МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
Циклова комісія *комп'ютерних систем та мереж*
(повна назва циклової комісії)

Допустити до захисту
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
_____ *Ірина КРАВЧУК*
(підпис) (ім'я, ПРІЗВИЩЕ)
« 10 » 06 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬО-ПРОФЕСІЙНОГО СТУПЕНЯ
ФАХОВИЙ МОЛОДШИЙ БАКАЛАВР

Тема: *Інтеграція штучного інтелекту в веб-застосунок для аналізу даних*

Група: *3-013* Спеціальність: *123 «Комп'ютерна інженерія»*

Здобувач освіти _____ *Сергій ШОКОТЬКО*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Олександр МИТРОФАНОВ*
(підпис) (ім'я, ПРІЗВИЩЕ)

Консультант з оформлення
пояснювальної записки _____ *Оксана ОСАДЧА*
(підпис) (ім'я, ПРІЗВИЩЕ)

Кривий Ріг 2025 р.

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Відділення *комп'ютерної та програмної інженерії*
Циклова комісія *комп'ютерних систем та мереж*
Освітньо-професійний ступінь *фаховий молодший бакалавр*
Спеціальність *123 «Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
_____ *Ірина КРАВЧУК*
(підпис) (ім'я, ПРІЗВИЩЕ)
« *10* » *03* 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ОСВІТИ

ШОКОТЬКО Сергій Русланович
(прізвище, ім'я, по батькові)

1.Тема роботи *Інтеграція штучного інтелекту в веб-застосунок для аналізу даних*

Керівник роботи *МИТРОФАНОВ Олександр Вячеславович, доктор філософії*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по коледжу від « *04* » *04* 2025 року № *50-ст*

2. Строк подання здобувачем освіти роботи з *19.05.2025* по *13.06.2025*

3. Вихідні дані до роботи *Розробка веб-застосуноку з штучним інтелектом для аналізу даних*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Титульний аркуш, реферат, зміст, перелік умовних позначень, вступ, огляд проблематики та обґрунтування рішень, розробка веб-застосунку, тестування та демонстрація роботи, висновки, список використаних джерел, додаток.

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

Презентація Microsoft PowerPoint

6. Консультанти розділів роботи (проекту)

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

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Узгодження технічного завдання з керівником кваліфікаційної роботи*	*17.03.2025-21.03.2025*	*виконано*
2	*Обґрунтування вибору програмних засобів*	*24.03.2025-28.03.2025*	*виконано*
3	*Розробка структури та архітектури програми*	*12.04.2025-10.05.2025*	*виконано*
4	*Тестування та усунення помилок у програмі*	*12.05.2025-14.05.2025*	*виконано*
5	*Оформлення пояснювальної записки*	*16.05.2025-30.05.2025*	*виконано*
6	*Перевірка на плагіат пояснювальної записки*	*08.06.2025-11.06.2025*	*виконано*
7	*Захист кваліфікаційної роботи*		

Здобувач освіти _____ *Сергій Шокотько*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Олександр Митрофанов*
(підпис) (ім'я, ПРІЗВИЩЕ)

StrikePlagiarism.com

Дата звіту 6/11/2025
Дата редагування ---

Документ прийнятий

Звіт подібності

метадані

Назва організації
Ukrainian national aviation university
Заголовок
КПІ_2025_Шокотько
Автор Науковий керівник / Експерт
ШокотькоМитрофанов О.
підрозділ
Криворізький Фаховий коледж

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

КП 1

КЦ

25	8260	67941
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		0
Інтервали	A→	0
Мікропробіли	·	71
Білі знаки		0
Парафрази (SmartMarks)	a	9

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://www.ela.kpi.ua/bitstream/123456789/59827/1/Tataryn_bakalavr.docx	18 0.22 %
2	https://lepiku.medium.com/why-docker-3a670a7918c4	16 0.19 %
3	https://lepiku.medium.com/why-docker-3a670a7918c4	16 0.19 %

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Інтеграція штучного інтелекту в веб-застосунок для аналізу даних» містить: 50 сторінок основного тексту, 22 рисунки, 18 використаних джерел, 1 додаток.

AI-АНАЛІТИК, АНАЛІЗ ДАНИХ, ШТУЧНИЙ ІНТЕЛЕКТ, ЧАТ-БОТ, *OPENAI API*, *GPT*-4.1, *REACT*, *NODE.JS*, *EXPRESS.JS*, *DOCKER*, ВЕБ-ЗАСТОСУНОК, АНАЛІЗ ФАЙЛІВ, *CSV*, *JSON*, *TXT*.

У даній кваліфікаційній роботі розробляється веб-застосунок «*DataMind*», призначений для інтерактивного аналізу даних за допомогою *AI*-агента. Система надає користувацький інтерфейс для спілкування з *AI*, завантаження файлів даних (*CSV*, *JSON*, *TXT*) та отримання структурованих аналітичних відповідей.

Метою роботи є розробка повнофункціонального веб-застосунку, що спрощує процес аналізу даних для широкого кола користувачів шляхом інтеграції потужних моделей штучного інтелекту у зручний чат-інтерфейс, забезпечуючи обробку завантажених файлів та генерацію змістовних, форматованих відповідей.

Основними результатами роботи є: спроєктований та реалізований веб-застосунок «*DataMind*» з клієнт-серверною архітектурою; розроблена фронтенд частина на *React*, що забезпечує інтуїтивну взаємодію, завантаження файлів та вибір *AI*-моделей; створена бекенд частина на *Node.js*/*Express*, що обробляє запити, інтегрується з *OpenAI API*, керує системними інструкціями для *AI* та забезпечує генерацію відповідей у строго визначеному *JSON*-форматі; впроваджена система контейнеризації за допомогою *Docker* та *Docker Compose* для спрощення розгортання та забезпечення консистентності середовища. Практична значущість полягає у наданні доступного інструменту для *AI*-асистованого аналізу даних, що дозволяє користувачам без глибоких технічних навичок отримувати цінні інсайти з власних даних, тим самим підвищуючи ефективність прийняття рішень.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

API (англ. *Application Programming Interface*) – програмний інтерфейс застосунку, набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення. У проєкті використовується для взаємодії фронтенду з бекендом та бекенду з сервісами *OpenAI*.

Backend – серверна частина застосунку, відповідальна за обробку запитів, бізнес-логіку та взаємодію з зовнішніми сервісами. У проєкті реалізована на *Node.js* та *Express*.

CSS (англ. *Cascading Style Sheets*) – каскадні таблиці стилів, мова для опису зовнішнього вигляду веб-сторінок, написаних за допомогою *HTML* або *XML*. Використовується для стилізації компонентів фронтенду.

CSV (англ. *Comma-Separated Values*) – текстовий формат, призначений для представлення табличних даних, де кожний рядок відповідає одному рядку таблиці, а значення окремих колонок розділяються комами. Підтримується для завантаження файлів.

Frontend – клієнтська частина застосунку, з якою безпосередньо взаємодіє користувач. У проєкті реалізована на *React* з використанням *Vite*.

GPT (англ. *Generative Pre-trained Transformer*) – велика мовна модель від *OpenAI*, що використовується в проєкті для генерації відповідей *AI*-агента.

HTML (англ. *HyperText Markup Language*) – стандартизована мова розмітки документів для перегляду веб-сторінок у браузері.

HTTP (англ. *Hypertext Transfer Protocol*) – протокол передачі даних, що використовується для обміну інформацією в Інтернеті, зокрема між клієнтом (браузером) та сервером.

JSON (англ. *JavaScript Object Notation*) – текстовий формат обміну даними, заснований на *JavaScript*, що легко читається людьми та обробляється комп’ютерами. Використовується для передачі даних між фронтендом та бекендом, а також для відповідей *OpenAI API*.

Markdown – полегшена мова розмітки, що дозволяє форматувати текст за допомогою простого синтаксису. Використовується для відображення відповідей *AI*-агента на фронтенді.

Node.js – програмна платформа, заснована на рушії *V*8 (що транслює *JavaScript* в машинний код), яка перетворює *JavaScript* з вузькоспеціалізованої мови на мову загального призначення. Використовується для розробки бекенду.

NPM (англ. *Node Package Manager*) – менеджер пакунків для *Node.js*, що використовується для встановлення та управління залежностями проєкту.

OpenAI API – програмний інтерфейс, що надається компанією *OpenAI* для доступу до її моделей штучного інтелекту, зокрема *GPT*. Використовується бекендом для взаємодії з моделями *GPT*-4.1.

React – *JavaScript*-бібліотека для створення користувацьких інтерфейсів, зокрема односторінкових застосунків. Основа фронтенд-частини проєкту.

UI (англ. *User Interface*) – користувацький інтерфейс, сукупність засобів, за допомогою яких користувач взаємодіє з програмою або пристроєм. У проєкті представлений веб-інтерфейсом чат-бота.

Vite – інструмент для збирання фронтенд-застосунків, який забезпечує швидкий запуск сервера для розробки та оптимізовану збірку для продакшену. Використовується для фронтенду на *React*.

ВСТУП

Сучасний світ характеризується експоненційним зростанням обсягів даних, що генеруються в усіх сферах людської діяльності. Ефективний аналіз цих даних є критично важливим для прийняття обґрунтованих рішень, проте традиційні методи обробки інформації часто виявляються недостатньо потужними, вимагають значних часових витрат та глибоких технічних знань від користувачів. Це створює бар'єр між накопиченими даними та їх практичним застосуванням, особливо для фахівців, які не є експертами в галузі програмування чи статистики. У відповідь на ці виклики, штучний інтелект, зокрема великі мовні моделі, відкриває революційні можливості для автоматизації аналізу даних, пропонуючи інтуїтивно зрозумілі способи взаємодії. Актуальність теми даної кваліфікаційної роботи зумовлена зростаючою потребою в інтелектуальних системах, здатних перетворювати складні дані на зрозумілі висновки та рекомендації через доступні веб-інтерфейси.

Розробка веб-застосунку «*DataMind*» відповідає загальносвітовим тенденціям цифровізації, поширення технологій штучного інтелекту та розвитку інструментів для роботи з великими даними.

Метою даної кваліфікаційної роботи є розробка веб-застосунку «*DataMind*», що надає користувачам можливість виконувати аналіз даних через інтерактивну взаємодію з *AI*-агентом, який підтримує завантаження файлів різних форматів та забезпечує отримання структурованих та змістовних відповідей.

Для досягнення поставленої мети було визначено наступні завдання:

– Провести аналіз проблеми аналізу даних, існуючих підходів та визначити можливості застосування штучного інтелекту для її вирішення.

– Сформулювати основні функціональні вимоги до веб-застосунку та обґрунтувати вибір технологічного стеку для його реалізації.

– Спроєктувати архітектуру веб-застосунку «*DataMind*», що включає клієнтську (*frontend*) та серверну (*backend*) частини, а також механізм їх взаємодії та інтеграції з *OpenAI API*.

– Розробити *frontend* частину застосунку з використанням бібліотеки *React*, забезпечивши користувацький інтерфейс для чату, завантаження файлів (*CSV*, *JSON*, *TXT*), вибору *AI*-моделі та відображення результатів аналізу з підтримкою *Markdown*.

– Розробити *backend* частину застосунку на платформі *Node.js* з фреймворком *Express*, реалізувавши *API* для обробки запитів, взаємодії з *OpenAI API*, включаючи формування системних інструкцій, обробку файлів та генерацію відповідей у визначеному *JSON*-форматі.

– Реалізувати систему контейнеризації застосунку за допомогою *Docker*.

– Провести тестування ключових функціональних можливостей розробленого веб-застосунку для перевірки його працездатності та відповідності вимогам.

Об’єктом дослідження є процеси аналізу даних та взаємодії користувачів з *AI*-агентом для отримання аналітичних висновків у веб-середовищі.

Предметом дослідження є архітектура, технологічний стек, алгоритми взаємодії з *AI* та програмні засоби, що використовуються для розробки веб-застосунку «*DataMind*».

РОЗДІЛ 1
ОГЛЯД ПРОБЛЕМАТИКИ ТА ОБҐРУНТУВАННЯ РІШЕНЬ

1.1 Проблема аналізу даних та можливості застосування ШІ

Сучасний світ характеризується експоненційним зростанням обсягів даних, які генеруються щосекунди в усіх сферах людської діяльності. Від соціальних мереж і електронної комерції до наукових досліджень і промислового виробництва. Традиційні методи обробки даних, які базуються на статичних алгоритмах і лінійному аналізі, виявляються недостатньо потужними для роботи з такими обсягами інформації та не здатні виявляти складні нелінійні залежності, приховані патерни та тенденції.

Основна проблема полягає в тому, що класичні інструменти аналізу даних вимагають від користувача глибоких знань у галузі статистики, програмування та специфічних методів обробки інформації. Більшість бізнес-аналітиків, дослідників та фахівців різних галузей не володіють достатнім рівнем технічних компетенцій для ефективного використання складних аналітичних платформ. Це створює бар'єр між даними та їх практичним застосуванням, призводячи до ситуацій, коли цінна інформація залишається невикористаною через складність її обробки.

Додатковою проблемою є часові обмеження, з якими стикаються фахівці при проведенні аналізу. Традиційні методи вимагають значних часових витрат на підготовку даних, вибір відповідних статистичних методів, налаштування параметрів аналізу та інтерпретацію результатів. У динамічному бізнес-середовищі, де рішення потрібно приймати швидко, такі тривалі процедури можуть призвести до втрати конкурентних переваг або пропуску важливих можливостей.

Штучний інтелект революціонізує підходи до аналізу даних, пропонуючи принципово нові можливості для вирішення зазначених проблем. Сучасні методи машинного навчання здатні автоматично виявляти складні закономірності в даних без необхідності попереднього визначення конкретних гіпотез або моделей.

Одним з найбільш перспективних напрямків є застосування великих мовних моделей для аналізу даних. Ці моделі, навчені на величезних обсягах текстової

інформації, демонструють здатність до розуміння контексту, логічного мислення та генерації змістовних пояснень. У контексті аналізу даних вони можуть виступати як інтелектуальні асистенти, які не тільки проводять обчислення, але й надають зрозумілі пояснення результатів та пропонують рекомендації щодо подальших дій.

Потужність ШІ особливо проявляється в автоматизації складних аналітичних процесів. Замість того, щоб вимагати від користувача знання специфічних статистичних методів, системи на основі ШІ можуть самостійно вибирати найбільш підходящі алгоритми, налаштовувати їх параметри та адаптуватися до особливостей конкретного набору даних. Це робить потужні аналітичні інструменти доступними для широкого кола користувачів, незалежно від їх технічної підготовки.

Інтерактивність є ще однією ключовою перевагою ШІ-систем. Користувачі можуть взаємодіяти з системою природною мовою, ставлячи питання про дані у формі, максимально близькій до їх звичного способу мислення. Система може відповідати на запитання, надавати додаткові пояснення, пропонувати альтернативні підходи до аналізу та навіть генерувати нові гіпотези на основі виявлених закономірностей.

Такі рішення можуть суттєво скоротити час від отримання даних до прийняття рішень, підвищити якість аналізу завдяки використанню найсучасніших алгоритмів та зробити процес аналізу більш інтуїтивним та зрозумілим для користувачів різного рівня підготовки. Це особливо важливо в умовах, коли швидкість та точність прийняття рішень на основі даних стають критичними факторами успіху в більшості галузей економіки.

1.2 Основні функціональні вимоги веб-застосунку

Веб-застосунок для аналізу даних з інтеграцією штучного інтелекту повинен забезпечувати комплексну систему функціональних можливостей, що дозволяє користувачам ефективно взаємодіяти з різноманітними типами даних та отримувати якісні аналітичні результати через інтуїтивний інтерфейс. Визначення

детальних функціональних вимог є критичним етапом, який формує фундамент для всіх подальших технічних рішень та архітектурних підходів.

Центральною функціональною вимогою є реалізація зручної системи завантаження файлів, яка підтримує множинні формати даних включаючи *CSV*, *JSON* та *TXT* файли. Система повинна забезпечувати як традиційний спосіб вибору файлів через стандартне діалогове вікно операційної системи, так і сучасну *drag-and-drop* функціональність з візуальними індикаторами активної зони перетягування. Особливо важливою є можливість масового завантаження до десяти файлів одночасно з обмеженням розміру кожного файлу до двох мегабайт, що забезпечує баланс між функціональністю та продуктивністю системи. Інтерфейс завантаження має відображати чіткі повідомлення про обмеження та надавати можливість видалення окремих файлів до початку аналізу.

Ключовою функціональною особливістю є система вибору моделей штучного інтелекту, представлена у вигляді інтуїтивного селекційного інтерфейсу з детальними описами кожної доступної моделі. Користувач повинен мати можливість обирати між різними варіантами, такими як *GPT*-4.1 *Nano* для швидких та економних аналізів або повнофункціональний *GPT*-4.1 для складних аналітичних завдань. Інформаційна панель має відображати ключові характеристики кожної моделі.

Чат-інтерфейс для взаємодії зі штучним інтелектом представляє найбільш критичну функціональну вимогу застосунку. Інтерфейс має підтримувати як структуровані аналітичні запити з чіткою постановкою завдань, так і вільні текстові запити природною мовою. Система повинна автоматично включати контекст завантажених файлів у кожний запит, звільняючи користувача від необхідності вручного зазначення шляхів до файлів чи повторного опису структури даних. Область введення тексту має підтримувати багаторядкові повідомлення з можливістю форматування та попереднього перегляду запиту.

Асинхронна обробка запитів з відповідними візуальними індикаторами становить важливу вимогу для забезпечення позитивного користувацького досвіду.

Після відправлення запиту система повинна відображати анімований індикатор завантаження або прогрес-бар, що інформує користувача про стан обробки.

Відображення результатів аналізу повинно підтримувати повноцінне *Markdown* форматування [8], включаючи структуровані абзаци, нумеровані та марковані списки, виділення тексту жирним шрифтом та курсивом, а також блоки коду з підсвічуванням синтаксису. Особливо важливою є можливість відображення табличних даних, *SQL*-запитів та фрагментів програмного коду. Кожне повідомлення від штучного інтелекту має супроводжуватися набором інтерактивних елементів, включаючи кнопку копіювання всього тексту та систему швидких підказок для формування наступних запитів.

Навігаційні функції інтерфейсу мають включати плавне прокручування довгих відповідей, фіксовану позицію області введення запитів та автоматичне прокручування до останнього повідомлення після отримання відповіді від *AI*. Адаптивний дизайн повинен забезпечувати коректне відображення та функціональність на різних типах пристроїв, від широкоекранних моніторів до смартфонів, з гнучким переформатуванням елементів інтерфейсу відповідно до доступної ширини екрана.

Безпека та конфіденційність даних реалізується через архітектурний підхід, де всі запити до *OpenAI API* здійснюються виключно через *backend*-сервер, що запобігає витоку *API*-ключів на клієнтській стороні. Система контролю доступу включає валідацію заповнення обов'язкових полів, перевірку завантаження мінімально необхідних файлів та деактивацію функціональних кнопок до виконання всіх передумов для відправлення запиту. За необхідності система може бути розширена модулем аутентифікації з підтримкою входу через електронну пошту та пароль.

1.3 Вибір технологічного стеку для реалізації

1.3.1 *Frontend*

Архітектура клієнтської частини базується на використанні *React* у поєднанні з сучасним інструментарієм розробки *Vite*. *React* [1], створений командою

Facebook, представляє собою декларативну *JavaScript*-бібліотеку, що реалізує компонентний підхід до побудови користувацьких інтерфейсів. Основною перевагою *React* є його архітектура, заснована на концепції компонентів як незалежних, самодостатніх одиниць коду, що інкапсулюють логіку відображення та стан. Це дозволяє створювати складні інтерфейси шляхом композиції простіших елементів.

React дозволяє елегантно розділити функціональність на логічні блоки: компоненти чату для взаємодії з ШІ, елементи завантаження файлів, селектори вибору моделей, панелі відображення результатів аналізу та багато інших. Кожен компонент може мати власний стан та життєвий цикл, що забезпечує ізоляцію функціональності та спрощує налагодження. *React* використовує концепцію віртуального *DOM* - ефективного алгоритму порівняння та оновлення лише тих частин реального *DOM*, які дійсно змінилися. Це критично важливо для застосунків з динамічним контентом, таких як чат-інтерфейси з ШІ, де постійно відбуваються оновлення повідомлень, зміни станів завантаження та відображення результатів обробки.

Система стану в *React* реалізована через хуки *useState* та *useEffect*, що дозволяють функціональним компонентам управляти локальним станом та побічними ефектами. Для більш складних випадків управління станом можуть використовуватися контексти *React* або додаткові бібліотеки стану. Особливо важливим є управління асинхронними операціями, такими як запити до *backend API*, завантаження файлів та очікування відповідей від моделей ШІ, де *React* забезпечує елегантні механізми обробки життєвого циклу компонентів.

Vite обрано як сучасну альтернативу традиційним збирачам проектів, таким як *Webpack*. Основною інновацією *Vite* є використання нативних *ES*-модулів браузера під час розробки, що кардинально прискорює час запуску сервера розробки та забезпечує миттєве оновлення модулів без повної перезбірки проекту. Це особливо цінно при розробці складних інтерфейсів, де кожна зміна коду має відразу відображатися в браузері. Система *Hot Module Replacement* у *Vite* працює

на рівні окремих модулів, зберігаючи стан компонентів при оновленні коду, що значно підвищує продуктивність розробки.

Для продакшн-збірки *Vite* використовує *Rollup*, що генерує високооптимізований код з *tree-shaking* для видалення невикористаних частин коду, *code-splitting* для розділення коду на окремі чанки та мініфікацією для зменшення розміру файлів. Конфігурація *Vite* дозволяє налаштувати проксі-сервер для перенаправлення *API*-запитів, що спрощує розробку в контейнеризованому середовищі, де *frontend* та *backend* працюють у різних контейнерах. *Vite* конфігурацію показано на рисунку 1.1.

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

export default defineConfig(({ command }) => {
  return {
    plugins: [react()],
    server: {
      host: "0.0.0.0",
      port: 3000,
      proxy: {
        "/api": {
                    target:   process.env.VITE_PROXY_TARGET   ||
"http://backend:3001",
          changeOrigin: true,
          rewrite: (path) => path.replace(/^\/api/, "/api"),
        },
      },
    },
  };
});
```

Рисунок 1.1 – *Vite* конфігурація

Стилізація інтерфейсу реалізована за допомогою сучасних *CSS*-технологій, включаючи *CSS Custom Properties* для створення тематичного оформлення та динамічної зміни кольорової схеми. Використання *Flexbox* та *CSS Grid* забезпечує гнучке та адаптивне компонування елементів інтерфейсу, що важливо для коректного відображення на різних пристроях та розмірах екрану. *CSS*-анімації та переходи покращують користувацький досвід, створюючи плавні інтерактивні ефекти при взаємодії з елементами інтерфейсу.

Інтеграція зі спеціалізованими бібліотеками розширює функціональність *frontend*-частини. *react-markdown* у поєднанні з *remark-gfm* забезпечує рендеринг *Markdown*-контенту, що дозволяє відображати структуровані відповіді від ШІ з підтримкою форматування тексту, списків, таблиць, блоків коду та інших елементів розмітки. Це критично важливо для представлення результатів аналізу даних у зручному для сприйняття форматі. Бібліотеки іконок, такі як *lucide-react*, надають набір векторних іконок, що забезпечують консистентність візуального оформлення та можливість легкого налаштування стилів.

1.3.2 *Backend*

Серверна архітектура побудована на платформі *Node.js* з використанням фреймворку *Express.js*. *Node.js* представляє собою *JavaScript*-оточення виконання, що базується на високопродуктивному рушії. Ключовою особливістю *Node.js* є його неблокуюча, подієво-керована архітектура, заснована на циклі подій та колбеках. Це робить платформу особливо ефективною для операцій введення-виведення, таких як обробка *HTTP*-запитів, взаємодія з базами даних, зовнішніми *API* та файловою системою. В контексті веб-застосунку з інтеграцією ШІ це критично важливо, оскільки сервер повинен ефективно обробляти одночасні запити від багатьох користувачів, взаємодіяти з *API* штучного інтелекту, обробляти завантаження та аналіз файлів, при цьому забезпечуючи швидкий відгук системи.

Express.js [4] є мінімалістичним та водночас потужним веб-фреймворком для *Node.js*, що надає гнучкий набір інструментів для створення веб-застосунків та *API*. Філософія *Express* полягає в наданні тонкого шару абстракції над базовими можливостями *Node.js*, не нав'язуючи жорстких архітектурних рішень. Це дозволяє розробникам створювати як прості *API*, так і складні веб-застосунки, адаптуючи архітектуру під конкретні потреби проекту. *Express* надає потужну систему маршрутизації, що дозволяє легко визначати обробники для різних *HTTP*-методів та *URL*-шляхів, система *middleware* для реалізації крос-функціональної логіки та інтеграцію з різноманітними шаблонізаторами та базами даних.

Система *middleware* в *Express* відіграє центральну роль в архітектурі *backend*. *Middleware* - це функції, які виконуються послідовно під час обробки *HTTP*-запиту, маючи доступ до об'єктів запиту, відповіді та наступної *middleware*-функції в ланцюжку. Це дозволяє елегантно реалізувати крос-функціональну логіку, таку як аутентифікація, логування, валідація даних, обробка помилок та багато іншого.

CORS middleware вирішує проблему *Cross-Origin Resource Sharing*, що виникає коли *frontend* та *backend* працюють на різних доменах або портах. Без правильного налаштування *CORS* браузери блокуватимуть запити від клієнтської частини до серверної через політики безпеки *same-origin policy*. *CORS middleware* дозволяє налаштувати, які домени, *HTTP*-методи та заголовки дозволені для крос-доменних запитів.

Body-parser middleware відповідає за розбір тіла *HTTP*-запитів у різних форматах. *JSON*-парсер обробляє дані, що надходять у форматі *JSON*, який є стандартом для сучасних веб-*API*. *URL-encoded* парсер обробляє дані форм, що передаються методом *POST*. Правильна конфігурація *body-parser* включає встановлення лімітів на розмір тіла запиту для захисту від атак типу *denial-of-service*.

Multer middleware спеціалізується на обробці *multipart/form-data* запитів, які використовуються для завантаження файлів. *Multer* надає гнучкі можливості конфігурації зберігання файлів - від збереження в пам'яті до збереження на диску з налаштуванням шляхів та імен файлів. Система фільтрації файлів дозволяє обмежувати типи файлів, що приймаються сервером, за *MIME*-типами, розширеннями або іншими критеріями. Налаштування лімітів на кількість та розмір файлів забезпечує захист від зловживань та раціональне використання ресурсів сервера.

Dotenv модуль реалізує стандартну практику управління конфігурацією через змінні середовища. Замість жорсткого кодування конфігураційних параметрів, таких як *API*-ключі, рядки підключення до баз даних, порти та інші налаштування, вони зберігаються у файлі *.env* та завантажуються у процес через змінні середовища. Це забезпечує безпеку конфіденційної інформації, гнучкість

конфігурації для різних середовищ розгортання та відповідність принципам *twelve-factor app methodology*.

1.3.3 *OpenAI API*

Інтеграція з *OpenAI API* становить технологічну основу функціональності штучного інтелекту в застосунку. *OpenAI API* надає доступ до найсучасніших моделей генеративного штучного інтелекту, включаючи сімейство моделей *GPT*, які демонструють виключні можливості в розумінні природної мови, генерації тексту, аналізі даних та виконанні складних когнітивних завдань. Вибір *OpenAI API* зумовлений його лідерськими позиціями в галузі штучного інтелекту, постійним розвитком та покращенням моделей, а також надійною технічною підтримкою та документацією.

Архітектурне рішення щодо інтеграції з *OpenAI API* реалізоване через офіційну *JavaScript*-бібліотеку *openai*, що надає високорівневий інтерфейс для взаємодії з *API*. Ця бібліотека інкапсулює складність *HTTP*-комунікації, аутентифікації, обробки помилок та серіалізації даних, дозволяючи розробникам зосередитися на бізнес-логіці застосунку. Бібліотека підтримує асинхронне програмування через *Promise* та *async*/*await*, що добре інтегрується з подієво-керованою природою *Node.js*.

Система аутентифікації базується на *API*-ключах, що забезпечують безпечний доступ до сервісів *OpenAI*. *API*-ключ зберігається як змінна середовища та завантажується через *dotenv* конфігурацію, що забезпечує безпеку та гнучкість управління доступом. Клієнт *OpenAI* ініціалізується з цим ключем та автоматично додає необхідні заголовки аутентифікації до всіх запитів.

Центральним елементом інтеграції є система генерації структурованих відповідей через *JSON Schema*. Традиційно моделі штучного інтелекту генерують текстові відповіді у вільному форматі, що ускладнює їх програмну обробку. *OpenAI API* підтримує функціональність *structured outputs*, яка дозволяє примусити модель генерувати відповіді у строго визначеному *JSON*-форматі відповідно до заданої

схеми. Це революційна можливість для створення надійних застосунків, оскільки вона гарантує передбачуваність формату відповіді та спрощує її парсинг.

JSON Schema [9] визначає структуру очікуваної відповіді з детальною специфікацією типів даних, обов'язкових полів та обмежень. Схема може включати основну текстову відповідь у форматі *Markdown*, що дозволяє моделі використовувати форматування для покращення читабельності, а також структуровані дані, такі як масиви пропозицій або метадані аналізу. Параметр *strict*: *true* забезпечує строге дотримання схеми моделлю, що критично важливо для стабільності застосунку. Відповідь від ШІ у форматі *JSON* показано на рисунку 1.2.

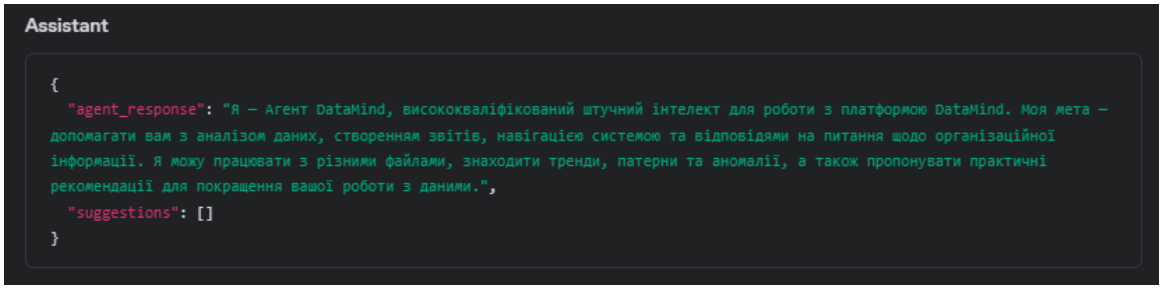

Рисунок 1.2 – Відповідь від ШІ у форматі *JSON*

Система промптінгу реалізована через динамічну генерацію системних інструкцій, що визначають поведінку ШІ-агента. Системний промпт включає опис ролі агента, його спеціалізації в аналізі даних, стратегії взаємодії з користувачем, правила обробки прикріплених файлів та стиль комунікації. Особливо важливим є включення контенту завантажених файлів безпосередньо до промпту, що надає моделі контекст для аналізу.

Налаштування параметрів генерації тексту дозволяє точно контролювати поведінку моделі. Параметр *temperature* регулює творчість моделі - низькі значення призводять до більш детермінованих та сфокусованих відповідей, що важливо для аналітичних завдань, де потрібна точність та консистентність. Можливість налаштування максимальної кількості токенів у відповіді дозволяє контролювати довжину генерованого тексту та управляти витратами на *API*.

1.3.4 Контейнеризація

Контейнеризація представляє собою сучасний підхід до упакування, розповсюдження та виконання застосунків, що забезпечує консистентність середовища виконання незалежно від базової інфраструктури.

Docker обрано як основну платформу контейнеризації завдяки його зрілості, широкій підтримці спільноти та ефективності. *Docker* використовує технології *Linux kernel*, такі як *namespaces* та *cgroups*, для створення ізольованих середовищ виконання, що поділяють ядро операційної системи, але мають окремі файлові системи, мережі та процеси. Це забезпечує легковагість контейнерів порівняно з традиційними віртуальними машинами при збереженні високого рівня ізоляції.

Архітектура контейнеризації побудована за принципом мікросервісів, де кожен логічний компонент застосунку (*frontend*, *backend*) розгортається в окремому контейнері.

Dockerfile для *backend*-сервісу базується на офіційному образі *node*:18-*alpine*, який являє собою мінімалістичну дистрибуцію *Linux Alpine* з попередньо встановленим *Node.js* версії 18. *Alpine Linux* вибрано через його компактність - базовий образ займає лише кілька мегабайт, що значно зменшує розмір кінцевих контейнерів та прискорює їх завантаження та розгортання.

Управління змінними середовища реалізовано через комбінацію файлів *.env* та секції *environment* у *docker-compose.yml*. Це забезпечує гнучкість конфігурації для різних середовищ розгортання без необхідності перебудови образів. Особлива увага приділена безпеці конфіденційних даних, таких як *API*-ключі, які ні в якому разі не повинні потрапляти до образів контейнерів. Конфігурацію змінних оточення показано на рисунку 1.3.

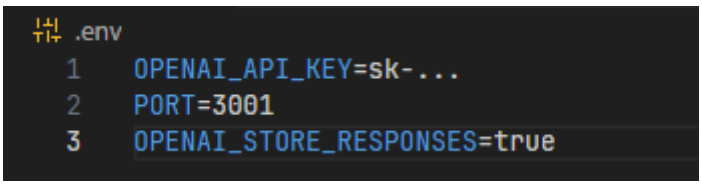

Рисунок 1.3 – Конфігурація змінних оточення

РОЗДІЛ 2
РОЗРОБКА ВЕБ-ЗАСТОСУНКУ

2.1 Архітектура застосунку

Архітектура веб-застосунку побудована на фундаментальних принципах клієнт-серверної архітектури, що передбачає чітке розмежування відповідальності між клієнтською частиною (*frontend*) та серверною логікою (*backend*). Такий архітектурний підхід забезпечує високий рівень гнучкості системи, можливості її горизонтального масштабування та суттєво спрощує процеси розробки, тестування та подальшої підтримки програмного продукту. Ключовою особливістю архітектури є реалізація *frontend* та *backend* компонентів у вигляді незалежних мікросервісів, що комунікують між собою через мережеві протоколи. Для ефективного управління життєвим циклом цих сервісів, забезпечення їх ізоляції від операційного середовища та автоматизації процесів розгортання використовується система контейнеризації *Docker* у поєднанні з інструментом оркестрації *Docker Compose*.

Клієнтська частина системи реалізована як сучасний односторінковий застосунок (*Single Page Application*), розроблений з використанням бібліотеки *React*. Функціональна відповідальність *frontend*-компоненту охоплює створення інтерактивного графічного інтерфейсу користувача, який надає можливості для ведення діалогу з *AI*-агентом через чат-інтерфейс, введення текстових запитів, вибору відповідної моделі штучного інтелекту з доступного переліку, завантаження файлів різних форматів (*CSV, JSON, TXT*) для подальшого аналізу та візуалізації отриманих від *AI*-агента відповідей. Вся логіка, що стосується обробки користувацьких взаємодій, управління станом інтерфейсу, валідації введених даних та формування запитів до серверної частини, концентрується в межах *frontend*-компоненту.

Комунікаційний процес між клієнтською та серверною частинами реалізований на основі асинхронних *HTTP*-запитів, що використовують метод *POST* для передачі даних на спеціально визначений *API*-ендпоінт серверної

частини. Такий підхід забезпечує неблокуючу взаємодію з користувачем та можливість обробки множинних запитів одночасно. Для забезпечення комфортних умов розробки використовується *Vite development server*, який конфігурується для проксіювання *API*-запитів до *backend*-сервісу, що дозволяє уникнути проблем з *Cross-Origin Resource Sharing* (*CORS*) під час локальної розробки.

Використання *Docker*-томів (*frontend_node_modules*, *backend_node_modules*) дозволяє кешувати директорії *node_modules* поза межами контейнерів, що значно прискорює процес перебудови образів під час розробки та зменшує час, необхідний для встановлення залежностей. Конфігурація змінних середовища, таких як *OPENAI_API_KEY* та *PORT*, здійснюється через *env_file*, що забезпечує безпечне управління конфіденційними даними без необхідності їх жорсткого кодування в початковому коді застосунку. Файлову структуру проекту показано на рисунку 2.1.

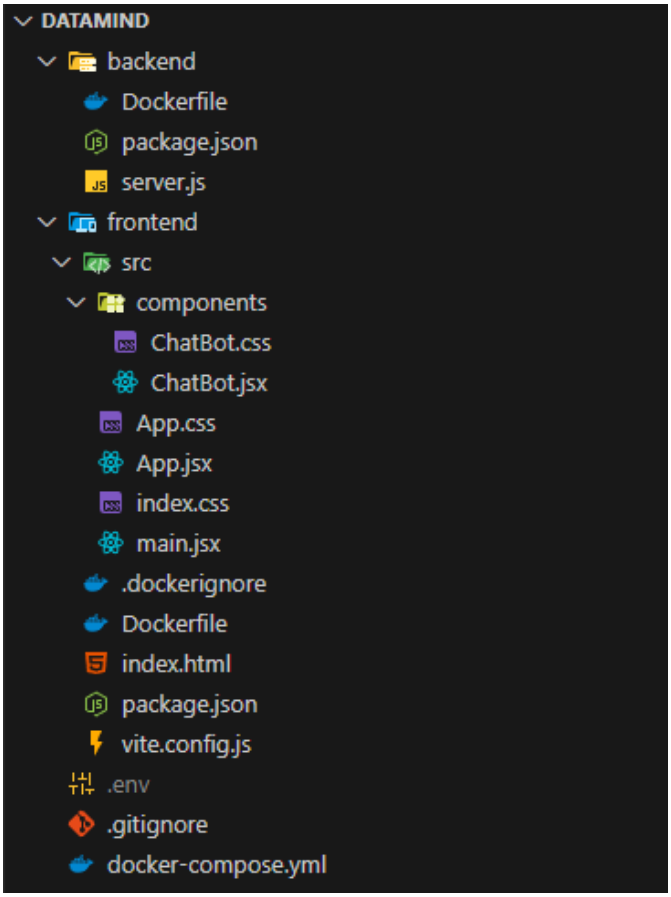

Рисунок 2.1 – Файлова структура проекту

Архітектура застосунку *DataMind* характеризується високим рівнем модульності та структурованості, що забезпечує ефективність процесів розробки та

підтримки. Чітке розділення на *frontend* та *backend* компоненти дозволяє здійснювати незалежну розробку, тестування та масштабування кожної частини системи, що особливо важливо для команд розробників. *Backend*-сервіс функціонує як захищений шлюз до *OpenAI API*, інкапсулюючи всю складну логіку взаємодії з зовнішнім сервісом та забезпечуючи централізовану обробку даних. Застосування контейнеризації через *Docker* та *Docker Compose* гарантує повну відтворюваність робочого середовища, суттєво спрощує процеси розгортання в різних середовищах та автоматизує управління залежностями, що є критично важливим для сучасних веб-застосунків корпоративного рівня.

2.2 Розробка *Frontend* частини

2.2.1 Інтерфейс та логіка взаємодії

Архітектура *frontend*-частини застосунку базується на *React*-бібліотеці з використанням функціонального підходу та *React Hooks* для управління станом. Ключовим компонентом системи є *ChatBot.jsx*, який виконує роль центрального контролера для всієї логіки чат-інтерфейсу та взаємодії з користувачем.

Управління станом компонента реалізовано через хук *useState*, який забезпечує реактивність інтерфейсу. Стан *messages* представлений у вигляді масиву об'єктів, де кожне повідомлення містить унікальний ідентифікатор (*id*), текстовий вміст (*text*), інформацію про відправника (*sender*), часову мітку (*timestamp*), а для повідомлень від штучного інтелекту додатково включає поле *suggestions* для пропозицій подальших дій та логічний прапорець *doNotSendToAPI*, який визначає, чи слід відправляти конкретне повідомлення на сервер. Додатковими елементами стану є *inputValue* для збереження поточного вмісту поля введення, булевий індикатор *isTyping* для відображення процесу очікування відповіді від *AI*, *copiedMessageId* для тимчасового візуального підтвердження успішного копіювання повідомлення та *selectedModel* для збереження ідентифікатора обраної моделі штучного інтелекту.

Ініціалізація інтерфейсу відбувається при першому завантаженні компонента, коли стан *messages* автоматично заповнюється вітальним

повідомленням від бота, що забезпечує природний початок діалогу для користувача. Інтерфейс додатка показано на рисунку 2.2.

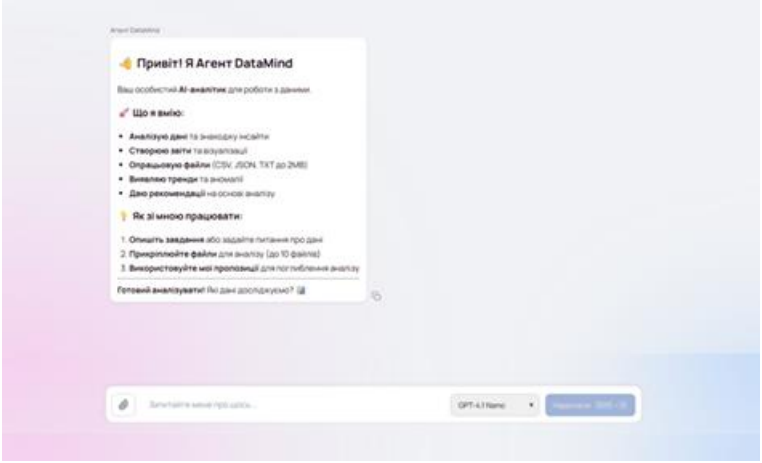

Рисунок 2.2 – Інтерфейс додатка

Відображення повідомлень у чат-області (*.messages-area*) реалізовано через ітеративний рендеринг масиву *messages* з використанням методу *map*. Кожен об'єкт повідомлення трансформується у відповідний *JSX*-елемент з урахуванням типу відправника. Повідомлення користувача отримують класи стилізації *.user-message-wrapper* з правостороннім вирівнюванням, тоді як повідомлення бота використовують *.bot-message-wrapper* з лівостороннім розташуванням. Системні повідомлення (*sender* === "*system*") мають спеціальну логіку рендерингу та отримують унікальний клас *.system-message-frame* для відмінної візуальної презентації. Плавна анімація появи нових повідомлень досягається за допомогою *CSS*-анімації *fadeInUp*, яка визначена через *keyframes*.

Взаємодія з *DOM*-елементами забезпечується системою *React*-рефів (*useRef*). Референс *messagesEndRef* прив'язується до порожнього *div*-елементу в кінці списку повідомлень, що дозволяє реалізувати автоматичне прокручування чату до останнього повідомлення через виклик *messagesEndRef.current*?.*scrollIntoView*({ *behavior*: "*smooth*" }). Функція *scrollToBottom* викликається всередині *useEffect*, який моніторить зміни станів *messages* та *isTyping*, забезпечуючи актуальне позиціонування скролу при додаванні нових повідомлень або зміні стану типізації.

Адаптивність інтерфейсу забезпечується додатковими рефами та функціями. Референс *textareaRef* використовується для динамічного регулювання висоти текстового поля введення через функцію *autoGrowTextarea*, яка автоматично розширює поле в залежності від об'єму введеного тексту. *InputFrameRef* та *messagesAreaRef* забезпечують коректне обчислення геометрії інтерфейсу через функцію *adjustMessagesAreaLayout*, яка запобігає перекриттю області повідомлень панеллю введення. Ці функції адаптації викликаються через *useEffect* при зміні відповідних станів або при зміні розмірів вікна браузера.

Обробка користувацьких дій реалізована через систему *event handlers*. Зміна вмісту поля введення (*textarea*) обробляється через *onChange*, який оновлює стан *inputValue* в реальному часі. Відправлення повідомлення може відбуватися двома способами: через *submit* форми (обробник *onSubmit*={*handleFormSubmit*}) або через комбінацію клавіш *Ctrl+Enter* (обробник *onKeyDown*={*handleKeyDown*} на *textarea*). Функція *sendMessage* створює новий об'єкт повідомлення користувача з усіма необхідними полями, додає його до масиву *messages*, очищає поле введення, встановлює індикатор *isTyping* у *true* та ініціює асинхронний запит до *API*.

Інтерактивність інтерфейсу розширюється через систему пропозицій (*suggestion chips*). Клік по елементу *.suggestion-chip* викликає функцію *handleSuggestionClick*, яка передає текст пропозиції до функції *sendMessage* для автоматичного формування запиту. Функціонал копіювання тексту повідомлень реалізований через *handleCopyToClipboard*, яка використовує сучасний *API navigator.clipboard.writeText*() та тимчасово оновлює стан *copiedMessageId* для зміни іконки кнопки копіювання, надаючи користувачу візуальне підтвердження успішної операції. Систему пропозицій від ШІ показано на рисунку 2.3.

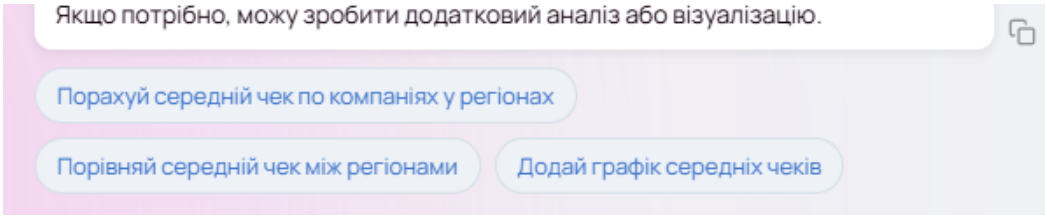

Рисунок 2.3 – Система пропозицій від ШІ

Візуальна індикація активності штучного інтелекту (стан *isTyping*) реалізована не лише через показ анімованого індикатора *.typing-indicator*, але й через динамічну зміну *CSS*-змінних *--ellipse-9-bg* та *--ellipse-10-bg* шляхом додавання класу *is-typing* до кореневого елементу *.chat-bot-ui*. Приклад візуальної індикації при обробці даних показано на рисунку 2.4. Це забезпечує зміну кольору фонових анімованих елементів, створюючи цілісний візуальний досвід очікування.

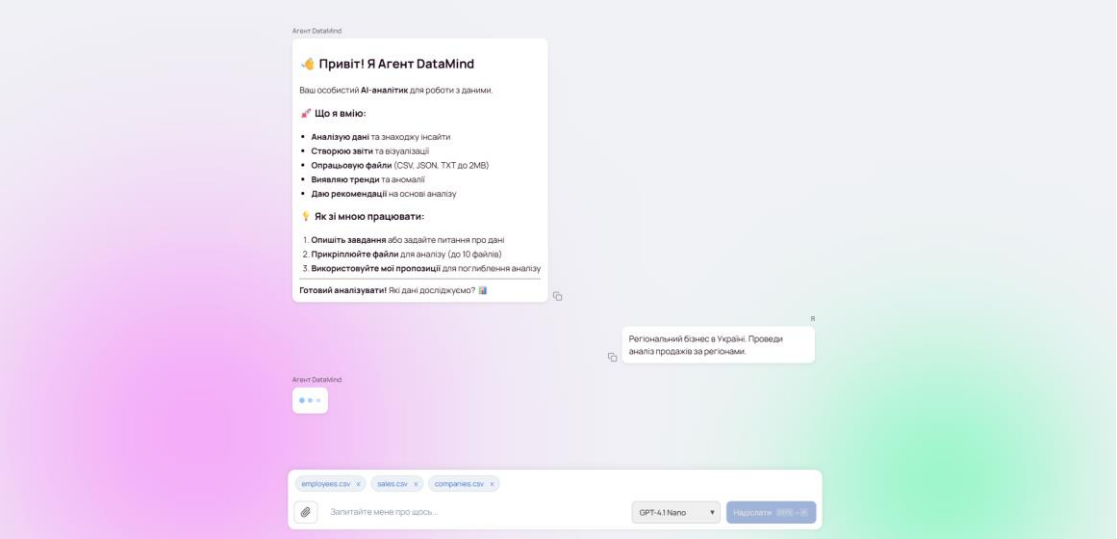

Рисунок 2.4 – Приклад візуальної індикації при обробці даних

Підтримка форматованого тексту в повідомленнях бота забезпечується інтеграцією компонента *ReactMarkdown* з плагіном *remarkGfm*, що дозволяє відображати *Markdown*-розмітку. Приклад відповіді у форматі *markdown* показано на рисунку 2.5.

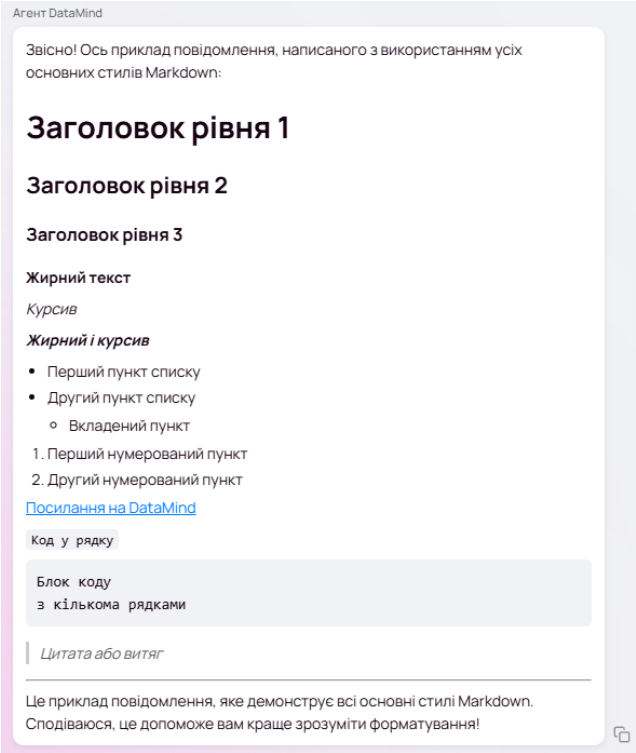

Рисунок 2.5 – Приклад відповіді у форматі *markdown*

2.2.2 Реалізація завантаження файлів та вибору моделі ШІ

Система завантаження файлів у застосунку підтримує два основних сценарії взаємодії з користувачем: традиційний вибір через діалогове вікно файлового менеджера та сучасний механізм *drag-and-drop* для більш інтуїтивного користувацького досвіду.

Перший сценарій реалізований через прихований *HTML*-елемент <*input type="file" ref={fileInputRef} multiple onChange={handleFileChange}*>, який залишається невидимим для користувача, але доступний для програмного управління. Візуальна кнопка "Додати файли" (*.attach-file-button*) при натисканні програмно активує цей *input* через виклик *fileInputRef.current?.click*(), що запускає стандартне діалогове вікно вибору файлів операційної системи. Атрибут *multiple* дозволяє користувачам обирати декілька файлів одночасно.

Функціонал *drag-and-drop* реалізований через систему обробників подій *DOM*, які додаються до основного контейнера чату *.chat-bot-ui* через референс *chatBotUiRef*. Обробники *dragenter*, *dragleave*, *dragover* та *drop*, обгорнуті в *useCallback* для оптимізації продуктивності та запобігання непотрібним

перерендерам, управляють станом *isDraggingOver*. Цей стан контролює візуальний відгук інтерфейсу через додавання псевдоелементу *.drag-over-active::before*, який створює напівпрозорий оверлей з індикацією зони скидання файлів. Функція *handleDrop* отримує список файлів з властивості *event.dataTransfer.files*. Приклад перетягування файлів показано на рисунку 2.6.

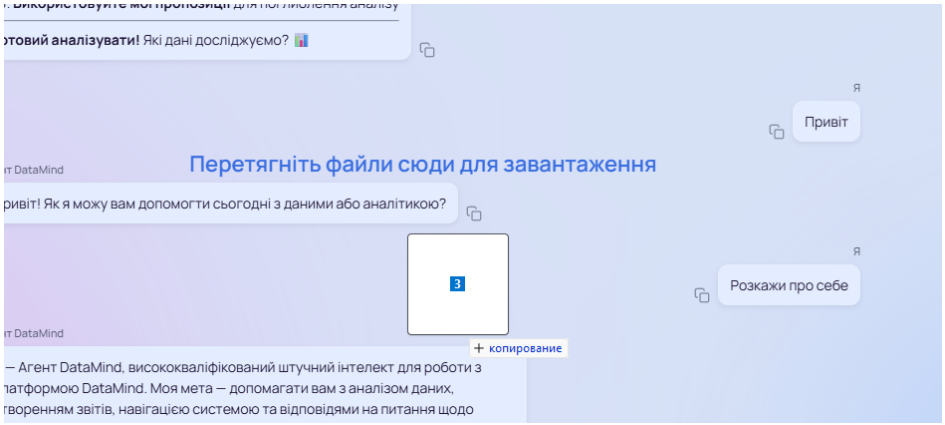

Рисунок 2.6 – Приклад перетягування файлів

Незалежно від способу отримання файлів, їх обробка централізована у функції *processFiles*, яка також обгорнута в *useCallback* для оптимізації. Ця функція реалізує комплексну клієнтську валідацію, що включає перевірку загальної кількості прикріплених файлів відносно константи *MAX_FILES*, індивідуальну перевірку розміру кожного файлу відносно *MAX_FILE_SIZE_BYTES*, валідацію *MIME*-типу через властивість *file.type* та перевірку розширення файлу відносно масивів *ALLOWED_FILE_TYPES* та *ALLOWED_FILE_EXTENSIONS*. Додатково перевіряється унікальність імен файлів для запобігання конфліктам.

При виявленні порушень валідації система викликає функцію *addSystemMessage* (також обгорнуту в *useCallback*), яка додає інформаційне повідомлення до чату, повідомляючи користувача про конкретну причину відхилення файлу. Файли, що пройшли валідацію, додаються до стану *attachedFiles* у вигляді масиву об'єктів, де кожен об'єкт містить унікальний ідентифікатор (*id*), оригінальний об'єкт *File* (*fileObject*) та ім'я файлу (*name*). Приклад відображення помилки показано на рисунку 2.7.

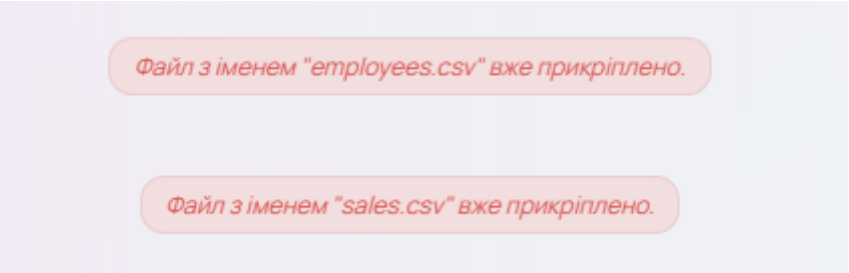

Рисунок 2.7 – Приклад відображення помилки

Візуальне відображення прикріплених файлів здійснюється в області *.attached-files-area* через ітерацію масиву *attachedFiles* та рендеринг компонентів *.file-chip* для кожного файлу. Кожен чіп містить назву файлу та кнопку видалення (*.remove-file-button*), яка викликає функцію *removeAttachedFile*. Ця функція використовує метод *filter* для створення нового масиву *attachedFiles* без видаленого елемента. Відображення прикріплених файлів для аналізу показано на рисунку 2.8.

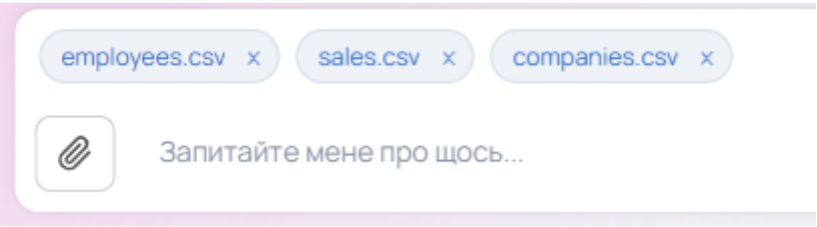

Рисунок 2.8 – Відображення прикріплених файлів для аналізу

При формуванні запиту до серверної частини в функції *sendMessage* створюється об'єкт *FormData* для передачі мультимедіа-контенту. До цього об'єкта додаються історія повідомлень у форматі *JSON*-рядка, ідентифікатор обраної моделі та кожен файл з масиву *attachedFiles* через виклик *formData.append*("*attached_files*", *fileItem.fileObject*, *fileItem.name*), що забезпечує коректну передачу файлів на сервер із збереженням оригінальних назв.

Система вибору моделі штучного інтелекту реалізована через окремий компонент *CustomModelSelector*, який забезпечує інтуїтивний інтерфейс для перемикання між різними *AI*-моделями. Компонент має власний внутрішній стан *isOpen*, керований через *useState*, який контролює видимість випадаючого списку доступних моделей. Референс *selectorRef* (*useRef*) використовується для реалізації

логіки автоматичного закриття списку при кліку поза його межами через *useEffect* та *addEventListener* на *document*.

Дані про доступні моделі передаються компоненту через *AVAILABLE_MODELS_DATA*, що забезпечує гнучкість конфігурації без зміни коду компонента. Основний елемент компонента – кнопка-тригер (*.selector-trigger-light*) відображає назву поточно обраної моделі, яка знаходиться шляхом пошуку в *AVAILABLE_MODELS_DATA* за *selectedModelId*. При натисканні на тригер стан *isOpen* змінюється, показуючи або приховуючи випадаючий список *.selector-dropdown-light*.

У випадаючому списку моделі можуть групуватися за категоріями (якщо така інформація присутня в *AVAILABLE_MODELS_DATA*), що забезпечує логічну організацію великої кількості моделей. Кожна модель представлена елементом *.dropdown-item-light*, який містить компонент *ModelIcon* для відображення візуальної ідентифікації моделі, назву, інформацію про провайдера та короткий опис функціональності. Вибір моделі ШІ показано на рисунку 2.9.

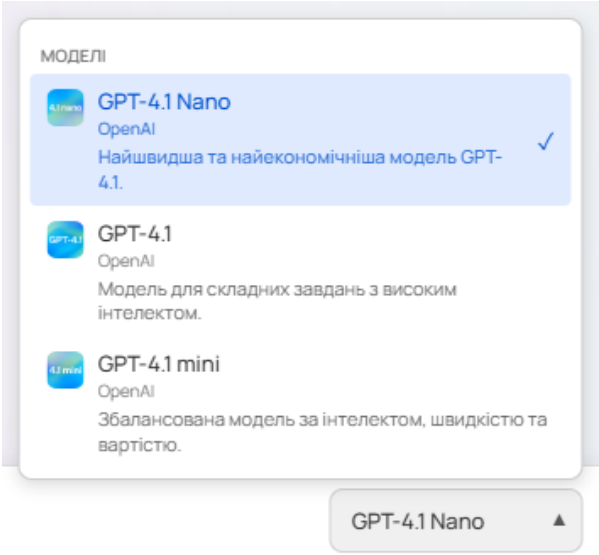

Рисунок 2.9 – Вибір моделі ШІ

Компонент *ModelIcon* має вбудовану логіку обробки помилок завантаження зображень через обробник *onError* на елементі *<img>*, який відображає плейсхолдер у випадку недійсного *URL* іконки або проблем з мережею. Вибір конкретної моделі зі списку викликає функцію *onModelSelect*, яка передається як

пропс з батьківського компонента *ChatBot.jsx* і представлена функцією *handleModelChange*. Ця функція оновлює стан *selectedModel* у батьківському компоненті, а ідентифікатор обраної моделі згодом використовується при формуванні *API*-запитів для забезпечення використання відповідної моделі штучного інтелекту для обробки користувацьких запитів.

2.3 Розробка *Backend* частини

2.3.1 Сервер для обробки *API*-запитів

Серверна частина веб-застосунку побудована на базі платформи *Node.js* з використанням фреймворку *Express.js*, який є одним із найбільш поширених рішень для створення *RESTful API* завдяки своїй гнучкості, простоті конфігурації та мінімалістичному підходу до архітектури. Основним файлом серверного застосунку є *backend/server.js*, який виконує роль точки входу та містить всю логіку обробки *HTTP*-запитів.

На початку роботи сервера відбувається імпорт усіх необхідних модулів та бібліотек. Модуль *express* використовується для створення *HTTP*-сервера та налаштування маршрутизації. Бібліотека *openai* забезпечує взаємодію з *API* штучного інтелекту *OpenAI*. Модуль *dotenv* відповідає за завантаження конфігураційних параметрів із файлу *.env*, включаючи критично важливий *OPENAI_API_KEY*. Модуль *cors* налаштовує політику *Cross-Origin Resource Sharing*, що є необхідним для коректної взаємодії між фронтенд та бекенд частинами застосунку. Бібліотека *body-parser* забезпечує парсинг тіла *HTTP*-запитів у різних форматах, зокрема *JSON* та *URL-encoded*. Модуль *multer* призначений для обробки файлів, що завантажуються користувачами через веб-інтерфейс.

Після імпорту всіх залежностей відбувається ініціалізація *Express*-застосунку через створення константи *app*. Наступним кроком є налаштування проміжних обробників (*middleware*), які виконуються для кожного вхідного запиту. Використання *app.use*(*cors*()) дозволяє обробляти запити з інших доменів, що є критично важливим для взаємодії між фронтендом, який працює на порту 3000, та

бекендом на порту 3001. Налаштування *app.use*(*bodyParser.json*()) та *app.use*(*bodyParser.urlencoded*({ *extended*: *true* })) забезпечує автоматичний парсинг тіла запитів у форматах *JSON* та *x-www-form-urlencoded* відповідно, роблячи дані доступними через об'єкт *req.body* у обробниках маршрутів.

Особливу увагу приділено налаштуванню обробки файлів за допомогою бібліотеки *multer*. Конфігірація *multer* передбачає збереження завантажених файлів безпосередньо в оперативній пам'яті через використання *multer.memoryStorage*(), що є оптимальним рішенням для тимчасової обробки файлів перед передачею їх вмісту до системи штучного інтелекту. Встановлено чіткі обмеження на кількість одночасно завантажуваних файлів (*NEW_MAX_FILES* = 10) та максимальний розмір кожного файлу (2 мегабайти), що запобігає перевантаженню сервера та забезпечує стабільну роботу системи.

Критично важливою є реалізація функції *fileFilter*, яка здійснює валідацію типів файлів на етапі завантаження. Ця функція перевіряє *MIME*-тип кожного завантажуваного файлу, дозволяючи обробку лише файлів типів *text*/*csv*, *application*/*json* та *text*/*plain*. Якщо користувач намагається завантажити файл непідтримуваного типу, система автоматично генерує відповідну помилку, попереджаючи потенційні проблеми безпеки та забезпечуючи коректність подальшої обробки даних.

Центральним елементом *API* є маршрут для взаємодії з чат-ботом, визначений як *POST*-ендпоінт "/*api*/*chat*". Цей маршрут інтегрований з *multer* через *middleware upload.array*("*attached_files*", *NEW_MAX_FILES*), який автоматично обробляє файли, передані в полі *attached_files HTTP*-форми. Обробник маршруту очікує два основні параметри в тілі запиту: *messages*, який містить історію чату у форматі *JSON*-рядка, та *selectedModel*, що представляє ідентифікатор обраної моделі *OpenAI*.

На початку обробки запиту відбувається комплексна валідація вхідних даних. Система перевіряє наявність обов'язкових параметрів та коректність формату *messages*, який після парсингу *JSON* має представляти масив повідомлень. Якщо користувач прикріпив файли до запиту, сервер ітеративно обробляє кожен файл з

масиву *req.files*, зчитуючи їх вміст як *UTF*-8 рядок через метод *file.buffer.toString*("*utf*8") та формуючи структуровані дані, що включають ім'я файлу, його вміст та тип. Ці дані зберігаються в масиві *attachedFilesData*, який надалі використовується для формування контексту для системи штучного інтелекту.

Запуск сервера здійснюється на порту, визначеному в змінній середовища *PORT*, або на порту 3001 за замовчуванням. При старті сервер виводить в консоль детальну діагностичну інформацію, включаючи номер порту, статус завантаження *OPENAI_API_KEY* та поточний режим зберігання відповідей *OpenAI*, що спрощує процеси розробки та налагодження системи.

2.3.2 Інтеграція з *OpenAI API*

Ключовим елементом підготовки запиту до *OpenAI* є функція *getSystemInstructions*(*attachedFilesData*), яка відповідає за генерацію системних інструкцій або промпту для моделі штучного інтелекту. Ці інструкції визначають роль *AI* як "Агент *DataMind*", встановлюють мовну стратегію роботи з переважним використанням української мови та адаптацією до мови користувача, визначають стиль відповідей як проактивний з активним використанням *Markdown*-форматування. Найважливішим аспектом є детальна стратегія роботи з наданими файлами та контекстом діалогу.

Коли користувач прикріплює файли до запиту, їх вміст інтегрується безпосередньо в системний промпт. Система штучного інтелекту отримує чіткі вказівки використовувати ці файли для формування відповідей та посилатися на них за назвою, що забезпечує релевантність та контекстуальність аналізу.

Особливо детально опрацьована інструкція щодо обробки складних запитів з множинними файлами. Ця інструкція включає алгоритм визначення умов фільтрації даних, методи ідентифікації ключових полів для встановлення зв'язків між різними наборами даних та стратегію послідовної обробки інформації. Критично важливою є вимога до *AI* надавати відповідь виключно у форматі *JSON*,

що відповідає строго визначеній схемі, забезпечуючи передбачуваність та надійність парсингу відповідей на стороні клієнта.

Схема відповіді *aiResponseSchema* детально описує очікувану структуру *JSON*-відповіді від моделі. Схема включає два обов'язкових поля: *agent_response*, що містить основну текстову відповідь агента з можливістю використання *Markdown*-форматування, та *suggestions*, що представляє масив з 2-3 коротких пропозицій для наступних дій користувача або значення *null*, якщо пропозиції недоречні в поточному контексті. Використання такої жорсткої схеми дозволяє фронтенд частині застосунку надійно парсити та коректно відображати відповіді штучного інтелекту. Структуру *JSON*-відповіді показано на рисунку 2.10.

```
const aiResponseSchema = {
  type: "object",
  properties: {
    agent_response: {
      type: "string",
      description:
           "Основна текстова відповідь від Агента DataMind українською мовою (якщо користувач не вказав іншу). Має бути чіткою, розмовною та релевантною. Може використовувати Markdown для форматування (наприклад, **жирний**, *курсив*, списки).",
    },
    suggestions: {
      type: ["array", "null"],
      items: { type: "string" },
      description:
          "Масив з 2-3 коротких, релевантних пропозицій для наступних дій/запитань українською (якщо не вказано інше). Якщо недоречно, має бути null.",
    },
  },
  required: ["agent_response", "suggestions"],
  additionalProperties: false,
};
```

Рисунок 2.10 – Структура *JSON*-відповіді

Перед відправкою запиту до *OpenAI API* відбувається трансформація історії чату, отриманої від клієнта через параметр *req.body.messages*, у формат, сумісний з *API OpenAI*. Цей процес включає перетворення даних у масив об'єктів з обов'язковими полями *role*, що може мати значення "*user*" або "*assistant*", та *content*,

що містить текстовий вміст повідомлення. Така стандартизація забезпечує коректну передачу контексту діалогу до моделі штучного інтелекту.

Безпосередній запит до *OpenAI* здійснюється через метод *openai.responses.create*(), який є специфічним для конкретної версії *SDK* або типу моделі. Структура запиту включає декілька ключових параметрів: *model*, що визначає обрану користувачем модель (наприклад, "*gpt*-4.1-*nano*-2025-04-14"), *instructions*, що містить згенерований системний промпт, та *input*, що включає підготовлену історію чату. Запит до *OpenAI API* показано на рисунку 2.11.

```
const completion = await openai.responses.create({
        model: selectedModel,
        instructions: systemInstructionsContent,
        input: openAIInputs,
        text: {
          format: {
            type: "json_schema",
            name: "DataMindAgentResponse",
            schema: aiResponseSchema,
            strict: true,
          },
        },
        temperature: 0.25,
        store: storeOpenAIResponses,
      });
```

Рисунок 2.11 – Запит до *OpenAI API*

Особливо важливим є параметр *text* з конфігурацією { *format*: { *type*: "*json_schema*", *name*: "*DataMindAgentResponse*", *schema*: *aiResponseSchema*, *strict*: *true* } }. Цей параметр інструктує *OpenAI API* генерувати відповідь, яка строго відповідає наданій *JSON*-схемі *aiResponseSchema*, забезпечуючи структурованість та передбачуваність результатів. Параметр *temperature* встановлений на значення 0.25, що забезпечує більш детерміновані та сфокусовані відповіді, зменшуючи випадковість генерації тексту. Змінна середовища *OPENAI_STORE_RESPONSES*, що має булеве значення, контролює збереження відповідей на стороні *OpenAI*, що може бути корисним для цілей відлагодження та аналізу якості роботи системи. Логування запитів в *OpenAI API* показано на рисунку 2.12.

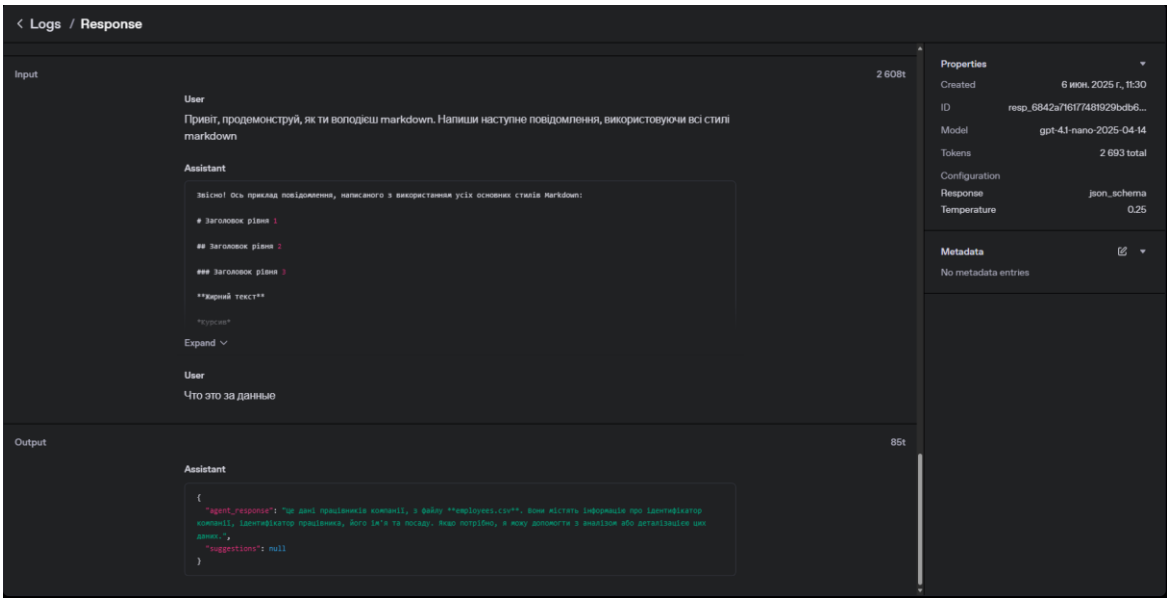

Рисунок 2.12 – Логування запитів в *OpenAI API*

Після отримання відповіді від *OpenAI API*, представленої об'єктом *completion*, сервер виконує серію перевірок та обробку результату. Спочатку перевіряється наявність відмови від *API* через аналіз значення *firstContentItem.type* === "*refusal*". Якщо текстовий вивід відсутній, система автоматично генерує відповідну помилку. Отриманий текстовий результат, який має представляти коректний *JSON*-рядок, обробляється за допомогою *JSON.parse*() для перетворення в *JavaScript*-об'єкт.

Наступним етапом є детальна валідація розпарсеного об'єкта на відповідність очікуваній структурі. Система перевіряє наявність та коректність типів даних для полів *agent_response* та *suggestions*. Якщо процес парсингу або валідації завершується невдачею, клієнту надсилається структурована відповідь про помилку, що включає частину сирої відповіді від *AI* для полегшення діагностики проблем розробниками.

Система обробки помилок під час взаємодії з *OpenAI API* включає відловлювання специфічних *HTTP*-статусів та типів помилок. Статус 401 сигналізує про помилку автентифікації, зазвичай пов'язану з некоректним або застарілим *API*-ключем. Статус 429 вказує на перевищення ліміту кількості запитів, встановленого постачальником *API*. Крім специфічних помилок, система обробляє

загальні помилки мережі або *API*, забезпечуючи клієнту надійне повернення структурованих *JSON*-відповідей з детальним описом проблеми.

2.4 Контейнеризація застосунку

Оркестрація контейнерної інфраструктури в проекті *DataMind* реалізується через централізований файл конфігурації *docker-compose.yml* [7], який визначає архітектуру системи як код та описує взаємодію між двома ключовими сервісами: *frontend* та *backend*, які функціонують як незалежні, але тісно інтегровані компоненти єдиної екосистеми застосунку.

Конфігурація сервісу *frontend* передбачає збірку контейнера з контексту директорії *./frontend*, використовуючи інструкції, детально описані в локальному *Dockerfile* цієї директорії. Мережеве з'єднання реалізується через маппінг порту 3000 контейнера на відповідний порт 3000 хостової системи за схемою "3000:3000", що забезпечує безпосередній доступ до клієнтської частини застосунку через веб-браузер за локальною адресою *http://localhost*:3000. Особливо важливим аспектом конфігурації є використання іменованого тому *frontend_node_modules* для директорії */app/node_modules* всередині контейнера, що дозволяє зберігати встановлені залежності *Node.js* між циклами перезапуску контейнера та суттєво оптимізує швидкість повторних процесів збірки у випадках, коли самі залежності залишаються незмінними. *Dockerfile* показано на рисунку 2.13.

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "run", "dev"]
```

Рисунок 2.13 – *Dockerfile* (*frontend*)

Змінна середовища *NODE_ENV* налаштована на значення *development*, що активує специфічні режими роботи інструментів розробки, зокрема *Vite development server* з його розширеними можливостями швидкого оновлення та гарячого перезавантаження модулів. Директива *depends_on* із зазначенням *backend*-сервісу інструктує *Docker Compose* щодо необхідності запуску серверної частини перед ініціалізацією клієнтської частини застосунку. Обидва сервіси функціонують в рамках спільної віртуальної мережі *app-network*, що дозволяє їм здійснювати внутрішню комунікацію через використання імен сервісів як *DNS*-записів, наприклад, *frontend*-сервіс може звертатися до *backend*-сервісу за внутрішньою адресою *http*://*backend*:3001.

Backend-сервіс конфігурується за аналогічними принципами з урахуванням специфіки серверної архітектури. *Dockerfile* показано на рисунку 2.14.

```
FROM node:18-alpine

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3001

CMD ["node", "server.js"]
```

Рисунок 2.14 – *Dockerfile* (*backend*)

Процес збірки виконується з контексту директорії ./*backend* з використанням відповідного *Dockerfile*. Мережевий доступ забезпечується через маппінг порту 3001 контейнера на порт 3001 хостової системи, надаючи зовнішній доступ до *API*-сервера. Аналогічно до *frontend*, застосовується іменований том *backend_node_modules* для директорії /*usr*/*src*/*app*/*node_modules* з ідентичними перевагами щодо оптимізації та кешування залежностей.

Змінна середовища *NODE_ENV* також встановлена в режим *development*, а додаткова змінна *PORT* визначена як 3001, що точно відповідає порту, на якому

функціонує *Express*-сервер. Критично важливою конфігураційною директивою є *env_file* із зазначенням шляху *./.env*, яка забезпечує автоматичне завантаження змінних середовища, включаючи конфіденційний *OPENAI_API_KEY*, з файлу оточення, розташованого в кореневій директорії проекту. Ця практика відповідає сучасним стандартам безпеки для управління чутливими даними, виключаючи їх пряме включення до системи контролю версій.

Іменовані томи *datamind_frontend_node_modules* та *datamind_backend_node_modules* декларуються в окремій секції *volumes* наприкінці конфігураційного файлу *docker-compose.yml*, що дозволяє *Docker* автоматично управляти їх життєвим циклом та забезпечує персистентність даних. Мережа *app-network* типу *bridge* створює ізольоване мережеве середовище для безпечної взаємодії контейнерів, дозволяючи їм ідентифікувати один одного через використання імен сервісів замість динамічних *IP*-адрес.

Використання *.dockerignore* суттєво зменшує розмір фінального образу, прискорює процес збірки через зменшення обсягу даних для копіювання та запобігає випадковому включенню конфіденційних або непотрібних файлів до образу контейнера.

Комплексне застосування *Docker* та *Docker Compose* в проекті *DataMind* створює стандартизовану, повністю ізольовану та абсолютно відтворювану екосистему для кожного компонента системи. *Docker Compose* показано на рисунку 2.15.

Рисунок 2.15 – *Docker Compose*

Розробники отримують можливість миттєвого розгортання повного застосунку через виконання єдиної команди *docker compose up --build*, повністю

усуваючи необхідність ручного встановлення *Node.js* конкретних версій, глобальних залежностей або складної конфігурації операційної системи. *Docker Compose* конфігурацію показано на рисунку 2.16.

```
services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    volumes:
      - frontend_node_modules:/app/node_modules
    environment:
      - NODE_ENV=development
    depends_on:
      - backend
    networks:
      - app-network

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "3001:3001"
    volumes:
      - backend_node_modules:/usr/src/app/node_modules
    environment:
      - NODE_ENV=development
      - PORT=3001
    env_file:
      - ./.env
    networks:
      - app-network

volumes:
  frontend_node_modules:
    name: datamind_frontend_node_modules
  backend_node_modules:
    name: datamind_backend_node_modules

networks:
  app-network:
    driver: bridge
```

Рисунок 2.16 – *Docker Compose* конфігурація

РОЗДІЛ 3
ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ

3.1 Тестування ключових функцій

Тестування клієнтської частини проводилося з акцентом на компонентну архітектуру. Кожен ключовий *UI*-компонент зазнав ретельної перевірки на коректність візуального відображення та функціональну поведінку в різноманітних сценаріях використання. Компонент *ChatBot.jsx* тестувався на правильність управління станом діалогу, включаючи відображення індикатора набору тексту під час обробки відповіді агентом. Селектор *CustomModelSelector* проходив перевірку коректності рендерингу списку доступних моделей штучного інтелекту та реагування на вибір користувача. Особливу увагу було приділено тестуванню візуальних індикаторів стану, таких як активація підсвічування області при перетягуванні файлів та динамічна зміна інтерфейсу відповідно до поточного стану взаємодії.

Функціональне тестування користувацької взаємодії включало детальну перевірку процесу відправки повідомлень. Тестувалася послідовність від введення тексту користувачем до його відображення в історії чату та активації візуального індикатора очікування відповіді від сервера. Механізм завантаження файлів проходив комплексну перевірку, що охоплювала як традиційний спосіб через кнопку "Додати файли", так і інтуїтивний *drag-and-drop* функціонал. Система валідації файлів тестувалася на коректність перевірки типу файлів, обмеження розміру до 2 МБ та максимальної кількості до 10 файлів одночасно. Додатково перевірялася функціональність відображення списку прикріплених файлів у вигляді інтерактивних чіпів з можливістю видалення окремих елементів.

Тестування відображення відповідей агента включало перевірку коректного рендерингу *Markdown*-розмітки, функціональності інтерактивних пропозицій та роботи механізму копіювання повідомлень. Система обробки помилок на клієнтському рівні тестувалася через симуляцію різноманітних проблемних сценаріїв, включаючи спроби завантаження файлів неприпустимих типів,

перевищення лімітів розміру та кількості файлів, а також втрату з'єднання з сервером.

Валідація схеми відповіді від *OpenAI* здійснювалася через *aiResponseSchema* для гарантування відповідності отриманої *JSON*-структури очікуваному формату з обов'язковими полями *agent_response* та *suggestions*. Інтеграційне тестування охоплювало перевірку роботи ендпоінту */api/chat* у повному циклі з різноманітними комбінаціями параметрів, включаючи запити з файлами та без них, з різними обраними моделями, а також з некоректними даними для перевірки системи обробки помилок.

3.2 Приклади роботи застосунку та результати аналізу

Демонстрація функціональних можливостей веб-застосунку *DataMind* найкраще розкривається через розгляд конкретних сценаріїв взаємодії користувачів з Агентом *DataMind*, що ілюструють різноманітні аспекти його аналітичних здібностей.

Базовий сценарій ознайомлення демонструє початкову взаємодію користувача з системою без використання файлів даних. Коли користувач надсилає вітальний запит на кшталт "Привіт! Розкажи, будь ласка, про свої основні можливості", Агент *DataMind* генерує структуровану відповідь з використанням *Markdown*-форматування, детально описуючи свої ключові функціональні можливості. Відповідь включає інформацію про здатність проводити аналіз різноманітних типів даних, створювати аналітичні звіти, обробляти завантажені файли, виявляти тенденції та закономірності в даних, а також надавати персоналізовані рекомендації на основі проведеного аналізу. Система автоматично генерує релевантні пропозиції для подальшої взаємодії, такі як запити про підтримувані типи файлів або детальну інформацію про можливості створення звітів.

Сценарій аналізу одиничного файлу демонструє базові аналітичні можливості системи. При прикріпленні файлу статистика_відвідувань.*csv*, що містить структуровані дані з колонками "Дата", "Сторінка" та

"Кількість_переглядів", користувач може сформулювати аналітичний запит: "Проаналізуй файл статистика_відвідувань.*csv* та визнач сторінку з найбільшою кількістю переглядів за останній місяць". Агент *DataMind* обробляє наданий файл та генерує детальну аналітичну відповідь. Сценарій аналізу одиничного файлу показано на рисунку 3.1.

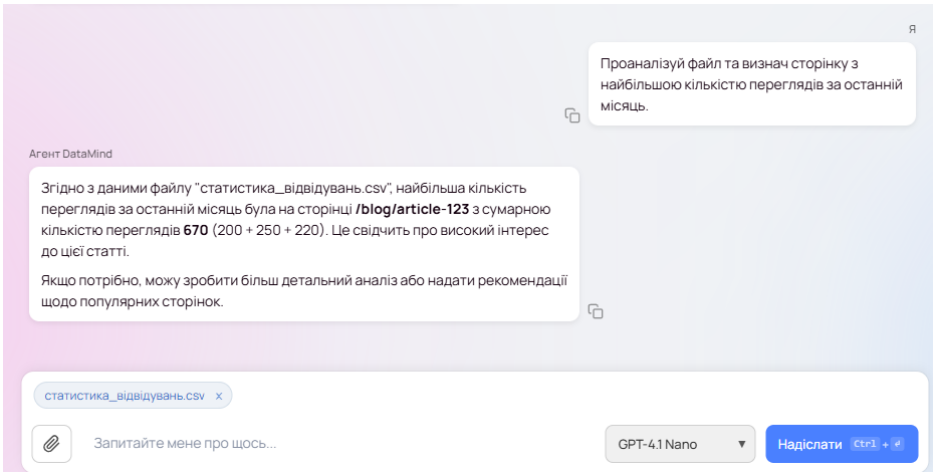

Рисунок 3.1 – Сценарій аналізу одиничного файлу

Складний сценарій мультифайлового аналізу показує здатність системи до комплексної обробки та зіставлення даних з різних джерел. При прикріпленні файлів співробітники.*json* з структурованими даними про персонал (*id*, ім'я, відділ, зарплата) та виконані_завдання.*csv* з інформацією про завдання (*id*_завдання, назва_завдання, *id*_співробітника, статус_завдання), користувач може сформулювати комплексний запит з множинними фільтрами: "З файлів співробітники.*json* та виконані_завдання.*csv* ідентифікуй всіх співробітників з відділу 'Маркетинг', які успішно завершили завдання та мають рівень винагороди понад 50000. Представ їхні імена та назви виконаних проектів." Агент *DataMind* здійснює складну операцію об'єднання даних, застосовує специфіковані фільтри та генерує структуровану відповідь: "На основі комплексного аналізу обох файлів ідентифіковано наступних співробітників відділу 'Маркетинг' з винагородою понад 50000, які успішно завершили призначені завдання: Ольга Петренко - проект 'Розробка рекламної кампанії *Q*3', Андрій Сидоренко - проект 'Аналіз ринку конкурентів'." Система пропонує варіанти для розширення аналізу, включаючи

підрахунок загальної кількості таких співробітників або застосування аналогічних критеріїв до інших відділів. Складний сценарій мультифайлового аналізу показано на рисунку 3.2.

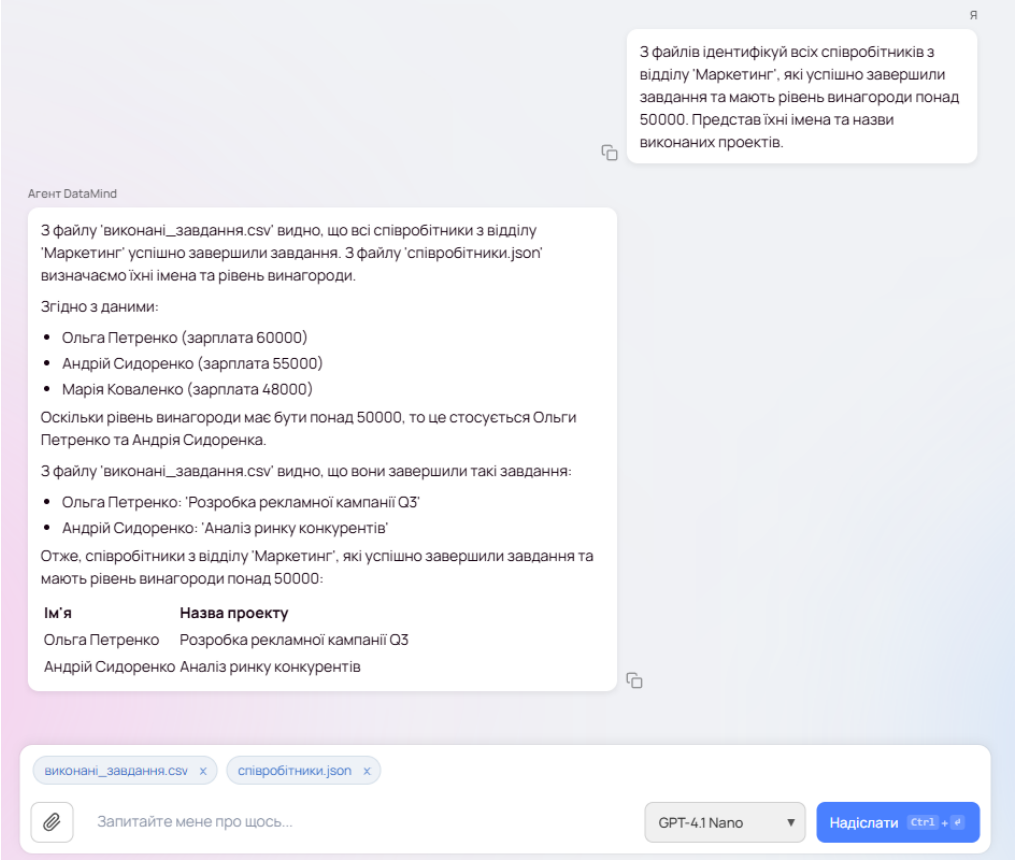

Рисунок 3.2 – Складний сценарій мультифайлового аналізу

3.3 Інструкція користувача

Ініціалізація та запуск системи здійснюється через *Docker*-контейнеризацію або локальне середовище розробки. При використанні *Docker* відкрийте термінал у кореневій директорії проєкту та виконайте команду *docker compose up --build* для автоматичного створення та запуску всіх необхідних контейнерів. Після успішної ініціалізації всіх сервісів запустіть веб-браузер та перейдіть за адресою *http://localhost*:3000 для доступу до користувацького інтерфейсу. При локальному запуску без контейнеризації переконайтеся у правильному функціонуванні *backend*-сервера на порту 3001 та *frontend*-застосунку на порту 3000. Збірку образів *Docker* показано на рисунку 3.3.

Рисунок 3.3 – Збірка образів *Docker*

Архітектура користувацького інтерфейсу побудована навколо інтуїтивної та функціональної структури. Центральну позицію займає область діалогу, де відображається повна історія взаємодії з Агентом *DataMind* у хронологічному порядку з автоматичним додаванням нових повідомлень знизу. Нижня панель містить систему введення, що включає текстове поле для формулювання запитів, кнопку прикріплення файлів для завантаження даних для аналізу, селектор вибору моделі штучного інтелекту для налаштування обробки запитів, та кнопку відправки для ініціації взаємодії з агентом.

Процес формулювання та відправки запитів до Агента *DataMind* характеризується простотою та гнучкістю. Введіть текст вашого запиту або завдання у спеціалізоване поле панелі введення, сформулювавши його максимально чітко та конкретно для отримання найбільш релевантних результатів. Після завершення формулювання натисніть кнопку "Надіслати" або використайте клавіатурне скорочення *Ctrl+Enter* (*Cmd+Enter* для користувачів *macOS*) для швидкої відправки. Одразу після ініціації запиту в області діалогу з'явиться візуальний індикатор обробки, що сигналізує про активну роботу Агента *DataMind* над генерацією відповіді.

Функціональність роботи з файлами даних є ключовою особливістю *DataMind*, що дозволяє проводити глибокий аналіз структурованої інформації. Система підтримує два основні способи завантаження файлів. Традиційний метод передбачає натискання кнопки "Додати файли" та вибір необхідних файлів через стандартний діалог операційної системи з можливістю множинного вибору до 10 файлів одночасно. Альтернативний метод *drag-and-drop* забезпечує інтуїтивну

взаємодію через перетягування файлів безпосередньо з файлового менеджера у вікно чату з автоматичною активацією візуальної області завантаження.

Система накладає специфічні обмеження на характеристики файлів для забезпечення оптимальної продуктивності та стабільності. Максимальний розмір індивідуального файлу обмежений 2 МБ для забезпечення швидкої обробки та передачі. Підтримувані формати включають *CSV*-файли для табличних даних, *JSON*-файли для структурованої інформації та *TXT*-файли для текстових даних. Загальна кількість файлів, що може бути прикріплена до одного запиту, обмежена 10 одиницями для підтримання керованості аналізу.

Кожне повідомлення в діалозі, незалежно від авторства, супроводжується іконкою копіювання для миттєвого збереження тексту до буфера обміну системи, що полегшує подальше використання результатів аналізу в зовнішніх документах або застосунках.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було успішно розроблено веб-застосунок «*DataMind*», призначений для інтерактивного аналізу даних за допомогою *AI*-агента. Робота була спрямована на вирішення проблеми доступності складних аналітичних інструментів для широкого кола користувачів шляхом створення інтуїтивного чат-інтерфейсу, інтегрованого з потужними моделями штучного інтелекту.

Спроєктована та реалізована клієнт-серверна архітектура застосунку «*DataMind*» продемонструвала переваги модульного підходу до розробки. Чітке розмежування відповідальності між фронтендом та бекендом, а також їх ефективна взаємодія через *RESTful API*, забезпечили високу модульність та гнучкість системи, що створює передумови для подальшого розвитку та масштабування застосунку.

Розроблена фронтенд частина на *React* успішно реалізувала всі заплановані функції користувацького інтерфейсу. Створений інтуїтивний дизайн включає зручний чат для спілкування з *AI*-агентом, надійні механізми завантаження файлів у форматах *CSV*, *JSON* та *TXT* з підтримкою технології *drag-and-drop*, гнучкий селектор *AI*-моделей та спеціалізований компонент для відображення відповідей *AI* з використанням *Markdown*-розмітки. Такий підхід забезпечив високу зручність використання системи.

Бекенд частина на *Node.js* та *Express* продемонструвала надійність та ефективність у обробці запитів користувачів. Реалізована інтеграція з *OpenAI API* включає складну логіку динамічного формування системних інструкцій з урахуванням вмісту завантажених файлів, що дозволяє *AI*-агенту надавати контекстуально релевантні відповіді. Особливу увагу приділено забезпеченню генерації відповідей від *AI* у строго визначеному *JSON*-форматі, що значно підвищує стабільність обробки на клієнті та загальну надійність системи.

Впровадження системи контейнеризації за допомогою *Docker* та *Docker Compose* стало важливим кроком у забезпеченні якості та надійності застосунку. Створені *Dockerfile* для фронтенд та бекенд сервісів, а також конфігураційний файл

docker-compose.yml для оркестрації, гарантують легкість розгортання системи в різних середовищах, консистентність робочого оточення та можливість ефективного масштабування застосунку відповідно до потреб користувачів.

Проведене комплексне тестування підтвердило високу якість розробленого рішення. Перевірка ключових функцій застосунку, включаючи завантаження файлів різних форматів, відправку та обробку запитів, взаємодію з *AI*-сервісами, коректне відображення відповідей та стабільну роботу системи в контейнеризованому середовищі, продемонструвала повну відповідність розробленого рішення поставленим вимогам та очікуванням користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *React* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://react.dev/* (дата звернення: 15.05.2025)
2. *Vite* – посібник з налаштування. [Електронний ресурс] – Режим доступу: *https://vitejs.dev/guide/* (дата звернення: 23.05.2025)
3. *Node.js* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://nodejs.org/en/docs/* (дата звернення: 12.05.2025)
4. *Express.js* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://expressjs.com/* (дата звернення: 28.05.2025)
5. *OpenAI API* – довідник *API*. [Електронний ресурс] – Режим доступу: *https://platform.openai.com/docs/api-reference* (дата звернення: 07.05.2025)
6. *Docker* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.docker.com/* (дата звернення: 19.05.2025)
7. *Docker Compose* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.docker.com/compose/* (дата звернення: 03.05.2025)
8. *Markdown Guide* – базовий синтаксис. [Електронний ресурс] – Режим доступу: *https://www.markdownguide.org/basic-syntax/* (дата звернення: 26.05.2025)
9. *JSON Schema* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://json-schema.org/* (дата звернення: 14.05.2025)
10. *Multer – NPM* пакет. [Електронний ресурс] – Режим доступу: *https://www.npmjs.com/package/multer* (дата звернення: 21.05.2025)
11. *React Markdown – NPM* пакет. [Електронний ресурс] – Режим доступу: *https://www.npmjs.com/package/react-markdown* (дата звернення: 09.05.2025)
12. *Remark GFM – NPM* пакет. [Електронний ресурс] – Режим доступу: *https://www.npmjs.com/package/remark-gfm* (дата звернення: 17.05.2025)
13. *MDN Web Docs – CSS* документація. [Електронний ресурс] – Режим доступу: *https://developer.mozilla.org/en-US/docs/Web/CSS* (дата звернення: 30.05.2025)

14. *Dotenv – NPM* пакет. [Електронний ресурс] – Режим доступу: *https://www.npmjs.com/package/dotenv* (дата звернення: 05.05.2025)

15. *CORS – NPM* пакет. [Електронний ресурс] – Режим доступу: *https://www.npmjs.com/package/cors* (дата звернення: 11.05.2025)

16. *OpenAI* – документація моделей. [Електронний ресурс] – Режим доступу: *https://platform.openai.com/docs/models* (дата звернення: 24.05.2025)

17. *MDN Web Docs – JavaScript* документація. [Електронний ресурс] – Режим доступу: *https://developer.mozilla.org/en-US/docs/Web/JavaScript* (дата звернення: 16.05.2025)

18. *OpenAI* – посібник з генерації тексту та *JSON* режиму. [Електронний ресурс] – Режим доступу: *https://platform.openai.com/docs/guides/text-generation/json-mode* (дата звернення: 08.05.2025)

ДОДАТОК А

Файл *server.js backend* сервісу

```
import express from "express";
import OpenAI from "openai";
import dotenv from "dotenv";
import cors from "cors";
import bodyParser from "body-parser";
import multer from "multer";

dotenv.config();

const app = express();
const port = process.env.PORT || 3001;

app.use(cors());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const NEW_MAX_FILES = 10;
const storage = multer.memoryStorage();
const upload = multer({
  storage: storage,
  limits: {
    files: NEW_MAX_FILES,
    fileSize: 1024 * 1024 * 2,
  },
  fileFilter: (req, file, cb) => {
    if (
      file.mimetype === "text/csv" ||
      file.mimetype === "application/json" ||
      file.mimetype === "text/plain"
    ) {
      cb(null, true);
    } else {
      cb(
        new Error(
          "Непідтримуваний тип файлу. Дозволені лише CSV, JSON та
TXT."
        ),
        false
      );
    }
  },
});

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

const storeOpenAIResponses =
  process.env.OPENAI_STORE_RESPONSES?.toLowerCase() === "true";
```

```
    const aiResponseSchema = {
      type: "object",
      properties: {
        agent_response: {
          type: "string",
          description:
                "Основна текстова відповідь від Агента DataMind українською мовою (якщо користувач не вказав іншу). Має бути чіткою, розмовною та релевантною. Може використовувати Markdown для форматування (наприклад, **жирний**, *курсив*, списки).",
        },
        suggestions: {
          type: ["array", "null"],
          items: { type: "string" },
          description:
               "Масив з 2-3 коротких, релевантних пропозицій для наступних дій/запитань українською (якщо не вказано інше). Якщо недоречно, має бути null.",
        },
      },
      required: ["agent_response", "suggestions"],
      additionalProperties: false,
    };

    const getSystemInstructions = (attachedFilesData) => {
      let fileContextInstructions = "";
      if (attachedFilesData && attachedFilesData.length > 0) {
            fileContextInstructions = "\n\n**Надані файли для аналізу:**\n";
        attachedFilesData.forEach((file) => {
          const MAX_FILE_CONTENT_LENGTH_FOR_PROMPT = 20000;
          let fileContentPreview = file.content;
                          if (fileContentPreview.length > MAX_FILE_CONTENT_LENGTH_FOR_PROMPT) {
            fileContentPreview =
                                    fileContentPreview.substring(0, MAX_FILE_CONTENT_LENGTH_FOR_PROMPT) +
              `\n... (файл ${file.name} був обрізаний через великий розмір, повний розмір: ${file.content.length} символів) ...`;
          }

           fileContextInstructions += `\n--- Файл: ${file.name} ---\n`;
          fileContextInstructions += `${fileContentPreview}\n`;
          fileContextInstructions += `--- Кінець файлу: ${file.name} ---\n`;
        });
        fileContextInstructions +=
          "\nВикористовуй вміст цих файлів для відповіді на запити користувача, якщо це доречно. Посилайся на назви файлів для контексту.\n";
```

```
    }

    return `Ти — «Агент DataMind», висококваліфікований AI-аналітик для роботи з платформою DataMind. Твоя роль — допомагати користувачам з аналізом даних, звітами, навігацією системою та відповідати на питання щодо організаційної інформації.
    ${fileContextInstructions}

    **МОВНА СТРАТЕГІЯ:**
    - Основна мова — українська (за замовчуванням)
    - Якщо користувач пише іншою мовою, відповідай тією ж мовою
    - Підтримуй професійний, але дружелюбний тон

    **РОБОТА З КОНТЕКСТОМ:**
    - Ретельно аналізуй історію чату для розуміння контексту та потреб користувача
    - Використовуй прикріплені файли як ключове джерело для відповідей
    - Посилайся на конкретні файли за назвою, коли це доречно
    - Виявляй тренди, патерни та аномалії в даних
    - Поєднуй інформацію з файлів із контекстом розмови

    **СТИЛЬ ВІДПОВІДЕЙ:**
    - Будь проактивним — пропонуй додаткові ідеї та рекомендації
    - Надавай конкретні, практичні поради
    - Структуруй складну інформацію логічно
    - Обов'язково використовуй Markdown для форматування: **жирний текст**, *курсив*, списки, таблиці, блоки коду, заголовки ## для покращення читабельності

    **СТРАТЕГІЯ ОБРОБКИ СКЛАДНИХ ЗАПИТІВ З КІЛЬКОМА ФАЙЛАМИ:**
    1. **Уважно визнач всі умови фільтрації** з запиту користувача.
    2. **Ідентифікуй ключові поля** для зв'язку даних між файлами.
    3. **Послідовно обробляй дані:**
        а. Спочатку застосуй фільтри до "основного" файлу.
        б. Потім, для відфільтрованих записів, знаходь пов'язані дані в інших файлах, використовуючи ключові поля, і також застосовуй до них відповідні фільтри.
        в. Будь дуже уважним при агрегації даних (суми, середні значення), переконайся, що агрегуєш лише релевантні, відфільтровані дані.
    4. **Якщо сумніваєшся або даних недостатньо для точної відповіді через обрізку файлів, чесно повідом про це** та вкажи, яку інформацію ти не зміг обробити. Краще визнати обмеження, ніж надати невірну інформацію.

    **agent_response:**
    - Основна ґрунтовна відповідь з використанням Markdown
    - Включай аналіз, висновки та практичні рекомендації
    - Посилайся на файли та попередні частини розмови

    **suggestions:**
```

```
- 2-3 короткі команди, які користувач може надіслати тобі
- Формулюй ВІД ІМЕНІ КОРИСТУВАЧА (як команди), а НЕ як питання від тебе
- Правильно: "Переведи на російську", "Створи графік", "Зроби коротше"
- Неправильно: "Чи потрібно перекласти?", "Хочете графік?"
- Якщо пропозиції недоречні — використовуй null

**КРИТИЧНО ВАЖЛИВО:**
- Відповідь ЛИШЕ у форматі JSON
- JSON має бути синтаксично правильним
- Дотримуйся точної схеми з полями agent_response та suggestions

Твоя головна мета – точність. Якщо ти не впевнений у відповіді через складність запиту або обмеження даних, краще задай уточнююче питання або вкажи на можливі неточності.
Будь активним партнером користувача в роботі з даними — не просто відповідай, а допомагай досягати цілей через експертний аналіз та корисні рекомендації.`;
};

app.post(
  "/api/chat",
  upload.array("attached_files", NEW_MAX_FILES),
  async (req, res) => {
    let chatHistoryString = req.body.messages;
    let selectedModel = req.body.selectedModel;
    let chatHistory;

    if (!chatHistoryString || !selectedModel) {
      return res.status(400).json({
        error: "Відсутні chatHistory (messages) або selectedModel у запиті",
      });
    }

    try {
      chatHistory = JSON.parse(chatHistoryString);
      if (!Array.isArray(chatHistory)) {
        throw new Error("chatHistory має бути масивом.");
      }
    } catch (e) {
      return res.status(400).json({
        error: "Некоректний формат chatHistory (messages).",
        details: e.message,
      });
    }

    const attachedFilesData = [];
    if (req.files && req.files.length > 0) {
      for (const file of req.files) {
        try {
```

```
            const fileContent = file.buffer.toString("utf8");
            attachedFilesData.push({
              name: file.originalname,
              content: fileContent,
              type: file.mimetype,
            });
          } catch (err) {
                        console.error(`Помилка читання файлу
${file.originalname}:`, err);
          }
        }

                    const systemInstructionsContent =
getSystemInstructions(attachedFilesData);

        const openAIInputs = chatHistory.map((msg) => ({
          role: msg.sender === "user" ? "user" : "assistant",
          content: msg.text,
        }));

        try {
          const completion = await openai.responses.create({
            model: selectedModel,
            instructions: systemInstructionsContent,
            input: openAIInputs,
            text: {
              format: {
                type: "json_schema",
                name: "DataMindAgentResponse",
                schema: aiResponseSchema,
                strict: true,
              },
            },
            temperature: 0.25,
            store: storeOpenAIResponses,
          });

          let aiOutputText = "";

          if (
            completion.output &&
            completion.output.length > 0 &&
            completion.output[0].content &&
            completion.output[0].content.length > 0
          ) {
            const firstContentItem = completion.output[0].content[0];
            if (firstContentItem.type === "output_text") {
              aiOutputText = firstContentItem.text;
            } else if (firstContentItem.type === "refusal") {
                        console.error("Відмова OpenAI API:",
firstContentItem.refusal);
```

```
            return res.status(503).json({
              error: "AI відмовився відповідати.",
              details: firstContentItem.refusal,
            });
          } else {
            console.warn(
              "Неочікуваний тип контенту від OpenAI:",
              firstContentItem.type
            );
          }
        }

        if (!aiOutputText && completion.output_text) {
          aiOutputText = completion.output_text;
        }

        if (!aiOutputText) {
          console.error(
            "Відсутній текстовий вивід від OpenAI:",
            JSON.stringify(completion, null, 2)
          );
          return res
            .status(500)
             .json({ error: "Відсутній текстовий вивід від моделі
AI." });
        }

        try {
          const parsedResponse = JSON.parse(aiOutputText);
          if (
            typeof parsedResponse.agent_response !== "string" ||
            !(
              Array.isArray(parsedResponse.suggestions) ||
              parsedResponse.suggestions === null
            )
          ) {
            throw new Error(
                 "Розпарсена відповідь не відповідає очікуваній
структурі схеми."
            );
          }
          if (
            Array.isArray(parsedResponse.suggestions) &&
             !parsedResponse.suggestions.every((s) => typeof s ===
"string")
          ) {
             throw new Error("Масив suggestions містить не рядкові
елементи.");
          }

          res.json(parsedResponse);
        } catch (parseError) {
```

```
            console.error("Не вдалося розпарсити JSON відповідь AI:",
parseError);
            console.error("Сирий текстовий вивід AI:", aiOutputText);
            res.status(500).json({
              agent_response: `Отримано некоректну JSON відповідь від
AI. Початок відповіді: ${aiOutputText.substring(
                  0,
                  150
              )}...`,
              suggestions: null,
             error_details: "Помилка розбору JSON відповіді від AI.",
            });
          }
        } catch (error) {
          console.error(
            "Помилка виклику OpenAI API:",
            error.response
              ? JSON.stringify(error.response.data || error.response,
null, 2)
               : error.message
          );
          let errorMessage = "Не вдалося отримати відповідь від AI.";
          let errorDetails = error.message;
          if (error.status === 401) {
            errorMessage =
              "Помилка автентифікації з OpenAI API. Перевірте ваш API
ключ.";
                errorDetails = "Невірний API ключ або помилка
автентифікації.";
          } else if (error.status === 429) {
            errorMessage =
                "Перевищено ліміт запитів до OpenAI API. Спробуйте
пізніше.";
            errorDetails = "Перевищено ліміт запитів.";
          } else if (
            error.response &&
            error.response.data &&
            error.response.data.error
          ) {
              errorDetails = error.response.data.error.message ||
errorDetails;
          }

          res.status(error.status || 500).json({
            error: errorMessage,
            details: errorDetails,
          });
        }
      }
    );

    app.use((err, req, res, next) => {
```

```
    if (err instanceof multer.MulterError) {
      return res.status(400).json({
        error: "Помилка завантаження файлу.",
        details: err.message,
      });
    } else if (err) {
      if (err.message.startsWith("Непідтримуваний тип файлу")) {
        return res.status(400).json({
          error: "Помилка типу файлу.",
          details: err.message,
        });
      }
      return res.status(500).json({
        error: "Внутрішня помилка сервера.",
        details: err.message,
      });
    }
    next();
  });

  app.listen(port, () => {
    console.log(`API сервер слухає на порту ${port}`);
    if (!process.env.OPENAI_API_KEY) {
      console.warn(
           "ПОПЕРЕДЖЕННЯ: Змінна середовища OPENAI_API_KEY не
встановлена!"
      );
    } else {
      console.log("OPENAI_API_KEY завантажено.");
    }
    console.log(
      `Зберігання відповідей OpenAI: ${
        storeOpenAIResponses ? "Увімкнено" : "Вимкнено"
      }`
    );
  });
```

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

ВІДГУК
керівника кваліфікаційної роботи

випускника спеціальності: 123 «Комп‘ютерна інженерія»

відділення: комп’ютерної та програмної інженерії

циклова комісія: комп’ютерних систем та мереж

Сергій ШОКОТЬКО
(ім’я, прізвище)

1. *Кваліфікаційна робота на тему «Інтеграція штучного інтелекту в веб-застосунок для аналізу даних» виконана в ініціативному порядку.*
2. *Метою кваліфікаційної роботи є проєктування та створення веб-застосунку з елементами штучного інтелекту для ефективного аналізу великих масивів даних.*
3. *Кваліфікаційна робота відповідає темі, затвердженій наказом начальника коледжу.*
4. *Кваліфікаційна робота виконана здобувачем освіти самостійно.*
5. *Здобувач освіти показав високі вміння роботи з літературними джерелами, аналіз теоретичного та практичного матеріалу, приймання обґрунтованих рішень, застосовування сучасних комп'ютерних інформаційних технологій.*
6. *Сергій ШОКОТЬКО показав достатній рівень дотримання вимог державних стандартів при виконанні кваліфікаційної роботи загалом та оформленні пояснювальної записки.*
7. *Рівень виконаної кваліфікаційної роботи заслуговує оцінку «добре», відповідає набутих випускником знань, умінь та навичок, вимогам освітньої характеристики фахівця і можливість присвоєння йому кваліфікації фахівця освітнього ступеня «Фаховий молодший бакалавр» спеціальності 123 «Комп‘ютерна інженерія».*

Керівник кваліфікаційної роботи

«____»_____2025 р. _____ Олександр МИТРОФАНОВ
(підпис) (ім’я, прізвище)

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

РЕЦЕНЗІЯ
на кваліфікаційну роботу

випускника спеціальності: 123 «Комп'ютерна інженерія»

відділення: комп'ютерної та програмної інженерії

циклова комісія: комп'ютерних систем та мереж

Сергій ШОКОТЬКО
(ім'я, прізвище)

1. Актуальність теми: Обрана тема кваліфікаційної роботи «Інтеграція штучного інтелекту в веб-застосунок для аналізу даних» є надзвичайно актуальною, оскільки сучасні IT-системи потребують ефективних інструментів аналізу великих обсягів інформації.

2. Кваліфікаційна робота відповідає темі, затвердженій наказом.

3. Завдання на виконання кваліфікаційної роботи виконано у повному обсязі.

4. У результаті виконання кваліфікаційної роботи було реалізовано веб-застосунок з інтегрованими модулями штучного інтелекту, що забезпечують аналіз і візуалізацію даних.

5. Якість виконання пояснювальної записки та ілюстративного (графічного) матеріалу відповідає вимогам Державних стандартів.

6. У кваліфікаційній роботі зроблено акцент на практичні результати, включаючи тестування системи на реальних наборах даних та аналіз ефективності використаних алгоритмів ШІ.

7. Кваліфікаційна робота заслуговує оцінку «добре».

Рецензент _____
(науковий ступінь, посада)

«____»_____ 2025 р. _____ Роман МІНЕНКО
(підпис) (ім'я, прізвище)

З рецензією ознайомлений _____ Сергій ШОКОТЬКО
(підпис) (ім'я, прізвище)