Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему **КОНСУЛЬТАТИВНО-ДОВІДКОВА СИСТЕМА ПІДТРИМКИ СТУДЕНТІВ НА БАЗІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ**

Виконав: студент 2 курсу, групи КНТ-514м
спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні системи та мережі

ЗАХАРЧЕНКО К.Р.
(ПРІЗВИЩЕ та ініціали)
Керівник ТЯГУНОВА М.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент ЩЕРБАКОВ В.І.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра «Комп'ютерні системи та мережі»
Ступінь вищої освіти магістерський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)

Освітня програма (спеціалізація) Комп'ютерні системи та мережі
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Зав. кафедри Кудерметов Р.К.
_____
"_15_" __жовтня__ 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ЗАХАРЧЕНКО Костянтина Романовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Консультативно-довідкова система підтримки студентів на базі великих мовних моделей
керівник проєкту (роботи) кан. техн. наук, доцент, ТЯГУНОВА Марія Юріївна
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом вищого навчального закладу від "30" жовтня 2025 року № 491
2. Строк подання студентом проєкту (роботи) 23 грудня 2025 року
3. Вихідні дані до проєкту (роботи) відкриті довідкові матеріали університету як база знань, підходи RAG для пошуку релевантних фрагментів, OpenAI API для доступу до LLM, вебтехнології React/TypeScript та Node.js/Express, система має забезпечувати генерацію відповідей на основі знайденого контексту у форматі чату.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та аналогів системи;
2. Проєктування консультативно-довідкової системи;
3. Реалізація програмних компонентів системи;
4. Тестування та аналіз роботи системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ТЯГУНОМА М.Ю., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області та аналогів системи	01.10.2025 р.	
2	Розробка структури системи	10.11.2025 р.	
3	Розробка алгоритму роботи системи	15.11.2025 р.	
4	Реалізація консультативно-довідкової системи	25.11.2025 р.	
5	Тестування консультативно-довідкової системи	30.11.2025 р.	
6	Оформлення отриманих результатів у ПЗ	05.12.2025 р.	
7	Оформлення графічного матеріалу	10.12.2025 р.	
8	Оформлення допоміжного матеріалу	15.12.2025 р.	

Студент _____ **Костянтин ЗАХАРЧЕНКО**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ **Марія ТЯГУНОВА**
(підпис) (ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 75 с., 28 рис., 10 ліст., 15 джерел.

LLM, REACT, TYPESCRIPT, ВЕБСЕРВІС, ДОВІДКОВА СИСТЕМА, КОНСУЛЬТАТИВНА СИСТЕМА, ПРОМПТ-ІНЖИНІРИНГ, СТУДЕНТСЬКА ПІДТРИМКА

Об’єкт дослідження – консультативно-довідкова система підтримки студентів закладу вищої освіти на базі великих мовних моделей із використанням RAG.

Мета проєкту – підвищення зручності доступу користувачів до довідкової інформації університету шляхом розроблення прототипу чат-асистента, що формує відповіді на основі релевантних фрагментів бази знань і керованих правил генерації.

У першому розділі узагальнено вимоги до довідкових цифрових сервісів і окреслено роль LLM у задачах підтримки користувачів.

У другому розділі сформовано технічні вимоги та описано архітектурну схему рішення з поділом відповідальностей між інтерфейсом користувача і серверним модулем.

У третьому розділі описано реалізацію веб-інтерфейсу на React/TypeScript, API-шлюзу та інтеграції з LLM через OpenAI API, а також формування промпта за схемою SYSTEM + CONTEXT + USER.

У четвертому розділі наведено та проаналізовано результати тестування та рекомендації щодо подальшого масштабування й супроводу.

Практична значущість полягає у можливості впровадження рішення як допоміжного каналу довідкової підтримки, що забезпечує оперативні відповіді 24/7 та зменшує навантаження на працівників підрозділів.

ABSTRACT

Explanatory note: 75 p., 28 figures, 10 listings, 15 references.

LLM, REACT, TYPESCRIPT, WEB SERVICE, INFORMATION SYSTEM, CONSULTATION SYSTEM, PROMPT ENGINEERING, STUDENT SUPPORT

The object of research is a consultation-and-information student support system for a higher education institution based on large language models using RAG.

The aim of the project is to improve user convenience in accessing the university’s reference information by developing a prototype chat assistant that generates answers based on relevant knowledge base fragments and controlled generation rules.

The first chapter summarizes the requirements for digital reference services and outlines the role of LLMs in user support tasks.

The second chapter defines the technical requirements and describes the system architecture with a separation of responsibilities between the user interface and the server module.

The third chapter presents the implementation of the web interface in React/TypeScript, the API gateway, and integration with an LLM via the OpenAI API, as well as prompt construction using the SYSTEM + CONTEXT + USER scheme.

The fourth chapter presents and analyzes the testing results and recommendations for further scaling and maintenance.

The practical significance lies in the possibility of deploying the solution as an auxiliary reference support channel that ensures prompt 24/7 responses and reduces the workload of departmental staff.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

LLM – Large Language Model, Велика мовна модель

RAG – Retrieval-Augmented Generation, Генерація з доповненним пошуком

AI – Artificial Intelligence, Штучний інтелект

NLP – Natural Language Processing, Обробка природної мови

FAQ – Frequently Asked Questions, Поширені запитання

API – Application Programming Interface, Прикладний програмний інтерфейс

HTTP – HyperText Transfer Protocol, Протокол передачі гіпертексту

JSON – JavaScript Object Notation, Запис об'єктів JavaScript

DOM – Document Object Model, Об’єктна модель документа

CSS – Cascading Style Sheets, Каскадні таблиці стилів

URL – Uniform Resource Locator, Уніфікований локатор ресурсу

UI – User Interface, Інтерфейс користувача

UX – User Experience, Користувацький досвід

CORS – Cross-Origin Resource Sharing, Спільне використання ресурсів між різними джерелами

ВНЗ – Вищий навчальний заклад

ВСТУП

У сучасному цифровому середовищі обсяги доступної інформації швидко зростають. У таких умовах традиційні веб-інтерфейси з ієрархічною навігацією не завжди забезпечують достатньо швидкий і зручний доступ до потрібних відповідей, особливо коли користувачеві необхідно оперативно знайти конкретне правило, розклад, контакт або регламент. Це підсилює потребу в інструментах, які підтримують природномовний пошук і подання інформації у формі діалогових повідомлень.

Одним із практично придатних напрямів розвитку довідкових сервісів є впровадження чат-ботів у сфері освіти. У наукових оглядах зазначається, що такі системи можуть забезпечувати масштабовану підтримку студентів і співробітників, зокрема через оперативне опрацювання типових запитів і цілодобову доступність сервісу. Водночас використання великих мовних моделей потребує механізмів, які обмежують генерацію рамками перевірених даних і зменшують ризик некоректних припущень.

Для розв’язання цього завдання у роботі розглянуто консультативно-довідкову систему у форматі чат-бота, що поєднує велику мовну модель із підходом Retrieval-Augmented Generation. RAG передбачає формування відповіді на основі попередньо відібраних фрагментів із зовнішнього сховища знань, які додаються до контексту генерації, що підвищує обґрунтованість результату та спрощує контроль джерел.

Метою кваліфікаційної роботи є підвищення зручності доступу користувачів до довідкової інформації університету шляхом розроблення прототипу чат-асистента, що формує відповіді на основі релевантних фрагментів бази знань і керованих правил генерації. Об’єктом дослідження є вебсистеми надання довідкової інформації користувачам. Предметом дослідження є методи й програмні засоби побудови діалогових інтерфейсів із доступом до доменної бази знань.

Для досягнення поставленої мети у роботі передбачено виконання комплексу взаємопов’язаних завдань, що охоплюють аналітичний і прикладний етапи. Зокрема, необхідно проаналізувати предметну область і наявні підходи до організації консультативно-довідкових сервісів, визначити вимоги до системи, спроєктувати її структуру та логіку функціонування, реалізувати програмні компоненти чат-бота з підтримкою пошуку в базі знань, а також провести перевірку працездатності та якості роботи на типових сценаріях використання і узагальнити результати в пояснювальній записці.

Елементи наукової новизни пов’язані з упорядкуванням і формалізацією принципів формування відповіді у консультативно-довідковому сервісі на основі поєднання мовної моделі з пошуком у доменній базі знань. Новизна полягає в описі керованої схеми використання контексту, за якої відповідь будується з опорою на релевантні фрагменти інформації, а за їх відсутності застосовується коректний механізм повідомлення про недостатність даних для надання обґрунтованої відповіді.

Практична цінність роботи полягає у можливості використання розробленої системи як цифрового асистента для студентів закладу вищої освіти, що забезпечує цілодобовий доступ до довідкової інформації та може бути адаптоване для інших освітніх установ або розширене інтеграцією з внутрішніми інформаційними системами.

Пояснювальна записка складається зі вступу, чотирьох розділів, висновків і переліку джерел посилань. У вступі обґрунтовано актуальність теми, сформульовано мету, об’єкт і предмет дослідження. У першому розділі наведено аналітичний огляд предметної області та підходів до побудови консультативно-довідкових сервісів. Другий розділ присвячено проєктуванню системи, зокрема визначенню вимог і вибору архітектурних та технологічних рішень. У третьому розділі описано реалізацію ключових компонентів і логіку їх взаємодії. У четвертому розділі подано підсумкові результати виконання роботи та їх стислий аналіз з погляду досягнення поставленої мети.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛОГІВ СИСТЕМИ

У цьому розділі розглядаються передумови створення консультативно-довідкової системи для студентів, проаналізувавши наявні цифрові освітні сервіси. Увага зосереджена на тому, як різні покоління рішень, такі як сценарні чат-боти, FAQ-платформи та асистенти на базі великих мовних моделей, задовольняють потреби користувачів в оперативній та достовірній інформації. Порівняння методів дозволяє з’ясувати недоліки традиційних систем, зокрема їхню нестабільність і залежність від вручну створеної бази відповідей. Крім того, це дозволяє з’ясувати переваги LLM у контексті запиту та роботі з природною мовою. Аналіз також включає вивчення технології RAG, яка дозволяє поєднати потенціал генеративних моделей із доступом до поточних даних.

1.1 Аналіз сучасних цифрових рішень у сфері освітніх асистентів

У сфері освіти вже є широкий вибір цифрових помічників, які можуть допомогти студентам. Перші варіанти таких систем були досить простими. Вони включали чатботи, які працювали за сценаріями або шаблонами, а також FAQ-системи, які давали заздалегідь підготовлені відповіді на типові запитання. Рішення, схожі на ці, використовують веб-чати, розміщені на сайтах університетів або освітніх порталів. Наприклад, у 2016 році Jill Watson, віртуальний помічник на базі IBM Watson, був створений, щоб відповідати на запитання студентів у онлайн-курсі. Цей штучний інтелект також успішно імітував функції викладацького помічника, а також обробляв запити на форумі курсу. Ранні системи демонстрували принципову можливість автоматизації консультацій, але вони мали деякі недоліки: вони базувалися на шаблонних

фразах і сценаріях і не могли виходити за рамки стандартних ситуацій. З цієї причини розмови стали одноманітними та негнучкими.

Чатботи, які зараз існують завдяки розвитку машинного навчання, можуть використовувати бази знань і компоненти природнього лінгвістичного процесу (NLP), щоб забезпечити більш інтелектуальні відповіді. Віртуальні помічники, такі як чатботи для студентів або вступників, набули популярності в освітній сфері. Нещодавня розробка НТУ «ХПІ» є одним з прикладів, що зображено на рисунку 1.1. На сайті kpi.kharkov.ua було створено інтелектуального чатбота, який допомагає абітурієнтам краще підготуватися до вступу. Цей веб-застосунок підтримує персоналізований діалог і поєднує адаптивний опитувальник із сучасною мовною моделлю (інтегрованою GPT-4o Mini).

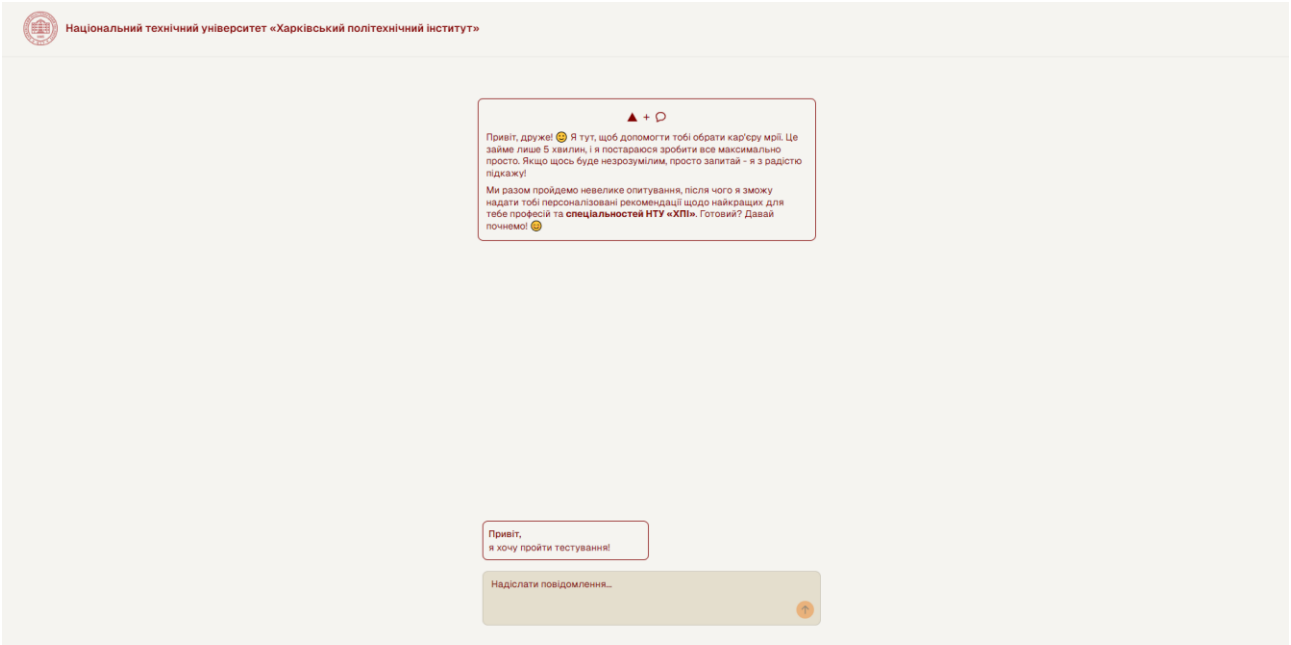

Рисунок 1.1 – Інтерфейс чатбота від НТУ «ХПІ»

Бот імітує реальну дискусію, змінюючи запитання залежно від відповідей користувача. Важливо, що чатбот доступний 24/7 і особливо корисний для майбутніх студентів з віддалених регіонів. Він не замінює повністю консультації приймальної комісії, а служить зручним початком для отримання інформації на kpi.kharkov.ua [2].

У всьому світі розвиваються універсальні моделі штучного інтелекту. Зокрема, модель ChatGPT, яка була запущена OpenAI у 2022 році, продемонструвала здатність генерувати детальні відповіді на довільні запити та навчальні завдання. З розвитком можливостей моделей штучного інтелекту також розширилась їх використання у різних сферах. Так, Khan Academy створила асистента Khanmigo, який спеціалізується на поясненні навчального матеріалу та допомозі з задачами, Quizlet має бота Q-Chat, який проводить інтерактивне навчання знань, а додатки, такі як Photomath, які розв'язують математичні задачі з поетапними поясненнями, і Elsa Speak, які навчають англійської мови. Усі ці інструменти мають різні функції, але їх об'єднує можливість взаємодіяти як у діалоговому режимі, коли користувач ставить запитання, а система надає розгорнуту відповідь або пораду [3].

Останніми мовними моделями можна значно покращити цифрові освітні рішення. Особливі версії LLM, такі як Scholar GPT для академічних досліджень, навчаються відповідати академічним вимогам стилю навчання та можуть автоматизувати рутинні наукові завдання (пошук літератури, форматування цитувань тощо) [4]. Загалом, використання штучного інтелекту в освіті має багато переваг. По-перше, ШІ-асистенти дозволяють розширювати консультаційну підтримку, оскільки один бот може одночасно допомагати необмеженій кількості студентів, знімаючи навантаження з викладачів, відповідаючи на стандартні запитання студентів. По-друге, такі системи можуть працювати 24/7, що особливо важливо для студентів, які навчаються дистанційно або за індивідуальним планом. По-третє, персоналізація, сучасні моделі можуть надавати більш відповідні рекомендації, аналізуючи історію запитів студента. Крім того, великі моделі можна навчити на внутрішніх матеріалах університету, таких як розклади та методичні вказівки. У такому випадку ці матеріали стануть єдиним вікном для доступу до інформації закладу. LLM дозволяє швидко створювати підручники, конспекти, посібники та інші матеріали, що дозволяє викладачам зосередитися на творчих аспектах навчання. Таким чином, сучасні аналоги варіюються від простих FAQ-ботів до

потужних LLM-асистентів, які є частиною освітнього середовища, і досвід їх використання демонструє високу продуктивність за правильного налаштування. Щоб визначити сильні та слабкі сторони, необхідно порівняти методи, що лежать в основі цих рішень.

1.2 Порівняльний аналіз шаблонних підходів, LLM та RAG

Технології, що використовуються для побудови консультативно-довідкових систем для студентів можна умовно поділити на три покоління. Шаблонні системи першого покоління, що базуються на наперед заданих правилах. Системи другого покоління, що базуються на LLM та гібридні системи, що поєднують RAG та LLM із механізмами пошуку та актуалізації інформації.

Традиційні чатботи працюють за певними сценаріями, вони складаються з набору шаблонів запитань і відповідей або дерева діалогу, які розробили розробники. Бот видає відповідь, пов’язану з вхідним повідомленням, коли користувач зв’язується з ботом. Перевага цього підходу полягає в передбачуваності та контролі: усі відповіді перевіряються людьми, тому бот не створює нічого непередбачуваного або хибного. Крім того, шаблонний бот не потребує великих ресурсів і працює швидко. Подібні боти можуть відповідати на часто задані запитання щодо умов вступу, розкладу, правил поселення тощо. Однак основним недоліком шаблонної системи є її низька гнучкість. Користувачеві потрібно створювати запити таким чином, щоб вони відповідали закладеному варіанту (або ключовим словам), будь-яке нестандартне запитання поставить бота в глухий кут. Приклад роботи сценарного чатбота зображено на рисунку 1.2. Незважаючи на те, що ручне розширення бази фраз частково вирішує проблему, масштабування вимагає великої кількості зусиль. Отже, коли сфера знань широка (наприклад, кожен аспект життя університету),

покриття кожного запиту шаблонами практично неможливо. Крім того, бот на основі сценаріїв не вчиться з досвіду – він не покращує свої відповіді, поки програміст не зробить це очевидним. Шаблонні рішення поступово витісняються потужнішими штучним інтелектом у наш час, коли очікується більш «розумна» взаємодія. Але деякі правила все ще корисні. Наприклад, формалізація діалогу за допомогою певної структури (привітання, уточнення питання, надання довідкової інформації та прощання) може використовуватися як шаблон поверх гнучкої моделі.

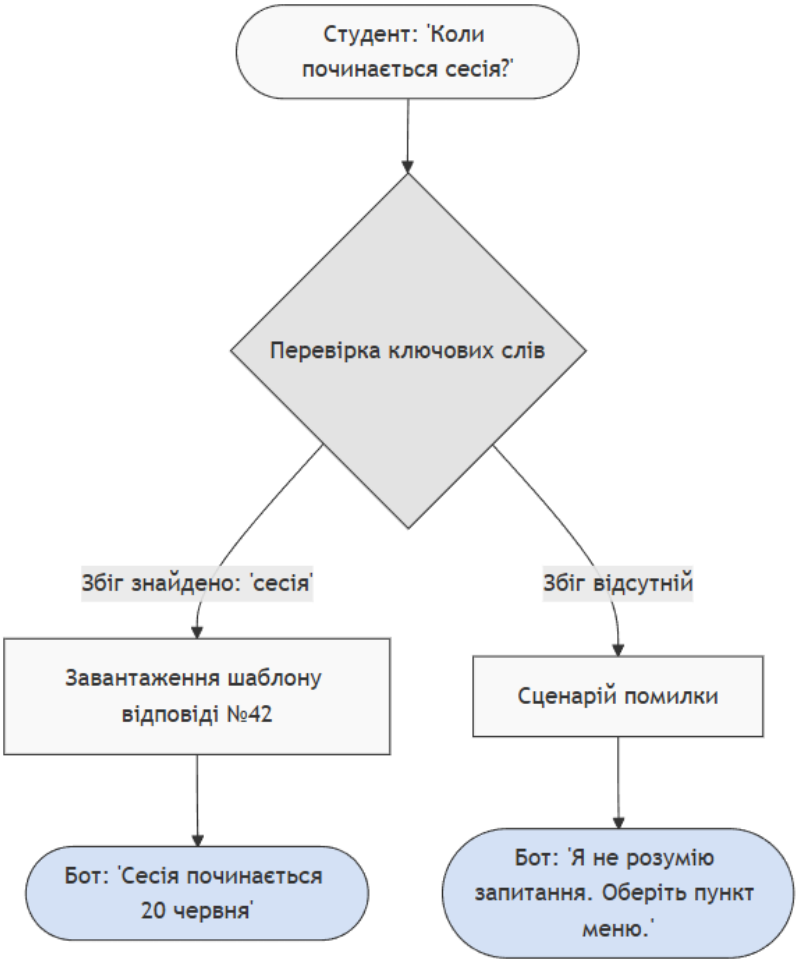

Рисунок 1.2 – Приклад алгоритму сценарного чатбота

Чатботи другого покоління побудовані на основі великої мовної моделі (наприклад, GPT-4). На відміну від шаблонних систем, такі системи генерують

відповіді динамічно на основі ймовірнісної моделі, навченої на великій кількості текстів. Рішення з використанням LLM мають ряд переваг. Вони мають здатність розуміти запитання, сформульовані природною мовою, навіть якщо розробник не передбачив їх, а відповіді створюються в реальному часі та можуть включати нові формулювання, пояснення та приклади, тобто все, що модель знає з даних свого навчання. Наприклад, асистент з LLM може пояснити складне наукове поняття простішими термінами або проаналізувати запит студента та запропонувати пораду, незважаючи на те, що таке запитання раніше ніколи не задавалося. Сила LLM – креативність і адаптивність. Однак є і недоліки прямого використання великої моделі. По-перше, відсутність актуалізації знань: модель не знає нових фактів після завершення свого навчання, оскільки вона навчилася на певному статичному корпусі даних. Це важливо для освітнього бота, оскільки інформація (про програми, контакти та розпорядки) постійно змінюється. LLM не може видалити застарілі дані без додаткових інструментів. По-друге, моделі часто помиляються. Це може призвести до того, що бот помилково порадить хибний порядок дій чи помилкову довідку під час консультації студентів, що неприйнятно. По-третє, якщо питання стосується конкретних або локальних даних, LLM часто дають надто загальні відповіді. Також якщо відповідь на запитання не була включена у данні на яких навчалась мовна модель та які не є загальнодоступними та такими які можна знайти за допомогою пошукового запиту, модель здатна надати загальну відповідь. Відповідь яка здається правильною але є фактично хибною. Приклад процесу взаємодії з таким типом чатботів зображений на рисунку 1.3. Для успішного впровадження модель повинна працювати під наглядом – з обмеженнями та можливістю валідації її відповідей.

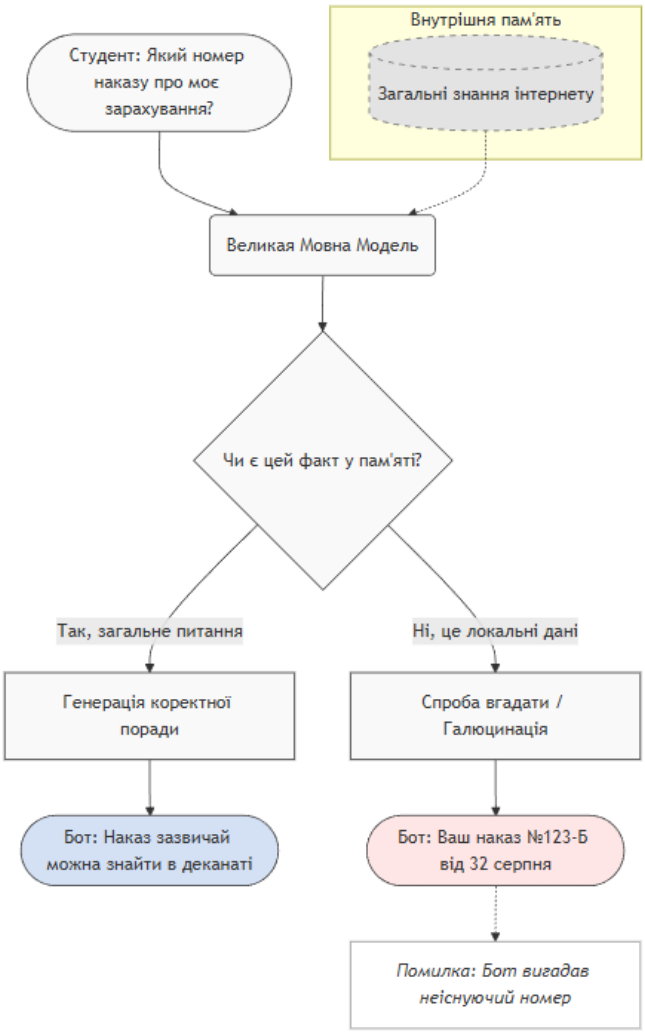

Рисунок 1.3 – Сценарій роботи чатбота на LLM

Найновішим підходом є поєднання переваг двох попередніх стратегій: гнучкості LLM і надійності перевіреної бази знань. RAG стверджує, що перед генерацією відповіді модель спочатку отримує важливі дані з зовнішнього джерела, такого як бази знань, а потім включає їх у свій контекст. Схема RAG складається з двох модулів: модуля пошуку інформації та модуля генерації відповіді. На практиці це виглядає таким чином: запит студента спочатку надсилається в систему пошуку, яка знайде відповідні документи або записи в базі даних університету. Ці документи можуть включати витяги з правил навчання, сторінки підручника, розклади тощо. Велика мовна модель створює кінцеву відповідь, надсилаючи знайдений фрагмент до запиту разом із розширеним контекстом. У цьому методі можна подолати обмеження LLM

щодо актуальності та точності, оскільки модель отримує актуальну та перевірену інформацію перед генерацією. Таким чином, відповідь містить реальні факти, а не лише те, що зберігається в пам’яті моделі. Крім того, система може підвищити довіру надаючи користувачеві посилання на джерела інформації у відповіді. Наприклад, RAG-асистент у освіті може діяти прозоро та доказово, не лише пояснюючи правило чи процедуру, але й цитуючи відповідний пункт положення чи сторінку підручника. Приклад алгоритму взаємодії з чатботом на такій системі зображено на рисунку 1.4.

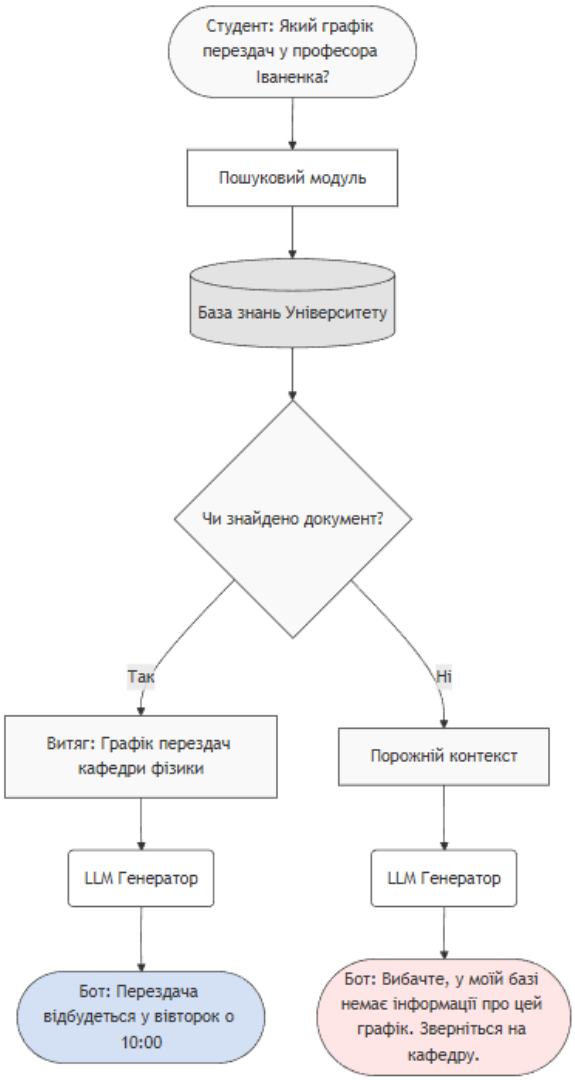

Рисунок 1.4 – Сценарій роботи чатбота на LLM з RAG

RAG-система складніша, ніж «чиста» LLM, оскільки вона вимагає впровадження додаткових процедур, таких як процес пошуку та індексування документів. Однак очевидні переваги цієї архітектури – вона об’єднує здатність генеративної моделі та надійність даних. RAG також виграє з точки зору продуктивності: компактна база даних і швидкий пошук достатньо, щоб тренувати гігантську модель на всі можливі факти. За словами експертів, цей метод є економічно вигідним способом додати нові знання до LLM, не перенавчаючись повністю. Фреймворки, такі як LangChain і LlamaIndex, вже з’явилися, щоб зробити інтеграцію генерації та пошуку більш простою. Таким чином, порівняльний аналіз показує, що шаблонні системи забезпечують контроль і простоту, але не гнучкість, LLM-асистенти дають багато відповідей, але ризикують бути менш точними, підхід RAG намагається поєднати переваги обох підходів, поєднуючи їх [9].

1.3 Практичні висновки щодо інтеграції LLM-асистента у вищу освіту

Аналіз сучасних технологій та рішень дозволяє сформулювати низку рекомендацій для впровадження консультативно-довідкової системи на базі великих мовних моделей у закладах вищої освіти. Передусім, результати переддипломного дослідження підтвердили доцільність обраного підходу – поєднання LLM з механізмом пошуку (RAG) для створення студентського асистента. Така система здатна забезпечити прийнятний баланс між інтелектуальністю відповідей та їхньою коректністю. Практичні експерименти показали, що різні реалізації LLM мають свої профілі сильних і слабких сторін. Наприклад, модель ChatGPT переважає у швидкості генерації відповіді, але припускається фактичних неточностей або помилок форматування, тоді як Gemini від Google дає більш стислі, структуровані та коректні відповіді, проте

займає багато часу на відповідь. Це підкреслює необхідність тонкого налаштування вибраної мовної моделі під конкретні завдання освітнього асистента, а також впровадження механізмів перевірки виходу. Одним із шляхів є реалізація режиму роботи з низькою стохастичністю (temp=0-0.3), що зменшує креативність моделі, але підвищує повторюваність результатів – для довідкової системи це часто виправдано.

З технічної точки зору, впровадження LLM-асистента у ВНЗ вимагає створення відповідної інфраструктури: збирання і постійного оновлення корпусу релевантних даних (навчальні програми, розклади, положення, FAQ тощо), налаштування векторної бази даних для швидкого пошуку, а також виділення обчислювальних ресурсів для роботи мовної моделі. Важливо забезпечити інформаційну безпеку: доступ до внутрішніх даних має бути контрольованим, щоб модель не розкривала конфіденційну інформацію у відповідях. Крім того, необхідно передбачити механізм зворотного зв’язку від користувачів (студентів і співробітників) – це допоможе виявляти помилки або прогалини у знаннях бота і оперативно їх коригувати.

На завершення, слід підкреслити, що впровадження AI-асистента – не разова акція, а постійний процес вдосконалення. Технології ШІ розвиваються дуже швидко, і університетська система має бути готова оновлювати свого цифрового помічника, інтегрувати нові моделі, алгоритми й найкращі практики. Уже зараз видно, що LLM-технології докорінно змінюють підходи до роботи з інформацією, і правильне їх використання здатне вирішити багато задач, які раніше здавалися непідйомними. Така система, за умови відповідальної інтеграції, підвищить ефективність навчального процесу, розвантажить персонал від типових запитів, забезпечить студентам швидкий доступ до необхідної інформації та сприятиме формуванню у них навичок взаємодії з сучасними цифровими технологіями.

2 ПРОЄКТУВАННЯ КОНСУЛЬТАТИВНО-ДОВІДКОВОЇ СИСТЕМИ

2.1 Вимоги до системи

Завданням розроблюваної системи є надання довідкової інформації у режимі діалогу на основі LLM із застосуванням технології RAG, яка забезпечує формування відповідей із використанням релевантних фрагментів внутрішньої бази знань. Система призначена для роботи з відкритими інформаційними матеріалами університету та орієнтована на використання через вебінтерфейс.

2.1.1 Функціональні вимоги

У системі розрізняються логічні ролі кінцевого користувача та адміністратора. Кінцевими користувачами можуть бути студент, абітурієнт або працівник університету, при цьому їхні права є однаковими, оскільки система працює виключно з відкритими довідковими даними і не передбачає механізмів автентифікації чи персоналізації. Адміністратор здійснює керування параметрами роботи системи, зокрема вибором LLM-провайдера з переліку реалізованих інтеграцій та актуалізацією бази знань.

Система має реалізовувати користувацький інтерфейс у вигляді вебзастосунку, що забезпечує введення запитів у діалоговому режимі та відображення відповідей у структурованому вигляді, придатному для швидкого сприйняття. Інтерфейс повинен підтримувати основний сценарій роботи користувача – формулювання запиту природною мовою та отримання відповіді, сформованої на основі знань системи, а також коректно обробляти типові помилки введення та мережеві збої.

Історія діалогу не зберігається на сервері та не використовується під час обробки нового повідомлення. Повідомлення, відображені у вікні чату, зберігаються лише у стані клієнтського інтерфейсу та скидаються після перезавантаження сторінки або перезапуску застосунку, унаслідок чого кожен новий запит обробляється незалежно від попереднього діалогу.

Серверна частина повинна бути спроєктована як окремий компонент, реалізований на сучасній мові програмування та здатний надавати API-шлюз для взаємодії з клієнтською частиною і зовнішнім провайдером LLM. Такий шлюз має приймати запити клієнта, виконувати попередню обробку, формувати запит до мовної моделі та повертати результат у стандартизованому форматі, щоб фронтенд міг однозначно інтерпретувати відповідь і відобразити її користувачу.

Окремою загальною вимогою є підтримка підходу RAG, тобто здатність системи розширювати вхідний запит до LLM релевантним контекстом із внутрішніх джерел даних. Це означає, що перед генерацією відповіді система повинна виконувати добір необхідних фрагментів інформації, узгоджувати їх із запитом користувача та включати до промпту у контрольованому вигляді. Такий механізм потрібен для підвищення точності та зменшення ризику вигаданих тверджень, а також для того, щоб відповіді спиралися на актуальні та перевірені дані.

Система повинна підтримувати перемикання LLM-провайдера з переліку вже реалізованих інтеграцій шляхом зміни конфігураційного параметра серверної частини. Перемикання провайдера повинно виконуватися без внесення змін у програмний код і вимагати лише оновлення конфігурації та перезапуску серверного застосунку. Незалежно від обраного провайдера має зберігатися єдиний контракт взаємодії з клієнтською частиною, зокрема незмінний формат запиту та відповіді API, а також уніфікована обробка помилок для забезпечення однакового способу відображення результату на клієнтській частині.

2.1.2 Нефункціональні вимоги

Система повинна працювати як консультативно-довідковий сервіс для студентів та абітурієнтів у форматі діалогу й надавати відповіді українською мовою за замовчуванням, дотримуючись офіційного академічного стилю викладу та узгоджених правил подання інформації у відповідях.

З погляду продуктивності система повинна забезпечувати прийнятний час обробки запитів у типових умовах експлуатації. Під час тестування на підготовленому наборі запитів середній час формування відповіді не повинен перевищувати 10 секунд, а час відповіді у 99% випадків не повинен перевищувати 25 секунд, що забезпечує придатність сервісу для інтерактивної консультації. Також система має коректно обробляти щонайменше 5 паралельних звернень без аварійного завершення процесу та без порушення коректності відповідей.

З погляду надійності система повинна зберігати працездатність при збоях зовнішніх сервісів. У разі недоступності або помилки LLM-провайдера серверна частина має повертати стандартизоване повідомлення про помилку у форматі, придатному для однозначної інтерпретації клієнтом.

З погляду безпеки конфігураційні параметри та секрети доступу повинні зберігатися поза межами програмного коду у змінних середовища. Серверна частина повинна застосовувати політику CORS для обмеження доступу до API лише дозволеними джерелами клієнтського застосунку, а також підтримувати базове обмеження частоти запитів з метою зниження ризику зловживань. Оскільки система не передбачає автентифікації та використаення cookie-сесії, основним механізмом зниження ризиків міжсайтових атак у межах клієнт–серверної взаємодії має виступати контроль джерела запитів через CORS та обмеження запитів до API [10].

2.2 Технології розробки

Для реалізації прототипу консультативно-довідкової системи обрано сучасний стек веброзробки, який забезпечує швидке створення користувацького інтерфейсу, централізовану серверну логіку та зручну інтеграцію з великими мовними моделями. Основу програмної реалізації

становлять TypeScript, React і Express, а доступ до LLM організовано через API OpenAI. Збірка та запуск клієнтського застосунку виконуються за допомогою Vite, що спрощує налаштування середовища розробки та пришвидшує ітерації [11].

Клієнтська частина реалізується як вебзастосунок на базі React. Компонентний підхід дозволяє ізолювати елементи інтерфейсу діалогу, керувати станами введення та відображення повідомлень, а також забезпечувати оновлення сторінки без повного перезавантаження, життєвий цикл цих компонентів зображено на рисунку 2.1. Використання TypeScript на рівні клієнта підвищує надійність коду завдяки статичній типізації та зменшує кількість помилок під час розвитку функціональності [12]. Vite застосовано як інструмент розробки і збірки, який забезпечує швидкий старт проєкту, зручну організацію коду та оптимізовану роботу в режимі розробки.

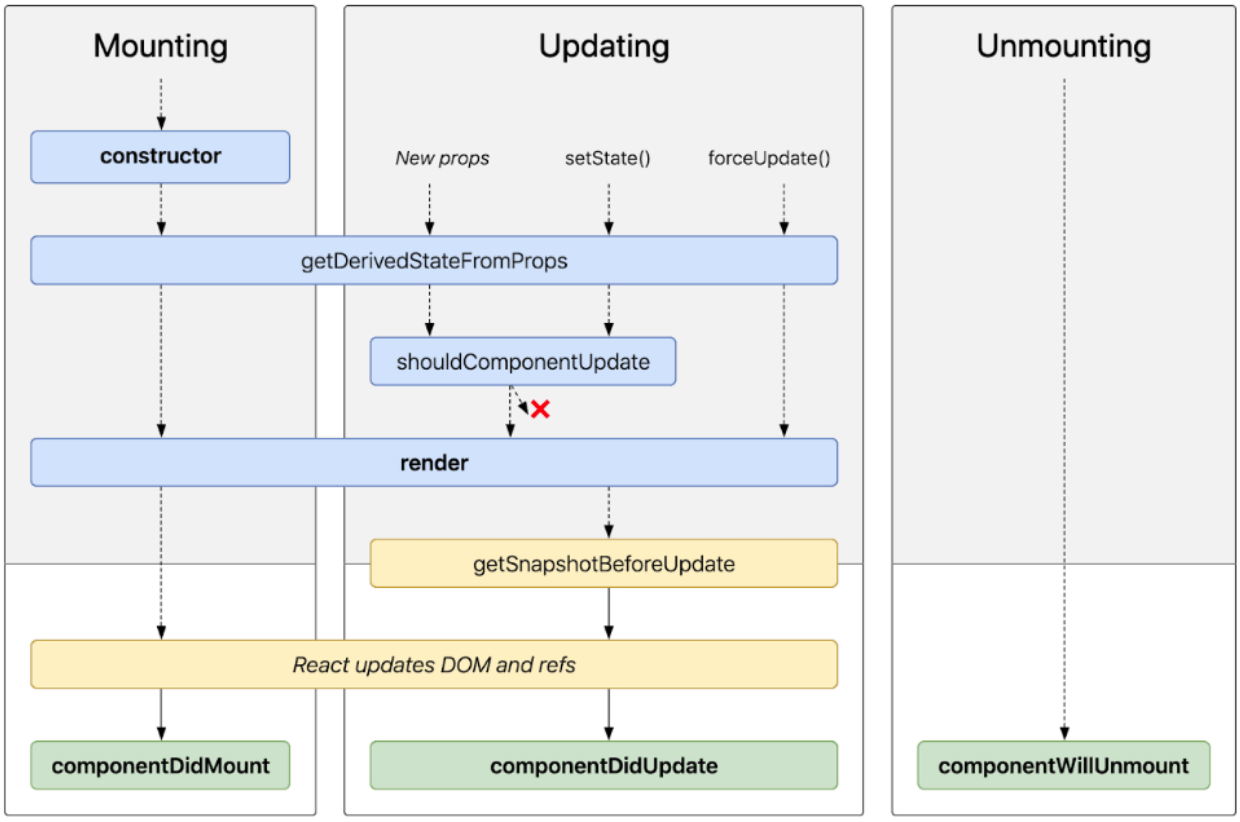

Рисунок 2.1 – Життєвий цикл компонента у React

Серверний рівень побудовано на Express з використанням TypeScript, що дозволяє сформувати компактний API-шлюз між клієнтською частиною та сервісом LLM. Сервер приймає запити у форматі HTTP/JSON, виконує попередню обробку вхідних даних, формує уніфіковану структуру запиту та повертає відповідь у стандартизованому вигляді, придатному для однозначного відображення у вебінтерфейсі. Таке розмежування відповідальностей зменшує зв’язність компонентів і спрощує подальший супровід системи.

Інтеграція з LLM здійснюється через OpenAI API, де як генеративний компонент використовується ChatGPT. Виклики до моделі виконуються виключно на серверній стороні, що дозволяє централізовано керувати параметрами генерації, контролювати формат відповіді та застосовувати єдині правила поведінки асистента. Для підтримки розширюваності в архітектурі передбачено єдиний інтерфейс провайдера та фабричний механізм створення реалізації, а вибір активної моделі задається конфігураційно; це дає змогу замінювати або розширювати набір моделей без змін у клієнтській логіці та основних модулях системи.

Підхід RAG реалізовано як окремий модуль у складі серверної частини на TypeScript: перед викликом ChatGPT система відбирає релевантні фрагменти з бази знань і додає їх до промпту як контекст. База знань організована у вигляді структурованих текстових блоків (категорії, ключові слова, зміст), що забезпечує тематичне зіставлення запиту та формування короткого, але інформативного контексту. Така організація підвищує точність відповідей і дозволяє актуалізувати інформаційне наповнення шляхом редагування бази знань без зміни програмного коду.

Загальна архітектура системи узгоджується з обраним стеком і зображена на рисунку 2.2, передбачено розподіл на клієнтський та серверний рівні, де інтерфейс реалізує взаємодію користувача, а сервер виконує роль API-шлюзу та точки інтеграції з LLM. Формування відповіді здійснюється із залученням RAG-підходу – перед генерацією відбирається релевантний контекст із бази

знань і додається до запиту, після чого результат повертається клієнту у стандартизованому вигляді.

Рисунок 2.2 – Архітектура консультативно-довідкової системи

У результаті обрані технології формують цілісне середовище розробки, у якому забезпечується чіткий розподіл відповідальностей між рівнями системи. Застосування інтеграції з LLM через OpenAI API та RAG-підходу з базою знань підвищує інформативність і релевантність відповідей.

2.3 Діаграма послідовності обробки користувацького запиту

Діаграми послідовностей застосовують для моделювання динамічної поведінки системи шляхом фіксації взаємодії між її учасниками у часовій послідовності. Вони відображають порядок обміну повідомленнями, моменти активації компонентів і повернення результатів під час реалізації визначеного сценарію, що дає змогу деталізувати не лише логіку виконання операцій, а й структуру викликів, залежності між модулями та умови взаємодії клієнтської частини, серверного рівня і зовнішніх сервісів.

У межах цієї роботи сформовано діаграму послідовностей, яка відтворює повний цикл обробки користувацького запиту з урахуванням усіх ключових архітектурних компонентів системи. Використання такої діаграми забезпечує

наочне представлення узгодженості взаємодії, дає підстави для виявлення критичних точок інтеграції та дозволяє ідентифікувати потенційні джерела помилок або затримок на етапі проєктування. Розроблену діаграму зображено на рисунку 2.3.

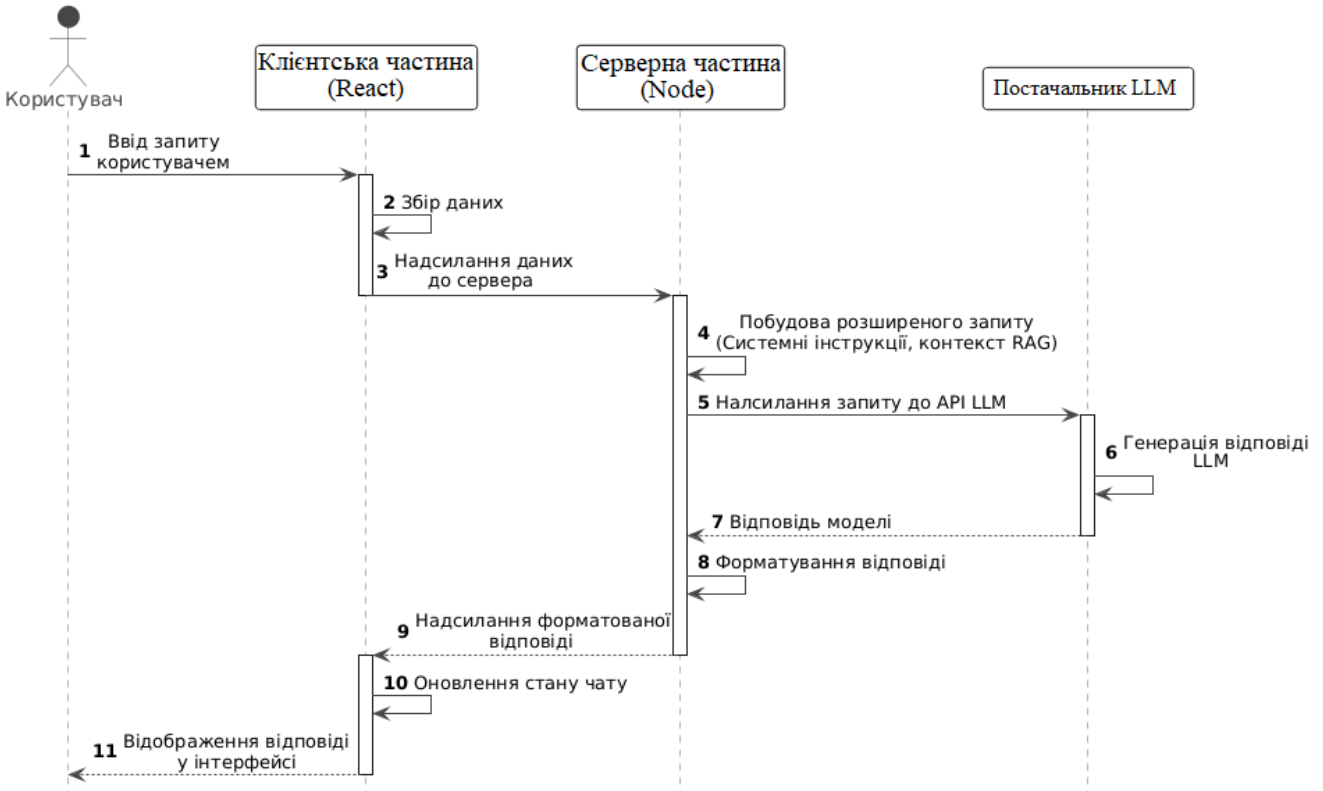

Рисунок 2.3 – Діаграма послідовності обробки користувацького запиту

У результаті було отримано діаграму послідовності обробки користувацького запиту, яка точно відтворює порядок обміну повідомлень, активації компонентів та повернення результатів поточного сценарію. Наявність діаграми послідовності в подальшому надає змогу деталізувати логіку виконання операцій та структуру викликів між модулями залежно від умов взаємодії частин системи.

3 РЕАЛІЗАЦІЯ ПРОГРАМНИХ КОМПОНЕНТІВ СИСТЕМИ

У цьому розділі наведено практичні аспекти реалізації консультативно-довідкової системи на основі великої мовної моделі. Розглянуто загальну архітектуру застосунку та взаємодію його складових. Окрему увагу приділено реалізації модулів інтеграції з LLM, підготовці та підключенню предметних даних, а також побудові конвеєра отримання релевантного контексту для підвищення точності відповідей. У межах розділу описано структуру програмних компонентів, їхні інтерфейси, принципи обміну даними, а також базові питання розгортання й перевірки працездатності прототипу.

3.1 Налаштування та підключення LLM

Провайдером LLM обрано сервіс OpenAI, що зумовлено сукупністю практичних критеріїв, серед яких: доступність і зрілість API, наявність детальної документації, стабільність роботи та прийнятне співвідношення вартості та якості згенерованих відповідей [14].

Архітектурні рішення спрямовано на мінімізацію залежності від конкретного постачальника LLM. Взаємодію з мовною моделлю інкапсульовано у спеціалізованому модулі провайдера, який реалізує уніфікований інтерфейс доступу до генерації. Такий підхід дозволяє замінювати або розширювати перелік провайдерів без модифікації клієнтської частини та без змін у ключових модулях серверної логіки, обмежуючи вплив змін конфігураційними параметрами і локалізованою реалізацією адаптера.

Підключення OpenAI здійснюється шляхом створення окремого проєкту в панелі керування сервісу, де генерується ключ доступу до API. Надалі цей ключ використовується для автентифікації під час виконання запитів до LLM з метою

отримання відповіді. Зосередження викликів LLM на сервері додатково забезпечує централізований контроль параметрів генерації та єдині правила формування запитів, що підвищує керованість і відтворюваність поведінки асистента.

Панель керування OpenAI також виконує функцію моніторингу використання ресурсів проєкту та контролю витрат. У ній надається інформація щодо доступного бюджету, кількості запитів і обсягу витрачених токенів, згрупованих по днях, а також відповідна візуалізація у вигляді стовпчастої діаграми. Застосування такого моніторингу дозволяє оперативно оцінювати інтенсивність використання LLM під час тестування, виявляти пікові навантаження та контролювати фінансові обмеження експлуатації. Інтерфейс панелі керування зображено на рисунку 3.1.

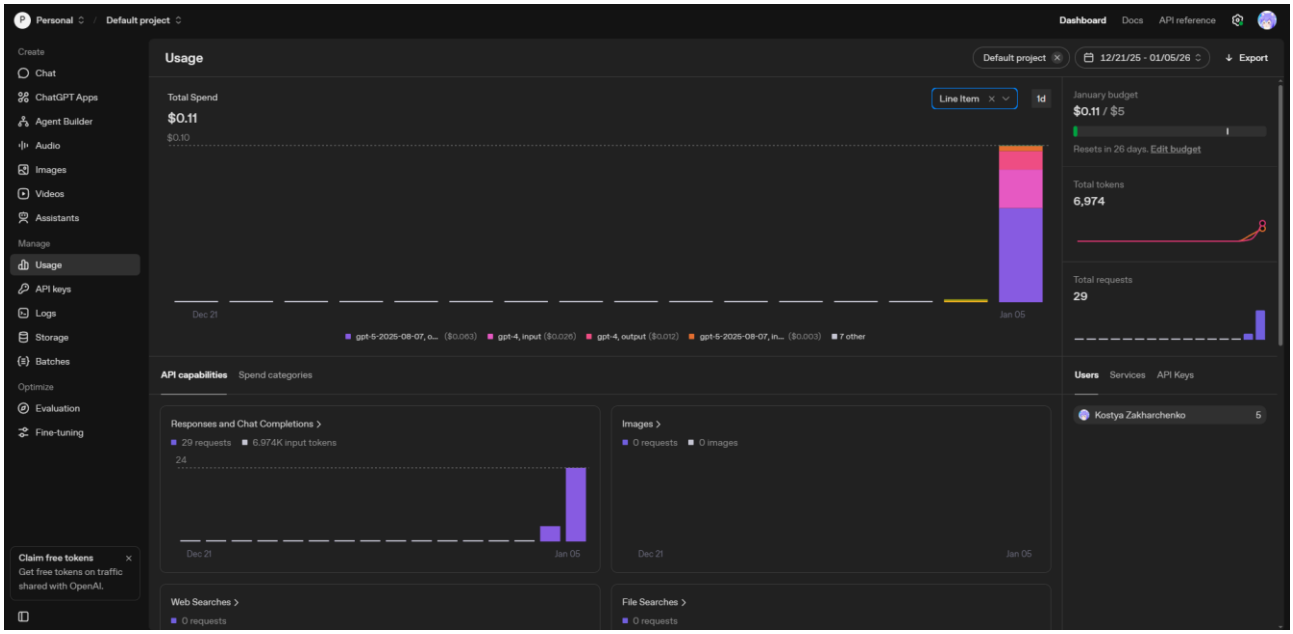

Рисунок 3.1 – Інтерфейс панелі керування проєктом OpenAI

Для генерації відповідей обрано модель gpt-4.1-mini, оскільки вона забезпечує високу швидкодію та достатню якість при помірній вартості використання. Важливою перевагою є можливість керування параметром temperature, що дозволяє налаштовувати характер відповідей відповідний довідковій системі.

3.1.1 Зберігання ключа доступу до API

З огляду на вимоги безпеки, ключ доступу необхідно зберігати на серверній стороні, оскільки передавання секретів у клієнтський застосунок створює ризик їх компрометації. У межах прототипу ключ зберігається у файлі .env, який містить перелік потрібних змінних середовища для запуску системи. Такий підхід спрощує налаштування проєкту та дозволяє відокремити конфігураційні дані від програмного коду. Файл .env не додається до репозиторію, а зберігається лише локально або у захищеному середовищі, що зменшує ймовірність витоку ключа та інших конфіденційних параметрів.

3.1.2 Побудова запиту до LLM

Побудова промпту для LLM є ключовим етапом, що визначає якість і керованість відповідей у консультативно-довідковій системі. У випадку використання RAG-підходу промпт фактично виконує роль “контракту” між системою та моделлю: він одночасно задає мету, правила поведінки, вимоги до формату відповіді й обмежує джерела інформації до наданого контексту. Чітка постановка задачі в промпті зменшує варіативність результатів, знижує ризик некоректних припущень і підвищує відтворюваність відповідей під час тестування.

У системі застосовано структуру SYSTEM + CONTEXT + USER, яка забезпечує розділення статичних правил і динамічних даних. Системний промпт (SYSTEM) визначає роль асистента, цільову аудиторію та предметну область, фіксує тон, мову відповіді та вимоги до оформлення. Особливо важливим є явне правило “джерела істини” – модель має формувати відповідь виключно на основі блоку “retrieved context”. За відсутності релевантних фрагментів передбачено стандартизовану відповідь про нестачу даних. Така умова є критичною для довідкових систем, оскільки запобігає домислюванню фактів і забезпечує узгодженість поведінки незалежно від змісту запиту. Приклад системного промпту наведено в лістингу 3.1.

Лістинг 3.1 – Системний промпт

```
export const SYSTEM_INSTRUCTION = `
You are the Official AI Assistant for National University
"Zaporizhzhia Polytechnic" (NUZP).
Your goal is to help students, applicants, and staff find accurate
information about the university.
ARCHITECTURE:
You are part of a RAG (Retrieval-Augmented Generation) system.
The system has ALREADY retrieved relevant chunks of text based on
the user's query and provided them below under "RETRIEVED
CONTEXT".
BEHAVIOR GUIDELINES:
1. **Source of Truth**: Base your answers STRICTLY on the
"RETRIEVED CONTEXT" provided below.
2. **Missing Info**: If the context is empty or does not contain
the answer, state politely in Ukrainian: "Вибачте, у мене немає
точної інформації про це в базі знань. Будь ласка, уточніть запит
або зверніться до деканату."
3. **Language**: Respond in UKRAINIAN by default.
4. **Tone**: Official, academic, helpful.
5. **Formatting**: Use Markdown lists and bold text for important
dates, URLs (make them clickable), and contacts.
6. **Identity**: You represent Zaporizhzhia Polytechnic. Refer to
"our university" or "NUZP".`;
```

Контекст формується автоматично як результат RAG. Система виконує пошук у базі знань, відбирає релевантні фрагменти та вставляє їх у промпт під окремим заголовком, що відокремлює фактичні дані від інструкцій та запиту користувача. На цьому етапі важливим є збереження семантичної відповідності між запитом і контекстом: до промпту додаються лише ті фрагменти, які безпосередньо стосуються поставленого питання та можуть бути використані як підстава для формування відповіді. Для підвищення точності доцільно обмежувати обсяг контексту, вилучати дублікати та мінімізувати шум, оскільки надлишкові або слабко пов’язані фрагменти погіршують якість відповіді, ускладнюють вибір суттєвої інформації та збільшують витрати токенів.

Користувацький запит додається як завершальний елемент промпту й задає конкретну інформаційну потребу. За необхідності запит може проходити попередню нормалізацію та доповнюватися коротким контекстом діалогу, якщо це потрібно для коректної інтерпретації питання. У результаті отримуємо послідовність композиції промпта що зображена на рисунку 3.2.

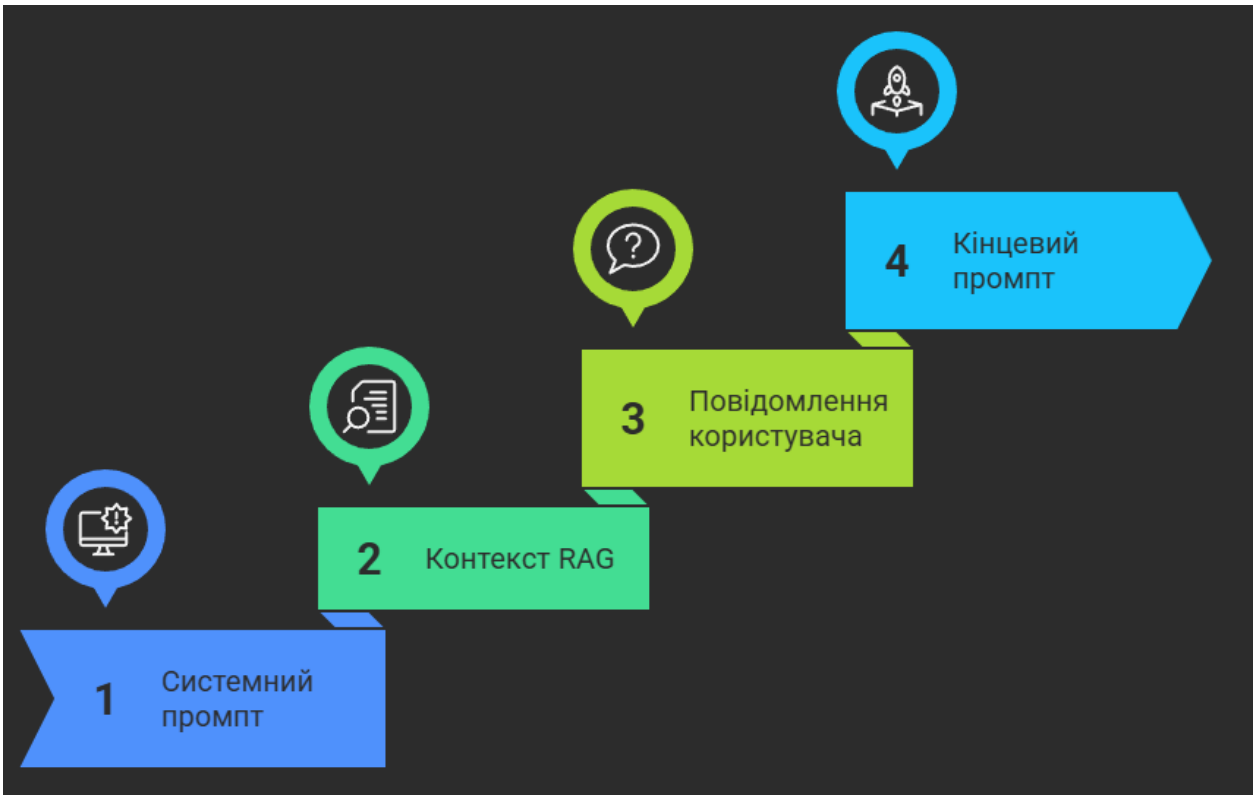

Рисунок 3.2 – Послідовність композиції промпта для LLM

Таким чином, схема, що поєднує системний промпт, контекст RAG та запит користувача забезпечує керовану генерацію: системна частина задає роль і правила, контекст надає фактичну основу відповіді, а запит визначає поточну задачу. Це дозволяє отримувати стабільні, формально коректні та перевірні відповіді в межах наявної бази знань.

3.2 Розробка клієнтської частини системи

Клієнтська частина системи реалізована із використанням React та TypeScript, що дозволяє поєднати гнучкість компонентного підходу з перевагами статичної типізації. React забезпечує роботу з віртуальним деревом DOM (Document Object Model), завдяки чому оновлення інтерфейсу відбувається без повного перезавантаження сторінки та з мінімальними

витратами на рендер. Це є критично важливим для діалогового режиму, де користувач очікує миттєвого відображення власного повідомлення, появи індикатора обробки та подальшого отримання відповіді від системи у тому ж вікні чату [15].

Архітектурно клієнтська частина організована як набір UI-компонентів, відповідальних за відображення історії діалогу, введення повідомлення та відображення станів взаємодії із сервером. Для збереження поточного стану розмови використовується стан застосунку, який включає масив повідомлень, ознаку активного запиту та текст помилки для випадків, коли відповідь не була отримана. Такий підхід спрощує підтримку узгодженості між відображенням інтерфейсу і фактичним станом обміну даними. Після відправлення повідомлення інтерфейс одразу показує новий запис користувача та сигналізує про очікування, а після завершення запиту додає відповідь моделі або повідомлення про помилку.

Ключовим елементом взаємодії клієнтської частини із сервером є функція, що виконує відправку тексту запиту та обробку відповіді. Логіка побудована таким чином, щоб забезпечити передбачувану послідовність станів і коректне оновлення історії діалогу навіть за умов асинхронності. Спочатку формується об’єкт повідомлення користувача із унікальним ідентифікатором і часовою міткою, після чого стан оновлюється через функціональну форму setState, що гарантує роботу з актуальним попереднім значенням. Далі виконується HTTP-запит до серверного маршруту /api/chat методом POST із передачею повідомлення у форматі JSON. Після успішного завершення запиту відповідь перетворюється з JSON у прикладний формат, витягується текст відповіді, створюється об’єкт повідомлення моделі та додається до історії чату. У випадку помилки, зокрема недоступності сервера або некоректного статус-коду, стан переводиться у режим завершення обробки, а користувачу відображається зрозуміле повідомлення про неможливість виконати запит. Реалізація описаного підходу наведена у лістингу 3.2.

Лістинг 3.2 – Функція відправки запиту на сервер та обробки відповіді

```
const handleSendMessage = async (text: string) => {
    const newUserMessage: Message = {
      id: Date.now().toString(),
      role: Role.USER,
      text: text,
      timestamp: new Date(),
    };
    setChatState((prev) => ({
      ...prev,
      messages: [...prev.messages, newUserMessage],
      isLoading: true,
      error: null,
    }));
    try {
      const response = await fetch('/api/chat', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ message: text }),
      });
      if (!response.ok) throw new Error('Server error');
      const data = await response.json();
      const responseText = data.reply;
      const newBotMessage: Message = {
        id: (Date.now() + 1).toString(),
        role: Role.MODEL,
        text: responseText,
        timestamp: new Date(),
      };
      setChatState((prev) => ({
        ...prev,
        messages: [...prev.messages, newBotMessage],
        isLoading: false,
      }));
    } catch (error) {
      setChatState((prev) => ({
        ...prev,
        isLoading: false,
        error: "Помилка сервера: Не вдалося обробити запит.
Спробуйте пізніше.",
      }));
    }
  };
```

Застосування функціонального оновлення стану дозволяє уникнути типових проблем, пов’язаних із конкурентними оновленнями під час швидкої послідовної відправки повідомлень, а також забезпечує незмінність структури даних, що відповідає рекомендаціям React. Додатково, введення ознаки

isLoading на рівні стану робить можливим контроль інтерфейсних сценаріїв, зокрема блокування повторної відправки під час виконання запиту або відображення індикатора очікування. Обробка помилок уніфікована і орієнтована на користувача, оскільки навіть у разі збоїв зберігається вже введена історія діалогу, а система не переходить у непрацюючий стан.

Реалізація блоку підказок у клієнтській частині покликана зробити перший контакт користувача з системою максимально простим і передбачуваним. На практиці це означає, що замість порожнього поля вводу у початковому стані інтерфейс пропонує декілька типових тем, які найчастіше виникають у студентів та абітурієнтів. Такий підхід зменшує поріг входу, оскільки користувачу не потрібно одразу формулювати запит самостійно, достатньо вибрати одну із запропонованих опцій і система автоматично підставляє готовий текст запиту в механізм надсилання. Крім цього, наявність прикладів задає правильний формат звернення до асистента і непрямо демонструє межі предметної області, на яку налаштована система.

Компонент підказок реалізований як окремий React-компонент із чітко визначеним списком аргументів через props. Клієнтська логіка навмисно розділяє відповідальність – компонент Suggestions відповідає лише за відображення списку та генерацію події вибору, тоді як фактична відправка повідомлення виконується на рівні батьківського компонента, який володіє станом чату і містить функцію надсилання. Завдяки цьому Suggestions залишається перевикористовуваним і не прив’язується до деталей мережевої взаємодії або структури стану. Механізм взаємодії побудований через callback onSelect, який приймає текст запиту і передає його у верхній рівень, де він обробляється так само, як і текст, введений вручну. Реалізацію компонента наведено у лістингу 3.3, де визначено інтерфейс властивостей, набір підказок із полями відображуваного підпису та фактичного запиту, а також розмітку, що відображає кнопки з прив’язкою до обробника натискання.

Лістинг 3.3 – Компонент з пропозиціями запитань

```
interface SuggestionsProps { onSelect: (text: string) => void }
const Suggestions: React.FC<SuggestionsProps> = ({ onSelect }) =>
{
  const suggestions = [
    { label: "🎓 Як зайти в Moodle?", query: "Як зайти в Moodle?"
},
    { label: "🗓️ Де знайти розклад?", query: "Де я можу знайти
розклад?" },
    { label: "📄 Документи для вступу", query: "Які документи
потрібні для вступу?" },
    { label: "📞 Приймальна комісія", query: "Які контакти
приймальної комісії?" }
  ];
  return (
    <div className="bg-white border border-gray-200 rounded-xl p-6
mb-6 shadow-sm text-center">
      <h2 className="text-gray-800 font-semibold text-lg mb-
2">Часті запитання</h2>
      <p className="text-gray-500 text-sm mb-6"> Виберіть тему або
напишіть своє запитання нижче: </p>
      <div className="grid grid-cols-1 sm:grid-cols-2 gap-3 text-
left">
        {suggestions.map((item, index) => (
          <button
            key={index}
            onClick={() => onSelect(item.query)}
            className="text-sm border border-gray-200
hover:border-blue-300 hover:bg-blue-50 text-gray-700 p-3 rounded-
lg transition-all"
          >
            {item.label}
          </button>
        ))}
      </div>
    </div>
  );
}
```

З погляду інтерфейсного дизайну, блок підказок є конфігурованим набором часто заданих питань, що фактично виконує роль “швидкого старту”. Оскільки масив suggestions зберігає запитання у структурованому вигляді, його можна легко адаптувати під конкретний університетський контент, додавати або прибирати теми без змін у логіці компонента. Окреме використання TypeScript-інтерфейсу для props забезпечує однозначність інтеграції –

батьківський компонент зобов’язаний передати функцію, що приймає рядок, і це зменшує ризики помилок під час розширення функціоналу.

Початковий стан інтерфейсу асистента, у якому відображається блок частих запитань, показано на рисунку 3.3. У цьому ж стані додатково виводиться коротке привітання, яке задає тон комунікації та підштовхує користувача до взаємодії, а також застереження про те, що модель може помилятися. Наявність такого повідомлення є важливою частиною UX, оскільки воно коректно формує очікування, що відповіді чат-бота доцільно сприймати як довідкову підказку і початкову точку пошуку, а не як джерело істини, особливо у випадках, коли йдеться про критично важливі дані або адміністративні процедури. Саме тому застереження розміщується у зоні, видимій ще до першої взаємодії, і працює як елемент відповідального використання ШІ.

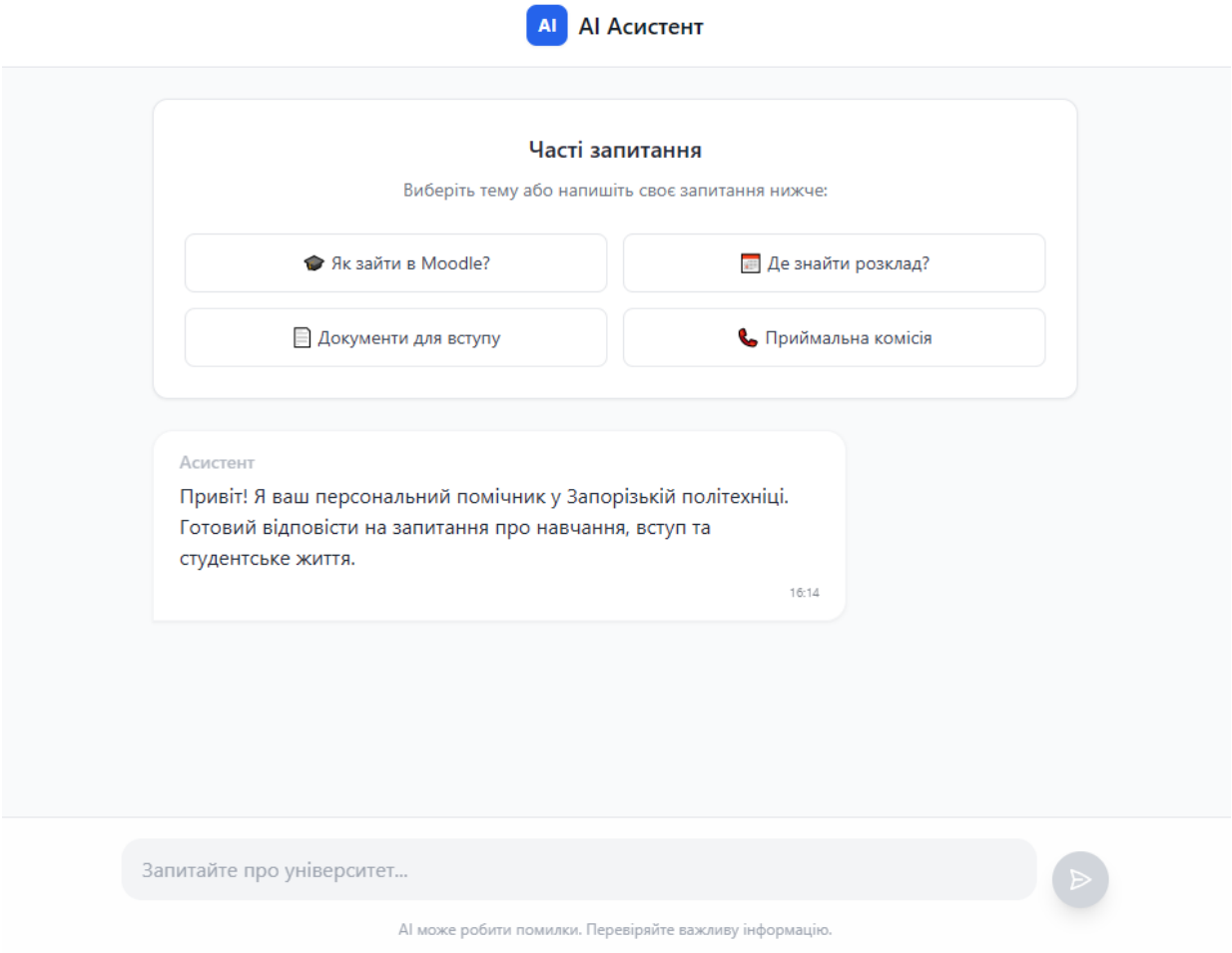

Рисунок 3.3 – Початковий стан асистента

Після отримання відповіді від сервера вона відображається у вікні чату з урахуванням форматування, оскільки текст відповіді на клієнтській частині обробляється як Markdown і рендериться у вигляді HTML-структури. Практично це реалізовано через використання бібліотеки ReactMarkdown, яка приймає текст повідомлення та перетворює його на відповідні елементи розмітки без необхідності вручну парсити посилання, виділення або переноси рядків. Завдяки такому підходу зберігається читабельність відповіді, а гіперпосилання відображаються як клікабельні, що є особливо важливим для довідкової системи, де користувачеві часто потрібно перейти на сторінку з розкладом, нормативними документами або контактами підрозділів. Додатковою перевагою Markdown-рендерингу є те, що сервер може формувати відповіді у напівструктурованому вигляді, наприклад із короткими абзацами, заголовками або вбудованими URL, а клієнтська частина гарантовано відобразить їх у зрозумілому вигляді, не порушуючи загальної стилістики інтерфейсу.

Конкретна імплементація відображення повідомлень винесена в окремий компонент MessageBubble, який відповідає за візуальне оформлення “бульбашки” повідомлення, вирівнювання відносно лівого або правого краю та рендеринг тексту з урахуванням ролі автора. У компоненті використовується перевірка message.role, яка визначає, чи належить повідомлення користувачу, і на основі цього застосовуються різні CSS-класи для фону, кольору тексту та форми заокруглення. Така диференціація робить діалог візуально зрозумілим навіть при швидкому перегляді: повідомлення користувача відображаються праворуч у контрастному стилі, а повідомлення асистента – ліворуч у нейтральному стилі з рамкою. Для підсилення сприйняття інтерфейс також містить короткий підпис ролі. Описаний механізм рендерингу повідомлень наведено у лістингу 3.4.

Лістинг 3.4 – Компонент відображення повідомлення

```
interface MessageBubbleProps { message: Message; }
const MessageBubble: React.FC<MessageBubbleProps> = ({ message }) => {
  const isUser = message.role === Role.USER;
  return (
    <div  className={`flex  w-full  ${isUser  ?  'justify-end'  : 'justify-start'} mb-4 animate-fade-in-up`}>
      <div
        className={`max-w-[85%]  sm:max-w-[75%]  rounded-2xl  px-5 py-3.5 shadow-sm ${
          isUser
            ? 'bg-blue-600 text-white rounded-br-none'
            :  'bg-white  text-gray-800  border  border-gray-100 rounded-bl-none'
        }`}
      >
        <div  className={`text-sm  font-medium  mb-1  opacity-75 ${isUser ? 'text-blue-100' : 'text-gray-400'}`}>
          {isUser ? 'Студент' : 'Асистент'}
        </div>
        <div  className={`prose  prose-sm  max-w-none  ${isUser  ? 'prose-invert' : 'prose-slate'}`}>
          <ReactMarkdown>{message.text}</ReactMarkdown>
        </div>
        <div className={`text-[10px] mt-2 text-right opacity-60`}>
          {message.timestamp.toLocaleTimeString([],  {  hour:  '2-digit', minute: '2-digit' })}
        </div>
      </div>
    </div>
  );
};
```

Використання ReactMarkdown створює основу для подальшого розширення відображення довідкової інформації, оскільки в межах цього підходу можна централізовано додати правила обробки посилань і спеціалізованих схем. Зокрема, підтримка mailto: для електронної пошти або tel: для телефонних номерів може бути реалізована через кастомізацію компонента посилань у ReactMarkdown, щоб такі значення автоматично ставали інтерактивними і коректно відкривали поштовий клієнт або телефонний набір на мобільних пристроях. За альтернативного підходу, якщо пріоритетом є швидке копіювання контактних даних, клієнтська частина може підміняти відображення певних патернів (адреса, телефон, email) на окремі інтерактивні

елементи з обробником натискання і копіюванням у буфер обміну. У результаті відповідь, що містить посилання та контактні дані, сприймається як структурована і практично придатна для подальших дій користувача, що ілюструється прикладом на рисунку 3.4.

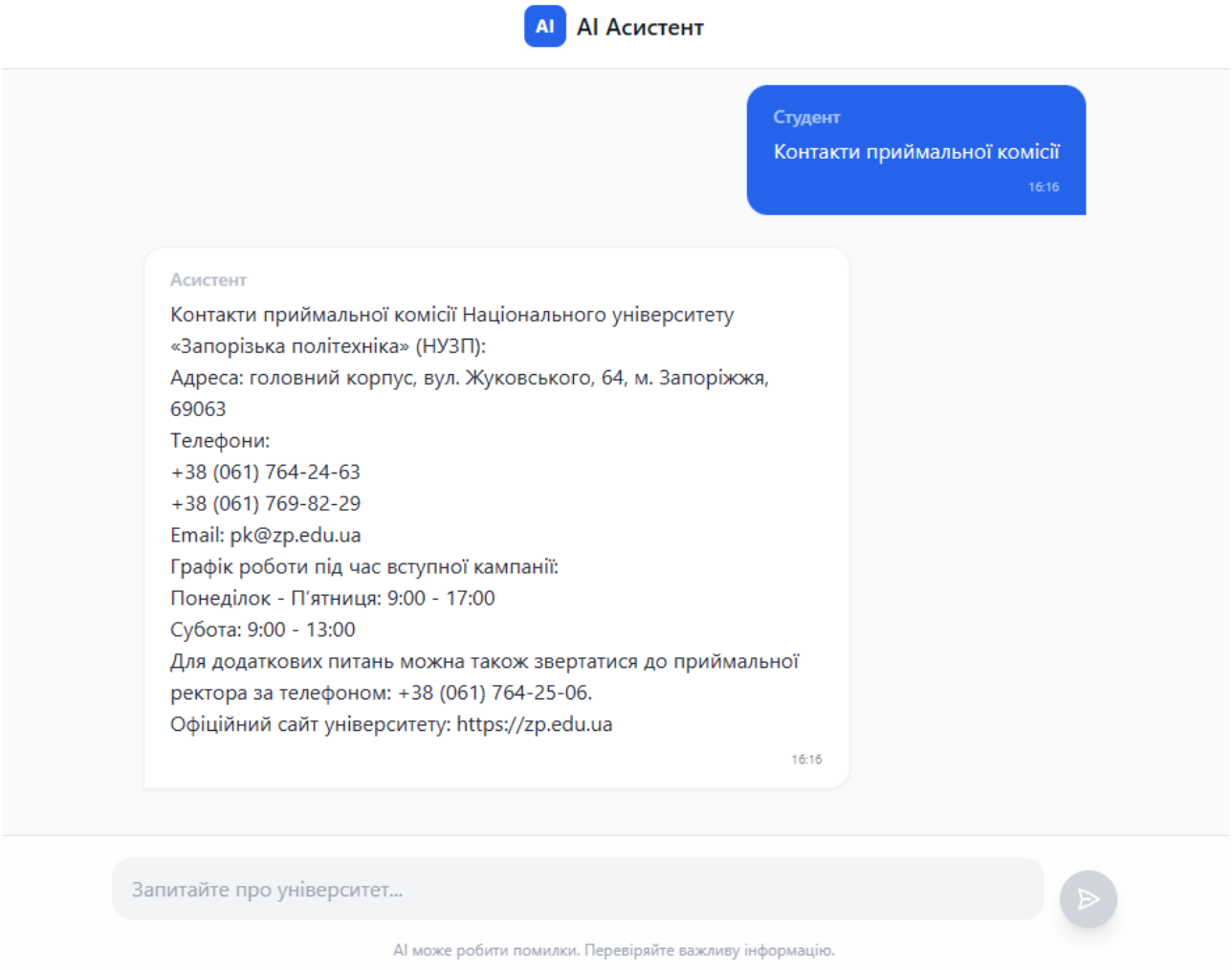

Рисунок 3.4 – Успішна відповідь асистента на запитання

У випадку, коли отриманого з RAG контексту недостатньо, аби дати однозначну відповідь, асистент повертає заздалегідь визначене повідомлення. Така стратегія є принциповою з погляду надійності системи, оскільки обмежує ризик генерації неправдивих або непідтверджених тверджень, характерних для LLM за умов дефіциту контексту. Відмова у відповіді реалізує безпечний режим роботи – система явно сигналізує про межі наявних знань і не підміняє

відсутню інформацію припущеннями. Окрім інформування про неможливість надати результат, повідомлення виконує навігаційну функцію – пропонує користувачу уточнити запит, переформулювати його або звернутися до відповідального підрозділу. Це також позитивно впливає на користувацьку довіру, адже поведінка асистента залишається передбачуваною – за наявності релевантних фрагментів система надає структуровану відповідь, а за їх відсутності коректно повідомляє про обмеження. Випадок, коли асистент не зміг відповісти на запитання, зображено на рисунку 3.5.

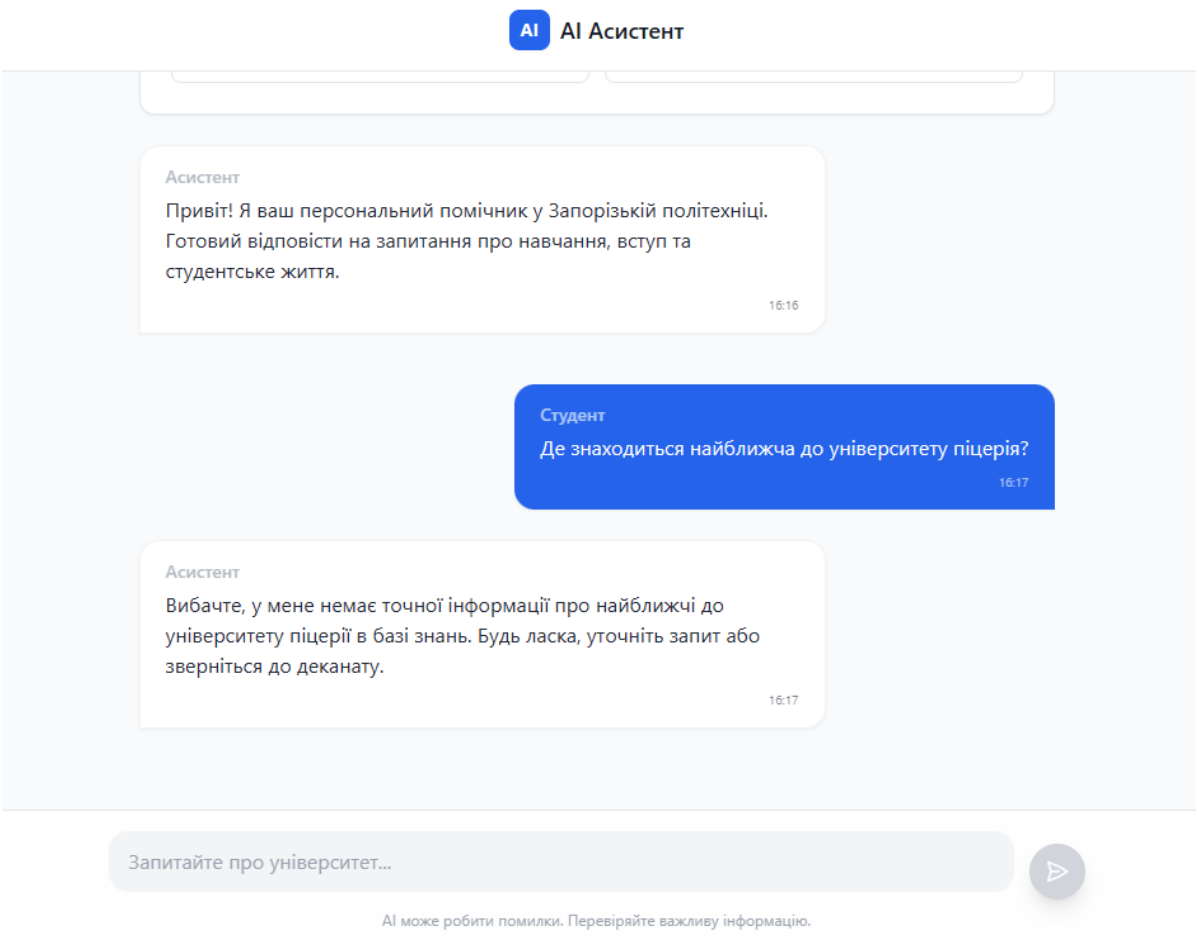

Рисунок 3.5 – Відмова у відповіді на запитання

Загалом у пункті 3.2 розроблено клієнтську частину консультативно-довідкової системи на базі React і TypeScript, що забезпечує інтерактивний діалоговий інтерфейс із передбачуваною зміною станів під час асинхронного обміну даними із сервером. Сформовано модель стану чату (історія

повідомлень, ознака обробки запиту, повідомлення про помилку) та реалізовано функцію надсилання запитів до маршруту /api/chat із коректним оновленням історії діалогу й уніфікованою обробкою збоїв. З метою підвищення зручності першої взаємодії реалізовано компонент підказок із типовими запитаннями, а також визначено правила візуального розділення ролей співрозмовників через компонент “бульбашки” повідомлень. Відображення відповідей організовано через рендеринг Markdown (ReactMarkdown), що підвищує читабельність, зберігає активні посилання й робить результати практично придатними для подальших дій користувача; у випадках нестачі контексту передбачено керовану відмову, яка підтримує надійність і узгодженість користувацького досвіду.

3.3 Розробка серверної частини системи

Серверна частина системи виконує роль проміжного шару між клієнтським вебінтерфейсом та зовнішніми сервісами генерації відповіді (LLM), а також забезпечує інтеграцію механізму Retrieval-Augmented Generation (RAG) для підвищення релевантності та фактичної узгодженості відповідей. Основними завданнями серверного застосунку є:

- прийом і валідація запитів від клієнта;
- формування контексту на основі локальної бази знань;
- вибір і ініціалізація провайдера мовної моделі;
- оркестрація запиту до LLM із параметрами генерації;
- повернення уніфікованої відповіді клієнту та обробка помилок.

3.3.1 Загальна структура та призначення серверу

Сервер реалізовано як HTTP API на базі Express, що дозволяє організувати просту й прогнозовану модель взаємодії: клієнт надсилає текстове повідомлення, сервер повертає поле reply з текстом відповіді. Такий підхід

спрощує клієнтську реалізацію, у якій достатньо одного POST-запиту, і централізує роботу з зовнішніми ключами доступу та конфігурацією моделей на серверному боці [13].

Налаштування серверу, наведене частково у лістингу 3.5 та повністю у лістингу А.2, описує стартову конфігурацію застосунку. На початковому етапі виконується завантаження змінних середовища через dotenv.config(), що дає змогу відокремити конфіденційні дані та параметри розгортання, зокрема ключі доступу, вибір активного провайдера і номер порту, від програмного коду. Після цього ініціалізується застосунок Express і підключаються проміжні обробники запитів, які забезпечують коректну взаємодію з клієнтською частиною та базові механізми захисту API. Зокрема, налаштовується CORS (Cross-Origin Resource Sharing), де перелік дозволених джерел формується зі змінної CORS_ORIGIN, а запити з невідповідних origin блокуються з поверненням коду 403 (Forbidden). Додатково перед обробкою маршрутів застосовується обмеження частоти запитів для /api/, параметри якого визначаються змінними RATE_LIMIT_WINDOW_MS і RATE_LIMIT_MAX, а у разі перевищення ліміту сервер повертає код 429 (Too Many Requests). Далі визначається маршрут POST /api/chat, який виступає основною точкою входу для передачі повідомлень у чаті та повернення сформованої відповіді, при цьому виконується базова валідація вхідних даних і повертається код 400 (Bad Request) за відсутності повідомлення. Для уніфікації обробки збоїв інтеграції з LLM використовується функція mapProviderError, яка перетворює помилки провайдера на стандартизовані HTTP-відповіді, зокрема 503 (Service Unavailable) у випадках відсутності ключів або перевантаження сервісу та 502 (Bad Gateway error) у разі мережевих збоїв. Завершальним кроком є запуск HTTP-серверу на порту, що береться зі змінної PORT або використовується значення 5000 за замовчуванням.

Лістинг 3.5 – Фрагмент налаштування серверу

```
dotenv.config();
const app = express();
const port = process.env.PORT || 5000;
app.set('trust proxy', true);
const allowedOrigins = (process.env.CORS_ORIGIN || '')
  .split(',')
  .map((v) => v.trim())
  .filter(Boolean);
app.use(
  cors({
    origin: (origin, callback) => {
      if (allowedOrigins.length === 0) return callback(null,
true);
      if (allowedOrigins.includes(origin)) return callback(null,
true);
      return callback(new Error('Not allowed by CORS'));
    },
  })
);
const rateWindowMs = Number(process.env.RATE_LIMIT_WINDOW_MS ||
60000);
const rateMax = Number(process.env.RATE_LIMIT_MAX || 60);
app.use('/api/', createRateLimiter({ windowMs: rateWindowMs, max:
rateMax }));
app.use(express.json());
app.post('/api/chat', async (req: Request, res: Response) => {
  const { message } = req.body;
  if (!message) return res.status(400).json({ error: 'Message is
required' });
  try {
    const reply = await processUserQuery(message);
    res.json({ reply });
  } catch (error: any) {
    const mapped = mapProviderError(error);
    console.error('Error:', { status: mapped.status, error:
mapped.error, details: mapped.details });
    res.status(mapped.status).json({ error: mapped.error });
  }
});
app.use((err: any, _req: Request, res: Response, _next:
NextFunction) => {
  if (String(err?.message || '').includes('Not allowed by CORS'))
return res.status(403).json({ error: 'CORS policy blocked this
request.' });
  return res.status(500).json({ error: 'Internal server error' });
});
app.listen(port, () => console.log(`Server running on port
${port}`));
```

3.3.2 Налаштування API та обробка запитів

Маршрут POST /api/chat приймає об’єкт запиту, що містить поле message. Валідація наявності цього поля виконується одразу на вході: за відсутності повідомлення сервер повертає відповідь зі статусом 400 та описом помилки. Така перевірка є мінімально необхідною, оскільки запобігає зайвим витратам на обробку порожніх або некоректних запитів та забезпечує формальний контракт взаємодії між клієнтом та сервером.

Подальша логіка обробки інкапсульована у функції processUserQuery(message), яка відповідає за повний цикл підготовки відповіді. Важливо, що виклик обгорнуто в try/catch: у разі винятку сервер фіксує помилку в логах і повертає статус 500. Такий підхід уніфікує помилки на рівні API та відокремлює внутрішні деталі реалізації від клієнта, що підвищує безпеку та стабільність системи.

З погляду відповідності інтерфейсу клієнтській частині, сервер завжди повертає структурований об’єкт { reply }, що спрощує обробку відповіді у фронтенді та дозволяє розширювати формат (наприклад, додавати джерела, впевненість, категорії) без зміни базового принципу взаємодії.

3.3.3 Конфігурація моделей та керування параметрами генерації

Для забезпечення гнучкості серверна частина підтримує декілька провайдерів і варіантів моделей, перелік яких задано як enum. Використання типізованого переліку зменшує імовірність помилок конфігурації, оскільки вибір обмежується визначеними значеннями, і водночас спрощує розширення системи, адже додавання нової моделі або провайдера зводиться до доповнення переліку та реалізації відповідного сервісу. Конфігурація бекенду описується інтерфейсом BackendConfig який наведено у лістингу 3.6, у якому фіксується активний провайдер і параметр temperature, що визначає ступінь випадковості генерації. Винесення цих параметрів у конфігурацію дозволяє централізовано змінювати поведінку системи без модифікації клієнтської частини, що є важливим для підтримки й експлуатації.

Лістинг 3.6 – Визначені моделі доступні до використання

```
export enum ModelProvider {
  GEMINI_FLASH = 'gemini-2.5-flash',
  GEMINI_PRO = 'gemini-2.5-pro',
  OPENAI_GPT5 = 'gpt-5-nano',
  OPENAI_GPT4 = 'gpt-4.1-mini',
  MOCK_MODEL = 'mock-server-response'
}
export interface BackendConfig {
  activeProvider: ModelProvider;
  temperature: number;
}
```

3.3.4 Реалізація RAG і формування контексту з бази знань

Контекстний RAG підхід реалізовано функцією retrieveContext(query), що наведено у лістингу 3.7, вона виконує добір найбільш релевантних фрагментів із локальної бази знань. База знань представлена об’єктами типу KnowledgeChunk, де фіксуються ідентифікатор, категорія, набір ключових слів і текстовий вміст. Обробка запиту починається з нормалізації до нижнього регістру, після чого формується набір токенів запиту з фільтрацією надто коротких слів для зменшення шуму. Далі для кожного фрагмента обчислюється показник доречності, який зростає у випадку збігу запиту з ключовими словами фрагмента та у випадку знаходження токенів запиту в його змісті. Отримані значення використовуються для відбору фрагментів із позитивною релевантністю, їх упорядкування за спаданням значущості та обмеження кількості до заданого максимуму.

Лістинг 3.7 – Імплементація RAG

```
interface KnowledgeChunk {
  id: string;
  category: string;
  keywords: string[];
  content: string;
}
export const retrieveContext = (query: string): string => {
  const normalizedQuery = query.toLowerCase();
  const queryTokens = normalizedQuery.split(/\s+/).filter(token =>
token.length > 2);
  const chunks: KnowledgeChunk[] = knowledgeBaseData;
  const scoredChunks = chunks.map(chunk => {
```

```
    let score = 0;
    chunk.keywords.forEach(kw          =>          score          +=
normalizedQuery.includes(kw.toLowerCase()) ? 5 : 0);
    queryTokens.forEach(token          =>          score          +=
chunk.content.toLowerCase().includes(token) ? 1 : 0);
    return { ...chunk, score };
  });
  const relevantChunks = scoredChunks
    .filter(chunk => chunk.score > 0)
    .sort((a, b) => b.score - a.score)
    .slice(0, 5);
  if (relevantChunks.length === 0) return
  return relevantChunks
    .map(chunk                    =>                    `[Category:
${chunk.category}]\n${chunk.content}`)
    .join("\n\n");
};
```

У підсумку сформований контекст збирається в один текстовий блок, при цьому категорія фрагмента зберігається в явному вигляді, що підвищує структурованість вхідних даних для мовної моделі. За відсутності релевантних фрагментів функція має повертати порожнє значення, щоб подальша логіка обробки могла коректно активувати сценарій керованої відмови або ініціювати потребу в уточненні запиту.

3.3.5 Абстракція провайдерів і фабричний підхід до ініціалізації

Для уніфікації роботи з різними постачальниками LLM застосовано фабрику провайдерів ProviderFactory, реалізацію якої наведено у лістингу 3.7. Її призначення полягає в ізоляції бізнес-логіки від деталей інтеграції конкретних LLM і в централізації правил створення сервісів доступу до моделей. У межах цього підходу вибір реалізації визначається значенням ModelProvider, а зовнішня логіка взаємодіє з провайдером через узгоджений інтерфейс AIProvider.

Лістинг 3.8 – Завод провайдерів

```
export class ProviderFactory {
  static createProvider(providerType: ModelProvider): AIProvider {
    switch (providerType) {
      case ModelProvider.GEMINI_FLASH:
      case ModelProvider.GEMINI_PRO:
        return new GeminiProviderService(providerType);
```

```
      case ModelProvider.OPENAI_GPT5:
      case ModelProvider.OPENAI_GPT4:
        return new OpenAiProvider(providerType);
      case ModelProvider.MOCK_MODEL:
        return new MockProvider();
      default:
          console.warn(`Unknown provider type: ${providerType}.
Defaulting to Gemini Flash.`);
                                          return new
GeminiProviderService(ModelProvider.GEMINI_FLASH);
    }
  }
}
```

Абстракція провайдерів у поєднанні з централізованим створенням реалізацій через ProviderFactory забезпечує стійкість системи до змін зовнішніх сервісів і знижує вартість супроводу інтеграцій, оскільки розширення переліку LLM або заміна активного постачальника зводяться до локальних змін у провайдерному шарі без впливу на основний сценарій обробки запитів.

Реалізація клієнтського та серверного рівнів у межах єдиного контракту взаємодії створює передумови для відтворюваної експлуатації системи, коли результати залежать не від випадкових налаштувань, а від контрольованих конфігураційних параметрів і узгоджених правил формування відповідей. Це підвищує практичну придатність рішення для довідкових сценаріїв, оскільки зменшує ризик непередбачуваної поведінки та спрощує подальше розширення функціоналу без порушення існуючих інтерфейсів.

Вбудовані механізми обмеження запитів, контроль джерел звернень і уніфікована обробка помилок формують базовий рівень надійності, який важливий для роботи з зовнішніми LLM сервісами у реальних мережевих умовах. У поєднанні з RAG підходом це дозволяє підтримувати баланс між якістю відповідей і керованістю системи, коли поведінка залишається передбачуваною навіть за нестачі даних або при тимчасових збоях провайдера.

Далі доцільно перевірити систему у типових сценаріях використання та оцінити її поведінку під навантаженням. Це дає змогу підтвердити стабільність роботи й зафіксувати ключові показники продуктивності.

4 ТЕСТУВАННЯ ТА АНАЛІЗ РОБОТИ СИСТЕМИ

4.1 Оцінка швидкодії

Тестування швидкодії та пропускної здатності серверного API виконувалося з використанням спеціалізованого навантажувального інструмента autocannon, який дозволяє відтворювати контрольовані серії HTTP запитів і фіксувати статистику затримок та інтенсивності обробки. Обраний підхід є доцільним для прототипу, оскільки забезпечує повторюваність експерименту і дає змогу порівнювати результати між різними конфігураціями без змін у клієнтській логіці. Для імітації реального сценарію взаємодії використовувався маршрут POST /api/chat з передаванням повідомлення у форматі JSON, а тривалість кожного тесту встановлювалась у 30 секунд, що забезпечує накопичення достатньої кількості спостережень для оцінювання середніх значень і перцентилів. Додатково встановлювався таймаут 60 секунд, що є важливим у випадку викликів до LLM, оскільки час генерації може коливатися залежно від поточного навантаження зовнішнього сервісу та мережевих умов.

Запуск проводився у двох режимах. Перший режим відповідає одиничному підключенню і відображає базову затримку в умовах послідовної взаємодії, яка є типовою для індивідуального користувача у чаті. Другий режим моделює паралельну роботу з п’ятьма одночасними підключеннями і дозволяє оцінити поведінку системи у ситуації, коли декілька користувачів звертаються до сервісу одночасно. Значення п’яти підключень обрано як практичний компроміс між реалістичністю та простотою інтерпретації, оскільки довідковий сервіс у межах університетського сайту зазвичай не функціонує як високонавантажена платформа, проте має забезпечувати коректну роботу при короткочасному зростанні активності. Для фіксації результатів використовувався JSON формат звіту з додатковим виведенням статистики

кодів відповіді, а вихідні дані зберігалися у файли bench-latency.json і bench-parallel.json для подальшої обробки та візуалізації.

4.1.1 Апаратне середовище тестування

Випробування швидкодії виконувались у локальному середовищі на апаратній платформі споживчого класу, що відповідає типовим умовам розгортання прототипу поза серверною інфраструктурою. Система була розгорнута на домашньому комп'ютері з процесором Intel Core i3-10100F та обсягом оперативної пам'яті у 16 ГБ. Слід враховувати, що отримані показники можуть відрізнятися від результатів на серверному обладнанні, яке оптимізоване під підтримку великої кількості паралельних підключень та має вищу пропускну здатність підсистем введення-виведення.

Окремо необхідно зазначити, що розмір бази знань у межах прототипу був обмеженим і становив менше 1 МБ. За таких умов модуль добору контексту не створював відчутного навантаження, тому продуктивність системи при суттєво більшому обсязі знань у межах цього тестування не оцінювались.

Мережеві умови під час експериментів були стабільними: затримка з'єднання не перевищувала 20 мс, а швидкість доступу до мережі була близькою до 1 Gbps. Це дозволяє обґрунтовано стверджувати, що внесок локальної мережевої інфраструктури в загальний час відповіді був мінімальним, а основна частка затримки формувалася на етапі обробки запиту провайдером LLM та генерації відповіді.

4.1.2 Час відповіді

Час відповіді має безпосередній вплив на сприйняття системи користувачем, оскільки діалоговий інтерфейс передбачає очікування швидкої реакції, а затримка формує відчуття надійності або навпаки створює сумніви щодо працездатності сервісу. Для коректного аналізу доцільно розглядати не лише середнє значення, але й перцентилі, оскільки вони відображають стабільність роботи та наявність поодиноких повільних відповідей. У режимі одного підключення середня затримка становила 977 мс, а для 99 відсотків запитів не перевищувала 1.386 секунд. У режимі п'яти паралельних підключень

середня затримка склала 872 мс, а 99 відсотків запитів вкладалися у 1.375 секунд. Детальніше розподіл затримок зображено на рисунку 4.1.

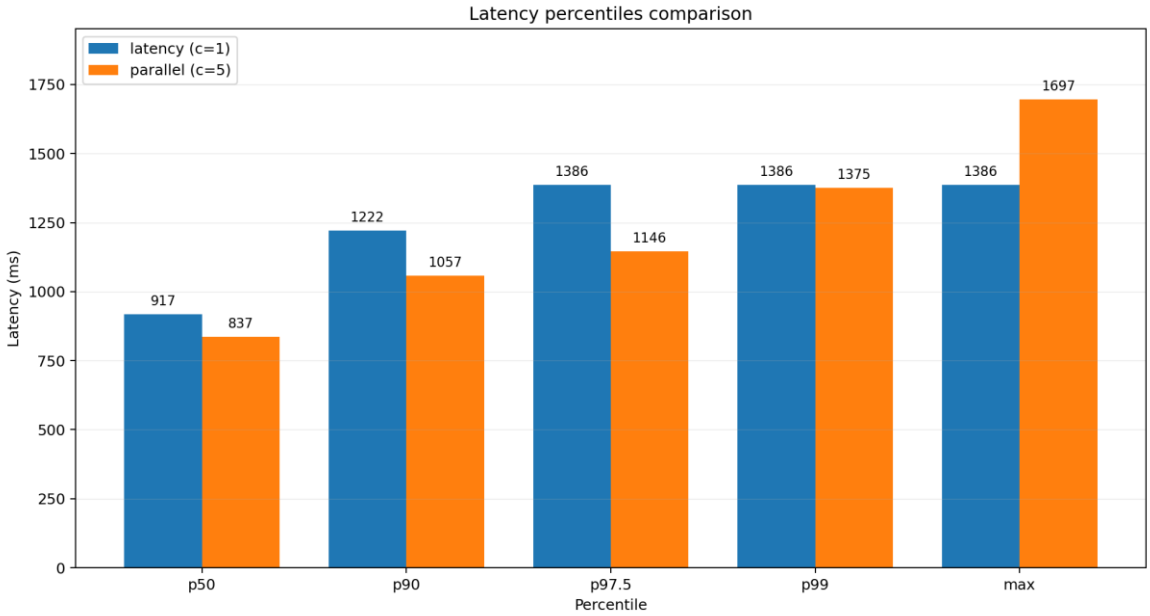

Рисунок 4.1 – Порівняння затримок запитів

Отримана картина є показовою з точки зору практичної експлуатації. По перше, у паралельному режимі не спостерігається погіршення ключових перцентилів, тобто система не демонструє тенденції до накопичення черги запитів при помірному зростанні одночасних звернень. По друге, різниця між середніми значеннями не суперечить очікуванням, оскільки при збільшенні кількості запитів вибірка стає статистично надійнішою і менше залежить від випадкових коливань часу генерації. По третє, значення 99 відсотків є близькими у двох режимах, що вказує на відсутність різких деградацій у хвостах розподілу, а це є важливим для сервісів, де навіть одиничні дуже повільні відповіді помітні користувачеві. Сукупно результати підтверджують, що система забезпечує прийнятну інтерактивність, а отриманий запас за часом відповіді є достатнім для типових довідкових сценаріїв.

4.1.3 Пропускна здатність

Показник кількості запитів на секунду відображає пропускну здатність системи та характеризує її здатність обслуговувати одночасні звернення без

переходу у деградований режим. На відміну від часу відповіді, який оцінює реакцію на окремий запит, пропускна здатність є агрегованою характеристикою, що дозволяє робити висновки щодо поведінки сервісу під навантаженням. Це має практичне значення у періоди, коли кілька користувачів одночасно запитують розклад, контакти або правила вступу, тобто коли навантаження зростає не через складність одного запиту, а через збільшення їх кількості.

У режимі одного підключення середнє значення склало 1 запит за секунду, а максимальне значення досягало 2. Такий результат є очікуваним, оскільки послідовний режим фактично відтворює взаємодію одного користувача, коли новий запит надсилається після отримання відповіді, тому інтенсивність обмежується затримкою обробки. У режимі п’яти підключень середнє значення становило 5.57 запитів за секунду, а максимальне досягало 8, що демонструє масштабування пропускної здатності при переході до паралельного сценарію. Візуалізація цього показника наведена на рисунку 4.2.

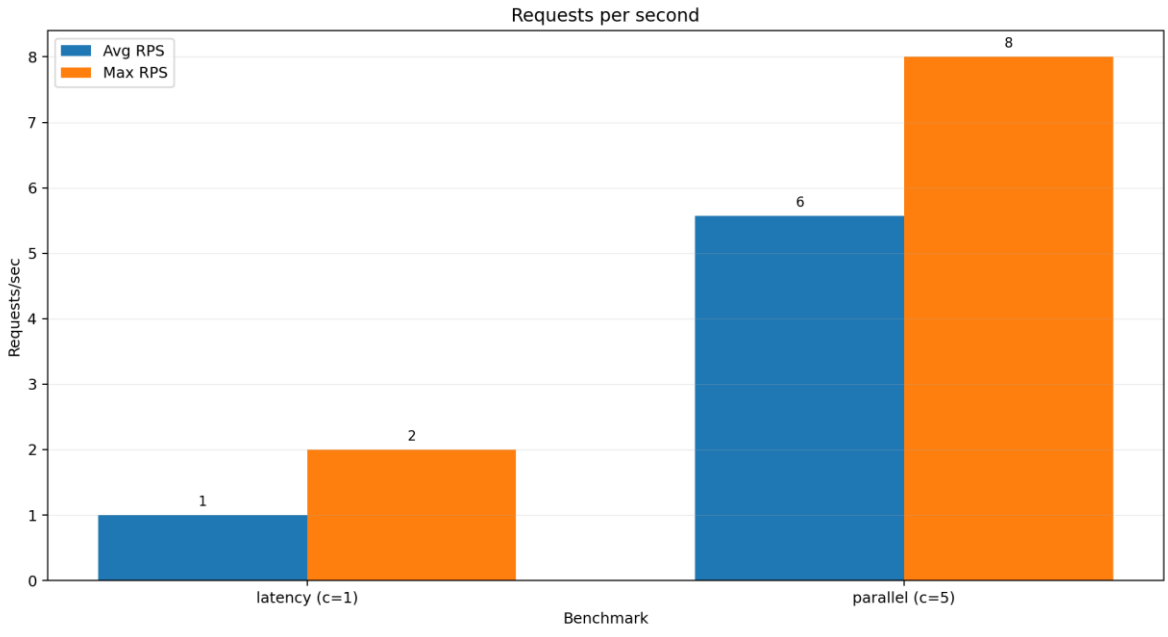

Рисунок 4.2 – Кількість запитів на секунду

Практична інтерпретація наведених значень полягає в тому, що система здатна підтримувати декілька одночасних діалогів без втрати стабільності та

без появи помилок, що узгоджується з цільовим призначенням прототипу як довідкового сервісу для вебсайту університету. Важливо також враховувати, що виміряна пропускна здатність у даному випадку визначається не лише локальною продуктивністю сервера, але й часовими характеристиками зовнішнього LLM сервісу, тому результати слід розглядати як інтегральну оцінку всієї ланки обробки. За потреби подальшого масштабування потенційними напрямами є оптимізація добору контексту, зменшення розміру промпта, кешування типових відповідей або асинхронні механізми обробки.

4.2 Тестування якості відповідей системи

Тестування відповідей на LLM присвячено оцінюванню якості текстових відповідей консультативно довідкової системи у сценаріях, що наближені до реального використання на сайті університету. На відміну від перевірки швидкодії, де ключовими є часові характеристики, у цьому підрозділі фокус зроблено на тому, наскільки коректно система інтерпретує запит, чи використовує вона наданий контекст RAG як підставу для відповіді, а також чи зберігає передбачувану поведінку у випадках, коли даних у базі знань недостатньо. Враховуючи, що кінцева взаємодія відбувається у діалоговому форматі, якість відповіді розглядається не лише як формальна правильність, але і як практична придатність для користувача, який очікує отримати стислу, зрозумілу і цільову довідку.

Методологія оцінювання побудована як експеримент на фіксованому наборі запитів, який відтворює типові наміри користувачів та характерні помилки формулювання. Для забезпечення репрезентативності тестовий набір сформовано на основі типових звернень студентів та абітурієнтів, що зазвичай стосуються розкладу, навчальних платформ, процедур вступу, контактів підрозділів, а також доступу до нормативної інформації. Окрім коректних

запитів, у датасет включено неоднозначні та некоректні формулювання, оскільки у реальних умовах користувачі часто ставлять запитання без достатнього контексту або з припущеннями, які система не може підтвердити. Додатково включено запити, що не належать до предметної області університетської довідки, щоб перевірити здатність системи зберігати межі теми та не генерувати довільні відповіді там, де очікуваною є відмова або перенаправлення.

Тестовий набір містить 30 запитів і поділений на чотири категорії, що дозволяє оцінити поведінку системи у різних класах складності. Розподіл запитів у датасеті обрано таким чином, щоб переважали проблемні випадки реальної комунікації, а саме неоднозначні й некоректні запити, при цьому зберігалася достатня частка коректних звернень для оцінки якості відповіді в основному режимі роботи. Розподіл запитань у наборі за типами наведено на рисунку 4.3.

Рисунок 4.3 – Розподіл запитань у наборі за типами

До категорії correct віднесено запитання, які відповідають предметній області та для яких у базі знань очікується наявність релевантних фрагментів, наприклад запити про те, де знайти розклад занять або як увійти до Moodle. Категорія ambiguous включає формулювання, які потенційно мають відповідь, але не задають достатньо умов для однозначного результату, наприклад коли користувач питає про дедлайн без уточнення процедури або запитує де подати документи без зазначення рівня освіти чи форми вступу. Категорія wrong охоплює запити про університет, які є некоректними з погляду можливостей системи та змісту бази знань, зокрема запити про майбутні показники, яких ще не існує у публічних даних, або вимоги до системи здійснити дію, яка виходить за межі довідкового сервісу, наприклад затвердити розклад або визначити прохідний бал на далекі роки. Категорія off_topic містить питання, що не стосуються університетської тематики, наприклад запити про побутові поради або загальні технологічні теми.

Очікувані відповіді в межах методології розглядаються як відповідність трьом основним поведінковим режимам системи. Для категорії correct очікується, що відповідь буде прямо спиратися на retrieved context і міститиме суттєву інформацію, релевантну запиту, без домислювань. Для категорії ambiguous очікуваною є стратегія уточнення, коли система або просить деталізувати запит, або надає загальну довідку з явним зазначенням, яких саме даних бракує для конкретної відповіді. Для категорій wrong та off_topic очікується керована відмова, коли система не намагається генерувати вигадану інформацію, а коректно повідомляє про відсутність даних у базі знань або про те, що запит не належить до предметної області сервісу. У всіх випадках важливою умовою є збереження узгодженого стилю відповіді та відсутність суперечностей з наданим контекстом.

У межах експериментальної перевірки працездатності консультативно довідкової системи було проведено тестування якості відповідей на підготовленому наборі запитань, що включав типові коректні звернення, неоднозначні формулювання, запити з некоректними або нереалістичними

припущеннями, а також питання поза предметною областю. Для кожного запиту фіксувалася відповідь системи та виконувалося подальше експертне оцінювання за трирівневою шкалою якості.

Категорія good означає, що відповідь є прийнятною з погляду поведінки системи в рамках довідкового сервісу. Вона або надає релевантну інформацію у випадках, коли запит має відповідь у базі знань, або коректно відмовляє чи просить уточнення, якщо даних недостатньо або запит виходить за межі можливостей системи. Категорія ok використовується для відповідей, які загалом не містять критичних помилок, однак мають помітні недоліки, наприклад надмірну узагальненість, неповну конкретизацію наступних кроків або неідеальне формулювання уточнення. Категорія bad відноситься до випадків, коли відповідь знижує практичну цінність взаємодії, наприклад через невиправдано конкретні твердження без підстав, помилкове припущення про намір користувача або відмову у ситуації, де очікується наявність відповіді в базі знань. У результаті оцінки було отримано розподіл відповідей зображений на рисунку 4.4.

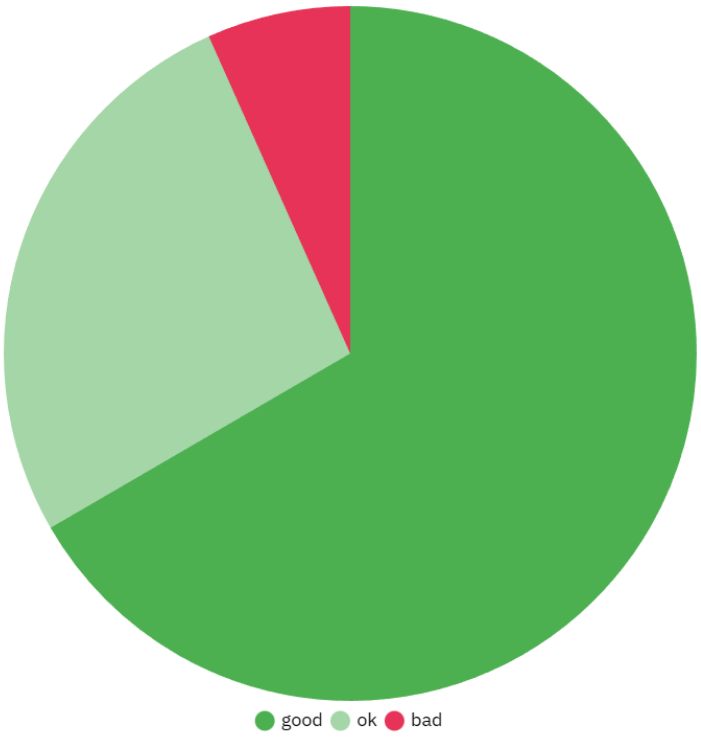

Рисунок 4.4 – Розподіл оцінок відповідей

Отримані результати свідчать, що в більшості тестових сценаріїв система демонструє керовану та узгоджену поведінку. Основна маса відповідей або містить релевантну довідкову інформацію, або коректно обробляє ситуації невизначеності шляхом відмови чи запиту на уточнення. Водночас наявність невеликої частки оцінок bad вказує на окремі випадки, де формулювання відповіді потребує посилення правил, які обмежують необґрунтовану конкретизацію та зменшують імовірність хибних припущень. На рисунку 4.5 наведено приклад відповіді з оцінкою bad яка потребує покращення системи.

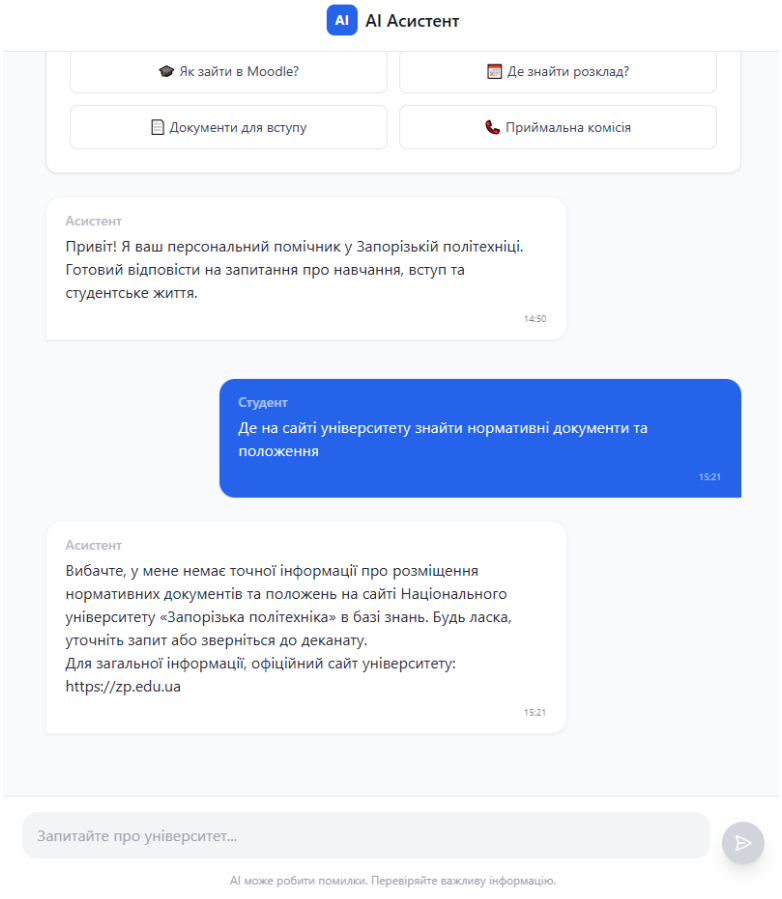

Рисунок 4.5 – Приклад поганої відповіді асистента

З практичної точки зору це означає, що для подальшого покращення якості доцільно уточнити шаблони реакцій у неоднозначних ситуаціях і розширити покриття бази знань для типових довідкових тем, щоб мінімізувати відмови там, де користувач очікує конкретну відповідь.

4.3 Рекомендації щодо використання та вдосконалення системи

Практична експлуатація консультативно-довідкової системи на базі LLM із RAG передбачає одночасну увагу до двох аспектів: коректного використання сервісу кінцевими користувачами та регулярного супроводу інформаційного наповнення і правил генерації з боку адміністратора. Результати оцінювання швидкодії підтверджують придатність системи до інтерактивного сценарію, а аналіз якості відповідей показує загалом керовану поведінку з окремими випадками необґрунтованої конкретизації. Це означає, що подальше підвищення надійності досягається насамперед організаційно-процедурними заходами, за потреби у майбутньому також можливі покращення зі швидкодії.

З погляду користувача, найбільш стабільні результати досягаються за умови, що запит містить мінімально необхідні уточнення. Для довідкових задач університетського середовища доцільно одразу зазначати контекст, який впливає на відповідь: факультет або підрозділ, рівень освіти, курс, форму навчання, а для питань щодо розкладу – групу та період. Така практика зменшує частку неоднозначних запитів і, відповідно, знижує ймовірність відповіді у форматі загальної довідки замість конкретного результату. Окремо слід враховувати, що система є довідковою, тому запити, які вимагають “затвердити”, “визначити наперед”, “надати персональні дані” або виконати адміністративну дію, мають оброблятися як некоректні: у таких випадках очікуваною є відмова або перенаправлення до відповідальних підрозділів.

Для адміністратора ключовим фактором якості є актуальність і структурованість бази знань. Доцільно організувати процес її оновлення як регулярну процедуру: визначити відповідальних за категорії, періодичність перегляду, а також правила фіксації змін. Практично виправданим є версіонування бази знань, що дозволяє відстежувати, які правки вплинули на поведінку асистента, та швидко відкотити некоректні оновлення. Оскільки у 4.2 зафіксовано поодинокі “bad” відповіді, корисною є стратегія “розширення

покриття типових тем”: якщо користувачі очікують конкретну відповідь, а система відмовляє або відповідає надто загально, це зазвичай означає нестачу релевантного контенту у знаннях або недостатню “видимість” цього контенту для механізму добору контексту.

Підвищення керованості відповідей доцільно реалізовувати через уточнення правил для двох класів ситуацій: неоднозначність і відсутність даних. Для неоднозначних запитів рекомендовано уніфікувати стратегію уточнення: асистент має не просто просити “уточніть”, а називати 1–2 конкретні параметри, яких бракує (наприклад, група/факультет/рік вступу). Для запитів поза предметною областю або з вимогами, які система не може підтвердити, доцільно посилити заборону на здогадки: відповідь має містити коротке пояснення обмеження і коректний маршрут подальших дій.

Окремий напрям покращення пов’язаний із модулем добору контексту RAG. Навіть за наявності релевантних фрагментів у базі знань, їх потрібно коректно “підняти” у retrieved context. Для цього практично корисними є: усунення дублікатів і шуму у фрагментах, обмеження максимальної кількості вставлених блоків, стандартизація структури фрагмента (назва категорії, коротке формулювання, посилання/контакти), а також контроль довжини контексту, щоб модель не “розмивала” відповідь надлишком слабко пов’язаного тексту. Якщо система еволюціонує до більшого обсягу даних, доцільним наступним кроком є перехід від переважно лексичного зіставлення до семантичного пошуку (векторного представлення), що підвищує чутливість до перефразувань і зменшує залежність від точних ключових слів.

Для підтримки відтворюваності оцінювання та керованого розвитку системи доцільно закріпити регресивне тестування відповідей як стандартну процедуру. Практично це реалізується через фіксований набір запитів, що надсилається до API та накопичення результатів із подальшим ручним експертним маркуванням. Порівняння розподілів оцінок до і після змін у промпті або базі знань дозволяє швидко виявляти деградацію якості і приймати рішення про корекцію правил. У випадках, коли “bad” відповіді спричинені

конкретними формулюваннями, ефективним є підхід локальних контрприкладів: для кожного проблемного кейсу фіксується очікувана поведінка (відмова/уточнення/посилання) і перевіряється, що після змін система стабільно відтворює потрібний результат.

З позиції експлуатації важливими є також нефункціональні заходи: журналювання запитів і помилок, моніторинг затримок і частоти відмов, контроль лімітів та вартості викликів LLM. З огляду на те, що час відповіді та пропускна здатність залежать не лише від локального сервера, а й від зовнішнього провайдера, доцільно передбачити механізми деградації сервісу: повідомлення про тимчасову недоступність, обмеження частоти запитів, а за потреби – кешування відповідей для найбільш типових звернень. У сукупності наведені рекомендації формують практичний план розвитку системи, за якого підвищення якості досягається не за рахунок “креативності” моделі, а через актуальність джерел, контрольовану поведінку в проблемних сценаріях і відтворювані процедури тестування.

Під час тестування системи на невеликих і середніх за складністю запитаннях було виконано майже 1000 звернень до провайдера LLM. Сукупна вартість цих викликів склала менше 1 долара США, що свідчить про низьку змінну собівартість обробки типового навантаження та робить підхід придатним для використання за значної кількості користувачів. Такий результат досягається за рахунок оптимального підбору моделі для довідкових сценаріїв і обмеження надлишкового контексту в промпті, що прямо зменшує витрати токенів.

Для вимірювань застосовувалась модель gpt-4.1-mini, яка забезпечує компроміс між якістю та ціною виклику. Отримані показники підтверджують, що система може економічно підтримувати високі обсяги запитів, а за потреби підвищення точності – бути переналаштована на більш потужну, але дорожчу модель без зміни загальної архітектури.

ВИСНОВКИ

У дипломній роботі було розв’язано задачу розроблення консультативно-довідкової системи підтримки студентів на базі великих мовних моделей з використанням підходу RAG. Актуальність обраної теми зумовлена зростанням обсягів інформації в освітньому середовищі та потребою у швидкому, зручному й достовірному доступі студентів до довідкових матеріалів у зручному форматі.

У ході виконання роботи проведено аналіз сучасних цифрових рішень у сфері освітніх асистентів та порівняльний аналіз підходів до побудови консультативних систем, зокрема шаблонних чатботів, систем на базі великих мовних моделей та гібридних рішень із використанням RAG. Встановлено, що застосування виключно генеративних моделей без доступу до актуальної доменної інформації є недостатнім для довідкових задач, тоді як інтеграція семантичного пошуку з LLM дозволяє суттєво підвищити точність і надійність відповідей.

У роботі розроблено архітектуру консультативно-довідкової системи з чітким розмежуванням клієнтського, серверного, інтелектуального рівнів та рівня даних. Запропонована структура забезпечує масштабованість, зручність супроводу та можливість адаптації системи до змін інформаційного наповнення без необхідності перенавчання мовної моделі. У межах практичної реалізації обґрунтовано вибір програмних засобів і технологій, що дозволили реалізувати описану модель функціонування системи.

Експериментальна перевірка підтвердила працездатність розробленої системи та коректність реалізації основних алгоритмічних рішень. Оцінка якості відповідей показала, що система здатна формувати змістовні та релевантні відповіді на типові студентські запити, зберігаючи стабільну швидкодію. Отримані результати підтверджують доцільність використання RAG підходу для задач консультативно-довідкової підтримки в освітньому середовищі.

Практична цінність роботи полягає у можливості використання розробленої системи як універсального цифрового асистента для студентів закладу вищої освіти. Система дозволяє зменшити навантаження на адміністративний і викладацький персонал, забезпечує цілодобовий доступ до довідкової інформації та створює умови для підвищення якості інформаційної підтримки студентів. Запропоноване рішення може бути адаптоване для використання в інших освітніх установах або розширене шляхом інтеграції з внутрішніми інформаційними системами університету, що визначає перспективність подальших досліджень у цьому напрямі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Тягунова М.Ю. Великі мовні моделі для домашніх систем / М.Ю. Тягунова, К.Р. Захарченко // Тиждень науки-2025. Факультет комп'ютерних наук і технологій: наук.- практ. конф., 14-18 квітня 2025 р.: тези доп. / Редкол.: Вадим ШАЛОМЄЄВ (відпов. ред.) Електрон. дані.- Запоріжжя : НУ «Запорізька політехніка», 2025. - С. 73-74 - 1 електрон. опт. диск (DVD-ROM). - назва з тит. екрана.

2. Фахівці Харківського Політеху створили інноваційний чатбот для профорієнтації абітурієнтів. Національний технічний університет «Харківський політехнічний інститут». URL: https://www.kpi.kharkov.ua/ukr/2025/04/29/fahivtsi-harkivskogo-politehu-stvoryly-innovatsijnyj-chatbot-dlya-proforiyentatsiyi-abituriyentiv/ (дата звернення: 26.11.2025).

3. Чи можуть чат-боти та AI замінити репетиторів з підготовки до НМТ?. LIGA.net. URL: https://life.liga.net/all/news/chy-mozhut-chat-boty-ta-ai-zaminyty-repetytoriv-z-pidhotovky-do-nmt (дата звернення: 30.11.2025).

4. Розблокуйте академічну ефективність з Scholar GPT, вашим асистентом на основі штучного інтелекту | CLAILA. CLAILA. URL: https://www.claila.com/uk/blog/rozblokujte-akademicnu-efektivnist-z-scholar-gpt-vasim-asistentom-na-osnovi-stucnogo-intelektu (дата звернення: 02.12.2025).

5. Сімченко С. В., Лиходєєва Г. В. та ін. Використання великих мовних моделей в освітній, науковій та дослідницькій діяльності, 2025, №1, с. 69–75 URL: https://jai.in.ua/archive/2025/2025-1-5.pdf (дата звернення: 03.12.2025).

6. Академічна спільнота і генеративний штучний інтелект. Наукова Бібліотека. URL: https://lib.udu.edu.ua/novyny/akademichna-spilnota-i-heneratyvnyy-shtuchnyy-intelekt (дата звернення: 03.12.2025).

7. Beyond Chat: how AI teaching assistants are transforming student support. THE Campus Learn, Share, Connect. URL:

https://www.timeshighereducation.com/campus/beyond-chat-how-ai-teaching-assistants-are-transforming-student-support (дата звернення: 05.12.2025).

8. What is RAG? - Retrieval-Augmented Generation AI Explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/retrieval-augmented-generation (дата звернення: 05.12.2025).

9. Bronnikov N. Як ми побудували власну RAG-систему: від проблеми до оптимального рішення. – DOU.ua URL: dou.ua/forums/topic/56395 (дата звернення: 08.12.2025).

10. Cross-Origin Resource Sharing (CORS) - HTTP | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (дата звернення: 12.12.2025).

11. Getting Started. vitejs. URL: https://vitejs.dev/guide/ (дата звернення: 12.12.2026).

12. Handbook - The TypeScript Handbook. TypeScript: JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 13.12.2025).

13. Express - Node.js web application framework. URL: https://expressjs.com/ (дата звернення: 13.12.2025).

14. OpenAI API Documentation. OpenAI. URL: https://platform.openai.com/docs (дата звернення: 13.12.2025).

15. Quick Start – React. React. URL: https://react.dev/learn (дата звернення: 14.12.2025).

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМНОГО КОДУ

Лістинг А.1 – Програмний код основного компонента клієнтської частини

```
const App: React.FC = () => {
  const [chatState, setChatState] = useState<ChatState>({
    messages: [
      {
        id: 'init-1',
        role: Role.MODEL,
        text: 'Привіт! Я ваш персональний помічник у Запорізькій політехніці. Готовий відповісти на запитання про навчання, вступ та студентське життя.',
        timestamp: new Date()
      }
    ],
    isLoading: false,
    error: null,
  });
  const messagesEndRef = useRef<HTMLDivElement>(null);
  const scrollToBottom = () => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  };
  useEffect(() => {
    scrollToBottom();
  }, [chatState.messages, chatState.isLoading]);
  const handleSendMessage = async (text: string) => {
    const newUserMessage: Message = {
      id: Date.now().toString(),
      role: Role.USER,
      text: text,
      timestamp: new Date(),
    };
    setChatState((prev) => ({
      ...prev,
      messages: [...prev.messages, newUserMessage],
      isLoading: true,
      error: null,
    }));
    try {
      const response = await fetch('/api/chat', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ message: text }),
      });
      if (!response.ok) throw new Error('Server error');
      const data = await response.json();
```

```
      const responseText = data.reply;
      const newBotMessage: Message = {
        id: (Date.now() + 1).toString(),
        role: Role.MODEL,
        text: responseText,
        timestamp: new Date(),
      };
      setChatState((prev) => ({
        ...prev,
        messages: [...prev.messages, newBotMessage],
        isLoading: false,
      }));
    } catch (error) {
      setChatState((prev) => ({
        ...prev,
        isLoading: false,
        error: "Помилка сервера: Не вдалося обробити запит. Спробуйте пізніше.",
      }));
    }
  };
  return (
    <div className="min-h-screen bg-gray-50 flex flex-col font-sans">
      <Header />
      <main className="flex-grow flex flex-col max-w-3xl w-full mx-auto p-4 sm:p-6 pb-0">

        <Suggestions onSelect={handleSendMessage} />
        <div className="flex-grow flex flex-col space-y-6 pb-4">
          {chatState.messages.map((msg) => (
            <MessageBubble key={msg.id} message={msg} />
          ))}

          {chatState.isLoading && (
             <div className="flex justify-start">
               <div className="bg-white border border-gray-100 rounded-2xl rounded-bl-none px-4 py-3 shadow-sm flex items-center space-x-2">
                 <div className="w-1.5 h-1.5 bg-gray-400 rounded-full animate-bounce" style={{ animationDelay: '0ms' }}></div>
                 <div className="w-1.5 h-1.5 bg-gray-400 rounded-full animate-bounce" style={{ animationDelay: '150ms' }}></div>
                 <div className="w-1.5 h-1.5 bg-gray-400 rounded-full animate-bounce" style={{ animationDelay: '300ms' }}></div>
               </div>
             </div>
          )}

          {chatState.error && (
            <div className="flex justify-center my-4">
              <span  className="bg-red-50  border  border-red-100 text-red-600 px-4 py-2 rounded-lg text-sm">
```

```
                {chatState.error}
              </span>
            </div>
          )}

          <div ref={messagesEndRef} />
        </div>
      </main>
      <InputArea                          onSend={handleSendMessage}
disabled={chatState.isLoading} />
    </div>
  );
};

export default App;
```

Лістинг А.2 – Повний програмний код налаштування серверу

```
dotenv.config();
const app = express();
const port = process.env.PORT || 5000;
app.set('trust proxy', true);
const allowedOrigins = (process.env.CORS_ORIGIN || '')
  .split(',')
  .map((v) => v.trim())
  .filter(Boolean);
app.use(
  cors({
    origin: (origin, callback) => {
      if (!origin) return callback(null, true);
      if  (allowedOrigins.length  ===  0)  return  callback(null,
true);
      if  (allowedOrigins.includes(origin))  return  callback(null,
true);
      return callback(new Error('Not allowed by CORS'));
    },
  })
);
const  rateWindowMs  =  Number(process.env.RATE_LIMIT_WINDOW_MS  ||
60000);
const rateMax = Number(process.env.RATE_LIMIT_MAX || 60);
app.use('/api/',  createRateLimiter({  windowMs:  rateWindowMs,  max:
rateMax }));
app.use(express.json());
app.post('/api/chat', async (req: Request, res: Response) => {
  const { message } = req.body;
  if  (!message)  return  res.status(400).json({  error:  'Message  is
required' });
  try {
    const reply = await processUserQuery(message);
    res.json({ reply });
  } catch (error: any) {
```

```
    const mapped = mapProviderError(error);
    console.error('Error:', { status: mapped.status, error:
mapped.error, details: mapped.details });
    res.status(mapped.status).json({ error: mapped.error });
  }
});
app.use((err: any, _req: Request, res: Response, _next:
NextFunction) => {
  if (String(err?.message || '').includes('Not allowed by CORS'))
return res.status(403).json({ error: 'CORS policy blocked this
request.' });
  return res.status(500).json({ error: 'Internal server error' });
});
app.listen(port, () => console.log(`Server running on port
${port}`));
function mapProviderError(error: any): { status: number; error:
string; details?: string } {
  const msg = String(error?.message || error || '');
  const code = String(error?.code || '');
  const name = String(error?.name || '');
  if (
    msg.includes('API_KEY is missing') ||
    msg.includes('OPENAI_API_KEY') ||
    msg.includes('GEMINI_API_KEY')
  ) return { status: 503, error: 'LLM provider is not configured.'
};
  if (msg.includes('429') || msg.toLowerCase().includes('rate
limit')) return { status: 503, error: 'LLM provider is overloaded.
Please retry later.' };
  const networkCodes = new Set(['ETIMEDOUT', 'ECONNRESET',
'ECONNREFUSED', 'ENOTFOUND', 'EAI_AGAIN']);
  if (networkCodes.has(code) ||
name.toLowerCase().includes('fetch') ||
msg.toLowerCase().includes('network')) return { status: 502,
error: 'LLM provider is temporarily unavailable.' };
  return { status: 500, error: 'Internal server error', details:
msg.slice(0, 200) };
}
function createRateLimiter(opts: { windowMs: number; max: number
}) {
  const { windowMs, max } = opts;
  type Entry = { count: number; resetAt: number };
  const store = new Map<string, Entry>();
  setInterval(() => {
    const now = Date.now();
    for (const [key, entry] of store) if (entry.resetAt <= now)
store.delete(key);
  }, Math.max(10000, Math.min(windowMs, 60000))).unref();
  return (req: Request, res: Response, next: NextFunction) => {
    const ip = req.ip || req.socket.remoteAddress || 'unknown';
    const now = Date.now();
    const current = store.get(ip);
    if (!current || current.resetAt <= now) {
```

```
      store.set(ip, { count: 1, resetAt: now + windowMs });
      res.setHeader('X-RateLimit-Limit', String(max));
      res.setHeader('X-RateLimit-Remaining',     String(Math.max(0,
max - 1)));
      res.setHeader('X-RateLimit-Reset',   String(Math.ceil((now   +
windowMs) / 1000)));
      return next();
    }
    current.count += 1;
    res.setHeader('X-RateLimit-Limit', String(max));
    res.setHeader('X-RateLimit-Remaining',  String(Math.max(0,  max
- current.count)));
    res.setHeader('X-RateLimit-Reset',
String(Math.ceil(current.resetAt / 1000)));
    if  (current.count  >  max)  return  res.status(429).json({  error:
'Too many requests. Please try again later.' });
    return next();
  };
};
```

ДОДАТОК Б
СЛАЙДИ ПРЕЗЕНТАЦІЇ

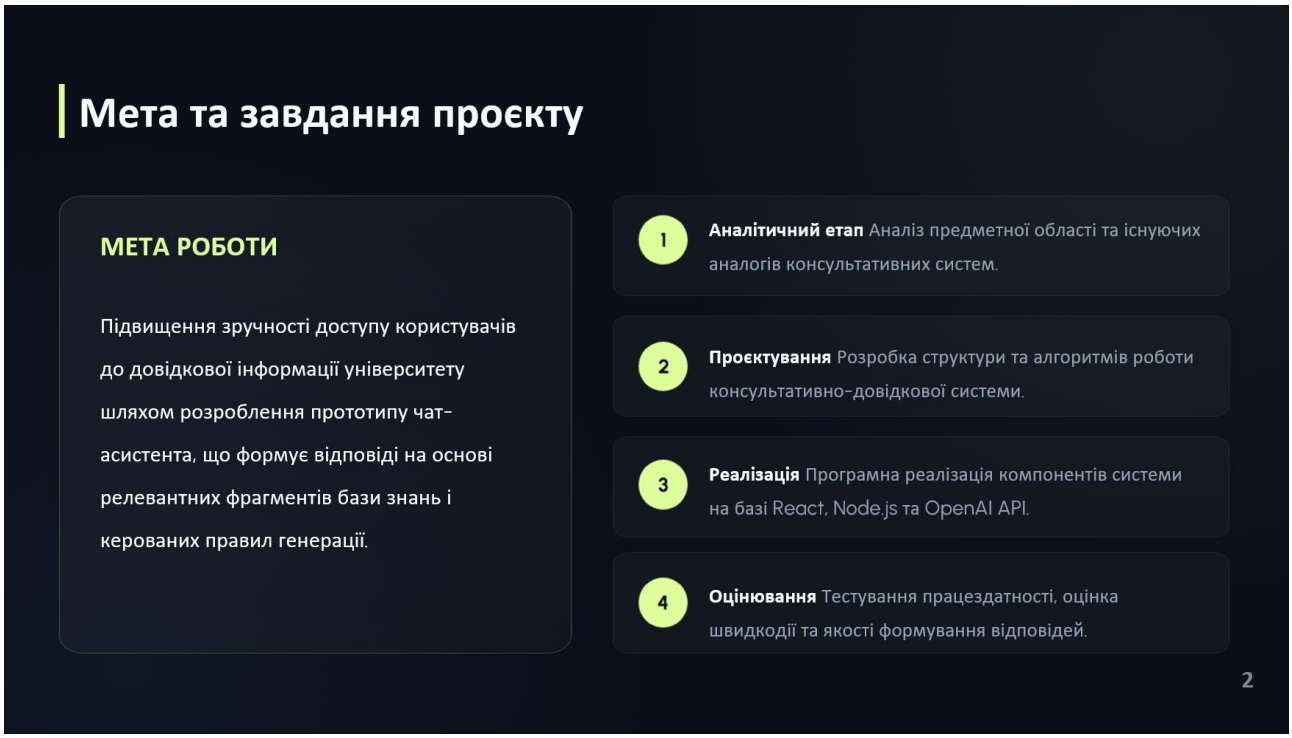

Рисунок Б.1 – Мета та завдання

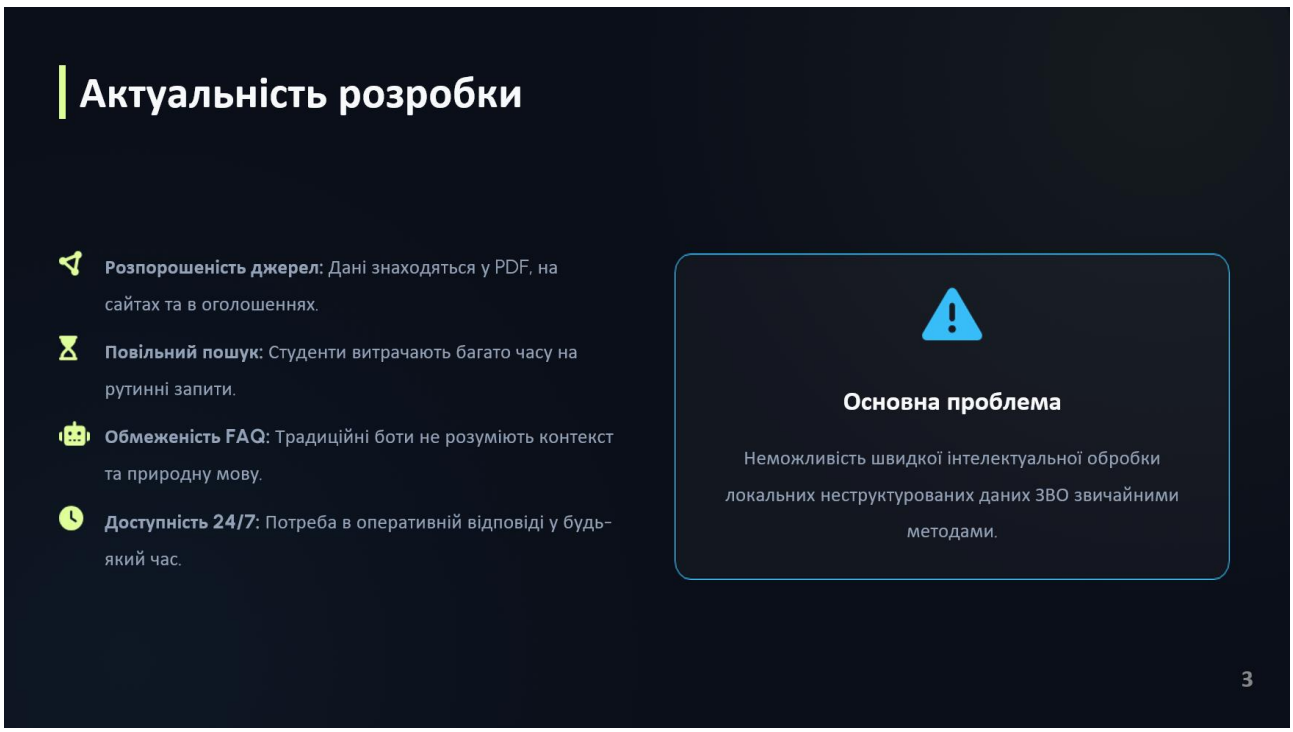

Рисунок Б.2 – Актальність розробки

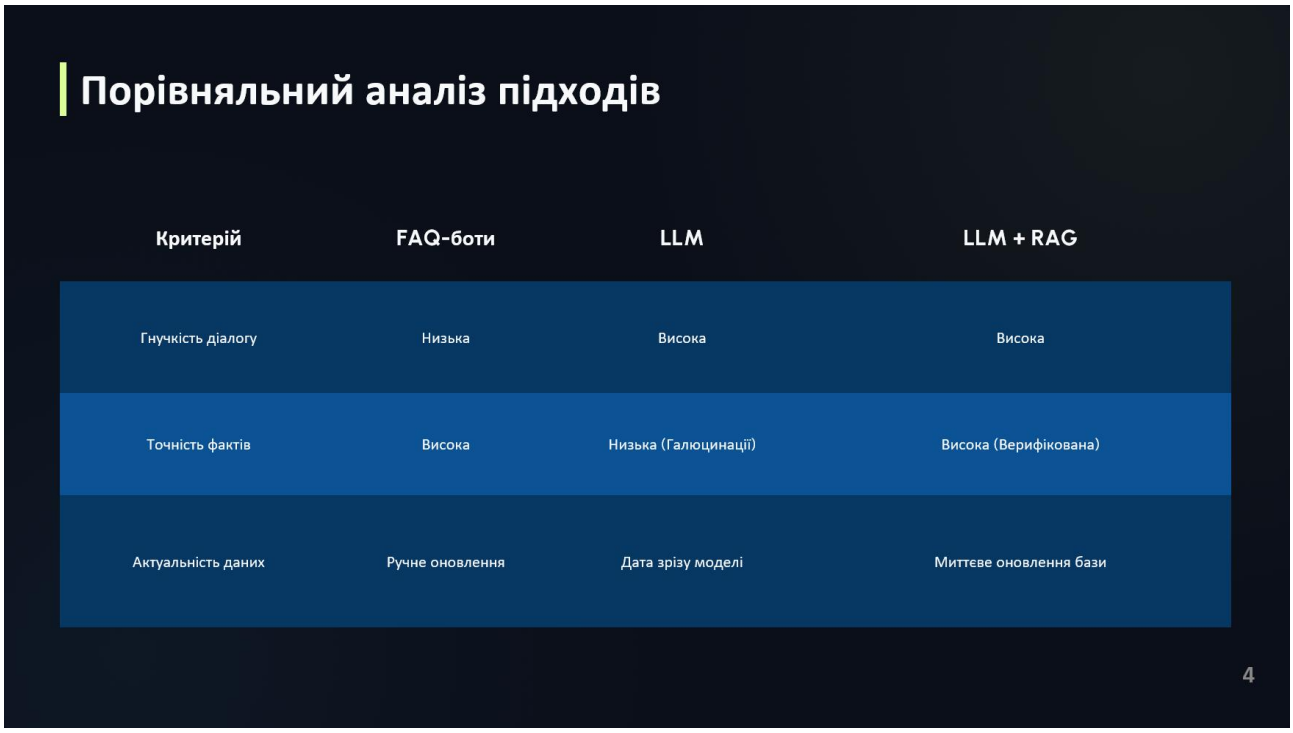

Рисунок Б.3 – Порівняльний аналіз підходів

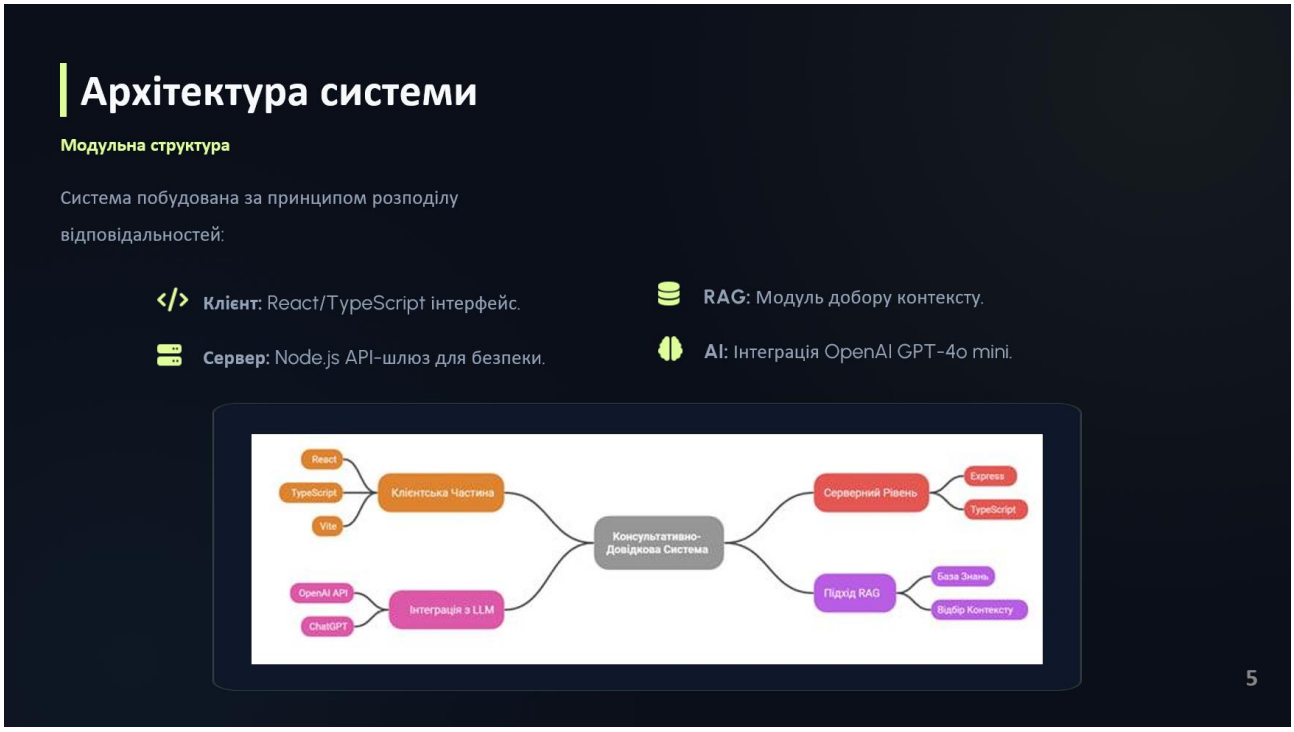

Рисунок Б.4 – Арзітектура системи

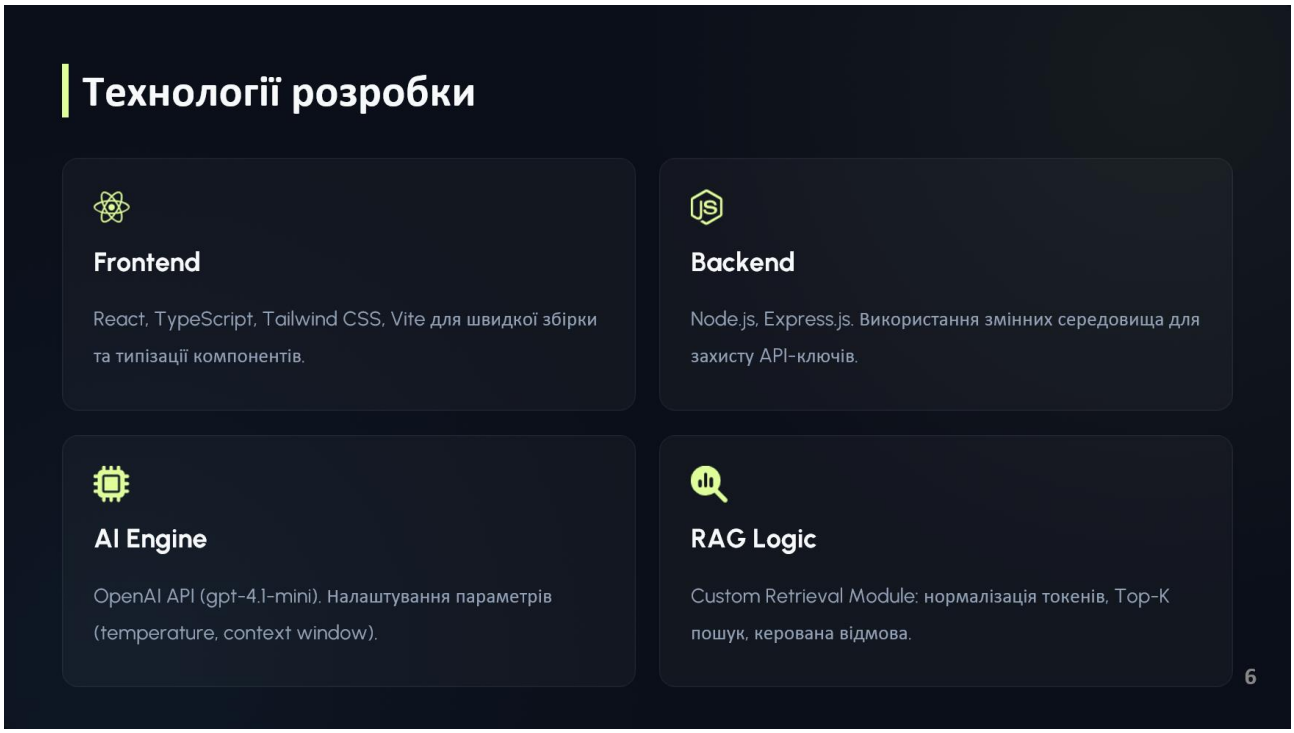

Рисунок Б.5 – Технології розробки

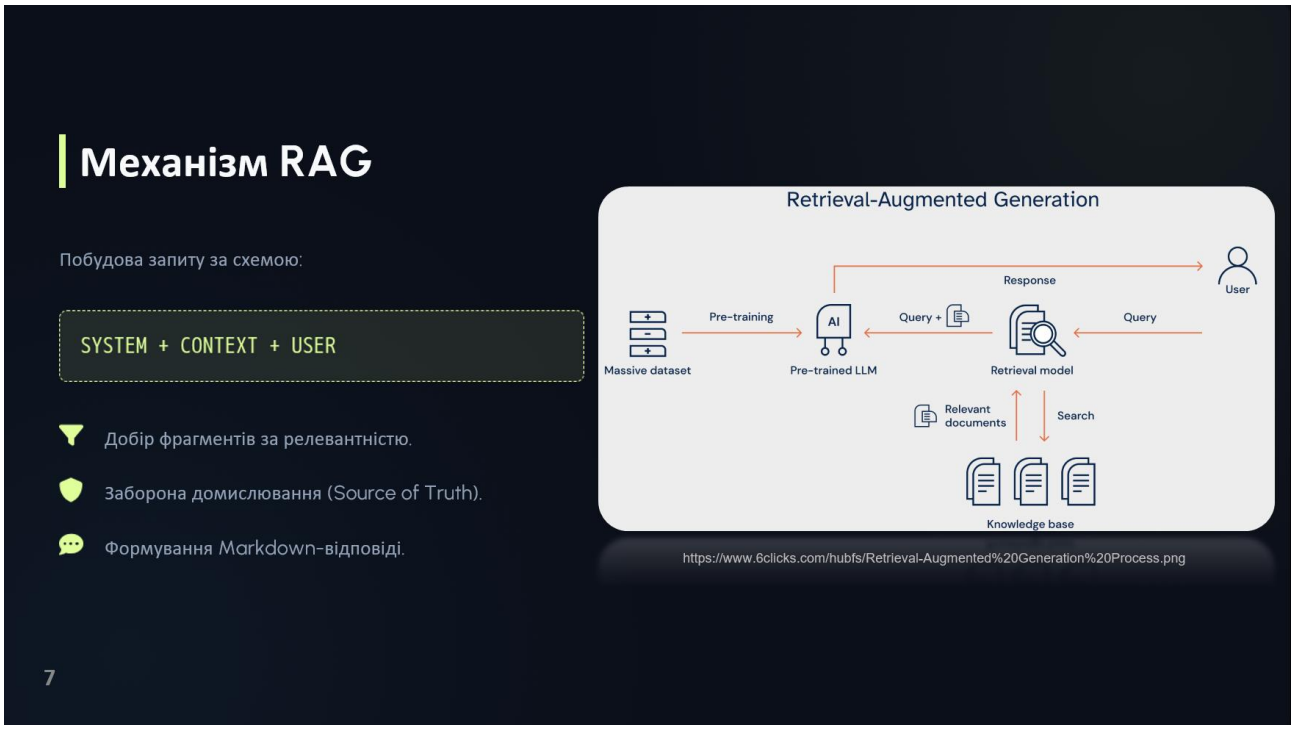

Рисунок Б.6 – Механізм RAG

Рисунок Б.7 – Результати оцінки швидкодії

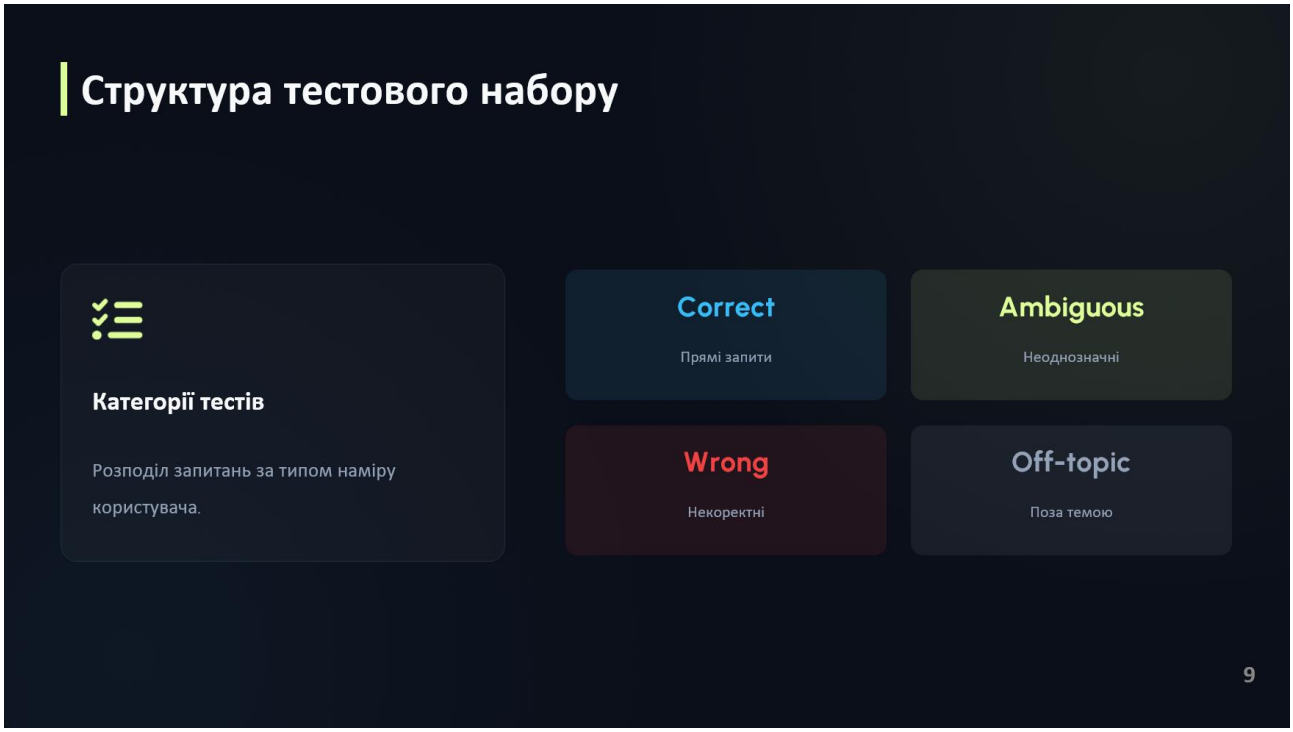

Рисунок Б.8 – Структура тестового набору

Рисунок Б.9 – Аналіз якості відповідей

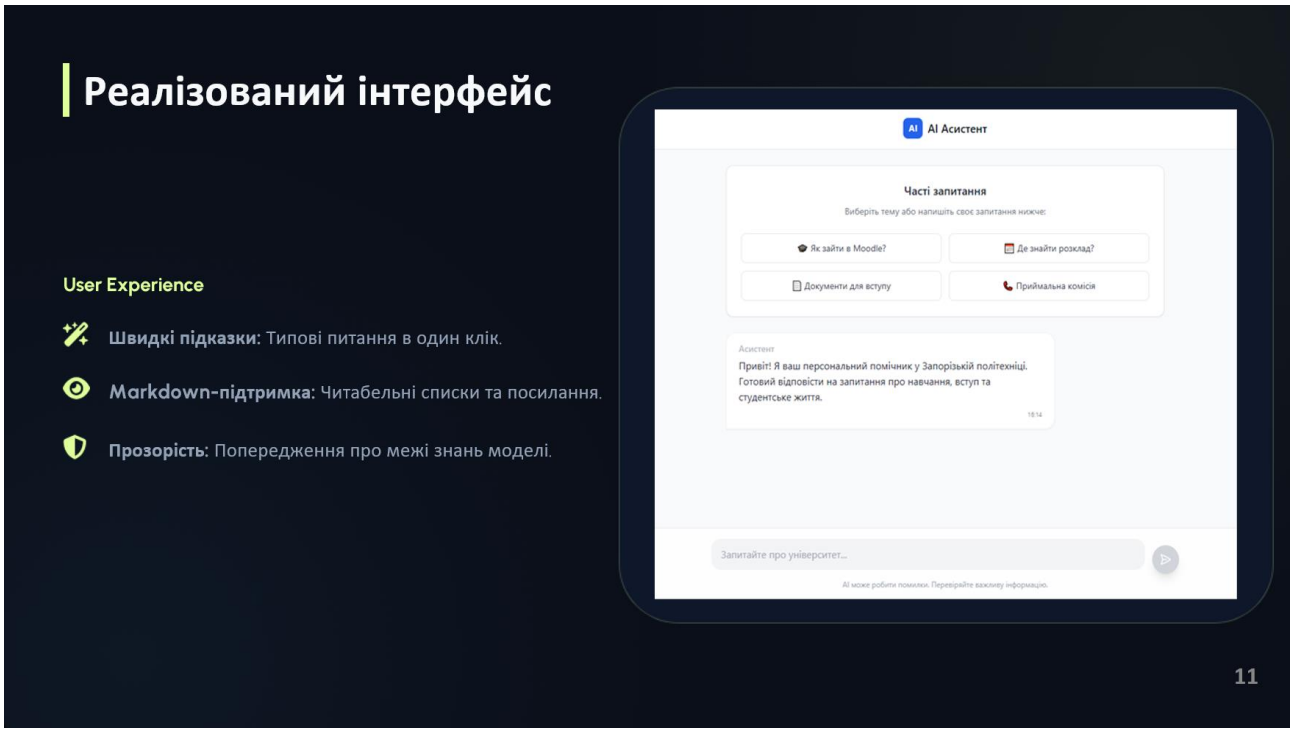

Рисунок Б.10 – Реалізований інтерфейс

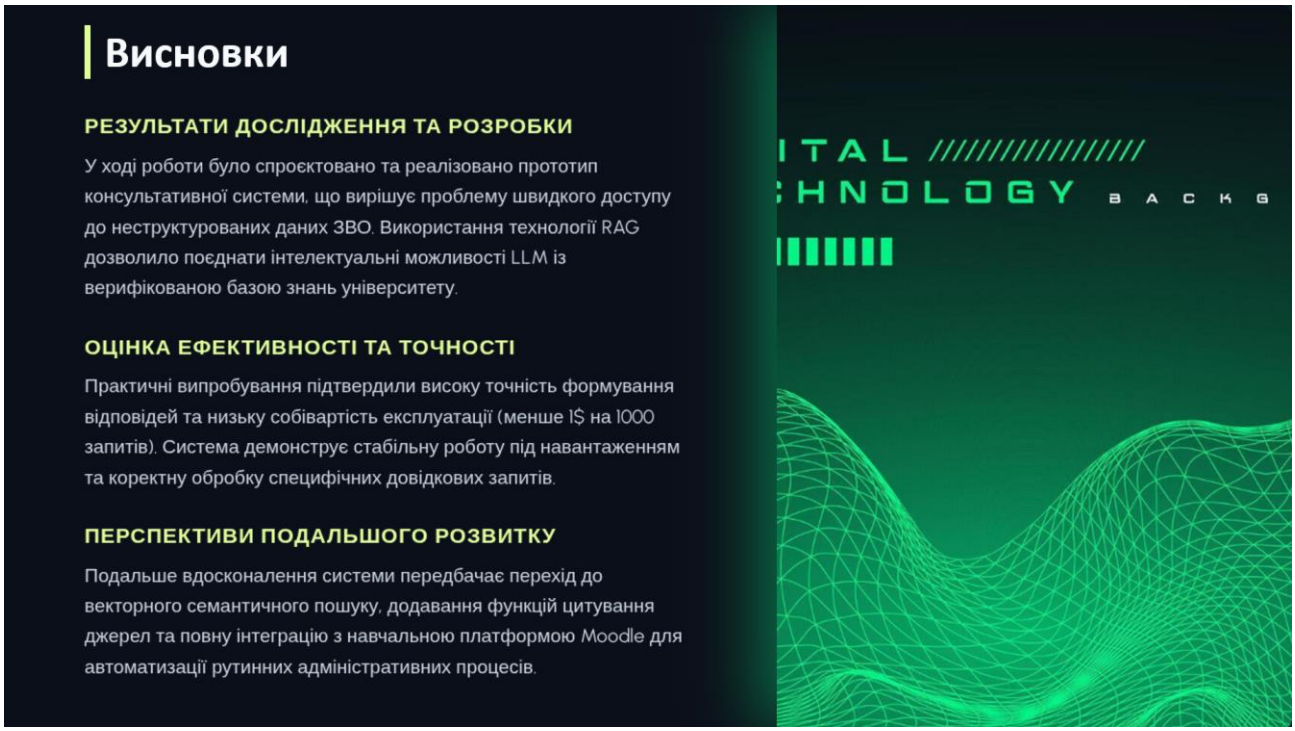

Рисунок Б.11 – Висновки