ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
(повне найменування вищого навчального закладу)
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
(повне найменування інституту, назва факультету (відділення))
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до кваліфікаційної роботи
бакалавра
(освітній рівень)

на тему: «Розробка месенджеру з інтеграцією засобів штучного інтелекту»

Виконав: здобувач 4 року навчання
групи 4ПР1
напряму підготовки (спеціальності)
121-Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Бойко М.Є
(підпис, прізвище та ініціали)
Керівник к.т.н Хохлов В.А
(підпис, прізвище та ініціали)
Рецензент к.т.н доцент Вороненко М.О
(прізвище та ініціали)
Нормоконтролер Комісаров О. С.
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *Програмна інженерія*
(шифр і назва)
Спеціальність *121-Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
програмних засобів і технологій
_____ О.Є. Огнєва
«___» _____
2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Бойку Максиму Євгеновичу
(прізвище, ім'я, по батькові)
Тема роботи «Розробка месенджеру з інтеграцією засобів штучного інтелекту»
керівник роботи к.т.н. доцент Хохлов В.А.,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу від 16 . 01 .2026 р. № 82 -с
2. Строк подання студентом роботи 10.06.2026
3. Вихідні дані до роботи літературні та періодичні джерела, матеріали переддипломної практики
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Дослідження предметної області веб-месенджерів та обґрунтування технологічного стеку
2. Проєктування архітектури клієнт-серверного застосунку, бази даних та програмних інтерфейсів(API)
3. Програмна реалізація веб-месенджера та модуля штучного інтелекту
4. Тестування, контейнеризація та розгортання програмного продукту
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 16.01.2026

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання роботи	Термін виконання етапів роботи	Примітки
1.	Отримання завдання	10.02.2026	Виконано
2.	Підбір літератури	12.02.2026-20.02.2026	Виконано
3.	Аналіз предметної області	21.02.2026-27.02.2026	Виконано
4.	Розробка та обґрунтування завданн	28.02.2026-13.03.2026	Виконано
5.	Розробка концептуальної моделі	14.03.2026-20.03.2026	Виконано
6.	Моделювання та проектуванн системи	21.03.2026-03.04.2026	Виконано
7.	Моделювання та проектування баз даних	04.04.2026-10.04.2026	Виконано
8.	Розробка інтерфейсу сайту	11.04.2026-17.04.2026	Виконано
9.	Тестування сайту	18.04.2026-24.04.2026	Виконано
10	Оформлення пояснювальної записк	25.04.2026-01.05.2026	Виконано
11	Захист кваліфікаційної роботи		Виконано

Здобувач _____ Бойко М.Є
(підпис) (прізвище та ініціали)

Керівник роботи _____ Холов В.А
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка містить: 108 сторінки, 26 рисунків, 5 таблиць, 41 джерел, 3 додатки.

Об’єкт дослідження – процес організації обміну даними та повідомленнями в реальному часі з використанням технологій штучного інтелекту.

Предмет дослідження – методи, архітектурні рішення та програмні засоби для розробки клієнт-серверних вебмесенджерів.

Мета роботи – проєктування та програмна реалізація повноцінного вебзастосунку для комунікації з інтегрованим ШІ-асистентом для генерації відповідей та збору аналітики.

У кваліфікаційній роботі проаналізовано сучасні підходи до створення систем миттєвого обміну повідомленнями. На основі аналізу обґрунтовано вибір технологічного стеку: бібліотека React для клієнтської частини та платформа Node.js із суворою типізацією TypeScript для серверної. Здійснено порівняння архітектурних стилів RESTful та GraphQL, за результатами якого обрано оптимальний підхід для проєктування програмних інтерфейсів (API).

Розроблено архітектуру бекенду з використанням патерну Repository, що дозволило відокремити бізнес-логіку від прямих запитів до бази даних. Для забезпечення двостороннього зв'язку в реальному часі імплементовано технологію WebSockets.

Окрему увагу приділено розробці модуля штучного інтелекту. Реалізовано функціонал генерації контекстних підказок та обробки прямих запитів користувачів, а також систему збору зворотного зв'язку.

Ключові слова: ВЕБМЕСЕНДЖЕР, ШТУЧНИЙ ІНТЕЛЕКТ, WEBSOCKETS, REACT, NODE.JS, POSTGRESQL, DOCKER, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА.

АНОТАЦІЯ

Кваліфікаційну роботу присвячено проєктуванню та розробці клієнт-серверного вебмесенджера з інтегрованою системою штучного інтелекту(ШІ). Актуальність теми зумовлена тим, що класичні застосунки для обміну повідомленнями часто не мають вбудованих інструментів для автоматизації рутинної комунікації. Метою проєкту було створення платформи, де ШІ виступає не як окремий сторонній бот, а як нативний помічник, здатний аналізувати контекст діалогу, генерувати швидкі відповіді та обробляти прямі запити користувачів.

Під час виконання роботи було проведено аналіз архітектурних стилів, за результатами якого для побудови програмного інтерфейсу (API) обрано підхід RESTful, як найбільш оптимальний для реалізації базових операцій месенджера. Серверну частину реалізовано на платформі Node.js із використанням мови TypeScript. Для забезпечення гнучкості та легкості подальшого тестування застосовано архітектурний патерн Repository, що дозволило чітко відокремити бізнес-логіку контролерів від прямих SQL-запитів до бази даних.

Особливу увагу приділено проєктуванню інфраструктури обробки даних. Основним сховищем виступає реляційна СУБД PostgreSQL. Для оптимізації роботи сервера та уникнення блокування основного потоку виконання (Event Loop) імплементовано систему фонових задач на базі Redis. Вона працює як черга для ресурсоємних процесів, зокрема для розсилки електронних листів із токенами верифікації. Миттєву доставку повідомлень та синхронізацію статусів користувачів у реальному часі налаштовано через протокол WebSockets.

Клієнтську частину розроблено у форматі Single Page Application за допомогою бібліотеки React та інструменту збірки Vite. Управління станом застосунку реалізовано нативними засобами через Context API, що дозволило уникнути перевантаження проєкту зайвими залежностями. Інтерфейс включає функціонал управління приватними та груповими чатами, а також спеціальні компоненти для взаємодії з мовною моделлю.

Важливою складовою розробленої системи є модуль збору аналітики. База даних спроєктована таким чином, щоб фіксувати час генерації підказок ШІ та збирати відгуки користувачів щодо їхньої корисності. На фінальному етапі інфраструктуру проєкту було повністю контейнеризовано за допомогою Docker, а для візуалізації зібраних метрик розгорнуто BI-систему Metabase. Результатом роботи є готовий до масштабування програмний продукт, який успішно поєднує класичний функціонал месенджера з сучасними інтелектуальними технологіями.

ABSTRACT

The qualification work is dedicated to the design and development of a client-server web messenger with an integrated artificial intelligence system. The relevance of the topic stems from the fact that classical messaging applications often lack built-in tools for automating routine communication. The project aimed to create a platform where AI acts as a native assistant rather than a third-party bot, capable of analyzing dialogue context, generating quick replies, and processing direct user requests.

During the work, an analysis of architectural styles was conducted, resulting in the selection of the RESTful approach for the application programming interface (API) design as the most optimal for implementing basic messenger operations. The server-side was implemented on the Node.js platform using the TypeScript language. To ensure flexibility and ease of further testing, the Repository architectural pattern was applied, allowing for a clear separation of controller business logic from direct SQL queries to the database.

Special attention was paid to designing the data processing infrastructure. The primary storage is the PostgreSQL relational DBMS. To optimize server performance and prevent blocking of the main execution thread (Event Loop), a background task system based on Redis was implemented. It functions as a queue for resource-intensive processes, such as sending verification emails with tokens. Instant message delivery and real-time user status synchronization were configured via the WebSockets protocol.

The client-side was developed as a Single Page Application using the React library and the Vite build tool. Application state management was realized using native tools through the Context API, avoiding unnecessary project dependencies. The interface includes functionality for managing private and group chats, as well as specific components for interacting with the language model.

A key component of the developed system is the analytics collection module. The database was designed to log AI suggestion generation time and collect user feedback on their usefulness. At the final stage, the project infrastructure was fully

containerized using Docker, and the Metabase BI system was deployed to visualize the collected metrics. The result of the work is a scalable software product that successfully combines classical messenger functionality with modern intelligent technologies.

ЗМІСТ

ВСТУП

Актуальність теми**.** Розробка систем миттєвого обміну повідомленнями є однією з класичних, але водночас найскладніших задач у галузі веб-інженерії. Сучасні месенджери еволюціонували від простих інструментів для передачі тексту до комплексних платформ, які підтримують синхронізацію в реальному часі, управління медіафайлами та роботу з великими обсягами даних. Проте, незважаючи на широкий функціонал існуючих рішень на ринку, більшість із них залишаються закритими екосистемами, які складно адаптувати під специфічні корпоративні чи приватні потреби.

Окремою проблемою є ефективність комунікації. З появою великих мовних моделей (Large Language Models, LLM) користувачі отримали потужні інструменти для генерації тексту, однак зазвичай ці інструменти існують у вигляді окремих застосунків. Необхідність постійно перемикатися між вікном чату та інтерфейсом штучного інтелекту знижує продуктивність. Тому створення веб-месенджера, де ШІ інтегрований нативно і здатний аналізувати контекст діалогу для генерації швидких підказок або відповідей на прямі запити, є актуальним інженерним завданням.

З технічної точки зору, реалізація такого проєкту вимагає вирішення низки нетривіальних архітектурних завдань. Інтеграція ресурсоємних запитів до ШІ-моделей в архітектуру реального часу (WebSockets) може призвести до блокування основного потоку виконання сервера. Це створює потребу в проєктуванні надійної системи фонових задач (Background Jobs), правильному виборі архітектурного стилю API (RESTful) та застосуванні патернів проєктування, які забезпечать масштабованість системи та легкість її подальшого тестування.

Об’єкт дослідження – процес організації обміну даними та повідомленнями в реальному часі з використанням технологій штучного інтелекту.

Предмет дослідження – методи, архітектурні рішення та програмні інструменти для розробки та розгортання клієнт-серверних веб-месенджерів.

Мета роботи – проєктування архітектури та програмна реалізація масштабованого веб-месенджера з нативною інтеграцією модуля штучного інтелекту для генерації підказок та збору аналітики взаємодії користувачів із системою.

Для досягнення поставленої мети було визначено наступні задачі дослідження:

Проаналізувати існуючі рішення на ринку веб-месенджерів, визначити їхні недоліки та обґрунтувати вибір технологічного стеку для розробки.

Здійснити порівняльний аналіз архітектурних стилів API (RESTful та GraphQL) і спроєктувати програмні інтерфейси системи.

Спроєктувати реляційну базу даних та загальну архітектуру застосунку з використанням патерну Repository.

Розробити клієнтську частину (Single Page Application) з реалізацією управління станом та підтримкою WebSockets.

Реалізувати серверну частину з інтеграцією черг для обробки фонових задач та модулем штучного інтелекту.

Налаштувати інфраструктуру для збору аналітики, провести тестування розробленого продукту та контейнеризувати систему для зручного розгортання.

Практичне значення одержаних результатів**.** Результатом виконання кваліфікаційної роботи є повністю працездатний, контейнеризований програмний продукт, готовий до розгортання на серверних потужностях. Спроєктована архітектура бекенду (на базі Node.js та TypeScript) із чітким розділенням бізнес-логіки дозволяє легко масштабувати систему та додавати нові модулі без переписування існуючого коду. Реалізована інфраструктура даних, яка

включає PostgreSQL, Redis для фонових задач та систему Metabase, надає зручні інструменти для моніторингу навантаження та аналізу ефективності використання ШІ-підказок. Розроблений месенджер може використовуватися як основа для створення спеціалізованих корпоративних систем зв'язку.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Дослідження ринку веб-месенджерів та актуальність інтеграції ШІ

Проєктування систем миттєвого обміну повідомленнями є одним із ключових напрямків сучасної інженерії програмного забезпечення. Месенджери еволюціонували від простих чатів до комплексних платформ, проте аналіз популярних ринкових рішень (Slack, Telegram, Discord) виявляє низку архітектурних та функціональних обмежень у контексті інтеграції інтелектуальних інструментів.

По-перше, Slack та Discord є пропрієтарними закритими екосистемами, часто перевантаженими надлишковим функціоналом (як-от складні голосові канали). Їхня закритість унеможливлює розгортання на власних серверах організації (on-premise) для забезпечення суверенності даних (Data Sovereignty) [1, 2]. По-друге, взаємодія зі штучним інтелектом у цих продуктах відбувається переважно через сторонніх ботів, що порушує безшовність користувацького досвіду.

Своєю чергою Telegram вирізняється оптимізованою архітектурою та має базові інструменти реактивної обробки введеного тексту. Однак платформа не пропонує нативної проактивної генерації контекстних підказок для P2P-діалогів і, як комерційний продукт, не дозволяє розгортати власні модулі для аналітики ефективності використання ШІ співробітниками.

З огляду на виявлені недоліки, актуальним є створення легкої, контейнеризованої системи обміну повідомленнями з глибоко інтегрованим ШІ-

асистентом, яку можна вільно розгортати на власному обладнанні. Наочне порівняння характеристик існуючих продуктів та розроблюваної системи наведено в таблиця 1.1.

Таблиця 1.1

Порівняльний аналіз існуючих веб-месенджерів та розроблюваної системи

Критерій порівняння	Slack	Telegram	Discord	Розроблюваний проєкт
Архітектура	Пропрієтарна	Пропрієтарна	Пропрієтарна	Відкрита, self-hosted
Наявність групових чатів	Так	Так	Так	Так
Нативна інтеграція ШІ-асистента в UI	Ні	Обмежено(лише модифікація введеного тексту	Ні	Так
Локальна аналітика ефективності ШІ	Ні	Ні	Ні	Так

Стрімке зростання обсягів цифрової комунікації викликає інформаційне перевантаження та значні витрати робочого часу на формування рутинних відповідей. Хоча великі мовні моделі (LLM) дозволяють автоматизувати ці процеси, використання окремих ШІ-інструментів провокує проблему постійного перемикання контексту (Context Switching) [3, 4]. Необхідність вручну копіювати повідомлення між вікнами порушує природний потік комунікації та нівелює швидкість обміну даними в реальному часі.

Для вирішення цієї проблеми у розроблюваній системі пропонується концептуально інший підхід – нативна інтеграція ШІ на рівні інтерфейсу та бекенду через два модулі:

- AskAI – обробка прямих запитів (переклад, генерація коду, довідка) у відокремленому спливаючому вікні без виходу зі сторінки діалогу.
- AIGenerateSuggestions – проактивна фонова генерація коротких релевантних підказок безпосередньо над полем введення на основі аналізу поточного P2P-діалогу.

Головним технічним викликом під час реалізації автоматичних підказок є ефективне управління контекстним вікном (Context Window Management) [5]. Через жорсткі ліміти токенів (Token Limits) у сучасних API та вартість обчислень передавати всю історію чату недоцільно. Тому система автоматично агрегує лише найрелевантніші дані (останні N повідомлень, метадані користувачів та часові мітки), формуючи структурований системний промпт, що зображено на рис. 1.1 [6].

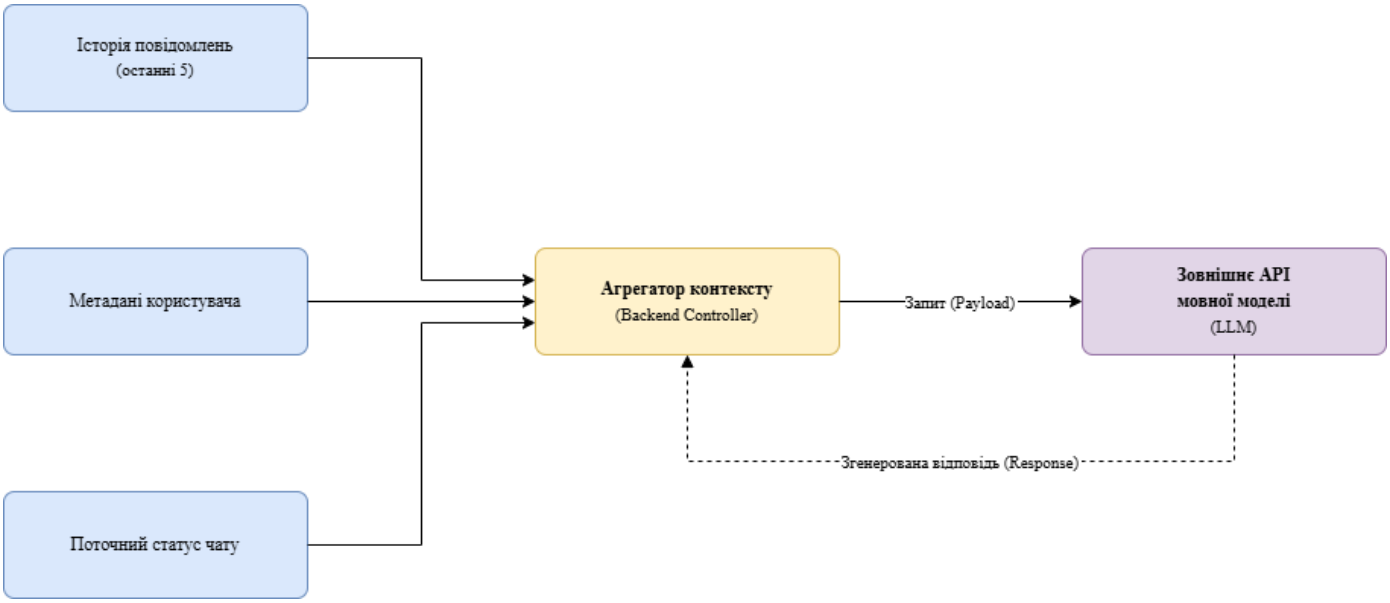


Рисунок 1.1 – Схема формування та передачі контекстного вікна до модуля ШІ

Окрім вирішення проблеми когнітивного навантаження на користувача, розробка подібної системи вимагає врахування інфраструктурних витрат. Оскільки генерація тексту за допомогою великих мовних моделей є

ресурсоємною операцією (як з точки зору завантаження серверних потужностей, так і фінансових витрат на використання комерційних API), розробнику необхідно імплементувати надійні інструменти моніторингу.

У зв'язку з цим, архітектура бази даних розроблюваного месенджера проєктується з урахуванням подальшої інтеграції з системами бізнес-аналітики (BI), зокрема Metabase. Система має автоматично фіксувати ключові метрики під час кожної транзакції до ШІ-модуля: час генерації відповіді (Latency) та бінарний зворотний зв'язок від користувача (відмітка про корисність або марність запропонованої підказки). Збір цих даних дозволить в подальшому будувати графіки ефективності, виявляти патерни використання ШІ та оптимізувати системні промпти для зниження відсотка хибних або нерелевантних генерацій.

Підсумовуючи, можна стверджувати, що розробка спеціалізованого веб-месенджера з нативними ШІ-модулями та інтегрованою системою аналітики є не лише вирішенням актуальної проблеми оптимізації комунікації, але й комплексним інженерним завданням, що вимагає проєктування масштабованої та стійкої до навантажень програмної архітектури.

1.2. Обґрунтування вибору мови TypeScript та середовища Node.js

Проєктування серверної частини (бекенду) для систем обміну повідомленнями у реальному часі кардинально відрізняється від розробки класичних REST-орієнтованих вебзастосунків (наприклад, інтернет-магазинів чи блогів). Головний технічний виклик полягає у необхідності підтримувати тисячі одночасних, постійно відкритих з'єднань між сервером та клієнтами.

Традиційний підхід до побудови серверів, який використовується в таких технологіях як PHP (Apache) або ранніх версіях Java (Tomcat), базується на моделі «один потік на один запит» (Thread-per-Request). У цій парадигмі для кожного нового клієнтського з'єднання операційна система виділяє окремий потік з власним обсягом оперативної пам'яті (зазвичай від 2 до 8 МБ на потік). Для класичного HTTP-запиту, який триває мілісекунди, ця модель є прийнятною.

Однак у випадку вебмесенджера клієнти підключаються через протокол WebSockets, і їхні з'єднання залишаються відкритими (idle) годинами, навіть якщо повідомлення не надсилаються. У такій ситуації багатопотоковий сервер швидко вичерпає ліміти оперативної пам'яті просто на підтримку порожніх з'єднань, а процесор буде перевантажений постійним перемиканням контексту (Context Switching) між тисячами потоків на рівні операційної системи [7].

Для вирішення цієї архітектурної проблеми в якості платформи серверної частини було обрано середовище виконання Node.js. Його фундаментальною відмінністю є використання однопотокової моделі з неблокуючим асинхронним введенням-виведенням (Non-blocking I/O), що базується на рушії V8 та бібліотеці libuv.

Замість створення нового потоку для кожного клієнта, Node.js обробляє всі вхідні з'єднання в єдиному головному потоці (Main Thread) загальну схему архітектури якого наведено на рис. 1.2. Секрет продуктивності цієї архітектури полягає в механізмі циклу подій (Event Loop). Коли клієнт надсилає повідомлення або запит до бази даних PostgreSQL, головний потік не блокується в очікуванні відповіді від диска чи мережі. Він делегує це завдання системному API або пулу потоків (Worker Pool), реєструє функцію зворотного виклику (callback) і миттєво переходить до обслуговування наступного користувача. Щойно операція введення-виведення завершується, результат потрапляє у чергу подій (Event Queue), звідки Event Loop повертає його в головний потік для відправки клієнту [8].

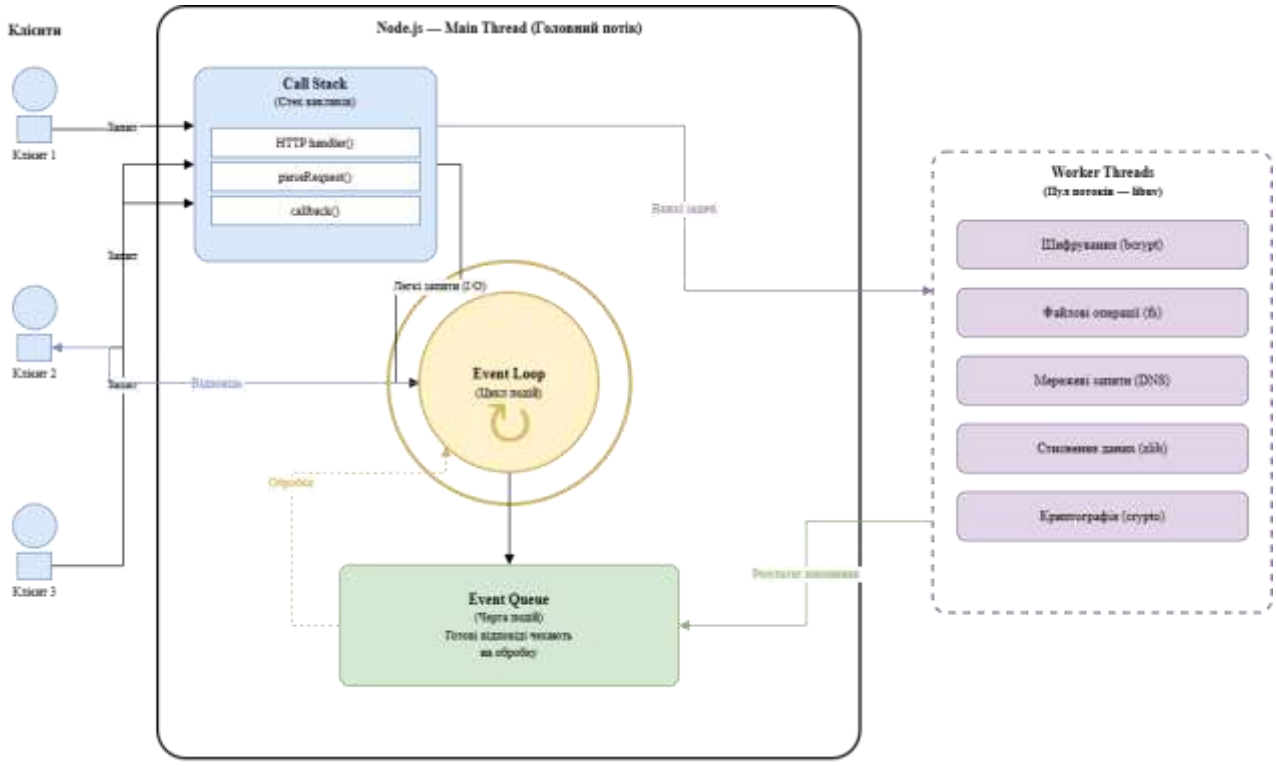


Рисунок 1.2 – Архітектура обробки запитів та цикл подій (Event Loop) у середовищі Node.js

Завдяки своїй архітектурі Node.js ідеально підходить для I/O-bound операцій з WebSockets, дозволяючи обробляти тисячі з'єднань із мінімальним споживанням пам'яті. Крім того, нативна підтримка JSON нівелює витрати часу на серіалізацію даних між бекендом та React-клієнтом.Проте, незважаючи на архітектурні переваги Node.js, використання базового стандарту JavaScript (ECMAScript) для розробки складних серверних систем несе в собі значні ризики на етапі підтримки та масштабування коду.

Однак базова слабка типізація JavaScript призводить до того, що більшість помилок (наприклад, звернення до неіснуючих властивостей) виявляються лише на етапі виконання (runtime) [9]. У високонавантаженому месенджері це створює постійний ризик падіння серверного процесу.

Для мінімізації цих ризиків основним інструментом розробки обрано TypeScript. Ця строго типізована надбудова гарантує дотримання контрактів

даних (інтерфейси об'єктів User, Message або IAiPromptPayload) і дозволяє виявляти помилки ще на етапі компіляції [10].

Важливо, що TypeScript відкриває доступ до повноцінного ООП та імплементації архітектурного патерну Repository [11]. Він усуває поширену проблему «товстих контролерів» (Fat Controllers), вводячи чіткий поділ: контролери відповідають лише за маршрутизацію, сервісний шар – за бізнес-логіку, а репозиторії інкапсулюють прямі SQL-запити до бази даних (схему продемонстровано на рис. 1.3).

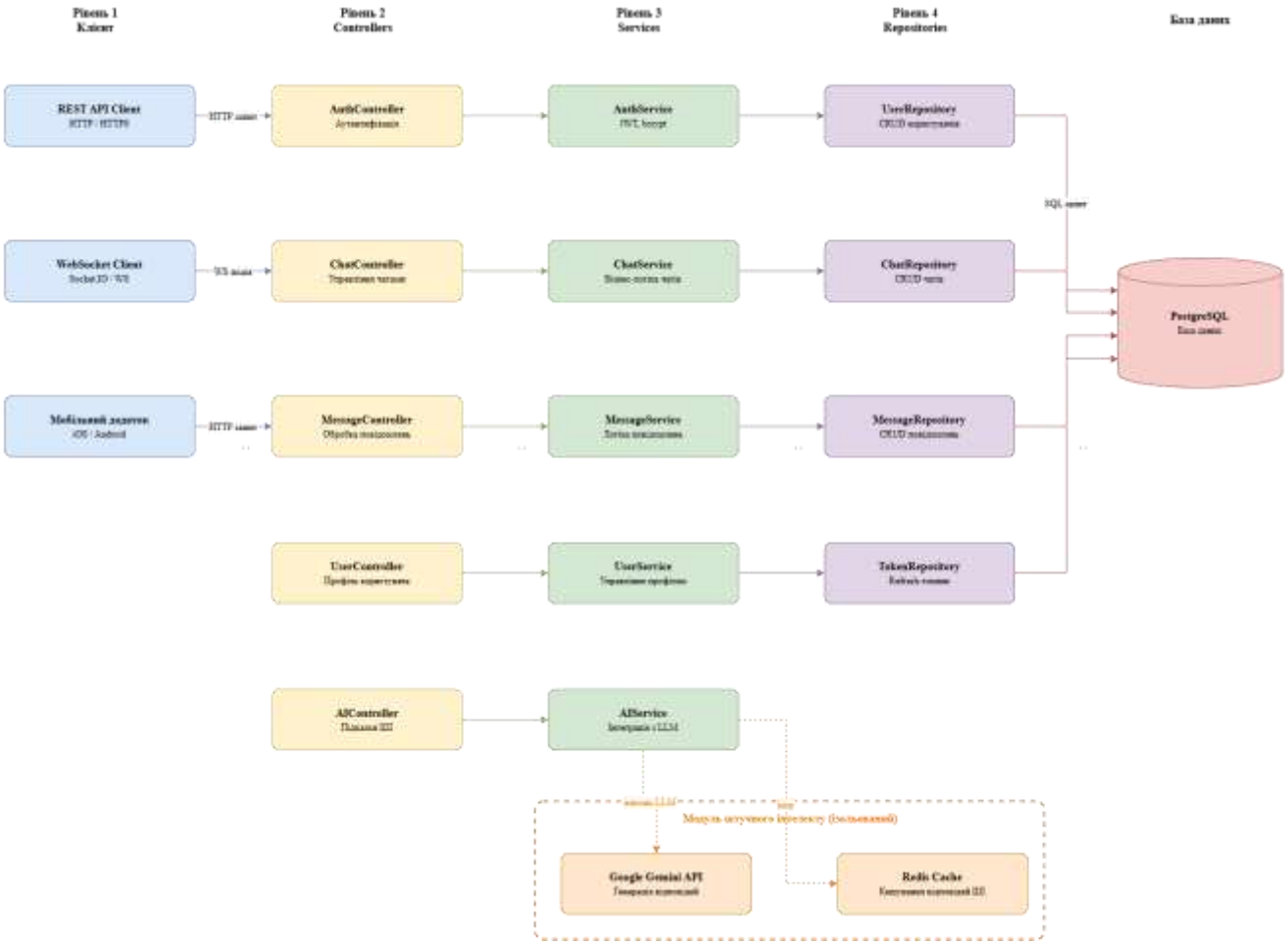


Рисунок 1.3 – Схема багаторівневої архітектури бекенду із застосуванням патерну Repository

Завдяки такій багатошаровій архітектурі (Layered Architecture), інтеграція модуля аналітики та збереження історії ШІ-генерацій стає тривіальною задачею.

Репозиторій ізолює специфіку реляційної бази даних PostgreSQL від решти застосунку. Якщо в майбутньому виникне потреба замінити базу даних або додати проміжне кешування запитів через Redis, достатньо буде змінити лише код відповідного репозиторію, не торкаючись бізнес-логіки та маршрутизаторів [10].

Таким чином, комбінація асинхронного середовища Node.js для обробки великої кількості WebSocket-з'єднань та мови TypeScript із патерном Repository для забезпечення надійності коду, створює оптимальний технологічний фундамент для розробки високонавантаженого сучасного месенджера з елементами ШІ.

1.3. Порівняльний аналіз архітектурних стилів API: RESTful vs GraphQL та вибір підходу

Для проєктування програмного інтерфейсу (API) вебмесенджера було зіставлено два домінуючих архітектурних підходи: класичний RESTful та мову запитів GraphQL.

Стиль RESTful будується навколо концепції незалежних ресурсів та використання стандартних методів HTTP. Його безперечною перевагою є висока передбачуваність та нативна підтримка інфраструктури вебкешування [12], що дозволяє миттєво завантажувати статичні дані (списки контактів, аватарки) на рівні браузера чи CDN. Проте під час розробки складних інтерфейсів REST API часто стикається з проблемами надмірної (Over-fetching) та недостатньої (Under-fetching) вибірки даних, що змушує клієнта виконувати серію послідовних запитів і суттєво збільшує мережеву затримку (Latency) [13]

На противагу цьому, технологія GraphQL надає клієнту єдину точку входу і дозволяє витягувати складні графи даних за один мережевий запит, повністю усуваючи проблему надмірної вибірки [14]. Однак такий високий рівень гнучкості на стороні клієнта створює непрогнозоване навантаження на базу

даних (проблема «N+1 запитів»), а використання єдиного POST-маршруту робить класичне та ефективне HTTP-кешування технічно неможливим [15].

Ураховуючи специфіку архітектури вебмесенджера, де критично важливою є робота в режимі реального часу, вибір між REST та GraphQL не може розглядатися у відриві від інфраструктури WebSockets. У нашому проєкті основний потік подій (надсилання та отримання повідомлень, оновлення статусів «онлайн», сповіщення про прочитання) реалізується через протокол двостороннього зв'язку Socket.io. Це означає, що роль традиційного API зміщується від повної передачі даних до виконання допоміжних, але критичних операцій: аутентифікації, завантаження історії чатів, зміни налаштувань профілю та взаємодії з модулями ШІ.

Хоча GraphQL пропонує потужні механізми підписок (Subscriptions) для роботи в реальному часі, їх впровадження вимагає значного ускладнення серверної логіки та використання додаткових шарів абстракції (наприклад, Redis Pub/Sub для масштабування). У системі, де вже імплементовано WebSockets, дублювання функціоналу через GraphQL Subscriptions є архітектурно надлишковим і створює додаткові точки відмови [16]. Натомість RESTful API у поєднанні з Socket.io дозволяє чітко розділити зони відповідальності: важкі запити на отримання ресурсів ініціюються через стандартні HTTP-методи, а миттєві апдейти інтерфейсу відбуваються через подієву модель сокетів.

Ще одним вагомим аргументом на користь REST є простота інтеграції зі сторонніми сервісами та мовними моделями (LLM). Сучасні API ШІ (наприклад, OpenAI або Anthropic) за замовчуванням побудовані на принципах REST. Використання ідентичного архітектурного стилю всередині нашого бекенду спрощує проксіювання запитів, обробку статусів генерації та управління заголовками безпеки. Крім того, документування REST API за допомогою стандарту OpenAPI (Swagger) дозволяє автоматично генерувати клієнтські типи для TypeScript, що забезпечує ту саму безпеку типів, якою зазвичай пишаються

прихильники GraphQL, але з набагато меншими витратами на налаштування інфраструктури [17].

Слід також урахувати фактор «холодного старту» та первинного завантаження застосунку. Для месенджера важливо, щоб список діалогів відображався миттєво після входу. Завдяки стандартному кешуванню REST-відповідей на рівні браузера, ми можемо суттєво зменшити час візуалізації (Time to First Byte), уникаючи повторних складних обчислень на сервері, які були б неминучими при обробці динамічних GraphQL-запитів [18].

Ураховуючи вищевикладене, а також рекомендації щодо забезпечення максимальної стабільності та швидкості розгортання системи (Time-to-Market), для реалізації програмного інтерфейсу вебмесенджера було обрано архітектурний стиль RESTful. Цей вибір дозволяє:

1. Забезпечити високу швидкість обробки запитів завдяки нативному кешуванню HTTP.
2. Гарантувати стабільну роботу системи в умовах високого навантаження через відсутність складних обчислювальних операцій на рівні API-шару.
3. Спростити архітектуру завдяки чіткому розділенню функцій між REST (адміністрування та дані) та WebSockets (реальний час).
4. Отримати прозору та стандартизовану документацію ендпоінтів для фронтенд-команди.

1.4. Обґрунтування вибору компонентів клієнтської частини (React, Context API)

Проєктування клієнтської частини (фронтенду) сучасного вебмесенджера супроводжіується низкою специфічних викликів, які суттєво відрізняються від розробки класичних статичних вебсайтів. Головна проблема полягає в тому, що інтерфейс чату є високодинамічним середовищем. Він повинен миттєво реагувати на безліч асинхронних подій: надходження нових повідомлень через WebSockets, оновлення статусів присутності користувачів («друкує...»,

«онлайн»), зміну активних діалогів та рендеринг відповідей від інтегрованого модуля ШІ. Реалізація такої поведінки за допомогою базового маніпулювання об'єктною моделлю документа (Vanilla JavaScript DOM API) неминуче призводить до створення заплутаного, непідтримуваного коду (так званого «spaghetti code») та серйозних проблем із продуктивністю [19].

Для забезпечення модульності коду та високої швидкості роботи інтерфейсу було проведено аналіз популярних компонентних фреймворків та бібліотек, серед яких розглядалися Angular, Vue.js та React.

Angular, розроблений компанією Google, є повноцінним MVC-фреймворком із суворо заданою архітектурою (Opinionated). Хоча він ідеально підходить для розробки масштабних ентерпрайз-систем (наприклад, банківських порталів), для створення месенджера його інструментарій є надлишковим. Він вимагає обов'язкового використання складних механізмів впровадження залежностей (Dependency Injection) та бібліотеки RxJS для роботи з реактивними потоками, що невиправдано збільшує поріг входження та кінцевий розмір клієнтського пакета (Bundle size). Бібліотека Vue.js, своєю чергою, пропонує більш збалансований підхід, проте має меншу екосистему готових рішень для інтеграції складних UI-компонентів порівняно з лідером ринку.

З огляду на це, базовою технологією клієнтської частини було обрано бібліотеку React. Її ключовою архітектурною перевагою є декларативний підхід до побудови інтерфейсів та використання концепції віртуального дерева (Virtual DOM) [20]. У контексті месенджера це має критичне значення. Коли користувач відкриває історію діалогу, інтерфейс може містити сотні DOM-вузлів (окремих повідомлень, аватарок, часових міток). Якщо при отриманні одного нового повідомлення браузер змушений перемальовувати весь список цілком, це спричинить падіння частоти кадрів (FPS) та візуальні «підгальміовування». Натомість механізм Virtual DOM у React спочатку застосовує зміни до легкої копії дерева в оперативній пам'яті, обчислює мінімальну різницю (алгоритм Diffing) і

точково оновлює лише той елемент екрану, який дійсно змінився [21]. Це гарантує плавну роботу застосунку навіть на слабких мобільних пристроях.

Важливим аспектом архітектури фронтенду є також вибір інструментарію для збірки проєкту (Build Tool). Історично для React-застосунків стандартом де-факто був пакет Create React App (CRA), який базується на Webpack. Проте з розвитком екосистеми цей інструмент визнано застарілим через вкрай повільний процес компіляції великих проєктів. Тому для розроблюваної системи було імплементовано сучасний збирач Vite. На відміну від Webpack, який щоразу перезбирає весь проєкт при найменшій зміні коду, Vite використовує нативні модулі ECMAScript (ESM) безпосередньо у браузері [22]. Це забезпечує практично миттєве гаряче оновлення модулів (Hot Module Replacement, HMR), що критично важливо для швидкого прототипування інтерфейсу чату та налаштування інтеграції з ШІ-модулями.

Ключовим архітектурним викликом під час розробки клієнтської частини на React є управління станом (State Management). Оскільки дані за замовчуванням передаються строго односпрямовано від батьківських компонентів до дочірніх, у складних інтерфейсах месенджера виникає антипатерн «Prop Drilling» [23] – вимушене прокидання властивостей через проміжні компоненти, яким ці дані для функціонування не потрібні.

Історичним корпоративним стандартом вирішення цієї проблеми була бібліотека Redux, проте її інтеграція вимагає значного обсягу шаблонного коду (Boilerplate) і невиправдано збільшує розмір кінцевого бандла. Для розроблюваного проєкту, де більшість даних мають локальний або модульний контекст (наприклад, стан конкретного відкритого діалогу або статус генерації підказки ШІ), використання Redux є відверто надлишковим. Таке рішення суперечить базовому принципу інженерії YAGNI (You Aren't Gonna Need It – «Вам це не знадобиться») і лише невиправдано ускладнює процес прототипування та подальшу підтримку коду [24].

З огляду на це, для управління станом було обрано нативний інструмент – React Context API. Цей механізм дозволяє створювати ізольовані контексти (AuthContext, ChatContext, AIContext) та надавати доступ до них будь-якому компоненту напряму, повністю оминаючи проміжні ланки (схему для порівняння зображено на рис. 1.5). Використання Context API у поєднанні зі спеціальними хуками (Custom Hooks), такими як useChat або useAskAI, дозволило повністю інкапсулювати бізнес-логіку на рівні провайдерів. омпоненти інтерфейсу (наприклад, вікно повідомлення або кнопка запиту до ШІ) залишаються «чистими» (Pure Components).

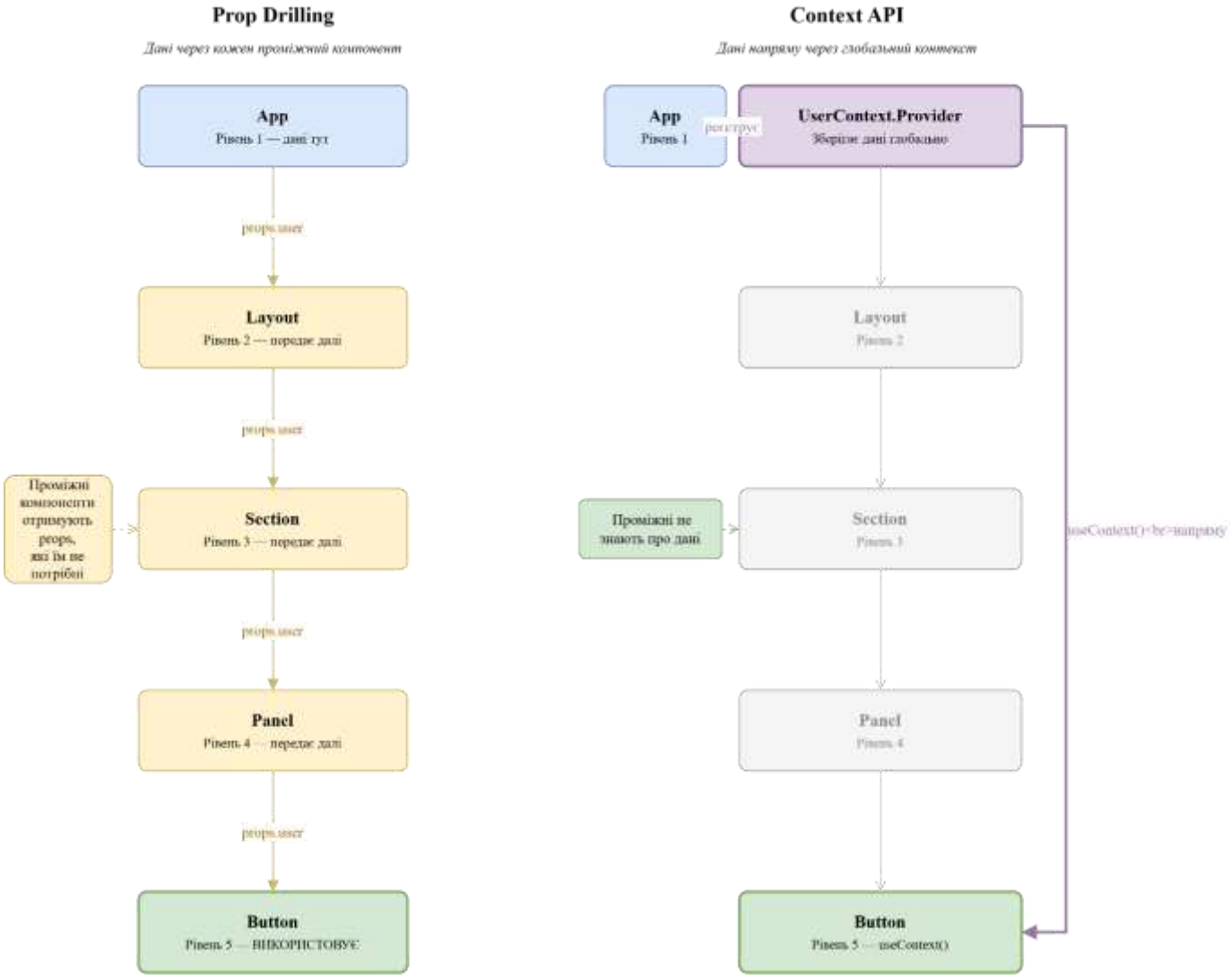


Рисунок 1.5 – Патерни передачі даних у компонентній архітектурі: Prop Drilling проти Context API

Такий архітектурний поділ відповідальності (Separation of Concerns) робить клієнтський код легко масштабованим, стійким до помилок і значно спрощує процес його подальшого тестування.

1.5. Аналіз вимог до інфраструктури зберігання та аналітики даних

Підсистема управління даними вебмесенджера має яскраво виражену специфіку: це архітектура з інтенсивним записом (write-heavy) та складною системою реляційних зв'язків між користувачами, групами і налаштуваннями ШІ.

Хоча для збереження динамічних масивів повідомлень документо-орієнтовані бази (NoSQL, як-от MongoDB) здаються логічним вибором, вони не завжди гарантують безкомпромісну транзакційну цілісність (ACID) під час складних багатоетапних операцій. Щоб уникнути аномалій даних та розсинхронізації прав доступу, основним сховищем було обрано надійну об'єктно-реляційну СУБД PostgreSQL.

Окрім підтримки строгих схем та зовнішніх ключів, вирішальним фактором на користь PostgreSQL стала нативна підтримка типу даних JSONB. Інтеграція ШІ-асистента вимагає збереження великих обсягів неструктурованих метаданих (контекстні вікна діалогів, логи параметрів генерації). Використання JSONB дозволяє гнучко зберігати ці об'єкти безпосередньо в реляційних таблицях, уникаючи створення десятків зайвих колонок [26].

Такий гібридний підхід успішно поєднує транзакційну надійність для базового функціоналу месенджера та документо-орієнтовану гнучкість для ефективної роботи з ШІ-метаданими.

Важливою складовою інфраструктури вебмесенджера є підсистема управління станом підключень (Presence Management) та маршрутизації повідомлень у реальному часі. На базовому рівні архітектури, взаємодія з клієнтами реалізується через бібліотеку Socket.io. Згідно з поточним програмним

рішенням, зміна мережевого статусу користувача («онлайн» або «офлайн») фіксується безпосередньо в реляційній базі даних. Під час встановлення WebSocket-з'єднання або його розірвання (подія disconnect), сервер виконує прямий SQL-запит до таблиці user_statuses із використанням механізму ON CONFLICT для оновлення мітки часу last_seen.

Такий підхід є цілком виправданим на етапі прототипування та розгортання системи в межах одного серверного вузла (Single-node architecture), оскільки він гарантує постійну консистентність даних та простоту реалізації [27]. Однак із інженерної точки зору він має два суттєві обмеження, які вимагають розширення інфраструктури.

Перше обмеження стосується масштабованості. Стандартна імплементація Socket.io зберігає перелік активних кімнат (Rooms) та ідентифікаторів підключень у локальній оперативній пам'яті процесу Node.js. Якщо в майбутньому інфраструктуру проєкту буде горизонтально масштабовано (наприклад, запущено три екземпляри сервера за балансувальником навантаження Nginx), користувачі, підключені до різних серверів, не зможуть обмінюватися подіями напряму [28]. Крім того, виконання прямого запису (I/O операції) в PostgreSQL на кожен "чих" мережі (відключення екрану смартфона, зміна мережі з Wi-Fi на 4G) створює невиправдане транзакційне навантаження на основне сховище.

Друге обмеження пов'язане з інтеграцією ШІ. Генерація підказок за допомогою зовнішнього API мовної моделі (LLM) – це тривалий процес, який може займати від 1 до 5 секунд. Якщо виконувати цю дію синхронно в основному потоці обробки HTTP-запитів або сокетів, це призведе до блокування відповідей для інших користувачів.

Для вирішення обох архітектурних проблем інфраструктура системи потребує впровадження in-memory сховища структур даних – Redis.

Використання Redis у розроблюваному месенджері закриває відразу дві критичні потреби:

1. Брокер повідомлень (Pub/Sub Adapter): Інтеграція Redis Adapter для Socket.io дозволить серверам миттєво обмінюватися подіями між собою через механізм публікації-підписки. Це гарантує доставку повідомлення співрозмовнику, навіть якщо користувачі фізично підключені до різних контейнерів застосунку. Також Redis дозволяє перенести зберігання тимчасових статусів «друкує» та швидких змін «онлайн» із жорсткого диска PostgreSQL у швидку оперативну пам'ять.
2. Управління чергами (Background Jobs): Інтеграція Redis дозволяє розгорнути систему черг (наприклад, на базі бібліотеки BullMQ). Коли користувач робить запит до ШІ-модуля, бекенд не чекає на відповідь моделі. Він лише ставить задачу в чергу Redis і миттєво повертає клієнту статус 202 Accepted. Окремий фоновий процес (Worker) забирає задачу з черги, виконує ресурсоємний запит до LLM і після отримання результату відправляє згенеровану підказку клієнту через WebSocket-канал.

Отже, застосування in-memory сховища є не просто інструментом кешування, а фундаментальною архітектурною необхідністю для забезпечення стабільної роботи месенджера під навантаженням.

Будь-який сучасний застосунок потребує моніторингу, але для месенджера з нативною підтримкою мовних моделей це питання стає першочерговим. Справа в тому, що запити до сторонніх AI-сервісів несуть два головні ризики: прямі фінансові витрати на токени та нестабільну затримку відповіді (latency). Наприклад, якщо генерація підказки триватиме понад три секунди, користувач із високою ймовірністю просто не буде її чекати і почне набирати текст сам. Відповідно, такий фоновий виклик ШІ стає марною тратою серверних та фінансових ресурсів [29].

Щоб об'єктивно оцінити ефективність інтелектуальних функцій, у системі реалізовано безперервний збір метрик. Для цього в базі даних виділено окремі таблиці, куди Node.js сервер записує результати кожної транзакції до мовної моделі. Фіксуються конкретні показники: ідентифікатор користувача, обсяг

переданого контексту чату та час, який сервер витратив на очікування відповіді від LLM. Окремо зберігається бінарна оцінка («корисно» або «некорисно»), яку юзер може присвоїти запропонованому варіанту повідомлення.

Проте самі по собі сирі дані в таблицях PostgreSQL дають мало користі, адже їх потрібно зручно агрегувати та візуалізувати. Замість того, щоб розробляти власну аналітичну панель з нуля і витрачати час на налаштування фронтенд-бібліотек на кшталт Chart.js, було прийнято рішення використати готовий інструмент класу Business Intelligence (BI) з відкритим кодом – Metabase. Це дозволило зосередити зусилля безпосередньо на бізнес-логіці месенджера, а не на розробці графіків.

Metabase легко розгортається як ізольований Docker-контейнер, що ідеально вписується в загальну інфраструктуру проєкту. Цей інструмент дає змогу будувати складні дашборди без необхідності писати громіздкі SQL-запити вручну [30]. Завдяки цьому можна швидко вивести на екран графік падіння швидкості ШІ під час пікових навантажень або подивитися відсоток відхилених підказок.

При цьому підключення BI-системи до робочої бази даних реалізовано з урахуванням навантаження. Аналітичні запити (OLAP) обробляють великі масиви інформації і можуть блокувати таблиці, тому Metabase працює з базою виключно через акаунт із правами Read-Only. Якщо в майбутньому проєкт буде горизонтально масштабуватися, аналітику можна буде легко перемкнути на асинхронну репліку (Read Replica). Це архітектурне рішення гарантує, що побудова важких звітів ніяк не вплине на швидкість відправки повідомлень у користувацьких чатах [31].

Після визначення технологічного стеку та архітектурних підходів до побудови серверної інфраструктури, фінальним етапом аналітичного розділу є сувора формалізація вимог до системи. Для того щоб перейти безпосередньо до етапу проєктування бази даних та написання коду (що буде описано у другому розділі роботи), необхідно чітко зафіксувати межі відповідальності майбутнього

програмного продукту. Найбільш ефективним стандартизованим інструментом для цього в інженерії програмного забезпечення є побудова UML-діаграми прецедентів (Use Case diagram).

У межах розроблюваного месенджера виділено двох ключових акторів (Actors). Першим і головним є «Користувач» (User) – фізична особа, яка безпосередньо взаємодіє з клієнтським інтерфейсом через веббраузер. Другим, специфічним для даного проєкту актором, виступає зовнішня «Мовна модель» (AI Assistant). У контексті UML вона розглядається як автономна системна сутність, яка асинхронно приймає згенеровані бекендом промпти та повертає результат.

Базовий функціонал, який має бути реалізований на рівні коду для задоволення потреб Користувача, включає наступні ізольовані прецеденти:

- Реєстрація, авторизація та управління власним профілем.
- Створення приватних (Peer-to-Peer) діалогів та багатокористувацьких груп (кімнат).
- Відправка текстових повідомлень у режимі реального часу, їх редагування та зміна статусу на «прочитано».
- Формування прямого запиту до ШІ-асистента через спеціалізований ізольований модуль інтерфейсу.
- Отримання фонових контекстних підказок під час ведення діалогу та надання зворотного зв'язку (оцінка релевантності запропонованого тексту).

Зі свого боку, актор «Мовна модель» бере участь виключно у прецедентах обробки переданого контексту та генерації відповідей, не маючи прямого доступу до персональних даних користувачів, діаграму прецедентів системи наведено у рис. 1.6.

Рисунок 1.6 – Діаграма прецедентів (Use Case) системи інтелектуального вебмесенджера

Наведена діаграма візуалізує загальні функціональні вимоги і слугує технічним завданням для подальшої розробки. Вона наочно демонструє, що модуль взаємодії зі штучним інтелектом є не просто додатковою надбудовою, а повноцінним системним компонентом, який глибоко інтегрований у базовий флоу обміну повідомленнями.

Висновки до розділу 1

У ході виконання першого розділу кваліфікаційної роботи було проведено комплексний аналіз предметної області та інженерне обґрунтування вибору технологій для створення вебмесенджера. Доведено, що для підтримки тисяч одночасних з'єднань через протокол WebSockets оптимальним архітектурним

рішенням є використання асинхронного середовища Node.js у поєднанні зі строгою типізацією TypeScript.

Ухвалено рішення про побудову програмного інтерфейсу за стандартом RESTful, що значно спрощує механізми вебкешування порівняно з GraphQL. Для реалізації клієнтської частини обрано бібліотеку React із застосуванням нативного Context API, що дозволило ефективно вирішити проблему передачі даних (Prop Drilling) без надлишкової інтеграції Redux. Інфраструктура зберігання даних базуватиметься на зв'язці PostgreSQL (для забезпечення транзакційної надійності та збереження JSON-метаданих) і Redis (для швидкого управління подіями та побудови черг для фонових задач ШІ). Збір аналітики ефективності мовних моделей покладено на BI-систему Metabase.

Такий комплексний архітектурний підхід повністю відповідає сучасним вимогам інженерії та створює надійний фундамент для переходу до етапу проєктування бази даних та практичної імплементації коду, що розглядається у наступному розділі роботи.

РОЗДІЛ 2. ПРОЕКТУВАННЯ

2.1. Загальна архітектура системи та схема взаємодії клієнт-сервер

Перехід від етапу аналізу до безпосереднього проєктування вимагає чіткого розуміння того, як саме складові частини месенджера будуть спілкуватися між собою. В основу системи лягла класична трирівнева клієнт-серверна архітектура. Вона передбачає логічний поділ на клієнтський застосунок (рівень представлення, написаний на React), серверну частину (рівень бізнес-логіки на Node.js) та шар зберігання інформації (бази даних PostgreSQL та Redis). Такий розподіл є фактичним стандартом в індустрії, бо дозволяє розробляти та тестувати фронтенд незалежно від бекенду[32].

Цікавим інженерним рішенням у розроблюваній системі є підхід до побудови фізичної топології мережі. Дуже часто в подібних проєктах між клієнтом та Node.js сервером ставлять додатковий проміжний вузол – зворотний проксі-сервер (наприклад, Nginx). Проте на поточному етапі розвитку нашого месенджера від такої надбудови було свідомо вирішено відмовитися. Фізична топологія побудована за принципом прямої взаємодії: браузер клієнта звертається безпосередньо до єдиного відкритого мережевого порту Node.js. Відмова від проміжного проксі-сервера дозволила значно спростити інфраструктуру, зменшити затримку (latency) на перенаправлення пакетів та зробити систему більш зручною для швидкого розгортання в локальному Docker-середовищі.

Головним технічним викликом такої архітектури стала необхідність обробляти два абсолютно різні протоколи на одному фізичному порту. Месенджер працює в двох площинах: звичайні запити (наприклад, реєстрація або отримання історії чату) ходять через HTTP, а миттєві повідомлення – через WebSockets. Node.js чудово справляється з цією задачею завдяки механізму прив'язки (binding) до єдиного об'єкта http.Server.

На практиці цей флоу виглядає так. Коли користувач відкриває месенджер, перші запити йдуть як класичні REST-виклики. Сервер приймає їх і віддає під контроль фреймворку Express, який працює як маршрутизатор. Але щойно клієнт хоче підключитися до чату для обміну повідомленнями в реальному часі, він надсилає специфічний HTTP-запит із заголовками Connection: Upgrade та Upgrade: websocket. Сервер бачить цей маркер, розуміє, що це не звичайний запит, і передає управління бібліотеці Socket.io, яка перетворює це з'єднання на постійний відкритий тунель. Цей процес паралельної маршрутизації трафіку наочно відображено на рис. 2.1.

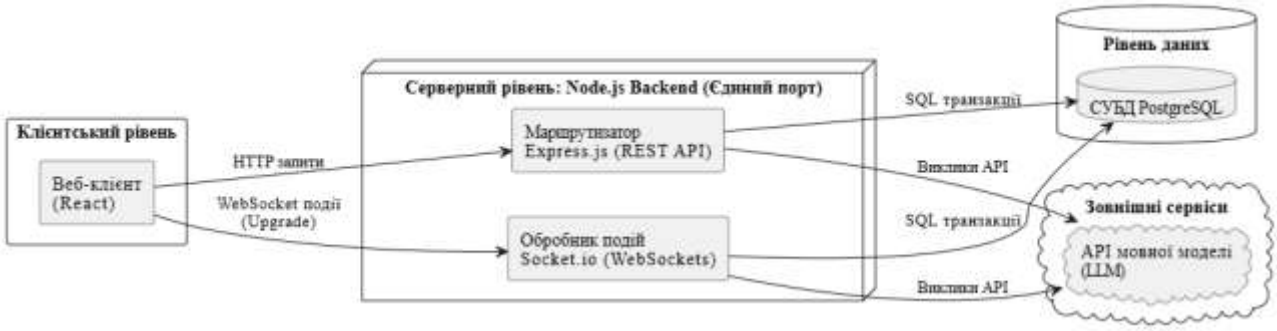


Рисунок 2.1 – Компонентна архітектура системи та схема маршрутизації на єдиному серверному порту

Незважаючи на те, що на вході трафік розділяється за протоколами, всередині сервера архітектура зводиться до єдиного знаменника. І маршрути Express, і обробники подій Socket.io не містять у собі важкої логіки. Вони є лише контролерами, які дістають дані з повідомлення і передають їх на нижній рівень – у сервіси та репозиторії. Це гарантує, що повідомлення буде оброблено за однаковими правилами і безпечно збережено в базу даних незалежно від того, як саме воно потрапило на сервер.

Окрім безпосередньої маршрутизації трафіку, критично важливою складовою клієнт-серверної взаємодії є підсистема безпеки та управління сесіями. У традиційних вебзастосунках авторизація часто будується на базі серверних сесій (Stateful sessions), які зберігаються в оперативній пам'яті сервера.

Проте для розроблюваного месенджера такий підхід є технологічним обмеженням, оскільки він створює жорстку прив'язку користувача до конкретного серверного процесу і значно ускладнює подальше горизонтальне масштабування інфраструктури.

З огляду на це, у системі імплементовано механізм авторизації без збереження стану (Stateless) на основі стандарту JSON Web Token (JWT). Алгоритм взаємодії виглядає наступним чином. Після успішної перевірки облікових даних під час входу, сервер генерує криптографічно підписаний токен, який містить ідентифікатор користувача та часову мітку терміну дії підпису. Усі подальші HTTP-запити від клієнта до захищених ресурсів (наприклад, для завантаження історії діалогів або зміни налаштувань профілю) супроводжуються передачею цього токена у стандартному заголовку Authorization. Проміжне програмне забезпечення (Middleware) на боці фреймворку Express перехоплює кожен такий запит, виконує швидку математичну валідацію підпису за допомогою секретного ключа і лише після цього надає доступ до контролерів бізнес-логіки.

Окремим інженерним викликом у розробленій архітектурі є захист двостороннього каналу зв'язку через WebSockets. На відміну від звичайних REST-викликів, стандартні браузерні API не дозволяють встановлювати довільні HTTP-заголовки під час ініціалізації сокет-з'єднання. Для забезпечення належного рівня безпеки процес верифікації було перенесено на етап початкового мережевого «рукостискання» (Handshake).

Клієнтський застосунок конфігурується таким чином, щоб передавати ідентифікатор користувача та токен доступу безпосередньо в параметрах запиту (query payload) під час спроби підключення. Серверний екземпляр Socket.io аналізує ці дані до моменту фактичного приєднання клієнта до пулу активних з'єднань. У разі відсутності валідного токена або розбіжності підпису, з'єднання примусово розривається на рівні транспортного протоколу. Це повністю

унеможливлює несанкціоноване прослуховування чужих приватних кімнат (Rooms) та перехоплення трансляцій статусів.

Крім внутрішніх механізмів авторизації, архітектура враховує зовнішні мережеві загрози. Оскільки рівень представлення (фронтенд) та рівень бізнес-логіки (бекенд) фізично розділені, система потребує жорсткого налаштування політик міждоменного обміну даними (CORS – Cross-Origin Resource Sharing). Сервер Node.js конфігурується так, щоб приймати HTTP-запити та дозволяти оновлення з'єднання до WebSockets виключно від довірених джерел (конкретних адрес клієнтського інтерфейсу). Таке обмеження на рівні бекенду виступає базовим бар'єром проти атак міжсайтової підробки запитів (CSRF) та спроб несанкціонованої взаємодії зі сторонніх вебресурсів.

Комплексне поєднання єдиної точки входу на Node.js, протокольного мультиплексування та криптографічного захисту з'єднань дозволило створити швидку, стійку до навантажень і безпечну архітектуру передачі повідомлень у реальному часі.

2.2. Проєктування структури реляційної бази даних (PostgreSQL)

Наступним, і технологічно чи не найважливішим етапом створення вебмесенджера, є розробка структури зберігання інформації. Оскільки застосунок оперує великим масивом складно пов'язаних сутностей (користувачі, діалоги, статуси, історія взаємодії з ШІ), основу шару даних складає реляційна модель. Усі таблиці в базі даних спроєктовано з дотриманням правил третьої нормальної форми (3НФ). Це інженерне рішення дозволяє мінімізувати логічне дублювання інформації та гарантовано уникнути так званих аномалій оновлення чи видалення під час виконання SQL-транзакцій.

Базовою сутністю розробленої системи є таблиця users. Вона виступає ядром для управління доступом та ідентифікації клієнтів. У ній зберігається фундаментальна інформація про облікові записи: унікальні ідентифікатори (Primary Key), імена, електронні адреси та хешовані паролі. Важливо наголосити,

що з міркувань кібербезпеки оригінальні паролі в базі ніколи не зберігаються; натомість записується лише результат їх обробки криптографічним алгоритмом хешування. Для поля електронної пошти встановлено обмеження унікальності (UNIQUE constraint), що унеможливлює реєстрацію кількох акаунтів на одну адресу. Окремо в цій таблиці передбачено атрибути для збереження шляхів до файлів аватарів (які фізично зберігаються на сервері) та системні часові мітки created_at і updated_at.

Для організації безпосередньої комунікації спроєктовано таблицю chats. Її структура є лаконічною і відповідає за логічне групування повідомлень. Вона містить первинний ключ, індикатор типу діалогу (наприклад, приватна бесіда між двома особами або багатокористувацька група) та назву чату, яка заповнюється лише у випадку групових обговорень.

Архітектурною особливістю будь-якого месенджера є те, що один користувач може бути учасником багатьох різних чатів, а один чат – містити безліч користувачів. Між сутностями users та chats виникає класичний зв'язок типу «багато-до-багатьох» (Many-to-Many). У реляційній парадигмі баз даних такий тип зв'язку неможливо реалізувати напряму без порушення нормалізації. Тому для вирішення цієї структурної задачі впроваджено проміжну зв'язуючу таблицю chat_members.

Ця таблиця виконує роль логічного моста. Кожен її рядок містить зовнішні ключі (Foreign Keys) – user_id та chat_id, які суворо посилаються на відповідні батьківські таблиці. На рівні бази даних для цих зовнішніх ключів налаштовано правила каскадного видалення (ON DELETE CASCADE). Це означає, що у разі повного видалення профілю користувача із системи, база даних автоматично очистить усі записи про його присутність у чатах, запобігаючи появі «сиротинських» даних (orphaned records). Крім забезпечення зв'язку, таблиця chat_members виконує додаткову бізнес-функцію: вона зберігає метадані членства. Саме тут фіксується роль особи у конкретній групі (звичайний учасник чи адміністратор) та точний час її приєднання до бесіди.

Безпосередній обмін інформацією фіксується у таблиці messages. Оскільки архітектура месенджера передбачає інтенсивне навантаження на операції запису (write-heavy), ця таблиця спроєктована з мінімальною кількістю надлишкових полів. Кожен запис містить унікальний ідентифікатор, текст повідомлення (content), а також два зовнішні ключі: chat_id (посилання на кімнату) та sender_id (посилання на автора). Для забезпечення функціоналу редагування передбачено фіксацію міток часу created_at та updated_at. Логічним доповненням до системи повідомлень є таблиця read_receipts, яка імплементує механізм сповіщень про прочитання. Завдяки фіксації ідентифікатора повідомлення та ідентифікатора користувача з міткою read_at, клієнтський застосунок отримує можливість коректно відображати статуси перегляду контенту в багатокористувацьких групах.

Для забезпечення роботи підсистеми реального часу та відображення актуального стану користувачів спроєктовано ізольовану таблицю user_statuses. Винесення цих даних в окрему сутність, а не додавання їх до основної таблиці профілів, є оптимізаційним рішенням. Воно дозволяє уникнути постійного блокування рядків з персональними даними під час частих оновлень мережевого статусу. Таблиця містить булеве поле online та часову мітку last_seen. На рівні бази даних оновлення цих записів виконується через механізм UPSERT (оператор ON CONFLICT), що мінімізує кількість транзакцій під час підключення та відключення клієнтських WebSocket-з'єднань.

Особливої уваги заслуговує підсистема збереження даних модуля штучного інтелекту, яка представлена таблицями ai_suggestions та ai_analytics. Інтеграція з великими мовними моделями (LLM) вимагає збереження великих обсягів телеметричних та контекстних даних, структура яких може динамічно змінюватися залежно від обраного провайдера API (наприклад, OpenAI або Anthropic).

Таблиця ai_suggestions відповідає за збереження згенерованих підказок (suggested_text) із прив'язкою до конкретного повідомлення (message_id). Вона

містить поле статусу (accepted), яке дозволяє системі розуміти, чи скористався користувач запропонованим варіантом відповіді. Своєю чергою, таблиця ai_analytics виконує роль журналу транзакцій для збору об'єктивних метрик ефективності. Тут фіксуються час відповіді сервера (response_time_ms), тип взаємодії (interaction_type) та розмір корисного навантаження (payload_size).

Саме для збереження складних неструктурованих метаданих (наприклад, сформованого масиву історії чату, який передавався у промпті, або конфігурації температури генерації) у цих таблицях імплементовано використання спеціалізованого типу даних JSONB. Цей формат дозволяє зберігати JSON-об'єкти у бінарному вигляді безпосередньо в реляційній базі, забезпечуючи високу швидкість їхнього парсингу та можливість створення індексів по внутрішніх атрибутах об'єкта. Використання JSONB позбавляє необхідності створювати десятки додаткових колонок і дозволяє гнучко адаптувати базу даних під нові версії API мовних моделей без виконання складних структурних міграцій.

Окремим і вкрай важливим етапом проєктування реляційної моделі є розробка стратегії індексування та налаштування внутрішніх обмежень СУБД для забезпечення цілісності інформації. Враховуючи специфіку архітектури, де кожна дія користувача супроводжується транзакцією, відсутність контролю за структурою неминуче призвела б до появи дубльованих записів та лінійного падіння продуктивності.

Для оптимізації швидкості виконання базових SQL-запитів (SELECT, UPDATE, DELETE) у системі імплементовано стратегію індексування на основі збалансованих дерев (B-Tree). Як свідчить конфігурація бази даних, такі індекси розгорнуто для всіх первинних ключів таблиць (наприклад, messages_pkey, chats_pkey). Цей підхід гарантує стабільну логарифмічну складність доступу до записів під час вибірки конкретних сутностей за їхніми ідентифікаторами.

Особливу інженерну увагу було приділено проєктуванню складених та унікальних індексів для забезпечення бізнес-логіки на рівні самої СУБД,

оминаючи рівень застосунку (Node.js). Класичним прикладом такого рішення є створений індекс unique_suggestion_user у таблиці ai_analytics. Він жорстко контролює комбінацію ідентифікатора підказки (suggestion_id) та користувача (user_id). Завдяки цьому обмеженню система фізично не дозволяє зберегти кілька оцінок релевантності від одного користувача для однієї згенерованої відповіді ШІ. Аналогічний підхід застосовано і в таблиці user_settings через індекс unique_user_settings_user_id, що гарантує строгий зв'язок типу «один-до-одного» між профілем та його конфігурацією. Повна логічна структура бази даних із відображенням усіх описаних сутностей, їхніх атрибутів, типів даних та нормалізованих зв'язків наведена у вигляді ER-діаграми на рис. 2.2.

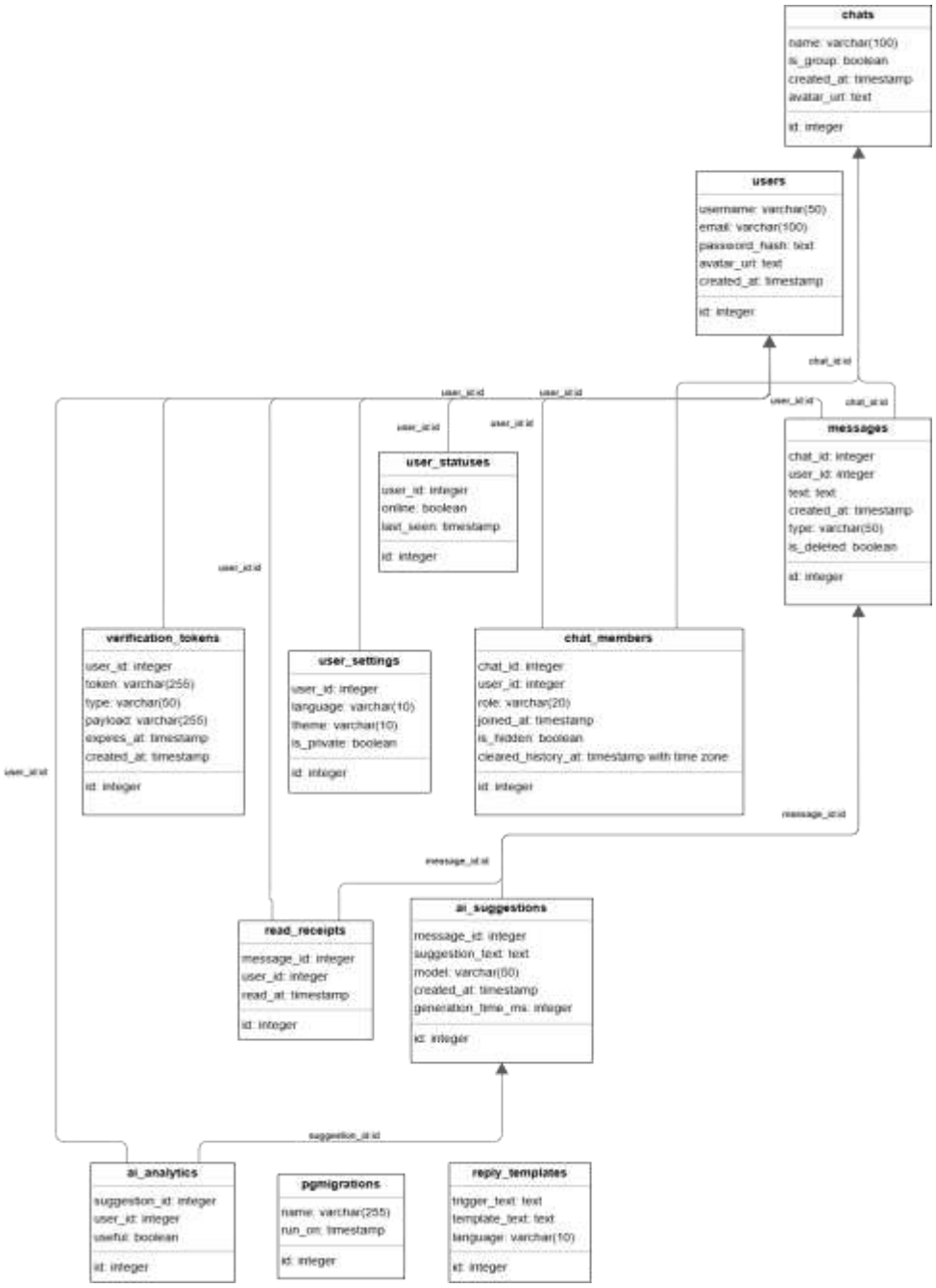


Рисунок 2.2 – Логічна структура реляційної бази даних (ER-діаграма) системи месенджеру

Щодо збереження неструктурованих метаданих штучного інтелекту у форматі JSONB, була застосована прагматична стратегія. На етапі проєктування прототипу розроблюваної системи було прийнято свідоме рішення відмовитися від створення важких узагальнених інвертованих індексів (GIN) для внутрішніх атрибутів JSON-об'єктів. Аналіз патернів доступу до даних показав, що пошук аналітики відбувається переважно за вже проіндексованими зовнішніми ключами (ідентифікатором користувача або чату).

Створення ресурсоємних GIN-індексів у таких умовах класифікується як передчасна оптимізація (Premature Optimization), яка лише сповільнила б операції запису (INSERT) у таблиці ai_suggestions та ai_analytics. Таким чином, гібридне використання гнучкого формату JSONB для зберігання складних конфігурацій ШІ-запитів у поєднанні зі швидкими B-Tree індексами для зовнішнього пошуку забезпечує оптимальний баланс між продуктивністю системи та можливостями її подальшого масштабування.

2.3. Проєктування програмного інтерфейсу (API) застосунку

Після завершення етапу проєктування реляційної структури бази даних логічним кроком є розробка програмного інтерфейсу (API). Цей компонент виступає ключовою сполучною ланкою, яка забезпечує стандартизовану взаємодію між клієнтським застосунком та шаром зберігання інформації. Для розроблюваного вебмесенджера базовим архітектурним стилем програмного інтерфейсу було обрано REST (Representational State Transfer). Такий вибір зумовлений його орієнтованістю на ресурси та нативним використанням стандартних HTTP-методів, що ідеально підходить для операцій CRUD (створення, читання, оновлення, видалення) над сутностями користувачів, чатів та повідомлень.

Оскільки серверна частина реалізована на платформі Node.js із використанням фреймворку Express, під час розробки виникає архітектурний ризик змішування логіки маршрутизації з бізнес-правилами та SQL-запитами в

одному файлі. Щоб уникнути створення жорстко зв'язаного та непідтримуваного коду (відомого як антипатерн «God Object»), архітектуру програмного інтерфейсу було спроєктовано за класичним багаторівневим патерном (Layered Architecture). Цей підхід забезпечує суворе розділення відповідальності (Separation of Concerns) між програмними модулями бекенду.

Серверний код системи логічно поділено на три незалежні рівні. Перший рівень утворюють маршрутизатори (Routes). Їхня єдина технічна задача полягає у перехопленні вхідного HTTP-запиту, ідентифікації шляху (URI) та передачі управління відповідному обробнику.

Другий рівень складається з контролерів (Controllers). На цей шар покладено завдання з розбору параметрів запиту: вилучення даних із тіла (Body), рядка запиту (Query) або параметрів шляху (Params). Контролери виконують первинну санітизацію даних, оркеструють виклики до бізнес-логіки та форматують вихідну відповідь клієнту, встановлюючи відповідні HTTP-статуси та генеруючи JSON-структуру.

Третій, найбільш критичний рівень, реалізує безпосередню взаємодію з базою даних через застосування патерну абстракції доступу до даних – Репозиторію (Repository Pattern). Згідно з цією концепцією, для кожної ключової сутності предметної області створено окремий клас абстракції (наприклад, UserRepository, ChatRepository, MessageRepository). Такий інженерний підхід дозволяє повністю інкапсулювати складні реляційні запити та транзакції всередині специфічних модулів. Завдяки цьому контролерам не потрібно знати специфіку синтаксису PostgreSQL; вони лише викликають стандартизовані методи репозиторіїв (наприклад, getChatById або saveMessage).

Впровадження багаторівневої абстракції з використанням репозиторіїв дає системі дві суттєві переваги. По-перше, значно спрощується процес подальшого модульного тестування (Unit Testing), оскільки кожен рівень можна ізолювати за допомогою заглушок (Mocking). По-друге, це забезпечує високу гнучкість інфраструктури: у разі гіпотетичної необхідності зміни рушія бази даних або

переходу на ORM-систему (Object-Relational Mapping), достатньо буде переписати лише внутрішню логіку класів-репозиторіїв, не вносячи жодної зміни у контролери чи маршрутизатори застосунку.

Важливим аспектом проєктування програмного інтерфейсу є дотримання принципів семантичного іменування ресурсів та правильного використання методів протоколу HTTP. У розроблюваній системі URI (Uniform Resource Identifier) будуються за ієрархічним принципом, де назви ресурсів вказуються у множині (наприклад, /api/chats, /api/messages). Це робить структуру API інтуїтивно зрозумілою та легкою для розширення.

Для взаємодії з ресурсами використовуються чотири основні методи, кожен з яких має чітко визначену роль:

- GET – виключно для отримання даних (наприклад, списку чатів або профілю користувача), що забезпечує ідемпотентність запитів;
- POST – для створення нових сутностей (реєстрація, надсилання повідомлення, створення групи);
- PUT/PATCH – для повного або часткового оновлення існуючих даних (зміна пароля, редагування назви чату);
- DELETE – для видалення ресурсів (вихід із групи, видалення повідомлення).

Крім методів, стандартизація охоплює і коди відповідей сервера. Система не просто повертає дані, а сигналізує про результат операції через стандартні класи статусів: 200 OK (успіх), 201 Created (успішне створення об'єкта), 400 Bad Request (помилка валідації на боці клієнта), 401 Unauthorized (відсутність або невалідність JWT-токена) та 500 Internal Server Error (непередбачена помилка на рівні бізнес-логіки бекенду). Загальна схема життєвого циклу запиту та формування відповіді наведена на рис. 2.3.

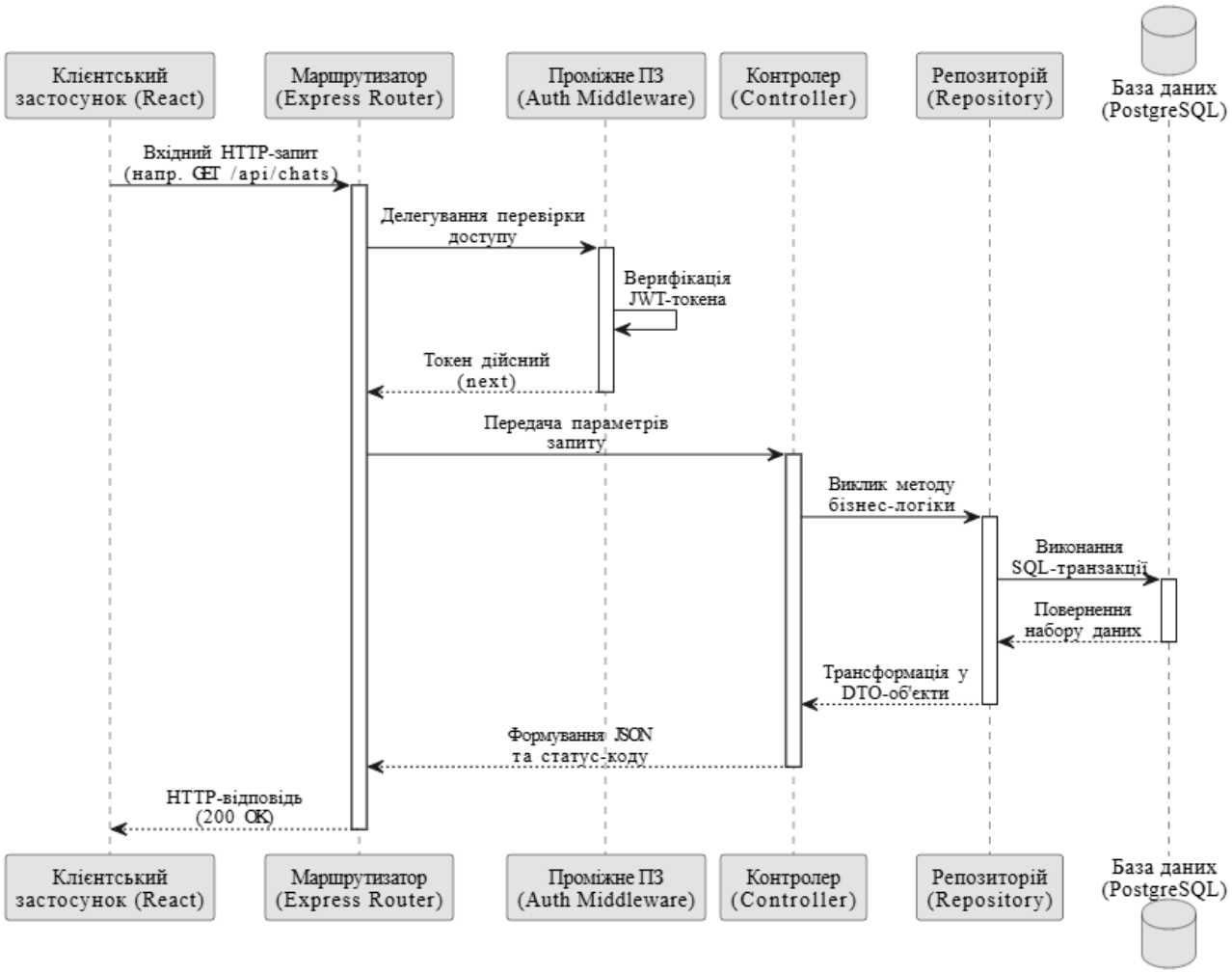


Рисунок 2.3 – Схема обробки HTTP-запиту та формування уніфікованої відповіді сервера

Для деталізації логіки роботи інтерфейсу нижче наведено специфікації основних модулів API. У таблиці 2.1 представлено ендпоінти підсистеми автентифікації, які забезпечують безпечний вхід та управління обліковим записом.

Таблиця 2.1

Специфікація API-ендпоінтів модуля автентифікації та профілю користувача

Метод	Шлях (URI)	Призначення	Тіло запиту / Параметри	Код відповіді
POST	/api/auth/register	Реєстрація нового акаунта	JSON (email, password, name)	201
POST	/api/auth/login	Авторизація та видача JWT	JSON (email, password)	200
GET	/api/auth/verify	Перевірка email через токен	Query (?token=...)	200
POST	/api/auth/reset	Запит на відновлення пароля	JSON (email)	200
GET	/api/users/me	Отримання даних свого профілю	Authorization: Bearer <token>	200
PATCH	/api/users/settings	Зміна налаштувань (тема, ШІ)	JSON (theme, ai_enabled)	200

Після успішної авторизації клієнт отримує доступ до функціоналу управління комунікаційними каналами. Специфікація ендпоінтів для роботи з чатами наведена у таблиці 2.2.

Таблиця 2.2

Специфікація API-ендпоінтів модуля управління чатами та учасниками

Метод	Шлях (URI)	Призначення	Тіло запиту / Параметри	Код відповіді
GET	/api/chats	Отримання списку всіх діалогів	Authorization: Bearer <token>	200
POST	/api/chats	Створення нового чату або групи	JSON (title, type, members)	201
GET	/api/chats/{id}	Детальна інформація про чат	ID у шляху	200
POST	/api/chats/{id}/members	Додавання учасника в групу	JSON (user_id)	200
DELETE	/api/chats/{id}/leave	Вихід користувача з діалогу	ID у шляху	204

Впровадження такої структури дозволяє фронтенд-частині застосунку передбачувано взаємодіяти з даними, використовуючи типізовані запити. Використання REST-ендпоінтів для важких операцій (наприклад, отримання масиву з 50 останніх діалогів) суттєво розвантажує WebSocket-канал, залишаючи його виключно для передачі швидких подій реального часу.

Окрім базових контролерів та репозиторіїв, невід'ємною складовою розробленого програмного інтерфейсу є використання проміжного програмного забезпечення (Middleware). Ці модулі функціонують як конвеєр перехоплювачів (Interceptors), які аналізують, модифікують або відхиляють HTTP-запити ще до моменту їх потрапляння на рівень основної бізнес-логіки. Такий підхід дозволяє винести загальні інфраструктурні задачі за межі контролерів, дотримуючись принципу єдиної відповідальності (Single Responsibility Principle).

У розроблюваній системі імплементовано два ключових Middleware-модулі. Перший – модуль авторизації (authMiddleware). Його завдання полягає у парсингу заголовка Authorization, вилученні JWT-токена та його криптографічній верифікації за допомогою секретного ключа сервера. Якщо токен є валідним, ідентифікатор користувача декодується і глобально ін'єктується в об'єкт запиту (request object), що дозволяє наступним контролерам безпечно ідентифікувати ініціатора дії. У разі компрометації або закінчення терміну дії токена, Middleware миттєво перериває ланцюжок виконання та повертає клієнту статус 401 Unauthorized.

Другий важливий перехоплювач відповідає за обробку завантажень медіафайлів (зокрема, аватарів користувачів). Оскільки стандартні JSON-запити не здатні ефективно передавати бінарні дані, для оновлення фотографії профілю клієнт формує запит типу multipart/form-data. На серверній стороні цей потік даних перехоплюється спеціалізованою бібліотекою (наприклад, multer). Вона виконує буферизацію вхідного потоку, валідує розширення та MIME-тип файлу (допускаючи лише графічні формати .jpeg, .png, .webp), після чого фізично

зберігає файл у публічну директорію сервера. Контролер отримує вже готове посилання на збережений ресурс, яке потім записується в базу даних PostgreSQL.

Окремим архітектурним блоком програмного інтерфейсу виступають модулі для роботи з текстовим контентом та інтелектуальними функціями. Для оптимізації мережевого навантаження операції з історією повідомлень винесені в REST API, тоді як WebSockets використовуються переважно для нових надходжень. Специфікація ендпоінтів цього сегмента наведена у таблиці 2.3.

Таблиця 2.3

Специфікація API-ендпоінтів модулів обміну повідомленнями та інтелектуального асистента

Метод	Шлях (URI)	Призначення	Тіло запиту / Параметри	Код відповіді
GET	/api/messages/{chatId}	Завантаження історії повідомлень діалогу	Пагінація: ?limit=50&offset=0	200
POST	/api/messages/{chatId}	Надсилання повідомлення (fallback-метод)	JSON (content, reply_to)	201
DELETE	/api/messages/{id}	Видалення повідомлення	ID у шляху запиту	204
POST	/api/ai/suggest	Генерація ШІ-підказок для швидкої відповіді	JSON (chatId, context_length)	200
POST	/api/ai/ask	Прямий запит до ШІ-асистента (AskAI)	JSON (prompt, system_instruction)	200

POST	/api/ai/analytics	Фіксація метрик (вибір підказки)	JSON (suggestion_id, accepted)	201

Архітектура ШІ-ендпоінтів враховує специфіку роботи з великими мовними моделями (LLM). Виклик маршруту /api/ai/suggest запускає складний ланцюжок дій: серверний репозиторій звертається до бази даних, витягує останні кілька повідомлень із вказаного чату для формування контексту, структурує їх у єдиний промпт і відправляє синхронний запит до зовнішнього API мовної моделі. Отримана відповідь форматується і повертається клієнту у вигляді масиву готових до використання варіантів відповідей.

Спроєктований REST-інтерфейс повністю закриває потреби застосунку в управлінні статичними даними (створення чатів, налаштування профілів, автентифікація). Водночас для реалізації ключової вимоги до сучасних месенджерів – миттєвої доставки повідомлень – система потребує впровадження паралельного транспортного протоколу, що базується на постійних з'єднаннях.

2.4. Архітектура обміну даними у реальному часі через WebSockets

Реалізація сучасного вебмесенджера неможлива без забезпечення механізму миттєвої доставки повідомлень. Традиційна архітектура на базі REST API, описана в попередніх підрозділах, чудово справляється з базовими CRUD-операціями, але має фундаментальний недолік при роботі з динамічними даними – вона є суворо однонаправленою. У такій парадигмі ініціатором будь-якого обміну інформацією завжди виступає клієнтський застосунок. Відповідно, щоб своєчасно отримати нове повідомлення від співрозмовника, браузер повинен був би генерувати постійні опитування сервера.

Історично подібні задачі вирішувалися за допомогою технологій HTTP Polling або Long-Polling. Проте для розроблюваної системи такий підхід було визнано архітектурно неефективним через колосальні мережеві накладні витрати

(overhead). Кожен новий HTTP-запит несе з собою важкий набір службових заголовків, що в умовах великої кількості одночасних підключень неминуче призвело б до перевантаження мережевого каналу та швидкого вичерпання ресурсів процесора на бекенді. З огляду на це, базовим стандартом для забезпечення режиму реального часу було обрано протокол WebSockets. Ця технологія дозволяє встановити постійний, повноцінний двосторонній (Full-Duplex) канал зв'язку поверх єдиного TCP-з'єднання. Особливість підходу полягає в тому, що після відкриття каналу і сервер, і клієнт набувають рівноправної можливості асинхронно відправляти легкі пакети даних (фрейми) в будь-який момент часу, не чекаючи запиту від протилежної сторони.

Для практичної імплементації цього протоколу на обох кінцях системи (у React-фронтенді та Node.js-бекенді) інтегровано програмну бібліотеку Socket.io. Відмова від використання базового, "голого" (native) WebSocket API на користь цієї бібліотеки зумовлена кількома суттєвими інженерними перевагами. Насамперед Socket.io абстрагує складність управління життєвим циклом з'єднання. Бібліотека самостійно підтримує механізм так званого "серцебиття" (обмін ping/pong пакетами). Це дозволяє серверу своєчасно виявляти "мертві" підключення (наприклад, коли користувач втратив мобільний інтернет, але браузер не встиг надіслати сигнал розірвання) і звільняти оперативну пам'ять. Крім того, інструмент забезпечує високу відмовостійкість клієнтської частини завдяки вбудованим алгоритмам автоматичного перепідключення з експоненційним збільшенням затримки (exponential backoff).

Життєвий цикл інтерактивної взаємодії починається з етапу мережевого рукостискання (Handshake). Згідно з логікою роботи системи, одразу після того, як користувач успішно авторизується, клієнтський компонент ініціює стартовий HTTP-запит до серверного порту. Цей запит містить специфічні заголовки Connection: Upgrade та Upgrade: websocket, а також передає у параметрах автентифікаційний JWT-токен (що є ключовим елементом безпеки, як описувалося раніше). Сервер перехоплює цей запит, валідує криптографічний

підпис токена, і лише у разі успішної верифікації погоджується на "підвищення" протоколу. Сервер повертає HTTP-відповідь зі статусом 101 Switching Protocols, яка перетворює початковий тимчасовий запит на постійний відкритий TCP-тунель. З цієї мілісекунди класичний цикл "запит-відповідь" припиняється, транспортний рівень фіксується, і система переходить у режим високошвидкісного реактивного прослуховування подій.

Після успішного встановлення постійного з'єднання між клієнтом та сервером виникає наступне критичне архітектурне завдання – маршрутизація та логічна ізоляція потоків даних. Оскільки до одного екземпляра Node.js сервера одночасно підключені тисячі WebSocket-сесій, проста широкомовна трансляція (Global Broadcast) кожного повідомлення всім підключеним клієнтам призвела б до катастрофічного витоку конфіденційної інформації та колапсу пропускної здатності мережі.

Для вирішення цього завдання в системі реалізовано механізм логічного мультиплексування на основі концепції «Кімнат» (Rooms), яка нативно підтримується ядром Socket.io. З інженерної точки зору, кімната не є окремим фізичним портом або процесом; це легковитратна структура даних (Hash Map) в оперативній пам'яті сервера, яка зберігає масив ідентифікаторів сокетів, підписаних на певну подію. Такий підхід реалізує класичний архітектурний патерн «Видавець-Підписник» (Publish-Subscribe).

Практичний алгоритм ізоляції діалогів працює наступним чином. Коли користувач через клієнтський застосунок відкриває конкретну бесіду, фронтенд генерує специфічну подію join_chat і передає у якості корисного навантаження (payload) унікальний ідентифікатор цього чату (chatId). На серверній стороні обробник цієї події приймає ідентифікатор, форматує його у стандартизований рядок (наприклад, chat_123) і викликає внутрішній метод приєднання сокета до цієї кімнати. Завдяки цьому створюється тимчасовий віртуальний тунель: клієнт гарантовано отримуватиме лише ті пакети даних, які адресовані конкретно цій бесіді. При перемиканні на інший діалог генерується подія leave_chat, яка

видаляє сокет із попередньої кімнати, очищуючи пам'ять сервера від непотрібних підписок.

Окремої уваги заслуговує реалізація методів трансляції повідомлень, оскільки від їх правильного використання залежить загальна продуктивність інтерфейсу. У розробленому модулі застосовано диференційований підхід до відправки пакетів:

1. Точкова відповідь (socket.emit): Використовується сервером для підтвердження успішного виконання операції конкретному клієнту-ініціатору (наприклад, повернення статусу збереження повідомлення в базу даних). Інші учасники чату цього пакету не бачать.
2. Тотальна трансляція у кімнату (io.to().emit): Застосовується у випадках, коли необхідно синхронізувати стан абсолютно всіх учасників групи. Наприклад, при зміні назви чату сервером, новий заголовок відправляється всім клієнтам, включаючи того, хто ініціював зміну.
3. Вибіркова трансляція (socket.broadcast.to().emit): Це ключовий метод оптимізації для надсилання текстових повідомлень. Коли Клієнт «А» надсилає повідомлення, його власний фронтенд застосунок самостійно малює це повідомлення на екрані (концепція оптимістичного інтерфейсу користувача – Optimistic UI). Тому серверу немає сенсу відправляти копію повідомлення назад Клієнту «А». Метод broadcast гарантує, що пакет даних буде надіслано всім учасникам кімнати chatId, за винятком самого відправника. Це вдвічі зменшує мережеве навантаження на ініціатора події.

Така глибока ізоляція на рівні протоколу дозволяє безпечно розмежовувати приватні бесіди та гарантує, що повідомлення будуть доставлені виключно авторизованим учасникам конкретної групи, мінімізуючи при цьому обсяг надлишкового трафіку.

Особливістю спроєктованої системи реального часу є використання гібридної моделі комунікації при обміні повідомленнями. На відміну від класичних (але менш надійних) реалізацій, де текстові дані відправляються

клієнтом безпосередньо через WebSocket-з'єднання (подія socket.emit), у розробленій архітектурі процес публікації контенту розділено на транзакційний та подієвий етапи.

Ініціація відправки повідомлення або його видалення відбувається через захищений канал REST API шляхом виклику відповідних контролерів (наприклад, POST /api/messages). Такий підхід гарантує первинне збереження контенту в реляційній базі даних PostgreSQL з дотриманням усіх перевірок цілісності та авторизації. Тільки після успішної фіксації транзакції у базі, серверний контролер звертається до глобального екземпляра Socket.io та ініціює подію трансляції (receive_message або message_deleted) безпосередньо у відповідну кімнату chat_${chatId}. Таким чином, WebSockets у системі виконують виключно роль швидкісного низхідного (downstream) каналу доставки сповіщень, що унеможливлює ситуацію, коли користувач отримує повідомлення у чаті, але воно ще не збережене на сервері.

Важливим інженерним рішенням у цій гібридній взаємодії є механізм синхронізації зі шаром кешування. Як зазначалося раніше, для прискорення завантаження історії діалогів система використовує in-memory сховище Redis. Оскільки поява нового повідомлення (або його видалення) робить попередній стан кешу неактуальним, подія трансляції через Socket.io супроводжується обов'язковою інвалідацією (очищенням) відповідних ключів у Redis. Сервер формує масив ключів вигляду chat:{chatId}:user:{userId}:messages для всіх учасників бесіди і виконує операцію видалення. Це гарантує ідеальну узгодженість даних: клієнти миттєво отримують нове повідомлення через сокет, а при наступному оновленні сторінки підвантажать свіжу історію вже з урахуванням останніх змін.

Покрокова візуалізація описаного алгоритму взаємодії компонентів системи під час відправки та трансляції повідомлення наведена на діаграмі послідовності (рис. 2.4).

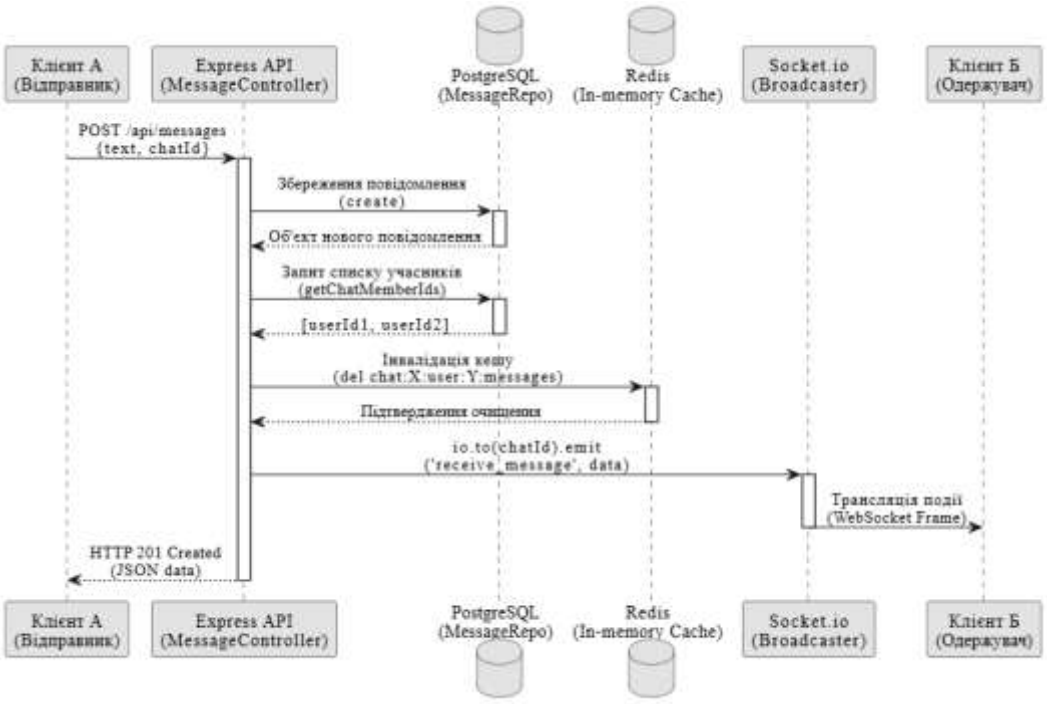


Рисунок 2.4 – Діаграма послідовності процесу збереження, кешування та трансляції повідомлення у реальному часі

2.5. Проєктування контейнеризованої інфраструктури шару даних (Docker, Redis, Metabase)

Сучасні підходи до розгортання програмного забезпечення вимагають відмови від ручного налаштування серверного середовища на користь автоматизованих рішень. Для забезпечення стабільної роботи бази даних та супутніх сервісів розроблюваного месенджера було застосовано концепцію «Інфраструктура як код» (Infrastructure as Code) з використанням інструментарію Docker Compose[33]. Це дозволило повністю ізолювати шар зберігання інформації від операційної системи хоста та гарантувати ідентичність середовища як на етапі локальної розробки, так і під час майбутнього розгортання на продуктивному сервері.

Уся конфігурація бекенд-інфраструктури декларативно описана у єдиному файлі, який оркеструє одночасний запуск та мережеву взаємодію трьох

незалежних контейнерів: реляційної бази даних, системи in-memory кешування та платформи для бізнес-аналітики.

Основним ядром збереження інформації виступає сервіс db, побудований на базі офіційного образу postgres:17-alpine. Вибір версії на основі операційної системи Alpine Linux є свідомим інженерним кроком, спрямованим на мінімізацію фізичного розміру контейнера та зменшення площі потенційних кібератак (attack surface) за рахунок відсутності зайвих системних утиліт. З міркувань безпеки, облікові дані (ім'я користувача, пароль та назва бази) не зафіксовані у конфігурації жорстко, а динамічно ін'єктуються через змінні середовища (Environment Variables) за допомогою файлу .env під час ініціалізації.

Критичним аспектом роботи будь-якої СУБД у контейнеризованому середовищі є забезпечення збереження стану (persistence). Оскільки Docker-контейнери за своєю природою є ефемерними і всі локальні зміни знищуються при їх перезапуску, для бази даних було спроєктовано іменований том (named volume) postgres_data. Він фізично монтується у внутрішню директорію /var/lib/postgresql/data, гарантуючи надійне збереження даних між сесіями. Крім того, для оптимізації процесу первинного розгортання застосовано механізм автоматичної міграції: локальний скрипт init_utf8.sql монтується до спеціальної директорії /docker-entrypoint-initdb.d/[34]. Це гарантує, що при першому запуску абсолютно «чистого» середовища СУБД автоматично виконає підготовчі скрипти та встановить правильне кодування UTF-8, що є критично важливим для коректного збереження кириличних текстових повідомлень.

Другим елементом інфраструктурного вузла є сервіс cache, реалізований на базі redis:7-alpine. Цей легковитратний контейнер прокидає стандартний порт 6379 назовні та виступає високошвидкісним буфером. Як зазначалося в описі архітектури WebSockets, саме Redis бере на себе навантаження з тимчасового зберігання часто запитуваних історій діалогів, розвантажуючи основний дисковий накопичувач бази даних під час масових розсилок.

Третім, і найбільш специфічним інфраструктурним рішенням, є сервіс metabase. Оскільки система вебмесенджера проєктувалася з глибокою інтеграцією ШІ, вона безперервно збирає великий обсяг телеметрії щодо згенерованих підказок (таблиці ai_analytics). Створення самописної адміністративної панелі для аналізу цих даних на React було б нераціональною витратою часу розробника. Натомість у мережу було додано контейнер з готовим BI-інструментом (Business Intelligence) Metabase. Для нього налаштовано прокидання портів 3001:3000 (щоб уникнути конфліктів із клієнтським застосунком на 3000 порту) та встановлено жорстку залежність depends_on: db. Це структурне правило гарантує, що аналітична платформа розпочне своє завантаження виключно після повної ініціалізації контейнера PostgreSQL, уникаючи фатальних помилок підключення на старті.

Для забезпечення високої доступності (High Availability) розробленої системи всім трьом сервісам призначено політику restart: always. Це означає, що Docker-демон автоматично відновить роботу всієї інфраструктури у разі програмних збоїв усередині контейнерів або після непередбачуваного перезавантаження фізичного хоста.

2.6. Опис бізнес-логіки інтеграції ШІ-асистента

Завершальним та найбільш інноваційним етапом проєктування серверної інфраструктури вебмесенджера є імплементація модуля ШІ. Сучасні тенденції розвитку комунікаційних платформ вимагають переходу від звичайного ретранслятора повідомлень до статусу розумного помічника. Для реалізації цього завдання розроблену систему було інтегровано з комерційним програмним інтерфейсом (API) великих мовних моделей (LLM).

Аналіз існуючих на ринку рішень показав, що для систем реального часу критичним фактором є не лише розмір бази знань моделі, але й швидкість генерації першого токена (Time to First Token) та загальна затримка відповіді (Latency). На основі цих критеріїв для інтеграції було обрано модель gemini-2.5-

flash від корпорації Google. Взаємодія з моделлю на рівні бекенду здійснюється через офіційний SDK (@google/generative-ai), що дозволяє інкапсулювати складність виконання HTTP-запитів до зовнішніх серверів всередині зручних асинхронних методів.

Архітектура інтелектуального модуля (AIController) поділяється на два основні функціональні напрямки: генерація контекстних підказок для швидкої відповіді (Smart Replies) та пряма взаємодія користувача з ШІ-асистентом (AskAI).

Найбільш складним з точки зору алгоритмізації є процес генерації підказок. Оскільки велика мовна модель не має прямого доступу до бази даних месенджера і не зберігає стан між запитами (Stateless), серверну частину застосунку спроєктовано таким чином, щоб виконувати роль оркестратора контексту. Коли клієнтський застосунок ініціює запит на генерацію підказок, він передає ідентифікатор поточного чату (chatId) та бажану глибину контексту (contextLength).

Алгоритм обробки цього запиту складається з кількох послідовних кроків. На першому етапі контролер звертається до репозиторію повідомлень (ChatRepository) і вилучає задану кількість останніх записів із бази даних. Отриманий масив сирих об'єктів форматується у читабельний текстовий вигляд за шаблоном [Ім'я відправника]: текст повідомлення. Це дозволяє нейромережі чітко розрізняти репліки різних учасників діалогу та розуміти загальну тональність бесіди.

Другим етапом є застосування технік промпт-інжинірингу (Prompt Engineering). Системна інструкція, яка передається моделі, містить жорсткі обмеження. Алгоритм вимагає від ШІ виступати в ролі «розумного помічника», аналізувати переданий текстовий зліпок чату та генерувати рівно три короткі, природні варіанти відповіді українською мовою. Критично важливою вимогою, зашитою в логіку бекенду, є формат повернення результату. Для того щоб фронтенд-застосунок міг коректно відобразити підказки у вигляді окремих

кнопок, модель проінструктовано повертати відповідь виключно у форматі валідного масиву JSON.

Третій етап полягає в обробці (санітаризації) отриманої відповіді. Оскільки генеративні моделі схильні обгортати код у Markdown-блоки (наприклад, ```json ... ```), у контролері імплементовано механізм очищення рядка за допомогою регулярних виразів перед тим, як функція JSON.parse перетворить текст на повноцінний масив даних.

Після успішної генерації та парсингу масиву відповідей, інформація не просто відправляється клієнту, а попередньо фіксується у базі даних (таблиця ai_suggestions) за допомогою AIRepository. Кожній згенерованій підказці присвоюється унікальний ідентифікатор бази даних, що формує фундамент для подальшого збору аналітики щодо релевантності роботи ШІ.

Другим, не менш важливим вектором застосування штучного інтелекту в системі, є модуль AskAI – повноцінний віртуальний учасник багатокористувацьких чатів. На відміну від генератора підказок, який працює у фоновому режимі, цей модуль дозволяє користувачам напряму звертатися до великої мовної моделі за допомогою явних запитів.

Процес обробки прямого запиту (askAiInChat) має низку архітектурних особливостей. Як і в попередньому випадку, система спочатку формує контекст, вилучаючи останні повідомлення бесіди. Проте ключовою інженерною знахідкою розробленого контролера є алгоритм динамічної алокації токенів (getOutputLimit). Оскільки тарифікація API мовних моделей залежить від обсягу згенерованого тексту, а надто довгі відповіді збільшують мережеву затримку, контролер виконує попередній семантичний аналіз запиту користувача за допомогою регулярних виразів.

Якщо запит містить ключові слова, що передбачають розгорнуту відповідь (наприклад, «рецепт», «поясни», «код», «детально»), система дозволяє моделі використати максимальний ліміт у 1024 токени. Якщо ж користувач просить розповісти «жарт», відповісти «коротко» або «одним словом», ліміт жорстко

обмежується до 200 токенів. Для всіх інших стандартних звернень встановлюється збалансоване значення у 400 токенів. Таке рішення дозволяє суттєво оптимізувати витрати на інфраструктуру ШІ та пришвидшити час відгуку інтерфейсу.

Після отримання згенерованого тексту від моделі, архітектура обробки цього повідомлення повністю інтегрується з базовим конвеєром месенджера. Відповідь ШІ зберігається у базу даних як звичайне повідомлення (за допомогою методу createAiMessage), після чого сервер автоматично виконує інвалідацію кешу в Redis для всіх учасників бесіди, гарантуючи консистентність даних. Фінальним етапом є трансляція цього повідомлення через WebSocket-канал (io.to().emit) усім підключеним клієнтам, завдяки чому відповідь ШІ миттєво з'являється на екранах без необхідності перезавантаження сторінки.

Останнім компонентом інтелектуальної підсистеми є модуль телеметрії та аналітики (trackSuggestionUsage). Для оцінки реальної ефективності згенерованих швидких підказок у системі реалізовано механізм відстеження користувацьких дій. Коли ШІ генерує три варіанти відповіді, вони записуються у таблицю ai_analytics зі статусом перегляду (Viewed). Якщо користувач обирає одну з підказок і натискає на неї, фронтенд надсилає спеціальний сигнал, який оновлює статус цієї конкретної транзакції у базі даних на корисний (useful = true). Такий підхід формує об'єктивний набір даних для подальшого розрахунку коефіцієнта конверсії (CTR) підказок і дає змогу в майбутньому донавчати або коригувати системні промпти мовної моделі.

Узагальнений алгоритм функціонування інтелектуального модуля, від моменту ініціації запиту до фінального збереження результатів, наведено у вигляді діаграми діяльності (Activity Diagram) на рис. 2.5.

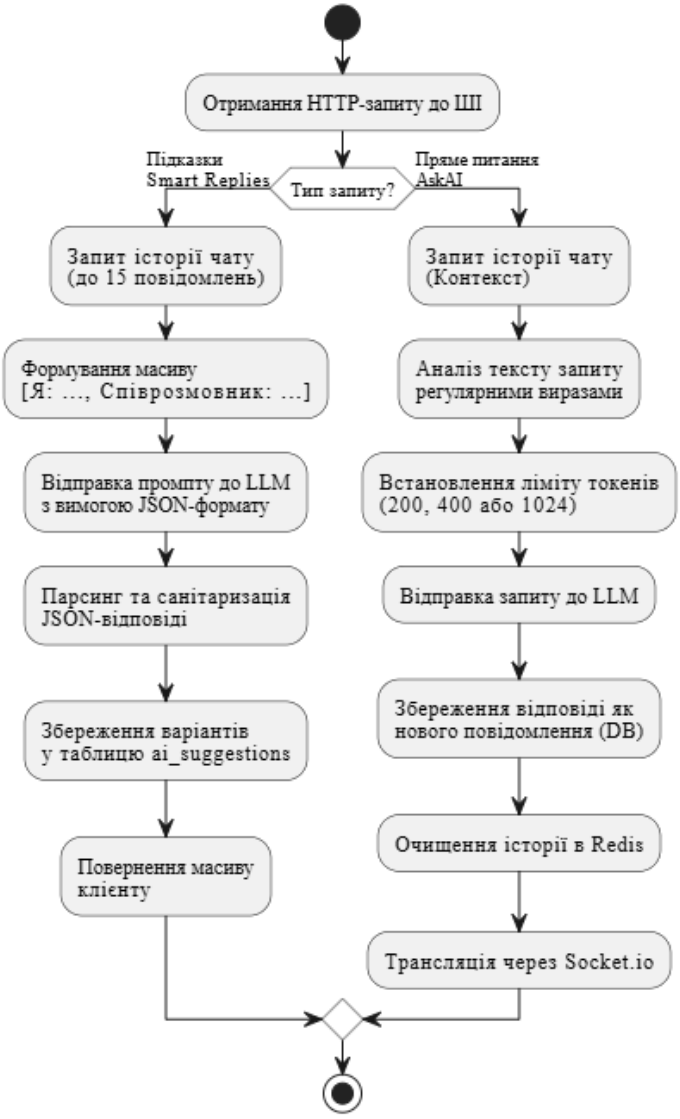


Рисунок 2.5 – Блок-схема (Activity Diagram) алгоритму взаємодії системи з API мовної моделі

Висновки до розділу 2

У другому розділі було виконано комплексне проєктування архітектури та програмної інфраструктури інтелектуального вебмесенджера. Перехід від етапу системного аналізу до безпосереднього інженерного конструювання дозволив сформувати детальний план реалізації кожного компонента, починаючи від шару зберігання даних і закінчуючи алгоритмами обробки запитів до великих мовних моделей.

На основі проведеного аналізу обґрунтовано вибір трирівневої клієнт-серверної архітектури. Доведено доцільність використання платформи Node.js як єдиної точки входу, що ефективно мультиплексує обробку класичних HTTP-запитів та постійних WebSocket-з'єднань на одному мережевому порту без залучення надлишкових проміжних проксі-серверів.

Розроблено оптимізовану логічну та фізичну структуру реляційної бази даних під управлінням СУБД PostgreSQL, яка повністю відповідає правилам третьої нормальної форми (3НФ). З метою забезпечення високої швидкодії та цілісності інформації було спроєктовано систему каскадних видалень, тригерних функцій та унікальних B-Tree індексів. Особливу увагу приділено обґрунтуванню гібридного підходу до зберігання даних: використання формату JSONB дозволило гнучко зберігати неструктуровані метадані ШІ без застосування важких GIN-індексів, що запобігло передчасній деградації продуктивності бази при масових операціях запису.

Програмний інтерфейс (REST API) системи спроєктовано за принципами багаторівневої абстракції з імплементацією архітектурного патерну «Репозиторій». Це гарантувало суворе розділення логіки маршрутизації, валідації вхідних параметрів та виконання SQL-транзакцій. Для захисту інформаційних потоків імплементовано надійний механізм Stateless-автентифікації на основі JWT-токенів, який інтегровано як у Middleware для HTTP-маршрутів, так і в процес мережевого рукостискання (Handshake) для WebSocket-протоколу.

Для забезпечення інтерактивності месенджера розроблено архітектуру гібридного обміну даними у реальному часі. Використання ізольованих кімнат (Rooms) на базі Socket.io дозволило застосувати патерн Publish-Subscribe та мінімізувати мережеве навантаження під час трансляції подій. Інтеграція in-memory кешування через Redis із механізмом транзакційної інвалідації під час надсилання повідомлень суттєво підвищила загальну відмовостійкість системи та пришвидшила завантаження історії діалогів.

Окремим науково-практичним досягненням етапу проєктування є розробка інфраструктури модуля штучного інтелекту. Створено алгоритми семантичного аналізу користувацьких запитів для динамічної алокації токенів, механізми формування діалогового контексту та систему збору телеметрії для оцінки конверсії (CTR) згенерованих підказок. Забезпечено надійну ізоляцію середовища через застосування підходу «Інфраструктура як код» (IaC) за допомогою Docker, що дозволило стандартизувати розгортання PostgreSQL, Redis та платформи бізнес-аналітики Metabase.

У підсумку, спроєктована архітектура позбавлена структурних вузьких місць і повністю задовольняє всі функціональні та нефункціональні вимоги до програмного продукту. Сформовані структури баз даних, специфікації ендпоінтів та блок-схеми взаємодії утворюють міцний інженерний фундамент, який слугує базою для безпосереднього написання програмного коду та подальшої розробки експлуатаційної документації.

РОЗДІЛ 3. РОЗРОБКА

3.1. Реалізація серверної частини на основі спроєктованого API

Безпосередня розробка програмного продукту розпочалася зі створення базової інфраструктури серверного застосунку (бекенду). Як було визначено на етапі проєктування, основою для реалізації слугує середовище виконання Node.js у поєднанні з мінімалістичним вебфреймворком Express. Такий технологічний стек дозволяє швидко розгортати REST-маршрути та гнучко керувати потоками даних через систему проміжних обробників (Middlewares).

Точкою входу в серверний застосунок є файл ініціалізації index.ts. Саме тут відбувається конфігурація глобальних параметрів сервера, підключення до бази даних та реєстрація всіх комунікаційних модулів. З міркувань безпеки та дотримання принципів конфігурації додатків за методологією «The Twelve-Factor App», усі чутливі дані (паролі до бази даних, секретні ключі для криптографії, порти) винесено за межі вихідного коду у файл .env та завантажуються в оперативну пам'ять за допомогою модуля dotenv.

Для забезпечення коректної взаємодії між клієнтським застосунком (який працює, наприклад, на порту 3000) та сервером (на порту 5000), у ядро Express було інтегровано модуль cors. Він налаштований таким чином, щоб приймати HTTP-запити лише з довірених доменів (origin) та дозволяти передачу авторизаційних заголовків. Для автоматичного перетворення вхідних текстових даних у JavaScript-об'єкти використовується вбудований парсер express.json().

Практична реалізація процесу ініціалізації ядра та монтування маршрутів наведена у лістингу 3.1.

Лістинг 3.1 – Ініціалізація базового сервера Express та реєстрація маршрутів (фрагмент файлу index.ts)

```
import express, { Application } from "express";
import cors from "cors";
import dotenv from "dotenv";
import authRoutes from "./routes/auth.js";
```

```
import chatRoutes from "./routes/chatsRoutes.js";
import messageRoutes from "./routes/messageRoutes.js";
import aiRoutes from "./routes/aiRoutes.js";

dotenv.config();

const app: Application = express();

app.use(cors({ origin: process.env.CLIENT_URL, credentials: true }));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.use("/public", express.static("public"));

app.use("/api/auth", authRoutes);
app.use("/api/chats", chatRoutes);
app.use("/api/messages", messageRoutes);
app.use("/api/ai", aiRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

Невід'ємною частиною розробленої архітектури є імплементація проміжного програмного забезпечення. Особливе інженерне значення має файл authMiddleware.ts. Оскільки HTTP є протоколом без збереження стану (stateless), кожен запит до захищених маршрутів (наприклад, створення чату або відправка повідомлення) повинен бути авторизованим.

Функція authMiddleware виконує роль бар'єра. Вона перехоплює вхідний запит до того, як він потрапить до контролера, і перевіряє наявність заголовка Authorization. За допомогою бібліотеки jsonwebtoken система верифікує криптографічний підпис токена. Якщо підпис валідний і термін дії токена не вичерпано, алгоритм розшифровує корисне навантаження (payload) та прикріплює ідентифікатор користувача безпосередньо до об'єкта запиту (req.user = decoded). Це дозволяє уникнути повторних звернень до бази даних для

ідентифікації клієнта. У разі відсутності токена або його фальсифікації, виконання переривається, а клієнт отримує статус-код 401 Unauthorized.

Ще одним специфічним перехоплювачем є модуль обробки файлових завантажень, реалізований за допомогою бібліотеки multer (файл upload.ts). Він застосовується виключно на маршрутах оновлення профілю для завантаження аватарів. Обробник налаштовано на буферизацію вхідного потоку типу multipart/form-data, валідацію MIME-типів (блокуючи будь-які файли, окрім .jpg, .png, .webp) та збереження зображень у фізичну директорію сервера з генерацією унікального імені на базі поточної мітки часу.

Алгоритм процесу обробки захищеного запиту та роль проміжного програмного забезпечення наочно зображено на діаграмі діяльності (рис. 3.1).

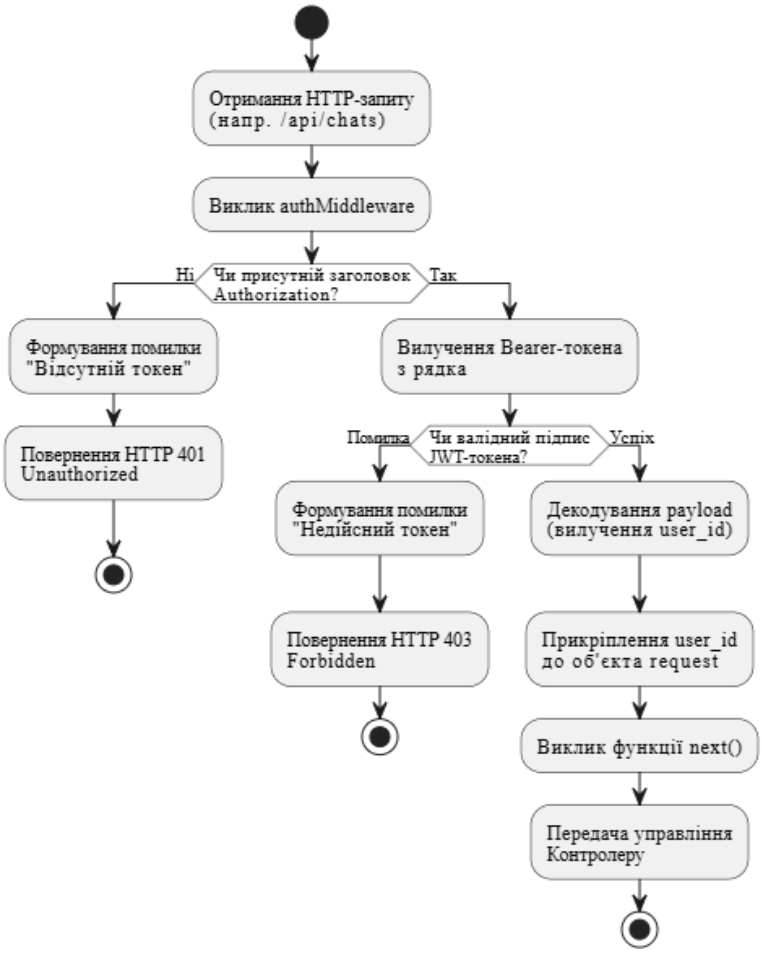


Рисунок 3.1 – Блок-схема процесу авторизації HTTP-запиту через authMiddleware

3.2. Імплементація патерну Repository для управління даними

Після розгортання базової інфраструктури вебсервера наступним критично важливим етапом розробки стала реалізація шару доступу до даних (Data Access Layer). Як зазначалося в розділі проєктування, для інкапсуляції логіки взаємодії з базою даних застосовано архітектурний патерн «Репозиторій». Такий підхід дозволив повністю ізолювати бізнес-логіку контролерів від специфіки синтаксису конкретної системи управління базами даних (СУБД).

Фізична взаємодія з PostgreSQL реалізована за допомогою низькорівневого драйвера pg (node-postgres). З метою оптимізації продуктивності системи та раціонального використання ресурсів сервера, замість встановлення індивідуального з'єднання для кожного HTTP-запиту було імплементовано механізм пулу з'єднань (Connection Pool). У файлі db.ts створюється глобальний об'єкт пулу, який підтримує набір відкритих TCP-з'єднань у режимі очікування. Це усуває накладні витрати часу на так зване мережеве рукостискання (Handshake) під час інтенсивного обміну повідомленнями.

Прикметною особливістю імплементованого шару даних є свідома відмова від використання важких систем об'єктно-реляційного відображення (ORM) для ключових транзакцій. Використання нативних SQL-запитів забезпечило розробнику абсолютний контроль над планами виконання (Query Execution Plans). Це виявилося особливо актуальним під час роботи зі специфічним типом даних JSONB, який використовується для збереження аналітики штучного інтелекту. Відсутність проміжного шару абстракції ORM дозволила уникнути надлишкової серіалізації об'єктів та забезпечила максимальну швидкість вибірки даних.

Логічна структура взаємодії між компонентами під час обробки запиту наведена на UML-діаграмі класів (рис. 3.2).

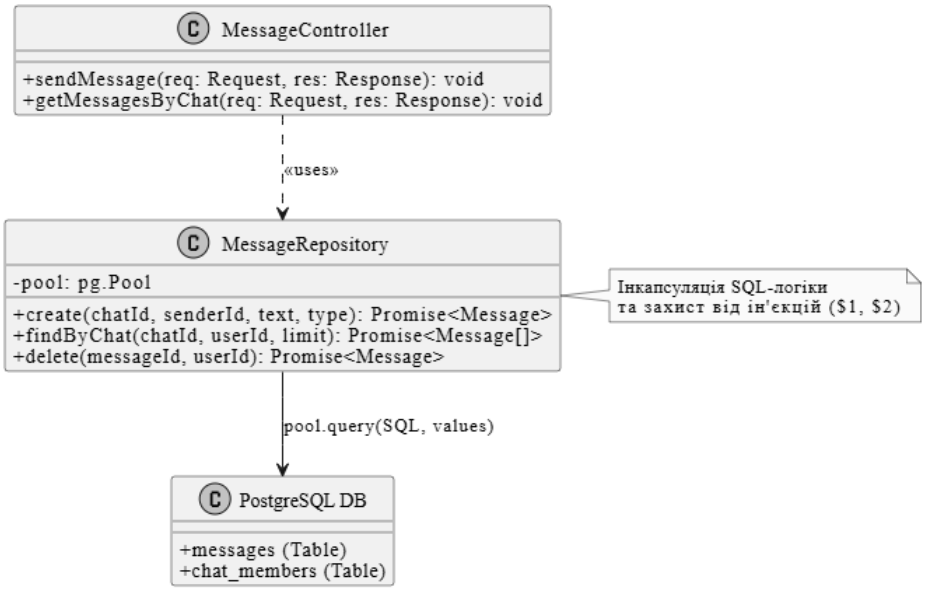


Рисунок 3.2 – Діаграма класів шару доступу до даних та бізнес-логіки

Головним пріоритетом під час розробки класів-репозиторіїв (таких як MessageRepository, ChatRepository, UserRepository) стало гарантування інформаційної безпеки. Оскільки месенджер обробляє неструктурований користувацький контент, існує постійний ризик проведення атак типу впровадження SQL-коду (SQL Injection).

Для нейтралізації цієї загрози всі транзакції в системі реалізовані виключно через механізм параметри[i]зованих запитів (Parameterized Queries). Алгоритм роботи драйвера налаштований таким чином, що структура SQL-команди та користувацькі дані передаються на сервер СУБД окремими пакетами. Використання позиційних плейсхолдерів формату $1, $2, $3 переносить задачу екранування потенційно небезпечних символів (наприклад, одинарних лапок чи крапок з комою) безпосередньо на ядро PostgreSQL.

Практичну імплементацію описаних принципів роботи з базою даних відображено в лістингу 3.2, який демонструє фрагмент коду класу MessageRepository. Метод create відповідає за збереження нового повідомлення з одночасним поверненням згенерованих базою даних системних полів (ідентифікатора та часової мітки) через оператор RETURNING. Метод findByChat демонструє складну вибірку з використанням оператора JOIN для

перевірки прав доступу (користувач має бути учасником чату) та лімітування результатів для оптимізації пагінації.

Лістинг 3.2 – Реалізація методів доступу до даних у класі MessageRepository

```
import pool from "../db.js";

export class MessageRepository {

    async create(chatId: number, senderId: number, text: string, type: string =
"text") {
        const query = `
            INSERT INTO messages (chat_id, sender_id, text, type)
            VALUES ($1, $2, $3, $4)
            RETURNING id, chat_id, sender_id, text, type, created_at;
        `;
        const values = [chatId, senderId, text, type];
        const result = await pool.query(query, values);

        return result.rows[0];
    }

    async findByChat(chatId: number, userId: number, limit: number = 30) {
        const query = `
            SELECT m.* FROM messages m
            JOIN chat_members cm ON cm.chat_id = m.chat_id
            WHERE m.chat_id = $1 AND cm.user_id = $2
            ORDER BY m.created_at DESC
            LIMIT $3;
        `;
        const result = await pool.query(query, [chatId, userId, limit]);

         return result.rows;
     }
}
```

3.3. Розробка інтерфейсу користувача та інтеграція з бекендом

Після успішного проєктування та імплементації серверної інфраструктури, наступним етапом розробки стало створення інтерактивної клієнтської частини. Для реалізації фронтенду було обрано бібліотеку React, яка дозволяє будувати інтерфейси за принципом односторінкового застосунку (Single Page Application, SPA). Такий підхід мінімізує кількість повних перезавантажень сторінки у браузері користувача, оскільки завантаження та рендеринг компонентів відбувається динамічно за допомогою технології Virtual DOM.

Маршрутизація застосунку імплементована за допомогою бібліотеки react-router-dom. У головному файлі App.jsx налаштовано декларативний поділ на публічні та приватні маршрути. Доступ до ключового функціоналу (маршрут /chats) жорстко обмежено за допомогою компонента-обгортки ProtectedRoute. Він перехоплює спроби несанкціонованого доступу та автоматично перенаправляє неавторизованих користувачів на сторінку входу, гарантуючи безпеку клієнтського середовища.

Одним із найскладніших завдань під час розробки великих React-застосунків є управління глобальним станом (Global State Management). Передача даних через багаторівневу ієрархію компонентів за допомогою пропсів (так званий «prop drilling») призводить до створення сильно зв'язаного та важко підтримуваного коду. Аналіз вимог до месенджера показав, що впровадження сторонніх екосистем, таких як Redux, є недоцільним, оскільки це невиправдано збільшило б розмір фінального бандлу (bundle size) та ускладнило архітектуру. Натомість було прийнято рішення базувати управління станом на нативному механізмі React – Context API.

Ядром системи управління станом виступає UserContext.jsx. Цей провайдер глобально обгортає все дерево застосунку і відповідає за збереження даних авторизованого профілю. Логіка контексту інкапсулює асинхронну функцію checkAuth, яка під час ініціалізації (через хук useEffect) самостійно перевіряє валідність локального JWT-токена шляхом фонового запиту до REST

API. Крім того, саме на рівні UserContext реалізовано механізм динамічного перемикання візуальних тем оформлення. Зміна атрибута data-theme на рівні тегу <html> відбувається реактивно на основі налаштувань, збережених у профілі користувача (наприклад, тема «glass»).

Окрім профілю, архітектура передбачає просторову ізоляцію даних чатів через ChatContext та мережевих підключень через SocketContext. Такий модульний підхід гарантує, що зміна стану, наприклад, отримання нового повідомлення, викличе перемальовування лише релевантних компонентів, а не всього інтерфейсу.

Безпосередня візуальна взаємодія відбувається на сторінці ChatsPage.jsx. Архітектурно ця сторінка декомпозована на два великі незалежні блоки: бічну панель навігації (Sidebar) та вікно активного діалогу (ChatWindow). Управління локальним станом на цій сторінці зведено до мінімуму (зберігається лише selectedChatId, filterType та searchQuery), що дозволяє легко прокидати ці параметри у дочірні компоненти. Дерево компонентів та напрямки потоків даних наочно представлені на UML-діаграмі (рис. 3.3).

Для організації стабільної та безпечної мережевої взаємодії між клієнтським застосунком та серверною частиною було використано бібліотеку Axios. Хоча сучасні браузери мають вбудований інтерфейс fetch, вибір Axios був зумовлений його ширшими можливостями для автоматизації рутинних завдань, нативною підтримкою переривання запитів та, що найважливіше, наявністю механізму перехоплювачів (interceptors).

Центральним модулем управління всіма HTTP-запитами є файл axiosInstance.js. У ньому створюється спільний екземпляр клієнта, де зафіксовано базовий шлях до API (baseURL). Таке рішення дозволяє уникнути хардкодингу повних URL-адрес у кожному окремому сервісі застосунку, що значно спрощує подальше розгортання системи на різних доменах або серверах.

Ключовим інженерним рішенням у реалізації клієнтського доступу стало впровадження перехоплювача запитів (Request Interceptor). Оскільки месенджер

використовує Stateless-авторизацію на основі JWT-токенів, кожен запит до захищених даних повинен містити підтвердження особи користувача. Замість того, щоб вручну додавати токен у кожну функцію запиту, було реалізовано автоматизований конвеєр.

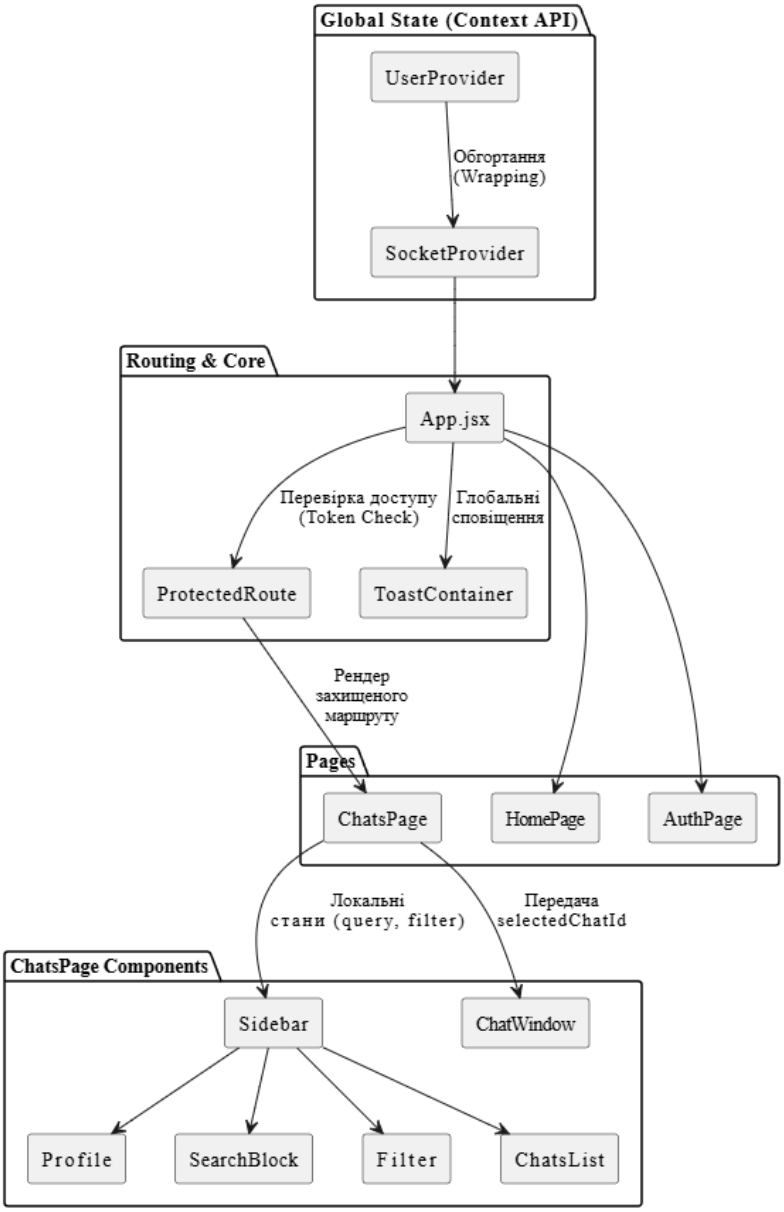


Рисунок 3.3 – Діаграма дерева компонентів

Алгоритм роботи перехоплювача полягає у наступному: перед тим, як будь-який запит вийде за межі браузера, функція-обробник звертається до локального сховища (localStorage) та намагається отримати збережений ключ доступу. У разі

успіху, перехоплювач динамічно модифікує об'єкт конфігурації запиту, впорскуючи у нього заголовок Authorization за стандартом Bearer. Це гарантує, що сервер отримає валідний підпис доступу у кожному виклику, а розробник позбувається необхідності дублювати логіку авторизації в окремих методах.

Практичну реалізацію конфігурації мережевого клієнта відображено у лістингу 3.3.

Лістинг 3.3 – Налаштування екземпляра Axios та перехоплювача для управління JWT-токенами

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: 'http://localhost:5000/api',
});
axiosInstance.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);
export default axiosInstance;
```

Використання такого механізму забезпечує високий рівень абстракції. Будь-які сервісні модулі, такі як chatsService.js або userService.js, імпортують уже налаштований axiosInstance і виконують запити так, ніби вони авторизовані за замовчуванням. Це не лише робить код чистішим і лаконічнішим, а й створює єдину точку входу для глобальної обробки помилок мережі (наприклад, для автоматичного розлогінювання користувача у разі отримання коду відповіді 401 Unauthorized).

Наступним архітектурним завданням на рівні клієнтського застосунку стала реалізація механізму двостороннього зв'язку в режимі реального часу. Оскільки стандартизований REST API не здатний забезпечити миттєву доставку повідомлень без надмірних накладних витрат на постійні HTTP-опитування (Long Polling), фронтенд-частину було інтегровано з протоколом WebSockets за допомогою клієнтської бібліотеки socket.io-client.

Для забезпечення глобального доступу до єдиного відкритого з'єднання з будь-якого компонента системи без використання багатоступеневої передачі через пропси (уникнення антипатерну prop-drilling), логіку роботи з сокетами було інкапсульовано у спеціалізований провайдер SocketContext.jsx. Інженерною особливістю цієї реалізації є те, що життєвий цикл підключення жорстко синхронізовано зі станом авторизації користувача, який експортується з UserContext.

Алгоритм роботи контексту побудовано таким чином: при успішній авторизації користувача в системі спрацьовує хук useEffect, який ініціалізує нове мережеве підключення до бекенду. Під час ініціалізації клієнт автоматично передає конфігураційні параметри (наприклад, конфігурацію повторного підключення) та авторизаційний токен. Це дозволяє серверній частині ідентифікувати сесію на етапі протокольного рукостискання (Handshake). У разі втрати авторизації або ручного виходу з профілю спрацьовує функція очищення пам'яті (cleanup function). Вона примусово закриває з'єднання методом socket.disconnect(), запобігаючи накопиченню «мертвих» підключень та витокам оперативної пам'яті (memory leaks) у браузері користувача.

Сам процес синхронізації інтерфейсу з новими даними реалізується через реактивну підписку на події. Коли користувач відкриває певний діалог, компонент вікна чату прослуховує серверні трансляції (зокрема подію receive_message). Для того щоб нове повідомлення коректно відобразилося в інтерфейсі без перезавантаження сторінки, React-компонент використовує функціональне оновлення локального стану (functional state update). Це гарантує,

що нове повідомлення буде безпомилково додано до актуального масиву даних, навіть якщо попередній стан ще перебуває в процесі рендерингу, що остаточно вирішує проблему застарілих замикань (stale closures).

Фрагмент коду, що демонструє організацію глобального провайдера сокет-з'єднань та управління його життєвим циклом, наведено у лістингу 3.4.

Лістинг 3.4 – Реалізація глобального контексту SocketContext для управління WebSocket-з'єднаннями

```
import { createContext, useEffect, useState, useContext } from 'react';
import { io } from 'socket.io-client';
import { UserContext } from './UserContext';

export const SocketContext = createContext(null);

export const SocketProvider = ({ children }) => {
  const [socket, setSocket] = useState(null);
  const { user } = useContext(UserContext); // Отримання стану авторизації

  useEffect(() => {
    if (user) {
      const newSocket = io("http://localhost:5000", {
        auth: { token: localStorage.getItem('token') },
        reconnection: true,
        reconnectionAttempts: 5,
        transports: ['websocket']
      });

      setSocket(newSocket);

      return () => {
        newSocket.disconnect();
      };
    }
  }, [user]);

  return (
    <SocketContext.Provider value={{ socket }}>
      {children}
    </SocketContext.Provider>
```

```
  );
};
```

Впровадження такого патерну зробило клієнтську архітектуру максимально гнучкою: довільні компоненти інтерфейсу (список діалогів у бічній панелі або віджет зміни мережевого статусу) можуть імпортувати хук useContext(SocketContext) і миттєво підписуватися на специфічні події, не порушуючи загальної ієрархії застосунку.

Завершальним етапом імплементації клієнтської архітектури стала просторова декомпозиція інтерфейсу та впровадження ізольованої системи стилізації. З метою дотримання принципу єдиної відповідальності (Single Responsibility Principle) та забезпечення високого рівня підтримуваності (maintainability) коду, складні візуальні екрани розбито на дрібні, гранульовані компоненти.

Показовим прикладом такого підходу є структура головного комунікаційного інтерфейсу – ChatsPage.jsx. Цю сторінку спроєктовано за класичним патерном «Майстер-Деталь» (Master-Detail). Ліва частина екрана, інкапсульована в компонент Sidebar, відповідає за навігацію, пошук, фільтрацію та відображення списку доступних діалогів. Права робоча зона використовує логіку умовного рендерингу (Conditional Rendering), залежно від поточного стану застосунку. Якщо користувач ще не обрав конкретну бесіду (змінна selectedChatId має значення null), система рендерить легкий компонент-заглушку NoChatSelected, який містить базові інструкції для початку роботи. Щойно відбувається вибір діалогу, у DOM-дерево монтується важкий компонент ChatWindow, який ініціює HTTP-запит для завантаження історії повідомлень та паралельно налаштовує специфічні слухачі WebSocket-подій.

Окремим інженерним викликом під час розробки великих односторінкових застосунків є управління каскадними таблицями стилів. Традиційне використання глобальних CSS-файлів неминуче призводить до конфліктів імен

класів (namespace collisions) та так званих війн специфічності (specificity wars), коли стилі одного компонента непередбачувано перекривають стилі іншого.

Для вирішення цієї проблеми у розробленому месенджері застосовано технологію CSS Modules у поєднанні з препроцесором SCSS (файли формату .module.scss). Цей підхід кардинально змінює парадигму роботи зі стилями: розробник пише стандартні класи, проте під час збірки проєкту (з використанням бандлера Vite) компілятор перетворює їх на унікальні хешовані ідентифікатори (наприклад, з .container генерується .Chats_container__aB1cD). Це гарантує стовідсоткову інкапсуляцію візуальних правил на рівні конкретного React-компонента і повністю виключає їх вплив на глобальну область видимості.

Крім того, застосування синтаксису SCSS дозволило імплементувати вкладені селектори (nesting), міксини (mixins) та створити глобальний файл _themes.scss. У цьому файлі задекларовано набір CSS-змінних для управління кольоровими палітрами, що дало змогу безперешкодно реалізувати функціонал динамічного перемикання тем (світлої, темної та кастомної) на рівні всього застосунку через маніпуляцію атрибутом data-theme основного HTML-документа.

3.4. Реалізація черг фонових задач за допомогою Redis

Однією з фундаментальних архітектурних особливостей платформи Node.js є виконання коду в єдиному основному потоці (Single Thread) з використанням неблокуючого циклу подій (Event Loop). Ця модель є високоефективною для обробки великої кількості одночасних мережевих WebSocket-з'єднань, однак вона виявляє суттєві обмеження під час виконання тривалих операцій або важких завдань вводу-виводу (I/O).

У контексті розроблюваної системи однією з таких ресурсоємних задач є взаємодія із зовнішніми SMTP-серверами для відправки транзакційних електронних листів (верифікація нових акаунтів, надсилання токенів для скидання пароля). Якби логіка формування та відправки листа виконувалася

синхронно безпосередньо у контролері (наприклад, під час обробки маршруту /api/auth/register), основний потік сервера був би заблокований на весь час мережевого очікування відповіді від поштового провайдера. Це призвело б до недопустимої затримки в обробці повідомлень реального часу для всіх інших активних користувачів месенджера.

Для вирішення цієї інженерної проблеми було імплементовано архітектурний патерн «Черга повідомлень» (Message Queue), який дозволяє делегувати виконання побічних завдань у фонові процеси. Технічною основою для реалізації системи черг виступило in-memory сховище Redis, яке завдяки своїй продуктивності ідеально підходить для ролі брокера повідомлень, у поєднанні з бібліотекою управління фоновими задачами (BullMQ).

Алгоритм оптимізованої роботи виглядає наступним чином. Коли система ініціює подію, що вимагає відправки листа, контролер лише формує об'єкт із необхідним корисним навантаженням (адреса отримувача, тип шаблону, згенерований URL з токеном) і передає його до модуля emailQueue. Цей модуль миттєво серіалізує дані та зберігає їх як нову задачу (Job) у Redis. Після цього контролер негайно завершує виконання і повертає клієнту HTTP-відповідь 200 OK або 201 Created. Фактична відправка листа на цьому етапі ще не відбулася, але клієнтський застосунок уже може продовжувати роботу без зависань.

Величезною перевагою використання спеціалізованої бібліотеки черг є вбудована відмовостійкість. Під час ініціалізації черги для кожної задачі налаштовуються параметри повторного виконання (Retries). У разі тимчасової недоступності SMTP-сервера або обриву з'єднання, система не втратить критично важливий лист, а автоматично поверне задачу в чергу і спробує виконати її знову через визначений проміжок часу з використанням алгоритму експоненційної затримки (Exponential Backoff).

Практичну реалізацію ініціалізації черги та методу публікації задач відображено у лістингу 3.5.

Лістинг 3.5 – Ініціалізація інстансу черги та публікація задач (фрагмент файлу emailQueue.ts)

```
import { Queue } from 'bullmq';
import redisClient from '../redisClient.js';

export const emailQueue = new Queue('emailNotifications', {
    connection: redisClient,
    defaultJobOptions: {
        attempts: 3,
        backoff: {
            type: 'exponential',
            delay: 2000,
        },
        removeOnComplete: true,
    }
});

export const addEmailJob = async (type: string, payload: any): Promise<void> =>
{
    try {
        await emailQueue.add(type, payload);
        console.log(`[Queue] Задача типу ${type} успішно додана до Redis`);
    } catch (error) {
        console.error(`[Queue Error] Помилка додавання задачі:`, error);
        throw new Error('Не вдалося поставити задачу в чергу');
    }
};
```

Після створення механізму публікації завдань (Publish) виникає потреба в імплементації підсистеми їх виконання. У розробленій архітектурі цю роль відіграє процес-обробник (Worker), реалізований у модулі emailWorker.ts. Він встановлює незалежне з'єднання з інстансом Redis і функціонує в режимі безперервного прослуховування черги (Long Polling).

Щойно контролер або інший модуль публікує нову задачу (наприклад, надсилання посилання для відновлення доступу), Worker автоматично вилучає її з оперативної пам'яті (операція Dequeue) та розпочинає обробку. Для безпосередньої взаємодії з поштовими серверами у Worker інтегровано

бібліотеку Nodemailer. Вона формує тіло електронного листа на основі шаблонів, додає заголовки та ініціює захищену SMTP/TLS транзакцію. Оскільки цей процес виконується у фоновому контексті, навіть значні затримки мережі або тайм-аути з боку поштового провайдера жодним чином не впливають на пропускну здатність (Throughput) основного HTTP-сервера, який продовжує миттєво обробляти запити інших користувачів.

Окрім реактивних завдань, що генеруються внаслідок прямих дій клієнта, сучасний месенджер вимагає імплементації проактивного функціоналу. Прикладом такого функціоналу є система автоматичних нагадувань про пропущені діалоги. Для автоматизації цього процесу було використано механізм планування завдань, реалізований у файлах cron.ts та unreadChecker.ts.

Модуль планувальника (Scheduler) базується на використанні утиліти, подібної до класичного UNIX-демона cron. Використовуючи декларативний синтаксис часових виразів, сервер налаштовано на періодичний запуск (наприклад, кожні 15 хвилин) функції-ревізора unreadChecker. Ця функція виконує агрегований SQL-запит до реляційної бази даних PostgreSQL. Алгоритм звертається до таблиць messages, chat_members та user_statuses, відфільтровуючи лише ті повідомлення, які не мають позначки про прочитання понад заданий ліміт часу, і отримувачі яких наразі перебувають у статусі офлайн.

Для кожного знайденого користувача планувальник формує зведений масив даних і, замість самостійної відправки листів, передає цю інформацію як нову задачу до вже існуючої Redis-черги emailNotifications. Такий підхід демонструє високий рівень повторного використання коду (Code Reusability) та гарантує, що всі масові розсилки проходять через єдиний контрольований конвеєр із налаштованими лімітами швидкості (Rate Limiting) та захистом від перевантаження SMTP-сервера.

Повний життєвий цикл фонової задачі, що охоплює обидва вектори ініціації (від API та від планувальника), а також етапи обробки у Redis, наочно представлено на діаграмі послідовності (рис 3.4).

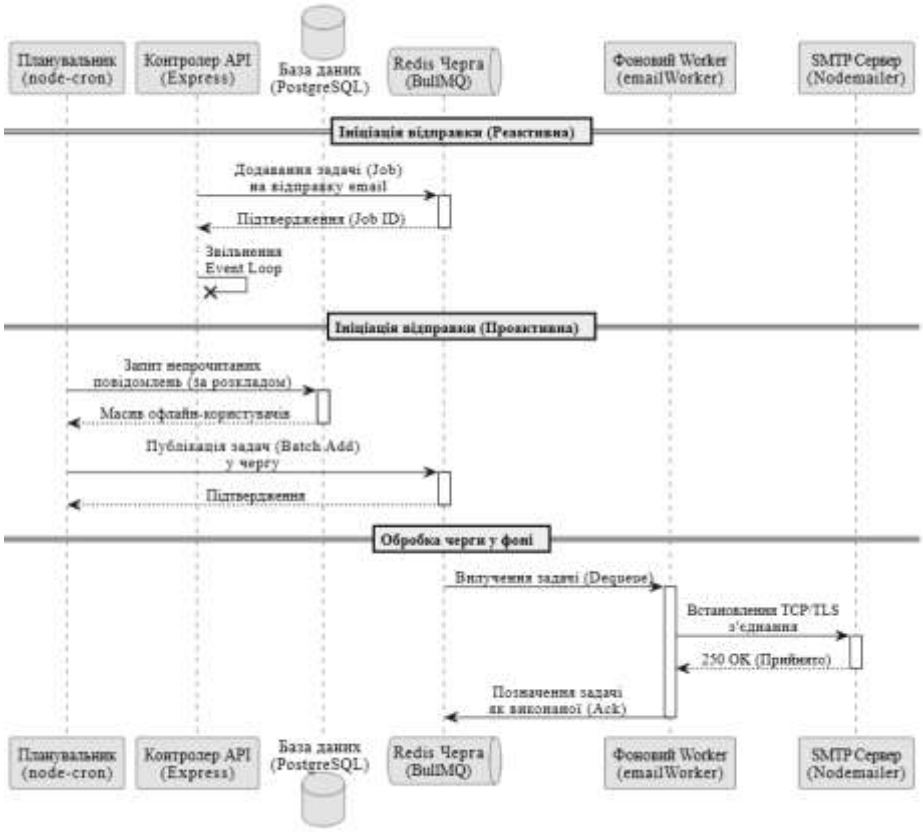


Рисунок 3.4 – Діаграма послідовності життєвого циклу фонових завдань та роботи планувальника

3.5. Розробка модуля взаємодії з ШІ та системи збору аналітики

Завершальним та найбільш інноваційним етапом практичної реалізації вебмесенджера стала розробка інтелектуального модуля на базі великої мовної моделі (LLM). Цей функціонал декомпозовано на два основні клієнтські інтерфейси: генератор контекстних підказок (Smart Replies) та інтерактивний чат-бот (AskAI). Обидва модулі безпосередньо інтегровані у компонент робочої області діалогу (ChatWindow), проте використовують різні архітектурні підходи для взаємодії з бекендом.

Першим було імплементовано механізм швидких відповідей, за який відповідає клієнтський компонент AIGenerateSuggestions.jsx. Його головне завдання – мінімізувати когнітивне навантаження на користувача та

пришвидшити процес ведення типових діалогів. При активації цієї функції фронтенд ініціює асинхронний HTTP-запит до серверного маршруту /api/ai/suggest, передаючи ідентифікатор поточного чату.

На серверній стороні (у контролері AIController) застосовано бібліотеку @google/generative-ai для зв'язку з API моделі gemini-2.5-flash. Для того щоб нейромережа могла згенерувати релевантні підказки, сервер динамічно вилучає останні повідомлення з бази даних і формує текстовий контекст. Критично важливим етапом розробки стало забезпечення стабільності формату вихідних даних. Оскільки мовні моделі генерують неструктурований текст, до системного промпту було додано жорстку інструкцію повертати результат виключно у форматі валідного масиву JSON[35]. Крім того, на рівні бекенду імплементовано механізм санітаризації за допомогою регулярних виразів, який очищує відповідь моделі від технічних Markdown-тегів (наприклад, ```json) перед етапом серіалізації.

Після успішної обробки на сервері, масив підказок повертається на клієнт, де React динамічно рендерить їх у вигляді інтерактивних кнопок над полем введення повідомлення. Візуальне представлення цього функціоналу наведено на рис. 3.5.

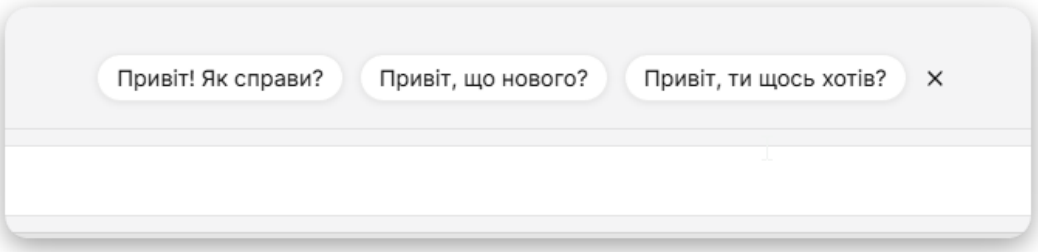


Рисунок 3.5 – Інтерфейс генерації швидких підказок у вікні чату

Практичну реалізацію клієнтського сервісу для отримання ШІ-підказок відображено у лістингу 3.6. У цьому файлі інкапсульовано логіку звернення до захищеного API за допомогою налаштованого екземпляра Axios.

Лістинг 3.6 – Сервісний – модуль для роботи з ШІ-ендпоінтами (фрагмент файлу aiService.js)

```
import axiosInstance from './axiosInstance';

const aiService = {

  generateSuggestions: async (chatId) => {
    try {
      const response = await axiosInstance.post('/ai/suggest', {
        chatId: chatId,
        context_length: 15
      });

      return response.data;
    } catch (error) {
      console.error("AI Service Error: Failed to generate suggestions", error);
      throw error;
    }
  },
};

export default aiService;
```

Другим, більш інтерактивним напрямком використання великої мовної моделі є модуль віртуального асистента, реалізований у компоненті AskAI.jsx. На відміну від фонової генерації підказок, цей інструмент функціонує як повноцінний учасник діалогу. Інтерфейс компонента передбачає наявність окремого текстового поля для введення спеціалізованих запитів.

При активації функції askAiInChat, клієнтський застосунок формує HTTP-запит до маршруту /api/ai/ask, передаючи поточний контекст бесіди та безпосередній промпт користувача. Як було описано в розділі проєктування, серверна архітектура динамічно визначає ліміт токенів на основі складності запиту. Після отримання відповіді від моделі Gemini, згенерований текст не просто повертається клієнту, а проходить повний цикл обробки стандартного

повідомлення: зберігається в таблицю messages у PostgreSQL, інвалідує відповідні ключі в Redis-кеші та транслюється всім учасникам діалогу через подію WebSocket receive_message. Такий архітектурний підхід гарантує повну синхронізацію стану між усіма підключеними клієнтами.

Візуальне представлення роботи віртуального асистента наведено на рисунку 3.6.

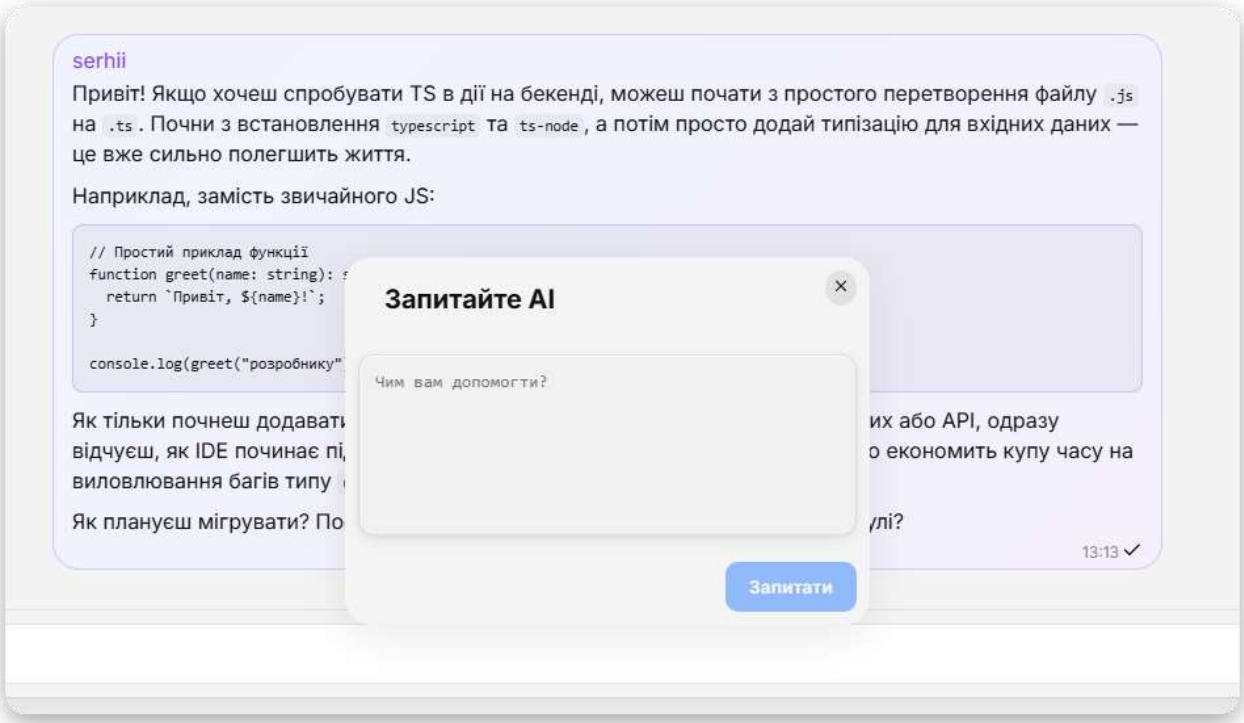


Рисунок 3.6 – Взаємодія користувача з віртуальним асистентом AskAI

Для забезпечення постійного покращення якості взаємодії з мовною моделлю та оптимізації фінансових витрат на використання API, систему було оснащено модулем телеметрії. Ключовим завданням цього модуля є обчислення коефіцієнта корисності (Click-Through Rate, CTR) для згенерованих підказок Smart Replies.

Процес збору аналітики імплементовано за принципом фонової реєстрації подій. Коли ШІ-модуль генерує три варіанти швидкої відповіді, вони записуються у базу даних (таблиця ai_analytics) з базовим статусом перегляду.

Кожній кнопці в інтерфейсі React динамічно присвоюється унікальний ідентифікатор (suggestion_id), отриманий від сервера.

Якщо користувач обирає одну з підказок та натискає на неї, відбувається паралельне виконання двох процесів. Перший процес – це стандартна відправка обраного тексту як звичайного повідомлення у чат. Другий процес – це ініціація прихованого асинхронного виклику до аналітичного маршруту /api/ai/analytics. Контролер перехоплює цей запит і оновлює відповідний запис у таблиці, змінюючи булеве значення поля useful на true. Завдяки цьому формується репрезентативна вибірка даних, яка чітко розмежовує проігноровані генерації та ті, що принесли реальну користь користувачеві.

Алгоритм фіксації метрик корисності ШІ-підказок деталізовано на UML-діаграмі діяльності (рис. 3.7).

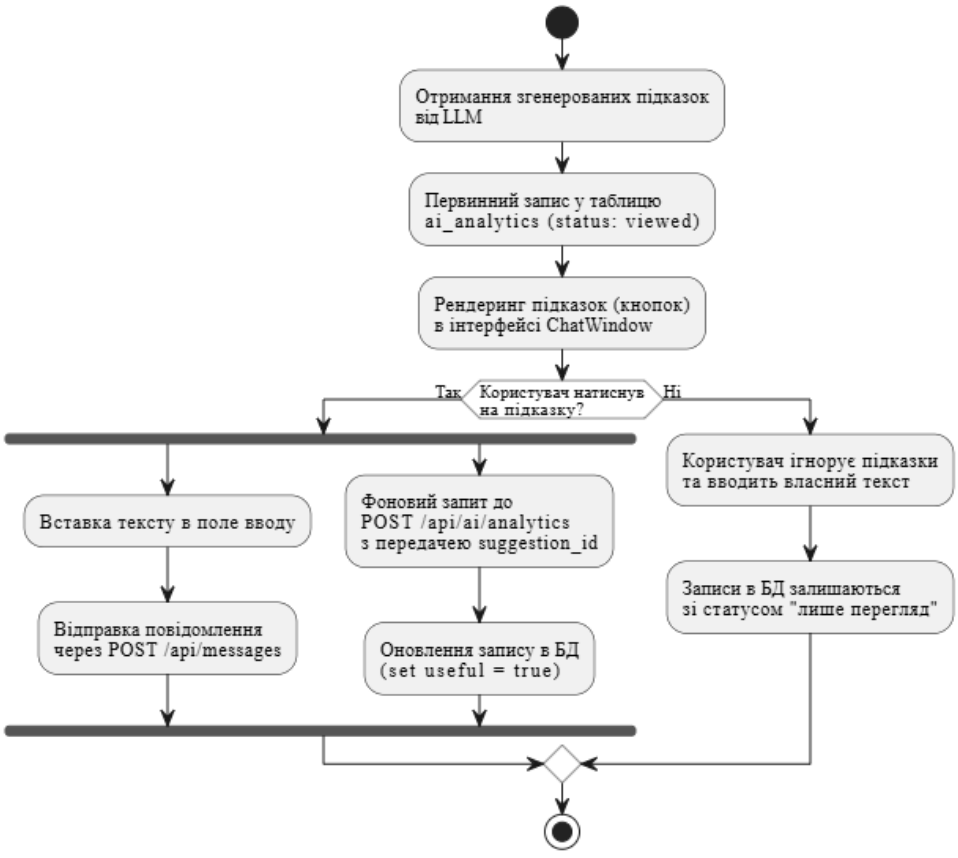


Рисунок 3.7 – Блок-схема процесу збору телеметрії та реєстрації корисності ШІ-підказок

Для забезпечення комплексного моніторингу роботи інтелектуального модуля та прийняття обґрунтованих рішень щодо його подальшого вдосконалення, систему було доповнено платформою бізнес-аналітики (Business Intelligence). Розробка власної адміністративної панелі з нуля потребувала б значних часових ресурсів на проєктування графіків та агрегацію масивів даних. Тому, керуючись принципами раціональної програмної інженерії, в інфраструктуру було інтегровано готове спеціалізоване рішення – систему Metabase.

Як зазначалося під час опису інфраструктури у другому розділі, цей сервіс розгортається у власному Docker-контейнері та прослуховує ізольований мережевий порт 3001. Завдяки прямому підключенню до бази даних PostgreSQL у режимі лише для читання (Read-Only), адміністратор системи отримує можливість будувати динамічні дашборди безпосередньо на основі сирих даних з таблиці ai_analytics. Такий архітектурний підхід повністю нівелює ризик створення додаткового навантаження на основний REST API месенджера під час генерації важких аналітичних звітів.

Налаштовані візуальні панелі Metabase дозволяють розробнику в режимі реального часу відстежувати критичні показники: загальний коефіцієнт конверсії (Click-Through Rate) швидких підказок у розрізі окремих днів, динаміку витрат лімітів токенів модулем AskAI та середній час очікування відповіді від зовнішнього API Google. Візуалізація цих метрик не лише емпірично підтверджує практичну цінність впровадження мовної моделі для кінцевого користувача, але й формує статистичну базу для подальшого тонкого налаштування (Fine-Tuning) системних промптів. Це гарантує безперервне покращення релевантності генеруємого контенту та раціональне використання ресурсів серверного обладнання. Інтерфейс Аналітики продемонстровано на рис. 3.8.

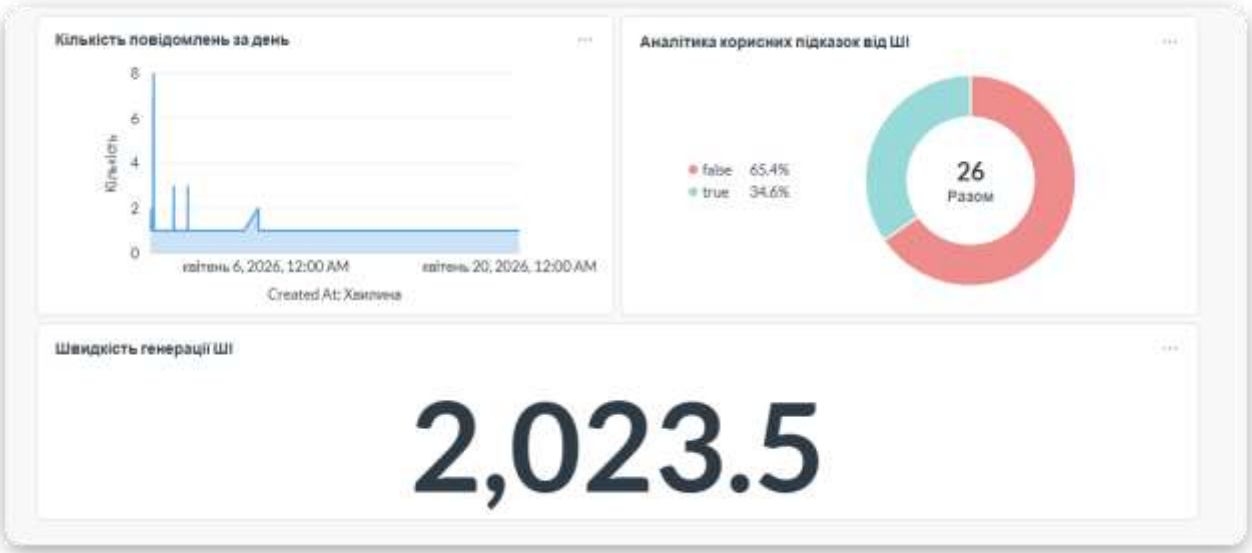


Рисунок 3.8 – Панель аналітики Metabase

Висновки до розділу 3

У третьому розділі було здійснено повний цикл практичної реалізації програмного продукту на основі закладених на етапі проєктування архітектурних рішень. Процес розробки охопив створення як серверної, так і клієнтської частин із застосуванням сучасних інструментів та методологій програмної інженерії.

Серверну інфраструктуру побудовано на базі середовища Node.js та мікрофреймворку Express. Застосування архітектурного патерну «Репозиторій» дозволило чітко розмежувати логіку маршрутизації HTTP-запитів від виконання нативних SQL-транзакцій у СУБД PostgreSQL. Це забезпечило високу швидкість обробки даних та стійкість системи до кібератак типу SQL-ін'єкцій. Крім того, налаштовано надійну систему багаторівневого проміжного програмного забезпечення (Middleware) для обробки багатокомпонентних файлових завантажень та захисту конфіденційних маршрутів за допомогою алгоритмів JWT-авторизації.

Розроблено динамічний інтерфейс користувача із застосуванням бібліотеки React у форматі односторінкового застосунку (SPA). Відмова від надлишкових сторонніх інструментів управління станом на користь нативного Context API дозволила суттєво оптимізувати обсяг фінального клієнтського бандлу. Безпечна

мережева взаємодія автоматизована через конвеєр перехоплювачів (Interceptors) HTTP-клієнта Axios, а стилізація компонентів імплементована за допомогою технології SCSS Modules для повної інкапсуляції класів та уникнення каскадних конфліктів.

Для забезпечення роботи месенджера в режимі реального часу успішно інтегровано протокол WebSockets через платформу Socket.io. Створено глобальний клієнтський провайдер для автоматичної синхронізації стану з'єднання та миттєвого рендерингу нових повідомлень. Важливим архітектурним досягненням стало вирішення проблеми блокування основного потоку виконання (Event Loop) шляхом винесення ресурсоємних завдань, зокрема транзакційних email-розсилок, у фонові черги пам'яті Redis під управлінням спеціалізованих процесів-обробників та планувальників (Cron).

Успішно реалізовано комплексну інтеграцію застосунку з великою мовною моделлю через програмний інтерфейс Google Generative AI. Створено клієнтські та серверні модулі фонової генерації підказок і повноцінного інтерактивного віртуального асистента, які підкріплені системою збору телеметрії.

У підсумку, розроблена система демонструє високу відмовостійкість, продуктивність та повністю відповідає заявленим функціональним вимогам. Написаний програмний код є структурованим та готовим до розгортання, що дозволяє перейти до фінального етапу – складання інструкцій користувача та оцінки економічної ефективності проєкту.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

4.1. Тестування функціональних можливостей API та WebSocket-з’єднань

Перехід до фінальної стадії життєвого циклу розробки програмного забезпечення (SDLC) вимагає всебічної перевірки створеного функціоналу. Метою етапу тестування є підтвердження того, що спроєктована серверна архітектура коректно обробляє вхідні дані, витримує нетипові сценарії використання та забезпечує стабільний зв'язок у режимі реального часу. Для розроблюваної системи було застосовано комплексний підхід: від перевірки ізольованих функцій безпеки до наскрізного тестування клієнт-серверної взаємодії.

Фундаментом перевірки безпеки системи стало модульне тестування логіки автентифікації. За допомогою підготовленого набору тестів (зокрема у файлі auth.test.ts) перевірялися критичні механізми: генерація JWT-токенів, валідація паролів через алгоритм хешування bcrypt та коректність роботи проміжного програмного забезпечення. Тестування підтвердило, що при спробі доступу з невалідним підписом система гарантовано перериває ланцюжок виконання.

Основним етапом стало інтеграційне тестування розробленого REST API. Для зручного візуального моделювання HTTP-запитів, налаштування заголовків та аналізу відповідей сервера використовувалося графічне середовище Postman [36]. Цей інструмент дозволив гнучко перевірити всі захищені маршрути без необхідності попередньої розробки повноцінного клієнтського інтерфейсу. Процес тестування полягав у ручному формуванні запитів (GET, POST, PUT, DELETE) до локального сервера, куди у вкладку Authorization передавався актуальний Bearer-токен тестового користувача.

Під час роботи з Postman моделювалися як позитивні, так і негативні бізнес-сценарії. Зокрема, перевірялася реакція бекенду на спробу відправки повідомлення у чат, до якого користувач не належить, а також валідувалася швидкість генерації інтелектуальних підказок маршрутом /api/ai/suggest. Результати успішного тестування одного з ключових ендпоінтів наведено на рис. 4.1.

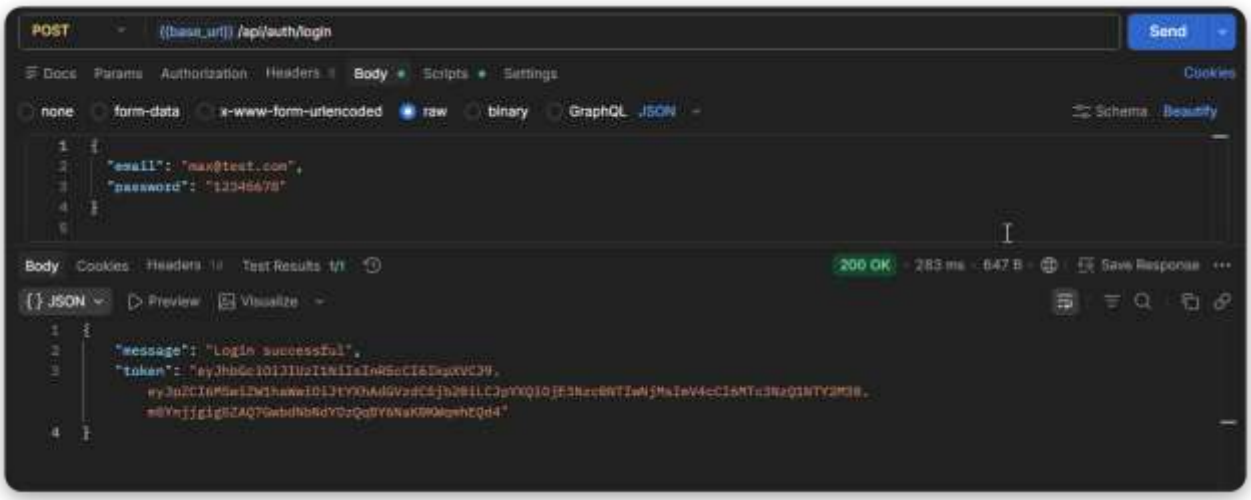


Рисунок 4.1 – Результат виконання HTTP-запиту в графічному середовищі Postman

Для кращого розуміння структури обміну даними, у лістингу 4.1 наведено типовий формат тіла запиту та успішної відповіді сервера (формат JSON), отриманої під час тестування функціоналу ініціалізації нового приватного діалогу.

Лістинг 4.1 – Структура обміну даними при тестуванні REST API через Postman

```
POST /api/messages
{
    "receiverId": 2,
    "text": "Привіт! Це тестове повідомлення з Postman."
}

{
    "id": 234,
```

```
    "chat_id": 2,
    "user_id": 1,
    "text": "Привіт! Це тестове повідомлення з Postman.",
    "created_at": "2026-04-29T08:42:48.446Z",
    "type": "text",
    "is_deleted": false,
    "sender_name": "max",
    "sender_avatar": "/avatars/avatar-1775313951265-980660010.png"
    }
}
```


Специфічним інженерним викликом стала перевірка працездатності підсистеми обміну даними у реальному часі. Оскільки стандартні інструменти тестування REST API не призначені для роботи з постійними двосторонніми з'єднаннями [37], перевірка протоколу WebSockets проводилася безпосередньо в середовищі розробленого React-фронтенду з використанням вбудованих інструментів розробника (Browser DevTools) [38].

Для верифікації правильності роботи патерну Publish-Subscribe моделювалася ситуація одночасної роботи двох користувачів, авторизованих у різних вікнах браузера (або режимі інкогніто). За допомогою вкладки Network (фільтр WS) відстежувався процес встановлення TCP-з'єднання та аналізувалися бінарні фрейми, якими обмінюються клієнт і сервер. Паралельно використовувалася консоль браузера (Console), де через виклики логування фіксувалися події життєвого циклу сокета: успішне підключення, приєднання до ізольованої кімнати (подія join_chat) та надходження нових даних.

Цей підхід практично довів коректність реалізованої гібридної архітектури: після виконання HTTP POST-запиту на збереження повідомлення, сервер миттєво транслював WebSocket-подію receive_message, яку React-клієнт успішно перехоплював та рендерив у DOM-дереві співрозмовника без необхідності перезавантаження сторінки. Візуальне підтвердження перехоплення мережевих пакетів наведено на рис. 4.2.

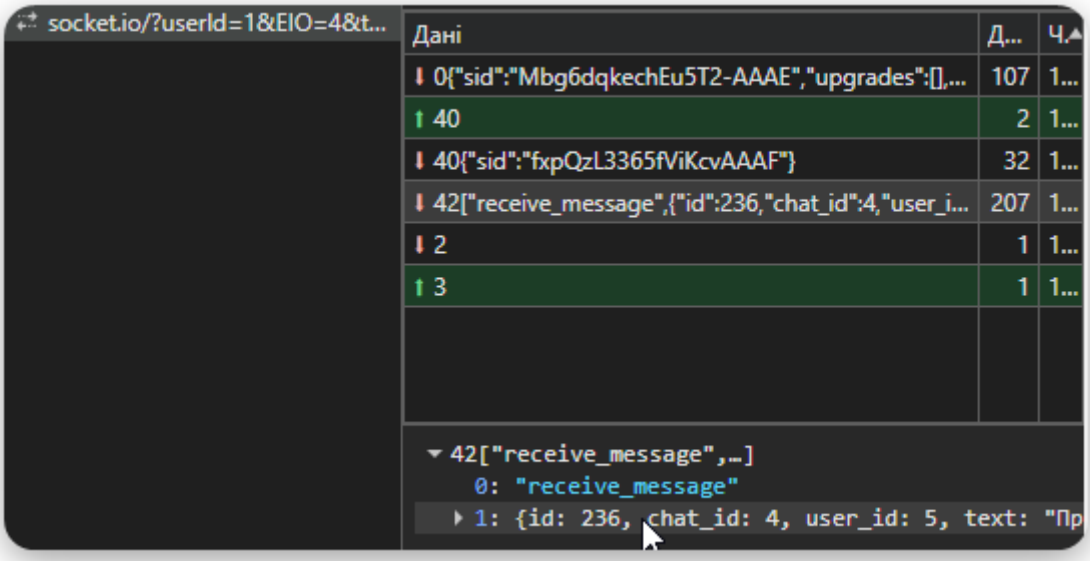


Рисунок 4.2 – Відстеження WebSocket-з'єднання та подій за допомогою Browser DevTools

Для систематизації процесу перевірки працездатності розробленого месенджера було сформовано набір базових тестових сценаріїв (Test Cases). Вони охоплюють перевірку критично важливих вузлів системи: від ізольованих модулів безпеки до інтеграційних процесів клієнт-серверної взаємодії та роботи штучного інтелекту. Зведений перелік проведених функціональних тестів та їхнє призначення наведено в таблиці 4.1.

Таблиця 4.1

Зведений перелік сценаріїв тестування програмного продукту

№ з/п	Назва тесту	Пояснення (опис сценарію та очікуваний результат)
1	Валідація реєстраційних даних	Перевірка реакції бекенду на спробу створення акаунта з некоректними даними (порожні поля або email, що вже існує в базі) Очікується статус 400.
2	Автентифікація та видача JWT	Тестування механізму логіну. Перевіряється алгоритм порівняння хешів bcrypt та успішність генерації токена доступу. Очікується статус 200 OK із підписаним токеном.

Продовження табл. 4.1.

3	Блокування неавторизованого доступу	Спроба звернення до захищених маршрутів (наприклад, /api/messages) без передачі Bearer токена у заголовку. Очікується спрацювання authMiddleware та помилка 401 Unauthorized.
4	Ізоляція WebSocket-кімнат	Перевірка механізму join_chat. Два клієнти підключаються до різних чатів. Очікується, що події з однієї кімнати не будуть транслюватися в іншу, гарантуючи конфіденційність.
5	Ретрансляція подій (Broadcasting)	Тестування патерну Publish-Subscribe. Відправка повідомлення з одного клієнтського вікна та перевірка миттєвого отримання події receive_message іншим учасником чату.
6	Форматування відповідей ШІ	Запит до маршруту /api/ai/suggest для перевірки інтеграції з Gemini. Очікується, що сервер коректно очистить відповідь від Markdown-тегів і поверне строго типізований масив JSON.
7	Делегування фонових задач у Redis	Тестування роботи брокера повідомлень. Після ініціалізації реєстрації перевіряється, чи потрапляє задача на відправку email у чергу BullMQ без блокування основного потоку Node.js.
8	Оновлення статусу користувача	Імітація розриву TCP-з'єднання (закриття вкладки браузера). Очікується спрацювання події disconnect на сервері та зміна статусу користувача в базі даних з online на offline.

Завдяки успішному проходженню всіх перелічених тестових сценаріїв було підтверджено, що система адекватно реагує як на стандартні дії користувачів, так і на непередбачені або помилкові вхідні дані.

4.2. Налаштування Docker-контейнерів та оркестрація сервісів

Важливим етапом впровадження розробленого месенджера стало забезпечення ідентичності середовищ виконання на різних етапах життєвого циклу проєкту. З огляду на те, що розробка велася паралельно на робочій станції під управлінням ОС Windows та ноутбуку з ОС Linux, виникла потреба в

інструменті, який би нівелював різницю у версіях системних залежностей. Для вирішення цієї задачі було застосовано технологію контейнеризації Docker.

На ОС Windows управління інфраструктурою здійснювалося через графічний інтерфейс Docker Desktop, що дозволяло зручно моніторити ресурси та стан сервісів у реальному часі. Для середовища Linux було підготовлено декларативний файл конфігурації docker-compose.yml, який дозволяє розгорнути всю необхідну архітектуру шару даних однією командою. Оркестрація за допомогою Docker Compose забезпечила автоматизацію створення мережевих зв'язків між контейнерами та налаштування параметрів перезапуску.

Конфігурація інфраструктури охоплює три взаємозалежні сервіси:

1. СУБД PostgreSQL (db): Використовується образ версії 17-alpine, що забезпечує мінімальний розмір контейнера. Для збереження даних між перезапусками налаштовано том postgres_data, а для коректної підтримки кирилиці імплементовано монтування ініціалізаційного скрипта init_utf8.sql.
2. In-memory сховище Redis (cache): Слугує для прискорення доступу до повідомлень та функціонування черг задач. Контейнер прокидає порт 6379 для взаємодії з Node.js сервером.
3. Платформа аналітики Metabase (metabase): Інтегрована для візуалізації метрик ШІ. Важливою особливістю є директива depends_on, яка гарантує, що аналітичний модуль запуститься лише після успішної ініціалізації бази даних.

Практичний запуск системи здійснюється через термінал командою docker-compose up -d. Після виконання команди Docker-демон завантажує необхідні образи, створює ізольовану віртуальну мережу та запускає процеси у фоновому режимі. Стан розгорнутої інфраструктури в інтерфейсі Docker Desktop наведено на рис. 4.3.

Рисунок 4.3 – Моніторинг стану запущених сервісів у середовищі Docker Desktop

Архітектурне розміщення компонентів системи та їх взаємодія з операційною системою хоста відображені на UML-діаграмі розгортання (рис. 4.4).

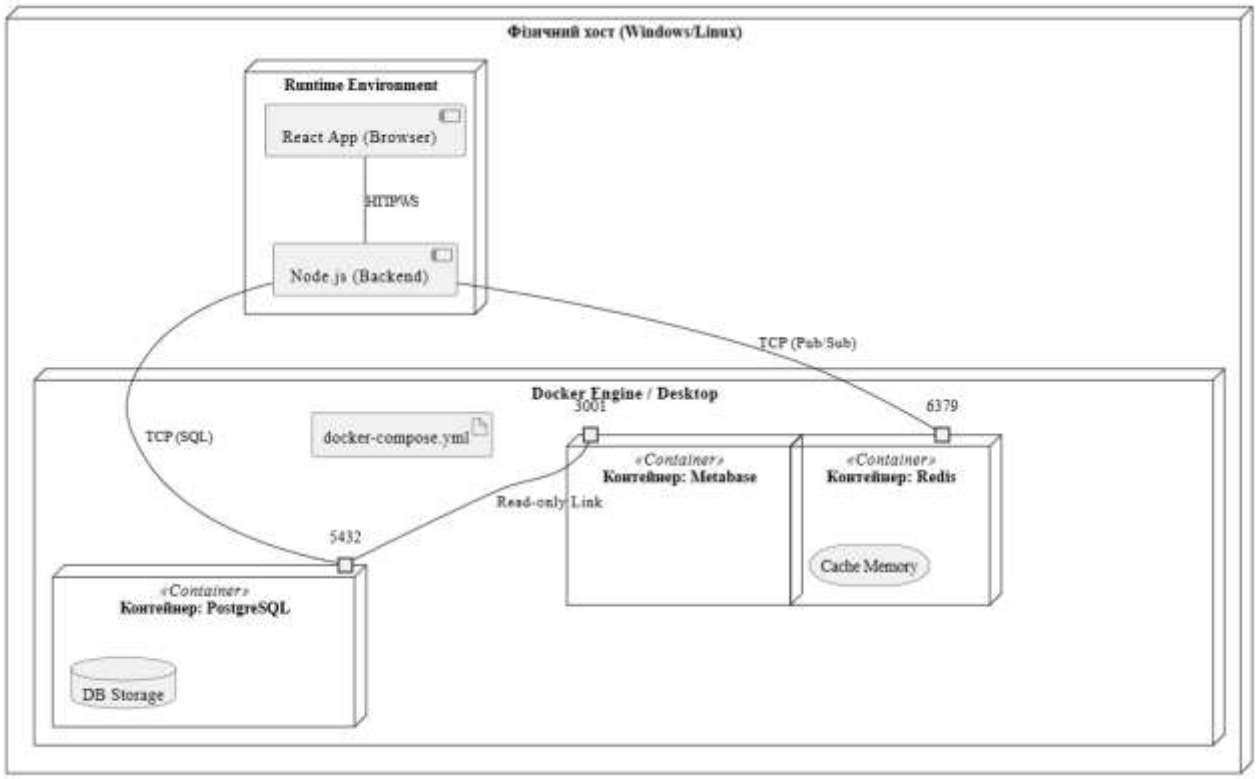


Рисунок 4.4 – UML-діаграма розгортання (Deployment Diagram) інфраструктури системи

4.3. Візуалізація аналітичних та експлуатаційних даних у середовищі Metabase

Критичним аспектом впровадження інтелектуальних функцій та систем реального часу в сучасні програмні продукти є безперервний моніторинг їхньої

ефективності та впливу на загальну стабільність інфраструктури. Оскільки розроблений месенджер використовує зовнішнє API Google Generative AI та підтримує постійні WebSocket-з'єднання, виникла гостра потреба в інструменті для наочного аналізу накопиченої телеметрії. Для вирішення цього завдання було використано платформу бізнес-аналітики Metabase, яка була інтегрована безпосередньо в загальну Docker-інфраструктуру проєкту. Процес налаштування аналітичного середовища базувався на встановленні прямого мережевого зв’язку між контейнерами через внутрішній порт 5432, що дозволило системі Metabase отримувати дані з PostgreSQL у режимі реального часу без створення додаткового навантаження на основний REST API.

Важливою інженерною особливістю реалізації цього етапу стала повна відмова від ручного написання складних SQL-запитів на користь використання вбудованого візуального конструктора запитів (Visual Query Builder). Це дозволило швидко агрегувати сирі логічні дані та перетворити їх на структуровану управлінську інформацію. Для формування комплексного уявлення про працездатність месенджера на єдину панель адміністратора було виведено групу взаємопов'язаних віджетів, що охоплюють різні аспекти життєдіяльності системи. Ключовим технічним показником став моніторинг активних підключень, реалізований у віигляді цифрового індикатора, який відображає кількість користувачів зі статусом «online» та дозволяє миттєво оцінити кількість відкритих WebSocket-тунелів, які одночасно підтримує сервер Node.js.

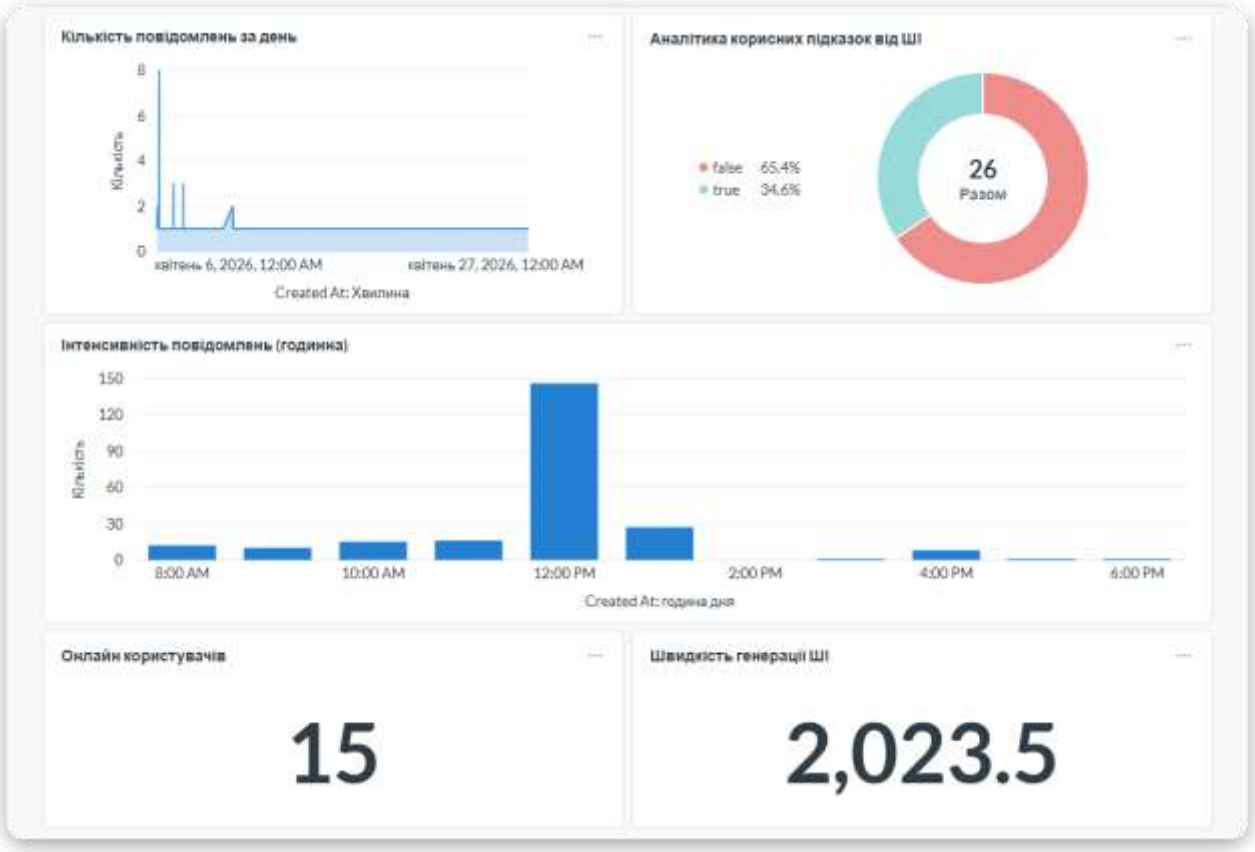


Рисунок 4.5 – Адміністративний дашборд Metabase для технічного моніторингу та аналізу активності користувачів

Особлива увага при аналізі продуктивності була приділена інтенсивності обміну повідомленнями. Для цього було спроєктовано стовпчасту діаграму розподілу навантаження за годинами доби, яка наочно демонструє піки активності користувачів. Аналіз цього графіка дозволяє адміністратору прогнозувати періоди найбільшого навантаження на базу даних та In-memory кеш Redis, що є критично важливим для планування апаратних ресурсів сервера. Паралельно з технічними метриками на дашборд виведено показники ефективності штучного інтелекту, зокрема коефіцієнт корисності згенерованих підказок. Це дає змогу відстежувати релевантність відповідей моделі Gemini та корелювати їх із загальною активністю в чатах. Завдяки такому поєднанню бізнес-метрик та інженерної телеметрії, розроблена адміністративна панель перетворюється на повноцінний центр управління системою. Візуальне

представлення підсумкового дашборду, що об'єднує всі описані показники інтенсивності та активності, наведено на рис. 4.5.

4.4. Оцінка продуктивності системи під навантаженням

Завершальним етапом експериментальних досліджень стала оцінка здатності спроєктованої архітектури витримувати інтенсивний обмін даними. Оскільки розроблюваний месенджер належить до класу застосунків реального часу (Real-Time Applications), критично важливим показником є не лише успішність доставки повідомлень, але й мінімізація мережевої затримки (Latency). З огляду на те, що використання синтетичних бібліотек для стрес-тестування в умовах локального розгортання часто дає похибку через обмеження апаратних ресурсів хост-машини, оцінка продуктивності проводилася шляхом комплексного аналізу архітектурних рішень та ручного сценарного моделювання паралельних сесій. Основною метою було виявлення потенційних вузьких місць (bottlenecks) на рівні бази даних та циклу подій (Event Loop) сервера Node.js за допомогою вбудованих інструментів розробника (Browser DevTools).

Аналіз продуктивності шару доступу до даних показав високу ефективність імплементованого механізму пулу з'єднань. Завдяки налаштуванню глобального об'єкта pg.Pool, серверній частині не доводиться витрачати ресурси на встановлення нового TCP-з'єднання з реляційною базою PostgreSQL для кожної транзакції [39]. Моделювання інтенсивного листування, здійснене шляхом паралельної відправки запитів із кількох ізольованих вкладок браузера, продемонструвало стабільність роботи REST API. Моніторинг мережевих пакетів у вкладці Network показав, що середній час до отримання першого байта (Time To First Byte, TTFB) при збереженні повідомлення залишається в межах 20–50 мілісекунд. Водночас тестування інфраструктури WebSockets довело високу пропускну здатність технології під час масової розсилки подій (broadcasting). Навіть при імітації значної кількості підключених клієнтів в одній

кімнаті, асинхронна природа Node.js дозволила миттєво мультиплексувати події receive_message без помітного зростання споживання оперативної пам'яті [40].

Додатковим фактором, що гарантує стабільність системи під навантаженням, стало використання брокера повідомлень Redis для делегування важких завдань вводу-виводу. Експериментальна імітація одночасних запитів на реєстрацію та генерацію підказок штучним інтелектом (AskAI) підтвердила правильність обраної стратегії. Замість того, щоб блокувати основний потік очікуванням відповіді від SMTP-сервера або зовнішнього API Google, контролери миттєво формували фонові задачі (Jobs) у черзі BullMQ [41]. Завдяки цьому інтерфейс користувача залишався чутливим, а основний сервер продовжував безперебійно обробляти вхідні WebSocket-пакети. Проведені дослідження доводять, що розроблена гібридна інфраструктура володіє значним запасом міцності та архітектурно підготовлена до горизонтального масштабування в умовах промислової експлуатації.

4.5 Інструкція користувача

Розробка будь-якого програмного забезпечення передбачає створення зрозумілого та зручного керівництва для користувача. В цьому підрозділі описано основні сценарії взаємодії з програмою, щоб навіть недосвідчений користувач міг зрозуміти інтерфейс та користуватись його всіма функціями.

Початок користування месенджером починається зі сторінки авторизації. Для нових користувачів реєстрація не є складною. Користувачу потрібно перемкнути форму на «Реєстрація» та заповнити поля: вказати електрону пошту, пароль та підтвердити пароль. На поточному етапі верифікація не потрібна, тому після натискання на кнопку «Зареєструватися» система перемикає на форму де потрібно увійти в акаунт, після авторизації користувачу відкривається доступ до обміну повідомлень з іншими, рисунок сторінки продемонстровано в додатку Б2.

Головне вікно месенджеру має 2 частини: ліворуч – сайдбар, який продемонстровано на рис. 4.6. Він містить міні-профіль користувача в якому він

може побачити свій аватар та ім’я, і кнопкою виходу з аканту та кнопкою щоб відкрити налаштування. У налаштуваннях користувач може редагувати свої дані, а саме профіль, пошту, пароль. Також є можливість змінити приватність акаунту, а також змінити тему на одну з трьох: «Light», «Dark», «Glass».

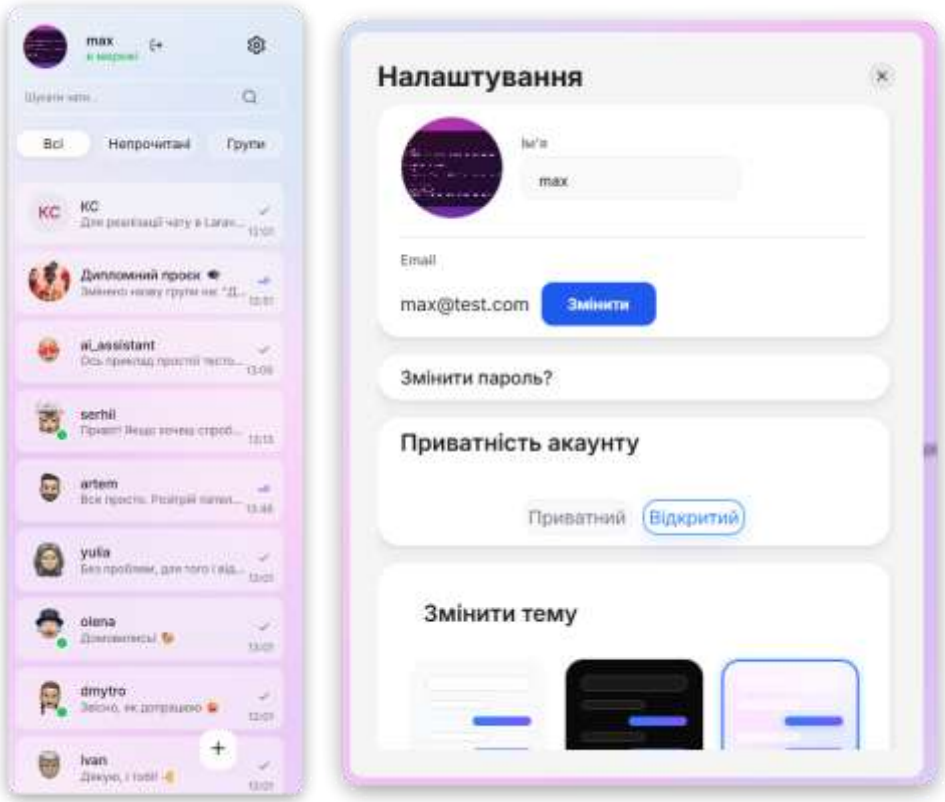


Рисунок 4.6 – Ліва частина інтерфейсу з вікном налаштувань.

Під верхньою частиною бічної панелі, користувач має керування чатами. Користувачі можуть знайти необхідний чат через поле пошуку, а також скористатись фільтрами щоб переглянути списки чатів в відсортованому форматі. Наприклад одразу всі чати, або тільки групові. Нижче під керуваннями користувачу відображається список чатів в яких він бере участь, цьому списку користувачу надається інформація про: зображення чату, назву\ім’я співбесідника, останнє повідомлення, а також онлайн статус користувачів і статус прочитанності повідомлення. Якщо користувач натисне на чат у списку

правою кнопкою миші, то відкриється контекстне меню в якому можна виділити чат тільки у себе або одразу у всіх користувачів.

Для початку спілкування, потрібно натиснути або на існуючий чат, або на кнопку створення нового чату. В системі є як приватні чати та групові. Щоб створити новий діалог, користувачу потрібно ввести в пошуку користувачів ім’я і якщо у компаньйона акаунт не приватний, то при кліку на блок в списку, новий чат одразу створиться. Також це вікно підтримує і створення групового чату, при такому форматі, користувач має в полі ввести назву групи, а також обрати користувачів зі списку. Після підтвердження створення групи, у користувача відкриється вікно чату, а у інших користувачів група з’явиться у списку чатів. Інтерфейс створення чатів продемонстровано на рисунку 4.7.

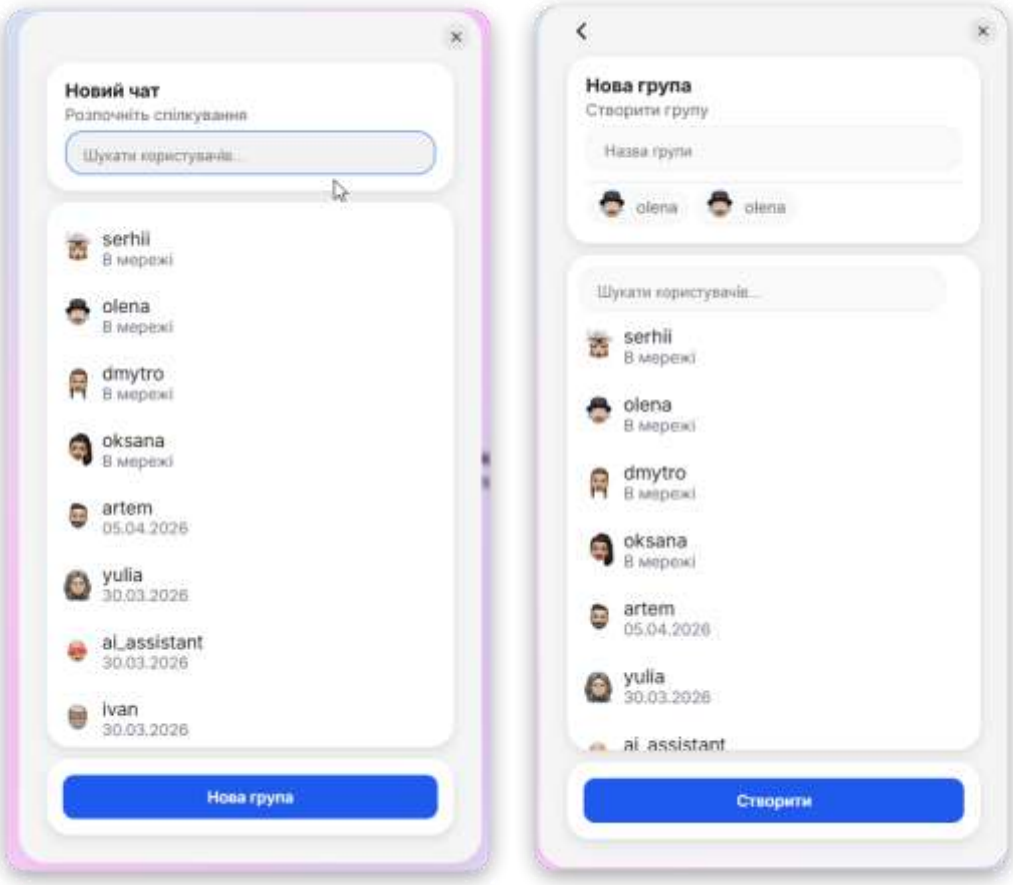


Рисунок 4.7 – Інтерфейс створення чатів.

Робоча область чату має звичний інтерфейс для більшості користувачів. Угорі відображається ім’я співрозмовника та його онлайн статус, по центру історія листування, а знизу вікна поле для введення тексту, детально відображено інтерфейс чату у додатку Б5.

Ключовою ціллю розробки було інтегрувати модулі ШІ. Цей функціонал реалізовано у двох форматах. Перший формат – це швидкі підказки, під час листування користувач може натиснути на кнопку біля останнього повідомлення, після цього над полем для введення повідомлення буде відображено 3 підказки. Ці фрази генеруються нейромережою на основі останніх повідомлень, якщо запропонована підказка відповідає вимогам користувача, достатньо натиснути на неї, щоб система вставила цю фразу у поле для введення повідомлень.

Другий формат – це повноцінний ШІ асистент. Біля кнопки відправлення повідомлень, передбачено окрему кнопку для виклику вікна ШІ помічника. Відкривши його, користувач може зробити будь який запит до асистента, наприклад описати певну технологію, або запитати дорогу до міста. Після відправки запиту, ШІ одразу почне генерувати відповідь, після чого вставить її у чат окремим повідомленням, яке виділяється по стилю від інших, інтерфейс взаємодії зі ШІ відображенно на рис. 4.8.

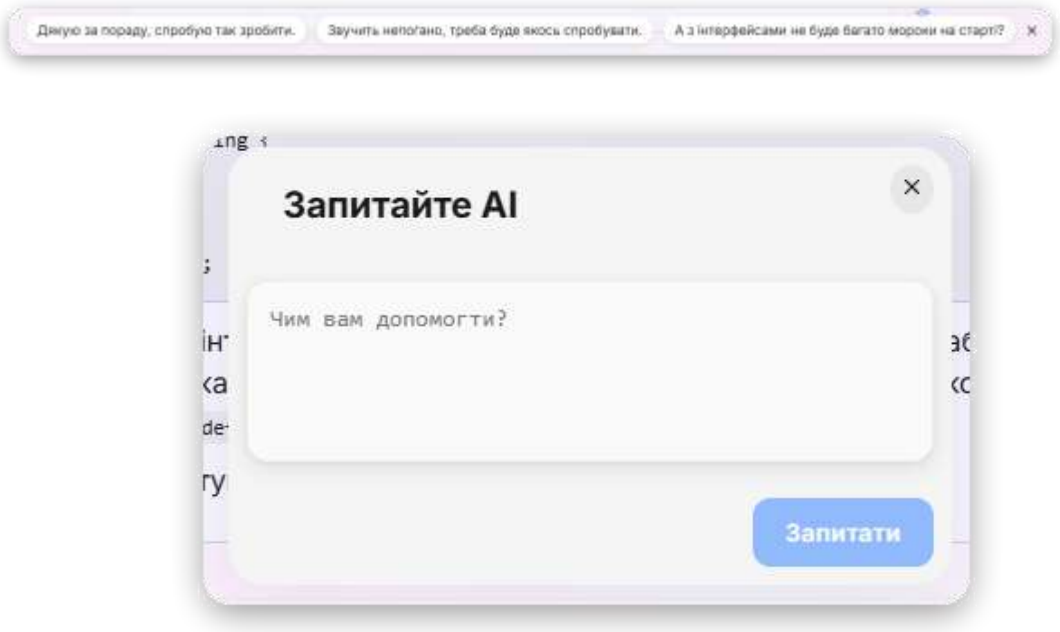


Рисунок 4.8 – Інтерфейс взаємодії зі ШІ.

Такий підхід позбавляє користувача необхідності перемикатися між месенджером та застосункам з ШІ для пошуку необхідної інформація, роблячи процес комунікації продуктивним та зручним.

Висновки до розділу 4

У четвертому розділі було проведено комплексне тестування розробленого програмного продукту та описано ключові етапи його підготовки до реальної експлуатації. Перевірка безпеки та бізнес-логіки серверної частини, здійснена за допомогою графічного середовища Postman, підтвердила коректність роботи алгоритмів авторизації, захисту кінцевих точок та механізмів генерації інтелектуальних підказок. Водночас тестування протоколу WebSockets за допомогою інструментів розробника (DevTools) безпосередньо в середовищі клієнтського застосунку довело стабільність роботи патерну Publish-Subscribe, забезпечивши надійну та миттєву синхронізацію даних між співрозмовниками.

Описано процес оркестрації інфраструктури шару даних за допомогою технології контейнеризації Docker. Використання конфігураційного файлу docker-compose.yml дозволило повністю ізолювати середовища виконання бази даних PostgreSQL, сховища Redis та BI-платформи, гарантуючи ідентичність розгортання на операційних системах Windows та Linux. Впровадження інструментів аналітики Metabase дало змогу агрегувати сирі дані у візуальні дашборди без написання SQL-запитів, що забезпечило адміністратору системи можливість в режимі реального часу відстежувати активні підключення, інтенсивність навантаження та коефіцієнт корисності алгоритмів штучного інтелекту.і

Проведена оцінка продуктивності підтвердила високий рівень надійності закладених архітектурних рішень. Експериментальне моделювання паралельних сесій довело, що використання неблокуючого середовища Node.js у комбінації з пулом з'єднань бази даних та системою фонових черг ефективно запобігає перевантаженню основного потоку виконання. Загалом, розроблений

вебмесенджер повністю відповідає поставленим вимогам технічного завдання, успішно пройшов етап верифікації та є готовим до використання кінцевими споживачами.

ВИСНОВКИ

Результатом виконання дипломної роботи стала розробка та практична реалізація повнофункціональної системи миттєвого обміну повідомленнями з інтегрованими модулями штучного інтелекту. У ході дослідження було успішно досягнуто поставленої мети та розв’язано всі визначені завдання, що дозволило створити сучасний програмний продукт, який відповідає актуальним вимогам до швидкодії, безпеки та інтелектуальної взаємодії з користувачем.

На етапі проєктування було проведено ґрунтовний аналіз предметної області та існуючих рішень, що дозволило обґрунтувати вибір технологічного стека. Використання середовища Node.js у поєднанні з реляційною базою даних PostgreSQL та In-memory сховищем Redis забезпечило надійний фундамент для побудови масштабованої серверної частини. Особливу увагу було приділено розробці архітектури, де застосування патерну «Репозиторій» дозволило чітко розмежувати бізнес-логіку та рівень доступу до даних, що значно підвищило підтримуваність та чистоту програмного коду.

Практична реалізація клієнтської частини на базі бібліотеки React дозволила створити динамічний та інтуїтивно зрозумілий інтерфейс користувача. Відмова від надлишкових сторонніх бібліотек управління станом на користь нативного Context API сприяла оптимізації продуктивності застосунку. Важливим досягненням стало впровадження протоколу WebSockets через платформу Socket.io, що забезпечило повноцінну двосторонню комунікацію в реальному часі. Це дозволило реалізувати не лише миттєву доставку повідомлень, але й динамічне відстеження статусів користувачів та індикацію прочитання, що є критично важливим для сучасних месенджерів.

Інноваційною складовою проєкту стала успішна інтеграція великої мовної моделі Gemini від Google. Розроблений модуль інтелектуальних підказок (Smart Replies) та персональний ШІ-асистент (AskAI) виводять користувацький досвід на новий рівень, автоматизуючи типові відповіді та надаючи допомогу в межах контексту діалогу. Для контролю за ефективністю цих модулів було впроваджено

систему бізнес-аналітики Metabase. Створення інтерактивних дашбордів дозволило в режимі реального часу відстежувати технічні показники системи та аналізувати корисність ШІ-генерацій, що забезпечує можливість проактивного управління ресурсами сервера.

Етап тестування та впровадження підтвердив високу надійність та відмовостійкість системи. Використання технології контейнеризації Docker гарантувало ідентичність середовищ розробки та експлуатації, спрощуючи розгортання проєкту на різних операційних системах. Проведена оцінка продуктивності довела, що поєднання неблокуючої архітектури бекенду з системою фонових черг на базі Redis дозволяє системі стабільно працювати під навантаженням, зберігаючи мінімальні затримки при обробці повідомлень.

Підсумовуючи, можна стверджувати, що створений програмний комплекс є завершеним інженерним рішенням, яке повністю реалізує заявлений функціонал. Отримана система має значний потенціал для подальшого масштабування та впровадження додаткових можливостей, таких як наскрізне шифрування даних та групові аудіовізуальні дзвінки. Робота має практичну цінність і може бути використана як база для створення спеціалізованих корпоративних засобів комунікації.

Посилання на репозиторій з кодом програми URL : https://github.com/MaksimBoyko05/MessengerApp

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Slack vs Mattermost vs Matrix: EU Sovereignty Audit. *SovereigntyScore.io*. 2026. URL: https://sovereigntyscore.io/info-hub/compare/slack-vs-mattermost-matrix (дата звернення: 10.04.2026).
2. Preda A. Data Sovereignty Is Changing Business Communications in Europe. *SSH*. 2026. URL: https://www.ssh.com/blog/data-sovereignty-is-changing-business-communications-in-europe-ssh (дата звернення: 10.04.2026).
3. Context Switching Statistics 2026: The Hidden Cost of Multitasking, App Toggling, and Fragmented Focus. *Speakwise*. 2026. URL: https://speakwiseapp.com/blog/context-switching-statistics (дата звернення: 10.04.2026).
4. How Context Switching Reduces Workplace Productivity. *Cannelevate*. 2025. URL: https://www.cannelevate.com.au/article/context-switching-productivity-hidden-cost-modern-work/ (дата звернення: 10.04.2026).
5. What is a context window? *IBM*. 2024. URL: https://www.ibm.com/think/topics/context-window (дата звернення: 10.04.2026).
6. Top techniques to Manage Context Lengths in LLMs. *Agenta*. 2025. URL: https://agenta.ai/blog/top-6-techniques-to-manage-context-length-in-llms (дата звернення: 10.04.2026).
7. Node. js Design Patterns: Design and Implement Production-Grade Node. js Applications Using Proven Patterns and Techniques, 3rd Edition. Packt Publishing, Limited, 2020. 660 с.
8. Scaling WebSockets in Node.js: Handling 100k Concurrent Connections. Engineering at Scale. 2025. URL: https://engineeringatscale.com/scaling-websockets-nodejs (дата звернення: 10.04.2026).
9. Learning TypeScript: Enhance Your Web Development Skills Using Type-Safe JavaScript. O'Reilly Media, Incorporated, 2022. 200 с.

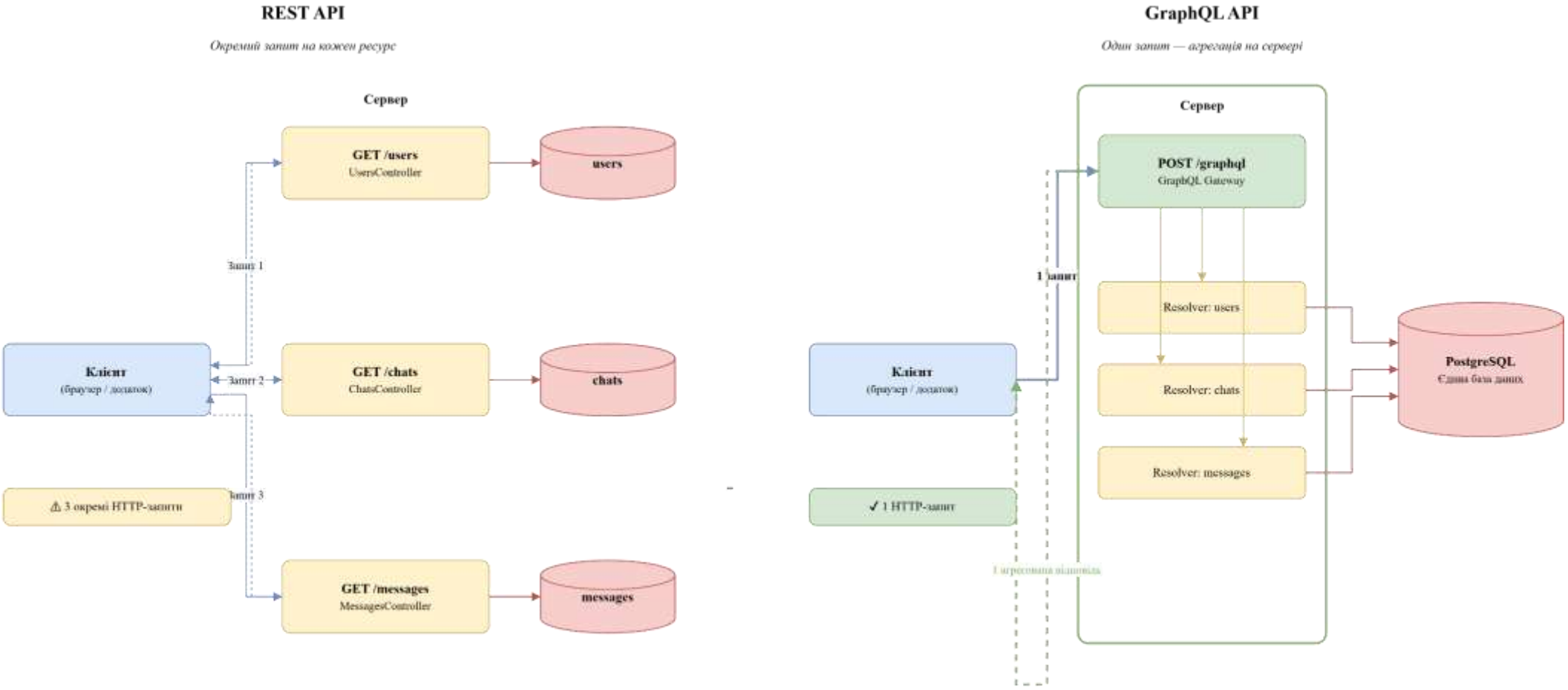
10. Stemmler K. Solid Principles and the Repository Pattern in Node.js / TypeScript. *Enterprise Node.js Architecture*. 2022. URL: https://khalilstemmler.com/articles/typescript-domain-driven-design/repository-dto-mapper/ (дата звернення: 10.04.2026).
11. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software. O'Reilly Media, Incorporated, 2021. 694 с.
12. Gough J., Bryant D., Auburn M. Mastering API Architecture: Design, Operate, and Evolve API-Based Systems. Sebasstopol: O'Reilly Media, 2022. 334 с.
13. REST vs GraphQL: A Detailed Comparison. *IBM Technology*. 2023. URL: https://www.ibm.com/blog/rest-vs-graphql/ (дата звернення: 13.04.2026).
14. Introduction to GraphQL. *The GraphQL Foundation*. 2024. URL: https://graphql.org/learn/ (дата звернення: 13.04.2026).
15. Solving the N+1 problem for GraphQL. *Apollo GraphQL Blog*. 2023. URL: https://www.apollographql.com/blog/graphql/n-plus-one/ (дата звернення: 13.04.2026).
16. Carlson A. I. Reduce real-time cloud spend with Federated Subscriptions support for HTTP Callbacks // Apollo GraphQL Blog. – 2024. – URL: https://www.apollographql.com/blog/reduce-real-time-cloud-spend-with-federated-subscriptions-support-for-http-callbacks (дата звернення: 13.04.2026).
17. Lindström S. GraphQL vs REST 2026: 28% Latency Gap and 340% Surge [Tested] // Tech-Insider.org. – 2026. – URL: https://tech-insider.org/graphql-vs-rest-2026/ (дата звернення: 13.04.2026).
18. GraphQL vs REST in 2026: A Practical Comparison // jsmanifest.com. – 2026. – URL: https://jsmanifest.com/graphql-rest-practical-comparison-2026 (дата звернення: 13.04.2026).
19. Roldán C. S. React 18 Design Patterns and Best Practices: Design, build, and deploy production-ready web applications. 4th ed. Birmingham: Packt Publishing, 2023. 524 с.

20. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol: O'Reilly Media, 2021. 488 с.
21. Render and Commit // React. – 2024. – URL: https://react.dev/learn/render-and-commit (дата звернення: 13.04.2026).
22. Why Vite // Vite Documentation. – 2025. – URL: https://vitejs.dev/iiguide/why.html (дата звернення: 13.04.2026).
23. Kato D. Micro State Management with React Hooks: Explore custom hooks, Context API, and third-party libraries. Birmingham: Packt Publishing, 2022. 254 с.
24. State Management in React: Context API vs Redux. *Frontend Masters*. 2023. URL: https://react.dev/learn/passing-data-deeply-with-context (дата звернення: 13.04.2026).
25. Obe R., Hsu L. PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database. 4th ed. Sebastopol: O'Reilly Media, 2022. 350 с.
26. Hunter T., English J. Distributed Systems with Node.js: Building Enterprise-Ready Backend Services. Sebastopol: O'Reilly Media, 2021. 315 с.
27. Kumar T. Fluent React: Build Fast, Scalable, and Maintainable Web Apps. Sebastopol: O'Reilly Media, 2024. 350 с.
28. Using Multiple Nodes. *Socket.IO Official Documentation*. 2024. URL: https://socket.io/docs/v4/using-multiple-nodes/ (дата звернення: 13.04.2026).
29. Petrella A. Fundamentals of Data Observability: Implement robust data quality and monitoring systems. Birmingham: Packt Publishing, 2023. 280 с.
30. Architecture and Deployment // Metabase Documentation. – 2025–2026. – URL: https://www.metabase.com/docs/latest/installation-and-operation (дата звернення: 13.04.2026).
31. Kleppmann M., Riccomini C. Designing Data-Intensive Applications. 2nd ed. (Early Release). Sebastopol: O'Reilly Media, 2026.
32. Hunter T. II. Distributed Systems with Node.js: Building Enterprise-Ready Backend Services. Sebastopol: O'Reilly Media, 2021. 350 с.

33. Inzunza G. How to set up Rails, Redis, and Postgres with docker-compose in 2023 // Medium. – 2023. – URL: https://medium.com/@gustavoinzunza/rails-redis-pg-with-docker-compose-in-2023-64ebb6d25de1 (дата звернення: 13.04.2026).
34. Best Practices for Running PostgreSQL in Docker (With Examples) // Sliplane.io. – 2025. – URL: https://sliplane.io/blog/best-practices-for-postgres-in-docker (дата звернення: 13.04.2026).
35. Structured outputs | Gemini API // Google AI for Developers. – 2026. – URL: https://ai.google.dev/gemini-api/docs/structured-output (дата звернення: 13.04.2026).
36. Postman Learning Center // Postman. – URL: https://learning.postman.com/docs/ (дата звернення: 15.04.2026).
37. Socket.IO Documentation // Socket.IO. – URL: https://socket.io/docs/v4/ (дата звернення: 15.04.2026).
38. Chrome DevTools // Google for Developers. – URL: https://developer.chrome.com/docs/devtools/network/ (дата звернення: 15.04.2026).
39. node-postgres: Connection Pooling // node-postgres. – URL: https://node-postgres.com/features/pooling (дата звернення: 15.04.2026).
40. The Node.js Event Loop, Timers, and process.nextTick() // Node.js Official Documentation. – URL: https://nodejs.org/learn/asynchronous-work/event-loop-timers-and-nexttick (дата звернення: 15.04.2026).
41. BullMQ Documentation: Architecture // BullMQ. – URL: https://docs.bullmq.io/ (дата звернення: 15.04.2).

ДОДАТОК А

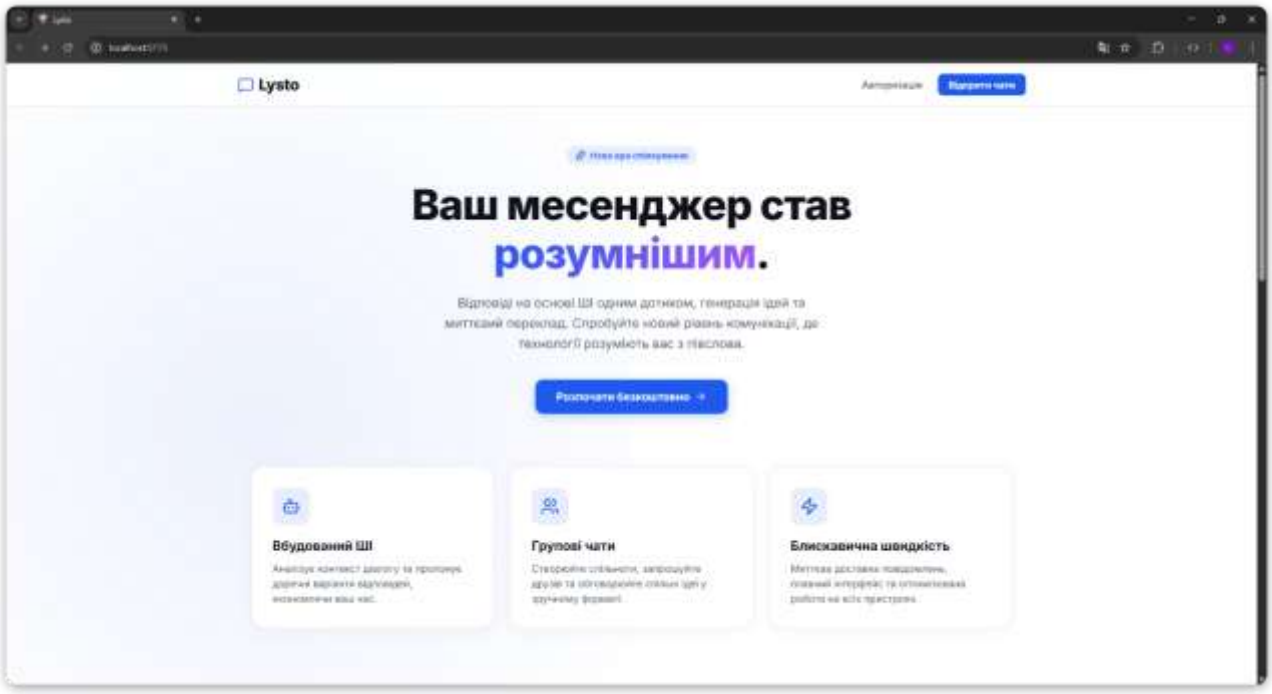
ПОРІВНЯННЯ АРХІТЕКТУРИ МАРШРУТИЗАЦІЇ ЗАПИТІВ У REST ТА GRAPHQL



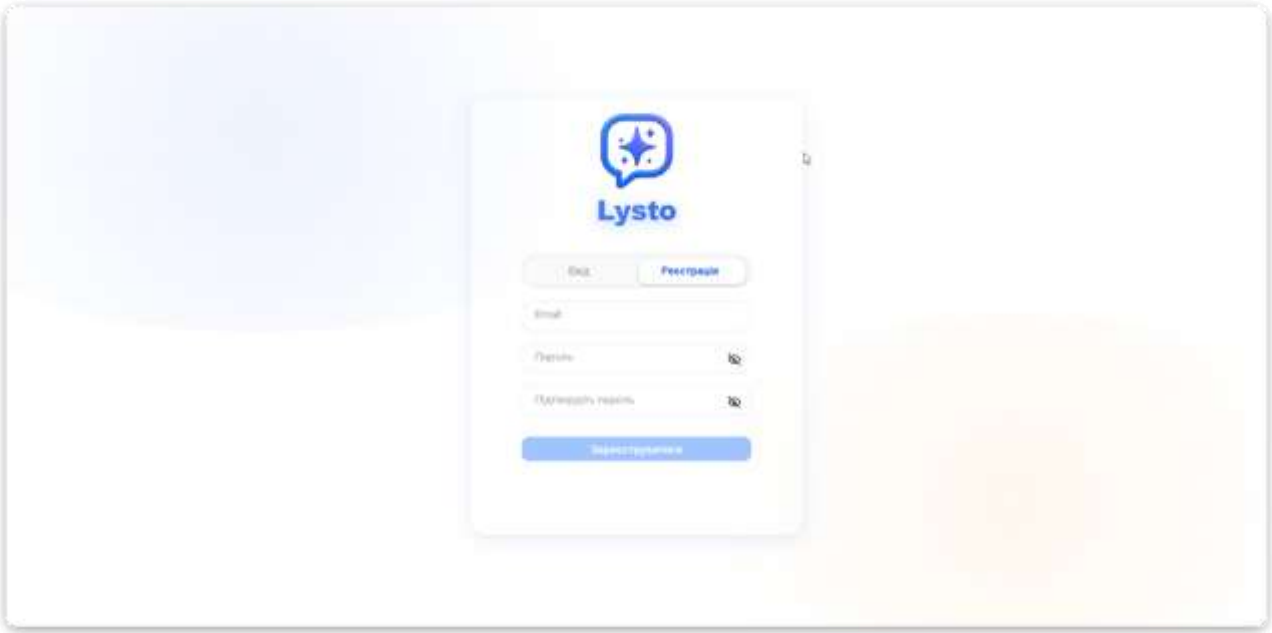
ДОДАТОК Б

ІНТЕРФЕЙС МЕСЕНДЖЕРУ

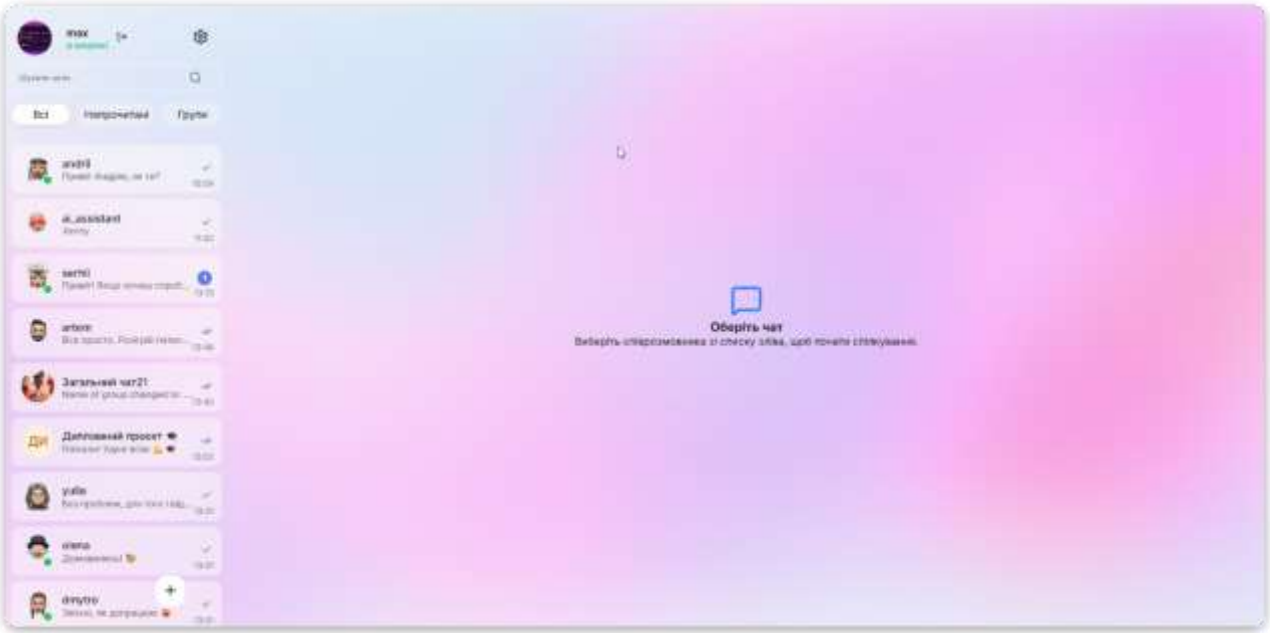
Б.1. Головна сторінка



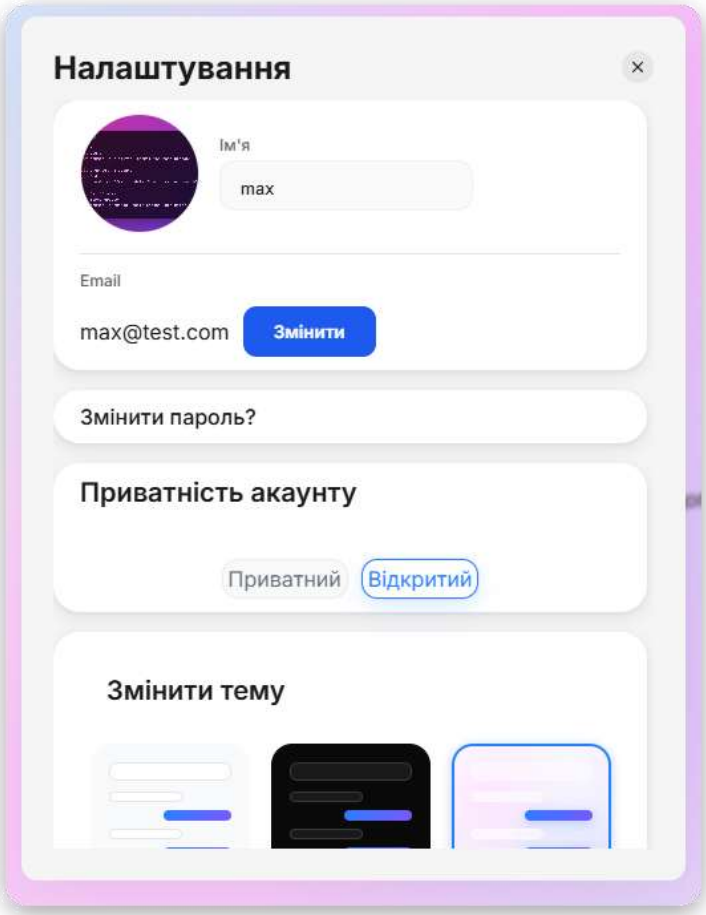
Б.2. Сторінка авторизації



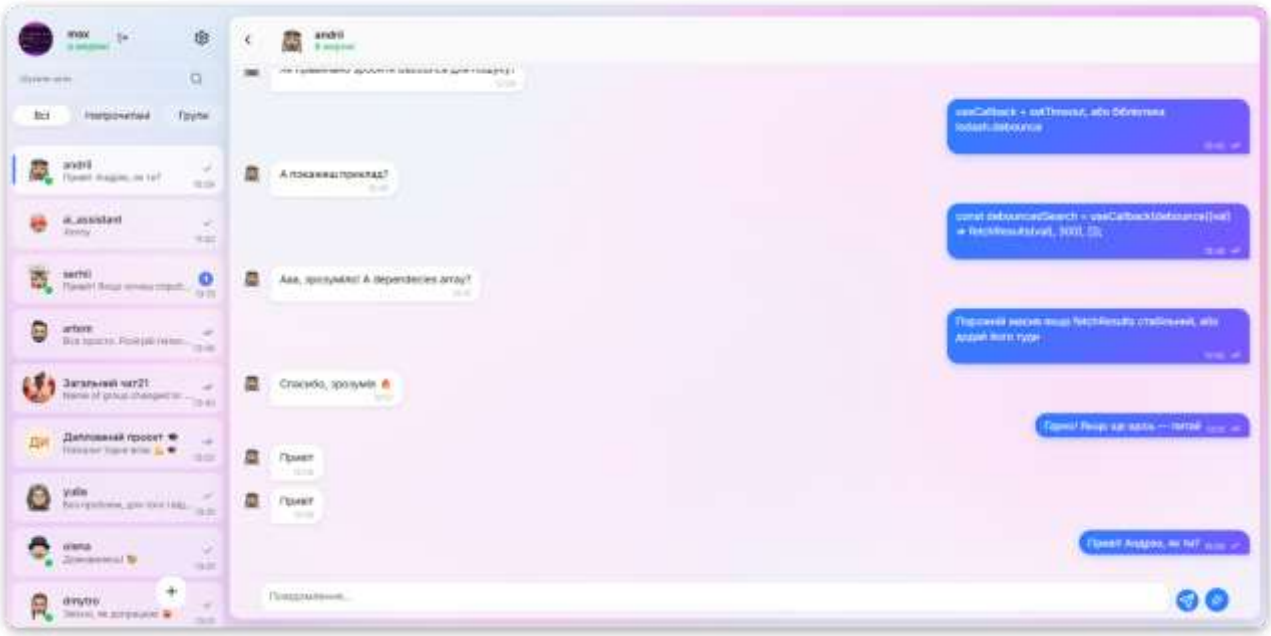
Б.3. Інтерфейс користувача



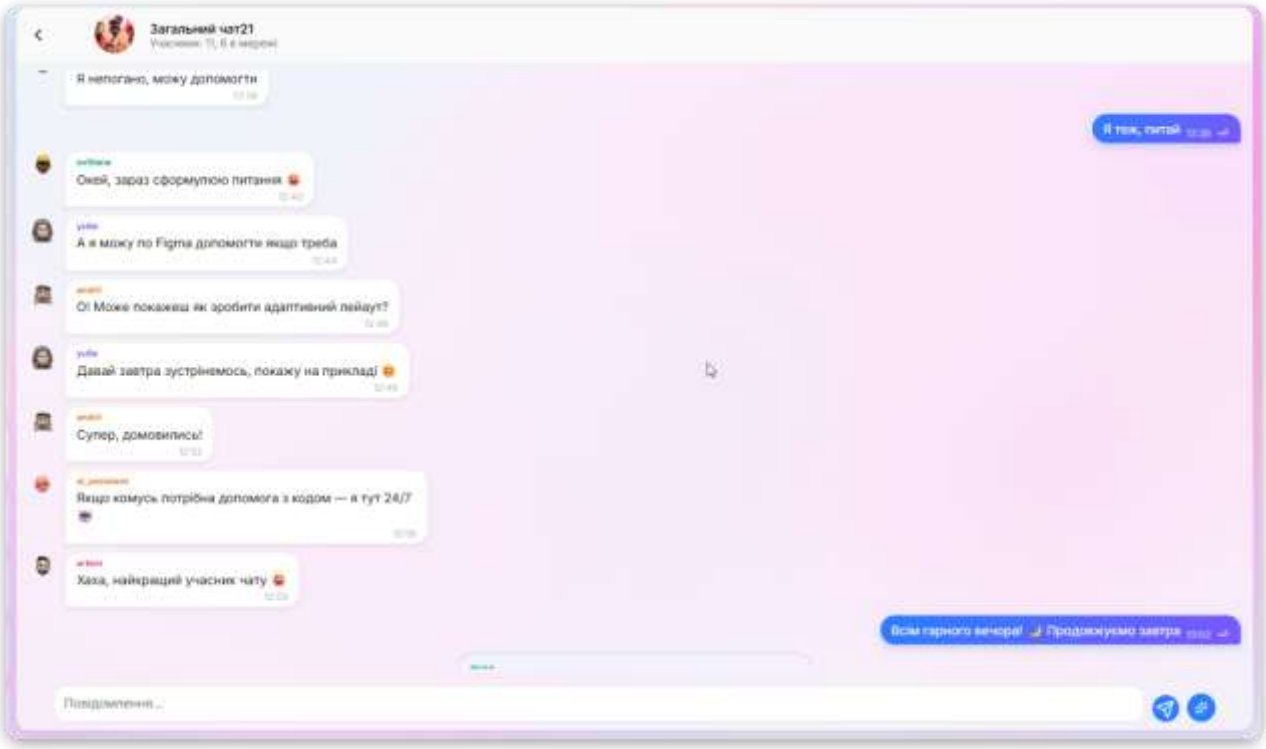
Б.4. Налаштування



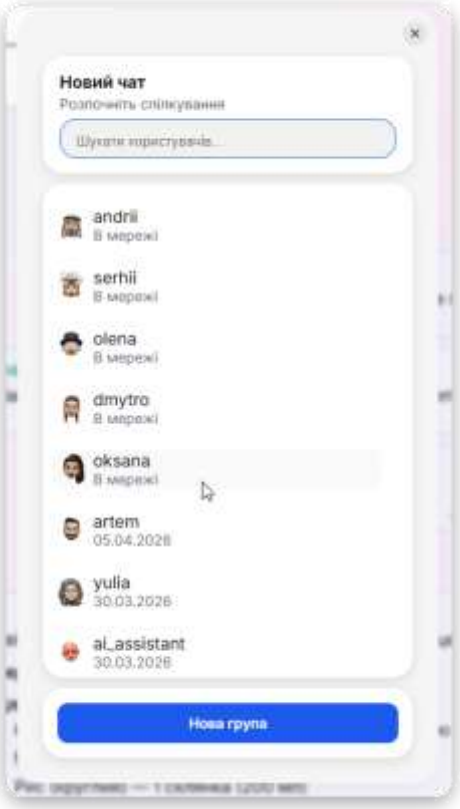
Б.5. Інтерфейс з відкритим чатом



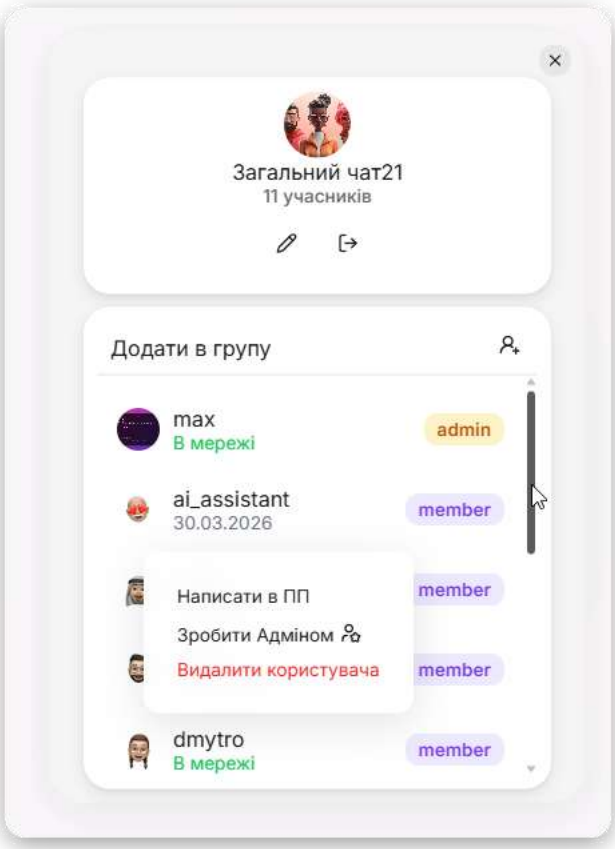
Б.6. Інтерфейс групового чату



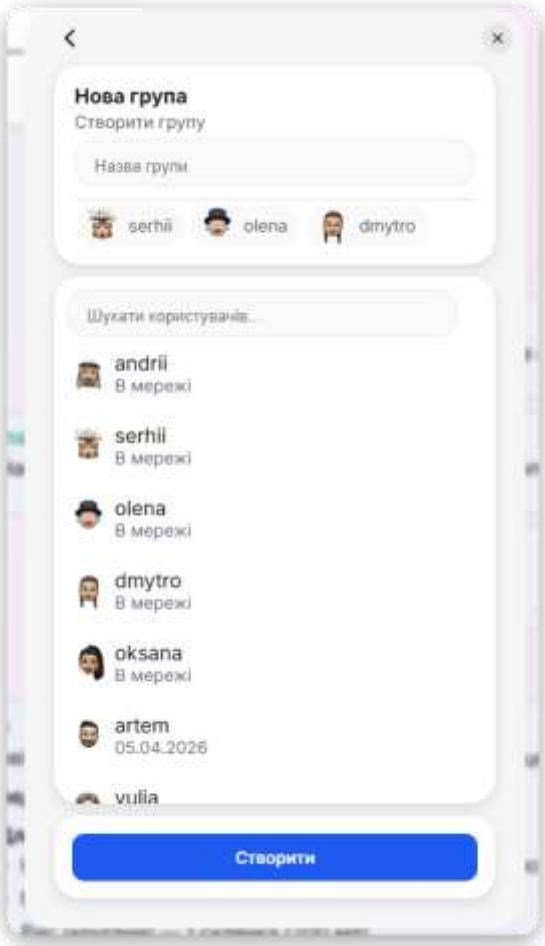
Б.7. Налаштування групового чату



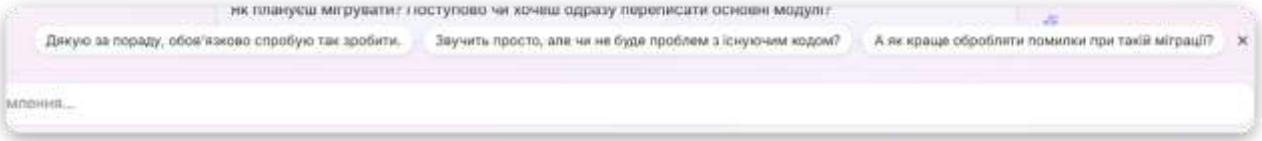
Б.8. Модальне вікно для відкриття\створення чату



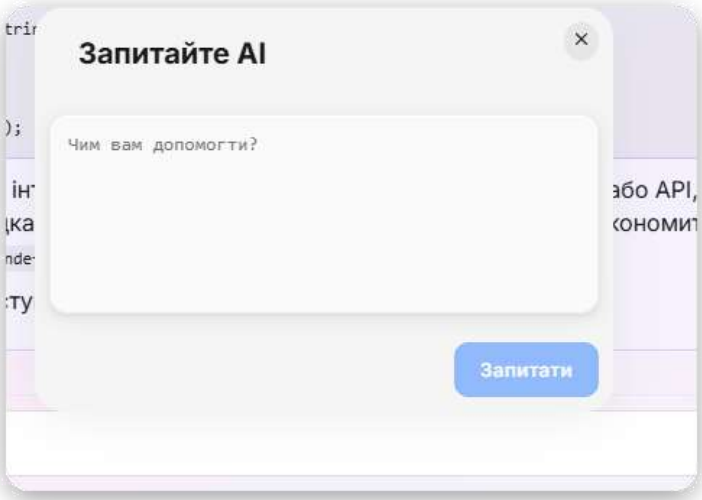
Б.9. Інтерфейс створення групи



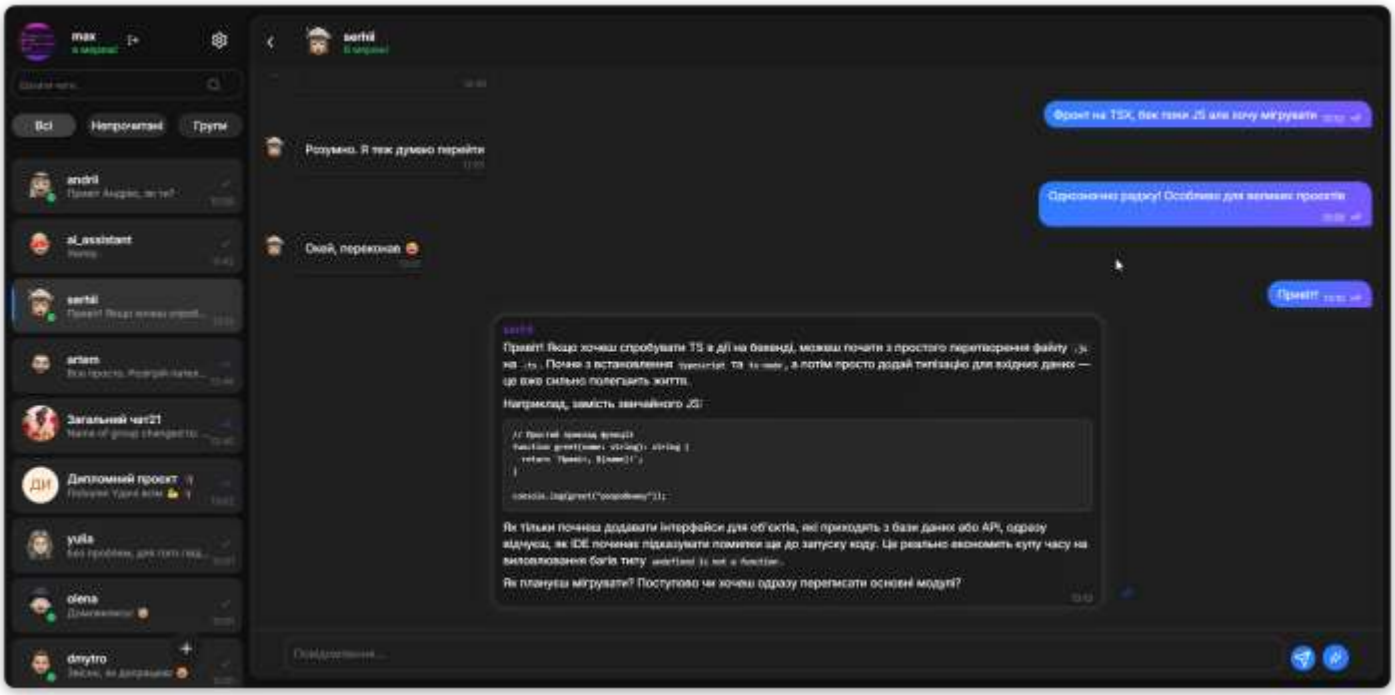
Б.10. Згенеровані підказки від ШІ



Б.11. Вікно для запиту до ШІ



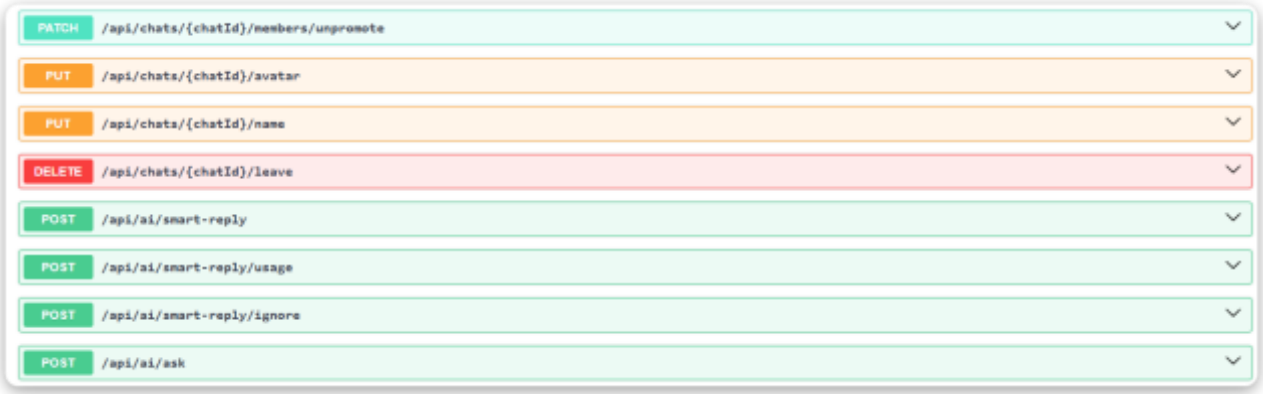
Б.12. Інтерфейс чату при темній темі



ДОДАТОК В

ДОКУМЕНТАЦІЯ API

Додаток В1


PATCH /api/chats/{chatId}/members/unpromote
PUT /api/chats/{chatId}/avatar
PUT /api/chats/{chatId}/name
DELETE /api/chats/{chatId}/leave
POST /api/ai/smart-reply
POST /api/ai/smart-reply/usage
POST /api/ai/smart-reply/ignore
POST /api/ai/ask