, and the direction of the overall conversation, so that users can more quickly find the specific information they are looking for without scrolling through large numbers of chat messages. Just like the rest of the system's internal architecture, the communications between the AI and other components are designed to be modular and scalable, allowing the system to add new features without requiring extensive changes to the existing architecture. Currently, the system has two primary functions: the ability to process directed sentiment and summarize chat conversations; but the system design is such that other features will likely be developed in the future, like topic detection, intent classification, response suggestions, and/or content moderation, using the existing system architecture. MongoDB’s data model and the event-driven backend architecture allow new metadata to be added to messages, conversations, or user profiles without affecting overall operation.

4.7 Development Environment and Hosting Platforms

Developing and hosting new versions, as well as maintaining applications, usually requires a specific methodology. A cloud-based set of tools simplifies local development and hosting, ensuring stable performance, with Render serving as the central hosting provider. The software has been developed and tested using Node.js with Express for the back end and React and Ionic for the front end with real-time communication through Socket.IO. The system has been designed for easy communication among all components and for the proper functioning of components. Development cycles are fast-tracked by using package managers, version control and debugging tools, while the flexible design allows for back-end service instances to run concurrently. The AI processing modules and mobile build pipeline enable simultaneous evaluation of each layer of the system, which enables realistic use cases to be simulated in real-time, thereby providing the opportunity to identify issues early in the development cycle. Render is used to host the backend, real-time services, and AI modules. It is a modern cloud platform that automates a large part of the deployment process. Subsequently the CI/CD capability directly from the project repository (where every commit results in a build and deployment) is a

significant advantage in terms of maintaining consistency in terms of updates and ensuring that the latest version of the codebase is represented in production. Moreover, the support for containerized services and scalable compute instances allows for parallel scaling of backend services as the number of users increases, which is important in terms of maintaining real-time communication and AI capabilities in a messaging system that needs to remain responsive as the system grows in terms of usage, with the potential for activating instances for load optimization as the system grows. Render's monitoring tools and logs facilitate tracking, diagnosis, and optimization. Furthermore, database hosting integration is usually done through the connection of Render services and MongoDB, hosted either through Render or a dedicated MongoDB provider, thus promoting secure communication between the backend database and the network connection, which is fast and efficient, especially regarding access to the history of messages, user profiles, and metadata created by the AI. The system is further enhanced by the inclusion of backup, environment variables, and access, which improves the integrity and security of the data, regardless of the environment, and the mobile application. The mobile application, developed using Ionic/Capacitor and connected directly to the backend, means that the cloud environment is critical to the stability and access to the authentication and message synchronization endpoints, all through Render. The performance and data synchronization in the mobile application, after its release to app stores, depend directly on AI requests and media uploads, with the chosen environment supporting gradual evolution. As the application evolves (e.g., with more complex AI models), the infrastructure scales without fundamental changes. Render's flexibility allows for microservices, background workers for AI processing, and resource increases during peak periods, ensuring responsiveness to needs and long-term goals, while Node.js and modern tools support backend, frontend, and AI layer development, with Render providing stability, scalability, and an automated deployment process. Together, the architecture enables real-time operations, intelligent processes, and cross-platform distribution, thus creating a strong foundation for both current functionality and future development.

Chapter 5 - System Implementation

5.1 Application Architecture

The application comprises a front and a back end that are separate, but still work together to allow for an independent system of handling events. The core elements of the application are real-time

communication through the use of Socket.IO and integration of artificial intelligence (AI) to provide a seamless experience regardless of the environment in which it is running.

The front end is located in the client folder, and it is built using React and Ionic, along with Capacitor, which is used to distribute the front end to web and mobile applications. The front end will interact with the backend through RESTful APIs to perform regular operations and Socket.IO to handle the sending of messages in real-time. This way, the front end will be able to inform the user immediately when it receives the messages.

The backend, which is stored in the server folder, is developed using Node.js and Express.js. The backend handles user authentication, message, users, media file processing, and AI request processing. Like the frontend, the backend uses REST endpoints for conventional tasks and Socket.IO for two-way real-time communication. Every time a user sends a new message, receives a notification that an update has been made to a message that was previously sent, notifies that the user is typing, or when the result of an AI request has been received, all users will be notified instantly. The backend also holds sensitive environment variable values (e.g. for database connection) and makes it possible to separate code from configuration. As such, the JWT secret used to create and authenticate JWTs, cloud storage, and external services tokens can be stored separately from the application and therefore cannot be compromised should the source code be compromised. Furthermore, MongoDB is used as the primary database for storing/retrieving users, conversations, messages, attachments, as well as metadata related to the AI. This is due to the fact that the optimized database schema allows for fast search as well as indexing for the smooth retrieval of the conversation history under high-volume messaging scenarios. The backend manages interactions with the database, data integrity, and secure access patterns, while the AI module is fully integrated as a component of the backend architecture.

When a user requests a summary or tone analysis, the backend forwards the content to the AI. Results are returned as structured objects, stored in MongoDB along with the corresponding messages, and immediately transmitted to the clients. The module is coupled with the rest of the backend logic in a way that allows easy replacement or extension of AI models without requiring changes to the overall system architecture.

5.2 Frontend Implementation

The frontend implemented with React and Ionic as mentioned in a previous chapter forms a hybrid architecture for web and mobile experiences from a single codebase where the componentbased structure of React is combined with crossplatform UI elements. The modular design it features separating presentation logic data management and network communication enhances maintainability and reusability. A service layer handles API communication integrating HTTP requests through Axios authentication headers and interaction with protected resources thus isolating components from lowlevel network issues. For state management and asynchronous operations React Query simplifies retrieval caching and updating of remote data automating load states error handling and synchronization of cached values with the backend. The application displays cached content concurrently with background updates improving responsiveness while components that require server data declare it via queries and mutations ensuring an updated UI without manual state management. Additionally Socket.IO supports realtime communication: active sessions maintain a persistent WebSocket connection via a custom hook for immediate message reception and updates. This hybrid model REST endpoints and realtime streams enables reliable HTTP communication combined with the immediacy of chat. Finally the UI consists of modular components that integrate UI logic and domainspecific behavior meaning pages such as the inbox chat view login screen and registration form are based on Ionic components like IonPage. Furthermore reusable components such as IonButton provide responsive styling interaction gestures and nativeoptimized rendering pipelines. Frontend functions such as managing reusable UI modules input fields titles modals and user authentication validation via Yup and React Hook Form utilize a global state management store Zustand for maintaining authentication tokens and session metadata accessible from the service layer for authorization headers and persistence via local storage. Finally the integration of device functions camera image collection via Capacitor in combination with Ionic UI components and React structures supports a modern experience with AIdriven processing. Thanks to modular components reactive state management realtime communication and crossplatform UI the frontend constitutes a responsive and extensible foundation for the messaging application maintaining its responsiveness under highinteraction conditions and enabling the integration of advanced features e.g. sentiment analysis summarization without architectural changes.

5.2.1 State Management and React Query

This is important since React Query allows developers to effectively manage their internal state that needs to be in sync with their backend server by enabling them to fetch, cache, and update data in a single method, as opposed to using complex and cumbersome logic that would be required when using implementations of useState or creating custom state management libraries. This allows developers to effectively create good UIs either by using Native components or by using React Native. This approach is considered more suitable for such an application, where data must remain up-to-date without triggering unnecessary network calls or excessive re-renders. At the foundation of the frontend, a single instance of QueryClient has been implemented, in which the overall behavior of queries across the entire application is configured with one key characteristic: staleTime is set to five minutes, after which fetched data is considered fresh for a few more minutes. Therefore, unless necessary, a reload is not required. This was implemented as follows:

```
const queryClient = new QueryClient({defaultOptions: { queries: { staleTime: 1000 * 60 * 5 } },});
```

The client is introduced via the QueryClientProvider, enabling the declaration of queries or mutations in any component. With this design, features such as background refetching and cache hydration are activated. An example of React Query usage appears on the inbox page, where the conversation list is retrieved using the specific hook (useQuery). The hook associates the dataset with the unique key ['chats'] and links it to the getChats() function, which communicates with the backend. The query for the inbox page is defined as follows:

```
const { data, isLoading, refetch } = useQuery({ queryKey: ['chats'],queryFn: getChats,
refetchOnMount: 'always', refetchIntervalInBackground: true,
});
```

In this way, React Query handles the entire lifecycle of a request, and the conversation list is loaded when the component is mounted and is cached. The data on your browser will be refreshed when the page enters into the foreground again and when loading an inbox page with cached data while waiting for the new responses from the backend. The refetch() function allows loading component data (e.g., Create Chat modal) for all new conversations to be created. Then, operations that can change state are implemented using the useMutation hook, which integrates asynchronous interactions with the backend. The chat page extensively applies this pattern. Sending a new message, updating message metadata, deleting conversations, or marking messages as read are based on mutations. A characteristic example is the mutation that sends a new message to the server:

```
const { mutate, isLoading: messageIsLoading } = useMutation({ mutationFn: ({ chatId, newMessage, image }) => sendMessage({ chatId, newMessage, image }), });
```

Once the mutation has been executed and the server has responded with the updated message, the component processes the response to add the message to the local state; then, it lets all the other users know that they have an updated version of the message via the WebSocket channel:

```
socket?.emit('send_message', messageData);
```

The retention of permanent data and the distribution of that data to other users occurs via the server when a user is presented with a message. Chat history (initial lists of messages), pagination of individual messages and the inbox are retrieved using REST calls which may be executed through either queries or mutations; however, the rest of the new messages that are added to the chat page are sent to users via WebSocket events and are passed to the local state by the chat page via a listener that adds those incoming messages directly to the local state.

```
socket?.on('receive_message', (message) => { setMessages((prev) => [...prev, message]);});
```

Using React Query to store both the original data and Socket.IO connections in a cache means no need for repetitive polling or manual syncs between different parts of the application. This process also helps avoid inconsistencies as users move between pages. For example, once the chat page is loaded, the component loads up the entire chat history via a mutation (mutateChat) and only the WebSocket events will be used to update the UI visually. Typically, Queries are for quickly loading data; Mutations are used to make controlled changes; and WebSockets provide access to real-time data and do not impact the flow of the application. Therefore, using this architecture improves the application's overall performance because there are fewer repeat API calls & keeps the component's logic simple so that it can grow easily in size (scalability).

5.2.2 WebSocket Architecture (Socket.IO)

All modern messaging applications need to support real-time communication between users. In this system, this is accomplished using Socket.IO to provide the client and backend server with a bidirectional communication channel that allows messages, status updates, and system events to be sent without the need for polling or multiple HTTP requests. The websocket architecture, which is used in the frontend, has been developed to be used as efficiently as possible with the react query library and the REST API layer.

A custom hook is used to develop a socket connection, and the name of the custom hook is useSocket. The custom hook is used to develop a socket connection, and the complete lifecycle of the Socket.IO client will be handled by the custom hook. The custom hook will handle the connection, the cleanup of the connection once the connection has ended, and the events that are bound to the socket connection. The complete structure of the custom hook is given below:

```
export const useSocket = () => {
 const [socket, setSocket] = useState<any>();
 useEffect(() => {const s = io(`${API_URL}`); console.log('Socket connected'); setSocket(s);
    return () => { s.disconnect(); console.log('Socket disconnected'); };
  }, []);
  return { socket };
};
```

A WebSocket connection is established by calling useSocket. This connection remains active while the component is mounted. The connection will be terminated correctly when the page is closed via the disconnect() method. This is done to avoid unnecessary active connections, which would consume resources and might cause the creation of duplicate listeners due to frequent renderings of the component. The socket created by the hook is used throughout the entire application. As soon as a user enters a chat, the socket joins a room corresponding to the chatId of the conversation:

```
useEffect(() => {socket?.emit('join_room', chatId); }, [socket]);
```

By joining the room, the server can emit events only to users in that specific room, thereby reducing network overhead and ensuring that clients receive updates exclusively for the conversation in which they are participating.

To send a new message, on the chat page, the system listens for the receive_message event:

```
useEffect(() => {
  socket?.on('receive_message', (message: any) => {
    setMessages((prevMessages) => [...prevMessages, message]);
  });
}, [socket]);
```

When the backend sends out the new message through broadcasting for the room, the frontend simply adds it to the local array for messages. This action is done independently from the cached server state via React Query, which allows the user to view the message immediately. Once the message has been stored through the REST-based mutation, the frontend sends out the real-time event:

```
socket?.emit('send_message', messageData)
```

The server sends the message to all active clients in the room, thus creating a synchronized environment. The inbox page also interacts with the WebSocket layer through the registration of the following listeners:

```
socket?.on('new_message', (data: any) => { … });
```

Socket.IO handles real-time network updates, while React Query serves the initial data retrieval and maintains server state integrity. This eliminates repeated query refetches and keeps the UI functional, so there is no need for polling and no network idle time, ensuring scalable performance with hundreds of concurrent users and active conversations.

5.2.3 Chat Interface

The chat interface serves as the most interactive and dynamic part of the frontend by integrating real-time communication, message retrieval on one screen, and managing different kinds of media into a single conversation experience with multiple subsystems. Some of those subsystems include using React Query for data fetching, utilizing Socket.IO for real-time synchronization between users, using Ionic components to render the UI elements of the interface, and using Capacitor to capture multimedia.

The chat page is structured as a React functional component that manages the entire lifecycle of the conversation via a combination of local and remote state. Requests in the local state of this functional component could include variables such as the value of the message input, the list of messages that have been loaded, the metadata of the active conversation, pagination flags, and temporary attachments of images. Together, these variables allow the chat page to respond quickly (in real time) to user input, scrolling events, and to incoming messages as they are being received without having to wait for any remote data to be received.

A chat session is initialized when the component is mounted or after the chatId has been updated in the component. A React Query mutation is used to get an initial history of the conversations via a mutation that will call upon the getChat function to load in the last message(s) that was sent between the two parties.

```
useEffect(() => { setIsLoading(true);  setPage(1); mutateChat({ chatId, page: 1 }); }, [chatId]);
```

Once the data is received, the component updates its state and automatically scrolls to the bottom of the conversation, ensuring the most recent messages remain visible. This scrolling is implemented through an IonContent reference that provides access to the underlying DOM element: υλοποιείται μέσω ενός IonContent reference που παρέχει πρόσβαση στο υποκείμενο DOM στοιχείο:

```
contentRef.current?.scrollToBottom(300);
```

Pagination is integrated into the interface via the onIonScroll event. When the user approaches the top of the message list, the component loads older messages by incrementing the page counter and calling mutateChat with the next page number. The interface maintains the user's scroll position by calculating the height difference before and after inserting the new messages. This technique provides a natural scrolling experience even in conversations with extensive history.

When the user composes a new message, the message lifecycle is in its most interactive stage. The user can send the message by clicking the send button or by pressing the Enter key:

```
const handleNewMessage = () => {
  if (newMessage === '' && !image) return;
  mutate(
    { chatId, newMessage, image },
    {
      onSuccess: (res) => {
        const messageData = {...res.chat.messages[res.chat.messages.length - 1], room: chatId};
        socket?.emit('send_message', messageData);
        setMessages((prev) => [...prev, messageData]);
        contentRef.current?.scrollToBottom();
        setNewMessage('');
      }} );};
```

The message follows a two-stage process. First, it is stored on the backend via a REST request, which returns the authoritative message object that has been stored in MongoDB. Then, the frontend emits a real-time send_message event through the WebSocket connection, so that all participants in the room receive the update immediately. In parallel, the message is added to the local messages array, providing instant visual confirmation to the sender.

Incoming messages from other users follow a complementary flow. The component registers a listener for the receive_message event:

```
socket?.on('receive_message', (message) => {
  setMessages((prevMessages) => [...prevMessages, message]);
});
```

This listener integrates real-time updates into the active conversation without requiring API refreshes, enabling low-latency communication. The message bubbles are rendered through a specialized MessageBox component, which formats the content depending on whether the message is text-based or contains an attached image. The alignment of the bubbles is determined dynamically by comparing the sender's identifier with the current user's identifier, resulting in left alignment for incoming content and right alignment for outgoing content.

Multimedia content exchange is supported through Capacitor's camera and photo gallery APIs, allowing users to take or select images:

```
const image = await Camera.getPhoto({
  resultType: CameraResultType.DataUrl, source: CameraSource.Camera,
});
```

The generated Base64 data URL is stored in the state and sent together with the text content when the message is submitted. This integration highlights how the chat interface bridges device-level capabilities with backend storage and real-time distribution.

Read receipts are also incorporated into the message lifecycle. When a new set of messages is loaded, the component checks whether the latest incoming message has been viewed. If not, a mutation is triggered to mark the message as read:

```
readMessageMutate({ chatId, messageId: lastMsg._id });
```

This ensures consistency across multiple devices and accurate tracking of unread messages on both the chat page and the inbox page.

The chat interface also has auxiliary actions, which include deleting a conversation, accessing chat details, and handling group members. These actions use additional mutations (deleteChat, addMembers, removeMember) and modal components that allow for contextual interactions without having to navigate away from the chat page.e.

Overall, the above mechanisms compose a complete message lifecycle. Messages are created locally, validated and stored via the backend, distributed to connected clients through Socket.IO, and maintained

in the frontend state for immediate rendering. Older messages are retrieved on demand via paginated requests, while read receipts ensure accurate synchronization across all facets of the application. The chat interface thus constitutes a highly dynamic environment, where the UI's immediate responsiveness, the integrity of remote data, and real-time event propagation converge into a cohesive user experience.

5.2.4 Authentication and Form Validation

The authentication in the application is achieved through a structured combination of controlled form management, client-side validation, and asynchronous communication with the backend API. The frontend offers two basic authentication methods, login and registration, which are built around a predictable validation and data submission process. This process ensures that only valid and well-formatted data is submitted to the server, while at the same time offering feedback to the user about input errors.

The form management is achieved using React Hook Form, a lightweight and efficient library that manages form state with minimal re-renders. Each authentication page initializes a form instance via the useForm hook, which tracks field values, handles submit events, and exposes validation information. The forms are strictly typed using the TypeScript interfaces LoginConfig and RegisterConfig. These interfaces precisely define the structure of the expected inputs, enhancing type safety and preventing accidental misuse of the authentication API.

Input data validation is performed through the Yup library. Validation rules are linked to React Hook Form using yupResolver, allowing form fields to be automatically checked against schema definitions before submission. The validation rules for the registration form are defined as follows:

```
export const registerSchema = yup.object().shape({
  phone: yup.string().required("Invalid phone"),
  username: yup.string().required("Invalid username"),
  password: yup.string().required("Invalid password"),
});
```

Similarly, the login form is validated according to the following schema:

```
export const loginSchema = yup.object().shape({
  username: yup.string().required("Invalid username"),
```

```
  password: yup.string().required("Invalid password"),
});
```

Every time the user attempts to submit these forms, the resolver will validate these fields according to the schema. If any field is invalid, React Hook Form provides clear error messages through the errors object. These error messages are displayed immediately below their respective input fields. This way, users can easily correct these errors without making any network requests. This makes the authentication process much more robust.

After this client-side validation, the data will then be passed to the useMutation hook, which is provided by React Query. This hook will handle all the asynchronous operations with the server. The authentication service, which includes functions like loginUser and registerUser, includes all the HTTP communication-related logic. This service is implemented using Axios and communicates with the server's authentication routes:

```
export const loginUser = async (payload: LoginConfig) => {
  const response = await Axios.post(`${API_URL}/auth/login`, {
    username: payload.username, password: payload.password,
  });
  return response.data;
};

export const registerUser = async (payload: RegisterConfig) => {
  const response = await Axios.post(`${API_URL}/auth/register`, {
    phone: payload.phone,
    username: payload.username,
    password: payload.password,
    avatar: payload.avatar,
  });
  return response.data;
};
```

The login page demonstrates how mutations are integrated into the component's flow. When the authentication request is completed successfully, the result is passed to the global state management system based on Zustand. The logIn method stores the token, user ID, username, and avatar inside a persistent state container:

```
logIn({ token: data.token, userId: data.userId,  avatar: data.avatar,  username: data.username });
```

The local storage is where the Zustand persistence middleware saves the state. The authentication state keeps the user logged in after refreshing their web browser or closing and re-opening the browser. The application will redirect the user to the inbox page, completing an entire login cycle.

The authentication user interface is created using Ionic components such as IonInput, IonButton, and IonCard to provide a clean and mobile-friendly user interface (UI). The UI also has a password visibility toggle provided by the HidePassword component. Additionally, the UI has toast notifications to provide users with immediate feedback when they submit the information entered into the form. The input fields in the form are styled to visually display validation errors by highlighting those fields when validation fails. All of the validation logic and feedback to the User for any validation errors occurs locally. Only after successfully validating the submitted data will the application communicate with the backend.

Using a combination of React Hook Form for forms and form validation, Yup for form validation, Axios for service functions, React Query for service function mutations and Zustand for persistent state management, the authentication process achieves a level of reliability, usability, and clarity. Each of these items in the authentication process has a distinctly defined role; validation maintains that the information is correct, form management dictates how the User interacts with the form to enter information, the service layer implements secure communication and provides a global store to maintain the authenticated User state. When combined, all of these parts represent a consistent and secure authentication solution that will integrate with the overall architecture of the frontend.

5.2.5 Centralized Authentication Management with Zustand

Zustand is a light and expressive state management library for React that manages the core authentication of the application. Zustand was chosen because it requires little boilerplate code and has great TypeScript integration, which allows for quickly creating a shared global state that does not have the complexity associated with larger libraries. The use of a central, persistent store for authentication information (access token, user profile information) allows for that data to be accessed across all pages of the client application, provides consistency across pages and allows for reliable session management, and separates presentation logic from application state.

Zustand's create function will create the authentication store and the persist middleware will automatically persist the value of certain fields of the authentication store to the browser's local storage. The user's authentication will remain persistent between page refreshes and upon restarting the application. The initial state of the authentication store includes fields for isLoggedIn, token, userId, username, and avatar that are initialized with a value of either an empty string or false. The authentication store exposes two methods for managing authentication transitions: logIn and logOutUser.

The authentication store will be implemented as follows:

```
export const authStore = create<AuthState>()(
  devtools(
    persist(
      (set) => ({isLoggedIn: false, token: "",  userId: "", avatar: "",  username: "",
        logOutUser: () => {
          set({ isLoggedIn: false,  token: "",   userId: "", avatar: "", username: "",});
        },
        logIn: (payload: any) =>
          set({ isLoggedIn: true,  token: payload.token,  userId: payload.userId, avatar: payload.avatar,
            username: payload.username,}),
        updateUser: (payload: any) =>
          set((state) => ({ ...state,  ...payload,  })),
      }), { name: "auth-storage-client",} ) ));
```

All authentication information is held in the global store. The global store is the only source of truth for authentication information. If a request to authenticate is successful, the logIn function is invoked. This function stores the data provided by the backend, including the token, and broadcasts it to all components that listen for changes in the global store. These components can then update the rest of the application to reflect that the authentication process has been successful. When a user decides to log out, the logOutUser function is invoked. This function removes all authentication data from the store and denies access to a restricted part of the application until the user logs in again.

The helper function getAuthState provides the ability to access the current state of authentication without requiring a component to subscribe to global store changes. This is particularly useful in the service layer since the service must use the authentication token when adding the token to the header of an HTTP request:

```
export const getAuthState = () => { return authStore.getState();};
```

The chat service functions rely on this helper to include the authorization token in the headers of every protected API call. The request configuration is built using the stored token:

```
const getConfig = () => { return {headers: {Authorization: `Bearer ${getAuthState().token}`, }, };};
```

By using a token or some unique ID for each user, this method allows for better and more consistent usage of that token in all service requests. It also avoids having much of the logic repeated throughout the entire codebase. Last, when an authentication token gets modified, updated, or refreshed, the token is sent back to all of the API calls throughout your application, thus ensuring that all API calls will have the updated token value.

The interaction between Zustand and React Query is just as important too. React Query is responsible for communicating with a remote server to get data; however, React Query does not include a place to store or maintain any information regarding authentication credentials. Zustand will help to bridge this gap by managing the authentication state of the user, while React Query performs the operation of communicating. When a user logs out of the application, even though there is nothing to remove from the Zustand store, the context for authenticated queries becomes invalid because the Zustand store no longer has any valid credentials. Therefore, while logging out of your account, the system will clear your interface of authenticated requests.

Another benefit of using Zustand is that your user information (avatar, user name, and token) will continue to be stored in local storage. This means that when you reload the page / application again, Zustand will check local storage, and if there is valid user information stored there, it will initialize your interface to an authenticated state. This helps significantly with user experience because users are not required to log back into their account every time they start the application, creating a better sense of continuous user experience.

5.2.6 UI Components and Ionic Integration

The frontend interface is based on a set of reusable components that include common interaction patterns and formatting conventions. The design of the frontend interface is based on a component-driven architecture, which uses small and well-defined components as building blocks for the entire interface.

The system breaks down common behaviors into independent components to ensure that the application's visual identity is consistent on all pages.

A characteristic example is the custom input component. This component extends the behavior of the Ionic IonInput element, connecting it with form-binding capabilities and applying predefined styling. It accepts props for labels, classes, and event handlers, allowing flexibility in every use while maintaining a unified design:

```
<IonInput {...register} label={label}  onIonChange={onIonChange} />
```

By using this template, all authentication and settings pages display input fields with identical structure and behavior. At the same time, the integration of validation is simplified, as each input passes through the same form bindings and event handlers of React Hook Form.

A similar logic is followed for the button component, which is a wrapper for the Ionic IonButton component. This component brings together the formatting rules for the shape, color, and layout of the button, so that all interface actions are consistent:

```
<IonButton onClick={onClick} routerLink={routerLink}>
        <IonIcon icon={icon} slot={iconSlot} size={size}></IonIcon> {name}
</IonButton>
```

By integrating both icon placement and visual styling into the same component, the application avoids repeating layout logic across pages such as login, registration, chat creation, and chat interaction.

The image picker component further highlights the value of abstraction. It handles file selection, Base64 conversion, and image preview rendering within a single, self-contained module. Pages that use this component do not need to implement any of the low-level details of image handling:

```
<input type="file" onChange={handleImage} ref={imageRef}/>
        {image && ( <img onClick={handleImageClick} src={image} alt={name} />
)}
```

This component can be used to create profiles, set up group chat settings, and execute all functionality that requires image uploads. The Base64 conversion is done internally to the component, leaving the rest of the application's code to send a ready-to-use value to the backend.

Using components like these is essential in producing an extensible and maintainable frontend. Each portion of your application experiences the benefits of an expected and reusable interface while giving developers the opportunity to introduce new features without having to re-implement shared UI functionality. Moreover, developing with components ensures visual uniformity, streamlines the design system's modification process, and allows you to isolate complex behaviour from the pages that consume them. Finally, using a modular structure will also enhance the ability to test as each unit of UI can be tested on its own.

5.2.7 Real-Time Voice and Video Calls with WebRTC and Socket.IO

The application's ability to make and receive voice and video calls in real time is achieved via a dedicated 'abstraction layer' that combines all WebRTC and Signaling Logic into a custom React Hook called 'useWebRTC'. The logic and implementation of WebRTC is fully decoupled from the front end interface of the chat application. As a result, chat components have simpler structure, are easier to maintain, and there are fewer issues with the real time flow of conversations (e.g., audio/video) between users.

Rather than embedding WebRTC within the chat component, the application exposes a declarative interface to peers/clients connected through the chat. The declarative interface has internal logic to handle Peer Connections, Media Streams, Signaling Events, and Call/Conversation States. This approach allows the chat component to have access only to the defined State Values, References, and Action Functions that are required for Presentation and User Interaction.

When initialized, the hook can use Context Data that includes information about the current Socket.IO Connection, Chat Room ID, Local & Remote User IDs, and the Call Ringtone Source; these contextual pieces of data limit all Calls to the specific Peer User Pair and associated Chat Room, which ties to the Room Based Model established in the application messaging.

```
const {
    callType, inCall, incomingCall, localVideoRef, remoteVideoRef, localAudioRef, remoteAudioRef, startCall, acceptCall, rejectCall, endCall, toggleMute, flipCamera} = useWebRTC({
    socket, roomId: chatId!, localUserId: userId, remoteUserId, ringtoneSrc: ringtonePlayer,
});
```

The controlled call state machine is where the actual implementation takes place. The machine can tell if there is an idle state, an incoming call state, or an active call state. Internally, the hook tracks if the local user initiated the call, if the call has been accepted, and if the media negotiation has occurred. As a result, premature SDP handling will not occur and WebRTC offers will only be handled when the user provides explicit consent. This is a critical need for usability as well as for security policies enforced by the browser.

Signaling for incoming calls is managed asynchronously through Socket.IO events. When an incoming call event is received, the hook stores the call metadata and activates a repetitive ringtone, while simultaneously exposing the incomingCall state to the UI. This allows the chat interface to display a modal-style overlay with accept and reject controls without blocking the underlying chat view.

```
socket.on('incoming_call', (data) => { setIncomingCall(data); startRingtone(); });
```

Once the user accepts the call, the hook transitions to an active call state, initializes the peer connection if required, and proceeds with the WebRTC negotiation process. The design explicitly separates call acceptance from SDP handling. If an offer arrives before the user accepts the call, it is temporarily stored and processed only after acceptance. This approach avoids race conditions and prevents the unintentional activation of media streams.

The WebRTC peer connection is created dynamically and configured with public STUN servers to support NAT traversal. Event listeners are added for receiving remote tracks and generating ICE candidates. Incoming media streams are directly attached to audio or video elements via React refs, eliminating the need for intermediate state updates and avoiding unnecessary re-renders.

```
pc.ontrack = (event) => {
  const remoteStream = event.streams[0];
  if (remoteVideoRef.current) { remoteVideoRef.current.srcObject = remoteStream; }
};
```

Local acquisition of media happens lazily and once per call only. Depending on if it's an audio or video call the right constraints will be requested through `getUserMedia` as part of a call to the `MediaDevices` instance. Media tracks will only be added to the peer connection once so that they can be reused throughout the life of that call; thus no duplicate track registrations and simplified negotiation logic will exist.

While actively connected to peers media level changes happen directly to the media tracks when switching cameras and muting microphones rather than renegotiating the associated audio or video streams on the peer connection. To mute a microphone just change the `enabled` property of the audio track(s) to mute/unmute (mute) them. For camera switches replace the currently active video track with another currently active video track using `RTCRtpSender.replaceTrack` Each of these actions is done without any disruption in call connectivity since they are performed seamlessly, thus allowing you to maintain your call while doing so.

```
const sender = pcRef.current?.getSenders().find((s) => s.track?.kind === 'video');
sender?.replaceTrack(newVideoTrack);
```

Using Socket.IO only for coordinating WebRTC offers, answers and ICE candidates, allows for exchange to occur entirely through Socket.IO as a coordination channel, then also utilize P2P for any subsequent transferring of audio/video (less latency and server workload).

The hook is responsible for centrally managing all possible call termination cases (rejection of call, termination by either party, or unexpected termination), by performing the necessary cleanup logic so that the media tracks stop playing, peer connections are closed, and the UI state is reset upon the next call. This way, a previously active microphone and/or camera will not be left in an active state causing both a problem with resources and/or creating a potential security risk (leak).

From the chat component's perspective, the rendering logic becomes simple and readable. When the `inCall` state is true, the interface transitions to an audio (or video-call layout). The component simply connects the provided refs to the respective media elements and renders call controls based on the exposed state.

```
{inCall && callType === 'video' && (
  <div className="video-call-ui">
    <video ref={localVideoRef} autoPlay muted playsInline />
    <video ref={remoteVideoRef} autoPlay playsInline />
  </div>
)}
```

By factoring out WebRTC-related functionality into a hook that can be reused throughout the application, we have been able to clearly separate our concerns. The chat interface remains easy to read and understand as a declarative piece of code, while our real-time communication layer operates independently with well-defined boundaries. This architectural pattern has greatly improved our

application's scalability and maintainability while providing opportunities for features like group calls and screen sharing without having to change our existing chat component.

5.3 Backend Implementation

The backend of the messaging application is based on a service-oriented architecture and meets the following three key requirements for a modern communication system: First and foremost, there must be an HTTP API for the naturally requested-response operation of registering an account, authenticating a user, creating a chat, and retrieving the message history. Secondly, there has to be an element for real-time communication that can transmit information in real time, which includes new messages, call invitations, and WebRTC signaling information. Thirdly, there has to be reliable data storage for all the information required by the application, which ensures the application is consistent regardless of the device/session and can survive network interruptions. The technologies used for the implementation of the project include Express.js for routing and middleware HTTP requests, MongoDB with Mongoose for reliable data storage, and Socket.IO for real-time communication.The architecture is designed as a hybrid; REST endpoints are the authoritative source for state modification and data storage, while Socket.IO will be used for real-Time client synchronization and establishing peer-to-peer calls. This is done on purpose, since this model allows each message, conversation, and user action to be stored in a deterministic way, while simultaneously providing the user with an interactive experience that feels immediate. From a practical point of view, the backend is organized into three main implementation areas. The first is the Express API layer, which is used for defining routes, implementing authentication, request validation, and response formatting. The second is the Socket.IO layer, used for managing rooms and the distribution of events for messaging and calls. The third is the database interaction and storage model layer, which is used for defining how users, conversations, and messages are represented in MongoDB.

5.3.1 Authentication and Access Control

User registration, login, and ID verification are all handled by the backend authentication feature and its various implementations, such as using Express.js as a framework for the HTTP layer, MongoDB with Mongoose as the data layer for persistent storage of data, JSON Web Tokens for stateless authentication, and bcrypt for hashing passwords. The result of using each of these technologies is a secure, scalable,

modern-day authentication mechanism that is ready for implementation in a real-time messaging platform.

At application startup, the Express server initializes global middleware for JSON-parsing, CORS handling, and processing of incoming request bodies. Furthermore, all authentication routes will be grouped together within the /*auth path to establish a clear separation of identity-based functionality from the rest of the application's business logic. This separation improves code readability, simplifies managing access controls, and allows the authentication subsystem to develop separately from chat and messaging capabilities.

```
app.use('/auth', authRoutes);
```

The data for each user is stored in MongoDB via the Mongoose schema containing defined constraints (structure) for each user document/schema (record). Each document/schema will include (at a minimum) a phone number, username and encrypted password; optionally an avatar image URL can also be included. By enforcing required fields at the schema level, data integrity is maintained; storing only hashed v. clear-text passwords ensures that sensitive data isn't ever stored in a clear-text fashion. Additionally, because of MongoDB's flexible document model, it is possible to extend (modify) the schema in the future without having to make major changes (radical modifications) to the schema.

```
const userSchema = mongoose.Schema({
  phone: { type: String, required: true }, username: { type: String, required: true }, password: { type: String, required: true },
  avatar: { type: String },
});
```

To register, a post request is sent to the endpoint /auth/register. This request contains the user's information as well as optionally their avatar image. The first step of the backend is to see if there's already a user with that user name since there cannot be two accounts with the same name in the system, and therefore when the user tries to log in, the system will be able to identify them consistently throughout all parts of the application. If there is no one using the same user name, the next step is to hash the password with bcrypt using a specified cost factor in order to protect against any breaches of security if the database is hacked. If the image is included as an avatar, the avatar will be sent to Cloudinary stored as a base64 string and then returned via a secure URL, which will be saved in the user document. After creating the user, the backend will generate and return a JSON Web Token (JWT) that will grant the client access to the user's Id and user name signed using an environmental secret key so that the client can begin to access protected resources immediately after registering.

```
existingUser = await User.findOne({ username: username });
hashedPassword = await bcrypt.hash(password, 12);
result = await uploadImage(avatar);
token = jwt.sign({ userId: createdUser.id, username: createdUser.username },process.env.JWT_SECRET);
```

The login process is restricted but also based on similar logic. With a POST to `/auth/login`, the backend will look for a user by their username. If there is no user, it will return a generic error message instead of allowing a user to know if an account exists. If there is a user, the submitted password is compared against the hashed password stored in the database.

If the password is validated, a new JWT will be generated and returned along with basic profile information on the user. The benefit of stateless authentication is that it allows you to avoid maintaining session information on the server and gives you the ability to vertically scale.

```
isValidPassword = await bcrypt.compare(password, existingUser.password);
res.status(200).json({ userId: existingUser.id, token: token, avatar: existingUser.avatar, username: existingUser.username });
```

JWT provides a way to secure any remaining endpoints. The middleware retrieves the token from the `Authorization` header, validates (and decodes) it; if it can be decoded successfully, `userId` will be added to the request object; if not, the request will be denied with an authorization error. This middleware is used for both chat and message routes, ensuring consistent access control across the entire application.

```
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.userId = decoded.userId;
```

5.3.2 User Management

User management constitutes a core subsystem of the application's backend, as it supports all functionalities related to retrieving, presenting, and updating user information during the application's use. In contrast to the authentication mechanism, which is focused on identification and access control, user management is primarily concerned with making use of stored profile information for proper functionality of social and communication features within the application. At the backend level, user management is carried out using specific REST endpoints, which can only be accessed after a user is authenticated. Using this, users can be searched, and information can be retrieved, including the basic user profile and the information required to create a conversation and identify the participants within it. This is done with the help of authentication middleware, which ensures that a JSON Web Token is sent with every request and

the user is already verified. The user model in the database includes core identity elements, such as username and phone number, as well as presentation details like the profile picture. This data is extensively used on the frontend for displaying user lists, conversations, and messages. It is particularly important that the backend never returns sensitive data, such as the encrypted password, limiting itself to information absolutely essential for the application's operation.The management of the profile picture is achieved through integration with another media storage service. When a user registers, the image is transmitted to the backend in base64 format and then to cloud storage, and a secure URL is received. The received URL is then stored in the database, and this URL will be used in the frontend to directly access the image. This method is useful in preventing the storage of large data in the database, hence improving the performance of the system. One of the major features of user management is the search functionality for users. This is important for starting a new conversation. The backend allows the frontend to search for users by username, returning lists that can be dynamically filtered and displayed. This function is especially critical for private chats, as it allows a user to locate other registered users without needing to know unique identifiers. Concurrently, the system maintains additional fields related to the application's functionality, such as notification tokens for future push notification support. The presence of such fields in the user model demonstrates that the backend design was carried out with future extensibility and the integration of additional features in mind, without requiring structural changes.

5.3.2 Conversation Creation and Chat Management

Conversation establishment and management via the back end come through the RESTful API using the /chat route. All business logic associated with conversations, participants, and messages exists within a single router, separating out the actual methods for implementing those business capabilities into their own respective controllers. As a result, we develop clear routes and develop business logic that is easy to understand and manage, ultimately supporting the maintainability and scalability of our project.

All routes import the protect middleware providing that only authenticated users can perform actions and that the back end is always aware of an authenticated user's identity in the request through req.userId.

```
router.post("/", protect, chatControllers.createChat);
router.get("/", protect, chatControllers.getChats);
router.get("/:chatId", protect, chatControllers.getChat);
router.put("/:chatId", protect, chatControllers.updateChat);
```

```
router.delete("/:chatId", protect, chatControllers.deleteChat);
```

Mongoose is utilized for storing conversations and supports two conversation types: private and group. Private conversations consist of two members, while group conversations additionally support a name and an avatar.

```
type: { type: String, enum: ["private", "group"], required: true },
members: { type: [mongoose.Schema.Types.ObjectId], ref: "User", required: true },
creatorId: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
```

A core design advantage is that messages are stored as an embedded array within the Chat document. The messages field is defined as an array of messageSchema, allowing the retrieval of a conversation along with its message history without requiring additional joins or separate collections.

```
messages: { type: [messageSchema], required: true }
```

The messageSchema includes elements such as the sender, the text, a potential image attachment, and the read status. The existence of the read boolean supports functionalities like unread indicators and read receipts on the frontend.

```
senderId: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
read: { type: Boolean, default: false },
image: { type: String },
```

The timezone for the moment-timezone package is set on the default timezone of "Europe/Athens" meaning that all of the dates will be stored using the same format regardless of what the client is.

```
moment.tz.setDefault("Europe/Athens");
createdAt: { type: String, default: () => moment().format("YYYY-MM-DD HH:mm:ss") },
```

When creating a private conversation, the backend checks to see if a private conversation already exists with the same participants using the $all method of the query structure, thus preventing multiple private conversations existing between the same users.

```
const existingChat = await Chat.findOne({ members: { $all: members },  type: "private" });
if (existingChat) { return res.status(200).json({ message: "ok", chat: existingChat }); }
```

For group conversations, users can set the conversation's avatar themselves for all participants. If set, this image is stored in cloud storage via the uploadImage function, while only the URL is saved in the Chat document. The creatorId is derived exclusively from the JWT (req.userId), protecting the system from spoofing.

```
let result;
if (avatar) { result = await uploadImage(avatar); }
const chat = await Chat.create({ name, type, avatar: result || "", members, creatorId: req.userId });
```

This endpoint retrieves all chats that the user is a member of. Using populate on members, creator, and message senders enables the frontend to have access to profile information without having to perform additional API calls.

```
const chats = await Chat.find({ members: req.userId }).populate( "members creatorId messages.senderId", "-password");
```

Retrieving a chat includes pagination of messages through slicing of the embedded array. This approach avoids complex querying and supports infinite scroll on the frontend.

```
const page = req.query.page ? parseInt(req.query.page) : 1;
const limit = req.query.limit ? parseInt(req.query.limit) : 20;
const total = chat.messages.length;
const start = Math.max(total - page * limit, 0);
const end = total - (page - 1) * limit;
const paginatedMessages = chat.messages.slice(start, end);
const hasMore = start > 0;
res.status(200).json({ message: "ok", chat: chat, hasMore: hasMore });
```

```
let query = { $set: {} };
for (let key in req.body) { query.$set[key] = req.body[key]; }
await Chat.updateOne({ _id: req.params.chatId }, query).exec()
const chat = await Chat.findByIdAndDelete(req.params.chatId).exec();
res.status(200).json({ message: "ok", chat: chat });
```

Adding a member checks if the user is already in the specific chat, while removal is implemented by filtering the members list.

```
const existingMembers = chat.members.map((member) => member.toString());
if (members.some((member) => existingMembers.includes(member.toString()))) {
  return res.status(400).json({ message: "Some members already exist in chat", chat: null });
}
chat.members.push(...members);
await chat.save();
chat.members = chat.members.filter((member) => member.toString() !== memberId);
await chat.save();
```

5.3.4 Message Management and Storage

The main part of this application's functionality is message management because it links data stored and the real-time experience that the chat system provides. The backend not only has the ability to create and store messages but also has the ability to correctly associate messages with their respective conversations, manage read status and keep data consistent.

At the API level, message operations are grouped in the routing structure as conversations using a path of /chat/:chatId. This method maintains that every message will always be processed with respect to the chat it belongs to, which simplifies both backend logic and how the frontend understands the API.

```
router.post("/:chatId/message", protect, messageControllers.createMessage);
router.delete("/:chatId/message/:messageId", protect, messageControllers.deleteMessage);
router.put("/:chatId/message/:messageId", protect, messageControllers.readMessage);
```

All API endpoints are protected by an authentication middleware; this is to ensure that the backend has a trustworthy way of knowing who sent the communication to be able to implement access restrictions without depending on the data being sent from the client side.

Rather than having messages stored in a separate collection in the database, chat messages will be embedded as sub-documents within the conversation document. This makes sense for many reasons, but primarily because it allows for instant access to both chat messages and the conversations to which they pertain.

```
const messageSchema = mongoose.Schema({
  senderId: { type: mongoose.Schema.Types.ObjectId,  ref: "User", required: true },
  message: { type: String },
  read: { type: Boolean, default: false },
  image: { type: String },
  createdAt: { type: String, default: () => moment().format("YYYY-MM-DD HH:mm:ss") },
});
```

All messages contain sender details, message data, optional image uploads, and read/unread status. The read/unread status field enables the system to support features such as unread notification and visual

indicators in the front end. Using a specific timezone for createdAt enables all timestamps in the application to be consistent regardless of the user's device or geographic area. Messages are created through POST requests and can include both plain text and an optional image. The backend uses chatId to identify an existing conversation, and it creates a new message object associated with that conversation and pushes it onto the messages array. At this point, when sending a message, the backend will securely attach the sender to the message by using req.userId (which was extracted from the JWT), establishing the relationship between the sender and the actual message. After saving, the backend returns the updated chat object to the front end so the client can synchronise its local state with the database while waiting for the real-time message delivery via Socket.io.

```
{ senderId: req.userId, message: newMessage, image: image || "" }
```

Using a specific endpoint, we provide a way to manage the read status of a message by updating the `read` property of a message. Managing the `read` property is important in managing the user experience when distinguishing between messages that have been read and messages that have not been read. The backend will find the message inside the `messages` array and update it accordingly. The only time this will happen is if the user sending the request is not the sender of the message. This functionality will work directly with the frontend and Socket.io and will be immediately reflected in the user interface.

```
router.put("/:chatId/message/:messageId", protect, messageControllers.readMessage);
```

The deletion of a message is also possible using the endpoint 'delete'. The deletion of a message is used to delete a specific message sent in a conversation. On the backend side, the message is found using its 'messageId'. Then, it is deleted from the list of the remaining messages. The deletion of a message is done on the server side, and hence, it is not reflected on other devices or during other requests.

5.3.2 Real-Time Communication with Socket.io

The real-time communication within the application is facilitated by the use of Socket.io. While the API is responsible for the handling of the data store, Socket.io is utilized for the sole purpose of facilitating the real-time transmission of events between connecting users. This ensures an architecturally clean system. The server for the use of Socket.io is initialized within the same HTTP server as the Express application,

which allows for the sharing of the port and the cohesive use of both REST and WebSocket communication. The use of CORS within the context of Socket.io is required because WebSocket connections are not made under the same rules as HTTP connections. This is to ensure that the authorized frontend client is the one establishing the connection.

```
const http = require('http').Server(app);
const io = require('socket.io');
const socket = io(http, { cors: { origin: 'http://localhost:5173', methods: ['GET', 'POST'] },});
```

A unique 'socketId' is generated for each user that establishes a connection with the server. This ID serves as a temporary identifier for the connection and is also useful when debugging, but it does not replace the need for user authentication through JWTs. The socket server uses the fact that the user has been previously authenticated via the REST API in order to send and receive real-time events via the socket.

```
socket.on('connection', (socket) => { console.log(`user connected: ${socket.id}`); });
```

Each chat corresponds to a room, which uses the `chatId` as its identifier. In this way, all real-time events are confined to the participants of that specific conversation.

```
socket.on('join_room', (roomId) => { socket.join(roomId); console.log(`user with id: ${socket.id} joined room: ${roomId}`);});
```

When a user sends a message, it is not created via Socket.io on the backend. The message has already been saved to the database through a REST endpoint. Socket.io is used exclusively for broadcasting it to the other participants. This pattern avoids duplicate storage, keeps the database as a single source of truth, and updates the frontend immediately without polling. The socket just broadcasts the payload that the API returns, which is already validated and synchronized.

```
socket.on('send_message', (data) => { socket.to(data.room).emit('receive_message', data);});
```

5.3.6 Real-Time Audio and Video Communication with WebRTC

This application provides functionality beyond basic text messaging by adding real-time audio & video calling capabilities via WebRTC. WebRTC creates peer-to-peer connections without the need of an intermediate media server between users (reducing latency and enabling better quality). In this case,

WebRTC has been integrated into the application along with Socket.io, which is only used as a signaling protocol to facilitate the connection negotiation process. The communication logic is contained within a single frontend custom hook (useWebRTC). This hook contains all applicable state and function(s) to allow it to be separated from the rest of the application's UI. The hook also maintains whether a user is currently engaged on a call, call type and whether an incoming call exists; thus enabling the UI to dynamically adjust based on those changes.

```
const [callType, setCallType] = useState<'audio' | 'video' | null>(null);
const [inCall, setInCall] = useState(false);
const [incomingCall, setIncomingCall] = useState<IncomingCallData | null>(null);
```

In the case of WebRTC, a new call is based on an object referred to as `RTCPeerConnection`, which is created dynamically as soon as the process of communication begins. The connection is achieved using a list of STUN servers in order for users behind a NAT or a firewall to be able to communicate. The peer connection receives incoming media tracks and forwards them to the correct DOM elements. The peer connection monitors incoming media tracks and forwards them to the appropriate DOM elements.

```
const pc = new RTCPeerConnection({ iceServers: [{ urls: 'stun:stun.l.google.com:19302' }] });
pc.ontrack = (event) => {
  const remoteStream = event.streams[0];
  if (remoteVideoRef.current) { remoteVideoRef.current.srcObject = remoteStream; }
  if (remoteAudioRef.current) { remoteAudioRef.current.srcObject = remoteStream; }
};
```

The exchange of ICE candidates is also done through Socket.io.

```
pc.onicecandidate = (event) => {
  if (event.candidate) {
    socket.emit('webrtc_ice_candidate', {
      roomId,
      candidate: event.candidate,
    });
  }
};
```

The app will ask if it can use either your microphone, camera, or both depending on what you're doing during your calls. The app will obtain your media streams through

`navigator.mediadevices.getUserMedia`. It will then use those media tracks by attaching them to the peer connection so that the user can see what they look like locally. This allows the app to give the user real-time feedback.

```
const constraints = type === 'audio' ? { audio: true } : { audio: true, video: { facingMode } };
const stream = await navigator.mediaDevices.getUserMedia(constraints);
localStreamRef.current = stream;
```

Media tracks are added to the peer connection only once, avoiding unnecessary renegotiations.

```
localStreamRef.current.getTracks().forEach(track => { pcRef.current.addTrack(track, localStreamRef.current); });
```

The negotiation of the connection relies on the exchange of SDP offers and answers through Socket.io. When a user makes a call, they make an offer and send it to the other user through the server.

```
const offer = await pcRef.current.createOffer();
await pcRef.current.setLocalDescription(offer);
socket.emit('webrtc_offer', { roomId,  sdp: offer, type });
```

The recipient, having accepted the call, then processes the offer and sends an answer.

```
await pcRef.current.setRemoteDescription(new RTCSessionDescription(data.sdp));
const answer = await pcRef.current.createAnswer();
await pcRef.current.setLocalDescription(answer);
socket.emit('webrtc_answer', { roomId,  sdp: answer });
```

Incoming calls are tracked using socket events, and a corresponding UI overlay and ringtone will be displayed, mimicking the functionality of a traditional phone application.

```
socket.on('incoming_call', (data) => {setIncomingCall(data);});
```

```
localStreamRef.current?.getAudioTracks().forEach(track => (track.enabled = false));
```

The user can also mute the microphone or change the camera during an ongoing call. This is achieved by simply disabling the audio track without terminating the call. The camera can be changed by replacing the video track of the sender. This ensures a seamless transition without terminating the call.

```
const sender = pcRef.current.getSenders().find(s => s.track?.kind === 'video');
sender.replaceTrack(newVideoTrack);
```

Terminating a call involves cleaning up all media resources, closing the peer connection, and notifying the remote user, ensuring no active connections or background streams remain.

```
localStreamRef.current.getTracks().forEach(track => track.stop());
pcRef.current.close();
```

5.4 Artificial Intelligence Integration

5.4.1 Integration of AI Tools on the Frontend

AI functionality in the front end is integrated and still separates from the architectural structure of the rest of the system. The client will NOT call an AI model directly. The client will only call protected backend endpoints that will retrieve messages, create a transcript, call the model, and return the results. The front end is purely presentation and interaction and will only display ready-made results.
The service file contains all communications to the AI endpoints from the backend. A service file is important because it keeps HTTP calls to the AI endpoints from being distributed to different components of the application, this also allows for centralized management of the API URL and authentication headers. The Authorization header should contain a token that is retrieved from the global auth state for every request.

```
const getConfig = () => {return { headers: { Authorization: `Bearer ${getAuthState().token}`},};};
```

The `getChatSummary` and `getChatEmotions` functions are tasked with making POST requests to the backend. While these functions do involve "retrieval" of information, POST is used to allow the backend to possibly process additional parameters in the future, such as the number of recent messages, filters, or formatting, without being constrained by query strings or caching rules.

```
export const getChatSummary = async (id: string) => {
  const response = await Axios.post(`${API_URL}/openai/chats/${id}/summary`,{},getConfig());
  return response.data;
```

```
};

export const getChatEmotions = async (id: string) => {
  const response = await Axios.post(`${API_URL}/openai/chats/${id}/emotions`,{},getConfig());
  return response.data;
};
```

The use of a service layer offers additional maintenance advantages. If the API prefix changes, if interceptors are added, if retries are implemented, or if central error handling is set up, modifications need only be made in a single location. Consequently, the component remains clean and focused on the UI.
On the interface side, the AI functionality is implemented in a dedicated component (AiTools), which is passed the chatId as a prop. This is important because the AI is always used in the context of some conversation. The component maintains local state to store and display either the summary or the sentiment analysis result.

```
const [summaryText, setSummaryText] = useState<string>();
const [emotionText, setEmotionText] = useState<any>();
```

AI requests are executed using React Query's `useMutation`. This approach is particularly appropriate here because the operations are triggered by user actions and are not data that needs to load automatically on every render. Mutations provide built-in loading state management, callbacks for success and error, and consistency with the application's other async operations.

```
const { mutate: mutateSummary, isLoading: summaryIsLoading } = useMutation({
  mutationFn: ({ chatId }: any) => getChatSummary(chatId),
});

const { mutate: mutateEmotions, isLoading: emotionsIsLoading } = useMutation({
  mutationFn: ({ chatId }: any) => getChatEmotions(chatId),
});
```

When the user selects "Get Chat Summary", the component executes the mutation, and in the success callback, stores the result in state. The backend returns the summary as a plain string, making it immediately suitable for display in a text block.

```
const getSummary = () => {
  mutateSummary(
```

```
    { chatId },
    {
      onSuccess: (data) => {setSummaryText(data.result);},
      onError: (error) => {console.error('Error fetching chat summary:', error);},
    }
  );
};
```

Similarly, for sentiment analysis, the result is returned as JSON containing `overall_mood`, `emotions[]`, and `explanation`. The choice of a structured object significantly aids the UI, as it can be rendered in the form of cards, labels, and tags without the need to parse natural language text.

```
const getEmotions = () => {
  mutateEmotions(
    { chatId },
    {
      onSuccess: (data) => {setEmotionText(data.result);},
      onError: (error) => {console.error('Error fetching chat emotions:', error);},
    });
};
```

The presentation of the result uses conditional rendering. If a summary exists, a summary card is displayed. If not, but an emotion analysis exists, a tone card is shown. If nothing has been requested yet, a placeholder image appears. This logic is important because it provides a clear user experience. The component always shows something relevant, without leaving empty space, and avoids confusion about which result is currently active.

```
{summaryText ? (
 <p className="ai-paragraph">{summaryText}</p>
) : emotionText ? (
  <span className="ai-title">{emotionText.overall_mood}</span>
) : (
  <img src={ai} className="ai-image" />
)}
```

The overall design of the software is consistent throughout because of the use of Ionic elements for building the interface. The buttons are also able to respond to user actions through a visual indication by replacing their original icon with a loading icon as the action is being executed. As a result, the

application has a more responsive feel since the latency for an AI call may be longer than that of a normal API call.

```
<IonIcon icon={summaryIsLoading ? ellipsisHorizontalOutline : readerOutline}/>
<IonIcon icon={emotionsIsLoading ? ellipsisHorizontalOutline : happyOutline}/>
```

In the case of emotion analysis, the display of individual emotions uses low-cost UI elements like "chips". This transforms the AI result into easily readable labels that resemble tags. Consequently, the user receives not a large block of analytical text, but a concise, structured overview of the conversation's tone.

```
{emotionText.emotions?.map((e: string) => (<span key={e} className="ai-chip">{e} </span> ))}
```

5.4.2 Artificial Intelligence Integration in the Backend

The inclusion of AI functionality in backend development is intended to improve the chat user experience by offering two primary features. The first feature is the automatic summarization of chat messages, which can help users get a quick overview of the chat without going through the entire content of the chat messages. The second feature is the analysis of the tone of the conversation, which is meant to provide an overview of the tone of the conversation (friendly, neutral, intense, aggressive) and provide insights into the primary emotions that are present in the conversation. In this scenario, the backend is the mediator between the information that is contained in the database and the AI model. The server retrieves the chat from the database, checks the user's access to the chat, merges the messages into a single text (transcript), and then requests the AI model for a summary or analysis of the chat. The client does not have to handle a lot of information and does not have to know anything about the AI API or how to connect to it directly. The connection to the OpenAI API is initialized in a separate utility file, eliminating the need to "scatter" the API key across many different areas of code and providing for a centralized point of management of the API key. This approach results in increased security, increased maintainability, since any configuration changes to the API key can be accomplished in one location.

```
const OpenAIApi = require('openai');
const openAi = new OpenAIApi({apiKey: process.env.OPENAI_API_KEY});
module.exports = openAi;
```

The summarization is done as a standalone operation, which takes a transcript as input and returns a summarized text. The operation invokes the model using the `responses.create` method and sets specific instructions so that only the summary is returned. The presence of these instructions is critical, because in real-time applications the server needs predictable output. If the model returned additional information or reproduced the history, it would create noise in the UI and unnecessarily increase the size of the response.

```
const openai = require('../utils/openAi');
module.exports = async function summary(transcript) {
  const response = await openai.responses.create({
    model: 'gpt-5-mini',
    instructions:
      'You summarize chat conversations. Respond in English. ' +
      'Write only the summary, not all chats, not information of chat. in third person' +
      'If there is not enough information, say so.',
    input: `Summarize the following chat conversation:\n\n${transcript}`,
  });
  return (response.output_text ?? '').trim();
};
```

Sentiment tone analysis is also implemented as a separate service. In this case, the backend requires the model to return strictly valid JSON with a specific structure. This constitutes a significant design choice because the model's response will be consumed programmatically. The client can present the result in a consistent manner by using a fixed schema, while the backend can parse the result in a safe manner without relying on "free-form text." Additionally, the system asks the model to respond in the same language as the conversation, which is more user-friendly and natural-sounding.

```
const openai = require('../utils/openAi');
module.exports = async function emotions(transcript) {
  const response = await openai.responses.create({
    model: 'gpt-5-mini',
    instructions:
      'Analyze the emotional tone of a chat conversation. ' +
      'Return ONLY valid JSON with this exact structure:\n' +
      '{ "overall_mood": string, "emotions": string[], "explanation": string }' +
       'Respond in the same language as the conversation. If the conversation is primarily in Greek, respond in Greek. Otherwise,
respond in English.',
    input: `Conversation:\n\n${transcript}`,
  });
```

```
  const text = (response.output_text ?? '').trim();
  return JSON.parse(text);
};
```

The backend has two endpoints for this functionality. A strict check should be done before the AI model is called. The user requesting the function should be a member of the conversation. This is a necessary step; otherwise, there would be a risk of leaking the content of the chat using the AI model. For this, a helper function is created to load a conversation. It checks if the chat exists, assigns the username to senderId, and then checks for membership.

```
async function loadChatForUser(chatId, userId) {
  const chat = await Chat.findById(chatId).populate('messages.senderId', 'username');
  if (!chat) return { error: { status: 404, message: 'Chat not found' } };

  const isMember = (chat.members || []).some((id) => id.toString() === userId.toString());
  if (!isMember) return { error: { status: 403, message: 'Not a member of this chat' } };

  return { chat };
}
```

With this mechanism, the AI endpoints inherit the same security logic as the rest of the application's endpoints. After securely loading the chat, the backend does not send the entire history to the model. It does this by limiting the number of messages to a certain window by the `lastN` parameter, rather than the number of messages overall. This is a very important decision for three reasons: it limits the cost and the time taken, the size of the prompt, and the leak of old sensitive information in real-world applications.

```
const lastN = Number(req.query.lastN || 50);
const messages = chat.messages.slice(-Math.max(1, Math.min(lastN, 200)));
```

The messages are converted into a unified transcript via the helper function `buildTranscript`. The presence of this step demonstrates correct architecture: the database data is structured, but the AI model performs better when it receives a natural dialogue sequence. Thus, the backend generates the text representing the conversation and forwards it to the AI services.

```
const transcript = buildTranscript(messages);
const result = await summary(transcript);
```

The corresponding functionality for the emotions endpoint is similar, but it calls the second service.

```
const transcript = buildTranscript(messages);
const result = await emotions(transcript);
```

The controllers return standardized responses with `ok: true` or `ok: false`, allowing the frontend to easily handle failures and display appropriate messages to the user. This is especially helpful for AI functionalities, where errors may arise from the network, usage limits, or parsing issues.

```
res.json({ ok: true, result });
res.status(500).json({ ok: false, error: e.message });
```

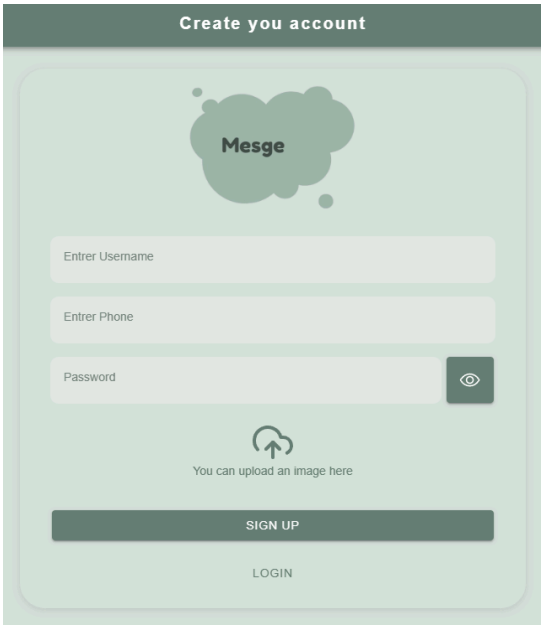
These are exposed at `/openai`, and they are protected by the same authentication middleware as other functionalities. This means that only logged-in users can request a summary or analysis.

```
router.post('/chats/:chatId/summary', protect, openAiControllers.getChatSummary);
router.post('/chats/:chatId/emotions', protect, openAiControllers.getChatEmotions);
app.use('/openai', openAiRoutes);
```

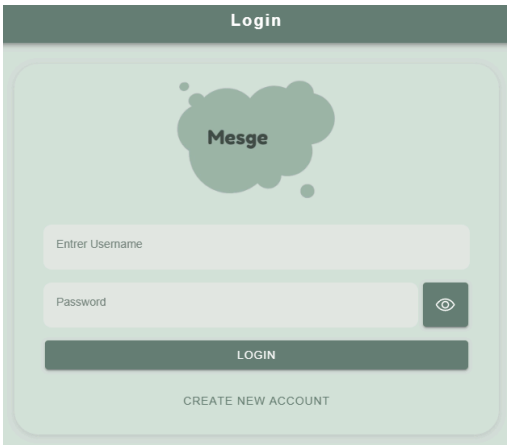
Chapter 6 - Results and Evaluation

6.1 Application Interface and User Experience

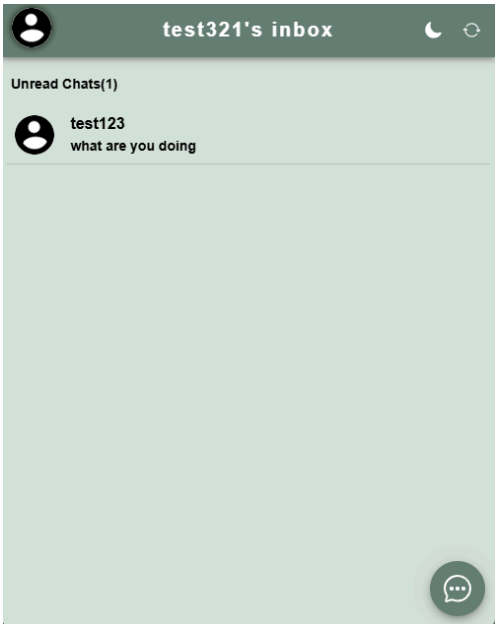
The application was designed in a way that maintains simplicity and clarity for the end-user. In the first step, the user can register through the following form, which requires them to provide a username, phone number, password, and optionally a profile picture.

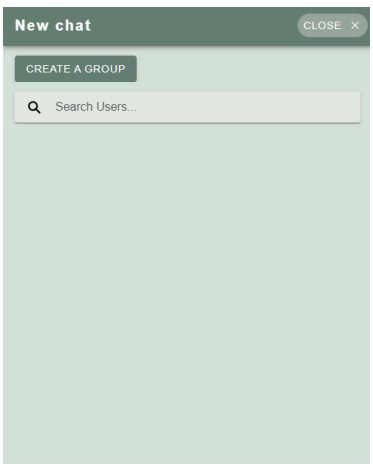


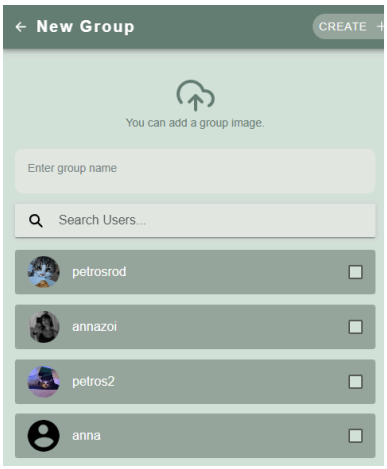
If they already have an account, they can log in using the form below.



Immediately afterward, the user has access to their conversations, can select private or group discussions, and send messages without delay.



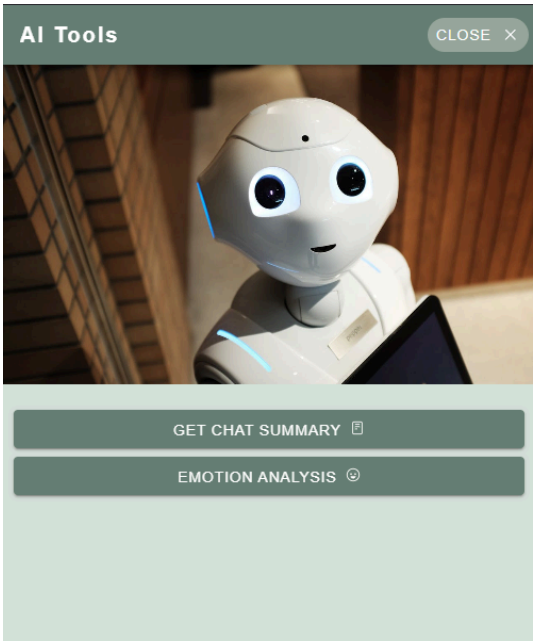




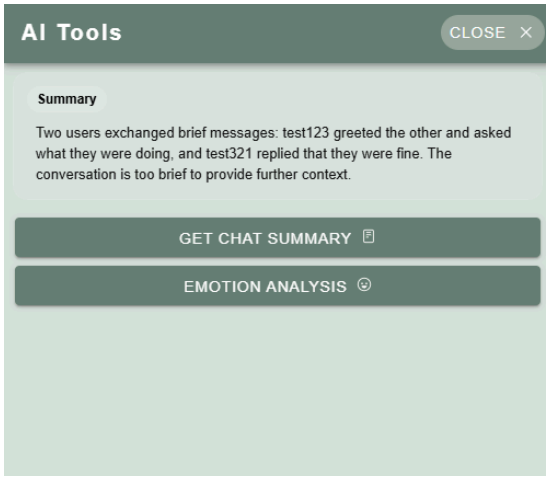
The layout of the chat is also simple and familiar, similar to other well-known applications, to make it easier to use without requiring special training.



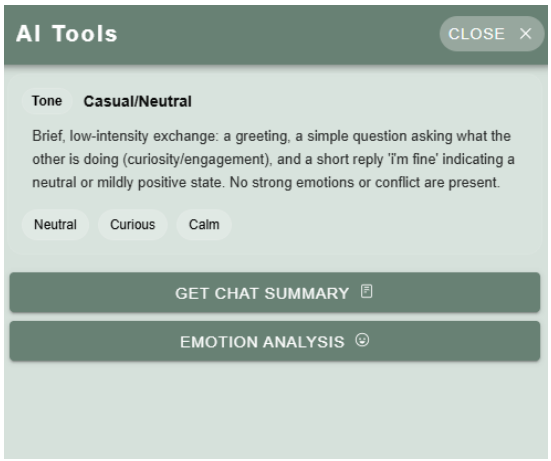
Furthermore, the integration of artificial intelligence functions is done discreetly in the lower left via a button, without interrupting the flow of messages.



In the event the user chooses to view the summary, they select the first button, which will display a message similar to the one below.



In the second case, they can be informed about the tone of the conversation, as shown below.



6.2 Functional Evaluation of the System

Correctness and reliability of the application's main features are assessed by functional evaluation. Testing and implementation of the messaging Application has confirmed that it can provide real-time exchange of messages between users and groups. The message is sent and received instantly with no data loss while the interface is dynamically updated. The system successfully controls login and logout operations and creates and manages conversations. All functionality around transferring files and images is part of the communication flow with no adverse effect on the overall performance of the Application. The Application allows users to make video or audio calls with the quality expected based on

standard network performance and confirms that real-time technologies work properly as integrated into the Application.

6.3 Usability and User Experience Evaluation

The usability of the application was assessed on the criteria of how clear the app interface is for users and the ease with which they can interact with the application through user-to-system interactions. The app structure also enables quick navigation between conversations and gives the user easy access to the core features of the app without having to go through multiple steps before reaching the final desired function. The artificial intelligence functionality is designed to operate in a way that supports and aids the user, rather than causing a disruption in the user's focus on completing the requested task. The user can only request a tone of the most recent conversation plus a summary if they choose to do so by exercising their ability to control the user-to-user communication experience. The overall experience of the user is that the application operates smoothly and consistently across both the web and mobile platforms, which enhances the usability of the application as a part of the daily life of the user.

6.4 Evaluation of Artificial Intelligence Functions

The artificial intelligence functionalities were evaluated in terms of their utility and practical value within the communication context. Sentiment analysis provides a general overview of the communicative tone, helping users perceive the conversation's atmosphere, especially in group settings. The summarization feature proved particularly useful for extensive conversations. The summaries produced are short and easy to understand, enabling the user to quickly comprehend the essential points without having to read the whole history. While the output relies on other AI services, the incorporation of these services into the system is done in such a way as to ensure the stability of the application.

Chapter 7 - Conclusions and Future ProspectsΑνάπτυξης

7.1 Conclusions

As a result of this project, a modern messaging app has been developed. The messaging app provides both real-time communication between users and uses AI to handle the fundamental functions of communication in addition to the use of intelligent tools to manage the conversation. The architecture of the system has been designed so that the frontend and backend are separate; the technologies used (WebRTC and Socket.IO) allow users to communicate with each other instantly. The AI functionalities have been integrated into the messaging app in such a way that they do not disrupt the communication process or cause instability in the messaging app.

7.2 Future Extensions

In the future, the system could be enriched with additional artificial intelligence capabilities, which may include topic analysis in conversations, automatic reply suggestions, or content monitoring mechanisms that detect aggressive or inappropriate material. Concurrently, upgrading security and adding more advanced data protection mechanisms would further enhance the application's reliability.

References

[1] V. K. Singh, P. Awasthi, L. Tripathi, S. Thakur, και P. Singh, “MERN (MongoDB, Express.js, React.js, Node.js) Stack Web-Based Streaming and Broadcasting Application,” in *Proceedings of the 2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT)*, Greater Noida, India, Aug. 2024. DOI: 10.1109/ICEECT61758.2024.10738869

[2] D. Gupta, N. Dhanda, και K. K. Gupta, “Enhanced Text Summarization System Using Machine Learning,” in *Proceedings of the 2025 3rd International Conference on Disruptive Technologies (ICDT)*, Greater Noida, India, Mar. 2025. DOI: 10.1109/ICDT63985.2025.10986688

[3] C. M. Novac, O. C. Novac, R. M. Sferle, M. I. Gordan, G. Bujdosó, και C. M. Dindelegan, “Comparative study of some applications made in the Vue.js and React.js frameworks,” in *Proceedings

of the 2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, Oradea, Romania, Jun. 2021. DOI: 10.1109/EMES52337.2021.9484149

[4] B. Wu, Y. Cao, και X. Liu, “Research and Analysis of Commonly Used Node.js Framework,” in *Proceedings of the 2024 Prognostics and System Health Management Conference (PHM)*, Stockholm, Sweden, May 2024. DOI: 10.1109/PHM61473.2024.00027

[5] B. Shamini P., K. S., N. Sivasankari, G. Chandran, A. Jency, και S. Sruthi, “People Interconnect – A Platform for Unified Communication,” in *Proceedings of the 2025 International Conference on Intelligent Computing and Control Systems (ICICCS)*, Erode, India, Mar. 2025. DOI: 10.1109/ICICCS65191.2025.10985250

[6] P. Murley, Z. Ma, J. Mason, M. Bailey, και A. Kharraz, “WebSocket Adoption and the Landscape of the Real-Time Web,” in *Proceedings of the Web Conference 2021 (WWW ’21)*, pp. 1192–1203, Jun. 2021. DOI: 10.1145/3442381.3450063

[7] S. Kumar, N. K. Kurmi, R. Kumar, S. Kumar, S. Mishra, και S. Tiwari, “Real Time Chat App,” *International Journal of Scientific Research & Engineering Trends*, vol. 11, no. 4, pp. xx–xx, Jul.–Aug. 2025.

[8] N. Shams-Vahdati, S. Shams Vahdati, και T. Samad-Soltani, “Design and evaluation of collaborative decision-making application for patient care in the emergency department,” *Health Science Reports*, vol. 7, no. 2, e1931, Feb. 2024. DOI: 10.1002/hsr2.1931

[9] N. Alimkhodjaeva, M. Minarova, E. Nabiev, και S. Aytieva, “Artificial Intelligence in Marketing Communication and Customer Satisfaction,” in *Proceedings of the 8th International Conference on Future Networks and Distributed Systems (ICFENDS ’24)*, pp. 165–174, Jul. 2025. DOI: 10.1145/3726122.3726147