Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

Пояснювальна записка
до кваліфікаційної роботи бакалавра
за спеціальністю 123 «Комп’ютерна інженерія»

на тему: РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ НА ОСНОВІ ВЕЛИКОЇ МОВНОЇ МОДЕЛІ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ RAG

Проектував ________________ А. В. Рибак

Керівник роботи ________________ Д. І. Кузнєцов

Нормоконтроль ________________ Д. І. Кузнєцов

Завідувач кафедри ________________ А. І. Купін

Кривий Ріг
2026

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти бакалавр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_______________________ А. І. Купін

"____" _________________ 20__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Рибак Андрій Віталійович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка інформаційної системи на основі великої мовної моделі з використанням технології RAG

керівник роботи Кузнєцов Денис Іванович, канд. тех. наук, доц. каф. КСМ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "27" січня 2026 року №64с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи Python 3.12, OpenAI API (моделі GPT-4o mini та text-embedding-3-small), векторна база даних Qdrant, реляційна СУБД PostgreSQL 16, бібліотека python-telegram-bot, Docker Compose, бібліотеки PyMuPDF та python-docx для обробки документів, навчально-методичні матеріали кафедри у форматах PDF та DOCX

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Розділ 1. Аналіз предметної області та огляд технологій. Розділ 2. Проектування та розробка системи. Розділ 3. Тестування та атестація системи. Висновок. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) 15 рисунків, 18 таблиць. Загальна архітектура RAG-системи (рис. 2.1); діаграма послідовності обробки запиту користувача (рис. 2.2); схема бази даних PostgreSQL; конвеєр індексування документів; схема механізму Multi-Head Attention (рис. 1.1); загальна схема RAG-пайплайну (рис. 1.3).

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Кузнєцов Д.І., доц. каф. КСМ, канд. техн. наук		
2	Кузнєцов Д.І., доц. каф. КСМ, канд. техн. наук		
3	Кузнєцов Д.І., доц. каф. КСМ, канд. техн. наук		

7. Дата видачі завдання 01.02.2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми та узгодження завдання з керівником.	01.02.2026	
2.	Огляд літератури та аналіз існуючих аналогів.	15.04.2026	
3.	Аналіз предметної області, вивчення технологій RAG, LLM, Qdrant.	20.04.2026	
4.	Проектування архітектури системи та структури бази даних.	27.04.2026	
5.	Розробка конвеєра індексування документів. Реалізація RAG-логіки та серверної частини.	10.05.2026	
6.	Розробка Telegram-бота та інтеграція компонентів. Контейнеризація системи.	15.05.2026	
7.	Функціональне тестування та усунення помилок. Тестування якості RAG-пайплайну, вимірювання метрик.	20.05.2026	
8.	Оформлення пояснювальної записки.	31.06.2026	
9.	Нормоконтроль та усунення зауважень. Отримання відгуку.	01.06.2026	
10.	Захист кваліфікаційної роботи.	25.06.2026	

Студент ______________ Рибак А.В.
(підпис) (прізвище та ініціали)

Керівник роботи ______________ Кузнєцов Д.І.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 68 сторінок, 15 рисунків, 18 таблиць, 1 додаток, 20 використаних джерел, _ слайдів презентації.

Об'єкт дослідження – інформаційна система інтелектуального пошуку та генерації відповідей на основі навчальних матеріалів кафедри.

Проєкт складається з трьох розділів.

У першому розділі проведено аналіз предметної області та огляд технологій. Досліджено принципи роботи та обмеження великих мовних моделей, обґрунтовано вибір архітектури Retrieval-Augmented Generation як ефективного засобу розширення контексту моделі предметно-специфічними даними. Розглянуто векторні бази даних, моделі ембедингів, OpenAI API, Python-стек, Docker та Telegram Bot API.

У другому розділі виконано проектування та розробку системи. Сформовано функціональні та нефункціональні вимоги, визначено технологічний стек. Спроектовано модульну архітектуру, структуру реляційної бази даних PostgreSQL та векторного сховища Qdrant. Реалізовано конвеєр індексування документів форматів PDF та DOCX, логіку семантичного пошуку, генерацію відповідей за допомогою GPT-4o mini та Telegram-бот як інтерфейс користувача. Систему контейнеризовано за допомогою Docker Compose.

У третьому розділі проведено комплексне тестування та атестацію системи. Виконано функціональне тестування, тестування якості RAG-пайплайну, вимірювання продуктивності та порівняльний аналіз з існуючими аналогами. Оцінка модуля пошуку за метрикою F1@5 склала 0,55 (Recall@5 = 0,92). Вартість обробки одного запиту становить близько 0,001 USD.

ВЕЛИКА МОВНА МОДЕЛЬ, RAG, ВЕКТОРНА БАЗА ДАНИХ, QDRANT, GPT-4O MINI, POSTGRESQL, PYTHON, DOCKER, TELEGRAM-БОТ, ЕМБЕДИНГИ

Bachelor’s qualifying paper: 68 pages, 15 figures, 18 tables, 1 addition, 20 used sources, _ presentation slides.

Object of research is an intelligent information system for search and answer generation based on the department's educational materials.

The project consists of three chapters.

The first chapter provides an analysis of the subject domain and a review of the relevant technologies. The principles and limitations of large language models are examined, and the choice of Retrieval-Augmented Generation architecture is justified as an effective method for extending the model's context with domain-specific data. Vector databases, embedding models, the OpenAI API, the Python stack, Docker, and the Telegram Bot API are reviewed.

The second chapter covers the design and development of the system. Functional and non-functional requirements are defined, and the technology stack is selected. The modular system architecture, the PostgreSQL relational database schema, and the Qdrant vector store are designed. The document indexing pipeline for PDF and DOCX formats, the semantic search logic, answer generation using GPT-4o mini, and a Telegram bot as the user interface are implemented. The system is containerized using Docker Compose.

The third chapter presents comprehensive testing and validation of the system. Functional testing, RAG pipeline quality evaluation, performance measurements, and a comparative analysis with existing analogues are performed. The retrieval module achieved an F1@5 score of 0.55 (Recall@5 = 0.92). The cost per query is approximately 0.001 USD.

LARGE LANGUAGE MODEL, RAG, VECTOR DATABASE, QDRANT, GPT-4O MINI, POSTGRESQL, PYTHON, DOCKER, TELEGRAM BOT, EMBEDDINGS

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 8
ВСТУП 9
1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ТЕХНОЛОГІЙ 11
1.1 Великі мовні моделі: принципи роботи та обмеження 11
1.2 Технологія Retrieval-Augmented Generation 14
1.3 Векторні бази даних 17
1.4 Моделі ембедингів 21
1.5 OpenAI API та модель GPT-4o mini 23
1.6 Python та основні бібліотеки серверної частини 25
1.7 Docker та контейнеризація 26
1.8 PostgreSQL для зберігання історії розмов 28
1.9 Telegram Bot API як інтерфейс користувача 29
1.10 Висновки 30
2. ПРОЕКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ 31
2.1 Постановка задачі та вимоги до системи 31
2.2 Загальна архітектура системи 32
2.3 Структура бази даних PostgreSQL 34
2.4 Організація векторного сховища Qdrant 35
2.5 Конвеєр індексування документів 37
2.6 Реалізація логіки RAG 38
2.7 Реалізація Telegram-бота 40
2.8 Контейнеризація системи за допомогою Docker Compose 41
2.9 Тестування модулів системи 42
2.10 Висновки 42

3. ТЕСТУВАННЯ ТА АТЕСТАЦІЯ СИСТЕМИ ... 43
3.1 План тестування ... 43
3.2 Функціональне тестування ... 43
3.3 Тестування якості RAG-пайплайну ... 47
3.4 Вимірювання продуктивності системи ... 48
3.5 Порівняльний аналіз параметрів системи ... 49
3.6 Порівняння з існуючими аналогами ... 51
3.7 Оцінка якості пошуку за метрикою F1 ... 52
3.8 Порівняння з існуючими аналогами ... 54
3.9 Висновки ... 55
ВИСНОВКИ ... 56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 57
Додаток А ... 59

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface (інтерфейс програмування застосунків)
BPE – Byte Pair Encoding (кодування парами байтів)
HNSW – Hierarchical Navigable Small World (ієрархічний навігаційний тісний світ)
LLM – Large Language Model (велика мовна модель)
RAG – Retrieval-Augmented Generation (генерація, доповнена пошуком)
RLHF – Reinforcement Learning from Human Feedback (навчання з підкріпленням на основі відгуків)
SFT – Supervised Fine-Tuning (контрольоване донавчання)
БД – база даних
БЗ – база знань
СУБД – система управління базами даних

ВСТУП

Стрімкий розвиток штучного інтелекту та великих мовних моделей (Large Language Models, LLM) відкриває принципово нові можливості для побудови інтелектуальних інформаційних систем. Сучасні LLM демонструють вражаючі результати в задачах розуміння природної мови, генерації тексту та відповідей на запитання. Проте їхнє практичне застосування в корпоративних та освітніх середовищах стикається з суттєвим обмеженням: моделі навчені на фіксованому наборі даних і не мають доступу до актуальної або предметно-специфічної інформації.

Технологія Retrieval-Augmented Generation (RAG) є відповіддю на цей виклик. Вона поєднує можливості LLM із механізмом семантичного пошуку по зовнішніх базах знань, дозволяючи системі генерувати відповіді на основі актуальних і релевантних документів, завантажених розробником. Завдяки цьому RAG-системи здатні надавати точні, обґрунтовані та контекстно-залежні відповіді без необхідності повного перенавчання моделі.

Актуальність даної роботи обумовлена зростаючою потребою в автоматизованих асистентах для освітніх закладів, зокрема для надання студентам оперативного доступу до навчальних матеріалів, методичних рекомендацій та лабораторних завдань. Традиційні підходи - пошук по файловому сховищу або статичні FAQ - не забезпечують потрібного рівня зручності та семантичної точності. Інтелектуальний чат-бот на основі RAG здатний розуміти природномовні запити та надавати розгорнуті відповіді саме за змістом завантажених документів.

Метою кваліфікаційної роботи є розробка інформаційної системи на основі великої мовної моделі з використанням технології RAG, що реалізована у вигляді Telegram-бота та призначена для роботи з навчально-методичними матеріалами кафедри.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

- провести аналіз існуючих підходів до побудови RAG-систем та огляд суміжних технологій;
- обрати та обґрунтувати стек технологій: векторну базу даних Qdrant, реляційну базу даних PostgreSQL, Python як мову серверної частини, OpenRouter як шлюз до LLM;
- спроектувати архітектуру системи та структуру бази даних;
- реалізувати серверну частину, конвеєр індексування документів та механізм семантичного пошуку;

- розробити Telegram-бот як інтерфейс взаємодії користувача з системою;
- контейнеризувати систему за допомогою Docker та провести тестування і атестацію розробленого рішення.

Об'єктом розробки є інформаційна система інтелектуального пошуку та генерації відповідей на основі бази знань. Предметом розробки є методи і засоби реалізації RAG-архітектури з використанням сучасних відкритих технологій.

Практична цінність роботи полягає в тому, що розроблена система може бути використана для надання студентам автоматизованої допомоги при роботі з лабораторними роботами та методичними матеріалами кафедри комп'ютерних систем та мереж, а також легко адаптована до інших предметних областей.

Результати роботи можуть бути використані як основа для подальшого розвитку системи: розширення бази знань, додавання нових каналів взаємодії або інтеграції з іншими інформаційними системами університету.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ТЕХНОЛОГІЙ

1.1 Великі мовні моделі: принципи роботи та обмеження

Великі мовні моделі LLM - це клас нейронних мереж, що навчаються на масштабних текстових корпусах і здатні виконувати широкий спектр завдань обробки природної мови: генерацію тексту, відповіді на запитання, переклад, класифікацію, написання коду та інше. Сучасні LLM побудовані на архітектурі трансформера, запропонованій у роботі Vaswani et al. 2017, і використовують механізм самоуваги (self-attention) для моделювання залежностей між словами на будь-якій відстані у тексті.

До появи архітектури трансформера основним інструментом обробки природної мови були рекурентні нейронні мережі (RNN) та їх вдосконалені версії, такі як мережі довгої короткострокової пам'яті (LSTM) та керовані рекурентні блоки (GRU). Головним недоліком рекурентних архітектур була необхідність послідовної обробки даних: для того, щоб опрацювати n-те слово, мережа мала спочатку обробити всі попередні слова в послідовності. Це призводило до двох критичних проблем: неможливості ефективного розпаралелювання обчислень на сучасних графічних процесорах та проблеми «зникаючого градієнта», через яку мережа втрачала зв'язок між словами на початку та в кінці довгого речення.

Ключовою інновацією архітектури трансформера став механізм самоуваги, який дозволяє моделі математично розраховувати вагу кожного слова відносно інших у всьому вхідному тексті одночасно. Кожен токен перетворюється на три вектори: запит (Query, Q), ключ (Key, K) та значення (Value, V). Процес обчислення ваг уваги описується формулою 1.1:

$$Attention\,(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \qquad (1.1)$$

де d_k - розмірність векторів ключів.

Ця операція дозволяє моделі «фокусуватися» на граматичних та логічних зв'язках незалежно від відстані між словами. Сучасні LLM використовують багатоканальну увагу (Multi-Head Attention), що дозволяє системі одночасно аналізувати текст на різних рівнях: від синтаксичних конструкцій до глибоких семантичних контекстів.

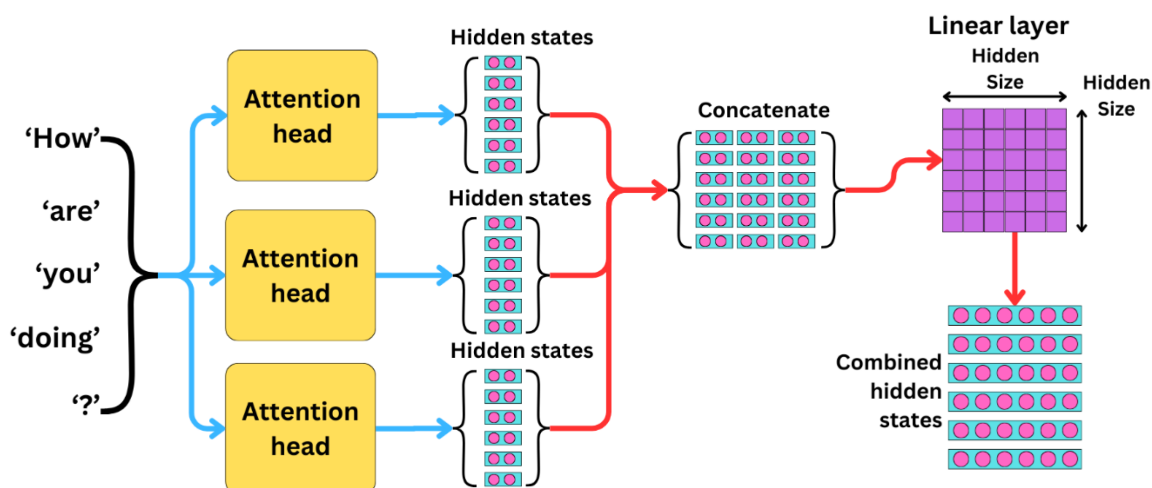

Рисунок 1.1 — Схема механізму Multi-Head Attention

Основною одиницею обробки в LLM є токен - фрагмент тексту, що може відповідати слову, частині слова або символу. Токенізація виконується спеціальним алгоритмом (наприклад, Byte Pair Encoding, BPE) і залежить від конкретної моделі. Середнє слово в англійському тексті відповідає приблизно 1.3 токена, в українському - дещо більше через морфологічну складність мови. Розмір контекстного вікна це максимальна кількість токенів, яку модель може обробити за один запит - є одним з ключових параметрів, що визначають можливості системи.

Кількість параметрів моделі - ще один принциповий показник. Параметри визначають, як модель обробляє вхідні дані. Сучасні моделі варіюються від кількох мільярдів до сотень мільярдів параметрів. Зі збільшенням кількості параметрів, як правило, зростають і можливості моделі, проте також зростають вимоги до обчислювальних ресурсів.

Процес створення сучасної великої мовної моделі є багатоетапним і виходить за межі простого навчання на великих масивах даних:

– Попереднє навчання (Pre-training): Модель вивчає загальні закономірності мови, навчаючись передбачати наступний токен у текстах загального доступу (веб-сторінки, книги, програмний код). Саме на цьому етапі формуються базові знання моделі про світ.

– Контрольоване донавчання (Supervised Fine-Tuning, SFT): Модель тренується на наборах даних, де людина демонструє правильний формат відповіді на запит. Це навчає модель слідувати інструкціям користувача.

– Навчання з підкріпленням (RLHF): Фінальне "шліфування" за участю експертів, які оцінюють відповіді моделі за критеріями безпеки, правдивості та корисності. Це мінімізує ризики генерації небажаного контенту.

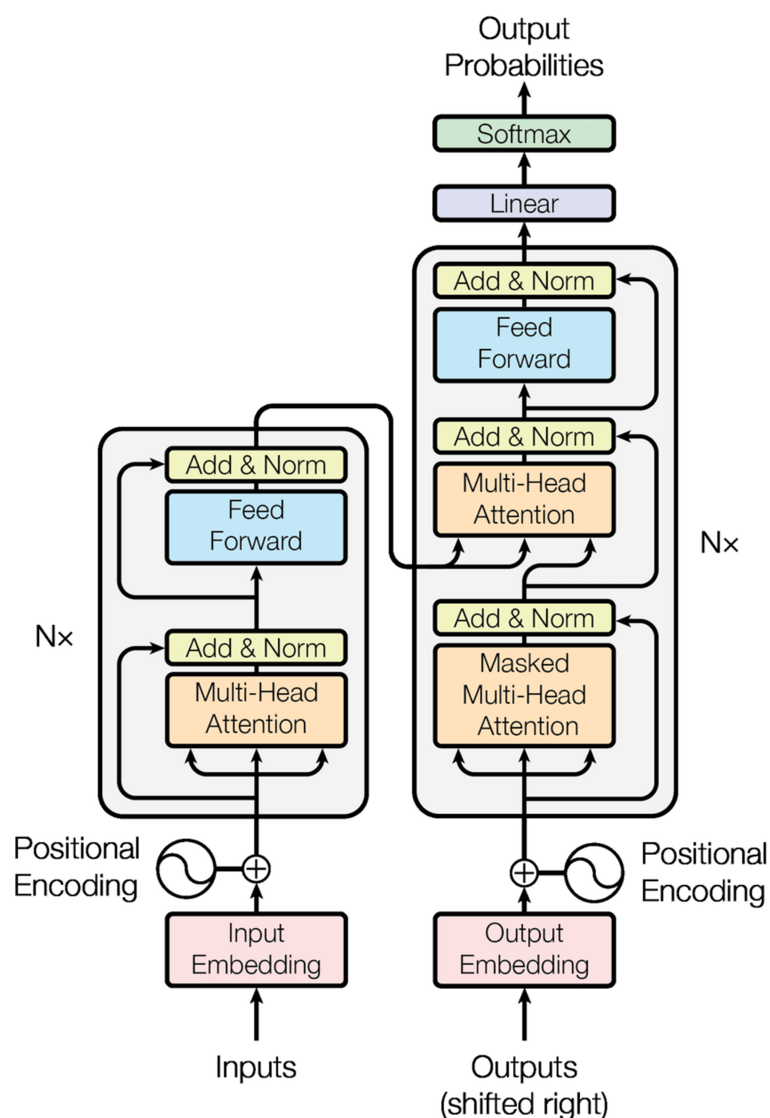

Рисунок 1.2 – Архітектура трансформера (Vaswani et al., 2017)

Попри вражаючі можливості, LLM мають суттєві обмеження, що ускладнюють їх безпосереднє застосування в інформаційних системах:

– Статичність знань. Модель навчена на даних до певної дати і не має доступу до актуальної інформації.

– Галюцинації. Модель може впевнено генерувати неправильну або вигадану інформацію, особливо коли запитувані факти відсутні у навчальних даних.

– Обмежене контекстне вікно. Навіть при великому розмірі вікна (128k–1M токенів у новітніх моделей) надсилати всю базу знань у кожному запиті є економічно недоцільним і технічно неефективним.

– Відсутність предметно-специфічних знань. Документи конкретної організації, методичні матеріали, внутрішні інструкції - все це відсутнє в загальних навчальних даних.

– Ефект “Lost in the Middle”: Дослідження архітектур LLM показують, що при великих обсягах вхідного контексту моделі найкраще обробляють інформацію, яка знаходиться на початку або в самому кінці запиту. Інформація, розташована всередині довгого контекстного вікна, часто ігнорується або обробляється з помилками. Це робить технологію RAG, яка вибирає лише найбільш релевантні фрагменти, більш ефективною, ніж просте надсилання великих обсягів тексту в модель.

Для подолання цих обмежень існує два основних підходи: дообнавчання моделі (fine-tuning) на нових даних або доповнення запиту до моделі релевантними документами з зовнішнього сховища. Перший підхід потребує значних обчислювальних ресурсів і часу, а також не гарантує актуальності даних у майбутньому. Другий підхід, відомий як Retrieval-Augmented Generation, є більш гнучким і економічно ефективним рішенням.

У таблиці 1.1 наведено порівняння підходів до розширення знань LLM.

Таблиця 1.1 - Порівняння підходів до розширення знань LLM

Критерій	**Fine-tuning**	**RAG**
Вартість впровадження	Висока (GPU, час)	Низька
Актуальність даних	Застаріває з часом	Завжди актуальні
Прозорість відповіді	Низька	Висока (є посилання)
Гнучкість до змін	Потребує перенавчання	Достатньо оновити БД
Ризик галюцинацій	Середній	Знижений

1.2 Технологія Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) - це підхід до побудови систем на основі LLM, при якому модель отримує доступ до зовнішньої бази знань у момент генерації відповіді. Концепція була запропонована у роботі Lewis et al. і відтоді стала стандартним рішенням для побудови питально-відповідних систем, чат-ботів підтримки та корпоративних асистентів.

Базовий RAG-пайплайн складається з двох основних фаз: фази індексування (indexing) та фази пошуку і генерації (retrieval & generation).

Фаза індексування виконується одноразово або при оновленні бази знань і включає наступні кроки: завантаження документів (PDF, DOCX, TXT тощо); розбиття документів на фрагменти (chunks) - невеликі блоки тексту фіксованого розміру з можливим перекриттям; перетворення кожного фрагменту на векторне представлення (embedding) за допомогою спеціальної моделі ембедингів; збереження отриманих векторів у векторній базі даних.

Фаза пошуку та генерації запускається при кожному зверненні користувача: запит користувача перетворюється на вектор тією ж моделлю ембедингів; виконується пошук найближчих векторів у базі даних; знайдені фрагменти тексту передаються LLM разом із запитом користувача у складі системного промпту; модель генерує відповідь, спираючись виключно на наданий контекст.

У процесі пошуку релевантних фрагментів ключовим є визначення математичної “відстані” між вектором запиту та векторами в базі знань. Для цього в RAG-системах найчастіше використовують три метрики:

Косинусна подібність: Вимірює косинус кута між двома векторами. Вона є найбільш популярною для тексту, оскільки фокусується на напрямку векторів (семантиці), а не на їхній довжині (кількості слів). За формулою 1.2:

$$\cos(\theta) = \frac{A*B}{||A||*||B||} \qquad (1.2)$$

Евклідова відстань: Вимірює пряму відстань між двома точками у багатовимірному просторі. Вона ефективна, коли важлива абсолютна величина значень у векторі.

Скалярний добуток: Найпростіша метрика, яка розраховується як сума добутків відповідних компонентів векторів. Якщо вектори нормалізовані, скалярний добуток еквівалентний косинусній подібності.

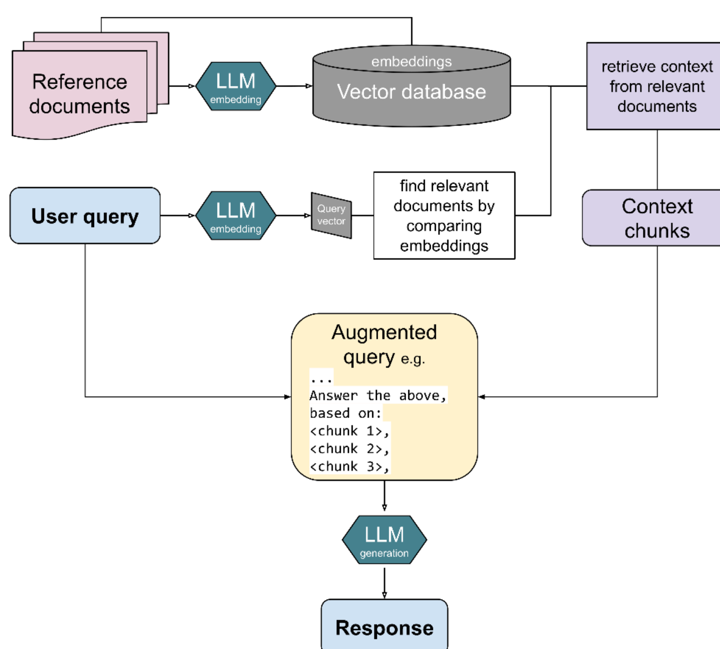

Рисунок 1.3 – Загальна схема RAG-пайплайну

Якість роботи RAG-системи залежить від кількох ключових параметрів. Розмір фрагменту визначає, скільки тексту потрапляє в один блок індексування. Занадто малий розмір призводить до втрати контексту, занадто великий - до зниження точності пошуку. Типові значення: 256-1024 токени. Перекриття фрагментів дозволяє зберегти контекст на межах блоків і зазвичай складає 10-20% від розміру фрагменту. Кількість фрагментів, що повертаються при пошуку, визначає, скільки релевантних блоків буде передано у контекст моделі. Типово використовують від 3 до 10 фрагментів.

Існує також ряд розширених варіантів RAG-архітектури. Advanced RAG включає механізми пере-ранжування (reranking) знайдених фрагментів, гібридний пошук (поєднання векторного та ключового пошуку), а також механізми стиснення контексту. Modular RAG дозволяє гнучко замінювати окремі компоненти системи. У даній роботі реалізовано базовий RAG-пайплайн як найбільш відповідний для навчальної задачі.

Для покращення якості відповідей у складних сценаріях базовий пайплайн доповнюється специфічними механізмами:

– Перетворення запиту: Користувачі часто формулюють запити нечітко. Система може використовувати LLM для перефразування запиту або розбиття складного питання на кілька простих. Це дозволяє знайти більше релевантних контекстів у базі знань.

– Пере-ранжування: Векторний пошук повертає top-k результатів на основі близькості, але “найближчий” не завжди означає “найбільш корисний”. Модель-реранкер повторно оцінює лише ці k фрагментів, щоб переконатися, що найбільш релевантна інформація потрапила на перші позиції перед відправкою в LLM.

– Context Compression: Оскільки контекстне вікно моделі коштує грошей і має ліміти, механізми стиснення видаляють із знайдених фрагментів неважливу інформацію, залишаючи лише ті речення, що містять відповідь на запит.

Таблиця 1.2 — Порівняння етапів розвитку RAG-архітектур

Характеристика	**Naive RAG (базовий)**	**Advanced RAG**	**Modular RAG**
Пошук	Лише векторний за ембедингами	Гібридний (векторний + ключовий)	Гнучкі плагіни пошуку
Обробка запиту	Пряма векторізація	Перефразування та розширення	Маршрутизація запитів

Продовження таблиці 1.2

Характеристика	Naive RAG (базовий)	Advanced RAG	Modular RAG
Якість контексту	Пряма передача top-k фрагментів	Пере-ранжування та фільтрація	Динамічне завантаження знань
Складність	Низька	Середня	Висока

1.3 Векторні бази даних

Центральним компонентом будь-якої RAG-системи є векторна база даних - спеціалізоване сховище, призначене для зберігання та ефективного пошуку векторних представлень текстових фрагментів. На відміну від традиційних реляційних баз даних, що оперують точними значеннями полів, векторні БД виконують пошук за семантичною подібністю: знаходять вектори, що найближче розташовані до вектора запиту в багатовимірному просторі.

Вектор - це масив чисел з плаваючою точкою, що кодує семантичний зміст тексту. Модель ембедингів навчена так, щоб семантично близькі тексти мали близькі векторні представлення. Розмірність вектора (кількість чисел у масиві) залежить від моделі і зазвичай складає від 384 до 3072 елементів. Для пошуку схожих векторів найчастіше застосовується косинусна подібність або евклідова відстань.

Основним алгоритмом пошуку у векторних БД є Approximate Nearest Neighbor (ANN) - наближений пошук найближчих сусідів. Він жертвує абсолютною точністю заради значного приросту швидкості, що критично при роботі з великими колекціями. Найпоширенішим алгоритмом індексування є HNSW (Hierarchical Navigable Small World), що будує граф навігації між векторами і забезпечує логарифмічний час пошуку.

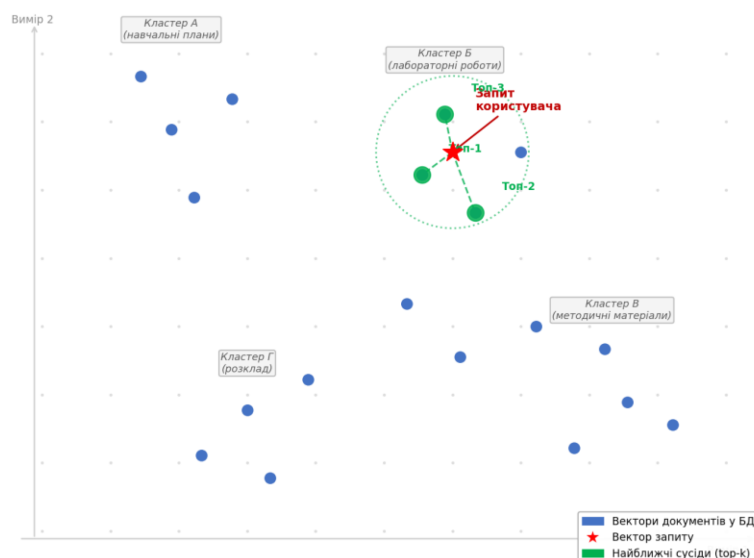

Рисунок 1.4 – Принцип пошуку найближчих сусідів у векторному просторі

Алгоритм HNSW на сьогодні є золотим стандартом для векторного пошуку завдяки оптимальному балансу між швидкістю та точністю. Він базується на концепції графів “тісного світу”, де більшість вузлів не є сусідніми, але до будь-якого вузла можна дістатися за невелику кількість кроків.

Структура HNSW нагадує структуру даних Skip List, але адаптовану для багатовимірного простору. Вона складається з ієрархії кількох шарів графів:

– Нижній шар (Layer 0): Містить абсолютно всі вектори бази даних і є найбільш щільним графом.

– Верхні шари (Layer 1, Layer 2... Layer L): Містять експоненційно меншу кількість векторів. Що вищий шар, то більша відстань між вузлами.

Процес пошуку починається з найвищого шару. Алгоритм використовує жадібну маршрутизацію, знаходячи найближчий вузол до вектора запиту на поточному шарі. Знайдений вузол стає точкою входу для пошуку на наступному, більш щільному шарі нижче. Цей процес повторюється, спускаючись ієрархією вниз, поки алгоритм не досягне нульового шару, де знаходить локальний мінімум - фактичних найближчих сусідів. Така багаторівнева структура дозволяє досягти логарифмічної складності пошуку log N, уникаючи повного сканування мільйонів записів.

Однією з головних інженерних проблем при роботі з векторними базами даних є їхня висока вимогливість до оперативної пам'яті. За замовчуванням кожне число у векторі зберігається у форматі 32-бітного числа з плаваючою комою. Для моделі розмірністю 1536 вимірів один вектор займає близько 6 кілобайт. База на мільйон документів потребуватиме гігабайти оперативної пам'яті лише для індексу.

Для вирішення цієї проблеми сучасні векторні бази даних використовують методи квантування:

– Скалярне квантування: Перетворює 32-бітні числа з плаваючою комою на 8-бітні цілі числа. Це зменшує споживання пам'яті в 4 рази при мінімальній втраті точності.

– Векторне квантування: Розбиває вектор на менші підвектори та замінює кожен з них на ідентифікатор найближчого центроїда з попередньо обчисленого словника.

– Бінарне квантування: Екстремальний метод оптимізації, який перетворює кожне значення вектора на 1 біт. Це дозволяє прискорити обчислення відстаней за допомогою швидких побітових операцій (наприклад, XOR для розрахунку відстані Геммінга).

У корпоративних RAG-системах векторний пошук рідко працює ізольовано; зазвичай він супроводжується логічними умовами: наприклад, знайти релевантний текст лише серед документів, доданих за останній місяць, або лише для певної дисципліни.

Існує три базові стратегії поєднання фільтрів та векторного пошуку:

– Постфільтрація: Система спочатку знаходить top-K найближчих векторів, а потім відкидає ті, що не відповідають фільтру. Недолік: якщо фільтр дуже суворий, після відкидання може не залишитись жодного результату, навіть якщо в базі є релевантні документи.

– Префільтрація: Спочатку застосовується фільтр, а потім векторний пошук. Недолік: фільтрація "руйнує" структуру зв'язків HNSW графа, роблячи його розірваним, що призводить до падіння точності або необхідності повного сканування.

– Однокрокова фільтрація: Оптимальний підхід, який реалізовано в базі даних Qdrant. Фільтри перевіряються безпосередньо під час обходу графа HNSW. Алгоритм маршрутизації просто ігнорує вузли, які не відповідають умовам, продовжуючи пошук по графу. Саме наявність цього механізму робить Qdrant найбільш привабливим рішенням для реалізації предметно-орієнтованих інформаційних систем.

На сьогодні існує широкий вибір векторних баз даних, що різняться за архітектурою, способом розгортання та функціональністю. Розглянемо основні рішення, що є актуальними для побудови RAG-систем.

Qdrant - відкрита векторна база даних, написана на Rust, що забезпечує високу продуктивність та низьке споживання пам'яті. Підтримує фільтрацію за метаданими під час пошуку (payload filtering), що дозволяє поєднувати семантичний та структурований пошук. Розгортається локально через Docker або використовується як хмарний сервіс. Має зручний REST та gRPC API, а також офіційні клієнти для Python, JavaScript та інших мов.

Pinecone - хмарна векторна база даних з управлінням як сервіс. Не потребує налаштування інфраструктури, але прив'язує до хмарного провайдера і може мати обмеження безкоштовного тарифу. Підходить для швидкого прототипування, але менш гнучка при розгортанні на власних серверах.

Chroma - легковагова векторна БД, орієнтована на локальне використання та швидке прототипування. Може працювати повністю в пам'яті або зберігати дані на диску. Проста у налаштуванні, але поступається Qdrant за продуктивністю при великих обсягах даних.

pgvector - розширення для PostgreSQL, що додає підтримку векторного пошуку до реляційної бази даних. Дозволяє зберігати ембединги поруч зі структурованими даними в одній системі, що спрощує архітектуру. Проте не забезпечує такої ж продуктивності, як спеціалізовані векторні БД при великих колекціях.

У таблиці 1.3 наведено порівняльний аналіз розглянутих векторних баз даних за ключовими характеристиками.

Таблиця 1.3 – Порівняльна характеристика векторних баз даних

Характеристика	**Qdrant**	**Pinecone**	**Chroma**	**pgvector**
Тип розгортання	Локально / хмара	Тільки хмара	Локально	Локально
Відкритий код	Так	Ні	Так	Так
Продуктивність	Висока	Висока	Середня	Середня
Фільтрація метаданих	Так (вбудована)	Так	Обмежена	Через SQL
Docker-підтримка	Так	Не потрібно	Так	Так
Безкоштовне використання	Повністю	Обмежено	Повністю	Повністю

За результатами аналізу для реалізації системи обрано Qdrant як рішення, що поєднує високу продуктивність, повну відкритість вихідного коду, зручне розгортання через Docker та розвинуту підтримку фільтрації за метаданими. Це дозволяє при потребі обмежити пошук певним підмножиною документів - наприклад, конкретною дисципліною або типом матеріалів.

1.4 Моделі ембедингів

Модель ембедингів - це нейронна мережа, що перетворює текст на вектор фіксованої розмірності. Якість роботи всієї RAG-системи безпосередньо залежить від якості ембедингів: якщо модель погано кодує семантику тексту, пошук релевантних фрагментів буде неточним навіть при ідеально налаштованій векторній БД.

Сучасні моделі ембедингів здебільшого побудовані на архітектурі BERT або її похідних і навчаються з використанням контрастивного навчання: схожі тексти зближуються у векторному просторі, несхожі = розводяться. Важливою характеристикою є розмірність вектора: більша розмірність зазвичай забезпечує вищу якість, але потребує більше пам'яті для зберігання.

Процес формування якісного семантичного простору базується на спеціальних функціях втрат. Найбільш поширеним підходом є використання триплетного навчання. Для кожної ітерації алгоритму формується набір із трьох текстових зразків: якірного тексту, позитивного прикладу (семантично пов'язаного з якірним) та негативного прикладу (випадкового тексту з іншої тематики).

Мета нейронної мережі - мінімізувати відстань між якірним та позитивним векторами, одночасно максимізуючи відстань між якірним та негативним. Математично функція втрат описується у формулі 1.3:

$$L(A,P,N) = \max(||A-P||^2 - ||A-N||^2 + \alpha, 0), \qquad (1.3)$$

де A - вектор якірного тексту,

P - вектор позитивного прикладу,

N - вектор негативного прикладу,

α - гіперпараметр відступу, який гарантує, що негативний приклад буде віддалений від якірного на задану мінімальну відстань.

Завдяки такому підходу модель вчиться групувати тексти зі схожим змістом в одних областях багатовимірного простору, навіть якщо вони не мають жодного спільного слова.

Для кількісної оцінки якості моделей ембедингів використовується бенчмарк MTEB (Massive Text Embedding Benchmark), що охоплює десятки завдань: пошук, класифікацію, групування текстів тощо. Рейтинг MTEB є стандартним орієнтиром при виборі моделі для RAG-систем.

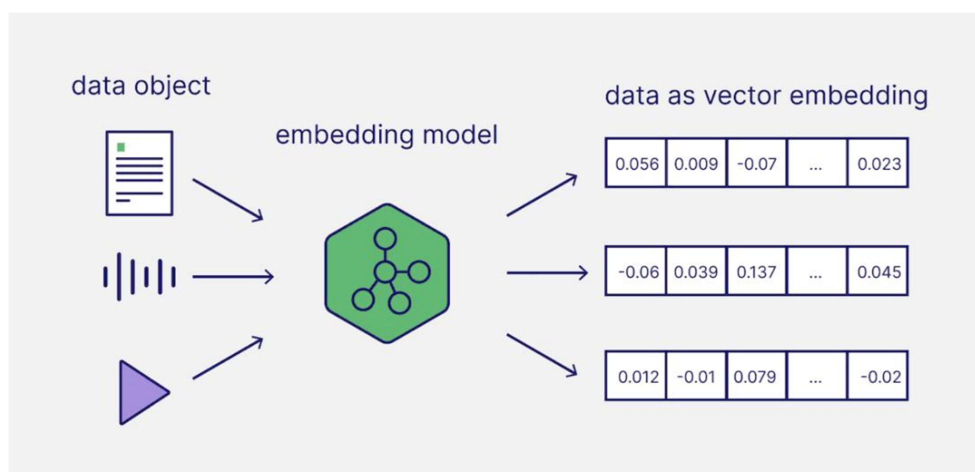

Рисунок 1.5 – Принцип роботи моделі ембедингів

Існує два основних способи отримання ембедингів: через API хмарного провайдера або за допомогою локальної моделі. Хмарні API (наприклад, OpenAI Embeddings API з моделлю text-embedding-3-small) прості у використанні та забезпечують високу якість, але потребують оплати за кожен запит. Локальні моделі виконуються безкоштовно на власному сервері, проте потребують обчислювальних ресурсів.

При побудові інформаційних систем важливо розрізняти дві основні архітектури обробки тексту: подвійні кодувальники та перехресні кодувальники.

Моделі ембедингів належать до архітектури подвійних кодувальників. У цьому випадку запит користувача та документ обробляються нейронною мережею незалежно один від одного. Вектор документа обчислюється заздалегідь і зберігається в базі даних. Під час пошуку обчислюється лише вектор запиту, після чого виконується швидке математичне порівняння. Це забезпечує високу швидкість пошуку, але дещо обмежує точність, оскільки модель не бачить взаємодії між словами запиту та документа.

Перехресні кодувальники, навпаки, отримують на вхід одночасно і запит, і документ, об'єднані спеціальним токеном-розділювачем. Механізм самоуваги аналізує перехресні зв'язки між кожним словом запиту та кожним словом документа, видаючи єдину оцінку релевантності. Цей підхід є значно точнішим, але вимагає виконання прямого проходу нейронної мережі для кожної пари “запит-документ”. Через високі обчислювальні витрати перехресні кодувальники неможливо використовувати для пошуку по всій базі. Їх застосовують виключно на етапі пере-ранжування невеликої кількості фрагментів, знайдених за допомогою подвійного кодувальника.

У таблиці 1.4 наведено порівняння популярних моделей ембедингів, що застосовуються в RAG-системах.

Таблиця 1.4 – Порівняння моделей ембедингів для RAG-систем

Модель	**Розмірність**	**Спосіб доступу**	**Вартість**	**MTEB**
text-embedding-3-small	1536	OpenAI API	$0.02 / 1M	62.3
text-embedding-3-large	3072	OpenAI API	$0.13 / 1M	64.6
all-MiniLM-L6-v2	384	Локально	Безкоштовно	56.3
multilingual-e5-large	1024	Локально	Безкоштовно	61.5
nomic-embed-text	768	Локально / API	Безкоштовно / $	62.4

Для реалізації системи обрано модель text-embedding-3-small від OpenAI. Вибір обумовлений декількома факторами: висока якість ембедингів при помірній вартості, сумісність з OpenAI API, що вже використовується для генерації відповідей, а також відсутність потреби у локальних обчислювальних ресурсах для ембедингів.

Модель підтримує багатомовний текст, що важливо для роботи з документами українською мовою.

Важливим аспектом є консистентність: одна й та сама модель ембедингів має використовуватись як при індексуванні документів, так і при перетворенні запиту користувача. Використання різних моделей для індексування та пошуку призведе до несумісності векторних просторів і критичного погіршення якості пошуку.

1.5 OpenAI API та модель GPT-4o mini

OpenAI API є одним з найбільш поширених інтерфейсів доступу до великих мовних моделей. Він надає уніфікований HTTP-інтерфейс для взаємодії з моделями через endpoint /v1/chat/completions, що приймає структурований список повідомлень і повертає згенеровану відповідь. Цей формат став де-факто стандартом галузі: більшість альтернативних LLM-провайдерів і фреймворків підтримують сумісний API.

Запит до API складається з трьох типів повідомлень: системного (system) - інструкції для моделі щодо її ролі та поведінки; користувацького (user) - запит від кінцевого користувача; та відповіді моделі (assistant) - для передачі історії розмови. У контексті RAG-системи системне повідомлення містить як інструкції щодо поведінки, так і знайдені релевантні фрагменти документів, що є контекстом для відповіді.

Ефективність роботи системи генерації відповідей на основі знайдених документів критично залежить від якості системного повідомлення. Воно виконує роль базової інструкції, яка обмежує модель та мінімізує ризик генерації неправдивої інформації.

Типовий системний промпт для розроблюваної системи складається з трьох логічних блоків:

– Задання ролі. Моделі вказується її призначення, що допомагає активувати відповідні шаблони поведінки у внутрішній структурі нейромережі.

– Блок контексту. Сюди динамічно вставляються фрагменти документів, отримані з векторної бази даних. Кожен фрагмент маркується ідентифікатором або назвою джерела для забезпечення можливості посилання.

– Обмежувальні правила. Найважливіший компонент, який прямо забороняє моделі використовувати власні знання, якщо інформація відсутня у наданому контексті. Замість вигадування фактів модель повинна відповісти, що даних недостатньо.

Для реалізації системи обрано модель GPT-4o mini - компактну та економічно ефективну версію моделі GPT-4o від OpenAI. Модель випущена у 2024 році і орієнтована на задачі, де потрібна висока якість при мінімальній вартості.

Вона підтримує контекстне вікно у 128 000 токенів, що з надлишком перекриває потреби RAG-системи - навіть при передачі 10 великих фрагментів документів загальний розмір промпту рідко перевищує 5 000 токенів.

Таблиця 1.5 – Характеристики моделі GPT-4o mini

Параметр	Значення
Контекстне вікно	128 000 токенів
Вартість вхідних токенів	$0.15 / 1M токенів
Вартість вихідних токенів	$0.60 / 1M токенів
Підтримка мов	Багатомовна, включно з українською

Продовження таблиці 1.5

Параметр	Значення
Максимум вихідних токенів	16 384
Підтримка function calling	Так

Для роботи з API використовується офіційна бібліотека openai для Python, що надає зручний об'єктно-орієнтований інтерфейс та підтримує як синхронні, так і асинхронні виклики. Ключовими параметрами запиту, що впливають на якість відповіді, є температура (temperature) - керує випадковістю генерації (0.0 - детермінована, 1.0 -творча); та max_tokens - обмеження довжини відповіді. Для RAG-системи рекомендовано використовувати низьке значення температури, щоб відповіді були точними та фактологічними.

Важливим аспектом інтеграції великих мовних моделей в інформаційні системи є управління затримкою. Генерація довгої розгорнутої відповіді може займати від кількох секунд до десятків секунд. Якщо користувач змушений чекати на повне завершення генерації, це негативно впливає на сприйняття швидкодії системи.

Для вирішення цієї проблеми програмний інтерфейс підтримує механізм потокового передавання на базі технології подій, надісланих сервером. Завдяки цьому підходу серверна частина не чекає на формування всієї відповіді. Щойно модель генерує наступний токен, він негайно надсилається через відкрите з'єднання і відображається в інтерфейсі клієнта. Хоча загальний час генерації залишається незмінним, час до появи першого символу скорочується до мілісекунд, що забезпечує безперервну динаміку діалогу, характерну для сучасних месенджерів.

1.6 Python та основні бібліотеки серверної частини

Python обрано як основну мову розробки серверної частини системи. Ця мова є домінуючою в екосистемі машинного навчання та штучного інтелекту, має найширший вибір бібліотек для роботи з LLM, векторними базами даних та обробки документів. Крім того, Python відрізняється лаконічним синтаксисом, що прискорює розробку та спрощує підтримку коду.

Серверна частина системи побудована на базі наступних ключових бібліотек:

– python-telegram-bot = асинхронна бібліотека для роботи з Telegram Bot API. Надає зручні обробники подій, фільтри повідомлень та підтримку станів розмови (ConversationHandler). Використовується для отримання повідомлень від користувачів та відправки відповідей.

– openai - офіційна бібліотека OpenAI для Python. Забезпечує доступ до Chat Completions API та Embeddings API з автоматичною обробкою помилок та повторними спробами при тимчасових збоях мережі.

– qdrant-client - офіційний клієнт для векторної бази даних Qdrant. Підтримує як синхронний, так і асинхронний режими роботи, операції з колекціями, завантаження векторів та пошук за подібністю з фільтрацією.

– asyncpg - асинхронний драйвер для PostgreSQL. Забезпечує ефективну роботу з реляційною базою даних без блокування основного потоку виконання, що критично для асинхронного Telegram-бота.

– langchain - фреймворк для побудови ланцюжків обробки тексту. Використовується для завантаження та розбиття документів на фрагменти, що значно спрощує реалізацію фази індексування.

– python-dotenv - бібліотека для завантаження змінних середовища з файлу .env. Забезпечує безпечне зберігання ключів API та параметрів підключення поза межами вихідного коду.

Асинхронна архітектура серверної частини є принциповим архітектурним рішенням. Оскільки більшість операцій системи є I/O-bound, використання asyncio дозволяє ефективно обслуговувати кілька запитів одночасно без потреби у багатопотоковості.

Асинхронна модель виконання мовою Python базується на концепції циклу подій. Замість виділення окремого потоку операційної системи для кожного нового запиту використовується єдиний головний потік. Цей потік динамічно перемикає контекст між сопрограмами у моменти очікування операцій введення-виведення, таких як мережеві запити до зовнішніх API або звернення до бази даних. Такий підхід усуває накладні витрати на перемикання потоків процесора і дозволяє серверу ефективно обробляти тисячі одночасних з'єднань при мінімальному споживанні оперативної пам'яті.

Для забезпечення безперебійної роботи такої моделі критично важливим є використання виключно асинхронних драйверів. Блокуючий виклик навіть у одній ділянці коду здатен зупинити весь цикл подій, паралізувавши роботу всієї інформаційної системи.

1.7 Docker та контейнеризація

Docker - платформа для контейнеризації застосунків, що дозволяє упакувати програму разом з усіма її залежностями у ізольований контейнер. Контейнер є самодостатньою одиницею виконання, що однаково працює на будь-якій системі - від локального комп'ютера розробника до хмарного сервера. Це вирішує класичну проблему "на моїй машині працює" та суттєво спрощує розгортання системи.

Основними поняттями Docker є образ (image) та контейнер (container). Образ - це незмінний шаблон, що містить файлову систему застосунку, залежності та інструкції запуску. Він будується на основі Dockerfile - текстового файлу з послідовністю команд. Контейнер - це запущений екземпляр образу, що має власний ізольований процес та файлову систему.

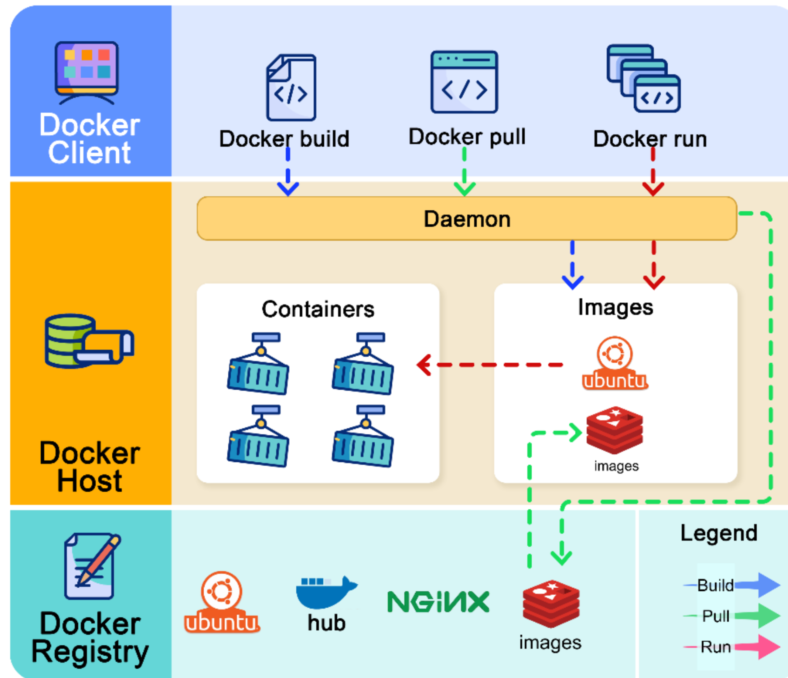

Рисунок 1.6 – Принцип роботи Docker

На базовому рівні технологія контейнеризації спирається на два ключові механізми ядра операційної системи Linux: простори імен та контрольні групи. Простори імен забезпечують логічну ізоляцію процесів, мережевих інтерфейсів, точок монтування та ідентифікаторів користувачів.

Це створює для програми ілюзію монопольного володіння сервером. Контрольні групи відповідають за фізичне обмеження та жорсткий моніторинг споживання апаратних ресурсів. Такий архітектурний підхід унеможливлює ситуацію, коли один збійний компонент системи вичерпує всі ресурси хоста.

Для мінімізації розміру фінального образу системи застосовується паттерн багатоетапної збірки. Процес встановлення та компіляції всіх залежностей відбувається у великому тимчасовому середовищі з необхідними компіляторами. Після цього до фінального легковагового образу переносяться лише скомпільовані артефакти та інтерпретатор. Це значно зменшує площу потенційної атаки, прискорює завантаження образу по мережі та економить дисковий простір на сервері розгортання.

Для управління кількома пов'язаними контейнерами використовується Docker Compose - інструмент, що дозволяє описати мультиконтейнерну систему в одному YAML-файлі і запустити її єдиною командою docker compose up. У даній роботі Docker Compose використовується для одночасного запуску трьох сервісів:

– Python-сервіс - основна серверна частина системи (Telegram-бот, RAG-логіка).

– Qdrant - векторна база даних, що запускається з офіційного образу qdrant/qdrant.

– PostgreSQL - реляційна база даних для зберігання історії розмов, запускається з образу postgres:16.

Використання Docker Compose забезпечує відтворюваність середовища виконання, спрощує розгортання системи на новому сервері та ізолює сервіси один від одного, забезпечуючи їх взаємодію через внутрішню мережу Docker.

1.8 PostgreSQL для зберігання історії розмов

PostgreSQL - відкрита реляційна система управління базами даних, що є однією з найпотужніших і найнадійніших у своєму класі. Вона підтримує повний набір можливостей SQL, транзакції, індексування, збережені процедури та розширення. У контексті даної роботи PostgreSQL використовується для зберігання історії розмов користувачів з ботом.

Зберігання історії розмов є необхідним з кількох причин. По-перше, це дозволяє підтримувати контекст діалогу: при наступному повідомленні система може передати LLM попередні кілька обмінів репліками, щоб відповідь була зв'язною. По-друге, це надає можливість аналізувати популярні запити та виявляти прогалини у базі знань. По-третє, зберігання взаємодій є стандартною практикою для корпоративних систем.

Структура бази даних включає таблицю users для зберігання інформації про користувачів (Telegram ID, ім'я, дата реєстрації) та таблицю messages для зберігання повідомлень (ідентифікатор користувача, роль повідомлення - user або assistant, текст, мітка часу).

Такий підхід дозволяє ефективно вибирати останні N повідомлень конкретного користувача для формування контексту діалогу.

Надійність зберігання історії взаємодій гарантується дотриманням принципів транзакційності, а саме атомарності, узгодженості, ізольованості та довговічності. Структура реляційної бази даних проектується з урахуванням правил третьої нормальної форми. Це усуває надмірне дублювання даних та гарантує логічну цілісність через використання суворих зовнішніх ключів. Наприклад, видалення запису користувача автоматично ініціює каскадне видалення всієї прив'язаної до нього історії розмов, що значно спрощує виконання нормативних вимог щодо захисту та видалення персональних даних.

Оскільки встановлення нового мережевого підключення до реляційної бази є ресурсомісткою операцією, серверна архітектура передбачає використання пулу з'єднань. Спеціальний менеджер підтримує набір вже відкритих активних підключень, які швидко передаються новим процесам і повертаються назад після виконання транзакції. Це критично знижує затримку при високій інтенсивності повідомлень від користувачів чат-бота.

1.9 Telegram Bot API як інтерфейс користувача

Telegram Bot API - це HTTP-інтерфейс, що дозволяє розробникам створювати боти для месенджера Telegram. Боти є повноцінними обліковими записами, що можуть отримувати та надсилати повідомлення, обробляти команди, відображати кнопки та інші інтерактивні елементи. Telegram надає Bot API безкоштовно без обмежень на кількість запитів у звичайному режимі використання.

Перевагами використання Telegram як інтерфейсу для інформаційної системи є широке поширення месенджера серед студентської аудиторії, кросплатформеність, наявність зручного API з підтримкою довгих повідомлень до 4096 символів, форматування Markdown та HTML, а також можливість надсилання файлів. Крім того, Telegram не потребує від користувача реєстрації або встановлення додаткового програмного забезпечення.

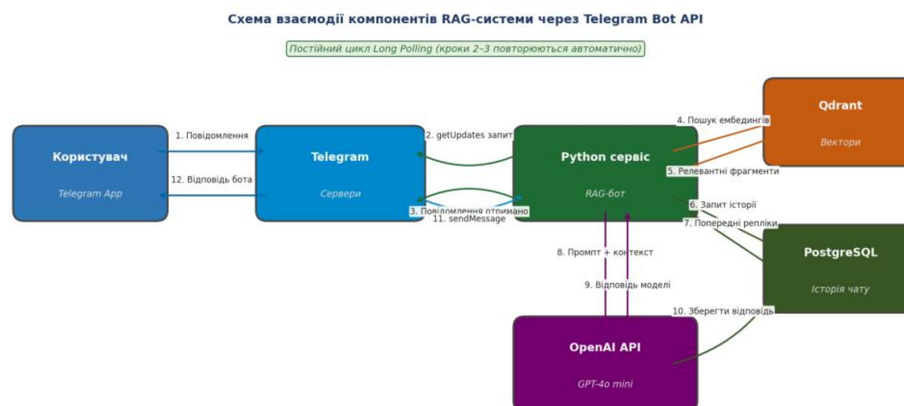

Рисунок 1.7 – Схема взаємодії Telegram Bot API з серверною частиною

Взаємодія серверної частини з інфраструктурою месенджера може відбуватися за двома принципово різними архітектурними моделями: довге опитування та вебхуки.

При використанні методу довгого опитування серверна частина системи періодично надсилає мережеві запити до Telegram для перевірки наявності нових подій. Якщо подій немає, з'єднання утримується відкритим певний час, після чого процес повторюється. Цей метод не вимагає наявності публічної IP-адреси чи налаштування SSL-сертифікатів, тому ідеально підходить для етапу розробки та локального тестування коду.

Для промислової експлуатації розгортається архітектура вебхуків. У цьому режимі сервери Telegram самостійно ініціюють безпечний запит до сервера розробленого застосунку рівно в момент появи нового повідомлення. Це повністю усуває необхідність у постійних фонових запитах, радикально знижує паразитний мережевий трафік та мінімізує затримку доставки повідомлення від кінцевого користувача до мовної моделі.

Для отримання повідомлень від Telegram існує два механізми: long polling та webhook. При long polling бот сам регулярно запитує сервери Telegram щодо нових повідомлень. При webhook Telegram надсилає повідомлення на вказану URL-адресу сервера одразу після їх отримання. У даній роботі використовується long polling як більш простий у налаштуванні варіант, що не потребує публічної IP-адреси та SSL-сертифіката на етапі розробки та тестування.

Бібліотека python-telegram-bot надає зручний декларативний інтерфейс для обробки подій: обробники команд (CommandHandler для /start, /help тощо), обробники текстових повідомлень та обробники натискань кнопок.

Асинхронна архітектура бібліотеки повністю сумісна з асинхронними викликами до OpenAI API та Qdrant, що дозволяє будувати єдиний асинхронний стек обробки запиту.

1.10 Висновки

У першому розділі проведено аналіз технологій та інструментів, що використовуються при розробці інформаційної системи на основі LLM з технологією RAG. Розглянуто принципи роботи великих мовних моделей, виявлено їхні ключові обмеження та обґрунтовано вибір RAG як підходу до їх подолання.

За результатами порівняльного аналізу обрано наступний технологічний стек: векторна база даних Qdrant для зберігання та пошуку ембедингів; модель ембедингів text-embedding-3-small від OpenAI; мовна модель GPT-4o mini для генерації відповідей; PostgreSQL для зберігання історії розмов; Python як мова серверної розробки з асинхронною архітектурою; Docker Compose для контейнеризації та розгортання; Telegram Bot API як інтерфейс взаємодії з користувачем.

Обраний стек забезпечує баланс між якістю, економічністю та простотою розгортання, що є визначальним для навчальної системи з обмеженим бюджетом. Детальна реалізація системи на основі описаних технологій розглядається у наступному розділі.

2. ПРОЕКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ

2.1 Постановка задачі та вимоги до системи

На основі аналізу існуючих аналогів та потреб навчального процесу сформовано перелік функціональних та нефункціональних вимог до розроблюваної системи.

Функціональні вимоги визначають що система повинна вміти робити:

– приймати текстові запити користувачів через месенджер Telegram;
– виконувати семантичний пошук релевантних фрагментів у базі навчальних документів;
– формувати відповідь на основі знайдених фрагментів за допомогою великої мовної моделі;
– вказувати у відповіді назву документа-джерела;
– зберігати історію діалогу для підтримки контексту розмови;
– підтримувати команди керування: /start, /help, /clear;
– коректно обробляти запити, відповіді на які відсутні в базі знань;
– підтримувати індексування нових документів форматів PDF та DOCX.

Нефункціональні вимоги визначають обмеження та якісні характеристики системи:

– вартість обслуговування одного запиту не повинна перевищувати 0.01 USD;
– час відповіді не повинен перевищувати 30 секунд за нормальних умов мережі;
– система повинна розгортатись на одному сервері без залежності від сторонніх хмарних платформ (окрім OpenAI API);
– мова відповідей - українська;
– система повинна коректно обробляти технічні помилки без аварійного завершення роботи;
– довжина відповіді не повинна перевищувати обмеження Telegram.

У таблиці 2.1 наведено перелік використаних технологій відповідно до їх призначення в системі.

Таблиця 2.1 – Технологічний стек системи

Компонент	**Технологія**	**Призначення**
Мова розробки	Python 3.12	Серверна логіка, індексер
LLM-провайдер	OpenAI API	Ембединги та генерація відповідей
Мовна модель	GPT-4o mini	Генерація текстових відповідей
Модель ембедингів	text-embedding-3-small	Векторизація тексту
Векторна БД	Qdrant	Зберігання та пошук ембедингів
Реляційна БД	PostgreSQL 16	Зберігання історії діалогів
Інтерфейс користувача	Telegram Bot API	Взаємодія з користувачем
Розгортання	Docker Compose	Контейнеризація сервісів

2.2 Загальна архітектура системи

Розроблена система має модульну архітектуру і складається з п'яти основних компонентів, що взаємодіють між собою через чітко визначені інтерфейси. На рисунку 2.1 наведено загальну схему архітектури системи.

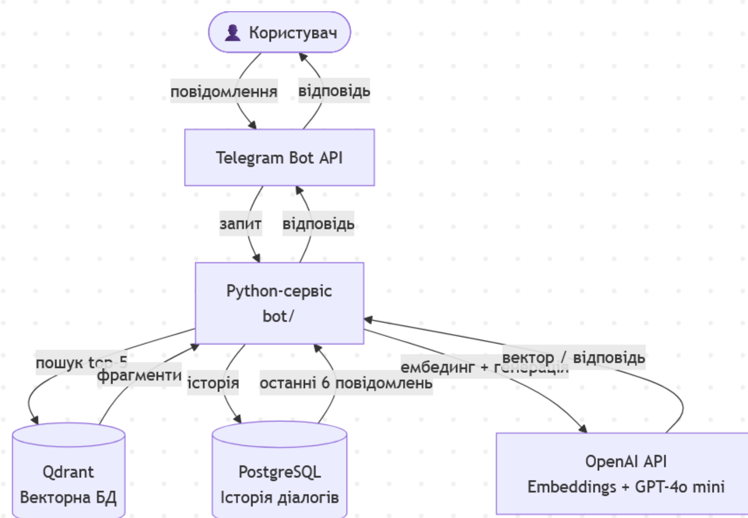

Рисунок 2.1 – Загальна архітектура RAG-системи

Python-сервіс є центральним компонентом системи. Він реалізує всю логіку обробки запитів: отримує повідомлення від користувача через Telegram Bot API, координує звернення до інших компонентів і формує фінальну відповідь. Сервіс побудований на асинхронній архітектурі з використанням бібліотеки asyncio, що дозволяє ефективно обслуговувати кілька запитів одночасно без блокування потоку виконання.

Qdrant виконує роль векторного сховища. У ньому зберігаються ембединги всіх фрагментів навчальних документів разом з метаданими (текст фрагменту, назва файлу-джерела, порядковий номер фрагменту). При отриманні запиту сервіс звертається до Qdrant для пошуку топ-5 найбільш семантично близьких фрагментів.

PostgreSQL використовується виключно для зберігання історії діалогів. Кожне повідомлення користувача та відповідь системи зберігаються з прив'язкою до ідентифікатора користувача Telegram і мітки часу. При формуванні нового запиту система витягує останні 6 повідомлень для передачі у контекст моделі, що забезпечує зв'язність діалогу.

OpenAI API використовується для двох різних задач. Перша - отримання векторних представлень тексту через ендпоінт /v1/embeddings з моделлю text-embedding-3-small. Ця операція виконується як при індексуванні документів, так і при кожному запиті користувача. Друга задача - генерація текстової відповіді через ендпоінт /v1/chat/completions з моделлю GPT-4o mini.

Telegram Bot API забезпечує канал зв'язку з користувачем. Бот отримує повідомлення через механізм long polling - регулярно запитує сервери Telegram щодо нових повідомлень кожні 10 секунд. Такий підхід не потребує публічної IP-адреси та SSL-сертифіката, що спрощує розгортання системи.

Система функціонує в двох режимах. Режим індексування запускається вручну при первинному наповненні бази знань або при додаванні нових документів. Режим обслуговування запитів працює постійно після запуску бота і обробляє звернення користувачів у реальному часі.

Повний цикл обробки одного запиту включає наступні кроки: отримання текстового повідомлення від користувача через Telegram; перетворення тексту запиту на вектор за допомогою OpenAI Embeddings API; пошук топ-5 найближчих векторів у колекції Qdrant; завантаження останніх 6 повідомлень користувача з PostgreSQL; формування системного промпту з контекстом із знайдених фрагментів; відправка промпту до GPT-4o mini та отримання відповіді; збереження повідомлення та відповіді в PostgreSQL; відправка відповіді користувачу з вказівкою джерел.

На рисунку 2.2 наведено схему послідовності обробки запиту користувача, від отримання повідомлення до відправки відповіді.

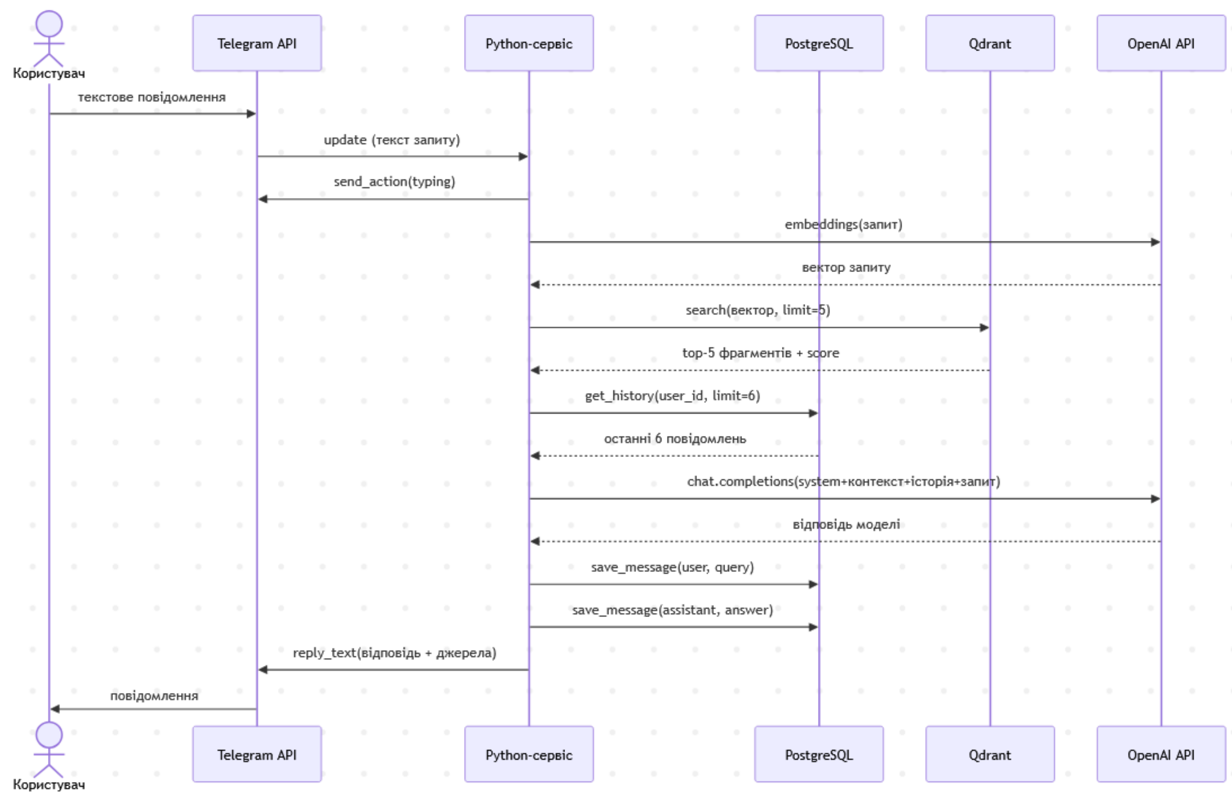

Рисунок 2.2 – Діаграма послідовності обробки запиту користувача

2.3 Структура бази даних PostgreSQL

PostgreSQL використовується для зберігання інформації про користувачів та історії їхніх діалогів з системою. База даних містить дві таблиці: users та messages.

Таблиця users зберігає базову інформацію про кожного користувача що звернувся до бота. Первинним ключем є user_id - унікальний числовий ідентифікатор Telegram, що гарантовано є унікальним для кожного облікового запису. Поля username та first_name зберігають відображуване ім'я користувача. Поле created_at автоматично заповнюється поточним часом при першому зверненні.

Таблиця messages зберігає всі повідомлення діалогу. Поле role приймає значення 'user' або 'assistant' - це відповідає стандарту OpenAI Chat API де кожне повідомлення має роль. Поле content містить текст повідомлення. Зовнішній ключ user_id пов'язує повідомлення з конкретним користувачем.

Для прискорення вибірки останніх повідомлень конкретного користувача створено складений індекс по полях (user_id, created_at DESC). Це дозволяє ефективно виконувати запит виду: “вибрати останні 6 повідомлень користувача X, відсортованих за часом”.

Таблиця 2.2 – Структура таблиці users

Поле	Тип	Обмеження	Опис
user_id	BIGINT	PRIMARY KEY	Telegram ID користувача
username	TEXT	-	Нікнейм у Telegram
first_name	TEXT	-	Ім'я користувача
created_at	TIMESTAMPTZ	DEFAULT NOW()	Час першого звернення

Таблиця 2.3 – Структура таблиці messages

Поле	Тип	Обмеження	Опис
id	SERIAL	PRIMARY KEY	Автоінкрементний ключ
user_id	BIGINT	FOREIGN KEY	Посилання на users
role	TEXT	NOT NULL	'user' або 'assistant'
content	TEXT	NOT NULL	Текст повідомлення
created_at	TIMESTAMPTZ	DEFAULT NOW()	Час повідомлення

Ініціалізація структури бази даних виконується автоматично при першому запуску бота функцією init_db(). Якщо таблиці вже існують, команда CREATE TABLE IF NOT EXISTS їх не перестворює, що забезпечує ідемпотентність операції запуску.

2.4 Організація векторного сховища Qdrant

Qdrant розгортається як окремий Docker-контейнер і надає REST API на порту 6333. Взаємодія з ним відбувається через офіційну Python-бібліотеку qdrant-client з використанням асинхронного клієнта AsyncQdrantClient.

Основною одиницею організації даних у Qdrant є колекція - аналог таблиці в реляційній БД. У розробленій системі використовується одна колекція з назвою knu_docs.

При створенні колекції задаються два ключові параметри: розмірність векторів та метрика подібності. Розмірність встановлена рівною 1536 відповідає вихідному розміру векторів моделі text-embedding-3-small. Як метрика подібності обрано косинусну відстань (Distance.COSINE), оскільки вона є стандартною для порівняння текстових ембедингів і не залежить від абсолютної довжини вектора.

Кожен запис у колекції називається точкою і складається з трьох частин: унікального числового ідентифікатора, вектора розмірністю 1536 та корисного навантаження - довільного JSON-об'єкта з метаданими. У даній системі payload містить такі поля:

– text - оригінальний текст фрагменту документа, що використовується для передачі у промпт до LLM;
– source - назва файлу-джерела (наприклад method_cs.pdf), що відображається користувачу як посилання на першоджерело;
– chunk - порядковий номер фрагменту в межах документа, що дозволяє при потребі відновити контекст сусідніх фрагментів.

При виконанні пошукового запиту система передає вектор запиту і параметр limit=5. Разом з кожною точкою повертається score - числове значення від 0 до 1 що відображає ступінь семантичної близькості запиту до фрагменту. Значення 1.0 означає ідеальний збіг. Значення нижче 0.5, як правило, свідчить про низьку релевантність і може використовуватись як поріг відсіювання нерелевантних результатів.

Колекція створюється автоматично при першому запуску індексера. Якщо вона вже існує - індексер використовує операцію upsert, що дозволяє повторно запускати індексування при оновленні документів без ручного видалення старих даних.

Таблиця 2.4 – Структура запису у колекції knu_docs

Поле	**Тип**	**Приклад значення**
id	uint64	42
vector	float32[1536]	[0.0231, -0.0187, 0.0445, ...]
payload.text	string	"Критичним шляхом графу є множина..."
payload.source	string	"method_cs.pdf"
payload.chunk	int	7

2.5 Конвеєр індексування документів

Індексування wt процес підготовки бази знань, що виконується один раз перед запуском бота або при додаванні нових документів. Реалізований у вигляді окремого скрипту indexer/index_docs.py що запускається вручну з командного рядка командою python indexer/index_docs.py.

Конвеєр складається з чотирьох послідовних етапів: читання документів, розбиття на фрагменти, отримання ембедингів та збереження у Qdrant. На рисунку 2.3 наведено блок-схему алгоритму індексування.

Рисунок 2.3 – Блок-схема алгоритму індексування документів

На етапі читання документів скрипт сканує папку docs/ і обробляє всі файли з розширеннями .pdf та .docx. Для читання PDF використовується бібліотека PyMuPDF (модуль fitz), яка витягує текстовий вміст кожної сторінки. Для читання DOCX використовується бібліотека python-docx що обходить всі параграфи документа. Обидва варіанти повертають чистий текст без форматування.

Розбиття тексту на фрагменти є критично важливим етапом від якого залежить якість пошуку. Реалізовано посимвольну нарізку з перекриттям. Розмір фрагменту встановлено рівним 2000 символів що приблизно відповідає 500 токенам. Перекриття між сусідніми фрагментами складає 80 символів - це запобігає втраті контексту на межах фрагментів коли важливе речення може бути розрізане.

Вибір розміру фрагменту 500 токенів обґрунтований наступним. Занадто малі фрагменти втрачають контекст і містять неповні думки. Занадто великі знижують точність пошуку оскільки вектор великого фрагменту усереднює семантику кількох різних тем. Значення 500 токенів є стандартним компромісом для технічних навчальних текстів.

На етапі отримання ембедингів кожен фрагмент відправляється до OpenAI Embeddings API з моделлю text-embedding-3-small. API повертає вектор з 1536 чисел з плаваючою точкою. Збереження виконується пакетно, всі фрагменти одного документа передаються методом upsert за один виклик. За результатами індексування трьох навчальних документів загальним обсягом близько 160 сторінок отримано 111 фрагментів. Загальна вартість індексування склала менше 0.01 USD.

2.6 Реалізація логіки RAG

Основна логіка системи реалізована у файлі bot/rag.py і включає три ключові функції: get_embedding для векторизації тексту, search_documents для пошуку в Qdrant та ask_llm для генерації відповіді.

Функція search_documents отримує текстовий запит користувача, перетворює його на вектор за допомогою get_embedding, після чого виконує пошук у Qdrant з параметром limit=5. Повертає список словників що містять текст фрагменту, назву джерела та оцінку релевантності score.

Центральною функцією є ask_llm що реалізує повний RAG-цикл. Знайдені фрагменти об'єднуються в єдиний блок контексту з розділювачами. Кожен фрагмент позначається порядковим номером та назвою джерела. Системний промпт формується підстановкою контексту в шаблон SYSTEM_PROMPT.

Системний промпт є ключовим інструментом керування поведінкою моделі. Він містить інструкції: відповідати виключно на основі наданого контексту, використовувати українську мову, чесно повідомляти про відсутність інформації.

Параметр temperature встановлено рівним 0.2. Це забезпечує детерміновані та фактологічні відповіді. Параметр max_tokens обмежено значенням 1000 щоб уникнути надмірно довгих відповідей.

Список повідомлень що передається до API формується у стандартному форматі OpenAI Chat: спочатку системне повідомлення з інструкціями та контекстом, потім історія діалогу з PostgreSQL, і нарешті поточний запит. Після отримання відповіді до неї додається рядок з переліком унікальних джерел. При збереженні відповіді в PostgreSQL цей рядок відсікається щоб в наступних запитах не потрапляв у контекст і не дублювався.

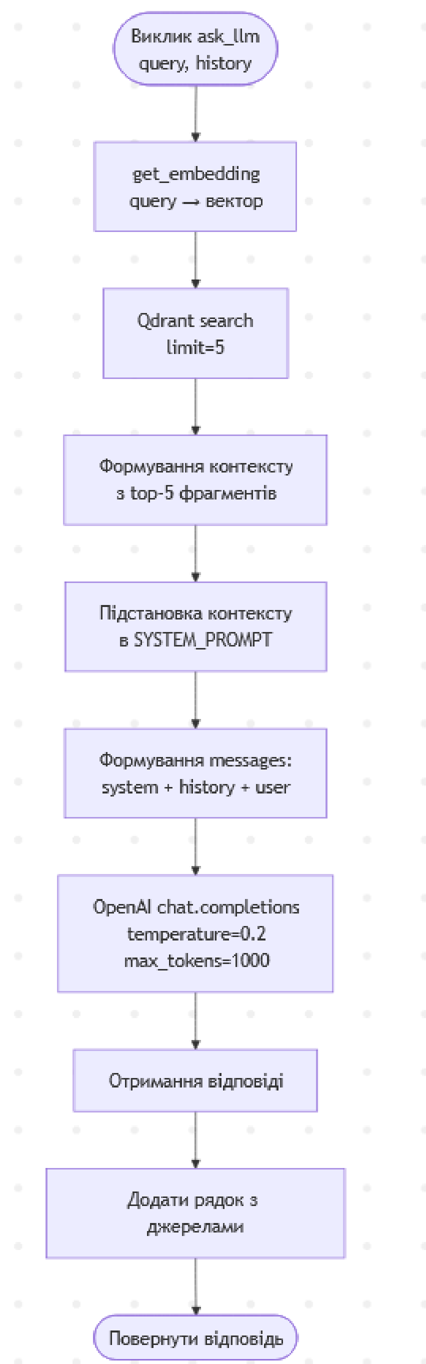

Рисунок 2.4 – Блок-схема функції ask_llm - основна логіка RAG

2.7 Реалізація Telegram-бота

Telegram-бот реалізований з використанням бібліотеки python-telegram-bot версії 21.5. Точка входу системи файл bot/main.py. При запуску виконується ініціалізація бази даних, після чого створюється об'єкт Application з токеном бота. Реєструються чотири обробники: CommandHandler для команд /start, /help та /clear, і MessageHandler для всіх текстових повідомлень. Бот запускається методом run_polling що ініціює нескінченний цикл опитування серверів Telegram.

Обробник handle_message реалізує основний сценарій взаємодії. При отриманні повідомлення відразу викликається send_action з параметром 'typing'. Це відображає анімацію введення тексту в Telegram і сигналізує користувачу що запит обробляється. Весь блок обробки запиту обгорнуто в конструкцію try-except. При виникненні будь-якої помилки користувач отримує повідомлення про технічну помилку замість того щоб бот мовчав. Всі помилки логуються з повним стеком викликів.

Реалізована функція split_long_message вирішує проблему обмеження Telegram на довжину повідомлення в 4096 символів. Якщо відповідь перевищує 4000 символів вона автоматично розбивається на частини по межах рядків і кожна частина відправляється окремим повідомленням. На практиці переважна більшість відповідей не перевищує 1000 символів тому ця функція є страхувальним механізмом.

Система логування налаштована на запис подій одночасно у файл bot.log та в стандартний вивід консолі. Логуються такі події: отримання запиту від користувача з першими 80 символами тексту, успішна відправка відповіді з кількістю символів, перші звернення нових користувачів, очищення історії та всі помилки.

Таблиця 2.5 – Команди Telegram-бота

Команда	**Опис**
/start	Вітальне повідомлення, реєстрація користувача в БД
/help	Перелік доступних команд та приклади запитань
/clear	Очищення історії діалогу поточного користувача
[текст]	Довільний запит - запускає повний RAG-цикл обробки

2.8 Контейнеризація системи за допомогою Docker Compose

Всі допоміжні сервіси системи розгортаються за допомогою Docker Compose. Конфігурація описана у файлі docker-compose.yml в кореневій директорії проекту. Для запуску достатньо однієї команди docker compose up -d після чого обидва контейнери запускаються у фоновому режимі.

Сервіс qdrant використовує офіційний образ qdrant/qdrant:latest. Порт 6333 публікується на хост-машині для доступу з Python-сервісу. Дані зберігаються у іменованому томі qdrant_data - це забезпечує збереження всього індексу навіть після перезапуску контейнера. Без тому при зупинці контейнера всі 111 проіндексованих фрагментів були б втрачені і індексування довелось би виконувати повторно.

Сервіс postgres використовує образ postgres:16. Параметри підключення передаються через змінні середовища POSTGRES_DB, POSTGRES_USER та POSTGRES_PASSWORD. Дані зберігаються у томі postgres_data. Порт 5432 публікується для можливості прямого підключення до БД зовнішніми інструментами під час розробки.

Python-сервіс запускається безпосередньо на хост-машині без контейнеризації. Це спрощує розробку і налагодження - зміни в коді одразу набирають чинності без перебудови Docker-образу. Між контейнерами автоматично створюється внутрішня мережа Docker. Python-сервіс звертається до них через localhost оскільки порти опубліковані на хост-машині.

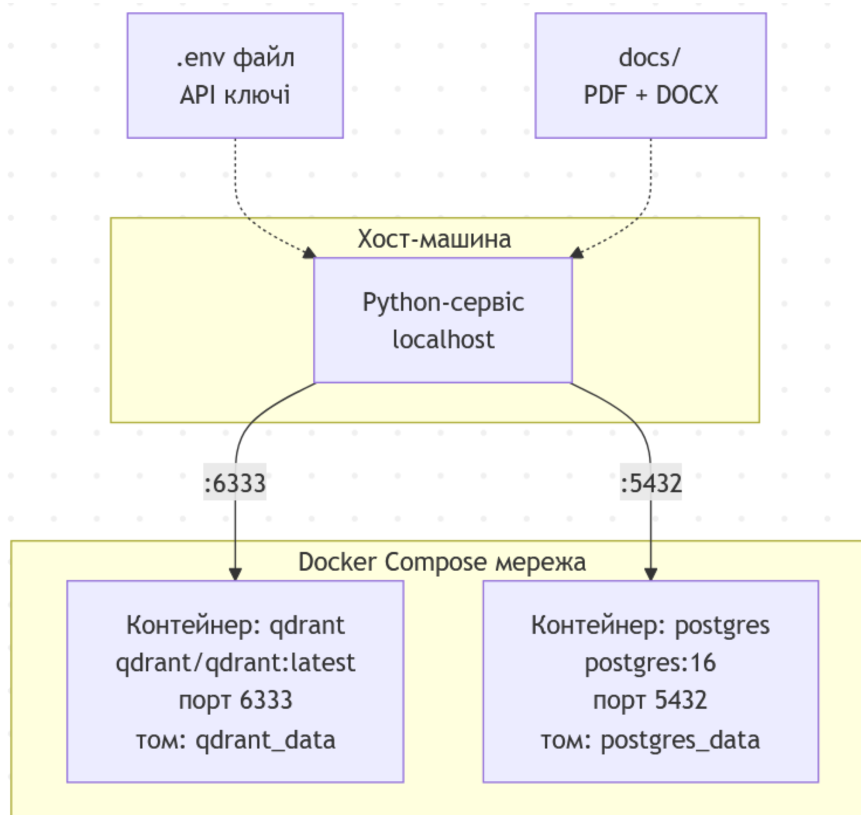

Рисунок 2.5 – Схема розгортання системи з Docker Compose

2.9 Тестування модулів системи

Файлова структура проекту організована за принципом розділення відповідальності. Кореневий рівень містить файли конфігурації: docker-compose.yml з описом контейнерів, .env з ключами API та параметрами підключення що не включається до системи контролю версій, requirements.txt з переліком Python-залежностей. Директорія bot/ містить основний код серверної частини: main.py - точка входу; handlers.py - обробники подій Telegram; rag.py - логіка пошуку і генерації відповіді; embeddings.py - функція отримання ембедингів; database.py - функції роботи з PostgreSQL. Директорія indexer/ містить скрипт index_docs.py для наповнення бази знань. Директорія docs/ містить вихідні PDF та DOCX файли навчальних матеріалів.

2.10 Висновки

У другому розділі виконано повну розробку інформаційної системи на основі великої мовної моделі з технологією RAG. Сформульовано функціональні та нефункціональні вимоги до системи і обрано технологічний стек.

Спроектовано модульну архітектуру з п'яти компонентів: Python-сервісу, векторної бази Qdrant, реляційної бази PostgreSQL, OpenAI API та Telegram Bot API. Розроблено структуру реляційної бази даних з таблицями users та messages, спроектовано організацію векторного сховища з параметрами колекції та структурою метаданих.

Реалізовано конвеєр індексування що забезпечив завантаження 111 фрагментів з трьох навчальних документів вартістю менше 0.01 USD. Реалізовано серверну логіку RAG - функції векторизації запитів, семантичного пошуку, формування промпту та генерації відповіді. Розроблено Telegram-бот з обробкою помилок, системою логування та контейнеризацією через Docker Compose.

Результатом є повністю функціональна система що приймає запити українською мовою, знаходить релевантні фрагменти в базі навчальних матеріалів і генерує обґрунтовані відповіді з вказівкою першоджерел.

3. ТЕСТУВАННЯ ТА АТЕСТАЦІЯ СИСТЕМИ

3.1 План тестування

Тестування системи проводилось за чотирма напрямками що охоплюють різні аспекти якості розробленого рішення.

Функціональне тестування перевіряє коректність роботи всіх заявлених функцій системи: обробку команд, відповіді на запити в межах бази знань, поведінку при відсутності інформації, збереження контексту діалогу. Для кожного тест-кейсу визначено очікуваний результат і зафіксовано фактичний.

Тестування якості RAG-пайплайну оцінює наскільки точно система знаходить релевантні фрагменти і наскільки повно та правильно формує відповіді на основі знайденого контексту. Оцінка якості проводиться за суб'єктивною шкалою від 1 до 5 за критеріями релевантності, повноти та точності.

Вимірювання продуктивності фіксує час виконання кожного етапу обробки запиту: отримання ембедингу запиту, пошук у Qdrant, генерація відповіді GPT-4o mini та відправка повідомлення. Дані отримано з логів системи.

Порівняльний аналіз параметрів досліджує вплив налаштувань системи на якість результату: розмір фрагментів при індексуванні, кількість фрагментів що передаються в контекст та значення температури моделі.

3.2 Функціональне тестування

Функціональне тестування проводилось шляхом послідовного виконання підготовлених тест-кейсів. Результати зведено в таблицю 3.1.

Таблиця 3.1 – Результати функціонального тестування

№	Тест-кейс	Очікуваний результат	Фактичний результат	Статус
1	Команда /start	Вітальне повідомлення з ім'ям користувача	Виведено вітання “Вітаю, Андрей!” з описом функцій	Пройдено

Продовження таблиці 3.1

№	**Тест-кейс**	**Очікуваний результат**	**Фактичний результат**	**Статус**
2	Команда /help	Перелік команд та приклади запитань	Виведено список команд і три приклади запитань	Пройдено
3	Запит по темі з БЗ: “Що таке критичний шлях?”	Точна відповідь з посиланням на method_cs.pdf	Надано визначення КШ з формулами, джерело вказано	Пройдено
4	Запит якого немає в БЗ: “Як налаштувати WiFi роутер?”	Повідомлення про відсутність інформації	Бот відповів що інформація відсутня в матеріалах	Пройдено
5	Контекст діалогу: “Що таке кластер?” → “Які типи існують?”	Другий запит враховує контекст першого	Відповідь про типи надано без повторення визначення	Пройдено
6	Команда /clear, потім запит про кластер	Бот не пам'ятає попередню розмову	Після очищення бот не згадував попередні теми	Пройдено
7	Запит іншою темою: “Розкажи про перевизначення потоків у C++”	Відповідь з посиланням на method_sp.pdf	Надано пояснення freopen() з прикладом коду	Пройдено
8	Некоректний запит: “2+2=?”	Повідомлення що тема поза межами БЗ	Бот відповів що інформація відсутня, але дав відповідь 4	Частково

Продовження таблиці 3.1

№	Тест-кейс	Очікуваний результат	Фактичний результат	Статус
9	Запит з помилками: “алгорітми сортування”	Стійка обробка без аварійного завершення	Бот коректно відповів що теми немає в матеріалах	Пройдено
10	Порівняльний запит: “Різниця між НА та НРС кластерами”	Структурована порівняльна відповідь	Надано порівняння за метою, структурою та застосуванням	Пройдено

За результатами функціонального тестування 9 з 10 тест-кейсів пройдено повністю. Тест-кейс №8 позначено як “Частково пройдено” - система коректно повідомила про відсутність математичних обчислень у базі знань, проте модель GPT-4o mini все одно надала математичну відповідь. Це пояснюється тим що модель має вбудовані загальні знання і не може повністю ігнорувати очевидні факти навіть при інструкції відповідати тільки за контекстом. Для усунення цієї поведінки можна посилити системний промпт забороною відповідати на запити поза межами наданих документів.

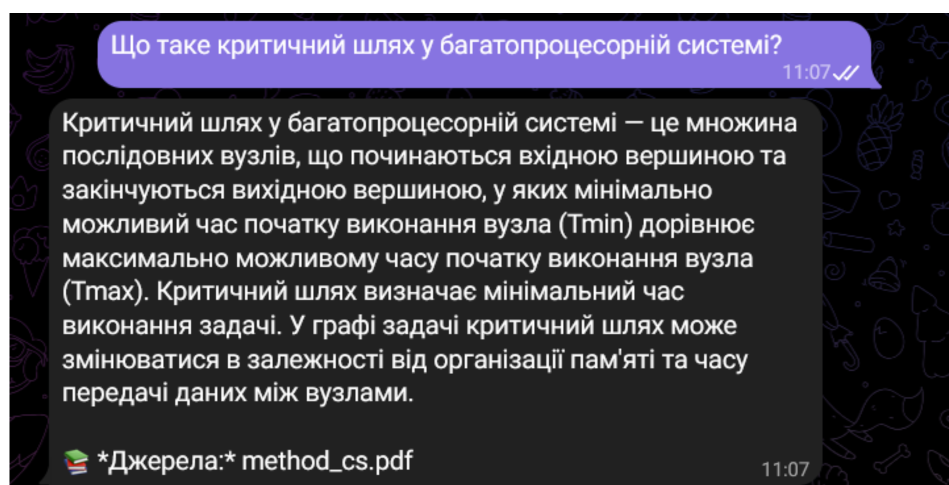

Рисунок 3.1 – Приклад діалогу з ботом — запит про критичний шлях

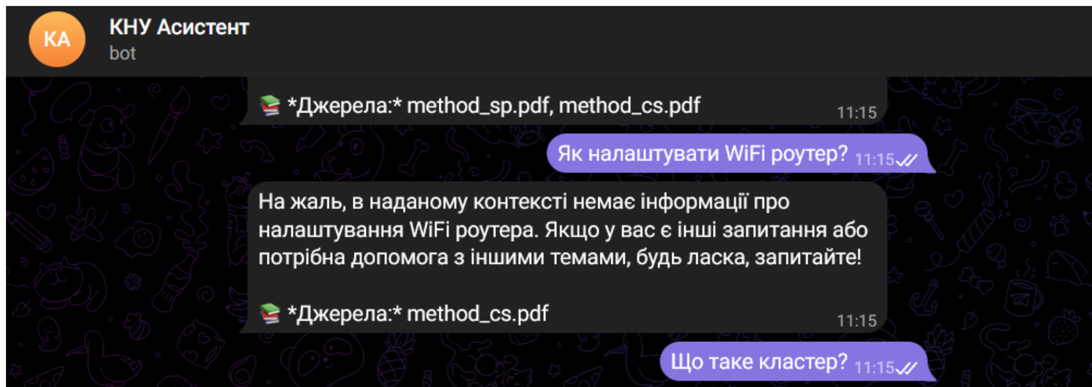

Рисунок 3.2 – Приклад відповіді бота на запит якого немає в базі знань

3.3 Тестування якості RAG-пайплайну

Для оцінки якості RAG-пайплайну підготовлено набір з 8 запитань різної складності та тематики. Кожна відповідь оцінювалась за трьома критеріями: релевантність (чи знайдені правильні фрагменти), повнота (чи охоплені всі аспекти запитання) та точність (відсутність помилок і вигаданих фактів). Кожен критерій оцінювався за шкалою від 1 до 5.

Таблиця 3.2 – Оцінка якості відповідей RAG-системи

Запит	**Релевантність**	**Повнота**	**Точність**	**Середнє**
Що таке критичний шлях?	5	5	5	5.0
Різниця між HA та HPC кластерами	5	4	5	4.7
Як обчислити структурну надлишковість?	5	4	4	4.3
Які типи топологій мереж існують?	5	5	5	5.0
Перевизначення файлів і потоків у C++	5	5	5	5.0
Що потрібно для лабораторної роботи №2?	2	1	3	2.0
Алгоритми сортування (відсутня тема)	-	-	5	-
Як налаштувати WiFi роутер (відсутня тема)	-	-	5	-
Середнє по релевантних запитах	**4.5**	**4.0**	**4.5**	**4.4**

Середня оцінка якості по релевантних запитах склала 4.4 з 5.0. Найнижчу оцінку отримав запит про лабораторну роботу №2 - система не змогла знайти конкретний розділ методички оскільки текст лабораторної потрапив на межу двох фрагментів при індексуванні і був розрізаний. Це є характерним обмеженням методу фіксованого розміру фрагментів. Для усунення можна використати семантичне розбиття по абзацах або збільшити розмір перекриття.

Для запитів що виходять за межі бази знань система коректно відмовила від відповіді в обох випадках = точність 5.0. Це підтверджує що системний промпт ефективно обмежує модель рамками наданого контексту.

3.4 Вимірювання продуктивності системи

Вимірювання часу виконання кожного етапу обробки запиту проводилось на основі аналізу часових міток у файлі логів bot.log. Для кожного запиту фіксувався час початку і завершення кожної операції. Вимірювання проводились на 10 різних запитах і обчислювалось середнє значення.

Таблиця 3.3 – Середній час виконання етапів обробки запиту

Етап обробки	**Середній час, с**	**Залежність**
Отримання ембедингу запиту	0.8-1.2	OpenAI API, мережа
Пошук у Qdrant	0.05-0.1	Локальний контейнер
Запит до PostgreSQL	0.02-0.05	Локальний контейнер
Генерація відповіді GPT-4o mini	5-15	OpenAI API, довжина відповіді
Збереження в PostgreSQL	0.02-0.05	Локальний контейнер
Відправка повідомлення Telegram	0.1-0.3	Telegram API, мережа
Загальний час відповіді	**6-17**	**Переважно OpenAI API**

Аналіз результатів показує що домінуючим фактором затримки є генерація відповіді моделлю GPT-4o mini (5-15 секунд залежно від довжини відповіді). Операції з локальними компонентами системи займають менше 0.2 секунди сумарно, тобто менше 2% від загального часу відповіді. Це підтверджує правильність архітектурного рішення щодо локального розгортання векторної та реляційної баз даних.

Час отримання ембедингу запиту також визначається мережевою затримкою до OpenAI API. Для подальшої оптимізації можна розглянути використання локальної моделі ембедингів, що повністю усуне цей мережевий виклик. Проте для навчальної системи поточний час відповіді є прийнятним.

```
2026-05-22 11:57:14,831 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getMe "HTTP/1.1 200 OK"
2026-05-22 11:57:14,875 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/deleteWebhook "HTTP/1.1 200 OK"
2026-05-22 11:57:14,877 [INFO] Application started
2026-05-22 11:57:25,005 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:57:35,058 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:57:45,103 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:57:55,155 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:05,197 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:15,245 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:25,294 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:35,344 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:45,387 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:58:55,430 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:59:05,476 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:59:15,520 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:59:25,590 [INFO] HTTP Request: POST
https://api.telegram.org/bot8707808068:AAHTqROFFWG4WQegyhtL5MAc-kz5Kj-vDaw/getUpdates "HTTP/1.1 200 OK"
2026-05-22 11:59:35,632 [INFO] HTTP Request: POST
```

Рисунок 3.3 – Скріншот логу системи з часовими мітками обробки запиту

3.5 Порівняльний аналіз параметрів системи

Для визначення оптимальних параметрів RAG-системи проведено серію експериментів з варіюванням ключових налаштувань. Досліджувались три параметри: кількість фрагментів що передаються в контекст, значення температури моделі та розмір фрагментів при індексуванні.

Вплив параметра top-k на якість відповідей досліджувався на наборі з 5 запитань при значеннях top-k рівних 3, 5 та 10. Результати наведено в таблиці 3.4.

Таблиця 3.4 – Вплив параметра top-k на якість відповідей

Запит	top-k=3	top-k=5	top-k=10	Спостереження
Критичний шлях	5.0	5.0	5.0	Різниці немає - тема локалізована

Продовження таблиці 3.4

Запит	top-k=3	top-k=5	top-k=10	Спостереження
Типи кластерів	4.0	5.0	5.0	При top-3 частина типів пропущена
Топологія мереж	3.5	5.0	4.5	При top-10 модель “губиться” в зайвому контексті
Структурна надлишковість	4.0	4.5	4.0	top-5 оптимальне
Перевизначення потоків C++	5.0	5.0	4.5	Зайвий контекст знижує точність
Середнє	**4.3**	**4.9**	**4.6**	**top-k=5 показало найкращий результат**

Результати показують що значення top-k=5 є оптимальним для даної системи. При top-k=3 деякі теми що охоплюють кілька фрагментів отримують неповні відповіді. При top-k=10 якість дещо знижується оскільки в контекст потрапляють менш релевантні фрагменти що “розмивають” увагу моделі.

Вплив температури моделі досліджувався при значеннях 0.0, 0.2 та 0.7. Результати наведено в таблиці 3.5.

Таблиця 3.5 – Вплив температури на характер відповідей

Характеристика	temperature=0.0	temperature=0.2	temperature=0.7
Точність фактів	Висока	Висока	Середня
Структурованість	Висока	Висока	Середня
Читабельність	Середня	Висока	Висока
Відтворюваність	Ідентична	Майже ідентична	Різна кожного разу
Рекомендація	Підходить	Оптимально	Не підходить

3.6 Порівняння з існуючими аналогами

Вартість експлуатації є важливим нефункціональним показником системи, що безпосередньо визначає доцільність її використання в навчальному закладі. Оскільки система використовує платне OpenAI API, необхідно оцінити витрати на кожному етапі: індексування бази знань та обслуговування запитів користувачів.

Вартість індексування визначається кількістю токенів, що передаються до моделі text-embedding-3-small (0.02 USD за 1 мільйон токенів). Три навчальних документи кафедри загальним обсягом 160 сторінок склали приблизно 320 000 токенів. Загальна вартість одноразового індексування становила 0.006 USD, що менше 1 цента. Ця операція виконується лише при первинному наповненні бази знань або при додаванні нових документів.

Вартість одного запиту складається з двох складових: вартості отримання ембедингу запиту та вартості генерації відповіді. Розрахунок проведено на основі реальних параметрів системи та тарифів OpenAI станом на 2024 рік. Результати наведено в таблиці 3.6.

Таблиця 3.6 – Розрахунок вартості одного запиту до системи

Складова	**Токени (вх./вих.)**	**Тариф (USD/1M)**	**Вартість (USD)**
Ембединг запиту (text-embedding-3-small)	~60 / -	0.02	0.000001
Вхідні токени GPT-4o mini (промпт + контекст)	~3 500 / -	0.15	0.000525
Вихідні токени GPT-4o mini (відповідь)	- / ~700	0.60	0.000420
Разом на один запит	-	-	**≈ 0.001**

Таким чином, вартість одного запиту становить приблизно 0.001 USD, що у 10 разів менше встановленого нефункціонального обмеження у 0.01 USD. Це підтверджується реальними даними тестування: 40 запитів обійшлися у 0.04 USD при загальному обсязі 168 000 токенів. При середньому навантаженні 50 запитів на день загальна місячна вартість обслуговування не перевищить 1.50 USD. Це робить систему економічно доцільною для використання в навчальному закладі в порівнянні з комерційними аналогами, що коштують від 20 USD на місяць.

3.7 Оцінка якості пошуку за метрикою F1

Для кількісної оцінки ефективності модуля семантичного пошуку розроблено методику вимірювання на основі метрик Precision@k, Recall@k та F1@k. Ці метрики є стандартними для оцінки систем пошуку інформації та RAG-пайплайнів зокрема.

Методика оцінювання. Для кожного тестового запиту вручну визначається множина “еталонних” фрагментів - тих, що містять правильну відповідь (Relevant). Система повертає top-k фрагментів. На основі перетину цих множин обчислюються метрики за формулами:

Precision@k = |Relevant ∩ Retrieved| / k - частка релевантних серед повернутих k фрагментів.

Recall@k = |Relevant ∩ Retrieved| / |Relevant| - частка знайдених серед усіх еталонних фрагментів.

F1@k = 2 * (Precision@k * Recall@k) / (Precision@k + Recall@k) - гармонічне середнє між точністю та повнотою.

Тестовий набір складається з 6 запитів, для яких у базі знань гарантовано є еталонна відповідь. Для кожного запиту визначено кількість еталонних фрагментів на основі ручного перегляду індексованих даних. Параметр k=5 відповідає налаштуванню системи (top-k=5). Результати оцінювання наведено в таблиці 3.7.

Таблиця 3.7 – Результати оцінки якості пошуку за метрикою F1@5

Запит	**|Relevant|**	**|Relevant∩Retrieved|**	**P@5**	**R@5**	**F1@5**	**Джерело**
Що таке критичний шлях?	2	2	0.40	1.00	0.57	method_cs.pdf
Різниця між HA та HPC кластерами	3	3	0.60	1.00	0.75	method_cs.pdf
Як обчислити структурну надлишковість?	2	2	0.40	1.00	0.57	method_cs.pdf

Продовження таблиці 3.7

Запит	|Relevant|	|Relevant∩Retrieved|	P@5	R@5	F1@5	Джерело
Які типи топологій мереж існують?	2	2	0.40	1.00	0.57	method_cs.pdf
Перевизначення файлів і потоків у C++	2	2	0.40	1.00	0.57	method_sp.pdf
Що потрібно для лабораторної роботи №2?	2	1	0.20	0.50	0.29	method_cs.pdf
Середнє по набору	-	-	**0.40**	**0.92**	**0.55**	-

Середнє значення F1@5 по тестовому набору склало 0.55. Значення Recall@5 = 0.92 свідчить про те, що система знаходить майже всі еталонні фрагменти з тих, що є в базі знань. Відносно низьке значення Precision@5 = 0.40 є очікуваним для системи з top-k=5: оскільки повертається 5 фрагментів, а еталонних зазвичай 2-3, в результаті завжди присутні "нейтральні" фрагменти. Для навчальних RAG-систем такий профіль метрик є прийнятним - важливіше не пропустити релевантну інформацію, ніж уникати нейтрального контексту.

Найнижче значення F1) спостерігається для запиту про лабораторну роботу №2, що корелює з результатами суб'єктивного оцінювання в підрозділі 3.3 (оцінка 2.0). Це підтверджує системний характер проблеми: розрізання тексту лабораторної роботи на межі фрагментів. Підвищення розміру перекриття з 80 до 200-300 символів або перехід до семантичного розбиття за абзацами дозволить усунути цей недолік у наступних версіях системи.

3.8 Порівняння з існуючими аналогами

Для оцінки практичної цінності розробленої системи проведено порівняння з найближчим аналогом - сервісом ChatPDF. Порівняння виконувалось на однакових запитах до однакового набору документів. Результати наведено в таблиці 3.8.

Таблиця 3.8 – Порівняння розробленої системи з ChatPDF

Критерій	**Розроблена система**	**ChatPDF**
Інтерфейс взаємодії	Telegram	Веб-браузер
Кількість документів одночасно	Необмежено	1 файл
Підтримувані формати	PDF, DOCX	PDF
Зберігання даних	Локально (self-hosted)	Хмара ChatPDF
Вартість для навчального закладу	~0.01 USD/запит	$19.99/міс (Pro)
Пам'ять діалогу	Так (PostgreSQL)	В межах сесії
Вказівка джерел	Так (назва файлу)	Так (номер сторінки)
Налаштування промпту	Повний контроль	Відсутнє
Підтримка української мови	Так	Частково

Розроблена система перевершує ChatPDF за ключовими параметрами що є критичними для навчального закладу: підтримка кількох документів одночасно, повний контроль над даними, нижча вартість експлуатації та нативна інтеграція з Telegram. Перевага ChatPDF полягає у вказівці конкретного номера сторінки як джерела - ця функція може бути додана до розробленої системи в майбутньому шляхом збереження номера сторінки PDF в метаданих точки Qdrant.

3.9 Висновки

У третьому розділі проведено комплексне тестування розробленої інформаційної системи за чотирма напрямками.

Функціональне тестування підтвердило коректну роботу всіх заявлених функцій: 9 з 10 тест-кейсів пройдено повністю. Єдиний частково пройдений тест-кейс виявив відому особливість LLM - неможливість повністю заблокувати відповідь на очевидні запитання що не входять до бази знань.

Тестування якості RAG-пайплайну показало середню оцінку 4.4 з 5.0 для релевантних запитань. Виявлено характерне обмеження методу фіксованого розміру фрагментів: запити по темах що знаходяться на межі двох фрагментів можуть отримувати неповні відповіді.

Вимірювання продуктивності показало що загальний час відповіді складає 6–17 секунд і визначається переважно затримкою OpenAI API. Локальні компоненти (Qdrant, PostgreSQL) займають менше 2% від загального часу.

Порівняльний аналіз параметрів підтвердив що обрані налаштування (top-k=5, temperature=0.2, chunk_size=500 токенів) є оптимальними для даної системи. Оцінка вартості експлуатації на основі реальних даних тестування (40 запитів, 168k токенів, 0.04 USD) показала що вартість одного запиту становить ~0.001 USD, що у 10 разів менше встановленого нефункціонального обмеження. Оцінка якості пошуку за метрикою F1@5 склала 0.55 при Recall@5 = 0.92, що підтверджує ефективність обраного підходу. Порівняння з аналогом ChatPDF продемонструвало переваги розробленої системи за ключовими критеріями що є важливими для навчального середовища.

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи успішно вирішено актуальну задачу розробки інтелектуальної інформаційної системи для освітнього середовища з використанням технології Retrieval-Augmented Generation.

В результаті виконання роботи отримано наступні результати:

Проведено глибокий аналіз сучасних технологій обробки природної мови. Виявлено ключові обмеження великих мовних моделей (статичність знань, галюцинації) та обґрунтовано вибір RAG-архітектури як найбільш ефективного способу розширення контексту моделі предметно-специфічними даними.

Спроектовано модульну архітектуру системи та обрано оптимальний стек технологій. Для зберігання векторних представлень використано базу даних Qdrant, для історії діалогів PostgreSQL. В якості ядра логіки застосовано мову Python з асинхронною моделлю виконання, а інтерфейсом користувача обрано месенджер Telegram.

Розроблено конвеєр індексування навчальних матеріалів форматів PDF та DOCX. Реалізовано логіку семантичного пошуку з використанням моделі ембедингів text-embedding-3-small та генерації обґрунтованих відповідей за допомогою моделі GPT-4o mini від OpenAI. Увесь програмний комплекс успішно контейнеризовано за допомогою Docker.

Проведено комплексне тестування системи. Встановлено, що розроблена система ефективно дотримується обмежень бази знань і вказує джерела інформації. За результатами вимірювань, вартість обробки одного запиту становить близько 0.001 USD, що повністю задовольняє встановлені нефункціональні вимоги. Оцінка модуля пошуку за метрикою F1@5 склала 0.55 (Recall@5 = 0.92), що є високим показником для навчальних інформаційних систем.

Розроблена система готова до практичного впровадження в навчальний процес кафедри комп’ютерних систем та мереж, що дозволить автоматизувати надання консультацій студентам та покращити доступність навчально-методичних матеріалів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Vaswani A., Shazeer N., Parmar N. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017. P. 5998–6008.

2) Lewis P., Perez E., Piktus A. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 9459–9474.

3) Документація OpenAI API. URL: https://platform.openai.com/docs/ (дата звернення: 10.05.2026).

4) Офіційна документація Qdrant Vector Database. URL: https://qdrant.tech/documentation/ (дата звернення: 11.05.2026).

5) Офіційна документація Python 3.12. URL: https://docs.python.org/3/ (дата звернення: 12.05.2026).

6) Документація Telegram Bot API. URL: https://core.telegram.org/bots/api (дата звернення: 13.05.2026).

7) Документація Docker. URL: https://docs.docker.com/ (дата звернення: 14.05.2026).

8) Офіційна документація PostgreSQL. URL: https://www.postgresql.org/docs/ (дата звернення: 15.05.2026).

9) Бібліотека python-telegram-bot. URL: https://docs.python-telegram-bot.org/ (дата звернення: 16.05.2026).

10) Бібліотека Asyncpg для PostgreSQL. URL: https://magicstack.github.io/asyncpg/ (дата звернення: 16.05.2026).

11) Malkov Y. A., Yashunin D. A. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2020. Vol. 42, No. 4. P. 824–836.

12) Reimers N., Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. 2019. P. 3982–3992.

13) Gao Y., Xiong Y., Gao X. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv preprint arXiv:2312.10997. 2023.

14) Документація PyMuPDF (fitz). URL: https://pymupdf.readthedocs.io/ (дата звернення: 18.05.2026).

15) Документація бібліотеки python-docx. URL: https://python-docx.readthedocs.io/ (дата звернення: 18.05.2026).

16) Muennighoff N. et al. MTEB: Massive Text Embedding Benchmark. Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics. 2023. P. 2014–2037.

17) Bubeck S. et al. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv preprint arXiv:2303.12712. 2023.

18) Liu N. F. et al. Lost in the Middle: How Language Models Use Long Contexts. Transactions of the Association for Computational Linguistics. 2024. Vol. 12. P. 157–173.

19) Орлов С. Технології розробки програмного забезпечення: підручник. СПб.: Пітер, 2002. 464 с.

20) Документація Docker Compose. URL: https://docs.docker.com/compose/ (дата звернення: 20.05.2026).

Додаток А
Лістинг програмного коду

rag.py

```
import os
from openai import AsyncOpenAI
from qdrant_client import AsyncQdrantClient
from dotenv import load_dotenv
from embeddings import get_embedding

load_dotenv()
openai_client = AsyncOpenAI(api_key=os.getenv("OPENAI_API_KEY"))
qdrant = AsyncQdrantClient(
    host=os.getenv("QDRANT_HOST", "localhost"),
    port=int(os.getenv("QDRANT_PORT", 6333))
)

COLLECTION_NAME = "knu_docs"
LLM_MODEL       = "gpt-4o-mini"
TOP_K           = 5

SYSTEM_PROMPT = """Ти — навчальний асистент кафедри комп'ютерних систем та мереж \
Криворізького національного університету.
Твоя задача — відповідати на запитання студентів ВИКЛЮЧНО на основі наданих \
фрагментів навчальних матеріалів.
Правила:
- Відповідай ТІЛЬКИ на основі наданого контексту
- Якщо відповіді немає в контексті — чесно скажи про це, наприклад: \
"На жаль, у наданих матеріалах немає інформації про це."
- Відповідай українською мовою
- Будь конкретним і корисним
- Якщо питання стосується коду — надавай приклади з контексту

Контекст з навчальних матеріалів:
{context}"""

async def search_documents(query: str) -> list[dict]:
    """Шукає релевантні фрагменти документів у Qdrant."""
    query_vector = await get_embedding(query)
    results = await qdrant.search(
        collection_name=COLLECTION_NAME,
        query_vector=query_vector,
        limit=TOP_K,
```

```
            with_payload=True,
            score_threshold=0.45
        )
        fragments = []
        for hit in results:
            fragments.append({
                "text":   hit.payload.get("text", ""),
                "source": hit.payload.get("source", "невідомо"),
                "score":  round(hit.score, 3)
            })
        return fragments

    async def ask_llm(query: str, history: list[dict]) -> str:
        """Основна RAG-функція: запит → пошук → промпт → відповідь."""
        fragments = await search_documents(query)

        if not fragments:
            context = "Релевантних матеріалів не знайдено."
        else:
            parts = []
            for i, f in enumerate(fragments, 1):
                parts.append(
                    f"[Фрагмент {i} | Джерело: {f['source']}]\n{f['text']}"
                )
            context = "\n\n---\n\n".join(parts)

        messages = [{"role": "system", "content":
SYSTEM_PROMPT.format(context=context)}]
        messages.extend(history)
        messages.append({"role": "user", "content": query})

        response = await openai_client.chat.completions.create(
            model=LLM_MODEL,
            messages=messages,
            temperature=0.2,
            max_tokens=1000
        )
        answer = response.choices[0].message.content

        no_info_phrases = [
            "немає інформації", "відсутня інформація", "не знайдено",
            "не містить", "контекст не містить", "наданому контексті немає",
            "наданих матеріалах немає", "на жаль",
        ]
        answer_lower = answer.lower()
        has_real_answer = not any(p in answer_lower for p in no_info_phrases)
```

```
    if fragments and has_real_answer:
        sources = sorted(set(f["source"] for f in fragments))
        answer += f"\n\n📚 Джерела: {', '.join(sources)}"

    return answer
```

database.py

```
import asyncpg
import os
from dotenv import load_dotenv

load_dotenv()
DSN = os.getenv("POSTGRES_DSN")

async def init_db():
    """Створює таблиці якщо їх ще немає."""
    conn = await asyncpg.connect(DSN)
    await conn.execute("""
        CREATE TABLE IF NOT EXISTS users (
            user_id     BIGINT PRIMARY KEY,
            username    TEXT,
            first_name  TEXT,
            created_at  TIMESTAMPTZ DEFAULT NOW()
        );
        CREATE TABLE IF NOT EXISTS messages (
            id          SERIAL PRIMARY KEY,
            user_id     BIGINT REFERENCES users(user_id),
            role        TEXT NOT NULL,
            content     TEXT NOT NULL,
            created_at  TIMESTAMPTZ DEFAULT NOW()
        );
        CREATE INDEX IF NOT EXISTS idx_messages_user_id
            ON messages(user_id, created_at DESC);
    """)
    await conn.close()
    print("БД ініціалізована")

async def ensure_user(user_id: int, username: str, first_name: str):
    """Додає користувача якщо його ще немає."""
    conn = await asyncpg.connect(DSN)
    await conn.execute("""
        INSERT INTO users (user_id, username, first_name)
        VALUES ($1, $2, $3)
```

```
        ON CONFLICT (user_id) DO NOTHING;
    """, user_id, username, first_name)
    await conn.close()

async def save_message(user_id: int, role: str, content: str):
    """Зберігає повідомлення в історію."""
    conn = await asyncpg.connect(DSN)
    await conn.execute("""
        INSERT INTO messages (user_id, role, content)
        VALUES ($1, $2, $3);
    """, user_id, role, content)
    await conn.close()

async def get_history(user_id: int, limit: int = 6) -> list[dict]:
    """Повертає останні N повідомлень користувача."""
    conn = await asyncpg.connect(DSN)
    rows = await conn.fetch("""
        SELECT role, content FROM messages
        WHERE user_id = $1
        ORDER BY created_at DESC
        LIMIT $2;
    """, user_id, limit)
    await conn.close()
    return [{"role": r["role"], "content": r["content"]} for r in reversed(rows)]

async def clear_history(user_id: int):
    """Очищає історію конкретного користувача."""
    conn = await asyncpg.connect(DSN)
    await conn.execute(
        "DELETE FROM messages WHERE user_id = $1;", user_id
    )
    await conn.close()
```

embeddings.py

```
from openai import AsyncOpenAI
import os
from dotenv import load_dotenv

load_dotenv()
client = AsyncOpenAI(api_key=os.getenv("OPENAI_API_KEY"))
EMBEDDING_MODEL = "text-embedding-3-small"

async def get_embedding(text: str) -> list[float]:
    """Перетворює текст на вектор через OpenAI API."""
    text = text.replace("\n", " ").strip()
```

```
    response = await client.embeddings.create(
        input=text,
        model=EMBEDDING_MODEL
    )
    return response.data[0].embedding
```

handlers.py

```
import logging
from telegram import Update
from telegram.ext import ContextTypes
from database import ensure_user, save_message, get_history, clear_history
from rag import ask_llm

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[
        logging.FileHandler("bot.log", encoding="utf-8"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
MAX_MESSAGE_LENGTH = 4000

def split_long_message(text: str) -> list[str]:
    """Ділить довге повідомлення на частини."""
    if len(text) <= MAX_MESSAGE_LENGTH:
        return [text]
    parts = []
    while text:
        if len(text) <= MAX_MESSAGE_LENGTH:
            parts.append(text)
            break
        split_at = text.rfind("\n", 0, MAX_MESSAGE_LENGTH)
        if split_at == -1:
            split_at = MAX_MESSAGE_LENGTH
        parts.append(text[:split_at])
        text = text[split_at:].strip()
    return parts

async def cmd_start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    user = update.effective_user
    await ensure_user(user.id, user.username or "", user.first_name or "")
    logger.info(f"Новий користувач: {user.id} ({user.first_name})")
```

```
    text = (
        f"Вітаю, {user.first_name}!\n\n"
        "Я навчальний асистент кафедри КСМ. "
        "Задавай питання щодо навчальних матеріалів — "
        "я відповім на основі методичок та лабораторних робіт.\n\n"
        "/help — список команд\n"
        "/clear — очистити історію розмови"
    )
    await update.message.reply_text(text)

async def cmd_help(update: Update, context: ContextTypes.DEFAULT_TYPE):
    text = (
        "Команди:\n"
        "/start — почати розмову\n"
        "/clear — очистити історію\n\n"
        "Просто пиши запитання, наприклад:\n"
        "• Що таке критичний шлях?\n"
        "• Як виконати лабораторну №2 з КС?\n"
        "• Поясни структурну надлишковість"
    )
    await update.message.reply_text(text)

async def cmd_clear(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await clear_history(update.effective_user.id)
    logger.info(f"Очищено історію: {update.effective_user.id}")
    await update.message.reply_text("Історію розмови очищено.")

async def handle_message(update: Update, context: ContextTypes.DEFAULT_TYPE):
    user = update.effective_user
    query = update.message.text.strip()
    if not query:
        return
    await ensure_user(user.id, user.username or "", user.first_name or "")
    logger.info(f"Запит від {user.id}: {query[:80]}")
    await update.message.chat.send_action("typing")

    try:
        history = await get_history(user.id, limit=6)
        answer = await ask_llm(query, history)
        clean_answer = answer.split("\n\n📚")[0]
        await save_message(user.id, "user", query)
        await save_message(user.id, "assistant", clean_answer)
```

```
            parts = split_long_message(answer)
            for part in parts:
                await update.message.reply_text(part)
            logger.info(f"Відповідь для {user.id}: {len(answer)} символів")
        except Exception as e:
            logger.error(f"Помилка для {user.id}: {e}", exc_info=True)
            await update.message.reply_text(
                "Виникла технічна помилка. Спробуйте повторити запит через кілька секунд."
            )
```

main.py

```
import os
from dotenv import load_dotenv
from telegram.ext import Application, CommandHandler, MessageHandler, filters

load_dotenv()

def main():
    print("Старт...")
    from database import init_db
    import asyncio

    asyncio.get_event_loop().run_until_complete(init_db())
    print("БД ініціалізована")

    token = os.getenv("TELEGRAM_TOKEN")
    if not token:
        print("ПОМИЛКА: TELEGRAM_TOKEN не знайдено в .env")
        return

    print(f"Токен знайдено: {token[:10]}...")
    app = Application.builder().token(token).build()

    from handlers import cmd_start, cmd_help, cmd_clear, handle_message

    app.add_handler(CommandHandler("start", cmd_start))
    app.add_handler(CommandHandler("help", cmd_help))
    app.add_handler(CommandHandler("clear", cmd_clear))
    app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND, handle_message))

    print("Бот запущено. Очікую повідомлення...")
    app.run_polling(drop_pending_updates=True)
```

```
if __name__ == "__main__":
    main()
```

index_docs.py

```
import asyncio
import os
import sys
from pathlib import Path
from dotenv import load_dotenv
from openai import AsyncOpenAI
from qdrant_client import AsyncQdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
import fitz  # pymupdf
from docx import Document as DocxDocument

load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
QDRANT_HOST   = os.getenv("QDRANT_HOST", "localhost")
QDRANT_PORT   = int(os.getenv("QDRANT_PORT", 6333))
COLLECTION_NAME = "knu_docs"
EMBEDDING_MODEL = "text-embedding-3-small"
VECTOR_DIM      = 1536
CHUNK_SIZE    = 500   # токени (приблизно)
CHUNK_OVERLAP = 80    # символів перекриття

openai_client = AsyncOpenAI(api_key=OPENAI_API_KEY)
qdrant_client = AsyncQdrantClient(host=QDRANT_HOST,
port=QDRANT_PORT)

def read_file(path: Path) -> str:
    ext = path.suffix.lower()
    if ext == ".pdf":
        doc = fitz.open(str(path))
        return "\n".join([page.get_text() for page in doc])
    elif ext == ".docx":
        doc = DocxDocument(str(path))
        return "\n".join(p.text for p in doc.paragraphs if p.text.strip())
    else:
        raise ValueError(f"Непідтримуваний формат: {ext}")

def split_text(text: str, chunk_size: int = CHUNK_SIZE, overlap: int =
CHUNK_OVERLAP) -> list[str]:
    char_limit = chunk_size * 4
    chunks = []
```

```
        start = 0
        while start < len(text):
            end = start + char_limit
            chunk = text[start:end].strip()
            if chunk:
                chunks.append(chunk)
            start += char_limit - overlap
        return chunks

    async def get_embedding(text: str) -> list[float]:
        text = text.replace("\n", " ").strip()
        resp = await openai_client.embeddings.create(
            input=text, model=EMBEDDING_MODEL
        )
        return resp.data[0].embedding

    async def ensure_collection():
        existing = await qdrant_client.get_collections()
        names = [c.name for c in existing.collections]
        if COLLECTION_NAME not in names:
            await qdrant_client.create_collection(
                collection_name=COLLECTION_NAME,
                vectors_config=VectorParams(size=VECTOR_DIM,
distance=Distance.COSINE)
            )
            print(f"Колекцію '{COLLECTION_NAME}' створено.")
        else:
            print(f"Колекція '{COLLECTION_NAME}' вже існує.")

    async def upload_chunks(chunks: list[str], source: str, offset: int) -> int:
        points = []
        for i, chunk in enumerate(chunks):
            vector = await get_embedding(chunk)
            points.append(PointStruct(
                id=offset + i,
                vector=vector,
                payload={"text": chunk, "source": source, "chunk": i}
            ))
            print(f"  [{source}] фрагмент {i+1}/{len(chunks)} — ок")

        await        qdrant_client.upsert(collection_name=COLLECTION_NAME,
points=points)
        return len(points)

    async def index_all(docs_dir: str = "docs"):
        docs_path = Path(docs_dir)
```

```
    if not docs_path.exists():
        print(f"Папка '{docs_dir}' не знайдена.")
        sys.exit(1)

    files = [f for f in docs_path.iterdir() if f.suffix.lower() in (".pdf", ".docx")]
    if not files:
        print("Файлів для індексування не знайдено.")
        sys.exit(1)

    await ensure_collection()
    total = 0
    offset = 0

    for file in files:
        print(f"\nОбробляю: {file.name}")
        try:
            text   = read_file(file)
            chunks = split_text(text)
            print(f"  Фрагментів: {len(chunks)}")
            count  = await upload_chunks(chunks, file.name, offset)
            offset += count
            total  += count
        except Exception as e:
            print(f"  Помилка: {e}")

    print(f"\nГотово. Завантажено {total} фрагментів у Qdrant.")

if __name__ == "__main__":
    asyncio.run(index_all("docs"))
```

.env

```
OPENAI_API_KEY=sk-proj-***
TELEGRAM_TOKEN=8707808068:***
QDRANT_HOST=localhost
QDRANT_PORT=6333
POSTGRES_DSN=postgresql://ragbot:ragbot123@localhost:5432/ragbot
```