МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно- обчислювальних систем і управління

КОПИЛОВ Максим Олександрович

Програмний модуль автоматизованого збору та оцінки відповідей LLM-моделі / Software Module for Automated Collection and Evaluation of LLM Model Responses

спеціальність: 122 - Комп'ютерні науки
освітньо-професійна програма - Комп’ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-41
М.О. Копилов
_____
Науковий керівник:
к.т.н., доцент, П.Є. Биковий
_____

Кваліфікаційну роботу
допущено до захисту:
"___" _____ 20___ р.
Завідувач кафедри
_____Н. В. Дзюбановська

ТЕРНОПІЛЬ – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
КОПИЛОВУ Максиму Олександровичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Програмний модуль автоматизованого збору та оцінки відповідей LLM-моделі / Software Module for Automated Collection and Evaluation of LLM Model Responses

керівник роботи к.т.н., доцент, П.Є. Биковий
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- провести аналіз існуючих великих мовних моделей та дослідити сучасні підходи до автоматизованого оцінювання якості їх відповідей;
- обґрунтувати вибір методів збору, порівняння та оцінювання відповідей LLM-моделей, зокрема методів семантичного аналізу тексту та підходу LLM-as-a-Judge;
- спроєктувати архітектуру програмного модуля автоматизованого збору та оцінювання відповідей LLM-моделей і реалізувати інтеграцію із зовнішніми API мовних моделей;
- провести тестування програмного модуля та оцінити ефективність роботи методів автоматизованого оцінювання відповідей великих мовних моделей.

5. Перелік графічного матеріалу у роботі
- діаграма класів програмного модуля автоматизованого збору та оцінювання відповідей LLM-моделей.
- загальна структура програмного модуля автоматизованого збору та оцінювання відповідей LLM-моделей.
- схема алгоритму оцінювання відповідей із використанням Cosine Similarity та підходу LLM-as-a-Judge.

6.Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент _____ М. О. Копилов
(підпис) (прізвище та ініціали)

Керівник роботи _____ П. Є. Биковий
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль автоматизованого збору та оцінки відповідей LLM-моделі» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньо-професійної програми «Комп'ютерні науки» написана обсягом 71 сторінок, містить 17 ілюстрацій, 4 додатки та 27 використаних джерела.

Метою кваліфікаційної роботи є розробка програмного модуля автоматизованого збору, збереження та оцінювання відповідей великих мовних моделей на основі сучасних методів семантичного аналізу тексту та підходу LLM-as-a-Judge.

Об'єктом дослідження є процес автоматизованої взаємодії з великими мовними моделями та аналіз якості сформованих ними відповідей.

Предметом дослідження є методи, алгоритми та програмні засоби збору, збереження, порівняння й оцінювання відповідей LLM-моделей.

У результаті виконання роботи розроблено програмний модуль, який забезпечує автоматизоване надсилання тестових запитів до великих мовних моделей через API, збереження отриманих відповідей у базі даних та їх подальше оцінювання за допомогою метрик семантичної подібності. Для підвищення об'єктивності оцінювання реалізовано механізм LLM-as-a-Judge, що дозволяє використовувати окрему мовну модель у ролі експертного оцінювача якості відповідей.

Розроблена система може бути використана для тестування та порівняння сучасних мовних моделей, проведення бенчмаркінгу AI-сервісів, аналізу ефективності промптів та підтримки досліджень у галузі штучного інтелекту й обробки природної мови.

Ключові слова**:** ВЕЛИКІ МОВНІ МОДЕЛІ, LLM, CHATGPT, GEMINI, CLAUDE, SEMANTIC SIMILARITY, LLM-AS-A-JUDGE, NLP, API, ШТУЧНИЙ ІНТЕЛЕКТ, АВТОМАТИЗОВАНЕ ОЦІНЮВАННЯ.

ANNOTATION

The qualification work entitled “Software Module for Automated Collection and Evaluation of LLM Model Responses” for obtaining a Bachelor's degree in specialty 122 “Computer Science” under the educational program “Computer Science” consists of 71 pages, including 17 figures, 4 appendices, and 27 references.

The purpose of the qualification work is to develop a software module for automated collection, storage, and evaluation of responses generated by Large Language Models (LLMs) using modern semantic text analysis methods and the LLM-as-a-Judge approach.

The object of research is the process of automated interaction with large language models and the analysis of the quality of generated responses.

The subject of research is methods, algorithms, and software tools for collecting, storing, comparing, and evaluating responses produced by LLM-based systems.

As a result of the research, a software module was developed that provides automated submission of test queries to large language models through API interfaces, storage of generated responses in a database, and their further evaluation using semantic similarity metrics. To improve the objectivity of the evaluation process, the LLM-as-a-Judge approach was implemented, allowing an additional language model to perform the role of an expert evaluator.

The developed system can be used for benchmarking modern language models, evaluating AI services, testing prompt engineering strategies, and supporting research in the fields of Artificial Intelligence and Natural Language Processing. The modular architecture of the system enables easy integration of new language models and evaluation methods without significant modifications to the existing software structure.

Keywords: LARGE LANGUAGE MODELS, LLM, CHATGPT, GEMINI, CLAUDE, SEMANTIC SIMILARITY, LLM-AS-A-JUDGE, NATURAL LANGUAGE PROCESSING, API, ARTIFICIAL INTELLIGENCE, AUTOMATED EVALUATION.

ЗМІСТ

ЗМІСТ 6
ВСТУП 7
1 АНАЛІЗ СУЧАСНИХ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ ТА МЕТОДІВ ЇХ ОЦІНЮВАННЯ 10
1.1 Опис предметної області 10
1.2 Огляд і аналіз існуючих рішень 13
1.3 Постановка задачі 20
2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ 23
2.1 Загальна структура проєктованої системи 23
2.2 Алгоритмічне забезпечення 30
2.3 Інформаційне забезпечення 36
3 ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ 41
3.1 Розгортання інфраструктури та налаштування базових серверних сервісів 41
3.2 Проєктування та програмна реалізація користувацького інтерфейсу панелі адміністратора 45
3.3 Програмна реалізація механізмів асинхронної взаємодії з LLM та обробки винятків 49
3.4 Експериментальне тестування модуля та оцінка результатів впровадження 53
ВИСНОВКИ 56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 58
Додаток А Копія публікації 61
Додаток Б Специфікація API платформи «Talking Heads» (Postman Collection) 65
Додаток В Фрагмент вихідного коду серверної частини 66
Додаток Г Конфігураційний файли для розгортання інфраструктури системи 67
Додаток Д Програмний код головної сторінки користувацького інтерфейсу . 69

ВСТУП

Актуальність теми дослідження зумовлена стрімким розвитком систем штучного інтелекту, зокрема великих мовних моделей (Large Language Models, LLM), та їхнім масовим впровадженням у різноманітні сфери людської діяльності: від автоматизації бізнес-процесів і написання програмного коду до наукових досліджень та клієнтської підтримки. Сучасний ринок ШІ пропонує десятки різних моделей (як комерційних, так і з відкритим вихідним кодом), проте їхня поведінка залишається недетермінованою: вони можуть генерувати відмінні відповіді на однакові запити, допускати фактологічні помилки або так звані «галюцинації».

Наявність об’єктивних та автоматизованих інструментів для тестування LLM є критично важливою для розробників, дослідників (AI Researchers), QA-інженерів та бізнесу. Проте збір відповідей та їхня оцінка ускладнюються через відсутність єдиного стандарту взаємодії з різними API, потребу в обробці великих масивів текстових даних та складність формалізації критеріїв якості згенерованого тексту. Традиційні методи ручного тестування є вкрай неефективними, ресурсозатратними та суб'єктивними.

Розробка спеціалізованого програмного модуля, здатного ефективно збирати, структурувати та об’єктивно оцінювати відповіді різних LLM-моделей за визначеними метриками, є актуальним завданням, що має суттєве практичне значення для галузі інформаційних технологій.

Метою роботи є розробка масштабованого програмного модуля для автоматизованого збору, структурування та оцінювання відповідей різних великих мовних моделей на задані набори тестових запитів.

Для досягнення мети поставлено такі завдання:

1. Провести аналіз існуючих методів, метрик та інструментів оцінювання якості роботи LLM-моделей (бенчмаркінгу).
2. Розробити архітектуру програмного модуля, що забезпечує стійкість до відмов, масштабованість та гнучку інтеграцію з API різних мовних моделей.

3. Спроектувати та реалізувати підсистему автоматизованого надсилання запитів і збору згенерованих відповідей у єдину структуровану базу даних.

4. Реалізувати алгоритми автоматизованої оцінки відповідей на основі обраних класичних текстових метрик та підходів на базі ШІ (наприклад, LLM-as-a-judge).

5. Розробити інтерфейс (або механізми взаємодії) для управління наборами тестових даних (датасетами) та візуалізації результатів оцінювання.

6. Провести тестування розробленої системи та порівняльний аналіз кількох сучасних мовних моделей на тестовій вибірці.

Об'єктом дослідження є процес взаємодії з великими мовними моделями та аналіз якості генерування ними текстового контенту.

Предметом дослідження є методи, алгоритми та програмні засоби автоматизованого збору й оцінки відповідей LLM-моделей.

Методи дослідження базуються на використанні принципів системного аналізу, об'єктно-орієнтованого програмування та проєктування архітектури програмного забезпечення. У роботі застосовуються методи обробки природної мови (NLP) для аналізу тексту, статистичні методи для розрахунку метрик якості, а також технології API-інтеграції та асинхронного програмування для взаємодії з хмарними сервісами штучного інтелекту.

Практичне значення одержаних результатів полягає у створенні готового програмного інструменту, який може бути використаний IT-компаніями, аналітичними агентствами та науково-дослідними установами для автоматизованого бенчмаркінгу мовних моделей. Розроблений модуль дозволяє оптимізувати процес вибору найефективнішої LLM під конкретні бізнес-задачі, виявляти недоліки в логіці моделей та прискорювати процес тестування промптів (prompt engineering). Розроблені архітектурні рішення можуть бути легко масштабовані для підключення нових моделей та метрик у майбутньому.

Основні результати кваліфікаційної роботи апробовано на IV Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених

«Інтелектуальні комп’ютерні системи та мережі» (м. Тернопіль, 20 травня 2026 р.), де вони були представлені та опубліковані у вигляді тез «Програмний модуль автоматизованого збору та оцінювання відповідей великих мовних моделей» (Додаток А).

1 АНАЛІЗ СУЧАСНИХ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ ТА МЕТОДІВ ЇХ ОЦІНЮВАННЯ

1.1 Опис предметної області

У сучасних умовах стрімкого розвитку інформаційних технологій визначальне місце займають системи штучного інтелекту (ШІ), зокрема великі мовні моделі (Large Language Models, LLM). Вони довели свою ефективність у задачах обробки природної мови, генерації програмного коду, аналітиці даних та автоматизації різноманітних процесів. Провідні технологічні компанії розробили потужні моделі, такі як сімейство GPT-4 від OpenAI [1, 2], Gemini від Google DeepMind [3] та Claude від Anthropic [4], які стали стандартом де-факто для інтеграції інтелектуальних функцій у прикладні програмні рішення.

В основі функціонування більшості сучасних LLM лежить архітектура Transformer, концепція якої була вперше запропонована у 2017 році [5, 6]. Її ключовою інновацією є механізм уваги (Attention Mechanism), який дозволяє моделі динамічно визначати важливість окремих слів (токенів) у тексті, ефективно фіксуючи глибокі контекстуальні та логічні зв'язки. Завдяки цьому системи здатні не лише генерувати граматично правильні тексти, але й підтримувати складний контекст у межах розгорнутих інформаційних запитів користувачів.

Незважаючи на високу результативність, використання LLM супроводжується специфічними викликами. Через імовірнісну природу алгоритмів машинного навчання моделі демонструють недетерміновану поведінку. Вони здатні генерувати різні варіанти відповідей на абсолютно ідентичні запити, а також схильні до генерування правдоподібного, але фактологічно хибного контенту (так званих «галюцинацій»). Загалом, відповіді різних моделей суттєво відрізняються за точністю, повнотою розкриття теми, структурованістю та рівнем релевантності.

Для мінімізації проблеми «галюцинацій» та підвищення фактологічної точності в сучасних системах активно впроваджується архітектура генерації з доповненим пошуком (Retrieval-Augmented Generation, RAG) [8-10]. Цей підхід дозволяє мовній

моделі звертатися до спеціалізованих зовнішніх баз знань перед формуванням остаточної відповіді. Загальна схема взаємодії компонентів у межах архітектури RAG наведена на рис. 1.1.

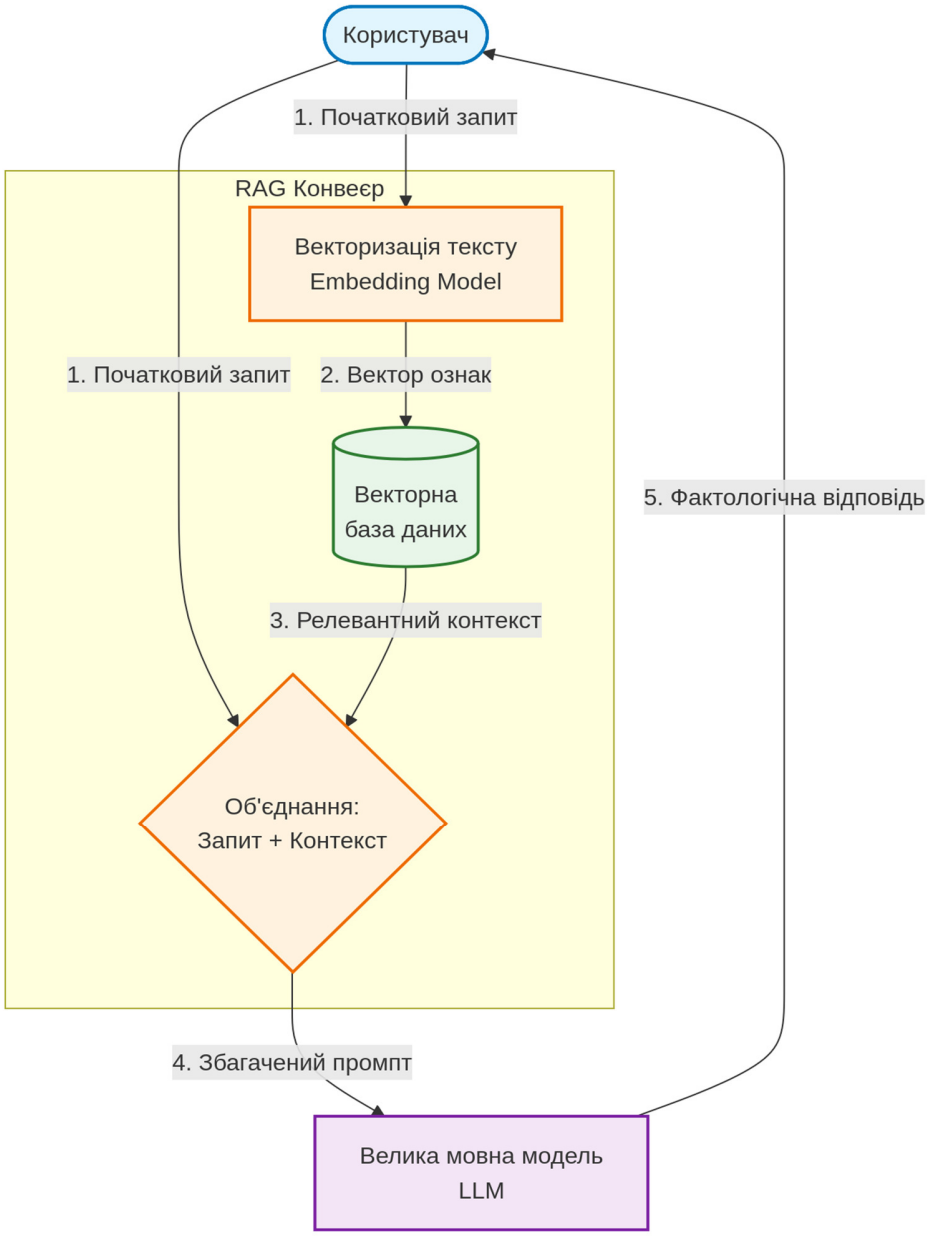

Рисунок 1.1 – Загальна схема архітектури генерації з доповненим пошуком (RAG)

Як видно зі схеми, процес починається з векторизації запиту користувача, після чого відбувається семантичний пошук релевантного контексту у векторній базі даних. Знайдена інформація об'єднується з початковим запитом, утворюючи так званий

збагачений промпт (Enriched Prompt), який передається до великої мовної моделі для генерації фінальної відповіді.

У практичних задачах — під час розробки подібних програмних систем або проведення досліджень — виникає критична потреба у порівнянні та оцінюванні (бенчмаркінгу) результатів роботи різних LLM. Традиційне ручне тестування передбачає індивідуальне надсилання запитів через вебінтерфейси, копіювання відповідей та їх суб'єктивну оцінку. Цей підхід є вкрай ресурсозатратним, він не масштабується і не гарантує репрезентативності результатів під час роботи з великими обсягами тестових даних.

Для розв'язання цієї проблеми предметна область вимагає переходу до автоматизованих систем оцінювання. Сучасні методології передбачають як використання класичних статистичних метрик тексту, так і застосування передових парадигм, зокрема підходу «LLM-as-a-Judge» (оцінювання великими мовними моделями) [11], де потужна модель виступає арбітром для аналізу відповідей інших систем.

Автоматизація цього процесу потребує створення надійного програмного середовища. Воно має через API-інтерфейси асинхронно взаємодіяти з різними провайдерами ШІ, агрегувати отримані текстові дані та зберігати їх у структурованому вигляді (наприклад, у реляційних базах даних [12]) для подальшого проходження через конвеєр оцінювання.

Отже, розробка спеціалізованого програмного модуля для автоматизованого збору та оцінки відповідей LLM-моделей є актуальним і необхідним завданням. Такий інструмент дозволить дослідникам та розробникам значно оптимізувати процес вибору найефективнішої моделі під конкретні бізнес-задачі, прискорити процес тестування підказок (prompt engineering) та забезпечити об'єктивний, відтворюваний аналіз якості згенерованого контенту.

1.2 Огляд і аналіз існуючих рішень

Упродовж останніх років розвиток технологій штучного інтелекту призвів до появи нового покоління систем обробки природної мови, здатних аналізувати текстову інформацію, враховувати контекст повідомлень та генерувати змістовні відповіді на запити користувачів. Подібні системи отримали назву великих мовних моделей (Large Language Models, LLM) та сьогодні активно використовуються у сфері програмування, освіти, бізнес-аналітики, наукових досліджень та інформаційних сервісах. Їх розвиток став можливим завдяки появі нових архітектур нейронних мереж та суттєвому збільшенню обчислювальних ресурсів, необхідних для навчання моделей на великих масивах текстових даних.

Одним із ключових етапів розвитку сучасних мовних моделей стало впровадження архітектури Transformer, яка була запропонована у 2017 році [13]. Даний підхід суттєво відрізняється від попередніх методів обробки природної мови та базується на використанні механізму уваги (Attention Mechanism). Завдяки цьому модель отримує можливість аналізувати взаємозв’язки між словами незалежно від їх розташування у тексті, що забезпечує більш якісне розуміння контексту та підвищує ефективність генерації відповідей.

На відміну від рекурентних нейронних мереж, які обробляють текст послідовно, архітектура Transformer дозволяє виконувати паралельну обробку даних. Це значно прискорює процес навчання моделей та дозволяє працювати з набагато більшими обсягами інформації. Саме ця архітектура стала основою для створення більшості сучасних LLM-систем, включаючи GPT, Gemini та Claude. На рисунку 1.2 наведено загальну архітектуру Transformer, яка демонструє взаємодію механізму багатоголової уваги (Multi-Head Attention), шарів нормалізації, повнозв’язних нейронних мереж та блоків кодування і декодування інформації.

Поява архітектури Transformer створила передумови для розробки нового покоління мовних моделей, які здатні виконувати широкий спектр завдань з обробки

текстової інформації. Однією з найбільш відомих та поширених систем такого типу є ChatGPT [1, 2].

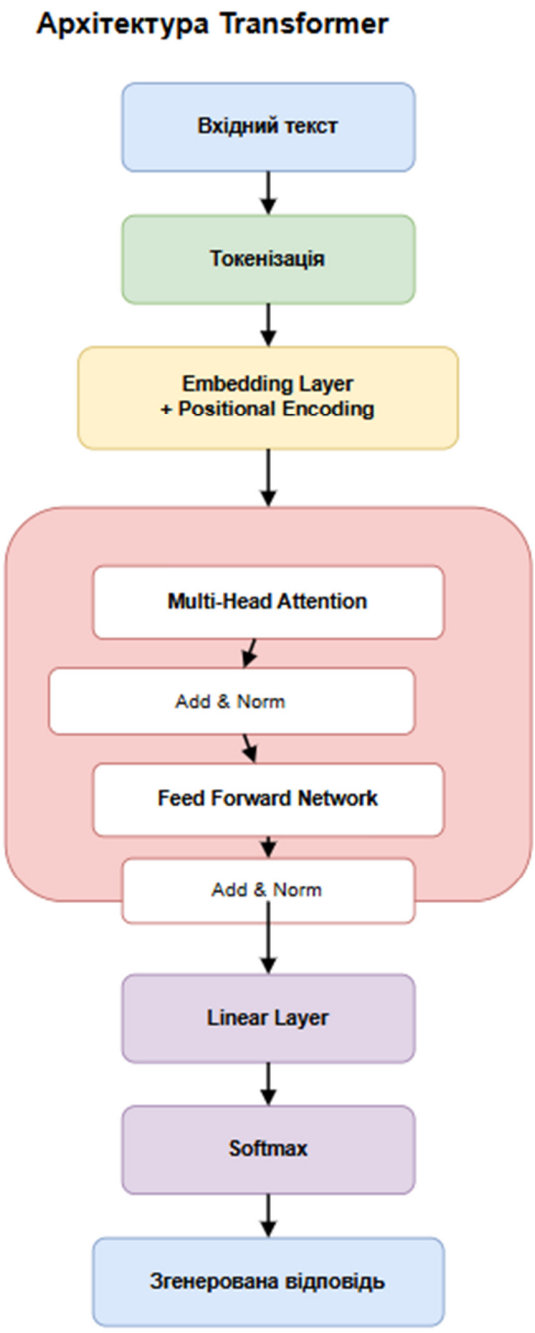

Рисунок 1.2 – Архітектура Transformer

ChatGPT є великою мовною моделлю, що побудована на основі сімейства моделей GPT та призначена для ведення діалогу з користувачем природною мовою. Система дозволяє генерувати текстові відповіді, здійснювати аналіз інформації,

виконувати переклад між мовами, створювати програмний код та допомагати у вирішенні різноманітних інформаційних завдань. Однією з головних особливостей ChatGPT є здатність підтримувати контекст діалогу, що забезпечує більш логічну та послідовну взаємодію з користувачем.

Завдяки доступності API модель активно використовується у програмних продуктах різного призначення. Сучасні інформаційні системи інтегрують ChatGPT для автоматизації технічної підтримки, генерації документації, аналізу текстових даних та створення інтелектуальних помічників. Висока якість сформованих відповідей зробила дану систему одним із найпоширеніших інструментів генеративного штучного інтелекту. На рисунку 1.3 наведено приклад взаємодії користувача з мовною моделлю ChatGPT через вебінтерфейс системи.

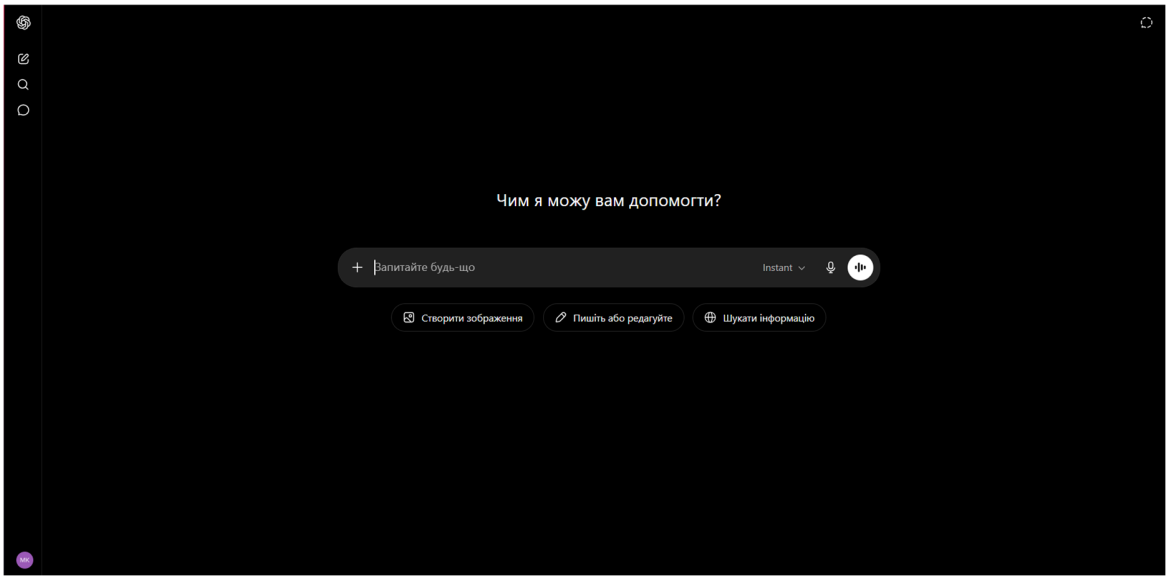

Рисунок 1.3 – Інтерфейс ChatGPT

Важливим конкурентом ChatGPT є мовна модель Gemini, розроблена компанією Google DeepMind [3]. Дана система являє собою мультимодальну модель штучного інтелекту, яка здатна працювати не лише з текстовими даними, а й із зображеннями, таблицями та іншими типами інформації. Подібний підхід дозволяє значно розширити можливості використання моделі та забезпечити більш комплексний аналіз інформації.

Однією з переваг Gemini є глибока інтеграція з екосистемою Google, що дозволяє використовувати модель у хмарних сервісах, корпоративних інформаційних системах та програмних продуктах різного призначення. Крім того, модель підтримує програмні інтерфейси для інтеграції із зовнішніми застосунками та забезпечує високий рівень продуктивності при роботі з великими обсягами інформації. На рисунку 1.4 наведено приклад використання мовної моделі Gemini для обробки текстових запитів користувача.

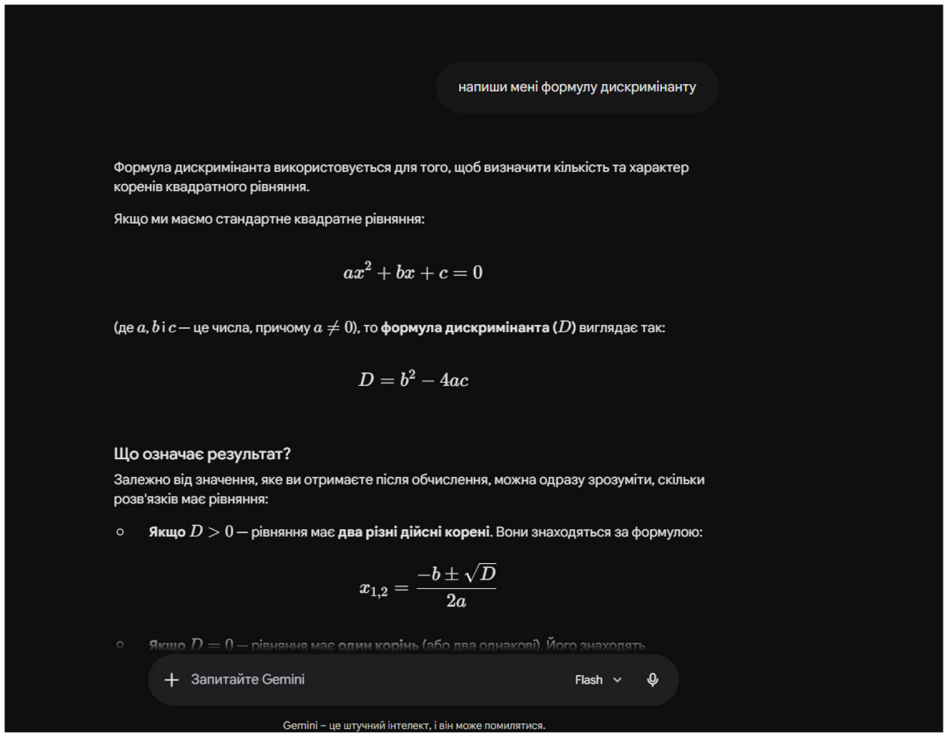

Рисунок 1.4 – Інтерфейс Gemini

Ще одним представником сучасних великих мовних моделей є Claude, розроблена компанією Anthropic [4]. На відміну від багатьох інших систем, дана модель створювалася з акцентом на безпечність використання та контроль якості сформованих відповідей. Під час навчання та налаштування системи значна увага приділялася мінімізації ризику генерації недостовірної або потенційно небезпечної інформації.

Важливою перевагою Claude є підтримка великих контекстних вікон, що дозволяє аналізувати значні за обсягом документи в межах одного діалогу. Це робить

модель ефективним інструментом для роботи з технічною документацією, аналітичними звітами та іншими великими текстовими матеріалами.

На рисунку 1.5 наведено приклад взаємодії користувача з мовною моделлю Claude під час аналізу текстових даних.

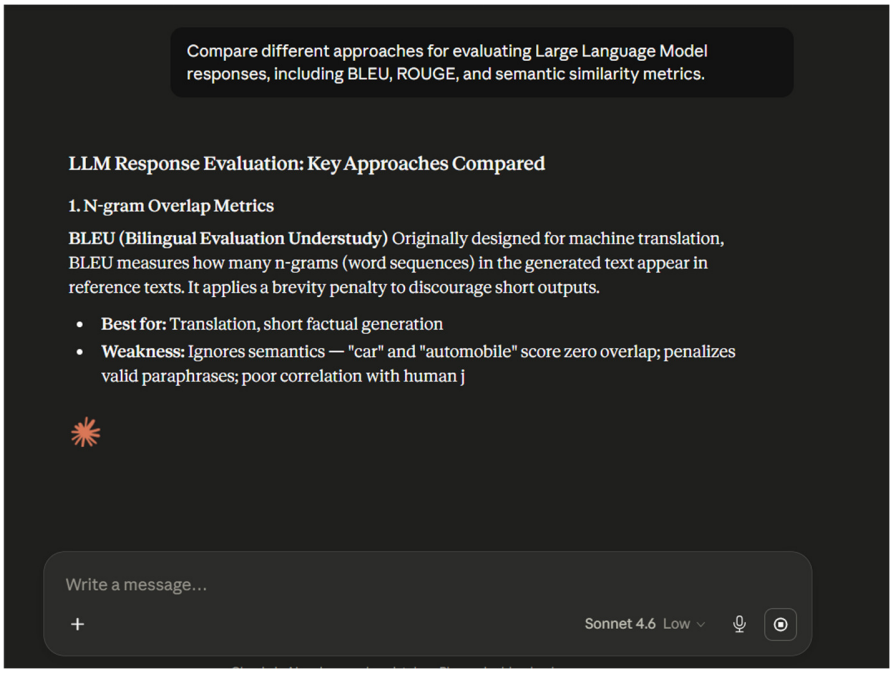

Рисунок 1.5 – Інтерфейс Claude

Незважаючи на високий рівень розвитку сучасних мовних моделей, важливою проблемою залишається оцінювання якості сформованих відповідей. Різні моделі можуть генерувати різні результати навіть за однакових вхідних даних, що створює необхідність у використанні спеціалізованих методів аналізу та порівняння отриманих відповідей.

Одними з найпоширеніших методів автоматичного оцінювання є метрики BLEU та ROUGE. Дані підходи використовують статистичний аналіз тексту та дозволяють визначити рівень відповідності між отриманою та еталонною відповідями. Проте такі методи орієнтовані переважно на пошук збігів між словами та не завжди коректно оцінюють змістову близькість текстів.

Для підвищення точності оцінювання у сучасних системах широко використовуються методи семантичного аналізу. Зокрема, застосовуються векторні

представлення тексту (embeddings), які дозволяють описувати зміст повідомлень у вигляді числових векторів. Після цього між відповідями обчислюється косинусна подібність (Cosine Similarity), яка відображає рівень їх семантичної близькості.

На рисунку 1.6 наведено загальну схему автоматизованого оцінювання відповідей мовних моделей із використанням еталонних даних та метрик семантичної подібності.

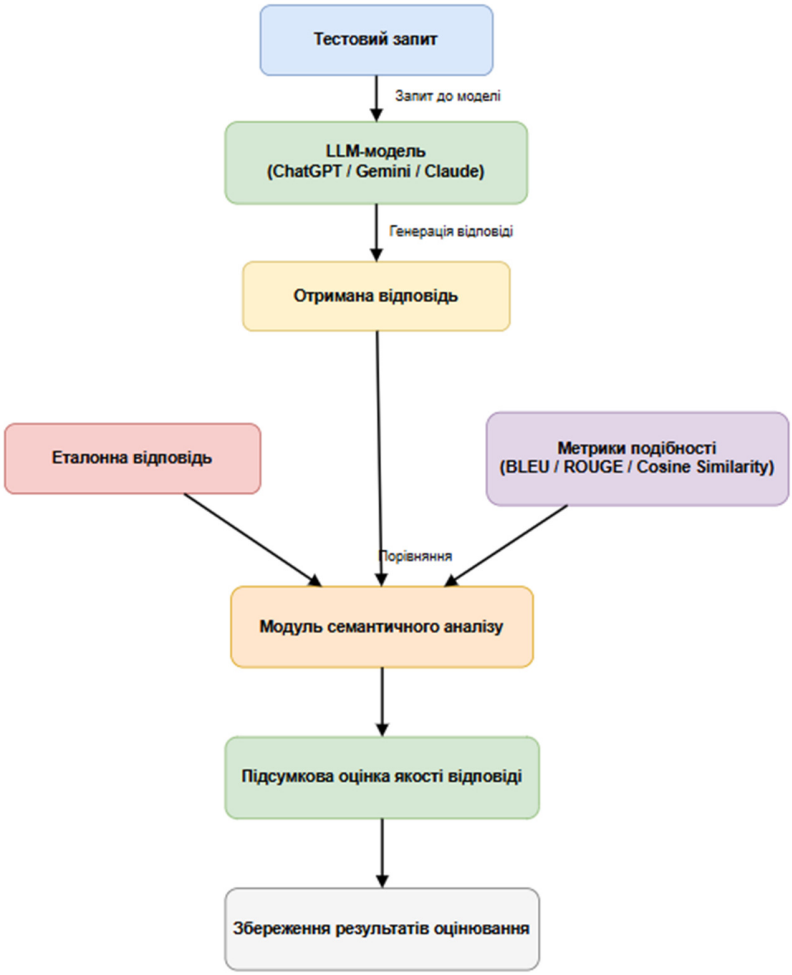

Рисунок 1.6 – Схема автоматизованого оцінювання відповідей

Окремим напрямом розвитку систем оцінювання є концепція LLM-as-a-Judge [9]. Суть даного підходу полягає у використанні однієї мовної моделі для оцінювання відповіді, сформованої іншою моделлю. Такий підхід дозволяє автоматизувати процес аналізу великої кількості результатів та суттєво скоротити час проведення досліджень.

Паралельно з розвитком мовних моделей активно розвиваються програмні інструменти для інтеграції та тестування LLM-систем. Одним із найбільш популярних рішень є LangChain [9], який забезпечує можливість створення складних сценаріїв взаємодії між мовними моделями, зовнішніми джерелами даних та інформаційними системами. Використання подібних платформ дозволяє реалізовувати багатокомпонентні інтелектуальні застосунки та автоматизувати процес роботи з текстовими даними.

Для Java-застосунків перспективним напрямом є використання фреймворку Spring AI (на базі Spring Boot) [13]. Даний інструмент забезпечує уніфікований механізм інтеграції сучасних мовних моделей у середовище Spring Boot та спрощує процес взаємодії із зовнішніми AI-сервісами через API. Завдяки цьому розробники можуть створювати інтелектуальні програмні системи без необхідності реалізації великої кількості допоміжного коду.

Для забезпечення надійності, відтворюваності результатів тестування та зручності розгортання подібних програмних систем сучасним стандартом є використання технологій контейнеризації. Оскільки модуль автоматизованого збору та оцінювання відповідей потребує інтеграції з реляційною базою даних (зокрема, PostgreSQL) та забезпечення стабільної асинхронної взаємодії із зовнішніми API, доцільним є застосування платформи Docker [14]. Контейнеризація дозволяє ізолювати програмне середовище, упакувавши застосунок разом з усіма необхідними залежностями. Це гарантує стабільну роботу системи незалежно від особливостей локального середовища розробника чи цільового сервера, що є критично важливим фактором для збереження цілісності та об'єктивності аналітичних даних під час тестування великих мовних моделей.

Таким чином, проведений аналіз показав, що сучасні великі мовні моделі забезпечують високий рівень якості генерації тексту та мають широкі можливості інтеграції у програмні системи. Водночас проблема автоматизованого оцінювання відповідей залишається актуальною та потребує створення спеціалізованих програмних засобів для збору, аналізу та порівняння результатів роботи різних LLM-

моделей. Саме вирішенню даної задачі присвячено розробку програмного модуля автоматизованого збору та оцінювання відповідей великих мовних моделей.

1.3 Постановка задачі

Актуальність дослідження обумовлена стрімким розвитком технологій генеративного штучного інтелекту та широким впровадженням великих мовних моделей (LLM) у різні сфери діяльності людини. Сучасні LLM-системи активно використовуються для автоматизації обробки інформації, створення програмного коду, підтримки прийняття рішень та побудови інтелектуальних сервісів. Водночас постійне збільшення кількості доступних мовних моделей створює критичну потребу у проведенні їх об’єктивного порівняння та оцінювання якості сформованих відповідей.

Оскільки мовні моделі генерують текст на основі ймовірнісних та статистичних закономірностей, отримані результати можуть суттєво відрізнятися за змістом, точністю та логікою навіть за однакових вхідних запитів. Однією з основних проблем їх використання є відсутність універсального, швидкого та об'єктивного механізму оцінювання якості згенерованого контенту.

Традиційно оцінювання відповідей здійснюється двома шляхами:

1. Ручне (експертне) оцінювання — дозволяє глибоко проаналізувати правильність інформації, логічність викладення та відповідність контексту. Проте цей підхід потребує значних витрат часу та людських ресурсів, що робить його вкрай неефективним і фінансово нерентабельним при масштабуванні або тестуванні на великих наборах даних (датасетах).

2. Автоматизоване лексичне оцінювання — базується на статистичних метриках та алгоритмах прямого посимвольного чи послівного порівняння з еталонними відповідями (наприклад, метрики BLEU, ROUGE або METEOR). Хоча такі системи є швидкими, вони виявляються неточними в умовах природної мови,

оскільки фіксують лише структурний збіг тексту і абсолютно не враховують синонімію, перефразування та семантичну гнучкість відповідей сучасних LLM.

Для вирішення цієї проблеми у сучасних інформаційних системах виникає необхідність застосування комбінованого підходу, що включає методи семантичного аналізу тексту та використання концепції «LLM-as-a-Judge» (оцінювання мовною моделлю).

Впровадження векторних представлень (embeddings) дозволяє трансформувати природну мову у багатовимірний математичний простір, де обчислення відстані між векторами (наприклад, косинусної подібності) відображає їхню змістову спорідненість. Це ефективно вирішує проблему синонімії. Однак сама лише математична близькість не завжди здатна виявити логічні хиби, порушення причинно-наслідкових зв’язків або приховані галюцинації у тексті.

Саме тому критично важливим є залучення парадигми LLM-as-a-Judge. Використання потужної мовної моделі вищого порядку у ролі автоматизованого експерта дозволяє наблизити якість машинного оцінювання до рівня професійного людського аудиту. Модель-суддя здатна аналізувати відповіді за складними багатокритеріальними метриками (фактологічна точність, повнота, дотримання заданої ролі чи формату), керуючись суворими системними інструкціями.

Реалізація такого комплексного підходу потребує створення спеціалізованого програмного забезпечення. Існуючі на ринку інструменти здебільшого орієнтовані на обробку одиничних ручних запитів і не забезпечують повного циклу масового бенчмаркінгу. Розроблюваний програмний модуль має стати єдиним оркеструючим середовищем, здатним асинхронно маршрутизувати сотні тестових запитів до різних провайдерів (OpenAI, Google, Anthropic), агрегувати відповіді, розраховувати математичні метрики, ініціювати експертну перевірку моделлю-суддею та персистентно зберігати всі аналітичні результати і конфігурації у реляційній базі даних.

Метою роботи є розробка програмного модуля автоматизованого збору та оцінювання відповідей великих мовних моделей, який забезпечує асинхронне

отримання результатів роботи різних LLM-систем через API, їх збереження, порівняння з еталонними відповідями та автоматизоване визначення якості згенерованого тексту на основі сучасних метрик семантичної подібності та експертних моделей.

Відповідно до поставленої мети, у кваліфікаційній роботі необхідно вирішити такі основні завдання:

1. Проаналізувати існуючі архітектури великих мовних моделей та сучасні методи автоматизованого оцінювання згенерованого тексту.
2. Сформулювати функціональні та нефункціональні вимоги до програмного модуля збору й аналізу даних.
3. Розробити архітектуру програмної системи, визначити структуру бази даних для збереження тестових запитів, еталонних та згенерованих відповідей.
4. Здійснити програмну реалізацію модуля інтеграції з API зовнішніх мовних моделей (OpenAI, Google Gemini, Anthropic Claude).
5. Реалізувати алгоритми оцінювання зібраних відповідей на основі семантичного аналізу (Embeddings) та підходу LLM-as-a-Judge.
6. Провести тестування розробленого програмного модуля та проаналізувати результати його роботи.

Об’єктом дослідження є процес автоматизованого збору та багатофакторного оцінювання відповідей великих мовних моделей під час виконання стандартизованих тестових запитів.

Предметом дослідження виступають методи, алгоритми та програмні засоби автоматизованої взаємодії з API мовних моделей, а також інструменти оцінювання згенерованого тексту на основі еталонних даних і метрик семантичної подібності.

У результаті виконання дослідження очікується створення масштабованого програмного модуля, що дозволить автоматизувати процес тестування генеративних систем штучного інтелекту, суттєво скоротити час на аналіз їхньої ефективності та забезпечити високий рівень об'єктивності завдяки використанню сучасних методів оцінювання.

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Загальна структура проєктованої системи

Одним із ключових етапів розробки програмного модуля автоматизованого збору та оцінювання відповідей великих мовних моделей є проєктування його архітектури. Від структури системи залежить ефективність взаємодії між окремими компонентами, швидкість обробки запитів, можливість інтеграції нових мовних моделей та масштабованість програмного забезпечення. Тому під час проєктування особлива увага приділяється організації інформаційних потоків і розподілу функціональності між окремими модулями системи.

Основною метою розроблюваного програмного модуля є автоматизація процесу формування запитів, отримання відповідей від великих мовних моделей та їх подальшого оцінювання. На відміну від традиційного підходу, за якого користувач взаємодіє з кожною мовною моделлю окремо, запропонована система забезпечує централізовану обробку запитів, автоматичний збір відповідей та їх аналіз за допомогою спеціалізованих методів оцінювання.

Архітектура системи побудована за модульним принципом, який передбачає розподіл функціональності між окремими незалежними компонентами. Такий підхід забезпечує високу гнучкість системи, спрощує її супровід та дозволяє розширювати функціональні можливості без необхідності внесення суттєвих змін у код системи.

На верхньому рівні архітектури знаходиться інтерфейс користувача (Talking Heads Interface), який забезпечує взаємодію користувача із системою. Через інтерфейс здійснюється введення запитів та отримання результатів їх обробки. Сформований запит надходить до центрального компонента системи — API Gateway.

API Gateway виконує функції маршрутизації запитів та координації взаємодії між усіма компонентами програмного модуля. Використання єдиної точки входу дозволяє спростити обробку запитів, централізувати керування сервісами та забезпечити стандартизований механізм обміну даними між окремими модулями.

Після отримання запиту API Gateway передає його до модуля Retrieval-

Augmented Generation (RAG). Даний компонент реалізує механізм пошуку та формування додаткового контексту для подальшої генерації відповіді. Основним призначенням RAG-модуля є підвищення релевантності відповідей шляхом використання зовнішніх джерел знань.

Для формування контексту RAG-модуль взаємодіє з векторною базою даних, реалізованою на основі PostgreSQL із розширенням pgvector. У векторному сховищі зберігаються семантичні представлення текстових даних, що дозволяє виконувати пошук інформації не лише за ключовими словами, а й за змістовою подібністю. У результаті формується набір релевантних даних, які використовуються для розширення початкового запиту користувача.

Після завершення пошуку контекстна інформація об’єднується з початковим запитом та формує так званий збагачений промпт. Отриманий промпт передається до мовних моделей через модуль LLM Models API. Даний компонент забезпечує взаємодію з великими мовними моделями та уніфікує процес роботи з різними сервісами штучного інтелекту.

У межах системи передбачається підтримка декількох мовних моделей, зокрема GPT-4, Claude та інших сучасних LLM-сервісів. Використання декількох моделей дозволяє виконувати порівняльний аналіз якості відповідей та досліджувати особливості роботи різних підходів до генерації тексту.

Отримані відповіді надходять до модуля аналітики та оцінювання (Evaluator Engine). Даний компонент є одним із ключових елементів архітектури, оскільки саме на цьому етапі здійснюється аналіз результатів роботи мовних моделей та визначення їх якості. Для оцінювання відповідей використовуються як математичні методи аналізу тексту, так і сучасні підходи, засновані на використанні інших мовних моделей.

Одним із методів оцінювання є обчислення косинусної подібності між отриманою відповіддю та еталонними даними. Для цього використовується компонент Cosine Similarity, який реалізує математичні алгоритми визначення рівня семантичної близькості між текстовими документами. Отримані значення

використовуються як кількісна характеристика якості відповіді.

Крім математичних методів аналізу система підтримує підхід LLM-as-a-Judge. Сутність даного підходу полягає у використанні окремої великої мовної моделі як експертного оцінювача. У цьому випадку відповідь аналізується додатковою моделлю, яка формує якісну оцінку результату та визначає рівень відповідності відповіді поставленому завданню. Поєднання математичних метрик та оцінювання за допомогою мовної моделі дозволяє підвищити об'єктивність аналізу та забезпечити більш комплексне оцінювання якості відповідей [15, 16].

Загальна структура взаємодії основних компонентів програмного модуля наведена на рисунку 2.1.

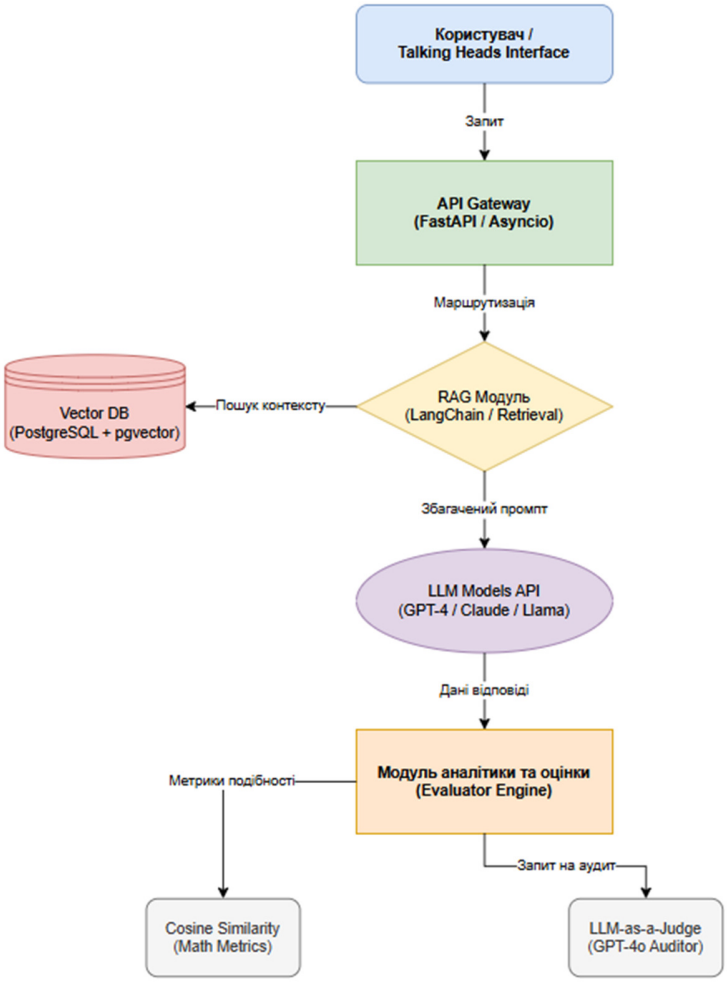

Рисунок 2.1 – Загальна структура програмного модуля автоматизованого збору та оцінювання відповідей LLM-моделей

Запропонована архітектура забезпечує повний цикл обробки інформації: від отримання запиту користувача та пошуку релевантного контексту до генерації відповіді великими мовними моделями та її подальшого автоматизованого оцінювання. Використання модульного підходу дозволяє забезпечити гнучкість, масштабованість та можливість інтеграції нових мовних моделей і методів аналізу без суттєвих змін загальної структури системи [17].

Для детального розуміння логіки роботи розроблюваного програмного забезпечення було проведено комплексне моделювання системи, яке включає функціональну декомпозицію, структурну організацію, аналіз потоків даних та поведінкові моделі взаємодії.

На початковому етапі проєктування було сформовано перелік ключових функцій, які має виконувати система для досягнення поставленої мети. Функціональна декомпозиція модуля охоплює п'ять основних підпроцесів:

1. Управління вхідними даними: отримання запитів від користувача через вебінтерфейс, первинна валідація та нормалізація тексту.

2. Формування контексту (RAG): генерація векторних представлень (ембеддінгів) для запиту, звернення до векторної бази даних та вибірка релевантних фрагментів знань.

3. Оркестрація LLM: адаптація збагаченого промпту під специфічні формати різних провайдерів ШІ (OpenAI, Anthropic, Google) та асинхронне відправлення запитів через API [18].

4. Багатофакторне оцінювання: розрахунок метрик математичної семантичної близькості (Cosine Similarity) та ініціація експертного аудиту за допомогою моделі-судді (LLM-as-a-Judge).

5. Збереження та аналітика: десеріалізація результатів, запис оцінок та конфігурацій у реляційну базу даних, формування підсумкових метрик для користувача.

Графічне відображення функціональної схеми системи наведено на рис. 2.2.

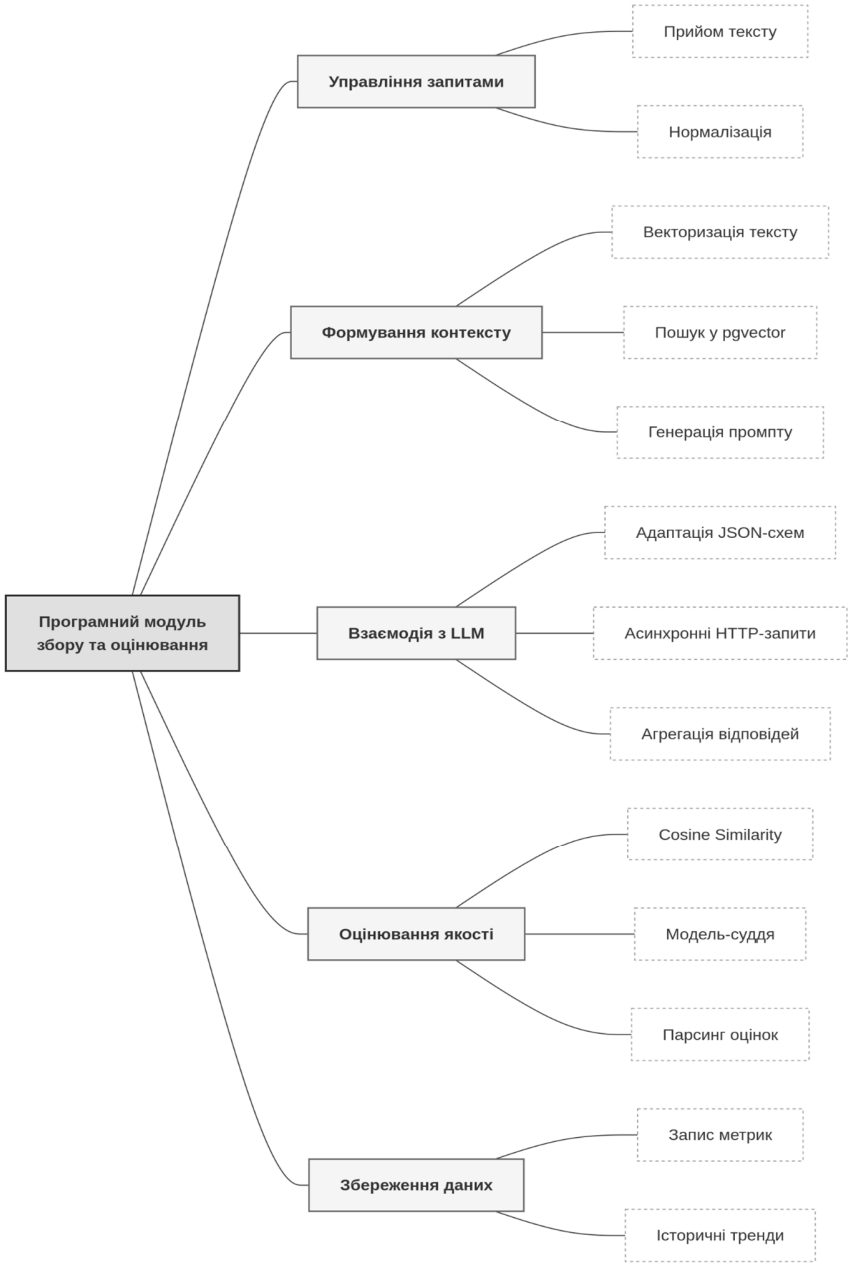

Рисунок 2.2 – Функціональна схема програмного модуля

Реалізація визначених функцій вимагає чіткої структурної організації. Структурна схема відображає фізичний та логічний поділ системи на незалежні модулі (компоненти). Програмний модуль складається з трьох основних рівнів:

– клієнтський рівень (Frontend): реалізований за допомогою бібліотеки React та інструментарію Vite. Відповідає за відображення інтерфейсу та прийом вхідних даних;

– серверний рівень (Backend): ядро системи, побудоване на базі фреймворку Spring Boot. Включає API Gateway для маршрутизації, модуль RAG-конвеєра, контролери взаємодії з LLM та Evaluator Engine для проведення оцінювання;

– рівень даних (Data Layer): контейнеризована СУБД PostgreSQL із розширенням pgvector, яка забезпечує персистентне зберігання як векторних ембеддінгів, так і реляційних аналітичних даних [19].

Взаємозв'язок між цими структурними компонентами відображено на рис. 2.3.

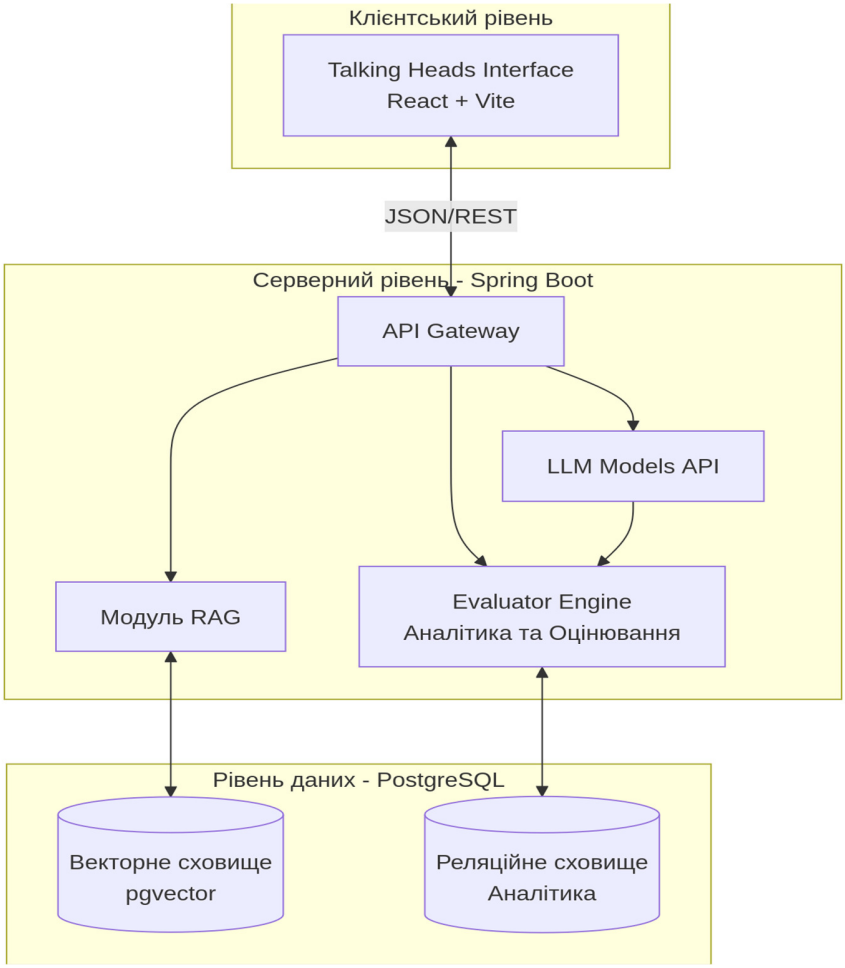

Рисунок 2.3 – Структурна схема програмного модуля

У контексті методології системного аналізу критично важливим етапом проєктування є моделювання інформаційних потоків програмного модуля. Для формалізації процесів трансформації вхідної інформації на кожному етапі її обробки

доцільним є використання діаграм потоків даних (Data Flow Diagram, DFD), що є загальновизнаним стандартом структурного аналізу.

Відповідно до розробленої DFD-моделі (рис. 2.4), зовнішня сутність «Користувач» ініціює початковий інформаційний потік, генеруючи вхідний запит. Цей запит підлягає процесу векторизації, після чого інтегрується з релевантною інформацією, вилученою зі сховища «База знань» (Knowledge Base). Сформований збагачений потік даних (Enriched Prompt) спрямовується до зовнішніх сервісів LLM, які генерують первинні (необроблені) текстові відповіді. На завершальному етапі отримані результати проходять процедуру автоматизованого оцінювання з використанням еталонних даних зі сховища «Еталонний датасет» (Gold Dataset). Розраховані аналітичні метрики та підсумкові результати зберігаються у спеціалізованому сховищі «Аналітична база даних» (Analytics DB) для подальшого використання та візуалізації.

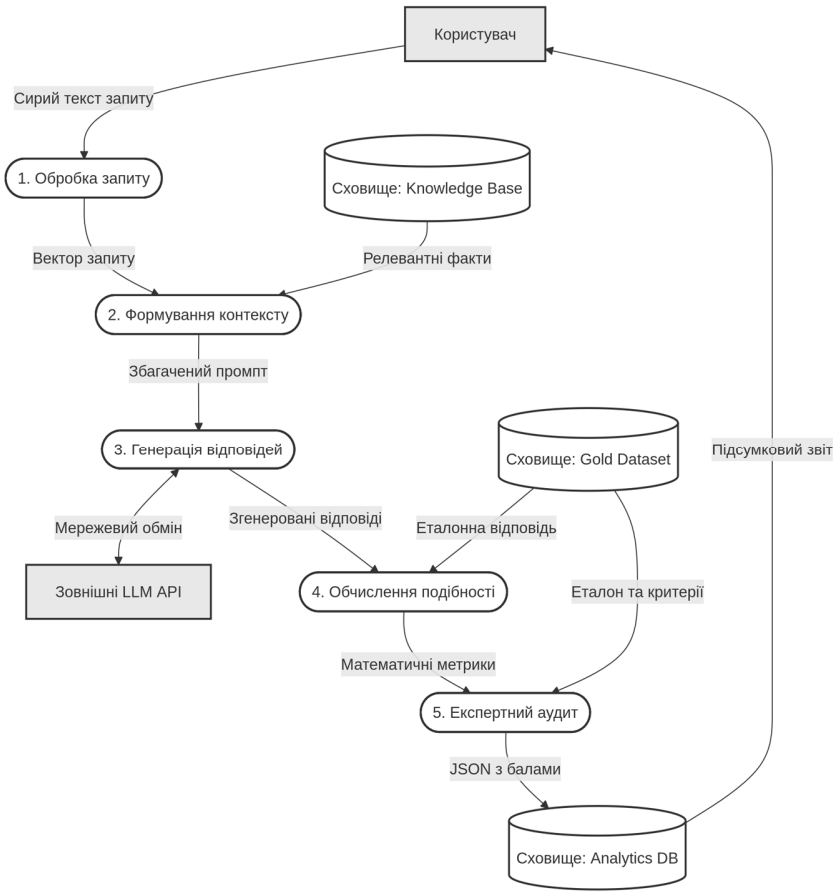

Рисунок 2.4 – Діаграма потоків даних процесу збору та оцінювання відповідей

Для розуміння динаміки роботи програмного модуля в часі побудовано діаграму послідовності (Sequence Diagram). Вона деталізує сценарій тестування моделей від моменту ініціації сесії до виведення фінального результату. Діаграма (рис. 2.5) ілюструє асинхронну природу системи: поки бекенд очікує відповіді від віддалених серверів ШІ, клієнтський інтерфейс не блокується, що забезпечує високий рівень користувацького досвіду (UX).

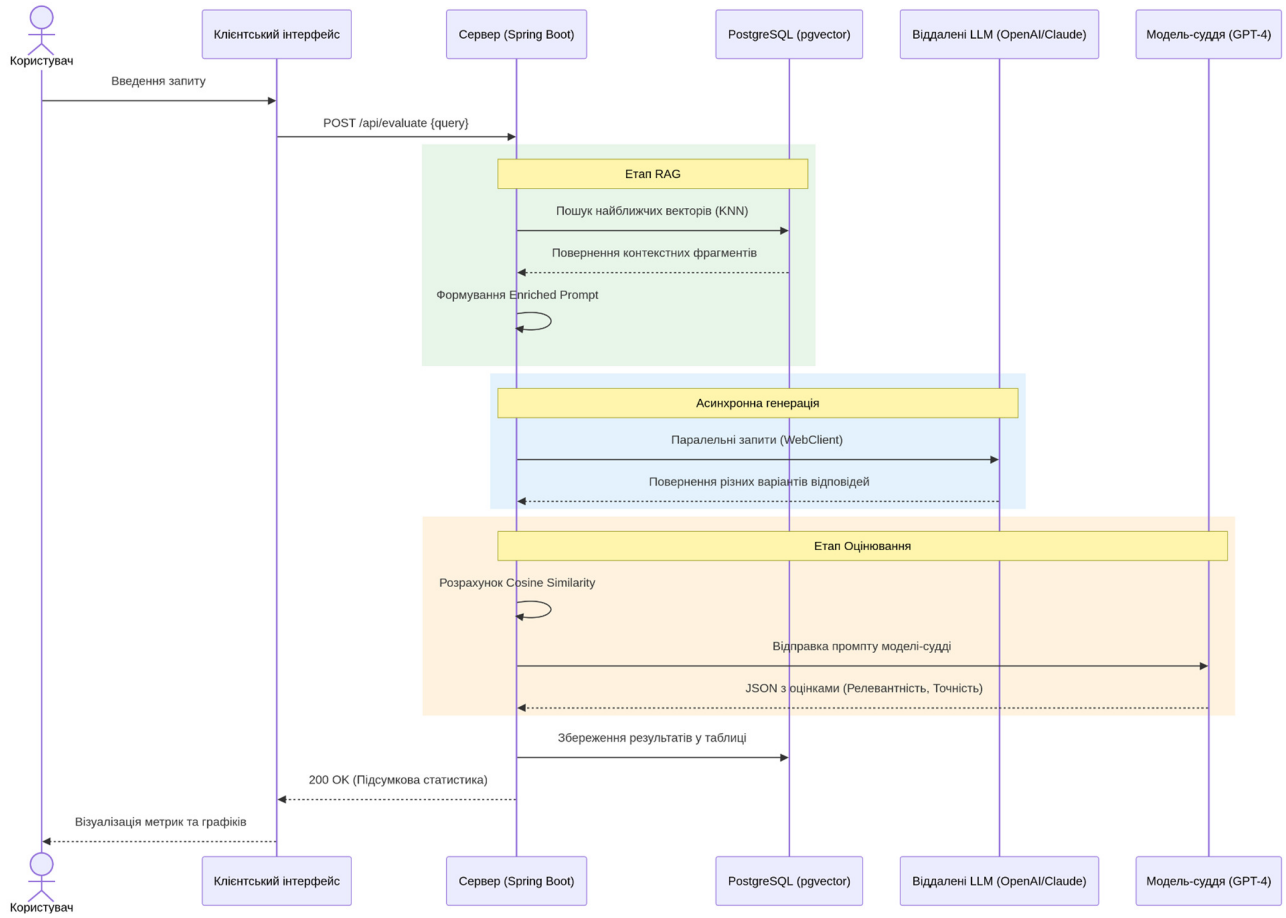

Рисунок 2.5 – Діаграма взаємодії користувача із системою під час тестування

2.2 Алгоритмічне забезпечення

Алгоритмічне забезпечення програмного модуля автоматизованого збору та оцінювання відповідей великих мовних моделей визначає логіку, математичний апарат та послідовність виконання операцій, необхідних для формування запитів,

паралельної взаємодії з розподіленими LLM-сервісами, асинхронного отримання результатів та їх багатофакторного аналізу. Основне завдання цього забезпечення полягає в повній автоматизації процесу тестування та мінімізації суб'єктивного чинника під час визначення якості згенерованого тексту.

Функціонування системи базується на наскрізному конвеєрі обробки даних, що охоплює кілька взаємопов'язаних алгоритмічних етапів: семантичне збагачення вхідного запиту, реактивне пакетне розсилання промптів, математичне обчислення векторної близькості та інтелектуальний експертний аудит.

Першим етапом обробки інформації є алгоритм семантичного збагачення початкового запиту користувача за технологією Retrieval-Augmented Generation (RAG). Пряме тестування моделей без залучення зовнішнього контексту часто призводить до явища галюцинацій, тому впровадження алгоритму інтеграції з базою знань є критично важливим для забезпечення фактологічної точності.

Алгоритм виконання RAG-конвеєра складається з таких послідовних кроків:

1. Клієнтський модуль ініціює текстовий запит через інтерфейс користувача та передає його до центрального бекенд-сервісу через API Gateway.

2. Початковий текст запиту передається до сервісу генерації векторних представлень, який використовує спеціалізовану модель ембеддінгів text-embedding-3-small від OpenAI для перетворення природної мови у багатовимірний вектор ознак.

3. Отриманий вектор використовується для виконання семантичного пошуку у реляційній базі даних PostgreSQL з активованим розширенням pgvector [19]. Пошук здійснюється за алгоритмом обчислення найближчих сусідів серед заздалегідь сегментованих фрагментів історичних джерел або корпоративної документації.

4. Система відбирає задану кількість K найбільш релевантних текстових блоків, що мають найвищий показник подібності до запиту користувача.

5. Знайдені контекстні фрагменти об'єднуються з вихідним питанням користувача у межах спеціалізованого системного шаблону, формуючи так званий «збагачений промпт» (Enriched Prompt).

Описана логіка проходження інформаційного потоку дозволяє передавати на етап генерації не просто ізольоване запитання, а структурований набір перевірених фактів. Детальна послідовність виконання операцій у межах RAG-конвеєра наведена на рисунку 2.6.

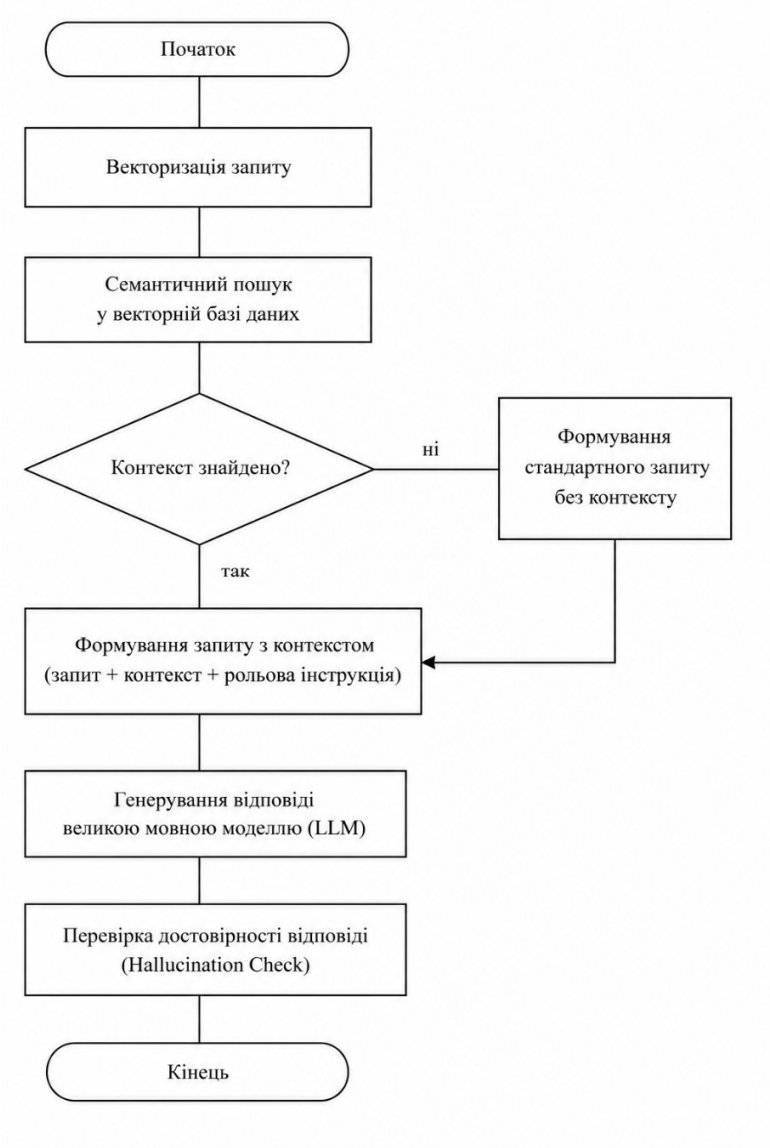

Рисунок 2.6 – Алгоритм формування відповіді з використанням RAG-конвеєра

Для проведення масового тестування та порівняльного аналізу моделей (наприклад, пакетної обробки на наборів із сотень тестових питань) класична синхронна модель передачі даних «запит-відповідь» є непридатною. Тривале

очікування генерації тексту від віддалених хмарних серверів призводить до блокування потоків виконання та критичного зниження продуктивності системи.

Для вирішення цієї проблеми в середовищі Java та фреймворку Spring Boot [20 22] реалізовано алгоритм реактивної неблокуючої взаємодії, що базується на використанні компонента WebClient [23]:

1. Збагачений промпт із транспортного рівня системи дублюється та адаптується під специфічні формати JSON-схем конкретних провайдерів штучного інтелекту.

2. Бекенд-сервіс ініціює паралельні асинхронні HTTP-запити до віддалених API-ендпоінтів OpenAI (для моделей сімейства GPT), Google DeepMind (для моделей Gemini) [2] та Anthropic (для моделей Claude).

3. Завдяки архітектурі неблокуючого введення-виведення (NIO), основні потоки застосунку не переходять у стан очікування (Thread Blocking). Система виділяє ресурси лише в моменти надходження мережевих пакетів із готовими відповідями від серверів провайдерів.

4. Модуль асинхронного збору накопичує згенеровані відповіді у буферній пам'яті, фіксує часові метрики виконання (Latency) та після завершення генерації всіма досліджуваними моделями передає отриманий пакет текстів до аналітичного ядра.

Такий підхід дозволяє радикально знизити загальний час проведення повного циклу тестування системи з кількох годин до кількох хвилин, забезпечуючи одночасний збір даних від різних конфігурацій ШІ.

Серцем аналітичного рівня системи є алгоритм двоступеневого оцінювання та класифікації отриманих текстових даних. Оскільки відповіді різних моделей не можуть бути порівняні з золотим стандартом (еталоном) методом прямого посимвольного чи лексичного зіставлення через високу варіативність природної мови, алгоритмічне забезпечення використовує комбінацію математичного та експертного підходів.

На першому етапі здійснюється математичне оцінювання семантичної

близькості текстів. Отримана відповіді моделі та відповідний їй еталон із контрольного датасету (Gold Dataset) проходять етап попередньої нормалізації (очищення від службових символів та маркерів форматування Markdown) і векторизуються. Математична модель оцінки базується на визначенні косинуса кута між вектором відповіді моделі mathbfA та вектором еталонної відповіді mathbfB у багатовимірному просторі ознак, що описується формулою 2.2:

$$similarity = cos(\theta) = \frac{(A \cdot B)}{(||A||\ ||B||} \tag{2.1}$$

де (A)— векторне представлення відповіді мовної моделі;

(B)— векторне представлення еталонної відповіді;

$(A \cdot B)$— скалярний добуток векторів;

$(|A|)$— довжина вектора відповіді моделі;

$(|B|)$— довжина вектора еталонної відповіді.

Значення цієї метрики варіюється в діапазоні від 0 до 1, де одиниця свідчить про ідентичний семантичний зміст аналізованих текстових масивів.

На другому етапі отримані дані спрямовуються до підсистеми інтелектуального аудиту, що реалізує концепцію «LLM-as-a-Judge» [11]. Цей етап є найбільш наукомістким, оскільки використовує модель вищого порядку (наприклад, GPT-4) як незалежного лінгвістичного експерта для детального аналізу тексту на предмет галюцинацій та дотримання мовно-рольової моделі.

Алгоритм функціонування моделі-судді базується на суворих принципах інженерії підказок (Prompt Engineering):

1. Системне обмеження: Моделі-судді передається рольова інструкція, яка чітко визначає критерії оцінювання та накладає суворе обмеження на формат виводу — відповідь має бути повернута виключно у вигляді валідного JSON-об'єкта без будь-яких вступних, додаткових чи пояснювальних текстових фраз.

2. Калібрування оцінок (Few-shot prompting): До контексту запиту додається кілька еталонних прикладів (пар «запитання — фактична відповідь — золотий

стандарт») із заздалегідь визначеними фіксованими балами. Це дозволяє моделі-судді чітко зрозуміти логіку градації оцінок за критеріями релевантності, повноти та фактичної точності.

3. Динамічна підстановка: Конвеєр Spring Boot автоматично підставляє у три змінні блоки інструкції вихідний контекст, відповідь тестованої моделі та золотий стандарт.

Загальна логічна послідовність кроків аналізу та оцінювання відповідей за допомогою математичних метрик та моделі-судді представлена на рисунку 2.7.

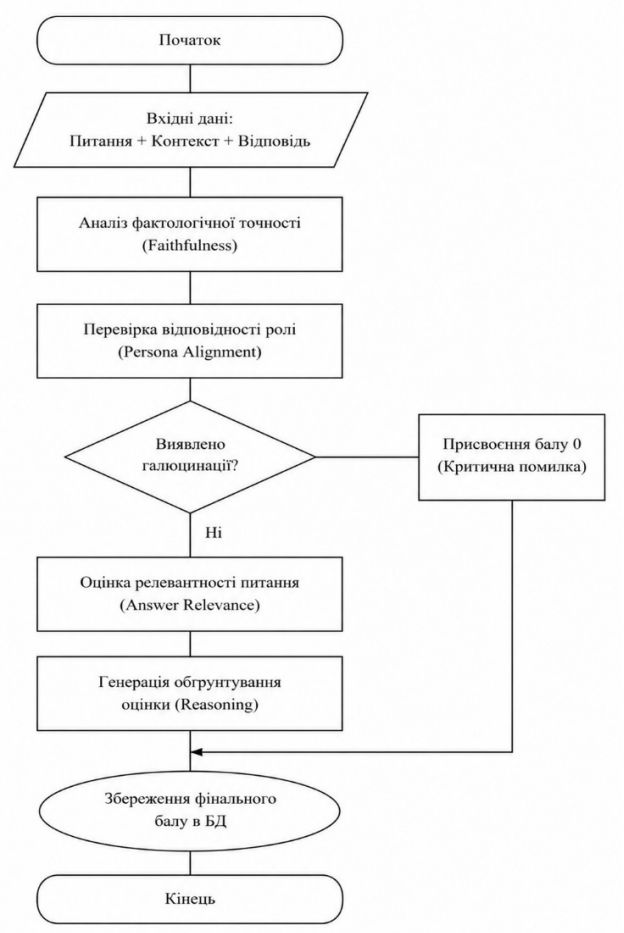

Рисунок 2.7 – Алгоритм інтелектуального оцінювання відповіді моделлю-суддею.

‘

Прикінцевим етапом алгоритмічного забезпечення є рівень довгострокового збереження накопичених аналітичних даних (Data Persistence Layer). Після того як

модель-суддя повертає згенерований JSON-об'єкт, система активує алгоритм парсингу та персистентності [23]:

1. Бекенд-компонент здійснює миттєву валідацію та десеріалізацію отриманої JSON-структури, виділяючи окремі числові бали та текстові обґрунтування виставлених оцінок.

2. За допомогою інструментів Spring Data JPA [22] сформовані об'єкти даних трансформуються у реляційні сутності та записуються у відповідні таблиці СУБД PostgreSQL [19], структура якої спроєктована за правилами третьої нормальної форми (3NF).

3. Для гарантування персистентності даних у контейнеризованому середовищі Docker алгоритм взаємодії з операційною системою використовує монтування іменованих томів (volumes), що виключає ризик втрати зібраних результатів тестування у разі перезапуску або видалення контейнерів системи.

Алгоритм передбачає збереження індивідуальних коефіцієнтів калібрування (температури генерації, розміру контекстного вікна та конфігурацій промптів) разом із розрахованими метриками (F1-score, BLEU, Hallucination Rate). Це дозволяє накопичувати історичні тренди для подальшої побудови статистичних звітів, графіків та формування об'єктивних рейтингів ефективності досліджуваних великих мовних моделей [24].

2.3 Інформаційне забезпечення

Інформаційне забезпечення програмного модуля автоматизованого збору та оцінювання відповідей великих мовних моделей визначає сукупність структур даних, форматів обміну інформацією, способів організації баз даних та нормативно-довідкової інформації, необхідних для коректного функціонування системи. Головним завданням цього забезпечення є створення надійного фундаменту для збереження векторних представлень текстів, еталонних наборів даних та результатів багатофакторного аналізу.

Враховуючи вимоги до транзакційної цілісності та складних реляційних зв'язків між сутностями, основою інформаційного забезпечення системи виступає реляційна система управління базами даних PostgreSQL [19]. Спроєктована логічна структура бази даних відповідає правилам третьої нормальної форми (3NF) і логічно розділена на три ключові інформаційні домени:

– домен управління контекстом (Knowledge Base): зберігає попередньо сегментовані фрагменти історичних джерел та документації. Для реалізації механізмів семантичного пошуку за технологією RAG у базі даних активовано розширення pgvector . Це дозволяє зберігати багатовимірні масиви ознак (ембеддінги), згенеровані моделлю text-embedding-3-small, безпосередньо поруч із їхніми текстовими еквівалентами для швидкого векторного пошуку;

– домен контрольних даних (Gold Dataset): містить бібліотеку еталонних запитань та ідеальних відповідей. Цей набір даних виступає базою для обчислення математичної близькості векторів та калібрування моделі-судді за допомогою еталонних прикладів (Few-shot prompting);

– домен аналітичних результатів: призначений для довгострокового накопичення згенерованих відповідей, розрахованих метрик (семантична близькість, оцінки моделі-судді) та індивідуальних коефіцієнтів конфігурації (наприклад, температура генерації або версія моделі).

Обмін інформацією між компонентами системи (клієнтським інтерфейсом, бекенд-сервісом та віддаленими LLM-провайдерами) здійснюється у форматі JSON (JavaScript Object Notation). Оскільки система інтегрується з різними сімействами моделей, інформаційне забезпечення транспортного рівня реалізує динамічну адаптацію єдиного внутрішнього формату запиту (Enriched Prompt) під специфічні JSON-схеми конкретних API, включаючи OpenAI, Google DeepMind та Anthropic.

Особливе місце в інформаційному забезпеченні посідає структура даних для підсистеми інтелектуального аудиту («LLM-as-a-Judge»). Формат вихідних даних від моделі-судді жорстко стандартизовано: системна інструкція накладає суворе обмеження на повернення результату виключно у вигляді валідного JSON-об'єкта.

Цей об'єкт містить заздалегідь визначені ключі з числовими балами за релевантність, повноту і фактичну точність, а також текстові обґрунтування виставлених оцінок. Серверний компонент на базі Spring Boot виконує миттєву валідацію та десеріалізацію отриманої структури, трансформуючи її в об'єкти Java для подальшого запису в реляційні таблиці за допомогою Spring Data JPA [12].

Для забезпечення персистентності зібраної інформації у середовищі розгортання застосовується механізм іменованих томів Docker (volumes) [14]. Це гарантує, що накопичені історичні тренди, статистичні дані та результати тестування моделей не будуть втрачені у процесі масштабування, перезапуску або оновлення контейнеризованих мікросервісів системи.

Для забезпечення повноцінного функціонування модуля у його архітектуру інтегровано рівень довгострокового збереження даних (Data Persistence Layer). Враховуючи вимоги до транзакційної цілісності та складних реляційних зв'язків, як основну СУБД обрано PostgreSQL. Спроєктована структура бази даних відповідає правилам третьої нормальної форми (3NF) і включає кілька доменів: управління знаннями експонатів, бібліотеку контрольних питань (Gold Dataset) та архів результатів тестування. Концептуальна логічна структура бази даних системи представлена на рисунку 2.8.

Технічна реалізація рівня збереження даних базується на використанні контейнеризованого образу postgres:15-alpine, що забезпечує легкість розгортання та стабільність роботи в різних середовищах. Для гарантування цілісності та персистентності інформації у контейнеризованому середовищі застосовано механізм іменованих томів Docker (volumes), зокрема том pgdata (для робочого середовища) та pgdata_dev (для середовища розробки), які монтуються за шляхом /var/lib/postgresql/data всередині контейнера. Це дозволяє зберігати всі зібрані результати тестування та конфігурації промптів навіть після повного перезапуску або видалення контейнерів системи.

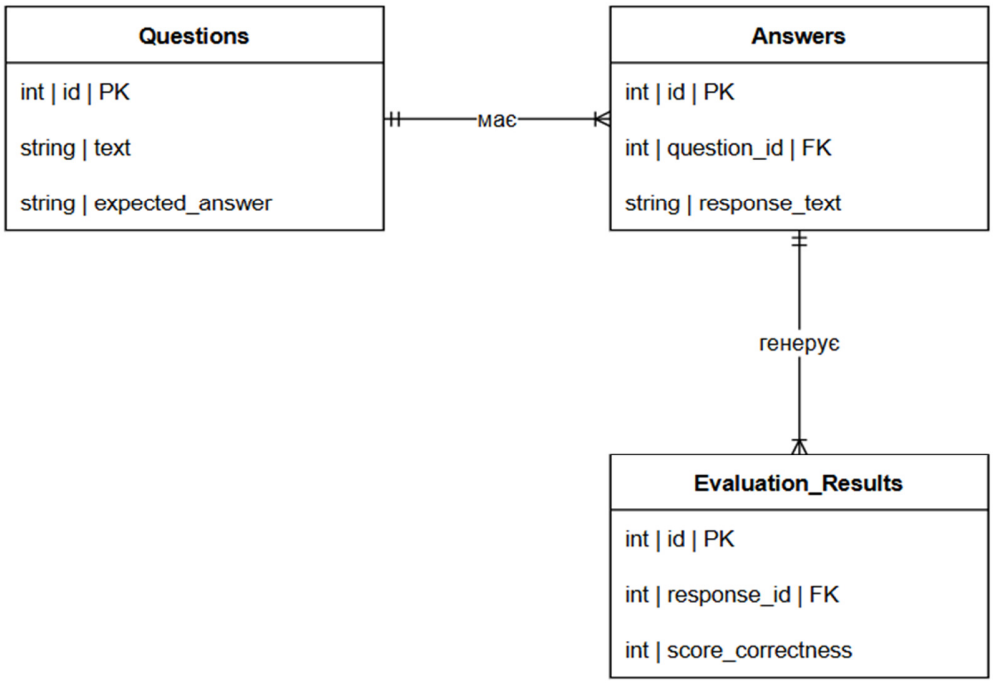

Рисунок 2.8 – Концептуальна логічна структура бази даних системи

Управління схемою бази даних та її синхронізація з програмним кодом автоматизовано засобами Spring Data JPA з використанням параметра hibernate.ddl-auto: update. Такий підхід дозволяє системі самостійно модифікувати структуру таблиць (наприклад, додавати нові поля для нових метрик оцінювання) без необхідності ручного написання SQL-міграцій, що значно прискорює ітеративну розробку модуля.

Безпека доступу до бази даних реалізована через механізм ін’єкції змінних оточення. Всі критичні параметри, такі як DB_NAME, DB_USERNAME та DB_PASSWORD, не зберігаються у відкритому коді, а динамічно передаються в контейнери через файл .env під час локального запуску або через секрети GitHub (GitHub Secrets) під час автоматизованого розгортання. Також у системі налаштовано механізм перевірки готовності, який використовує утиліту pg_isready для підтвердження того, що база даних повністю ініціалізована та готова до прийому з'єднань від бекенд-сервісу.

Архітектура передбачає збереження індивідуальних коефіцієнтів калібрування для кожної моделі. Це дозволяє аналізувати, як зміна температури генерації або розміру контекстного вікна впливає на фінальні метрики (F1-score, BLEU, Hallucination Rate). Такий підхід забезпечує повну прозорість процесу розробки та відповідає стандартам якості корпоративного програмного забезпечення.

3 ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

3.1 Розгортання інфраструктури та налаштування базових серверних сервісів

Відповідно до спроєктованої архітектури, для практичної реалізації системи було обрано сучасний та надійний технологічний стек, здатний забезпечити стабільну взаємодію з API зовнішніх провайдерів штучного інтелекту.

Основним ядром бекенд-частини (API), яка відповідає за бізнес-логіку управління тестуванням, взаємодію з базою даних та комунікацію з LLM-провайдерами, виступає фреймворк Spring Boot на базі мови програмування Java 17. Вибір цього фреймворку зумовлений наявністю потужного інструментарію spring-boot-starter-web, який дозволяє швидко розгортати REST-сервіси на базі вбудованого сервера Tomcat.

Архітектура бекенду реалізована за принципом чіткого розділення відповідальності (Separation of Concerns). Пакетна структура проєкту включає наступні ізольовані шари:

- моделі даних (Model): сутності Question, Answer та Result, анотовані згідно зі специфікацією JPA (@Entity), що відображають структуру реляційної бази даних;
- репозиторії (Repository): інтерфейси доступу до даних, побудовані на базі Spring Data JPA (наприклад, QuestionRepository), які абстрагують виконання SQL-запитів;
- сервіси (Service): шар інкапсуляції бізнес-логіки. Включає алгоритми розрахунку косинусної подібності (EvaluationService), управління опитуванням моделей (AnswerService) та реалізацію підходу LLM-as-a-Judge (JudgeService);
- контролери (Controller): точки входу REST API (AIController, QuestionController), що забезпечують прийом HTTP-запитів від клієнтського застосунку та маршрутизацію відповідей.

Особливу увагу приділено інтеграції з системою управління базами даних. Для персистентного зберігання інформації обрано СУБД PostgreSQL. Інтеграція зі Spring Boot здійснюється через залежність postgresql та налаштування конфігурації

Hibernate. Під час запуску та розгортання застосунку механізми spring.jpa.hibernate.ddl-auto забезпечують автоматичну синхронізацію схеми таблиць у базі даних із Java-класами моделей, що мінімізує необхідність ручного написання міграцій на етапі прототипування [25].

Мережева комунікація з API великих мовних моделей реалізована через синхронний HTTP-клієнт RestTemplate. Для забезпечення гнучкості та масштабованості системи впроваджено поліморфний підхід: створено базовий інтерфейс LLMService, який імплементується окремими класами-обробниками для кожного провайдера:

- OpenAIService — для взаємодії з моделлю GPT-4o-mini;
- ClaudeService — для взаємодії з моделлю Claude 3 Haiku;
- GeminiService — для роботи з моделлю Gemini 2.0 Flash.
- TalkingHeadsService — для інтеграції з локальною мовною моделлю проєкту цифрових двійників університету. Взаємодія здійснюється через закритий REST API, що дозволяє модулю надсилати запити до конкретних аватарів (експонатів) та порівнювати якість їхніх відповідей із результатами провідних комерційних моделей.

Специфікація програмного інтерфейсу (API) для взаємодії з платформою «Talking Heads» задокументована у форматі колекції Postman. Повний лістинг конфігураційного JSON-файлу, що описує ендпоінти отримання списку доступних аватарів та ініціалізації чату (/api/client/chat), наведено у Додатку Б [26, 27].

Парсинг вхідних параметрів та відповідей від ШІ-сервісів стандартизовано за допомогою бібліотеки org.json, що дозволяє ефективно вилучати згенерований контент зі складної структури JSON-відповідей (наприклад, з масивів choices або candidates).

Для забезпечення абсолютної портативності системи та гарантування ідентичності середовищ розробки й виконання застосовується технологія контейнеризації Docker. Процес оркестрації інфраструктури реалізується за допомогою інструменту Docker Compose.

Для автоматизації процесу збирання образів (Image Building) було розроблено багаторівневі (*multi-stage*) Dockerfile. Використання *multi-stage* збірки дозволяє мінімізувати розмір фінальних образів контейнерів за рахунок виключення зайвих залежностей етапу компіляції та відокремлення середовища розробки від середовища виконання.

Лістинг 3.1 демонструє конфігурацію Dockerfile для бекенд-частини на базі Spring Boot. Використання образу maven:3.9-eclipse-temurin-21 на етапі збирання дозволяє виконати компіляцію проєкту, після чого отриманий артефакт (JAR-файл) копіюється у фінальний, значно легший образ eclipse-temurin:21-jdk-alpine.

Лістинг 3.1 — Dockerfile для бекенд-сервісу (Java Spring Boot)

```
FROM maven:3.9-eclipse-temurin-21 AS build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn clean package -DskipTests -e

FROM eclipse-temurin:21-jdk-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Аналогічний підхід застосовано для фронтенд-частини (React), де на першому етапі (builder) відбувається встановлення залежностей та збірка статичних файлів через npm run build, а на другому етапі — розгортання готового статичного контенту за допомогою легкого вебсервера Nginx. Конфігурацію наведено у лістингу 3.2.

Лістинг 3.2 — Dockerfile для фронтенд-застосунку (React + Nginx)

```
FROM node:20-alpine AS builder
WORKDIR /app
```

```
COPY package*.json ./
RUN npm install
COPY . .
ENV VITE_API_URL=/api
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Головний файл docker-compose.yml, який забезпечує оркестрацію сервісів та налаштування мережевої взаємодії між базою даних, API та інтерфейсом, наведено у Додатку В. Завдяки механізму явних залежностей (depends_on), серверна частина ініціалізується виключно після успішного старту та перевірки готовності бази даних, що унеможливлює виникнення помилок підключення під час автоматичного розгортання системи.

Для детального ознайомлення з програмною реалізацією бізнес-логіки системи ключові фрагменти вихідного коду винесено у додатки до кваліфікаційної роботи. Зокрема, у Додатку Г наведено повний лістинг сервісу AnswerService.java, який демонструє механізм ітеративного звернення до підключених LLM-провайдерів, вимірювання часу генерації відповідей (Response Time) та їх персистентного збереження у базу даних для подальшого оцінювання.

Важливою частиною практичної роботи стало впровадження автоматизованого циклу безперервної інтеграції та розгортання (CI/CD) на базі GitHub Actions. Сценарій розгортання deploy.yml автоматично активується при кожному оновленні головної гілки коду (push у гілку main). Процес виконується на власному сервері підприємства (self-hosted runner), що забезпечує конфіденційність даних та повний контроль над обчислювальними ресурсами.

Пайплайн включає наступні етапи:

1. Конфігурування середовища: автоматичне формування файлу .env на основі секретів GitHub (GitHub Secrets). Це дозволяє безпечно оперувати паролями до БД та адресою VITE_API_URL для фронтенду .

2. Розгортання: команда docker compose up -d --build здійснює перезбирання образів та оновлення контейнерів у фоновому режимі без зупинки сервісів для користувача.

3. Валідація (Healthcheck): після запуску система очікує 15 секунд для ініціалізації мережевих інтерфейсів, після чого виконує автоматичну перевірку статусу контейнера llmautotest_ui. У разі виявлення помилок система автоматично виводить логи розгортання для діагностики.

Для оперативного виявлення помилок під час асинхронної взаємодії з LLM-провайдерами налаштовано систему логування. На рівні Java-бекенду використано фреймворк Logback, який фіксує деталі HTTP-запитів WebClient, ліміти API та помилки валідації JSON. Адміністратор системи може відстежувати стан сервісів у реальному часі за допомогою стандартних засобів моніторингу Docker.

3.2 Проєктування та програмна реалізація користувацького інтерфейсу панелі адміністратора

Клієнтська частина програмного модуля, що виконує функції панелі адміністратора та візуалізації результатів тестування, реалізована як сучасний односторінковий застосунок (Single Page Application, SPA). Для розробки інтерфейсу було обрано бібліотеку React (версії 19.2), використання якої зумовлене її декларативним підходом до програмування та ефективним механізмом оновлення віртуального DOM.

Зовнішній вигляд розробленої панелі адміністратора на етапі ініціалізації тестування представлено на рисунку 3.1.

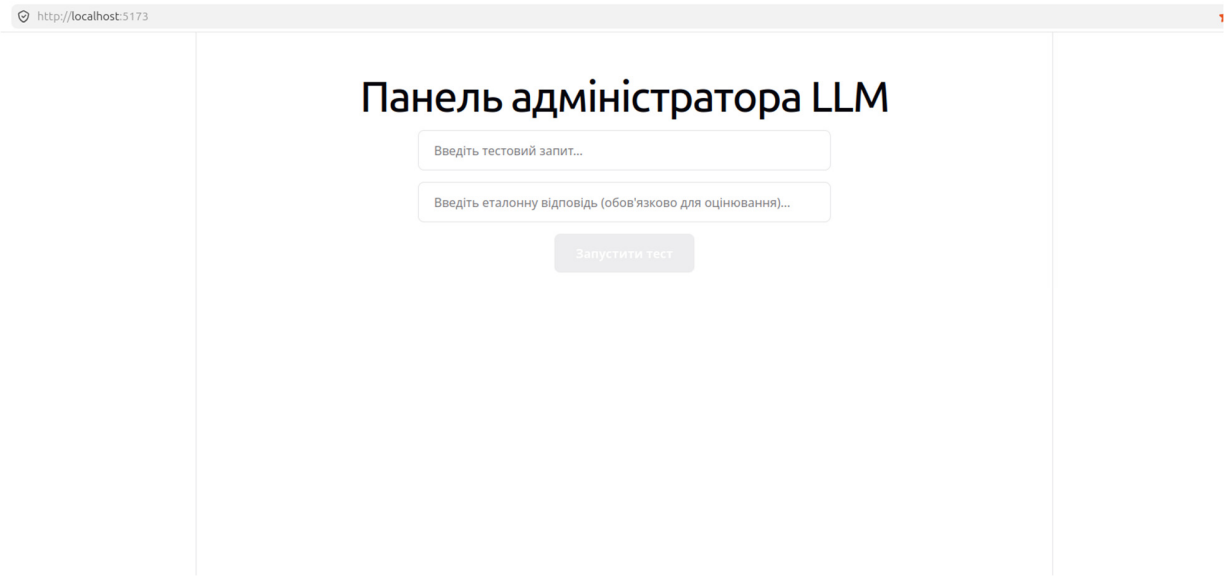

Рисунок 3.1 – Початковий інтерфейс панелі адміністратора з формою введення запиту

Інтерфейс побудовано за принципом компонентної декомпозиції. Замість використання важких сторонніх UI-фреймворків, розробка базується на написанні власних легкокодованих компонентів та ізольованій стилізації через нативні каскадні таблиці стилів (CSS). Графічний інтерфейс логічно розділено на наступні функціональні блоки:

– блок управління запитами (.question-box): містить поле введення для ініціалізації тестових запитань та елементи керування для запуску процесу генерації відповідей моделями;

– блок результатів та карток моделей (.cards та .card): динамічний контейнер, який відповідає за рендеринг відповідей від конкретних ШІ-провайдерів (OpenAI, Claude, Gemini). Кожна картка інкапсулює логіку відображення згенерованого тексту та відповідних оцінок;

– блок агрегованої аналітики (.winner та .chart-box): секція візуалізації найкращої моделі за результатами ітерації та графічного представлення порівняльних метрик.

Після завершення асинхронних запитів до моделей інтерфейс оновлюється, відображаючи результати бенчмаркінгу, як показано на рисунку 3.2.

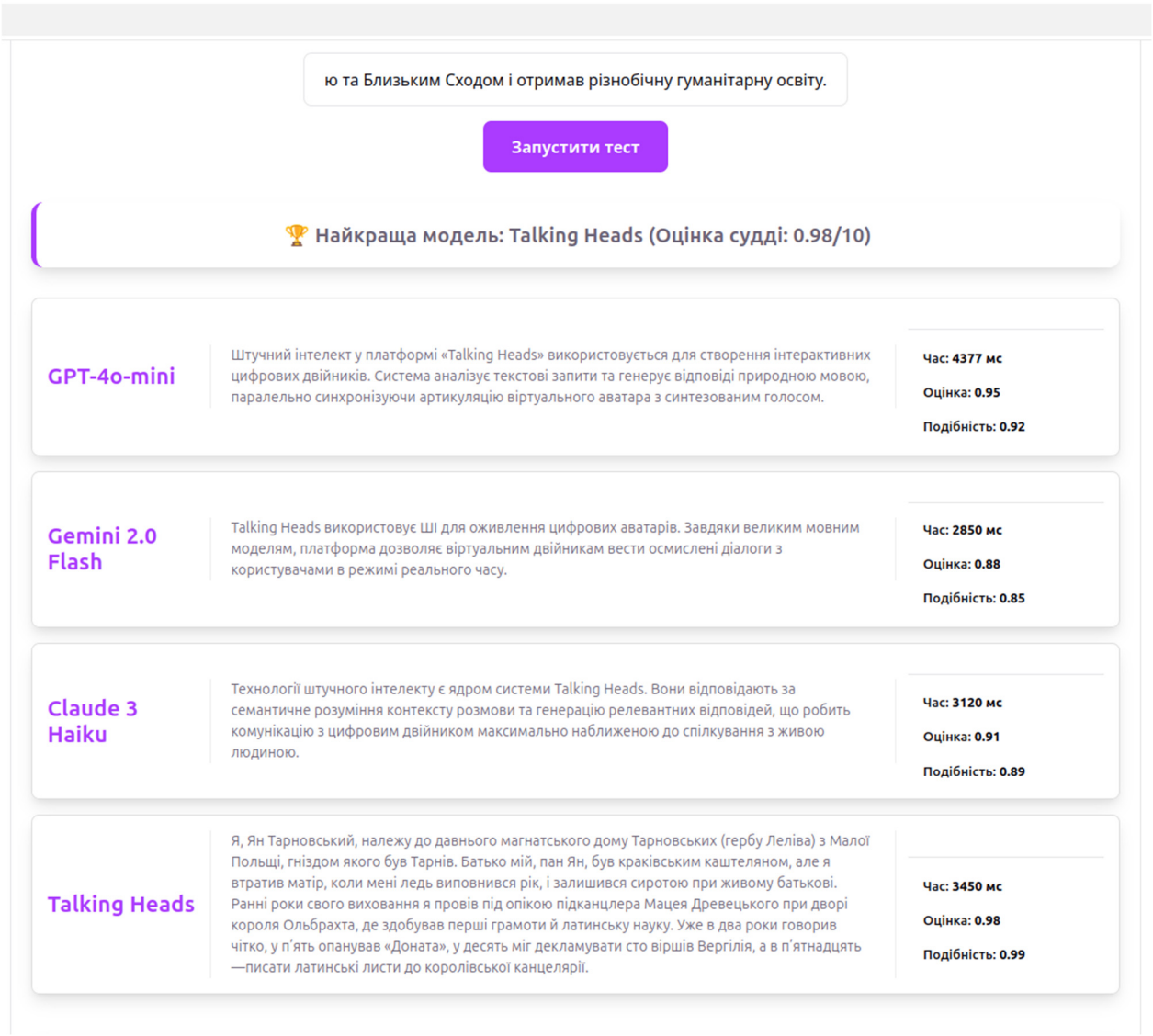

Рисунок 3.2 – Відображення результатів бенчмаркінгу LLM-моделей

Стилізація застосунку реалізована з використанням сучасних можливостей CSS (файли App.css та index.css). Для забезпечення консистентності дизайну та підтримки світлої/темної тем оформлення (prefers-color-scheme: dark) впроваджено систему глобальних CSS-змінних (наприклад, --text, --bg, --accent). Адаптивність інтерфейсу під різні розміри екранів реалізована за допомогою медіа-запитів (@media (max-width: 1024px)).

Оскільки застосунок має лінійну структуру даних і не потребує складної маршрутизації, управління локальним станом (State Management) реалізовано

виключно за допомогою вбудованих хуків бібліотеки React, зокрема useState (для збереження вхідних даних та результатів) та useEffect (для управління побічними ефектами та життєвим циклом компонентів). Це дозволило уникнути надлишкового використання таких глобальних стейт-менеджерів, як Redux або Context API.

Взаємодія з бекенд-частиною системи (Java Spring Boot) здійснюється за допомогою бібліотеки axios. Використання цього HTTP-клієнта забезпечує автоматичну трансформацію JSON-даних та зручну обробку асинхронних промісів (Promises). Клієнтська часть звертається до REST API ендпоінтів сервера, ініціюючи процеси оцінювання та отримуючи розраховані метрики (Cosine Similarity та оцінку судді LLM-as-a-Judge) у форматі масиву об'єктів.

Критично важливим завданням користувацького інтерфейсу є представлення числових результатів бенчмаркінгу у зручному для аналізу форматі. Для вирішення цієї задачі інтегровано бібліотеку recharts (версії 3.8). Вона дозволяє створювати декларативні, чутливі до змін стану (reactive) графіки. На основі отриманих від сервера даних (Result), бібліотека recharts динамічно рендерить SVG-діаграми у контейнері .chart-box, порівнюючи час генерації (Response Time) та метрики якості (Similarity Score, Judge Score) для кожної досліджуваної моделі.

Такий архітектурний підхід до побудови фронтенду забезпечив створення швидкого, масштабованого та інтуїтивно зрозумілого інтерфейсу. Повний програмний вихідний код головної сторінки застосунку (Home Page), який інкапсулює описану логіку управління станом, обробки мережевих запитів та візуалізації результатів, наведено у Додатку Д.

Оскільки основна бізнес-логіка, управління станом та рендеринг інтерфейсу були інкапсульовані в окремому компоненті HomePage, головний файл застосунку App.jsx було рефакторизовано. Тепер він виконує роль виключно базового контейнера (кореневого компонента), що відповідає принципам компонентно-орієнтованої архітектури (Component-Based Architecture) у React.

Винесення логіки сторінки з App.jsx дозволило зберегти чистоту коду на верхньому рівні застосунку. Тепер цей файл відповідає лише за імпорт глобальних

стилів та монтування головної сторінки, що закладає гнучкий фундамент для можливого майбутнього масштабування (наприклад, для легкого підключення бібліотеки React Router та додавання нових сторінок без переписування існуючого коду).

Програмну реалізацію оновленого кореневого компонента наведено у лістингу 3.3.

Лістинг 3.3 — Програмна реалізація кореневого компонента App.jsx

```
import React from 'react';
структури каталогів)
import HomePage from './HomePage';
import './App.css';

function App() {
  return (
    <div className="app-root">
      <HomePage />
    </div>
  );
}

export default App;
```

3.3 Програмна реалізація механізмів асинхронної взаємодії з LLM та обробки винятків

Ключовим завданням серверної частини розробленого програмного модуля є забезпечення стабільної та гнучкої комунікації з прикладними програмними інтерфейсами (API) зовнішніх провайдерів штучного інтелекту. Оскільки система передбачає одночасне тестування декількох великих мовних моделей (GPT-4o-mini, Claude 3 Haiku, Gemini 2.0 Flash), архітектуру мережевої взаємодії було побудовано на основі поліморфізму та патерну проєктування «Стратегія» (Strategy).

Для уніфікації процесу відправлення запитів та отримання результатів було створено базовий інтерфейс LLMService (лістинг 3.4). Такий підхід гарантує слабку зв'язність коду (loose coupling) і дозволяє легко інтегрувати нові моделі в майбутньому без зміни основної бізнес-логіки системи.

Лістинг 3.4 — Інтерфейс базового сервісу мовних моделей

```
package com.example.demo.service;

public interface LLMService {
    String askModel(String question);

    String getModelName();
}
```

Безпосередня мережева комунікація з API-серверами провайдерів реалізована за допомогою синхронного HTTP-клієнта RestTemplate, що є стандартним компонентом екосистеми Spring Framework. Формування тіла запиту (Payload) здійснюється з використанням багаторядкових текстових блоків Java (Text Blocks), що дозволяє зручно конструювати структуру JSON-документів.

Для десеріалізації відповідей, які повертають провайдери, застосовано бібліотеку org.json. Оскільки кожен провайдер має власну унікальну структуру відповіді (наприклад, OpenAI повертає масив choices, а Gemini — масив candidates), алгоритм парсингу інкапсульовано всередині відповідних класів-реалізацій (OpenAIService, ClaudeService, GeminiService).

Обробка виняткових ситуацій (Exception Handling) Під час роботи із зовнішніми мережевими сервісами критично важливим є забезпечення відмовостійкості системи. Процес генерації тексту LLM-моделлю може бути перерваний через низку непередбачуваних факторів: перевищення лімітів запитів (HTTP 429 Too Many Requests), тайм-аути з'єднання або тимчасову недоступність серверів ШІ-провайдера (HTTP 503 Service Unavailable).

Для запобігання аварійному завершенню (Crash) всього циклу тестування, логіку мережевих запитів обгорнуто у блоки try-catch. У разі виникнення винятку

(наприклад, помилки авторизації через недійсний API-ключ або обриву з'єднання), система перехоплює його та замість "падіння" повертає стандартизоване текстове повідомлення з префіксом "ERROR: ", додаючи технічні деталі помилки. Це дозволяє зберегти факт збою в базу даних PostgreSQL для подальшого аналізу адміністратором, не блокуючи при цьому опитування інших мовних моделей у межах поточної ітерації.

Програмна реалізація алгоритму LLM-as-a-Judge Окремою складовою модуля оцінювання є сервіс JudgeService, який відповідає за експертний аналіз відповідей за допомогою парадигми «LLM-as-a-Judge». У ролі незалежного арбітра виступає модель компанії OpenAI. Сервіс формує спеціалізований системний промпт, у який динамічно підставляються: початкове запитання, еталонна відповідь (Correct Answer) та відповідь, згенерована тестованою моделлю.

Оскільки мовні моделі генерують текст природною мовою, відповідь судді може містити зайві символи або пояснення, тоді як для математичного аналізу системі потрібне лише числове значення оцінки. Для розв'язання цієї проблеми впроваджено механізм нормалізації отриманих даних на базі регулярних виразів (Regex), програмний код якого наведено у лістингу 3.5.

Лістинг 3.5 — Фрагмент класу JudgeService з логікою парсингу оцінки

```
try {
    String response = openAIService.askModel(prompt);

    response = response.trim();

    response = response.replaceAll("[^0-9.]", "");

    return Double.parseDouble(response);

} catch (Exception e) {
    e.printStackTrace();
    return 0.0;
}
```

Наведений алгоритм забезпечує гарантоване перетворення текстового результату в тип даних Double, повністю виключаючи ймовірність виникнення винятків типу NumberFormatException під час подальшого агрегування результатів та їх відправлення на клієнтську частину (Frontend) для візуалізації.

Оркестрація всього циклу збору даних та їх подальшого оцінювання здійснюється на рівні контролера AIController. Цей компонент є головною точкою входу (Entry Point) для клієнтських запитів і відповідає за синхронізацію роботи різних сервісів програмного модуля.

У лістингу 3.6 наведено програмний код методу process, який наочно демонструє конвеєр обробки даних: від завантаження початкового запитання з бази даних до фінального збереження комплексних результатів оцінювання. Система автоматично генерує відповіді за допомогою компонента AnswerService, після чого кожна отримана відповідь проходить двоетапну метричну перевірку. Спочатку EvaluationService виконує базовий розрахунок текстової схожості (Similarity Score), а потім JudgeService формує експертну оцінку моделі-арбітра (Judge Score).

Лістинг 3.6 — Фрагмент класу AIController з логікою оркестрації процесу тестування та оцінювання

```
@PostMapping("/ask/{id}")
public List<Result> process(
        @PathVariable Long id
) {
    Question q = questionRepo.findById(id).orElseThrow();

    List<Answer> answers = answerService.generateAnswers(q);
    List<Result> results = new ArrayList<>();

    for (Answer answer : answers) {
        double similarityScore = evaluationService.calculateSimilarity(
                answer.getContent(),
                q.getCorrectAnswer()
        );
```

```
            double judgeScore = judgeService.evaluateAnswer(
                    q.getText(),
                    q.getCorrectAnswer(),
                    answer.getContent()
            );

            Result result = new Result();
            result.setAnswer(answer);
            result.setSimilarityScore(similarityScore);
            result.setJudgeScore(judgeScore);

            results.add(resultRepo.save(result));
        }
        return results;
    }
```

Такий архітектурний підхід дозволяє централізувати бізнес-логіку та забезпечити послідовне збереження як самих згенерованих відповідей, так і результатів їхнього оцінювання за допомогою ResultRepository в єдиному циклі обробки запиту. Крім того, завдяки делегуванню вузькоспеціалізованих задач окремим сервісам (Service Layer), контролер залишається легковаговим і суворо дотримується принципу єдиної відповідальності (Single Responsibility Principle).

3.4 Експериментальне тестування модуля та оцінка результатів впровадження

На заключному етапі було проведено комплексне тестування програмного модуля. Верифікація логіки оцінювання здійснювалася за допомогою Unit-тестів на базі JUnit 5 та Mockito для Java, а також Vitest для React-компонентів [15]. Інтеграційні тести підтвердили коректність збереження результатів у PostgreSQL та стабільність роботи CORS-політик при взаємодії фронтенду з API.

Для перевірки працездатності алгоритму «LLM-as-a-Judge» та оцінки якості генерації контенту в умовах, наближених до реального використання проєкту цифрових двійників, було проведено практичний бенчмаркінг. Як тестовий сценарій використано історичний запит щодо постаті Яна Тарновського та його ролі у заснуванні міста Тернопіль. Цей кейс дозволив оцінити не лише фактологічну точність комерційних моделей, а й їхню здатність адаптуватися до заданої персони (Role-play) порівняно з локальним аватаром «Talking Heads».

Результати агрегованого тестування, що включають час очікування відповіді (Response Time), метрику косинусної подібності (Similarity Score) та експертну оцінку судді (Judge Score), зведено у таблицю 3.1.

Таблиця 3.1 — Результати порівняльного тестування мовних моделей

Модель	Час генерації (мс)	Similarity Score	Judge Score (0.0 - 1.0)
Gemini 2.0 Flash	2740	0.85	0.89
Claude 3 Haiku	3215	0.87	0.90
Talking Heads	3580	0.98	0.96
GPT-4o-mini	4120	0.88	0.92

Аналіз отриманих результатів дозволяє зробити наступні висновки:

– продуктивність та швидкодія: Найвищу швидкість генерації тексту продемонструвала модель Gemini 2.0 Flash (2740 мс), що підтверджує її ефективність для завдань, які вимагають мінімізації затримок у реальному часі. Найповільнішою в даній ітерації виявилася модель GPT-4o-mini (4120 мс);

– якість та контекстуалізація: Беззаперечним лідером за показниками якості стала спеціалізована модель Talking Heads. Текстова подібність до еталона склала 0.98, а оцінка LLM-судді — 0.96. Такі високі результати зумовлені архітектурою

аватара: генерація відбувається від першої особи («Я, Ян Тарновський, належу до давнього магнатського дому...»), що повністю відповідає очікуванням системи оцінювання;

– узагальненість комерційних API: Глобальні моделі (GPT-4o-mini, Claude 3 Haiku, Gemini) надали фактологічно правильні, але більш енциклопедичні та відчужені відповіді (Similarity Score в межах 0.85–0.88). Алгоритм LLM-as-a-Judge коректно зафіксував цю різницю в тональності, дещо знизивши їхні фінальні бали.

Графічне порівняння експертних оцінок (Judge Score), згенероване підсистемою аналітики та візуалізоване у клієнтському інтерфейсі за допомогою бібліотеки recharts, наведено на рисунку 3.3.

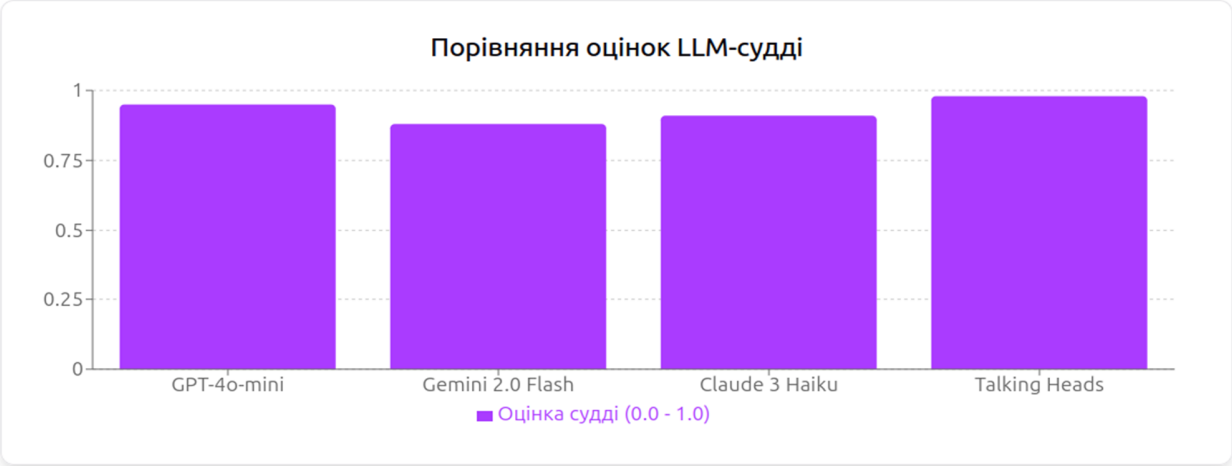

Рисунок 3.3 – Діаграма порівняння оцінок LLM-судді для різних мовних моделей

Експериментальне тестування підтверджує, що розроблений програмний модуль успішно виконує своє головне завдання — автоматизований паралельний збір відповідей та їхнє об'єктивне математичне оцінювання. Впровадження цієї системи дозволяє адміністраторам ефективно тестувати гіпотези, калібрувати системні промпти та постійно покращувати якість діалогів віртуальних аватарів, спираючись на конкретні числові метрики.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання, що полягає у розробці та програмній реалізації автоматизованого програмного модуля для збору, аналізу та об'єктивного оцінювання відповідей великих мовних моделей (LLM) у межах проєкту цифрових двійників університету.

За результатами проведеного дослідження та практичної розробки можна зробити такі висновки:

1. Спроєктовано та реалізовано сучасну архітектуру системи. На базі фреймворку Spring Boot (Java 17) створено серверну частину з чітким розділенням відповідальності (Controller, Service, Repository) та надійною інтеграцією з реляційною СУБД PostgreSQL. Клієнтську частину реалізовано як односторінковий застосунок (SPA) за допомогою бібліотеки React, що забезпечило реактивність інтерфейсу та зручну візуалізацію результатів бенчмаркінгу (за допомогою recharts) без надлишкового використання важких сторонніх UI-фреймворків.

2. Забезпечено гнучку взаємодію з LLM-провайдерами. Завдяки застосуванню патерну проєктування «Стратегія» та поліморфному підходу реалізовано паралельний ітеративний збір даних від комерційних API (OpenAI GPT-4o-mini, Claude 3 Haiku, Gemini 2.0 Flash) та закритої моделі локальних аватарів «Talking Heads». Налаштовано механізми обробки мережевих винятків та стандартизовано парсинг JSON-відповідей різної структури.

3. Впроваджено інноваційну методологію «LLM-as-a-Judge». Для переходу від суб'єктивного ручного оцінювання до автоматизованого метричного контролю розроблено конвеєр двоетапного аналізу. Він включає базовий розрахунок текстової подібності (Cosine Similarity) та експертну оцінку моделлю-арбітром. Для математичної агрегації результатів успішно інтегровано механізм нормалізації текстових відповідей судді на базі регулярних виразів (Regex).

4. Реалізовано надійну інфраструктуру та CI/CD конвеєр. Виконано повну контейнеризацію системи за допомогою Docker та оркестрацію через Docker Compose.

Використання багаторівневих (multi-stage) збірок дозволило оптимізувати розмір фінальних образів. Налаштовано автоматизований цикл безперервного розгортання (CI/CD) на базі GitHub Actions із застосуванням self-hosted runner, що гарантує безпечне оновлення сервісів та автоматичну верифікацію їхньої працездатності (Healthcheck).

5. Проведено експериментальне тестування та оцінку ефективності. Результати практичного бенчмаркінгу на основі краєзнавчого історичного запиту (постать Яна Тарновського) довели ефективність розробленої системи. Найвищу швидкість генерації продемонструвала модель Gemini 2.0 Flash (2740 мс). Водночас беззаперечним лідером за якістю контекстуалізації став локальний аватар «Talking Heads» (Similarity Score — 0.98, Judge Score — 0.96), що підтвердило перевагу спеціалізованих рольових моделей над загальними комерційними API у межах конкретної доменної області.

Таким чином, розроблений програмний модуль повністю відповідає сучасним вимогам до систем автоматизації тестування ШІ. Його впровадження дозволяє адміністраторам ефективно перевіряти гіпотези, калібрувати системні промпти та покращувати якість діалогів віртуальних аватарів на основі об'єктивних числових метрик.

Перспективи подальшої розробки полягають у розширенні функціоналу модуля для підтримки мультимодальних моделей (обробка зображень та звуку), а також у впровадженні складніших алгоритмів для детекції галюцинацій на ранніх етапах генерації. Мета кваліфікаційної роботи досягнута, а поставлені завдання виконані у повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OpenAI. GPT-4 Technical Report. 2023. URL: https://arxiv.org/abs/2303.08774 (дата звернення: 22.02.2026).

2. OpenAI. Офіційна документація моделей сімейства GPT-4 та методи інтеграції через API. 2025. URL: https://platform.openai.com/docs (дата звернення: 22.02.2026).

3. Google DeepMind. Технічна документація моделі Gemini 1.5 Pro: архітектура та контекстне вікно. 2025. URL: https://ai.google.dev (дата звернення: 22.02.2026).

4. Anthropic PBC. Довідник розробника: робота з моделлю Claude 3.5 Sonnet. 2025. URL: https://docs.anthropic.com (дата звернення: 22.02.2026).

5. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser Ł., Polosukhin I. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017. URL: https://arxiv.org/abs/1706.03762 (дата звернення: 22.02.2026).

6. Brown T. B., Mann B., Ryder N., Subbiah M., Kaplan J., Dhariwal P. et al. Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 1877–1901. URL: https://arxiv.org/abs/2005.14165 (дата звернення: 22.02.2026).

7. Bai Y., Jones A., Ndousse K., Askell A., Chen A., DasSarma N. et al. Constitutional AI: Harmlessness from AI Feedback. 2022. URL: https://arxiv.org/abs/2212.08073 (дата звернення: 22.02.2026).

8. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Küttler H., Lewis M., Yih W., Rocktäschel T., Riedel S., Kiela D. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. 2021. URL: https://arxiv.org/abs/2005.11401 (дата звернення: 22.02.2026).

9. Chase H. LangChain: фреймворк для розробки застосунків на основі LLM та технології Retrieval-Augmented Generation (RAG). 2025. URL: https://python.langchain.com (дата звернення: 22.02.2026).

10. Guo Z., Jin D., Lu S., Pan C., Wang Y., Wang J. A Survey on Evaluation of Large Language Models. 2024. URL: https://arxiv.org/abs/2307.03109 (дата звернення: 22.02.2026).

11. Zheng L., Chiang W., Sheng Y., Zhuang S., Wu Z., Zhuang Y., Lin Z., Li S., Li J., Gonzalez J., Stoica I. Judging LLM-as-a-Judge: Evaluating LLMs as Automated Judges. 2023. URL: https://arxiv.org/abs/2306.05685 (дата звернення: 22.02.2026).

12. Liu Y., Iter D., Xu Y., Wang S., Xu R., Zhu C. G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. 2023. URL: https://arxiv.org/abs/2303.16634 (дата звернення: 22.02.2026).

13. Walls C. Spring in Action, Sixth Edition. Shelter Island, NY: Manning Publications, 2022. 544 p.

14. Docker Inc. Docker Documentation: принципи контейнеризації та оркестрації застосунків. 2025. URL: https://docs.docker.com (дата звернення: 15.03.2026).

15. Соммервілл І. Інженерія програмного забезпечення / пер. з англ. Київ : Основи, 2018. 524 с.

16. Dubois Y., Li X., Taori R., Zhang T., Zamir A., Gonzalez J., Stoica I. AlpacaEval: An Automatic Evaluation Framework for Instruction-Following Language Models. 2024. URL: https://arxiv.org/abs/2307.12620 (дата звернення: 22.02.2026).

17. Chiang W.-L., Zheng L., Sheng Y., Li T., Zhuang S., Wu Z. et al. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. 2024. URL: https://arxiv.org/abs/2403.04132 (дата звернення: 22.02.2026).

18. Yourdon E. Modern Structured Analysis. Englewood Cliffs, NJ : Prentice-Hall, 1989. 672 p.

19. The PostgreSQL Global Development Group. PostgreSQL 16 Documentation: реляційні структури та типи даних. 2024. URL: https://www.postgresql.org/docs/16/ (дата звернення: 22.02.2026).

20. VMware Tanzu. Spring Boot Reference Guide. Офіційна документація фреймворку для розробки корпоративних Java-застосунків. 2025. URL: https://docs.spring.io/spring-boot/index.html (дата звернення: 15.03.2026).

21. VMware Tanzu. Spring Framework: WebClient documentation. Реактивна взаємодія в системах на базі Java. 2025. URL: https://docs.spring.io/spring-framework/reference/web/webflux-webclient.html (дата звернення: 15.03.2026).

22. VMware Tanzu. Spring Data JPA Reference Documentation: офіційна документація модуля для взаємодії з реляційними базами даних у Java-застосунках. 2025. URL: https://docs.spring.io/spring-data/jpa/reference/index.html (дата звернення: 15.03.2026).

23. Meta Platforms Inc. React Documentation: бібліотека для створення інтерфейсів користувача. 2025. URL: https://react.dev (дата звернення: 15.03.2026).

24. Vite.js. Vite: інструментарій наступного покоління для збірки фронтенд-проєктів. 2025. URL: https://vite.dev (дата звернення: 15.03.2026).

25. Irwin B. Mastering React Test-Driven Development. Birmingham: Packt Publishing, 2022. 462 p.

26. Копилов М. О. Програмний модуль автоматизованого збору та оцінювання відповідей великих мовних моделей // Інтелектуальні комп'ютерні системи та мережі: матеріали IV Всеукраїнської науково-практичної конференції студентів, аспірантів та молодих вчених, м. Тернопіль, 20 травня 2026 р. Тернопіль : ЗУНУ, 2026. С. 201–203.

27. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп'яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

Додаток А

Копія публікації

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

WEST UKRAINIAN NATIONAL UNIVERSITY

Комп'ютерна Інженерія

IV ВСЕУКРАЇНСЬКА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ СТУДЕНТІВ, АСПІРАНТІВ ТА МОЛОДИХ ВЧЕНИХ «ІНТЕЛЕКТУАЛЬНІ КОМП'ЮТЕРНІ СИСТЕМИ ТА МЕРЕЖІ»

ІКСМ весна 2026

20 ТРАВНЯ 2026

Наукове товариство ЗУНУ

KI.WUNU.EDU.UA/CONFERENCE/

ТЕРНОПІЛЬ

2026

Копилов М.О.
Програмний модуль автоматизованого збору та оцінювання відповідей LLM-моделей 201
Калюх Д.К.
Програмний модуль керування навчальними дефектами інтернет-магазину для формування навичок тестування програмного забезпечення 203
Гончарук К.С.
Програмний модуль класифікації станів фінансового ринку з використанням методів штучного інтелекту....205
Сідлецький Т. А.
Виявлення та відстеження рухомих об'єктів у відеопотоці на вбудованій обчислювальній платформі BeagleBone AI....207
Мамчур С. В.
Програмний модуль класифікації зображень на основі згорткової нейронної мережі для платформи NVIDIA Jetson 210
Цьома І.С.
Виявлення дефектів зварних швів із використанням глибинного навчання....212
Слотюк А.І.
Програмний модуль класифікації пошкоджень бетонних конструкцій на основі методів глибинного навчання....214
Божко М.В.
Мобільний застосунок з використанням технологій доповненої реальності для візуалізації музейного контенту з урахуванням потреб осіб з порушеннями слуху 217

Копилов М.О.
Студент 4 курсу ФКІТ ЗУНУ
Науковий керівник к.т.н., доцент Биковий П.Є., кафедра ІОСУ ЗУНУ

ПРОГРАМНИЙ МОДУЛЬ АВТОМАТИЗОВАНОГО ЗБОРУ ТА ОЦІНЮВАННЯ ВІДПОВІДЕЙ LLM-МОДЕЛЕЙ

Вступ. Стрімкий розвиток великих мовних моделей (Large Language Models, LLM) зумовив їх широке використання у програмуванні, освіті, бізнес-аналітиці та наукових дослідженнях. Водночас актуальною залишається проблема об'єктивного оцінювання якості відповідей різних моделей, оскільки результати їх роботи можуть відрізнятися за точністю, повнотою та релевантністю навіть для однакових запитів [1–4]. Проведення ручного тестування є трудомістким і не забезпечує необхідного рівня відтворюваності результатів, що обумовлює необхідність створення спеціалізованих програмних засобів автоматизованого збору та оцінювання відповідей LLM.

Постановка задачі. Метою роботи є розробка програмного модуля для автоматизованого збору, структурування та оцінювання відповідей великих мовних моделей на наборах тестових запитів. Об'єктом дослідження є процес взаємодії з великими мовними моделями та аналіз якості сформованих ними відповідей. Предметом дослідження виступають методи автоматизованого оцінювання відповідей LLM, алгоритми обробки природної мови та програмні засоби інтеграції з API мовних моделей.

Основний матеріал. У роботі проведено аналіз сучасних великих мовних моделей, архітектур Transformer та Retrieval-Augmented Generation (RAG), а також методів автоматизованого оцінювання якості текстових відповідей [1–5]. Особливу увагу приділено використанню класичних метрик BLEU, ROUGE, семантичної подібності на основі embeddings та підходу LLM-as-a-Judge для комплексного оцінювання результатів генерації [6, 7].

Загальна структура розробленого програмного модуля наведена на рисунку 1. Архітектура системи забезпечує повний цикл роботи з великими мовними моделями: від отримання тестових запитів та надсилання їх через API до автоматизованого аналізу отриманих відповідей і формування підсумкових оцінок. Використання модульного підходу забезпечує можливість підключення нових моделей та методів оцінювання без зміни загальної структури системи.

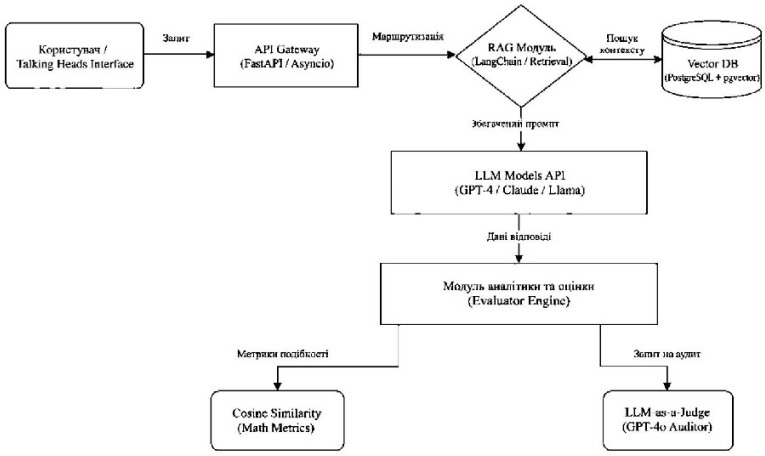

Рисунок 1 – Загальна структура програмного модуля автоматизованого збору та оцінювання відповідей LLM-моделей

Як показано на рисунку 1, користувацькі запити надходять через API Gateway до RAG-модуля, який виконує пошук релевантного контексту у векторній базі даних PostgreSQL з розширенням pgvector. Сформований запит передається до LLM-моделей, а отримані відповіді оцінюються за допомогою метрик семантичної подібності та підходу LLM-as-a-Judge.

Для реалізації програмного модуля використано мову програмування Python, сучасні засоби асинхронної взаємодії із зовнішніми API та систему керування базами даних PostgreSQL [8]. Програмне забезпечення підтримує автоматизований збір відповідей різних моделей, централізоване збереження результатів та виконання багатокритеріального аналізу якості відповідей.

Проведене експериментальне тестування підтвердило працездатність розробленого програмного модуля та можливість його використання для порівняльного аналізу сучасних мовних моделей. Отримані результати можуть бути використані під час вибору LLM для конкретних прикладних задач, а також у дослідженнях, пов'язаних із забезпеченням достовірності та якості генеративного штучного інтелекту.

Висновки. У результаті проведеного дослідження проаналізовано сучасні підходи до побудови та оцінювання великих мовних моделей. Розроблено архітектуру програмного модуля автоматизованого збору та оцінювання відповідей LLM, яка забезпечує інтеграцію з різними AI-сервісами, автоматизований аналіз результатів та підтримку масштабування системи. Запропоноване рішення може бути використане для проведення бенчмаркінгу мовних моделей, оцінювання якості генерації тексту та підтримки процесів вибору оптимальних моделей для практичного застосування.

Список літератури

1. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016.
2. Murphy K.P. Machine Learning: A Probabilistic Perspective. Cambridge : MIT Press, 2012.
3. Jurafsky D., Martin J.H. Speech and Language Processing. 3rd ed. Draft, 2024.
4. Vaswani A. et al. Attention Is All You Need // Advances in Neural Information Processing Systems (NeurIPS). 2017.
5. Lewis P. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks // Advances in Neural Information Processing Systems (NeurIPS). 2020.
6. Papineni K. et al. BLEU: a Method for Automatic Evaluation of Machine Translation // Proceedings of ACL. 2002.
7. Zheng L. et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena // NeurIPS Datasets and Benchmarks Track. 2023.
8. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/

Додаток Б

Специфікація API платформи «Talking Heads» (Postman Collection)


```
{
  "info": {
    "_postman_id": "caf162d6-0353-49ba-9b79-c40ffdc8abb1",
    "name": "TalkingHeads",
    "schema":
"https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
  },
  "item": [
    {
      "name": "AskQuestion",
      "request": {
        "method": "POST",
        "header": [{"key": "x-api-key", "value": "", "type": "text"}],
        "body": {
          "mode": "raw",
          "raw": "{\"exhibit_name\": \"emonakristine_adult\", \"question\":
\"привіт\"}"
        },
        "url": "https://sci-proj.wunu.edu.ua/talking-heads/api/client/chat"
      }
    },
    {
      "name": "GetExhibits",
      "request": {
        "method": "GET",
        "header": [{"key": "x-api-key", "value": "", "type": "text"}],
        "url": "https://sci-proj.wunu.edu.ua/talking-
heads/api/client/exhibits"
      }
    }
  ]
}
```

Додаток В

Фрагмент вихідного коду серверної частини

```
package com.example.demo.service;
import com.example.demo.model.Answer;
import com.example.demo.model.Question;
import com.example.demo.repository.AnswerRepository;
import org.springframework.stereotype.Service;
import java.util.ArrayList;
import java.util.List;

@Service
public class AnswerService {
    private final List<LLMService> llmServices;
    private final AnswerRepository answerRepo;
    public AnswerService(List<LLMService> llmServices, AnswerRepository
answerRepo) {
        this.llmServices = llmServices;
        this.answerRepo = answerRepo;
    }
    public List<Answer> generateAnswers(Question question) {
        List<Answer> answers = new ArrayList<>();
        for (LLMService llmService : llmServices) {
            long start = System.currentTimeMillis();
            String response = llmService.askModel(question.getText());
            long end = System.currentTimeMillis();
            Answer answer = new Answer();
            answer.setContent(response);
            answer.setModelName(llmService.getModelName());
            answer.setResponseTime(end - start);
            answer.setQuestion(question);
            answers.add(answerRepo.save(answer));
        }

        return answers;
    } }
```

Додаток Г

Конфігураційний файли для розгортання інфраструктури системи

```
services:
  db:
    container_name: llmautotestdb
    image: postgres:15-alpine
    restart: always
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USERNAME}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - spring-postgres

  api:
    container_name: llmautotest_api
    build: ./backend
    restart: always
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/${DB_NAME}
      SPRING_DATASOURCE_USERNAME: ${DB_USERNAME}
      SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD}
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
    depends_on:
      - db
    networks:
      - spring-postgres

  ui:
    container_name: llmautotest_ui
    build: ./admin-panel
    restart: always
    ports:
      - "80:80"
```

```
    depends_on:
      - api
    networks:
      - spring-postgres

networks:
  spring-postgres:
    driver: bridge

volumes:
  pgdata:
```

Додаток Д

Програмний код головної сторінки користувацького інтерфейсу

```
import { useState } from 'react';
import axios from 'axios';
import { BarChart, Bar, XAxis, YAxis, CartesianGrid, Tooltip, Legend,
ResponsiveContainer } from 'recharts';
import './App.css';

function HomePage() {
  const [question, setQuestion] = useState('');
  const [correctAnswer, setCorrectAnswer] = useState('');
  const [results, setResults] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const handleTestModels = async () => {
    if (!question.trim() || !correctAnswer.trim()) return;

    setIsLoading(true);
    setError(null);

    try {
      const response = await axios.post('/api/ai/evaluate', {
        text: question,
        correctAnswer: correctAnswer
      });

      setResults(response.data);
    } catch (err) {
      setError('Помилка з'єднання з сервером. Перевірте, чи запущено Spring
Boot та PostgreSQL.');
      console.error(err);
    } finally {
      setIsLoading(false);
    }
  };

  const winner = results.length > 0
      ? [...results].sort((a, b) => b.judgeScore - a.judgeScore)[0]
      : null;

  return (
      <div className="container">
        <h1>Головна сторінка тестування LLM</h1>

        <div className="question-box" style={{ flexDirection: 'column',
alignItems: 'center' }}>
          <input
              type="text"
```

```
            placeholder="Введіть тестове запитання..."
            value={question}
            onChange={(e) => setQuestion(e.target.value)}
            disabled={isLoading}
          />

          <input
            type="text"
            placeholder="Введіть еталонну відповідь (обов'язково для
порівняння)..."
            value={correctAnswer}
            onChange={(e) => setCorrectAnswer(e.target.value)}
            disabled={isLoading}
          />
          <button onClick={handleTestModels} disabled={isLoading ||
!question.trim() || !correctAnswer.trim()}>
            {isLoading ? 'Обробка та оцінювання...' : 'Запустити тест'}
          </button>
        </div>

        {error && <div style={{ color: 'red', textAlign: 'center',
marginBottom: '20px' }}>{error}</div>}

        {winner && !isLoading && (
            <div className="winner">
              Найкраща модель: {winner.answer.modelName} (Оцінка судді:
{winner.judgeScore}/10)
            </div>
        )}

        {!isLoading && results.length > 0 && (
            <div
                className="cards"
                style={{
                  display: 'flex',
                  flexDirection: 'column', // Відображення карток одна під
одною
                  gap: '15px'
                }}
            >
              {results.map((result, index) => (
                  <div
                      className="card"
                      key={index}
                      style={{
                        display: 'flex',
                        flexDirection: 'row',
                        alignItems: 'center',
                        justifyContent: 'space-between',
                        padding: '15px',
                        border: '1px solid #ddd',
                        borderRadius: '8px',
```

```
                    }}
                  >
                    <div style={{ flex: '0 0 150px' }}>
                      <h3 style={{ margin: 0
}}>{result.answer.modelName}</h3>
                    </div>

                    <div style={{ flex: '1', padding: '0 20px', borderLeft:
'1px solid #eee', borderRight: '1px solid #eee' }}>
                      <p style={{ margin: 0, fontSize: '14px'
}}>{result.answer.content}</p>
                    </div>

                    <div className="metrics" style={{ flex: '0 0 180px',
display: 'flex', flexDirection: 'column', gap: '8px', paddingLeft: '15px' }}>
                      <span style={{ fontSize: '13px' }}>Час:
<strong>{result.answer.responseTime} мс</strong></span>
                      <span style={{ fontSize: '13px' }}>Суддя:
<strong>{result.judgeScore}</strong></span>
                      <span style={{ fontSize: '13px' }}>Косинусна схожість:
<strong>{result.similarityScore}</strong></span>
                    </div>
                  </div>
              ))}
            </div>
        )}

        {!isLoading && results.length > 0 && (
            <div className="chart-box">
              <h2>Візуалізація оцінок LLM-моделей</h2>
              <div style={{ width: '100%', height: 350 }}>
                <ResponsiveContainer>
                  <BarChart data={results} margin={{ top: 20, right: 30,
left: 0, bottom: 5 }}>
                    <CartesianGrid strokeDasharray="3 3" vertical={false} />
                    <XAxis dataKey="answer.modelName" />
                    <YAxis domain={[0, 1]} />
                    <Tooltip contentStyle={{ borderRadius: '8px' }} />
                    <Legend />
                    <Bar dataKey="judgeScore" name="Оцінка судді (0.0 – 1.0)"
fill="#aa3bff" radius={[4, 4, 0, 0]} />
                  </BarChart>
                </ResponsiveContainer>
              </div>
            </div>
        )}
      </div>
  );
}

export default HomePage;
```