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 **ІНТЕЛЕКТУАЛЬНА АГЕНТО-ОРІЄНТОВАНА СИСТЕМА ПІДБОРУ ТОВАРІВ**

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

Освітня програма (спеціалізація)
Комп'ютерні системи та мережі

ВИНИЧЕНКО Д. В.
(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

ВИНИЧЕНКО Даніїла Віталійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Інтелектуальна агенто-орієнтована система підбору товарів
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) наявні підходи рекомендаційних систем та агентно-орієнтованих систем, сервіс OpenAI API для доступу до LLM, сучасні вебтехнології JavaScript/React та Python/FastAPI, система має забезпечувати формування персоналізованої добірки товарів та текстового пояснення рекомендацій.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області;
2. Проєктування інтелектуального агента для підбору товарів;
3. Розробка системи;
4. Результати та рекомендації щодо застосування.

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	ТЯГУНОВА 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)	1 Примітка
1	Аналіз предметної області та аналогів системи	01.10.2025 р.	
2	Проєктування інтелектуального агента для підбору	20.10.2025 р.	
	товарів		
3	Розробка системи	10.11.2025 р.	
4	Тестування системи	15.11.2025 р.	
5	Оформлення отриманих результатів у ПЗ	05.12.2025 р.	
6	Оформлення графічного матеріалу	10.12.2025 р.	
7	Оформлення допоміжного матеріалу	15.12.2025 р.	

Студент _____ **Даніїл ВИНИЧЕНКО**
(підпис) (ім'я, ПРІЗВИЩЕ)

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

РЕФЕРАТ

ПЗ: 77 с., 19 рис., 18 лістингів., 18 джерел.

FASTAPI, JAVASCRIPT, LLM, OPENAI API, PYTHON, REACT, АГЕНТО-ОРІЄНТОВАНА СИСТЕМА, ВЕБСЕРВІС, ІНТЕЛЕКТУАЛЬНА СИСТЕМА, ПРОМПТ-ІНЖИНІРИНГ, РЕКОМЕНДАЦІЙНА СИСТЕМА

Об’єкт розробки – інтелектуальна агенто-орієнтована система підбору товарів у вебсистемах електронної комерції.

Мета проєкту – підвищення якості та зручності взаємодії з користувачем вебсайту інтернет-магазину за рахунок розробки прототипу інтелектуальної агенто-орієнтованої системи підбору товарів на базі великої мовної моделі (LLM).

Перший розділ складається з аналізу предметної області. У ньому розглянуто сучасні рекомендаційні підходи, особливості агентно-орієнтованих систем, використання LLM у задачах підбору товарів та існуючі AI-асистенти.

У другому розділі розглянуто усі етапи проєктування інтелектуального агента для підбору товарів. Описано вимоги та архітектуру системи, обґрунтовано вибір технологій.

Третій розділ складається з програмної реалізації системи. Описано підключення до віддаленої LLM через OpenAI API, розробку клієнтської частини на JavaScript/React та серверної частини на Python/FastAPI, розгортання та тестування основних сценаріїв роботи системи.

У четвертому розділі наведено узагальнені результати розробки та експериментів, а також сформульовано рекомендації щодо практичного застосування системи в вебсистемах електронної комерції.

Практична цінність роботи полягає у можливості застосування розробленого прототипу в реальних вебсистемах електронної комерції для підвищення якості сервісу.

ABSTRACT

Explanatory note to the master’s work: 77 p., 19 figures, 18 listings, 18 sources.

AGENT-ORIENTED SYSTEM, FASTAPI, INTELLIGENT SYSTEM, JAVASCRIPT, LLM, OPENAI API, PROMPT ENGINEERING, PYTHON, REACT, RECOMMENDATION SYSTEM, WEB SERVICE

The object of development is an intelligent agent-oriented system for selecting goods in e-commerce web systems.

The goal of the project is to improve the quality and convenience of interaction with the user of an online store website by developing a prototype of an intelligent agent-oriented product selection system based on a large language model (LLM).

The first section consists of an analysis of the subject area. It examines modern recommendation approaches, the features of agent-oriented systems, the use of LLM in product selection tasks, and existing AI assistants.

The second section examines all stages of designing an intelligent agent for product selection. The requirements and architecture of the system are described, and the choice of technologies is justified.

The third section consists of the software implementation of the system. It describes the connection to a remote LLM via the OpenAI API, the development of the client part in JavaScript/React and the server part in Python/FastAPI, the deployment and testing of the main scenarios of the system.

The fourth section presents the generalized results of development and experiments, as well as recommendations for the practical application of the system in e-commerce web systems.

The practical value of the work lies in the possibility of applying the developed prototype in real e-commerce web systems to improve service quality.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AI	–	Artificial Intelligence, Штучний інтелект
API	–	Application Programming Interface, Інтерфейс прикладного програмування
CSS	–	Cascading Style Sheets, Каскадні таблиці стилів
DEMO	–	Демонстраційний режим роботи системи
HTML	–	HyperText Markup Language, Мова гіпертекстової розмітки
HTTP	–	HyperText Transfer Protocol, Протокол передавання гіпертексту
JSON	–	JavaScript Object Notation, Текстовий формат обміну даними
LLM	–	Large Language Models, Великі мовні моделі
RAG	–	Retrieval-Augmented Generation, Пошуково-посилене генерування

ВСТУП

У сучасному інтернет-магазині клієнти очікують швидкого та персоналізованого підбору товарів відповідно до своїх потреб та уподобань. Традиційні механізми фільтрації й сортування товарів за фіксованими атрибутами часто не враховують контекст запиту, стиль спілкування або приховані критерії вибору. Це знижує зручність користування вебсистемою та негативно впливає на конверсію. Розвиток технологій штучного інтелекту, зокрема великих мовних моделей (LLM) та сучасних рекомендаційних алгоритмів, відкриває можливість побудови інтелектуальних асистентів, які ведуть природну діалогову взаємодію з користувачем, уточнюють запит і формують релевантні добірки товарів.

Використання агенто-орієнтованих підходів дає змогу розділити складне завдання підбору на низку спеціалізованих ролей: агент збору контексту, агент роботи з каталогом, агент діалогу, агент оцінювання якості рекомендацій тощо. Інтеграція таких агентів з LLM та Retrieval-Augmented Generation-підходи (RAG) дозволяє поєднати гнучкість генеративних моделей із точністю семантичного пошуку у товарному каталозі. У поєднанні з вебтехнологіями це створює основу для інтелектуальної системи підбору товарів, яку можна вбудувати у вже існуючі рішення електронної комерції.

Об’єктом дослідження є процеси автоматизованого підбору товарів у вебсистемах електронної комерції.

Предметом дослідження є методи, алгоритми та програмні засоби, що забезпечують роботу інтелектуальної агенто-орієнтованої системи підбору товарів на основі LLM, рекомендаційних підходів та вебсервісів.

Мета проєкту – підвищення якості та зручності взаємодії з користувачем вебсайту інтернет-магазину за рахунок розробки прототипу інтелектуальної агенто-орієнтованої системи підбору товарів на базі великої мовної моделі (LLM).

Для досягнення поставленої мети в роботі необхідно вирішити такі завдання:

- дослідити особливості агентно-орієнтованих систем та проаналізувати сучасні підходи до використання великих мовних моделей в задачах підбору товарів;

- спроєктувати архітектуру інтелектуальної системи підбору товарів;

- розробити вебсистему з AI-асистентом підбору товарів;

- протестувати реалізовану систему та надати рекомендації щодо застосування.

Наукова новизна роботи полягає в обґрунтуванні комплексного підходу до побудови системи підбору товарів, що поєднує агенто-орієнтовану організацію компонентів, LLM з пошуково-посиленим генеруванням RAG та сучасні рекомендаційні алгоритми у контексті електронної комерції. Практична цінність полягає у створенні програмного прототипу AI-асистента підбору товарів, який може бути інтегрований у вебсайт інтернет-магазину та використаний як основа для впровадження персоналізованих сервісів.

Структура роботи складається зі вступу, чотирьох основних розділів, висновків, переліку посилань та додатків. У першому розділі наведено аналіз предметної області, розглянуто особливості агентно-орієнтованих систем, сучасний стан використання LLM у задачах підбору та рекомендацій, а також проаналізовано наявні аналоги AI-асистентів для шопінгу й суміжних сервісів. У другому розділі описано архітектуру запропонованої системи, сформовано вимоги, визначено компоненти та їх взаємодію. У третьому розділі подано програмну реалізацію системи – описано підключення до віддаленої LLM через OpenAI API, розробку серверної частини, клієнтської частини та проведено тестування реалізованого функціоналу. Завершальна частина роботи складається з висновків та рекомендацій щодо застосування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз рекомендаційних систем

Рекомендаційні системи аналізують поведінку користувачів і пропонують персоналізовані товари або контент. Найпоширеніші підходи – це системи колаборативної фільтрації, контент-орієнтовані та гібридні. У моделях колаборативної фільтрації, рекомендації будуються на основі схожості між користувачами та товарами. Такий підхід використовує інформацію про вподобання багатьох користувачів, що дозволяє отримувати неочікувані рекомендації – тобто пропонувати неочікувані, але релевантні товари на основі смаків схожих користувачів. Модель колаборативної фільтрації зазвичай навчається за допомогою зведених матриць подібності, які автоматично вивчають ембеддінги для користувачів та предметів. Проте такі системи мають недоліки, а саме проблеми при нових користувачах чи товари, для яких немає історії вподобань, та розрідженість даних, що може погіршувати рекомендації [2].

З іншого боку, контент-орієнтовані системи генерують рекомендації на основі характеристик самих товарів або поведінки конкретного користувача. Вони аналізують властивості об’єктів, такі як категорії, ключові слова, атрибути та знаходять ті предмети, що подібні до вже вподобаних користувачем. Наприклад, система може представити кожен товар у вигляді вектору ознак, а потім за допомогою метрик подібності обирати найбільш релевантні товари до профілю користувача. До переваг контент-орієнтованого підходу належить незалежність від інших користувачів (можна рекомендувати товари навіть новим користувачам, маючи тільки їхній список бажань) і прозорість моделі. Водночас такий підхід часто пропонує користувачу товари, схожі на вже відомі йому, тому менш схильний до несподіваних рекомендацій і потребує добре охарактеризованих товарів [3].

Гібридні системи поєднують обидва підходи, прагнучі комбінувати їхні переваги. Наприклад, вони можуть використовувати контент-орієнтовані для

рекомендацій на початку, а згодом переходити на колаборативну фільтрацію, коли накопичується достатньо даних. Гібридні моделі усувають частину недоліків простих підходів, а саме використовують як історію користувачів, так і семантичну інформацію про товари. Згідно з оглядом, такий комбінований підхід часто забезпечує вищу якість персоналізації, особливо в умовах невеликої кількості даних або неоднорідних товарних каталогів.

Метод пошуково-підсиленого генерування рекомендацій є новим підходом, що поєднує семантичний пошук у векторизованому індексі з можливостями великих мовних моделей. Спочатку система відбирає релевантні документи або описи товарів за допомогою ембеддінгів, а потім передає ці фрагменти в LLM для формування рекомендацій та пояснень. Такий підхід зменшує кількість помилкових або вигаданих відповідей, забезпечує актуальність даних і дозволяє отримувати природномовні обґрунтування вибору. Завдяки поєднанню точності пошуку та гнучкості налаштувань генеративної моделі RAG вважається перспективним рішенням для систем рекомендацій, особливо в умовах великого та різнорідного контенту, де класичні методи втрачають ефективність.

1.2 Особливості агентно-орієнтованих систем

Агентно-орієнтовані системи побудовані з автономних агентів – незалежних програмних компонентів, які самостійно приймають рішення і взаємодіють з іншими агентами чи середовищем. У мультиагентних системах (MAS) кожен агент має обмежений локальний огляд і цілі, а загальна поведінка системи виникає з координації їх діяльності. Властивості агентів включають автономність, локальний огляд і відсутність єдиного контролера. Для досягнення узгоджених результатів у MAS використовуються різні моделі взаємодії та координації. Один із підходів – централізована оркестрація, коли спеціальний агент-оркестратор керує роботою інших, забезпечуючи синхронізацію та розподіл завдань. За такого

сценарію кожен підлеглий агент відповідає за окремий аспект (агенти-пошуку, агенти-аналітики, агенти-інтерфейсу), а оркестратор активує потрібного агента в потрібний час. Існують також децентралізовані підходи, де агенти безпосередньо обмінюються повідомленнями або користуються загальними середовищами для взаємодії. Отже, особливістю агентно-орієнтованих систем є здатність модульно розподіляти складні задачі між взаємодіючими автономними одиницями, гарантуючи гнучкість, масштабованість і відмовостійкість системи [4].

1.3 Використання LLM у задачах підбору та рекомендацій

Наразі великі мовні моделі відкрили нові можливості для рекомендаційних систем та персоналізованих асистентів. Одним із перспективних підходів є Retrieval-Augmented Generation – пошуково-посилене генерування, коли до запиту LLM додають релевантні факти з зовнішньої бази знань або каталогу товарів. Це дозволяє моделям вирішувати відсутності вбудованих знань про специфічний каталог і зменшує кількість хибних фактів у відповідях. Наприклад, у моделі KERAG_R для рекомендацій використано граф знань та механізм вибірки трійних інструкцій, що забезпечує додатковий контекст при обчисленні уподобань користувача. Іншим прикладом є академічна система рекомендації курсів – LLM перетворює вільний текстовий запит студента у ніби ідеальний опис курсу, який потім використовується як векторний ключ для пошуку схожих курсів, після чого GPT-4 формує фінальний рейтинг рекомендацій із поясненнями. Такий двоступеневий підхід (генерація проміжного запиту, ембеддінговий-пошук та фінальна генерація) демонструє ефективність LLM для складних завдань підбору рекомендацій [5,6].

Ембеддінги (векторні представлення) відіграють ключову роль у сучасних системах – товари, запити і профілі користувачів перекладаються у спільний векторний простір. Наприклад, у мультиагентних рішеннях для рекомендацій

часто передбачено збереження характеристик товарів у векторній базі для швидкого семантичного пошуку. Після знаходження релевантних елементів ці векторні уявлення разом з LLM використовують для формування персоналізованих відповідей.

Особливу увагу в задачах підбору приділяють промпт-інжинірингу (prompt engineering). Правильно сформульовані системні і користувацькі повідомлення значно впливають на якість результату. Існують деякі рекомендації для ітеративного вдосконалення шаблонів запитів, до прикладу, фреймворк RecPrompt комбінує компонент оптимізації промптів (Prompt Optimizer), сама рекомендаційна система та монітор продуктивності. Як зазначено у дослідженні, підходи на основі LLM з ефективною настройкою промптів можуть у сотні разів прискорити розробку систем рекомендацій. Приклади таких практик: введення системних ролей (“ви – консультант з підбору товарів”), обмеження формату відповіді й налаштування гіперпараметрів LLM забезпечують більш інформативні та релевантні рекомендації [7].

Необхідно зазначити, що при роботі з LLM, існує особлива валюта яку називають токенами. Токени являють собою одиниці, якими модель розбиває текст, і саме на основі кількості використаних токенів відбувається облік обчислень та виставлення рахунків за виклики моделі. При кожному запиті до LLM сумуються токени вхідного контексту (промпт, включно з системною роллю і підстановками) та токени вихідної відповіді – загальна сума визначає обсяг оплати і впливає на затримку обробки. Окремі сервіси також тарифікують ембеддінги та інші API-виклики за токенами. Через це оптимізація промптів є важливим способом економії. Стиск шаблонів, винесення статичного контенту у зовнішнє сховище, використання менших моделей для простих завдань або кешування результатів зменшують витрати. Додатково важливо контролювати довжину контексту через ліміти вікна контексту моделі, дуже довгі запити можуть бути недопустимими та надто дорогими.

1.4 Аналіз систем на базі LLM

У цьому підрозділі розглянуто шість існуючих подібних систем, які використовують LLM та суміжні підходи для підтримки шопінгу й управління торговими процесами, а саме: Amazon Rufus, Google Shopping AI, Shopify Sidekick, Amazon Personalize, Klarna AI Shopping Assistant та TikTok Shop Seller Assistant. Вибір обумовлений тим, що ці рішення репрезентують різні архітектури та сценарії застосування – від клієнтських чат-асистентів та візуального пошуку до інструментів для продавців і класичних рекомендаторів. В них застосовані різноманітні підходи – чисті LLM-чат-асистенти, гібридні RAG-системи з графами продуктів, а також масштабовані рекомендаційні сервіси, що дозволяють порівняти вимоги до даних, приватності, обчислювальних ресурсів та швидкості інтеграції.

1.4.1 Amazon AI Chat Assistant

Amazon розробила AI-асистента для шопінгу Rufus, який зображено на рисунку 1.1, його інтегровано в мобільний додаток та вебверсію магазину, щоб зробити покупки швидшими й зручнішими. Rufus використовує генеративний і агентний AI, поєднуючи LLM та індивідуальні дані користувача, щоб відповідати на запитання покупців і пропонувати персоналізовані рекомендації товарів. Асистент аналізує історію переглядів і покупок кожного клієнта та формує поради, спираючись на контекст бесіди та особисті уподобання. Це дозволяє йому виконувати як загальні консультаційні завдання (порівнювати різні товари), так і деталізовані питання (уточнювати характеристики товарів) [8,9].

Крім індивідуального консультування, Rufus виконує роль інтерактивного помічника, який веде діалог у реальному часі, уточнює вимоги користувача та адаптує своє спілкування в залежності від поставленого завдання. Система аналізує поведінку покупця, пропонує підбірки товарів та додаткові товари, пропонує порівняння та може зменшувати інформаційне навантаження на користувача шляхом узагальнення технічних даних у прості рекомендації.

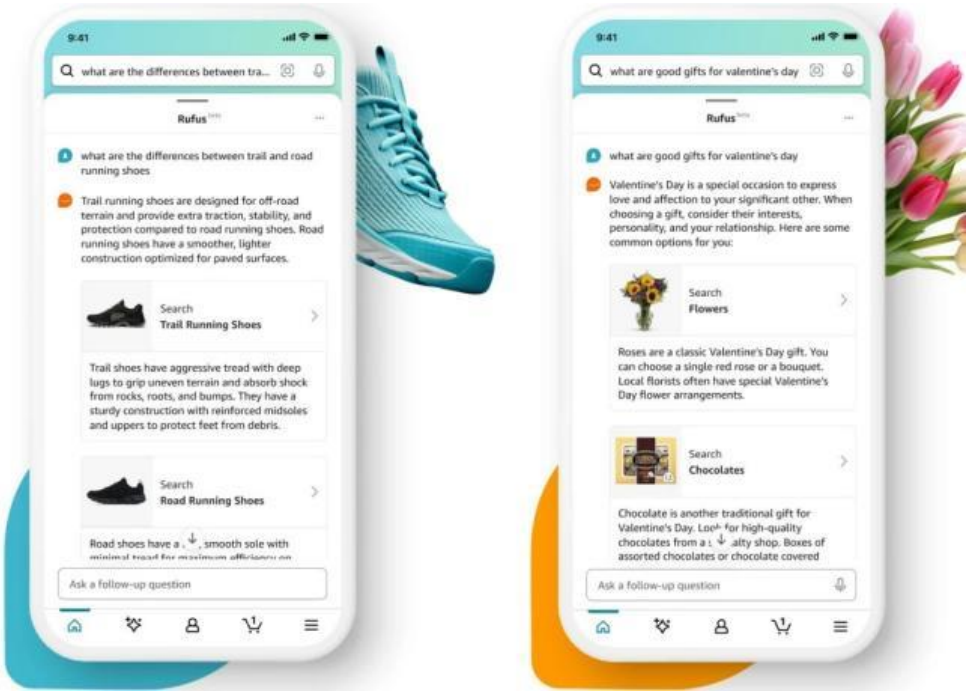

Рисунок 1.1 – Інтерфейс AI помічника Rufus

За архітектурою Rufus базується на AWS Bedrock та багатомодельній системі - сервер визначає через роутер оптимальну модель (Anthropic Claude, Amazon Nova чи власну модель Amazon) залежно від типу запиту. Система має власний граф знань атрибутів товарів і уподобань покупців, що дозволяє уточнювати результати на основі ключових характеристик товарів. Також застосовується підхід Retrieval-Augmented Generation – модель спочатку знаходить релевантну інформацію (відгуки, базу даних товарів, рейтинги, статті) і на її основі генерує обґрунтовані відповіді. Така архітектура розрахована на масштабування до сотень мільйонів користувачів та комбінує велику обчислювальну потужність з гнучким маршрутизатором запитів.

Водночас подібні системи мають потенційні вразливості. Дослідники компанії 0DIN виявили у Rufus небезпечну вразливість, а саме зашифровані ASCII-запити могли обійти фільтри безпеки та змусити систему видавати заборонену інформацію. Це підкреслює проблему галюцинацій та необхідність постійно посилювати механізми модерації. Крім того, збір великих обсягів персональних даних викликає занепокоєння щодо приватності й можливих упереджень у рекомендаціях. Наявність кількох великих моделей також означає

високі обчислювальні витрати і обмежує впровадження подібного рішення на невеликих платформах [10].

1.4.2 Google Shopping AI

Компанія Google представила оновлений сервіс Shopping, побудований на основі генеративного AI (моделі Gemini від DeepMind) та власного графа товарів. Система обробляє понад 45 мільярдів записів про продукти, формуючи для кожного пошукового запиту адаптований AI-бриф, що містить ключові аспекти пошуку (врахування клімату чи призначення товару) та переліки найбільш релевантних продуктів. Рекомендації на Google Shopping супроводжуються поясненнями і категоризованим переглядом варіантів. Крім того, на головній сторінці користувачів формується персоналізована стрічка з тематичними товарами та відео на основі їхніх інтересів. Такий підхід значно спрощує пошук товару, замість стандартного пошуку за ключовими словами покупець може ввести описовий запит і отримати згенеровану AI-підказку з переліком речей, які відповідають потребам, а також новини та статті з порадами [11].

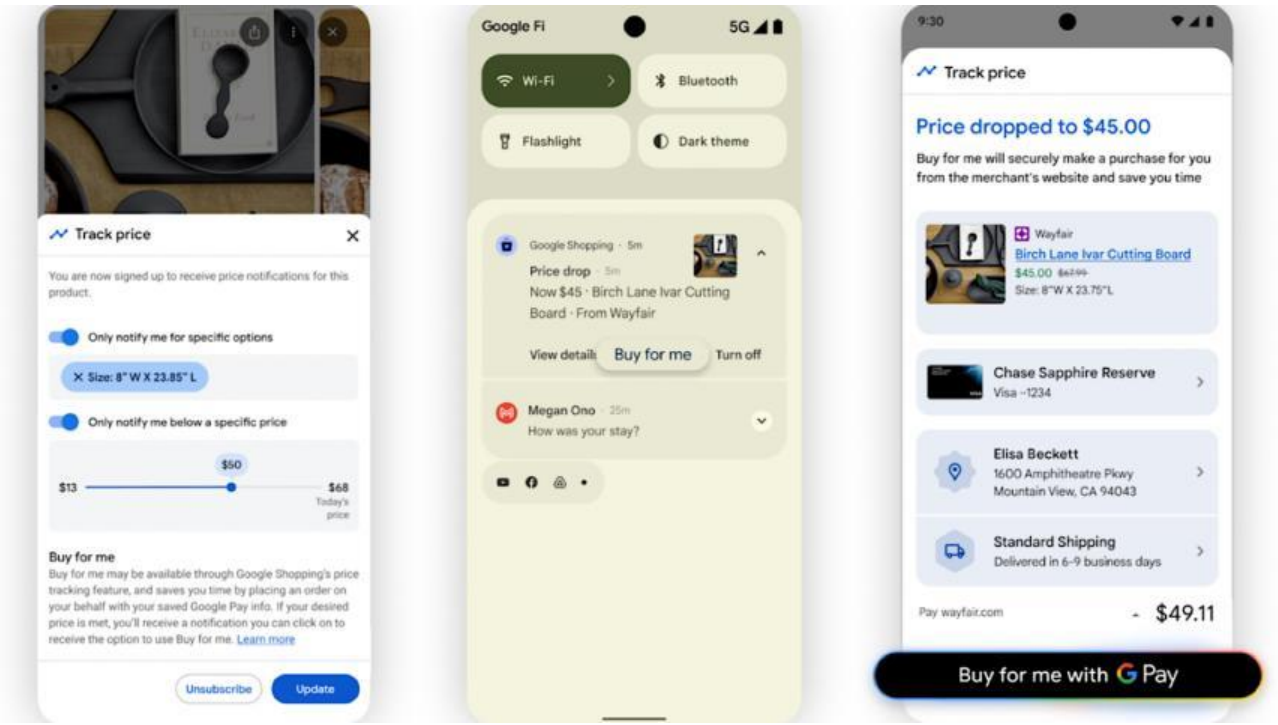

Рисунок 1.2 – Інтерфейс Google Shopping AI

У реалізації Google Shopping AI, який зображено на рисунку 1.2, поєднано модель Gemini з базою графу покупок – централізованою системою інформації про товари й продавців. Gemini аналізує текстовий запит і користувацький контекст, виділяючи ключові характеристики товарів та використовує їх для пошуку відповідних продуктів. Для побудови відповіді застосовується RAG-подібний підхід - система одночасно звертається до внутрішніх та зовнішніх джерел інформації і надає злиту інформацію. Інакше кажучи, Google поєднує свій багатий товарний граф із LLM, що дозволяє створювати комплексні і контекстуалізовані рекомендації. Крім того, у новій версії Shopping передбачено динамічні фільтри, віртуальну примірку в режимі доповненої реальності та інші інструменти, що допомагають уточнити результати.

Однак Google Shopping AI має обмеження. Рекомендації ґрунтуються на аналізі особистих даних та історії дій користувача, що порушує приватність та користувачу можуть показувати лише схожі товари з урахуванням попередніх вподобань. Можливі також помилки розуміння запитів – якщо модель неправильно інтерпретує запит, то й підсумкові рекомендації можуть бути нерелевантними. До того ж Gemini як LLM інколи може генерувати не точну інформацію у випадку нестачі якісних вихідних даних про нові або нішеві продукти. Ця проблема вимагає уважного контролю якості контенту, що подається покупцеві.

1.4.3 Shopify AI Assistant

Shopify презентувала Sidekick – персонального AI-помічника для власників інтернет-магазинів, інтегрованого у адміністративну панель платформи. Sidekick, який зображено на рисунку 1.3, постійно доступний для продавців, допомагаючи виконувати рутинні завдання і покращувати маркетинг. З його допомогою можна автоматично генерувати тексти (SEO-орієнтовані описи товарів чи листи email-розсилок), створювати рекламні банери та навіть готувати контент для соцмереж, використовуючи дані конкретного магазину. У чаті Shopify Inbox функція “Magic” дозволяє створювати персоналізовані відповіді на питання клієнтів, переводячи більше звернень у покупки. Крім того, Sidekick аналізує статистику магазину і

видає поради щодо оптимізації каталогу, показників конверсії та рекламних стратегій [12, 13].

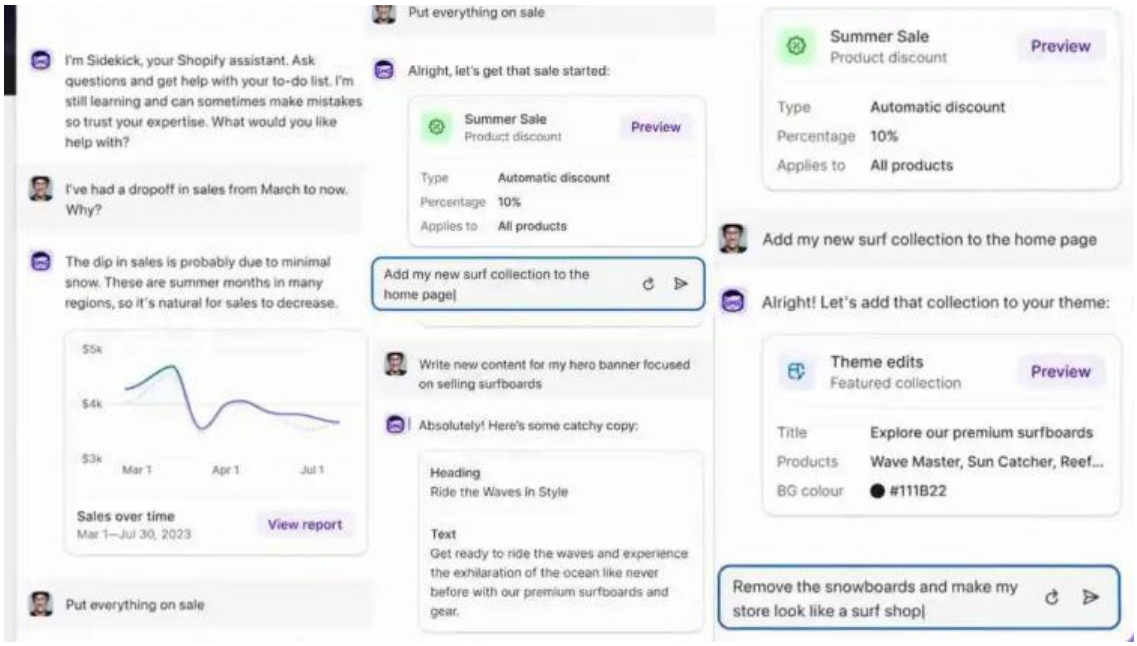

Рисунок 1.3 – Панель Shopify Sidekick

Технічно Sidekick побудований на основі великих мовних моделей і навчається на даних Shopify. Він має доступ до метаданих магазину – інформації про товари, ціни, запаси і статистику – через спеціальні API та протоколи по типу Model Context Protocol. Це дозволяє AI-агенту отримати каталог магазину та використовувати його у розмові. Наприклад, користувач може запитати “Які товари найбільше продавалися цього тижня?” чи “Додай у кошик найпопулярнішу футболку з червоною графікою”, і Sidekick зможе виконати ці дії за рахунок прямого підключення до даних магазину. Shopify також розвиває агентні можливості Sidekick, власники магазинів можуть додавати користувацькі AI-скрипти, що дозволяють здійснювати пошук, рекомендації та навіть оформлення замовлень у чаті безпосередньо через інтерфейс свого магазину.

Оскільки Sidekick отримує доступ до критичних даних магазину і може ініціювати транзакції, є потреба у забезпеченні надійних заходів безпеки. Неправильна генерація або некоректне формування опису товару можуть призвести до невірного відображення інформації. Shopify вводить жорсткі

правила використання автономних агентів та рекомендує передбачати резервні схеми передачі управління людині, на випадок помилок AI. Великі обчислювальні витрати LLM роблять подібні рішення менш доступними для малих компаній.

1.4.4 Amazon Personalize

Amazon Personalize – це хмарний сервіс машинного навчання AWS для побудови систем персоналізованих рекомендацій, архітектура якої зображена на рисунку 1.4. Він дозволяє розробникам швидко створити власний механізм рекомендацій на основі даних про поведінку користувачів (перегляди товарів, історія покупок тощо). Personalize використовує глибинні нейронні мережі, колаборативну фільтрацію та методи ранжування для того, щоб у реальному часі видавати релевантні рекомендації. Сервіс повністю керований – AWS самостійно організовує інфраструктуру і масштабування моделі, що дає змогу впровадити персоналізовані рекомендації за кілька годин замість тижнів розробки [14].

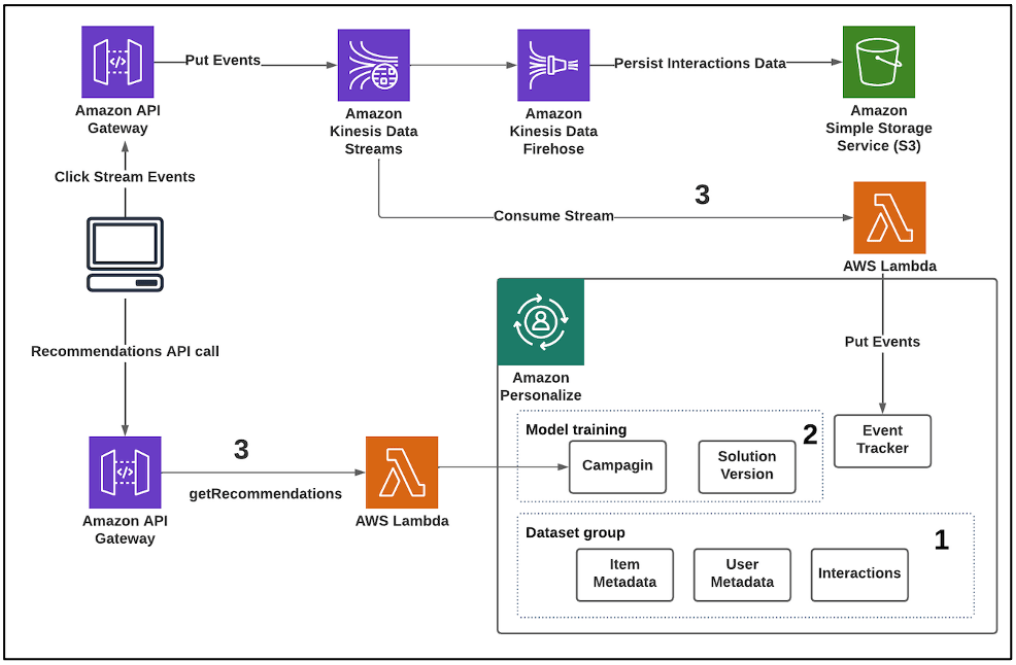

Рисунок 1.4 – Архітектура Amazon Personalize

Архітектурно Amazon Personalize має потік даних - інформація про взаємодії користувачів надходить у хмарне сховище, де попередньо обробляється, а потім передається в рекомендуючу модель. Користувач може обрати готовий

алгоритм або налаштувати власну архітектуру. Система підтримує оновлення моделей у реальному часі – якщо користувач взаємодіє з новими товарами, рекомендації зразу адаптуються. У 2025 році AWS додав можливість інтеграції Personalize з Amazon Bedrock - це дозволяє поєднувати рекомендації з генеративними моделями для створення контенту та тематичних річних звітів, а також для більш креативних продукт-рев'ю. Наприклад, можна генерувати опис рекомендацій з урахуванням сезонних тенденцій або створювати динамічні оголошення, доповнюючи традиційні рекомендації персоналізованим контентом.

Personalize має обмеження, для хорошої якості необхідні великі обсяги достовірних даних. Сервіс не призначений для безпосереднього діалогу з покупцем, а тільки генерує ранжування товарів за історією подій. Amazon Personalize сам по собі не відповідає на запитання в чаті чи не пояснює рекомендації користувачеві. Моделі також можуть підсилювати наявні тренди, дуже популярні товари отримують ще більше рекомендацій, що може зменшувати виявлення нішевих продуктів. Крім того, автоматичні системи рекомендацій потребують обережності щодо упереджень, якщо у тренувальних даних є історичні перекоси за цінностями, популярністю певних груп товарів тощо - то й результати рекомендацій можуть їх відтворювати. Для уникнення цього необхідне уважне тестування моделей та постійна оцінка якості рекомендацій.

1.4.5 Klarna AI Shopping Assistant

Шведська компанія Klarna впровадила в свій мобільний додаток AI-асистента для покупок, він виступає потужним помічником при покупках, допомагаючи користувачам знаходити та обирати товари відповідно до їхніх потреб. Асистент підтримує візуальний пошук - користувач може сфотографувати річ або скріншот, а система знайде аналогічні товари в різних магазинах. Також передбачено функцію чату на базі GPT від OpenAI. За допомогою чату, покупець може запитати у Klarna порівняння між товарами різних брендів або попросити різноманітні поради, асистент надає відповіді у розмовній формі у вигляді переліку релевантних продуктів з різних ритейлерів. Інтерфейс AI помічника Klarna зображено на рисунку 1.5. Додатково Klarna відстежує зміни цін на товари

та прогнозує ймовірність майбутніх знижок – це допомагає покупцям вирішити, чи варто чекати на акції, аби не переплачувати [15,16].

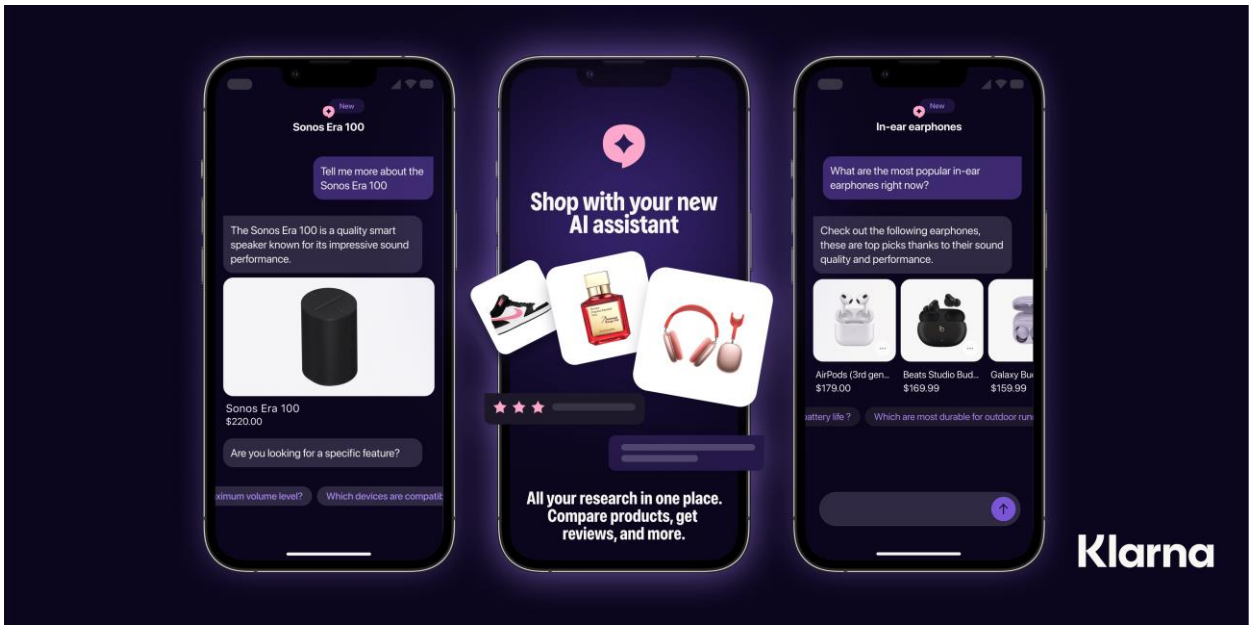

Рисунок 1.5 – AI помічник Klarna

Технічна реалізація Klarna AI поєднує LLM з вбудованими базами даних цін та товарів. Коли користувач формує запит або надсилає фото, система виконує пошук у глобальному каталозі партнерських магазинів, визначає відповідні пропозиції і передає їх LLM. Модель у свою чергу генерує зрозуміле пояснення і рекомендації, опираючись на контекст запиту. Якщо користувач цікавиться ціною або акціями, використовуються алгоритми аналізу історії цін – вони оцінюють, чи найнижча поточна ціна, та повідомляють про очікувані знижки. Таким чином, Klarna поєднує потужності LLM для генерації тексту, пошук за зображеннями і аналітичні моделі цін.

Досвід Klarna свідчить про потенційні ризики надмірної автоматизації. Наприклад, спроби компанії перевести значну частину служби підтримки на AI чат-боти призвели до зниження якості обслуговування і критики з боку клієнтів. Через це компанія зараз використовує змішану модель, де AI доповнює, але не повністю замінює людину. Цей приклад показує, що AI-асистент може давати неточні відповіді або неправильно інтерпретувати складні ситуації. Важливо

забезпечити контроль якості рекомендацій, перевіряти фактичні дані і давати покупцям можливість звернутися до живого консультанта, якщо це необхідно.

1.4.6 TikTok Shop AI Assistant

TikTok Shop створив AI-асистента для продавців Seller Assistant, інтерфейс якого зображено на рисунку 1.6, вбудованого в платформу управління продажами. Він являє собою чат-бота, який призначений допомагати брендам і магазинам оптимізувати свої товари та рекламу в додатку. Seller Assistant надає продавцям поради в реальному часі, він аналізує активність об’яв і дає персоналізовані рекомендації щодо оформлення товарів і маркетингових стратегій. Окрім того, продавець може отримати миттєвий доступ до ключової статистики (перегляди, продажі, коефіцієнт конверсії) та одразу отримати пояснення, що вплинуло на результати. Seller Assistant призначено покращити роботу з товарами, щоб власники магазинів могли оптимізувати їх з легкістю, він може підказати, які хештеги або підписи найкраще підходять для конкретної продукції [17].

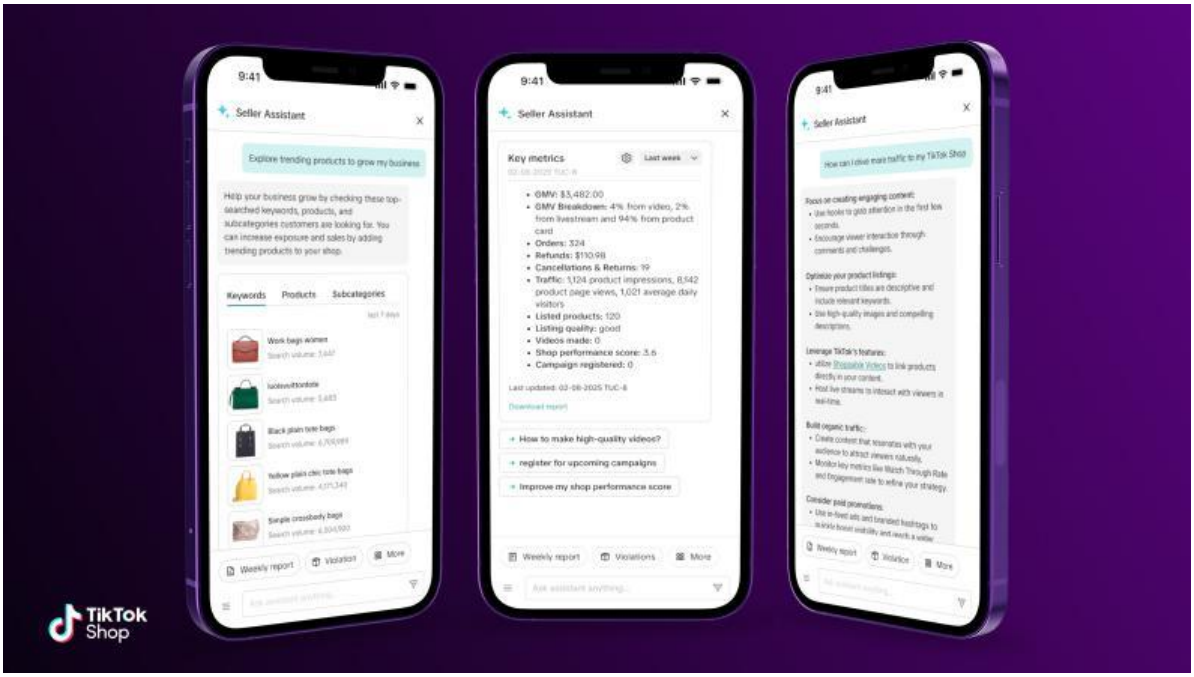

Рисунок 1.6 – AI помічник Seller Assistant

З точки зору архітектури, TikTok використовує власну платформу генеративного AI для забезпечення Seller Assistant. Модель отримує дані про

товари продавця та метрики кампаній і обробляє питання від користувача у діалоговому форматі. Наприклад, якщо продавець запитує “Які ключові слова краще використовувати для цієї сукні?”, асистент аналізує актуальні тренди і статистику та пропонує оптимальні варіанти. Інтеграція з аналітичними сервісами TikTok дозволяє моделі брати за основу реальні внутрішні дані, що підвищує релевантність порад. Варто зазначити, що наразі функціонал Seller Assistant орієнтовано насамперед на внутрішні процеси продавця, а не на безпосередні рекомендації покупцям.

Як і інші AI-рішення, ця система має свої обмеження. Її поради базуються на внутрішніх даних платформи і не враховують всі зовнішні фактори, такі як дії конкурентів чи зміну трендів поза TikTok. Точність рекомендацій залежить від якості вхідних даних і тренування моделі – неправильно зібрана статистика або неактуальні налаштування можуть призвести до нерелевантних порад. Наразі Seller Assistant являє собою допоміжний інструмент та має великі плани по розвитку.

1.5 Порівняння аналогів та висновок

За призначенням ці системи поділяються на клієнтські асистенти (Rufus, Google Shopping, Klarna), інструменти для продавців (Shopify Sidekick, TikTok Seller Assistant) та спеціалізований рекомендаційний сервіс (Amazon Personalize). Ключові відмінності у ролях використання LLM, джерела даних і рівень автономії.

Rufus, Google Shopping і Klarna широко використовують LLM для генерації природних відповідей і створення рекомендаційних брифів. Багато реалізацій застосовують RAG, щоб прив’язати генерацію до каталогу та відгуків. Sidekick і Seller Assistant більше орієнтовані на автоматизацію бізнес-процесів і контенту (описи, маркетинг), хоча теж можуть використовувати LLM для тексту.

Personalize – це класичний рекомендатор без вбудованої генерації відповідей його призначення у ранжуванні й швидкому оновленні рекомендацій в реальному часі.

Google і Amazon покладаються на великі графи продуктів й каталоги. Klarna додає візуальний пошук і історію цін, Shopify і TikTok працюють із метаданими магазинів і аналітикою продавця. Там, де використовується персоналізація на основі історії користувача, зростає ризик приватних витоків і упереджень – це загальна слабкість LLM-орієнтованих рішень.

Рішення з багатомодельною маршрутизацією дають гнучкість, але підвищують складність і витрати, службові рекомендатори ефективніші по ціні для ранжування. Агентні можливості створюють ризики, тому потрібні додаткові заходи безпеки й людський контроль.

У висновку, для швидкої інтеграції в існуючу систему найзручнішим та дешевшим шляхом є використання асистента у вигляді AI агента рекомендації товарів. Оренда LLM для генерації відповідей у парі з локальним модулем пошуку й отримання інформації з бази даних забезпечує мінімальні витрати на розробку й дозволяє швидко надати користувачам природнього асистента, водночас залишаючи критичні дані й логіку бізнес-операцій під контролем власної інфраструктури. Водночас рекомендовано передбачити кешування й фільтрацію чутливих даних, механізми модерації відповіді LLM та залучення людей для операцій, що впливають на транзакції або конфіденційність.

2 ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОГО АГЕНТА ДЛЯ ПІДБОРУ ТОВАРІВ

2.1 Вимоги до системи

Завданням розроблюваної інтелектуальної системи є надання персоналізованих підборів товарів у режимі діалогу з користувачем на основі LLM з технологією RAG. Кінцевими користувачами системи є відвідувачі

інтернет-магазину та зареєстровані користувачі, які очікують швидких, релевантних та аргументованих рекомендацій.

2.1.1 Загальні вимоги

Система повинна забезпечувати користувацький інтерфейс у вигляді вебдодатку, що дозволяє вводити запити в діалоговому режимі та отримувати структуровані відповіді з рекомендаціями. Додатково необхідно додати зручну можливість налаштування AI асистента для керівників бізнесу.

Серверна частина повинна бути реалізована на будь-якій сучасній мові програмування та надавати API-шлюз для взаємодії з клієнтом і зовнішнім LLM. Система повинна вміти розширювати вміст запиту до LLM (RAG підхід). Дані користувача, такі як історія покупок, сесії та його налаштування, повинні зберігаються в локальній БД і використовуються для персоналізації рекомендацій. Архітектура має бути сервісно-орієнтованою для можливості зручної інтеграції AI асистента у інші системи.

2.1.2 Функціональні вимоги

Система побудована на сервісно-орієнтованій архітектурі, тому необхідно налагодити зручний інтерфейс пересилання повідомлень між частинами сервісу. Прийом запитів від користувача відбувається у текстовій формі, та передача у формі JSON об’єктів, включаючи доповнення параметрів через RAG підхід. Аргументований промпт повинен бути доповнений витягом релевантної історії покупок для конкретного користувача перед формуванням. Сервер повинен формувати аргументований запит із чотирьох частин, а саме: системна інструкція, користувацький запит, розширення контексту та правила. Валідація та обробка відповіді LLM – обов’язкова перевірка формату запитів на серверній частині та підготовка остаточного JSON для клієнтської частини. Можливий віддалений виклик окремих функцій на серверній або клієнтських частинах від LLM через JSON об’єкт. Це знадобиться для показу користувачу списку рекомендованих товарів, у вигляді окремих іконок, із поясненнями, або виклик іншого функціоналу системи. Необхідна можливість збереження діалогової сесії для

відновлення після перезавантаження сторінки на клієнтській частині. Додатково необхідні обробки помилок та вивід їх у повідомленнях.

2.1.3 Нефункціональні вимоги

Клієнтоорієнтовані системи мають постійно оброблювати велику кількість запитів та витримувати навантаження, для цього необхідно зазначити вимоги до продуктивності. Середній час обробки запиту, при нормальному навантаженні, не повинен перевищувати 3 секунд, при пікових навантаженнях – не більше 6 секунд.

Окремо необхідно розглянути безпекові питання. Передача даних має здійснюватись тільки за HTTPS протоколом, API-ключі необхідно зберігати віддалено від системи. До зовнішнього LLM слід передавати лише необхідний мінімум даних, чутливі персональні дані повинні анонімізуватись або фільтруватись. Додатково необхідно реалізувати механізм обмеження частоти запитів для уникнення зловживань системою.

2.2 Архітектура системи

Для розробки AI асистента обрано підхід на базі LLM з емуляцією RAG. RAG-підхід означає, що до запиту LLM додаються релевантні факти з перевірених джерел такі як інформація з товарного каталогу або історія попередніх покупок. Такий підхід дозволяє моделі враховувати індивідуальний контекст користувача та зменшує кількість хибних або вигаданих відповідей. У агенті до запиту клієнта додається історія його покупок – набір описів чи ідентифікаторів раніше придбаних товарів. Після чого LLM обробляє комбінований запит, що дає змогу формувати персоналізовану рекомендацію товарів з поясненням. Двоступеневий процес запиту й пошуку по історії та перехід до генерація відповіді вважається ефективним для рекомендацій.

Системна архітектура, яка зображена на рисунку 2.1, складається з клієнтської вебчастини, серверної частини з FastAPI та орендована LLM. Для клієнтської частини інтерфейсу використано сучасні найпопулярніші підходи, а саме написання на мові програмування JavaScript з використанням бібліотеки React. Клієнтська частина забезпечує користувацький інтерфейс чату – введення описового запиту та відображення отриманої відповіді, та зв’язок із серверною частиною.

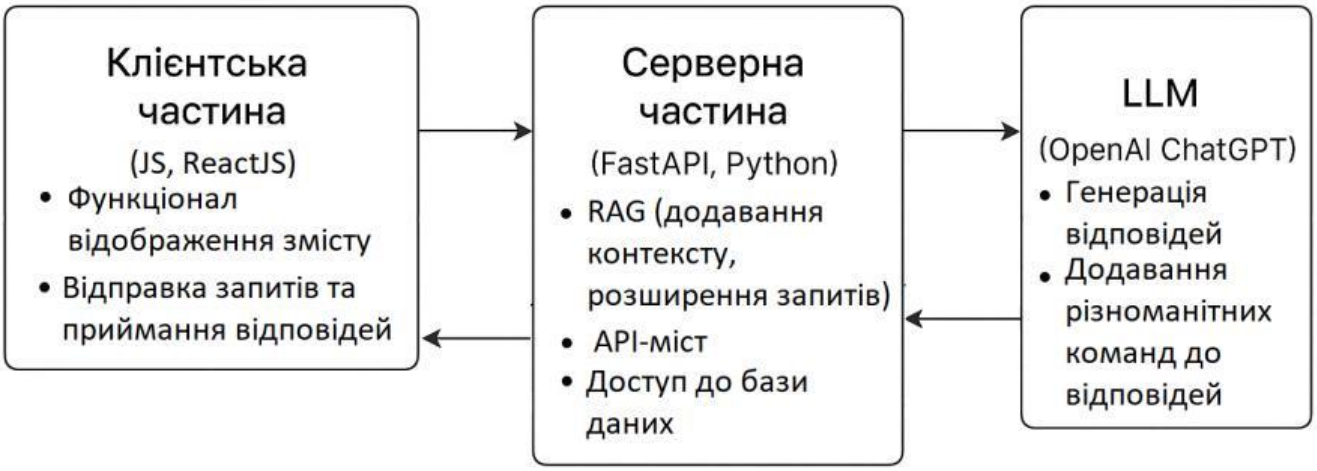

Рисунок 2.1 – Архітектура системи та її загальне призначення

Серверна частина оброблює клієнтські запити та відправляє їх до LLM, після чого повертає оброблений запит з LLM до клієнтської частини. Клієнт надсилає текстовий запит через HTTP до бекенду. Серверна частина побудована з використанням FastAPI на мові Python. FastAPI являє собою сучасний високопродуктивний фреймворк для побудови веб-API, що забезпечує швидку обробку запитів, автоматичну валідацію даних. Після отримання запиту серверний обробник звертається до локальної бази даних, де зберігається історія покупок користувача, і витягує релевантні записи. Отриманий контекст передається в модуль взаємодії з OpenAI через API. По завершенні генерації відповіді LLM, FastAPI-сервіс формує відповідь у форматі JSON і надсилає її назад на клієнт.

З великої кількості LLM обрано ChatGPT, таке рішення зумовлено дослідженням у переддипломній практиці, яке показало його схильність к вигідній власникам торгівлі, низьку ціну оренди, високу швидкість обробки запитів, відмовостійкість та популярність. Створення власної LLM, її навчання та підтримка, вимагає від власників великих грошових витрат на власний сервер із дорогими комплектуючими, тому цей варіант було відкинуто як неліквідний для малого та середнього бізнесу.

Необхідно зазначити як працює RAG-підхід, спочатку користувач вводить запит у клієнтському застосунку. Сервер обробляє його та з бази даних отримує історію покупок цього користувача. Далі сервер формує розширений запит, додаючи отримані дані до початкової фрази, і надсилає його до LLM. Модель генерує відповідь з урахуванням наданого контексту (відповідь є більш релевантною персональній інформації користувача), після чого результат повертається до клієнтської частини. Такий підхід забезпечує формування рекомендації з урахуванням індивідуальних вподобань та історії дій користувача. Через відсутність великої бази даних з товарами, RAG підхід частково емульовано. Таке рішення не впливає на функціонал AI асистента, але зробить його налаштування та тестування легшим.

2.3 Технології розробки

Для реалізації ключових компонентів системи обрані сучасні найрозповсюджені технології розробки вебсистем. Для клієнтської частини обрано мову програмування JavaScript з використанням бібліотеки React, такий вибір додатково обгрунтовано розподілом елементів сторінки на компоненти та іншим підходом до рендерингу сторінок, який зображено на рисунку 2.2.

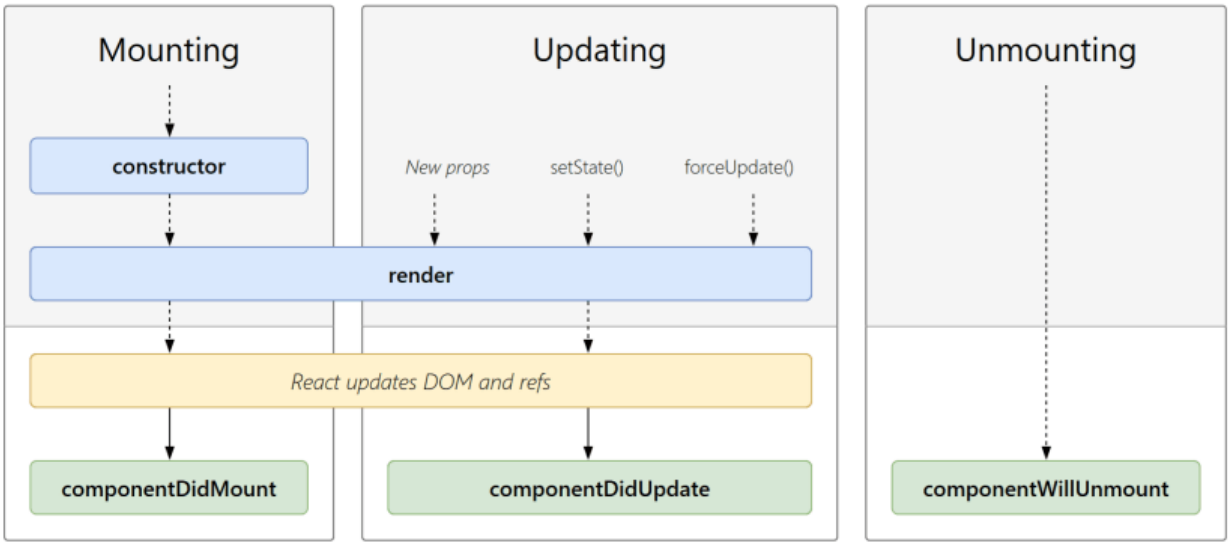

Рисунок 2.2 – Життєвий цикл компонентів React

Цей підхід дозволяє реалізувати віття інших сторінки системи без перезавантаження сторінок, таким чином діалогове вікно AI асистента не буде постійно зникати та перезавантажуватись, що є дуже важливим для клієнтів.

JavaScript є основною мовою програмування для вебклієнтів, яка виконується безпосередньо у браузері та забезпечує динамічну взаємодію користувача з інтерфейсом. Її ключова перевага у можливісті реалізовувати інтерактивну логіку на стороні клієнта. Завдяки JavaScript клієнтська частина працює стабільно, швидко та забезпечує зручну взаємодію користувача з інтелектуальним асистентом. React є сучасним популярним фреймворком, він має компонентну структуру, що спрощує побудову складних графічних інтерфейсів з багатьма станами, дозволяє ефективно оновлювати лише необхідні частини сторінки і має велику спільноту розробників.

FastAPI – для створення серверної логіки. Цей фреймворк забезпечує високу швидкодію і можливість обробки асинхронних запитів, а також інтегровану валідацію даних через Pydantic. Завдяки типізації Python-код FastAPI чітко структурований і добре документований.

У ролі LLM обрано OpenAI ChatGPT. Використання OpenAI API дає можливість швидко отримувати доступ до передових моделей через простий текстовий інтерфейс. API OpenAI спроєктовано як гнучкий і зручний для

розробників, це означає, що інтеграція з ChatGPT не вимагає складної інфраструктури, і модель можна легко оновлювати або замінювати без змін у клієнтській логіці.

Запропонована сервісно-орієнтована архітектура є легкою для розширення та масштабування. Компоненти фронтенду та бекенду розвиваються незалежно, а за потреби до них можна підключати додаткові мікросервіси. Модульний підхід з чітким розподілом функцій гарантує гнучкість системи та її відмовостійкість. Використання OpenAI API дозволяє масштабувати обчислення за рахунок хмарного сервісу без необхідності власної складної ML-інфраструктури. Усі ці властивості забезпечують стабільну роботу сервісу при зростанні числа користувачів і дозволяють легко додавати нові функціональності у майбутньому.

2.4 Діаграма послідовностей

Діаграми послідовностей використовуються для моделювання поведінки системи через взаємодію між об’єктами у часі. Вони відображають послідовність обміну повідомленнями, активації компонентів та повернення результатів під час реалізації конкретного функціоналу. За допомогою такого підходу можна продемонструвати не лише логіку виконання операцій, а й порядок викликів, залежності між компонентами та умови взаємодії між клієнтською частиною, сервером та зовнішніми сервісами.

У межах роботи створено діаграму послідовностей, яка ілюструє життєвий цикл обробки користувацького запиту з використанням усіх архітектурних частин системи Така діаграма дозволяє наочно оцінити логіку роботи компонентів, визначити ключові точки взаємодії та місця можливих помилок або затримок. Розроблену діаграму зображено на рисунку 2.3.

Рисунок 2.3 – Діаграма послідовності обробки користувацького запиту

Користувач вводить запит у інтерфейсі клієнтської частини, після чого ці дані збираються та формується повідомлення у форматі HTTP/JSON і надсилається до серверної частини. На стороні клієнта може відбутися попередній збір локального контексту (поточний діалог, параметри сесії), який додається до запиту перед відправленням. Сервер приймає запит, оброблює дані та звертається до внутрішніх джерел даних з яких витягує історію покупок користувача та інші релевантні метадані для формування розширеного запиту в RAG-форматі. Після побудови аргументованого промпту сервер виконує виклик до зовнішнього LLM-сервісу через OpenAI API. Модель обробляє отриманий промпт і генерує відповідь у вигляді природномовного тексту загорнутого у структурований JSON об’єкт (кандидати товарів, пояснення, метаінформація), після чого повертає результат серверу. Серверна частина проводить валідацію та постобробку відповіді, можливий виклик серверних функцій якщо є необхідність. Після цього сформований та перевірений результат передається назад на клієнтську частину, де відбувається повторний рендер частини інтерфейсу чату – показ списку

рекомендованих товарів та супровідних пояснень для користувача. У разі помилок або невідповідності формату передбачено гілку обробки винятків на сервері з поверненням контрольованого повідомлення на клієнт. Таким чином, діаграма демонструє послідовність дій від ініціації запиту до відображення персоналізованої відповіді, підкреслюючи роль серверної логіки у доповненні контекстом (RAG), контролі якості вихідних даних LLM та забезпеченні безпечної й передбачуваної взаємодії з користувачем.

У висновку обрана архітектура поєднує переваги вебтехнологій і сучасних LLM. Така система є достатньо гнучкою і масштабованою. Це відповідає більшості сучасних вебсистем, що дозволить виявити та вирішити сучасні проблеми інтеграції AI асистентів до вебсистем.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

У цьому розділі подано детальний опис реалізації розробленої інтелектуальної агенто-орієнтованої системи підбору товарів. Описано структуру та логіку роботи усіх частин системи, включно із принципами промпт-інжинірингу. Необхідно зазначити, що у дипломній роботі розглядається створення та інтеграція AI асистента у вже готові рішення, тому частина вебсистеми, яка відповідає стандартному інтернет магазину буде частково емульовано. Метою дипломної роботи є розробка AI асистента підбору товарів, тому реалізація інших частин системи, окрім інтерфейсів взаємодії з асистентом, не розглядається.

Основна увага зосереджена на демонстрації працездатності запропонованого підходу та практичних аспектів інтеграції асистента у реальні веб-сервіси. Реалізація допоміжної функціональності наведена лише в обсязі, необхідному для коректної роботи асистента та візуалізації результатів.

3.1 Оренда та налаштування віддаленої LLM

Для реалізації сервісу як віддаленої LLM використано хмарний сервіс API OpenAI. Фактично систему спроєктовано на мікросервісній архітектурі. Вона являє собою систему з набору слабко-пов’язаних сервісів. Базується на основі розподілу на різні модулі та спілкування їх за допомогою повідомлень. Мікросервісна архітектура наведена на рисунку 3.1.

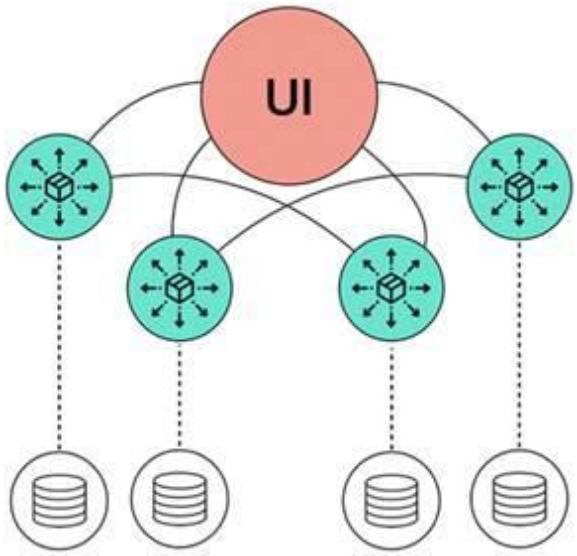

Рисунок 3.1 – Мікросервісна архітектура

Модель орендується у вигляді викликів до REST/SDK інтерфейсу OpenAI. Необхідно відправляти сформований запит до віддаленого сервера та отримувати відповідь і повертати її клієнту. Інфраструктура орендованої LLM побудована на серверах OpenAI, які розміщені у хмарі постачальника, тому власникам бізнесу та розробникам не потрібно керувати кластерами LLM або витрачати кошти на масштабування моделі. Інтерфейс орендованої системи, який наведено на рисунку 3.2, відображає усі необхідні власникам бізнесу дані про використання моделі. Інтеграція з LLM забезпечується через API-ключ та конфігурацію запитів. Такий підхід спрощує розгортання й дозволяє зосередитися на бізнес-логіці системи, але додаючи необхідність контролю витрат та безпеки при роботі з ключем API.

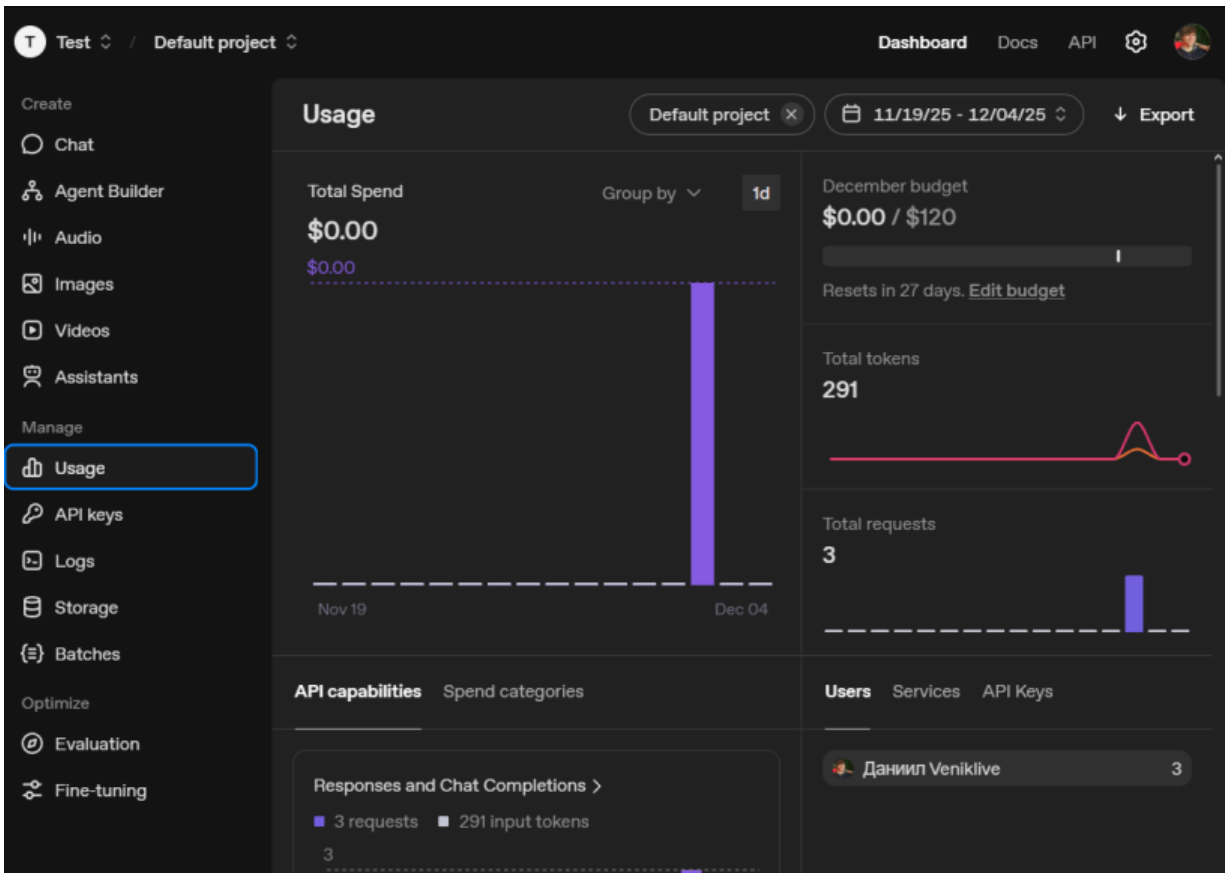

Рисунок 3.2 – Інтерфейс сервісу API OpenAI

Для розробки системи обрано модель “gpt-3.5-turbo-0125”. Такий вибір оснований на високій швидкодії, низькій вартості та стабільності LLM. Ця модель забезпечує достатню якість відповідей для стандартних консультаційних та інформаційних задач, працює швидко навіть при підвищеному навантаженні й має значно нижчу ціну за токен порівняно з новішими моделями. У межах дипломного проєкту це дає оптимальний баланс між якістю, продуктивністю та економічною доцільністю, що робить її практичним рішенням для побудови помічника інтернет-магазину.

3.1.1 Отримання та зберігання ключа API

API-ключ отримується через обліковий запис OpenAI та після отримання ключ зберігається тільки на серверній стороні у конфігурації середовища. У коді ключ необхідно підвантажувати при старті системи та використовувати для ініціалізації клієнта OpenAI. Важливо дотримуватись правил безпеки – ключ не має потрапити в клієнтську частину застосунку, не зберігатися в системах

контролю версій у відкритому вигляді, виконувати періодичну ротацію ключів, налаштувати мінімальні права використання.

3.1.2 Структура та побудова промптів

Промпт є набором вхідних повідомлень та інструкцій, які передаються до моделі. Перед створенням клієнтської та серверної частини необхідно визначитись із структурою інструкцій. У розроблюємій системі обрано створення промпту із чотирьох логічних частин, а саме: системна роль, правила й формат відповіді, контекст (RAG), та користувацький запит.

Системна роль являє собою глобальну інструкція для моделі, що задає її роль та тон. Ця частина визначає загальну поведінку асистента та обмеження формату відповіді.

Правила й формат відповіді – набір чітких правил за якими формується відповідь. До них входять обмеження довжини відповіді, форматування (JSON, список, таблиця), заборонені теми, політика щодо невизначеної інформації. Ця частина промпту створить вихідні відповіді передбачуваними та готовими до подальшої обробки.

RAG-контекст – релевантні факти, витягнуті з зовнішньої бази знань або каталогу товарів. Тут включаються або уривки товарних описів та атрибутів, або короткий підсумок знайдених записів. Контекст дає моделі фактичну інформацію, яка зменшує кількість вигаданих тверджень.

Користувацьким запитом є безпосередній текст, який ввів користувач. Він має містити запит на підбір товарів, уточнення фільтрів чи інші уточнюючі питання.

Під час розробки ці частини необхідно зкомбінувати у впорядкований масив повідомлень, що гарантує правильне трактування LLM. Далі у клієнтській частині необхідно створити ліміт користувацького запиту, для того щоб запобігти зловживанню токенами.

3.2 Клієнтська частина системи

Клієнтська частина реалізована як односторінковий вебдодаток на основі JavaScript та бібліотеки React з використанням стандартної мови розмітки HTML та таблиці каскадних стилей CSS. Використання рендерингу від бібліотеки React дозволяє позбавитись від основної проблеми AI асистентів, а саме постійне перезавантаження діалогового вікна при переході на інші сторінки, або інші дії з перерендирингом сторінки, та втрати контексту при цьому. Робота з рендерингом React наведена на рисунку 3.3. React використовує концепцію віртуального DOM, яка дозволяє ефективно оновлювати інтерфейс при зміні стану. Це критично важливо для діалогового вікна, де повідомлення додаються часто, а інтерфейс повинен постійно оновлюватись.

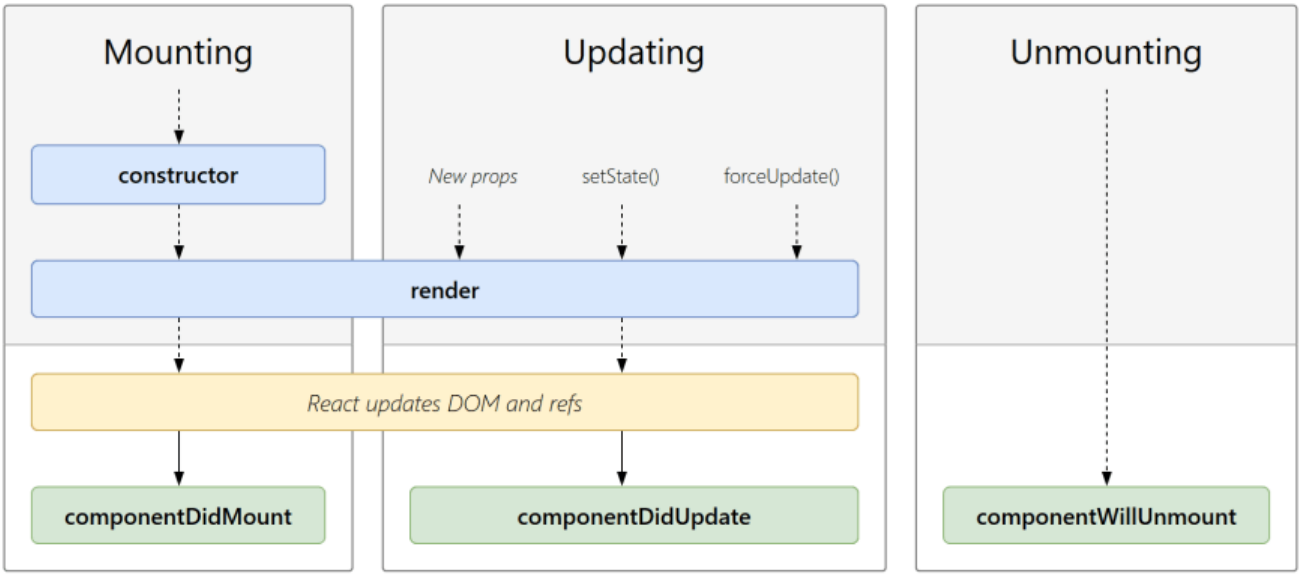

Рисунок 3.3 – Життєвий цикл React

Клієнтська частина складається з функціоналу діалогового вікна AI асистента, меню налаштувань асистента, збереження діалогу у локальному сховищі, відправка та прийняття даних від серверної частини та інший обов’язковий, типовий для клієнтської частини, функціонал.

3.2.1 Створення діалогового інтерфейсу

Діалогове вікно, яке зображено на малюнку 3.4 та його код у додатку А, є головним інтерфейсом взаємодії клієнта з AI частиною системи. Він відображає історію повідомлень у хронологічному порядку, розділених на репліки користувача та агента. Кожне повідомлення оформлене окремим компонентом React. Поле вводу тексту реалізоване як керований компонент React із відстеженням введеного тексту у стані, який називається “state”. При натисканні кнопки відправки повідомлення, поточний текст передається серверу для обробки, а в локальний стан додається нове повідомлення користувача.

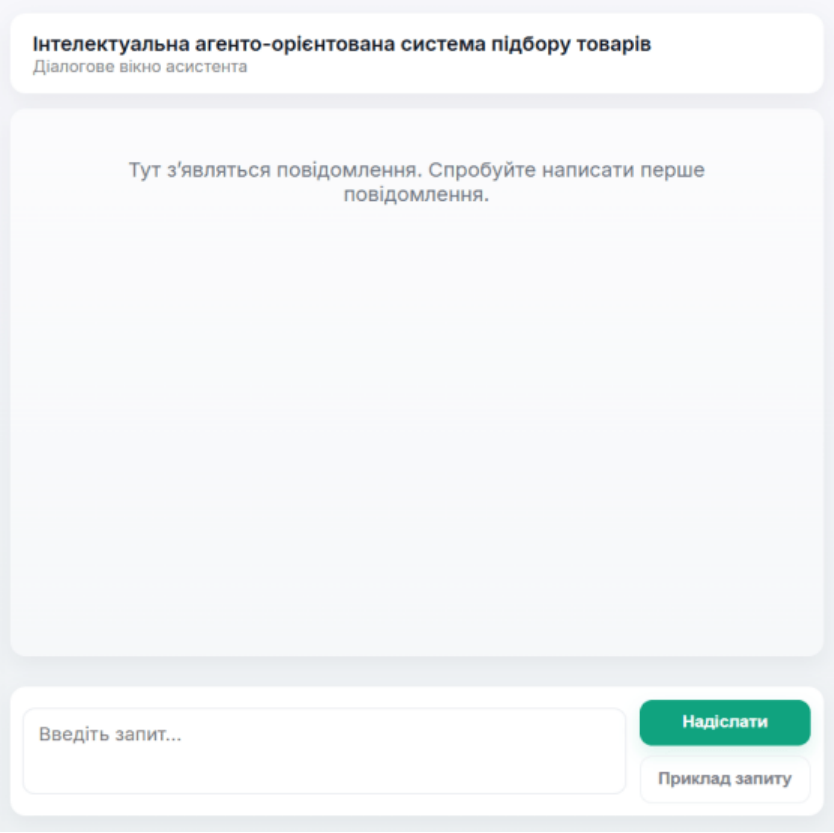

Рисунок 3.4 – Інтерфейс діалогового вікна із AI асистентом

Після реалізації діалогового вікна, необхідно реалізувати подальшу відправку та прийняття повідомлень. Реалізація цього функціоналу наведена у лістингу 3.1.

Лістинг 3.1 – Функція відправки повідомлень до серверу та назад

```
const handleSend = async () => {
    if (!input.trim()) return;
    setLoading(true);
    const newMsg = {
      role: "user",
      content: input.trim(),
      timestamp: Date.now(),
    };
    setMessages((m) => [...m, newMsg]);
    try {
      const res = await fetch("http://127.0.0.1:8000/api/llm/chat",
{
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ message: input.trim() }),
      });

      if (!res.ok) {
        const txt = await res.text();
        console.error("Server returned non-ok:", res.status, txt);
        throw new Error(`Server error ${res.status}`);
      }
      const data = await res.json();
      setMessages((m) => [
        ...m,
        {
          role: "assistant",
          content: data.reply ?? "Порожня відповідь сервера",
          timestamp: Date.now(),
        },
      ]);
    } catch (err) {
      console.error(err);
      setMessages((m) => [
        ...m,
        {
          role: "assistant",
          content: "Помилка під час запиту до сервера.",
          timestamp: Date.now(),
        },
      ]);
    } finally {
      setLoading(false);
      setInput("");
    }
  };
```

Функція “handleSend” реалізує повний клієнтський цикл обміну повідомленнями. Вона спочатку перевіряє та нормалізує введений текст,

обрізаючи зайві пробіли і відсікаючи пусті рядки, після чого встановлює стан завантаження для блокування повторних відправлень і відображення індикатора. Одразу після цього до історії повідомлень додається тимчасове повідомлення користувача, сформоване з ролі, вмісту та мітки часу, причому оновлення стану виконується функціонально щоб уникнути конфліктів при паралельних оновленнях. Далі відправляється асинхронний POST-запит до серверної частини з заголовком “Content-Type application JSON” та тілом, що містить текст запиту, після чого перевіряється HTTP-статус відповіді, і в разі невдачі спочатку читається текст помилки для діагностики і лише потім викликається загальна обробка помилки. Якщо відповідь успішна, її тіло розбирається як JSON об’єкт, звідки дістається поле “reply” та додається до історії як відповідь асистента з власною міткою часу. У випадку відсутності очікуваного поля застосовується запасне значення для запобігання помилок у інтерфейсі. Блок “catch” гарантує, що будь-які мережеві або типові помилки будуть зафіксовані у логах та відображені користувачу у вигляді повідомлення про помилку в чаті. Блок “finally” скидає індикатор завантаження та очищує поле вводу, готуючи інтерфейс до наступної взаємодії.

3.2.2 Функціонал збереження та відновлення повідомлень

Наступним кроком є реалізація функціоналу збереження повідомлень та відновлення чату, це дає можливість не втрачати користувачу діалог з асистентом навіть після перезавантаження вебсайту чи усієї системи. Реалізація функціоналу наведена у лістингах 3.2, 3.3 та 3.4.

Лістинг 3.2 – Ініціалізація “messages” та читання із “localStorage”

```
const CHAT_KEY = "llm_chat_history_v1";

const [messages, setMessages] = useState(() => {
  try {
    const raw = localStorage.getItem(CHAT_KEY);
    return raw ? JSON.parse(raw) : [];
  } catch (err) {
    console.error("Error parsing chat from localStorage:", err);
    return [];
  }});
```

У лістингу 3.3 наведено фрагмент, який зчитує збережену історію чату під ключем “llm_chat_history_v1” під час першого рендера компонента. Далі використано React хук “ useState(() => {...})”, який дозволяє ініціалізувати функції для зчитування та зміни у стані. Така реалізація дозволяє уникнути перезапису стану порожнім масивом при монтуванні та значення з “localStorage” застосовується лише один раз при ініціалізації. “JSON.parse” загорнуто в конструкцію try/catch для запобіганню виникання критичних помилок приневалідних або пошкоджених даних. При помилці у консоль виводиться помилка, а стан встановлюється у порожній масив.

Лістинг 3.3 – Збереження повідомлень при зміні стану

```
useEffect(() => {
  try {
    localStorage.setItem(CHAT_KEY, JSON.stringify(messages));
  } catch (err) {
    console.error("Не вдалося зберегти чат у localStorage:", err);
  }
  messagesRef.current?.scrollIntoView({ behavior: "smooth", block: 
"end" });
}, [messages]);
```

Хук “useEffect”, який наведено у лістингу 3.4, спрацьовує щоразу при зміні змінної “messages” та перетворює масив у рядок JSON за допомогою функції “JSON.stringify”, після чого записує повідомлення у локальне сховище під тим самим ключем. Операція огорнута в конструкцію try/catch захищає від помилок, якщо місця в сховищі недостатньо або об'єкт містить циклічні структури. Додатковий рядок коду “messagesRef.current?.scrollIntoView” являє собою зручність для користувацького досвіду (забороняє скрол) та не стосується збереження повідомлень.

Лістинг 3.4 – Очищення збереження при видаленні чату

```
function clearChatAndStorage() {
  setMessages([]);
  setEvaluation("");
  try {
    localStorage.removeItem(CHAT_KEY);
```

```
  } catch (err) {
    console.error("Не вдалося очистити localStorage:", err);
  }
}
```

Функція, яка наведена у лістингу 3.5, очищує стан чату методом запису порожнього масиву у змінну “messages” та видаляє ключ “CHAT_KEY” з локального сховища. Операція додатково обгорнута у конструкцію try/catch на випадок отримання помилок при доступі до сховища. При отриманні помилки, у консоль виводиться повідомлення "Не вдалося очистити localStorage:" та код помилки.

3.2.3 Створення інтерфейсу налаштувань LLM

Розповсюдженою практикою є написання промптів на серверній частині, але при цьому не враховується зручність власників бізнесу, кожна зміна або додання нового правила змушує власників найняти спеціалістів. Щоб цього уникнути та зробити це зручним, промпт систематизовано на окремі частини та створено інтерфейс для його налаштування у клієнтській частині.

У цьому розділі наведено код інтерфейсу для налаштування промпту LLM, а саме поля для системної ролі, редактор правил, обмежень формату та контексту (RAG), а також логіку їх завантаження зі серверу й збереження назад. Така реалізація дозволить власникам або адміністраторам системи зручно контролювати поведінку асистента, формувати передбачувані й відформатовані відповіді, швидко підлаштовувати систему під бізнес-вимоги та економічно оптимізувати використання токенів під час викликів моделі. Елементи користувацького інтерфейсу, що зв’язують редактор зі збереженням та передачею наведені у додатку А.

Першим етапом, який наведено у лістингу 3.5, проініціалізовано усі стани, які відповідають за промпт до LLM.

Лістинг 3.5 – Стани для промпту

```
const [promptSystemRole, setPromptSystemRole] = useState("");
const [promptRules, setPromptRules] = useState([]); // array of
strings
```

```
const [ruleInput, setRuleInput] = useState("");
const [promptFormat, setPromptFormat] = useState("");
const [promptContextText, setPromptContextText] = useState("");
const [promptSaved, setPromptSaved] = useState(false);
```

Ці “useState” зберігають поточні значення трьох частин промпту, до якого входить системна інструкція, набір правил і обмеження формату та текст контексту для RAG. “ruleInput” є тимчасовим полем для додавання нового правила. “promptSaved” – прапорець для короткої відповіді після успішного збереження. Усе це реалізовано як локальний редактор у клієнта, фінальна версія промпту передається на сервер при натисканні кнопки “Зберегти”.

Налаштування промпту необхідно зберігати у серверній частині, тому наступним кроком є реалізація підвантаження промпту із серверної частини для подальшого його редагування у клієнтській частині.

Лістинг 3.6 – Завантаження конфігурації промпту з серверу

```
async function loadPromptFromServer() {
  try {
    const res = await fetch("http://127.0.0.1:8000/api/llm/prompt");
    if (!res.ok) {
      console.warn("loadPromptFromServer: non-ok response",
res.status);
      return;
    }
    const data = await res.json();
    setPromptSystemRole(data.system_role || "");
    setPromptRules(Array.isArray(data.rules) ? data.rules : []);
    setPromptFormat(data.format_constraints || "");
    setPromptContextText(data.context_text || "");
  } catch (err) {
    console.error("loadPromptFromServer:", err);
  }
}
```

Функція, яка наведена у лістингу 3.6, робить GET-запит до “/api/llm/prompt” та очікує відповідь у вигляді JSON об’єкту “{system_role, rules, format_constraints, context_text}”. Отримані поля записуються у локальні стани редактора. Прописана основна перевірка для типів отриманих даних. При отриманні критичних помилок не вудбувається припинення виконання коду через конструкцію try/catch.

Зворотна функція передачі налаштувань до сервера наведена у лістингу 3.7.

Лістинг 3.7 – Передача налаштувань промпту до сервера

```
async function savePromptToServer() {
  try {
    const payload = {
      system_role: promptSystemRole,
      rules: promptRules,
      format_constraints: promptFormat,
      context_text: promptContextText,
    };
    const res = await fetch("http://127.0.0.1:8000/api/llm/prompt",
{
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload),
    });
    if (!res.ok) throw new Error(`Save failed ${res.status}`);
    setPromptSaved(true);
    setTimeout(() => setPromptSaved(false), 2000);
  } catch (err) {
    console.error("savePromptToServer:", err);
    alert("Не вдалося зберегти налаштування промпту.");
  }
}
```

Ця функція складає із поточних станів редактора усі налаштування у один JSON об’єкт та надсилає його методом POST до серверної частини системи. У подальшому сервер має зберігати ці налаштування у конфігураційний файл. Після успішної передачі даних, до консолі виводиться повідомлення “promptSaved”.

Функціонал редагування промптів простий та реалізований за допомогою інпутів з підтягуванням з них даних, але функціонал додавання нових правил реалізований інакше. Замість одного текстового поля, він приймає масив, у якому можна редагувати кожне окреме правило, його видаляти або додавати нове. До його реалізації доданий допоміжний код, який наведено у лістингу 3.8.

Лістинг 3.8 – Зручне додавання та видалення правил

```
const addRule = () => {
  if (ruleInput.trim()) {
    setPromptRules((prev) => [...prev, ruleInput.trim()]);
    setRuleInput("");
```

```
  }
};
const removeRule = (idx) => {
  setPromptRules((prev) => prev.filter((_, i) => i !== idx));
};
```

Функція “addRule” додає новий рядок до масиву правил лише якщо введено не пустий текст. “removeRule” видаляє правило за індексом. Обидві операції виконують оновлення через функціональне оновлення “setPromptRules(prev => ...)”, що є безпечним при паралельних операціях.

У висновку, логіка клієнтської частини зводиться до оновлення стану та передачі даних серверу. Коли користувач вводить текст або завантажує файл, відповідні функції-обробники оновлюють внутрішній стан React-компонентів і відправляють HTTP-запит до бекенду з необхідними даними. Отримавши відповідь від сервера, фронтенд оновлює інтерфейс та зберігає новий стан історії. Завдяки використанню CSS, налаштовано зовнішній вигляд компонентів формується через класи, що забезпечує адаптивність та однорідність дизайну. Загалом архітектура клієнтської частини реалізована за принципами одно-сторінкового додатку, де вміст оновлюється динамічно без перезавантаження сторінки.

3.3 Розробка серверної частини

Серверну частину побудовано на мові Python та з використанням фреймворку FastAPI, що дозволило забезпечити асинхронну обробку HTTP-запитів. FastAPI обрано завдяки можливості обробляти запити асинхронно, що важливо при очікуванні відповідей від зовнішніх сервісів та при надходженні інших запитів. Першочергово необхідно ініціалізувати клієнта OpenAI API, щоб один раз встановити з’єднання з OpenAI і повторно використовувати його при

кожному виклику, без переналаштування між запитами. Конфігурація та ініціалізація OpenAI клієнта наведено у лістингу 3.9.

Лістинг 3.9 – Конфігурація та ініціалізація OpenAI клієнта

```
import os
import logging
from dotenv import load_dotenv
from openai import OpenAI
load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
OPENAI_MODEL = os.getenv("OPENAI_MODEL", "gpt-3.5-turbo-0125")
OPENAI_FALLBACK = os.getenv("OPENAI_FALLBACK", "").lower()
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("llm-backend")
client = None
try:
    if OPENAI_API_KEY:
        client = OpenAI(api_key=OPENAI_API_KEY)
        logger.info("OpenAI client ініціалізовано. Модель: %s",
OPENAI_MODEL)
    else:
        client = None
except Exception as e:
    logger.exception("Не вдалося ініціалізувати OpenAI client: %s",
e)
    client = None
```

Необхідно зазначити, що у коді, API ключ завантажується з локального файлу “.env” та використовується лише на сервері, це зроблено для безпеки ключа та згідно загальноприйнятих безпекових вимог. Далі встановлюються константи для API та ініціалізується клієнт OpenAI. Логування повідомляє, чи вдалось створити клієнта та у відсутності ключа або при помилці сервер переходить у DEMO-режим. Цей режим створено для відправки відповідей у випадку присутності проблем з LLM (відсутності облікового запису/квоти).

Лістинг 3.10 – Ініціалізація FastAPI та CORS

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
app = FastAPI(title="LLM Testing Backend")
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
```

```
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

У лістингу 3.10 наведено створення екземпляру FastAPI та додання проміжного забезпечення CORS, необхідного для дозволу запитів із браузера.

Наступним кроком є використання формування правильних запитів, для цього застосовано бібліотеку Pydantic для Python. У лістингу 3.6 наведено використання Pydantic-моделі, які задають структуру вхідних запитів. “ChatRequest” для основного чату та додаткове логування.

Після організації структур повідомлень, ініціалізації та інших налаштувань, розроблено основну логіку сервера, яка наведена у лістингу 3.11.

Лістинг 3.11 – Логіка чату та виклик OpenAI

```
@app.post("/api/llm/chat")
def chat(req: ChatRequest):
    user_message = req.message
    logger.info("New chat request. msg=%s", user_message[:200])
    if OPENAI_FALLBACK == "demo" or client is None:
        reply = demo_reply_for(user_message)
        LOGS.append({"user": user_message, "reply": reply, "demo":
True})
        return {"reply": reply, "note": "DEMO"}
    effective_prompt = PROMPT_CONFIG
    if req.prompt_override:
        override = PromptConfig(...)
        effective_prompt = override
    system_messages = [
       {"role": "system", "content": effective_prompt.system_role},]
    messages_payload = system_messages + [{"role": "user",
"content": user_message}]
    response = client.chat.completions.create(
        model=OPENAI_MODEL,
        messages=messages_payload,)
    reply = response.choices[0].message.content
    LOGS.append({"user": user_message, "reply": reply})
    return {"reply": reply}
```

Основна логіка серверної частини полягає в прийомі POST-запитів від фронтенду, побудові системного промпту, виклику OpenAI через SDK та поверненні відповіді клієнту. Кожен запит містить JSON-дані з параметрами сесії.

Сервер формує повний набір повідомлень для моделі, додаючи системний промпт на початку, викликає метод “client.chat.completions” OpenAI і витягує з відповіді текст, який надсилає назад у JSON-форматі.

Реалізована серверна частина підтримує централізоване управління промптами, тимчасові перезаписи для ad-hoc тестування. Для підвищення стабільності й зручності тестування передбачено механізм надсилання відповідей у випадку якщо сервер запущено в DEMO-режимі або клієнт OpenAI не ініціалізований ключ API. У цьому випадку сервер повертає просту демонстраційну відповідь замість виклику моделі. У разі помилок під час виклику OpenAI код повертає помилки, пов’язані з вичерпаним лімітом LLM. Така реалізація необхідна щоб інтерфейс клієнтської частини не отримав критичної помилки. Інші невідновлювані помилки повертаються у вигляді “HTTP 500” для подальшої діагностики.

Для налагодження і моніторингу роботи використовується стандартний модуль “logging” мови Python. На початку обробника запиту реєструється інформація про вхідні дані, після отримання відповіді – результат виклику моделі. FastAPI вміє генерувати стандартні HTTP-логи та додаткове логування викликів до LLM полегшує діагностику відмов, зокрема випадків перевищення квот OpenAI.

3.4 Збільшення інтерактивності системи

Розроблена вебсистема вміє передавати усі необхідні повідомлення до LLM, але відповіді є повністю текстовими, без посилань та різноманітних карток предметів. Реалізація виводу зображень, та інших зручних для користувача нововведень, потребує додаткової обробки зворотніх повідомлень на серверній та клієнтських частинах системи. Найпростіший та надійнішим підходом є проінструктувати модель повертати структурований JSON зі спеціальним полем

“items: [...]”. Серверна частина системи після отримання тексту від моделі спробує розгорнути JSON об’єкт і, якщо знайде масив “items”, поверне назад клієнту додаткове поле “items” та тип “cards”. Після отримання клієнтською частиною повідомлення, необхідно обробити дані за допомогою конструкції switch case по типу відповіді та відобразити картки, якщо масив предметів присутній.

Першим етапом модифікації є розширення формату промпту, а саме додано вимогу повертати JSON з масива “items”. У системний промпт додано жорсткі форматні обмеження, які змушують LLM повертати структурований JSON з масивом товарів “items”, а текстове пояснення залишати поза JSON.

Розроблено функцію перетворення відповіді “try_extract_items_from_reply”, яка наведена у додатку А.3. Сервер після відповіді LLM починає пошук та перетворення JSON-блоку із масивом “items” навіть якщо модель повернула його всередині JSON об’єкта або з невеликими помилками. Разом із масивом повертається текст-обґрунтування.

Додане очищення та валідація товарів “sanitize_items”, яке наведено у лістингу 3.12.

Лістинг 3.12 – Очищення та валідація товарів

```
def sanitize_items(items: List[Dict[str, Any]], max_items: int = MAX_ITEMS_RETURN) -> List[Dict[str, Any]]:
    safe: List[Dict[str, Any]] = []
    for it in items:
        if not isinstance(it, dict):
            continue
        id_ = str(it.get("id", "")).strip()
        title = str(it.get("title", "")).strip()
        url = str(it.get("url", "")).strip()
        if not id_ or not title or not url:
            continue
        if not re.match(r"^https?://", url, re.IGNORECASE):
            continue
        if len(title) > 300 or len(url) > 1000:
            continue
        safe_item = {
            "id": id_,
            "title": title,
            "url": url,
            "price": it.get("price"),
            "image": it.get("image"),
```

```
            "description": it.get("description"),
            "specs": it.get("specs") if isinstance(it.get("specs"),
list) else [],
        }
        safe.append(safe_item)
        if len(safe) >= max_items:
            break
    return safe
```

Перед відправкою до клієнтської частини, усі елементи “items” проходять фільтрацію. Перевіряються обов’язкові поля, формат посилань, обмежуються довжини, нормалізуються характеристики специфікацій. Така реалізація захищає інтерфейс від помилок та гарантує коректний формат даних для карток.

Лістинг 3.13 – Зміна формату відповіді в основному обробнику

```
@app.post("/api/llm/chat")
def chat(req: ChatRequest):
    ...
    messages_payload = system_messages + [{"role": "user",
"content": user_message}]
    try:
        logger.info("Calling OpenAI model=%s", OPENAI_MODEL)
        response = client.chat.completions.create(
            model=OPENAI_MODEL,
            messages=messages_payload,
        )
        try:
            reply_raw = response.choices[0].message.content
        except Exception:
            reply_raw = str(response)
        logger.info("OpenAI reply length=%d", len(reply_raw) if
isinstance(reply_raw, str) else 0)
        items_raw, justification =
try_extract_items_from_reply(reply_raw)
        if items_raw:
            items_safe = sanitize_items(items_raw,
max_items=MAX_ITEMS_RETURN)
            LOGS.append({"user": user_message, "reply":
justification or "[items]", "items_count": len(items_safe)})
            return {"items": items_safe, "justification":
justification or "", "type": "cards"}
        else:
            LOGS.append({"user": user_message, "reply": reply_raw})
            return {"reply": reply_raw, "type": "text"}
```

Модифікований обробник, який наведено у лістингу 3.13, вміє повертати два типи відповідей, а саме звичайну текстову відповідь типу “{"reply": "...", "type": "text"}” та нову відповідь з картками товарів “{"items": [...], "justification": "...", "type": "cards"}”, який активується, якщо у відповіді LLM знайдено коректний масив “items”. Клієнтська частина по полю “type” визначає, що замість простого тексту потрібно створити горизонтальний список карток товарів з

Основні модифікації для обробки нових відповідей наведено у лістингу 3.14.

Лістинг 3.14 – Доданий код до функції “handleSend”

```
    const data = await res.json();
    if (data.items && Array.isArray(data.items) &&
data.items.length > 0) {
      const justification = data.justification || "";
      setMessages((m) => [
        ...m,
        {
          role: "assistant",
          type: "cards",
          items: data.items,
          justification,
          content: justification, // для сумісності з існуючим
рендером
          timestamp: Date.now(),
        },
      ]);
    } else {
      setMessages((m) => [
        ...m,
        {
          role: "assistant",
          content: data.reply ?? "Порожня відповідь сервера",
          timestamp: Date.now(),
        },
      ]);
    }
```

Цей код реалізує розгалуження обробника на клієнтській частині, якщо сервер повернув масив “items”, у історію додається спеціальне повідомлення типу “cards” з товарами та обґрунтуванням, інакше зберігається звичайна текстова

відповідь асистента. Інші модифікації клієнтської частини впливають на візуальну частину відображення карток з товарами, тому їх код не розглянуто.

Подібним чином можна додавати до системи обробку нових, необхідних для бізнесу, типів відповідей – блоки з ключовими характеристиками товарів, інтерактивні порівняння декількох моделей, спеціальні картки, підбірки супутніх товарів, посилання на статті підтримки або інструкції, візуалізацію статусу замовлення тощо. Фактично описаний підхід із JSON-структурою та клієнтським розгалуженням за типом відповіді є прикладом додавання нового функціоналу до вебсистеми без радикальної зміни архітектури. Серверна частина лише розширює протокол відповіді, а клієнтська підключає новий рендер-компонент, не чіпаючи вже реалізовану логіку діалогового чату. Така реалізація модульності дозволяє поступово перетворювати простий чат у повноцінний інтерфейс взаємодії з користувачем, де LLM не тільки зможе писати текст, а й керувати динамічними елементами користувацького інтерфейсу, що можуть бути адаптовані під конкретні задачі бізнесу.

3.5 Розгортання системи

Розроблену інтелектуальну агенто-орієнтовану систему підбору товарів розгорнуто в локальному середовищі розробника. Розгортання передбачало окремий запуск серверної та клієнтської частин, а також використання хмарного сервісу LLM через OpenAI API.

Програмне забезпечення, використане для розгортання:

- середовище розробки Visual Studio Code з розширеннями для Python та JavaScript/React;

- інтерпретатор CPython – Python 3.14;

- серверний фреймворк FastAPI з використанням ASGI-сервера uvicorn у режимі розробки;

- веб-браузери для запуску та тестування на базі Chromium (Google Chrome та Opera GX);

Серверна частина запускалася локально командою запуску FastAPI/uvicorn, клієнтський застосунок – через стандартний development-сервер React (npm run dev).

Апаратне забезпечення, на якому розгорталась система:

- операційна система – Windows 10 Pro x64;
- процесор – AMD Ryzen 5 3550H (4 фізичні ядра, 8 потоків);
- графічний прискорювач – інтегрована AMD Radeon Vega 8;
- оперативна пам’ять – 32 ГБ DDR4.

Вказана конфігурація забезпечує комфортну роботу середовища розробки, швидкий запуск серверної та клієнтської частин, а також стабільне тестування системи. Обчислювальні операції, пов’язані з роботою LLM, виконувались на стороні хмарного сервісу OpenAI, тому вимоги до локальних ресурсів є малими, а використання графічного прискорювача не є необхідним. Необхідно зазначити, що апаратні вимоги до серверної частини збільшуються зі зростанням числа паралельних запитів та навантаженням на систему.

3.6 Тестування функціональності системи

Функціональне тестування системи включає перевірку всіх ключових сценаріїв роботи. Метою тестування є перевірка коректності роботи розробленої інтелектуальної агенто-орієнтованої системи підбору товарів, а також виявлення можливих помилок у клієнтській та серверній частинах і на етапі інтеграції з LLM-сервісом OpenAI. Тестування проводилося методом “чорної скриньки” на основі заздалегідь визначених сценаріїв використання системи кінцевим користувачем та адміністратором.

Тестування включає такі групи перевірок:

- функціональність діалогового інтерфейсу;
- збереження та відновлення історії повідомлень;
- роботу інтерфейсу налаштувань промпту;
- коректність роботи серверної частини та інтеграції з OpenAI API;
- обробку та відображення карток товарів;
- поведінку системи при помилках та у DEMO-режимі.

3.6.1 Тестування діалогового інтерфейсу

Для клієнтської частини було сформовано набір тестових сценаріїв, які перевіряють логіку роботи компонента діалогового вікна.

У першому випадку користувач вводить непорожній текст у поле вводу та натискає кнопку відправки. Отриманий результат зображено на рисунку 3.5 Очікуваний результат:

- повідомлення додається в історію чату з міткою часу, кнопка відправки блокується під час очікування відповіді;
- після отримання відповіді від сервера у чат додається повідомлення асистента;
- поле вводу очищується.

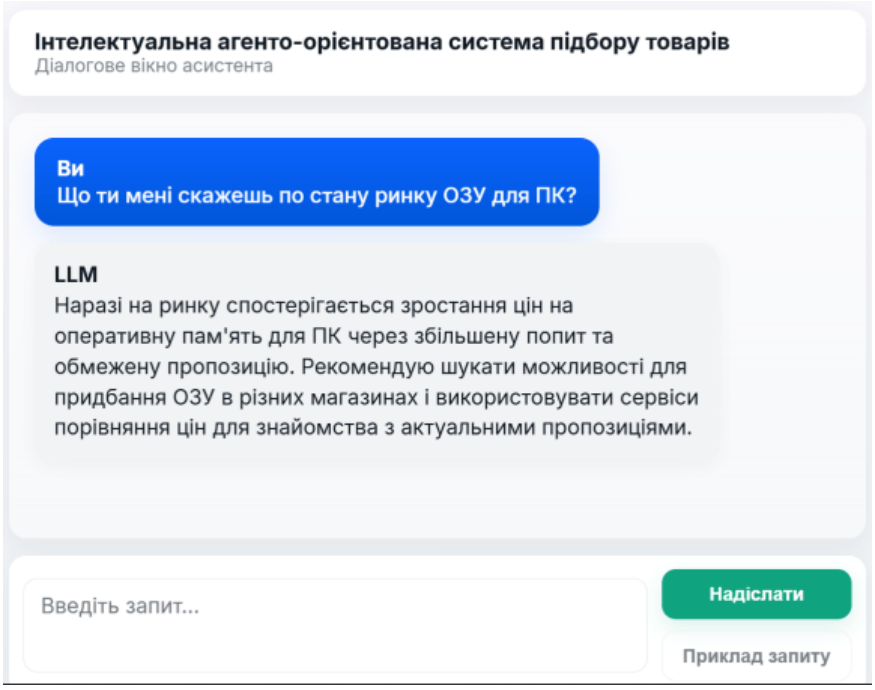

Рисунок 3.5 – Діалогове вікно асистента

У другому випадку, введення порожнього повідомлення. Користувач натискає кнопку відправки з порожнім рядком або рядком, що містить тільки пробіли. Очікуваний результат:

- запит до сервера не надсилається;

- історія повідомлень не змінюється.

Перевірка наявності індикатора завантаження після натискання кнопки відправки. Очікуваний результат:

- під час запиту відображається індикатор завантаження;

- повторна відправка тимчасово блокується;

- після отримання відповіді індикатор зникає.

Усі описані сценарії успішно пройдені та виявлено, що логіка обробки введеного тексту, блокування повторних відправлень і відображення відповідей асистента працює відповідно до розроблених вимог.

3.6.2 Перевірка збереження та відновлення історії чату

Окремо протестовано механізм збереження історії повідомлень у локальному сховищі браузера та її подальшого відновлення. Після виконання кількох запитів до асистента, у діалоговому вікні має створитись кілька пар повідомлень “користувач – асистент”. Очікуваним результатом є:

- поява JSON об’єкта з масивом повідомлень у локальному сховищі під ключем “llm_chat_history_v1”;

- при зміні масиву “messages” дані в сховищі повинні оновлюватись без помилок;

Відновлення діалогу після перезавантаження сторінки. Очікуваний результат:

- чат ініціалізується історією з “localStorage”;

- усі попередні повідомлення зберігаються та відображаються у правильному порядку;

- останнє повідомлення знаходиться в зоні видимості завдяки автоматичному прокручуванню.

При очищення історії, а саме натисканні кнопки очищення чату (виклик функції “clearChatAndStorage”, очікується результат:

- масив “messages” очищується;

- ключ “llm_chat_history_v1” видаляється з локального сховища;

- інтерфейс відображає порожній чат.

У процесі тестування помилок у роботі механізму збереження та відновлення не виявлено. Випадки некоректних або пошкоджених даних у “localStorage” додатково обробляються конструкціями try/catch, що запобігає критичним збоям інтерфейсу.

3.6.3 Тестування інтерфейсу налаштувань LLM-промпту

Важливим є перевірка працездатності інтерфейсу налаштувань промпту системи, який зображено на рисунку 3.6.

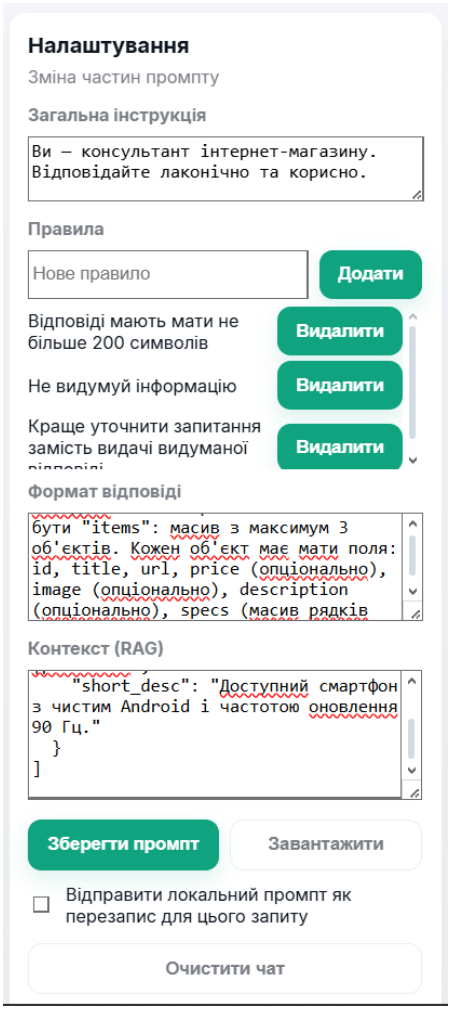

Рисунок 3.6 – Інтерфейс налаштувань промпту

Під час завантаження поточної конфігурації з сервера, а саме відкриття сторінки налаштувань та виклику функцію “loadPromptFromServer”, очікується результат:

- виконується GET-запит до http://127.0.0.1:8000/api/llm/prompt;

- поля “системна роль”, “правила”, “формат” та “контекст” заповнюються значеннями, отриманими із сервера;

- при некоректній відповіді або помилці у консолі виводиться попередження, але робота інтерфейсу не блокується.

Під час редагування та збереження промпту (зміни системної ролі, додавання або видалення правил, зміни контексту та натисканні кнопки збереження) очікується:

- формування коректного JSON-об’єкту “payload” із полями “system_role”, “rules”, “format_constraints”, “context_text”;

- виконання POST-запиту до http://127.0.0.1:8000/api/llm/prompt;

- при успішній відповіді на короткий час відображається індикація успішного збереження;

- у разі помилки користувач отримує повідомлення про неможливість збереження налаштувань.

У результаті тестування підтверджено, що адміністратор може змінювати конфігурацію промпту без втручання у код серверної частини.

3.6.4 Тестування серверної частини та інтеграції з LLM

Для серверної частини, реалізованої на основі FastAPI, перевірено обробку стандартного чат-запиту, роботу у DEMO-режимі та обробку помилок OpenAI.

При надсиланні POST-запиту до /api/llm/chat з коректним текстом повідомлення очікується такий результат:

- сервер логуватиме новий запит;

- формується масив повідомлень з системною роллю та повідомленням користувача;

- викликається клієнт OpenAI з вказаною моделлю;

- відповідь моделі повертається у форматі {"reply": "...", "type": "text"} або у форматі карток.

Перевірка роботи у DEMO-режимі включає запуск сервер без коректного API-ключа та очікування, замість фактичного звернення до OpenAI, виклику функції формування демонстраційної відповіді;

При моделюванні ситуацій з недоступністю OpenAI або некоректною конфігурацією моделі очікується логування помилок та отримання клієнтом повідомлення про помилку в чаті або сервер повертає коректний HTTP-статус для подальшої діагностики.

Результати тестування показали, що серверна частина правильно оброблює як успішні, так і помилкові сценарії, не допускаючи критичних збоїв у роботі клієнтського інтерфейсу.

3.6.5 Тестування обробки карток товарів

Окремий блок тестування присвячено перевірці механізму формування та відображення карток товарів, які повертаються у структурованому форматі JSON об’єкту. Існує необхідність перевирки наявності масиву “items” у відповіді моделі, правильного формату відповіді сервера з картками та відображення карток у клієнтському інтерфейсі.

При надсиланні запиту, який вимагає підбору товарів очікується:

- модель повертає відповідь до сервера, що містить JSON об’єкт з масивом “items”;

- від сервера до клієнта повертається JSON структура {"items": [...], "justification": "...", "type": "cards"};

- до історії повідомлень додається спеціальний запис з типом cards, масивом товарів та текстом-обґрунтуванням;

- інтерфейс відображає горизонтальний список карток з основними характеристиками та посиланнями.

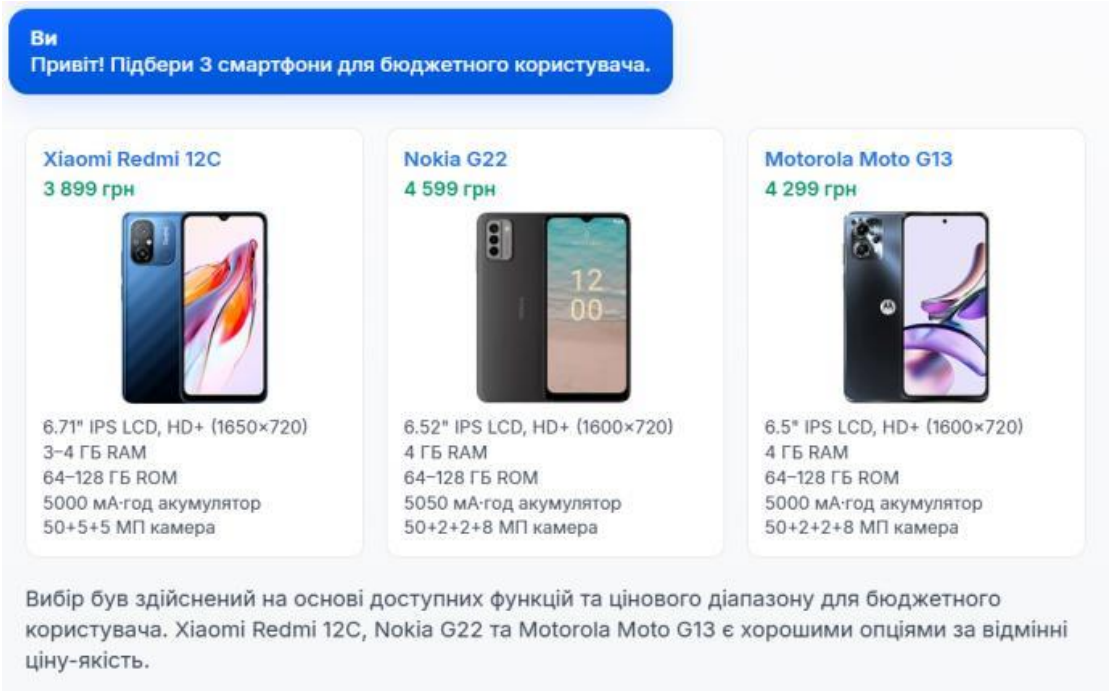

Рисунок 3.7 – Відображення карток з товарами

Результат тестування, який зображено на рисунку 3.7 підтвердило, що система коректно обробляє відповіді LLM у новому форматі, а також залишається сумісною зі звичайними текстовими відповідями.

4 РЕЗУЛЬТАТИ ТА РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ

4.1 Результати тестування

За результатами проведеного тестування створено висновок, що розроблена система в відповідає вимогам, визначеним до функціональності агенто-орієнтованої системи підбору товарів. Діалоговий інтерфейс забезпечує стабільний обмін повідомленнями між користувачем та асистентом. Історія спілкування коректно зберігається та відновлюється з локального сховища браузера. Інтерфейс налаштувань промпту дозволяє адміністраторам гнучко керувати поведінкою LLM без змін у коді. Серверна частина на базі FastAPI стійко працює як у штатному режимі, так і при відмові зовнішнього сервісу.

Механізм формування та відображення карток товарів працює коректно, дані попередньо фільтруються та перевіряються.

Критичних помилок у роботі системи не було виявлено. У консолях серверної та клієнтської частини, які наведені на рисунку 4.1, помилок не виявлено. Ініціалізація компонентів проходить успішно, запуск системи не займає більше одної хвилини.

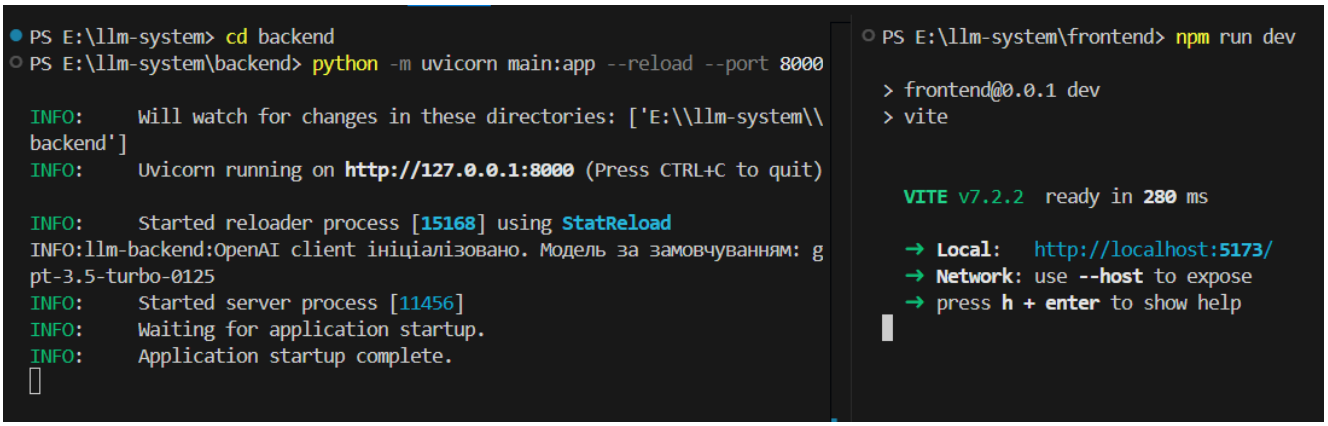

Рисунок 4.1 – Консоль серверної та клієнтської частини системи

Зафіксовані незначні недоліки не є критичними та не впливають на основну функціональність системи, але можуть бути усунуті в процесі інтеграції у інші системи та подальшого супроводу й модернізації.

4.2 Результат розробки системи

Результатом роботи є програмний прототип інтелектуальної агенто-орієнтованої системи підбору товарів для вебсистем електронної комерції. Реалізовано діалоговий AI-асистент, який спілкується з користувачем природною мовою, приймає запити на підбір товарів, уточнення фільтрів та додаткові запитання, а також формує текстові рекомендації й добірки товарів. Діалогове вікно асистента зображено на рисунку 4.2.

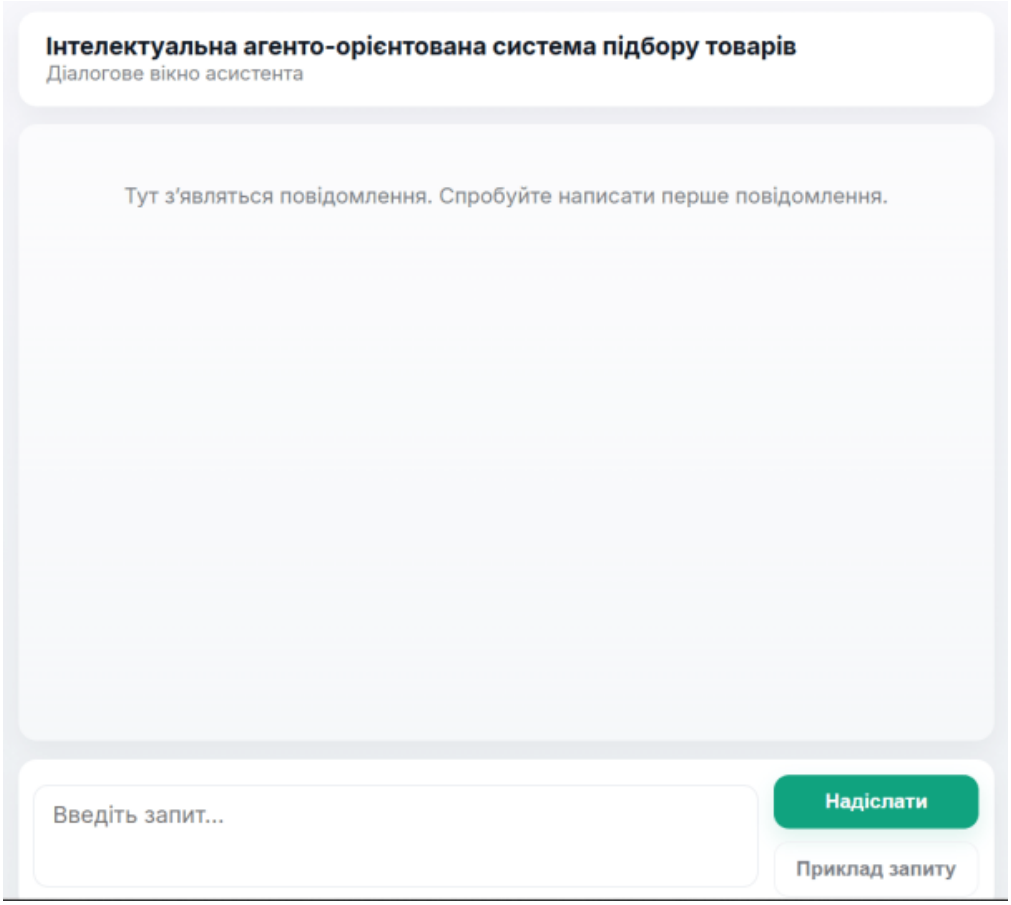

Рисунок 4.2 – Діалоговий інтерфейс AI асистента

Клієнтська частина системи реалізована у вигляді односторінкового вебдодатку на JavaScript з використанням бібліотеки React. Компонентний підхід React спрощує подальший розвиток клієнтської частини. Інтерфейс розділено на окремі незалежні компоненти (чат, список повідомлень, панель налаштувань, картки товарів), які можна повторно використовувати та модифікувати без впливу на інші частини системи.

Забезпечено рендеринг діалогового вікна без перезавантаження сторінки, збереження та відновлення історії спілкування з локального сховища браузера, а також відображення карток товарів з основними характеристиками у відповідь на структуровані відповіді від LLM. Передбачено інтерфейс для редагування промпту, який зображено на рисунку 4.3, що дозволяє адміністраторам змінювати поведінку асистента без модифікації програмного коду.

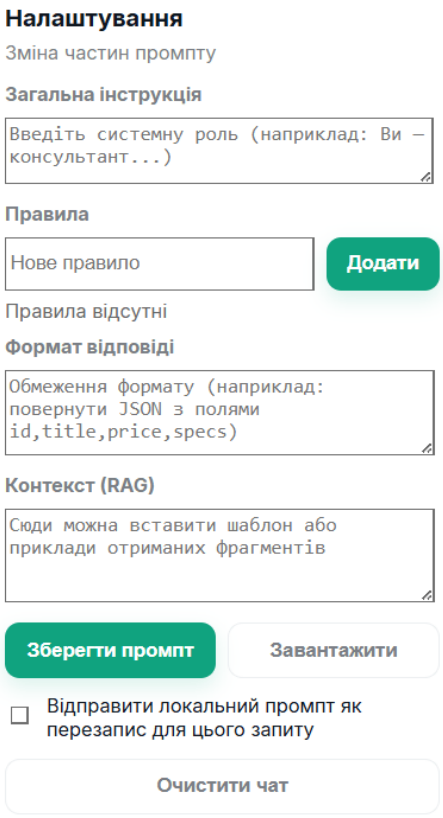

Рисунок 4.3 – Діалоговий інтерфейс AI асистента

Серверна частина побудована на основі фреймворку FastAPI мовою Python і забезпечує інтеграцію з віддаленою LLM через OpenAI API. Реалізовано централізоване керування системним промптом та параметрами запитів, підтримку демонстраційного режиму при відсутності або вичерпанні токенів API-ключа, обробку помилок зовнішнього сервісу без критичних збоїв для клієнтського інтерфейсу.

Функціональне тестування підтвердило коректність роботи основних сценаріїв – обмін повідомленнями між користувачем та асистентом, збереження та відновлення історії діалогу, робота інтерфейсу налаштувань промпту, стабільність серверної частини у штатному та DEMO-режимах, а також формування й відображення карток товарів. Критичних помилок не виявлено, виявлені незначні недоліки можуть бути усунені в процесі подальшого розвитку системи.

4.3 Рекомендації щодо застосування системи

Розроблений прототип доцільно застосовувати як модуль AI-асистента у складі існуючих інтернет-магазинів та інших вебсистем електронної комерції. Система може бути інтегрована у вебсистеми магазинів як окреме діалогове вікно, що допомагає користувачеві сформулювати вимоги, підібрати товари за кількома критеріями, порівняти альтернативи та отримати пояснення щодо рекомендацій.

З технічної точки зору рекомендовано розгортати серверну частину на окремому сервісному вузлі, що матиме доступ до OpenAI API та внутрішніх служб інтернет-магазину. Клієнтський компонент можна підключати як окремий віджет або React-компонент, який взаємодіє з існуючим бекендом через HTTP/HTTPS протокол. Для зниження витрат на використання LLM і підвищення стабільності роботи варто передбачити кешування типових відповідей, використання DEMO-режиму для тестування та обмеження довжини користувацьких запитів.

З точки зору бізнесу система рекомендується для магазинів з розгалуженим асортиментом, де користувачам складно самостійно знайти потрібні товари стандартними засобами фільтрації. Подальший розвиток рішення може включати інтеграцію з реальною базою даних каталогу, використання ембеддінгів і RAG-підходів для підбору товарів за семантичним змістом запитів, розширення набору типів відповідей та підключення аналітики для оцінки впливу асистента на конверсію продажів.

ВИСНОВКИ

Результатом дипломної роботи є розроблений програмний прототип інтелектуальної агенто-орієнтованої системи підбору товарів для вебсистем електронної комерції, який відповідає поставленій меті та задачам проєкту. Система реалізує діалоговий AI-асистент, що допомагає користувачам формувати запити природною мовою та отримувати персоналізовані підбори товарів у зручному форматі.

Актуальність розробленої системи зумовлена стрімким розвитком онлайн-торгівлі, зростанням обсягів товарних каталогів і потребою в персоналізованому сервісі. На ринку бракує доступних рішень, які поєднують можливості великих мовних моделей з вебінтерфейсом інтернет-магазину та орієнтовані на інтеграцію у вже існуючі бізнес-системи.

Перевагами системи є використання LLM для гнучкої обробки запитів користувача, можливість масштабування та подальшого розширення функціональності, а також сервісно-орієнтована архітектура, що спрощує інтеграцію з іншими компонентами вебплатформи. Розроблений вебінтерфейс має зрозумілий та зручний дизайн, підтримує збереження історії діалогу та відображення рекомендацій у вигляді карток товарів з поясненнями, що полегшує взаємодію користувача з системою.

У підсумку, інтелектуальна агенто-орієнтована система підбору товарів відповідає вимогам автоматизації процесу підбору та рекомендації товарів в інтернет-магазині. Впровадження такого AI-асистента може підвищити ефективність взаємодії користувачів із каталогом, покращити якість рекомендацій і сприяти зростанню задоволеності клієнтів та показників бізнесу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Тягунова М.Ю. Аналіз асистентів на базі штучного інтелекту для вебсистем / М.Ю. Тягунова, Д.В. Виниченко // 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. 69-70 - 1 електрон. опт. диск (DVD-ROM). - назва з тит. Екрана;

2. Google for Developers: Collaborative filtering. URL: https://developers.google.com/machine-learning/recommendation/collaborative/basics (date of access: 21.11.2025);

3. Google for Developers: Content-based filtering . URL: https://developers.google.com/machine-learning/recommendation/content-based/basics (date of access: 21.11.2025);

4 IBM: What is AI Agent Orchestration? URL: https://www.ibm.com/think/topics/ai-agent-orchestration (date of access: 22.11.2025);

5. Cornell Univercity: Knowledge-Enhanced Retrieval-Augmented Generation for Recommendation. URL: https://arxiv.org/abs/2507.05863 (date of access: 22.11.2025);

6. Cornell Univercity: An LLM Approach to Course Recommendations Using Natural Language Queries. URL: https://arxiv.org/html/2412.19312v2 (date of access: 22.11.2025);

7. PromptHub: A Prompt Engineering Framework for LLM Recommendations. URL: https://www.prompthub.us/blog/recprompt-a-prompt-engineering-framework-for-llm-recommendations (date of access: 23.11.2025);

8. Amazon: Amazon's AI shopping assistant gets smarter and more personal. URL: https://www.aboutamazon.com/news/retail/amazon-rufus-ai-assistant-personalized-shopping-features (date of access: 25.11.2025);

9. The technology behind Amazon’s GenAI-powered shopping assistant, Rufus - Amazon Science. URL: https://www.amazon.science/blog/the-technology-behind-amazons-genai-powered-shopping-assistant-rufus (date of access: 25.11.2025);

10. 0DIN: Secures the Future of AI Shopping. URL: https://0din.ai/blog/odin-secures-the-future-of-ai-shopping (date of access: 25.11.2025);

11. Google: The new Google Shopping is rebuilt with AI. URL: https://blog.google/products/shopping/google-shopping-ai-update-october-2024/ (date of access: 25.11.2025);

12. Shopify: AI for Commerce. URL: https://www.shopify.com/magic (date of access: 26.11.2025);

13. GoIT Global: Must-have AI-інструменти для малого бізнесу. URL: https://goit.global/ua/blog/ai-dlia-malogo-biznesy/ (дата звернення: 26.11.2025);

14. Amazon: Amazon Personalize. URL: https://aws.amazon.com/personalize/ (date of access: 27.11.2025);

15. The Page: Фінансова компанія Klarna хоче замінити 2 тисячі працівників ІІІІ. URL: https://thepage.ua/ua/news/finansova-kompaniya-klarna-hoche-zaminiti-2-tisyachi-pracivnikiv-shi (дата звернення: 27.11.2025);

16. eWeek: Best AI Tools for Holiday Shopping Deals. URL: https://www.eweek.com/news/news-best-ai-tools-for-holiday-bargain-shopping/ (date of access: 27.11.2025);

17. Social Media Today: TikTok Adds a Dedicated AI Assistant for In-App Sellers. URL: https://www.socialmediatoday.com/news/tiktok-adds-seller-assistant-ai-chatbot/748785/ (date of access: 27.11.2025).

ДОДАТОК А
ЛІСТИНГИ ФРАГМЕНТІВ КОДУ

Лістинг А.1 – Код елементів інтерфейсу діалогового вікна

```
import React, { useState, useRef, useEffect } from "react";
import "../style.css";
export default function LLMTestPage() {
  const [input, setInput] = useState("");
  const [messages, setMessages] = useState(() => {
    try {
      const raw = localStorage.getItem("llm_chat_history_v1");
      return raw ? JSON.parse(raw) : [];
    } catch (err) {
      console.error("Error parsing chat from localStorage:",
err);
      return [];
    }
  });
  const [dbData, setDbData] = useState(null);
  const [evaluation, setEvaluation] = useState("");
  const [loading, setLoading] = useState(false);
  const messagesRef = useRef(null);

  const CHAT_KEY = "llm_chat_history_v1";

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

  useEffect(() => {
    saveChatToLocalStorage(messages);
    messagesRef.current?.scrollIntoView({ behavior: "smooth",
block: "end" });
  }, [messages]);
  const handleKeyDown = (e) => {
    if (e.key === "Enter" && !e.shiftKey) {
      e.preventDefault();
      handleSend();
    }
  };
      <section className="chat-panel">
        <header className="chat-header">
          <div>
            <div className="title">Інтелектуальна агенто-
орієнтована система підбору товарів</div>
              <div className="sub">Діалогове вікно
асистента</div>
          </div>
        </header>
```

```
            <div className="messages" role="log" aria-
live="polite">
              {messages.length === 0 && (
                <div className="placeholder">Тут з’являться
повідомлення. Спробуйте написати перше повідомлення.</div>
              )}

              {messages.map((m, i) => (
                <div className="message-row" key={i}>
                  <div className={`message ${m.role === "user" ?
"user" : "assistant"}`}>
                    <div><strong>{m.role === "user" ? "Ви" :
"LLM"}</strong></div>
                    <div>{m.content}</div>
                  </div>
                </div>
              ))}
              <div ref={messagesRef} />
            </div>
            <div>
              <div className="input-area">
                <textarea
                  placeholder="Введіть запит..."
                  value={input}
                  onChange={(e) => setInput(e.target.value)}
                  onKeyDown={handleKeyDown}
                  rows={2}
                />
                <div className="input-actions">
                  <button className="btn minwide"
onClick={handleSend} disabled={loading}>
                    {loading ? "Відправка..." : "Надіслати"}
                  </button>
                  <button className="btn secondary minwide"
onClick={() =>
                    setInput("Підбери 3 смартфони для бюджетного
користувача.")
                  }>
                    Приклад запиту
                  </button>
                </div>
              </div>
            </div>
          </section>
        </div>
      );
    }
```

Лістинг А.2 – Код елементів форми редагування промпту

```
<textarea
  rows={3}
  value={promptSystemRole}
```

```
        onChange={(e) => setPromptSystemRole(e.target.value)}
        placeholder="Введіть системну роль (наприклад: Ви -
консультант...)"
      />
      <input
        value={ruleInput}
        onChange={(e) => setRuleInput(e.target.value)}
        placeholder="Нове правило"
      />
      <button className="btn" onClick={addRule}>Додати</button>
      {/* mapped rules with remove button */}
      {promptRules.map((r, idx) => (
        <div key={idx}>
          <div>{r}</div>
          <button onClick={() => removeRule(idx)}>Видалити</button>
        </div>
      ))}
      <textarea rows={3} value={promptFormat} onChange={(e) =>
setPromptFormat(e.target.value)} />
      <textarea rows={4} value={promptContextText} onChange={(e) =>
setPromptContextText(e.target.value)} />
      <div>
        <button className="btn full"
onClick={savePromptToServer}>Зберегти промпт</button>
        <button className="btn secondary full"
onClick={loadPromptFromServer}>Завантажити</button>
      </div>
      {promptSaved && <div style={{ color: "green"
}}>Збережено</div>}
```

Лістинг А.3 – Функція розбору відповіді від LLM

```
def try_extract_items_from_reply(reply_text: str) ->
Tuple[Optional[List[Dict[str, Any]]], str]:
    if not reply_text:
        return None, ""
    reply_text = reply_text.strip()
    fence_re = re.compile(r"```(?:json)?\s*([\s\S]*?)\s*```",
re.IGNORECASE)
    m = fence_re.search(reply_text)
    if m:
        block = m.group(1).strip()
        after = reply_text[m.end():].strip()
    else:
        block = reply_text
        after = ""
    def try_load(s: str):
        try:
            return json.loads(s)
        except Exception:
            return None
    candidate = block
    first_brace = candidate.find("{")
```

```
    first_bracket = candidate.find("[")
    starts = [p for p in (first_brace, first_bracket) if p != -1]
    if starts:
        start = min(starts)
        last_brace = candidate.rfind("}")
        last_bracket = candidate.rfind("]")
        ends = [p for p in (last_brace, last_bracket) if p != -1 and
p > start]
        if ends:
            end = max(ends)
            candidate = candidate[start:end+1]
    candidate_fixed = re.sub(r",\s*(?=[}\]])", "", candidate)
    parsed = try_load(candidate_fixed)
    if parsed is None:
        parsed = try_load(candidate_fixed.replace("'", '"'))
    if parsed is not None:
        if isinstance(parsed, dict) and
isinstance(parsed.get("items"), list):
            items = parsed["items"]
            justification = parsed.get("text", "") or after or ""
            return items, justification
        if isinstance(parsed, list):
            return parsed, after or ""
        if isinstance(parsed, dict):
            best = None
            for v in parsed.values():
                if isinstance(v, list) and all(isinstance(x, dict)
for x in v):
                    if best is None or len(v) > len(best):
                        best = v
            if best:
                justification = parsed.get("text", "") or after or
""
                return best, justification
    return No
```

ДОДАТОК Б
СЛАЙДИ ПРЕЗЕНТАЦІЇ

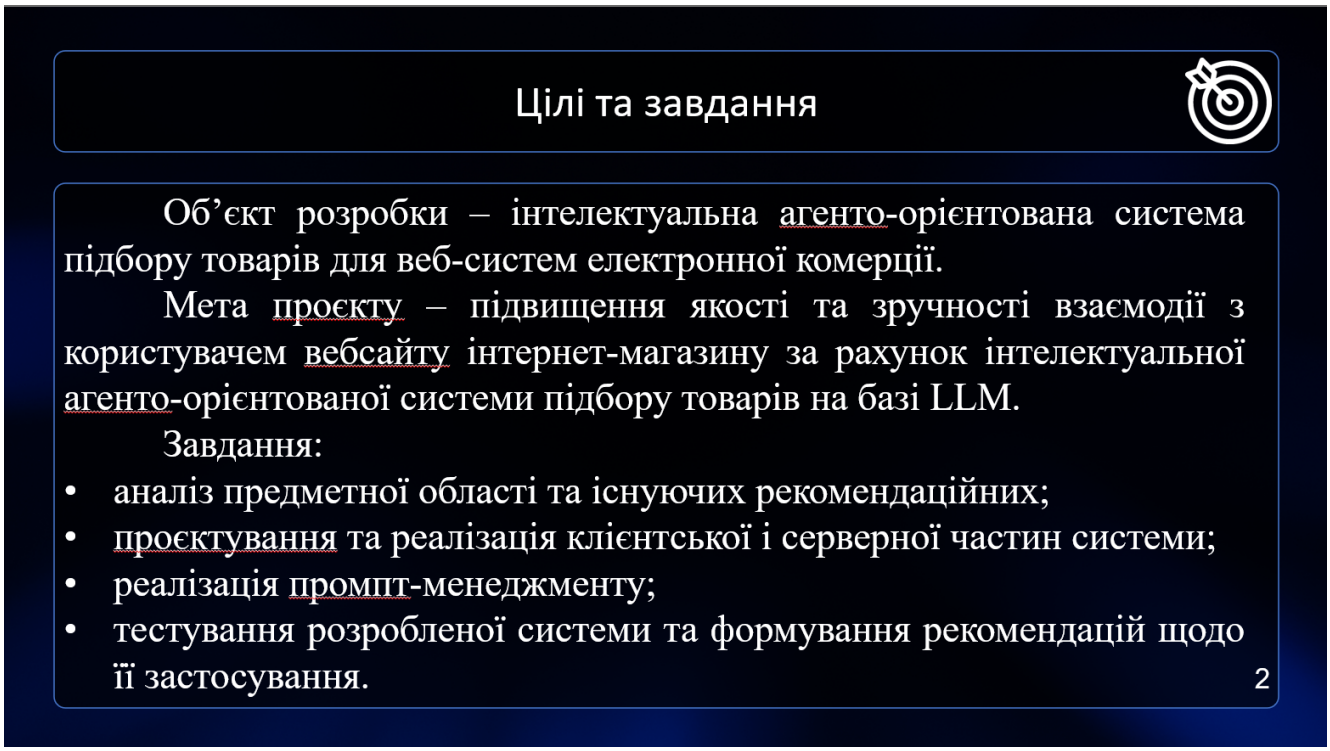

Рисунок Б.1 – Цілі та завдання

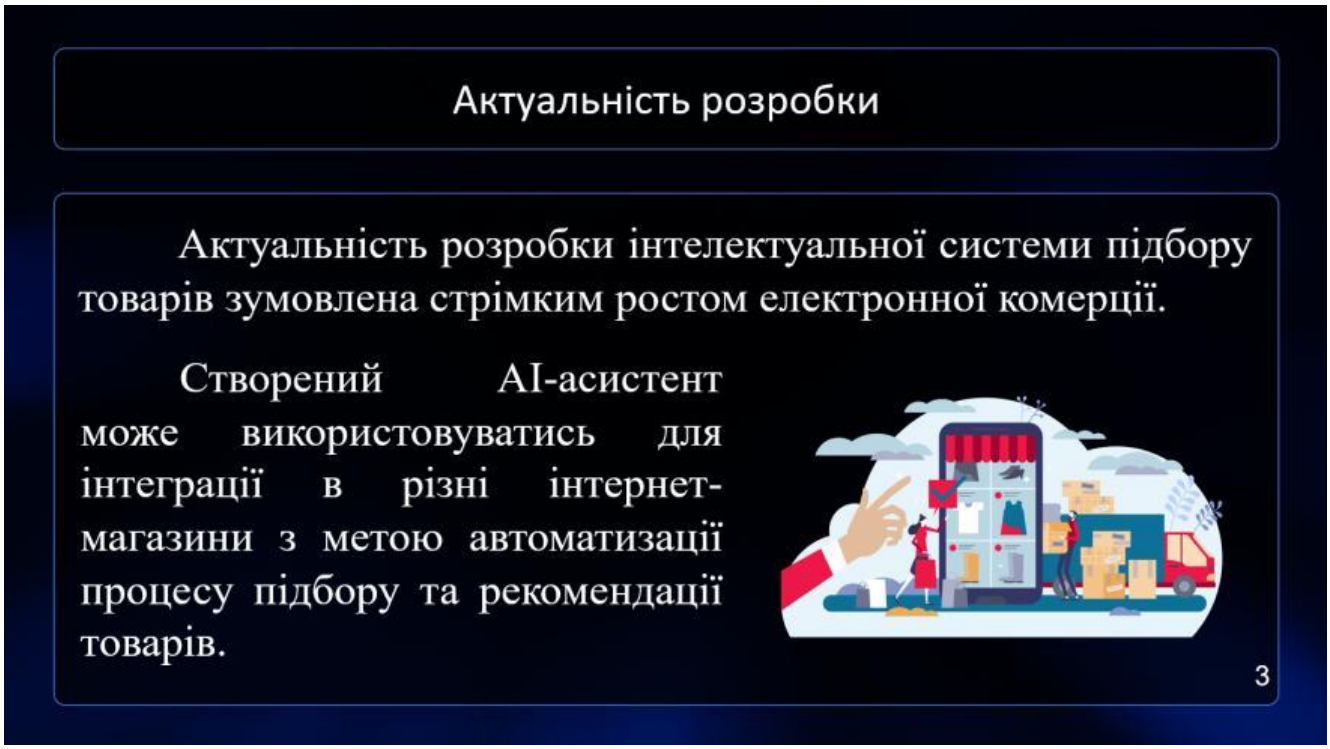

Рисунок Б.2 – Актуальність розробки

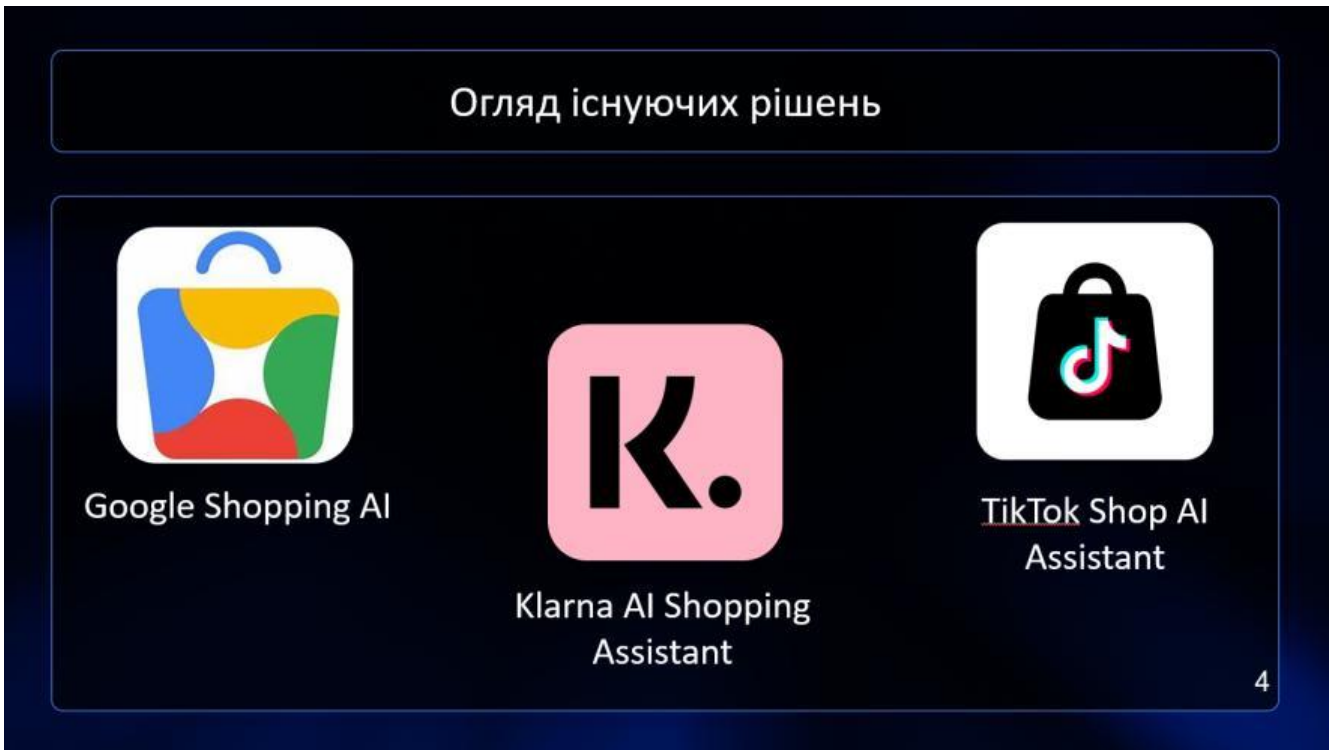

Рисунок Б.3 – Огляд існуючих рішень

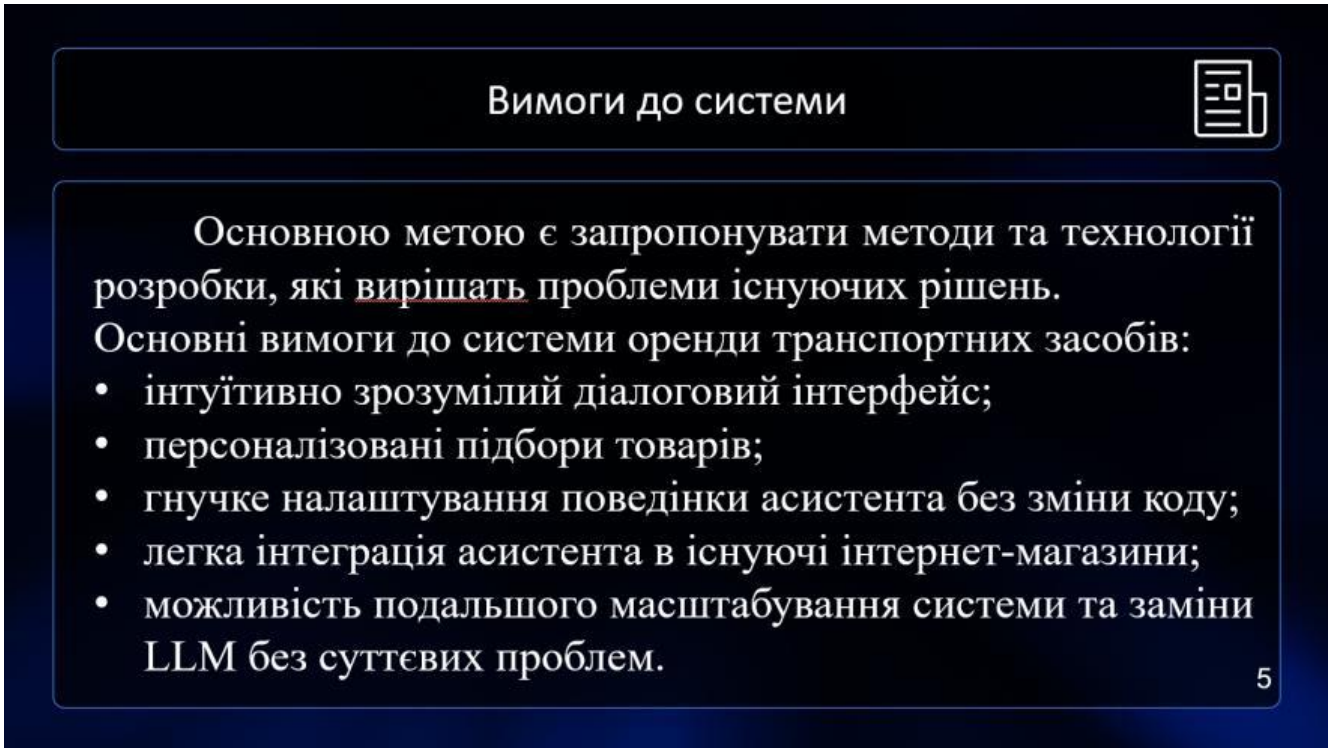

Рисунок Б.4 – Вимоги до системи

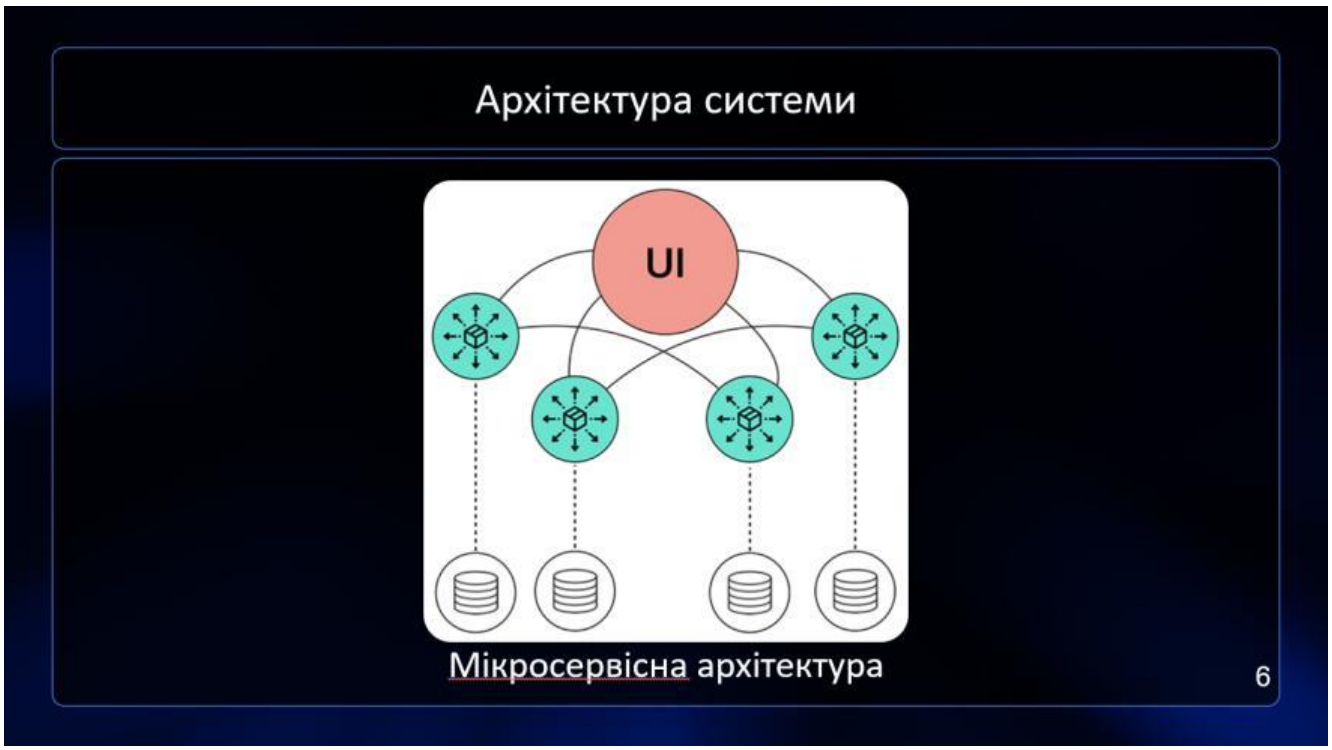

Рисунок Б.5 – Архітектура системи

Рисунок Б.6 – Технології розробки клієнтської частини

Рисунок Б.7 – Переваги React JS

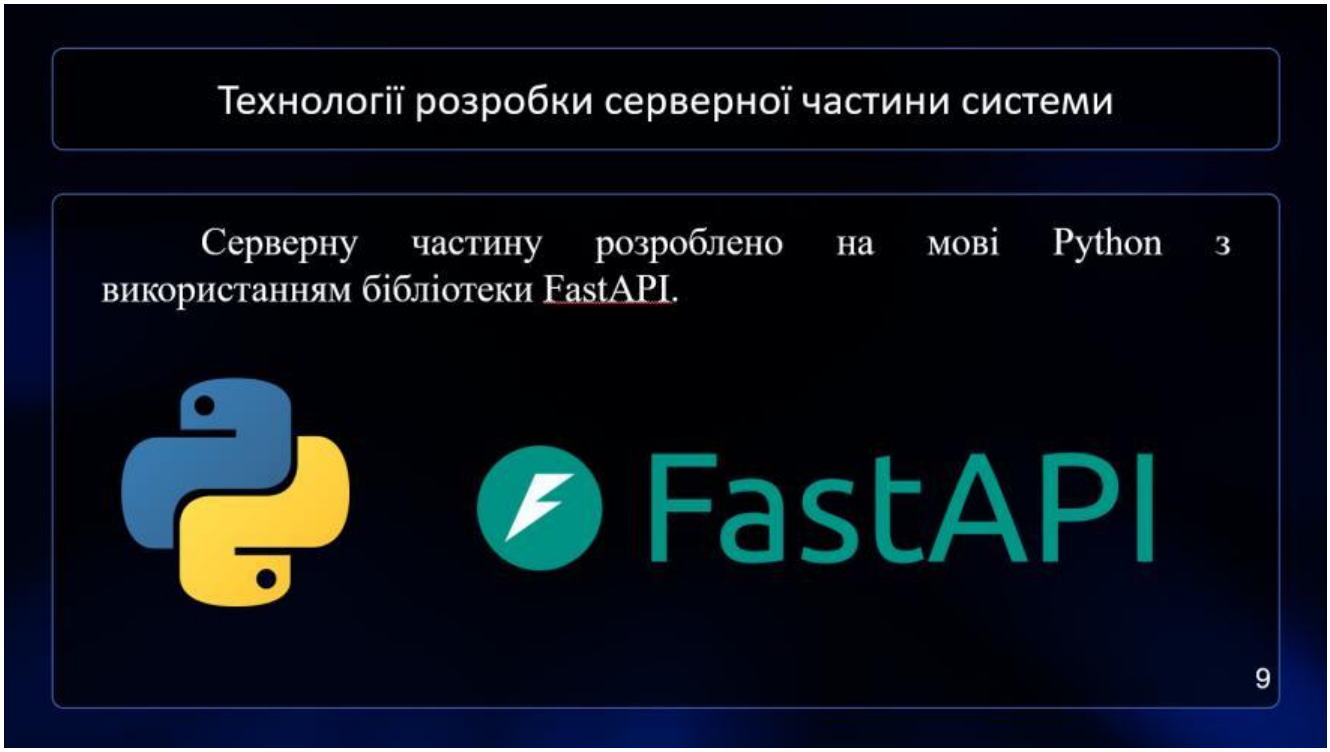

Рисунок Б.8 – Технології розробки серверної частини

Рисунок Б.9 – Переваги застосування FastAPI

Рисунок Б.10 – Велика мовна модель ChatGPT

Рисунок Б.11 – Складові промпту

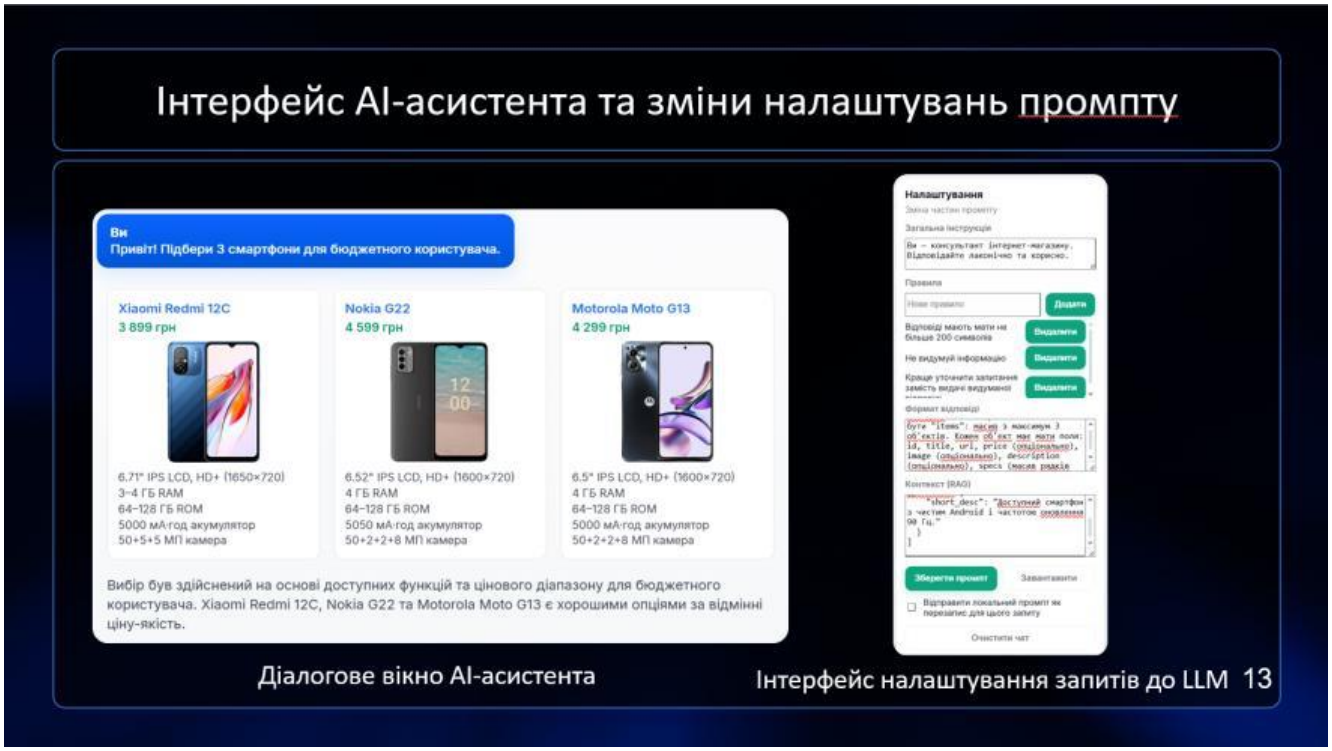

Рисунок Б.12 – Інтерфейс AI-асистента та налаштувань промпту

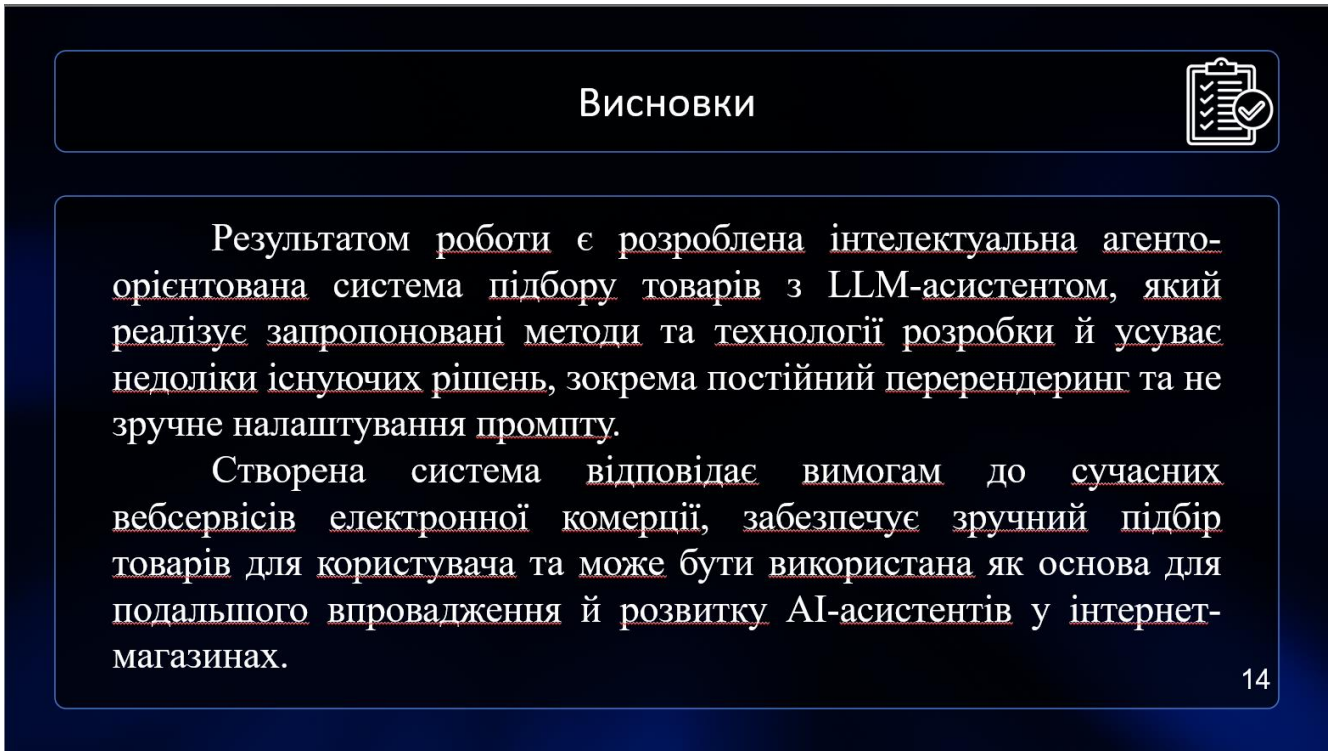

Рисунок Б.13 – Висновки