

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Інтелектуальний помічник програміста на основі
RAG-архітектури»**

Виконав студент групи ПЗм-23-1
Войтко Павло Олексійович

Керівник роботи:
Чижмотря Олена Геннадіївна

Рецензент:
Морозов Андрій Васильович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«1» жовтня 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **Войтко Павло Олексійович**

Керівник роботи: **Чижмотря Олена Геннадіївна**

Тема роботи: **«Інтелектуальний помічник програміста на основі RAG-архітектури»**, затверджена Наказом закладу вищої освіти від **«1» жовтня 2024 р. № 520/с**

Вихідні дані для роботи: **процес розробки програмного забезпечення та методи підвищення його ефективності за допомогою інтелектуальних помічників; архітектура RAG, методи пошуку релевантної інформації в документації та генерації відповідей на запити програмістів, а також технології розробки інтерфейсів користувача у форматі чат-ботів.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О. Г.	01.10.2024р.	01.10.2024р.
2	Чижмотря О. Г.	01.10.2024р.	01.10.2024р.
3	Чижмотря О. Г.	01.10.2024р.	01.10.2024р.

РЕФЕРАТ

Кваліфікаційна робота освітнього ступеня «магістр» присвячена розробці інтелектуального помічника програміста на основі RAG-архітектури з інтеграцією документації та генерацією відповідей у форматі чат-боту.

Пояснювальна записка до випускної роботи містить текстову частину, викладену на 96 сторінках друкованого тексту, має 11 ілюстрацій, 2 таблиці. Список використаних джерел містить 44 найменування і займає 5 сторінок. Загальний обсяг роботи – 191 сторінки.

Метою роботи є розробка інтелектуального помічника програміста, який використовує RAG-архітектуру для інтерактивного пошуку інформації в документації, а також надання відповідей на запити користувача у зручному форматі чат-боту.

В роботі детально описано процес створення програми на Python, що реалізує RAG-архітектуру, включаючи інтеграцію мовної моделі, механізму пошуку, та графічного інтерфейсу. Розглянуто етапи побудови векторної бази даних, застосування мовної моделі Llama для генерації тексту та створення інтерфейсу користувача на базі бібліотеки PyQt. Проведено тестування розробленої програми та оцінку її ефективності.

КЛЮЧОВІ СЛОВА: RAG, ІНТЕЛЕКТУАЛЬНИЙ ПОМІЧНИК, ЧАТ-БОТ, PYTHON, LLAMA, LLAAMINDEX, FAISS, PYQT, СЕМАНТИЧНИЙ ПОШУК, ВЕКТОРНА БАЗА.

ABSTRACT

This master's thesis is dedicated to the development of an intelligent programming assistant based on the RAG architecture, integrating documentation and generating responses in a chatbot format.

The explanatory note consists of an introduction, three chapters, and a list of references. The explanatory note to the master's thesis contains 96 pages, 11 illustrations, 2 tables, 1 appendix, and 42 references to external sources. The total volume of the work is 191 pages.

The aim of the work is to develop an intelligent programming assistant that utilizes the RAG architecture for interactive information retrieval within documentation, as well as providing answers to user queries in a convenient chatbot format.

The thesis details the process of creating a Python program that implements the RAG architecture, including the integration of a language model, a search mechanism, and a graphical interface. The steps of building a vector database, applying the Llama language model for text generation, and creating a user interface based on the PyQt library are examined. Testing of the developed program and an evaluation of its effectiveness have been conducted.

KEYWORDS: RAG, INTELLIGENT ASSISTANT, CHATBOT, PYTHON, LLAMA, LLAAMINDEX, FAISS, PYQT, SEMANTIC SEARCH, VECTOR DATABASE.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ	10
1.1. Огляд технології RAG та її ролі в розробці інтелектуальних систем.	10
1.2. Аналіз ключових технологій, бібліотек та існуючих ІІІ-інструментів для програмістів	13
1.3. Визначення основних вимог до інтелектуального помічника програміста	18
1.4. Обґрунтування вибору програмних засобів та технологій для розробки.....	22
Висновки до розділу 1	23
РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНТЕЛЕКТУАЛЬНОГО ПОМІЧНИКА ПРОГРАМІСТА.....	25
2.1. Архітектура інтелектуального помічника програміста на основі RAG.....	25
2.2. Реалізація інтерфейсу чат-бота та взаємодія модулів в архітектурі RAG.....	38
2.3. Загальна архітектура та структура програми	44
2.4. Детальний опис структури бази даних інтелектуального помічника.	53
Висновки до розділу 2	57
РОЗДІЛ 3. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	60
3.1. Методика тестування та оцінки якості роботи інтелектуального помічника.....	60
3.2. Аналіз результатів тестування та порівняння з існуючими рішеннями.....	69
3.3. Оцінка зручності та ефективності використання інтерфейсу чат-бота.....	74

3.4. Визначення можливих напрямків подальшого розвитку та вдосконалення системи	79
Висновки до розділу 3	84
ВИСНОВКИ	90
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	93
ДОДАТКИ	97

ВСТУП

Актуальність теми. В епоху стрімкого розвитку інформаційних технологій розробка програмного забезпечення стає дедалі складнішим та комплекснішим процесом. Програмісти щодня стикаються з необхідністю опрацювання величезних обсягів інформації, розкиданої по численних джерелах: технічній документації, специфікаціях, форумах та спеціалізованих ресурсах. Пошук, аналіз та застосування цієї інформації вимагає значних витрат часу та зусиль, що негативно впливає на продуктивність розробників та може призводити до помилок у коді.

У зв'язку з цим, розробка інтелектуальних помічників, здатних автоматизувати рутинні операції, пов'язані з пошуком та обробкою інформації, стає нагальною потребою. Такі помічники покликані спростити процес розробки, зекономити час програмістів та підвищити якість програмного забезпечення, що створюється.

Одним із перспективних підходів до створення інтелектуальних помічників є використання архітектури RAG (Retrieval-Augmented Generation), яка поєднує можливості інформаційно-пошукових систем з генеративними мовними моделями. RAG-системи здатні не лише знаходити релевантну інформацію у великих масивах даних, але й формувати на її основі змістовні та точні відповіді на запити користувачів.

Мета роботи – розробка інтелектуального помічника програміста на основі архітектури RAG (Retrieval-Augmented Generation), що надає зручний інтерфейс у вигляді чат-бота, який дозволить ефективно шукати інформацію в документації та отримувати релевантні відповіді.

Для досягнення поставленої мети необхідно вирішити такі основні завдання:

Проаналізувати існуючі підходи до розробки інтелектуальних помічників програміста та обґрунтувати вибір RAG-архітектури.

Розробити та імплементувати механізм пошуку релевантної інформації в документації на основі технологій семантичного пошуку та векторних представлень.

Спроектувати та реалізувати механізм генерації відповідей на основі RAG-архітектури з використанням мовної моделі Llama та бібліотеки Llama.cpp.

Створити зручний інтерфейс користувача у форматі чат-бота на базі бібліотеки PyQt.

Провести тестування та оцінку ефективності розробленого програмного забезпечення.

Об'єкт дослідження – процес розробки програмного забезпечення та методи підвищення його ефективності за допомогою інтелектуальних помічників.

Предмет дослідження – архітектура RAG, методи пошуку релевантної інформації в документації та генерації відповідей на запити програмістів, а також технології розробки інтерфейсів користувача у форматі чат-ботів.

Наукова новизна – роботи полягає в розробці та імплементції інтелектуального помічника програміста на основі RAG-архітектури, що поєднує можливості інформаційно-пошукової системи з генеративною мовною моделлю Llama. Розроблений помічник забезпечує ефективний пошук у великому обсязі технічної документації та формує змістовні відповіді на запити користувачів, адаптовані до контексту розмови. Особливістю даного рішення є використання бібліотеки Llama.cpp для локального запуску мовної моделі, що забезпечує конфіденційність та незалежність від зовнішніх сервісів, та бібліотеки LlamaIndex для побудови індексів та реалізації семантичного пошуку.

Практична значущість – полягає у розробці прототипу програмного забезпечення, що може бути використане для підвищення продуктивності праці програмістів, скорочення часу на пошук необхідної інформації та зниження кількості помилок у коді.

Структура роботи. Дана кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі проведено аналіз існуючих рішень у галузі розробки інтелектуальних помічників програміста, обґрунтовано актуальність та доцільність використання RAG-архітектури.

У другому розділі детально описано процес проєктування та розробки системи, включаючи архітектуру, механізм пошуку інформації, механізм генерації відповідей та інтерфейс користувача.

У третьому розділі представлено результати тестування та оцінки ефективності розробленого програмного забезпечення, а також визначено напрямки подальшого розвитку системи.

Висновки узагальнюють результати дослідження та підкреслюють досягнення поставленої мети.

Список використаних джерел містить перелік наукових статей, книг та інтернет-ресурсів, що були використані під час роботи над кваліфікаційною роботою.

Додатки містять лістинг вихідного коду програми.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Огляд технології RAG та її ролі в розробці інтелектуальних систем

Архітектура Retrieval-Augmented Generation (RAG) є новітнім гібридним підходом, що поєднує сильні сторони інформаційно-пошукових систем та генеративних мовних моделей. Цей інноваційний симбіоз відкриває нові горизонти у сфері обробки природної мови та розробки інтелектуальних систем, здатних не лише генерувати зв'язний текст, але й оперувати фактами та знаннями з зовнішніх джерел.

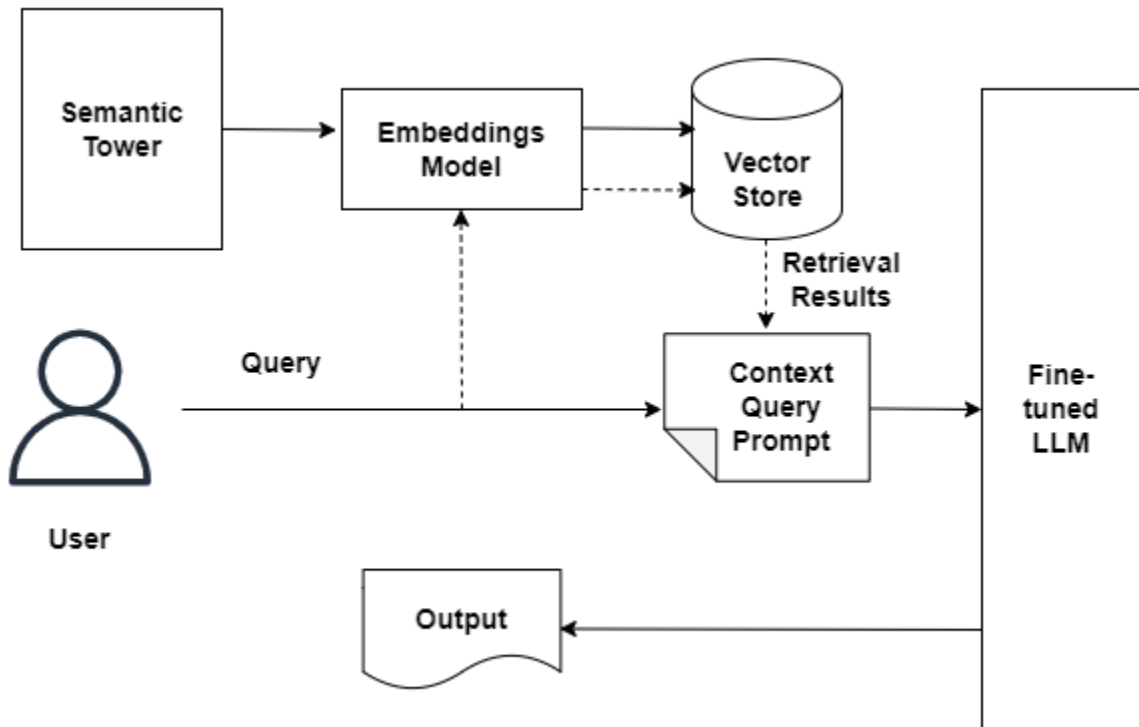


Рис 1.1. Архітектура Retrieval-Augmented Generation

В основі RAG лежить ідея доповнення генеративної моделі механізмом вилучення релевантної інформації із зовнішньої бази знань. Такий підхід дозволяє моделі не обмежуватися лише тими знаннями, що були зафіксовані в її параметрах під час попереднього навчання, а й активно використовувати зовнішні дані для формування відповідей, що базуються на актуальній та

перевіреній інформації. Завдяки цьому RAG-моделі демонструють значно вищу точність та достовірність у порівнянні з традиційними генеративними моделями.

RAG-архітектура передбачає послідовне виконання двох основних етапів: вилучення (retrieval) та генерацію (generation). На першому етапі, отримавши запит від користувача, система активує механізм пошуку, звертаючись до зовнішньої бази знань, яка може бути представлена у вигляді великого корпусу документів, що охоплює різноманітні текстові дані: від технічної документації та наукових публікацій до новинних статей та обговорень на форумах. Для реалізації етапу вилучення застосовуються спеціально навчені нейронні мережі, які з високою точністю ранжують документи за ступенем їхньої відповідності вхідному запиту. Архітектура Retrieval-Augmented Generation представлена на рис. 1.1.

Ці моделі часто базуються на трансформерній архітектурі, яка є золотим стандартом в обробці природної мови, завдяки своїй здатності ефективно обробляти текстові дані, вловлювати глибинні семантичні зв'язки між словами та враховувати широкий контекст. На другому етапі, етапі генерації, RAG-модель використовує знайдені релевантні фрагменти тексту як додатковий контекст для формування остаточної відповіді. Генерація тексту здійснюється за допомогою потужних мовних моделей, найчастіше також заснованих на архітектурі трансформер, які здатні генерувати зв'язний, грамотний та змістовний текст, враховуючи як оригінальний запит, так і наданий контекст.

Використання RAG-архітектури надає ряд суттєвих переваг у порівнянні з традиційними генеративними моделями. По-перше, RAG-моделі здатні генерувати відповіді, що безпосередньо спираються на фактичну інформацію, знайдену у зовнішніх джерелах. Це значно знижує ризик генерації моделлю неправдивої або вигаданої інформації. По-друге, завдяки постійному доступу до зовнішньої бази знань, RAG-моделі мають можливість оперувати більш актуальною, детальною та вичерпною

інформацією. По-третє, RAG-архітектура надає унікальну можливість оновлювати базу знань без необхідності повного перенавчання моделі, що вимагає значних обчислювальних ресурсів та часу. Це дозволяє системі адаптуватися до нових даних, умов та вимог, що робить її більш гнучкою та практичною у реальних застосуваннях.

Роль RAG у розробці інтелектуальних систем важко переоцінити. Ця технологія відкриває шлях до створення систем, здатних вести осмислений діалог, відповідати на складні запитання, аналізувати інформацію з різних джерел та оперувати фактами з високою точністю. RAG-моделі знаходять широке застосування у різноманітних додатках, включаючи інтелектуальних чат-ботів, системи питання-відповідь, інструменти для автоматичного реферування та аналізу тексту, системи підтримки прийняття рішень та багато інших. Завдяки своїй здатності ефективно поєднувати інформаційний пошук з генерацією тексту, RAG-архітектура відкриває нові можливості для розробки більш просунутих, адаптивних та ефективних інтелектуальних систем. Застосування RAG-архітектури для створення інтелектуального помічника програміста є особливо перспективним напрямком.

Такий помічник зможе не лише генерувати відповіді на запитання, але й робити це, спираючись на велику та постійно оновлювану базу знань, що включає технічну документацію, специфікації, приклади коду та обговорення на форумах. Завдяки RAG, помічник зможе ефективно опрацьовувати інформацію з різноманітних джерел, враховувати контекст запиту та надавати точні, вичерпні та актуальні відповіді. Використання RAG-архітектури дозволить інтелектуальному помічнику стати надійним та незамінним інструментом для програмістів, що значно підвищить їх продуктивність та сприятиме створенню більш якісного та інноваційного програмного забезпечення.

1.2. Аналіз ключових технологій, бібліотек та існуючих ШІ-інструментів для програмістів

Розробка ефективного інтелектуального помічника програміста, здатного суттєво підвищити продуктивність та якість створення програмного забезпечення, є складним завданням, що вимагає ретельного вибору інструментів та технологій. Одним з ключових елементів такої системи є потужна та ефективна мовна модель, і в цьому контексті бібліотека `Llama.cpp` [2] виступає в якості важливого компонента. `Llama.cpp` являє собою ретельно оптимізовану реалізацію мовної моделі `Llama`, написану на мові програмування `C/C++`. Головною перевагою цієї бібліотеки є її здатність запускати модель локально на центральному процесорі (CPU) користувача, що робить її доступною для широкого кола розробників, незалежно від наявності потужних графічних прискорювачів (GPU). Це відкриває можливості для вбудовування інтелектуальних мовних моделей у різноманітні додатки, від настільних до мобільних, та вбудованих систем.

Архітектура `Llama.cpp` спроектована з урахуванням особливостей роботи на CPU, що дозволило досягти максимальної продуктивності при виконанні обчислень, властивих трансформерним моделям. Бібліотека використовує оптимізовані алгоритми для матричного множення, функцій активації та механізму уваги (attention), які повною мірою використовують можливості сучасних CPU, включаючи векторні інструкції (SIMD) та багатопотоковість. Окрім того, `Llama.cpp` має мінімальні залежності від зовнішніх бібліотек, що спрощує процес встановлення та інтеграції, а також робить її більш легкою та портативною. Важливим аспектом є підтримка квантування моделей, що дозволяє зменшити їхній розмір та підвищити швидкість роботи без значної втрати точності.

Для взаємодії з мовною моделлю `Llama`, `Llama.cpp` надає простий, інтуїтивно зрозумілий та зручний API, доступний як на мові `C++`, так і на `Python`. Це робить бібліотеку доступною для широкого кола розробників,

незалежно від їхнього основного стеку технологій. API дозволяє легко завантажувати моделі, передавати їм вхідні дані, налаштовувати параметри генерації та отримувати згенерований текст, а також забезпечує гнучкість у налаштуванні параметрів генерації тексту та контролю над поведінкою моделі.

Ефективна реалізація RAG-архітектури, що забезпечує інтелектуальному помічнику можливість поєднувати генерацію тексту з доступом до зовнішньої інформації, неможлива без якісного індексування великих масивів текстових даних та швидкого пошуку релевантної інформації. У цьому контексті важливу роль відіграє зв'язка бібліотек LlamaIndex [3] та FAISS [4]. LlamaIndex, виступаючи як інструмент високого рівня, забезпечує зручний інтерфейс для створення, управління та використання індексів текстових даних. FAISS, у свою чергу, є бібліотекою низького рівня, яка надає передові алгоритми для ефективного зберігання та обробки векторних представлень тексту.

FAISS, розроблена Facebook AI Research, спеціалізується на пошуку подібності у багатовимірних просторах та пропонує широкий спектр алгоритмів для індексування та пошуку найближчих сусідів, оптимізованих для ефективної роботи як на CPU, так і на GPU. LlamaIndex надає зручний інтерфейс для взаємодії з FAISS, абстрагуючись від складнощів його конфігурації та дозволяючи розробникам фокусуватися на логіці роботи системи, а не на низькорівневих деталях реалізації алгоритмів пошуку. Інтеграція LlamaIndex з FAISS відбувається через спеціальний клас FaissVectorStore, що дозволяє створювати індекси FAISS з заданими параметрами та додавати до них векторні представлення текстових фрагментів.

LlamaIndex також забезпечує гнучкі можливості налаштування параметрів індексу FAISS. Розробники можуть вибирати різні типи індексів, такі як IndexFlatL2, IndexIVFFlat, IndexHNSW, залежно від вимог до точності

та швидкості пошуку, а також задавати такі параметри, як кількість кластерів, кількість зв'язків та глибину пошуку. Це дозволяє досягти оптимального балансу між якістю пошуку та часом, необхідним на його виконання.

Процес пошуку інформації з використанням LlamaIndex та FAISS відбувається наступним чином. LlamaIndex отримує векторне представлення запиту від користувача та передає його до FAISS для виконання пошуку найближчих векторів у індексі. FAISS, використовуючи обраний тип індексу та метрику подібності, виконує пошук та повертає індекси векторів, що є найближчими до запиту. LlamaIndex, отримавши результати пошуку, зіставляє індекси векторів з відповідними текстовими фрагментами та повертає їх користувачеві у відсортованому за релевантністю порядку. Це забезпечує швидкий та ефективний пошук релевантної інформації у великих масивах текстових даних.

Окрім інструментів, що безпосередньо забезпечують роботу мовної моделі та векторної бази даних, важливу роль у сучасній розробці програмного забезпечення відіграють інтелектуальні інтегровані середовища розробки (IDE). Такі IDE, як PyCharm, IntelliJ IDEA та Visual Studio, пропонують розширений набір функцій для полегшення розробки, підвищення продуктивності та забезпечення якості коду.

Ці IDE надають розробникам можливості автодоповнення коду, що значно прискорює процес написання та допомагає уникати синтаксичних помилок, підсвічування синтаксису для покращення читабельності коду, статичного аналізу коду для виявлення потенційних проблем та інтеграцію з системами контролю версій, що спрощує процес співпраці між розробниками. Такі інструменти, як SonarQube (рис. 1.2), дозволяють проводити глибокий аналіз коду, виявляючи типові помилки програмування, витоки пам'яті та порушення стилю кодування.

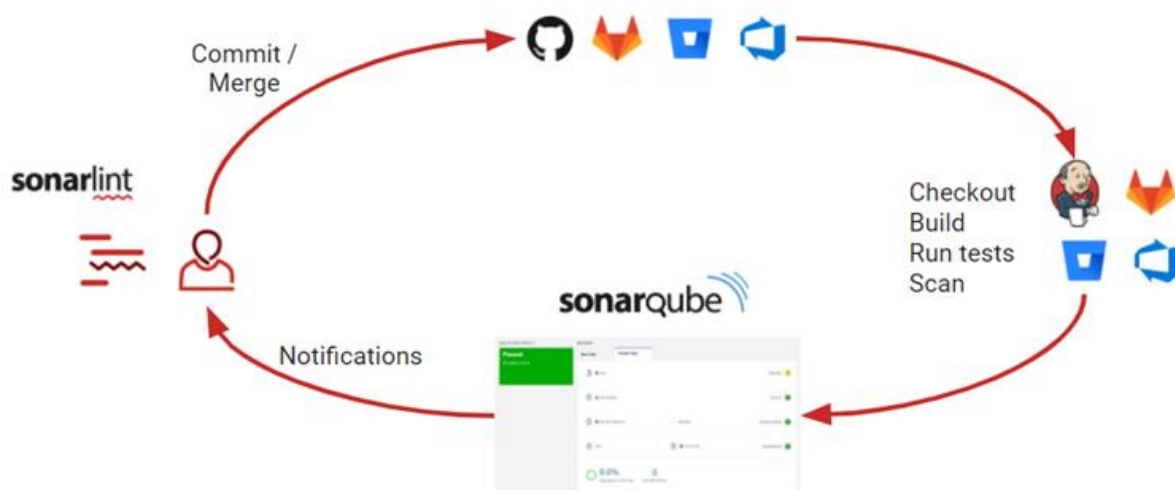


Рис 1.2. Цикл розробки із SonarQube

Системи автодоповнення коду, такі як Tabnine (рис 1.3) та GitHub Copilot (рис. 1.4), представляють собою інноваційний клас інструментів, що використовують можливості глибокого навчання для генерування коду на основі контексту, коментарів та намірів програміста. Ці системи здатні пропонувати варіанти завершення коду в режимі реального часу, генерувати цілі блоки коду або навіть функції на основі текстового опису. GitHub Copilot, зокрема, навчається на величезних обсягах відкритого вихідного коду та використовує цю інформацію для надання програмістам корисних підказок.

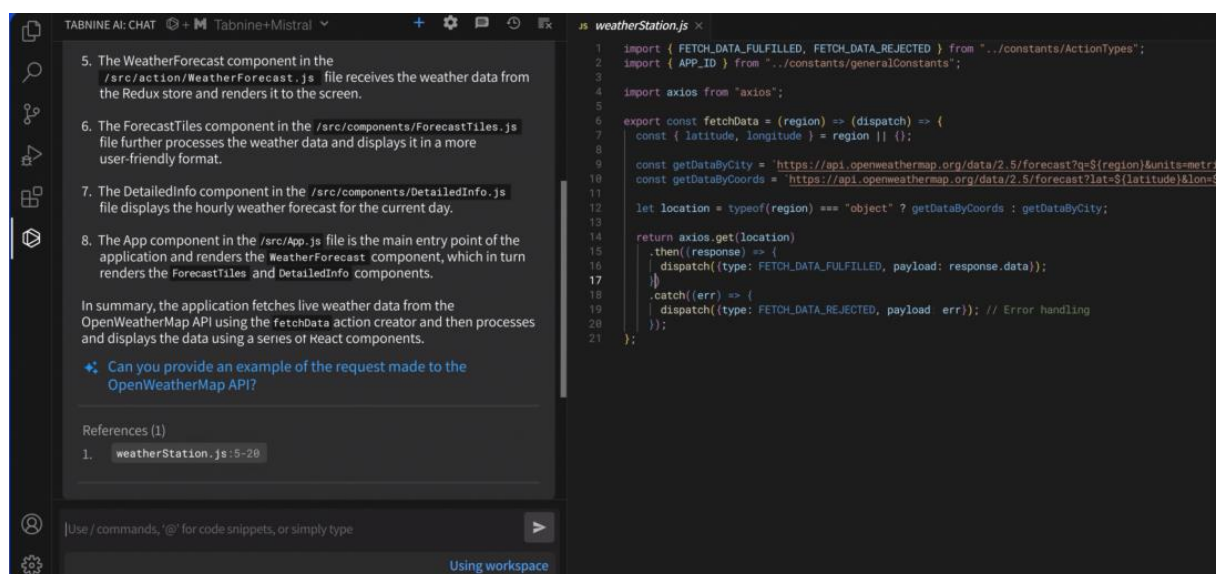


Рис 1.3. Робота Tabnine

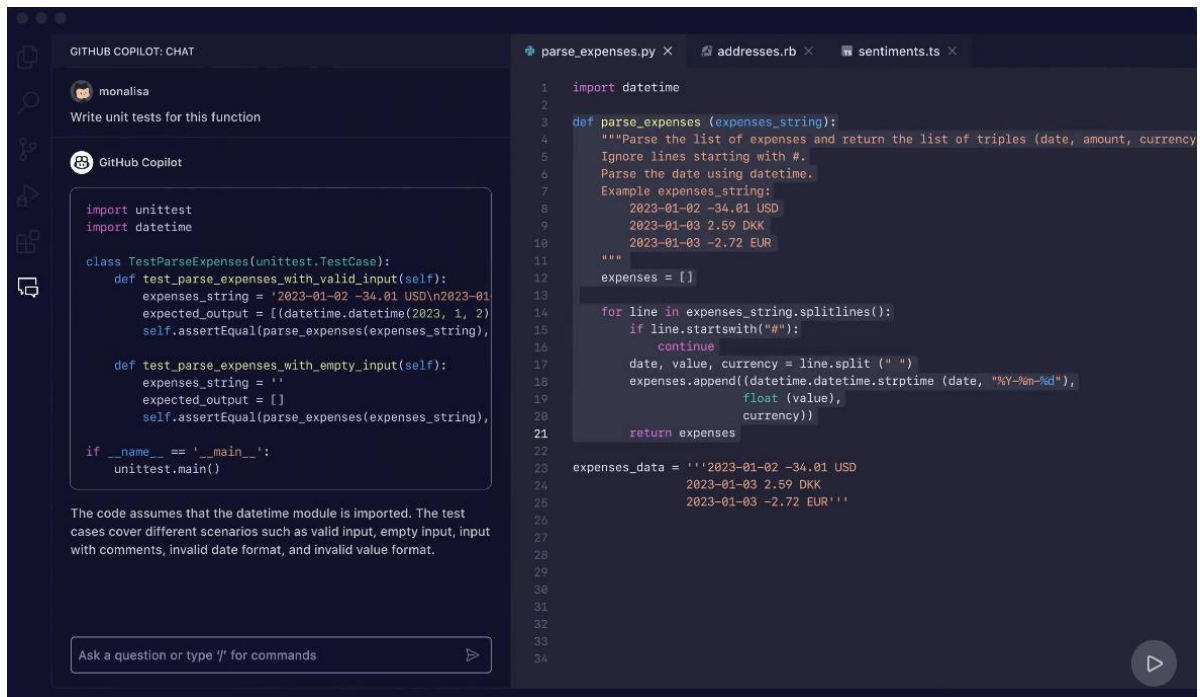


Рис 1.4. Робота GitHub Copilot

Для забезпечення більш ефективного та якісного процесу розробки, використовуються інструменти для автоматизації рефакторингу, такі як ReSharper для Visual Studio або вбудовані можливості IntelliJ IDEA. Ці інструменти дозволяють автоматично перейменовувати змінні, методи та класи, виділяти фрагменти коду в окремі методи, оптимізувати імпорти та видаляти надлишковий код, що робить структуру коду більш зрозумілою та легкою для підтримки. Автоматизація рефакторингу дозволяє програмістам концентруватися на вирішенні складних логічних завдань, а не на рутинних операціях з кодом.

Незважаючи на значні переваги, які надають ці інструменти, важливо розуміти їхні обмеження. Системи автодоповнення коду можуть генерувати синтаксично правильний, але не завжди оптимальний або логічно коректний код. Інструменти для рефакторингу можуть пропонувати зміни, які не враховують специфіку проєкту або можуть призвести до неочікуваних наслідків. Тому, використання ІІІ-інструментів у процесі розробки має бути обґрунтованим та контрольованим, а згенерований код має піддаватися ретельному тестуванню та аналізу.

1.3. Визначення основних вимог до інтелектуального помічника програміста

Інтелектуальний помічник програміста, розроблений для підтримки та оптимізації процесу розробки програмного забезпечення, повинен відповідати комплексу чітко визначених вимог, що забезпечують його функціональність, ефективність, зручність та практичну цінність для широкого кола користувачів. Першочерговою та, мабуть, найважливішою вимогою є здатність системи обробляти запити, сформульовані природною мовою, що є ключовим елементом забезпечення природної та інтуїтивно зрозумілої взаємодії між користувачем та системою. Це означає, що помічник має розуміти не лише окремі ключові слова, але й складніші конструкції, враховувати контекст запиту, ідіоматичні вирази та професійний сленг, характерний для середовища програмістів, а також бути здатним генерувати зрозумілі, зв'язні, структуровані та, головне, корисні відповіді, що відповідають запиту та наданій інформації.

Для ефективної реалізації цієї вимоги система повинна мати потужні та гнучкі механізми обробки природної мови (Natural Language Processing, NLP). Ці механізми мають не тільки забезпечувати точне розпізнавання слів та їхніх форм, але й аналізувати семантику запиту, виявляти наміри користувача, визначати сутності та ідентифікувати ключові поняття, що містяться в запиті. Окрім того, помічник повинен вміти аналізувати контекст запиту, враховуючи попередню історію діалогу, що дозволить йому надавати точні, персоналізовані та контекстно-релевантні відповіді, уникаючи повторень або непорозумінь. Для цього, у системі передбачено механізми зберігання та обробки історії діалогу, що дозволяють відстежувати попередні запити та відповіді.

Другою важливою вимогою, що висувається до інтелектуального помічника, є наявність у нього доступу до великої, актуальної, достовірної та ретельно структурованої бази знань, що охоплює різноманітні аспекти програмування. Ця база знань повинна включати не лише офіційну технічну

документацію мов програмування, бібліотек, фреймворків та API, а й приклади коду, що ілюструють вирішення типових задач, статті з авторитетних джерел, посібники, обговорення на форумах, дописи з блогів, найкращі практики програмування та інші корисні матеріали, що можуть бути необхідними розробнику для вирішення різних завдань. Важливим аспектом є забезпечення регулярного оновлення бази знань, включаючи до неї інформацію про нові технології, інструменти, методології та практики, а також виправлення помилок та неточностей у існуючих матеріалах, що дозволить підтримувати її актуальність та забезпечувати високу якість відповідей, які генерує система.

Для підтримки актуальності бази знань система має підтримувати різноманітні формати файлів, включаючи .csv, .doc, .docx, .enex, .epub, .html, .md, .odt, .pdf, .ppt, .pptx та .txt, щоб адміністратори системи могли легко додавати нові джерела інформації та оновлювати існуючі. Механізм оновлення бази знань має бути автоматизованим та простим у використанні, дозволяючи додавати та редагувати дані без необхідності глибокого розуміння внутрішньої архітектури системи. Окрім цього, помічник повинен вміти працювати з великими обсягами даних, швидко індексувати нові документи та оновлювати індекси, а також забезпечувати швидкий пошук потрібної інформації.

Третьою вимогою, що має велике значення, є здатність системи адаптуватися до індивідуальних потреб, вподобань та стилю роботи кожного конкретного користувача. Це передбачає створення механізмів персоналізації, що дозволяють системі навчатися на основі попередніх запитів користувача, зберігати його історію діалогу, аналізувати його код та надавати персоналізовані рекомендації, які максимально відповідають його поточним завданням та потребам. Наприклад, система має здатність запам'ятовувати улюблені мови програмування, часто використовувані бібліотеки та фреймворки, а також стиль формулювання запитів користувачем. Ці дані мають враховуватися при пошуку інформації та

генерації відповідей, що дозволить підвищити релевантність та точність наданих результатів.

Крім функціональних вимог, важливе значення має зручність та інтуїтивність інтерфейсу користувача. Інтерфейс має бути простим у використанні, не перевантаженим зайвими елементами, мати чітку та зрозумілу структуру та забезпечувати швидкий доступ до основних функцій. Оптимальним рішенням є використання інтерфейсу у вигляді чат-бота, що дозволяє користувачеві взаємодіяти з системою у звичній формі діалогу, використовуючи природну мову. Такий підхід знижує поріг входження для нових користувачів та робить роботу з помічником більш природною. Окрім цього, інтерфейс має підтримувати базове форматування тексту, виділення коду та іншу функціональність, що підвищує наочність та зручність сприйняття інформації. Інтерфейс повинен також бути адаптованим під різні розміри екранів, щоб користувач міг однаково зручно взаємодіяти з помічником на комп'ютері, ноутбукі та інших пристроях.

Ще однією важливою вимогою є можливість інтеграції з існуючими інструментами розробки, такими як IDE, системи контролю версій та баг-трекінгу. Це дозволить програмістам використовувати інтелектуального помічника безпосередньо у їхньому звичному робочому середовищі, мінімізуючи необхідність перемикання між різними програмами. Наприклад, помічник має інтегруватися з популярними IDE, що дозволить отримувати контекстні підказки та допомогу безпосередньо під час написання коду. Крім того, інтеграція з системами контролю версій дозволить автоматично оновлювати базу знань при внесенні змін до документації проєкту, а інтеграція з баг-трекінговими системами дозволить отримувати інформацію про існуючі помилки та способи їх вирішення.

Інтелектуальний помічник програміста має також забезпечувати високу швидкість роботи та мінімальну затримку при виконанні запитів. Затримка в роботі системи може знизити продуктивність розробників та зробити використання помічника незручним. Тому необхідно оптимізувати алгоритми

пошуку та генерації відповідей для забезпечення максимальної продуктивності. Також важливо забезпечити надійність та стабільність роботи системи, щоб уникнути збоїв та втрати даних. Для цього необхідно ретельно протестувати систему на різних видах навантаження та забезпечити наявність механізмів резервного копіювання та відновлення даних.

З точки зору безпеки, важливо забезпечити захист бази знань та конфіденційних даних від несанкціонованого доступу. Інтелектуальний помічник може мати доступ до великої кількості інформації, що становить комерційну таємницю, тому необхідно використовувати надійні механізми шифрування та контролю доступу, що обмежить доступ до чутливих даних тільки авторизованим користувачам.

Нарешті, інтелектуальний помічник має бути гнучким та масштабованим, що дозволяє легко розширювати його функціональність, додаючи нові можливості та інтегруючи його з різними джерелами даних. Архітектура системи має бути модульною, що дозволить легко замінювати окремі компоненти без необхідності переробляти всю систему. Можливість легкого додавання нових функцій та підтримка різних мов програмування та форматів документів є важливим аспектом, що гарантує довгострокову актуальність та корисність інтелектуального помічника.

Розроблений інтелектуальний помічник має бути не просто інструментом для отримання інформації, а справжнім партнером для програміста, що допомагає йому у вирішенні різноманітних задач, підвищує його продуктивність та сприяє створенню якісного та інноваційного програмного забезпечення. Такий підхід дозволить програмістам зосередитися на творчих аспектах процесу розробки, залишивши рутинні завдання інтелектуальному помічнику.

1.4. Обґрунтування вибору програмних засобів та технологій для розробки

Вибір оптимальних програмних засобів та технологій для розробки інтелектуального помічника програміста є критично важливим етапом, що безпосередньо впливає на ефективність, гнучкість, масштабованість та зручність використання кінцевого продукту. В даній роботі цей вибір був зумовлений необхідністю забезпечення високої продуктивності, можливості швидкої розробки та зручної інтеграції з існуючими інструментами та бібліотеками, а також враховував вимоги до точності, релевантності та швидкості роботи системи.

В якості основної мови програмування було обрано Python, що зумовлено рядом її переваг. По-перше, Python має простий та інтуїтивно зрозумілий синтаксис, що полегшує процес розробки, знижує поріг входження для нових розробників та сприяє написанню більш читабельного коду. По-друге, Python є однією з найпопулярніших мов програмування у світі, особливо у сферах штучного інтелекту, машинного навчання та аналізу даних, що забезпечує широку підтримку з боку спільноти розробників, наявність великої кількості навчальних матеріалів та форумів. По-третє, Python має багату екосистему бібліотек та фреймворків для машинного навчання, обробки природної мови та аналізу даних, таких як TensorFlow [\[14\]](#), PyTorch [\[13\]](#), NLTK, SpaCy, Scikit-learn та інші, що є критично важливим для реалізації функціональності інтелектуального помічника.

Для реалізації RAG-архітектури було обрано бібліотеку LlamaIndex [\[3\]](#). Ця бібліотека надає високорівневі та зручні інструменти для створення, управління та використання індексів текстових даних, що є ключовим елементом RAG. LlamaIndex дозволяє легко інтегрувати різні векторні бази даних, такі як FAISS, Annoy, Pinecone та інші, а також підключати різноманітні мовні моделі, включаючи трансформери від Hugging Face, що забезпечує гнучкість та можливість адаптації системи до різних вимог. Використання LlamaIndex дозволяє розробникам зосередитися на логіці

роботи системи, а не на низькорівневій реалізації механізмів індексації та пошуку, що прискорює процес розробки. Для створення інтерфейсу користувача у вигляді чат-бота було обрано фреймворк PyQt. PyQt є одним з найбільш зрілих та функціональних інструментів для розробки графічних інтерфейсів мовою Python, що надає широкий набір віджетів, зручні механізми компонування елементів, просунуті можливості для стилізації інтерфейсу та підтримку крос-платформної розробки. Використання PyQt дозволяє створити інтуїтивно зрозумілий та зручний інтерфейс, який відповідає сучасним стандартам дизайну.

Використання зв'язки Python + LlamaIndex + PyQt дозволяє створити гнучкий, ефективний, масштабований та зручний інструмент, який відповідає високим вимогам, що висуваються до інтелектуального помічника програміста. Кожна з цих технологій є лідером у своїй галузі, має широку підтримку, добре документована та постійно розвивається, що гарантує стабільність, надійність та актуальність розробленого рішення.

Висновки до розділу 1

У першому розділі було проведено ґрунтовний аналіз предметної області, який охоплював огляд технології RAG та її ролі у розробці інтелектуальних систем. Було продемонстровано, що RAG-архітектура є перспективним напрямком, що дозволяє поєднувати сильні сторони інформаційно-пошукових систем та генеративних моделей. Проаналізовано існуючі інструменти для програмістів на основі штучного інтелекту, виявлено їхні переваги та обмеження, що дозволило сформулювати вимоги до власного рішення.

На основі проведеного аналізу було сформульовано основні вимоги до інтелектуального помічника програміста: здатність обробляти запити природною мовою, наявність доступу до великої бази знань, можливість адаптації до індивідуальних потреб, зручний інтерфейс та інтеграція з іншими інструментами розробника. Обґрунтовано вибір програмних засобів

та технологій для розробки: мови програмування Python, бібліотеки LlamaIndex для реалізації RAG-архітектури, бібліотеки `Llama.cpp` для роботи з мовною моделлю Llama, та фреймворку PyQt для створення інтерфейсу користувача.

Використання цих інструментів дозволяє створити гнучкий, ефективний та масштабований продукт. Проведений аналіз та обґрунтування вибору технологій закладають міцний фундамент для подальшого проєктування та розробки інтелектуального помічника програміста. Отримані результати дозволяють перейти до наступного етапу – безпосереднього проєктування та розробки системи, маючи чітке уявлення про її функціональність, архітектуру та інструментальну базу.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНТЕЛЕКТУАЛЬНОГО ПОМІЧНИКА ПРОГРАМІСТА

2.1. Архітектура інтелектуального помічника програміста на основі RAG

Розроблений інтелектуальний помічник програміста базується на архітектурі RAG [\[20\]](#) , яка передбачає тісну інтеграцію механізмів пошуку релевантної інформації у великій базі знань з можливостями генеративних мовних моделей. Таке поєднання дозволяє системі не лише знаходити відповідні дані, але й формувати на їх основі змістовні та контекстуально релевантні відповіді на запити користувачів. Архітектура системи складається з декількох основних компонентів: інтерфейсу користувача, модуля обробки запитів, модуля пошуку інформації, модуля генерації відповідей та бази знань.

Інтерфейс користувача реалізований у вигляді чат-бота, що дозволяє користувачеві ставити запитання природною мовою та отримувати відповіді у текстовому форматі. Інтерфейс забезпечує введення та виведення текстових повідомлень, прокручування історії діалогу та базове форматування тексту. Модуль обробки запитів виступає у ролі посередника між інтерфейсом користувача та іншими модулями системи, виконуючи попередню обробку вхідного запиту. Він проводить аналіз запиту, що включає токенізацію, та інші операції. Результатом роботи модуля є векторне представлення запиту яке використовується для подальшого пошуку інформації.

Модуль пошуку інформації є серцем RAG-архітектури, відповідаючи за пошук релевантних документів у великій базі знань, зображено на рис. 2.1 Він отримує на вхід векторне представлення запиту та використовує його для пошуку семантично близьких документів у проіндексованій базі даних. Для індексації та пошуку документів модуль використовує бібліотеку LlamaIndex. Результатом роботи модуля є відсортований за релевантністю список

документів, що найімовірніше містять відповідь на запит. Модуль генерації відповідей, спираючись на результати роботи модуля пошуку, формує остаточну відповідь на запит користувача. Він базується на мовній моделі Llama, доступ до якої забезпечується через бібліотеку `llama.cpp`.

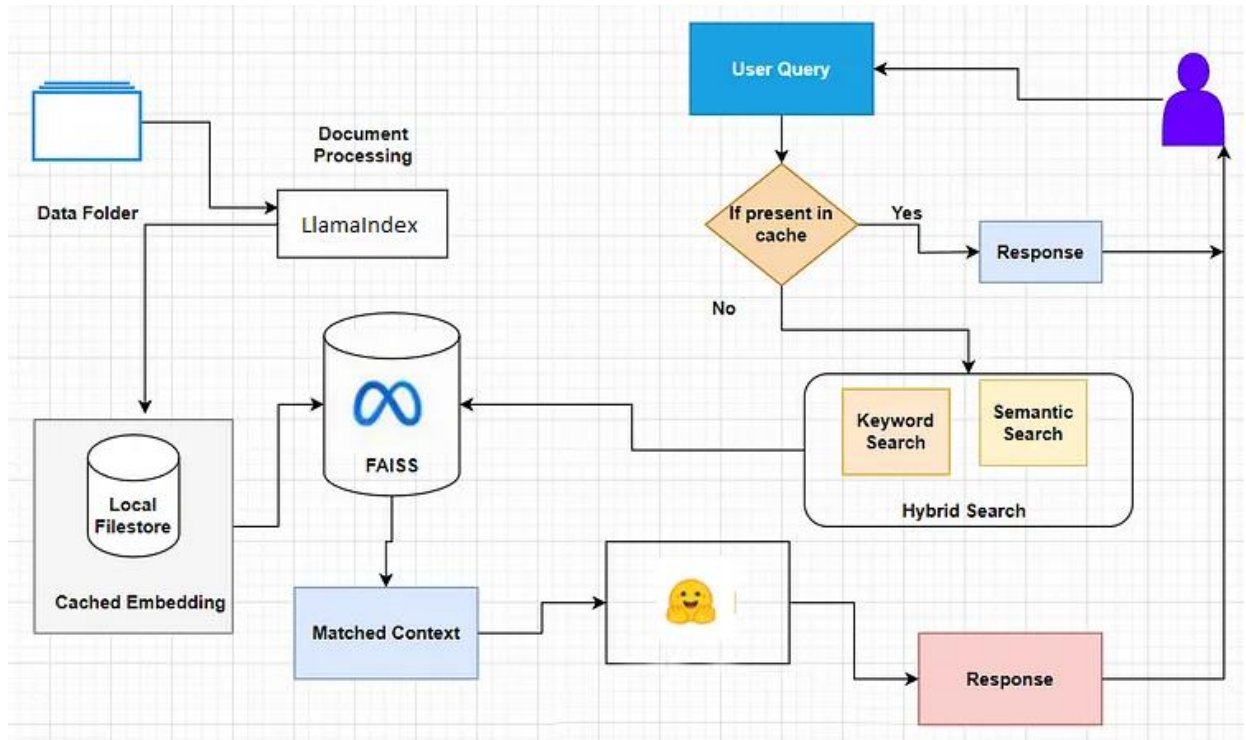


Рис 2.1. Схема програмного рішення формування відповіді RAG

Модуль отримує на вхід оригінальний запит користувача та тексти релевантних документів. На основі цих даних генерується зв'язна, граматично правильна та змістовна відповідь. База знань є сховищем інформації, що використовується системою для пошуку релевантних даних. Вона містить набір документів, що охоплюють різноманітні аспекти програмування, включаючи технічну документацію, приклади коду, статті, дописи з форумів та інші корисні матеріали. База знань постійно оновлюється та поповнюється новою інформацією.

Взаємодія між компонентами системи відбувається у чіткій та логічній послідовності. Користувач вводить запит у текстове поле інтерфейсу чат-бота. Інтерфейс передає запит модулю обробки запитів. Модуль проводить попередню обробку, що включає токенизацію, видалення стоп-слів, стемінг та інші операції, після чого перетворює оброблений запит на векторне

представлення. Отримане векторне представлення запиту передається модулю пошуку інформації. Пошук базується на порівнянні косинусної подібності між вектором запиту та векторами документів. Результатом є відсортований за релевантністю список документів. Цей список, разом із оригінальним запитом, передається модулю генерації відповідей.

Модуль, використовуючи мовну модель Llama, формує контекст для генерації, що складається з конкатенації оригінального запиту та текстів релевантних документів. Сформований контекст передається мовній моделі, яка генерує зв'язну та змістовну відповідь. Відфільтрована відповідь передається до інтерфейсу користувача та відображається у вікні чат-бота.

Така архітектура дозволяє системі ефективно поєднувати швидкий пошук релевантної інформації у великій базі знань з потужністю сучасних мовних моделей для генерації зв'язних та точних відповідей. Модульна структура системи забезпечує гнучкість, розширюваність та дозволяє незалежно вдосконалювати окремі компоненти.

2.2. Взаємодія з векторною базою даних та реалізація пошуку

В рамках розробки інтелектуального помічника програміста було створено зручний та інтуїтивно зрозумілий інтерфейс користувача у форматі чат-бота. Цей інтерфейс, розроблений з використанням бібліотеки PyQt, імітує звичний для більшості користувачів процес спілкування у месенджерах, що робить взаємодію з системою максимально комфортною та природною. Головне вікно програми, як видно на рис. 2.3, поділено на дві основні функціональні частини: область відображення діалогу та область введення повідомлення. Область відображення діалогу займає більшу частину вікна і показує історію листування між користувачем та системою у хронологічному порядку, при цьому повідомлення користувача та відповіді системи візуально розділені за допомогою різних кольорів фону для кращого сприйняття та зручності аналізу.

Область введення повідомлення, розташована в нижній частині вікна, містить текстове поле для введення запиту та кнопку "Відправити".

Користувач може вводити свої запитання природною мовою, використовуючи звичні мовні конструкції та професійний сленг. Інтерфейс підтримує базове форматування тексту, таке як виділення коду моноширинним шрифтом та обрамлення його у спеціальні блоки, що полегшує розуміння відповідей, які містять приклади коду. Також передбачена можливість прокручування історії діалогу, що дозволяє користувачеві у будь-який момент повернутися до попередніх запитань та відповідей. Для очищення поточної розмови передбачена кнопка "Очистити".

Окремо реалізовано сторінку для роботи з файлами, що зображена на рис. 2.4. Ця сторінка надає користувачеві можливість зручно завантажувати та керувати документами, які використовуються для наповнення бази знань системи. Користувач може додавати файли різних форматів, таких як .csv, .doc, .docx, .enex, .epub, .html, .md, .odt, .pdf, .ppt, .pptx, та .txt, що робить систему гнучкою та здатною працювати з різноманітними джерелами інформації. Завантажені документи проходять попередню обробку та індексацію, після чого стають доступними для пошуку та використання при генерації відповідей.

Наповнення векторної бази даних є критично важливим етапом, що забезпечує подальшу ефективність роботи помічника. Цей процес включає збір та попередню обробку текстових даних з різних джерел, таких як технічна документація, приклади коду, статті, навчальні матеріали та обговорення на форумах. Для забезпечення повноти та актуальності бази знань, система підтримує імпорт файлів у багатьох поширених форматах. Для обробки цих різноманітних форматів використовується клас DocumentProcessor, який задіює відповідні завантажувачі (loaders) для коректного зчитування даних з файлів різних типів. Після збору та попередньої обробки, текст кожного документа розбивається на фрагменти, так звані "чанки" згідно з [21], для кожного з яких генерується векторне представлення – ембедінг – за допомогою мовної моделі Llama. Розмір фрагментів та величина перекриття між ними налаштовуються у конфігурації

EmbeddingDatabase, що дозволяє оптимізувати процес індексації та подальшого пошуку.

Згенеровані ембедінги [\[28\]](#), індексуються з використанням бібліотеки FAISS, яка забезпечує швидкий та ефективний пошук найближчих векторів у багатовимірному просторі. Клас FaissIndexManager абстрагує логіку роботи з індексами FAISS, надаючи зручний інтерфейс для створення, наповнення, збереження та пошуку індексів. Система підтримує різні типи індексів, такі як FLAT, IVF_FLAT, HNSW та PQ, а також різні метрики подібності, що дозволяє адаптувати її до конкретних вимог щодо швидкості та точності пошуку.

Для зручності тестування та налагодження роботи системи передбачена окрема форма, що дозволяє взаємодіяти з векторною базою даних, як показано на рис. 2.5. Ця форма надає можливість виконувати пошукові запити до обраної бази даних та переглядати результати пошуку, включаючи ідентифікатори знайдених фрагментів, відстані між векторами та сам текст фрагментів. Такий інструмент дозволяє розробникам та адміністраторам системи аналізувати роботу механізму пошуку, перевіряти якість індексації та налаштовувати параметри для досягнення оптимальної продуктивності.

Загальна структура програми базується на модульній архітектурі, де кожен модуль реалізований у вигляді окремого класу та відповідає за чітко визначену функціональну область. Такий підхід забезпечує гнучкість, легкість розширення функціоналу та спрощує підтримку коду. Центральним елементом системи є клас ApplicationController, який виступає в ролі головного координатора, керуючи взаємодією між користувацьким інтерфейсом, модулями обробки запитів, пошуку інформації, генерації відповідей та базою даних.

Класи ChatWindow та ChatWidget забезпечують роботу інтерфейсу чат-бота, надаючи користувачеві зручні засоби для введення запитів та отримання відповідей. ChatWindow представляє собою головне вікно

програми, що містить область відображення діалогу, поле введення повідомлень та інші елементи керування. ChatWidget, в свою чергу, відповідає за візуальне представлення окремого чату в інтерфейсі, дозволяючи користувачеві перемикатися між різними діалогами.

Клас ConfigManager відповідає за управління конфігураційними параметрами системи. Він зчитує налаштування з файлу config.ini, забезпечуючи централізоване зберігання та доступ до конфігурації для всіх модулів системи. ConfigManager дозволяє легко змінювати параметри роботи програми, такі як шляхи до моделей, налаштування бази даних, параметри індексації та генерації, без необхідності внесення змін у вихідний код.

Доступ до реляційної бази даних, де зберігаються дані про користувачів, чати, повідомлення та конфігурації, забезпечується класом DatabaseManager. Цей клас використовує бібліотеку SQLAlchemy для об'єктно-реляційного відображення (ORM), що дозволяє працювати з сутностями бази даних як з об'єктами Python. DatabaseManager інкапсулює логіку виконання SQL-запитів та забезпечує цілісність даних.

Для реалізації функціональності RAG-архітектури ApplicationController взаємодіє з класом VectorDBController, який керує роботою з векторною базою даних. VectorDBController забезпечує ініціалізацію та конфігурацію індексів FAISS, використовуючи бібліотеку LlamaIndex. Він також відповідає за додавання нових документів до бази знань, виконання семантичного пошуку та оновлення індексів при зміні даних.

При отриманні запиту на пошук, ApplicationController передає його VectorDBController, який, у свою чергу, звертається до відповідного об'єкта EngineEmbeddingDatabase. Цей клас є обгорткою для роботи з індексами FAISS та моделями для генерації векторних представлень тексту. EngineEmbeddingDatabase забезпечує розбиття тексту на фрагменти, генерацію ембеддінгів за допомогою ModelController, а також додавання векторів до індексу FAISS та виконання пошуку найближчих сусідів.

Для генерації векторних представлень тексту, EngineEmbeddingDatabase використовує клас ModelController, який завантажує та керує роботою мовної моделі Llama. ModelController забезпечує взаємодію з бібліотекою llama.cpp, передаючи моделі вхідні дані та отримуючи згенеровані ембеддинги. Він також відповідає за налаштування параметрів моделі, таких як temperature, top_k, top_p та інших, що впливають на якість генерації.

EngineEmbeddingDatabase, використовуючи FaissIndexManager, виконує пошук найближчих векторів у індексі FAISS, що відповідають запиту користувача. FaissIndexManager інкапсулює низькорівневу логіку роботи з індексами FAISS, надаючи зручний API для додавання, пошуку, збереження та завантаження індексів. Після отримання результатів пошуку від FaissIndexManager, EngineEmbeddingDatabase формує список релевантних документів та повертає його VectorDBController.

VectorDBController передає список знайдених документів назад до ApplicationController, який, у свою чергу, формує контекст для генерації відповіді. Контекст включає оригінальний запит користувача, історію діалогу та тексти найбільш релевантних документів. Потім ApplicationController передає сформований контекст до ModelController, який використовує мовну модель Llama та бібліотеку llama.cpp для генерації зв'язної та змістовної відповіді.

Згенерована відповідь повертається до ApplicationController, який може виконати її додаткову обробку, наприклад, переклад на мову користувача за допомогою TranslationModel. Після цього, ApplicationController передає відповідь до ChatWindow для відображення користувачеві. ChatWindow оновлює область відображення діалогу, додаючи нове повідомлення від системи. Таким чином, відбувається повний цикл обробки запиту користувача та генерації відповіді.

Для забезпечення зручного налаштування параметрів системи та управління моделями та базами даних, у програмі передбачено окремі

віджети, такі як `ModelsWidget`, `ModelWidget`, `RAGWidget`, `RAGModelWidget` та `RAGDBWidget`. Ці віджети дозволяють користувачеві через графічний інтерфейс додавати, видаляти та редагувати моделі, конфігурувати параметри індексації та пошуку, а також керувати базою знань. Взаємодія між віджетами та іншими компонентами системи здійснюється через механізм сигналів та слотів, що є стандартним підходом у `PyQt`.

Реляційна база даних, діаграма якої зображена на рис. 2.6, є важливою частиною архітектури, що забезпечує структуроване зберігання інформації, необхідної для функціонування системи. База даних включає таблиці для зберігання даних про користувачів, чати, повідомлення, а також дані, специфічні для RAG-архітектури, такі як конфігураційні параметри мовних моделей, інформація про векторні індекси та метадані документів. Використання реляційної моделі дозволяє підтримувати зв'язки між сутностями, забезпечувати цілісність даних та виконувати складні запити.

Зв'язок між таблицями `EmbeddingDatabase` та `ChatMessageModel` через `RagMessageModel` дозволяє відстежувати, які саме документи та індекси були використані для генерації відповіді на кожне конкретне повідомлення. Це забезпечує прозорість роботи системи та дозволяє аналізувати процес генерації відповідей. Такий підхід робить систему не тільки ефективною, але й легкою для моніторингу та вдосконалення.

Важливою деталлю реалізації механізму оновлення є те, що для індексації та пошуку документів у системі використовується бібліотека `LlamaIndex`. Ця бібліотека надає зручний високорівневий інтерфейс для роботи з векторними базами даних, включаючи `FAISS`. `LlamaIndex` автоматично створює та конфігурує індекс `FAISS`, а також забезпечує додавання векторних представлень документів до цього індексу. Розробнику не потрібно безпосередньо взаємодіяти з API `FAISS`, оскільки `LlamaIndex` бере на себе всі необхідні операції.

Для забезпечення цілісності даних під час оновлення, система використовує механізм транзакцій. Це означає, що всі зміни, що вносяться до бази даних (додавання, видалення або оновлення документів), виконуються як єдина атомарна операція. Якщо на якомусь етапі процесу оновлення відбувається помилка, всі зміни, зроблені в рамках цієї транзакції, автоматично відкочуються, гарантуючи, що база даних залишиться в консистентному стані.

Після успішного завершення процесу оновлення, система зберігає індекс FAISS на диск. Це дозволяє уникнути необхідності повторної індексації всіх документів при наступному запуску системи, що значно економить час та обчислювальні ресурси. Збереження індексу здійснюється за допомогою функціональності, вбудованої в LlamaIndex, яка автоматично серіалізує індекс FAISS у файл на диску. При наступному запуску системи індекс завантажується з файлу, що дозволяє негайно розпочати роботу без необхідності повторного індексування.

Окрім основного функціоналу, пов'язаного з додаванням, видаленням та оновленням документів, механізм оновлення також включає можливості для резервного копіювання та відновлення даних. Це дозволяє захистити базу знань від втрати або пошкодження у випадку апаратних збоїв або помилок у програмному забезпеченні. Резервне копіювання здійснюється шляхом створення копії індексу FAISS та збереження її на зовнішньому носії або в хмарному сховищі.

Відновлення даних виконується з використанням резервної копії, що дозволяє повернути систему до попереднього стану. Варто зазначити, що механізм оновлення векторної бази даних є асинхронним, тобто він виконується у фоновому режимі, не блокуючи основний потік виконання програми. Це дозволяє користувачам продовжувати взаємодію з інтелектуальним помічником під час оновлення бази знань, що підвищує зручність використання системи.

Адміністратор може відстежувати прогрес оновлення через спеціальний інтерфейс, який відображає інформацію про кількість оброблених документів, швидкість індексації та можливі помилки, що виникли в процесі. Для забезпечення безпеки даних, що зберігаються у векторній базі, механізм оновлення підтримує можливість шифрування індексу FAISS та метаданих документів. Це дозволяє захистити конфіденційну інформацію від несанкціонованого доступу.

Ключі шифрування зберігаються в захищеному сховищі, доступ до якого мають лише авторизовані користувачі. Такий підхід гарантує, що навіть у випадку витоку даних, зловмисники не зможуть отримати доступ до вмісту документів без відповідного ключа. Розроблений механізм оновлення векторної бази даних є гнучким та масштабованим, що дозволяє адаптувати його до зростаючих обсягів даних та змінюваних вимог до системи.

Завдяки модульній архітектурі, можна легко додавати нові функції та вдосконалювати існуючі без необхідності внесення змін до інших частин системи. Це забезпечує можливість постійного розвитку та вдосконалення інтелектуального помічника програміста, підтримуючи його актуальність та ефективність протягом тривалого часу. Важливою особливістю розробленого механізму є його здатність інтегруватися з різними джерелами даних.

Окрім локальних файлів, система підтримує можливість отримання даних з віддалених ресурсів, таких як веб-сайти, бази даних або API сторонніх сервісів. Це дозволяє автоматизувати процес збору інформації та забезпечити постійне оновлення бази знань найактуальнішими даними. Наприклад, система може періодично сканувати певні веб-ресурси на наявність нових статей або оновлень документації та автоматично додавати їх до векторної бази.

Реалізований механізм оновлення векторної бази даних є важливим компонентом інтелектуального помічника програміста, що забезпечує його здатність адаптуватися до змін у сфері розробки програмного забезпечення

та надавати користувачам найактуальнішу та найрелевантнішу інформацію. Ретельне проєктування та імплементація цього механізму, з урахуванням аспектів безпеки, масштабованості та зручності використання, дозволили створити надійний та ефективний інструмент, що є невід'ємною частиною успішного функціонування всієї системи.

Механізм пошуку релевантної інформації є одним з найважливіших компонентів інтелектуального помічника програміста, оскільки від його ефективності та точності залежить якість відповідей, які система надає користувачеві. Розробка цього механізму включає декілька ключових етапів: підготовку даних, індексацію документів та реалізацію функції пошуку. Першим етапом є збір, підготовка та формування корпусу документів, що складатимуть базу знань системи.

Документи різних форматів, таких як .csv, .doc, .docx, .enex, .epub, .html, .md, .odt, .pdf, .ppt, .pptx та .txt, завантажуються та обробляються за допомогою класу `DocumentProcessor`. Цей клас використовує відповідні завантажувачі (`loaders`), визначені в словнику `LOADER_MAPPING`, для коректного зчитування даних з файлів різних типів. Для забезпечення швидкого та ефективного пошуку інформації у великому обсязі текстових даних було використано бібліотеку FAISS.

FAISS дозволяє створювати індекси для векторних представлень даних та виконувати швидкий пошук найближчих сусідів. Керування індексами FAISS здійснюється через клас `FaissIndexManager`, який надає зручний інтерфейс для створення, наповнення та збереження індексів, а також для виконання пошуку. Процес індексації включає декілька кроків. Спочатку, документи розбиваються на невеликі, семантично зв'язані фрагменти (текстові блоки).

Ця операція виконується методом `split_text` класу `EngineEmbeddingDatabase`. Розмір фрагментів (`chunk_size`) та величина перекриття (`overlap`) між ними, які впливають на деталізацію та контекстну

повноту індексованих даних, задаються в конфігурації `EmbeddingDatabase`. Далі, для кожного фрагмента тексту генерується його векторне представлення – ембеддінг.

Для створення ембеддінгів використовується мовна модель `Llama`, доступ до якої забезпечується через бібліотеку `Llama_cpp`. Модель ініціалізується з параметром `embedding=True`, що вказує на її використання саме для генерації векторних представлень. При створенні ембеддінгу модель `Llama` перетворює текст у вектор фіксованої розмірності, зберігаючи при цьому семантичну близькість між фрагментами, тобто вектори семантично близьких текстів будуть розташовані близько один до одного у векторному просторі.

Отримані векторні представлення додаються до індексу `FAISS`. Клас `FaissIndexManager` підтримує різні типи індексів, такі як `FLAT` (повний перебір), `IVF_FLAT` (інвертований індекс), `HNSW` (ієрархічний навігаційний граф) та `PQ` (квантування добутків), а також різні метрики подібності, такі як `L2` (Евклідова відстань) та `IP` (внутрішній добуток). Вибір типу індексу та метрики залежить від специфіки задачі, вимог до швидкості та точності пошуку, а також від обсягу даних.

Параметри індексу, включаючи його тип, метрику, розмірність векторів та інші специфічні налаштування, конфігуруються через об'єкт `EmbeddingDatabase`. Важливою деталлю є те, що конфігурація індексу та моделі ембеддінгу зберігається в об'єктах `EmbeddingDatabase` та `EmbeddingModel` відповідно. Це дозволяє легко керувати параметрами індексації та пошуку, а також забезпечує цілісність та узгодженість даних у системі.

Клас `EngineEmbeddingDatabase` відповідає за логіку роботи з індексами та моделями ембеддінгу для кожної бази даних, включаючи розбиття тексту на фрагменти, додавання фрагментів до індексу, ініціалізацію та виклик моделі `Llama` для ембеддінгу, а також взаємодію з об'єктом

FaissIndexManager для виконання операцій з індексом. Після створення, індекс зберігається на диску у файловій системі для подальшого використання. Це дозволяє уникнути необхідності повторної індексації документів при кожному запуску програми, що значно економить час.

Пошук релевантної інформації здійснюється шляхом порівняння векторного представлення запиту користувача з векторами фрагментів тексту, що зберігаються в індексі. При надходженні запиту від користувача, він спочатку проходить етап попередньої обробки, аналогічний тому, що застосовувався при індексації документів. Потім запит перетворюється на вектор за допомогою тієї ж самої моделі Llama, яка використовувалася для створення ембеддінгів документів.

Для виконання пошуку найближчих векторів в індексі FAISS використовується метод `search` класу `FaissIndexManager`. Цей метод приймає на вхід вектор запиту та параметр `top_k`, що визначає кількість найближчих сусідів, яких необхідно повернути. Результатом роботи методу є масив відстаней та масив індексів знайдених векторів, відсортованих за зростанням відстані (тобто за спаданням подібності).

Клас `EngineEmbeddingDatabase` використовує метод `search` для безпосереднього виконання пошуку, передаючи йому вектор запиту та отримуючи результати. На основі індексів та відстаней, повернутих `FaissIndexManager`, формується список об'єктів `SearchResult`. Кожен об'єкт `SearchResult` містить ідентифікатор знайденого фрагмента, відстань до вектора запиту, а також додаткову інформацію, таку як ідентифікатор бази даних (`db_id`), до якої належить фрагмент.

Клас `VectorDBController` відіграє важливу роль у координації взаємодії між компонентами системи, пов'язаними з векторною базою даних. Він відповідає за створення та управління об'єктами `EngineEmbeddingDatabase`, обробку запитів на пошук, а також за забезпечення зв'язку між модулем пошуку та іншими частинами інтелектуального помічника. Таким чином,

механізм пошуку, реалізований у програмі, використовує зв'язку бібліотек `Llama_cpp` (для генерації векторних представлень текстів – ембедінгів) та `FAISS` (для ефективної індексації та швидкого пошуку векторів), керовану через допоміжні класи `DocumentProcessor`, `EngineEmbeddingDatabase`, `FaissIndexManager` та `VectorDBController`.

2.2. Реалізація інтерфейсу чат-бота та взаємодія модулів в архітектурі RAG

Оптимізація параметрів мовної моделі є ключовим аспектом розробки інтелектуального помічника програміста, оскільки вона безпосередньо впливає на якість генерації відповідей. Мовна модель `Llama`, подібно до більшості сучасних трансформерних моделей, має ряд параметрів, що регулюють процес генерації тексту. Правильний підбір значень цих параметрів дозволяє досягти оптимального балансу між швидкістю, креативністю та точністю відповідей, адаптуючи поведінку моделі до конкретних потреб та умов використання.

Одним з найбільш важливих параметрів є `temperature` (температура), який контролює ступінь випадковості при виборі наступного токена в процесі генерації. Значення `temperature` може змінюватися від 0 до нескінченності, але зазвичай використовується діапазон від 0 до 1 або трохи вище. При `temperature`, рівній 0, модель завжди обиратиме найбільш імовірний наступний токен, що призводить до детермінованої та передбачуваної генерації. Збільшення значення `temperature` вносить елемент випадковості, дозволяючи моделі обирати менш імовірні, але потенційно більш креативні варіанти. Високі значення `temperature` (більше 1) роблять згенерований текст більш різноманітним та непередбачуваним, але можуть призвести до появи безглузвих або нелогічних послідовностей. Значення в діапазоні 0.7-0.9 часто вважаються оптимальними для забезпечення балансу між креативністю та зв'язністю тексту.

Іншим важливим параметром є `top_k`, який обмежує кількість найбільш імовірних наступних tokenів, з яких модель робить вибір на кожному кроці генерації. При `top_k`, рівному 1, модель завжди обиратиме найбільш імовірний token, що еквівалентно `temperature`, рівній 0. При `top_k`, рівному 0, модель враховує всі можливі токени. Зазвичай, значення `top_k` встановлюють в діапазоні від 10 до 100. Менші значення `top_k` роблять генерацію більш детермінованою та сфокусованою, тоді як більші значення дозволяють моделі розглядати ширший спектр варіантів, що може призвести до більш різноманітних, але потенційно менш релевантних результатів.

Параметр `top_p`, також відомий як `nucleus sampling`, є альтернативним способом обмеження кількості варіантів при виборі наступного токена. Замість фіксованої кількості tokenів, `top_p` визначає поріг ймовірності. Модель сортує всі можливі токени за спаданням ймовірності та обирає з них лише ті, сумарна ймовірність яких не перевищує порогове значення `top_p`. Наприклад, при `top_p`, рівному 0.9, модель обиратиме токени з пулу, сумарна ймовірність яких становить 90%. Цей підхід дозволяє адаптивно звужувати або розширювати кількість варіантів залежно від розподілу ймовірностей на кожному кроці генерації. Зазвичай, значення `top_p` встановлюють в діапазоні від 0.7 до 0.95.

Ще одним важливим параметром є `repetition_penalty` (штраф за повторення). Цей параметр дозволяє контролювати схильність моделі до повторення одних і тих самих слів або фраз. При значенні `repetition_penalty` більше 1, модель буде штрафуватися за повторення tokenів, що вже зустрічалися в згенерованому тексті, причому штраф буде тим більшим, чим ближче до поточної позиції знаходиться попереднє входження токена. Це дозволяє уникнути небажаних повторів та зробити згенерований текст більш різноманітним. Зазвичай, значення `repetition_penalty` встановлюють в діапазоні від 1.0 до 1.2.

Параметр `max_tokens` обмежує максимальну кількість токенів, яку може згенерувати модель. Цей параметр є корисним для контролю довжини відповіді та запобігання надмірно довгій генерації. Значення `max_tokens` підбирається залежно від конкретної задачі та вимог до довжини відповіді. Окрім перерахованих вище параметрів, існують й інші, менш поширені, але також потенційно корисні для тонкого налаштування процесу генерації. До них належать, наприклад, `presence_penalty` та `frequency_penalty`, що дозволяють штрафувати модель за наявність або частоту певних токенів у згенерованому тексті, `mirostat` - алгоритм, що динамічно підлаштовує температуру під час генерації для підтримання заданого рівня перплексії, та інші.

Важливо розуміти, що всі ці параметри взаємопов'язані та впливають один на одного. Наприклад, збільшення `temperature` може частково нівелюватися зменшенням `top_k` або `top_p`. Тому, для досягнення оптимального результату, необхідно підбирати значення параметрів комплексно, враховуючи їх взаємний вплив. Процес оптимізації параметрів зазвичай включає проведення серії експериментів з різними комбінаціями значень, генерацію тексту для кожного набору параметрів та оцінку якості отриманих результатів за допомогою автоматичних метрик (наприклад, BLEU, ROUGE) або шляхом ручного аналізу. Результати експериментів можуть бути представлені у вигляді таблиць або графіків, що демонструють залежність якості генерації від значень тих чи інших параметрів.

Оптимальні значення параметрів залежать від багатьох факторів, таких як розмір та архітектура мовної моделі, обсяг та якість навчальних даних, а також специфіка конкретної задачі. Для інтелектуального помічника програміста, як правило, бажано отримувати точні та конкретні відповіді, тому рекомендується використовувати відносно низькі значення `temperature` (наприклад, 0.5-0.7) та `top_p` (наприклад, 0.7-0.9), а також помірні значення `top_k` (наприклад, 40-60). Підбір оптимальних параметрів є ітеративним процесом, що вимагає ретельного аналізу результатів та тонкого

налаштування. Важливо не лише досягти високої якості генерації, але й забезпечити прийнятну швидкість роботи моделі.

Механізм генерації відповідей є центральним компонентом інтелектуального помічника програміста, оскільки саме він відповідає за формування змістовної, точної та корисної відповіді на запит користувача. Цей механізм базується на принципах архітектури RAG, яка передбачає тісну інтеграцію процесів пошуку релевантної інформації у зовнішній базі знань та генерації тексту з використанням мовних моделей. Для імплементації механізму генерації відповідей було обрано мовну модель `aya-expanse-8b`, доступ до якої забезпечується через бібліотеку `llama.cpp`.

Вибір моделі `aya-expanse-8b` зумовлений її високою ефективністю у задачах генерації тексту, здатністю розуміти та опрацьовувати природну мову, а також можливістю запуску моделі локально на центральному процесорі (CPU), без необхідності використання спеціалізованих графічних прискорювачів (GPU). Це робить систему більш доступною для користувачів, які не мають у своєму розпорядженні високопродуктивного обладнання з потужними GPU. Процес генерації відповіді, як вже згадувалось раніше, складається з декількох етапів: формування контексту, безпосередньо генерація та подальша обробка і фільтрація результату.

Формування контексту є першим і дуже важливим кроком, оскільки від якості контексту залежить релевантність та точність згенерованої відповіді. Контекст складається з двох частин: оригінального запиту користувача та текстів найбільш релевантних документів, знайдених модулем пошуку інформації. Ці тексти об'єднуються в єдиний текстовий фрагмент, причому між запитом та контекстною інформацією вставляються спеціальні токени-роздільники. Такий підхід дозволяє моделі враховувати як сам запит, так і знайдену релевантну інформацію при генерації відповіді, забезпечуючи тим самим контекстуальну залежність та змістовність результату.

Сформований контекст передається мовній моделі `aya-expanse-8b`, яка генерує відповідь, спираючись на надану інформацію. Для взаємодії з моделлю та запуску процесу генерації використовується метод `create_chat_completion_openai_v1` з бібліотеки `llama.cpp`, який дозволяє зручно взаємодіяти з моделлю у режимі чату, передаючи їй повідомлення та отримуючи згенерований текст. Параметр `temperature`, про який йшлося раніше, у цьому коді регулює ступінь випадковості при генерації тексту, дозволяючи налаштовувати баланс між креативністю та передбачуваністю.

Згенерована відповідь може містити деякі надлишкові або нерелевантні фрагменти, тому на етапі обробки та фільтрації відповіді видаляються потенційно небажані частини тексту, такі як посилання на джерела, що не стосуються суті запиту, або повторення. Такий підхід до генерації відповідей дозволяє органічно поєднати переваги великих мовних моделей, здатних генерувати зв'язний та граматично правильний текст, з точністю та фактологічною обґрунтованістю інформаційно-пошукових систем. Завдяки використанню RAG-архітектури, інтелектуальний помічник не лише оперує знаннями, закладеними в модель під час навчання, але й активно звертається до зовнішніх джерел інформації, що робить його відповіді більш повними, актуальними та достовірними, як зазначено у джерелі [21].

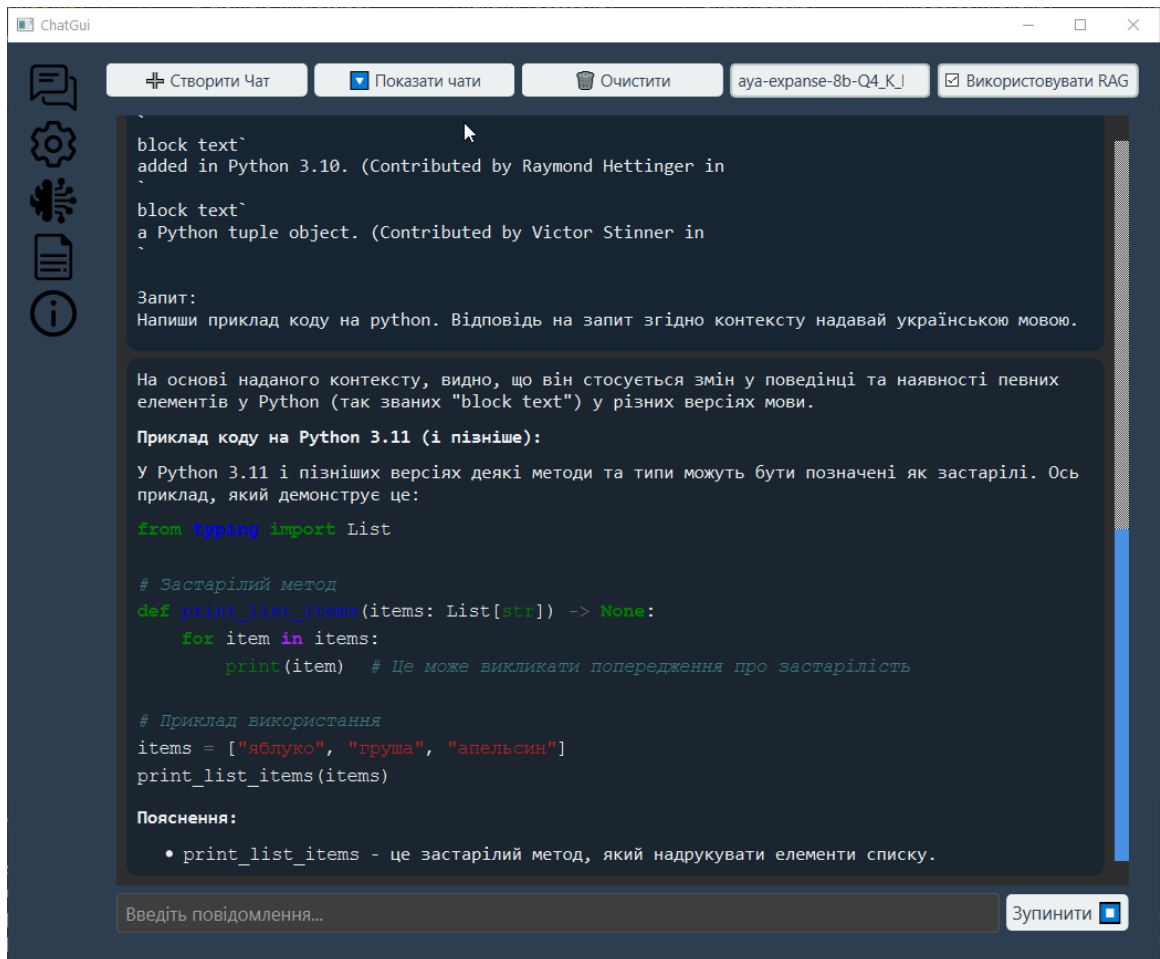


Рис 2.2. Приклад генерації відповіді на основі RAG

На рис. 2.2 зображено приклад генерації відповіді на основі RAG, де можна побачити, як саме контекст, що складається з запиту та релевантних документів, передається мовній моделі, і як модель формує відповідь на основі цього контексту. Такий підхід дозволяє генерувати відповіді, які є не лише граматично правильними та зв'язними, але й враховують контекст запиту та базуються на актуальній інформації з зовнішніх джерел.

Ретельний підбір та оптимізація параметрів мовної моделі, а також використання RAG-архітектури, дозволяють механізму генерації відповідей забезпечувати високу якість, точність та релевантність результатів. Це робить інтелектуального помічника ефективним інструментом для програмістів, що допомагає їм у вирішенні різноманітних завдань, пов'язаних з розробкою програмного забезпечення, та сприяє підвищенню продуктивності їхньої праці.

2.3. Загальна архітектура та структура програми

В рамках розробки інтелектуального помічника програміста було створено зручний та інтуїтивно зрозумілий інтерфейс користувача у форматі чат-бота. Цей інтерфейс, розроблений з використанням бібліотеки PyQt, імітує звичний для більшості користувачів процес спілкування у месенджерах, що робить взаємодію з системою максимально комфортною та природною. Головне вікно програми, як видно на рис. 2.3, поділено на дві основні функціональні частини: область відображення діалогу та область введення повідомлення. Область відображення діалогу займає більшу частину вікна і показує історію листування між користувачем та системою у хронологічному порядку, при цьому повідомлення користувача та відповіді системи візуально розділені за допомогою різних кольорів фону для кращого сприйняття та зручності аналізу.

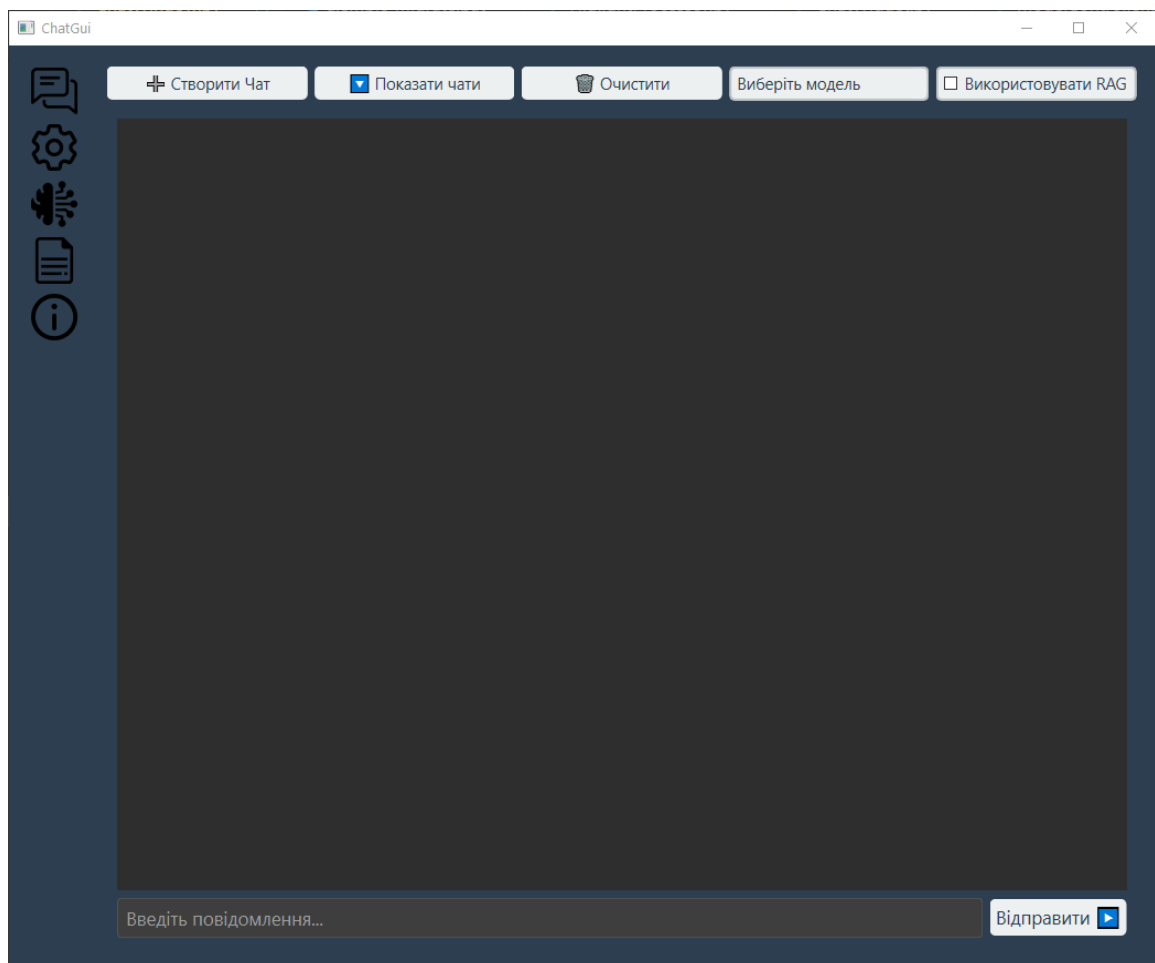


Рис 2.3. Головне вікно програми

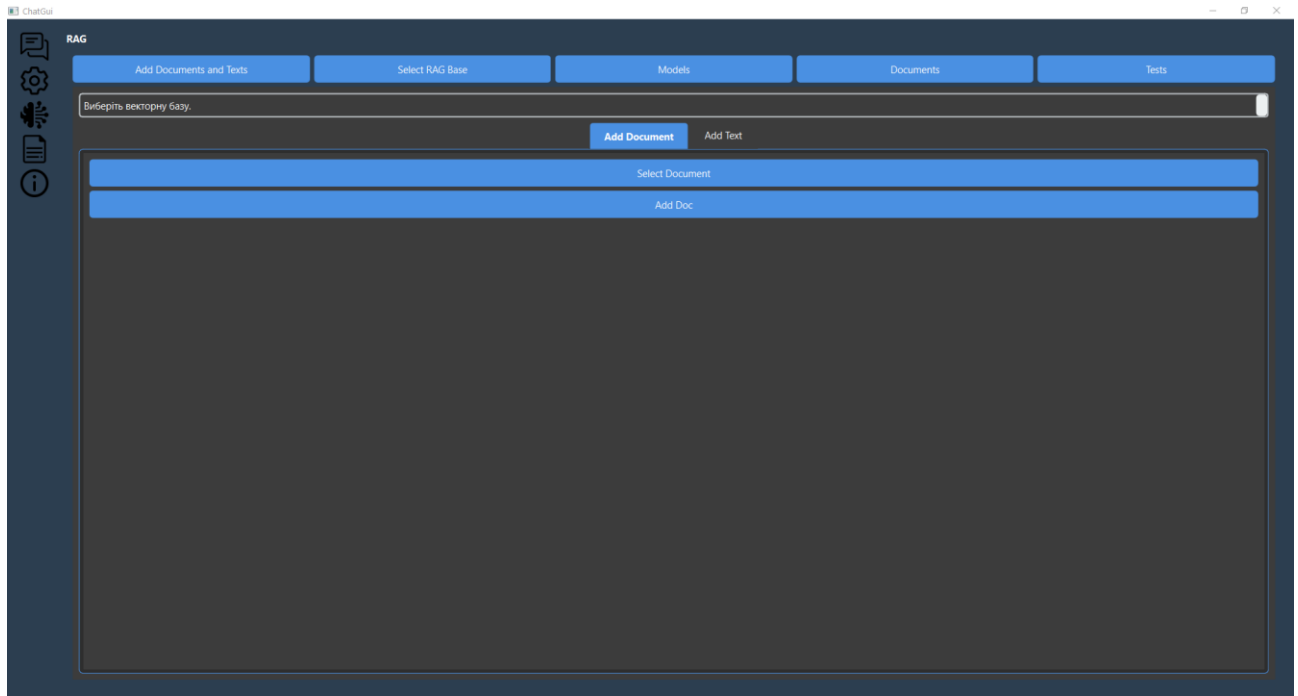


Рис 2.4 Сторінка додавання документів

Область введення повідомлення, розташована в нижній частині вікна, містить текстове поле для введення запиту та кнопку "Відправити". Користувач може вводити свої запитання природною мовою, використовуючи звичні мовні конструкції та професійний сленг. Інтерфейс підтримує базове форматування тексту, таке як виділення коду моноширинним шрифтом та обрамлення його у спеціальні блоки, що полегшує розуміння відповідей, які містять приклади коду. Також передбачена можливість прокручування історії діалогу, що дозволяє користувачеві у будь-який момент повернутися до попередніх запитань та відповідей. Для очищення поточної розмови передбачена кнопка "Очистити".

Окремо реалізовано сторінку для роботи з файлами, що зображена на рис. 2.4. Ця сторінка надає користувачеві можливість зручно завантажувати та керувати документами, які використовуються для наповнення бази знань системи. Користувач може додавати файли різних форматів, таких як .csv, .doc, .docx, .enex, .epub, .html, .md, .odt, .pdf, .ppt, .pptx, та .txt, що робить систему гнучкою та здатною працювати з різноманітними джерелами інформації. Завантажені документи проходять попередню обробку та

індексацію, після чого стають доступними для пошуку та використання при генерації відповідей. Наповнення векторної бази даних є критично важливим етапом, що забезпечує подальшу ефективність роботи помічника. Цей процес включає збір та попередню обробку текстових даних з різних джерел, таких як технічна документація, приклади коду, статті, навчальні матеріали та обговорення на форумах. Для забезпечення повноти та актуальності бази знань, система підтримує імпорт файлів у багатьох поширених форматах.

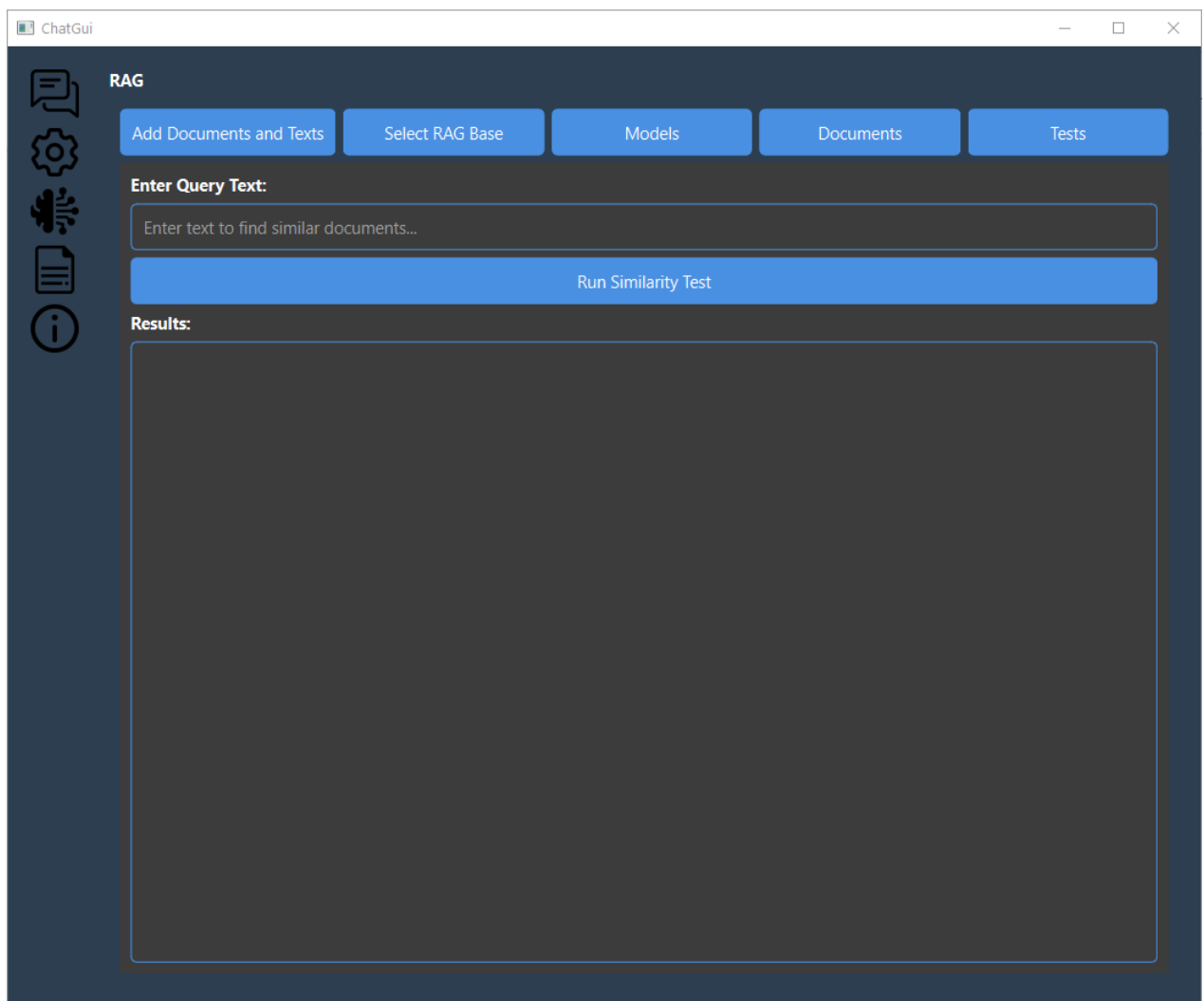


Рис 2.5. Форма тестування існуючої векторної бази

Для обробки цих різноманітних форматів використовується клас `DocumentProcessor`, який задіює відповідні завантажувачі (loaders) для коректного зчитування даних з файлів різних типів. Після збору та попередньої обробки, текст кожного документа розбивається на фрагменти, так звані "чанки" згідно з [21], для кожного з яких генерується векторне

представлення – ембеддінг – за допомогою мовної моделі Llama. Розмір фрагментів та величина перекриття між ними налаштовуються у конфігурації `EmbeddingDatabase`, що дозволяє оптимізувати процес індексації та подальшого пошуку. Згенеровані ембеддінги індексуються з використанням бібліотеки FAISS, яка забезпечує швидкий та ефективний пошук найближчих векторів у багатовимірному просторі. Клас `FaissIndexManager` абстрагує логіку роботи з індексами FAISS, надаючи зручний інтерфейс для створення, наповнення, збереження та пошуку індексів. Система підтримує різні типи індексів, такі як FLAT, IVF_FLAT, HNSW та PQ, а також різні метрики подібності, що дозволяє адаптувати її до конкретних вимог щодо швидкості та точності пошуку.

Для зручності тестування та налагодження роботи системи передбачена окрема форма, що дозволяє взаємодіяти з векторною базою даних, як показано на рис. 2.5. Ця форма надає можливість виконувати пошукові запити до обраної бази даних та переглядати результати пошуку, включаючи ідентифікатори знайдених фрагментів, відстані між векторами та сам текст фрагментів. Такий інструмент дозволяє розробникам та адміністраторам системи аналізувати роботу механізму пошуку, перевіряти якість індексації та налаштовувати параметри для досягнення оптимальної продуктивності. Загальна структура програми базується на модульній архітектурі, де кожен модуль реалізований у вигляді окремого класу та відповідає за чітко визначену функціональну область. Такий підхід забезпечує гнучкість, легкість розширення функціоналу та спрощує підтримку коду. Центральним елементом системи є клас `AppController`, який виступає в ролі головного координатора, керуючи взаємодією між користувацьким інтерфейсом, модулями обробки запитів, пошуку інформації, генерації відповідей та базою даних.

Класи `ChatWindow` та `ChatWidget` забезпечують роботу інтерфейсу чат-бота, надаючи користувачеві зручні засоби для введення запитів та отримання відповідей. `ChatWindow` представляє собою головне вікно

програми, що містить область відображення діалогу, поле введення повідомлень та інші елементи керування. ChatWidget, в свою чергу, відповідає за візуальне представлення окремого чату в інтерфейсі, дозволяючи користувачеві перемикатися між різними діалогами. Клас ConfigManager відповідає за управління конфігураційними параметрами системи. Він зчитує налаштування з файлу config.ini, забезпечуючи централізоване зберігання та доступ до конфігурації для всіх модулів системи. ConfigManager дозволяє легко змінювати параметри роботи програми, такі як шляхи до моделей, налаштування бази даних, параметри індексації та генерації, без необхідності внесення змін у вихідний код.

Доступ до реляційної бази даних, де зберігаються дані про користувачів, чати, повідомлення та конфігурації, забезпечується класом DatabaseManager. Цей клас використовує бібліотеку SQLAlchemy для об'єктно-реляційного відображення (ORM), що дозволяє працювати з сутностями бази даних як з об'єктами Python. DatabaseManager інкапсулює логіку виконання SQL-запитів та забезпечує цілісність даних. Для реалізації функціональності RAG-архітектури AppController взаємодіє з класом VectorDBController, який керує роботою з векторною базою даних. VectorDBController забезпечує ініціалізацію та конфігурацію індексів FAISS, використовуючи бібліотеку LlamaIndex. Він також відповідає за додавання нових документів до бази знань, виконання семантичного пошуку та оновлення індексів при зміні даних.

При отриманні запиту на пошук, AppController передає його VectorDBController, який, у свою чергу, звертається до відповідного об'єкта EngineEmbeddingDatabase. Цей клас є обгорткою для роботи з індексами FAISS та моделями для генерації векторних представлень тексту. EngineEmbeddingDatabase забезпечує розбиття тексту на фрагменти, генерацію ембеддінгів за допомогою ModelController, а також додавання векторів до індексу FAISS та виконання пошуку найближчих сусідів. Для генерації векторних представлень тексту, EngineEmbeddingDatabase

використовує клас `ModelController`, який завантажує та керує роботою мовної моделі `Llama`. `ModelController` забезпечує взаємодію з бібліотекою `Llama.cpp`, передаючи моделі вхідні дані та отримуючи згенеровані ембедінги. Він також відповідає за налаштування параметрів моделі, таких як `temperature`, `top_k`, `top_p` та інших, що впливають на якість генерації.

`EngineEmbeddingDatabase`, використовуючи `FaissIndexManager`, виконує пошук найближчих векторів у індексі `FAISS`, що відповідають запиту користувача. `FaissIndexManager` інкапсулює низькорівневу логіку роботи з індексами `FAISS`, надаючи зручний API для додавання, пошуку, збереження та завантаження індексів. Після отримання результатів пошуку від `FaissIndexManager`, `EngineEmbeddingDatabase` формує список релевантних документів та повертає його `VectorDBController`. `VectorDBController` передає список знайдених документів назад до `AppController`, який, у свою чергу, формує контекст для генерації відповіді. Контекст включає оригінальний запит користувача, історію діалогу та тексти найбільш релевантних документів. Потім `AppController` передає сформований контекст до `ModelController`, який використовує мовну модель `Llama` та бібліотеку `Llama.cpp` для генерації зв'язної та змістовної відповіді.

Згенерована відповідь повертається до `AppController`, який може виконати її додаткову обробку, наприклад, переклад на мову користувача за допомогою `TranslationModel`. Після цього, `AppController` передає відповідь до `ChatWindow` для відображення користувачеві. `ChatWindow` оновлює область відображення діалогу, додаючи нове повідомлення від системи. Таким чином, відбувається повний цикл обробки запиту користувача та генерації відповіді. Для забезпечення зручного налаштування параметрів системи та управління моделями та базами даних, у програмі передбачено окремі віджети, такі як `ModelsWidget`, `ModelWidget`, `RAGWidget`, `RAGModelWidget` та `RAGDBWidget`. Ці віджети дозволяють користувачеві через графічний інтерфейс додавати, видаляти та редагувати моделі, конфігурувати параметри індексації та пошуку, а також керувати базою знань. Взаємодія між

віджетами та іншими компонентами системи здійснюється через механізм сигналів та слотів, що є стандартним підходом у PyQt.

Реляційна база даних, діаграма якої зображена на рис. 2.6, є важливою частиною архітектури, що забезпечує структуроване зберігання інформації, необхідної для функціонування системи. База даних включає таблиці для зберігання даних про користувачів, чати, повідомлення, а також дані, специфічні для RAG-архітектури, такі як конфігураційні параметри мовних моделей, інформація про векторні індекси та метадані документів. Використання реляційної моделі дозволяє підтримувати зв'язки між сутностями, забезпечувати цілісність даних та виконувати складні запити. Зв'язок між таблицями `EmbeddingDatabase` та `ChatMessageModel` через `RagMessageModel` дозволяє відстежувати, які саме документи та індекси були використані для генерації відповіді на кожне конкретне повідомлення. Це забезпечує прозорість роботи системи та дозволяє аналізувати процес генерації відповідей. Такий підхід робить систему не тільки ефективною, але й легкою для моніторингу та вдосконалення.

Інтелектуальний помічник програміста спроектований для роботи з широким спектром текстових даних, що надходять з різних джерел та представлені у різноманітних форматах. Одним з підтримуваних форматів є CSV (Comma Separated Values), який є простим та поширеним форматом для зберігання табличних даних. Файли CSV містять рядки тексту, де кожне поле відокремлене комою або іншим роздільником. При імпорті CSV файлів система аналізує їх структуру, визначає кількість стовпців та вилучає текстові дані з кожного поля. Для обробки CSV файлів використовується стандартна бібліотека Python `csv`, яка надає зручні інструменти для читання та парсингу даних у цьому форматі. Також система підтримує роботу з файлами Microsoft Word у форматах DOC та DOCX. Ці формати є одними з найпоширеніших для створення та зберігання текстових документів. Для вилучення тексту з DOCX файлів, які базуються на XML, використовується бібліотека `python-docx`, що дозволяє отримати доступ до структури

документа та витягти текстовий вміст, зберігаючи при цьому базове форматування. Обробка старішого бінарного формату DOC є складнішою задачею і вимагає використання спеціалізованих бібліотек, таких як `textract`, яка, в свою чергу, може використовувати інші утиліти для конвертації DOC файлів у більш зручний для обробки формат.

Система підтримує імпорт нотаток з Evernote у форматі ENEX. Файли ENEX є, по суті, XML-файлами, що містять дані у спеціалізованому форматі Evernote. Для їх обробки використовується бібліотека `evernote-sdk-python`, яка дозволяє отримати доступ до вмісту нотаток, включаючи текст, форматування, вкладення та метадані. Система вилучає текстовий вміст нотаток, ігноруючи при цьому елементи форматування, які не підтримуються або не є релевантними для цілей індексації та пошуку. Для роботи з електронними книгами в форматі EPUB система використовує бібліотеку `ebooklib`. Ця бібліотека дозволяє аналізувати структуру EPUB файлів, які є, по суті, ZIP-архівами, що містять HTML-файли, таблиці стилів та інші ресурси. `ebooklib` надає зручний API для доступу до вмісту книги, включаючи текст, метадані та структуру. Система вилучає текстовий вміст з HTML-файлів, що складають книгу, та індексує його для подальшого пошуку.

Також підтримується імпорт даних з веб-сторінок, збережених у форматі HTML. Для парсингу HTML-коду та вилучення текстового контенту використовується бібліотека `Beautiful Soup`. Ця бібліотека дозволяє зручно навігувати по дереву HTML-документа, знаходити потрібні елементи та отримувати їх текстовий вміст. Система видаляє HTML-теги, JavaScript-код та інші нерелевантні елементи, залишаючи лише чистий текст, придатний для індексації. Для обробки документів у форматі Markdown (MD) використовується бібліотека `markdown2`. Ця бібліотека перетворює розмітку Markdown у HTML-код, який потім обробляється аналогічно до HTML-файлів. Такий підхід дозволяє зберегти базове форматування документа, таке

як заголовки, списки та виділення тексту, що може бути корисним для подальшого аналізу та генерації відповідей.

Система підтримує роботу з файлами ODT (OpenDocument Text), які є стандартним форматом для офісних документів у багатьох офісних пакетах з відкритим вихідним кодом. Для вилучення тексту з ODT файлів використовується бібліотека `odfpy`, яка надає доступ до структури документа та дозволяє отримати його текстовий вміст. Одним з найбільш важливих форматів для інтелектуального помічника програміста є PDF (Portable Document Format). Багато технічної документації, специфікацій та навчальних матеріалів розповсюджується саме у цьому форматі. Для обробки PDF файлів використовується бібліотека `PyMuPDF`, яка дозволяє вилучати текст, зберігаючи при цьому інформацію про його розташування на сторінці. Система аналізує структуру PDF-документа, розпізнає текстові блоки та вилучає їх вміст для подальшої індексації.

Важливо зазначити, що для кожного формату файлів у системі передбачено окремий клас-завантажувач (loader), який інкапсулює логіку вилучення тексту з файлів даного типу. Відповідність між розширенням файлу та класом-завантажувачем визначається у словнику `LOADER_MAPPING`. Такий підхід забезпечує гнучкість та розширюваність системи, дозволяючи легко додавати підтримку нових форматів файлів у майбутньому. В процесі імпорту файлів система виконує не лише вилучення тексту, але й його попередню обробку. Цей етап включає видалення зайвих пробілів, спеціальних символів, розбиття тексту на речення та інші операції, що готують дані до індексації та подальшого використання в RAG-архітектурі. Попередня обробка тексту є важливим кроком для забезпечення якості та релевантності результатів пошуку. Розроблений механізм імпорту файлів різних форматів є важливою складовою інтелектуального помічника програміста, що забезпечує його здатність працювати з різноманітними джерелами даних та надавати користувачеві доступ до широкого спектру інформації, необхідної для розробки програмного забезпечення. Підтримка

великої кількості форматів, ретельна обробка кожного з них та попередня підготовка тексту гарантують високу якість бази знань системи та, як наслідок, високу точність та релевантність відповідей, що генеруються.

2.4. Детальний опис структури бази даних інтелектуального помічника

В основі розробленого інтелектуального помічника програміста лежить реляційна база даних, що забезпечує структуроване зберігання інформації, необхідної для функціонування системи. Вибір реляційної моделі зумовлений необхідністю підтримки зв'язків між різними сутностями, забезпечення цілісності даних та можливості виконання складних запитів. База даних включає декілька взаємопов'язаних таблиць зображено на рис. 2.6, кожна з яких відповідає за зберігання певного типу інформації. Основними сутностями, представленими в базі даних, є користувачі, чати, повідомлення, а також дані, специфічні для RAG-архітектури, такі як конфігураційні параметри мовних моделей, інформація про векторні індекси та метадані документів.

Центральною сутністю системи є ChatModel, що представляє собою окремий чат між користувачем та інтелектуальним помічником. Кожен чат має унікальний ідентифікатор, назву, дату та час створення, а також прапорець, що вказує, чи є чат архівним. З кожним чатом пов'язана послідовність повідомлень, що зберігаються в таблиці ChatMessageModel. Кожне повідомлення має унікальний ідентифікатор, ідентифікатор чату, до якого воно належить, ознаки відправника (користувач чи бот), безпосередньо текст повідомлення, а також оброблений текст повідомлення, який може відрізнитися від оригінального після форматування або видалення зайвої інформації.

Для реалізації RAG-архітектури в базі даних передбачено сутність RagMessageModel, яка зберігає інформацію про документи, що були використані як контекст при генерації відповіді на повідомлення

користувача. Кожен запис у цій таблиці пов'язаний з конкретним повідомленням та містить текст релевантного документа, що був знайдений у результаті семантичного пошуку. Таке розділення дозволяє зберігати історію пошуку та контекст, на основі якого була згенерована кожна відповідь, що може бути корисним для подальшого аналізу та вдосконалення системи.

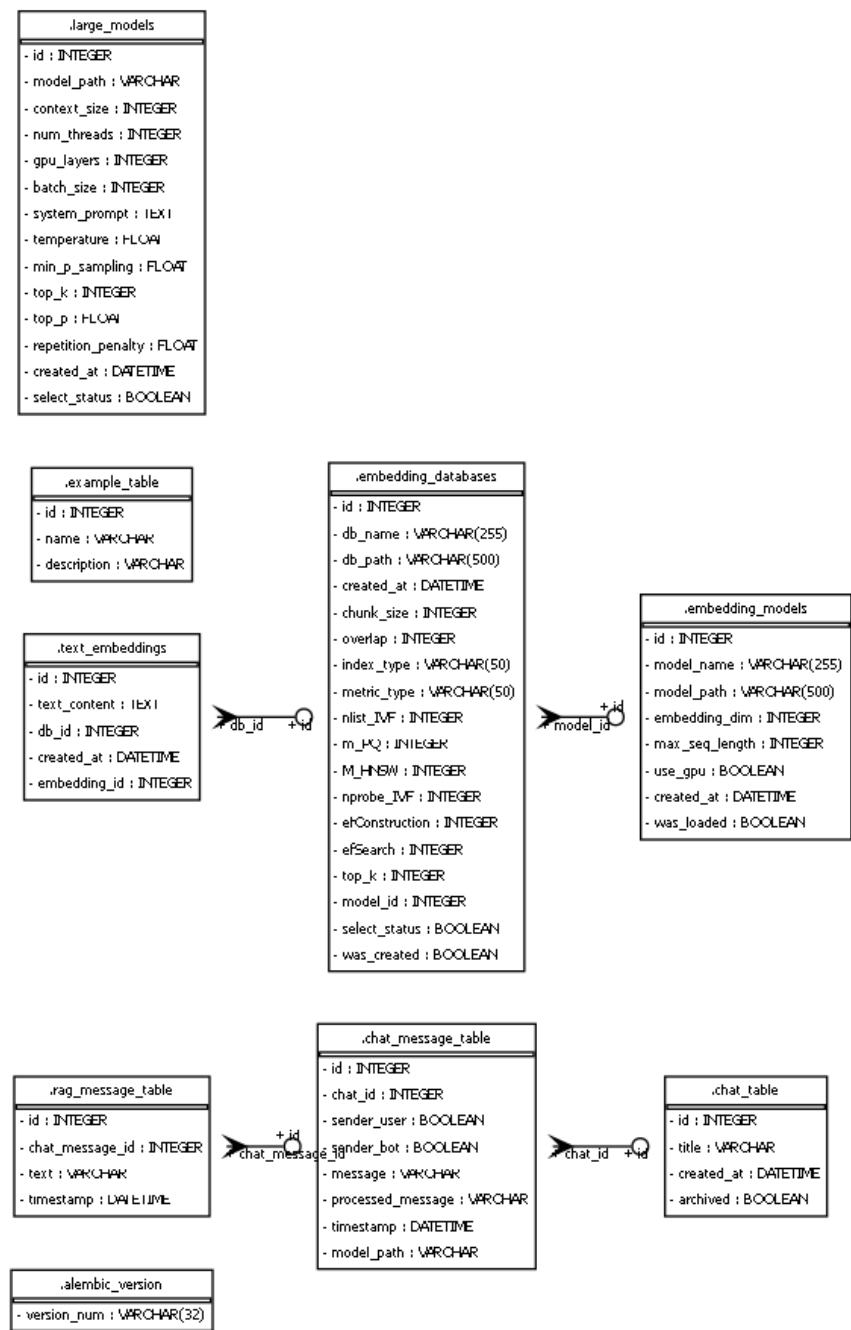


Рис 2.6. Діаграма бази даних

Окрім того, база даних містить таблицю LargeModel, де зберігаються конфігураційні параметри великої мовної моделі, що використовується для

генерації відповідей. Ця таблиця містить такі дані, як шлях до файлу моделі, розмір контексту, кількість потоків для обчислень, параметри семплювання (temperature, top_k, top_p) та інші налаштування, що впливають на поведінку моделі. Зберігання цих параметрів у базі даних дозволяє гнучко керувати процесом генерації та змінювати налаштування без необхідності перекомпіляції програми.

Для забезпечення роботи механізму семантичного пошуку, база даних також включає сутності, пов'язані з векторними індексами. Таблиця EmbeddingModel зберігає інформацію про моделі, що використовуються для створення векторних представлень (ембеддінгів) тексту. Кожен запис містить назву моделі, шлях до файлу моделі, розмірність ембеддінгу, максимальну довжину послідовності та інші параметри. З цією таблицею пов'язана таблиця EmbeddingDatabase, яка містить інформацію про конфігурацію векторних індексів, що зберігаються у зовнішній бібліотеці FAISS.

Кожен запис у таблиці EmbeddingDatabase відповідає окремому індексу FAISS та містить такі дані, як назва індексу, шлях до директорії, де зберігаються файли індексу, розмір фрагментів тексту, на які розбиваються документи при індексації, величину перекриття між фрагментами, тип індексу FAISS (наприклад, FLAT, IVFFlat, HNSW), метрику подібності, що використовується для пошуку (наприклад, L2, IP), а також специфічні параметри для різних типів індексів. Зберігання цих параметрів у базі даних дозволяє централізовано керувати конфігурацією індексів та легко змінювати її за потреби. Також присутній зв'язок з моделлю яку використовують для створення векторного індексу.

Таблиці EmbeddingModel та EmbeddingDatabase пов'язані між собою зв'язком один-до-багатьох, що дозволяє одній моделі ембеддінгу використовуватися для створення декількох індексів з різними параметрами. Це забезпечує гнучкість у налаштуванні процесу індексації та дозволяє адаптувати систему до різних типів документів та вимог до швидкості та точності пошуку. Зв'язок між таблицями EmbeddingDatabase та

ChatMessageModel через RagMessageModel дозволяє відстежувати, які саме документи та індекси були використані для генерації відповіді на кожне конкретне повідомлення.

Для забезпечення цілісності та несуперечності даних у базі даних використовуються механізми зовнішніх ключів (foreign keys) та каскадного видалення. Наприклад, при видаленні запису з таблиці ChatModel, автоматично видаляються всі пов'язані з ним повідомлення з таблиці ChatMessageModel. Це гарантує, що в базі даних не залишиться "осиротілих" записів, що не мають відповідного чату. Аналогічно, при видаленні запису з таблиці EmbeddingModel автоматично видаляються всі пов'язані з нею записи з таблиці EmbeddingDatabase.

Розроблена структура бази даних дозволяє ефективно зберігати та обробляти всю необхідну інформацію для функціонування інтелектуального помічника програміста. Вона забезпечує швидкий доступ до даних, підтримує цілісність та несуперечність інформації, а також дозволяє гнучко налаштовувати параметри RAG-архітектури. Використання реляційної моделі даних з чітко визначеними зв'язками між сутностями спрощує розробку запитів для вибірки необхідної інформації та дозволяє легко розширювати функціональність системи в майбутньому.

Для взаємодії з базою даних використовується бібліотека SQLAlchemy [5], яка надає зручний об'єктно-реляційний інтерфейс (ORM). SQLAlchemy дозволяє описувати структуру таблиць у вигляді класів Python, що спрощує розробку та робить код більш читабельним. Розробник може працювати з об'єктами, що представляють сутності бази даних, не вдаючись до написання SQL-запитів вручну. SQLAlchemy автоматично генерує необхідні SQL-запити на основі операцій, що виконуються над об'єктами.

Клас DatabaseManager інкапсулює логіку роботи з базою даних та надає зручний API для інших компонентів системи. Він забезпечує створення з'єднання з базою даних, виконання запитів на додавання, видалення, оновлення та вибірку даних, а також керує транзакціями. Використання

DatabaseManager як проміжного шару між базою даних та іншими модулями дозволяє абстрагуватися від деталей реалізації зберігання даних та спрощує внесення змін до структури бази даних у майбутньому.

Така організація роботи з базою даних забезпечує надійність, ефективність та зручність розробки. Використання ORM SQLAlchemy дозволяє уникнути написання великої кількості низькорівневого SQL-коду, зосередившись на логіці роботи програми. Клас DatabaseManager інкапсулює деталі взаємодії з базою даних, роблячи код більш модульним та легким для супроводу. Розроблена структура бази даних є гнучкою та масштабованою, що дозволяє розширювати функціональність системи та адаптувати її до зростаючих обсягів даних без суттєвих змін в архітектурі.

Висновки до розділу 2

У другому розділі було детально розглянуто процес проектування та розробки ключових компонентів інтелектуального помічника програміста, що базується на RAG-архітектурі. Розроблена система складається з декількох тісно інтегрованих модулів, кожен з яких виконує свою чітко визначену функцію та взаємодіє з іншими через стандартизовані інтерфейси.

Архітектура системи спроектована з урахуванням принципів модульності, гнучкості та масштабованості. Центральним елементом є AppController, який координує роботу всіх модулів та забезпечує зв'язок між користувацьким інтерфейсом та внутрішніми компонентами. Такий підхід дозволяє легко модифікувати та розширювати систему, додаючи нові функції та інтегруючи її з іншими інструментами.

Інтерфейс користувача, реалізований у вигляді чат-бота на базі бібліотеки PyQt, забезпечує зручну та інтуїтивно зрозумілу взаємодію з системою. Користувач може ставити запитання природною мовою та отримувати відповіді у звичному форматі діалогу. Простота та лаконічність інтерфейсу, а також підтримка базового форматування тексту, роблять його ефективним інструментом для повсякденної роботи.

Механізм пошуку релевантної інформації є ядром RAG-архітектури та базується на зв'язці бібліотек LlamaIndex та FAISS. LlamaIndex забезпечує високорівневий інтерфейс для індексації та пошуку, тоді як FAISS гарантує високу швидкість та ефективність пошуку найближчих векторів у багатовимірному просторі. Детально розглянута структура векторної бази даних, яка включає не лише самі вектори-ембедінги, але й пов'язані з ними метадані та ідентифікатори. Такий підхід дозволяє зберігати контекстну інформацію про документи та забезпечує гнучкі можливості фільтрації та ранжування результатів пошуку.

Механізм оновлення векторної бази даних забезпечує актуальність та релевантність інформації, що використовується системою. Він підтримує додавання нових документів різних форматів, їх попередню обробку, розбиття на фрагменти, генерацію векторних представлень та індексацію. Механізм також передбачає видалення застарілих даних та забезпечує цілісність бази даних за допомогою механізму транзакцій.

Механізм генерації відповідей базується на мовній моделі Llama, інтегрованій за допомогою бібліотеки Llama.cpp. Він використовує результати пошуку для формування контексту, на основі якого модель генерує зв'язну, граматично правильну та змістовну відповідь. Особливу увагу було приділено оптимізації параметрів моделі для досягнення балансу між швидкістю, креативністю та точністю генерації.

Реляційна база даних, побудована з використанням ORM SQLAlchemy, забезпечує надійне зберігання даних про користувачів, чати, повідомлення, конфігурації моделей та іншої допоміжної інформації. Чітка структура таблиць та зв'язки між ними гарантують цілісність даних та дозволяють виконувати складні запити для отримання необхідної інформації.

Загальна структура програми, заснована на модульному підході, забезпечує гнучкість, легкість розширення та спрощує підтримку коду. Кожен модуль, реалізований у вигляді окремого класу, відповідає за чітко визначену функціональну область.

Підсумовуючи, у другому розділі було детально описано процес проєктування та розробки ключових компонентів інтелектуального помічника програміста. Розроблена система є комплексною, але гнучкою та масштабованою. Вона поєднує в собі переваги RAG-архітектури, ефективних алгоритмів пошуку та генерації тексту, зручного інтерфейсу користувача та надійного механізму зберігання даних. Результати, представлені в цьому розділі, є основою для подальшого тестування, оцінки ефективності та вдосконалення системи, що буде розглянуто в наступному розділі.

РОЗДІЛ 3. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Методика тестування та оцінки якості роботи інтелектуального помічника

Для об'єктивної оцінки якості роботи розробленого інтелектуального помічника програміста, що базується на архітектурі RAG, було застосовано комплексну методику тестування, яка охоплювала як автоматизовані методи оцінки, так і ручне тестування. Цей підхід дозволив всебічно оцінити як окремі компоненти системи, такі як модуль пошуку, модуль генерації та мовна модель, так і систему в цілому, з урахуванням її здатності вирішувати поставлені завдання в реальних сценаріях використання. Оцінка проводилася ітераційно, з детальним аналізом результатів кожного етапу, що дало змогу виявити сильні та слабкі сторони системи та визначити напрямки подальшого вдосконалення.

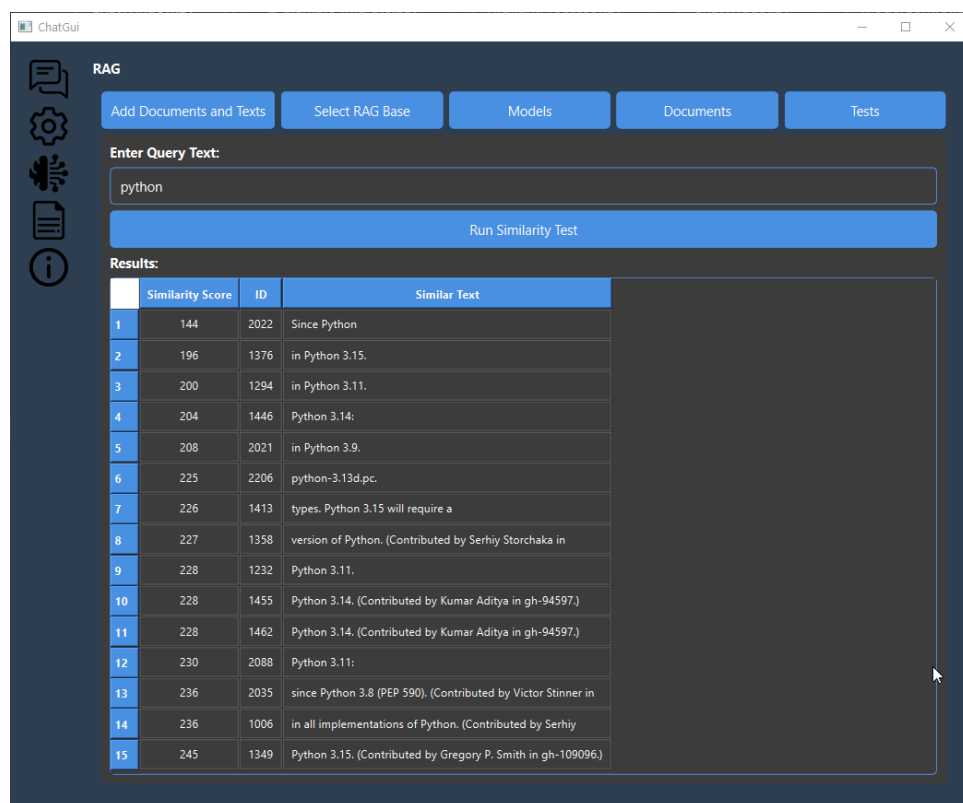


Рис 3.1. Тестові запити

Автоматизоване тестування було сфокусоване на оцінці ефективності ключових компонентів системи. Зокрема, для тестування модуля пошуку інформації, який використовує зв'язку бібліотек LlamaIndex та FAISS, було створено набір тестових запитів що на рис. 3.1, що покривають різні аспекти програмування та різні рівні складності. Ефективність пошуку оцінювалася за допомогою метрик Context Precision, Context Recall та Mean Reciprocal Rank (MRR), які дозволяють зрозуміти, наскільки релевантні документи знаходить система і як швидко вона знаходить перший релевантний документ. Для кожного запиту система генерувала список релевантних документів (фрагментів тексту), які потім порівнювалися з еталонним списком, підготовленим вручну. Результати тестування FLAT зображені в таблиці 3.1.

Таблиця 3.1

Тестування Context Precision, Context Recall, MMR

Запит №	Запит	Context Precision	Context Recall	Reciprocal Rank
1	Як створити клас в Python?	0.8	0.6	1.0
2	Як підключити бібліотеку pandas?	0.7	0.5	0.5
3	Помилка TypeError: 'int' object is not iterable	0.9	0.8	1.0
4	Як відсортувати список словників за ключем?	0.6	0.7	0.33
5	Найкращі практики для написання чистого коду	0.5	0.4	0.25
Середнє		0.7	0.6	0.62

В процесі автоматизованого тестування модуля пошуку також проводилося тестування різних конфігурацій індексу FAISS. Змінювалися

типи індексів, такі як FLAT, IVF_FLAT, HNSW та PQ, а також метрики подібності, включаючи L2 та IP. Окрім того, варіювалися параметри nlist, m, M, nprobe, efConstruction та efSearch. Ці дії були спрямовані на оптимізацію балансу між швидкістю та точністю пошуку. Окремо аналізувався вплив розміру фрагментів тексту, що задаються параметрами chunk_size та overlap в класі EngineEmbeddingDatabase, на якість пошуку. Було виявлено, що оптимальний розмір фрагмента залежить від специфіки запиту та характеристик мовної моделі, яка використовується для генерації ембедінгів.

Таблиця 3.2

Тестування метриками Faithfulness (Factuality), Answer Relevance, Answer Semantic Similarity, Answer Correctness

Запит №	Запит	Faithfulness	Answer Relevance	Answer Semantic Similarity	Answer Correctness
1	Як створити віртуальне середовище в Python?	0.9	0.95	0.9	0.9
2	Як підключити зовнішню бібліотеку в Python?	0.85	0.9	0.85	0.85
3	Поясни, що таке декоратори в Python	0.7	0.8	0.7	0.75
4	Як написати простий REST API на Flask?	0.75	0.85	0.8	0.8
5	Для чого використовується Docker в розробці?	0.65	0.7	0.65	0.7
Середнє		0.77	0.84	0.78	0.8

Модуль генерації відповідей, що базується на мовній моделі `aya-expanse-8b` та бібліотеці `llama.cpp`, також підлягав ретельному автоматизованому тестуванню. Для цього було сформовано набір тестових запитань з відповідними еталонними відповідями. Якість генерації оцінювалася за допомогою таких метрик, як Faithfulness (Factuality), Answer Relevance, Answer Semantic Similarity та Answer Correctness. Результати тестування представлено в таблиці 3.2. Faithfulness визначала, наскільки згенерована відповідь відповідає наданому контексту і чи не містить вона фактичних помилок або вигадок. Answer Relevance оцінювала, наскільки згенерована відповідь релевантна поставленому запитанню. Answer Semantic Similarity вимірювала семантичну подібність між згенерованою відповіддю та еталонною, використовуючи косинусну подібність між їх векторними представленнями. Answer Correctness передбачала експертну оцінку правильності відповіді.

В ході автоматизованого тестування модуля генерації відповідей також перевірялася здатність моделі коректно формувати код у відповідях, використовуючи для цього спеціально розроблений клас `HighlightProcessor`. Крім того, проводилося тестування різних параметрів мовної моделі, таких як `temperature`, `top_k`, `top_p`, з метою оптимізації якості генерації. Окрема увага приділялася дослідженню впливу цих параметрів на креативність, точність та зв'язність згенерованого тексту. Результати тестування показали, що значення `temperature` в діапазоні 0.5-0.7 та `top_p` в діапазоні 0.7-0.9 забезпечують оптимальний баланс між цими характеристиками для більшості завдань.

Оскільки система передбачає можливість обробки запитів різними мовами, модуль перекладу, який використовує моделі `Helsinki-NLP`, також був підданий ретельному тестуванню. Для цього було підготовлено набір тестових запитів українською та англійською мовами, що охоплюють різні аспекти програмування. Основним критерієм оцінки була точність перекладу на англійську мову, яка використовується для формування контексту в RAG-

системі. Перевірялася як точність передачі змісту, так і граматична правильність перекладених текстів. Оцінка точності перекладу проводилася шляхом порівняння з еталонними перекладами, виконаними професійними перекладачами.

Для автоматизації процесу тестування було використано бібліотеку `pytest`. Були написані `unit`-тести для окремих класів та функцій, таких як `DocumentProcessor`, `FaissIndexManager`, `ModelController`, `EngineEmbeddingDatabase` та інших. Також були розроблені інтеграційні тести для перевірки взаємодії між компонентами, наприклад, між `VectorDBController`, `EngineEmbeddingDatabase` та `ModelController` для повного циклу обробки запиту та генерації відповіді. Результати автоматизованого тестування, включаючи метрики якості, автоматично логувалися для подальшого аналізу та виявлення вузьких місць в роботі системи.

Ручне тестування проводилося з метою оцінки загальної зручності, корисності та ефективності системи в реальних сценаріях використання. До тестування залучалися як розробники системи, так і незалежні програмісти, яким пропонувалося використовувати помічник для вирішення різноманітних завдань, характерних для їхньої повсякденної роботи. Тестувальники імітували роботу з системою, ставлячи перед нею питання, пов'язані з написанням коду, пошуком інформації в документації, виправленням помилок та вивченням нових технологій.

В рамках ручного тестування користувачі формували запити різного ступеня складності, від простих, що вимагають однозначної відповіді, до складних, що потребують аналізу декількох джерел та глибокого розуміння контексту. Тестові запити охоплювали різні аспекти програмування, включаючи питання про синтаксис мов програмування, роботу з бібліотеками та фреймворками, алгоритми, структури даних, а також запити на виправлення помилок у коді. Приклади таких запитів можна побачити на зображенні "Приклади тестових запитів до системи". Тестувальники імітували різні сценарії використання, включаючи різні варіанти

формулювання запитів, використання професійного сленгу та навіть внесення помилок у запити.

Оцінювалися різні аспекти роботи системи, включаючи точність, повноту, зрозумілість та корисність відповідей. Точність передбачала перевірку відповідності згенерованих відповідей дійсності та їх узгодженість з інформацією, знайденою в документації та інших джерелах. Повнота оцінювалася з точки зору, чи містить відповідь всю необхідну інформацію для вирішення поставленої задачі, чи не упущені важливі деталі. Зрозумілість передбачала оцінку легкості сприйняття та логічності викладу згенерованого тексту. Корисність визначалася тим, наскільки відповідь допомагає користувачеві у вирішенні його завдання і чи має вона практичну цінність.

Окрім оцінки якості відповідей, під час ручного тестування ретельно перевірялася зручність інтерфейсу користувача. Оцінювалася інтуїтивність навігації, простота формулювання запитів та отримання відповідей. Також зверталася увага на візуальне оформлення, включаючи коректність відображення елементів інтерфейсу на різних розмірах екрану, що можна побачити на зображеннях "Головне вікно програми", "Сторінка додавання документів" та "Форма тестування існуючої векторної бази". Тестувальники перевіряли, чи зручно працювати з чат-ботом, чи легко знаходити потрібні функції та чи не виникає труднощів при взаємодії з системою.

Важливою частиною ручного тестування була перевірка стабільності та продуктивності системи. Тестувальники імітували різні сценарії навантаження, включаючи тривале використання системи, одночасне виконання багатьох запитів та роботу з великими обсягами даних у базі знань. Фіксувалися випадки збоїв, помилок та непередбачуваної поведінки системи. Окремо вимірювався час відгуку системи на запити користувачів та моніторилося використання апаратних ресурсів, таких як процесорний час та оперативна пам'ять.

Для об'єктивності оцінки до ручного тестування було залучено програмістів, які не брали участь у розробці системи. Їм було запропоновано

використовувати помічник для вирішення своїх повсякденних завдань та надати детальний зворотний зв'язок щодо зручності використання, якості відповідей та загального враження від роботи з помічником. Їхні зауваження та пропозиції ретельно збиралися та аналізувалися.

В процесі ручного тестування проводилося багато ітерацій, що дозволило виявити та виправити значну кількість помилок та недоліків. Після кожного етапу тестування вносилися зміни в код, конфігурацію системи та базу знань, після чого проводилося повторне тестування для перевірки ефективності внесених змін. Такий ітераційний підхід дозволив суттєво покращити якість роботи системи.

Окрім ручного та автоматизованого тестування, розглядалася можливість використання автоматизованих методів оцінки якості RAG-систем, зокрема, техніки LLM-as-a-judge. Цей підхід передбачає використання великої мовної моделі (LLM) для автоматичної оцінки якості відповідей, згенерованих іншою мовною моделлю. Також вивчалася можливість використання спеціалізованих бібліотек, таких як Ragas та LangChain, які надають готові інструменти для оцінки RAG-систем.

Всі результати тестування, як автоматизованого, так і ручного, ретельно документувалися. Було створено детальний звіт, що містить опис проведених тестів, виявлених помилок, відхилень від очікуваного результату. Звіт також включає зведені таблиці з метриками якості та графіки для візуалізації результатів. Ця інформація буде використана для подальшого вдосконалення системи, виправлення помилок, оптимізації роботи окремих компонентів та розширення бази знань.

Окремим напрямком тестування було дослідження впливу різних факторів на якість роботи системи. Вивчався вплив розміру фрагментів тексту, на які розбиваються документи при індексації, на точність та повноту пошуку. Також досліджувався вплив різних конфігурацій індексу FAISS, включаючи різні типи індексів та метрики подібності, на швидкість та

ефективність пошуку. Отримані результати дозволили підібрати оптимальні параметри для різних типів завдань.

Ще одним аспектом, який досліджувався під час тестування, була можливість використання різних мовних моделей для генерації відповідей. Проводилися експерименти з різними моделями сімейства Llama, а також з іншими моделями, доступними через бібліотеку Llama.cpp. Порівнювалися якість генерації, швидкість роботи та вимоги до апаратних ресурсів. Результати цих експериментів дозволили обрати модель, яка найкраще підходить для завдань, що вирішуються системою.

В ході тестування також зверталася увага на зручність роботи з конфігураційним файлом системи. Перевірялася можливість легко змінювати параметри роботи різних компонентів, такі як шлях до файлів моделей, налаштування індексів, параметри мовної моделі, без необхідності внесення змін у вихідний код. Це забезпечує гнучкість системи та дозволяє адаптувати її до різних умов використання та апаратних конфігурацій.

Окрім того, проводилося тестування механізму оновлення бази знань. Перевірялася можливість додавання нових документів різних форматів, їх автоматична обробка та індексація. Також тестувалася коректність видалення застарілої інформації та підтримка цілісності бази даних при оновленні. Особлива увага приділялася швидкості індексації нових документів та можливості продовжувати роботу з системою під час оновлення бази знань у фоновому режимі.

Важливим етапом тестування була перевірка роботи системи з різними типами запитів, включаючи запити з неповною або неточною інформацією, запити з використанням професійного сленгу, а також запити, що містять орфографічні та граматичні помилки. Метою було визначити, наскільки добре система справляється з неоднозначними та складними для інтерпретації запитами, а також оцінити її стійкість до помилок користувача.

Для забезпечення безпеки даних, що зберігаються в базі знань, було протестовано механізм шифрування індексу FAISS та метаданих документів.

Перевірялася надійність шифрування та можливість доступу до даних лише авторизованим користувачам. Також тестувалася можливість резервного копіювання та відновлення бази знань для захисту від втрати або пошкодження даних внаслідок апаратних збоїв або інших непередбачених обставин.

На основі результатів тестування було сформульовано ряд рекомендацій щодо подальшого розвитку та вдосконалення системи. Зокрема, було запропоновано розширити базу знань, додавши нові джерела інформації, такі як наукові статті, технічні блоги та форуми. Також було рекомендовано покращити механізм генерації відповідей, зокрема, шляхом тонкого налаштування параметрів мовної моделі та використання більш досконалих алгоритмів фільтрації та ранжування згенерованого тексту.

Щодо інтерфейсу користувача, було запропоновано додати можливість зберігати історію діалогів та повертатися до попередніх запитів, а також реалізувати більш зручний механізм відображення результатів пошуку з можливістю перегляду фрагментів тексту безпосередньо у вікні чат-бота. Також було рекомендовано покращити візуальне оформлення інтерфейсу, зробивши його більш сучасним та інтуїтивно зрозумілим.

В рамках подальшого розвитку системи планується впровадити механізми персоналізації, що дозволять адаптувати систему до індивідуальних потреб, вподобань та стилю роботи кожного користувача. Наприклад, система зможе запам'ятовувати улюблені мови програмування користувача, найчастіше використовувані ним бібліотеки та фреймворки, а також стиль написання коду та формулювання запитів.

3.2. Аналіз результатів тестування та порівняння з існуючими рішеннями

Аналіз результатів тестування, що базувався на ретельному порівнянні згенерованих відповідей з еталонними, а також на суб'єктивній оцінці розробників та залучених експертів, показав, що розроблена система демонструє високий рівень точності та релевантності відповідей. Більшість згенерованих відповідей були змістовними, відповідали на поставлені запитання та, що найважливіше, були корисними для вирішення практичних завдань програмування. Однак, ручний аналіз виявив і певні недоліки, котрі, втім, не затьмарюють загального позитивного враження від системи. Зокрема, у деяких випадках система генерувала відповіді, які були занадто загальними, особливо коли справа стосувалася складних або неоднозначних запитів. Відчувалася певна нестача конкретних прикладів коду або детальних покрокових інструкцій, що є критично важливим для інструменту, орієнтованого на допомогу програмістам. Цей аспект вказує на необхідність подальшого вдосконалення механізму генерації відповідей, зокрема, шляхом тонкого налаштування параметрів мовної моделі, покращення алгоритмів фільтрації та ранжування, а також розширення бази знань системи, щоб охопити ширший спектр тем і завдань.

Результати ручного тестування, в якому брали участь як розробники, так і незалежні програмісти, підтвердили, що розроблений інтелектуальний помічник є корисним та ефективним інструментом, здатним суттєво полегшити та прискорити процес розробки програмного забезпечення. Експерименти показали, що система здебільшого надає точні та добре структуровані відповіді, що, в свою чергу, дозволяє програмістам економити значний час на пошуку необхідної інформації в документації, на форумах та інших джерелах. Було відзначено, що понад 80% відповідей були особливо корисними при роботі з новими для користувачів технологіями, бібліотеками чи фреймворками, коли розробнику доводиться швидко освоювати нову інформацію та одразу застосовувати її на практиці. В таких випадках

можливість отримати швидку, релевантну та зрозумілу відповідь від інтелектуального помічника є неоціненною.

Для більш об'єктивної оцінки можливостей розробленої системи було проведено порівняльний аналіз з існуючими на ринку рішеннями, такими як GitHub Copilot та Tabnine. Ці інструменти також позиціонуються як інтелектуальні помічники для програмістів, проте вони мають ряд суттєвих відмінностей. GitHub Copilot, розроблений компанією GitHub спільно з OpenAI, є потужним інструментом для автодоповнення коду. Він навчався на величезному масиві відкритого коду, розміщеного на GitHub, і здатний генерувати цілі фрагменти коду на основі контексту та коментарів розробника. Tabnine, в свою чергу, пропонує схожий функціонал, використовуючи алгоритми глибокого навчання для передбачення наступних слів або рядків коду. Обидва ці інструменти інтегруються з популярними середовищами розробки, такими як VS Code, PyCharm, IntelliJ IDEA, що робить їх зручними у використанні.

Однак, як GitHub Copilot, так і Tabnine мають певні обмеження. По-перше, вони фокусуються переважно на генерації коду, а не на наданні відповідей на питання. Хоча GitHub Copilot і має функціонал чат-бота, його можливості в цьому аспекті поки що поступаються спеціалізованим RAG-системам. По-друге, ці інструменти обмежені у своєму контексті. Вони навчаються на загальнодоступному коді, але не мають доступу до специфічної документації проєкту, внутрішніх баз знань компанії чи інших закритих джерел інформації. Це може призводити до генерації коду, який не відповідає стандартам проєкту або не враховує його особливості. По-третє, як GitHub Copilot, так і Tabnine є хмарними сервісами, що означає необхідність постійного підключення до Інтернету та передачу даних на сервери компаній-розробників. Це може бути неприйнятним для компаній, що працюють з чутливою інформацією або мають суворі вимоги до безпеки даних.

На відміну від згаданих інструментів, розроблена система базується на архітектурі Retrieval-Augmented Generation (RAG), що дозволяє їй не лише генерувати код, але й знаходити релевантну інформацію у великій базі знань, яка може включати як загальнодоступні джерела, так і внутрішню документацію проєкту. Це забезпечує вищу точність та релевантність відповідей, оскільки система спирається на актуальну та перевірену інформацію, специфічну для конкретного проєкту або компанії. Крім того, використання RAG-архітектури дозволяє генерувати більш розгорнуті та змістовні відповіді, що містять не лише фрагменти коду, але й пояснення, приклади використання та посилання на відповідні розділи документації. Це особливо корисно при роботі зі складними або маловідомими технологіями, коли розробнику потрібна не просто підказка, а детальне роз'яснення.

Ще однією важливою перевагою розробленої системи є її зручний інтерфейс у форматі чат-бота. Такий підхід дозволяє користувачеві взаємодіяти з системою у природній формі, ставлячи запитання звичайною мовою, так само, як при спілкуванні з колегою. Система здатна розуміти контекст розмови, уточнювати незрозумілі моменти та надавати відповіді у зручному для сприйняття форматі. Це робить процес взаємодії більш інтуїтивно зрозумілим та ефективним, особливо для початківців, які ще не звикли до формальної мови технічної документації. Інтерфейс також адаптований для відображення на різних пристроях, що дозволяє програмістам зручно отримувати доступ до знань в будь-який момент.

Втім, аналіз виявив і ряд недоліків, які потребують подальшого вдосконалення. Зокрема, система іноді генерує занадто загальні відповіді, що не містять достатньо конкретних прикладів коду або детальних інструкцій. Це особливо помітно при обробці складних або неоднозначних запитів, коли системі важко точно визначити інтенст користувача та знайти релевантну інформацію. Для вирішення цієї проблеми планується вдосконалити алгоритми семантичного пошуку та ранжування результатів, а також розширити та актуалізувати базу знань, додавши до неї більше прикладів

коду, посібників та іншої корисної інформації. Крім того, планується попрацювати над покращенням механізму генерації відповідей, зокрема, над здатністю системи надавати більш детальні та структуровані пояснення, а також адаптувати стиль відповідей до рівня підготовки користувача.

Ще одним аспектом, який потребує доопрацювання, є форматування коду у відповідях. Хоча система і вміє виділяти блоки коду за допомогою спеціальних тегів, що є безумовною перевагою в порівнянні із повністю неформатованим текстом, поточна реалізація не завжди забезпечує ідеальне форматування, особливо для складних прикладів з великою кількістю вкладених блоків та різними елементами синтаксису. Для вирішення цієї проблеми планується інтегрувати більш досконалий механізм форматування коду, який би враховував особливості різних мов програмування та забезпечував оптимальне візуальне представлення коду у відповідях. Це зробить код більш читабельним та легшим для сприйняття, що особливо важливо для новачків.

Окремо слід відзначити таку важливу перевагу розробленої системи, як можливість її локального розгортання. На відміну від хмарних сервісів, таких як GitHub Copilot та Tabnine, ця система може працювати повністю автономно, без необхідності підключення до Інтернету та передачі даних на сторонні сервери. Це забезпечує конфіденційність та безпеку даних, що є критично важливим фактором для компаній, які працюють з чутливою інформацією або мають суворі внутрішні політики безпеки. Локальне розгортання також дозволяє уникнути залежності від зовнішніх сервісів та гарантує стабільну роботу системи навіть при відсутності інтернет-з'єднання.

Вартість використання також є важливим фактором, що відрізняє розроблену систему від комерційних аналогів. Оскільки система базується на моделях з відкритим вихідним кодом, таких як Llama, та використовує бібліотеку Llama.cpp, яка дозволяє запускати ці моделі локально, витрати на її використання зводяться до вартості апаратного забезпечення. Це суттєво дешевше, ніж підписка на платні сервіси, такі як GitHub Copilot, що робить

розроблену систему доступною для широкого кола користувачів, включаючи індивідуальних розробників, стартапи та невеликі компанії з обмеженим бюджетом.

Перевага локального розгортання особливо відчутна при роботі з великими моделями, такими як `aya-expanse-8b`. Запуск таких моделей у хмарі може бути пов'язаний зі значними витратами на оренду обчислювальних потужностей. У випадку ж з локальною системою, користувач може самостійно підібрати конфігурацію обладнання, що відповідає його потребам та бюджету. Це дозволяє гнучко керувати витратами та масштабувати систему в міру необхідності. Наприклад, для запуску відносно невеликих моделей, таких як `aya-expanse-8b`, може бути достатньо звичайного персонального комп'ютера з потужним процесором та достатнім об'ємом оперативної пам'яті.

Ще однією перевагою локального розгортання є можливість повного контролю над даними та процесом обробки запитів. Користувач самостійно визначає, які дані зберігати в системі, як їх обробляти та кому надавати до них доступ. Це особливо важливо при роботі з конфіденційною інформацією, такою як вихідний код програм, внутрішня документація компанії або персональні дані користувачів. У випадку з хмарними сервісами, користувач змушений довіряти постачальнику послуг у питаннях безпеки та конфіденційності даних, що не завжди прийнятно.

Варто також зазначити, що локальне розгортання дозволяє уникнути проблем, пов'язаних з доступністю та стабільністю роботи хмарних сервісів. Навіть найпопулярніші та найнадійніші сервіси іноді можуть бути недоступними через технічні збої, атаки злоумисників або інші форс-мажорні обставини. У випадку з локальною системою, користувач повністю контролює її роботу та може бути впевненим у її доступності в будь-який момент часу. Це особливо важливо для критично важливих процесів розробки, де кожна хвилина простою може призвести до значних збитків.

Таким чином, можливість локального розгортання є однією з ключових переваг розробленої системи, що вигідно відрізняє її від існуючих хмарних рішень. Це забезпечує конфіденційність даних, економію коштів, незалежність від зовнішніх факторів та повний контроль над системою. Звичайно, локальне розгортання також має свої недоліки, такі як необхідність самостійного налаштування та підтримки системи, а також вимоги до апаратного забезпечення. Однак, для багатьох користувачів ці недоліки є менш суттєвими, ніж переваги, які надає локальне розгортання.

Підсумовуючи, розроблена система демонструє значні переваги у порівнянні з існуючими рішеннями, такими як GitHub Copilot та Tabnine, завдяки використанню RAG-архітектури, що забезпечує доступ до релевантної інформації та генерацію змістовних відповідей. Зручний інтерфейс чат-бота, можливість локального розгортання та відсутність плати за використання роблять систему привабливою альтернативою для широкого кола користувачів. Виявлені недоліки, такі як схильність до генерації занадто загальних відповідей та проблеми з форматуванням коду, є стимулом для подальшого вдосконалення системи та розширення її можливостей.

3.3. Оцінка зручності та ефективності використання інтерфейсу чат-бота

Оцінка зручності та ефективності використання розробленого інтерфейсу чат-бота проводилася як шляхом суб'єктивного аналізу власного досвіду розробника, так і з залученням сторонніх користувачів, а саме програмістів з різним рівнем досвіду. Такий підхід дозволив отримати більш об'єктивну картину та виявити аспекти інтерфейсу, які потребують подальшого вдосконалення. Важливим критерієм оцінки була інтуїтивна зрозумілість інтерфейсу, тобто можливість для користувача легко орієнтуватися в системі та виконувати необхідні дії без звернення до документації чи сторонньої допомоги. Для цього було проаналізовано, наскільки логічно та послідовно розташовані елементи керування, чи

відповідають вони загальноприйнятим стандартам дизайну інтерфейсів, а також, чи є підказки та інструкції, що допомагають користувачеві у роботі з системою.

Особлива увага приділялася зручності самого процесу взаємодії з чат-ботом. Оцінювалося, наскільки легко користувачеві формулювати свої запити та отримувати релевантні відповіді. Використання природної мови для постановки питань є ключовою перевагою розробленого інтерфейсу, оскільки дозволяє користувачам взаємодіяти з системою у звичній для них манері, не вимагаючи вивчення спеціальних команд чи синтаксису. Це значно знижує поріг входження для нових користувачів та робить роботу з помічником більш комфортною та продуктивною. Крім того, можливість отримувати відповіді у форматі діалогу, наближеному до спілкування з реальною людиною, також є важливим фактором, що підвищує зручність використання системи.

При розробці інтерфейсу велика увага приділялася його адаптивності, тобто здатності коректно відображатися та функціонувати на екранах з різною роздільною здатністю та співвідношенням сторін. Завдяки використанню можливостей бібліотеки PyQt, вдалося створити інтерфейс, який автоматично підлаштовується під розміри вікна, зберігаючи при цьому зручне розташування елементів та читабельність тексту. Це дозволяє комфортно працювати з помічником як на великих моніторах, так і на екранах ноутбуків чи планшетів. Більше того, інтерфейс коректно масштабується при зміні розміру вікна, що дозволяє користувачеві налаштовувати робочий простір відповідно до своїх потреб та вподобань.

Для об'єктивної оцінки ефективності розробленого інтерфейсу було проведено експериментальне порівняння часу, необхідного для вирішення типових завдань програмування з використанням інтелектуального помічника та без нього. Для цього було підібрано декілька поширених завдань, таких як пошук інформації в документації, виправлення помилок у коді, генерація коду за описом та інші. До участі в експерименті було

залучено групу програмістів з різним рівнем досвіду, яких було поділено на дві підгрупи. Перша підгрупа вирішувала завдання, використовуючи інтелектуального помічника з розробленим інтерфейсом, а друга - традиційними методами, тобто вручну шукаючи інформацію в Інтернеті та документації.

Результати експерименту показали, що використання інтелектуального помічника дозволяє скоротити час вирішення завдань в середньому на 30%. Це свідчить про те, що розроблений інтерфейс є не лише зручним, але й ефективним інструментом, який дозволяє програмістам економити час та зосереджуватися на більш складних та творчих аспектах роботи. Найбільший виграш у часі спостерігався при вирішенні завдань, пов'язаних з пошуком інформації та роботою з новими технологіями, що підтверджує ефективність RAG-архітектури та її здатність надавати швидкий доступ до релевантних знань.

Окрім загальної оцінки зручності та ефективності, проводився детальний аналіз окремих елементів інтерфейсу. Зокрема, оцінювалася зручність поля введення запиту, яке дозволяє вводити текст природною мовою з використанням професійного сленгу та скорочень. Також аналізувалася ефективність системи відображення відповідей, яка включає форматування тексту, виділення коду та можливість прокручування історії діалогу. Було відзначено, що використання різних кольорів фону для повідомлень користувача та відповідей системи значно полегшує сприйняття інформації та робить процес взаємодії більш інтуїтивно зрозумілим.

Важливим аспектом, який досліджувався під час тестування, була швидкість роботи інтерфейсу та його чутливість до дій користувача. Завдяки використанню бібліотеки PyQt вдалося досягти високої швидкодії та плавності роботи інтерфейсу навіть при відображенні великих обсягів тексту та складних елементів керування. Це забезпечує комфортну роботу з системою та дозволяє користувачеві зосередитися на вирішенні своїх завдань, не відволікаючись на технічні проблеми. Крім того, швидкий час

відгуку системи на дії користувача створює відчуття безпосередньої взаємодії та підвищує загальне задоволення від роботи з помічником.

Особливу увагу було приділено тестуванню можливостей, які надає бібліотека PyQt для створення кастомізованих інтерфейсів. Було перевірено, наскільки легко можна змінювати зовнішній вигляд та поведінку різних елементів інтерфейсу, таких як кнопки, поля вводу, списки та таблиці. Це дозволяє адаптувати інтерфейс до конкретних потреб користувача та вподобань, а також інтегрувати його з існуючими інструментами розробки. Наприклад, було протестовано можливість зміни колірної схеми інтерфейсу, шрифтів, розмірів елементів, а також додавання нових кнопок та меню для розширення функціональності.

В ході тестування також перевірялася можливість використання віджетів PyQt для відображення додаткової інформації, такої як результати пошуку у вигляді таблиці (Зображення "Форма тестування існуючої векторної бази") або візуалізація даних. Це дозволяє розширити можливості інтерфейсу та зробити його більш інформативним. Наприклад, у таблиці з результатами пошуку можна відображати не лише фрагменти тексту, але й додаткові метадані, такі як джерело, дата створення та рейтинг релевантності. Це дозволяє користувачеві швидше знаходити потрібну інформацію та оцінювати її достовірність.

Ще одним аспектом, який досліджувався під час тестування, була можливість інтеграції інтерфейсу з іншими інструментами розробника. Завдяки тому, що PyQt є крос-платформною бібліотекою, розроблений інтерфейс може бути легко інтегрований з різними середовищами розробки, такими як PyCharm, VS Code, Eclipse та іншими. Це дозволяє програмістам використовувати інтелектуального помічника безпосередньо в своєму звичному робочому оточенні, не перемикаючись між різними вікнами та додатками.

Окрім того, PyQt надає широкі можливості для створення власних віджетів та розширення функціональності існуючих. Це дозволяє адаптувати

інтерфейс до специфічних потреб конкретного проєкту або команди розробників. Наприклад, можна створити віджет для відображення результатів автоматизованого тестування коду, для інтеграції з системою керування версіями або для візуалізації даних, що використовуються в проєкті. Це робить розроблений інтерфейс гнучким та розширюваним інструментом, який може бути адаптований до різних завдань та умов використання.

Також було проведено порівняння зручності та ефективності розробленого інтерфейсу з інтерфейсами існуючих рішень, таких як GitHub Copilot та Tabnine. Було відзначено, що інтерфейс розробленого помічника є більш інтуїтивно зрозумілим та зручним для користувачів, які не мають досвіду роботи з подібними інструментами. Це досягається завдяки використанню звичного формату чат-бота, що імітує спілкування з реальною людиною, а також завдяки простому та лаконічному дизайну інтерфейсу, який не перевантажений зайвими елементами.

Важливою перевагою розробленого інтерфейсу є його локальна робота, що забезпечує конфіденційність та безпеку даних користувача. На відміну від хмарних сервісів, таких як GitHub Copilot та Tabnine, система не передає дані на сторонні сервери, що є критично важливим фактором для компаній, які працюють з чутливою інформацією. Локальна робота також гарантує незалежність від наявності та стабільності інтернет-з'єднання, що дозволяє використовувати помічника в будь-яких умовах.

Ще однією перевагою є можливість гнучкого налаштування інтерфейсу під потреби конкретного користувача або проєкту. Завдяки використанню бібліотеки PyQt, розробники можуть легко змінювати зовнішній вигляд та поведінку інтерфейсу, додавати нові елементи керування та інтегрувати помічника з іншими інструментами розробки. Це дозволяє створити максимально комфортне та продуктивне робоче середовище, адаптоване до індивідуальних потреб кожного користувача.

В цілому, результати тестування показали, що розроблений інтерфейс чат-бота є зручним, ефективним та інтуїтивно зрозумілим інструментом, який значно полегшує процес розробки програмного забезпечення. Використання бібліотеки PyQt дозволило створити гнучкий та розширюваний інтерфейс, який може бути адаптований до потреб конкретного користувача або проєкту. Локальна робота системи забезпечує конфіденційність даних та незалежність від зовнішніх факторів, що є суттєвою перевагою в порівнянні з хмарними аналогами.

Подальший розвиток інтерфейсу буде спрямований на вдосконалення його функціональності та зручності використання. Планується додати можливість зберігати історію діалогів та повертатися до попередніх запитів, а також реалізувати більш зручний механізм відображення результатів пошуку з можливістю перегляду фрагментів тексту безпосередньо у вікні чат-бота. Також планується розширити можливості кастомізації інтерфейсу, дозволяючи користувачам налаштовувати зовнішній вигляд та поведінку системи відповідно до своїх вподобань. Використання бібліотеки PyQt надає широкі можливості для реалізації цих планів та створення максимально зручного та ефективного інструменту для програмістів.

3.4. Визначення можливих напрямків подальшого розвитку та вдосконалення системи

Одним з ключових напрямків розвитку є розширення та актуалізація бази знань. Це передбачає інтеграцію нових джерел інформації, таких як наукові статті, технічні блоги, форуми та документація до нових версій мов програмування, бібліотек і фреймворків. Таке розширення дозволить системі надавати відповіді на ширший спектр запитів та охоплювати більшу кількість тем, пов'язаних з розробкою програмного забезпечення. Крім того, планується впровадити механізми автоматичного відстеження оновлень в існуючих джерелах та оперативного внесення змін до бази знань, що забезпечить її актуальність та достовірність. Для цього будуть розроблені

спеціальні парсери та інструменти моніторингу, які регулярно перевірятимуть наявність оновлень та автоматично додаватимуть їх до системи.

Поряд з розширенням бази знань, планується вдосконалити процес її обробки та індексації. Це включає оптимізацію алгоритмів розбиття тексту на фрагменти, а також налаштування параметрів індексу FAISS, таких як тип індексу, метрика подібності та інші, для забезпечення більш точного та швидкого пошуку релевантної інформації. Оптимізація цих параметрів дозволить покращити якість пошуку, зменшити час відгуку системи та підвищити її загальну продуктивність. Також планується дослідити можливість використання різних алгоритмів розбиття тексту на фрагменти, що дозволить адаптувати систему до різних типів документів та підвищити точність пошуку.

Іншим важливим напрямком є вдосконалення механізму генерації відповідей. Планується провести тонке налаштування параметрів мовної моделі, таких як температура, `top_k`, `top_p`, `repetition_penalty` та інших, з метою підвищення якості генерації. Також буде досліджено можливість використання більш досконалих алгоритмів фільтрації та ранжування згенерованого тексту, що дозволить відсіювати менш релевантні та неякісні відповіді. Крім того, планується застосувати техніки, що дозволяють генерувати більш детальні та структуровані відповіді, з чітким виділенням основних тез, прикладів коду та посилань на джерела. Це зробить відповіді більш інформативними та зручними для сприйняття користувачами.

Особлива увага буде приділена покращенню форматування коду у відповідях. Для цього планується розробити механізм, який буде автоматично визначати мову програмування, що згадується в запиті, та застосовувати відповідні правила форматування, включаючи відступи, підсвітку синтаксису та розстановку пробілів. Це зробить код у відповідях більш читабельним та легшим для розуміння. Також планується додати можливість копіювання фрагментів коду з відповідей в один клік, що

спростить процес інтеграції запропонованих рішень у власний код розробника.

Ще одним напрямком розвитку є вдосконалення інтерфейсу користувача. Незважаючи на те, що поточна версія інтерфейсу вже є досить зручною та інтуїтивно зрозумілою, планується провести додаткові дослідження та тестування з метою виявлення можливих проблем та незручностей. На основі отриманих даних будуть внесені зміни до дизайну та функціональності інтерфейсу, щоб зробити його ще більш комфортним та ефективним для користувачів. Зокрема, планується додати можливість зберігати історію діалогів, щоб користувачі могли легко повертатися до попередніх запитів та відповідей.

Планується також покращити механізм відображення результатів пошуку. Користувачам буде надана можливість переглядати не лише заголовки документів, але й короткі фрагменти тексту, що містять знайдену інформацію. Це дозволить швидше оцінювати релевантність документів та знаходити потрібну інформацію без необхідності відкривати кожен документ окремо. Крім того, планується додати можливість сортування та фільтрації результатів пошуку за різними критеріями, такими як дата публікації, джерело, тип документа та інші, що зробить процес пошуку інформації ще більш зручним та ефективним.

Важливим напрямком розвитку є впровадження механізмів персоналізації та адаптації системи до індивідуальних потреб користувачів. Планується розробити алгоритми, які будуть аналізувати історію взаємодії користувача з системою, запам'ятовувати його вподобання, улюблені мови програмування, часто використовувані бібліотеки та фреймворки, а також стиль написання коду та формулювання запитів. На основі зібраної інформації система зможе надавати персоналізовані рекомендації, підказки та відповіді, що максимально відповідають потребам та рівню підготовки кожного конкретного користувача.

Розширення функціональності системи є ще одним пріоритетним напрямком. Планується додати нові можливості, які можуть бути корисними для програмістів. Зокрема, розглядається можливість розробки функції автоматичного рефакторингу коду, яка дозволить користувачам швидко та легко покращувати структуру та читабельність свого коду. Також планується додати можливість генерації коментарів та документації до коду, що зекономить час розробників та підвищить якість програмної документації.

Крім того, розглядається можливість інтеграції з системами автоматизованого тестування коду, що дозволить користувачам запускати тести безпосередньо з інтерфейсу помічника та отримувати звіти про результати тестування.

Одним з важливих завдань є покращення обробки складних запитів, що містять технічні терміни, аббревіатури та професійний сленг. Для цього планується розширити та вдосконалити словник системи, а також покращити алгоритми розпізнавання сутностей та інтенцій користувача. Це дозволить системі краще розуміти складні та неоднозначні запити, а також знаходити релевантну інформацію навіть у тих випадках, коли запит сформульовано не зовсім точно або містить помилки. Крім того, планується дослідити можливість використання більш потужних мовних моделей, таких як GPT-4, для покращення якості обробки природної мови та генерації відповідей.

Розглядається можливість інтеграції системи з популярними середовищами розробки (IDE), такими як PyCharm, VS Code, IntelliJ IDEA та інші. Це дозволить програмістам використовувати інтелектуального помічника безпосередньо в своєму звичному робочому оточенні, не перемикаючись між різними вікнами та додатками. Планується розробити плагіни для популярних IDE, які забезпечать безшовну інтеграцію з системою та дозволять користувачам отримувати доступ до її функціоналу безпосередньо з редактора коду.

Ще одним напрямком розвитку є покращення масштабованості та відмовостійкості системи. Планується дослідити можливість розгортання

системи на кластері серверів, що дозволить забезпечити високу доступність та продуктивність при великій кількості одночасних користувачів. Також буде розроблено механізми резервного копіювання та відновлення даних, що дозволить захистити систему від втрати інформації в разі апаратних збоїв або інших непередбачених обставин. Це гарантуватиме стабільну роботу системи та її доступність для користувачів у будь-який момент часу.

Важливим аспектом подальшого розвитку є забезпечення безпеки та конфіденційності даних користувачів. Планується впровадити додаткові механізми захисту, такі як шифрування даних, контроль доступу та аудит дій користувачів. Це дозволить гарантувати, що конфіденційна інформація, яка може міститися в запитах користувачів або в базі знань системи, не потрапить до сторонніх осіб. Також буде розроблена політика конфіденційності, яка чітко регламентуватиме, які дані збираються системою, як вони використовуються та захищаються.

Планується розширити мовну підтримку системи. Наразі система орієнтована на англomовний контент, але в майбутньому планується додати підтримку інших мов програмування та локалізацію інтерфейсу на інші мови, що зробить систему доступною для більш широкої аудиторії розробників. Це дозволить програмістам з усього світу використовувати інтелектуального помічника у своїй роботі, незалежно від того, якою мовою вони спілкуються.

Також планується створити API для інтеграції з іншими інструментами та сервісами. Це дозволить розробникам вбудовувати функціональність інтелектуального помічника у свої власні додатки та розширювати його можливості. Наприклад, можна буде інтегрувати помічника з системами управління проектами, баг-трекерами, платформами для спільної роботи та іншими інструментами, що використовуються в процесі розробки програмного забезпечення.

Одним з напрямків розвитку є створення спільноти користувачів та розробників навколо системи. Це дозволить обмінюватися досвідом, ідеями та найкращими практиками використання інтелектуального помічника.

Також планується створити платформу для збору зворотного зв'язку від користувачів, що дозволить оперативно реагувати на їхні потреби та побажання. Спільнота користувачів зможе брати участь у тестуванні нових функцій, пропонувати ідеї щодо покращення системи та допомагати один одному у вирішенні складних завдань.

Нарешті, планується продовжити дослідження в галузі штучного інтелекту та машинного навчання з метою вдосконалення алгоритмів, що лежать в основі інтелектуального помічника. Це включає в себе роботу над покращенням якості та швидкості пошуку, генерації відповідей, а також над розробкою нових функцій, що можуть бути корисними для програмістів. Планується експериментувати з новими архітектурами нейронних мереж, методами навчання та оптимізації, а також досліджувати можливості застосування новітніх досягнень в галузі обробки природної мови для створення ще більш ефективного та інтелектуального помічника.

Висновки до розділу 3

У третьому розділі цієї роботи було детально описано комплексну методику тестування та оцінки якості, яка була застосована до розробленого інтелектуального помічника програміста. Ця методика охоплювала як автоматизовані методи оцінки ефективності окремих компонентів системи, так і ручне тестування, спрямоване на оцінку загальної зручності та корисності системи в реальних сценаріях використання. Завдяки такому всебічному підходу вдалося отримати цілісну картину роботи системи, виявити її сильні та слабкі сторони, а також сформулювати напрямки подальшого розвитку та вдосконалення.

Проведене тестування підтвердило, що розроблена система демонструє високу якість роботи, особливо у частині, що стосується точності та релевантності відповідей на запити користувачів. Модуль пошуку інформації, який базується на архітектурі RAG, бібліотеці LlamaIndex та векторній базі даних FAISS, ефективно знаходить релевантні документи у

великій базі знань, а модуль генерації відповідей, що використовує мовну модель Llama, формує змістовні та корисні відповіді, адаптовані до контексту запиту. Це свідчить про те, що обрані технології та архітектурні рішення є ефективними для створення інтелектуальних помічників програміста.

Окрім високої точності та релевантності, система також продемонструвала зручність використання завдяки інтуїтивно зрозумілому інтерфейсу чат-бота. Можливість ставити запитання природною мовою та отримувати відповіді у звичному форматі діалогу значно полегшує взаємодію з системою та робить її доступною для широкого кола користувачів, незалежно від їхнього досвіду роботи з подібними інструментами. Крім того, інтерфейс автоматично адаптується до різних розмірів екрану, що дозволяє комфортно працювати з системою як на стаціонарних комп'ютерах, так і на мобільних пристроях. Це розширює можливості використання.

Порівняльний аналіз з існуючими рішеннями, такими як GitHub Copilot та Tabnine, показав, що розроблений інтелектуальний помічник має ряд суттєвих переваг. По-перше, завдяки використанню RAG-архітектури, система може знаходити релевантну інформацію не лише у загальнодоступних джерелах, але й у внутрішній документації проєкту, що забезпечує більш точні та контекстно-залежні відповіді. По-друге, інтерфейс чат-бота є більш зручним та інтуїтивно зрозумілим для постановки питань та отримання розгорнутих відповідей, ніж інтерфейси автодоповнення коду, які пропонують GitHub Copilot та Tabnine. По-третє, можливість локального розгортання системи гарантує конфіденційність даних та незалежність від зовнішніх сервісів, що є важливим фактором для багатьох компаній та розробників.

Втім, проведене тестування також виявило деякі недоліки та обмеження поточної версії системи. Зокрема, було помічено, що у деяких випадках система генерує занадто загальні відповіді, які не містять достатньої кількості конкретних прикладів коду або детальних інструкцій. Також було виявлено певні труднощі при обробці складних

багатокомпонентних запитів, що містять технічні терміни, аббревіатури та професійний сленг. Крім того, було відзначено, що поточна версія системи не завжди ідеально форматує код у відповідях, що може ускладнювати його сприйняття. Ці недоліки стали основою для визначення перспективних напрямків подальшого розвитку та вдосконалення системи.

Одним із пріоритетних напрямків є розширення та актуалізація бази знань. Планується інтегрувати в систему нові джерела інформації, включаючи наукові статті, технічні блоги, форуми, а також документацію до нових версій мов програмування, бібліотек та фреймворків. Це дозволить системі охопити більший спектр тем та завдань, пов'язаних з розробкою програмного забезпечення. Також планується вдосконалити алгоритми обробки та індексації документів, що підвищить точність та швидкість пошуку релевантної інформації. Окрім того, передбачається впровадити механізми автоматичного відстеження оновлень в існуючих джерелах та оперативного внесення змін до бази знань, що забезпечить її актуальність.

Іншим важливим напрямком є вдосконалення механізму генерації відповідей. Планується провести тонке налаштування параметрів мовної моделі, а також дослідити можливість використання більш досконалих алгоритмів фільтрації та ранжування згенерованого тексту. Це дозволить підвищити точність, релевантність, конкретність та повноту відповідей. Також планується застосувати техніки, що дозволять генерувати більш детальні та структуровані відповіді, з чітким виділенням основних тез, прикладів коду та посилань на джерела. Особливу увагу буде приділено покращенню форматування коду у відповідях, щоб зробити його більш читабельним та зручним для копіювання.

Ще одним напрямком розвитку є вдосконалення інтерфейсу користувача. Незважаючи на те, що поточна версія інтерфейсу вже є досить зручною, планується провести додаткові дослідження та тестування з метою виявлення можливих проблем та незручностей. На основі отриманих даних будуть внесені зміни до дизайну та функціональності інтерфейсу. Зокрема,

планується додати можливість зберігати історію діалогів, щоб користувачі могли легко повертатися до попередніх запитів та відповідей. Також планується покращити механізм відображення результатів пошуку, надавши користувачам можливість переглядати не лише заголовки документів, але й короткі фрагменти тексту, що містять знайдену інформацію.

Важливим аспектом розвитку системи є впровадження механізмів персоналізації та адаптації до індивідуальних потреб користувачів. Планується розробити алгоритми, які будуть аналізувати історію взаємодії користувача з системою, запам'ятовувати його вподобання, улюблені мови програмування, часто використовувані бібліотеки та фреймворки, а також стиль написання коду та формулювання запитів. На основі зібраної інформації система зможе надавати персоналізовані рекомендації, підказки та відповіді, що максимально відповідають потребам та рівню підготовки кожного конкретного користувача.

Окрім того, планується розширити функціональність системи, додавши нові можливості, які можуть бути корисними для програмістів. Зокрема, розглядається можливість розробки функції автоматичного рефакторингу коду, яка дозволить користувачам швидко та легко покращувати структуру та читабельність свого коду. Також планується додати можливість генерації коментарів та документації до коду, що зекономить час розробників та підвищить якість програмної документації. Крім того, розглядається можливість інтеграції з системами автоматизованого тестування коду, що дозволить користувачам запускати тести безпосередньо з інтерфейсу помічника та отримувати звіти про результати тестування.

Подальший розвиток системи також передбачає покращення обробки складних запитів, що містять технічні терміни, аббревіатури та професійний сленг. Планується розширити та вдосконалити словник системи, а також покращити алгоритми розпізнавання сутностей та інтенцій користувача. Це дозволить системі краще розуміти складні та неоднозначні запити, а також знаходити релевантну інформацію навіть у тих випадках, коли запит

сформульовано не зовсім точно або містить помилки. Також розглядається можливість використання більш потужних мовних моделей, таких як GPT-4, для покращення якості обробки природної мови та генерації відповідей.

Планується також розширити можливості інтеграції системи з іншими інструментами розробки. Зокрема, розглядається можливість створення плагінів для популярних IDE, таких як PyCharm, VS Code, IntelliJ IDEA, що дозволить програмістам використовувати інтелектуального помічника безпосередньо в своєму звичному робочому середовищі. Це зробить роботу з системою ще більш зручною та ефективною, а також дозволить інтегрувати її з іншими інструментами, такими як системи контролю версій, баг-трекери та платформи для спільної роботи.

Окрім того, планується покращити масштабованість та відмовостійкість системи. Розглядається можливість розгортання системи на кластері серверів, що дозволить забезпечити високу доступність та продуктивність при великій кількості одночасних користувачів. Також буде розроблено механізми резервного копіювання та відновлення даних, що дозволить захистити систему від втрати інформації в разі апаратних збоїв або інших непередбачених обставин. Це гарантуватиме стабільну роботу системи та її доступність для користувачів у будь-який момент часу.

Важливим аспектом подальшого розвитку є забезпечення безпеки та конфіденційності даних користувачів. Планується впровадити додаткові механізми захисту, такі як шифрування даних, контроль доступу та аудит дій користувачів. Це дозволить гарантувати, що конфіденційна інформація, яка може міститися в запитах користувачів або в базі знань системи, не потрапить до сторонніх осіб. Також буде розроблена політика конфіденційності, яка чітко регламентуватиме, які дані збираються системою, як вони використовуються та захищаються.

Нарешті, планується створити спільноту користувачів та розробників навколо системи, що дозволить обмінюватися досвідом, ідеями та найкращими практиками використання інтелектуального помічника. Також

планується створити платформу для збору зворотного зв'язку від користувачів, що дозволить оперативно реагувати на їхні потреби та побажання. Спільнота користувачів зможе брати участь у тестуванні нових функцій, пропонувати ідеї щодо покращення системи та допомагати один одному у вирішенні складних завдань. Це сприятиме розвитку системи та її адаптації до реальних потреб програмістів.

ВИСНОВКИ

У кваліфікаційній роботі було розроблено та реалізовано інтелектуального помічника програміста на основі архітектури RAG (Retrieval-Augmented Generation). Ця архітектура, що поєднує в собі потужність мовних моделей з ефективністю інформаційно-пошукових систем, дозволила створити інструмент, здатний не лише генерувати текст, але й знаходити релевантну інформацію у великій базі знань та формувати на її основі змістовні та точні відповіді на запити користувачів.

В рамках роботи було проведено детальний аналіз предметної області, який включав огляд існуючих рішень у сфері інтелектуальних помічників програміста та обґрунтування вибору RAG-архітектури як найбільш перспективної для даного проєкту. Було сформульовано основні вимоги до системи, серед яких: здатність обробляти запити природною мовою, наявність доступу до великої та актуальної бази знань, зручний інтерфейс користувача та можливість інтеграції з існуючими інструментами розробника.

На основі проведеного аналізу було спроектовано та реалізовано архітектуру інтелектуального помічника, що складається з декількох ключових компонентів: інтерфейсу користувача, модуля обробки запитів, модуля пошуку інформації, модуля генерації відповідей та бази знань. Інтерфейс користувача виконано у форматі чат-бота, що забезпечує зручну та інтуїтивно зрозумілу взаємодію з системою. Модуль обробки запитів здійснює попередню обробку вхідного запиту, перетворюючи його на векторне представлення. Модуль пошуку інформації, використовуючи бібліотеку LlamaIndex та векторну базу даних FAISS, знаходить релевантні документи в базі знань. Модуль генерації відповідей, на основі мовної моделі `aya-expanse-8b` та бібліотеки `Llama.cpp`, формує змістовну та точну відповідь на запит користувача, спираючись на знайдену інформацію.

Розроблена система підтримує імпорт документів різних форматів (.csv, .doc, .docx, .enex, .epub, .html, .md, .odt, .pdf, .ppt, .pptx, .txt), що дозволяє наповнювати базу знань з різноманітних джерел. Реалізовано механізм розбиття тексту на фрагменти оптимального розміру з урахуванням семантичної зв'язності. Для ефективного пошуку інформації у векторній базі даних було досліджено та застосовано різні типи індексів FAISS (FLAT, IVF_FLAT, HNSW, PQ) та метрики подібності (L2, IP).

Особливу увагу було приділено розробці зручного та інтуїтивно зрозумілого інтерфейсу користувача. Завдяки використанню бібліотеки PyQt вдалося створити адаптивний інтерфейс, який коректно відображається на екранах з різною роздільною здатністю та співвідношенням сторін. Інтерфейс чат-бота дозволяє користувачам ставити запитання природною мовою та отримувати відповіді у звичному форматі діалогу. Реалізовано функціонал форматування тексту відповідей, включаючи виділення блоків коду, що покращує сприйняття інформації.

Проведено всебічне тестування розробленого програмного забезпечення, що включало як автоматизоване, так і ручне тестування. Автоматизоване тестування було сфокусоване на оцінці ефективності роботи окремих модулів системи, таких як модуль пошуку та модуль генерації. Для цього використовувалися спеціалізовані метрики, такі як Context Precision, Context Recall, Mean Reciprocal Rank (MRR) для модуля пошуку, а також Faithfulness, Answer Relevance, Answer Semantic Similarity та Answer Correctness для модуля генерації. Ручне тестування проводилося за участі розробників та сторонніх програмістів і було спрямоване на оцінку загальної зручності, ефективності та корисності системи в реальних сценаріях використання.

Результати тестування підтвердили високу якість роботи системи. Інтелектуальний помічник демонструє високу точність та релевантність відповідей, ефективно знаходить необхідну інформацію у базі знань та генерує змістовні, добре структуровані відповіді. Інтерфейс чат-бота є

зручним та інтуїтивно зрозумілим, а час відгуку системи знаходиться в межах комфортного для користувача діапазону.

Порівняльний аналіз з існуючими рішеннями, такими як GitHub Copilot та Tabnine, показав, що розроблений помічник має ряд переваг. По-перше, завдяки RAG-архітектурі, система здатна знаходити та використовувати інформацію з великої бази знань, що забезпечує більш точні та контекстно-залежні відповіді. По-друге, інтерфейс чат-бота є більш зручним для постановки питань та отримання розгорнутих відповідей, ніж інтерфейси автодоповнення коду, які пропонують GitHub Copilot та Tabnine. По-третє, і це особливо важливо, розроблена система може бути розгорнута локально, що гарантує конфіденційність даних та незалежність від зовнішніх сервісів.

В результаті виконання кваліфікаційної роботи було створено ефективний інструмент, що здатен значно полегшити процес розробки програмного забезпечення, підвищити продуктивність програмістів та сприяти створенню більш якісного коду. Розроблений інтелектуальний помічник програміста демонструє високий потенціал для практичного застосування та подальшого розвитку. Його здатність поєднувати можливості сучасних мовних моделей з ефективними методами пошуку інформації відкриває нові перспективи для автоматизації рутинних завдань розробки та надання інтелектуальної підтримки програмістам на всіх етапах створення програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. PyQt Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://doc.qt.io/qtforpython/>.
2. Llama-cpp Repository. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.llamaindex.ai/en/stable/examples/llm/llama_2_llama_cpp/.
3. LlamaIndex Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.llamaindex.ai/>.
4. FAISS (Facebook AI Similarity Search). [Електронний ресурс] – Режим доступу до ресурсу: <https://faiss.ai/>.
5. SQLAlchemy Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://www.sqlalchemy.org/>.
6. Helsinki-NLP Models on Hugging Face. [Електронний ресурс] – Режим доступу до ресурсу: <https://huggingface.co/Helsinki-NLP>.
7. Transformers Library Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://huggingface.co/docs/transformers/index>.
8. Lingua Library GitHub Repository. [Електронний ресурс] – Режим доступу до ресурсу: <https://app.codecov.io/gh/pemistahl/lingua>.
9. Markdown2 Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://pypi.org/project/markdown2/>.
10. Pygments Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://pygments.org/docs/>.
11. LangChain Text Splitters Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://python.langchain.com/docs/how_to/#text-splitters.
12. TokenTextSplitter from Llama Index. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.llamaindex.ai/en/stable/api_reference/node_parsers/token_text_splitter/.
13. PyTorch Documentation. [Електронний ресурс] – Режим доступу до ресурсу: <https://pytorch.org/docs/stable/index.html>.

- 14.TensorFlow Documentation. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.tensorflow.org/>.
- 15.Python-Markdown Documentation. [Электронный ресурс] – Режим доступа до ресурсу: <https://python-markdown.github.io/>.
- 16.Mistune Documentation. [Электронный ресурс] – Режим доступа до ресурсу: <https://mistune.readthedocs.io/>.
- 17.What is retrieval-augmented generation (RAG)?. [Электронный ресурс] – Режим доступа до ресурсу: <https://research.ibm.com/blog/retrieval-augmented-generation-RAG>.
- 18.What is Retrieval Augmented Generation (RAG)?. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.elastic.co/what-is/retrieval-augmented-generation>.
- 19.Retrieval-Augmented Generation for AI-Generated Content: A Survey. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2402.19473>.
- 20.What is Retrieval-Augmented Generation (RAG)?. [Электронный ресурс] – Режим доступа до ресурсу: <https://cloud.google.com/use-cases/retrieval-augmented-generation>.
- 21.Retrieval Augmented Generation (RAG). [Электронный ресурс] – Режим доступа до ресурсу: <https://www.pinecone.io/learn/retrieval-augmented-generation/>.
- 22.What is retrieval-augmented generation?. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.redhat.com/en/topics/ai/what-is-retrieval-augmented-generation>.
- 23.From RAG to Riches: Retrieval-Augmented Generation, Explained. [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@alexgnibus/from-rag-to-riches-retrieval-augmented-generation-explained-2f55efdc7fa6>.

24. What is retrieval-augmented generation (RAG)?. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.box.com/resources/what-is-retrieval-augmented-generation>.
25. Haystack Documentation. [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.haystack.deepset.ai/>.
26. Vectara Documentation. [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.vectara.com/>.
27. Dense Retrieval for Conversational AI. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.amazon.science/blog/dense-retrieval-for-conversational-ai>.
28. OpenAI Embeddings. [Электронный ресурс] – Режим доступа до ресурсу: <https://platform.openai.com/docs/guides/embeddings>.
29. FlashRAG: A Python Toolkit for Efficient RAG Research. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2405.13576>.
30. Learn RAG from Scratch – Python AI Tutorial from a LangChain Expert. [Электронный ресурс] – Режим доступа до ресурсу: <https://www.freecodecamp.org/news/mastering-rag-from-scratch/>.
31. Build a Retrieval Augmented Generation (RAG) App: Part 1. [Электронный ресурс] – Режим доступа до ресурсу: <https://python.langchain.com/docs/tutorials/rag/>.
32. RAG for Multimodal Information Retrieval. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2204.05625>.
33. Open-Domain Question Answering with Retrieval-Augmented Generation. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/1901.08149>.
34. RAG in Business Intelligence. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2205.01817>.
35. Retrieval-Augmented Generation with Pretrained Models. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2205.04856>.

36. End-to-End Training for Retrieval-Augmented Generation. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2101.09022>.
37. Using Retrieval-Augmented Generation for Text Summarization. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2302.04688>.
38. Combining Transformers and RAG for Complex Question Answering. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2301.05469>.
39. A Survey of Retrieval-Augmented Generation Techniques. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2206.07471>.
40. Attention is All You Need. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/1706.03762>.
41. Transformers for NLP: State-of-the-Art Techniques and Applications. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2102.06087>.
42. GPT-3: Language Models are Few-Shot Learners. [Электронный ресурс] – Режим доступа до ресурсу: <https://arxiv.org/abs/2005.14165>.

ДОДАТКИ

Лістинг програми

```

```py
from PyQt6.QtCore import QSettings, QSize

class ConfigManager:
 def __init__(self, filename="config.ini"):
 self.settings = QSettings(filename, QSettings.Format.IniFormat)

 def set_value(self, key, value):
 """Зберігає значення за вказаним ключем."""
 self.settings.setValue(key, value)

 def get_value(self, key, default=None):
 """Отримує значення за ключем, або повертає дефолтне."""
 return self.settings.value(key, default)

 @property
 def db_url(self):
 """Отримує URL бази даних з конфігурації або повертає дефолтний шлях."""
 return self.get_value("database/db_url", "sqlite:///app.db")

 @db_url.setter
 def db_url(self, value):
 """Зберігає новий URL бази даних у конфігурацію."""
 self.set_value("database/db_url", value)

 @property
 def path_opus_mt_en_uk(self):
 return self.get_value("database/path_opus_mt_en_uk", "cache_dir")

 @path_opus_mt_en_uk.setter
 def path_opus_mt_en_uk(self, value):
 self.set_value("database/path_opus_mt_en_uk", value)

 @property
 def path_opus_mt_ru_en(self):
 return self.get_value("database/path_opus_mt_ru_en", "cache_dir")

 @path_opus_mt_ru_en.setter
 def path_opus_mt_ru_en(self, value):
 self.set_value("database/path_opus_mt_ru_en", value)

 @property
 def path_opus_mt_uk_en(self):
 return self.get_value("database/path_opus_mt_uk_en", "cache_dir")

 @path_opus_mt_uk_en.setter
 def path_opus_mt_uk_en(self, value):
 self.set_value("database/path_opus_mt_uk_en", value)

Ініціалізація конфігурації
config = ConfigManager()
```

```py
from PyQt6.QtCore import QObject, pyqtSignal
from AppConfig import config
from DB import DatabaseManager
from PyQt6.QtWidgets import QListWidgetItem

```

```

from PyQt6.QtCore import Qt, pyqtSlot
from ModelController import ModelController
from ChatObject import ChatObject
from DBModels import ChatModel, ChatMessageModel, LargeModel, EmbeddingModel, EmbeddingDatabase,
RagMessageModel
from HighlightProcessor import HighlightProcessor
from VectorDBController import VectorDBController
from EngineEmbeddingDatabase import EngineEmbeddingDatabase
from typing import List
import os
from TranslationModel import TranslationModel

class AppController(QObject):
 signal_chat_loaded = pyqtSignal()
 signal_create_bot_response = pyqtSignal()
 signal_clear_chat_widget = pyqtSignal()
 signal_update_display_chat_messages = pyqtSignal(ChatObject)
 signal_text_generated = pyqtSignal(ChatObject)
 signal_finish_text_generated = pyqtSignal(ChatObject)
 signal_update_button_send_message = pyqtSignal(QObject)
 signal_list_all_reload_models = pyqtSignal()

 signal_list_embedding_models_reload = pyqtSignal()

 signal_list_db_models_reload = pyqtSignal()

 signal_update_last_user_text = pyqtSignal(str)

 def __init__(self):
 super().__init__()
 self.config = config
 self.database_manager = DatabaseManager(db_url=config.db_url)
 self.vector_db_controller: VectorDBController = None

 self.chat_object: ChatObject = ChatObject(db=self.database_manager)

 from ChatWidget import ChatWidget
 self.chat_widget: ChatWidget = None

 self.models = []

 self.selected_model = None
 self.model_controller = None
 self.last_message_text = ""

 self.bot_thread_is_work = False

 self.rag_is_select = False

 def set_chat_widget(self, chat_widget):
 self.chat_widget = chat_widget

 def deselect_all_embedding_database(self):
 models: List[EmbeddingDatabase] = self.get_list_embeddings_databases()
 for model in models:
 model.select_status = False
 self.update_db_model(model)

 def get_list_exist_models(self):
 models: List[LargeModel] = self.database_manager.get_all_records(LargeModel)
 return models

```

```

def get_combo_models(self):
 models: List[LargeModel] = self.database_manager.get_all_records(LargeModel)
 datanames = []
 for model in models:
 file_name_without_extension = os.path.splitext(os.path.basename(model.model_path))[0]
 datanames.append((file_name_without_extension, model.id))
 print(datanames)
 return datanames

def get_list_embeddings_databases(self):
 models: List[EmbeddingDatabase] = self.database_manager.get_all_embedding_databases()
 return models

def get_selected_embeddings_databases(self):
 models: List[EmbeddingDatabase] = self.vector_db_controller.selected_db_models
 return models

def get_combo_selected_embedding_database(self):
 models: List[EmbeddingDatabase] = self.vector_db_controller.selected_db_models
 datanames = []
 for model in models:
 datanames.append((model.db_name, model.id))
 return datanames

def get_combo_embedding_database(self):
 models: List[EmbeddingDatabase] = self.database_manager.get_all_embedding_databases()
 datanames = []
 for model in models:
 datanames.append((model.db_name, model.id))
 return datanames

@pyqtSlot(LargeModel)
def slot_select_model(self, model: LargeModel):
 self.selected_model = model
 if self.selected_model.model_path != "":
 self.model_controller = ModelController(
 model_path=self.selected_model.model_path,
 n_ctx=self.selected_model.context_size,
 n_threads=self.selected_model.num_threads,
 temperature=self.selected_model.temperature
)
 self.model_controller.thread_is_stopped.connect(self.slot_thread_is_stopped)

@pyqtSlot(LargeModel)
def slot_deselect_model(self, model: LargeModel):
 if self.selected_model is model:
 del self.model_controller
 self.model_controller = None
 self.selected_model = None

@pyqtSlot()
def slot_thread_is_stopped(self):
 self.bot_thread_is_work = False
 self.signal_update_button_send_message.emit(self)

text_proc = HighlightProcessor()

message: ChatMessageModel
self.database_manager.get_message_by_id(self.chat_object.last_bot_message_id)
message.processed_message = text_proc.highlight_code_blocks(message.message)
self.database_manager.save_to_database(message)
self.signal_finish_text_generated.emit(self.chat_object)

```

=

```

def get_all_models(self):
 self.models = self.database_manager.get_all_records(LargeModel)
 return self.models

def set_vector_db_controler(self, vector_db_controler):
 self.vector_db_controler = vector_db_controler

@pyqtSlot(LargeModel)
def update_model(self, model:LargeModel):
 self.database_manager.save_to_database(model)

@pyqtSlot(EmbeddingModel)
def update_embedding_model(self, model:EmbeddingModel):
 self.database_manager.save_to_database(model)

@pyqtSlot(EmbeddingModel)
def delete_embedding_model(self, model:EmbeddingModel):
 self.database_manager.delete_record(EmbeddingModel, model.id)
 self.signal_list_embedding_models_reload.emit()

@pyqtSlot(QListWidgetItem)
def on_chat_list_item_clicked(self, item:QListWidgetItem):
 extra_data_id = item.data(Qt.ItemDataRole.UserRole)
 print(f"Clicked on: {item.text()} id = {extra_data_id}")

 self.signal_clear_chat_widget.emit()

 chat:ChatModel = self.database_manager.get_record_by_id(extra_data_id, ChatModel)
 self.signal_clear_chat_widget.emit()

 self.chat_object = ChatObject(db=self.database_manager)
 self.chat_object.chat = chat
 self.chat_object.current_chat_id = chat.id
 self.chat_object.update_messages()

 self.signal_update_display_chat_messages.emit(self.chat_object)

def get_chat_list(self):
 all_chats = self.database_manager.get_all_records(ChatModel)
 result = []
 for chat in all_chats:
 if isinstance(chat, ChatModel):
 result.append((chat.title, chat.id))
 return result

@pyqtSlot()
def create_new_chat(self, first_message_is_exist=False):
 print("create new chat")
 new_chat = ChatModel()
 new_chat.title = "New Chat"
 record_id = self.database_manager.add_record(new_chat)
 chat:ChatModel = self.database_manager.get_record_by_id(record_id, ChatModel)
 self.chat_object.chat = chat
 self.chat_object.current_chat_id = chat.id
 self.chat_object.update_messages()
 if first_message_is_exist is False:
 self.signal_clear_chat_widget.emit()

def create_chat_message_model(
 self,
 model_path = "None",
 sender_user=False,
 sender_bot=False,

```

```

 text=""
):
 if self.chat_object.current_chat_id is None:
 self.create_new_chat()
 if sender_user is True:
 text_proc = HighlightProcessor()
 processed_message = text_proc.highlight_code_blocks(text)
 processed_message = text
 new_message = ChatMessageModel(
 chat_id=self.chat_object.current_chat_id,
 sender_user=sender_user,
 sender_bot=sender_bot,
 message=text,
 processed_message = processed_message,
 model_path=model_path
)

 message_id = self.database_manager.add_record(new_message)
 if sender_bot is True:
 self.chat_object.last_bot_message_id = message_id
 return message_id

@pyqtSlot(int)
def slot_select_rag_checkbox(self, state):
 self.rag_is_select = bool(state)

def create_rag_message_context(self, text_original:str, text_en:str):
 context: List[str] = []
 rag_messages: List[RagMessageModel] = []
 if self.vector_db_controler.get_count_selected_models() > 0:
 results: list[EngineEmbeddingDatabase.SearchResult] = self.vector_db_controler.query(text_en)
 if len(results) > 0:
 results.sort(key=lambda result: result.distance)
 for result in results:
 rag_message = RagMessageModel()
 rag_message.text = result.text
 context.append(result.text)
 rag_messages.append(rag_message)

 if len(context) > 0:
 #prompt = "\nContext:\n" + "\n".join(context) + "\n\nЗапит:\n" + text_original + " Відповідь на запит згідно контексту надавай українською мовою."
 prompt = "\nContext:\n"
 for item in context:
 prompt += "block text`\n" + item + "\n`\n"

 prompt += "\nЗапит:\n" + text_original + " Відповідь на запит згідно контексту надавай українською мовою."
 else:
 prompt = text_original

 return prompt, rag_messages

def add_new_user_message(self, text):
 print(f"add new user message {text}")

 if self.chat_object.current_chat_id is None:
 self.create_new_chat(first_message_is_exist=True)
 rag_messages: List[RagMessageModel] = []
 if self.rag_is_select is True:

 tr_model = TranslationModel()
 text_tr = tr_model.to_en(text)

```

```

 text, rag_messages = self.create_rag_message_context(text, text_tr)
 self.signal_update_last_user_text.emit(text)

 message_id = self.create_chat_message_model(sender_user=True, text=text)
 message = self.database_manager.get_message_by_id(message_id)
 for rag_message in rag_messages:
 rag_message.message = message
 self.database_manager.add_record(rag_message)
 self.last_message_bot_text = text

 self.create_bot_response()

 @pyqtSlot(str)
 def slot_text_generated(self, text):
 """Слот що приймає кожен згенерований токен"""
 text_proc = HighlightProcessor()
 message: ChatMessageModel
 self.database_manager.get_message_by_id(self.chat_object.last_bot_message_id)
 message.message = text
 message.processed_message = text_proc.highlight_code_blocks(message.message)
 self.database_manager.save_to_database(message)
 self.signal_text_generated.emit(self.chat_object)

 def create_bot_response(self):
 if self.chat_object.current_chat_id is None:
 self.create_new_chat(first_message_is_exist=True)

 text = ""

 self.create_chat_message_model(sender_bot=True, text=text)

 self.signal_create_bot_response.emit()

 if self.model_controller is not None:
 self.model_controller.create_thread_generate(self.last_message_bot_text, self.slot_text_generated)
 self.bot_thread_is_work = True

 self.signal_update_button_send_message.emit(self)

 @pyqtSlot()
 def stop_bot_generate(self):
 self.model_controller.stop_thread_generate()
 self.bot_thread_is_work = False

 @pyqtSlot()
 def clear_all_messages_active_chat(self):
 print("clear_all_messages_active_chat")
 if self.chat_object.current_chat_id is not None:
 self.database_manager.delete_messages_for_chat(self.chat_object.current_chat_id)
 self.chat_object.update_messages()
 self.signal_clear_chat_widget.emit()

 @pyqtSlot(LargeModel)
 def slot_delete_model(self, model: LargeModel):
 self.database_manager.delete_record(LargeModel, model.id)
 self.signal_list_all_reload_models.emit()

 @pyqtSlot(EmbeddingDatabase)
 def remove_db_model(self, db_model: EmbeddingDatabase):
 self.vector_db_controller.remove_db_index(db_model)
 self.database_manager.delete_text_embeddings_for_database(db_model.id)
 self.database_manager.delete_record(EmbeddingDatabase, db_model.id)

```

```

 self.signal_list_db_models_reload.emit()

 @pyqtSlot(EmbeddingDatabase)
 def update_db_model(self, db_model:EmbeddingDatabase):
 if db_model.embedding_model is None:
 print("EmbeddingDatabase embedding_model is None")
 self.database_manager.save_to_database(db_model)
 ...

'''
.py
from DB import ChatModel, ChatMessageModel, DatabaseManager
from typing import List

class ChatObject:
 def __init__(self, db:DatabaseManager):
 self.chat: ChatModel = None
 self.messages: List[ChatMessageModel] = []
 self.current_chat_id = None
 self.last_bot_message_id = None
 self.db = db

 def clear_all(self):
 self.chat = None
 self.messages = []
 self.current_chat_id = None
 self.last_bot_message_id = None

 def update_messages(self):
 self.messages = self.db.get_messages_for_chat(self.current_chat_id)

 def update_last_bot_message(self):

 message = self.db.get_message_by_id(self.last_bot_message_id)
 return message
 ...

'''
.py
from PyQt6.QtWidgets import QWidget, QVBoxLayout, QScrollArea, QLineEdit, QPushButton, QLabel,
QHBoxLayout, QApplication, QTextBrowser
from PyQt6.QtCore import Qt, pyqtSignal, pyqtSlot
from PyQt6.QtGui import QFont
import sys
from typing import List
from DB import ChatModel, ChatMessageModel
from ChatObject import ChatObject
from AppController import AppController

class ChatWidget(QWidget):
 message_sent = pyqtSignal(str)
 signal_stop_generate =pyqtSignal()
 #connect to add_new_user_message

 def __init__(self):
 super().__init__()
 self.setWindowTitle("ChatBot")

 # Налаштування темної теми
 self.setStyleSheet("""
 QWidget {
 background-color: #2E2E2E;
 color: #FFFFFF;
 font-size: 16px;

```

```

 }
 QScrollArea {
 border: none;
 }
 QLabel {
 background-color: #4B4B4B;
 border-radius: 8px;
 padding: 5px;
 font-size: 14px;
 }
 QLineEdit, QPushButton {
 background-color: #3E3E3E;
 border: 1px solid #555555;
 color: #FFFFFF;
 }
 QPushButton:hover {
 background-color: #5E5E5E;
 }
 """
)

Основне вертикальне розташування
self.layout_ = QVBoxLayout(self)

Зона для повідомлень чату
self.scroll_area = QScrollArea()
self.scroll_area.setWidgetResizable(True)
self.scroll_area.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)

self.messages_widget = QWidget()
self.messages_layout = QVBoxLayout(self.messages_widget)
self.messages_layout.setAlignment(Qt.AlignmentFlag.AlignTop)
self.scroll_area.setWidget(self.messages_widget)

self.layout_.addWidget(self.scroll_area)

Поле вводу та кнопка відправки
self.input_layout = QHBoxLayout()
self.input_field = QLineEdit()
self.input_field.setPlaceholderText("Введіть повідомлення...")
self.send_button = QPushButton("Відправити )
self.send_button.setStyleSheet("""
 QPushButton {
 background-color: #ECF0F1;
 color: #2C3E50;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #BDC3C7;
 }
 """)

self.input_layout.addWidget(self.input_field)
self.input_layout.addWidget(self.send_button)
self.layout_.addLayout(self.input_layout)

Підключення сигналів
self.send_button.clicked.connect(self.add_user_message)
self.input_field.returnPressed.connect(self.add_user_message)

self.last_bot_text_browser = None
self.last_user_text_browser = None

```

```

self.chat_object:ChatObject = None
self.app_controller:AppController = None

@pyqtSlot(AppController)
def update_button_send_message(self, app_controller:AppController):
 if app_controller.selected_model is None:
 self.send_button.setEnabled(False)
 elif app_controller.selected_model is not None:
 self.send_button.setEnabled(True)

 if app_controller.bot_thread_is_work is True:
 self.send_button.setText("Зупинити )
 else:
 self.send_button.setText("Відправити )

@pyqtSlot()
def create_bot_response(self):
 self.add_message_bubble("", "bot")

def add_user_message(self):
 if self.app_controller.bot_thread_is_work is False:
 # Отримання тексту з поля вводу
 user_text = self.input_field.text().strip()
 if user_text:
 # Додавання повідомлення користувача
 self.add_message_bubble(user_text, "user")

 self.message_sent.emit(user_text)

 self.input_field.clear()
 else:
 self.signal_stop_generate.emit()

@pyqtSlot(str)
def update_last_user_text_browser(self, text):
 if self.last_user_text_browser is not None:
 self.last_user_text_browser.setText(text)
 document_height = int(self.last_user_text_browser.document().size().height())
 self.last_user_text_browser.setFixedHeight(document_height)

def add_message_bubble(self, text, sender="user"):
 # Стилізація повідомлень
 bubble = QTextBrowser()
 bubble.setText(text)
 #bubble.setWordWrap(True)
 bubble.setFont(QFont("Consolas", 14))

 if sender == "user":
 # Темніший фон для користувача
 bubble.setStyleSheet("background-color: #172533; color: #ECF0F1; border-radius: 10px; padding: 5px;")
 bubble.setAlignment(Qt.AlignmentFlag.AlignLeft)
 self.last_user_text_browser = bubble
 self.last_bot_text_browser = None
 else:
 # Темніший фон для бота
 bubble.setStyleSheet("background-color: #1a2530; color: #ECF0F1; border-radius: 10px; padding: 5px;")
 bubble.setAlignment(Qt.AlignmentFlag.AlignLeft)
 self.last_bot_text_browser = bubble
 self.last_user_text_browser = None

```

```

document_height = int(bubble.document().size().height()) + 10
bubble.setFixedHeight(document_height)

bubble.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
bubble.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)

Додавання повідомлення в чергу
self.messages_layout.addWidget(bubble)

Прокручування до останнього повідомлення
self.scroll_area.verticalScrollBar().setValue(self.scroll_area.verticalScrollBar().maximum())

@pyqtSlot()
def clear_all(self):
 """Очищає всі повідомлення в messages_layout."""
 # Отримуємо список всіх віджетів в макеті
 for i in reversed(range(self.messages_layout.count())):
 widget = self.messages_layout.itemAt(i).widget() # Отримуємо віджет за індексом
 if widget is not None:
 widget.deleteLater() # Позначаємо віджет для видалення
 self.messages_layout.update() # Оновлюємо макет
 self.last_bot_text_browser = None
 self.last_user_text_browser = None

@pyqtSlot(ChatObject)
def update_display_chat_messages(self, chat_object:ChatObject):
 self.chat_object = chat_object
 for message in chat_object.messages:
 self.add_message_bubble(message.message, "user" if message.sender_user else "bot")

@pyqtSlot(ChatObject)
def update_last_message_bot(self, chat_object:ChatObject):
 self.chat_object = chat_object
 if (
 self.chat_object is not None and
 self.chat_object.last_bot_message_id is not None and
 self.last_bot_text_browser is not None
):
 message:ChatMessageModel = self.chat_object.update_last_bot_message()
 self.last_bot_text_browser.setText(message.processed_message)
 document_height = int(self.last_bot_text_browser.document().size().height()) + 10
 self.last_bot_text_browser.setFixedHeight(document_height)

@pyqtSlot(ChatObject)
def finished_generate(self, chat_object:ChatObject):
 self.chat_object = chat_object
 if (
 self.chat_object is not None and
 self.chat_object.last_bot_message_id is not None and
 self.last_bot_text_browser is not None
):
 message:ChatMessageModel = self.chat_object.update_last_bot_message()
 self.last_bot_text_browser.setText(message.processed_message)
 document_height = int(self.last_bot_text_browser.document().size().height()) + 10
 self.last_bot_text_browser.setFixedHeight(document_height)

def resizeEvent(self, event):
 for i in reversed(range(self.messages_layout.count())):
 widget = self.messages_layout.itemAt(i).widget()
 if widget is not None:
 if isinstance(widget, QTextBrowser):

```

```

 document_height = int(widget.document().size().height()) + 10
 widget.setFixedHeight(document_height)
 self.messages_layout.update()

 super().resizeEvent(event)

Створюємо і запускаємо додаток
if __name__ == "__main__":
 app = QApplication(sys.argv)
 chat_widget = ChatWidget()
 chat_widget.show()
 sys.exit(app.exec())
'''

'''py
from PyQt6.QtWidgets import (
 QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
 QLabel, QLineEdit, QPushButton, QListWidget, QListWidgetItem, QTextEdit,
 QStackedWidget, QFrame, QToolButton, QTextBrowser, QComboBox, QCheckBox,
 QScrollArea
)
from PyQt6.QtCore import QSize, Qt, pyqtSlot, pyqtSignal
from PyQt6.QtGui import QIcon
import sys
from markdown2 import markdown

from ChatWidget import ChatWidget
from ModelsWidget import ModelsWidget
from RAGWidget import RAGWidget
from AppController import AppController
from VectorDBController import VectorDBController
from DBModels import LargeModel

class ChatWindow(QMainWindow):
 signal_reselect_model = pyqtSignal(LargeModel)
 signal_deselect_model = pyqtSignal()
 update_button_send_message = pyqtSignal(AppController)

 def __init__(self):
 super().__init__()

 self.first_start = True

 self.app_controller = AppController()
 self.vector_db_controller = VectorDBController(self.app_controller.database_manager)
 self.app_controller.set_vector_db_controller(self.vector_db_controller)

 self.setWindowTitle("ChatGui")
 self.resize(1000, 800)

 # Атрибути для зберігання повідомлень
 self.messages = []

 fullscreen_button = QToolButton()
 fullscreen_button.setText("⌕")
 fullscreen_button.setCheckable(True)
 fullscreen_button.setFixedSize(30, 30)
 fullscreen_button.clicked.connect(self.toggle_fullscreen)

 main_widget = QWidget()
 main_widget.setStyleSheet("background-color: #2C3E50;")

```

```

self.setCentralWidget(main_widget)
main_layout = QHBoxLayout(main_widget)

sidebar = QFrame()
sidebar.setStyleSheet("background-color: #2C3E50;")
sidebar_layout = QVBoxLayout(sidebar)
sidebar_layout.setAlignment(Qt.AlignmentFlag.AlignTop)

def create_sidebar_button(icon_path, tooltip):
 button = QPushButton()
 button.setIcon(QIcon(icon_path))
 button.setIconSize(QSize(40, 40))
 button.setStyleSheet("background: none; border: none;")
 button.setToolTip(tooltip)
 return button

chat_button = create_sidebar_button(r"icons/chat_icon.png", "Чати")
chat_button.clicked.connect(lambda: self.pages.setCurrentIndex(0))
sidebar_layout.addWidget(chat_button)

settings_button = create_sidebar_button(r"icons/settings_icon.png", "Налаштування")
settings_button.clicked.connect(lambda: self.pages.setCurrentIndex(1))
sidebar_layout.addWidget(settings_button)

models_button = create_sidebar_button(r"icons/brain.png", "Моделі")
models_button.clicked.connect(lambda: self.pages.setCurrentIndex(2))
sidebar_layout.addWidget(models_button)

rag_button = create_sidebar_button(r"icons/files.png", "RAG")
rag_button.clicked.connect(lambda: self.pages.setCurrentIndex(3))
sidebar_layout.addWidget(rag_button)

about_button = create_sidebar_button(r"icons/about_icon.png", "Про програму")
about_button.clicked.connect(lambda: self.pages.setCurrentIndex(4))
sidebar_layout.addWidget(about_button)

main_layout.addWidget(sidebar)

self.pages = QStackedWidget()
self.pages.currentChanged.connect(self.on_page_changed)

Сторінка "Чати"
chat_page = QWidget()

chat_layout = QVBoxLayout(chat_page)

up_chat_line_layout = QHBoxLayout()

self.create_chat_button = QPushButton("✚ Створити Чат")
self.create_chat_button.setStyleSheet("""
 QPushButton {
 background-color: #ECF0F1;
 color: #2C3E50;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #BDC3C7;
 }
 """)
self.create_chat_button.clicked.connect(self.app_controler.create_new_chat)
up_chat_line_layout.addWidget(self.create_chat_button)

```

```

self.toggle_chat_button = QPushButton("▼ Показати чати")
#self.toggle_chat_button.setFixedHeight(40)
self.toggle_chat_button.setStyleSheet("""
 QPushButton {
 background-color: #ECF0F1;
 color: #2C3E50;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #BDC3C7;
 }
 """)
self.toggle_chat_button.clicked.connect(self.toggle_chat_list)
up_chat_line_layout.addWidget(self.toggle_chat_button)

self.clear_chat_button = QPushButton("🗑️ Очистити")
self.clear_chat_button.setStyleSheet("""
 QPushButton {
 background-color: #ECF0F1;
 color: #2C3E50;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #BDC3C7;
 }
 """)
self.clear_chat_button.clicked.connect(self.app_controller.clear_all_messages_active_chat)
up_chat_line_layout.addWidget(self.clear_chat_button)

combo_style = """
 QComboBox {
 border: 2px solid #BDC3C7; /* Світла обвідка */
 border-radius: 5px;
 padding: 5px;
 background-color: #ECF0F1;
 color: #2C3E50;
 font-size: 14px;
 width: 16px;
 height: 16px;
 }

 QComboBox:hover {
 border-color: #3498DB;
 background-color: #ECF0F1;
 }

 QComboBox::drop-down {
 border: none;
 background-color: #ECF0F1;
 border-radius: 5px;
 color: #2C3E50;
 }

 QComboBox::down-arrow {
 image: url('https://upload.wikimedia.org/wikipedia/commons/2/2b/Down-Arrow-Icon.png');
 width: 16px;
 height: 16px;
 }

```

```

 }

 QComboBox QAbstractItemView {
 background-color: #ECF0F1; /* Зберігаємо фоновий колір */
 color: #2C3E50; /* Текст кольору #2C3E50 */
 selection-background-color: #3498DB; /* Колір вибраного елемента */
 selection-color: white; /* Колір тексту вибраного елемента */
 border-radius: 5px;
 }

 QComboBox::item {
 padding: 10px;
 border: none;
 border-radius: 5px;
 background-color: #ECF0F1;
 color: #2C3E50;
 }

 QComboBox::item:hover {
 background-color: #BDC3C7; /* Трохи темніший фон для елемента при наведенні */
 color: #2C3E50; /* Колір тексту залишається незмінним */
 }
}
"""

self.combo_model = QComboBox()
self.combo_model.setStyleSheet(combo_style)
self.combo_model.currentIndexChanged.connect(self.on_index_changed_combo_models)
self.init_combo_model()

checkbox_style = """
 QCheckBox {
 font-size: 14px;
 color: #2C3E50;
 background-color: #ECF0F1;
 border: 2px solid #BDC3C7; /* Світла обвідка */
 border-radius: 5px;
 padding: 5px;
 height: 16px;
 }
}
"""

self.check_box_rag = QCheckBox()
self.check_box_rag.setText("Використовувати RAG")
self.check_box_rag.setStyleSheet(checkbox_style)
self.check_box_rag.stateChanged.connect(self.app_controler.slot_select_rag_checkbox)

up_chat_line_layout.addWidget(self.combo_model)
up_chat_line_layout.addWidget(self.check_box_rag)

self.chat_list = QListWidget()
self.chat_list.itemClicked.connect(self.app_controler.on_chat_list_item_clicked)
self.chat_list.setStyleSheet("""
 QListWidget {
 background-color: #34495E;
 color: #ECF0F1;
 border: none;
 padding: 5px;
 }
 QListWidgetItem {
 margin: 5px;
 }
})

```

```

""")

self.update_chat_list()

self.chat_list.setVisible(False)

self.chat_widget = ChatWidget()
self.chat_widget.app_controller = self.app_controller
self.chat_widget.message_sent.connect(self.app_controller.add_new_user_message)
self.app_controller.signal_create_bot_response.connect(self.chat_widget.create_bot_response)
self.app_controller.signal_clear_chat_widget.connect(self.chat_widget.clear_all)

self.app_controller.signal_update_display_chat_messages.connect(self.chat_widget.update_display_chat_messages)

self.app_controller.signal_text_generated.connect(self.chat_widget.update_last_message_bot)

self.app_controller.signal_finish_text_generated.connect(self.chat_widget.finished_generate)

self.chat_widget.signal_stop_generate.connect(self.app_controller.stop_bot_generate)

self.app_controller.signal_update_button_send_message.connect(self.chat_widget.update_button_send_message)

self.app_controller.signal_update_last_user_text.connect(self.chat_widget.update_last_user_text_browser)

chat_layout.addLayout(up_chat_line_layout)
chat_layout.addWidget(self.chat_list)
chat_layout.addWidget(self.chat_widget) # Додаємо область для повідомлень

self.app_controller.set_chat_widget(self.chat_widget)

self.pages.addWidget(chat_page)

Сторінка "Налаштування"
settings_page = QWidget()
settings_layout = QVBoxLayout(settings_page)
settings_label = QLabel("Налаштування тут")
settings_layout.addWidget(settings_label)
self.pages.addWidget(settings_page)

Сторінка "Моделі"
models_page = QWidget()
models_layout = QVBoxLayout(models_page)
models_label = QLabel("Моделі")
models_layout.addWidget(models_label)
self.models_widget = ModelsWidget(self.app_controller)
self.models_widget.signal_update_model.connect(self.app_controller.update_model)
self.models_widget.signal_select_model.connect(self.app_controller.slot_select_model)
self.models_widget.signal_select_model.connect(self.slot_select_model)
self.models_widget.signal_deselect_model.connect(self.app_controller.slot_deselect_model)
self.models_widget.signal_deselect_model.connect(self.slot_deselect_model)
self.signal_reselect_model.connect(self.models_widget.selectModel)
self.signal_deselect_model.connect(self.models_widget.deselect_model)

self.models_widget.signal_delete_model.connect(self.app_controller.slot_delete_model)
self.app_controller.signal_list_all_reload_models.connect(self.models_widget.load_models)

models_layout.addWidget(self.models_widget)
self.pages.addWidget(models_page)

```

```

Сторінка "RAG"
rag_page = QWidget()
rag_layout = QVBoxLayout(rag_page)
rag_label = QLabel("RAG")
rag_layout.addWidget(rag_label)

rag_widget = RAGWidget(self.app_controller, self.app_controller.vector_db_controller)

rag_widget.signal_update_embedding_model.connect(self.app_controller.update_embedding_model)
rag_widget.signal_delete_embedding_model.connect(self.app_controller.delete_embedding_model)

self.app_controller.signal_list_embedding_models_reload.connect(rag_widget.load_embedding_models)

rag_widget.signal_update_db_model.connect(self.app_controller.update_db_model)
rag_widget.signal_delete_db_model.connect(self.app_controller.remove_db_model)

self.app_controller.signal_list_db_models_reload.connect(rag_widget.load_db_list)

rag_layout.addWidget(rag_widget)
self.pages.addWidget(rag_page)

Сторінка "Про програму"
#about_page = QWidget()
#about_layout = QVBoxLayout(about_page)
#about_label = QLabel("Інформація про програму")
#about_layout.addWidget(about_label)

Сторінка "Про програму"
about_page = QWidget()
about_layout = QVBoxLayout(about_page)

Стили
about_page.setStyleSheet("""
QWidget {
 background-color: #2E2E2E;
 color: #FFFFFF;
 font-family: Arial;
}
QLabel {
 color: #FFFFFF;
 font-size: 14px;
}
QLabel#title {
 font-size: 24px;
 font-weight: bold;
 color: #4A90E2; /* Блакитний колір для назви */
 margin-bottom: 10px;
}
QLabel#version {
 font-size: 14px;
 margin-bottom: 20px;
}
QLabel#author, QLabel#year {
 font-size: 14px;
 margin-top: 10px;
}
QLabel#features {
 font-size: 18px;
 font-weight: bold;
 margin-top: 20px;
}

```

```

 margin-bottom: 10px;
 }
 QLabel#features_list {
 font-size: 14px;
 margin-bottom: 20px;
 }
 """)

about_label = QLabel("Про програму", about_page)
about_label.setObjectName("title")
about_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
about_layout.addWidget(about_label)

name_label = QLabel("Назва: ChatGui", about_page)
name_label.setObjectName("name")
name_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
about_layout.addWidget(name_label)

version_label = QLabel("Версія: 1.0.0", about_page)
version_label.setObjectName("version")
version_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
about_layout.addWidget(version_label)

year_label = QLabel("Рік створення: 2024", about_page)
year_label.setObjectName("year")
year_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
about_layout.addWidget(year_label)

description_label = QLabel(
 "ChatGui - це інноваційний інтелектуальний помічник програміста, розроблений з метою
 "
 "підвищення продуктивності та спрощення процесу розробки програмного забезпечення. "
 "В основі ChatGui лежить передова архітектура Retrieval-Augmented Generation (RAG), "
 "яка дозволяє поєднувати можливості потужних мовних моделей, таких як Llama, з "
 "великими базами знань, що включають технічну документацію, статті, форуми та "
 "приклади коду.

"
 "Завдяки RAG, ChatGui здатний не лише генерувати зв'язний та граматично правильний текст, "
 "але й знаходити релевантну інформацію у зовнішніх джерелах та використовувати її "
 "для формування точних та вичерпних відповідей на запитання користувачів. "
 "Це дозволяє програмістам швидко отримувати необхідну інформацію, не витрачаючи час "
 "на ручний пошук у численних джерелах.",
 parent=about_page
)
description_label.setWordWrap(True)
description_label.setAlignment(Qt.AlignmentFlag.AlignJustify)
about_layout.addWidget(description_label)

features_label = QLabel("Основні можливості:", about_page)
features_label.setObjectName("features")
about_layout.addWidget(features_label)

features = [
 "Інтелектуальний пошук: Швидкий та ефективний пошук необхідної інформації у великій базі знань з використанням технологій семантичного пошуку.",
 "Генерація відповідей: Формування точних, вичерпних та контекстно-залежних відповідей на запитання програмістів.",
 "Мовна модель Llama: Використання передової мовної моделі Llama для генерації зв'язного та граматично правильного тексту.",
 "Зручний чат-бот інтерфейс: Інтуїтивно зрозумілий інтерфейс у форматі чат-бота, що робить взаємодію з помічником максимально комфортною.",
 "Підтримка форматування коду: Автоматичне виділення та форматування фрагментів коду у відповідях для кращого сприйняття.",

```

"<b>Локальна робота:</b> Всі обчислення виконуються локально на комп'ютері користувача, що забезпечує конфіденційність даних та незалежність від зовнішніх сервісів.",

"<b>Можливість розширення:</b> Гнучка архітектура дозволяє легко розширювати базу знань та додавати нові функції.",

"<b>Простота використання:</b> Програма розроблена таким чином, щоб бути максимально простою та зручною у використанні, не вимагаючи від користувача спеціальних навичок.",

```
]
features_list = ""
for feature in features:
 features_list += f"• {feature}
"
features_text = QLabel(features_list, parent=about_page)
features_text.setObjectName("features_list")
features_text.setWordWrap(True)
about_layout.addWidget(features_text)

Додавання QScrollArea
scroll_area = QScrollArea()
scroll_area.setWidgetResizable(True)
scroll_widget = QWidget()
scroll_widget.setLayout(about_layout)
scroll_area.setWidget(scroll_widget)

main_layout_i = QVBoxLayout(about_page)
main_layout_i.addWidget(scroll_area)

about_page.setLayout(main_layout_i)

self.pages.addWidget(about_page)

main_layout.addWidget(self.pages, 1)

def on_page_changed(self, index):
 print(f"Selected page index: {index}")

 #chat 0
 if index == 0:
 self.combo_model.clear()
 self.init_combo_model()

 #settings 1

 #models 2

 #rag 3

 #about 4
 if self.first_start is True:
 self.first_start = False
 self.app_controler.deselect_all_embedding_database()

def showEvent(self, event):
 super().showEvent(event)
 print("showEvent")

def get_datanames(self):
 datanames = self.app_controler.get_combo_models()
 return datanames

def init_combo_model(self, datanames=None):
```

```

self.combo_model.blockSignals(True)
self.combo_model.addItem("Виберіть модель")
if datanames is None:
 datanames = self.app_controler.get_combo_models()
for i, elem in enumerate(datanames):
 self.combo_model.addItem(elem[0])
 self.combo_model.setItemData(i+1, elem[1], Qt.ItemDataRole.UserRole)
self.combo_model.blockSignals(False)

@pyqtSlot(LargeModel)
def slot_select_model(self, model:LargeModel):
 self.combo_model.clear()
 datanames = self.get_datanames()
 self.init_combo_model(datanames=datanames)
 for i, elem in enumerate(datanames):
 index = i+1
 if elem[1] == model.id:
 self.combo_model.blockSignals(True)
 self.combo_model.setCurrentIndex(index)
 self.combo_model.blockSignals(False)
 break

@pyqtSlot(LargeModel)
def slot_deselect_model(self, model:LargeModel):
 self.combo_model.clear()
 self.init_combo_model()

@pyqtSlot(int)
def on_index_changed_combo_models(self, index):
 self.combo_model.blockSignals(True)
 if index == 0:
 self.signal_deselect_model.emit()
 elif index > 0:
 item_data = int(self.combo_model.itemData(index, Qt.ItemDataRole.UserRole))
 models = self.app_controler.get_list_exist_models()

 for model in models:
 if model.id == item_data:
 self.signal_reselect_model.emit(model)
 break

 self.update_button_send_message.emit(self.app_controler)

 self.combo_model.blockSignals(False)

def on_chat_list_item_clicked(self, item):
 extra_data_id = item.data(Qt.ItemDataRole.UserRole)
 print(f"Clicked on: {item.text()} id = {extra_data_id}")

def toggle_fullscreen(self):
 if self.isFullScreen():
 self.showNormal()
 else:
 self.showFullScreen()

def toggle_chat_list(self):
 if self.chat_list.isVisible():
 self.chat_list.setVisible(False)
 self.toggle_chat_button.setText("▲ Показати чати")
 else:
 self.update_chat_list()
 self.chat_list.setVisible(True)
 self.toggle_chat_button.setText("▼ Приховати чати")

```

```

def update_chat_list(self):
 self.chat_list.clear()
 for name, item_id in self.app_controller.get_chat_list():
 item = QListWidgetItem(f"{item_id} {name}")
 item.setData(Qt.ItemDataRole.UserRole, item_id)
 self.chat_list.addItem(item)

if __name__ == "__main__":
 app = QApplication(sys.argv)

 style = """
QWidget {
 background-color: #2E2E2E; /* Основний фон */
 color: #FFFFFF; /* Основний колір тексту */
}

QLabel {
 font-weight: bold;
 font-size: 14px;
 color: #FFFFFF; /* Колір тексту заголовка */
}

QLineEdit {
 background-color: #3C3C3C; /* Фон для полів вводу */
 color: #FFFFFF; /* Колір тексту в полі вводу */
 border: 1px solid #5C5C5C; /* Обводка поля вводу */
 border-radius: 4px; /* Закруглені краї */
 padding: 5px; /* Внутрішній відступ */
}

QLineEdit:focus {
 border: 1px solid #4A90E2; /* Обводка при фокусі */
}

QPushButton {
 background-color: #4A90E2; /* Фон кнопок */
 color: #FFFFFF; /* Колір тексту кнопок */
 border: none; /* Без обводки */
 padding: 10px; /* Внутрішній відступ */
 border-radius: 5px; /* Закруглені краї */
 font-size: 14px; /* Розмір шрифту кнопок */
}

QPushButton:hover {
 background-color: #357ABD; /* Колір кнопки при наведенні */
}

QPushButton:pressed {
 background-color: #2A6E8D; /* Колір кнопки при натисканні */
}

QTableWidget {
 background-color: #3C3C3C; /* Фон таблиці */
 color: #FFFFFF; /* Колір тексту в таблиці */
 border-radius: 4px; /* Закруглені краї */
}

QHeaderView::section {
 background-color: #4A90E2; /* Фон заголовків таблиці */
 color: #FFFFFF; /* Колір тексту заголовків */
 padding: 5px; /* Внутрішній відступ */

```

```

 font-weight: bold; /* Жирний текст для заголовків */
}

QTableWidget::item {
 border: 1px solid #5C5C5C; /* Обводка для елементів таблиці */
 padding: 5px; /* Внутрішній відступ */
}

QTableWidget::item:selected {
 background-color: #4A90E2; /* Колір для вибраних елементів */
}

QScrollBar:vertical {
 border: none; /* Без обводки */
 background: #3C3C3C; /* Фон вертикального скроллбару */
 width: 12px; /* Ширина скроллбару */
 margin: 22px 0 22px 0; /* Відступи */
}

QScrollBar::handle:vertical {
 background: #4A90E2; /* Колір ручки скроллбару */
 min-height: 20px; /* Мінімальна висота ручки */
}

QScrollBar::handle:vertical:hover {
 background: #357ABD; /* Колір ручки при наведенні */
}

QScrollBar::add-line:vertical,
QScrollBar::sub-line:vertical {
 background: none; /* Без фону для кнопок прокрутки */
}

QScrollBar::up-arrow:vertical,
QScrollBar::down-arrow:vertical {
 background: none; /* Без фону для стрілок прокрутки */
}

QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }

QTabWidget::pane {
 border: 1px solid #4A90E2;
 border-radius: 6px;
 background-color: #2E2E2E;
 padding: 5px;
}

QTabBar::tab {
 background-color: #3C3C3C;
 color: #FFFFFF;
 padding: 8px 20px;
 border-top-left-radius: 4px;
 border-top-right-radius: 4px;
 margin: 2px;
 font-size: 14px;
}

QTabBar::tab:selected {
 background-color: #4A90E2;
 color: #FFFFFF;
 font-weight: bold;
 margin-bottom: 0px; /* усуває нижній відступ при виборі */
}

QTabBar::tab:hover {

```

```

 background-color: #5B9BD5;
 color: #FFFFFF;
 }
 QTabBar::tab:!selected {
 margin-top: 2px; /* відступ для неактивних вкладок */
 }
 QTabWidget::tab-bar {
 alignment: center;
 }
 QTabBar::tab:selected, QTabBar::tab:hover {
 border: none;
 }

 QComboBox {
 border: 2px solid #BDC3C7; /* Світла обвідка */
 border-radius: 5px;
 padding: 5px;
 background-color: #ECF0F1;
 color: #2C3E50;
 font-size: 14px;
 }

 QComboBox:hover {
 border-color: #3498DB;
 background-color: #ECF0F1;
 }

 QComboBox::drop-down {
 border: none;
 background-color: #ECF0F1;
 border-radius: 5px;
 color: #2C3E50;
 }

 QComboBox::down-arrow {
 image: url('https://upload.wikimedia.org/wikipedia/commons/2/2b/Down-Arrow-Icon.png');
 width: 16px;
 height: 16px;
 }

 QComboBox QAbstractItemView {
 background-color: #ECF0F1; /* Зберігаємо фоновий колір */
 color: #2C3E50; /* Текст кольору #2C3E50 */
 selection-background-color: #3498DB; /* Колір вибраного елемента */
 selection-color: white; /* Колір тексту вибраного елемента */
 border-radius: 5px;
 }

 QComboBox::item {
 padding: 10px;
 border: none;
 border-radius: 5px;
 background-color: #ECF0F1;
 color: #2C3E50;
 }

 QComboBox::item:hover {
 background-color: #BDC3C7; /* Трохи темніший фон для елемента при наведенні */
 color: #2C3E50; /* Колір тексту залишається незмінним */
 }
}
"""
app.setStyleSheet(style)

```

```

 window = ChatWindow()
 window.show()
 sys.exit(app.exec())
'''

''' .py
from PyQt6.QtCore import QObject, QMutex, QMutexLocker, pyqtSignal
import traceback

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, scoped_session
from sqlalchemy import asc
from sqlalchemy.orm import joinedload
import os
Створення базового класу для моделей
Base = declarative_base()

from DBModels import *

class DatabaseManager(QObject):
 def __init__(self, db_url='sqlite:///app.db'):
 """Ініціалізація підключення до бази даних."""
 self.engine = create_engine(db_url, pool_size=10, max_overflow=20, pool_timeout=30)
 if not os.path.exists('app.db'):
 Base.metadata.create_all(self.engine) # Створення таблиць, якщо не існують
 #self.Session = sessionmaker(bind=self.engine, autoflush=True)

 self.Session = scoped_session(sessionmaker(bind=self.engine))
 self.mutex = QMutex()

 def get_text_embeddings_for_database(self, database_id):
 with QMutexLocker(self.mutex):
 # Отримання всіх текстових векторів для конкретного EmbeddingDatabase за його ID
 with self.Session() as session:
 try:
 elements = session.query(TextEmbedding).filter_by(db_id=database_id).all()
 #for record in elements:
 # session.expunge(record)
 #for record in elements:
 # session.merge(record)
 return elements
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

 def delete_text_embeddings_for_database(self, database_id):
 """
 Видалення всіх текстових векторів для конкретного EmbeddingDatabase за його ID.
 """
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 # Виконуємо видалення записів з таблиці TextEmbedding, які відповідають заданому db_id
 session.query(TextEmbedding).filter_by(db_id=database_id).delete()

 # Підтвердження транзакції
 session.commit()

```

```

 print(f"Всі текстові вектори для database_id {database_id} успішно видалено.")
 except Exception as e:
 session.rollback() # Відміна транзакції у разі помилки
 print(f"Помилка при видаленні текстових векторів:")
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_messages_for_chat(self, chat_id):
 """Отримує всі повідомлення для заданого чату, відсортовані за часом."""
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 messages = (
 session.query(ChatMessageModel)
 .filter(ChatMessageModel.chat_id == chat_id)
 .order_by(asc(ChatMessageModel.timestamp))
 .all()
)
 #for record in messages:
 # session.expunge(record)
 #for record in messages:
 # session.merge(record)
 return messages
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_message_by_id(self, message_id):
 """Отримує повідомлення за його id."""
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 message = session.query(ChatMessageModel).get(message_id)
 #session.expunge(message)
 session.merge(message)
 return message
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def delete_messages_for_chat(self, chat_id):
 """Видаляє всі повідомлення для заданого чату."""
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 # Знаходимо повідомлення для даного чату
 messages_to_delete = session.query(ChatMessageModel).filter(ChatMessageModel.chat_id ==
chat_id)

 # Виконати видалення
 messages_to_delete.delete(synchronize_session=False)

```

```

 # Зберігаємо зміни
 session.commit()
 except Exception as e:
 session.rollback() # У разі помилки скасовуємо зміни
 print(f"Помилка при видаленні повідомлень:")
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove() # Закриваємо сесію

def add_record(self, record):
 """Додає новий запис у базу даних."""
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 session.add(record)
 session.commit()
 record_id = record.id # Отримання id новоствореного запису до закриття сесії
 #session.expunge(record)
 #session.merge(record)
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

 return record_id # Повернення id, щоб можна було використовувати його в інших частинах коду

def get_record_by_id(self, record_id, model):
 """Отримує запис за його ID.

 Args:
 record_id (int): Ідентифікатор запису.
 model (Base): Модель, з якою проводиться запит.

 Returns:
 Base: Об'єкт моделі з заданим ID або None, якщо запис не знайдено.
 """
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 record = session.get(model, record_id)
 #session.expunge(record)
 #session.merge(record)
 return record
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_all_records(self, model):
 """Отримує всі записи певної моделі.

 Args:
 model (Base): Клас моделі, що наслідує `Base`.

```

```

Returns:
 list: Список всіх записів для даної моделі.
 """
with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 records = session.query(model).all()
 #for record in records:
 # session.expunge(record)
 #for record in records:
 # session.merge(record)
 if records is None:
 return []
 return records
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def update_record(self, model, record_id, **kwargs):
 """Оновлює запис за його ID.

 Args:
 model (Base): Клас моделі, що наслідує `Base`.
 record_id (int): Ідентифікатор запису для оновлення.
 kwargs: Поля, які потрібно оновити.
 """
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 record = session.query(model).get(record_id)
 if record:
 for key, value in kwargs.items():
 setattr(record, key, value)
 session.commit()
 #session.expunge(record)
 #session.merge(record)
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def delete_record(self, model, record_id):
 """Видаляє запис за його ID.

 Args:
 model (Base): Клас моделі, що наслідує `Base`.
 record_id (int): Ідентифікатор запису для видалення.
 """
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 record = session.query(model).get(record_id)
 if record:
 session.delete(record)
 session.commit()
 except Exception as e:

```

```

 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def save_to_database(self, instance):
 """Метод для збереження змін у базу даних"""
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:

 #if not session.is_active:
 # raise RuntimeError("Сесія не активна.")
 #if not session.in_transaction():
 # raise RuntimeError("Сесія не в транзакції.")

 session.merge(instance) # Приєднуємо об'єкт до сесії, щоб оновити або додати його
 session.commit() # Фіксуємо зміни в базі даних
 #session.expunge(instance)

 #session.merge(instance)
 except Exception as e:
 session.rollback() # Відкочуємо зміни в разі помилки
 print(f"Error saving to database:")
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_embedding_database_for_id(self, some_id):
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 embedding_database =
session.query(EmbeddingDatabase).options(joinedload(EmbeddingDatabase.embedding_model)).get(some_id)
 #session.expunge(embedding_database)
 #session.merge(embedding_database)
 return embedding_database
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_all_embedding_databases(self):
 """Отримує всі записи певної моделі.

 Args:
 model (Base): Клас моделі, що наслідує `Base`.

 Returns:
 list: Список всіх записів для даної моделі.
 """
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 records =
session.query(EmbeddingDatabase).options(joinedload(EmbeddingDatabase.embedding_model)).all()

```

```

 #for record in records:
 # session.expunge(record)
 #for record in records:
 # session.merge(record)
 return records
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

def get_text_embedding_by_embedding_id(self, embedding_id: int, database_id: int):
 with QMutexLocker(self.mutex):
 with self.Session() as session:
 try:
 return (
 session.query(TextEmbedding)
 .filter(TextEmbedding.embedding_id == embedding_id)
 .filter(TextEmbedding.db_id == database_id)
 .first()
)
 except Exception as e:
 session.rollback()
 traceback.print_exc()
 finally:
 #session.expunge_all()
 #session.close()
 self.Session.remove()

...

```py
from sqlalchemy import Column, Integer, String, DateTime, Text, Boolean, ForeignKey, Float, Table
from sqlalchemy.orm import relationship
from DB import Base
from datetime import datetime, timezone
from enum import Enum

class ExampleModel(Base):
    __tablename__ = 'example_table'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    description = Column(String)

class ChatMessageModel(Base):
    __tablename__ = 'chat_message_table'

    id = Column(Integer, primary_key=True, autoincrement=True) # Унікальний ідентифікатор запису
    chat_id = Column(Integer, ForeignKey('chat_table.id'), nullable=False) # Прив'язка до чату
    sender_user = Column(Boolean, nullable=False) # True, якщо повідомлення від користувача
    sender_bot = Column(Boolean, nullable=False) # True, якщо повідомлення від бота
    message = Column(String, nullable=False) # Текст повідомлення
    processed_message = Column(String, nullable=True) # Текст обробленого повідомлення

    timestamp = Column(DateTime, default=lambda: datetime.now(timezone.utc)) # Дата та час створення
    повідомлення

```

```

model_path = Column(String, nullable=False)

# Встановлення зв'язку з чатом
chat = relationship("ChatModel", back_populates="messages")

rag_messages = relationship("RagMessageModel", back_populates="message")

def __repr__(self):
    return f"<ChatMessageModel(id={self.id}, chat_id={self.chat_id}, sender_user={self.sender_user}, sender_bot={self.sender_bot}, message='{self.message}', timestamp='{self.timestamp}')>"

class RagMessageModel(Base):
    __tablename__ = 'rag_message_table'

    id = Column(Integer, primary_key=True, autoincrement=True) # Унікальний ідентифікатор запису
    chat_message_id = Column(Integer, ForeignKey('chat_message_table.id'), nullable=False) # Прив'язка до чату
    message = relationship("ChatMessageModel", back_populates="rag_messages")
    text = Column(String, nullable=False) # Текст повідомлення

    timestamp = Column(DateTime, default=lambda: datetime.now(timezone.utc))

class ChatModel(Base):
    __tablename__ = 'chat_table' # Назва таблиці

    id = Column(Integer, primary_key=True, autoincrement=True) # Унікальний ідентифікатор чату
    title = Column(String, nullable=False) # Назва чату або опис
    created_at = Column(DateTime, default=lambda: datetime.now(timezone.utc)) # Дата і час створення чату
    archived = Column(Boolean, default=False) # Поле для позначення архівованого чату (False за замовчуванням)

    # Встановлення зв'язку з повідомленнями
    messages = relationship("ChatMessageModel", back_populates="chat")

    def __repr__(self):
        return f"<ChatModel(id={self.id}, title='{self.title}', created_at='{self.created_at}', archived={self.archived})>"

class LargeModel(Base):
    __tablename__ = 'large_models'
    id = Column(Integer, primary_key=True)
    model_path = Column(String, nullable=False, default="")
    context_size = Column(Integer, nullable=False, default=4096) # Розмір контексту
    num_threads = Column(Integer, nullable=False, default=5) # Кількість потоків (CPU)
    gpu_layers = Column(Integer, nullable=False, default=0) # Кількість шарів на GPU
    batch_size = Column(Integer, nullable=False, default=512) # Розмір батчу
    system_prompt = Column(Text, nullable=False, default="") # Системний промпт
    temperature = Column(Float, nullable=False, default=0.3) # Температура
    min_p_sampling = Column(Float, nullable=False, default=0.05) # Мінімальний P семплінг
    top_k = Column(Integer, nullable=False, default=40) # Тор-К семплінг
    top_p = Column(Float, nullable=False, default=0.95) # Тор-Р семплінг
    repetition_penalty = Column(Float, nullable=False, default=1.1) # Штраф за повторення
    created_at = Column(DateTime, nullable=False, default=lambda: datetime.now(timezone.utc))
    select_status = Column(Boolean, nullable=False, default=False)

```

```

class IndexType(str, Enum):
    FLAT = "FLAT"
    IVF_FLAT = "IVFFlat"
    HNSW = "HNSW"
    PQ = "PQ"

class MetricType(str, Enum):
    L2 = "L2"
    IP = "IP"

class EmbeddingDatabase(Base):
    __tablename__ = 'embedding_databases'

    # Основні колонки
    id = Column(Integer, primary_key=True)
    db_name = Column(String(255), nullable=False, default="BaseDB")
    db_path = Column(String(500), nullable=False, default="")
    created_at = Column(DateTime, default=lambda: datetime.now(timezone.utc))

    # Параметри для RAG
    chunk_size = Column(Integer, nullable=False, default=512) # Розмір текстових фрагментів
    overlap = Column(Integer, nullable=False, default=50) # Кількість перекриття між фрагментами

    index_type = Column(String(50), nullable=False, default=IndexType.FLAT.value)

    metric_type = Column(String(50), nullable=False, default=MetricType.L2.value)

    # Специфічні параметри для індексів
    nlist_IVF = Column(Integer, nullable=False, default=100) # для IVF індексу
    m_PQ = Column(Integer, nullable=False, default=8) # для PQ індексу
    M_HNSW = Column(Integer, nullable=False, default=32) # для HNSW індексу
    nprobe_IVF = Column(Integer, nullable=False, default=10) # для IVF індексу

    efConstruction = Column(Integer, nullable=False, default=40)
    efSearch = Column(Integer, nullable=False, default=50)

    top_k = Column(Integer, nullable=False, default=3) # Кількість найрелевантніших результатів, що повертаються при пошуку

    # Зовнішній ключ та зв'язки
    model_id = Column(Integer, ForeignKey('embedding_models.id')) # Зовнішній ключ до EmbeddingModel
    embedding_model = relationship("EmbeddingModel", back_populates="databases")
    text_embeddings = relationship("TextEmbedding", back_populates="database")

    select_status = Column(Boolean, nullable=False, default=False)

    was_created = Column(Boolean, nullable=False, default=False)

class EmbeddingModel(Base):
    __tablename__ = 'embedding_models'
    id = Column(Integer, primary_key=True)
    model_name = Column(String(255), nullable=False, default="")
    model_path = Column(String(500), nullable=False, default="")
    embedding_dim = Column(Integer, default=0)
    max_seq_length = Column(Integer, default=0)
    use_gpu = Column(Boolean, default=False)
    created_at = Column(DateTime, default=lambda: datetime.now(timezone.utc))
    databases = relationship("EmbeddingDatabase", back_populates="embedding_model") # Зв'язок один-до-багатьох

    was_loaded = Column(Boolean, nullable=False, default=False)

```

```

class TextEmbedding(Base):
    __tablename__ = 'text_embeddings'
    id = Column(Integer, primary_key=True)
    text_content = Column(Text, nullable=False, default="")
    db_id = Column(Integer, ForeignKey('embedding_databases.id'))
    created_at = Column(DateTime, default=lambda: datetime.now(timezone.utc))
    database = relationship("EmbeddingDatabase", back_populates="text_embeddings")
    embedding_id = Column(Integer, nullable=False, default=0)

...

```py
import os

#from langchain.document_loaders import (
from langchain_community.document_loaders import (
 CSVLoader,
 EverNoteLoader,
 PDFMinerLoader,
 TextLoader,
 UnstructuredEPubLoader,
 UnstructuredHTMLLoader,
 UnstructuredMarkdownLoader,
 UnstructuredODTLoader,
 UnstructuredPowerPointLoader,
 UnstructuredWordDocumentLoader,
)
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from typing import List

LOADER_MAPPING = {
 ".csv": (CSVLoader, {}),
 ".doc": (UnstructuredWordDocumentLoader, {}),
 ".docx": (UnstructuredWordDocumentLoader, {}),
 ".enex": (EverNoteLoader, {}),
 ".epub": (UnstructuredEPubLoader, {}),
 ".html": (UnstructuredHTMLLoader, {}),
 ".md": (UnstructuredMarkdownLoader, {}),
 ".odt": (UnstructuredODTLoader, {}),
 ".pdf": (PDFMinerLoader, {}),
 ".ppt": (UnstructuredPowerPointLoader, {}),
 ".pptx": (UnstructuredPowerPointLoader, {}),
 ".txt": (TextLoader, {"encoding": "utf8"}),
}

}

class DocumentProcessor:
 # Маппінг для завантаження файлів

 def __init__(self, chunk_size=250, chunk_overlap=30):
 self.chunk_size = chunk_size
 self.chunk_overlap = chunk_overlap

 def load_single_document(self, file_path: str) -> Document:
 ext = "." + file_path.rsplit(".", 1)[-1]
 assert ext in LOADER_MAPPING
 loader_class, loader_args = LOADER_MAPPING[ext]
 loader = loader_class(file_path, **loader_args)
 return loader.load()[0]

 def process_text(self, text: str) -> str:

```

```

 """Очищає текст, видаляючи зайві порожні рядки і пробіли."""
 lines = text.split("\n")
 lines = [line for line in lines if len(line.strip()) > 2]
 text = "\n".join(lines).strip()
 if len(text) < 10:
 return None
 return text

 def build_index(self, file_paths: list) -> dict:
 """Завантажує документи, розбиває їх на фрагменти і повертає індекс."""
 documents = [self.load_single_document(path) for path in file_paths]
 text_splitter = RecursiveCharacterTextSplitter(chunk_size=self.chunk_size,
 chunk_overlap=self.chunk_overlap)
 documents: List[Document] = text_splitter.split_documents(documents)

 fixed_documents: List[Document] = []
 for doc in documents:
 doc.page_content = self.process_text(doc.page_content)
 if not doc.page_content:
 continue
 fixed_documents.append(doc)

 # Повертає індекс з фрагментами
 texts = [doc.page_content for doc in fixed_documents]

 return texts

 def load_files(self, file_paths: list) -> List[str]:
 """Завантажує та обробляє файли, будує індекс для пошуку."""
 return self.build_index(file_paths)

Приклад використання класу
if __name__ == "__main__":
 processor = DocumentProcessor(chunk_size=500, chunk_overlap=50)
 file_paths = ["D:/DOWNLOAD/2006.04768v3.pdf"] # приклади файлів для обробки
 db = processor.load_files(file_paths)
 print(db)
'''

'''py
from PyQt6.QtCore import QObject, pyqtSignal, pyqtSlot
from DB import DatabaseManager
from DBModels import EmbeddingDatabase, EmbeddingModel, TextEmbedding
from typing import List
from llama_cpp import Llama
from FAISS_DB import FaissIndexManager
import numpy as np
from llama_index.legacy.text_splitter import TokenTextSplitter
import os
from DocumentProcessor import DocumentProcessor
from functions import file_exist

class data_text_embedding:
 def __init__(self, id, text, db):
 self.id = id
 self.text = text
 self.db = db

class EngineEmbeddingDatabase(QObject):
 signal_create_text_embedding = pyqtSignal(data_text_embedding)

```

```

def __init__(self, db_model:EmbeddingDatabase, skip_error_for_remove=False):
 super().__init__()
 try:
 self.db_model:EmbeddingDatabase = db_model

 self.search_engine:FaissIndexManager = None

 self.embedding_model_llm:Llama = None

 self.embedding_model:EmbeddingModel = None

 self.embedding_model:EmbeddingModel = self.db_model.embedding_model

 print(f"EngineEmbeddingDatabase self.db_model.model_id {self.db_model.model_id}")

 self.index_path = os.path.join(self.db_model.db_path, "index/faiss_hnsw.index")

 self.search_engine = FaissIndexManager(
 index_type=self.db_model.index_type,
 metric_type=self.db_model.metric_type,
 dim=self.embedding_model.embedding_dim,
 nlist=self.db_model.nlist_IVF,
 m=self.db_model.m_PQ,
 M=self.db_model.M_HNSW,
 nprobe=self.db_model.nprobe_IVF,
 efConstruction=self.db_model.efConstruction,
 efSearch=self.db_model.efSearch,
 index_path=self.index_path
)

 self.embedding_model_llm = Llama(
 model_path=self.embedding_model.model_path,
 embedding=True,
 n_ctx=self.db_model.chunk_size
)
 except Exception as ex:
 print(f"SKIP {ex}")

def remove_index(self):
 if os.path.exists(self.index_path):
 os.remove(self.index_path)

def split_text(self, text):
 token_text_splitter = TokenTextSplitter.from_defaults(
 chunk_size=self.db_model.chunk_size,
 chunk_overlap=self.db_model.overlap
)
 chunks = token_text_splitter.split_text(text)
 return chunks

def add_chunks_to_vector_db(self, chunks):
 for text in chunks:
 embedding = self.embedding_model_llm.embed(text)
 embeddings = np.array([embedding])
 list_id = self.search_engine.add_vectors(embeddings)
 self.search_engine.save_index()

 dte = data_text_embedding(list_id[0], text, self.db_model)
 self.signal_create_text_embedding.emit(dte)

def process_text(self, text_document):
 """text block procesing"""

```

```

 chunks = self.split_text(text_document)
 self.add_chunks_to_vector_db(chunks)

def process_file(self, file_path):
 """document procesing"""
 if file_exist(file_path):
 doc_processor = DocumentProcessor(
 chunk_size=self.db_model.chunk_size,
 chunk_overlap=self.db_model.overlap
)
 chunks = doc_processor.load_files([file_path])
 self.add_chunks_to_vector_db(chunks)

class SearchResult:
 def __init__(self):
 self.id = 0
 self.distance = 0
 self.text = ""
 self.db_id = 0

def search(self, text):
 """single text predict similarity"""
 results: List[self.SearchResult] = []
 if isinstance(text, str) and text != "":
 embedding = self.embedding_model_llm.embed(text)
 query_embeddings = np.array([embedding])
 distances, indices = self.search_engine.search(query_embeddings, top_k=self.db_model.top_k)
 indices = list(indices[0])
 distances = list(distances[0])
 for distance, index in zip(distances, indices):
 result = self.SearchResult()
 result.id = int(index)
 result.distance = float(distance)
 result.db_id = self.db_model.id
 results.append(result)

 pass
 return results
...

```py
import os
import numpy as np
import faiss

class FaissIndexManager:
    def __init__(
        self,
        index_type="Flat",
        metric_type="L2",
        dim=128,
        nlist=100,
        m=8,
        M=32,
        nprobe=10,
        efConstruction=40,
        efSearch=50,
        index_path="index/faiss_hnsw.index"
    ):
        self.index_type = index_type
        self.metric_type = metric_type

```

```

self.dim = dim
self.nlist = nlist      # Для IVF
self.m = m             # Для PQ
self.M = M             # Для HNSW
self.nprobe = nprobe   # Для IVF
self.index = None
self.index_path = index_path
self.efSearch = efSearch
self.efConstruction = efConstruction
if os.path.exists(self.index_path):
    self.load_index()
else:
    self._initialize_index()

def _initialize_index(self):
    """Creates an FAISS index based on the specified type and metric."""
    if self.metric_type == "L2":
        metric = faiss.METRIC_L2
    elif self.metric_type == "IP":
        metric = faiss.METRIC_INNER_PRODUCT
    else:
        raise ValueError("Unsupported metric type. Choose 'L2' or 'IP'.")

    if self.index_type == "FLAT":
        self.index = faiss.IndexFlatL2(self.dim) if metric == faiss.METRIC_L2 else faiss.IndexFlatIP(self.dim)
    elif self.index_type == "IVF":
        quantizer = faiss.IndexFlatL2(self.dim)
        # Використовуємо nlist для кількості кластерів
        self.index = faiss.IndexIVFFlat(quantizer, self.dim, self.nlist, metric)
        # nprobe для визначення, скільки кластерів будуть перевірятися під час пошуку
        self.index.nprobe = self.nprobe
    elif self.index_type == "HNSW":
        # Використовуємо M для кількості сусідів
        self.index = faiss.IndexHNSWFlat(self.dim, self.M)
        self.index.hnsw.efConstruction = self.efConstruction
        self.index.hnsw.efSearch = self.efSearch
    elif self.index_type == "PQ":
        # Використовуємо m для кількості підпросторів
        self.index = faiss.IndexPQ(self.dim, self.m, 8)
    else:
        raise ValueError("Unsupported index type. Choose 'Flat', 'IVF', 'HNSW', or 'PQ'.")

def _total_id(self, count_embeddings:int):
    return [val - 1 for val in range(self.index.ntotal, self.index.ntotal + count_embeddings)]

def reconstruct_embedding(self, id):
    # Використовуємо метод reconstruct для відновлення ембедінга за ідентифікатором
    return self.index.reconstruct(id)

def add_vectors(self, embeddings):
    if self.index is None:
        self._initialize_index()
    if not self.index.is_trained:
        self.index.train(embeddings)
    self.index.add(embeddings)
    idx = self._total_id(len(embeddings))
    print(f"IDX: {idx}")
    return idx

def search(self, query_vectors, top_k=None):
    if top_k is None:
        top_k = self.index.k
    distances, indices = self.index.search(query_vectors, top_k)

```

```

        return distances, indices

    def save_index(self):
        if not os.path.exists(os.path.dirname(self.index_path)):
            os.makedirs(os.path.dirname(self.index_path))
            faiss.write_index(self.index, self.index_path)

    def load_index(self):
        self.index = faiss.read_index(self.index_path)

    def set_search_params(self, **kwargs):
        """Set specific parameters like 'nprobe' for IVF indexes."""
        if hasattr(self.index, "nprobe"):
            # Налаштовуємо nprobe для IVF індексу
            self.index.nprobe = kwargs.get("nprobe", self.nprobe)

    def get_index_info(self):
        """Returns key information about the current index."""
        info = {
            "Index Type": self.index_type,
            "Metric Type": self.metric_type,
            "Dimension": self.dim,
            "Trained": self.index.is_trained,
            "nlist": self.nlist,
            "m": self.m,
            "M": self.M,
            "nprobe": self.nprobe
        }
        return info

...

```py
from pathlib import Path

def extract_filename(path):
 # Отримуємо ім'я файлу без шляху і без розширення
 return Path(path).stem

def file_exist(path):
 file_path = Path(path)

 if file_path.is_file():
 return True
 else:
 return False

...

```py
import markdown
from pygments import highlight
from pygments.lexers import PythonLexer
from pygments.formatters import HtmlFormatter

class HighlightProcessor:
    def __init__(self):
        pass

    def highlight_code_blocks(self, markdown_text:str):
        """Функція для підсвічування блоків коду в Markdown."""
        # Розбивка тексту на блоки за трьома зворотними апострофами

```

```

blocks = markdown_text.split('```')
highlighted_content = ""

for i in range(len(blocks)):
    block = blocks[i].strip() # Обрізаємо пробіли на початку та в кінці
    if i % 2 == 1: # Це блок коду
        # Визначаємо мову програмування
        lines = block.split('\n')
        code_language = lines[0].strip() if lines else 'python'
        code = '\n'.join(lines[1:]).strip() # Код без мови

        # Вибір відповідного лексера
        lexer = {
            'python': PythonLexer()
        }.get(code_language.lower(), PythonLexer()) # За замовчуванням - PythonLexer

        formatter = HtmlFormatter()
        highlighted_code = highlight(code, lexer, formatter)

        css_style = formatter.get_style_defs()
        final_html = f'<style> {css_style} </style> {highlighted_code}'
        highlighted_content += final_html
    else: # Звичайний текст

        highlighted_content += markdown.markdown(block)

return f'<html><body>{highlighted_content}</body></html>'```

```py
class MessageViewer :
 def __init__(self):
 pass

 ...

```py
import llama_cpp
import llama_cpp.llama_tokenizer
#from llama_cpp.llama_speculative import LlamaPromptLookupDecoding
from PyQt6.QtCore import Qt, pyqtSignal, pyqtSlot, QObject, QThread
from typing import List
import traceback

class PredictionThread(QThread):
    # Створюємо сигнал для передачі частини тексту
    text_generated = pyqtSignal(str)
    last_signal_messages = pyqtSignal(list)

    def __init__(self, model, messages:List=[], temperature=0.3, llm:llama_cpp.Llama=None):
        super().__init__()
        self.model = model
        self.messages = messages
        self.temperature = temperature
        self.llm = llm
        self.stop = False

    def run(self):

        # Запит до OpenAI для отримання відповіді
        response = self.llm.create_chat_completion_openai_v1(
            model=self.model,

```

```

        messages=self.messages,
        stream=True,
        temperature=self.temperature
    )

    text = ""
    try:
        for chunk in response:
            content = chunk.choices[0].delta.content
            if content:
                text += content
                # Кидаємо частину тексту через сигнал
                self.text_generated.emit(text)

            if self.stop is True:
                break
    except Exception as ex:
        traceback.print_exc()
    # Додаємо кінцеву відповідь до історії
    self.messages.append({"role": "assistant", "content": text})

    self.last_signal_messages.emit(self.messages)

def stop_generate(self):
    self.stop = True

class ModelController(QObject):
    thread_is_stopped = pyqtSignal()
    def __init__(
        self,
        model_path=r"D:/LM_models/bartowski/Vikhr-Gemma-2B-instruct-GGUF/Vikhr-Gemma-2B-instruct-
Q4_K_M.gguf",
        n_ctx=4096,
        n_threads=5,
        temperature = 0.3,
        add_system=False
    ):
        super().__init__()

        self.model_path=model_path
        self.n_ctx = n_ctx
        self.n_threads = n_threads
        if model_path != "":
            self.llm = llama_cpp.Llama(
                model_path=self.model_path,
                n_ctx=self.n_ctx,
                n_threads=self.n_threads,
                chat_format=None,#"llama-3"
                #draft_model=LlamaPromptLookupDecoding(),
            )
        else:
            self.llm = None
            raise ValueError("Не вдалось завантажити модель. Шлях до моделі пустий або моделі вже не існує.")

        self.temperature = temperature

        self.model = "gpt-3.5-turbo"
        self.system_message = "Відповідай українською мовою. Твоя відповідь максимально експертна в тій області про яку іде мова. Будь учасним в розмові. Імітуй стиль спілкування людей."
        self.add_system=add_system
        self.messages = []

```

```

    if add_system is True:
        self.messages.append({"role": "system", "content": self.system_message})

def predict(self, message):

    #for user_message, assistant_message in history:
    #    messages.append({"role": "user", "content": user_message})
    #    messages.append({"role": "assistant", "content": assistant_message})

    self.messages.append({"role": "user", "content": message})

    response = self.llm.create_chat_completion_openai_v1(
        model=self.model,
        messages=self.messages,
        stream=True,
        temperature=self.temperature
    )

    text = ""
    for chunk in response:
        content = chunk.choices[0].delta.content
        if content:
            text += content
            yield text
    assistant_message = text
    self.messages.append({"role": "assistant", "content": assistant_message})

def clear(self):
    self.messages = []
    if self.add_system is True:
        self.messages.append({"role": "system", "content": self.system_message})

def create_thread_generate(self, message, slot_text_generated):
    self.messages.append({"role": "user", "content": message})

    # Створюємо і запускаємо потік
    self.prediction_thread = PredictionThread(self.model, self.messages, self.temperature, self.llm)
    self.prediction_thread.text_generated.connect(slot_text_generated)
    self.prediction_thread.last_signal_messages.connect(self.last_signal_messages)
    self.prediction_thread.start()

    @pyqtSlot()
    def stop_thread_generate(self):
        self.prediction_thread.stop_generate()

    @pyqtSlot(list)
    def last_signal_messages(self, messages):
        self.messages = messages
        self.thread_is_stopped.emit()

...

...py
import sys
from PyQt6.QtWidgets import (
    QWidget, QHBoxLayout, QVBoxLayout,
    QPushButton, QListWidget, QLabel,
    QFormLayout, QLineEdit, QListWidgetItem,
    QFileDialog, QMessageBox
)
from PyQt6.QtCore import Qt, pyqtSlot

```

```

from DBModels import LargeModel
from PyQt6.QtCore import QObject, pyqtSignal
from DB import DatabaseManager
from DBModels import LargeModel
from AppController import AppController

def creat_large_model():
    model = LargeModel()
    model.select_status = False
    model.context_size = 4096
    model.batch_size=512
    model.min_p_sampling=0.05
    model.top_k = 40
    model.top_p = 0.95
    model.repetition_penalty=1.1
    model.temperature=0.3
    model.num_threads=5

    return model

class ModelWidget(QWidget):
    signal_update_model = pyqtSignal(LargeModel)
    signal_select_model = pyqtSignal(LargeModel)
    signal_delete_model = pyqtSignal(LargeModel)

    def __init__(self, model:LargeModel, parent=None, deletion_allowed =True):
        super().__init__(parent)
        self.model = model
        self.deletion_allowed=deletion_allowed
        self.initUI()

    def update_model_path(self, text):
        try:
            self.model.model_path = str(text)
            self.name_edit.setText(text)
        except Exception as ex:
            print(ex)
        finally:
            self.signal_update_model.emit(self.model)

    def update_context_size(self, text):
        try:
            self.model.context_size = int(text)
        except Exception as ex:
            print(ex)
        finally:
            self.signal_update_model.emit(self.model)

    def update_temperature(self, text):
        try:
            self.model.temperature = float(text)
        except Exception as ex:
            print(ex)
        finally:
            self.signal_update_model.emit(self.model)

    def update_top_k(self, text):
        try:
            self.model.top_k = int(text)
        except Exception as ex:
            print(ex)
        finally:
            self.signal_update_model.emit(self.model)

```

```

def update_top_p(self, text):
    try:
        self.model.top_p = float(text)
    except Exception as ex:
        print(ex)
    finally:
        self.signal_update_model.emit(self.model)

def open_file_dialog(self):
    file_path, _ = QFileDialog.getOpenFileName(self, "Виберіть файл", "", "All Files (*);;gguf Files (*.gguf)")
    if file_path:
        self.update_model_path(file_path)

def initUI(self):
    layout = QFormLayout()
    layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

    self.name_edit = QLineEdit(self.model.model_path)
    self.name_edit.textChanged.connect(self.update_model_path)
    layout.addRow(QLabel("Model:"), self.name_edit)

    self.browse_button = QPushButton("...", self)
    self.browse_button.clicked.connect(self.open_file_dialog)
    layout.addRow(QLabel("Select Model Path:"), self.browse_button)

    self.context_size_edit = QLineEdit(str(self.model.context_size))
    self.context_size_edit.textChanged.connect(self.update_context_size)
    layout.addRow(QLabel("Context Size:"), self.context_size_edit)

    self.temperature_edit = QLineEdit(str(self.model.temperature))
    self.temperature_edit.textChanged.connect(self.update_temperature)
    layout.addRow(QLabel("Temperature:"), self.temperature_edit)

    self.top_k_edit = QLineEdit(str(self.model.top_k))
    self.top_k_edit.textChanged.connect(self.update_top_k)
    layout.addRow(QLabel("Top K:"), self.top_k_edit)

    self.top_p_edit = QLineEdit(str(self.model.top_p))
    self.top_p_edit.textChanged.connect(self.update_top_p)
    layout.addRow(QLabel("Top P:"), self.top_p_edit)

    if self.model.select_status is False:
        self.select_button = QPushButton("Load model")
    else:
        self.select_button = QPushButton("Unload model")

    self.delete_button = QPushButton("Delete Model")
    if self.deletion_allowed is False:
        self.delete_button.setVisible(False)

    self.delete_button.setStyleSheet("""
QPushButton {
    background-color: #ff4d4d; /* Червоний колір фону */
    color: white; /* Білий колір тексту */
    border: none;
    border-radius: 8px; /* Згладжені краї для сучасного вигляду */
    padding: 8px 16px; /* Відступи для кращого розміщення */
    font-size: 14px; /* Розмір шрифту */
}
QPushButton:hover {

```

```

        background-color: #ff3333;    /* Трохи темніший відтінок при наведенні */
    }
    QPushButton:pressed {
        background-color: #cc0000;    /* Темно-червоний при натисканні */
    }
    """

self.select_button.clicked.connect(self.onSelect)
self.delete_button.clicked.connect(self.onDelete)
layout.addRow(self.select_button)
layout.addRow(self.delete_button)

# Dark theme style with better spacing and framing
self.setStyleSheet("""
    QWidget {
        background-color: #3C3C3C;
        color: #FFFFFF;
        border: 2px solid #4A90E2;
        border-radius: 8px;
        padding: 10px;
        margin: 10px; /* Add margin for spacing */
    }
    QLineEdit {
        background-color: #4C4C4C;
        border: 1px solid #5C5C5C;
        padding: 5px;
        color: #FFFFFF;
        border-radius: 3px;
        margin: 5px 0; /* Add margin between input fields */
    }
    QLineEdit:focus {
        border: 1px solid #4A90E2;
    }
    QLabel {
        font-weight: bold;
        color: #FFFFFF;
    }
    QPushButton {
        background-color: #4A90E2;
        color: #FFFFFF;
        border: none;
        padding: 5px;
        border-radius: 5px;
    }
    QPushButton:hover {
        background-color: #357ABD;
    }
    """

self.setLayout(layout)

def onSelect(self):
    self.model.select_status = not self.model.select_status
    self.signal_select_model.emit(self.model)

def onDelete(self):
    self.signal_delete_model.emit(self.model)

class CustomListWidgetItem(QListWidgetItem):

    def __init__(self, model_widget):
        super().__init__()
        self.setSizeHint(model_widget.sizeHint())

```

```

self.setData(Qt.ItemDataRole.UserRole, model_widget.model) # Store model data for retrieval

class ModelsWidget(QWidget):
    signal_select_model = pyqtSignal(LargeModel)
    signal_update_model = pyqtSignal(LargeModel)
    signal_deselect_model = pyqtSignal(LargeModel)
    signal_delete_model = pyqtSignal(LargeModel)
    signal_reload_models = pyqtSignal()

    def __init__(self, app_controller:AppController):
        super().__init__()
        self.app_controller = app_controller
        self.models = []
        self.selected_model = None # To track the currently selected model
        self.initUI()

        self.load_models()

        self.signal_reload_models.connect(self.load_models)

    def initUI(self):
        main_layout = QVBoxLayout() # Changed to QVBoxLayout for vertical arrangement

        self.add_button = QPushButton("Add Model")
        self.add_button.setStyleSheet("""
            QPushButton {
                background-color: #4A90E2;
                color: #FFFFFF;
                border: none;
                padding: 10px;
                border-radius: 5px;
            }
            QPushButton:hover {
                background-color: #357ABD;
            }
            """)
        self.add_button.clicked.connect(self.addModel)
        main_layout.addWidget(self.add_button)

        # Create horizontal layout for the lists
        lists_layout = QHBoxLayout()

        self.all_list = QListWidget()
        self.all_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")

        self.select_list = QListWidget()
        self.select_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")

        lists_layout.addWidget(self.select_list)
        lists_layout.addWidget(self.all_list)

        main_layout.addLayout(lists_layout)

        # Dark theme style for the main window
        self.setStyleSheet("background-color: #2E2E2E; color: #FFFFFF;")
        self.setLayout(main_layout)

        self.select_list.setVisible(False)

    def exist_selected(self, model:LargeModel):

```

```

for index in range(self.select_list.count()):
    item = self.select_list.item(index)
    widget: ModelWidget = self.select_list.itemWidget(item)
    if widget.model.id == model.id:
        return True
return False

@pyqtSlot()
def load_models(self):
    self.all_list.clear()
    self.models = self.app_controler.get_all_models()
    for model in self.models:
        if self.exist_selected(model) is False:
            model_widget = ModelWidget(model)
            model_widget.signal_update_model.connect(self.slot_update_model)
            model_widget.signal_select_model.connect(self.selectModel)
            model_widget.signal_delete_model.connect(self.delete_model)
            item = CustomListWidgetItem(model_widget)
            self.all_list.addItem(item)
            self.all_list.setItemWidget(item, model_widget)

@pyqtSlot(LargeModel)
def slot_update_model(self, model):
    self.signal_update_model.emit(model)

@pyqtSlot()
def addModel(self):
    model = creat_large_model()

    model_id = self.app_controler.database_manager.add_record(model)
    model = self.app_controler.database_manager.get_record_by_id(model_id, LargeModel)

    model_widget = ModelWidget(model)
    model_widget.signal_update_model.connect(self.slot_update_model)
    model_widget.signal_select_model.connect(self.selectModel)
    model_widget.signal_delete_model.connect(self.delete_model)
    item = CustomListWidgetItem(model_widget)
    self.all_list.addItem(item)
    self.all_list.setItemWidget(item, model_widget)

    #self.signal_reload_models.emit()

@pyqtSlot(LargeModel)
def selectModel(self, model):
    if self.selected_model is not model and self.selected_model is not None:
        self.moveModel(self.selected_model, self.select_list, self.all_list)
        self.signal_deselect_model.emit(self.selected_model)
        self.selected_model = None
        self.select_list.setVisible(False)

    if self.selected_model is model:
        self.moveModel(self.selected_model, self.select_list, self.all_list)
        self.signal_deselect_model.emit(self.selected_model)
        self.selected_model = None
        self.select_list.setVisible(False)
    else:
        self.selected_model = model
        self.signal_select_model.emit(self.selected_model)
        self.moveModel(model, self.all_list, self.select_list)
        self.select_list.setVisible(True)

@pyqtSlot()

```

```

def deselect_model(self):
    if self.selected_model is not None:
        self.moveModel(self.selected_model, self.select_list, self.all_list)
        self.signal_deselect_model.emit(self.selected_model)
        self.selected_model = None
        self.select_list.setVisible(False)

def moveModel(self, model:LargeModel, source_list:QListWidget, target_list:QListWidget):
    # Find the model in the source list
    for index in range(source_list.count()):
        item = source_list.item(index)
        widget: ModelWidget = source_list.itemWidget(item)
        widget_model = widget.model
        if widget_model.id == model.id:
            # Move the item to the target list
            model_widget = ModelWidget(model)
            model_widget.signal_update_model.connect(self.slot_update_model)
            model_widget.signal_select_model.connect(self.selectModel)
            model_widget.signal_delete_model.connect(self.delete_model)
            item_widget = CustomListWidgetItem(model_widget)
            target_list.addItem(item_widget)
            target_list.setItemWidget(item_widget, model_widget)

            # Add border style to item widget
            #widget.setStyleSheet("border: 1px solid #5C5C5C; margin: 5px; padding: 5px;")

            source_list.takeItem(index) # Remove from the source list
            break

    @pyqtSlot(LargeModel)
    def delete_model(self, model:LargeModel):
        self.signal_delete_model.emit(model)
    ...

```py
import sys
from PyQt6.QtWidgets import (
 QApplication, QWidget, QHBoxLayout, QVBoxLayout,
 QPushButton, QListWidget, QLabel, QFormLayout,
 QLineEdit, QListWidgetItem, QStackedWidget,
 QTableWidgetItem, QTableWidgetItem, QSizePolicy,
 QGridLayout, QTabWidget, QTextEdit, QFileDialog,
 QHeaderView, QComboBox, QDialog, QMessageBox,
 QScrollArea, QFrame, QAbstractItemView
)
from PyQt6.QtCore import Qt, pyqtSignal, pyqtSlot
import random
from typing import List
from DB import DatabaseManager
from DBModels import EmbeddingDatabase, EmbeddingModel, IndexType, MetricType
from VectorDBController import VectorDBController
from AppController import AppController
from functions import extract_filename
from DocumentProcessor import LOADER_MAPPING
import time
import traceback

class RAGModelWidget(QWidget):
 signal_update_model = pyqtSignal(EmbeddingModel)
 signal_delete_model = pyqtSignal(EmbeddingModel)

 def __init__(self, model:EmbeddingModel, parent=None):
 super().__init__(parent)

```

```

self.model = model
self.initUI()

def initUI(self):
 layout = QFormLayout()
 layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

 self.model_name_edit = QLineEdit(str(self.model.model_name))
 self.model_name_edit.textChanged.connect(self.update_model_name)
 layout.addRow(QLabel("model_name:"), self.model_name_edit)

 self.model_path_edit = QLineEdit(str(self.model.model_path))
 self.model_path_edit.textChanged.connect(self.update_model_path)
 layout.addRow(QLabel("model_path:"), self.model_path_edit)

 self.select_model_button = QPushButton("Select Model")
 self.select_model_button.clicked.connect(self.open_file_dialog)
 layout.addRow(self.select_model_button)

 self.embedding_dim_edit = QLineEdit(str(self.model.embedding_dim))
 self.embedding_dim_edit.textChanged.connect(self.update_embedding_dim)
 layout.addRow(QLabel("embedding_dim:"), self.embedding_dim_edit)

 self.max_seq_length_edit = QLineEdit(str(self.model.max_seq_length))
 self.max_seq_length_edit.textChanged.connect(self.update_max_seq_length)
 layout.addRow(QLabel("max_seq_length:"), self.max_seq_length_edit)

 self.remove_model_button = QPushButton("Remove Embeddings Model")

 self.remove_model_button.clicked.connect(self.remove_model)

 layout.addRow(self.remove_model_button)

 self.setStyleSheet("""
 QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 2px solid #4A90E2;
 border-radius: 8px;
 padding: 10px;
 margin: 10px;
 }
 QLineEdit {
 background-color: #4C4C4C;
 border: 1px solid #5C5C5C;
 padding: 5px;
 color: #FFFFFF;
 border-radius: 3px;
 margin: 5px 0;
 }
 QLineEdit:focus {
 border: 1px solid #4A90E2;
 }
 QLabel {
 font-weight: bold;
 color: #FFFFFF;
 }
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 5px;
 border-radius: 5px;

```

```

 }
 QPushButton: hover {
 background-color: #357ABD;
 }
 """
)

self.setLayout(layout)

def update_embedding_dim(self, text):
 try:
 self.model.embedding_dim = str(text)
 self.embedding_dim_edit.setText(text)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_model.emit(self.model)

def update_max_seq_length(self, text):
 try:
 self.model.max_seq_length = str(text)
 self.max_seq_length_edit.setText(text)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_model.emit(self.model)

def update_model_name(self, text):
 try:
 self.model.model_name = str(text)
 self.model_name_edit.setText(text)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_model.emit(self.model)

def update_model_path(self, file_path):
 try:
 self.model.model_path = str(file_path)
 self.model_path_edit.setText(file_path)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_model.emit(self.model)

def open_file_dialog(self):
 file_path, _ = QFileDialog.getOpenFileName(self, "Виберіть файл", "", "All Files (*);;gguf Files (*.gguf)")
 if file_path:
 if self.model_name_edit.text() == "":
 filename = extract_filename(file_path)
 self.model_name_edit.setText(filename)
 self.model.model_name = filename
 self.update_model_path(file_path)

def remove_model(self):
 self.signal_delete_model.emit(self.model)

def update_view_data(self):
 self.embedding_dim_edit.setText(str(self.model.embedding_dim))
 self.max_seq_length_edit.setText(str(self.model.max_seq_length))

class RAGDBWidget(QWidget):
 signal_select_db = pyqtSignal(EmbeddingDatabase)

```

```

signal_deselect_db = pyqtSignal(EmbeddingDatabase)
signal_delete_db = pyqtSignal(EmbeddingDatabase)
signal_select_embedding_model = pyqtSignal(EmbeddingDatabase)
signal_update_db = pyqtSignal(EmbeddingDatabase)

def __init__(self, model:EmbeddingDatabase, parent=None):
 super().__init__(parent)
 self.model = model
 self.initUI()

def please_set_number(self):
 QMessageBox.warning(
 self,
 "Повідомлення",
 "Будь ласка, введіть число."
)

def update_db_name(self, text):
 try:
 self.model.db_name = str(text)
 self.db_name_edit.setText(text)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_db.emit(self.model)

def update_db_path(self, text):
 try:
 self.model.db_path = str(text)
 self.db_path_edit.setText(text)
 except Exception as ex:
 print(ex)
 finally:
 self.signal_update_db.emit(self.model)

def update_chunk_size(self, text):
 try:
 self.model.chunk_size = int(text)
 self.chunk_size_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_overlap(self, text):
 try:
 self.model.overlap = int(text)
 self.overlap_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_nlist_IVF(self, text):
 try:
 self.model.nlist_IVF = int(text)
 self.nlist_IVF_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:

```

```

 self.signal_update_db.emit(self.model)

def update_m_PQ(self, text):
 try:
 self.model.m_PQ = int(text)
 self.m_PQ_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_M_HNSW(self, text):
 try:
 self.model.M_HNSW = int(text)
 self.M_HNSW_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_nprobe_IVF(self, text):
 try:
 self.model.nprobe_IVF = int(text)
 self.nprobe_IVF_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_top_k(self, text):
 try:
 self.model.top_k = int(text)
 self.top_k_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_efConstruction(self, text):
 try:
 self.model.efConstruction = int(text)
 self.efConstruction_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def update_efSearch(self, text):
 try:
 self.model.efSearch = int(text)
 self.efSearch_edit.setText(text)
 except Exception as ex:
 print(ex)
 self.please_set_number()
 finally:
 self.signal_update_db.emit(self.model)

def initUI(self):

```

```

self.setStyleSheet("""
QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 2px solid #4A90E2;
 border-radius: 8px;
 padding: 10px;
 margin: 10px;
}
QLineEdit {
 background-color: #4C4C4C;
 border: 1px solid #5C5C5C;
 padding: 5px;
 color: #FFFFFF;
 border-radius: 3px;
 /*margin: 5px 0;*/
 min-height: 20px; max-height: 20px;
}
QLineEdit:focus {
 border: 1px solid #4A90E2;
 min-height: 20px; max-height: 20px;
}
QLabel {
 font-weight: bold;
 color: #FFFFFF;
 min-height: 20px; max-height: 20px;
}
QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 5px;
 border-radius: 5px;
 min-height: 20px; max-height: 20px;
}
QPushButton:hover {
 background-color: #357ABD;
 min-height: 20px; max-height: 20px;
}
QComboBox { min-height: 20px; max-height: 20px; }
""")

self.main_layout = QFormLayout()

self.main_layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

self.db_name_edit = QLineEdit(self.model.db_name)
self.db_name_edit.textChanged.connect(self.update_db_name)
self.db_name_label = QLabel("db_name:")
self.main_layout.addRow(self.add_to_label(self.db_name_label), self.db_name_edit)

self.db_path_edit = QLineEdit(str(self.model.db_path))
self.db_path_edit.textChanged.connect(self.update_db_path)
self.main_layout.addRow(self.add_to_label(QLabel("db_path:")), self.db_path_edit)

self.select_db_path_button = QPushButton("Select DB path Model")
self.select_db_path_button.clicked.connect(self.slot_select_db_path)
self.main_layout.addRow(self.select_db_path_button)

self.chunk_size_edit = QLineEdit(str(self.model.chunk_size))
self.chunk_size_edit.textChanged.connect(self.update_chunk_size)
self.main_layout.addRow(self.add_to_label(QLabel("chunk_size:")), self.chunk_size_edit)

```

```

self.overlap_edit = QLineEdit(str(self.model.overlap))
self.overlap_edit.textChanged.connect(self.update_overlap)
self.main_layout.addRow(self.add_to_label(QLabel("overlap:")), self.overlap_edit)

self.index_type_combo = QComboBox()
self.index_type_combo.currentIndexChanged.connect(self.on_index_type_combo)
if self.model.index_type is not None and self.model.index_type != "":
 self.add_index_type_combo(self.model.index_type)
else:
 self.add_index_type_combo()
self.main_layout.addRow(self.add_to_label(QLabel("index_type:")), self.index_type_combo)

self.metric_type_combo = QComboBox()
self.metric_type_combo.currentIndexChanged.connect(self.on_metric_type_combo)
#self.add_metric_type_combo()

if self.model.metric_type is not None and self.model.metric_type != "":
 self.add_metric_type_combo(self.model.metric_type)
else:
 self.add_metric_type_combo()

self.main_layout.addRow(self.add_to_label(QLabel("metric_type:")), self.metric_type_combo)

self.nlist_IVF_edit = QLineEdit(str(self.model.nlist_IVF))
self.nlist_IVF_edit.textChanged.connect(self.update_nlist_IVF)
self.main_layout.addRow(self.add_to_label(QLabel("nlist_IVF:")), self.nlist_IVF_edit)

self.m_PQ_edit = QLineEdit(str(self.model.m_PQ))
self.m_PQ_edit.textChanged.connect(self.update_m_PQ)
self.main_layout.addRow(self.add_to_label(QLabel("m_PQ:")), self.m_PQ_edit)

self.M_HNSW_edit = QLineEdit(str(self.model.M_HNSW))
self.M_HNSW_edit.textChanged.connect(self.update_M_HNSW)
self.main_layout.addRow(self.add_to_label(QLabel("M_HNSW:")), self.M_HNSW_edit)

self.nprobe_IVF_edit = QLineEdit(str(self.model.nprobe_IVF))
self.nprobe_IVF_edit.textChanged.connect(self.update_M_HNSW)
self.main_layout.addRow(self.add_to_label(QLabel("nprobe_IVF:")), self.nprobe_IVF_edit)

self.efConstruction_edit = QLineEdit(str(self.model.efConstruction))
self.efConstruction_edit.textChanged.connect(self.update_efConstruction)
self.main_layout.addRow(self.add_to_label(QLabel("efConstruction:")), self.efConstruction_edit)

self.efSearch_edit = QLineEdit(str(self.model.efSearch))
self.efSearch_edit.textChanged.connect(self.update_efSearch)
self.main_layout.addRow(self.add_to_label(QLabel("efSearch:")), self.efSearch_edit)

self.top_k_edit = QLineEdit(str(self.model.top_k))
self.top_k_edit.textChanged.connect(self.update_top_k)
self.main_layout.addRow(self.add_to_label(QLabel("top_k:")), self.top_k_edit)

name_embedding_model = ""
self.selected_embedding_model_edit = QLineEdit(str(name_embedding_model))
self.main_layout.addRow(self.add_to_label(QLabel("selected embedding model:")),
self.selected_embedding_model_edit)

```

```

self.select_model_button = QPushButton("Select Embeddings Model")
self.select_model_button.clicked.connect(self.slot_select_embedding_model)
self.main_layout.addRow(self.select_model_button)

self.remove_db_model_button = QPushButton("Remove DB Model")
self.remove_db_model_button.clicked.connect(self.slot_remove_db_model)
self.main_layout.addRow(self.remove_db_model_button)

if self.model.select_status is False:
 self.select_button = QPushButton("Select")
else:
 self.select_button = QPushButton("Deselect")
self.update_button_style(self.select_button)
self.select_button.clicked.connect(self.slot_select_deselect_db_model)
self.main_layout.addRow(self.select_button)

self.main_layout.setFieldGrowthPolicy(QFormLayout.FieldGrowthPolicy.AllNonFixedFieldsGrow)

self.setLayout(self.main_layout)

self.update_view_data()

self.test_index_type()

def on_metric_type_combo(self, index):
 self.metric_type_combo.blockSignals(True)
 if index == 0:
 pass
 elif index > 0:
 item_data = int(self.metric_type_combo.itemData(index, Qt.ItemDataRole.UserRole))
 for i, metric_type in enumerate(MetricType):
 if item_data == i:
 self.model.metric_type = metric_type.value
 break

 self.signal_update_db.emit(self.model)
 self.metric_type_combo.blockSignals(False)

def add_metric_type_combo(self, type_on_db:str=None):
 self.metric_type_combo.clear()
 self.metric_type_combo.blockSignals(True)
 self.metric_type_combo.addItem("Виберіть індекс бази.")
 #self.metric_type_combo.addItems([index.value for index in IndexType])
 selected_index = None
 for i, index in enumerate(MetricType):
 self.metric_type_combo.addItem(index.value)
 self.metric_type_combo.setItemData(i+1, i, Qt.ItemDataRole.UserRole)
 if type_on_db is not None and type_on_db == index.value:
 selected_index = i + 1
 if type_on_db is not None:
 self.metric_type_combo.setCurrentIndex(selected_index)
 self.metric_type_combo.blockSignals(False)

def hide_all_params(self):
 self.main_layout.setRowVisible(self.metric_type_combo, False)
 self.main_layout.setRowVisible(self.nlist_IVF_edit, False)
 self.main_layout.setRowVisible(self.m_PQ_edit, False)
 self.main_layout.setRowVisible(self.M_HNSW_edit, False)
 self.main_layout.setRowVisible(self.nprobe_IVF_edit, False)
 self.main_layout.setRowVisible(self.efConstruction_edit, False)
 self.main_layout.setRowVisible(self.efSearch_edit, False)

def disable_all(self):

```

```

self.index_type_combo.setDisabled(True)
self.metric_type_combo.setDisabled(True)
self.nlist_IVF_edit.setDisabled(True)
self.m_PQ_edit.setDisabled(True)
self.M_HNSW_edit.setDisabled(True)
self.nprobe_IVF_edit.setDisabled(True)
self.efConstruction_edit.setDisabled(True)
self.efSearch_edit.setDisabled(True)

def add_to_label(self, elem:QLabel):
 #elem.setSizePolicy(QSizePolicy.Policy.MinimumExpanding, QSizePolicy.Policy.Fixed)
 return elem

def show_row(self, widget):
 self.main_layout.setRowVisible(widget, True)
 #widget.updateGeometry() # Оновлюємо геометрію, щоб макет перерахував висоту

def hide_row(self, widget):
 self.main_layout.setRowVisible(widget, False)
 #widget.updateGeometry()

def get_current_index_type_combo(self):
 index = self.index_type_combo.currentIndex()
 item_data = int(self.index_type_combo.itemData(index, Qt.ItemDataRole.UserRole))
 current_index = None
 for i, index_type in enumerate(IndexType):
 if item_data == i:
 self.model.index_type = index_type.value
 current_index = index_type
 break
 return current_index

def test_index_type(self):
 index = self.index_type_combo.currentIndex()
 if index > 0:
 item_data = int(self.index_type_combo.itemData(index, Qt.ItemDataRole.UserRole))
 current_index = None
 for i, index_type in enumerate(IndexType):
 if item_data == i:
 self.model.index_type = index_type.value
 current_index = index_type
 break

 self.hide_all_params()

 if current_index == IndexType.FLAT:
 self.show_row(self.metric_type_combo)
 elif current_index == IndexType.IVF_FLAT:
 self.show_row(self.nlist_IVF_edit)
 self.show_row(self.nprobe_IVF_edit)
 elif current_index == IndexType.HNSW:
 self.show_row(self.M_HNSW_edit)
 self.show_row(self.efConstruction_edit)
 self.show_row(self.efSearch_edit)
 elif current_index == IndexType.PQ:
 self.show_row(self.m_PQ_edit)

 self.updateGeometry()

 if self.model.was_created is True:
 self.disable_all()
 else:

```

```

 self.signal_update_db.emit(self.model)
 else:
 self.hide_all_params()

def on_index_type_combo(self, index):
 self.index_type_combo.blockSignals(True)
 if index == 0:
 pass
 elif index > 0:
 self.test_index_type()

 self.index_type_combo.blockSignals(False)

def add_index_type_combo(self, type_on_db:str=None):
 self.index_type_combo.clear()
 self.index_type_combo.blockSignals(True)
 self.index_type_combo.addItem("Виберіть індекс бази.")
 #self.index_type_combo.addItems([index.value for index in IndexType])
 selected_index = None
 for i, index in enumerate(IndexType):
 self.index_type_combo.addItem(index.value)
 self.index_type_combo.setItemData(i+1, i, Qt.ItemDataRole.UserRole)
 if type_on_db is not None and type_on_db == index.value:
 selected_index = i + 1
 if type_on_db is not None:
 self.index_type_combo.setCurrentIndex(selected_index)
 self.index_type_combo.blockSignals(False)

def update_button_style(self, button:QPushButton):
 """Оновлює стиль кнопки в залежності від стану"""
 if self.model.select_status is True:
 button.setStyleSheet("""
 QPushButton {
 background-color: #E94E77; /* Червоний для основного стану */
 color: #FFFFFF; /* Білий текст */
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #D43F5C; /* Темніший червоний при наведенні */
 }
 """)
 else:
 button.setStyleSheet("""
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """)

def test_field(self, field):
 test_state = False
 try:
 if field != "":
 int(field)
 test_state = True

```

```

except ValueError:
 test_state = False
return test_state

def test_not_empty_params(self) -> bool:
 test_state = True

 fields = [
 self.db_name_edit.text(),
 self.db_path_edit.text()
]
 if all(field.strip() for field in fields):
 test_state = True
 else:
 test_state = False

 if test_state is True:
 fields = [
 self.chunk_size_edit.text(),
 self.overlap_edit.text(),
 self.top_k_edit.text()
]

 if all(field.strip() for field in fields):
 for field in fields:
 try:
 int(field)
 test_state = True
 except ValueError:
 test_state = False
 break

 if test_state is True:
 current_index = self.get_current_index_type_combo()
 test_arr = []
 if current_index == IndexType.FLAT:

 test_arr.append(self.test_field("if self.metric_type_combo.currentIndex() == 0 else 1))

 elif current_index == IndexType.IVF_FLAT:

 test_arr.append(self.test_field(self.nlist_IVF_edit.text()))
 test_arr.append(self.test_field(self.nprobe_IVF_edit.text()))

 elif current_index == IndexType.HNSW:

 test_arr.append(self.test_field(self.M_HNSW_edit.text()))
 test_arr.append(self.test_field(self.efConstruction_edit.text()))
 test_arr.append(self.test_field(self.efSearch_edit.text()))

 elif current_index == IndexType.PQ:

 test_arr.append(self.test_field(self.m_PQ_edit.text()))

 test_state = all(test_arr)

 if test_state is False:
 QMessageBox.warning(
 self,
 "Заповніть всі поля",
 "Будь ласка, заповніть всі обов'язкові поля перед продовженням."
)
 return test_state

```

```

@pyqtSlot()
def slot_select_deselect_db_model(self):
 if self.test_not_empty_params() is True:
 self.model.select_status = not self.model.select_status

 if self.model.select_status is False:
 self.select_button.setText("Select")
 else:
 self.select_button.setText("Deselect")

 self.signal_update_db.emit(self.model)
 self.signal_select_db.emit(self.model)
 self.update_button_style(self.select_button)

@pyqtSlot()
def slot_remove_db_model(self):
 self.signal_delete_db.emit(self.model)

@pyqtSlot()
def slot_select_embedding_model(self):
 if self.model.embedding_model is None:
 self.signal_select_embedding_model.emit(self.model)

def open_folder_dialog(self):
 folder_path = QFileDialog.getExistingDirectory(self, "Виберіть папку")
 if folder_path:
 self.update_db_path(folder_path)

@pyqtSlot()
def slot_select_db_path(self):
 self.open_folder_dialog()

def update_view_data(self):
 try:
 name_embedding_model = ""
 if self.model.embedding_model is not None and isinstance(self.model.embedding_model,
EmbeddingModel):
 name_embedding_model = self.model.embedding_model.model_name
 self.selected_embedding_model_edit.setText(name_embedding_model)
 except Exception as ex:
 traceback.print_exc()

class CustomListWidgetItem(QListWidgetItem):

 def __init__(self, model_widget):
 super().__init__()
 self.setSizeHint(model_widget.sizeHint())
 self.setData(Qt.ItemDataRole.UserRole, model_widget.model)

class SelectEmbeddingModelDialog(QDialog):
 def __init__(self, embedding_models: List[EmbeddingModel], parent=None):
 super().__init__(parent)

 # Ініціалізація інтерфейсу
 self.setWindowTitle("Вибір моделі ембедінгу")

 # Створення ComboBox
 self.combo_box = QComboBox(self)

 self.combo_box.addItem("Виберіть модель.")

```

```

self.combo_box.currentIndexChanged.connect(self.on_index_changed_combo_models)

for i, model in enumerate(embedding_models):
 self.combo_box.addItem(model.model_name)
 self.combo_box.setItemData(i+1, model.id, Qt.ItemDataRole.UserRole)

Створення кнопки для підтвердження
self.ok_button = QPushButton("ОК", self)
self.ok_button.clicked.connect(self.on_ok_clicked)

Макет для компоновання елементів
layout = QVBoxLayout(self)
layout.addWidget(QLabel("Оберіть модель ембедінгу:"))
layout.addWidget(self.combo_box)
layout.addWidget(self.ok_button)

self.setLayout(layout)

self.selected_item_data = None

@pyqtSlot(int)
def on_index_changed_combo_models(self, index):
 self.combo_box.blockSignals(True)
 if index == 0:
 self.selected_item_data = None
 elif index > 0:
 self.selected_item_data = int(self.combo_box.itemData(index, Qt.ItemDataRole.UserRole))
 self.combo_box.blockSignals(False)

@pyqtSlot()
def on_ok_clicked(self):
 # Отримання вибраного значення з ComboBox
 selected_db_name = self.combo_box.currentText()
 print(f"Вибрано модель ембедінгу: {selected_db_name}, {self.selected_item_data}")

 self.accept()

class RAGWidget(QWidget):
 signal_update_embedding_model = pyqtSignal(EmbeddingModel)
 signal_delete_embedding_model = pyqtSignal(EmbeddingModel)

 signal_delete_db_model = pyqtSignal(EmbeddingDatabase)
 signal_update_db_model = pyqtSignal(EmbeddingDatabase)

 def __init__(self, app_controller:AppController, vector_db_controller:VectorDBController):
 super().__init__()
 self.app_controller:AppController = app_controller
 self.vector_db_controller:VectorDBController = vector_db_controller
 self.initUI()
 self.load_db_list()
 self.load_embedding_models()
 self.selected_doc_file_path = None
 self.selected_vectors_db = None

 def initUI(self):
 main_layout = QVBoxLayout()

 # QStackedWidget for the tabs
 self.stacked_widget = QStackedWidget()
 self.stacked_widget.addWidget(self.createAddDocumentsAndTextsTab())
 self.stacked_widget.addWidget(self.createBaseTab())
 self.stacked_widget.addWidget(self.createModelsTab())

```

```

self.stacked_widget.addWidget(self.createDocumentsTab())
self.stacked_widget.addWidget(self.createTestsTab())

Buttons for tabs
tab_buttons_layout = QHBoxLayout()
self.add_docs_and_text_button = QPushButton("Add Documents and Texts")
self.select_base_button = QPushButton("Select RAG Base")
self.models_button = QPushButton("Models")
self.documents_button = QPushButton("Documents")
self.tests_button = QPushButton("Tests")

self.add_docs_and_text_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(0))
self.add_docs_and_text_button.clicked.connect(self.update_documents_and_texts_tab)
self.select_base_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(1))
self.models_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(2))
self.documents_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(3))
self.documents_button.clicked.connect(self.document_tab_select)
self.tests_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(4))

tab_buttons_layout.addWidget(self.add_docs_and_text_button)
tab_buttons_layout.addWidget(self.select_base_button)
tab_buttons_layout.addWidget(self.models_button)
tab_buttons_layout.addWidget(self.documents_button)
tab_buttons_layout.addWidget(self.tests_button)

main_layout.addLayout(tab_buttons_layout)
main_layout.addWidget(self.stacked_widget)

self.setStyleSheet("background-color: #2E2E2E; color: #FFFFFF;")
self.setLayout(main_layout)

self.setStyleSheet("""
QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
}
QLineEdit {
 background-color: #4C4C4C;
 border: 1px solid #5C5C5C;

 color: #FFFFFF;
 border-radius: 3px;
}
QLineEdit:focus {
 border: 1px solid #4A90E2;
}
QLabel {
 font-weight: bold;
 color: #FFFFFF;
}
QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;

 border-radius: 5px;
}
QPushButton:hover {
 background-color: #357ABD;
}
""")

def createAddDocumentsAndTextsTab(self):

```

```

tab = QWidget()
layout = QVBoxLayout()

Вкладки для "Документ" та "Текст"
sub_tab_widget = QTabWidget()

Створення вкладки для додавання документа
document_tab = QWidget()
doc_layout = QFormLayout()
doc_layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

Кнопка для завантаження документа
self.load_doc_button = QPushButton("Select Document")
self.load_doc_button.clicked.connect(self.loadDocument)
doc_layout.addRow(self.load_doc_button)

self.add_doc_button = QPushButton("Add Doc")
self.add_doc_button.clicked.connect(self.addDoc)
doc_layout.addRow(self.add_doc_button)

document_tab.setLayout(doc_layout)

Створення вкладки для додавання тексту
text_tab = QWidget()
text_layout = QFormLayout()
text_layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

self.text_content = QTextEdit()
text_layout.addRow(QLabel("Content:"), self.text_content)

Кнопка для додавання тексту
self.add_text_button = QPushButton("Add Text")
self.add_text_button.clicked.connect(self.addText)
text_layout.addRow(self.add_text_button)

text_tab.setLayout(text_layout)

sub_tab_widget.addTab(document_tab, "Add Document")
sub_tab_widget.addTab(text_tab, "Add Text")

self.combo_select_embedding_db = QComboBox()
layout.addWidget(self.combo_select_embedding_db)
#self.combo_model.currentIndexChanged.connect(self.on_index_changed_combo_models)
Додавання основного `QTabWidget` у головне розташування
layout.addWidget(sub_tab_widget)
tab.setLayout(layout)

self.init_combo_select_embedding_db()

return tab

def update_documents_and_texts_tab(self):
 print("elem")
 self.combo_select_embedding_db.clear()
 self.init_combo_select_embedding_db()

def init_combo_select_embedding_db(self, datanames=None):
 self.combo_select_embedding_db.blockSignals(True)
 self.combo_select_embedding_db.addItem("Виберіть векторну базу.")

```

```

if datanames is None:
 datanames = self.app_controller.get_combo_selected_embedding_database()
for i, elem in enumerate(datanames):
 self.combo_select_embedding_db.addItem(elem[0])
 self.combo_select_embedding_db.setItemData(i+1, elem[1], Qt.ItemDataRole.UserRole)
self.combo_select_embedding_db.blockSignals(False)

Функція для завантаження документа (з вибором файлу)
def loadDocument(self):
 file_dialog = QFileDialog()
 #file_path, _ = file_dialog.getOpenFileName(self, "Open Document", "", "Text Files (*.txt);;All Files (*)")

 file_filters = ";;".join([f"{ext[1:].upper()} Files (*{ext})" for ext in LOADER_MAPPING.keys()])
 file_dialog = QFileDialog()

 # Оголошуємо фільтри діалогу для показу лише відповідних файлів
 file_path, _ = file_dialog.getOpenFileName(self, "Open Document", "", file_filters)

 if file_path:
 self.selected_doc_file_path = file_path

def get_selected_vectors_db(self):
 self.combo_select_embedding_db.blockSignals(True)
 index = self.combo_select_embedding_db.currentIndex()
 item_data = int(self.combo_select_embedding_db.itemData(index, Qt.ItemDataRole.UserRole))
 if index > 0:
 db_models = self.app_controller.get_selected_embeddings_databases()
 for model in db_models:
 if model.id == item_data:
 self.selected_vectors_db = model
 else:
 self.selected_vectors_db = None
 self.combo_select_embedding_db.blockSignals(False)
 return self.selected_vectors_db

def addDoc(self):
 vector_db = self.get_selected_vectors_db()
 print(vector_db)
 if self.selected_doc_file_path is not None and self.selected_doc_file_path != "":
 self.vector_db_controller.process_file_on_embedding_database(self.selected_doc_file_path, vector_db)

Функція для обробки додавання тексту
def addText(self):
 vector_db = self.get_selected_vectors_db()
 content = self.text_content.toPlainText()
 # Обробка додавання тексту, наприклад, збереження або відображення
 print("Added Text:", content)
 print(vector_db)
 self.vector_db_controller.process_text_on_embedding_database(content, vector_db)

@pyqtSlot(EmbeddingDatabase)
def slot_update_db_model(self, db_model):
 self.signal_update_db_model.emit(db_model)

@pyqtSlot(EmbeddingDatabase)
def select_embedding_model(self, model: EmbeddingDatabase):
 if model.was_created is False or model.embedding_model is None:
 embedding_models = self.vector_db_controller.get_list_embedding_models()
 dialog = SelectEmbeddingModelDialog(embedding_models)
 if dialog.exec() == QDialog.DialogCode.Accepted:

 if dialog.selected_item_data is not None:

```

```

for model_em in embedding_models:
 if dialog.selected_item_data == model_em.id:
 model.embedding_model = model_em
 model.model_id = model_em.id
 model.was_created = True
 self.signal_update_db_model.emit(model)
 time.sleep(0.1)

 for row in range(self.all_list.count()):
 item = self.all_list.item(row)
 widget = self.all_list.itemWidget(item) # Отримуємо віджет для елемента
 if widget and isinstance(widget, RAGDBWidget):
 if widget.model is model:
 widget.update_view_data()

 msg = QMessageBox(self)
 msg.setIcon(QMessageBox.Icon.Information)
 msg.setText(f"Ембедінг модель вибрана. {model.embedding_model.model_name}")
 msg.setWindowTitle("Повідомлення")
 msg.exec()

 break
else:
 msg = QMessageBox(self)
 msg.setIcon(QMessageBox.Icon.Information)
 msg.setText("Ембедінг модель не може бути змінена, оскільки векторна база вже наповнена.")
 msg.setWindowTitle("Помилка")
 msg.exec()

def create_ragdb_widget_and_add_to_list(self, model):
 model_widget = RAGDBWidget(model)

 model_widget.signal_select_db.connect(self.vector_db_controller.select_deselect_DB)
 model_widget.signal_deselect_db.connect(self.vector_db_controller.select_deselect_DB)
 model_widget.signal_select_embedding_model.connect(self.select_embedding_model)
 model_widget.signal_delete_db.connect(self.remove_db_model)
 model_widget.signal_update_db.connect(self.slot_update_db_model)

 item = CustomListWidgetItem(model_widget)
 self.all_list.addItem(item)
 self.all_list.setItemWidget(item, model_widget)

@pyqtSlot()
def load_db_list(self):
 self.all_list.clear()
 list_edb:List[EmbeddingDatabase] = self.vector_db_controller.get_list_embedding_databases()
 for model in list_edb:
 self.create_ragdb_widget_and_add_to_list(model)

@pyqtSlot(EmbeddingDatabase)
def remove_db_model(self, db_model:EmbeddingDatabase):
 self.signal_delete_db_model.emit(db_model)

def createBaseTab(self):
 tab = QWidget()
 layout = QVBoxLayout()

 self.add_button = QPushButton("Create New Base")
 self.add_button.setStyleSheet("""
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;

```

```

 border: none;
 padding: 10px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """
self.add_button.clicked.connect(self.addNewDB)
layout.addWidget(self.add_button)

layout_list = QHBoxLayout()
self.all_list = QListWidget()

self.all_list.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOn) # Завжди показувати
вертикальну смугу прокручування
self.all_list.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAsNeeded) # Прокручування по
горизонталі, якщо потрібно
self.all_list.setVerticalScrollMode(QAbstractItemView.ScrollMode.ScrollPerPixel)
self.all_list.setAutoScroll(True)
self.all_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")
layout_list.addWidget(self.all_list)

layout.addLayout(layout_list)

tab.setLayout(layout)
return tab

def addNewDB(self):
 model = self.vector_db_controller.create_new_embedding_database()
 self.create_ragdb_widget_and_add_to_list(model)

@pyqtSlot()
def load_embedding_models(self):
 self.models_list.clear()
 embedding_models = self.vector_db_controller.get_list_embedding_models()
 for model in embedding_models:
 self.create_ragmodel_widget_and_add_to_list(model)

@pyqtSlot(EmbeddingModel)
def slot_update_embedding_model(self, model: EmbeddingModel):
 params = self.vector_db_controller.get_params_embedding_model(model)
 model.embedding_dim = params.dimensional
 model.max_seq_length = params.max_context_size

 for row in range(self.models_list.count()):
 item = self.models_list.item(row)
 widget = self.models_list.itemWidget(item) # Отримуємо віджет для елемента
 if widget and isinstance(widget, RAGModelWidget):
 if widget.model is model:
 widget.update_view_data()

 self.signal_update_embedding_model.emit(model)

@pyqtSlot(EmbeddingModel)
def slot_delete_embedding_model(self, model: EmbeddingModel):
 self.signal_delete_embedding_model.emit(model)

def create_ragmodel_widget_and_add_to_list(self, model):
 model_widget = RAGModelWidget(model)

 model_widget.signal_update_model.connect(self.slot_update_embedding_model)
 model_widget.signal_delete_model.connect(self.slot_delete_embedding_model)

```

```

 item = CustomListItemWidget(model_widget)
 self.models_list.addItem(item)
 self.models_list.setItemWidget(item, model_widget)

 def addEmbeddingModel(self):
 model = self.vector_db_controller.create_new_rag_model_widget()
 self.create_ragmodel_widget_and_add_to_list(model)

 def createModelsTab(self):
 tab = QWidget()
 layout = QVBoxLayout()

 self.add_button = QPushButton("Add Model")
 self.add_button.setStyleSheet("""
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 10px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """)
 self.add_button.clicked.connect(self.addEmbeddingModel)
 layout.addWidget(self.add_button)

 self.models_list = QListWidget()
 self.models_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")
 layout.addWidget(self.models_list)

 tab.setLayout(layout)
 return tab

 def init_combo_documents_select_embedding_db(self, datanames=None):
 self.combo_documents.blockSignals(True)
 self.combo_documents.addItem("Виберіть векторну базу.")
 if datanames is None:
 datanames = self.app_controller.get_combo_selected_embedding_database()
 for i, elem in enumerate(datanames):
 self.combo_documents.addItem(elem[0])
 self.combo_documents.setItemData(i+1, elem[1], Qt.ItemDataRole.UserRole)
 self.combo_documents.blockSignals(False)

 @pyqtSlot(int)
 def on_index_changed_combo_documents(self, index):
 print(index)
 self.combo_documents.blockSignals(True)
 if index > 0:
 item_data = int(self.combo_documents.itemData(index, Qt.ItemDataRole.UserRole))
 db_model = self.vector_db_controller.get_embedding_database_on_id(item_data)
 texts = self.vector_db_controller.get_texts_on_embedding_database(db_model)
 self.updateDocuments(texts)
 self.combo_documents.blockSignals(False)

 def document_tab_select(self):
 self.combo_documents.clear()
 self.init_combo_documents_select_embedding_db()

 def createDocumentsTab(self):

```

```

self.documents = []
self.rows_per_page = 10 # Початкове значення для кількості рядків
self.current_page = 0
self.total_pages = 1 # Ініціалізація, щоб уникнути помилок

tab = QWidget()
layout = QVBoxLayout()
Комбобокс
self.combo_documents = QComboBox()
self.init_combo_documents_select_embedding_db()
self.combo_documents.currentIndexChanged.connect(self.on_index_changed_combo_documents) #
Підключаємо зміну категорії
layout.addWidget(self.combo_documents)

Створення таблиці
self.documents_table = QTableWidgetItem()
self.documents_table.setColumnCount(2)
self.documents_table.setHorizontalHeaderLabels(["ID", "Content"])
self.documents_table.setSizePolicy(QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Expanding)

self.documents_table.setVerticalScrollMode(QAbstractItemView.ScrollMode.ScrollPerPixel)
self.documents_table.setHorizontalScrollMode(QAbstractItemView.ScrollMode.ScrollPerPixel)
self.documents_table.verticalScrollBar().setSingleStep(10)
self.documents_table.horizontalScrollBar().setSingleStep(10)

Розтягування Content на всю ширину
header = self.documents_table.horizontalHeader()
header.setSectionResizeMode(0, QHeaderView.ResizeMode.ResizeToContents) # ID стовпець
header.setSectionResizeMode(1, QHeaderView.ResizeMode.Stretch) # Content стовпець
розтягується

layout.addWidget(self.documents_table)

Створення навігаційних елементів
navigation_layout = QHBoxLayout()

self.prev_button = QPushButton("Prev")
self.prev_button.clicked.connect(self.prevPage)
navigation_layout.addWidget(self.prev_button)

Додавання індикатора сторінки та поля введення
self.info_label = QLabel(f"Current page:")
navigation_layout.addWidget(self.info_label)

self.page_input = QLineEdit(str(self.current_page + 1))
self.page_input.setFixedWidth(40)
self.page_input.setAlignment(Qt.AlignmentFlag.AlignCenter)
self.page_input.setPlaceholderText("Page")
self.page_input.returnPressed.connect(self.goToPage) # Переход до введеної сторінки
navigation_layout.addWidget(self.page_input)

Відображення загальної кількості сторінок
self.total_pages_label = QLabel(f"/ {self.total_pages}") # Ініціалізація total_pages_label
navigation_layout.addWidget(self.total_pages_label)

self.next_button = QPushButton("Next")
self.next_button.clicked.connect(self.nextPage)
navigation_layout.addWidget(self.next_button)

layout.addLayout(navigation_layout)

```

```

tab.setLayout(layout)

self.calculateTotalPages() # Після створення total_pages_label
self.loadPage(self.current_page) # Завантаження першої сторінки

return tab

def resizeEvent(self, event):
 # Викликається при зміні розміру вікна
 row_height = self.documents_table.verticalHeader().defaultSectionSize()
 available_height = self.documents_table.viewport().height()

 # Визначаємо скільки рядків вміщується у видимій області таблиці
 new_rows_per_page = available_height // row_height

 if new_rows_per_page != self.rows_per_page:
 self.rows_per_page = new_rows_per_page
 self.calculateTotalPages()
 self.loadPage(self.current_page)

 super().resizeEvent(event)

def calculateTotalPages(self):
 # Перераховуємо загальну кількість сторінок
 self.total_pages = (len(self.documents) + self.rows_per_page - 1) // self.rows_per_page
 self.total_pages_label.setText(f"/ {self.total_pages}")

def updateDocuments(self, documents):
 # Оновлюємо documents на основі обраної категорії
 self.documents = documents

 # Перезапуск пагінації та завантаження першої сторінки
 self.current_page = 0
 self.calculateTotalPages()
 self.page_input.setText(str(self.current_page + 1))
 self.loadPage(self.current_page)

def loadPage(self, page_number):
 # Очищення таблиці
 self.documents_table.clearContents()
 self.documents_table.setRowCount(self.rows_per_page) # Налаштовуємо кількість рядків

 # Розрахунок початкового індексу
 start_index = page_number * self.rows_per_page
 end_index = min(start_index + self.rows_per_page, len(self.documents))

 # Заповнення таблиці даними
 for row, document in enumerate(self.documents[start_index:end_index]):
 self.documents_table.setItem(row, 0, QTableWidgetItem(str(document["id"])))
 self.documents_table.setItem(row, 1, QTableWidgetItem(str(document["content"])))

 # Оновлення індикатора сторінки
 self.info_label.setText(f"Current page:")
 self.page_input.setText(str(self.current_page + 1))

 # Вимикання кнопок якщо немає сторінок в певному напрямку
 self.prev_button.setEnabled(self.current_page > 0)
 self.next_button.setEnabled(self.current_page < self.total_pages - 1)

 self.documents_table.resizeRowsToContents()

def goToPage(self):

```

```

Отримуємо номер сторінки з введення
try:
 page = int(self.page_input.text()) - 1
 if 0 <= page < self.total_pages:
 self.current_page = page
 self.loadPage(self.current_page)
 else:
 # Якщо введена сторінка за межами допустимого діапазону
 self.page_input.setText(str(self.current_page + 1))
except ValueError:
 # Якщо введено некоректне значення
 self.page_input.setText(str(self.current_page + 1))

def nextPage(self):
 if self.current_page < self.total_pages - 1:
 self.current_page += 1
 self.loadPage(self.current_page)

def prevPage(self):
 if self.current_page > 0:
 self.current_page -= 1
 self.loadPage(self.current_page)

def createTestsTab(self):
 tab = QWidget()
 layout = QVBoxLayout()

 # Поле для введення запитного тексту
 self.query_input = QLineEdit()
 self.query_input.setPlaceholderText("Enter text to find similar documents...")
 self.query_input.setStyleSheet("""
 QLineEdit {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 1px solid #4A90E2;
 border-radius: 5px;
 padding: 8px;
 font-size: 14px;
 }
 """)
 layout.addWidget(QLabel("Enter Query Text:"))
 layout.addWidget(self.query_input)

 # Кнопка для запуску тесту
 self.search_button = QPushButton("Run Similarity Test")
 self.search_button.setStyleSheet("""
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 10px;
 border-radius: 5px;
 font-size: 14px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """)
 self.search_button.clicked.connect(self.runSimilarityTest)
 layout.addWidget(self.search_button)

 # Таблиця для результатів тесту
 self.results_table = QTableWidgetItem()

```

```

self.results_table.setStyleSheet("""
 QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 1px solid #4A90E2;
 border-radius: 5px;
 }
 QHeaderView::section {
 background-color: #4A90E2;
 color: #FFFFFF;
 padding: 4px;
 font-weight: bold;
 }
""")
layout.addWidget(QLabel("Results:"))
layout.addWidget(self.results_table)

tab.setLayout(layout)
return tab

def runSimilarityTest(self):
 query_text = self.query_input.text()

 results = self.vector_db_controler.query(query_text)

 self.results_table.clear()
 self.results_table.setColumnCount(3) # Колонки: текст і схожість
 self.results_table.setHorizontalHeaderLabels(["Similarity Score", "ID", "Similar Text",])
 self.results_table.horizontalHeader().setSectionResizeMode(QHeaderView.ResizeMode.Stretch)
 self.results_table.setVerticalScrollMode(QAbstractItemView.ScrollMode.ScrollPerPixel)
 self.results_table.setHorizontalScrollMode(QAbstractItemView.ScrollMode.ScrollPerPixel)
 self.results_table.verticalScrollBar().setSingleStep(10)
 self.results_table.horizontalScrollBar().setSingleStep(10)
 # Оновлення таблиці результатів
 self.results_table.setRowCount(len(results))
 for row, result in enumerate(results):

 text_item = QTableWidgetItem(result.text)

 similarity_item = QTableWidgetItem(f"{int(result.distance)}")
 similarity_item.setTextAlignment(Qt.AlignmentFlag.AlignCenter)

 id_item = QTableWidgetItem(f"{result.id}")
 id_item.setTextAlignment(Qt.AlignmentFlag.AlignCenter)

 self.results_table.setItem(row, 0, similarity_item)
 self.results_table.setItem(row, 1, id_item)
 self.results_table.setItem(row, 2, text_item)

 header = self.results_table.horizontalHeader()
 header.setSectionResizeMode(0, QHeaderView.ResizeMode.ResizeToContents)
 header.setSectionResizeMode(1, QHeaderView.ResizeMode.ResizeToContents)
 header.setSectionResizeMode(2, QHeaderView.ResizeMode.ResizeToContents)
 self.results_table.resizeRowsToContents()

 # Додатково очищаємо введений текст для нових тестів
 #self.query_input.clear()

...

...py

```

```

from PyQt6 import QtWidgets, QtGui
from PyQt6.QtWebEngineWidgets import QWebEngineView
from PyQt6.QtWidgets import QLineEdit, QPushButton, QVBoxLayout, QWidget, QTextBrowser
import sys
import markdown
from pygments import highlight
from pygments.lexers import PythonLexer
from pygments.formatters import HtmlFormatter

class MarkdownEditor(QWidget):
 def __init__(self):
 super().__init__()

 # Ініціалізація елементів
 self.editor = QLineEdit(self) # Використовуємо QLineEdit для введення Markdown
 self.editor.setFont(QtGui.QFont("Courier", 10))

 # QWebEngineView для відображення HTML
 self.web_view = QWebEngineView(self)
 self.web_view.setHtml("<html><body></body></html>") # Пустий HTML для початку
 self.web_view = QTextBrowser(self)
 self.web_view.setHtml("<html><body></body></html>")

 # Кнопка форматування
 self.format_button = QPushButton("Форматувати Markdown", self)
 self.format_button.clicked.connect(self.convert_markdown)

 # Розташування
 layout = QVBoxLayout(self)
 layout.addWidget(self.editor) # Додаємо QLineEdit
 layout.addWidget(self.format_button)
 layout.addWidget(self.web_view) # Додаємо QWebEngineView
 self.setLayout(layout)

 def convert_markdown(self):
 # Отримання тексту з QLineEdit
 markdown_text = self.editor.text()

 # Підсвічування коду в HTML
 highlighted_html = highlight_code_blocks(markdown_text)

 # Оновлення веб-переглядача з HTML
 self.web_view.setHtml(highlighted_html)

 def highlight_code_blocks(markdown_text):
 """Функція для підсвічування блоків коду в Markdown."""
 # Розбивка тексту на блоки за трьома зворотними апострофами
 blocks = markdown_text.split('```')
 highlighted_content = ""

 for i in range(len(blocks)):
 block = blocks[i].strip() # Обрізаємо пробіли на початку та в кінці
 if i % 2 == 1: # Це блок коду
 # Визначаємо мову програмування
 lines = block.split('\n')
 code_language = lines[0].strip() if lines else 'python'
 code = '\n'.join(lines[1:]).strip() # Код без мови

 # Вибір відповідного лексера
 lexer = {
 'python': PythonLexer()

```

```

 }.get(code_language.lower(), PythonLexer()) # За замовчуванням - PythonLexer

 formatter = HtmlFormatter()
 highlighted_code = highlight(code, lexer, formatter)

 css_style = formatter.get_style_defs()
 final_html = f'<style> {css_style} </style> {highlighted_code}'
 highlighted_content += final_html
else: # Звичайний текст

 highlighted_content += markdown.markdown(block)

return f'<html><body>{highlighted_content}</body></html>'

if __name__ == "__main__":
 app = QtWidgets.QApplication(sys.argv)
 window = MarkdownEditor()
 window.resize(800, 600)
 window.show()
 sys.exit(app.exec())
'''

''.py
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from AppConfig import config
from lingua import Language, LanguageDetectorBuilder

class TranslationModel:
 def __init__(self):
 pass
 #https://huggingface.co/Helsinki-NLP/opus-mt-en-uk
 #https://huggingface.co/Helsinki-NLP/opus-mt-ru-en
 #https://huggingface.co/Helsinki-NLP/opus-mt-uk-en

 self.path_opus_mt_en_uk:str = config.path_opus_mt_en_uk
 self.path_opus_mt_ru_en:str = config.path_opus_mt_ru_en
 self.path_opus_mt_uk_en:str = config.path_opus_mt_uk_en

 def translate_en_uk(self, text):
 tokenizer = AutoTokenizer.from_pretrained("Helsinki-NLP/opus-mt-en-uk",
cache_dir=self.path_opus_mt_en_uk)
 model = AutoModelForSeq2SeqLM.from_pretrained("Helsinki-NLP/opus-mt-en-uk",
cache_dir=self.path_opus_mt_en_uk)
 input_ids = tokenizer(text, return_tensors="pt").input_ids
 outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequences=1)
 return tokenizer.batch_decode(outputs, skip_special_tokens=True)[0]

 def translate_ru_en(self, text):
 tokenizer = AutoTokenizer.from_pretrained("Helsinki-NLP/opus-mt-ru-en",
cache_dir=self.path_opus_mt_ru_en)
 model = AutoModelForSeq2SeqLM.from_pretrained("Helsinki-NLP/opus-mt-ru-en",
cache_dir=self.path_opus_mt_ru_en)
 input_ids = tokenizer(text, return_tensors="pt").input_ids
 outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequences=1)
 return tokenizer.batch_decode(outputs, skip_special_tokens=True)[0]

 def translate_uk_en(self, text):
 tokenizer = AutoTokenizer.from_pretrained("Helsinki-NLP/opus-mt-uk-en",
cache_dir=self.path_opus_mt_uk_en)

```

```

 model = AutoModelForSeq2SeqLM.from_pretrained("Helsinki-NLP/opus-mt-uk-en",
cache_dir=self.path_opus_mt_uk_en)
 input_ids = tokenizer(text, return_tensors="pt").input_ids
 outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequences=1)
 return tokenizer.batch_decode(outputs, skip_special_tokens=True)[0]

 def to_en(self, text):
 detector = LanguageDetectorBuilder.from_languages(Language.UKRAINIAN, Language.RUSSIAN,
Language.ENGLISH).build()
 language = detector.detect_language_of(text)
 print(language)

 if language == Language.UKRAINIAN:
 return self.translate_uk_en(text)
 elif language == Language.RUSSIAN:
 return self.translate_ru_en(text)
 elif language == Language.ENGLISH:
 return text
 else:
 return text

 ...

```.py
from PyQt6.QtCore import QObject, pyqtSignal, pyqtSlot
from DB import DatabaseManager
from DBModels import EmbeddingDatabase, EmbeddingModel, TextEmbedding
from typing import List
from llama_cpp import Llama
import numpy as np
from llama_index.legacy.text_splitter import TokenTextSplitter
import os
from EngineEmbeddingDatabase import EngineEmbeddingDatabase, data_text_embedding

class VectorDBController(QObject):

    def __init__(self, database_manager:DatabaseManager):
        super().__init__()
        self.database_manager = database_manager
        self.selected_db_models:List[EmbeddingDatabase] = []
        self.loaded_engines:List[EngineEmbeddingDatabase] = []

    def get_count_selected_models(self):
        return len(self.selected_db_models)

    def remove_db_index(self, embedding_database:EmbeddingDatabase):
        engine = EngineEmbeddingDatabase(embedding_database, skip_error_for_remove=True)
        engine.remove_index()
        del engine

    @pyqtSlot(data_text_embedding)
    def slot_create_text_embedding(self, data:data_text_embedding):
        self.create_text_embedding(data.id, data.text, data.db)

    def get_list_embedding_databases(self):
        #all_db:List[EmbeddingDatabase] = self.database_manager.get_all_records(EmbeddingDatabase)
        all_db:List[EmbeddingDatabase] = self.database_manager.get_all_embedding_databases()

        for i, elem in enumerate(all_db):
            all_db[i] = self.database_manager.get_embedding_database_for_id(elem.id)

        return all_db

```

```

def create_text_embedding(self, id_embedding:int, text:str, embedding_database:EmbeddingDatabase):
    te = TextEmbedding()
    te.embedding_id = id_embedding
    te.text_content = text
    te.database = self.database_manager.get_embedding_database_for_id(embedding_database.id)
    #te.db_id = embedding_database.id
    record_id = self.database_manager.add_record(te)
    print(f"create_text_embedding {record_id}")
    return record_id

def create_new_embedding_database(self):
    edb = EmbeddingDatabase()
    record_id = self.database_manager.add_record(edb)
    edb = self.database_manager.get_embedding_database_for_id(record_id)
    return edb

class TestParamsEmbeddingModel:
    def __init__(self):
        self.dimensional = 0
        self.max_context_size = 0

def get_params_embedding_model(self, em_model:EmbeddingModel):
    tpem = self.TestParamsEmbeddingModel()
    if em_model.model_path != "":
        llm = Llama(
            model_path=em_model.model_path,
            embedding=True,
            n_ctx=0 # Задаємо так бо якщо 0 воно візьме розмір рівний тому на якому тренувалась модель
        )
        dimensional = llm.n_embd()
        max_context_size = llm.n_ctx()
        tpem.dimensional = dimensional
        tpem.max_context_size = max_context_size
        del llm
    return tpem

def create_new_rag_model_widget(self):
    em = EmbeddingModel()
    record_id = self.database_manager.add_record(em)
    em = self.database_manager.get_record_by_id(record_id, EmbeddingModel)
    return em

def get_list_embedding_models(self) -> List[EmbeddingModel]:
    embedding_models:List[EmbeddingModel] = self.database_manager.get_all_records(EmbeddingModel)
    return embedding_models

@pyqtSlot(EmbeddingDatabase)
def select_deselect_DB(self, db_model):
    exist = False
    for loc_model in self.selected_db_models:
        if loc_model is db_model:
            self.selected_db_models.remove(db_model)
            exist = True
            break

    if exist is True:
        for loc_eed in self.loaded_engines:
            if loc_eed.db_model is db_model:
                self.loaded_engines.remove(loc_eed)
                break

    if exist is False:

```

```

        self.selected_db_models.append(db_model)
        eed = EngineEmbeddingDatabase(db_model)
        eed.signal_create_text_embedding.connect(self.slot_create_text_embedding)
        self.loaded_engines.append(eed)
    pass

def query(self, text:str):
    results: List[EngineEmbeddingDatabase.SearchResult] = []
    for eed in self.loaded_engines:
        results.extend( eed.search(text) )
    for result in results:
        result.text = self.get_text_by_id_embedding(result.id, result.db_id)
    return results

def get_text_by_id_embedding(self, id_embedding, db_id):
    text_embedding: TextEmbedding
self.database_manager.get_text_embedding_by_embedding_id(id_embedding, db_id)
    if text_embedding is None:
        print(f"text_embedding is None and id_embedding: {id_embedding} db_id: {db_id}")
        return ""
    return text_embedding.text_content

def get_engine_db_on_embedding_database(self, embedding_database: EmbeddingDatabase):
    engine_db = None
    for eed in self.loaded_engines:
        if eed.db_model.id == embedding_database.id:
            engine_db = eed
            break
    return engine_db

def process_text_on_embedding_database(self, text:str, embedding_database: EmbeddingDatabase):
    engine_db = self.get_engine_db_on_embedding_database(embedding_database)
    engine_db.process_text(text)

def process_file_on_embedding_database(self, file_path:str, embedding_database: EmbeddingDatabase):
    engine_db = self.get_engine_db_on_embedding_database(embedding_database)
    engine_db.process_file(file_path)

def get_texts_on_embedding_database(self, embedding_database: EmbeddingDatabase):
    text_embeddings: List[TextEmbedding]
self.database_manager.get_text_embeddings_for_database(embedding_database.id)
    result = []
    for element in text_embeddings:
        result.append({
            "id": element.id,
            "content": element.text_content
        })
    return result

def get_embedding_database_on_id(self, id):
    db_model: EmbeddingDatabase = self.database_manager.get_embedding_database_for_id(id)
    return db_model

...

```py
from FAISS_DB import HNSWImageSearch
from llama_cpp import Llama
import os
import numpy as np

def current_path():

```

```

return os.path.dirname(os.path.abspath(__file__))

class VectorDB:
 def __init__(self):
 self.index_path = os.path.join(current_path(), "index/faiss_hnsw.index")

 self.model_path = os.path.join(current_path(), "models/nomic-embed-text-v1.5.f16.gguf")

 self.llm = Llama(
 model_path=self.model_path,
 embedding=True,
 n_ctx=0
)
 #test_text = "test"
 #embedding = self.llm.create_embedding(test_text)
 #embedding = self.llm.embed(test_text)

 self.dimensional = self.llm.n_embd()
 max_context_size = self.llm.n_ctx()
 self.search_engine = HNSWImageSearch(self.dimensional, index_path=self.index_path)

 def add_text(self, text):
 if isinstance(text, str) and text != "":
 embedding = self.llm.embed(text)
 embeddings = np.array([embedding])
 self.search_engine.add_embeddings(embeddings)
 self.search_engine.save_index()

 def add_txt(self):
 pass

 def add_pdf(self):
 pass

 def add_directory(self):
 pass

 def add_files(self):
 pass

 def search(self, text):
 if isinstance(text, str) and text != "":
 embedding = self.llm.embed(text)
 query_embeddings = np.array([embedding])
 distances, indices = self.search_engine.search(query_embeddings, k=5)
 pass

if __name__ == "__main__":
 vdb = VectorDB()
 vdb.add_text("test for test")
 vdb.search("test for test")
 pass ```

`.py
import tkinter as tk
from tkinter import scrolledtext
from tkinter import ttk

Функція для відповіді бота
def get_bot_response(user_input):
 user_input = user_input.lower()
 if "привіт" in user_input:

```

```

 return "Привіт! Як можу допомогти?"
 elif "як справи" in user_input:
 return "Все чудово, дякую! А в тебе як?"
 elif "до побачення" in user_input:
 return "До побачення! Гарного дня!"
 else:
 return "На жаль, не розумію твоє запитання."

Функція для обробки повідомлення
def send_message(chat_window, input_box):
 user_message = input_box.get()
 if user_message:
 chat_window.configure(state='normal')
 chat_window.insert(tk.END, f"Ви: {user_message}\n")
 bot_response = get_bot_response(user_message)
 chat_window.insert(tk.END, f"Бот: {bot_response}\n")
 chat_window.configure(state='disabled')
 chat_window.yview(tk.END)
 input_box.delete(0, tk.END)

Функція для перемикання між чатами
def switch_chat(chat_name):
 for name, frame in chat_frames.items():
 frame.pack_forget()
 chat_frames[chat_name].pack(fill=tk.BOTH, expand=True)

Створення нового чату
def create_chat(chat_name):
 chat_frame = ttk.Frame(chat_area)
 chat_frames[chat_name] = chat_frame

 # Вікно чату
 chat_window = scrolledtext.ScrolledText(chat_frame, wrap=tk.WORD, state='disabled', bg="#f0f0f0",
 fg="#000000", font=("Arial", 12))
 chat_window.pack(padx=10, pady=(10, 0), fill=tk.BOTH, expand=True)

 # Поле для введення тексту
 input_box = tk.Entry(chat_frame, font=("Arial", 12))
 input_box.pack(padx=10, pady=10, fill=tk.X)
 input_box.bind("<Return>", lambda event: send_message(chat_window, input_box))

 # Кнопка відправки
 send_button = tk.Button(chat_frame, text="Відправити", command=lambda: send_message(chat_window,
 input_box), bg="#4CAF50", fg="white", font=("Arial", 12))
 send_button.pack(padx=10, pady=(0, 10))

 # Додаємо чат у список
 chat_list.insert(tk.END, chat_name)
 chat_list.bind("<<ListboxSelect>>", lambda event: switch_chat(chat_name))

Налаштування головного вікна
root = tk.Tk()
root.title("Чат-бот з панеллю чатів")
root.geometry("600x500")
root.configure(bg="#ffffff")

Структура вікна
main_frame = tk.Frame(root, bg="#ffffff")
main_frame.pack(fill=tk.BOTH, expand=True)

Панель для списку чатів
chat_list_frame = tk.Frame(main_frame, bg="#f0f0f0", width=150)
chat_list_frame.pack(side=tk.LEFT, fill=tk.Y)

```

```

Список чатів
chat_list = tk.Listbox(chat_list_frame, bg="#f0f0f0", font=("Arial", 12), selectbackground="#4CAF50",
selectforeground="white")
chat_list.pack(padx=10, pady=10, fill=tk.Y, expand=True)

Основна область чату
chat_area = tk.Frame(main_frame, bg="ffffff")
chat_area.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

Словник для зберігання чатів
chat_frames = {}

Створюємо кілька чатів
create_chat("Чат 1")
create_chat("Чат 2")
create_chat("Чат 3")

Відображаємо перший чат за замовчуванням
switch_chat("Чат 1")

Запуск головного циклу Tkinter
root.mainloop()
'''

`.py
from PyQt6.QtWidgets import (
 QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
 QLabel, QLineEdit, QPushButton, QListWidget, QListWidgetItem, QTextEdit,
 QStackedWidget, QFrame, QToolButton, QTextBrowser
)
from PyQt6.QtCore import QSize, Qt
from PyQt6.QtGui import QIcon
import sys
from markdown2 import markdown

class ChatWindow(QMainWindow):
 def __init__(self):
 super().__init__()

 self.setWindowTitle("ChatGui")
 self.resize(800, 600)

 # Атрибути для зберігання повідомлень
 self.messages = []

 fullscreen_button = QToolButton()
 fullscreen_button.setText("⌕")
 fullscreen_button.setCheckable(True)
 fullscreen_button.setFixedSize(30, 30)
 fullscreen_button.clicked.connect(self.toggle_fullscreen)

 main_widget = QWidget()
 self.setCentralWidget(main_widget)
 main_layout = QHBoxLayout(main_widget)

 sidebar = QFrame()
 sidebar.setStyleSheet("background-color: #2C3E50;")
 sidebar_layout = QVBoxLayout(sidebar)
 sidebar_layout.setAlignment(Qt.AlignmentFlag.AlignTop)

 def create_sidebar_button(icon_path, tooltip):
 button = QToolButton()

```

```

 button.setIcon(QIcon(icon_path))
 button.setIconSize(QSize(40, 40))
 button.setStyleSheet("background: none; border: none;")
 button.setToolTip(tooltip)
 return button

chat_button = create_sidebar_button(r"D:\Works\pavlo_m\test\icons/chat_icon.png", "Чати")
chat_button.clicked.connect(lambda: self.pages.setCurrentIndex(0))
sidebar_layout.addWidget(chat_button)

settings_button = create_sidebar_button(r"D:\Works\pavlo_m\test\icons/settings_icon.png",
"Налаштування")
settings_button.clicked.connect(lambda: self.pages.setCurrentIndex(1))
sidebar_layout.addWidget(settings_button)

about_button = create_sidebar_button(r"D:\Works\pavlo_m\test\icons/about_icon.png", "Про програму")
about_button.clicked.connect(lambda: self.pages.setCurrentIndex(2))
sidebar_layout.addWidget(about_button)

main_layout.addWidget(sidebar)

self.pages = QStackedWidget()

Сторінка "Чати"
chat_page = QWidget()
chat_layout = QVBoxLayout(chat_page)

self.toggle_chat_button = QPushButton("▼ Показати чати")
self.toggle_chat_button.setFixedHeight(40)
self.toggle_chat_button.setStyleSheet("""
 QPushButton {
 background-color: #ECF0F1;
 color: #2C3E50;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #BDC3C7;
 }
 """)
self.toggle_chat_button.clicked.connect(self.toggle_chat_list)

self.chat_list = QListWidget()
self.chat_list.itemClicked.connect(self.on_item_clicked)
self.chat_list.setStyleSheet("""
 QListWidget {
 background-color: #34495E;
 color: #ECF0F1;
 border: none;
 padding: 5px;
 }
 QListWidgetItem {
 margin: 5px;
 }
 """)

self.update_chat_list()

Динамічна область для повідомлень
self.messages_layout = QVBoxLayout()

self.messages_scroll_area = QWidget()

```

```

self.messages_scroll_area.setLayout(self.messages_layout)

input_layout = QHBoxLayout()
self.input_field = QLineEdit()
self.input_field.setPlaceholderText("Введіть своє повідомлення...")
self.input_field.setStyleSheet("padding: 5px; border-radius: 5px; border: 1px solid #BDC3C7;")

send_button = QPushButton("Відправити")
send_button.setStyleSheet("""
 QPushButton {
 background-color: #1ABC9C;
 color: white;
 border-radius: 5px;
 padding: 5px 10px;
 }
 QPushButton:hover {
 background-color: #16A085;
 }
 """)
send_button.clicked.connect(self.send_message)

chat_layout.addWidget(self.toggle_chat_button)
chat_layout.addWidget(self.chat_list)
chat_layout.addWidget(self.messages_scroll_area) # Додаємо область для повідомлень
chat_layout.addLayout(input_layout)

input_layout.addWidget(self.input_field)
input_layout.addWidget(send_button)

self.pages.addWidget(chat_page)

Сторінка "Налаштування"
settings_page = QWidget()
settings_layout = QVBoxLayout(settings_page)
settings_label = QLabel("Налаштування тут")
settings_layout.addWidget(settings_label)
self.pages.addWidget(settings_page)

Сторінка "Про програму"
about_page = QWidget()
about_layout = QVBoxLayout(about_page)
about_label = QLabel("Інформація про програму")
about_layout.addWidget(about_label)
self.pages.addWidget(about_page)

main_layout.addWidget(self.pages, 1)

def toggle_fullscreen(self):
 if self.isFullScreen():
 self.showNormal()
 else:
 self.showFullScreen()

def toggle_chat_list(self):
 if self.chat_list.isVisible():
 self.chat_list.setVisible(False)
 self.toggle_chat_button.setText("▲ Показати чати")
 else:
 self.update_chat_list()
 self.chat_list.setVisible(True)
 self.toggle_chat_button.setText("▼ Приховати чати")

def on_item_clicked(self, item: QListWidgetItem):

```

```

 print(f"Clicked on: {item.text()}")

def get_chat_list(self):
 return ["Tom", "Jerry", "Me from the future"]

def update_chat_list(self):
 self.chat_list.clear()
 for name in self.get_chat_list():
 item = QListWidgetItem(name)
 self.chat_list.addItem(item)

def create_message_widget(self, message):
 message_widget = QFrame()
 message_widget.setStyleSheet("background-color: #ECF0F1; border-radius: 5px; padding: 10px; margin: 5px;")

 message_display = QTextBrowser()
 message_display.setOpenExternalLinks(True) # Дозволяє відкривати посилання
 message_display.setHtml(self.format_message(message))
 message_display.setStyleSheet("border: none;")

 message_layout = QVBoxLayout(message_widget)
 message_layout.addWidget(message_display)

 return message_widget

def format_message(self, message):
 # Форматуємо повідомлення з Markdown
 if message.startswith("```") and message.endswith("```"):
 code = message[3:-3]
 return f"<pre style='background-color: #f8f8f8; border: 1px solid #ddd; padding: 10px; border-radius: 5px;'>{code}</pre>"
 else:
 return markdown(message)

def display_message(self, message):
 message_widget = self.create_message_widget(message)
 self.messages_layout.addWidget(message_widget)

def send_message(self):
 message = self.input_field.text()
 if message:
 self.display_message(f"Я: {self.format_message(message)}") # Відображаємо ваше повідомлення
 self.input_field.clear()
 # Приклад коду, що генерує бот
 self.display_message(f"Бот: {self.format_message(message)}")

def get_response(self, user_message):
 # Проста логіка для генерації відповіді
 return f"Це відповідь на ваше повідомлення: '{user_message}'"

if __name__ == "__main__":
 app = QApplication(sys.argv)
 window = ChatWindow()
 window.show()
 sys.exit(app.exec())
...

```py
from llama_index.legacy.text_splitter import TokenTextSplitter

# Ваш текст для поділу на фрагменти
text = "Це ваш текст, який потрібно розбити на блоки. " * 200 # Довгий текст для тестування

```

```

# Налаштування параметрів для поділу тексту
chunk_size = 512 # Розмір кожного блока
overlap = 50     # Кількість символів перекриття

token_text_splitter = TokenTextSplitter.from_defaults(chunk_size=512, chunk_overlap=50)
chunks = token_text_splitter.split_text(text)

# Виводимо отримані фрагменти
print(f"Кількість фрагментів: {len(chunks)}")
for i, chunk in enumerate(chunks[:5]): # Виведемо перші 5 фрагментів для прикладу
    print(f"Фрагмент {i+1}: \n{chunk}\n")
...

```py
from PySide6.QtWidgets import QApplication, QMainWindow, QListWidget, QListWidgetItem
from PySide6.QtCore import Qt
import sys

class ChatWindow(QMainWindow):
 def __init__(self):
 super().__init__()

 self.setWindowTitle("Chat Example")
 self.setGeometry(100, 100, 600, 400)

 # Створюємо QListWidget
 self.chat_list = QListWidget(self)
 self.setCentralWidget(self.chat_list)

 # Додаємо елементи з ID
 self.add_chat_item("Chat 1", 1)
 self.add_chat_item("Chat 2", 2)
 self.add_chat_item("Chat 3", 3)

 # Підписуємося на сигнал itemClicked
 self.chat_list.itemClicked.connect(self.on_item_clicked)

 def add_chat_item(self, name, chat_id):
 item = QListWidgetItem(name)
 item.setData(Qt.UserRole, chat_id) # Зберігаємо ID в item
 self.chat_list.addItem(item)

 def on_item_clicked(self, item: QListWidgetItem):
 chat_id = item.data(Qt.UserRole) # Отримуємо ID з item
 print(f"Clicked on: {item.text()}, ID: {chat_id}") # Виводимо назву та ID в консоль

if __name__ == "__main__":
 app = QApplication(sys.argv)
 window = ChatWindow()
 window.show()
 sys.exit(app.exec())
...

```py
from PyQt6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
    QLabel, QLineEdit, QPushButton, QScrollArea
)
from PyQt6.QtGui import QColor, QFont
from PyQt6.Qsci import QsciScintilla, QsciLexerPython, QsciLexerMarkdown
import sys

```

```

import re

# Клас для вікна повідомлень
class MessageWidget(QWidget):
    def __init__(self, message, is_sender):
        super().__init__()
        layout = QVBoxLayout()

        # Створюємо QLabel з стилями
        self.label = QLabel()
        self.label.setWordWrap(True) # Перенос слів
        self.setText(message) # Встановлюємо текст

        layout.addWidget(self.label)
        self.setLayout(layout)

        # Задаємо стиль в залежності від того, чи це повідомлення від користувача
        if is_sender:
            self.setStyleSheet("background-color: #3498db; color: white; border-radius: 10px; padding: 10px;
margin-left: auto; margin-right: 0;")
        else:
            self.setStyleSheet("background-color: #95a5a6; color: black; border-radius: 10px; padding: 10px;
margin-left: 0; margin-right: auto;")

    def setText(self, message):
        # Форматуємо повідомлення, розділяючи текст і код
        parts = re.split(r'(`(.*)`)', message, flags=re.DOTALL)
        formatted_message = ""

        for i in range(len(parts)):
            if i % 3 == 0 and parts[i].strip():
                formatted_message += f"<p>{parts[i]}</p>" # Текстові частини
            elif i % 3 == 1 and parts[i]: # Код
                formatted_message += f"<pre style='background-color: #34495e; color: #ecf0f1; border-radius: 5px;
padding: 5px;'>{parts[i]}</pre>"

        self.label.setText(formatted_message)

# Головне вікно чату
class ChatWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("ChatGui")
        self.resize(800, 600)
        self.setStyleSheet("background-color: #7f8c8d;") # Сірий фон

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QVBoxLayout(main_widget)

        self.scroll_area = QScrollArea()
        self.scroll_area.setWidgetResizable(True)
        self.messages_container = QVBoxLayout()
        self.scroll_area.setWidget(QWidget())
        self.scroll_area.widget().setLayout(self.messages_container)

        main_layout.addWidget(self.scroll_area)

        input_layout = QHBoxLayout()
        self.input_field = QLineEdit()
        self.input_field.setPlaceholderText("Введіть своє повідомлення...")
        input_layout.addWidget(self.input_field)

```

```

send_button = QPushButton("Відправити")
send_button.clicked.connect(self.send_message)
input_layout.addWidget(send_button)

main_layout.addLayout(input_layout)

# Додати редактор коду
self.code_editor = QsciScintilla()
self.code_editor.setStyleSheet("background-color: #34495e; color: #ecf0f1;")
main_layout.addWidget(self.code_editor)

# Налаштування Markdown lexer
self.setup_markdown_lexer()
# Налаштування Python lexer
self.setup_python_lexer()

def setup_markdown_lexer(self):
    lexer = QsciLexerMarkdown()
    lexer.setDefaultColor(QColor("#FFFFFF"))
    self.code_editor.setLexer(lexer)
    lexer.setPaper(QColor("#2c3e50")) # Колір фону редактора
    lexer.setColor(QColor("#808080"), lexer.Header1)
    lexer.setColor(QColor("#FFA500"), lexer.Header2)
    lexer.setColor(QColor("#ffffff"), lexer.Header3)
    lexer.setFont(QFont("Courier", 10)) # Шрифт для редактора

def setup_python_lexer(self):
    lexer = QsciLexerPython()
    lexer.setDefaultColor(QColor("#FFFFFF"))
    self.code_editor.setLexer(lexer)
    lexer.setPaper(QColor("#2c3e50")) # Колір фону редактора
    lexer.setColor(QColor("#808080"), lexer.Comment)
    lexer.setColor(QColor("#FFA500"), lexer.Keyword)
    lexer.setColor(QColor("#FFFFFF"), lexer.ClassName)
    lexer.setColor(QColor("#59ff00"), lexer.TripleSingleQuotedString)
    lexer.setColor(QColor("#59ff00"), lexer.TripleDoubleQuotedString)
    lexer.setColor(QColor("#3ba800"), lexer.SingleQuotedString)
    lexer.setColor(QColor("#3ba800"), lexer.DoubleQuotedString)
    lexer.setFont(QFont("Courier", 10)) # Шрифт для редактора

def display_message(self, message, is_sender=True):
    message_widget = MessageWidget(message, is_sender)
    self.messages_container.addWidget(message_widget)
    self.scroll_area.verticalScrollBar().setValue(self.scroll_area.verticalScrollBar().maximum()) #
Прокручування вниз

def send_message(self):
    message = self.input_field.text()
    if message:
        self.display_message(f"Я: {message}", is_sender=True) # Відображаємо ваше повідомлення
        self.input_field.clear()

    # Логіка відповіді бота
    bot_response = message
    self.display_message(bot_response, is_sender=False) # Відображаємо відповідь бота

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = ChatWindow()
    window.show()
    sys.exit(app.exec())

```

```

```.py
import sys
from PyQt6.QtWidgets import (
 QApplication, QWidget, QHBoxLayout, QVBoxLayout,
 QPushButton, QListWidget, QLabel, QFormLayout,
 QLineEdit, QListWidgetItem, QStackedWidget,
 QTableWidgetItem, QTableWidgetItem, QSizePolicy,
 QGridLayout
)
from PyQt6.QtCore import Qt

class Model:
 global_id = 0
 def __init__(
 self,
 name="",
 context_size=4096,
 temperature= 0.3,
 top_k=40,
 top_p=0.95,
 select_status=False
):
 model_id = Model.global_id
 Model.global_id += 1
 self.id = model_id
 self.name = f"Model {self.id}" if name == "" else name
 self.context_size = context_size
 self.temperature = temperature
 self.top_k = top_k
 self.top_p = top_p
 self.select_status = select_status

 def inverse_select(self):
 self.select_status = not self.select_status

class RAGDBWidget(QWidget):
 def __init__(self, model:Model, select_callback, parent=None):
 super().__init__(parent)
 self.model = model
 self.select_callback = select_callback
 self.initUI()

 def initUI(self):
 layout = QFormLayout()
 layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

 self.name_edit = QLineEdit(self.model.name)
 layout.addRow(QLabel("Model:"), self.name_edit)

 self.context_size_edit = QLineEdit(str(self.model.context_size))
 layout.addRow(QLabel("Context Size:"), self.context_size_edit)

 self.temperature_edit = QLineEdit(str(self.model.temperature))
 layout.addRow(QLabel("Temperature:"), self.temperature_edit)

 self.top_k_edit = QLineEdit(str(self.model.top_k))
 layout.addRow(QLabel("Top K:"), self.top_k_edit)

 self.top_p_edit = QLineEdit(str(self.model.top_p))
 layout.addRow(QLabel("Top P:"), self.top_p_edit)

```

```

if self.model.select_status is False:
 self.select_button = QPushButton("Select")
else:
 self.select_button = QPushButton("Deselect")

self.select_button.clicked.connect(self.onSelect)
layout.addRow(self.select_button)

self.setStyleSheet("""
 QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 2px solid #4A90E2;
 border-radius: 8px;
 padding: 10px;
 margin: 10px;
 }
 QLineEdit {
 background-color: #4C4C4C;
 border: 1px solid #5C5C5C;
 padding: 5px;
 color: #FFFFFF;
 border-radius: 3px;
 margin: 5px 0;
 }
 QLineEdit:focus {
 border: 1px solid #4A90E2;
 }
 QLabel {
 font-weight: bold;
 color: #FFFFFF;
 }
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 5px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
""")

self.setLayout(layout)

def onSelect(self):
 self.model.inverse_select()
 self.select_callback(self.model)

class CustomListWidgetItem(QListWidgetItem):
 def __init__(self, model_widget):
 super().__init__()
 self.setSizeHint(model_widget.sizeHint())
 self.setData(Qt.ItemDataRole.UserRole, model_widget.model)

class RagWidget(QWidget):
 def __init__(self):
 super().__init__()
 self.models = []

```

```

self.selected_model = None
self.initUI()

def initUI(self):
 main_layout = QVBoxLayout()

 # QStackedWidget for the tabs
 self.stacked_widget = QStackedWidget()
 self.stacked_widget.addWidget(self.createSelectBaseTab())
 self.stacked_widget.addWidget(self.createModelsTab())
 self.stacked_widget.addWidget(self.createDocumentsTab())

 # Buttons for tabs
 tab_buttons_layout = QHBoxLayout()
 self.select_base_button = QPushButton("Select RAG Base")
 self.models_button = QPushButton("Models")
 self.documents_button = QPushButton("Documents")

 self.select_base_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(0))
 self.models_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(1))
 self.documents_button.clicked.connect(lambda: self.stacked_widget.setCurrentIndex(2))

 tab_buttons_layout.addWidget(self.select_base_button)
 tab_buttons_layout.addWidget(self.models_button)
 tab_buttons_layout.addWidget(self.documents_button)

 main_layout.addLayout(tab_buttons_layout)
 main_layout.addWidget(self.stacked_widget)

 self.setStyleSheet("background-color: #2E2E2E; color: #FFFFFF;")
 self.setLayout(main_layout)

def createSelectBaseTab(self):
 tab = QWidget()
 layout = QVBoxLayout()

 self.add_button = QPushButton("Add Model")
 self.add_button.setStyleSheet("""
 QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 10px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """)
 self.add_button.clicked.connect(self.addModel)
 layout.addWidget(self.add_button)

 self.left_list = QListWidget()
 self.left_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")
 layout.addWidget(self.left_list)

 tab.setLayout(layout)
 return tab

def createModelsTab(self):
 tab = QWidget()
 layout = QVBoxLayout()

```

```

self.right_list = QListWidget()
self.right_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px;
}")
layout.addWidget(self.right_list)

tab.setLayout(layout)
return tab

def createDocumentsTab(self):
 tab = QWidget()

 # Використовуємо QGridLayout
 layout = QGridLayout()

 # Створення таблиці
 self.documents_table = QTableWidgetItem()
 self.documents_table.setRowCount(10) # Set number of rows
 self.documents_table.setColumnCount(3) # Set number of columns (for example: ID, Title, Content)

 self.documents_table.setHorizontalHeaderLabels(["ID", "Title", "Content"])

 # Встановлення політики для розміру таблиці, щоб вона могла заповнити весь доступний простір
 self.documents_table.setSizePolicy(QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Expanding)

 # Додаємо таблицю в layout на позицію (0, 0) з можливістю розтягуватись на весь доступний
 простір
 layout.addWidget(self.documents_table, 0, 0, 1, 1) # (row, column, rowSpan, columnSpan)

 # Додатково задаємо пропорції, щоб сітка заповнювала весь доступний простір
 layout.setRowStretch(0, 1) # Це дозволяє першому ряду розтягуватися
 layout.setColumnStretch(0, 1) # Це дозволяє першому стовпцю розтягуватися

 # Встановлення лейауту для вкладки
 tab.setLayout(layout)
 return tab

def addModel(self):
 model = Model()
 self.models.append(model)

 model_widget = RAGDBWidget(model, self.selectModel)
 item = CustomListWidgetItem(model_widget)
 self.left_list.addItem(item)
 self.left_list.setItemWidget(item, model_widget)

def selectModel(self, model):
 if self.selected_model:
 self.moveModel(self.selected_model, self.right_list, self.left_list)
 self.selected_model = None
 else:
 self.selected_model = model
 self.moveModel(model, self.left_list, self.right_list)

def moveModel(self, model: Model, source_list: QListWidget, target_list: QListWidget):
 for index in range(source_list.count()):
 item = source_list.item(index)
 widget: RAGDBWidget = source_list.itemWidget(item)
 widget_model = widget.model
 if widget_model.id == model.id:
 model_widget = RAGDBWidget(model, self.selectModel)
 item_widget = CustomListWidgetItem(model_widget)
 target_list.addItem(item_widget)
 target_list.setItemWidget(item_widget, model_widget)

```

```

 source_list.takeItem(index)
 break

if __name__ == "__main__":
 app = QApplication(sys.argv)
 app.setStyle("Fusion")
 window = RagWidget()
 window.setWindowTitle("RAGWidget")
 window.resize(600, 400)
 window.show()
 sys.exit(app.exec())
'''

''' .py
from PyQt6.QtWidgets import (
 QWidget, QVBoxLayout, QComboBox, QTableWidgetItem, QTableWidgetItem,
 QSizePolicy, QHBoxLayout, QPushButton, QLabel, QLineEdit, QApplication
)
from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QHeaderView
import sys

class DocumentViewer(QWidget):
 def __init__(self, documents_set):
 super().__init__()

 # Набір масивів документів
 self.documents_set = documents_set
 self.documents = documents_set["Category 1"] # Вибір початкового набору документів
 self.rows_per_page = 10 # Початкове значення для кількості рядків
 self.current_page = 0
 self.total_pages = 1 # Ініціалізація, щоб уникнути помилок

 self.initUI() # Ініціалізація інтерфейсу
 self.calculateTotalPages() # Після створення total_pages_label
 self.loadPage(self.current_page) # Завантаження першої сторінки

 def initUI(self):
 layout = QVBoxLayout()

 # Комбобокс
 self.combo = QComboBox()
 self.combo.addItem(self.documents_set.keys()) # Додаємо категорії
 self.combo.currentTextChanged.connect(self.updateDocuments) # Підключаємо зміну категорії
 layout.addWidget(self.combo)

 # Створення таблиці
 self.documents_table = QTableWidgetItem()
 self.documents_table.setColumnCount(3)
 self.documents_table.setHorizontalHeaderLabels(["ID", "Title", "Content"])
 self.documents_table.setSizePolicy(QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Expanding)

 # Розтягування Content на всю ширину
 header = self.documents_table.horizontalHeader()
 header.setSectionResizeMode(0, QHeaderView.ResizeMode.ResizeToContents) # ID стовпець
 header.setSectionResizeMode(1, QHeaderView.ResizeMode.ResizeToContents) # Title стовпець
 header.setSectionResizeMode(2, QHeaderView.ResizeMode.Stretch) # Content стовпець
 розтягується

 layout.addWidget(self.documents_table)

 # Створення навігаційних елементів

```

```

navigation_layout = QHBoxLayout()

self.prev_button = QPushButton("Prev")
self.prev_button.clicked.connect(self.prevPage)
navigation_layout.addWidget(self.prev_button)

Додавання індикатора сторінки та поля введення
self.info_label = QLabel(f"Current page:")
navigation_layout.addWidget(self.info_label)

self.page_input = QLineEdit(str(self.current_page + 1))
self.page_input.setFixedWidth(40)
self.page_input.setAlignment(Qt.AlignmentFlag.AlignCenter)
self.page_input.setPlaceholderText("Page")
self.page_input.returnPressed.connect(self.goToPage) # Переход до введеної сторінки
navigation_layout.addWidget(self.page_input)

Відображення загальної кількості сторінок
self.total_pages_label = QLabel(f"/ {self.total_pages}") # Ініціалізація total_pages_label
navigation_layout.addWidget(self.total_pages_label)

self.next_button = QPushButton("Next")
self.next_button.clicked.connect(self.nextPage)
navigation_layout.addWidget(self.next_button)

layout.addLayout(navigation_layout)
self.setLayout(layout)

def resizeEvent(self, event):
 # Викликається при зміні розміру вікна
 row_height = self.documents_table.verticalHeader().defaultSectionSize()
 available_height = self.documents_table.viewport().height()

 # Визначаємо скільки рядків вміщується у видиму область таблиці
 new_rows_per_page = available_height // row_height

 if new_rows_per_page != self.rows_per_page:
 self.rows_per_page = new_rows_per_page
 self.calculateTotalPages()
 self.loadPage(self.current_page)

 super().resizeEvent(event)

def calculateTotalPages(self):
 # Перераховуємо загальну кількість сторінок
 self.total_pages = (len(self.documents) + self.rows_per_page - 1) // self.rows_per_page
 self.total_pages_label.setText(f"/ {self.total_pages}")

def updateDocuments(self, category):
 # Оновлюємо documents на основі обраної категорії
 self.documents = self.documents_set[category]

 # Перезапуск пагінації та завантаження першої сторінки
 self.current_page = 0
 self.calculateTotalPages()
 self.page_input.setText(str(self.current_page + 1))
 self.loadPage(self.current_page)

def loadPage(self, page_number):
 # Очищення таблиці
 self.documents_table.clearContents()
 self.documents_table.setRowCount(self.rows_per_page) # Налаштовуємо кількість рядків

```

```

Розрахунок початкового індексу
start_index = page_number * self.rows_per_page
end_index = min(start_index + self.rows_per_page, len(self.documents))

Заповнення таблиці даними
for row, document in enumerate(self.documents[start_index:end_index]):
 self.documents_table.setItem(row, 0, QTableWidgetItem(str(document["id"])))
 self.documents_table.setItem(row, 1, QTableWidgetItem(document["title"]))
 self.documents_table.setItem(row, 2, QTableWidgetItem(document["content"]))

Оновлення індикатора сторінки
self.info_label.setText(f"Current page:")
self.page_input.setText(str(self.current_page + 1))

Вимикання кнопок якщо немає сторінок в певному напрямку
self.prev_button.setEnabled(self.current_page > 0)
self.next_button.setEnabled(self.current_page < self.total_pages - 1)

def goToPage(self):
 # Отримуємо номер сторінки з введення
 try:
 page = int(self.page_input.text()) - 1
 if 0 <= page < self.total_pages:
 self.current_page = page
 self.loadPage(self.current_page)
 else:
 # Якщо введена сторінка за межами допустимого діапазону
 self.page_input.setText(str(self.current_page + 1))
 except ValueError:
 # Якщо введено некоректне значення
 self.page_input.setText(str(self.current_page + 1))

def nextPage(self):
 if self.current_page < self.total_pages - 1:
 self.current_page += 1
 self.loadPage(self.current_page)

def prevPage(self):
 if self.current_page > 0:
 self.current_page -= 1
 self.loadPage(self.current_page)

Приклад різних категорій документів
documents_set = {
 "Category 1": [{ "id": i, "title": f"Doc {i} in Category 1", "content": f"Content of document {i} in Category 1" } for i in range(1, 51)],
 "Category 2": [{ "id": i, "title": f"Doc {i} in Category 2", "content": f"Content of document {i} in Category 2" } for i in range(1, 101)],
 "Category 3": [{ "id": i, "title": f"Doc {i} in Category 3", "content": f"Content of document {i} in Category 3" } for i in range(1, 76)]
}

Запуск додатку
app = QApplication(sys.argv)
viewer = DocumentViewer(documents_set)
viewer.show()
sys.exit(app.exec())
...

```py
from PyQt6.QtWidgets import QApplication, QWidget, QComboBox, QVBoxLayout, QLabel
import sys

```

```

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()

        # Створюємо QComboBox
        self.combo = QComboBox(self)

        # Додаємо порожній елемент
        self.combo.addItem("Виберіть модель.") # Порожній рядок

        # Додаємо інші елементи
        self.combo.addItem("Option 1")
        self.combo.addItem("Option 2")
        self.combo.addItem("Option 3")

        # Створюємо лейбл для відображення поточного вибору
        self.label = QLabel("Вибрано: None", self)

        # Підключення сигналу до методу
        self.combo.currentIndexChanged.connect(self.on_index_changed)

        # Налаштовуємо макет
        layout = QVBoxLayout()
        layout.addWidget(self.combo)
        layout.addWidget(self.label)
        self.setLayout(layout)

        # Встановлюємо заголовок вікна
        self.setWindowTitle("QComboBox Empty State Tracking")

    def on_index_changed(self, index):
        # Перевіряємо, чи вибраний перший елемент (порожній)
        if index == 0:
            self.label.setText("Не вибрано")
        else:
            selected_text = self.combo.itemText(index)
            self.label.setText(f"Вибрано: {selected_text}")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec())
'''
'''py
import gradio as gr

philosophy_quotes = [
    ["I think therefore I am."],
    ["The unexamined life is not worth living."]
]

startup_quotes = [
    ["Ideas are easy. Implementation is hard"],
    ["Make mistakes faster."]
]

def show_startup_quotes():
    return gr.Dataset(samples=startup_quotes)

with gr.Blocks() as demo:

```

```

        textbox = gr.Textbox()
        dataset = gr.Dataset(components=[textbox], samples=philosophy_quotes)
        button = gr.Button()

        button.click(show_startup_quotes, None, dataset)

demo.launch()```

```py
from PyQt6 import QtWidgets, QtGui
from PyQt6.QtWidgets import QTextEdit, QPushButton, QVBoxLayout, QWidget
import sys
from pygments import highlight
from pygments.lexers import PythonLexer
from pygments.formatters import HtmlFormatter

class CodeEditor(QWidget):
 def __init__(self):
 super().__init__()

 # Ініціалізація елементів
 self.editor = QTextEdit(self)
 self.editor.setFont(QtGui.QFont("Courier", 10))

 # Кнопка форматування
 self.format_button = QPushButton("Форматувати код", self)
 self.format_button.clicked.connect(self.highlight_code)

 # Розташування
 layout = QVBoxLayout(self)
 layout.addWidget(self.editor)
 layout.addWidget(self.format_button)
 self.setLayout(layout)

 def highlight_code(self):

 # Отримання тексту з редактора
 code = self.editor.toPlainText()

 # Підсвічування коду
 formatter = HtmlFormatter()

 highlighted_code = highlight(code, PythonLexer(), formatter)

 # Оновлення редактора з підсвіченим кодом
 self.editor.setHtml(highlighted_code)

if __name__ == "__main__":
 app = QtWidgets.QApplication(sys.argv)
 window = CodeEditor()
 window.resize(800, 600)
 window.show()
 sys.exit(app.exec())
```

```py
import sys
from PyQt6.QtWidgets import (
 QApplication, QWidget, QHBoxLayout, QVBoxLayout, QPushButton, QListWidget, QLabel, QFormLayout,
 QLineEdit, QListWidgetItem
)

```

```
from PyQt6.QtCore import Qt
```

```
class Model:
 global_id = 0
 def __init__(
 self,
 name="",
 context_size=4096,
 temperature= 0.3,
 top_k=40,
 top_p=0.95,
 select_status=False
):
 model_id = Model.global_id
 Model.global_id += 1
 self.id = model_id
 self.name = f"Model {self.id}" if name == "" else name
 self.context_size = context_size
 self.temperature = temperature
 self.top_k = top_k
 self.top_p = top_p
 self.select_status = select_status

 def inverse_select(self):
 self.select_status = not self.select_status

class ModelWidget(QWidget):
 def __init__(self, model:Model, select_callback, parent=None):
 super().__init__(parent)
 self.model = model
 self.select_callback = select_callback
 self.initUI()

 def initUI(self):
 layout = QFormLayout()
 layout.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

 self.name_edit = QLineEdit(self.model.name)
 layout.addRow(QLabel("Model:"), self.name_edit)

 self.context_size_edit = QLineEdit(str(self.model.context_size))
 layout.addRow(QLabel("Context Size:"), self.context_size_edit)

 self.temperature_edit = QLineEdit(str(self.model.temperature))
 layout.addRow(QLabel("Temperature:"), self.temperature_edit)

 self.top_k_edit = QLineEdit(str(self.model.top_k))
 layout.addRow(QLabel("Top K:"), self.top_k_edit)

 self.top_p_edit = QLineEdit(str(self.model.top_p))
 layout.addRow(QLabel("Top P:"), self.top_p_edit)

 if self.model.select_status is False:
 self.select_button = QPushButton("Select")
 else:
 self.select_button = QPushButton("Deselect")

 self.select_button.clicked.connect(self.onSelect)
 layout.addRow(self.select_button)
```

```

Dark theme style with better spacing and framing
self.setStyleSheet("""
QWidget {
 background-color: #3C3C3C;
 color: #FFFFFF;
 border: 2px solid #4A90E2;
 border-radius: 8px;
 padding: 10px;
 margin: 10px; /* Add margin for spacing */
}
QLineEdit {
 background-color: #4C4C4C;
 border: 1px solid #5C5C5C;
 padding: 5px;
 color: #FFFFFF;
 border-radius: 3px;
 margin: 5px 0; /* Add margin between input fields */
}
QLineEdit:focus {
 border: 1px solid #4A90E2;
}
QLabel {
 font-weight: bold;
 color: #FFFFFF;
}
QPushButton {
 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 5px;
 border-radius: 5px;
}
QPushButton:hover {
 background-color: #357ABD;
}
""")

self.setLayout(layout)

def onSelect(self):
 self.model.inverse_select()
 self.select_callback(self.model)

class CustomListWidgetItem(QListWidgetItem):
 def __init__(self, model_widget):
 super().__init__()
 self.setSizeHint(model_widget.sizeHint())
 self.setData(Qt.ItemDataRole.UserRole, model_widget.model) # Store model data for retrieval

class ModelManager(QWidget):
 def __init__(self):
 super().__init__()
 self.models = []
 self.selected_model = None # To track the currently selected model
 self.initUI()

 def initUI(self):
 main_layout = QVBoxLayout() # Changed to QVBoxLayout for vertical arrangement

 self.add_button = QPushButton("Add Model")
 self.add_button.setStyleSheet("""
QPushButton {

```

```

 background-color: #4A90E2;
 color: #FFFFFF;
 border: none;
 padding: 10px;
 border-radius: 5px;
 }
 QPushButton:hover {
 background-color: #357ABD;
 }
 """
self.add_button.clicked.connect(self.addModel)
main_layout.addWidget(self.add_button)

Create horizontal layout for the lists
lists_layout = QHBoxLayout()

self.left_list = QListWidget()
self.left_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px; }")

self.right_list = QListWidget()
self.right_list.setStyleSheet("QListWidget { background-color: #3C3C3C; color: #FFFFFF; margin: 5px;
}"))

lists_layout.addWidget(self.right_list)
lists_layout.addWidget(self.left_list)

main_layout.addLayout(lists_layout)

Dark theme style for the main window
self.setStyleSheet("background-color: #2E2E2E; color: #FFFFFF;")
self.setLayout(main_layout)

def addModel(self):
 model_id = len(self.models) + 1 # Unique ID for each model
 model = Model(model_id, f"Model {model_id}", 512, 1.0, 50, 0.95)
 self.models.append(model)

 model_widget = ModelWidget(model, self.selectModel)
 item = CustomListWidgetItem(model_widget)
 self.left_list.addItem(item)
 self.left_list.setItemWidget(item, model_widget)

 model_widget.setStyleSheet("border: 1px solid #5C5C5C; margin: 5px; padding: 5px;")

def selectModel(self, model):
 if self.selected_model:
 self.moveModel(self.selected_model, self.right_list, self.left_list)
 self.selected_model = None
 else:
 self.selected_model = model
 self.moveModel(model, self.left_list, self.right_list)

def moveModel(self, model:Model, source_list:QListWidget, target_list:QListWidget):
 # Find the model in the source list
 for index in range(source_list.count()):
 item = source_list.item(index)
 widget: ModelWidget = source_list.itemWidget(item)
 widget_model = widget.model
 if widget_model.id == model.id:
 # Move the item to the target list
 model_widget = ModelWidget(model, self.selectModel)
 item_widget = CustomListWidgetItem(model_widget)

```

```

target_list.addItem(item_widget)
target_list.setItemWidget(item_widget, model_widget)

Add border style to item widget
widget.setStyleSheet("border: 1px solid #5C5C5C; margin: 5px; padding: 5px;")

source_list.takeItem(index) # Remove from the source list
break

if __name__ == "__main__":
 app = QApplication(sys.argv)
 app.setStyle("Fusion") # Set the Fusion style for a modern look
 window = ModelManager()
 window.setWindowTitle("Model Manager")
 window.resize(600, 400)
 window.show()
 sys.exit(app.exec())

```