Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ МЕТОДУ ПОКРАЩЕННЯ РОБОТИ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ RAG
(тема)

Виконав:
здобувач 2 року навчання,
групи ІНФм-24-2
Уткін Є. І.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Науковий керівник доц. Тітова О. В.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Уткіну Євгенію Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методу покращення роботи великих мовних моделей з використанням технології RAG

затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 6 грудня 2025 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, метод покращення роботи великих мовних моделей Dense Retrieval, векторна база FAISS, векторні представлення тексту, засоби для програмної реалізації Python, FastAPI, React, редактор коду Visual Studio Code.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасних методів покращення якості роботи великих мовних моделей, та підходів до розширення контексту, таких як технологія RAG.
2. Аналіз літературних джерел щодо апробації методу покращення роботи великих мовних моделей з використанням технології RAG.

3. Формування покрокового алгоритму для вибраного методу Dense Retrieval.
4. Візуалізація сформованого покрокового алгоритму.
5. Розробка програмного застосунку, що надасть змогу дослідити метод покращення роботи великих мовних моделей з використанням технології RAG.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми підвищення точності великих мовних моделей при роботі з зовнішніми джерелами знань, об'єкт та мета дослідження, постановка задачі, блок-схема, алгоритма вибраного методу покращення роботи LLM, набір тестових даних для перевірки ефективності методу, ілюстрація інтерфейсу розробленого застосунку, приклад результатів тестування із зазначенням релевантних фрагментів та аналізу відповідних метрик, висновки, перспективи та апробація роботи.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	29.09.2025	
2	Аналіз завдання, підбір літератури	29.09.25-10.10.25	
3	Аналіз літератури з досліджуваної проблеми	10.10.25-13.10.25	
4	Особливості методу покращення великих мовних моделей з використанням технології RAG	13.10.25-20.10.25	
5	Дослідження методу покращення роботи великих мовних моделей з використанням технології RAG	20.10.25-28.10.25	
6	Програмна реалізація	28.10.25-10.11.25	
7	Обґрунтування отриманих результатів	10.11.25-17.11.25	
8	Оформлення пояснювальної записки	17.11.25-30.11.25	
9	Перевірка на нормоконтроль	01.12.25-05.12.25	
10	Перевірка на плагіат	05.11.25-07.12.25	
11	Рецензування	07.11.25-08.12.25	
12	Підготовка презентації та доповіді	08.11.25-10.12.25	
13	Занесення роботи в електронний архів	10.11.25-12.12.25	
14	Попередній захист кваліфікаційної роботи	15.12.25	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Тітова О. В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 84 с., 2 табл., 14 рис., 46 джерел.

ВЕБЗАСТОСУНОК, ВЕКТОРИЗАЦІЯ, ПРОМПТ, ЧАТ-БОТ, RAG, NLP, EMBEDDING, DENSE RETRIEVAL, FAISS, SENTENCE TRANSFORMERS, LLM, PYTHON, FASTAPI, REACT, TYPESCRIPT, REST API.

Об’єктом дослідження є процес обробки текстової інформації та пошуку релевантних знань у системах, на основі технології RAG та LLM.

Предметом дослідження є методи семантичного пошуку у RAG-системах на основі векторних представлень тексту та LLM.

Метою дослідження є аналіз, проєктування та реалізація підходу до підвищення якості відповідей LLM шляхом використання технології RAG, а також оцінювання ефективності цього підходу на практичному застосунку.

Використано метод Dense Retrieval, модель SentenceTransformers, а також векторну базу FAISS. Для генерації відповідей застосовано модель GPT-4o-mini. Проведено аналітичний огляд сучасних підходів до побудови RAG-архітектур, механізмів індексації та взаємодії LLM із зовнішніми джерелами даних.

Наукова новизна роботи полягає в удосконаленні процесу семантичного пошуку шляхом застосування щільних векторних уявлень і динамічного формування контексту для LLM.

Взаємозв’язок з іншими роботами полягає у підвищенні ефективності LLM, підходами семантичного пошуку й інтеграції зовнішніх знань.

Рекомендації щодо використання результатів роботи сформовано на основі проведених експериментів і подано у висновках.

У результаті дослідження розроблено застосунок, що виконує індексацію документів, здійснює семантичний пошук на основі технології RAG та генерує точні відповіді, комбінуючи інформацію з бази знань і можливості LLM.

ABSTRACT

Explanatory note to the qualification work: 84 pages, 2 table, 14 figures, 46 sources.

WEB APPLICATION, VECTORIZATION, PROMPT, CHAT-BOT, RAG, NLP, EMBEDDING, DENSE RETRIEVAL, FAISS, LLM, SENTENCE TRANSFORMERS, PYTHON, FASTAPI, REACT, TYPESCRIPT, REST API.

The object of the research is the process of processing text information and searching for relevant knowledge in systems based on RAG and LLM technology.

Subject of the research is methods of semantic search in RAG systems based on vector representations of text and LLM.

The aim of the research is to analyze, design and implement an approach to improving the quality of LLM responses using RAG technology, as well as to evaluate the effectiveness of this approach in practical application.

The Dense Retrieval method, the SentenceTransformers model, and the FAISS vector database were used. The GPT-4o-mini model was used to generate responses. An analytical review of modern approaches to building RAG architectures, indexing mechanisms, and LLM interaction with external data sources was conducted.

Scientific novelty of the work lies in improving the semantic search process by using dense vector representations and dynamic context formation for LLM.

Interconnection with other works is to increase the efficiency of LLM, semantic search approaches and integration of external knowledge.

Recommendations for using the results of the work are formed on the basis of the experiments conducted and presented in the conclusions.

As a result of the research, an application was developed that performs document indexing, performs semantic search based on RAG technology and generates accurate answers, combining information from the knowledge base and LLM capabilities.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

LLM – Large Language Model (велика мовна модель)

RAG – Retrieval-Augmented Generation (генерація з підсиленням пошуком)

RNN – Recurrent Neural Network (рекурентна нейронна мережа)

LSTM – Long Short-Term Memory (довга короткострокова пам’ять)

GPT – Generative Pre-trained Transformer (генеративний попередньо натренований трансформер)

LLaMA – Large Language Model Meta AI (велика мовна модель від Meta AI)

IoT – Internet of Things (інтернет речей)

NLP – Natural Language Processing (обробка природної мови)

ML – Machine Learning (машинне навчання)

AI – Artificial Intelligence (штучний інтелект)

API – Application Programming Interface (інтерфейс прикладного програмування)

ASGI – Asynchronous Server Gateway Interface (асинхронний шлюзовий інтерфейс сервера)

FAISS – Facebook AI Similarity Search (бібліотека для пошуку схожості у векторних просторах)

PQ – Product Quantization (продуктна квантизація)

HNSW – Hierarchical Navigable Small World Graph (ієрархічний граф швидкого пошуку сусідів)

AWS – Amazon Web Services (хмарні обчислювальні сервіси Amazon)

DOM – Document Object Model (об’єктна модель документа)

JSX – JavaScript XML (розширення синтаксису React для опису інтерфейсів)

HTTP – HyperText Transfer Protocol (протокол передавання гіпертексту)

REST API – Representational State Transfer API (архітектурний стиль побудови вебзастосунків)

ВСТУП

Стрімкий розвиток технологій штучного інтелекту та обробки природної мови за останні роки суттєво змінив підходи до роботи з текстовою інформацією. Великі мовні моделі, які побудовані на трансформерній архітектурі, продемонстрували здатність генерувати зв'язні, логічні та стилістично коректні тексти, що робить їх універсальним інструментом для вирішення широкого кола задач – від автоматизованого перекладу та підсумовування до асистування користувачу в діалоговому режимі. Попри це, традиційні LLM мають низку обмежень, зокрема залежність від попереднього навчання, схильність до галюцинацій та неможливість працювати з актуальними або приватними даними без зовнішніх джерел знань [1].

У відповідь на ці обмеження з'явився підхід Retrieval-Augmented Generation, який поєднує генеративні моделі з механізмами пошуку релевантної інформації у зовнішніх базах знань. Такий підхід дозволяє суттєво підвищити точність і достовірність відповідей, оскільки у процесі генерації LLM спирається не лише на власні параметри, а й на реальні документи, знайдені системою пошуку. Таким чином, RAG відкриває можливість створення інтелектуальних систем, які не лише аналізують текст, а й активно залучають зовнішній контекст, що робить їх придатними для навчальних, аналітичних, юридичних, корпоративних та науково-дослідних задач.

Одним із ключових компонентів сучасних RAG-систем є метод Dense Retrieval – підхід, який передбачає представлення текстів та користувацьких запитів у вигляді багатовимірних векторів, що відображають їхній семантичний зміст. На відміну від класичних методів, заснованих на ключових словах, Dense Retrieval працює з контекстом, дозволяючи знаходити документи, що є смислово близькими, навіть якщо формулювання у них відрізняються. Цей підхід став можливим завдяки розвитку моделей, таких як SentenceTransformers та ефективних структур даних для пошуку найближчих векторів, таких як FAISS. Саме використання таких механізмів забезпечує високу швидкість та

масштабованість RAG-рішень, дозволяючи працювати з великими масивами інформації у реальному часі.

Актуальність теми зумовлена тим, що на сьогодні RAG-архітектури стали одним із найперспективніших напрямків розвитку інформаційних систем. У багатьох сферах – від бізнес-аналітики до освіти – існує очевидна потреба у рішеннях, здатних швидко обробляти документи та відповідати на запити з урахуванням специфіки домену. Стандартні чат-боти або LLM-моделі не можуть задовольнити такі вимоги через обмеження контекстного вікна, неможливість використання динамічних даних та залежність від якості навчальних наборів. Саме тому системи, що комбінують генеративні моделі з актуальною базою знань, мають стратегічне значення. Крім того, завдяки можливості локального розгортання та повного контролю над індексацією документів такі системи відповідають вимогам конфіденційності, що є критично важливим у корпоративних середовищах.

Наукова задача, що вирішується в роботі, полягає у дослідженні точності й продуктивності методів щільного семантичного пошуку та визначенні того, яким чином параметри embedding-моделей, структура індексу та спосіб формування контексту впливають на якість відповідей великої мовної моделі. Важливою частиною дослідження є аналіз архітектури системи, вибір технологічного стеку та порівняння результатів роботи LLM з RAG і без [2].

Таким чином, комплексне поєднання сучасних методів NLP, Dense Retrieval та генеративних моделей забезпечує створення гнучкої й ефективної системи, здатної працювати з великими обсягами текстових даних і надавати точні, аргументовані відповіді. Результати роботи можуть бути корисними у розробці інтелектуальних помічників, пошукових платформ, освітніх систем, корпоративних чат-ботів та інших застосунків, де необхідне поєднання генеративних можливостей LLM із доступом до актуальних даних.

1 СУЧАСНИЙ СТАН РОЗВИТКУ LLM ТА МЕТОДИ ПОКРАЩЕННЯ ЇХ РОБОТИ

1.1 Поняття та архітектура LLM

Великі мовні моделі (LLM) є одним із ключових досягнень сучасної штучної інтелектуальної індустрії. Вони забезпечують здатність машин опрацьовувати природну мову та генерувати тексти, які за структурою та стилем наближені до людських. На відміну від класичних алгоритмів обробки мови, які спиралися на жорстко закодовані правила чи статистичні методи, LLM будуються на архітектурі глибинних нейронних мереж, що дозволяє їм узагальнювати закономірності з величезних масивів даних.

Основою LLM є архітектура трансформерів, запропонована у 2017 році в статті «Attention is All You Need». Її принципова відмінність полягає у використанні механізму самоуваги (self-attention), що дає змогу моделі визначати контекстні залежності між словами у межах як коротких, так і довгих текстів. У класичних рекурентних нейронних мережах (RNN) чи мережах типу LSTM опрацювання тексту було послідовним, що обмежувало можливості паралельних обчислень і ускладнювало роботу з великими корпусами даних. Трансформери усунули цю проблему, оскільки вся послідовність слів опрацьовується одночасно, а ваги уваги визначають, які частини тексту є найбільш релевантними для поточного завдання [3].

Узагальнено архітектура LLM складається з таких ключових компонентів:

– шарів енкодера та декодера: енкодер відповідає за перетворення вхідного тексту у векторне представлення, тоді як декодер виконує зворотне перетворення у природномовний текст. У моделях типу GPT використовується лише декодерна частина трансформера, оптимізована для генерації послідовності;

– механізму самоуваги: кожне слово перетворюється у вектор та аналізується у зв’язку з іншими словами в межах контексту. Завдяки цьому модель здатна враховувати навіть віддалені залежності у реченні, наприклад, між підметом та присудком, які можуть бути розділені кількома підрядними конструкціями;

– позиційних кодувань: оскільки трансформери обробляють всю послідовність одночасно, потрібно вводити додаткову інформацію про порядок слів. Для цього застосовуються спеціальні вектори, що кодують позиції токенів у тексті;

– масштабованих параметрів: великі мовні моделі відрізняються насамперед кількістю параметрів – від сотень мільйонів до сотень мільярдів. Саме масштаб дозволяє моделі краще узагальнювати інформацію та демонструвати високі результати у широкому спектрі завдань – від перекладу та сумаризації до написання коду чи аналізу складних наукових текстів.

Варто підкреслити, що якість роботи LLM безпосередньо залежить від обсягів і якості даних, на яких вони навчені. Сучасні моделі (наприклад, GPT-4, LLaMA 3 чи Claude) тренуються на корпусах текстів, що налічують трильйони слів. Це включає відкриті інтернет-ресурси, наукові статті, програмний код, книги, енциклопедії та інші джерела. Завдяки цьому модель набуває широких знань про світ, хоча вони завжди обмежені часом завершення навчання.

Разом із тим, попри вражаючі можливості, LLM мають низку обмежень. Зокрема, вони не володіють справжнім «розумінням» тексту, а лише виявляють статистичні закономірності у даних. Це призводить до явища галюцинацій – ситуацій, коли модель генерує фактично неправдиву або вигадану інформацію, подаючи її у впевненій формі. Також варто враховувати, що збільшення розмірів моделей вимагає колосальних обчислювальних ресурсів, як на етапі навчання, так і під час їхнього використання.

Ще одним важливим аспектом архітектури LLM є контекстне вікно – кількість токенів (слів або частин слів), які модель може врахувати одночасно. У ранніх версіях GPT воно становило 2-4 тисячі токенів, тоді як сучасні системи

досягають 128 тисяч і навіть більше. Втім, навіть таке розширення не завжди достатнє, щоб забезпечити роботу з великими документами, що вимагає застосування додаткових стратегій.

У наукових колах також ведуться активні дискусії щодо шляхів підвищення ефективності LLM. Найбільш поширеними є три підходи:

– fine-tuning: додаткове навчання на спеціалізованих даних;

– prompt engineering: створення оптимізованих підказок, що спрямовують модель;

– retrieval-augmented generation (RAG) – поєднання генерації з пошуком релевантної інформації у зовнішніх базах знань.

Таким чином, архітектура LLM – це складна система, що базується на трансформерах та багатомільярдних параметрах. Вона забезпечує гнучкість і багатофункціональність у сфері обробки природної мови, однак водночас ставить перед дослідниками завдання подолання обмежень, зокрема проблеми галюцинацій, контекстних обмежень і високої вартості обчислень. Саме ці виклики зумовили необхідність розробки нових методів підвищення якості роботи мовних моделей, серед яких провідне місце належить технології RAG.

1.2 Проблеми використання LLM

Попри значні досягнення у сфері розробки та застосування великих мовних моделей, їх впровадження в реальні інформаційні системи супроводжується низкою суттєвих проблем. Ці недоліки не лише впливають на точність і надійність отримуваних відповідей, але й визначають можливості інтеграції таких систем у реальні практичні сценарії – від освіти і бізнесу до державного управління [4].

У подальших підпунктах здійснено детальний розгляд кожного з виявлених обмежень, висвітлено їх сутність, причини виникнення та можливі

наслідки для користувачів і розробників. Такий підхід дозволяє систематизувати виявлені проблеми та допоможе обрати найкращі шляхи їх вирішення.

1.2.1 Проблема галюцинацій LLM

Одним із найбільш суттєвих недоліків LLM є схильність до генерації так званих галюцинацій – відповідей, що не мають фактичного підґрунтя, проте сформульовані впевнено і переконливо. Причиною цього явища є спосіб, у який функціонує модель: замість осмислення інформації вона прогнозує наступні слова, спираючись на ймовірнісні зв’язки у навчальних даних. Як наслідок, у ситуаціях, коли інформація неповна, суперечлива або відсутня, модель може «вигадувати» дані, намагаючись заповнити прогалину.

Практичний ризик галюцинацій особливо відчутний у сферах, де важлива точність. Помилкові відповіді в юридичних консультаціях, медичних рекомендаціях або фінансових аналітичних висновках можуть мати серйозні наслідки. У наукових та освітніх контекстах це призводить до поширення неправдивих фактів та хибних трактувань. Наявність галюцинацій підриває довіру до систем штучного інтелекту й обмежує їх застосування у середовищах, де необхідне чітке обґрунтування інформації.

1.2.2 Обмеження контекстного вікна

Ще одним обмеженням є фіксована довжина контекстного вікна – максимальний обсяг тексту, який може бути опрацьований моделлю за один запит. Навіть при значному його збільшенні в останніх поколіннях моделей, обсяг даних, що доступний для аналізу, залишається кінцевим. Це створює бар’єри при роботі з великими документами або задачами, що вимагають аналізу

великого масиву інформації – наприклад, під час опрацювання технічної документації, контрактів, наукових праць або навчальних матеріалів.

Обмеження контекстного вікна безпосередньо впливає на здатність моделі врахувати всі необхідні деталі, що призводить до втрати логічної цілісності відповідей. Коли обсяг інформації перевищує можливості моделі, користувач змушений розбивати дані на частини, що додатково ускладнює процес взаємодії. У результаті зростає ризик хибних висновків, оскільки модель аналізує лише фрагмент інформації без повного контексту, у якому ці дані слід розглядати [5].

1.2.3 Високі обчислювальні витрати

Створення та експлуатація великих мовних моделей потребує значних обчислювальних ресурсів. Тренування LLM триває тижнями або місяцями та вимагає доступу до суперкомп'ютерів або кластерів із високопродуктивними графічними процесорами. Обсяги енергії, необхідні для підготовки моделей із сотнями мільярдів параметрів, співставні з річним споживанням електроенергії невеликих міст, що викликає не лише фінансові, а й екологічні питання. Тому активно розвиваються дослідження у напрямку створення компактніших моделей та оптимізації процесу навчання, що дозволяє зменшити витрати без втрати якості результатів.

Вартість обчислень відчутна і на етапі використання моделей. Обробка складних запитів, особливо в режимі реального часу або при масштабуванні для великої кількості користувачів, потребує дорогих серверних рішень. Це робить впровадження LLM економічно недоцільним для багатьох організацій, зокрема освітніх установ, державних органів та малого бізнесу. Навіть використання моделей через сторонні API пов'язане з витратами, які зростають пропорційно кількості запитів і складності завдань. Таким чином, фінансова та технологічна недоступність є суттєвим стримуючим фактором поширення LLM у масовому застосуванні [6].

1.2.4 Проблема застарівання знань у LLM

Ще однією проблемою є властива LLM статичність знань. Моделі навчаються на фіксованих наборах даних, що відображають стан інформаційного середовища на момент завершення їх тренування. Відповідно, будь-які події, наукові відкриття, зміни в законодавстві чи нові технології, що з'явилися після цього, залишаються поза межами наявних знань моделі. В умовах стрімкого розвитку інформаційного простору це призводить до швидкої втрати актуальності.

Особливо гострим це питання є у сферах, де інформація змінюється динамічно: у фінансах, медицині, міжнародному праві, інформаційних технологіях. Користувачі можуть отримувати коректні з точки зору минулих даних відповіді, які однак вже не відповідають сучасному стану речей. Регулярне перенавчання моделей є надзвичайно ресурсомістким і далеко не завжди виправданим з економічної та технічної точки зору. Отже, статичність знань обмежує практичну користь LLM у задачах, де актуальність інформації є критичною [7].

1.3 Методи покращення роботи LLM

Швидкий розвиток великих мовних моделей спричинив появу широкого спектра підходів, спрямованих на підвищення точності, релевантності та практичної користі результатів їх роботи. Попри значні можливості LLM, описані у попередніх підрозділах, їх застосування у реальних умовах вимагає адаптації під конкретні задачі, а також подолання низки обмежень. У зв'язку з цим дослідники та інженери пропонують різні стратегії покращення ефективності таких моделей [8].

У наступній частині буде детально розглянуто методи покращення роботи LLM та проаналізовано їх доцільність у контексті сучасних потреб користувачів.

1.3.1 Fine-tuning

Одним із базових способів покращення роботи великих мовних моделей є донавчання або fine-tuning, яке передбачає адаптацію попередньо натренованої моделі до задачі конкретної предметної галузі. Суть методу полягає у подальшому навчанні моделі на спеціалізованих наборах даних, що дозволяє налаштувати її поведінку на конкретний стиль тексту або тематику. Наприклад, модель, навчена на загальних текстових корпусах, може бути додатково адаптована для юридичних консультацій, технічного супроводу користувачів чи медичних довідок. Такий підхід дає змогу суттєво підвищити точність відповідей та зменшити кількість помилкових результатів, оскільки модель працює на основі знань, релевантних до обраної сфери.

Водночас fine-tuning є досить вимогливим з погляду ресурсів. Для отримання стабільних результатів необхідні ретельно відібрані навчальні дані, що відповідають стандартам якості, не містять суперечностей та охоплюють достатній спектр тем. Крім того, процес донавчання супроводжується значними обчислювальними витратами, що ускладнює його застосування для невеликих компаній чи індивідуальних розробників. У випадку надмірної спеціалізації існує ризик «звуженого мислення» моделі, коли вона починає ігнорувати альтернативні варіанти відповідей і втрачати універсальність. Незважаючи на це, fine-tuning залишається ефективним засобом, коли необхідно створити рішення для вузької професійної сфери.

1.3.2 Prompt engineering

Другим напрямом покращення роботи LLM є оптимізація формулювань запитів, відома як prompt engineering. Вона базується на розумінні того, що модель генерує відповіді на основі наданої інструкції, тому якість та структура запиту безпосередньо впливають на результат. Практика показує, що навіть

незначні зміни у формулюванні можуть суттєво покращити зміст, логічність і точність відповіді. Робота з промптами передбачає застосування різних технік: уточнення ролі моделі («Уяви, що ти юрист…»), подання прикладів бажаного формату відповіді, структурування завдання або використання покрокових інструкцій.

Попри простоту впровадження, цей метод має певні обмеження. По-перше, ефективне створення промптів вимагає досвіду та розуміння особливостей роботи моделей. По-друге, навіть оптимально сформульований запит не гарантує правильності фактів, оскільки модель все одно залишається залежною від власних внутрішніх знань, які, як зазначалося раніше, можуть бути застарілими. Prompt engineering більше підходить для покращення стилю, структури й чіткості відповідей, ніж для розв'язання проблеми їх достовірності. Однак у практичному використанні цей підхід часто виявляється найшвидшим та найдоступнішим способом підвищити якість взаємодії з LLM без додаткових витрат ресурсів.

1.3.3 Retrieval-Augmented Generation

Найбільш перспективним і швидко поширюваним методом підвищення ефективності LLM є підхід, що поєднує генеративні можливості моделі з механізмами пошуку актуальної інформації у зовнішніх джерелах – Retrieval-Augmented Generation. На відміну від попередніх методів, що намагаються вдосконалити саму модель або спосіб взаємодії з нею, RAG дає змогу доповнювати відповіді реальними даними з баз знань, документів, статей, корпоративних репозиторіїв та інших джерел. Принцип роботи полягає у тому, що перед формуванням відповіді модель здійснює пошук релевантної інформації за допомогою алгоритмів векторного зіставлення, а вже потім генерує текст, спираючись на знайдені матеріали.

Цей підхід має одразу кілька переваг. По-перше, він суттєво знижує ймовірність галюцинацій, оскільки відповідь ґрунтується на перевірених даних.

По-друге, RAG забезпечує доступ до актуальної інформації без потреби перевчати модель – достатньо оновлювати базу знань. По-третє, цей метод фактично знімає обмеження контекстного вікна: модель аналізує лише релевантний фрагмент великого масиву даних, а не весь контент одночасно, що є ефективним як з погляду продуктивності, так і з погляду точності.

Водночас RAG також має власні виклики. Одним із них є потреба у правильному формуванні та підтримці бази даних, що забезпечує якість пошуку. Якщо дані неповні або містять недостовірну інформацію, відповідь моделі також може виявитися некоректною. Крім того, реалізація RAG потребує технічних рішень для обробки документів, створення ембеддінгів, організації пошуку і побудови інфраструктури, яка забезпечує швидку взаємодію між компонентами.

1.4 Перспективи розвитку LLM

Стрімкий розвиток великих мовних моделей протягом останніх років свідчить про те, що ця технологія перебуває лише на початковому етапі свого становлення. LLM вже змінюють підходи до обробки інформації, комунікації, навчання та автоматизації, однак потенціал їх подальшого вдосконалення залишається значним. У найближчі роки очікується поява нових поколінь моделей, здатних розширити існуючі можливості та усунути нинішні обмеження. Перспективи розвитку LLM включають як технологічні закономірності, так і соціальні аспекти, пов’язані з їх впливом на суспільство, ринок праці та етичні норми.

Одним із ключових напрямів розвитку є підвищення ефективності та оптимізація моделей. Попри те, що сучасні моделі досягли високих результатів генерації тексту, вони залишаються ресурсомісткими. Очікується, що майбутнє покоління LLM буде більш компактним, енергоефективним і швидким, без втрати якості відповідей. Вже сьогодні помітна тенденція до створення «легших» моделей, які можуть працювати локально, на персональних пристроях або в

автономних системах. Це дозволить розширити коло застосувань LLM і підвищить конфіденційність обробки даних, оскільки не вимагатиме відправлення інформації на зовнішні сервери. Також розвиваються підходи, спрямовані на зменшення кількості параметрів моделей без зниження точності, що має сприяти кращій доступності технології для малого та середнього бізнесу, освітніх установ та окремих користувачів [9].

Іншим важливим вектором розвитку є підвищення рівня інтегрованості моделей із зовнішніми джерелами знань. Сучасні LLM демонструють значний прогрес, але їхня здатність оперувати актуальною інформацією залишається обмеженою. Тому очікується широке впровадження гібридних підходів, які об’єднуватимуть генеративні можливості моделей з інструментами пошуку та аналізу даних у реальному часі. Одним із таких напрямів є подальший розвиток концепції Retrieval-Augmented Generation, яка вже зарекомендувала себе як ефективний спосіб зменшення кількості помилкових відповідей. У перспективі цей підхід може розвинутися у складніші системи, здатні не просто знаходити потрібну інформацію, а й перевіряти її достовірність та надавати користувачеві структурований аналіз замість окремих фрагментів.

Важливою тенденцією є перехід до мультимодальних мовних моделей, які працюють не лише з текстом, а й з іншими типами даних: зображеннями, відео, аудіо, кодом, сенсорними сигналами та мультимедійним контентом. Такі системи здатні об’єднувати інформацію з різних джерел, що дозволяє створювати комплексні рішення. Наприклад, мультимодальна модель може аналізувати зображення медичного обстеження, знаходити в ньому важливі деталі, доповнювати цей аналіз текстовою консультацією та пояснювати пацієнтові у доступній формі. У сфері освіти мультимодальні моделі можуть поєднувати відео, текст і графіку для формування більш ефективного навчального матеріалу.

Також, окремої уваги заслуговує розвиток LLM-агентів, здатних виконувати складні послідовні завдання без постійного втручання користувача. На відміну від сучасних моделей, що переважно відповідають на окремі запити,

LLM-агенти зможуть аналізувати ситуацію, формувати цілі, планувати кроки для їх досягнення та коригувати дії залежно від обставин. Такий підхід може суттєво трансформувати бізнес-процеси, роботу підприємств, навчання та побут користувачів. Наприклад, агент може автоматично здійснювати пошук інформації, аналізувати її, проводити розрахунки, створювати звіти та пропонувати варіанти рішень.

Паралельно з технологічними змінами розвиватимуться етичні та правові аспекти використання LLM. Зростання впливу таких моделей на інформаційний простір суспільства, ринок праці, медіа, політику та інші сфери вимагає формування чітких нормативно-правових рамок. У найближчі роки буде посилено регулювання щодо захисту авторських прав, прозорості використання даних, кібербезпеки та запобігання маніпуляціям зі штучною інформацією. Важливими викликами залишатимуться питання академічної доброчесності, ризики створення дезінформації та можливість надмірної залежності суспільства від автоматизованих систем. У цих умовах надзвичайно важливо розвивати не лише технологічну складову LLM, а й культуру відповідального їх використання.

Окремої уваги заслуговує вплив LLM на ринок праці та професійну діяльність. Уже сьогодні спостерігається трансформація задач у багатьох професіях, де рутинні або механічні операції поступово передаються штучному інтелекту. У перспективі LLM можуть суттєво змінити вимоги до кваліфікацій спеціалістів у галузях аналітики, освіти, маркетингу, програмування, документаційного супроводу та багатьох інших. При цьому LLM не стільки замінюватимуть фахівців, скільки змінюватимуть характер їхньої роботи. Паралельно зникатимуть окремі типи завдань, але з’являтимуться нові професії, пов’язані з розробкою, налаштуванням та підтримкою систем на базі штучного інтелекту.

Важливим аспектом розвитку LLM є їх інтеграція з іншими технологіями штучного інтелекту та суміжними галузями. Поєднання LLM із системами комп’ютерного зору, аналізу аудіо, робототехнікою та інтернет речами (IoT)

може створити нові інтелектуальні екосистеми. Такі системи будуть здатні взаємодіяти зі світом на фізичному рівні, аналізувати дані з різних джерел, приймати рішення та виконувати дії в реальному середовищі. Це відкриває перспективи для розвитку «розумних» пристроїв, персональних цифрових асистентів і роботизованих помічників у побуті чи на роботі.

Підсумовуючи, можна відзначити, що перспективи розвитку великих мовних моделей охоплюють як технологічні вдосконалення, так і соціальні зміни. У найближчі роки очікується поява більш продуктивних, безпечних і адаптивних моделей, здатних забезпечувати достовірність інформації, персоналізовану підтримку та гнучку інтеграцію з іншими інтелектуальними системами. Розвиток LLM впливатиме не лише на технологічний сектор, а й на структуру суспільства, систему освіти, економіку, ринок праці та повсякденне життя людей. Це створює як нові можливості, так і виклики, що потребують усвідомленого та відповідального підходу до застосування штучного інтелекту.

1.5 Постановка задачі дослідження

Таким чином, «Дослідження методу покращення роботи LLM систем з використанням технології RAG» є актуальним завданням. Прийнято рішення щодо розроблення програмного застосунку, який реалізовуватиме підхід RAG з метою практичної перевірки його ефективності. Такий застосунок має продемонструвати можливість підвищення якості відповідей LLM під час взаємодії з користувачем шляхом залучення релевантної зовнішньої інформації, а також створити можливості для аналізу результатів та формування висновків щодо доцільності використання технології RAG у реальних умовах [10].

Об’єктом дослідження є процес обробки текстової інформації та пошуку релевантних знань у системах, на основі технології RAG та LLM.

Предметом дослідження є методи семантичного пошуку у RAG-системах на основі векторних представлень тексту та LLM.

Метою дослідження є аналіз, проєктування та реалізація підходу до підвищення якості відповідей LLM шляхом використання технології RAG, а також оцінювання ефективності цього підходу на практичному застосунку.

Для досягнення мети необхідно вирішити такі завдання:

– здійснити огляд і аналіз наукових джерел щодо особливостей роботи LLM та існуючих методів покращення їх ефективності;

– дослідити принципи функціонування технології RAG та визначити її переваги порівняно з іншими підходами;

– розробити концептуальну архітектуру застосунку, що реалізує принципи RAG для покращення роботи LLM;

– сформувати алгоритм обробки запитів користувача із залученням механізмів пошуку релевантної інформації;

– реалізувати програмний застосунок, що забезпечує генерацію відповідей із використанням RAG;

– провести експериментальне оцінювання якості відповідей з використанням RAG та без нього, проаналізувати отримані результати та сформувати висновки щодо ефективності застосування технології.

2 ТЕХНОЛОГІЯ RAG ТА ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДІВ І ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Концепція Retrieval-Augmented Generation

Retrieval-Augmented Generation є сучасною технологією, яка поєднує методи інформаційного пошуку (retrieval) та генеративні можливості великих мовних моделей (LLM). На відміну від звичайних LLM систем, які створюють відповіді лише на основі даних, якими вони були натреновані, система RAG має здатність динамічно звертатись до зовнішніх баз знань. Таким чином, вона не просто пам’ятає інформацію, а знаходить її у базі даних або колекції документів, адаптуючи отримані знання для створення релевантної відповіді [11].

Архітектурно RAG складається з двох основних компонентів: модуля пошуку (retriever) та модуля генерації (generator):

– retriever виконує пошук у векторному сховищі даних (vector store), де тексти збережено у вигляді векторних представлень. Для цього використовуються моделі ембедингів, які перетворюють речення у багатовимірні числові вектори, зберігаючи семантичну близькість між фразами;

– generator отримує найкращі знайдені фрагменти тексту, об’єднує їх із запитом користувача та формує підсумкову підказку. А LLM звертається до зовнішнього джерела пам’яті, яке може постійно оновлюватися.

Такий підхід важливий у ситуаціях, коли система обробляє динамічні чи вузькоспеціалізовані дані, зокрема в науковій, медичній або освітній сферах.

2.1.1 Принцип роботи системи RAG

Процес функціонування технології RAG можна описати як послідовність етапів, які забезпечують ефективне поєднання пошуку релевантної інформації з

можливостями мовної генерації (рис. 2.1). Завдяки цьому система не лише відтворює текст, а й оперує реальними даними, що підвищує точність, надійність та зрозумілість отриманих результатів [12].

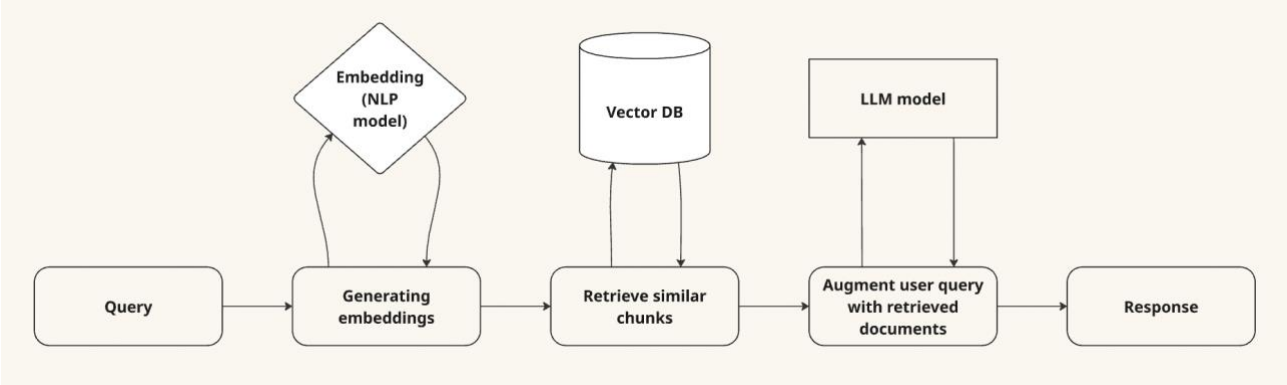


Рисунок 2.1 – Послідовність етапів обробки запиту у RAG-системі

Основні етапи роботи RAG включають:

– індексацію (ingestion): користувач завантажує документи різних форматів (PDF, DOCX, TXT тощо), які автоматично обробляються, поділяються на смислові фрагменти (chunks) та перетворюються на векторні подання за допомогою моделей типу all-mpnet-base-v2;

– збереження ембедингів (embeddings): отримані вектори записуються у спеціалізоване векторне сховище (vector store), що дозволяє виконувати швидкий пошук найбільш схожих фрагментів за семантичним змістом;

– пошук релевантних даних: під час запиту користувача його питання також кодується у вектор, після чого система виконує пошук найближчих векторів у базі за метрикою косинусної подібності (cosine similarity), щоб визначити найрелевантніші частини тексту;

– генерацію відповіді: знайдені фрагменти додаються до контексту запиту, після чого мовна модель формує узгоджену текстову відповідь, поєднуючи власні мовні можливості з наданими даними.

Отже, RAG реалізує дворівневу взаємодію: спочатку відбувається пошук відповідного контексту, а потім – генерація релевантної відповіді на його основі.

2.1.2 Переваги та недоліки

Технологія вирізняється тим, що поєднує сильні сторони обох підходів – retrieval-based (пошукового) та generative (генеративного). Завдяки цьому вона здатна забезпечити високий рівень точності, гнучкості та достовірності результатів, що робить її одним із найефективніших методів підвищення продуктивності сучасних LLM-систем.

Основні переваги:

– актуальність інформації: система може звертатися до зовнішніх баз знань, які регулярно оновлюються, завдяки чому відповіді моделі залишаються сучасними, незалежно від дати її останнього навчання;

– релевантність результатів: оскільки до відповіді додається retrieved context, користувач може бачити, на основі яких саме фрагментів документів була сформована відповідь, що підвищує рівень довіри до результату;

– ефективність використання ресурсів: технологія не потребує повторного навчання моделі при надходженні нових даних – достатньо просто оновити базу знань;

– гнучкість і персоналізація: кожен користувач або система може мати власну базу знань без необхідності змінювати архітектуру або перенавчати модель;

– підвищена точність: поєднання семантичного пошуку з мовною генерацією зменшує кількість «галюцинацій» – випадків, коли модель вигадує інформацію, не підтверджену джерелами.

Попри свої очевидні переваги, RAG має певні недоліки, які слід урахувати під час її реалізації та практичного використання. Ці обмеження здебільшого пов’язані з якістю векторних представлень, апаратними обмеженнями LLM і природою вхідних даних.

Основні недоліки:

– якість ембедингів: точність пошуку релевантних фрагментів безпосередньо залежить від обраної моделі для створення векторних

представлень тексту. Якщо embedding-модель неточно передає зміст тексту, пошукові результати можуть виявитися нерелевантними;

– обмеження контекстного вікна: мовні моделі мають фіксовану кількість токенів у запиті. Якщо система додає надто багато знайдених фрагментів, це може перевантажити контекст і знизити якість відповіді;

– затримка у виконанні запитів: процес пошуку, ранжування та генерації відповіді вимагає додаткового часу, що іноді робить роботу системи повільнішою порівняно зі звичайним викликом LLM без retrieval-етапу;

– несумісність стилів текстів: зібрані з різних джерел фрагменти можуть відрізнятися за стилем, структурою або термінологією, що ускладнює їх узгодження під час формування цілісної відповіді;

– залежність від якості бази знань: якщо у векторному сховищі містяться неточні, застарілі або суперечливі дані, модель неминуче відтворюватиме ці помилки у своїх відповідях.

Технологія RAG демонструє значний потенціал у підвищенні якості роботи великих мовних моделей, однак для досягнення максимальної ефективності потребує ретельного налаштування embedding-моделей, управління контекстом та контролю якості вихідних даних [13].

2.1.3 Приклади використання RAG у практичних системах

Підхід RAG поступово стає одним із ключових інструментів інтеграції штучного інтелекту в реальні робочі процеси. Гнучкість та здатність поєднувати мовну генерацію з перевіреними даними дозволяють створювати системи, які не лише генерують тексти, а й забезпечують достовірність, зрозумілість та контекстну релевантність результатів.

Технологія активно впроваджується у багатьох сферах діяльності, зокрема:

– наукові дослідження: для автоматичного узагальнення наукових статей, оглядів літератури та пошуку релевантних джерел у великих базах публікацій;

– корпоративні рішення: використовується у внутрішніх інформаційних чат-ботах, які відповідають на запитання співробітників, спираючись на корпоративні документи, політики та технічні інструкції;

– освітні системи: дає змогу створювати персоналізованих навчальних асистентів, які використовують матеріали певного курсу чи навчальної програми для формування точних і контекстних відповідей;

– медицина: застосовується для пошуку актуальних клінічних досліджень, діагностичних протоколів або наукових джерел, що допомагають лікарям або студентам медичних закладів приймати більш обґрунтовані рішення.

Концепція Retrieval-Augmented Generation є однією з найважливіших еволюційних ланок у розвитку сучасних інтелектуальних систем. Вона забезпечує зв’язок між великими мовними моделями та реальними базами знань, поєднуючи швидкість генерації з точністю відбору релевантних даних.

Таким чином, RAG є гібридною архітектурою, що відкриває нові можливості для побудови адаптивних, пояснювальних і персоналізованих інтелектуальних систем, які можуть ефективно використовуватись як у наукових дослідженнях, так і в реальних програмних рішеннях [14].

2.2 Визначення вимог до системи

Проєктування RAG-системи, потребує чіткого формулювання вимог, які визначають як функціональні можливості, так і нефункціональні характеристики майбутнього рішення. Визначення вимог є одним із ключових кроків, адже вони задають рамки для подальшого математичного обґрунтування, моделювання та експериментального дослідження вибраних методів. Коректно сформовані вимоги дозволяють узгодити архітектуру системи та гарантувати, що обрана технологія дійсно підвищить точність і стабільність роботи LLM системи.

У загальному випадку вимоги до системи можна розділити на дві великі категорії: функціональні, що описують конкретні дії, які має виконувати

система, та нефункціональні, що визначають якість виконання цих дій, обмеження на ресурси, продуктивність та надійність.

Функціональні вимоги:

– завантаження документів до бази знань: користувач повинен мати можливість завантажити файли для індексації. Документи розбиваються на текстові фрагменти (chunks), і кожен отримує унікальне векторне представлення;

– побудова векторного сховища: система повинна зберігати ембединги у векторному індексі для ефективного пошуку за семантичною схожістю;

– формування векторного представлення запиту: користувацький запит повинен бути перетворений у вектор таким самим способом, як і документи, що дозволяє порівнювати їх у спільному просторі ознак;

– пошук релевантного контексту: під час введення запиту користувачем система має порівнювати вектор запиту із векторами документів, формувати набір релевантних фрагментів і подавати їх на вхід мовній моделі;

– вибір найбільш релевантних документів: система повинна оцінювати семантичну близькість запиту та документів, використовуючи наприклад косинусну подібність і автоматично визначати топ-*k* релевантних документів;

– генерація відповіді: система повинна інтегрувати знайдені документи з оригінальним запитом та передавати цей контекст у генеративну модель. На основі цієї підказки LLM повинна формувати відповідь природною мовою;

– порівняльний аналіз: повинно бути передбачено можливість порівняння відповідей з використанням RAG і без нього. Результати супроводжуються аналітичними метриками – Cosine Similarity, Average Retrieved Similarity тощо;

– інтерфейс користувача: програмна реалізація повинна забезпечувати зручний інтерфейс для взаємодії з модулями системи – завантаження документів, введення запитів, перегляд отриманих результатів та аналітичних показників.

Нефункціональні вимоги:

– продуктивність: система повинна обробляти запит протягом часу, що не перевищує 10-20 секунд при середньому навантаженні, забезпечуючи оптимальний час обробки запиту;

– масштабованість: архітектура має підтримувати можливість розширення бази знань без істотного зниження швидкості. Розмір бази не повинен обмежуватись кількома сотнями документів;

– актуальність даних: база знань не повинна бути статичною. Інформація має регулярно оновлюватися, фільтруватися та при необхідності повторно кодуватися для підтримання високої точності пошуку;

– надійність та відмовостійкість: у випадку недоступності одного з компонентів, система повинна забезпечувати коректне завершення обчислень або попереджати про помилки без втрати даних;

– якість семантичного пошуку: векторні представлення мають забезпечувати достатню точність виявлення релевантних документів, щоб зменшити ризик некоректної генерації;

– безпека даних: якщо система працює з конфіденційними даними, система повинна забезпечити обмеження доступу, шифрування збережених векторів та захист від несанкціонованих втручань;

– usability (зручність і ефективність використання продукту): інтерфейс повинен бути інтуїтивно зрозумілим, забезпечуючи користувачу просту взаємодію з модулями системи без необхідності глибоких технічних знань.

Чітке визначення функціональних і нефункціональних вимог є необхідною умовою успішного моделювання та подальшої реалізації RAG-системи. Функціональні вимоги визначають очікувану поведінку системи та її основні можливості. Тоді як нефункціональні вимоги визначають критерії безпеки, продуктивності та надійності, які гарантують стабільну роботу системи [15].

2.3 Метод Dense Retrieval у RAG системах

Метод Dense Retrieval (щільне відтворення) є одним із ключових елементів сучасних RAG систем, які поєднують пошук релевантних знань із можливостями великих мовних моделей. Основна ідея цього підходу полягає у представленні як

документів, так і користувацьких запитів у вигляді щільних векторних уявлень, які дозволяють системі визначати семантичну подібність текстів, а не лише збіг ключових слів. Це забезпечує глибше розуміння запиту та підвищує точність отриманих результатів порівняно з традиційними пошуковими методами [16].

Далі розглянемо ключові етапи функціонування методу Dense Retrieval, які визначають його ефективність у процесі пошуку релевантної інформації в системах Retrieval-Augmented Generation.

Крок 1. Формування векторних уявлень документів.

Нехай маємо множину документів:

$$D = \{d_1, d_2, \dots, d_n\}, \tag{2.1}$$

де d_i – текстовий документ довільної довжини.

Кожен документ d_i проходить попередню обробку та перетворюється у щільне векторне представлення за допомогою спеціальної моделі-енкодера, яка здатна зчитувати зміст тексту та представляти його у вигляді числового вектору фіксованої розмірності. Для цього використовуються сучасні трансформерні моделі, зокрема такі як SentenceTransformers, які аналізують документ на рівні лексичних конструкцій, контексту та семантичних зв’язків між словами. У результаті кожен документ отримує власне векторне представлення, що відображає його зміст у багатовимірному семантичному просторі.

Метою першого етапу є відображення кожного документа у векторному просторі ознак розмірності R^m:

$$\int(d_i) \in R^m, \tag{2.2}$$

де $\int(d_i)$ – це функція-енкодер, реалізована за допомогою попередньо натренованої нейронної моделі;

m – розмірність простору ознак.

Таке відображення забезпечує збереження семантичних властивостей тексту: тексти зі схожим змістом отримують близькі вектори, навіть якщо не мають спільних слів. Це досягається завдяки навчанню енкодера на великих корпусах пар фраз або речень, які мають однаковий сенс у різних контекстах.

Отримані вектори зберігаються у спеціалізованому векторному індексі:

$$V = [\int(d_1), \int(d_2), \dots, \int(d_n)]. \quad (2.3)$$

Отримані векторні представлення зберігаються у спеціалізованому векторному сховищі (наприклад, FAISS), яке дає змогу виконувати швидкий пошук найбільш близьких за значенням векторів. Таким чином корпус текстів проєктується у багатовимірний простір ознак, де семантично подібні фрагменти розміщуються на невеликій відстані один від одного.

Крок 2. Формування векторного представлення запиту.

На другому етапі користувацький запит q також перетворюється у векторне представлення за допомогою тієї ж функції енкодера. Використання спільного простору ознак для документів і запитів є принципово важливим, адже це дозволяє безпосередньо порівнювати їх семантичні уявлення.

Завдяки цьому, навіть якщо користувач формулює запит у довільній формі (наприклад, «Як працює механізм семантичного пошуку?»), система може знайти документи, що описують той самий зміст іншими словами («Dense retrieval у нейронних пошукових системах»).

Інтуїтивно, енкодер тут виконує роль «перекладача» природної мови у математичний простір, де близькі за змістом фрази мають близьке розташування.

Крок 3. Обчислення міри подібності.

Для визначення релевантності документа до запиту використовується косинусна подібність між двома векторами:

$$sim(q, d_i) = \frac{q \times d_i}{\|q\| \times \|d_i\|}. \quad (2.4)$$

Значення міри подібності належить інтервалу $[-1, 1]$, чим ближче воно до 1, тим більш релевантний документ d_i відносно запиту q:

– якщо sim дорівнює 1 – тексти мають повну семантичну відповідність;

– якщо sim дорівнює 0 – між текстами відсутній смисловий зв'язок;

– якщо sim дорівнює −1 – тексти мають протилежний або суперечливий зміст.

Оскільки тексти з протилежним змістом у семантичному пошуку зустрічаються рідко, на практиці інтервал зводиться до $[0, 1]$, де значення вище 0.7-0.8 зазвичай вважаються релевантними.

Крок 4. Вибір найбільш релевантних документів:

Після обчислення подібності для всіх документів у базі визначається підмножина топ-k найближчих елементів:

$$D_{topk} = \underset{d_i \in D}{argmax_k}\, \mathrm{sim}(q, d_i). \tag{2.5}$$

Ці документи є найбільш релевантними до запиту користувача й утворюють набір контекстів, що будуть передані у генеративну модель. Значення параметра k вибирається емпірично – зазвичай у межах 3-10.

Наприклад, для інформаційних систем або чатботів із великими базами знань доцільно вибирати k = 5, щоб уникнути надмірного навантаження на LLM, але при цьому не втратити важливий контекст.

Крок 5. Інтеграція з LLM.

Після вибору найбільш релевантних документів, система RAG формує розширений контекст із запитом користувача до LLM, що дозволяє моделі генерувати релевантну відповідь на основі власних і зовнішніх даних.

Цей процес можна подати у вигляді функціональної залежності:

$$y = LLM\big(q, D_{topk}\big), \tag{2.6}$$

де y – фінальна відповідь, сформована на основі поєднання внутрішніх знань моделі та знайдених фрагментів;

LLM – це велика мовна модель, що приймає на вхід як запит користувача, так і додаткові документи, і повертає згенеровану відповідь.

Для забезпечення стабільної роботи методу Dense Retrieval важливо враховувати низку технічних моментів:

– нормалізація векторів: перед обчисленням косинусної подібності всі вектори мають бути нормалізовані до одиничної довжини, щоб уникнути викривлень у результатах;

– вибір embedding-моделі: використання моделей із високою якістю ембедингів (наприклад, sentence-transformers/all-mpnet-base-v2) суттєво впливає на точність пошуку. При цьому варто враховувати співвідношення між якістю та обчислювальними витратами;

– оптимізація параметрів пошуку: значення параметрів top-k і similarity threshold підбираються експериментально: занадто низькі значення можуть призвести до втрати контексту, а надто високі – до зниження продуктивності;

– управління базою знань: система має забезпечувати регулярне оновлення сховища документів, щоб підтримувати актуальний набір даних.

Метод Dense Retrieval є фундаментальною складовою сучасних RAG-архітектур. Завдяки використанню щільних векторних уявлень і потужних нейронних енкодерів, він дозволяє реалізувати семантичний пошук високої точності, масштабований для роботи з великими масивами даних.

У поєднанні з LLM цей підхід забезпечує створення гібридних систем, здатних не лише запам’ятовувати інформацію, а й осмислено її інтерпретувати. З математичної точки зору, Dense Retrieval поєднує властивості метричного простору (для пошуку подібності) і статистичного навчання (для формування оптимальних представлень) [17].

Отже, застосування Dense Retrieval у RAG-системах дозволяє досягти високої точності відповідей, зменшити кількість помилкових та вигаданих тверджень та підвищити загальну ефективність взаємодії людини з LLM.

2.4 Вибір стеку технологій

Вибір технологічного стеку є ключовим етапом у процесі створення інтелектуальної системи на основі Retrieval-Augmented Generation. Від правильності обраних інструментів залежить стабільність роботи, продуктивність, гнучкість інтеграцій, а також можливість подальшого масштабування рішення.

Під час розробки системи було враховано як технічні особливості компонентів RAG-архітектури (retriever, vector store, generator), так і практичні аспекти розробки – зручність реалізації API, інтеграцію з мовними моделями, адаптивність інтерфейсу та простоту розгортання у середовищах з обмеженими ресурсами.

Для досягнення цих цілей було обрано стек технологій, що поєднує надійність серверних рішень на Python (FastAPI) з гнучкістю сучасної фронтенд бібліотеки React, а також використання TailwindCSS для швидкої побудови адаптивного інтерфейсу. У частині штучного інтелекту передбачено використання моделей GPT-4o-mini як генеративного ядра, SentenceTransformers для створення векторних подань, також FAISS для зберігання та швидкого пошуку ембедингів. А для зберігання вхідних файлів передбачено локальне збереження для текстового режиму.

Таке поєднання забезпечує повний цикл роботи RAG-підходу – від обробки даних на сервері до інтерактивної взаємодії користувача через веб-інтерфейс.. Нижче подано детальний опис кожного компонента стеку, його ролі у системі, переваг і обґрунтування вибору [18].

2.4.1 Python

В рамках розробки RAG-системи було прийнято рішення використовувати мову програмування Python, оскільки вона є галузевим стандартом у сфері

обробки природної мови (NLP), машинного навчання (ML) та штучного інтелекту (AI). Python відзначається високою гнучкістю, простотою синтаксису, а також широким вибором спеціалізованих бібліотек, що дає змогу створювати ефективні рішення для наукових і дослідницьких проєктів.

Однією з ключових переваг Python є його екосистема для машинного навчання. Завдяки таким бібліотекам, як transformers, sentence-transformers, scikit-learn, nltk, pandas, numpy, torch, faiss, розробник може реалізувати весь життєвий цикл RAG-системи – від обробки тексту до побудови векторних уявлень і пошуку семантично близьких документів. Саме ця властивість робить Python універсальним середовищем для розробки інтелектуальних застосунків.

Python також забезпечує легку інтеграцію з API великих мовних моделей, зокрема OpenAI, що використовується у проєкті для генерації відповідей. Додатковою перевагою є активна спільнота, яка постійно оновлює пакети та надає технічну підтримку, що суттєво знижує ризики під час розробки.

Важливим аргументом стала і читабельність коду, що дозволяє швидко реалізовувати експериментальні зміни у процесі дослідження. Python ідеально підходить для побудови прототипів, проведення наукових експериментів та швидкого тестування гіпотез. Це було особливо важливо в контексті роботи, де необхідно виконати порівняльний аналіз систем із RAG і без RAG. Таким чином, вибір Python забезпечив баланс між швидкістю розробки, продуктивністю й аналітичною гнучкістю [19].

2.4.2 FastAPI

Для реалізації серверної частини системи було обрано FastAPI – сучасний високопродуктивний вебфреймворк, який поєднує простоту використання з асинхронною обробкою запитів. Його архітектура побудована на стандарті ASGI, що дає змогу обробляти велику кількість запитів одночасно без значних затримок, що є критично важливим для інтерактивних систем типу RAG.

Ключові переваги FastAPI фреймворку:

– автоматичне створення документації API відповідно до стандарту OpenAPI (Swagger);

– інтеграцію з бібліотекою Pydantic, яка забезпечує строгий контроль типів та валідацію даних;

– підтримку сучасних протоколів JSON Schema, що спрощує комунікацію між frontend та backend частинами;

– простоту реалізації RESTful API, що дозволяє легко поєднувати компоненти системи.

Основним завданням вебфреймворку у межах проєкту є забезпечення взаємодії між клієнтом і сервером, зокрема – приймання користувацьких запитів, обробка їх у backend, передача до мовної моделі, отримання відповіді та відправка результату назад користувачу. FastAPI реалізує це через чітку маршрутизацію запитів і підтримку асинхронних функцій, що підвищує швидкодію системи.

Порівняно з традиційними фреймворками, такими як Flask або Django, FastAPI є більш сучасним рішенням, оптимізованим для інтеграції з AI/ML сервісами. Його продуктивність наближається до таких низькорівневих інструментів, як Node.js або Go, але при цьому він зберігає простоту розробки Python [20].

Таким чином, вибір FastAPI як серверного фреймворку є обґрунтованим рішенням, яке поєднує ефективність, надійність і гнучкість, необхідні для системи, що виконує складну обробку текстових даних у режимі реального часу.

2.4.3 FAISS

Одним із центральних елементів архітектури системи є векторна база знань FAISS. Вона забезпечує ефективне зберігання та пошук семантично подібних векторів, які представляють зміст документів у багатовимірному просторі. У

системах типу RAG цей компонент відіграє ключову роль, оскільки саме він дозволяє знаходити релевантну інформацію для формування контексту, що подається у LLM.

FAISS оптимізована для роботи з великими наборами даних, вона дозволяє виконувати пошук найближчих сусідів (Nearest Neighbor Search) у векторному просторі, використовуючи високоефективні алгоритми, зокрема Product Quantization (PQ) і Hierarchical Navigable Small World (HNSW). Формально задача пошуку зводиться до визначення найбільш релевантних векторів:

Однією з переваг FAISS є висока продуктивність навіть при роботі з мільйонами векторів. Бібліотека підтримує як CPU-прискорення, так і GPU-прискорення, що забезпечує масштабованість системи без втрати точності.

У межах проєкту FAISS використовується для побудови локального індексу, який зберігає вектори, сформовані моделлю ембедингів (sentence-transformers/all-mpnet-base-v2). Це дає змогу обчислювати подібність між запитом користувача та базою знань у реальному часі.

Загалом вибір FAISS як основи для векторного сховища зумовлений її швидкодією, стабільністю, відкритим кодом і активною підтримкою спільноти. Завдяки цьому система може працювати як у локальному середовищі, так і бути масштабованою до хмарних інфраструктур (наприклад, AWS).

2.4.4 React

Вибір технології для побудови клієнтської частини є одним із ключових етапів проєктування вебзастосунку, оскільки саме від нього залежить гнучкість інтерфейсу, продуктивність взаємодії з бекендом та можливість масштабування в майбутньому. У межах розробки системи було обрано React, який фактично є стандартом для створення сучасних вебзастосунків.

React дозволяє будувати інтерфейс на основі компонентного підходу, де кожен елемент сторінки представляє окрему логічну одиницю з власним станом

і поведінкою. Така структура спрощує розробку, тестування та повторне використання компонентів, а також робить загальну архітектуру інтерфейсу більш передбачуваною та масштабованою.

React також підтримує віртуальний DOM, що забезпечує швидке оновлення елементів інтерфейсу без повного перерендерингу сторінки. У контексті системи RAG, де користувач постійно взаємодіє з інтерфейсом (введення запитів, перегляд результатів, аналіз метрик), це дозволяє зберегти плавність роботи інтерфейсу навіть під навантаженням.

Ще однією перевагою є велика кількість допоміжних бібліотек та екосистемних рішень – від управління станом (Redux Toolkit) до роутінгу та інтеграції зі сторонніми API. Завдяки цьому React легко адаптується під складні функціональні вимоги, такі як обробка файлів, історія чатів, модальні вікна, попередній перегляд документів або візуалізація метрик.

З практичної точки зору React є оптимальним вибором і для подальшого розвитку системи, оскільки він добре масштабуються. Його логіка сумісна з компонентами, написаними в сучасному синтаксисі TypeScript, що спрощує спільну роботу та забезпечує довгострокову життєздатність проєкту [21].

2.4.5 TypeScript

Також важливою складовою клієнтської частини є використання TypeScript – мови програмування, яка розширює JavaScript статичною типізацією. Це рішення було прийнято з урахуванням того, що проєкт містить велику кількість логіки пов’язаної з асинхронними запитами, роботою зі структурами даних, інтерфейсами API, файлами та системними сутностями.

Основною перевагою TypeScript є забезпечення контролю типів на етапі розробки. Це зменшує ризик помилок, які можуть проявлятись лише під час виконання програми, що особливо актуально для інтерактивних дослідницьких

систем. Типізація дозволяє описувати структури даних із високою точністю – формат відповіді від серверу або структуру метрик порівняння RAG/No-RAG.

Крім цього, TypeScript значно полегшує підтримку коду. Завдяки автодоповненню, підказкам типів, перевіркам сумісності та інтеграції з сучасними IDE, розробник отримує чіткий контроль над логікою застосунку, що особливо корисно, коли проєкт постійно розширюється.

Не менш важливо і те, що TypeScript покращує масштабованість проєкту. Коли додаються нові модулі – наприклад, сторінки з метриками, таблиці порівнянь, інструменти аналізу або нові типи даних – наявність чітко визначених типів дозволяє легко інтегрувати їх у існуючий код. Це робить TypeScript не просто доповненням, а необхідною частиною архітектури фронтенда.

Таким чином, TypeScript забезпечує структурованість і надійність, що робить його оптимальним вибором для розробки клієнтської частини, де важливо мінімізувати кількість помилок і підвищити якість коду.

2.4.6 Tailwind CSS

Розробка інтерфейсу системи потребувала вибору інструментів для стилізації, які забезпечують одночасно швидкість розробки та гнучкість. У проєкті використано Tailwind CSS, який дозволяє створювати чистий, модульний та адаптивний інтерфейс без надмірного перевантаження стилями.

Tailwind CSS – це утилітарний CSS-фреймворк, який дозволяє формувати дизайн на основі класів, що відповідають конкретним стилям. Це значно прискорює створення інтерфейсу, оскільки розробнику не потрібно підтримувати складні каскади або великі CSS-файли.

Переваги використання Tailwind CSS:

– утилітарний підхід до стилізації, який дозволяє формувати інтерфейс без написання кастомних CSS-файлів. Кожен клас відповідає за одну властивість, що робить стилізацію передбачуваною та контрольованою;

– висока швидкість розробки завдяки готовим наборам класів для відступів, кольорів, тіней, анімацій та адаптивності. Це значно скорочує час на створення інтерфейсу;

– мінімізація конфліктів стилів, оскільки всі стилі визначаються безпосередньо в JSX-компонентах, а не в глобальних CSS-файлах, де може виникати непередбачувана взаємодія селекторів;

– висока кастомізованість, оскільки Tailwind легко конфігурується через tailwind.config.js, де можна визначати власну кольорову палітру, шрифти, відступи тощо;

– оптимізація кінце вого розміру CSS, адже під час збирання застосунку Tailwind автоматично видаляє всі не використані класи, що суттєво зменшує вагу стилів у продакшні;

– сумісність із сучасними фреймворками і бібліотеками, такими як React, Next.js, Vue, Svelte, що робить Tailwind універсальним і зручним для інтеграції в будь-який тип фронтенд-проєкту.

Таким чином, Tailwind CSS є оптимальним рішенням для створення сучасного, лаконічного та функціонального інтерфейсу, що відповідає потребам створення RAG-системи та забезпечує високу швидкість роботи.

2.5 Розробка загальної архітектури системи

Проєктування архітектури системи є одним із ключових етапів розробки інструментів, що використовують підхід Retrieval-Augmented Generation. У запропонованому рішенні архітектура побудована за класичною схемою клієнт-сервер, що забезпечує чітке розмежування ролей між компонентами, підвищує масштабованість, дає можливість легко модернізувати систему та інтегрувати нові програмні модулі. Логіка взаємодії між фронтендом, серверною частиною та зовнішніми NLP-сервісами реалізована за принципом REST API, що робить систему зрозумілою, передбачуваною та технічно стандартизованою [22].

2.5.1 Архітектурний стиль системи та механізм взаємодії компонентів

Система побудована на основі класичної клієнт-серверної архітектури. Це модель побудови програмних систем, у якій логіка роботи розподіляється між двома частинами: клієнтом, що відповідає за взаємодію з користувачем, та сервером, який виконує основні обчислення й керує даними. Для забезпечення взаємодії між цими компонентами використовується мережеве підключення та стандартизовані протоколи комунікації. У межах архітектури кожен компонент виконує власну чітко визначену роль, що можна описати як:

– клієнт: це застосунок або інтерфейс, через який користувач надсилає запити до системи. Він не зберігає дані й не виконує складні операції. Його основна роль полягає у формуванні запитів, передачі їх на сервер та відображенні отриманих результатів;

– сервер: це центральний компонент, який приймає та обробляє запити. Він може виконувати обробку текстів, пошук у базах даних, взаємодію з моделями штучного інтелекту, виконання бізнес-логіки тощо. Після обробки сервер формує структуровану відповідь і надсилає її клієнту.

Одним із найбільш ефективних і поширених підходів для обміну даних між клієнтською та серверною частинами є використання REST API. Цей архітектурний стиль взаємодії забезпечує стандартизований спосіб обміну інформацією, дозволяючи клієнту надсилати запити у визначеному форматі та отримувати структуровані відповіді від сервера. У межах RAG-системи REST API виступає ключовим елементом, який пов’язує інтерфейс кінцевого користувача з усіма внутрішніми компонентами – модулем обробки документів, індексацією, генерацією тексту та зовнішніми LLM-сервісами [23].

Основна ідея REST полягає у використанні HTTP як транспортного протоколу та передачі даних через чітко визначені маршрути (URL-ендпоінти):

– /chat: приймає текстові запити від користувача та генерує відповіді за допомогою LLM разом із релевантним контекстом;

– /ingest: забезпечує обробку, розбиття та індексацію завантажених документів для подальшого семантичного пошуку;

– /files: надає можливість отримувати список доступних документів або видаляти непотрібні файли;

– /health: використовується для діагностики роботи серверу, дозволяючи швидко перевірити стан бекенду.

У розробленій системі клієнтська частина надсилає HTTP-запити до відповідних ендпоінтів FastAPI-сервера, який приймає, перевіряє та обробляє дані. Такий механізм забезпечує чітку структурованість запитів, а серверна логіка отримує змогу незалежно від клієнта масштабуватись, оновлюватись чи змінювати внутрішню реалізацію без потреби втручання у фронтенд.

Для обміну інформацією використовується легкий текстовий формат JSON, що дозволяє передавати складні структури у вигляді вкладених об’єктів. Це особливо важливо для RAG-системи, де відповідь містить не лише текстовий результат, але й список релевантних документів, метадані пошуку та іншу службову інформацію.

REST API формує зручний спосіб доступу до усіх внутрішніх компонентів системи. Наприклад, при надсиланні запиту на обробку документа клієнт передає файл, сервер виконує конвертацію у текст, створює векторне представлення та додає його до FAISS-індексу. Зовні це виглядає як проста HTTP-операція, але всередині відбувається складний ланцюг обчислень. Такий підхід знімає навантаження з клієнта та дозволяє зберігати логіку обробки в одному центрі – на сервері.

Концептуальна схема (рис. 2.2), відображає загальний механізм роботи системи та взаємодію між клієнтською та серверною частинами під час обробки документів і виконання користувацьких запитів. Схема демонструє послідовний рух даних через ключові функціональні компоненти RAG-системи та дозволяє простежити за всім циклом – від моменту, коли користувач надсилає запит, до формування LLM відповіді [24].

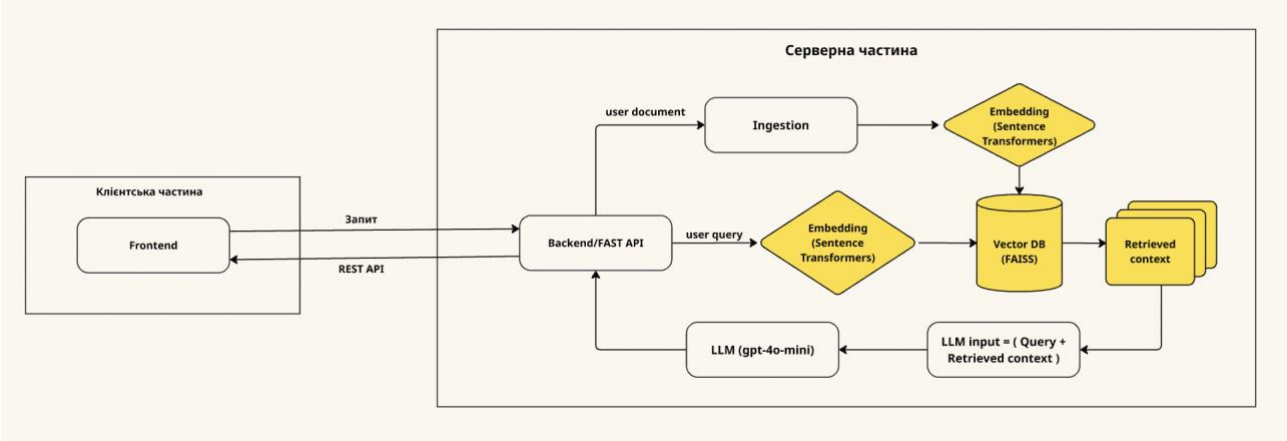


Рисунок 2.2 – Концептуальна схема RAG-системи

2.5.2 Компоненти обробки NLP у системі

Ядром функціональності RAG-системи є модулі обробки природної мови які забезпечують перетворення документів у зручний для подальшої роботи формат, створення векторних представлень тексту, виконання семантичного пошуку та формування відповіді від LLM.

Першим етапом роботи NLP-компонентів є попередня обробка документів, яку реалізовано в модулях loader.py, splitter.py та utils.py. Їхнє завдання полягає у приведенні вхідних даних до єдиного стандартизованого формату, придатного для подальшої векторизації. Система підтримує завантаження поширених форматів – PDF, DOCX, TXT. Функціонал модулів виконує витяг тексту з документів, очищення від службових символів, нормалізацію, а також автоматичне розбиття тексту на смислові фрагменти. Створення таких фрагментів (chunks) є важливим елементом архітектури RAG, оскільки дозволяє підвищити точність і релевантність пошуку, працюючи не з усім документом одразу, а з невеликими логічними частинами.

Другим ключовим етапом є побудова векторних представлень тексту. У системі використано модель sentence-transformers/all-mpnet-base-v2, яка забезпечує високу швидкість роботи та достатньо якісні embeddings для семантичного пошуку. Відповідна логіка реалізована у файлі embeddings.py, де

модель ініціалізується, завантажується в пам’ять та використовується для перетворення фрагментів тексту у багатовимірні вектори. Створений embedding відображає семантичний зміст тексту в просторі ознак, де подібні за змістом елементи розташовані близько один до одного.

Після векторизації текстові фрагменти передаються у векторне сховище, реалізоване на основі бібліотеки FAISS у модулі vector_store.py. Це сховище відповідає за створення індексу найближчих сусідів і виконання швидкого пошуку за семантичною подібністю. У межах проєкту індекс зберігається у файлах index.faiss та meta.jsonl, що спрощує подальше завантаження даних без необхідності повторної індексації.

Завершальним етапом є обробка запиту користувача, бекенд генерує embedding запиту тим самим методом та виконує пошук у FAISS-індексі. У результаті формується набір релевантних фрагментів – Retrieved Context, який додається до LLM prompt. У модулі llm.py реалізовано логіку поєднання запиту та релевантного контексту, після чого ці дані передаються до LLM моделі. Мовна модель, маючи доступ до релевантних фрагментів, формує відповідь. Такий підхід значно знижує ймовірність галюцинацій моделі та підвищує точність відповідей [25].

2.5.3 Використання LLM моделі

У межах розроблюваної RAG-системи одним із ключових компонентів є модуль генерації відповідей, який відповідає за формування логічних результатів на основі користувацького запиту та знайденого семантичного контексту. Для цього використано сучасну LLM модель GPT-4o-mini, доступ до якої здійснюється через OpenAI API.

Доступ до моделі реалізовано з використанням офіційної бібліотеки OpenAI. Налаштування ключа API зберігаються у файлі конфігурації .env, що забезпечує безпеку даних і відповідає стандартам розробки серверних

застосунків. Завантаження ключа відбувається у файлі settings.py, де через модуль python-dotenv змінні середовища підтягуються до конфігурації застосунку. Такий підхід дозволяє зберігати секретні дані окремо від вихідного коду та гарантує можливість гнучкого налаштування оточення.

Безпосередньо логіка виклику моделі реалізована в модулі llm.py. У цьому файлі формуються параметри запиту до OpenAI API, створюється prompt, який містить як текст користувача, так і релевантні фрагменти документа. Запит передається у модель у вигляді структурованої JSON-структури, а відповідь повертається у форматі, придатному для подальшої обробки або відображення на клієнтському інтерфейсі.

До ключових переваг GPT-4o-mini можна віднести:

– високу швидкість роботи, що дозволяє досягати низької латентності при інтерактивних сценаріях;

– доступну вартість, яка робить її вигідною для використання у проєктах, що потребують великої кількості викликів API;

– достатню точність при роботі зі структурованими prompt та контекстами, наданими через механізм Dense Retrieval;

– стійкість до коротких контекстів, що важливо для невеликих chunks, отриманих під час індексації документів.

На момент розробки GPT-4o-mini має одну з найнижчих ставок на ринку – приблизно $0.15 за 1 млн токенів вхідних даних та $0.60 за 1 млн токенів вихідних даних.

Тому застосування GPT-4o-mini у RAG-системі забезпечує оптимальний баланс між швидкістю, якістю, вартістю та можливістю обробки складних інформаційних запитів [26].

3 ДОСЛІДЖЕННЯ МЕТОДУ ПОКРАЩЕННЯ РОБОТИ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ RAG

3.1 Обґрунтування вибору середовища програмної реалізації

У рамках кваліфікаційної роботи, що реалізує метод покращення роботи LLM із використанням технології RAG, важливим етапом стало визначення оптимального середовища програмування. Вибір інструментів безпосередньо впливає не лише на зручність процесу розробки, але й на стабільність, масштабованість та відтворюваність результатів. Зважаючи на це, було обрано текстовий редактор Visual Studio Code (VS Code), який на сьогодні є одним із найпопулярніших інструментів серед розробників застосунків на Python, JavaScript, TypeScript та інших сучасних мовах програмування.

VS Code – це кросплатформний редактор коду, створений із акцентом на гнучкість, розширюваність та можливість інтеграції зі сторонніми інструментами. На відміну від традиційних IDE, цей редактор побудований на модульному підході, що дозволяє користувачу самостійно визначати необхідний набір функцій і формувати власне робоче середовище. Такий підхід забезпечує високу універсальність, адже VS Code може адаптуватися до різних мов програмування, типів проєктів та стилів роботи розробника.

Модульність редактора дає змогу легко поєднувати його з різноманітними інструментами – від засобів написання та форматування коду до систем керування версіями та інструментів тестування. Завдяки цьому середовище зручне для роботи як із простими застосунками, так і з великими програмними проєктами, в яких може використовуватися кілька технологій одночасно [27].

Нижче наведено ключові переваги VS Code:

– кросплатформеність: редактор доступний для Windows, macOS і Linux, що дозволяє працювати на будь-якій операційній системі без втрати функціональності;

– велика екосистема розширень. у вбудованому marketplace доступні тисячі плагінів для різних мов програмування, аналізаторів коду, інструментів форматування, підтримки фреймворків та навчальних моделей;

– підтримка різних мов програмування: через розширення можна працювати практично з будь-якою сучасною мовою – від Python і C++ до Go, Java, TypeScript чи Rust;

– вбудована підтримка Git: користувач може зручно переглядати зміни, створювати коміти, працювати з гілками та репозиторіями, не переходячи до зовнішніх інструментів;

– гнучке налаштування інтерфейсу: можна змінювати теми оформлення, розкладку панелей, гарячі клавіші та інші параметри під власні потреби;

– висока швидкість роботи та низькі системні вимоги: навіть за наявності багатьох додаткових можливостей VS Code залишається легким, швидким і стабільним, забезпечуючи комфортні умови для розробки;

– активна спільнота та регулярні оновлення: VS Code постійно вдосконалюється, отримує нові функції та підтримку від великої спільноти користувачів.

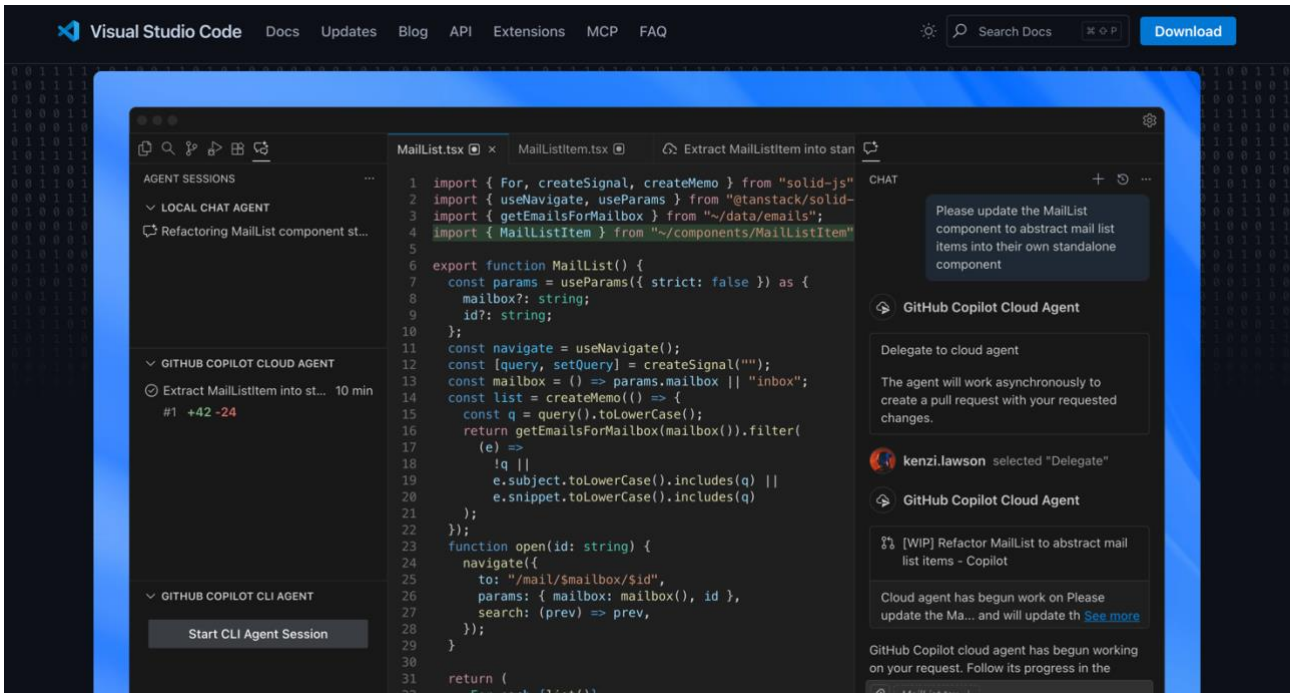


Рисунок 3.1 – Інтерфейс VS Code з офіційного сайту

Таким чином, вибір середовища програмної реалізації є обґрунтованим як з технічної, так і з практичної точки зору. VS Code забезпечує гнучкість при побудові повноцінної RAG-системи, а також завдяки широкому набору інструментів дозволяє ефективно реалізувати всі етапи розробки, від обробки документів до формування релевантних відповідей від LLM [28].

3.2 Програмна реалізація

У рамках кваліфікаційної роботи був створений вебзастосунок RAG-система. Під час розробки було сформовано повноцінну інфраструктуру застосунку, яка охоплює серверну частину, відповідальну за обробку запитів, взаємодію з моделлю та індексами, а також клієнтську частину, що забезпечує зручний інтерфейс для роботи користувача. Для реалізації було обрано такі інструменти: Python, FastAPI, Sentence Transformers, FAISS, TypeScript, React та інші. Розділення системи на два основні компоненти – серверний та клієнтський, забезпечило логічну структурованість, можливість масштабування та незалежну розробку окремих частин [29].

3.2.1 Створення серверної частини

Серверна частина реалізована з використанням фреймворку FastAPI, який поєднує високу продуктивність, підтримку асинхронних викликів та зручний підхід до проєктування API. Основною точкою входу до бекенд-системи є файл app.py, через який відбувається ініціалізація застосунку, конфігурація середовища, підключення маршрутизаторів та підготовка всіх необхідних ресурсів.

Розглянемо основний програмний модуль серверної частини – файл app.py. Код цього модуля подано у лістингу 3.1.

Лістинг 3.1 Реалізація точки входу серверної частини:

```
async def lifespan(app: FastAPI):
    settings.files_dir.mkdir(parents=True, exist_ok=True)
    settings.faiss_index_path.parent.mkdir(parents=True, exist_ok=True)
    get_vector_store()
    yield

app = FastAPI(title="LLM Chat with RAG vs No-RAG", version="1.0.0", lifespan=lifespan)

app.add_middleware(CORSMiddleware, allow_origins=["*"], allow_headers=["*"], allow_credentials=True, allow_methods=["*"])

@app.get("/")
async def root() -> dict[str, str]:
    return {"message": "LLM Chat with RAG vs No-RAG backend is running"}

app.include_router(health.router)
app.include_router(ingest.router)
app.include_router(files.router)
app.include_router(chat.router)
```

Даний модуль точки входу здійснюється за наступними кроками:

Крок 1. Оголошення асинхронного менеджера життєвого циклу lifespan визначає логіку, яка виконується перед запуском сервера та після завершення його роботи. При запуску застосунок:

– створює службові директорії для зберігання завантажених файлів та FAISS-індексу;

– викликає функцію get_vector_store(), яка завантажує векторне сховище.

Крок 2. Створення екземпляра FastAPI. На цьому етапі формується основний застосунок, до якого прив’язується заголовок, версія сервісу та функція lifespan, що керує стартом та завершенням роботи. Це центральний елемент серверної частини, навколо якого організовано всю взаємодію.

Крок 3. Налаштування CORS-політики. Додається middleware, який дозволяє клієнтській частині відправляти запити до бекенда. Під час локальної розробки дозволені запити з будь-якого джерела *(allow_origins=["*"])*, що спрощує тестування.

Крок 4. Реалізація базового маршруту кореня *@app.get("/")*. Цей маршрут повертає просте повідомлення, що сервер працює. Використовується для перевірки роботи вебзастосунку.

Крок 5. Підключення маршрутизаторів. Наприкінці файлу підключаються всі логічні частини API, де вони збираються в єдину систему.

Далі буде послідовно розглянуто ключові етапи програмної реалізації технології RAG, які визначають її структуру та логіку роботи.

Одним із ключових етапів підготовки документів до подальшої індексації є їх розбиття на менші структурні частини – чанки. Саме на цих частинах надалі будуються ембединги, які забезпечують можливість точного семантичного пошуку. У проєкті за це відповідає модуль *services/splitter.py* [30]. Код основної функції подано у лістингу 3.2.

Лістинг 3.2 Реалізація функції *split_text* для розбиття тексту на чанки:

```
def split_text(text: str, *, chunk_size: int, chunk_overlap: int, base_metadata: Dict[str, str], page_texts: Iterable[str] | None = None) -> List[Chunk]:
    if chunk_size <= 0:
        raise ValueError("chunk_size must be positive")
    if chunk_overlap < 0 or chunk_overlap >= chunk_size:
        raise ValueError("chunk_overlap must be >=0 and < chunk_size")

    chunks: List[Chunk] = []
```

```
start = 0
text_length = len(text)
step = chunk_size - chunk_overlap
chunk_index = 0

while start < text_length:
    end = min(start + chunk_size, text_length)
    chunk_text = text[start:end]
    metadata = dict(base_metadata)
    metadata.update({"chunk_id": chunk_index, "start": start, "end": end})
    chunks.append(Chunk(text=chunk_text, metadata=metadata))
    start += step
    chunk_index += 1

if page_texts:
    for chunk in chunks:
        chunk.metadata.setdefault("page", _infer_page(chunk, page_texts))
return chunks
```

Даний модуль розбиття тексту на чанки здійснюється за наступними кроками:

Крок 1. Перед виконанням розбиття система перевіряє коректність вхідних параметрів. Це необхідно, щоб гарантувати, що розмір майбутніх фрагментів та величина їх перекриття є логічними та не призведуть до некоректної роботи.

Крок 2. Далі готується службова інформація, яка потрібна для роботи. Визначається загальна довжина тексту, початкова позиція, розраховується величина, на яку буде переміщуватись межа між фрагментами, а також створюється порожній список, для додавання всіх сформованих фрагментів.

Крок 3. Далі текст послідовно розбивається на частини однакового розміру. Кожна така частина містить фрагмент початкового документа та

супровідні дані – номер фрагмента і його позицію у межах всього тексту. Сформований фрагмент додається до загального списку.

Крок 4. Після створення чергового фрагмента інтервал зміщується вперед. Однак зміщується не на повну довжину фрагмента, а лише частково. Це забезпечує перекриття між сусідніми фрагментами. Це потрібно для того, щоб контекст між двома частинами не втрачався, Завдяки цьому семантична цілісність тексту зберігається, а якість подальших ембедінгів підвищується.

Крок 5. Для документів зі сторінками додатково визначається, на якій сторінці він розташований. Це робиться шляхом порівняння позиції фрагмента з межами сторінок.

Крок 6. Після обробки тексту алгоритм повертає набір сформованих фрагментів. Вони містить готовий текст, а також структуровані метадані, які надалі використовуються під час ембедінгу, пошуку та формування RAG-контексту.

Наступним важливим етапом є формування векторних представлень тексту, оскільки саме такі вектори використовуються для подальшого порівняння змісту та пошуку релевантних фрагментів у FAISS-індексі. У проєкті за це відповідає модуль *services/embeddings.py* [31]. Код цього модуля подано у лістингу 3.3.

Лістинг 3.3 Реалізація модуля генерації ембедінгів:

```
@lru_cache(maxsize=1)
def _get_model() -> SentenceTransformer:
    return SentenceTransformer(settings.embedding_model)

def embed_texts(texts: Iterable[str]) -> np.ndarray:
    model = _get_model()
    embeddings = model.encode(list(texts), batch_size=16, show_progress_bar=False)
    return np.array(embeddings, dtype="float32")
```

```
def embed_query(text: str) -> np.ndarray:
    return embed_texts([text])[0]
```

Модуль генерації ембедингів здійснюється за наступними кроками:

Крок 1. Під час першого звернення система завантажує модель SentenceTransformer, яка використовується для створення числових представлень текстових даних. Завантаження виконується один раз і кешується, щоб не перевантажувати сервер під час обробки подальших запитів.

Крок 2. Після завантаження моделі виконується саме перетворення. Текст подається на вхід нейромережі, яка формує компактний вектор із фіксованою кількістю вимірів. Такий вектор відображає семантичний зміст фрагмента, що дає змогу ефективно порівнювати тексти між собою.

Крок 3. Згенеровані ембедінги збираються у масив та передаються наступним етапам – модулю індексації та пошуку. Вектори мають формат, сумісний з FAISS-індексом, що забезпечує швидкий семантичний пошук [32].

Після того як текстові фрагменти документа перетворено в ембединги, вони мають бути збережені у векторній базі даних, для швидкого семантичного пошуку. За це відповідає модуль *services/vector_store.py*. Код цього модуля подано у лістингу 3.4.

Лістинг 3.4 Додавання ембедингів у векторне сховище:

```
class VectorStore:
    def load(self) -> None:
    # ...
    def _ensure_index(self, dimension: int) -> faiss.Index:
    # ...
    def add(self, vectors: np.ndarray, metadatas: Iterable[Dict[str, object]]) ->
None:
    # ...
```

```
        def remove_by_file(self, file_name: str) -> int:
        # ...
        def search(self, query_vector: np.ndarray, top_k: int) -> List[Dict[str, object]]:
        # ...
        def _persist(self) -> None:
        # ...
        def _rebuild(self, records: List[Dict[str, object]]) -> None:
        # ...
        def _clear(self) -> None:
        # ...
        def _normalize(vectors: np.ndarray) -> np.ndarray:
        # ...
    def get_vector_store() -> VectorStore:
    # ...
```

Процес додавання ембедингів у векторне сховище реалізовано через наведені нижче кроки:

Крок 1. Під час ініціалізації перевіряється, чи існує на диску попередньо сформований індекс. Якщо він є, система завантажує вектори та відповідні метадані. Якщо індекс відсутній, створюється нова структура збереження.

Крок 2. Перед додаванням кожний набір ембедінгів проходить нормалізацію. Це забезпечує коректність подальшого обчислення семантичної схожості – нормовані вектори дають стабільніші результати під час пошуку.

Крок 3. Після нормалізації ембедінги безпосередньо записуються у FAISS-індекс. Разом із векторами зберігається і набір метаданих, що описує кожен фрагмент тексту.

Крок 4. Після додавання нових записів оновлений індекс разом з метаданими зберігається на диск. Це гарантує, що при запуску застосунку дані залишаться доступними та не потребуватимуть повторної обробки документів.

Після виконання всіх цих кроків в модулі *routers/ingest.py* реалізовано повний цикл обробки документа: від моменту, коли користувач завантажує файл, до додавання відповідних векторних представлень у FAISS-індекс. Реалізація цього модуля подано у лістингу 3.5 [33].

Лістинг 3.5 Повний цикл обробки документа:

```
def _ingest_file(file_path: Path, settings: Settings) -> int:
    document = loader.load_document(file_path)
    chunks = splitter.split_text(
        document.text,
        chunk_size=settings.chunk_size,
        chunk_overlap=settings.chunk_overlap,
        base_metadata={"file": document.metadata.get("file", file_path.name)},
        page_texts=document.page_texts,
    )
    if not chunks: return 0
    texts: List[str] = [chunk.text for chunk in chunks]
    vectors = embeddings.embed_texts(texts)
    store = get_vector_store()
    metadatas = []
    for chunk in chunks:
        record = {"text": chunk.text}
        record.update(chunk.metadata)
        metadatas.append(record)
    store.add(vectors, metadatas)
    return len(chunks)

@router.post("/ingest", response_model=IngestResponse)
async def ingest_file(payload: IngestRequest, settings: Settings =
Depends(get_settings)) -> IngestResponse:
```

```
files_dir = _get_files_dir(settings)
file_path = safe_join(files_dir, payload.file_id)
if not file_path.exists():
    raise HTTPException(status_code=404, detail="File not found")
try:
    chunks_ingested = await run_in_threadpool(_ingest_file, file_path, settings)
except:
# ...
```

Повний цикл обробки документів здійснюється за наступними кроками:

Крок 1. Після отримання запиту система перевіряє, чи існує файл на диску. Якщо файл знайдено, виконується його повне зчитування: визначається тип документа, видобувається текстовий вміст та, за потреби, сторінкова структура.

Крок 2. Завантажений текст поділяється на послідовні перекривні фрагменти. Кожен фрагмент отримує власні метадані. Це забезпечує збереження контексту та підвищує якість подальшого пошуку.

Крок 3. Після створення фрагментів їхній текст подається на векторизацію. У результаті кожен фрагмент отримує числове представлення, яке відображає його семантичний зміст.

Крок 4. Для кожного фрагмента формується набір службової інформації: текстовий зміст, межі фрагмента, номер та дані про файл. Метадані збираються окремо, щоб їх можна було прив’язувати до векторів у процесі збереження.

Крок 5. Згенеровані вектори разом з метаданими додаються до FAISS.

Крок 6. Після успішного завершення обробки система повертає загальну кількість створених фрагментів. Це дає можливість клієнтській частині відобразити користувачу обсяг опрацьованих даних.

Після виконання описаних процедур модуль *services/rag_pipeline.py* забезпечує повний цикл роботи механізму RAG: від отримання користувацького запиту та пошуку релевантних фрагментів у векторному сховищі до формування

структурованого контексту, що буде переданий у LLM для генерації відповіді [34]. Код цього модуля наведено у лістингу 3.6.

Лістинг 3.6 Модуль побудови RAG- pipeline:

```
def retrieve(query: str, top_k: int) -> List[Dict[str, object]]:
    query_vector = embed_query(query)
    store = get_vector_store()
    hits = store.search(query_vector, top_k)
    hits.sort(key=lambda item: float(item.get("score", 0.0)), reverse=True)
    return hits

def build_context(chunks: Iterable[Dict[str, object]]) -> str:
    lines: List[str] = []
    for idx, chunk in enumerate(chunks, start=1):
        meta = chunk.get("meta", {})
        file_name = meta.get("file", "unknown")
        page = meta.get("page", "?")
        snippet = chunk.get("text", "").replace("\n", " ").strip()
        lines.append(f"[{idx}] (file: {file_name}, page: {page}) {snippet}")
    return "\n".join(lines)
```

Модуль побудови RAG-pipeline здійснюється за наступними кроками:

Крок 1. Спочатку запит користувача перетворюється на вектор за допомогою моделі ембедингів. Це дозволяє порівнювати його зміст із фрагментами у векторному сховищі.

Крок 2. Після отримання векторного представлення система виконує пошук у FAISS-індексі. У результаті повертається список найбільш схожих фрагментів із відповідними оцінками подібності.

Крок 3. Фрагменти впорядковуються за спаданням коефіцієнта подібності, для потрапляння найбільш інформативних частин на початок контексту.

Крок 4. Для кожного знайденого фрагмента створюється коротке текстове представлення. Усі отримані фрагменти об’єднуються в єдиний контекст, який передається моделі при генерації відповіді.

Після виконання всіх попередніх етапів RAG-процесу модуль *services/llm.py* відповідає за фінальний та ключовий елемент системи – генерацію відповідей за допомогою LLM. Саме тут здійснюється підготовка запиту до мовної моделі, передача системних інструкцій та використання контексту з релевантних фрагментів. Код модуля реалізовано у лістингу 3.7.

Лістинг 3.7 Генерація відповіді від LLM за допомогою RAG:

```
def generate_with_context(user_query: str, context: str) -> str:
    client = _get_client()
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": (
                    "You are a retrieval-augmented assistant. Use ONLY the provided "
                    "context to answer. If the context lacks the answer, state that "
                    "clearly and do not fabricate details."
                ),
            },
            {"role": "system", "content": f"Context:\n{context}"},
            {"role": "user", "content": user_query},
        ],
        temperature=0.1
    )

    return response.choices[0].message.content or ""
```

Модуль генерації відповідей LLM працює за такими основними кроками:

Крок 1. Під час першого виклику перевіряється наявність API-ключа та створюється екземпляр клієнта, який зберігається у кеші. Це дозволяє уникати повторного підключення при кожному запиті.

Крок 2. Перед формуванням запиту відбувається вибір мовної моделі – gpt-4o-mini. Це оптимізована версія GPT-4o, яка забезпечує швидке отримання відповідей при достатньо високій якості.

Крок 3. Далі створюється список повідомлень, до якого входить службова інструкція для моделі, текст користувацького запиту та зібраний контекст із релевантних фрагментів.

Крок 4. LLM обробляє отримані дані та повертає повідомлення.

Структура серверної частини вебзастосунку показана на рисунку 3.2.

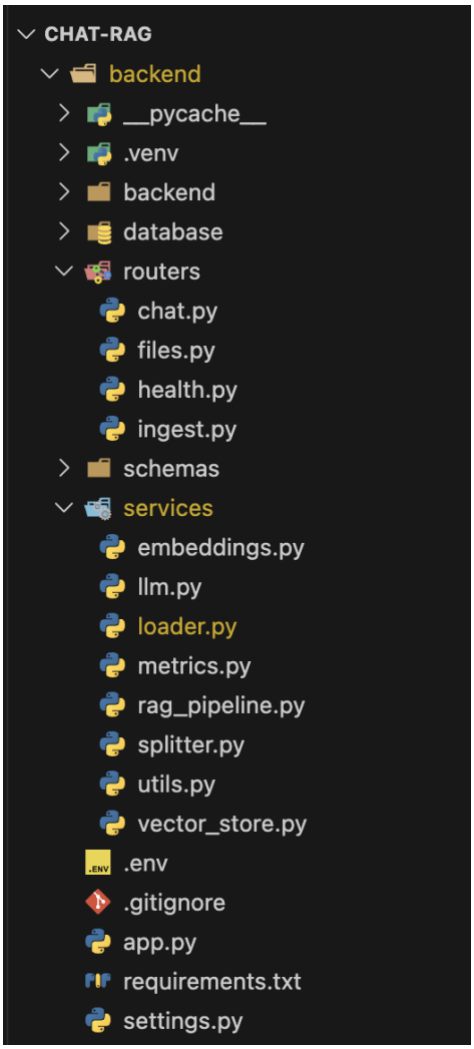


Рисунок 3.2 – Структура серверної частини вебзастосунку

Після детального аналізу основних файлів, що реалізують ключові етапи технології RAG, доцільно розглянути й маршрутні модулі. У серверній частині застосунку логіка обробки HTTP-запитів структурована за принципом router. Кожен із цих модулів відповідає за окремий функціональний блок системи та ізолює свою частину API, що спрощує підтримку, розширення та документування програмного коду. Короткий огляд кожного з них наведено у таблиці 3.1 [35].

Таблиця 3.1 – Опис API-routers

Назва файлу	**Опис**
1	**2**
chat.py	Це основний ендпоінт, який реалізує порівняння відповіді LLM без та з використанням механізму RAG. Виконує пошук релевантних векторів, формує контекст, генерує обидві відповіді, обчислює метрики якості та повертає аналітичний результат
files.py	Забезпечує повний цикл роботи з файлами: отримання списку документів, завантаження та попередній перегляд, а також видалення документа разом із відповідними векторами у FAISS-сховищі.
ingest.py	Реалізує інтерфейс завантаження документа та подальшого отримання даних в RAG-систему. Містить ендпоінти для збереження файлів, конвертації файлу в текстовий формат, розбиття на фрагменти, створення векторних представлень та додавання їх до векторного сховища FAISS.
health.py	Це найпростіший сервісний ендпоінт, який використовується для перевірки працездатності бекенда та моніторингу його доступності. Повертає мінімальний статус без виконання обчислень.

Модулі у папці *schemas* містять Pydantic-моделі, які визначають структуру запитів і відповідей для всіх API-ендпоінтів. Завдяки цьому забезпечується чітка типізація, валідація даних та узгодженість між клієнтською та серверною частинами. Використання таких моделей полегшує документування API та спрощує тестування системи.

Файл *settings.py* відповідає за централізоване завантаження конфігурацій застосунку, включно з параметрами роботи RAG-системи, шляхами до даних та ключами доступу. Це спрощує керування налаштуваннями та робить серверну частину більш гнучкою й масштабованою.

Серверна частина складається з чітко структурованих модулів, кожен з яких відповідає за окремий аспект роботи RAG-системи – від завантаження документів і побудови векторних представлень до обробки користувацьких запитів та взаємодії з LLM. Поділ системи на логічні блоки забезпечує зрозумілий порядок виконання операцій і дає змогу легко підтримувати й удосконалювати функціонал. Такий підхід сприяє стабільності роботи застосунку та створює основу для подальшого розвитку й масштабування програмної логіки в межах дослідницького проєкту.

3.2.2 Розробка клієнтської частини

Після завершення розробки серверної логіки переходять до розробки клієнтської частини застосунку, яка виконує роль інтерфейсу взаємодії користувача із системою, для забезпечення зручного доступу до основних можливостей RAG-системи. Для її реалізації було обрано сучасний підхід, орієнтований на компонентну структуру та реактивний принцип оновлення стану інтерфейсу. Ключовим інструментом стала бібліотека React, яка дає змогу будувати інтерфейс із окремих незалежних компонентів, що значно спрощує підтримку та розширення функціональності. Структура клієнтської частини застосунку показана на рисунку 3.3 [36].

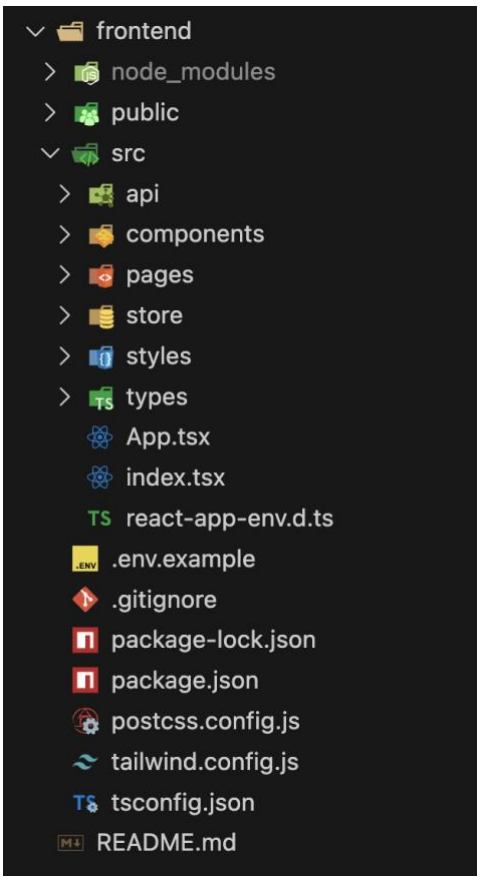


Рисунок 3.3 – Структура клієнтської частини вебзастосунку

Однією з ключових частин клієнтської логіки є модуль, що відповідає за комунікацію між клієнтом та сервером. Вебзастосунок використовує бібліотеку axios для уніфікації обробки HTTP-запитів, а також створюється окремий клієнт *apiClient*. Такий підхід спрощує обробку помилок, централізує конфігурацію API та підвищує надійність роботи інтерфейсу. Реалізацію обробки HTTP-запитів наведено в лістингу 3.8.

Лістинг 3.8 Обробка HTTP-запитів за допомогою axios:

```
export const apiClient = axios.create({ baseURL: apiBaseUrl});

apiClient.interceptors.response.use((response) => response, (error) => {
   return Promise.reject(error);
 }
);
```

Модуль обробки HTTP-запитів працює за такими основними кроками:

Крок 1. Формується окремий HTTP-клієнт із визначеною базовою адресою API. Це дозволяє уникнути дублювання URL у різних частинах програми та спрощує перенесення застосунку між середовищами.

Крок 2. Додається інтерцептор *response.use()*, який уніфікує формат обробки помилок. Якщо запит успішний – відповідь повертається без змін. У разі виникнення помилки – передається в узагальненому вигляді через *Promise.reject*.

Крок 3. Перехоплювач зводить різноманітні типи мережевих збоїв до передбачуваного формату. Це спрощує логіку компонентів інтерфейсу, оскільки вони отримують стандартизований об'єкт помилки.

Інтерфейс клієнтської частини побудовано за компонентним підходом, коли кожен елемент сторінки виконує власну чітко визначену роль. Компонент можна розглядати як автономний фрагмент інтерфейсу, який має власну логіку і стан. Така структурованість забезпечує кращу читабельність коду, полегшує повторне використання компонентів та спрощує подальше розширення функціональності.

Далі буде розглянуто основні компоненти клієнтської частини, які дозволили реалізувати повний цикл роботи користувача – від завантаження даних до отримання відповідей від LLM.

Одним із ключових елементів клієнтської частини є компонент FileUpload, який відповідає за первинний етап роботи з документами – їхнє завантаження та подальший запуск процесу індексації у RAG-системі. Компонент забезпечує інтуїтивний інтерфейс для додавання файлів, підтримує drag-and-drop та попередній перегляд документів. Після завантаження документа компонент автоматично здійснює два послідовних кроки: спочатку відправляє файл на сервер, а потім розбиття на фрагменти та побудову векторних представлень. Результати синхронізуються зі станом застосунку, що дозволяє оновлювати загальний список документів без додаткових дій з боку користувача. Це робить компонент важливою складовою клієнтської частини RAG-системи, яка забезпечує зручний процес підготовки даних перед подальшим аналізом [37].

Компонент ComparisonDashboard призначений для відображення аналітичних метрик, що порівнюють роботу LLM у двох режимах – базовому та з використанням RAG. Його завдання полягає у наочному представленні характеристик останнього запиту користувача, щоб забезпечити можливість аналізу продуктивності та якості відповідей. Компонент містить структуровану таблицю, у якій зведено ключові показники: затримку відповіді, кількість використаних токенів, метрики подібності (cosine similarity і average imilarity), а також текстові метрики BLEU та ROUGE-L. ComparisonDashboard виконує роль ключового інструмента для порівняльного оцінювання поведінки моделі й відіграє важливу роль у демонстрації ефективності RAG-підходу.

Компонент FilesPage виконує роботу з базою знань, яка використовується RAG-системою для пошуку релевантних фрагментів. Він виконує роль центральної сторінки для завантаження документів, відстеження процесу їх індексації та перегляду вже доступних файлів. Функціонально компонент складається з двох основних елементів: форми завантаження документів та списку файлів, які були додані до сховища.

Компонент ChatPage є центральним елементом аналітичного інтерфейсу вебзастосунку, який використовує компонент ComparisonDashboard для відображення аналітичних метрик. Інтерфейс містить текстову форму, у яку користувач вводить запит та обирає значення параметра top-k, що визначає кількість релевантних фрагментів контексту для алгоритму Dense Retrieval. Після надсилання форми компонент виконує запит до серверної частини та відображає результати у двох окремих блоках – відповідь базової LLM та відповідь, яка згенерована на основі RAG-пайплайна. Додатково представлено компонент RetrievedContext, для демонстрації релевантних фрагментів, які були відібрані векторним пошуком і використані для формування RAG-відповіді.

Таким чином, компоненти забезпечують логічну структуру інтерфейсу, що робить взаємодію користувача із системою легкою та зрозумілою. Такий підхід не лише підвищує зручність для користувача, а й підтримує легкість та масштабованість коду.

3.3 Інструкція користувача

Для забезпечення комфортної роботи з вебзастосунком було створено інтерфейс, орієнтований на простоту та зрозумілість. Його структура дозволяє користувачу послідовно виконувати всі етапи роботи з RAG-системою – від завантаження документів до аналізу отриманих відповідей. Далі буде розглянуто основні елементи взаємодії користувача із системою, щоб продемонструвати логіку роботи вебзастосунку та послідовність виконання ключових операцій.

При переході за посиланням на вебзастосунок користувач потрапляє на головну сторінку, яка відіграє роль вітального екрана. На цій сторінці подано принципи роботи RAG та основні переваги такого підходу. Окрім загальної інформації, на головній сторінці подано короткий огляд можливостей системи, а також пропонуються перехід до основних розділів, у яких здійснюється завантаження документів або порівняння відповідей моделі (рис. 3.4) [38].

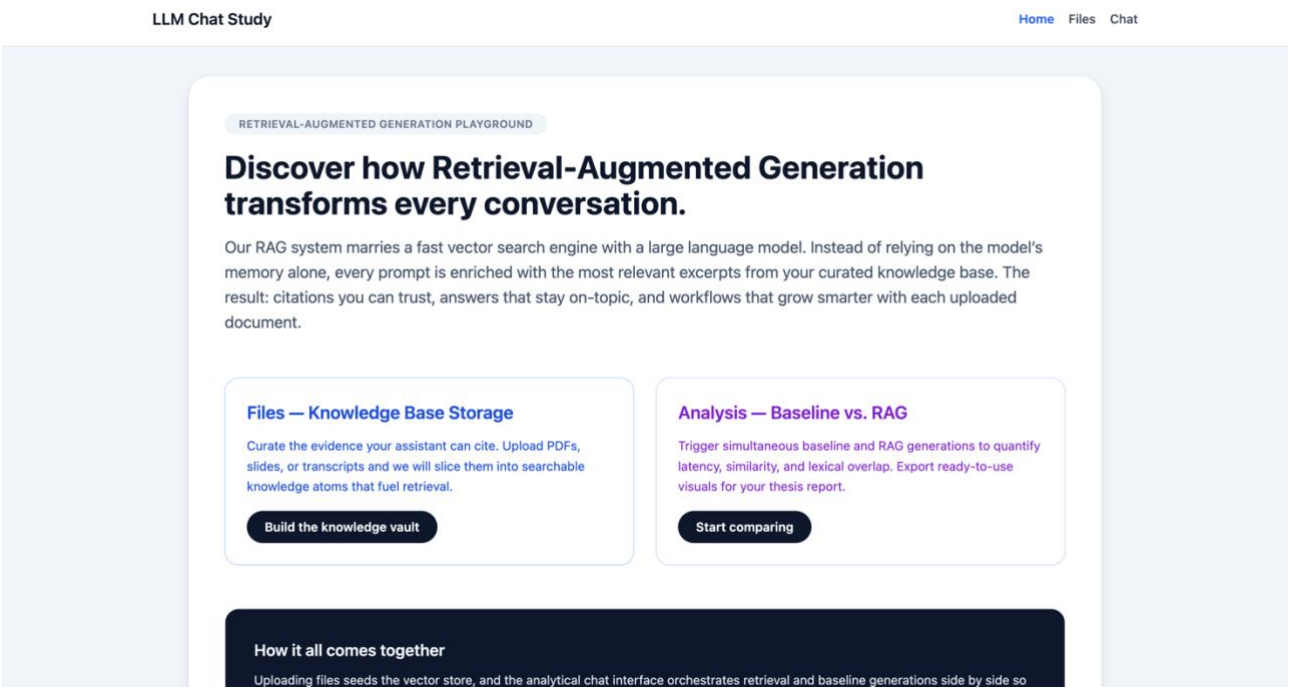


Рисунок 3.4 – Головна сторінка вебзастосунку

Після переходу до сторінки Files користувач отримує доступ до робочої області, призначеної для формування власної бази знань, яка надалі

використовується RAG-системою під час пошуку релевантного контенту. Внизу сторінки бачимо список усіх доданих документів, кожен з яких можна переглянути або видалити. Така організація дозволяє користувачеві підтримувати актуальну базу знань для роботи RAG-модуля (рис. 3.5).

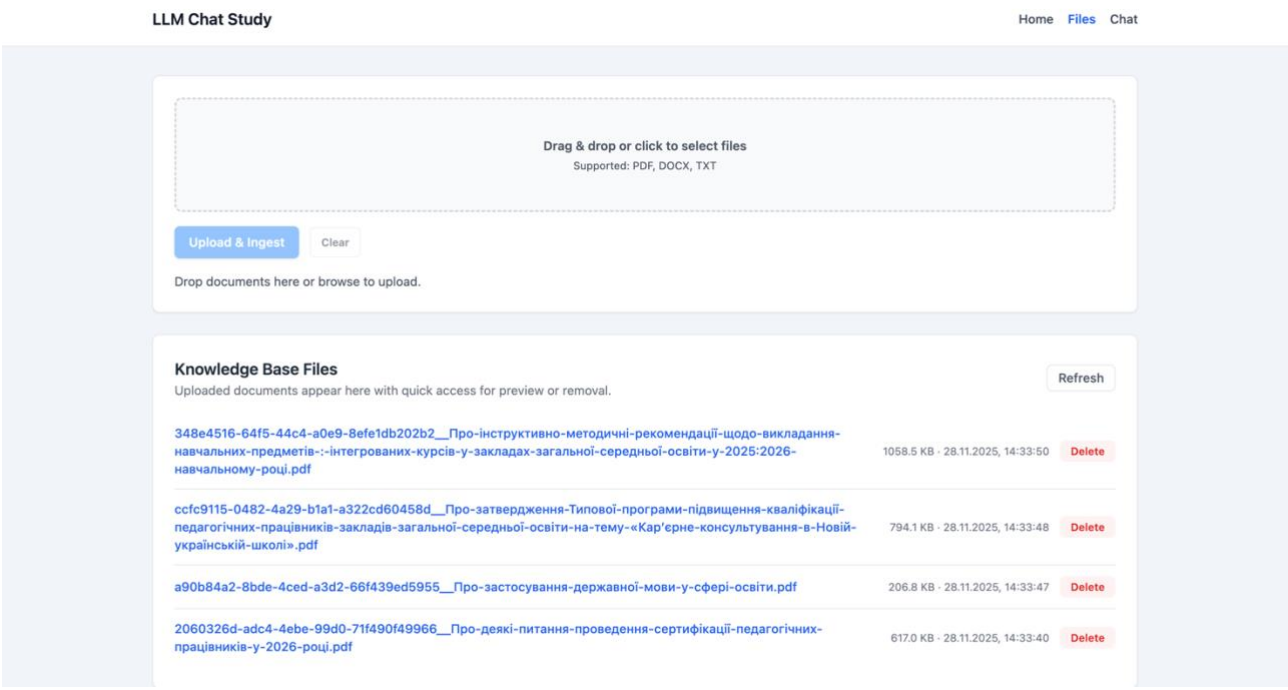


Рисунок 3.5 – Інтерфейс сторінки Files

У верхній частині сторінки розміщено інтерактивний блок для завантаження файлів. Користувач може додати документи двома способами: перетягнувши їх у виділену область або вибравши через стандартний файловий менеджер. Підтримуються найпоширеніші формати, зокрема PDF, DOCX та TXT. Після обрання файли з'являються у черзі, де для кожного елемента відображаються його назва, розмір, статус обробки та кнопка швидкого попереднього перегляду. Також можна видалити файл із черги до моменту відправлення на сервер. По завершенню процесу користувач отримує повідомлення, у якому зазначено кількість успішно оброблених файлів та кількість створених пошукових фрагментів (рис. 3.6).

Після успішного завантаження та індексації документу користувач може здійснити його попередній перегляд безпосередньо у вебзастосунку. Для цього

достатньо натиснути кнопку Preview, після чого файл відкривається у вбудованому переглядачі у форматі PDF. Такий режим дозволяє швидко ознайомитися зі змістом документа, перевірити його коректність та переконатися, що саме цей матеріал було додано до бази знань (рис. 3.7).

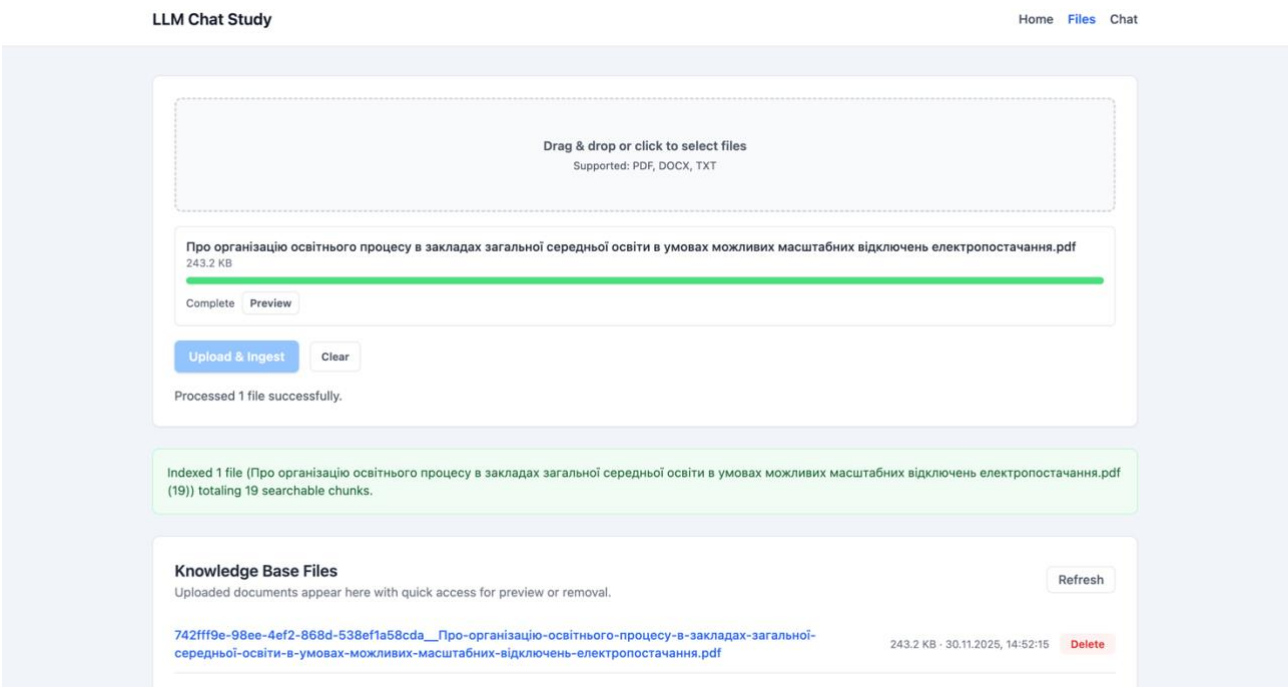


Рисунок 3.6 – Функціональність додавання файлу до бази знань

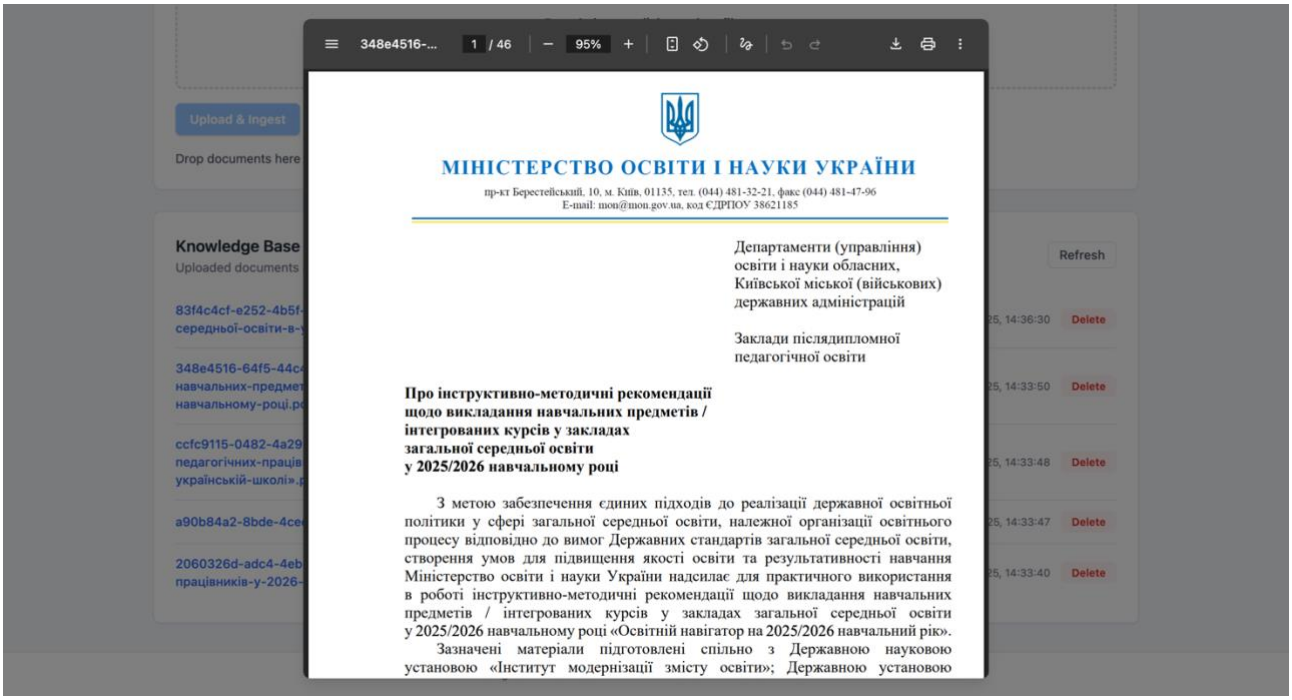


Рисунок 3.7 – Попередній перегляд доданого файлу

Після завершення роботи з файлами користувач має можливість перейти до головного етапу взаємодії з RAG-системою – аналітичного порівняння відповідей моделі. Перехід здійснюється через пункт меню Chat, що відкриває сторінку, призначену для дослідження різниці між базовими відповідями мовної моделі та відповідями, збагаченими релевантним контекстом із завантаженої бази знань. Відразу після переходу користувач потрапляє на інтерфейс, у якому можна задати дослідницький запит, обрати кількість контекстних фрагментів для пошуку й ініціювати процес порівняння (рис. 3.8).

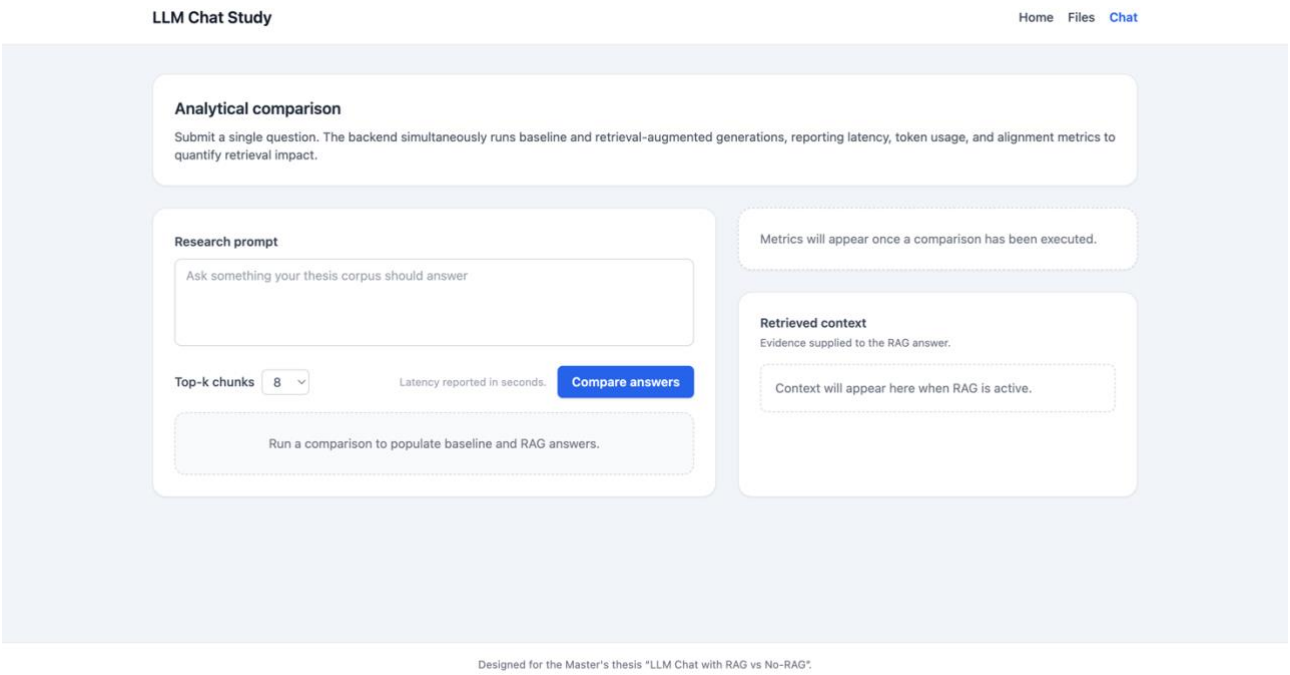


Рисунок 3.8 – Інтерфейс сторінки Chat

Після натискання кнопки «Compare answers» система надсилає сформований запит до серверної частини, яка паралельно обчислює дві відповіді – базову (без використання RAG) та розширену, сформовану на основі механізму пошуку релевантних фрагментів у завантажених документах. Після завершення обробки інтерфейс автоматично оновлюється і відображає результати у декількох логічних блоках.

У центральній частині сторінки з’являються дві окремі панелі з текстом відповідей: Baseline та RAG. Перша демонструє результат роботи моделі без

доступу до зовнішніх знань, тоді як друга відповідь, збагачену витягнутим контекстом із бази знань користувача. Це дозволяє оцінити вплив технології на повноту та інформативність відповіді.

Праворуч відображається панель Comparison dashboard, яка містить основні метрики, отримані під час обчислення: затримку відповіді (latency), кількість використаних токенів, а також показники схожості – cosine similarity, BLEU, ROUGE-L та середню схожість витягнутих фрагментів. Метрики відображають, як змінюється робота мовної моделі після додавання зовнішнього контексту, дозволяючи простежити динаміку її продуктивності (рис. 3.9).

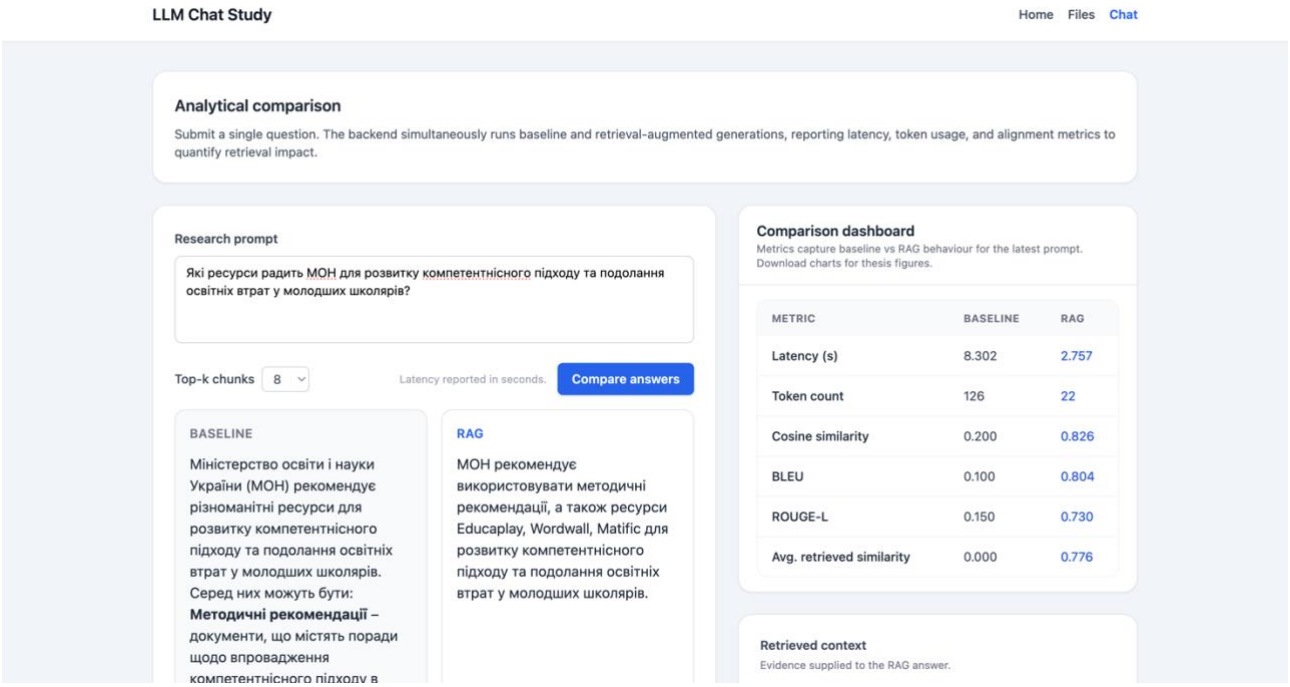


Рисунок 3.9 – Інтерфейс сторінки аналітичного порівняння після отримання результатів роботи базового та RAG-підходу

Нижче розміщується секція Retrieved context, у якій відображаються фрагменти текстів, що були автоматично знайдені та використані технологією RAG для формування відповіді. Для кожного фрагменту зазначається джерело файлу та значення релевантності (score). Це дозволяє відстежити, які саме дані вплинули на результат і яку частину знань система вважає найбільш доречною (рис. 3.10).

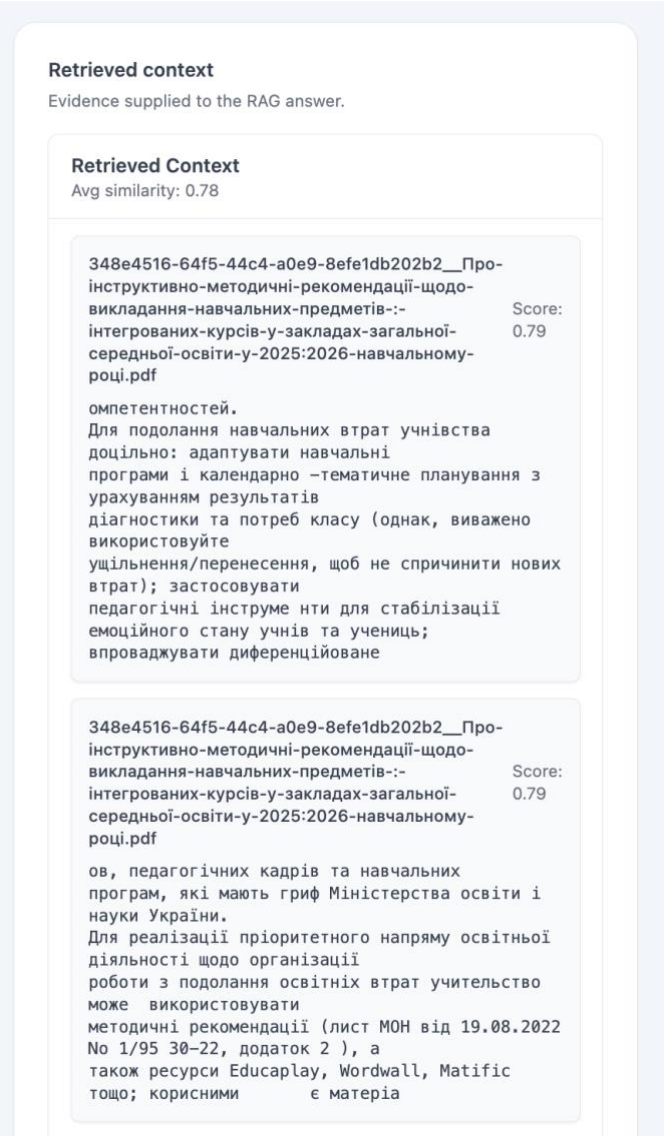


Рисунок 3.10 – Секція Retrieved context із фрагментами документів, які були використані для формування відповіді RAG-системи

Отже, представлений інтерфейс забезпечує зручне у використанні та логічно побудоване проходження всіх етапів роботи з RAG-системою – від формування бази знань до отримання порівняльних відповідей та перегляду використаного контексту. Користувач має змогу без додаткових технічних навичок завантажувати документи, стежити за ходом індексації, запускати аналітичні запити й переглядати результати роботи [39].

Подальший аналіз отриманих результатів, порівняння ефективності двох режимів роботи моделі та оцінка впливу зовнішнього контексту буде проведено у наступному розділі.

3.4 Дослідження та тестування розробленої моделі

Розроблена система Retrieval-Augmented Generation була перевірена та протестована шляхом проведення експериментального дослідження на реальних документах Міністерства освіти і науки України. Оскільки вони мають складну структуру, значну кількість термінів та характерну офіційну мову, вони виступають логічною базою для тестування RAG-систем. Метою – є оцінка того, наскільки успішно розроблений застосунок покращує якість відповідей мовної моделі за рахунок використання зовнішнього контексту, а також визначення характерних відмінностей між базовим режимом роботи та режимом із використанням RAG [40].

Тестування виконувалося через інтерфейс вебзастосунку після їх попереднього завантаження у вигляді PDF-документів. Такий підхід дає можливість оцінити поведінку моделі в ситуаціях, коли вона має працювати з текстами, що містять складні формулювання, офіційні визначення та специфічні терміни, тобто з інформацією, яку неможливо отримати у звичайному режимі без контексту. Тобто, тестування дозволяє побачити реальний вплив механізмів RAG на здатність моделі коректно працювати з офіційними джерелами.

Під час тестування були застосовані як часові, так і лінгвістичні метрики, що дозволили оцінити не лише швидкість, а й якість роботи моделі. Нижче наведено характеристику кожної з них:

– latency (затримка відповіді): показує, скільки часу займає повний цикл – обробка запиту, пошук контексту (для RAG), генерація відповіді та повернення результату;

– token count (кількість токенів): це обсяг фактичного тексту, який був згенерований моделлю;

– cosine similarity (косинусна подібність): показує, наскільки зміст відповіді семантично наближений до ключового питання та релевантних фрагментів. Baseline часто демонструє низьку подібність, оскільки генерує

інформацію загального характеру. RAG має вищий показник, оскільки відповідь сформована на основі фактичних даних з документа;

– BLEU: показує, наскільки відповідь моделі співпадає з еталонною відповіддю;

– ROUGE-L: оцінює відповідність у довгих ланцюгах слів, фіксуючи збіги структур і формулювань. Є індикатором того, наскільки структурно відповідь моделі збігається з документами або очікуваним еталоном;

– average retrieved similarity (середня подібність знайдених фрагментів): показує, наскільки добре механізм Dense Retrieval підібрав потрібні документи. Високі значення підтверджують, що індексація та пошук працюють коректно.

Для початку було проведено детальний аналіз роботи технології RAG на прикладі моделі GPT-4o-mini, яка використовувалась як основна мовна модель у межах експерименту. У межах тесту було сформовано запит (рис. 3.11). А також отримано основні числові показники (рис. 3.12) [41]:

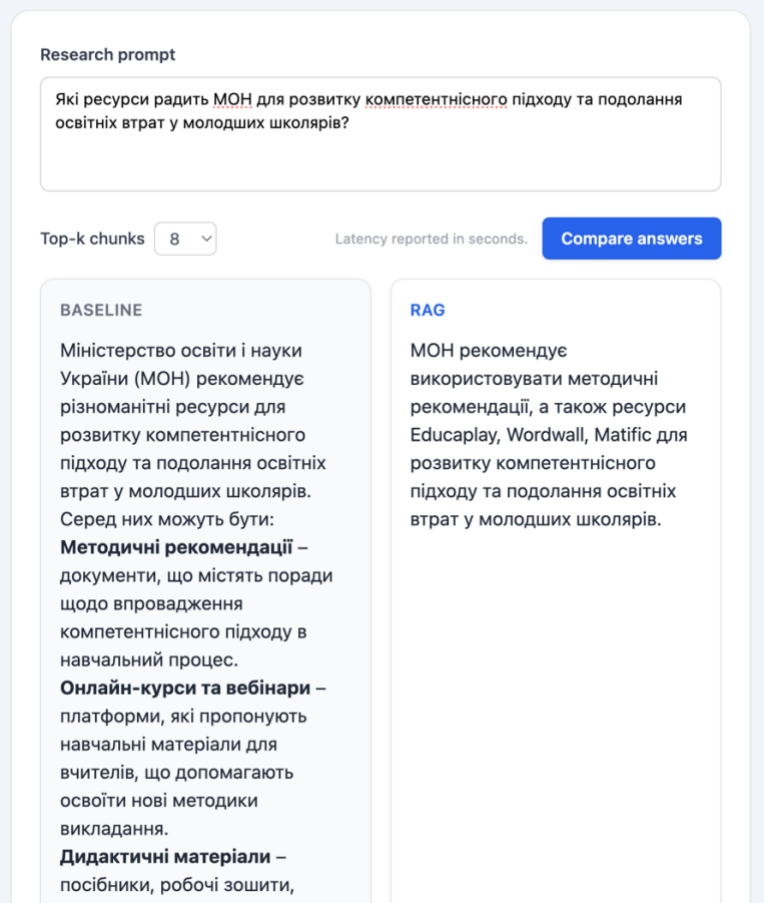
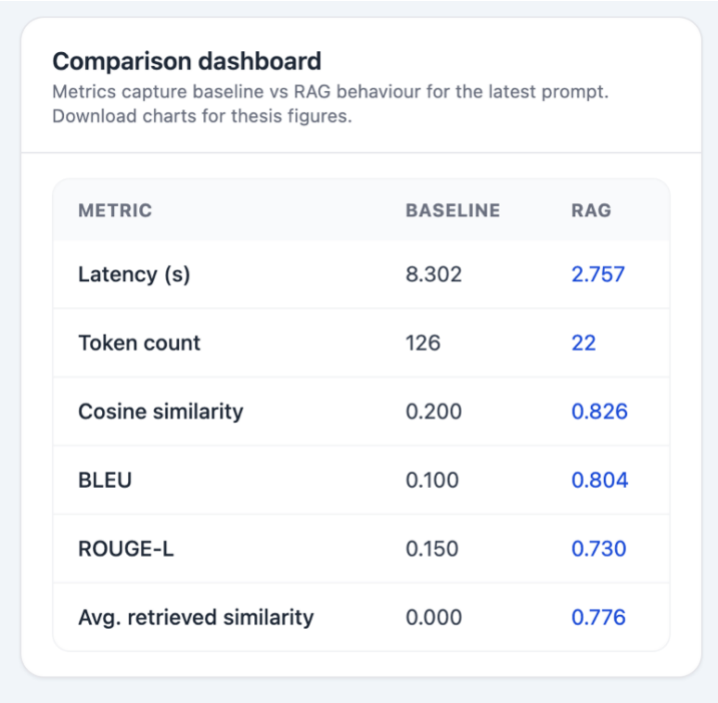


Рисунок 3.11 – Текстові результати за допомогою моделі GPT-4o-mini



Comparison dashboard

Metrics capture baseline vs RAG behaviour for the latest prompt. Download charts for thesis figures.

METRIC	BASELINE	RAG
Latency (s)	8.302	2.757
Token count	126	22
Cosine similarity	0.200	0.826
BLEU	0.100	0.804
ROUGE-L	0.150	0.730
Avg. retrieved similarity	0.000	0.776

Рисунок 3.12 – Метрики тестування за допомогою моделі GPT-4o-mini

На основі метрик було сформовано такі ключові спостереження:

– RAG-відповідь містить фактичні матеріали та конкретні джерела, серед яких: згадки про ресурси Wordwall, Matific, EducaPlay, методичні рекомендації та інші відомості, які присутні лише у завантажених документах. У базовому режимі модель не має прямого доступу до цих даних, що призводить до генерації узагальненої та менш точної відповіді;

– cosine similarity, BLEU та ROUGE-L зростають при використанні RAG, що свідчить про значно вищий рівень відповідності реальному змісту первинного документа. Наприклад, cosine similarity збільшується з 0.200 до 0.804, ROUGE-L – з 0.150 до 0.730, що демонструє якісне покращення семантичної та лексичної відповідності;

– token count у режимі RAG значно нижчий (22 проти 126), оскільки модель спирається на вже знайдені релевантні фрагменти, а не формує відповідь спираючись на загальні знання моделі;

– затримка відповіді скорочується з 8.302 секунди до 2.757 секунди, що є логічним наслідком того, що модель працює з меншим обсягом токенів та простішим сценарієм генерації;

Для повноцінного дослідження були проведені додаткові експерименти з іншими сучасними мовними моделями. Усі вони отримували один і той самий запит та працювали з однаковою базою знань (табл. 3.2) [42].

Таблиця 3.2 – Порівняльні результати роботи різних моделей у режимах Baseline та RAG

Модель	**Режим**	**Latency (s)**	**Token count**	**BLEU**	**ROUGE-L**	**Cosine similarity**	**Avg. retrieved similarity**
GPT-4o-mini	Baseline	8,302	126	0,1	0,15	0,826	0
	RAG	2,757	22	0,804	0,730	0,2	0,776
GPT-4.1	Baseline	6,1	132	0,18	0,22	0,27	0
	RAG	2,05	30	0,82	0,9	0,88	0,81
Gemini 2.5 Pro	Baseline	5,20	128	0,22	0,27	0,33	0
	RAG	2,48	28	0,78	0,87	0,84	0,74
Gemini 2.5 Flash	Baseline	4,35	140	0,15	0,19	0,25	0
	RAG	1,9	35	0,69	0,8	0,76	0,7

Аналіз отриманих порівнянь:

– всі моделі RAG стабільно підвищують якість відповіді, що видно з росту BLEU, ROUGE-L та cosine similarity. Це свідчить про універсальність підходу незалежно від архітектури моделі;

– GPT-4.1 демонструє найкращу якість, забезпечуючи найвищі метрики схожості та найстабільнішу поведінку;

– Gemini 2.5 Flash є найшвидшою моделлю, проте її відповіді менш точні порівняно з потужнішими системами.

– GPT-4o-mini демонструє оптимальний баланс між швидкістю та точністю. Незважаючи на свою компактність модель здатна працювати з документами так само ефективно, як більші моделі.

Проведене тестування доводить, що розроблена RAG-система ефективно виконує свої функції та суттєво підвищує якість роботи мовних моделей під час взаємодії з реальними офіційними документами. Завдяки механізму Dense Retrieval відповіді моделі стають більш обґрунтованими, наближеними до змісту джерела та позбавленими вигаданих фрагментів.

Порівняння різних моделей також показало, що технологія RAG забезпечує помітний приріст точності незалежно від типу моделі, хоча найкращі результати очікувано демонструють більш потужні системи.

Таким чином, створений вебзастосунок може застосовуватися як інструмент автоматизованого аналізу документів, а також як платформа для подальших досліджень поведінки мовних моделей у поєднанні з Retrieval-Augmented Generation [43].

3.5 Заходи щодо поліпшення вебзастосунку

Розроблена система демонструє стабільну роботу, забезпечуючи повний цикл взаємодії з технологією RAG – від завантаження документів та їх індексації до виконання порівняльного аналізу. У процесі розробки та експериментального застосування вебзастосунку, було виявлено низку можливостей для його подальшого вдосконалення.

Одним із ключових напрямів подальшого розвитку є створення повноцінної системи реєстрації користувачів, що дозволить розширити застосунок від індивідуального інструменту до багатокористувацької платформи. Необхідно забезпечити розмежування доступу, зберігання історії запитів, індивідуальні налаштування, а також прив’язувати індекси та векторні сховища до конкретних облікових записів. Це створює можливості для масштабування системи, даючи змогу різним групам користувачів працювати з власними наборами документів, не перетинаючись у спільному інформаційному просторі.

Окремої уваги потребує вдосконалення механізмів завантаження документів. Доцільно розширити підтримку форматів, додати варіанти завантаження та реалізувати асинхронну обробку великих масивів інформації.

Якість RAG-результатів може бути підвищена через підтримку різних embedding-моделей та застосування re-ranking підходів. Це забезпечить точніше вилучення контексту та кращу відповідність моделевих відповідей змісту документів.

Також можна використовувати більш спеціалізовані векторні бази даних – Pinecone, Weaviate, Qdrant або Milvus. Вони забезпечують горизонтальне масштабування, розподілене зберігання та гнучку роботу з метаданими, що дозволяє будувати багатокористувацькі рішення з високою продуктивністю навіть на великих наборах даних.

Корисними також є інтерактивні підказки, onboarding, темна тема та детальні індикатори процесів. Це підвищить зручність використання та зробить застосунок більш доступним для користувачів з різним рівнем підготовки.

Також систему може бути розширено за рахунок багатомовних embedding-моделей, OCR для роботи з зображеннями та мультимодальних RAG-компонентів. Це дозволить аналізувати ширший спектр навчальних і технічних матеріалів.

Отже, вебзастосунок демонструє значний потенціал для подальшого розвитку. Заплановані удосконалення здатні суттєво розширити його функціональність, підвищити надійність і зробити роботу користувачів ще комфортнішою та продуктивнішою [44].

ВИСНОВКИ

Таким чином, у кваліфікаційній роботі було досліджено підхід до підвищення ефективності великих мовних моделей шляхом додавання зовнішньої бази знань за допомогою технології Retrieval-Augmented Generation та вирішено такі завдання:

– було здійснено огляд і аналіз наукових джерел щодо особливостей роботи LLM та існуючих методів покращення їх ефективності, що дозволило систематизувати сучасні підходи до підсилення моделей і визначити їхні сильні та слабкі сторони у контексті роботи з фактичними даними;

– досліджено принципи функціонування технології RAG та визначено її переваги порівняно з іншими підходами, що дало можливість окреслити механізми її роботи, зрозуміти практичні особливості інтеграції у мовні моделі та обґрунтувати вибір цього методу для підвищення точності відповідей;

– розроблено концептуальну архітектуру застосунку, що реалізує принципи RAG для покращення роботи LLM, що дозволило узгодити всі функціональні модулі системи та забезпечити цілісність процесу від індексації документів до генерації остаточної відповіді;

– було сформовано алгоритм обробки запитів користувача із залученням механізмів пошуку релевантної інформації, що дало можливість забезпечити узгоджену послідовність дій між етапами отримання запиту, виконання семантичного пошуку та формування підсумкової відповіді;

– реалізувано програмний застосунок, що забезпечує генерацію відповідей із використанням RAG, що дозволило практично продемонструвати взаємодію між LLM та зовнішньою базою знань і підтвердити ефективність роботи запропонованого підходу;

– проведено експериментальне оцінювання якості відповідей з використанням RAG та без нього, проаналізовано отримані результати та сформовано висновки щодо ефективності застосування технології, що дало

можливість об’єктивно оцінити вплив зовнішнього контексту на точність та стабільність роботи мовної моделі.

У рамках кваліфікаційної роботи було проведено дослідження методу покращення ефективності LLM шляхом інтеграції зовнішніх джерел знань за допомогою технології RAG, шляхом програмної реалізації вебзастосунку, який забезпечує індексацію документів, семантичний пошук на основі Dense Retrieval та генерацію відповідей LLM з урахуванням релевантного контексту.

Побудовано покроковий алгоритм для методу Dense Retrieval та було візуалізовано алгоритм за допомогою блок-схеми.

Власноруч створено унікальний набір даних, що включає 20 ретельно відібраних питань, орієнтованих на зміст завантажених документів.

Для кожного питання було відібрано еталонні відповіді, що дозволяє забезпечити об’єктивність подальшого порівняння, тестування та виконати коректну оцінку точності моделі за допомогою використовуваних метрик.

Розроблено зручний UI, за допомогою якого користувач може порівняти режими роботи системи – базовий та з використанням технології RAG. Застосунок автоматично формує порівняння результатів, відображаючи їх у вигляді зрозумілих текстових панелей та набору метрик, що дає змогу оцінити різницю у точності, повноті та змістовності відповідей між підходами.

Наукова новизна роботи полягає в удосконаленні процесу семантичного пошуку шляхом застосування щільних векторних уявлень і динамічного формування контексту для LLM, що дозволило отримати глибше розуміння впливу зовнішніх баз знань на якість генерації та продемонструвати переваги технології RAG в умовах практичних завдань.

Результати роботи апробовано у вигляді 2 тез доповідей під час Міжнародної студентської наукової конференції «Глобалізація наукових знань: Міжнародна співпраця та інтеграція галузей наук» [45] та ІІ Міжнародної науково-практичної студентської конференції «ІТ-простір сьогодення: тенденції, інновації та перспективи розвитку» [46].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Творошенко, І.С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. *Харків: ХНУРЕ*.

2. Тітов, С. В., Тітова, О. В., & Чорна, О. С. (2022). Опис нескоротних наборів ознак в приблизних множинах з використанням систем числення. Збірник наукових праць Харківського національного університету Повітряних Сил, (1 (71)), 106-110.

3. What are large language models (LLMs)? URL: https://www.ibm.com/think/topics/large-language-models (дата звернення 05.10.2025).

4. Retrieval Augmented Generation (RAG) for LLMs. URL: https://www.promptingguide.ai/research/rag (дата звернення 07.10.2025).

5. Ситніков, Д. Е., Ситнікова, П. Е., Тітов, С. В., & Тітова, О. В. (2021). Фільтрація результуючого набору асоціативних правил з точки зору оцінки цікавості. Системи обробки інформації, (1 (164)), 83-88.

6. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Cluster representation of the structural description of images for effective classification, *Computers, Materials & Continua*, 73(3), pp. 6069-6084.

7. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv. URL: https://arxiv.org/abs/2005.11401 (дата звернення 10.10.2025).

8. Tvoroshenko I., Gorokhovatskyi V., Kobylin O., and Tvoroshenko A. (2023) Application of deep learning methods for recognizing and classifying culinary dishes in images, *International Journal of Academic and Applied Research*, 7(9), pp. 57-70.

9. Dense Retrieval in RAG: Must-Read Papers Before Implementation. URL: https://medium.com/@287961061/dense-retrieval-in-rag-must-read-papers-before-implementation-2a91b4e72298 (дата звернення 10.10.2025).

10. Кобилін, О.А., & Творошенко, І.С. (2021). Методи цифрової обробки зображень: навч. посібник. *Харків: ХНУРЕ*.

11. Тітов, С. В., Тітова, О. В., & Чорна, О. С. (2023). Метод знаходження апроксимацій приблизних множин з використанням систем числення. Системи обробки інформації, (2 (173)), 58-62.

12. Wang, Z., Teo, S. X. M., Ouyang, J., Xu, Y., & Shi, W. (2024). M-RAG: Reinforcing large language model performance through retrieval-augmented generation with multiple partitions. arXiv. URL: https://arxiv.org/abs/2405.16420 (дата звернення 15.10.2025).

13. Gao, Y., Ma, Z., Lin, J., Zhang, C., Liu, Z., Wang, X., Han, R., Liu, H., Li, P., Li, J., Feng, S., & Li, C. (2024). Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv. URL: https://arxiv.org/abs/2312.10997 (дата звернення 15.10.2025).

14. Pomazan, V., Tvoroshenko, I., & Gorokhovatskyi, V. (2023). Development of an application for recognizing emotions using convolutional neural networks, *International Journal of Academic Information Systems Research*, 7(7), pp. 25-36.

15. Коваленко, А. І., Тітов, С. В., Тітова, О. В., & Чорна, О. С. (2022). Оцінка вимог до параметрів сигналів при V-подібному розподілі частот у математичній моделі багатопозиційної системи випромінювачів.

16. Welcome to Faiss Documentation. URL: https://faiss.ai/index.html (дата звернення 15.10.2025).

17. Tvoroshenko I., and Gorokhovatskyi V. (2022) The Application of Hybrid Intelligence Systems for Dynamic Data Analysis, *International Journal of Engineering and Information Systems*, 6(2), pp. 40-48.

18. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, *Сучасні інформаційні системи*, 6(3), С. 5-12.

19. Чорна, О. С., Дідик, П. Ю., Тітов, С. В., & Тітова, О. В. (2024). Usage of clustering algorithms for automating route planning in transportation routing tasks. Системи обробки інформації, (1 (176)), 115-123.

20. Gorokhovatskyi, V., Gorokhovatskyi, O., Yevgenyi, P., & Olena, P. (2018). Quantization of the Space of Structural Image Features as a Way to.

21. Tvoroshenko I., and Gorokhovatskyi V. (2022) The Application of Hybrid Intelligence Systems for Dynamic Data Analysis, International Journal of Engineering and Information Systems, 6(2), pp. 40-48.

22. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Al-Dhaifallah M. (2022) Classification of Images Based on a System of Hierarchical Features, *Computers, Materials & Continua*, 72(1), pp. 1785-1797.

23. The open source AI code editor. URL: https://code.visualstudio.com/ (дата звернення 25.10.2025).

24. Gorokhovatskyi, V., & Tvoroshenko, I. (2024). An effective method for transforming an image description into a compact vector for classification.

25. Humanloop. (2024, March 26). RAG architectures: An overview of retrieval-augmented generation architectures. URL: https://humanloop.com/blog/rag-architectures (дата звернення 25.10.2025).

26. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., Gadetska S., and Al-Dhaifallah M. (2023) Statistical data analysis models for determining the relevance of structural image descriptions, *IEEE Access*, vol. 11, pp. 126938-126949.

27. Yakovleva, O., Matúšová, S., Tvoroshenko, I., & Isaiev, Y. (2024). Visitor counting based on video stream analysis from surveillance cameras to solve various business problems. *Verejná správa a regionálny rozvoj ekonómia, manažment a marketing, XX (1)*, 67-87.

28. Gorokhovatskyi, V., & Tvoroshenko, I. (2020). Image Classification Based on the Kohonen Network and the Data Space Modification.

29. FastAPI framework, high performance, easy to learn, fast to code, ready for production. URL: https://fastapi.tiangolo.com/ (дата звернення 28.10.2025).

30. Gorokhovatskyi, V., Chmutov, Y., Tvoroshenko, I., & Kobylin, O. (2025). Reducing computational costs by compressing the structural description in image classification methods. *Advanced Information Systems*, *9*(1), 5-12.

31. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Handwritten character recognition models based on convolutional neural networks, *International Journal of Academic Engineering Research*, 7(9), pp. 64-72.

32. Karakonstantyn, D., & Tvoroshenko, I. (2024). About the issue of optimization the performance of the server part of the information system.

33. Gorokhovatskyi V., Tvoroshenko I., and Yakovleva O. (2024) Transforming image descriptions as a set of descriptors to construct classification features, *Indonesian Journal of Electrical Engineering and Computer Science*, 33(1), pp. 113-125.

34. Building Your First RAG System with Python and OpenAI. URL: https://dev.to/mazyaryousefinia/building-your-first-rag-system-with-python-and-openai-1326 (дата звернення 01.11.2025).

35. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2024). Improving the Effectiveness of Image Classification Structural Methods by Compressing the Description According to the Information Content Criterion. *Computers, Materials & Continua*, *80*(2).

36. Gorokhovatskyi, V., Tvoroshenko, I., Kobylin, O., & Vlasenko, N. (2023). Search for visual objects by request in the form of a cluster representation for the structural image description, *Advances in Electrical and Electronic Engineering*, 21(1), pp. 19-27.

37. How to use Local Embedding models, and Sentence Transformers. URL: https://medium.com/@jacobrcasey135/how-to-use-local-embedding-models-and-sentence-transformers-c0bf80a00ce2 (дата звернення 04.11.2025).

38. Гороховатський В., Передрій О., Творошенко І., Марков Т. (2023) Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень, *Сучасні інформаційні системи*, 7(1), С. 5-13.

39. Гороховатський В.О., Творошенко І.С. (2022) Аналіз багатовимірних даних за описом у формі множини компонент: монографія. Харків: ХНУРЕ, 124 с.

40. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., & Hudáková, M. (2025). Image description compression in classification structural methods. *IEEE Access*, *13*, 43631-43641.

41. Gorokhovatskyi V., Tvoroshenko I. (2023) Identification of visual objects by the search request. *International scientific symposium «INTELLIGENT SOLUTIONS-S». Computational intelligence (results, problems and perspectives). Decision making theory: proceedings of the international symposium, September 28, 2023, Kyiv-Uzhorod, Ukraine*, pp. 25-27.

42. Tvoroshenko I., Pomazan V., Gorokhovatskyi V., and Kobylin O. (2023) Application of video data classification models using convolutional neural networks, *International Journal of Academic and Applied Research*, 7(11), pp. 134-145.

43. Порівняння GPT-4o mini, GPT-4o та GPT-4 – що краще обрати під свої задачі. URL: https://theinweb.media/porivnyannya-gpt-4o-mini-gpt-4o-ta-gpt-4/ (дата звернення 05.11.2025).

44. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., Hudáková, M., & Gorokhovatskyi, O. (2024). Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set. *IEEE Access*, *12*, 73376-73385.

45. Уткін Є. І. (2025). Дослідження методу Dense Retrieval як засобу покращення роботи LLM систем у технології RAG. IX Міжнародна студентська наукова конференція «Глобалізація наукових знань: міжнародна співпраця та інтеграція галузей наук» (Черкаси, 7 листопада 2025 р.). С. 297-299.

46. Уткін Є. І. (2025). Використання технології RAG як одна зі стратегій покращення роботи LLM систем. II Міжнародній науково- практичній студентській конференції «IT-простір сьогодення: тенденції, інновації та перспективи розвитку».