

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу**

До захисту допущено:

Завідувач кафедри

_____ Оксана ТИМОЩУК

«__» _____ 2025 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системний аналіз і управління»

спеціальності 124 «Системний аналіз»

**на тему: «Месенджер із використанням технологій штучного інтелекту
для підсумування вмісту повідомлень»**

Виконав :

студент IV курсу, групи КА-13

Грабовецький Нікіта Богданович

Керівник:

доцент, к. т. н., Савченко Ілля Олександрович

Консультант з економічного розділу:

доцент, к. е. н., Рощина Надія Василівна

Консультант з нормконтролю:

асистент каф. ММСА Канцедал Георгій Олегович

Рецензент:

доцент, к.н., Булах Богдан Вікторович

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 124 «Системний аналіз»

Освітньо-професійна програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Оксана ТИМОЩУК

«___» _____ 2025 р.

ЗАВДАННЯ
на дипломну роботу студенту
Грабовецькому Нікіті Богдановичу

1. Тема роботи «Месенджер із використанням технологій штучного інтелекту для підсумування вмісту повідомлень», керівник роботи Савченко Ілля Олександрович доцент, к. т. н., затверджені наказом по університету від «___» _____ 20__ р. № _____
2. Термін подання студентом роботи _____
3. Вихідні дані до роботи – доступ до прикладного програмного інтерфейсу від OpenAI.
4. Зміст роботи – огляд сучасних інструментів для створення веб-застосунків, аналіз технологій інтеграції генеративного штучного інтелекту для обробки тексту, розробка месенджера з функцією підсумування повідомлень, тестування функціоналу та проведення функціонально-вартісного аналізу.
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) – презентація до захисту дипломної роботи складається з 17 слайдів і включає постановку задачі, стислий огляд використаних технологій та демонстрацію результатів розробленого програмного продукту.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В., доцент, к. е. н.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Формулювання тематики дослідження.	15.02.2025-01.03.2025	Виконано
2	Аналіз актуальності задачі стосовно тематики дослідження	02.03.2025 – 20.03.2025	Виконано
3	Аналіз відомих результатів стосовно тематики дослідження	22.03.2025 – 19.04.2025	Виконано
4	Формулювання теми дипломної роботи	14.04.2025 – 19.04.2025	Виконано
5	Ознайомлення технологій для реалізації програмної частини	21.04.2025 – 26.04.2025	Виконано
6	Розробка клієнтської та серверної частини застосунку	27.04.2025 – 10.05.2025	Виконано
7	Інтеграція штучного інтелекту для підсумування вмісту повідомлень	29.04.2025 – 11.05.2025	Виконано
8	Написання звіту з переддипломної практики	12.05.2025 – 17.05.2025	Виконано
9	Кінцеве оформлення записок та розділів дипломної роботи	18.05.2025 – 03.06.2025	Виконано
10	Попередній захист дипломної роботи	05.06.2025 – 06.06.2025	Виконано
11	Захист дипломної роботи	16.06.2025 – 20.06.2025	Виконано

Студент

Нікіта ГРАБОВЕЦЬКИЙ

Керівник

Ілля САВЧЕНКО

РЕФЕРАТ

Дипломна робота: 116 ст., 9 таблиць, 13 рис., 15 джерел, 2 дод.

ВЕБ-ЗАСТОСУНОК, МЕССЕНДЖЕР, ШТУЧНИЙ ІНТЕЛЕКТ, ПІДСУМУВАННЯ ТЕКСТ, КЛІЄНТСЬКА ЧАСТИНА, СЕРВЕРНА ЧАСТИНА.

Об'єкт дослідження – процеси розробки та функціонування веб-месенджера з можливістю інтелектуального аналізу вмісту повідомлень.

Постановка задачі – розробити веб-застосунок, який дозволяє користувачам обмінюватися повідомленнями та файлами, з можливістю редагування й видалення власних повідомлень. Інтегрувати штучний інтелект для підсумування змісту окремих повідомлень. Реалізувати систему авторизації та автентифікації для безпечного доступу користувачів до функціоналу застосунку.

Мета роботи – розробка веб-застосунку для обміну повідомленнями з інтеграцією моделі штучного інтелекту GPT-4o для підсумування текстового вмісту повідомлень з метою підвищення ефективності спілкування та зручності користування.

Предмет дослідження – методи та засоби інтеграції штучного інтелекту у веб-застосунки для автоматизованої обробки й узагальнення текстових повідомлень.

Практична значимість проєкту полягає у створенні месенджера з функцією підсумування повідомлень за допомогою штучного інтелекту. Це дозволяє підвищити ефективність цифрового спілкування, зменшити інформаційне навантаження та заощадити час користувачів. Проєкт демонструє приклад практичного застосування великих мовних моделей у повсякденному комунікаційному середовищі, що сприяє популяризації та подальшому розвитку технологій ШІ.

ABSTRACT

Thesis: 116 p., 9 tabl., 13 fig., 15 sources, 2 appendices.

WEB APPLICATION, MESSENGER, ARTIFICIAL INTELLIGENCE,
TEXT SUMMARIZATION, CLIENT SIDE, SERVER SIDE.

The object of research – the processes of development and operation of a web messenger with the ability to intelligently analyze the content of messages.

Task statement – to develop a web application that allows users to exchange messages and files, with the ability to edit and delete their own messages. Integrate artificial intelligence to summarize the content of individual messages. Implement an authorization and authentication system for secure user access to the application's functionality.

The goal of the project is to develop a web application for messaging with the integration of the GPT-4o artificial intelligence model to summarize the text content of messages in order to increase communication efficiency and ease of use.

The subject of the research – methods and tools for integrating artificial intelligence into web applications for automated processing and summarizing text messages.

The practical significance of the project is the creation of a messenger with the function of summarizing messages using artificial intelligence. This allows to increase the efficiency of digital communication, reduce the information load, and save users' time. The project demonstrates an example of the practical application of large language models in the everyday communication environment, which contributes to the popularization and further development of AI technologies.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА ТЕРМІНІВ	8
ВСТУП	9
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ	10
1.1 Визначення месенджер та його історія розвитку	10
1.2 Класифікація месенджерів за сферою застосування	11
1.3 Огляд сучасних месенджерів	13
1.4 Формалізація задачі	15
1.5 Висновки до першого розділу	17
РОЗРОБКА МЕСЕНДЖЕРА ТА ОГЛЯД ТЕХНОЛОГІЙ ШІ ДЛЯ ПІДСУМУВАННЯ ТЕКСТУ	18
2.1 Методи та технології для побудови веб-застосунку	18
2.1.1 Огляд розробки клієнтської частини веб-застосунку	19
2.1.2 Огляд розробки серверної частини веб-застосунку	22
2.1.3 Огляд проектування та реалізації БД	25
2.2 Вибір технологій розробки та середовища	33
2.3 Вибір моделі ШІ для підсумування тексту	35
2.4 Проектування функціоналу	36
2.5 Висновки до другого розділу	37
ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ	39
3.1 Набір технологій	39
3.2 Архітектура програмного продукту	40
3.3 Інтерфейс веб-застосунку	43
3.4 Порівняння результатів	47
3.5 Висновки до третього розділу	49
РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	50
4.1 Постановка завдання	50
4.2 Обґрунтування функцій програмного продукту	50
4.3 Обґрунтування системи параметрів програмного продукту	56
4.4. Аналіз експертного оцінювання параметрів	58
4.5 Аналіз рівня якості обраного варіанту	62

4.7 Вибір найкращої реалізації техніко-економічного рівня	67
4.8 Висновки до четвертого розділу	68
ВИСНОВКИ.....	69
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ	70
ДОДАТОК А ЛІСТИНГ ПРОГРАМИ	72
ДОДАТОК Б ГРАФІЧНІ МАТЕРІАЛИ	108

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА ТЕРМІНІВ

БД – бази даних;

ШІ – штучний інтелект

HTML – hyper text markup language

CSS – cascading style sheets

API – application programming interface

REST – representational state transfer

SQL – structured query language

URL – uniform resource locator

CRUD – create read update delete

GPT – generative pre-training transformer

AI – artificial intelligence

ВСТУП

Сьогодні, в умовах стрімкого розвитку цифрових технологій, обсяг текстової інформації, що передається через месенджери, невідомо зростає. Регулярне спілкування в особистому та професійному середовищі сприяє накопиченню значного обсягу повідомлень, що, своєю чергою, призводить до інформаційного перевантаження та створює потребу в інструментах для ефективної обробки змісту.

У межах дипломного проєкту передбачається розробка месенджера з вбудованою функцією підсумування повідомлень на основі технологій штучного інтелекту. Такий підхід має на меті не лише забезпечити зручну комунікацію, а й надати користувачам інноваційний інструмент для швидкого ознайомлення зі змістом довгих текстів. Це дозволить економити час, знижувати когнітивне навантаження та підвищувати загальну ефективність цифрової взаємодії.

Подібна функціональність є особливо актуальною в професійному середовищі, де щоденна обробка великих обсягів інформації є ключовим елементом продуктивної діяльності. Саме тому обрана тема має не лише практичне значення, а й значний потенціал для подальшого розвитку в галузі штучного інтелекту та цифрових комунікацій.

Інтеграція ШІ в повсякденні засоби спілкування відповідає сучасним тенденціям цифрової трансформації та відкриває нові можливості для створення інтелектуальних систем, які не лише передають інформацію, а й допомагають користувачам ефективніше її осмислювати. Реалізація подібного функціоналу відкриває перспективи для появи інноваційних рішень, що здатні змінити підхід до цифрового спілкування, зробивши його більш зручним і продуктивним.

ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Визначення месенджер та його історія розвитку

Месенджер – телекомунікаційна служба для обміну текстовими повідомленнями між комп'ютерами або іншими пристроями користувачів через комп'ютерні мережі [1].

Історія розвитку месенджерів бере початок ще з кінця 1960-х років, коли в рамках проєкту ARPANET, попередника сучасного Інтернету, було здійснено перші експерименти з передачею текстових повідомлень між віддаленими комп'ютерами. Одним із ключових кроків у цьому напрямі стало створення протоколів обміну даними, які дозволяли реалізувати функціональність, подібну до сучасних електронних листів, хоча й у дуже обмеженому вигляді. Ці перші спроби дали поштовх до формування концепції інтерактивного цифрового спілкування в реальному часі.

У 1980-х роках розвиток локальних обчислювальних мереж у компаніях і університетах сприяв появі систем миттєвого обміну повідомленнями, які дозволяли співробітникам або студентам спілкуватися між собою всередині однієї мережі. Визначним прикладом є IRC (Internet Relay Chat), створений у 1988 році. Цей протокол став першою широкодоступною інтернет-технологією для живого спілкування, переважно в групових чатах, що називаються «каналами», хоча також підтримував особисті повідомлення, передачу файлів та виконання різних клієнтських і серверних команд [2].

У 1990-х роках, із розповсюдженням персональних комп'ютерів і підключенням до Інтернету все більшої кількості користувачів, з'явилися перші масові графічні клієнти для миттєвого обміну повідомленнями.

Серед найбільш популярних месенджерів 1990-х років варто згадати ICQ, створений у 1996 році ізраїльською компанією Mirabilis. Цей сервіс вирізнявся впровадженням унікальної системи ідентифікації користувачів (UIN) та низкою функцій, які згодом стали стандартом для галузі: список контактів, статуси онлайн/офлайн, можливість залишати повідомлення, коли користувач не в мережі. У США широкого поширення набув AOL Instant Messenger (AIM), тісно інтегрований з екосистемою America Online, що забезпечувало зручність використання для мільйонів абонентів цього провайдера. Ще одним важливим представником у сфері миттєвого обміну повідомленнями став MSN Messenger (пізніше – Windows Live Messenger) від компанії Microsoft, який дозволяв користувачам Windows спілкуватися в режимі реального часу, а згодом – здійснювати голосові дзвінки та використовувати додаткові інтерактивні функції.

Далі, з появою смартфонів і мобільного Інтернету, почалася нова ера месенджерів. Новітні платформи стали не лише засобами комунікації, а й частиною цифрового стилю життя. Вони еволюціонували в багатофункціональні комунікаційні середовища, які підтримують канали новин, боти, оплати, голосові чати та інше.

1.2 Класифікація месенджерів за сферою застосування

Месенджери є важливими інструментами цифрової комунікації, які активно застосовуються в найрізноманітніших сферах діяльності. Залежно від призначення, цільової аудиторії та функціональних можливостей, вони можуть виконувати різні ролі – від засобу особистого спілкування до інструменту командної взаємодії або вузькоспеціалізованого сервісу.

Класифікація месенджерів за сферою використання дозволяє краще зрозуміти специфіку їхнього застосування в різних контекстах, а також виявити відмінності у вимогах до функціональності, безпеки та інтеграції.

1. Персональні месенджери. Це найбільш поширений тип месенджерів, орієнтований на повсякденне спілкування між друзями, родиною та знайомими. Такі додатки мають інтуїтивно зрозумілий інтерфейс, підтримують текстові та голосові повідомлення, обмін файлами, емодзі, стікери, відеодзвінки.
2. Корпоративні месенджери. Розроблені для внутрішньої комунікації в організаціях. Вони підтримують інтеграцію з іншими корпоративними системами, мають функції контролю доступу, адміністрування користувачів, збереження історії, захисту даних тощо. Такі платформи сприяють ефективній взаємодії в командах і підвищують продуктивність праці.
3. Спеціалізовані месенджери. Ці месенджери призначені для конкретних галузей або цільових аудиторій.
4. Соціальні платформи з функціями месенджера. Деякі соціальні мережі мають вбудовані месенджери, які виконують роль допоміжного засобу спілкування. Наприклад, Instagram Direct, LinkedIn Messaging або месенджер у TikTok. Хоч вони й не є класичними месенджерами, їх активно використовують у повсякденній комунікації.

Загалом, різноманіття месенджерів зумовлено широким спектром потреб користувачів у різних сферах життя. Залежно від призначення, функціональності та контексту використання, месенджери можуть бути як універсальними засобами спілкування, так і спеціалізованими платформами, орієнтованими на вузьконаправленні завдання.

1.3 Огляд сучасних месенджерів

У сучасному цифровому світі месенджери стали невід’ємною частиною повсякденного життя. Кожен із популярних месенджерів має власну специфіку, набір функцій, цільову аудиторію, а також відрізняється ступенем інтеграції новітніх технологій, зокрема штучного інтелекту.

Загальна характеристика сучасних месенджерів наведена в таблиці 1.1.

Таблиця 1.1 – Загальна характеристика сучасних месенджерів

Месенджер	Основна аудиторія	Ключові функції	Особливості
Telegram	Універсальна, техно-орієнтована	Боти, канали, чати, групи до 200 000 учасників, обмін файлами до 2 ГБ	Відкрите API, гнучка система ботів, секретні чати
WhatsApp	Масова аудиторія	Голосові/відеодзвінки, текст, документи, статуси	Простота, наскрізне шифрування
Facebook Messenger	Користувачі Facebook	Інтеграція з Facebook, відеочати, ігри, платежі	Частина соцмережі, залежність від Facebook
Signal	Користувачі з фокусом на приватність	Наскрізне шифрування, мінімалізм, дзвінки, файли	Максимальна безпека, відкритий код
Viber	Центральна та Східна Європа	Чати, дзвінки, канали, стікери, Viber Out	Популярний в Україні, інтеграція з банками
Discord	Геймери, спільноти	Сервери, голосові/відеоканали, боти, стрімінг	Активні спільноти, підтримка ролей
Slack	Бізнес, команди	Канали, інтеграції, повідомлення, нагадування	Робочий простір, інтеграція з Jira, GitHub

Порівняння функціональності та ключових характеристик провідних месенджерів.

1. Безпека та приватність. Signal і Telegram пропонують найвищий рівень захисту. Signal застосовує наскрізне шифрування за замовчуванням для всіх повідомлень, а Telegram – лише в «секретних чатах». WhatsApp також використовує наскрізне шифрування, проте належить корпорації Meta, що викликає занепокоєння у деяких користувачів.
2. Гнучкість і розширюваність. Telegram і Discord мають потужні API та підтримують створення ботів, інструментів автоматизації, інтеграцій. Slack є лідером серед бізнес-месенджерів завдяки інтеграції з системами управління завданнями та CRM.
3. Підтримка спільнот. Discord і Telegram пропонують потужні інструменти для адміністрування великих груп та каналів. Discord дозволяє створювати сервери з окремими каналами, ролями, правами доступу – ідеально підходить для спільнот, курсів і ігрових гільдій.
4. Якість відео та голосового зв'язку. WhatsApp, Messenger та Discord забезпечують стабільні відеодзвінки, але Discord вирізняється можливістю високоякісного аудіо навіть при великій кількості учасників.
5. Інтеграція штучного інтелекту. Останнім часом месенджери почали інтегрувати технології штучного інтелекту, що дозволяє реалізовувати такі функції, як автоматичне підсумування повідомлень, переклад, фільтрація спаму, генерація відповідей, а також персоналізовані рекомендації. Такі інновації змінюють уявлення про комунікацію, відкриваючи нові можливості для підвищення ефективності взаємодії користувачів. Telegram активно підтримує ботів з інтеграцією GPT-моделей, що дозволяє реалізувати ШІ-функції безпосередньо у чатах. Slack інтегрує

OpenAI GPT для генерації відповідей та підсумування обговорень у корпоративному середовищі. WhatsApp також тестує вбудовану підтримку ШІ через Meta AI. Discord вже має інтегрованого бота Clyde на основі ChatGPT. Такі можливості стають стандартом для нової генерації месенджерів.

1.4 Формалізація задачі

У цій роботі розглядається задача автоматичного формування стислого змісту вхідного повідомлення у месенджері. Метою є створення веб-застосунку, який здатний аналізувати текст повідомлення, виокремлювати ключову інформацію та формулювати стислий зміст у формі речення або короткого абзацу. Такий зміст повинен передавати основну ідею без повторів і другорядних деталей.

Для реалізації цієї функціональності планується використання сучасних мовних моделей штучного інтелекту. Такі моделі навчені на великих обсягах текстових даних і здатні ефективно обробляти природну мову, виявляючи суть повідомлення, приховані смислові зв'язки та логічну структуру. Вони повинні забезпечувати високу якість генерації підсумків, навіть для складних текстів із неформальною мовою, розмовною лексикою або емодзі.

Особливістю поставленої задачі є необхідність роботи системи в реальному часі. Це означає, що кожне нове повідомлення користувача має оброблятися миттєво, тобто система повинна протягом декількох секунд повертати узагальнений варіант тексту, не порушуючи швидкість комунікації. Крім того, месенджер повинен надавати можливість переглядати як оригінальний текст повідомлення, так і його коротку версію.

Крім основної функції автоматичного підсумування повідомлень,

месенджер повинен надавати базові, але критично важливі можливості для повноцінного користувацького досвіду. До таких можливостей належать функції редагування та видалення повідомлень, а також можливість спілкування з іншими користувачами в межах платформи.

Редагування повідомлень є важливою складовою сучасних комунікаційних сервісів. Користувачі повинні мати змогу виправляти помилки, змінювати формулювання або уточнювати зміст вже надісланого повідомлення. Це дозволяє підтримувати чіткість і коректність спілкування, особливо в разі, коли інформація, що надсилається, є діловою, технічною або містить інструкції.

Так само важливою є функція видалення повідомлень. Вона дозволяє користувачеві повністю прибрати повідомлення з історії чату. Це особливо актуально у випадках, коли повідомлення було надіслане помилково, містило чутливу інформацію або стало недоречним.

Окрім роботи з окремими повідомленнями, месенджер має надавати повноцінний механізм взаємодії між користувачами. Це передбачає можливість обирати іншого користувача з доступного списку. Інтерфейс має бути організований таким чином, щоб користувач міг легко перемикатися між чатами, переглядати історію спілкування, а також мати змогу швидко надіслати нове повідомлення іншому учаснику. Забезпечення такої функціональності є базовою вимогою для будь-якої сучасної комунікаційної платформи.

Формалізована задача проекту полягає у розробці веб-застосунку месенджера, що здатен підсумовувати вхідні повідомлення користувачів за допомогою технологій штучного інтелекту. Такий функціонал має значно підвищити ефективність комунікації, зробити спілкування більш інформативним, а також зменшити когнітивне навантаження на користувача під час читання довгих чи складних текстів.

1.5 Висновки до першого розділу

Отже, у першому розділі було здійснено всебічний аналіз поняття месенджера, його історичного розвитку, різновидів та сучасних функціональних можливостей. Розглянуто, як месенджери еволюціонували від експериментів з передачею текстових повідомлень між віддаленими комп'ютерами до багатофункціональних платформ, що об'єднують обмін повідомленнями, медіа, документами та навіть інтеграцію зі сторонніми сервісами.

Проведено класифікацію месенджерів за сферою їх застосування, що дозволило виділити такі категорії, як корпоративні, особисті, спеціалізовані та персональні. Такий підхід допомагає краще зрозуміти цільову аудиторію, основні вимоги та технічні особливості різних типів комунікаційних сервісів.

В рамках огляду сучасних месенджерів було проаналізовано ключові гравці ринку, їх функціонал та новітні технології, які вони впроваджують, зокрема – інтеграцію елементів штучного інтелекту. Особливу увагу приділено функціям, що покращують взаємодію з текстовою інформацією, таким як переклад, фільтрація спаму та генерація відповідей.

Окремо було сформульовано задачу, яку вирішує ця робота: створення веб-застосунку месенджера, здатного підсумовувати вміст повідомлень. Додатково визначено функціональні вимоги до месенджера: підтримка редагування та видалення повідомлень, можливість комунікації між користувачами, а також збереження швидкодії системи під час обробки.

Таким чином, у першому розділі сформовано теоретичні та концептуальні засади для розробки інтелектуального месенджера, який поєднує базовий функціонал текстової комунікації з інструментами штучного інтелекту з метою підвищення ефективності обміну інформацією.

РОЗРОБКА МЕСЕНДЖЕРА ТА ОГЛЯД ТЕХНОЛОГІЙ ШІ ДЛЯ ПІДСУМУВАННЯ ТЕКСТУ

2.1 Методи та технології для побудови веб-застосунку

На перший погляд, сучасні месенджери – це прості, зручні у використанні сервіси, які дозволяють користувачам взаємодіяти між собою, надсилати повідомлення або обмінюватися файлами. Проте за цією зручністю стоїть складна архітектура, що поєднує низку технологій, інструментів і програмних рішень. Звичайний користувач, який взаємодіє з веб-застосунком через інтерфейс, рідко замислюється над тим, як саме працює система, яка обробляє його запити, зберігає повідомлення, формує відповіді чи забезпечує безперебійну роботу сервісу.

У цьому розділі буде розглянуто основні методи та технології, які використовуються для створення повноцінного веб-застосунку, а саме месенджера з функцією підсумування повідомлень. Умовно, усі ці технології можна поділити на дві основні частини: Frontend тобто те, що бачить і з чим взаємодіє користувач, та Backend тобто внутрішня логіка, база даних, алгоритми обробки запитів та модулі штучного інтелекту. Обидві частини тісно пов'язані між собою, і лише завдяки їх злагодженій роботі веб-застосунок функціонує належним чином.

Далі будуть розглянуті конкретні інструменти, мови програмування, бібліотеки та фреймворки, які використовуються для реалізації інтерфейсу користувача, серверної логіки, зберігання даних, а також підключення моделей штучного інтелекту для аналізу та підсумування текстових повідомлень.

2.1.1 Огляд розробки клієнтської частини веб-застосунку

Клієнтська частина веб-застосунку, або frontend – це сукупність програмних компонентів, які відповідають за відображення інтерфейсу користувача та забезпечення його взаємодії з функціоналом системи. Саме через неї користувач бачить структуру сторінки, заповнює форми, натискає кнопки та отримує відповіді на свої дії. У сучасних веб-застосунках клієнтська частина не обмежується статичним HTML, а є динамічною, інтерактивною і тісно пов'язаною з серверною логікою.

HTML (HyperText Markup Language) – це мова розмітки, яка слугує для побудови структури веб-сторінок. Вона складається з тегів, що задають типи елементів і визначають їхнє місце розташування на сторінці. За допомогою HTML можна описувати як логічну структуру документа, так і елементи його зовнішнього вигляду. Кожен елемент HTML має свої властивості та може включати текст, зображення, гіперпосилання й інші компоненти. Комбінуючи ці елементи, можна створювати різноманітні та привабливі веб-сторінки [3].

CSS (Cascading Style Sheets) – це мова стилів, призначена для оформлення зовнішнього вигляду веб-сторінок, створених за допомогою HTML. Вона задає, як саме повинні виглядати елементи на сторінці: їхній колір, шрифт, розміри, розташування, межі та інші стилістичні характеристики. Завдяки CSS можна відокремити візуальне оформлення сайту від його структури та змісту. Це дозволяє задавати стилі один раз і застосовувати їх до всіх елементів з подібними властивостями, замість того щоб формувати кожен елемент окремо [4].

Однак лише HTML та CSS недостатньо для створення повноцінного функціонального веб-застосунку, оскільки вони не забезпечують динамічну взаємодію з користувачем та обробку даних. Для створення веб-застосунку використовується JavaScript – об'єктно-прототипна мова програмування, яка

широко застосовується у сфері веб розробки. Вона дає змогу додавати динамічний і інтерактивний функціонал до веб сторінок, реалізовувати взаємодію з користувачем та розширювати можливості веб інтерфейсу [5].

Для стилізації інтерфейсу веб-застосунків використовуються CSS-фреймворки та утилітарні бібліотеки, які значно спрощують процес створення сучасного, адаптивного та привабливого дизайну. Вони дозволяють розробникам уникнути трудомісткого ручного написання стилів, використовуючи готові рішення, що відповідають стандартам веб-дизайну, забезпечують кросбраузерність і адаптивність до різних типів пристроїв.

Одним із найпоширеніших CSS-фреймворків є Bootstrap. Це комплексне рішення, яке містить готові компоненти (кнопки, форми, навігаційні панелі, модальні вікна тощо), систему сіток для побудови адаптивного макета, а також набір допоміжних класів для вирівнювання, відступів і стилізації тексту. Bootstrap дозволяє створювати повноцінний інтерфейс за короткий час із мінімальними зусиллями.

Іншим популярним інструментом є Tailwind CSS – утилітарна CSS-бібліотека, яка базується на іншій концепції. Замість готових компонентів вона надає набір коротких класів, які відповідають за конкретні стилі (відступи, кольори, шрифти, тіні, позиціонування тощо). Це дозволяє розробникам будувати унікальний дизайн без написання власних CSS-стилів, працюючи безпосередньо в HTML-розмітці. Tailwind CSS забезпечує високу гнучкість та зручність у стилізації, особливо при використанні компонентного підходу у фреймворках на кшталт React або Vue.

Застосування таких інструментів дозволяє значно зменшити час на розробку інтерфейсу, підтримувати чисту структуру коду, покращити вигляд застосунку на всіх пристроях та забезпечити єдину стилістичну логіку протягом усього проєкту.

З метою оптимізації процесу розробки часто застосовуються фреймворки та бібліотеки, які спрощують написання коду, структурування

проєкту та повторне використання компонентів. Найпопулярнішими серед них є : React, Vue.js, Angular.

1. React – бібліотека від компанії Meta, що дозволяє будувати інтерфейс як набір незалежних компонентів і ефективно оновлювати DOM завдяки концепції віртуального DOM.
2. Vue.js – прогресивний фреймворк, орієнтований на поступове впровадження та зручну інтеграцію.
3. Angular – комплексний фреймворк від Google з вбудованими рішеннями для маршрутизації, форм, валідації тощо.

Нижче наведена таблиця 2.1 з порівнянням цих основних JavaScript-фреймворків/бібліотек

Таблиця 2.1 – Порівняння основних JavaScript-фреймворків/бібліотек

Критерій	React	Vue.js	Angular
Тип	Бібліотека для побудови інтерфейсу	Прогресивний фреймворк	Повноцінний фреймворк
Популярність	Висока, велика спільнота, багато ресурсів	Нижча, але зростає, активна спільнота	Висока, особливо у великих компаніях
Структура	Гнучкий, але без чіткої структури	Простий у структурі	Чітка структура з жорсткою архітектурою
Простота навчання	Середня, потребує знань JavaScript та JSX	Легкий для початківців	Важкий у вивченні, потребує TypeScript
Синтаксис	JSX (поєднання JS і HTML)	Звичний HTML + JavaScript	TypeScript, шаблони HTML
Продуктивність	Висока, завдяки віртуальному DOM	Дуже швидкий рендеринг	Висока, але об'ємний код

Продовження таблиці 2.1

Гнучкість	Висока, легко інтегрується з іншими бібліотеками	Висока	Менш гнучкий через строго визначену структуру
Сфера застосування	SPA, динамічні UI, середні та великі проекти	Невеликі та середні проекти, легкий старт	Великі корпоративні додатки
Недоліки	Немає вбудованої структури, потрібно обирати бібліотеки вручну	Мала екосистема, не така популярна серед великих компаній	Складність, велика крива навчання, багато коду

2.1.2 Огляд розробки серверної частини веб-застосунку

Бекенд – це сукупність технологій і програмного коду, що відповідають за внутрішню логіку та функціонування веб-сайту чи застосунку, залишаючись прихованими від користувача. Основними складовими бекенду є сервер, база даних і програмна логіка. Відповідно, завдання Backend-розробника полягає в розробці та обслуговуванні технічної інфраструктури, яка підтримує функціонування фронтенду – тієї частини застосунку, з якою безпосередньо взаємодіє користувач [6].

Серверна частина забезпечує реалізацію широкого спектра функцій, зокрема:

- 1) обробку HTTP-запитів від клієнта;
- 2) автентифікацію та авторизацію користувачів;
- 3) валідацію вхідних даних;

- 4) роботу з базами даних (створення, читання, оновлення, видалення даних);
- 5) підключення зовнішніх API або бібліотек (наприклад, для підсумування повідомлень за допомогою моделей штучного інтелекту);
- 6) ведення журналів (логування), моніторинг та обробку помилок;
- 7) забезпечення безпеки, зокрема захисту від несанкціонованого доступу або атак.

У типовому веб-застосунку клієнт (браузер або мобільний додаток) надсилає запити до сервера, наприклад, при надсиланні повідомлення, отриманні списку чатів, авторизації користувача чи зверненні до функції підсумування повідомлення. Сервер обробляє цей запит, за потреби взаємодіє з базою даних або зовнішніми сервісами, формує відповідь і надсилає її назад на клієнт. Усе це виконується безпосередньо серверною частиною застосунку.

Архітектура серверної частини веб-застосунку найчастіше реалізується у вигляді RESTful API (Representational State Transfer) або GraphQL API, які визначають спосіб взаємодії між клієнтською та серверною частинами через набір чітко визначених маршрутів (endpoint'ів).

REST – це архітектурний стиль побудови веб-сервісів, що базується на принципах стандартного HTTP-протоколу. У REST API кожен ресурс, такий як користувач, повідомлення чи чат, має унікальну адресу (URL), а взаємодія з цими ресурсами здійснюється за допомогою стандартних HTTP-методів: GET використовується для отримання даних (наприклад, списку повідомлень), POST – для створення нового ресурсу (наприклад, нового повідомлення), PUT або PATCH – для оновлення вже існуючого ресурсу (наприклад, редагування повідомлення), а DELETE – для його видалення. REST API є простим у реалізації, добре читається, легко тестується та інтегрується в інші системи. Основна його перевага полягає в чіткому розділенні клієнтської та серверної логіки, можливості кешування

відповідей, масштабуванні архітектури та використанні зрозумілого формату обміну даними – JSON.

GraphQL – це альтернативний до REST підхід до створення API, розроблений компанією Facebook, який надає клієнту можливість самостійно визначати, які саме дані йому потрібні. На відміну від REST, де для кожного ресурсу створюються окремі маршрути, у GraphQL усі запити обробляються через один універсальний маршрут на сервері, що дозволяє централізовано керувати всіма операціями з даними. Такий підхід дозволяє запитувати лише необхідні поля для конкретного сценарію, що значно знижує обсяг переданої інформації та кількість запитів до сервера. Наприклад, замість кількох REST-запитів для отримання користувача та його повідомлень, у GraphQL достатньо сформулювати один запит із відповідною структурою. Система також підтримує об'єднання кількох ресурсів в одному запиті, автоматичну генерацію документації та чітко визначену схему типів даних. Водночас, через більшу складність реалізації, потребу у налаштуванні схеми, резолверів та контролі безпеки, GraphQL частіше застосовується у масштабних або складноструктурованих застосунках із великою кількістю пов'язаних даних.

У сучасній розробці серверна частина може бути реалізована різними мовами програмування.

1. JavaScript (Node.js) – одна з найпопулярніших платформ, яка дозволяє створювати швидкі асинхронні сервери, підтримує тисячі бібліотек і фреймворків, зокрема Express, NestJS.
2. Python – використовується переважно для проєктів, що включають складну логіку обробки даних або машинне навчання; популярні фреймворки: Django, FastAPI.
3. Java – мова для побудови надійних, масштабованих систем; найвідоміший фреймворк – Spring Boot [7].
4. PHP – традиційна технологія для серверної розробки з фреймворками Laravel, Symfony [8].

5. Ruby – застосовується для швидкої розробки з використанням Ruby on Rails.
6. Go (Golang) – ефективна мова для створення високонавантажених сервісів.

Вибір мови програмування та фреймворку залежить від складності проєкту, навантаження, кількості одночасних користувачів, а також специфіки обробки даних. Наприклад, для створення легких і швидких RESTful API часто використовуються JavaScript-фреймворки на базі Node.js, тоді як для побудови складних монолітних систем перевагу надають Java або Python.

Слід окремо наголосити на важливості асинхронної обробки запитів у сучасних веб-застосунках. Асинхронне програмування є особливо корисним у випадках, коли система звертається до зовнішніх джерел даних, виконує операції введення/виведення (I/O) або обробляє велику кількість паралельних процесів. Такий підхід дозволяє уникнути блокування потоку виконання та забезпечити швидку реакцію застосунку.

Нижче наведено приклади ситуацій, у яких доцільно застосовувати асинхронність:

- 1) запити до API та отримання даних з мережі [9];
- 2) читання або запис великих файлів [9];
- 3) обробка паралельних завдань у мікросервісній архітектурі [9];
- 4) робота з базами даних [9];
- 5) таймери, фонове опрацювання та черги завдань [9].

2.1.3 Огляд проектування та реалізації БД

Бази даних відіграють ключову роль у зберіганні, організації та обробці інформації у веб-застосунках. Вони дозволяють забезпечити надійне та

структуроване збереження даних, а також швидкий доступ до них за допомогою запитів. У контексті серверної частини застосунку база даних є основним інструментом для зберігання інформації про користувачів, повідомлення, чати, журнали подій тощо. Бази даних поділяються на реляційні та нереляційні БД.

Реляційна база даних (РБД) – це набір даних, організованих у вигляді таблиць. Кожна таблиця має іменовані стовпці та рядки, що містять конкретні значення. Взаємозв'язки між таблицями визначаються за допомогою ключів. Доступ до таблиць здійснюється за допомогою мови структурованих запитів (SQL) [10].

Нижче наведено що включає в себе реляційна база даних, і які елементи необхідні для її роботи .

1. Реляційні таблиці. Данні організовуються у вигляді таблиць із рядками та стовпцями. Кожна таблиця має ім'я та структуру, визначену схемою даних [11].
2. Рядки (кортежі). Кожен рядок у реляційній таблиці представляє окремий запис і містить інформацію про об'єкт. Кожен запис є унікальним та ідентифікується за допомогою ключа [11].
3. Стовпці (атрибути). Стовпці таблиці являють собою атрибути або характеристики об'єктів, що описуються в таблиці. Кожен стовпець має ім'я і тип даних, який визначає, якого роду інформацію можна зберігати в цьому стовпчику (наприклад, текст, числа, дати і так далі) [11].
4. Ключі. У реляційній моделі використовуються ключі для ідентифікації рядків у таблиці. Основний ключ (Primary Key) забезпечує унікальність кожного рядка, а зовнішній ключ (Foreign Key) створює зв'язки між таблицями [11].
5. Зв'язки. Реляційні бази даних дають змогу встановлювати зв'язки між таблицями, що дає змогу об'єднувати дані з різних таблиць для виконання складних запитів [11].

6. SQL (Structured Query Language). SQL використовується для виконання операцій з даними в реляційних базах даних. Він дає змогу створювати, змінювати, видаляти та витягувати дані з таблиць, а також визначати правила для цілісності даних [11].
7. Нормалізація. Реляційні бази даних використовують процес нормалізації для зменшення надмірності даних і забезпечення їхньої цілісності [11].
8. Транзакції. РБД підтримують транзакції, що забезпечує атомарність, узгодженість, ізолюваність і довговічність даних (ACID-властивості) під час одночасного доступу кількох користувачів до бази даних [11].

У сфері розробки веб-застосунків найбільш широко використовуються реляційні системи управління базами даних (СУБД). Основні реляційні системи управління базами даних, які найчастіше використовуються на практиці, розглянуто нижче. Кожна з них має свої характерні риси, переваги та відповідні сфери застосування.

1. MySQL – це одна з найпопулярніших систем управління базами даних з відкритим вихідним кодом. Вона вирізняється високою швидкістю роботи, широкою підтримкою спільноти, легкістю інтеграції з різними мовами програмування та CMS. MySQL використовується у безлічі веб-проектів, від блогів до складних корпоративних систем. Вона підтримує транзакції, індекси, зовнішні ключі, реплікацію та інші інструменти для масштабування.
2. PostgreSQL – це потужна об'єктно-реляційна СУБД, яка підтримує розширену функціональність, зокрема зберігання складних типів даних, виконання користувацьких функцій, повноцінну роботу з транзакціями та розширення бази за допомогою сторонніх модулів. PostgreSQL відома своєю надійністю, дотриманням стандартів SQL та високим рівнем безпеки, що робить її популярною у фінансових,

державних та аналітичних системах. Вона також підтримує роботу з просторовими даними завдяки модулю PostGIS.

3. SQLite – легка, вбудована СУБД, яка не вимагає окремого серверного процесу. Вона часто використовується у мобільних застосунках, десктопних програмах, а також у невеликих веб-проектах або прототипах. SQLite зберігає дані у вигляді одного файлу на диску, що робить її простою у використанні, хоча і менш придатною для систем із високим навантаженням або великою кількістю одночасних користувачів.
4. Microsoft SQL Server – це комерційна реляційна СУБД, розроблена компанією Microsoft. Вона орієнтована на корпоративний сектор і має потужний набір засобів для адміністрування, резервного копіювання, реплікації, бізнес-аналітики (SSAS), інтеграції з іншими продуктами Microsoft та хмарними рішеннями Azure. Вона забезпечує високу безпеку даних, масштабованість і зручність у супроводі великих інформаційних систем.

Реляційні бази ідеально підходять для структурованих даних із чіткими зв'язками та обмеженнями. Вони забезпечують цілісність даних, підтримують транзакції, індекси, складні запити, агрегування та аналітику.

У сучасних умовах, коли цифрові продукти генерують великі обсяги даних у режимі реального часу, а вимоги до масштабованості, гнучкості й продуктивності систем постійно зростають, нереляційні бази даних (NoSQL) стали невід'ємною частиною архітектури багатьох сучасних веб-застосунків. Вони особливо актуальні у високонавантажених системах, хмарних сервісах, інтернеті речей, аналітичних платформах, а також у випадках, коли дані не мають стабільної структури або схильні часто змінюватися. Головною перевагою NoSQL-баз є можливість зберігати та обробляти дані без попередньо визначеної схеми, що значно пришвидшує розробку та спрощує адаптацію до нових вимог.

NoSQL-бази даних з'явилися як відповідь на потребу в гнучкішій, горизонтально масштабованій системі зберігання, особливо для великих обсягів даних і високої швидкості читання/запису. Вони не використовують табличну структуру, як у реляційних СУБД, а базуються на інших моделях зберігання: документи, ключ-значення, графи, колонки тощо.

Серед основних типів NoSQL-баз можна виділити такі.

1. Документоорієнтовані бази даних, зокрема MongoDB та CouchDB, зберігають інформацію у вигляді окремих документів, найчастіше у форматі JSON або BSON. Кожен документ є самостійною одиницею зберігання, яка може містити вкладені структури, масиви та змінну кількість полів. Такий підхід забезпечує гнучкість – структура документа не обов'язково має відповідати загальному шаблону, що дозволяє працювати з різномірними даними. Ці системи особливо ефективні в проєктах, де необхідно швидко додавати або змінювати поля у записах без складних міграцій, наприклад, у CMS або персоналізованих профілях користувачів.
2. Ключ-значення бази даних (наприклад, Redis, DynamoDB) реалізують найпростішу форму зберігання даних, де кожному унікальному ключу відповідає одне значення. Такі системи забезпечують надзвичайно високу швидкодію при читанні й записі, тому широко використовуються для кешування, зберігання сесій, обліку кількостей (лічильників), а також у потоковій обробці подій. Redis, зокрема, підтримує типи даних, як-от списки, множини та упорядковані множини, що робить його зручним для задач реального часу.
3. Графові бази даних (такі як Neo4j, ArangoDB, Dgraph, OrientDB) призначені для зберігання та обробки складних зв'язків між об'єктами за допомогою графової моделі. У таких базах даних об'єкти представлені у вигляді вузлів, а їхні взаємозв'язки – у вигляді ребер графа. Цей підхід особливо корисний для аналізу

взаємин між користувачами в соціальних мережах, наприклад, коли один користувач підписаний на іншого або належить до певної спільноти. Окрім того, графові бази даних ефективно застосовуються для виявлення підозрілих транзакцій і фінансових схем, де можна простежити зв'язки між рахунками, картками або іншими реквізитами контрагентів. Серед головних переваг – висока продуктивність: пошук і аналіз зв'язків між вузлами відбувається набагато швидше, ніж при використанні традиційних реляційних баз з великою кількістю таблиць і складними JOIN-запитами [12].

4. Колонкові бази даних, це системи баз даних, у яких інформація організована не за рядками, як у традиційних реляційних базах, а за стовпцями. На відміну від класичних таблиць, де дані зберігаються рядками, у колонкових базах кожен стовпець зберігається окремо – у спеціальному файлі або сегменті сховища. При цьому зберігається відповідність між елементами різних колонок: перше значення в одному стовпці пов'язане з першими значеннями інших. Такий підхід дозволяє обробляти лише необхідні стовпці при виконанні запиту, що значно знижує навантаження на пам'ять і пришвидшує роботу з великими обсягами даних [13].

Усі перелічені NoSQL-рішення зазвичай є *schema-less* – тобто не вимагають попереднього визначення структури даних, що дає змогу гнучко реагувати на зміни у застосунку. Крім того, вони відзначаються високою горизонтальною масштабованістю, дозволяючи легко збільшувати потужність системи шляхом додавання нових серверів. Це робить їх ефективними інструментами у сучасній веб-розробці, де важливі швидкість, адаптивність і надійність обробки великих та нестабільних обсягів інформації.

У сучасній розробці застосунків важливою складовою є використання технології ORM (Object-Relational Mapping) – об'єктно-реляційного відображення. Цей підхід значно спрощує взаємодію між програмним кодом

і базою даних, підвищуючи ефективність розробки та зменшуючи кількість потенційних помилок. Замість того, щоб писати складні SQL-запити вручну, розробники можуть працювати з об'єктами й класами мови програмування, які відповідають таблицям і записам у базі даних. Такий підхід робить код читабельнішим, легшим у підтримці та незалежнішим від конкретної СУБД.

Суть ORM полягає в тому, що кожна таблиця в базі даних відповідає певному класу, а рядки цієї таблиці – об'єктам цього класу. Завдяки цьому зчитування, модифікація або створення записів у базі даних здійснюються так само, як і робота з об'єктами у мові програмування. ORM автоматизує процес генерації запитів до бази, дозволяє будувати складні зв'язки між сутностями, підтримує транзакції та забезпечує контроль цілісності даних. Основні переваги використання ORM полягають у зменшенні кількості шаблонного коду, оскільки більшість CRUD-операцій виконується простими методами, підвищенні безпеки завдяки вбудованому захисту від SQL-ін'єкцій, можливості роботи з різними СУБД без необхідності змінювати основну логіку коду, підтримці міграцій бази даних, що дає змогу зручно змінювати структуру таблиць через код, а також в автоматичній генерації запитів, яка значно спрощує вибірку, фільтрацію, сортування та об'єднання даних.

Приклади популярних ORM.

1. Sequelize – підтримує PostgreSQL, MySQL, SQLite, MSSQL; широко використовується у JavaScript-застосунках.
2. TypeORM – ORM для TypeScript і Node.js із підтримкою декораторів, відображення зв'язків і вбудованої системи міграцій.
3. Prisma – це інструмент для об'єктно-реляційного відображення (ORM), що забезпечує з'єднання між застосунком і базою даних. Він пропонує типобезпечний API, який робить роботу з базами даних більш надійною та зменшує ризик помилок. Prisma сумісний із різними СУБД, зокрема PostgreSQL, MySQL, SQLite та MongoDB [14].

Використання ORM у розробці месенджера є особливо важливим, оскільки дозволяє ефективно управляти складною структурою зв'язаних даних, таких як користувачі, повідомлення та вкладення. Завдяки ORM розробник може легко реалізовувати створення, оновлення та отримання повідомлень, встановлювати зв'язки між користувачами та їхньою активністю, забезпечувати цілісність даних і одночасно зменшити обсяг шаблонного SQL-коду. Це значно прискорює розробку, спрощує масштабування проєкту та зменшує ризик помилок у логіці доступу до бази даних, що критично для стабільної роботи системи обміну повідомленнями в режимі реального часу.

Проектування бази даних є одним із ключових етапів розробки будь-якого веб-застосунку, оскільки саме від структури БД залежить ефективність зберігання, доступу та обробки даних, стабільність роботи системи та можливість її подальшого масштабування. Грамотно спроектована база даних дозволяє уникнути надмірності, забезпечити цілісність даних, а також оптимізувати виконання запитів, особливо при роботі з великими обсягами інформації.

У контексті розробки месенджера, мінімально необхідна структура бази даних може складатися з трьох основних таблиць: таблиця користувачів, таблиця повідомлень та таблиця файлів, прикріплених до повідомлень. Таблиця користувачів містить основну інформацію про кожного зареєстрованого користувача (ID, логін, пароль тощо) та слугує базовою сутністю для встановлення зв'язків з іншими таблицями. Таблиця повідомлень зберігає безпосередньо текст повідомлень, їхній час створення, ID відправника і, за потреби, одержувача чи чату. Таблиця файлів відповідає за зберігання метаданих щодо вкладених файлів (назва, розширення, шлях до файлу, розмір) і містить зовнішній ключ, який пов'язує файл із конкретним повідомленням. На рисунку 2.1 зображено цю структуру бази даних, що використовується у проєкті месенджера.

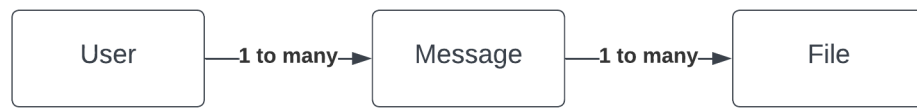


Рисунок 2.1 – Структура БД, що використовується месенджері.

Така структура забезпечує логічну цілісність системи, дозволяє масштабувати функціонал (наприклад, додавати чати або статуси повідомлень), а також ефективно виконувати вибірки та фільтрацію під час роботи з інтерфейсом.

2.2 Вибір технологій розробки та середовища

Успішна реалізація веб-застосунку напряду залежить від правильно обраного технологічного стеку, який має відповідати вимогам проєкту, забезпечувати високу продуктивність, масштабованість, безпеку, а також зручність розробки та супроводу. Зважаючи на функціональні особливості застосунку типу месенджера – з обробкою повідомлень, прикріпленням файлів, роботою з користувачами в реальному часі – було обрано сучасні, перевірені технології з активною спільнотою та широкою екосистемою.

Для реалізації клієнтської частини обрано React – сучасна JavaScript-бібліотека для створення користувацького інтерфейсу. Вона забезпечує високу продуктивність завдяки використанню віртуального DOM, дозволяє будувати інтерфейс на основі компонентів та підтримує гнучку структуру коду. React особливо добре підходить для динамічних застосунків, де вміст сторінки часто оновлюється без повного перезавантаження. Для стилізації інтерфейсу обрано Tailwind CSS – утилітарну бібліотеку, яка надає набір класів для швидкого створення адаптивного, сучасного дизайну без потреби

написання власних CSS-файлів. Її перевагами є швидкість верстки, зручна настройка та хороша інтеграція з компонентною структурою React.

Для серверної частини веб-застосунку обрано Express – мінімалістичний веб-фреймворк для Node.js. Express дозволяє швидко налаштовувати RESTful API, організовувати маршрутизацію, обробку запитів і відповідей, реалізовувати проміжні обробники (middleware) та інтегрувати з базами даних або сторонніми сервісами. Його простота, гнучкість і активна підтримка спільноти роблять Express популярним вибором для побудови сучасних серверних застосунків.

Для зберігання та обробки даних обрано PostgreSQL – потужну об'єктно-реляційну систему управління базами даних, яка поєднує підтримку класичної реляційної моделі з розширеними можливостями, такими як зберігання JSON, транзакції, індексація, повнотекстовий пошук тощо. PostgreSQL має високу продуктивність, забезпечує цілісність даних і добре масштабується, що є важливим для застосунків зі зростаючою кількістю користувачів і повідомлень.

Для взаємодії між сервером і базою даних обрано Prisma – сучасний ORM для Node.js та TypeScript. Prisma дозволяє працювати з базою даних через об'єктну модель, автоматично генерує типізовані клієнти для доступу до даних, спрощує реалізацію CRUD-операцій, підтримує створення міграцій і забезпечує прозору інтеграцію з PostgreSQL. Це зменшує кількість шаблонного коду, підвищує безпеку та покращує читабельність бізнес-логіки.

В якості основного середовища розробки обрано Visual Studio Code – легкий, швидкий та розширюваний текстовий редактор з широким спектром розширень для роботи з JavaScript, TypeScript, React, Express і Prisma. Його гнучкість, інтеграція з терміналом, автодоповненням, форматуванням коду та інструментами налагодження значно пришвидшує процес розробки й тестування.

2.3 Вибір моделі ШІ для підсумування тексту

Підсумовування тексту є ключовою функцією розробленого месенджера, оскільки саме вона забезпечує стислий виклад змісту повідомлень, дозволяючи користувачеві швидко ознайомлюватися з великою кількістю інформації. Реалізація цієї функції потребує використання сучасної моделі штучного інтелекту, здатної до високоякісного розуміння тексту, виділення головного та формування граматично коректних і логічно зв'язних узагальнень.

Для реалізації інтелектуального підсумовування було обрано модель GPT-4o – новітню мультимодальну модель від OpenAI, яка підтримує роботу з текстом, зображеннями та аудіо в одному середовищі, але в межах цього проєкту використовується для роботи з текстом. Вибір GPT-4o обумовлений її здатністю до розуміння довгих контекстів, високою якістю генерованого тексту, стабільністю відповіді та адаптивністю до стилю вхідного запиту. Крім того, GPT-4o є швидшою та менш ресурсоємною, ніж попередні версії, що дозволяє використовувати її навіть у проєктах із реальним часом та обмеженими затримками.

Під час аналізу було також розглянуто альтернативні варіанти – Hugging Face Inference API та Replicate API, однак вони не були обрані з низки причин.

Hugging Face Inference API, хоча й надає доступ до великої кількості відкритих мовних моделей (наприклад, BART, T5, Pegasus тощо), має суттєві обмеження: продуктивність і якість відповіді сильно залежать від обраної моделі, необхідність окремо налаштовувати кожну модель для роботи з українською або змішаною мовою, а також відсутність єдиного інтерфейсу з високою стабільністю для різних задач. Крім того, навіть найсильніші доступні моделі часто демонструють нижчу якість узагальнень порівняно з

GPT-4o, особливо в задачах, що потребують емоційної інтерпретації чи адаптивного стилю.

Replicate API є зручним сервісом для розгортання моделей із відкритим кодом, однак він не надає єдиної універсальної моделі, що оптимально підходила б саме для підсумовування. Окрім того, ці моделі часто мають нестабільну продуктивність, тривалий час відповіді та обмежену підтримку складних мовних структур. Інтеграція з Replicate API також потребує додаткової обробки форматів вхідних і вихідних даних, що ускладнює побудову єдиного API-інтерфейсу застосунку.

На відміну від зазначених сервісів, GPT-4o надає можливість отримувати точні, граматично правильні й стилістично природні підсумки, не потребуючи окремого налаштування чи донавчання. Її інтеграція через OpenAI API є стабільною, добре документованою та масштабованою.

2.4 Проєктування функціоналу

Функціональність веб-застосунку месенджера визначає його практичну цінність для кінцевого користувача. Під час проєктування особливу увагу було приділено зручності взаємодії, швидкості виконання дій, а також забезпеченню інтелектуальних можливостей, які відрізняють даний застосунок від типових засобів обміну повідомленнями.

Першочергово, користувач повинен мати змогу зареєструватися у системі та авторизуватися під своїм обліковим записом. У рамках цього функціоналу реалізовано перевірку унікальності логіну, захищене зберігання пароля, а також систему аутентифікації для подальшої взаємодії з сервером. Після входу до системи користувач потрапляє до основного інтерфейсу – списку доступних чатів.

Кожен користувач може обирати чат для перегляду та спілкуватися з іншими користувачами, надсилаючи текстові повідомлення. Повідомлення можуть містити один або кілька прикріплених файлів – документів, зображень або інших типів контенту.

Ключовою особливістю застосунку є можливість підсумування повідомлень за допомогою штучного інтелекту. Користувач може обрати окреме повідомлення і викликати функцію стислого викладу його змісту, що особливо корисно для великих текстів або повідомлень із надмірною деталізацією. В результаті замість оригінального повідомлення відображається його скорочена версія, сформована мовною моделлю. При цьому реалізована можливість повернення повідомлення до його первісного вигляду, що дозволяє зберегти повну інформацію та уникнути втрати змісту.

Усі дії користувача реалізуються через відповідні маршрути API, з дотриманням принципів безпеки, валідації даних і логічної цілісності бази. Така структура функціоналу забезпечує не лише зручність використання, але й надає платформі потенціал для подальшого розширення.

2.5 Висновки до другого розділу

У другому розділі було здійснено обґрунтований вибір технологій, архітектури та інструментів, необхідних для реалізації веб-застосунку такого як месенджер. Розглянуто переваги використання реляційних та нереляційних баз даних, а також доцільність впровадження ORM-технології для зручної та безпечної взаємодії із системою управління базами даних. Обрано стек технологій, що включає React, Tailwind CSS, Express, PostgreSQL та Prisma, які забезпечують продуктивну, гнучку й масштабовану архітектуру застосунку. Окрему увагу приділено вибору моделі штучного інтелекту для функції підсумовування тексту: проаналізовано кілька

альтернатив і обґрунтовано використання моделі GPT-4o як найоптимальнішої. Також було проєктовано ключовий функціонал застосунку, включно з обміном повідомленнями, прикріпленням файлів та інтеграцією інтелектуального підсумовування з можливістю повернення до оригінального тексту. Усі розглянуті рішення створюють основу для реалізації повнофункціонального сучасного месенджера.

ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Набір технологій

Для реалізації програмного продукту було використано сучасний технологічний стек, який забезпечує ефективну взаємодію між клієнтською та серверною частинами, надійне зберігання і обробку даних, а також гнучкість і можливість масштабування в подальшому. На клієнтському боці (frontend) використано React – популярну JavaScript бібліотеку, яка дозволяє створювати динамічні веб-інтерфейси на основі компонентного підходу. Завдяки цьому забезпечується повторне використання коду, зручне структурування логіки відображення та швидка розробка. Для стилізації інтерфейсу застосовано Tailwind CSS – утилітарний CSS-фреймворк, що дозволяє створювати адаптивні й сучасні інтерфейси без необхідності писати окремі CSS-файли. Використання Tailwind дозволяє пришвидшити верстку, зменшити кількість коду та підтримувати єдиний стиль на всіх сторінках застосунку.

Серверна частина (backend) побудована на основі Express – легкого й продуктивного фреймворку для Node.js, який дозволяє реалізовувати RESTful API, обробку HTTP-запитів, маршрутизацію, автентифікацію та взаємодію з базами даних. Для зберігання структурованої інформації використовується PostgreSQL – надійна об'єктно-реляційна СУБД, що забезпечує підтримку складних запитів, транзакцій, індексації та розширень. Запити до бази реалізовано через Prisma – сучасну ORM-систему для Node.js, яка дозволяє описувати модель даних декларативно, виконувати автоматичні міграції, працювати з типізованими запитами й зменшити обсяг шаблонного SQL-коду. Для інтеграції логіки з ШІ, використано модель від OpenAI, а саме GPT-4o. Як середовище розробки застосовано Visual Studio Code, яке

забезпечує зручну інтеграцію з інструментами розробника: терміналом, системою контролю версій Git, автодоповненням коду, налагодженням та розширеннями для роботи з React, Node.js.

3.2 Архітектура програмного продукту

Архітектура програмного продукту побудована за класичною моделлю клієнт-сервер, де клієнтська частина відповідає за взаємодію з користувачем, а серверна частина відповідає за обробку логіки, доступ до бази даних і реалізацію функцій застосунку. Такий підхід дозволяє чітко розмежувати відповідальність між компонентами, підвищити гнучкість системи та спростити супровід коду.

На рисунку 3.1 зображено базову структуру архітектури взаємодії між основними частинами системи. Користувач взаємодіє з клієнтською частиною, створеною на базі React, яка надсилає HTTP-запити до серверу, реалізованого на Express. Серверна частина обробляє запити, виконує необхідну бізнес-логіку та звертається до бази даних PostgreSQL через ORM Prisma. Отримані дані повертаються користувачу у вигляді JSON-відповідей. Така структура забезпечує швидку реакцію системи, стабільний обмін даними та можливість масштабування при зростанні навантаження.



Рисунок 3.1 – Базова архітектура взаємодії клієнта, сервера та БД

Детальніше архітектуру програмного продукту можна розглянути через структуру компонентів реалізації, де показано, як саме організовано

код та які модулі відповідають за окремі функціональні частини застосунку. Така схема дозволяє краще зрозуміти логіку взаємодії між частинами проєкту. Схема основної структури фронкенду зображена на рисунку 3.2.

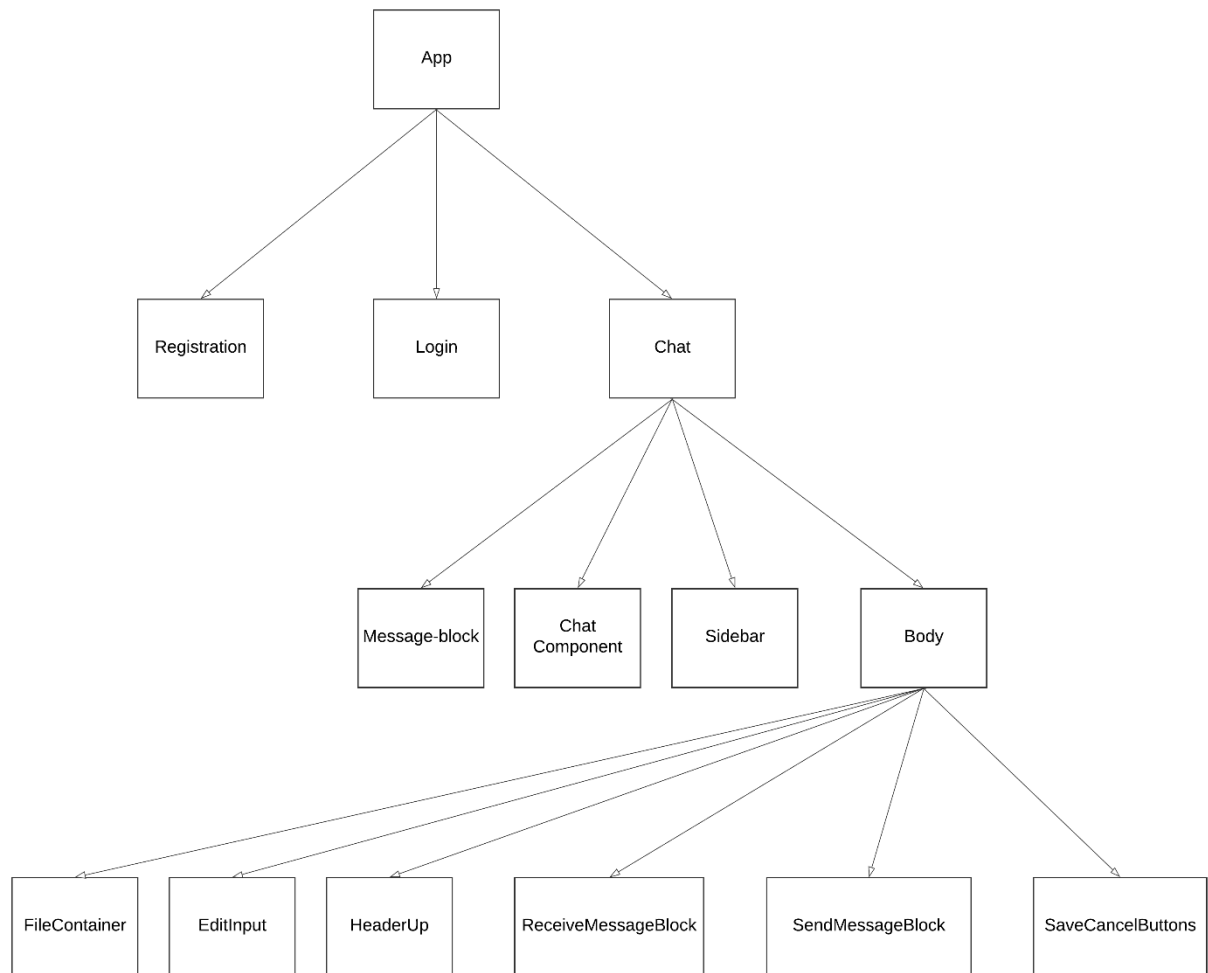


Рисунок 3.2 – Схема основної структури фронкенду

Компонент App є кореневим і відповідає за маршрутизацію між сторінками такими як Registration, Login та основної сторінки для користувача - Chat. Сторінка Chat логічно розділена на кілька компонентів: Message-block (відповідає за відображення повідомлень), Chat Component, Sidebar (бічна панель чатами) та Body, що є основним контейнером для контенту повідомлень.

Компонент Body додатково поділений на менші функціональні блоки.

1. FileContainer – відображення прикріплених файлів.
2. EditInput – поле введення тексту при редагуванні.

3. HeaderUp – верхній компонент вікна чату.
4. ReceiveMessageBlock – блок отриманих повідомлень.
5. SendMessageBlock – форма відправлених повідомлень.
6. SaveCancelButton – кнопки збереження та скасування змін.

Схема основної структури бекенду зображена на рисунку 3.3

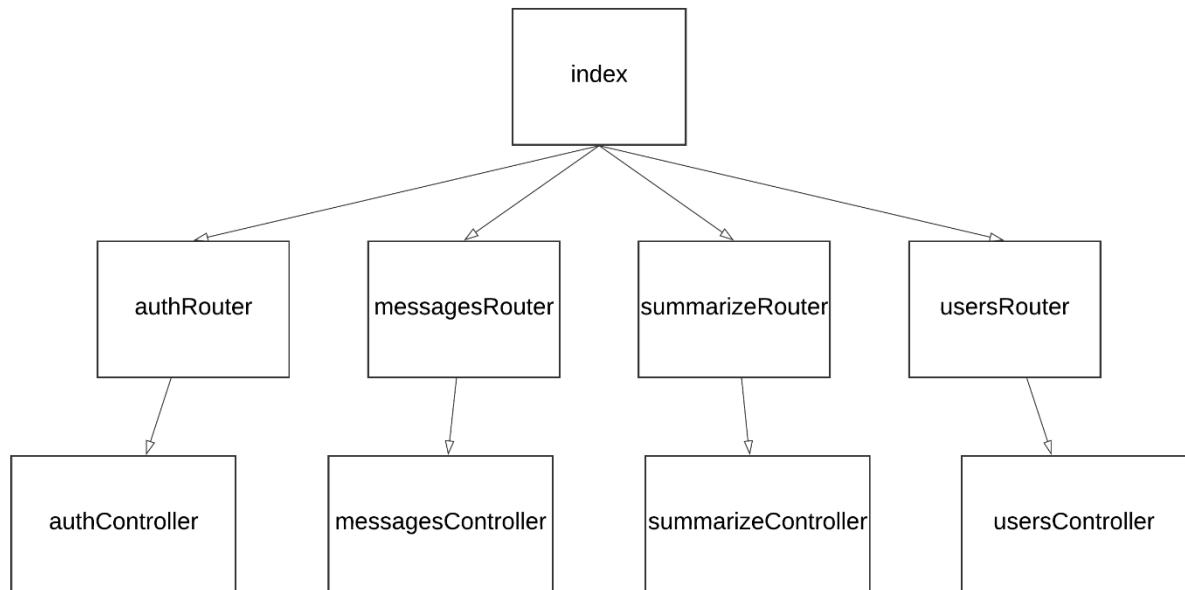


Рисунок 3.3 – Схема основної структури бекенду

Архітектура серверної частини побудована таким чином, що кожен модуль виконує окрему роль у логіці обробки запитів. Центральним елементом є файл `index`, який виконує ініціалізацію серверу, підключення маршрутів та запуск додатку. Далі йдуть чотири основні маршрути: `authRouter`, `messagesRouter`, `summarizeRouter` та `usersRouter`, кожен з яких відповідає за обробку запитів до відповідної групи функціональностей.

Кожен маршрут пов'язаний із відповідним контролером:

- 1) `authController` – обробка логіки реєстрації та автентифікації;
- 2) `messagesController` – логіка обміну повідомленнями, включно з обробкою вкладених файлів;
- 3) `summarizeController` – підключення до модуля штучного інтелекту для реалізації функції підсумування тексту;

4) `usersController` – дії над профілями користувачів та їх даними.

Така модульна побудова дозволяє легко масштабувати проєкт і підтримувати кожну частину окремо.

Окрім контролерів та маршрутів, важливою частиною архітектури є `middleware` – проміжне програмне забезпечення, що обробляє запити до потрапляння в основну логіку. У проєкті використано `middleware auth`, що забезпечує перевірку токенів і контроль доступу до захищених маршрутів, а також `multer-config`, який відповідає за обробку завантаження файлів. Це дозволяє централізовано контролювати безпеку, валідацію даних та обробку файлів, не дублюючи логіку в контролерах.

Такий підхід забезпечує високу організованість коду, зменшує зв'язність між модулями та сприяє спрощенню тестування і підтримки серверної частини застосунку.

3.3 Інтерфейс веб-застосунку

На рисунку 3.4 зображено сторінку входу користувача. Вона містить два основні поля: `Username` (введення імені користувача) та `Password` (введення пароля), а також кнопку для виконання авторизації. Нижче розташоване посилання, що дозволяє перейти до форми реєстрації нового користувача.

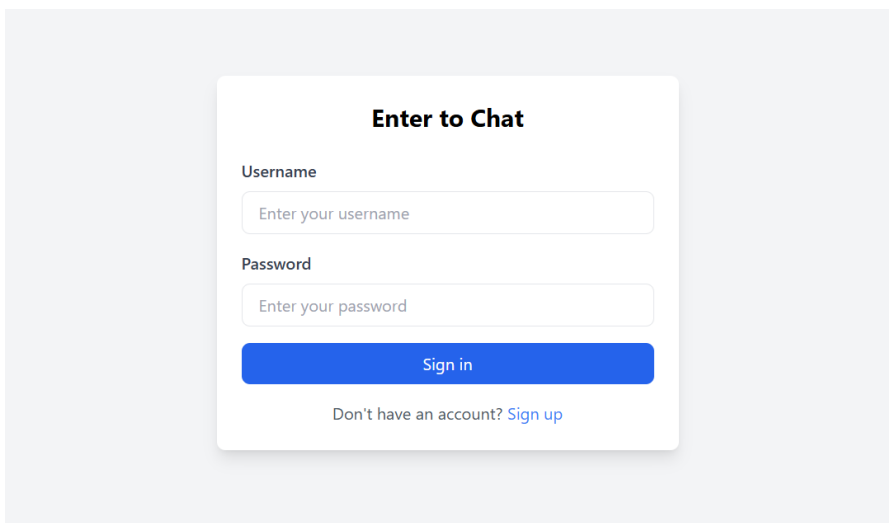
The image shows a login form titled "Enter to Chat" centered on a light gray background. The form is a white card with a subtle shadow. It contains two input fields: "Username" with a placeholder "Enter your username" and "Password" with a placeholder "Enter your password". Below these fields is a blue button labeled "Sign in". At the bottom of the card, there is a link that says "Don't have an account? Sign up".

Рисунок 3.4 – Сторінка авторизації користувача

На рисунку 3.5 зображено інтерфейс веб-застосунку, який відкривається після успішної авторизації користувача. Ліворуч розміщено список зареєстрованих користувачів, з якими можна розпочати або продовжити чат. Користувач обирає одного зі списку, після чого відкривається діалогове вікно з історією повідомлень. Вигляд діалогового вікна зображено на рисунку 3.6

У центрі сторінки розміщено вітальне повідомлення, яке повідомляє про головну особливість застосунку – можливість підсумовувати текст повідомлень. У правому верхньому куті розташована кнопка виходу із чату яка дозволяє завершити поточну сесію та повернутися до сторінки авторизації.

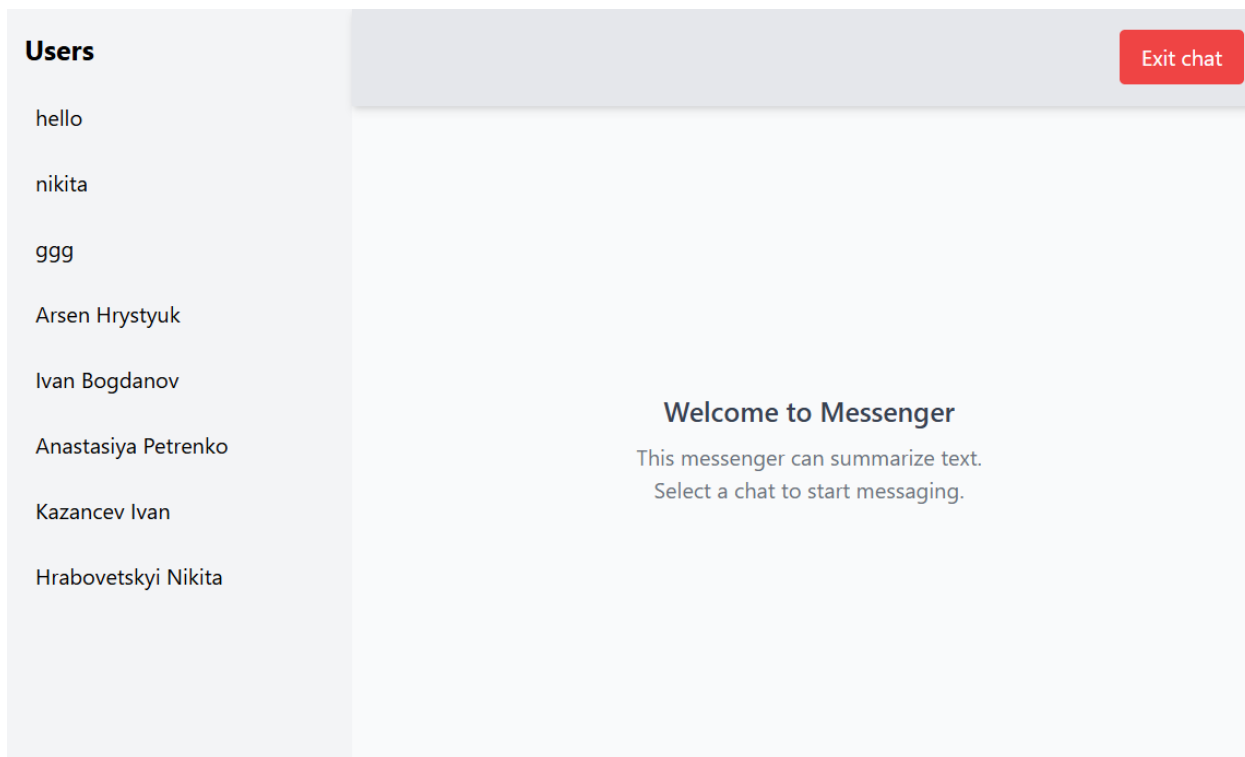


Рисунок 3.5 – Інтерфейс веб-застосунку після авторизації користувача

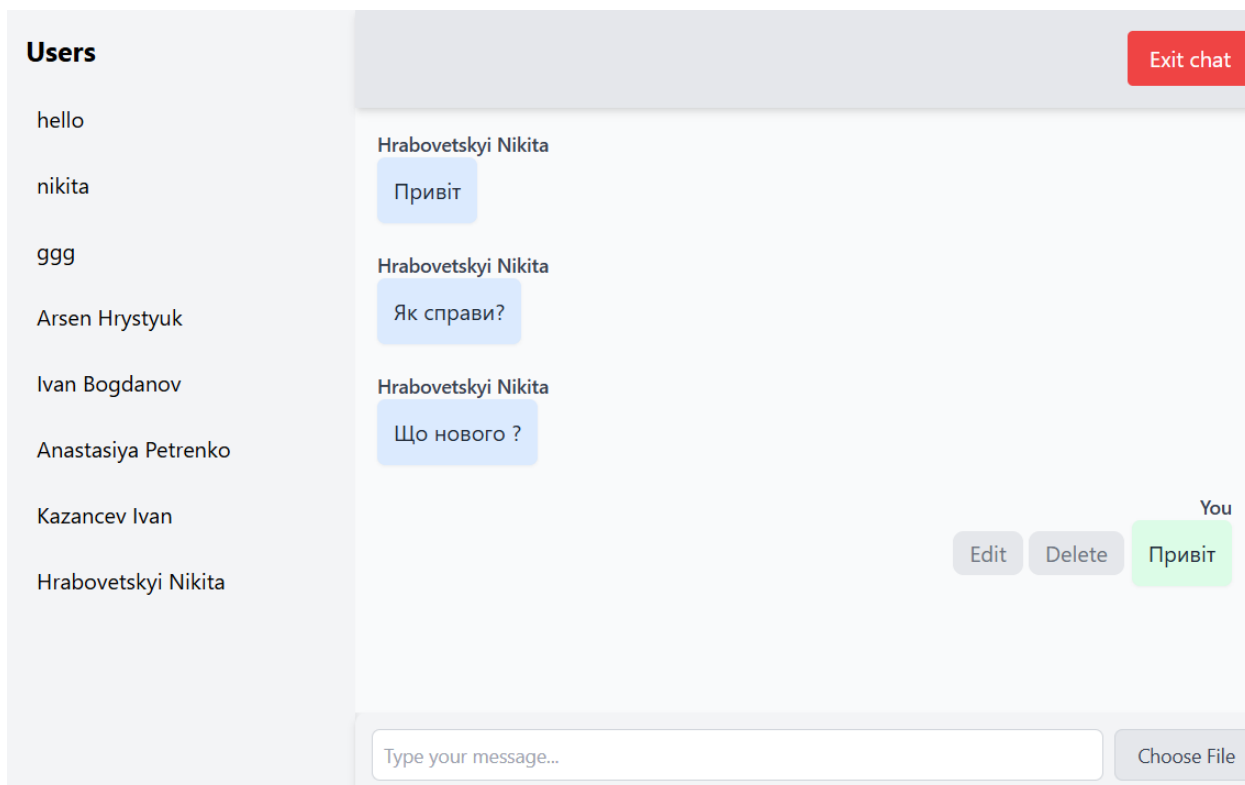


Рисунок 3.6 – Вигляд діалогового вікна

На рисунку 3.6 зображено основний інтерфейс чату. При наведенні на повідомлення з'являються кнопки редагування та видалення, що дозволяє

керувати власними повідомленнями після їх надсилання. У нижній частині вікна розташовано поле введення нового повідомлення та кнопка для прикріплення файлів, що забезпечує зручну інтерактивну комунікацію.

На рисунку 3.7 представлено інтерфейс чату з інтегрованою функцією інтелектуального підсумування. Праворуч від повідомлення, яке містить великий обсяг тексту, розташована кнопка для підсумування вмісту повідомлення. Після натискання кнопки виконується запит до API моделі GPT, яка формує короткий, зрозумілий зміст повідомлення. Поки клієнт очікує відповідь від сервера після натискання кнопки підсумування, на екрані відображається індикатор завантаження, що сигналізує про те, що обробка запиту триває.

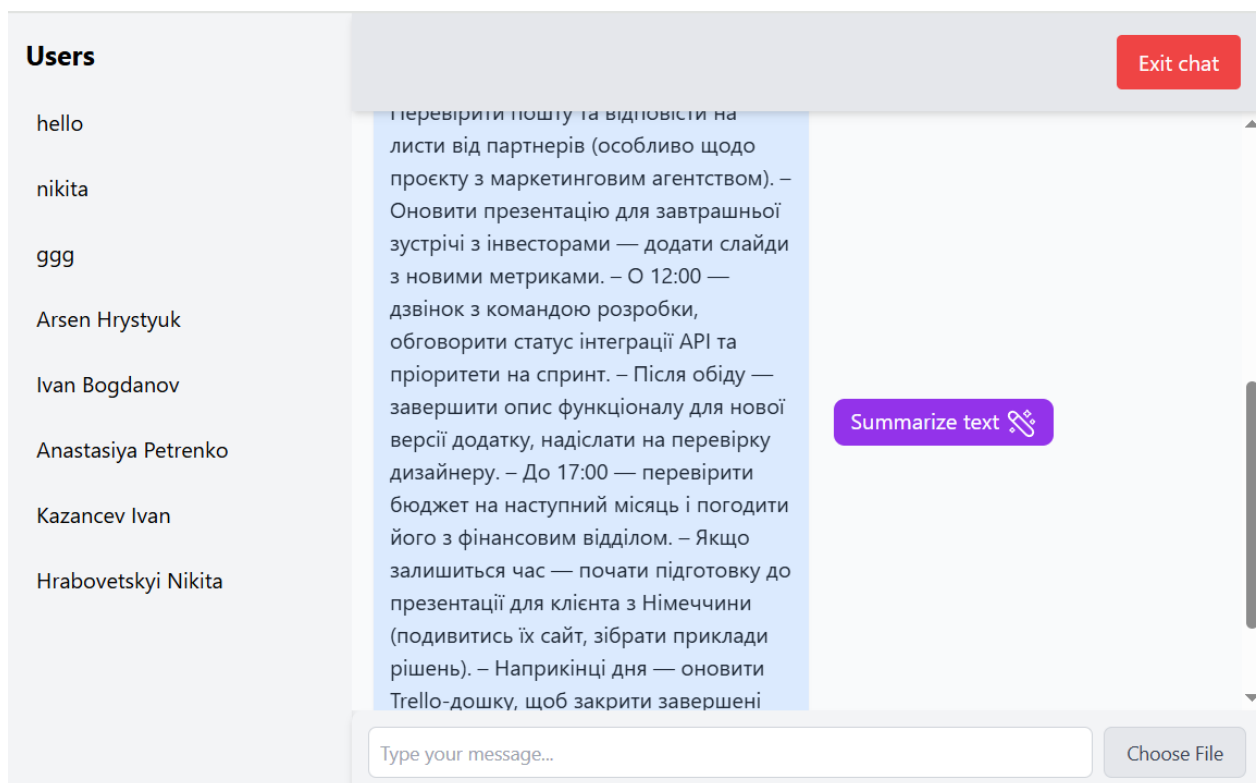


Рисунок 3.7 – Інтерфейс чату з функцією підсумування повідомлення

На рисунку 3.8 зображено функціональність чату після виконання підсумування повідомлення. Повідомлення користувача, що містило великий обсяг інформації, було скорочено до стислого викладу, а біля нього з'явилась кнопка повернутися до початкового тексту. Натиснувши цю кнопку,

користувач може швидко переглянути повний зміст повідомлення у його початковому вигляді.

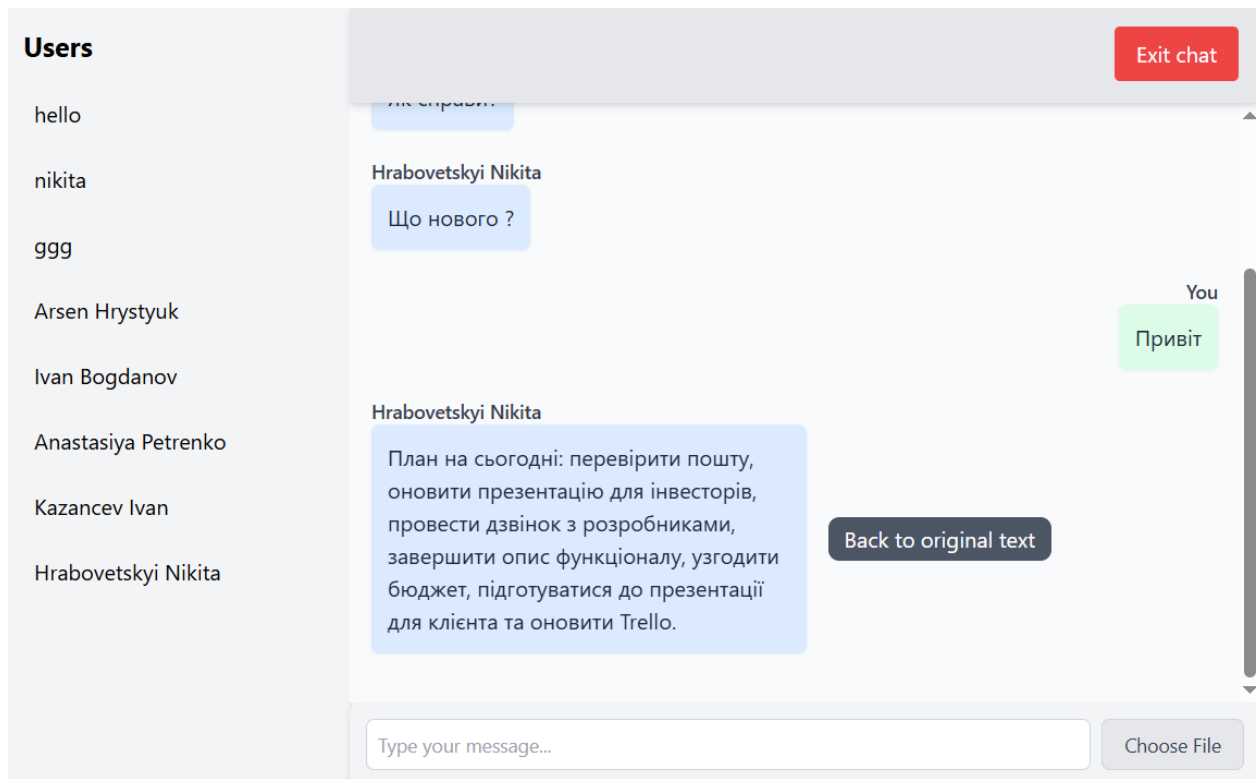


Рисунок 3.8 – Функціональність після підсумування повідомлення

3.4 Порівняння результатів

Для оцінки ефективності підсумування повідомлень, реалізованого у розробленому веб-застосунку з використанням GPT-4o, було проведено порівняльний аналіз результатів із публічним онлайн-інструментом NeuralWriter Summarize Tool [15], який також використовує технології штучного інтелекту.

Обидві системи отримували на вхід одне й те саме текстове повідомлення, сформоване у стилі щоденних задач користувача. Нижче наведена таблиця 3.1 з результатами підсумування:

Таблиця 3.1 - Порівняння результатів підсумування

Критерій	Мій веб-застосунок (GPT-4o)	NeuralWriter
Згенерований підсумок	План на день: завершити редагування дипломної роботи, переглянути лекцію з інформаційних технологій, підготуватися до зустрічі з науковим керівником, відповісти на електронні листи, доопрацювати портфоліо, зайнятися технічним завданням на NestJS, виконати тренування, пов'язати рукави до светра для релаксації.	Сьогодні планується багато робіт. Спочатку потрібно закінчити редагування четвертого розділу дипломної роботи, перевірити посилання на джерела та переформулювати висновки. Після обіду потрібно переглянути лекцію з "Інформаційними технологіями в архітектурі", зробити конспект та підготувати запитання до консультації. Також потрібно підготуватися до зустрічі з науковим керівником, відповісти на електронні листи та доповнити портфоліо. У другій половині дня планується технічне завдання на NestJS, а ввечері - тренування.
Стислість	Дуже стислий, чіткий перелік без зайвих пояснень	Розгорнутий переказ, наближений до повного повторення вихідного тексту
Охоплення змісту	Враховано всі ключові завдання, включно з побутовими деталями	Одна з дій (в'язання) була пропущена
Зручність сприйняття	Виглядає як список дій, придатний для планування	Виглядає як опис

Проведений аналіз підтвердив переваги підходу на основі GPT-4o. Застосунок формує більш компактні, точні й структуровані підсумки. Натомість підсумування повідомлення, сформоване за допомогою NeuralWriter, має форму переказу вихідного тексту, характеризується менш

чіткою структурою та в окремих випадках не охоплює всі суттєві аспекти повідомлення.

3.5 Висновки до третього розділу

У третьому розділі було продемонстровано архітектуру клієнтської та серверної частин, а також структуру організації коду, зокрема розподіл на компоненти, контролери, маршрути та middleware. Окрему увагу приділено реалізації інтерфейсу користувача, де забезпечено зручну взаємодію з чатами, повідомленнями, файлами та функцією підсумування. Інтеграція інтелектуального механізму узагальнення тексту, заснованого на моделі GPT, дозволяє користувачам швидко ознайомлюватися зі змістом великих повідомлень, зменшуючи інформаційне навантаження.

Таким чином, реалізований програмний продукт відповідає поставленим завданням, є технічно цілісним, функціональним і має потенціал для подальшого розвитку.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка завдання

У сучасних умовах активної цифровізації особливої актуальності набувають програмні інструменти, які дозволяють автоматизувати рутинні процеси взаємодії з інформацією. Одним із таких рішень є інтеграція штучного інтелекту у месенджери для автоматичного підсумування змісту повідомлень. Це забезпечує користувачам швидкий доступ до суттєвих і важливих частин розмов, підвищуючи ефективність спілкування як у приватному, так і в професійному середовищі.

Дане програмне рішення передбачає створення програмного продукту який може автоматично сформулювати стислий зміст з вхідних повідомлень за допомогою генеративних моделей ШІ. Такий підхід особливо корисний у випадках великих обсягів тексту або коли необхідно швидко орієнтуватися в історії переписки.

Щоб оцінити ефективність розробки, застосовується функціонально-вартісний аналіз (ФВА), який дозволяє порівняти різні варіанти реалізації з урахуванням якісних, технічних і економічних параметрів.

4.2 Обґрунтування функцій програмного продукту

Головною метою є створення веб застосунку , який може підсумувати зміст вхідних повідомлень за допомогою технологій ШІ. Було виділено наступні головні функції вибору.

F1 – вибір бібліотеки або фреймворку для відображення клієнтської частини веб застосунку:

- 1) react (а) ;
- 2) vue (б);
- 3) angular (в).

F2 – вибір фреймворку для серверної частини веб застосунку:

- 1) express (а);
- 2) nest (б);
- 3) koa (в).

F3 – вибір бази даних:

- 1) mySQL (а);
- 2) mongoDB (б);
- 3) postgresSQL (в).

F4 – вибір ШІ для підсумування тексту:

- 1) openAI (а);
- 2) hugging Face Inference API (б);
- 3) replicate API (в).

Можливі способи реалізації головних функцій системи представлені у вигляді морфологічної карти рисунку 4.1.

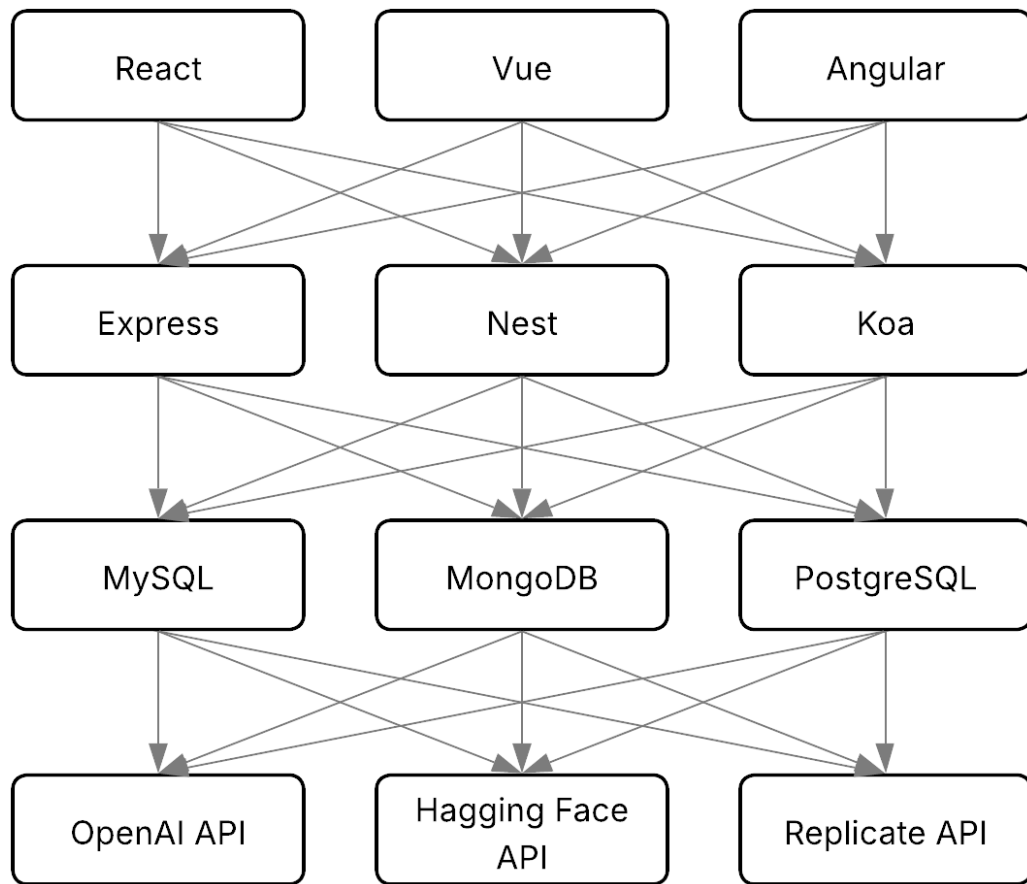


Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображає всю множину комбінацій для варіантів реалізації окремих функцій.

Позитивно-негативну матрицю наведено у таблиці 4.1.

Таблиця 4.1 Позитивно-негативна матриця

Основні функції	Варіант	Переваги	Недоліки
F1	а) React	Велика спільнота, популярний, гнучкий, великий вибір бібліотек	Немає вбудованої структури
	б) Vue	Швидкість в браузері, простий синтаксис	Мала спільнота, мала популярність, обмежена екосистема
	в) Angular	Стабільність, чітка структура, підходить для великих проєктів	Складність, великий об'єм коду
F2	а) Express	Простий синтаксис, мінімум залежностей, велика спільнота, гнучкість	Немає вбудованої структури, потребує ручної реалізації багатьох функцій
	б) Nest	Строга архітектура, вбудовані модулі, підходить для великих проєктів	Складність, великий об'єм коду, потребує знання Typescript
	в) Koa	Простий синтаксис, чистий middleware підхід	Немає вбудованого роутера, мала популярність

Продовження таблиці 4.1

F3	а) MySQL	Простота у використанні, швидкість на читанні даних.	Обмеженість підтримки складної логіки, менша гнучкість у структурі даних
	б) MongoDB	Гнучка NoSQL модель, підходить для проєктів без чіткої схеми, швидкість розробки.	Обмеженість при складних запитах та зв'язках
	в) PostgreSQL	Багатий функціонал, висока відповідність стандартам, швидко працює зі складними запитамми, масштабованість	Складність в адмініструванні
F4	а) OpenAI	Висока якість моделей, проста інтеграція, масштабованість	Платний доступ
	б) Hugging Face Inference API	Великий вибір моделей, можливий безкоштовний доступ	Швидкість безкоштовних моделей, нестабільність
	в) Replicate API	Підтримка кастомних моделей, відкрита екосистема	Складність інтегрування, платний доступ

Проаналізувавши позитивно-негативну матрицю, можна зробити висновки щодо доцільності використання тих чи інших варіантів реалізації функцій. Деякі з них виявилися невідповідними до вимог програмного продукту, враховуючи технічні особливості та обсяг передбачуваної роботи. Такий аналіз дозволив відсіяти менш ефективні рішення та зосередитися на тих підходах, які найкраще відповідають завданню проєкту.

Функція F_1 .

Для побудування користувацького інтерфейсу месенджера, найкраще підійде React, оскільки він має велику спільноту, гнучкість і широкий вибір бібліотек, що спрощує розробку та масштабування. Vue не підходить через обмежену екосистему та меншу популярність. Та Angular – надто складний і громіздкий для проєкту з невеликим обсягом.

Функція F_2 .

Express обрано як найкращий варіант для реалізації серверної частини месенджера з ШІ. Оскільки має простий синтаксис, велику спільноту, мінімум залежностей та високу гнучкість. Це дає змогу швидко інтегрувати сторонні сервіси та налаштовувати потрібну логіку підсумування повідомлень. Nest має надто складну і громіздку структуру для невеликого проєкту, та потребує знання TypeScript. Koa є мінімалістичним, але має меншу популярність і не містить вбудованого роутера, що ускладнює розробку.

Функція F_3 .

Для бази даних обрано PostgreSQL, бо має багатий функціонал, високу відповідність стандартам, підтримку складних запитів і добру масштабованість. MySQL є менш гнучкий у структурі даних і не підтримує складну логіку на рівні запитів. MongoDB має обмеження при роботі з запитами, які містять багато зв'язків, тому не підходить для мого проєкту.

Функція F_4 .

ШІ для реалізації функції підсумування повідомлень у месенджері буде OpenAI API, оскільки забезпечує високу якість моделей (GPT-4, GPT-3.5), просту інтеграцію через API та стабільну роботу з великими обсягами даних.

Hugging Face Inference API є зручний для тестування проєкту, але менш стабільний і повільніший при безкоштовному використанні. Replicate API підтримує кастомні моделі, але складніший в інтеграції та також є платним.

Отже, таким чином, буде розглянуто такий варіант реалізації:

$$F_1a - F_2a - F_3b - F_4a$$

4.3 Обґрунтування системи параметрів програмного продукту

У цьому розділі розглядаються основні характеристики проєкту з метою визначення коефіцієнта його технічного рівня. Для характеристики системи обрано три параметри.

1. X1 – швидкодія веб застосунку.
2. X2 – об'єм коду.
3. X3 – швидкість підсумування тексту.

Згідно з установленними вимогами були визначені гірші, середні та кращі значення характеристик, які наведено у таблиці 4.2.

Таблиця 4.2 – Основні характеристики програмного продукту

Параметр	Позначення	Одиниця виміру	Гірше	Середньо	Краще
Швидкодія веб застосунку	X1	мс	800	400	100
Об'єм коду	X2	Кількість рядків	10000	8000	5000
Швидкість підсумовування тексту	X3	с	8	4	1

За даними з таблиці 4.2 було побудовано графічні характеристики параметрів - рисунки 4.2, 4.3, 4.4.

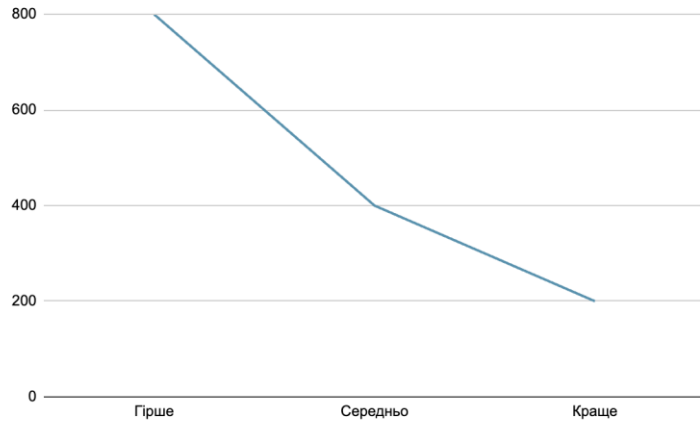


Рисунок 4.2 Графік швидкодії веб-застосунку

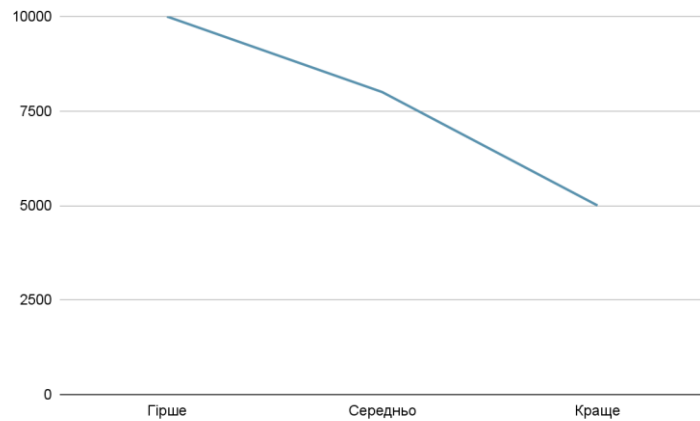


Рисунок 4.3 Графік об'єму коду

Обчислено наступні параметри.

1. Сума рангів кожного з параметрів і загальна сума рангів (4.1):

$$R_i = \sum_{j=1}^N r_{ij} = 42. \quad (4.1)$$

2. Середня сума рангів (4.2):

$$T = \frac{1}{n} R_i \approx 14. \quad (4.2)$$

3. Відхилення суми рангів кожного параметра від середньої суми рангів (4.3):

$$\Delta_i = R_i - T. \quad (4.3)$$

Сума всіх відхилень повинна дорівнювати 0.

4. Загальна сума квадратів відхилення:

$$S = \sum_{i=1}^N \Delta_i^2 = 78. \quad (4.4)$$

5. Коефіцієнт узгодженості (4.5):

$$W = \frac{12S}{N^2(n^3-n)} = \frac{12 \cdot 78}{7^2(3^3-3)} \approx 0.796 > 0.67. \quad (4.5)$$

Обчислений коефіцієнт узгодженості становить приблизно 0.796, що перевищує нормативне значення 0.67, встановлене для параметрів програмного продукту. Це свідчить про високий рівень згоди між експертами.

На наступному етапі було здійснено попарне порівняння параметрів, при цьому числові значення ступеня переваги визначались згідно з формулою 4.6.

$$X_{ij} = \{0.5 \text{ при } X_i < X_j \text{ } 1.0 \text{ при } X_i = X_j \text{ } 1.5 \text{ при } X_i > X_j\}. \quad (4.6)$$

Результати наведені у таблиці 4.4.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	<	<	<	<	<	<	<	0.5
X1 і X3	<	<	<	<	<	<	<	<	0.5
X2 і X3	<	>	>	>	<	>	>	>	1.5

На основі отриманих числових оцінок переваги будується матриця $A = \|x_{ij}\|$. Далі для кожного з параметрів обчислюється коефіцієнт вагомості K_{bi} за формулами (4.7)-(4.10).

$$K_{bi} = \frac{b_i}{\sum_{i=1}^n b_i}, \quad (4.7)$$

$$b_i = \sum_{j=1}^N x_{ij}. \quad (4.8)$$

Починаючи з другого кроку, на всіх етапах розрахунків застосовувалися відповідні формули.

$$K_{bi} = \frac{b'_i}{\sum_{i=1}^n b'_i}, \quad (4.9)$$

$$b'_i = \sum_{j=1}^N x_{ij} b_j. \quad (4.10)$$

Процес ітерацій продовжується до моменту, коли відхилення між поточними та попередніми значеннями не перевищує 5%. Отримані результати наведені в таблиці 4.5.

Таблиця 4.5 – Розрахунок вагових коефіцієнтів параметрів

Параметри x_i	Параметри x_j			Перша ітер.		Друга ітер.		Третя ітер.	
	X1	X2	X3	b_i	K	b_i	K_{bi}	b_i	K_{bi}
X1	1	0.5	0.5	4	0.44	0.72	0.22	0.61	0.22
X2	1.5	1	1.5	2	0.22	1.38	0.43	1.29	0.46
X3	1.5	0.5	1	3	0.33	1.1	0.35	0.89	0.32
Всього:				9	1	3.19	1	2.79	1

4.5 Аналіз рівня якості обраного варіанту

На основі даних таблиці 4.5 було проведено оцінку потенційних абсолютних значень параметрів проєкту. Для порівняння розглянуто альтернативну конфігурацію, у якій замість Express використовується фреймворк Nest. Таким чином, були сформовані наступні варіації:

$$F_1a - F_2a - F_3b - F_4a$$

$$F_1a - F_2b - F_3b - F_4a$$

Для обчислення коефіцієнта технічного рівня було застосовано формулу (4.11):

$$K_K(j) = \sum_{i=1}^n K_{vi,j} B_{i,j} . \quad (4.11)$$

Розраховані значення показників технічного рівня наведено в таблиці 4.6.

Таблиця 4.6 Розрахунок показників рівня якості

Основні функції	Варіант реалізації функції	Параметр	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	A	X1	100	10	0.22	2.2
F2	A	X2	5500	9	0.46	4.14
	B	X2	7000	6	0.46	2.76
F3	B	X1	100	10	0.22	2.2
F4	A	X3	2	8	0.32	2.56

На основі даних з таблиці 4.6 та за формулою (4.11) було здійснено розрахунок рівня якості для обох варіантів реалізації.

$$K_1 = 2.2 + 4.14 + 2.2 + 2.56 = 11.1$$

$$K_2 = 2.2 + 2.76 + 2.2 + 2.56 = 9.72$$

Варіант із використанням Express має більший рівень якості, отже є кращим варіантом.

4.6 Економічний аналіз варіантів розробки програмного продукту

З метою обчислення загальної трудомісткості розробки проєкту було використано формулу (4.12).

Загальна трудомісткість обчислюється як:

$$T_0 = T_P \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТМ}, \quad (4.12)$$

Було визначено початкові дані для розробки програмного продукту, що дозволило отримати поправочні коефіцієнти для подальшого розрахунку загальної трудомісткості.

$$T_P = 80 \text{ людино-днів};$$

$$K_{\Pi} = 1.8;$$

$$K_{СК} = 1;$$

$$K_M = 1;$$

$$K_{СТ} = 0.8;$$

$$K_{СТМ} = 1.$$

У результаті обчислень за початковим рівнянням було отримано таке значення:

$$T_1 = 80 \cdot 1.8 \cdot 0.8 = 115.2 \text{ людино-днів} = 115.2 \cdot 8 = 921.6 \text{ людино-годин.}$$

У розробці задіяний один спеціаліст у сфері веб-розробки з місячним окладом 40000 грн. На основі цього було розраховано середню погодинну заробітну плату:

$$C_{\text{ч}} = \frac{20000}{21 \cdot 8} = 119 \text{ (грн)}$$

Звідси, розраховано заробітну плату за формулою (4.13):

$$C_{\text{зп}} = C_{\text{ч}} \cdot T_i \cdot K_{\text{д}}, \quad (4.13)$$

де $C_{\text{ч}}$ – величина погодинної оплати праці програміста;
 T_i – трудомісткість відповідного завдання;
 $K_{\text{д}}$ – норматив, який враховує додаткову заробітну плату.

Для обраного варіанту реалізації було отримано таке значення:

$$C_{\text{зп}} = 119 \cdot 115.2 \cdot 4 \cdot 8 \cdot 1.2 = 526417.92 \text{ (грн)}$$

З урахування податків 22%:

$$C_{\text{вїд}} = C_{\text{зп}} \cdot 0.22 = 526417.92 \cdot 0.22 = 115811.94 \text{ (грн)}$$

Наступним етапом став розрахунок витрат на одну машино-годину. Оскільки над проєктом працює один розробник, використовується одна робоча машина з коефіцієнтом зайнятості 0.2.

$$C_{\Gamma} = 12 \cdot M \cdot K_3 = 12 \cdot 20000 \cdot 0.2 = 48000 \text{ (грн)}$$

Враховано додаткову заробітню плату:

$$C_{3П} = C_{\Gamma} \cdot (1 + K_3) = 48000 \cdot (1 + 0.2) = 57600 \text{ (грн)}$$

Врахування податків:

$$C_{ВІД} = C_{3П} \cdot 0.22 = 57600 \cdot 0.22 = 12672 \text{ (грн)}$$

Були розраховані амортизаційні відрахування за умови вартості техніки 20 000 грн та річної норми амортизації 25%. Додатково враховано витрати на перевезення й монтаж обладнання.

$$C_A = K_{TM} \cdot K_A \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 10000 = 2875 \text{ (грн)}$$

Витрати на ремонт та обслуговування враховано, виходячи з 5% від загальної суми:

$$C_P = K_{TM} \cdot Ц_{ПР} \cdot K_P = 1.15 \cdot 10000 \cdot 0.05 = 575 \text{ (грн)}$$

Розраховано ефективний годинний фонд часу роботи комп'ютера за рік:

$$\begin{aligned} T_{\text{ЕФ}} &= (D_K - D_B - D_C - D_P) \cdot t_3 \cdot K_B = (365 - 104 - 10 - 16) \cdot 8 \cdot 0.8 = \\ &= 1504 \text{ (години)} \end{aligned}$$

- де D_K – кількість днів у році;
 D_B – кількість вихідних днів у році;
 D_C – кількість свят у році;
 D_P – кількість днів ремонту;
 t – кількість робочих годин в день;
 K_B – коефіцієнт використання ПК протягом робочого часу.

Витрати на оплату електроенергії розраховуємо за формулою:

$$C_{\text{ЕЛ}} = T_{\text{ЕФ}} \cdot N_C \cdot K_3 \cdot C_{\text{ЕН}} = 1504 \cdot 0,2 \cdot 0,8 \cdot 9,43 = 2269,24 \text{ (грн)}$$

- де N_C – середньо-споживча потужність приладу;
 K_3 – коефіцієнтом зайнятості приладу;
 $C_{\text{ЕН}}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$C_H = C_{\text{ГР}} \cdot 0,67 = 10000 \cdot 0,67 = 6700 \text{ (грн)}$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{\text{ЗАГ}} = C_{\text{ЗП}} + C_{\text{ВІД}} + C_A + C_{\text{ЕЛ}} + C_H = 57600 + 12672 + 2875 + 2269,24 + 6700 = 82116,24 \text{ (грн)}$$

Вартість однієї машино-години буде рівною:

$$C_{\text{МГ}} = C_{\text{ЗАГ}} / T_{\text{ЕФ}} = 82116,24 / 1504 = 54,6 \text{ (грн/год)}$$

Розраховано витрати на оплату машинного часу:

$$C_M = C_{\text{МГ}} \cdot T = 54,6 \cdot 921,6 = 50319,36 \text{ (грн)}$$

Накладні витрати:

$$C_H = C_{3П} \cdot 0.4 = 526417,92 \cdot 0.4 = 210567,17 \text{ (грн)}$$

Отже, загальна вартість веб-застосунку:

$$C_{ПП} = C_{3П} + C_{ВІД} + C_M + C_H = 526417,92 + 12672 + 50319,36 + 210567,17 = 799976,45 \text{ (грн)}$$

4.7 Вибір найкращої реалізації техніко-економічного рівня

Визначимо коефіцієнт техніко-економічного рівня за допомогою формули:

$$K_{TEPj} = K_{Kj} / C_{Фj}, \quad (4.14)$$

$$K_{TEP1} = 11,1/799976,45 = 1.388 \cdot 10^{-5}$$

$$K_{TEP2} = 9,72/799976,45 = 1.215 \cdot 10^{-5}$$

Аналіз розрахунків показує, що коефіцієнт техніко-економічного рівня для першого варіанта перевищує показник альтернативної реалізації на базі NestJS. Це пояснюється тим, що, попри подібні витрати на розробку, стек технологій, до якого входять React, Express, PostgreSQL та OpenAI API, забезпечує вищу ефективність, більшу гнучкість і простішу реалізацію. Усе це сприяє підвищенню загального технічного рівня проєкту.

Таким чином, за результатами функціонально-вартісного аналізу, оптимальним варіантом реалізації веб-застосунку для підсумування повідомлень у месенджері із використанням штучного інтелекту є варіант

F1a–F2a–F3в–F4a. Він забезпечує найкраще співвідношення між функціональністю, зручністю підтримки та витратами на розробку. Обраний стек технологій дозволяє реалізувати сучасний, швидкий і масштабований продукт, здатний ефективно обробляти повідомлення та формувати стислий зміст за допомогою GPT-моделі.

4.8 Висновки до четвертого розділу

У даному розділі було розглянуто можливі варіанти реалізації веб-застосунку для месенджера, який використовує штучний інтелект для підсумування вмісту повідомлень. З метою обґрунтування вибору оптимального технічного рішення було проведено функціонально-вартісний аналіз, а також визначено вагомість характеристик за допомогою експертних оцінок. Вартість розробки оцінювалася за умови участі одного фахівця у сфері веб-розробки з місячним окладом 40 000 гривень.

Порівнювалися два варіанти реалізації програмного продукту, які відрізнялись стеком технологій на бекенді: перший варіант базувався на Express, а другий – на NestJS. Результати аналізу показали, що обидва варіанти мають приблизно однакові витрати на розробку, однак реалізація на базі React, Express, PostgreSQL та OpenAI API (варіант F1a–F2a–F3в–F4a) забезпечує вищий коефіцієнт техніко-економічного рівня.

Таким чином, вибір Express як основа для бекенд-частини проекту є обґрунтованим, оскільки він забезпечує оптимальний баланс між функціональністю, продуктивністю та витратами на реалізацію і підтримку.

ВИСНОВКИ

У межах дипломної роботи було реалізовано повнофункціональний веб-застосунок для обміну повідомленнями, доповнений інтеграцією штучного інтелекту з метою підвищення зручності користування. Основною особливістю розробленої системи є можливість стислого викладу змісту окремих повідомлень за допомогою моделі GPT-4o від OpenAI. Це дозволяє користувачам швидко ознайомлюватися з великою за обсягом інформацією без необхідності читати все повідомлення повністю. Після виконання запиту до моделі, користувачу повертається підсумований зміст повідомлення, а за бажанням він може повернутися до перегляду оригінального повідомлення.

Окрім функції підсумування, у застосунку реалізовано базовий набір можливостей сучасного месенджера: обмін текстовими повідомленнями, передавання файлів, редагування й видалення власних повідомлень. Усі зміни зберігаються у базі даних, що забезпечує цілісність історії листування та зручність у подальшому використанні.

Клієнтська частина застосунку створена з використанням React і Tailwind CSS, що дозволило реалізувати адаптивний і привабливий інтерфейс. На боці сервера використано Express, а для взаємодії з базою даних – PostgreSQL у поєднанні з ORM Prisma. Це забезпечило високу продуктивність, масштабованість і зручність у підтримці та розвитку системи.

У результаті виконання дипломної роботи поставлену мету було досягнуто. Веб-застосунок стабільно функціонує, має зручний інтерфейс і дозволяє користувачам ефективно взаємодіяти з великими обсягами текстової інформації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Перекладаємо слово месенджер. URL: <https://slovotvir.org.ua/words/mesendzher/> (дата звернення: 02.05.2025).
2. Що таке IRC у мережі? URL: <https://svoboda.home.cx.ua/vzaiemodiya/shho-take-irc-u-merezhi.html> (дата звернення: 02.05.2025).
3. Що таке HTML? URL: <https://bdut.co.ua/pro-nas/scho-take-html/> (дата звернення: 08.05.2025).
4. Що таке CSS? URL: <https://css.tutorial.in.ua/what-is-css/> (дата звернення: 08.05.2025).
5. Що таке JavaScript та де він використовується URL: <https://dzudzylo.com/javascript/shho-take-javascript-ta-de-vin-vykorystovuyetsya.html> (дата звернення: 08.05.2025).
6. Хто такий Back-end розробник? URL: <https://javarush.com/ua/groups/posts/uk.2830.byс-nevidimogo-frontu-khto-takiy-backend-rozrobnik-ta-jak-nim-stati> (дата звернення: 08.05.2025).
7. Ткачук І. Розробка на Java – з чого почати знайомство з мовою. URL: <https://dou.ua/lenta/articles/how-to-learn-java> (дата звернення: 08.05.2025).
8. Ключові відмінності між Laravel і Symfony. URL: <https://asabix.com.ua/key-differences-between-laravel-and-sym> (дата звернення: 08.05.2025).
9. Асинхронне програмування: коли, навіщо та як його використовувати. URL: <https://itproger.com.ua/news/asinhronnoe-programmirovanie-kogda-zachem-i-kak-ego-ispolzovat> (дата звернення: 08.05.2025).
10. Реляційні бази даних. URL: <https://data-b-i.com/uk/article/yak-organyzoveny-relyatsiyni-bazy-danykh.html> (дата звернення: 08.05.2025).

11. Структура реляційних баз даних. URL: <https://foxminded.ua/reliatsiini-bazy-danykh/> (дата звернення: 08.05.2025).
12. Ткачук І. Типи баз даних: особливості, відмінності та приклади. URL: <https://dou.ua/lenta/articles/types-of-databases/> (дата звернення: 08.05.2025).
13. Що таке база даних. URL: <https://hostiq.ua/wiki/ukr/database> (дата звернення: 08.05.2025).
14. Prisma ORM: Спрощення управління базами даних у Node.js. URL: <https://javascript.org.ua/prisma-orm-sproshhennya-upravlinnya-bazami-danih-u-node-js/> (дата звернення: 08.05.2025).
15. NeuralWriter Summarize Tool. Інструмент підсумування тексту. URL: <https://neuralwriter.com/uk/summarize-tool> (дата звернення: 08.05.2025).

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

```

#client/ Body.tsx
import SendMessageBlock from "../SendMessageBlock";
import ReceiveMessageBlock from "../ReceiveMessageBlock";
import { BodyProps } from "../../types/BodyComponents/BodyProps.interface.ts";
import React, { useEffect, useRef } from "react";

const Body: React.FC<BodyProps> = ({
  messages,
  recipientUser,
  loadMessages,
}) => {
  const messagesEndRef = useRef<HTMLDivElement | null>(null);

  useEffect(() => {
    messagesEndRef.current?.scrollIntoView({ behavior: "smooth" });
  }, [messages]);

  return (
    <div className="flex flex-col h-screen">
      <div className="flex-1 p-4 bg-gray-50">
        {recipientUser && (
          <div className="space-y-4">
            {messages.map((element) => {
              if (
                element.sender_name === localStorage.getItem("user") &&
                element.recipient_name === recipientUser
              ) {
                return (
                  <SendMessageBlock
                    key={element.id}
                    element={element}

```

```

        loadMessages={loadMessages}
      />
    );
  } else if (
    element.sender_name === recipientUser &&
    element.recipient_name === localStorage.getItem("user")
  ) {
    return (
      <ReceiveMessageBlock key={element.id} element={element} />
    );
  } else {
    return null;
  }
}
}
<div ref={messagesEndRef} />
</div>
))
</div>
</div>
);
};

export default Body;
#client/Body/EditInput.tsx
import {EditInputProps} from "../../types/BodyComponents/EditInputProps.interface.ts";
import React from "react";
const EditInput: React.FC<EditInputProps> = ({editedContent,setEditedContent}) => {
  return (
    <input
      type="text"
      value={editedContent}
      onChange={(e) => setEditedContent(e.target.value)}
      className="w-full p-2 border rounded-md"
    />
  );
};

```

```
};
```

```
export default EditInput;
```

```
#client/Body/FilesContainer.tsx
```

```
import React from 'react';
```

```
import {FilesContainerProps} from
```

```
"../../types/BodyComponents/FilesContainerProps.interface.ts";
```

```
const FilesContainer: React.FC<FilesContainerProps> = ({element}) => {
```

```
  return (
```

```
    <div>
```

```
      {element.files && element.files.length > 0 && (
```

```
        <div className="space-y-4">
```

```
          {element.files.map((file, index) => (
```

```
            <div
```

```
              key={index}
```

```
              className="flex flex-col items-start bg-gray-50 p-3 rounded-lg shadow-
```

```
md border border-gray-200 w-full"
```

```
            >
```

```
              {file.mimetype.startsWith("image/") ? (
```

```
                <a
```

```
                  href={`http://localhost:5000/uploads/${file.filename}`}
```

```
                  target="_blank"
```

```
                  rel="noopener noreferrer"
```

```
                >
```

```
                  <img
```

```
                    src={`http://localhost:5000/uploads/${file.filename}`}
```

```
                    alt={file.filename}
```

```
                    className="w-40 h-40 object-cover rounded-lg border border-
```

```
gray-300 shadow-sm"
```

```
                  />
```

```
                </a>
```

```
              ) : (
```

```
                <div className="flex flex-col space-y-1">
```

```
                  <a
```

```

        href={`http://localhost:5000/uploads/${file.filename}`}
        target="_blank"
        rel="noopener noreferrer"
        className="text-blue-500 underline hover:text-blue-700 font-
medium break-all"
      >
        {file.filename}
      </a>
      <span className="text-sm text-gray-500">
        Type: {file.mimetype}
      </span>
    </div>
  )}
</div>
)))
</div>
)}
</div>
);
};

```

```

export default FilesContainer;
#client/Body/HeaderUp.tsx
import {HeaderUpProps} from "../../types/BodyComponents/HeaderUpProps.ts";
import React from "react";
const HeaderUp: React.FC<HeaderUpProps> = ({handleLeave}) => {
  return (
    <header className="sticky top-0 bg-gray-200 p-4 shadow-md z-10">
      <div className="flex justify-between items-center">
        <div className="flex-1"></div>
        <button className="bg-red-500 text-white px-4 py-2 rounded hover:bg-red-600
transition"
          onClick={handleLeave}>
          Exit chat
        </button>
      </div>
    </header>
  );
};

```

```

        </div>
      </header>
    );
  };

export default HeaderUp;
#client/Body/ReceiveMessageBlock.tsx
import FilesContainer from "../FilesContainer";
import React, { useState, useRef } from "react";
import { ReceiveMessageBlockProps } from
"../../types/BodyComponents/ReceiveMessageBlock.interface.ts";
import magicStick from "../../public/magic-stick.svg";
import { summarizeMessage } from "../../services/message-service.ts";

const ReceiveMessageBlock: React.FC<ReceiveMessageBlockProps> = ({
  element,
}) => {
  const [displayedMessage, setDisplayedMessage] = useState<string>(
    element.content
  );
  const [isLoading, setIsLoading] = useState<boolean>(false);
  const originalMessageRef = useRef<string>(element.content);
  const [isSummarized, setIsSummarized] = useState<boolean>(false);

  const handleSummarize = async () => {
    try {
      if (element.content.length < 10) {
        alert("Too short message to summarize");
        return;
      }
      setIsLoading(true);
      const result = await summarizeMessage(element.content);
      setDisplayedMessage(result);
      setIsSummarized(true);
      setIsLoading(false);
    }
  };

```

```

    } catch (error) {
      console.error("Помилка при підсумовуванні:", error);
      setDisplayedMessage("Не вдалося підсумувати повідомлення.");
      setIsSummarized(true);
      setIsLoading(false);
    }
  };

```

```

const handleBackToOriginal = () => {
  setDisplayedMessage(originalMessageRef.current);
  setIsSummarized(false);
};

```

```

return (
  <div className="flex flex-col items-start group" key={element.id}>
    <p className="text-sm font-semibold text-gray-700">
      {element.sender_name}
    </p>
    <div className="flex items-center bg-blue-100 p-3 rounded-md shadow-sm max-w-xs
relative mb-1">
      {!isLoading &&
        (isLoading ? (
          <div
            className="text-gray-700 text-s px-3 py-1 rounded-lg bg-purple-200
            transition-all duration-200 ease-in-out absolute right-[-130px] animate-pulse"
          >
            Loading...
          </div>
        ) : (
          <button
            onClick={handleSummarize}
            className="text-white text-s px-3 py-1 rounded-lg bg-purple-600 hover:bg-purple-
300
            transition-all duration-200 ease-in-out opacity-0 group-hover:opacity-100
            absolute right-[-180px]"

```

```

    >
    Summarize text
    <img
      className="w-6 h-6 inline-block ml-1"
      src={magicStick}
      alt="Logo"
    />
  </button>
)}}
```

{isSummarized && (

```

  <button
    onClick={handleBackToOriginal}
    className="text-white text-s px-3 py-1 rounded-lg bg-gray-600 hover:bg-gray-400
      transition-all duration-200 ease-in-out opacity-0 group-hover:opacity-100
      absolute right-[-180px]"
  >
    Back to original text
  </button>
)}
<p className="text-gray-800">{displayedMessage}</p>
</div>
<FilesContainer element={element} />
</div>
);
};

export default ReceiveMessageBlock;
#client/Body/SaveCancelButton.tsx
import {SaveCancelButtonProps} from
"../../types/BodyComponents/SaveCancelButtonProps.interface.ts";
import React from "react";

const SaveCancelButton: React.FC<SaveCancelButtonProps> =
({saveEdit,setIsEditing,setEditedContent,element}) => {
  return (
```

```

<div>
  <div className="flex space-x-2 mt-2">
    <button
      onClick={saveEdit}
      className="bg-green-500 text-white px-3 py-1 rounded-lg hover:bg-green-600
transition-all"
    >
      Save
    </button>
    <button
      onClick={() => {
        setIsEditing(false);
        setEditedContent(element.content);
      }}
      className="bg-gray-500 text-white px-3 py-1 rounded-lg hover:bg-gray-600
transition-all"
    >
      Cancel
    </button>
  </div>
</div>
);
};

```

```
export default SaveCancelButton;
```

```
#client/Body/HeaderUp.tsx
```

```
#client/Body/SendMessageBlock.tsx
```

```
import React, { useState } from "react";
```

```
import EditInput from "../EditInput";
```

```
import SaveCancelButton from "../SaveCancelButton";
```

```
import {
```

```
  handleDeleteMessage,
```

```
  handleEditMessage,
```

```
} from "../../services/message-service.ts";
```

```

import FilesContainer from "../FilesContainer";
import { SendMessageBlockProps } from
"../../types/BodyComponents/SendMessageBlockProps.interface.ts";

const SendMessageBlock: React.FC<SendMessageBlockProps> = ({
  element,
  loadMessages,
}) => {
  const [isEditing, setIsEditing] = useState<boolean>(false);
  const [editedContent, setEditedContent] = useState<string>(element.content);

  const saveEdit = (): void => {
    handleEditMessage(element.id, editedContent, loadMessages);
    setIsEditing(false);
  };

  return (
    <div className="flex flex-col items-end group">
      <p className="text-sm font-semibold text-gray-700">You</p>
      <div className="flex items-center bg-green-100 p-3 rounded-md shadow-sm max-w-xs
relative mb-1">
        <button
          onClick={() => handleDeleteMessage(element.id, loadMessages)}
          className="text-gray-500 text-s px-3 py-1 rounded-lg bg-gray-200 hover:bg-gray-300
          transition-all duration-200 ease-in-out opacity-0 group-hover:opacity-100
          absolute left-[-75px]"
        >
          Delete
        </button>

        {!isEditing && (
          <button
            onClick={() => setIsEditing(true)}
            className="text-gray-500 text-s px-3 py-1 rounded-lg bg-gray-200 hover:bg-gray-300
            transition-all duration-200 ease-in-out opacity-0 group-hover:opacity-100

```

```

        absolute left-[-130px]"
    >
        Edit
    </button>
  )}

  {!isEditing ? (
    <p className="text-gray-800 break-words">{element.content}</p>
  ) : (
    <EditInput
      editedContent={editedContent}
      setEditedContent={setEditedContent}
    />
  )}
  {isEditing && (
    <SaveCancelButtons
      saveEdit={saveEdit}
      setIsEditing={setIsEditing}
      setEditedContent={setEditedContent}
      element={element}
    />
  )}
</div>
<FilesContainer element={element} />
</div>
);
};

```

```

export default SendMessageBlock;
#client/Chat/ChatPage.tsx
import Sidebar from "../Sidebar";
import Body from "../Body";
import MessageBlock from "../Message-block";
import React, { useEffect, useState } from "react";
import axios from "axios";

```

```

import { Message } from "../../types/Message.interface.ts";
import HeaderUp from "../Body/HeaderUp";
import { useNavigate } from "react-router-dom";

const ChatPage: React.FC = () => {
  const [messages, setMessages] = useState<Message[]>([]);
  const [recipientUser, setRecipientUser] = useState<string>("");

  const loadMessages = async (): Promise<void> => {
    const token = localStorage.getItem("token") || "";
    const response = await axios.get<Message[]>(
      "http://localhost:5000/api/messages",
      {
        headers: {
          "Content-Type": "application/json",
          "x-access-token": token,
        },
      }
    );
    setMessages(response.data);
  };

  useEffect(() => {
    loadMessages();
  }, []);

  const navigate = useNavigate();

  const handleLeave = (): void => {
    navigate("/");
  };

  return (
    <div className="flex h-screen bg-gray-100">
      <Sidebar setRecipientUser={setRecipientUser} />

```

```

<main className="flex-1 flex flex-col ">
  <HeaderUp handleLeave={handleLeave} />
  {recipientUser ? (
    ◇
    <div className="flex-1 overflow-y-auto">
      <Body
        messages={messages}
        recipientUser={recipientUser}
        loadMessages={loadMessages}
      />
    </div>
    <MessageBlock
      recipientUser={recipientUser}
      loadMessages={loadMessages}
    />
  </>
  ) : (
    <div className="flex-1 overflow-y-auto flex items-center justify-center bg-gray-50">
      <div className="text-center px-6">
        <h2 className="text-xl font-semibold text-gray-700 mb-2">
          Welcome to Messenger
        </h2>
        <p className="text-gray-500">
          This messenger can summarize text. <br /> Select a chat to start
          messaging.
        </p>
      </div>
    </div>
  )}
</main>
</div>
);
};

```

```
export default ChatPage;
```

```
#client/Login/Login.tsx
```

```
import React, { useState } from "react";
```

```
import { useNavigate } from "react-router-dom";
```

```
import axios from "axios";
```

```
import { FormData } from "../../types/FormData.interface.ts";
```

```
function Login() {
```

```
  const [formData, setFormData] = useState<FormData>({
    name: "",
    password: "",
  });
```

```
  const navigate = useNavigate();
```

```
  const handleChange = (e: React.ChangeEvent<HTMLInputElement>):void => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

```

```
  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>): Promise<void>
=> {
    e.preventDefault();
    try {
      const response = await axios.post("http://localhost:5000/api/auth/login", formData);
      localStorage.setItem("token", response.data.token);
      localStorage.setItem("user", response.data.name);
      alert("Login successful!");
      navigate("/chat");
    } catch (err) {
      alert("Invalid username or password");
      console.error(err);
    }
  };

```

```
  return (
```

```
    <div className="flex justify-center items-center h-screen bg-gray-100">
```

```

<form
  onSubmit={handleSubmit}
  className="w-full max-w-md bg-white rounded-lg shadow-lg p-6"
>
  <h2 className="text-2xl font-bold text-center mb-6">Enter to Chat</h2>
  <div className="mb-4">
    <label
      htmlFor="name"
      className="block text-gray-700 font-medium mb-2"
    >
      Username
    </label>
    <input
      type="text"
      id="name"
      name="name"
      value={formData.name}
      onChange={handleChange}
      className="w-full px-4 py-2 border rounded-lg focus:outline-none focus:ring
focus:ring-blue-300"
      placeholder="Enter your username"
      required
    />
  </div>
  <div className="mb-4">
    <label
      htmlFor="password"
      className="block text-gray-700 font-medium mb-2"
    >
      Password
    </label>
    <input
      type="password"
      id="password"
      name="password"

```

```

        value={formData.password}
        onChange={handleChange}
        className="w-full px-4 py-2 border rounded-lg focus:outline-none focus:ring
focus:ring-blue-300"
        placeholder="Enter your password"
        required
      />
    </div>
    <button
      type="submit"
      className="w-full bg-blue-500 text-white py-2 rounded-lg hover:bg-blue-600
transition duration-300"
    >
      Sign in
    </button>
    <p className="text-center text-gray-600 mt-4">
      Don't have an account? {" "}
      <span
        className="text-blue-500 cursor-pointer hover:underline"
        onClick={() => navigate("/register")}
      >
        Sign up
      </span>
    </p>
  </form>
</div>
);
}

```

```

export default Login;
#client/MessageBlock/MessageBlock.tsx
import React, {useState} from "react";
import {handleSend} from "../../services/message-service.ts";
import {MessageBlockProps} from "../../types/MessageBlockProps.interface.ts";

```

```

const MessageBlock: React.FC<MessageBlockProps> = ({recipientUser,loadMessages}) => {

  const [message,setMessage] = useState<string>("")
  const [files, setFiles] = useState<File[]>([]);
  const handleFileChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    if (event.target.files) {
      const selectedFiles = Array.from(event.target.files);
      setFiles((prevFiles) => [...prevFiles, ...selectedFiles]);
    }
  };
  const handleFileRemove = (index: number):void => {
    setFiles((prevFiles) => prevFiles.filter((_, i) => i !== index));
  };

  return (
    <div className="bg-gray-100 p-3 shadow-md w-full rounded-lg flex flex-col space-y-2">
      {files.length > 0 && (
        <div
          className="bg-white border border-gray-300 rounded-md shadow-md p-2 max-h-32 w-full overflow-y-auto text-sm">
            <ul>
              {files.map((file, index) => (
                <li
                  key={index}
                  className="flex justify-between items-center mb-1 last:mb-0">
                    <span className="truncate w-3/4">{file.name}</span>
                    <button
                      type="button"
                      onClick={() => handleFileRemove(index)}
                      className="text-xs">
                        ✕
                    </button>

```

```

        </li>
      )}
    </ul>
  </div>
)}

<div className="flex items-center space-x-2">
  <input
    type="text"
    placeholder="Type your message..."
    className="flex-grow p-2 border border-gray-300 rounded-md focus:outline-none
focus:ring-2 focus:ring-blue-500 text-sm"
    value={message}
    onChange={(e) => setMessage(e.target.value)}
  />

  <div className="relative">
    <label
      htmlFor="file-upload"
      className="flex items-center justify-center bg-gray-200 text-gray-700 px-4 py-
2 rounded-md border border-gray-300 cursor-pointer hover:bg-gray-300 text-sm transition"
    >
      Choose File
    </label>
    <input
      id="file-upload"
      type="file"
      multiple
      className="hidden"
      onChange={handleFileChange}
    />
  </div>

  {message &&
  <button

```

```

        type="submit"
        onClick={ (event) => handleSend(event,message,recipientUser,setMessage,loadMessages,files,setFiles)}
        className="bg-blue-500 text-white px-4 py-2 rounded-md hover:bg-blue-600 text-sm transition"
      >
        Send
      </button>
    }
  </div>
</div>

);
};

```

```

export default MessageBlock;
#client/Register/Register.tsx
import React, {useState} from "react";
import {useNavigate} from "react-router-dom";
import axios from "axios";
import {FormData} from "../../types/FormData.interface.ts";

```

```

const Register = () => {

  const [formData, setFormData] = useState<FormData>({
    name: "",
    password: "",
  });

  const navigate = useNavigate();

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>):void => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

```

```

const handleRegister = async (e: React.FormEvent<HTMLFormElement>):Promise<void>
=> {
  e.preventDefault();
  try {
    const response = await axios.post("http://localhost:5000/api/auth/register", formData);
    localStorage.setItem("token", response.data.token);
    localStorage.setItem("user",response.data.name)
    alert("Sign up is successful!");
    navigate("/chat");
  } catch (err) {
    alert("Invalid username or password");
    console.error(err);
  }
};

return(
  <div>
    <div className="flex justify-center items-center h-screen bg-gray-100">
      <form
        onSubmit={handleRegister}
        className="w-full max-w-md bg-white rounded-lg shadow-lg p-6"
      >
        <h2 className="text-2xl font-bold text-center mb-6">Sign up</h2>
        <div className="mb-4">
          <label
            htmlFor="name"
            className="block text-gray-700 font-medium mb-2"
          >
            Username
          </label>
          <input
            type="text"
            id="name"
            name="name"
            value={formData.name}

```

```

        onChange={handleChange}
        className="w-full px-4 py-2 border rounded-lg focus:outline-none
focus:ring focus:ring-blue-300"
        placeholder="Enter your username"
        required
      />
    </div>
    <div className="mb-4">
      <label
        htmlFor="password"
        className="block text-gray-700 font-medium mb-2"
      >
        Password
      </label>
      <input
        type="password"
        id="password"
        name="password"
        value={formData.password}
        onChange={handleChange}
        className="w-full px-4 py-2 border rounded-lg focus:outline-none
focus:ring focus:ring-blue-300"
        placeholder="Enter your password"
        required
      />
    </div>
    <button
      type="submit"
      className="w-full bg-blue-500 text-white py-2 rounded-lg hover:bg-blue-600
transition duration-300"
    >
      Sign up
    </button>
    <p className="text-center text-gray-600 mt-4">
      Already have an account? {" "}

```

```

        <span
            className="text-blue-500 cursor-pointer hover:underline"
            onClick={() => navigate("/")}>
        >
        Sign in
    </span>
</p>
</form>
</div>
</div>
)
}

export default Register
#client/Sidebar/Sidebar.tsx
import React, {useEffect, useState} from "react";
import axios from "axios";
import {SidebarProps} from "../../types/SidebarProps.interface.ts";
import {User} from "../../types/User.interface.ts";

const Sidebar: React.FC<SidebarProps> = ({setRecipientUser}) => {
    const [users,setUsers] = useState<User[]>([])

    useEffect(() => {
        const token = localStorage.getItem('token') || "";
        const user = localStorage.getItem("user")

        axios
            .get<User[]>('http://localhost:5000/api/users', {
                headers: {
                    'Content-Type': 'application/json',
                    'x-access-token': token,
                },
            })
            .then((response) => {
                setUsers(response.data.filter(data=>data.name !== user));
            })
    })
}

```

```

    })
  }, []);

const handleSetRecipientUser = (user: string): void =>{
  setRecipientUser(user)
}

return (
  <div className="w-64 h-screen bg-gray-100 p-4 shadow-md">
    <h4 className="text-xl font-bold mb-4">Users</h4>
    <ul className="space-y-2">
      {users.map(el=>{
        return <li className="p-2 rounded hover:bg-gray-200 cursor-pointer"
          key={el.id} onClick={() => handleSetRecipientUser(el.name)}
        >
          {el.name}
        </li>
      })}
    </ul>
  </div>
);
};

```

```

export default Sidebar;

#client/services/message-services.ts

import axios from "axios";
import React from "react";

export const handleSend = async (
  e: React.FormEvent,
  message: string,
  recipientUser: string,
  setMessage: (value: string) => void,
  loadMessages: () => void,
  files: File[],

```

```

    setFiles: (files: File[]) => void
  ): Promise<void>=>{
    e.preventDefault()
    const token = localStorage.getItem('token') || "";

    const formData = new FormData();
    const senderName = localStorage.getItem('user') || "";
    formData.append("sender_name",senderName)
    formData.append("content", message)
    formData.append("recipient_name", recipientUser)
    files.forEach((file) => {
      formData.append("files", file);
    });

    if(message.trim() && localStorage.getItem('user')){
      await axios.post(
        'http://localhost:5000/api/messages', formData, {
          headers: {
            'Content-Type': 'multipart/form-data',
            'x-access-token': token,
          },
        }
      );
    }
    setMessage("")
    setFiles([])
    loadMessages();
  }

export const handleDeleteMessage = async (
  id: string,
  loadMessages: () => void
): Promise<void> =>{
  const token = localStorage.getItem('token') || "";
  await axios.delete(

```

```

    `http://localhost:5000/api/messages/${id}`,
    {
      headers: {
        'Content-Type': 'application/json',
        'x-access-token': token,
      },
    }
  );
  loadMessages();
}

export const handleEditMessage = async (
  id: string,
  content: string,
  loadMessages: () => void
): Promise<void>=>{
  const token = localStorage.getItem('token') || "";
  await axios.patch(
    `http://localhost:5000/api/messages/${id}`,
    {
      content: content,
    },
    {
      headers: {
        'Content-Type': 'application/json',
        'x-access-token': token,
      },
    }
  );
  loadMessages();
}

export const summarizeMessage = async(content:string):Promise<string>=>{
  const token = localStorage.getItem('token') || "";
  const message = await axios.post(
    `http://localhost:5000/api/summarize`,

```

```

    {
      content: content,
    },
    {
      headers: {
        'Content-Type': 'application/json',
        'x-access-token': token,
      },
    }
  );
  return message.data
}

#client /App.tsx
import {Route, Routes} from "react-router-dom";
import Login from "../components/Login"
import ChatPage from "../components/Chat";
import Register from "../components/Register";

function App() {

  return (
    <div>
      <Routes>
        <Route path="/" element={<Login/>}/>
        <Route path="/register" element={<Register/>}/>

        <Route path="/chat" element={<ChatPage/>}/>
      </Routes>
    </div>
  )
}

```

```

export default App
#server/controllers/authController.js
require("dotenv").config();
const { PrismaClient } = require("@prisma/client");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");

const prisma = new PrismaClient();

exports.registerUser = async (req,res)=>{
  try {
    const {name,password} = req.body;

    if (!(name && password)){
      res.status(400).send("All input is required");
    }

    const existUser = await prisma.user.findUnique({
      where: {
        name: name
      }
    })
    if(existUser){
      return res.status(409).send("User Already Exist. Please Login");
    }
    const encryptedPassword = await bcrypt.hash(password, 10);

    const user = await prisma.user.create({
      data: {
        name:name,
        password: encryptedPassword
      }
    });

    const token = jwt.sign(
      {user_id: user.id, name},

```

```

    process.env.TOKEN_KEY,
    {
      expiresIn: "2h",
    }
  )

  user.token = token;
  res.status(201).json(user)

} catch (error) {
  console.log(error)
}
}

exports.loginUser = async (req,res)=>{
  try {
    const {name,password} = req.body;
    if (!(name && password)){
      res.status(400).send("All input is required");
    }
    const user = await prisma.user.findUnique({
      where: {
        name: name
      }
    })
    if (user && (await bcrypt.compare(password, user.password ))) {
      const token = jwt.sign(
        {user_id: user.id, name},
        process.env.TOKEN_KEY,
        {
          expiresIn: "2h",
        }
      );
      user.token = token;

```

```

        res.status(200).json(user);
    }
    else res.status(400).send("Invalid Credentials");
} catch (error) {
    console.log(error)
}
}
}
#server/controllers/messagesController.js
const { PrismaClient } = require("@prisma/client");
const fs = require("fs").promises;
const prisma = new PrismaClient();

exports.getAllMessages = async (req, res) => {
    try {
        const messages = await prisma.message.findMany({
            include: { files: true },
            orderBy: {
                created_at: "asc",
            },
        })
        res.status(200).json(messages);
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "Failed to fetch messages" });
    }
};

exports.postMessage = async (req,res)=>{
    try {
        const {sender_name,content,recipient_name} = req.body;
        const files = req.files.map((file) => ({
            filename: file.filename,
            path: file.path,
            size: file.size,
            mimetype: file.mimetype,

```

```

    ));

    const message = await prisma.message.create({
      data: {
        sender_name,
        content,
        recipient_name,
        files: {
          create: files,
        },
      },
      include: {
        files: true,
      },
    });

    res.status(201).json(message)

  } catch (error) {
    console.log(error)
  }
}

exports.deleteMessage = async (req,res)=>{
  try{
    const messageId = req.params.id

    const existingMessage = await prisma.message.findUnique({
      where: { id: messageId },
      include: { files: true },
    });

    if (!existingMessage) {
      return res.status(404).json({ error: "Message not found" });
    }

    for (const file of existingMessage.files) {

```

```

    try {
      await fs.unlink(file.path);
    } catch (err) {
      console.error(`Failed to delete file: ${file.path}`, err);
    }
  }
}

const deletedMessage = await prisma.$transaction([
  prisma.file.deleteMany({
    where: {
      messageId: messageId, // Видаляємо всі файли, пов'язані з повідомленням
    },
  }),
  prisma.message.delete({
    where: {
      id: messageId,
    },
  }),
]);

res.status(204).json(deletedMessage)

} catch (err){
  console.log(`Fail to delete message ${err}`)
}
}

exports.updateMessage = async (req,res) =>{
  try{
    const messageId = req.params.id
    const {content} = req.body;

    const updatedMessage = await prisma.message.update({
      where: {
        id: messageId,
      },
    },

```

```

        data: {
            content,
        },
    });

    res.status(200).json(updatedMessage);

} catch (err){
    console.log(`Fail to update message ${err}`)
}
}

#server/controllers/summarizeController.js
const { OpenAI } = require("openai");

const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

exports.summarizeMessage = async (req, res) => {
    try {
        const { content } = req.body;
        const response = await openai.chat.completions.create({
            model: "gpt-4o",
            messages: [
                {
                    role: "user",
                    content: `Summrize this text in very short format:\n${content}`,
                },
            ],
        });

        const summary = response.choices[0].message.content.trim();

        res.status(200).json(summary);
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "Failed to fetch req" });
    }
}

```

```

    }
  };
#server/controllers/usersController.js
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

exports.getAllUsers = async (req, res) => {
  try {
    const users = await prisma.user.findMany();

    res.status(200).json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Failed to fetch users" });
  }
};
#server/middlewares/auth.js
const jwt = require("jsonwebtoken");

const auth = (req, res, next) => {
  const token =
    req.body.token || req.query.token || req.headers["x-access-token"];

  if (!token) {
    return res.status(403).send("A token is required for authentication");
  }

  try {
    const decoded = jwt.verify(token, process.env.TOKEN_KEY);
    req.user = decoded;
  } catch (err) {
    return res.status(401).send("Invalid token");
  }

  return next();
};

```

```
}
```

```
module.exports = auth;
#server/middlewares/multer-config.js
const multer = require("multer");
const path = require("path");

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, path.join(__dirname, "../uploads"));
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + "-" + file.originalname;
    cb(null, uniqueSuffix);
  },
});
```

```
const upload = multer({
  storage,
  limits: {
    fileSize: 5 * 1024 * 1024,
  },
});
```

```
module.exports = upload;
#server/prisma/schema.prisma
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema
```

```
// Looking for ways to speed up your queries, or scale easily with your serverless or edge
functions?
```

```
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init
```

```
generator client {
  provider = "prisma-client-js"
```

```

}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id Int @id @default(autoincrement())

  name   String @unique
  password String
  token  String?
}

model Message {
  id String @id @default(uuid())

  sender_name String @default("Unknown Sender")
  content      String
  recipient_name String @default("Unknown Recipient")
  created_at   DateTime @default(now())
  files        File[]
}

model File {
  id      String @id @default(uuid())
  filename String
  path    String
  size    Int
  mimetype String
  Message Message? @relation(fields: [messageId], references: [id])
  messageId String?
}

#server/routes/authRouter.js

```

```
const express = require("express");
const authController = require("../controllers/authController")
const router = express.Router();
```

```
router.route("/register").post(authController.registerUser)
```

```
router.route("/login").post(authController.loginUser)
```

```
module.exports = router;
```

```
#server/routes/messagesRouter.js
```

```
const express = require("express");
const messagesController = require('../controllers/messagesController')
const auth = require("../middlewares/auth")
const upload = require("../middlewares/multer-config")
const router = express.Router();
```

```
router
```

```
  .route("/")
  .get(auth,messagesController.getAllMessages)
  .post(auth,upload.array("files", 10),messagesController.postMessage)
```

```
router.route("/:id")
```

```
  .delete(auth,messagesController.deleteMessage)
  .patch(auth,messagesController.updateMessage)
```

```
module.exports = router;
```

```
#server/routes/summarizeRouter.js
```

```
const express = require("express");
const summarizeController = require("../controllers/summarizeController")
const auth = require("../middlewares/auth")
const router = express.Router();
```

```
router
```

```
  .route("/")
  .post(auth,summarizeController.summarizeMessage)
```

```

module.exports = router;

#server/routes/usersRouter.js

const express = require("express");
const usersController = require("../controllers/usersController")
const auth = require("../middlewares/auth")
const router = express.Router();

```

```

router
  .route("/")
  .get(auth,usersController.getAllUsers)

```

```

module.exports = router;

```

```

#server/index.js

const express = require("express")
const app = express()
const cors = require("cors")
const usersRouter = require("../routes/usersRouter")
const authRouter = require("../routes/authRouter")
const messagesRouter = require("../routes/messagesRouter")
const summarizeRouter = require("../routes/summarizeRouter")
const path = require("path");
const PORT = 5000

app.use(cors())
app.use(express.json());
app.use("/api/users", usersRouter);
app.use("/api/auth",authRouter)
app.use("/api/messages",messagesRouter)
app.use("/api/summarize", summarizeRouter)
app.use("/uploads", express.static(path.join(__dirname, "uploads")));

app.listen(PORT,()=>{
  console.log("app started")
})

```

ДОДАТОК Б ГРАФІЧНІ МАТЕРІАЛИ

Месенджер з використанням технологій штучного інтелекту для підсумування вмісту повідомлень

Виконав студент 4 курсу, групи КА-13

Грабовецький Нікіта Богданович

Керівник: доцент, к. т. н., Савченко Ілля Олександрович

Постановка задачі

- створити веб-застосунок з інтеграцією ШІ для підсумування вмісту повідомлень;
- реалізувати можливість обміну повідомленнями між користувачами;
- додати функції редагування та видалення власних повідомлень;
- додати можливість обміну файлами між користувачами;
- забезпечити авторизацію та автентифікацію користувачів.

Актуальність роботи

- актуальність роботи зумовлена щоденним отриманням великої кількості повідомлень, що ускладнює сприйняття важливої інформації;
- у професійному та особистому листуванні зростає потреба швидко орієнтуватися в змісті без прочитання всього тексту;
- більшість популярних месенджерів не мають функції підсумування повідомлень, що робить цю розробку інноваційною.

3

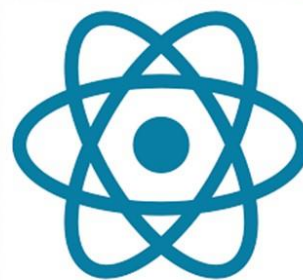
Мета роботи — розробити веб-застосунок для обміну повідомленнями з інтеграцією моделі штучного інтелекту GPT-4o для підсумування текстового вмісту повідомлень з метою підвищення ефективності спілкування та зручності користування.

Об'єкт дослідження — процеси розробки та функціонування веб-месенджера з можливістю інтелектуального аналізу вмісту повідомлень.

Предмет дослідження — методи та засоби інтеграції штучного інтелекту у веб-застосунки для автоматизованої обробки й узагальнення текстових повідомлень.

4

Обрані технології розробки клієнтської частини



5

Обрані технології розробки серверної частини

- Express.js — створення REST API;
- PostgreSQL — зберігання структурованих даних;
- Prisma — зручна робота з БД через ORM;
- JWT — авторизація та аутентифікація

express



6

Обраний ШІ

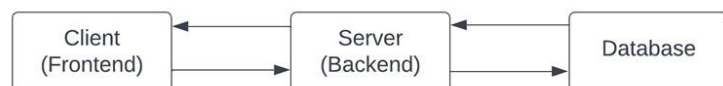
Використано GPT-4o від OpenAI — потужну модель, що забезпечує якісне стислий виклад тексту, швидку інтеграцію через API та стабільну роботу в реальному часі.



7

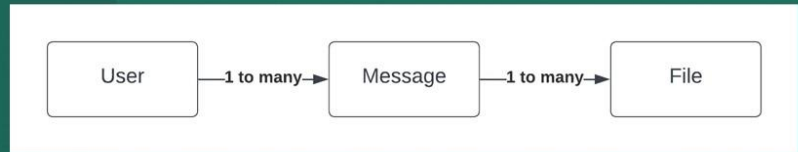
Архітектура веб-застосунку

Клієнт надсилає запити до сервера, який обробляє логіку, звертається до бази даних для зберігання або отримання даних, а також здійснює запити до OpenAI API для формування підсумку повідомлення за допомогою моделі GPT-4o.



8

Структура бази даних



9

Enter to Chat

Username

Password

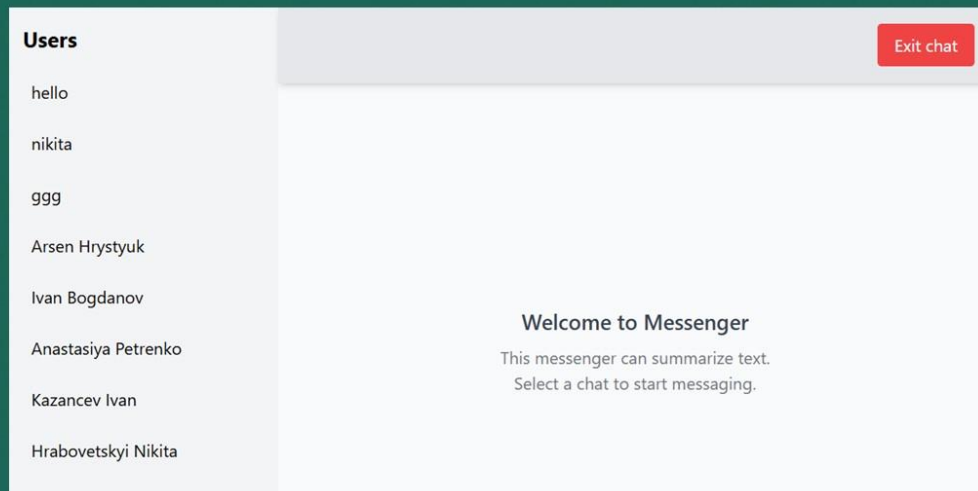
Sign in

Don't have an account? [Sign up](#)

Сторінка авторизації користувача

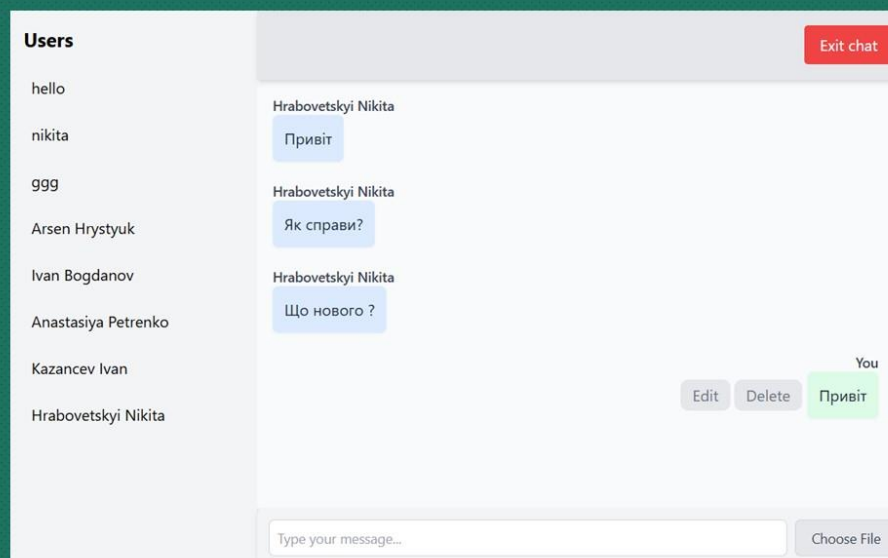
11

Робота головної сторінки



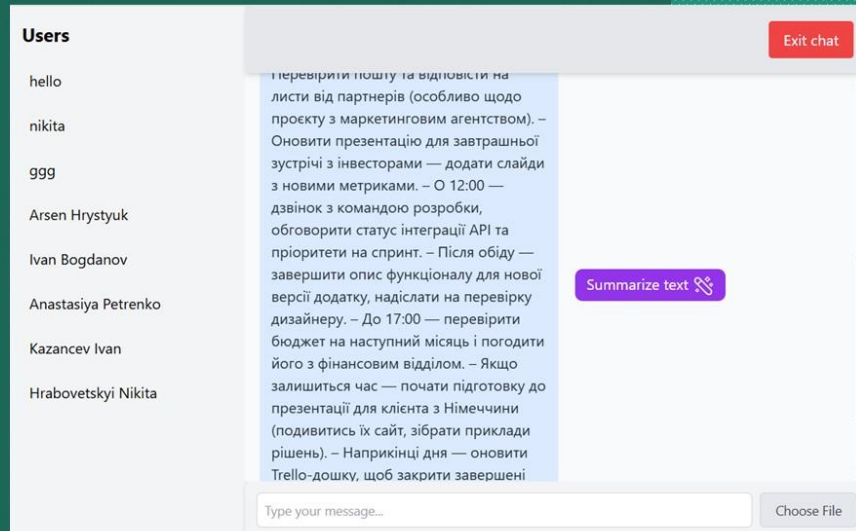
12

Інтерфейс діалогового вікна



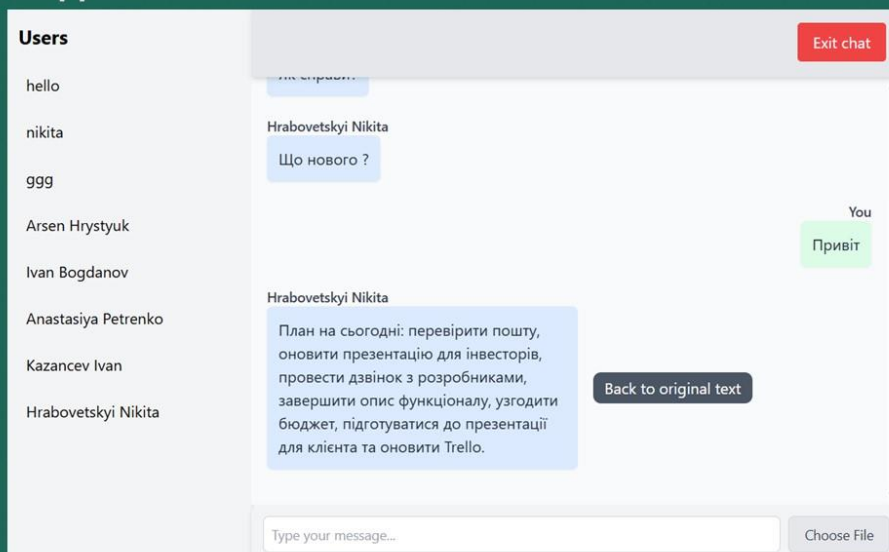
13

Інтерфейс діалогового вікна з функцією підсумування повідомлення



14

Функціональність після підсумування повідомлення



15

Порівняння результатів підсумування

Реалізований застосунок (Gpt-4o):

План на день: завершити редагування дипломної роботи, переглянути лекцію з інформаційних технологій, підготуватися до зустрічі з науковим керівником, відповісти на електронні листи, доопрацювати портфоліо, зайнятися технічним завданням на NestJS, виконати тренування, пов'язати рукави до светра для релаксації.

NeuralWriter:

Сьогодні планується багато робіт. Спочатку потрібно закінчити редагування четвертого розділу дипломної роботи, перевірити посилання на джерела та переформулювати висновки. Після обіду потрібно переглянути лекцію з "Інформаційними технологіями в архітектурі", зробити конспект та підготувати запитання до консультації. Також потрібно підготуватися до зустрічі з науковим керівником, відповісти на електронні листи та доповнити портфоліо. У другій половині дня планується технічне завдання на NestJS, а ввечері - тренування.

15

Висновки

Загалом, даний проєкт є завершеним, відповідно до початкової постановки задачі. Розроблено месенджер із функцією підсумування повідомлень за допомогою GPT-4o. Веб-застосунок підтримує обмін повідомленнями та файлами, редагування, видалення, авторизацію, автентифікацію та підсумування тексту за допомогою ШІ. Мету роботи досягнуто, система працює стабільно та має потенціал для подальшого розвитку.

У процесі подальшого удосконалення програмного продукту доцільно передбачити реалізацію функції групових чатів і покращення графічного інтерфейсу для підвищення зручності користування.

16

Дякую за увагу