Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього рівня бакалавр

Бакалавр
(назва освітнього ступеня)

на тему: Розробка програмного забезпечення та тестування AI-агента для інтерактивного консультування під час вивчення мови програмування Python

Виконав(ла): студент(ка) 4 курсу, групи СП-41
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

(підпис) (прізвище та ініціали)

Керівник (підпис) Багрій-Заяць О.А. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю.М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М.Р. (прізвище та ініціали)

Рецензент (підпис) Яцишин В.В. (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М.Р.
(підпис) (прізвище та ініціали)
« » 2026 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Хоренку Владиславу Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програмного забезпечення та тестування AI-агента для інтерактивного консультування під час вивчення мови програмування Python.

Керівник роботи Багрій-Заяць Оксана Андріївна, канд. техн. наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 20___ року № _____

2. Термін подання студентом завершеної роботи _____

3. Вихідні дані до роботи Потреба замовника, вимога замовника

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
Аналіз предметної області та постановка задачі
Теоретичні основи та обґрунтування вибору технологій
Реалізація AI-агента для консультування з Python на основі RAG та Telegram-бота
Безпека життєдіяльності та основи охорони праці
Висновки

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності та основи охорони праці			
Нормоконтроль			

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	**Примітка**
1	Ознайомлення з завданням кваліфікаційної роботи	27.01-02.02.26	Виконано
2	Збір та аналіз інформації за темою дослідження (огляд	18.03.2026	Виконано
	існуючих AI-агентів, чат-ботів та RAG-систем)		
3	Формування структури пояснювальної записки	20.03.2026	Виконано
4	Розробка технічного завдання та вибір методів реалізації	11.04.2026	Виконано
5	Реалізація AI-агента для консультування:	19.04.2026	Виконано
	розробка RAG-конвеєра, серверної частини FastAPI,		
	інтеграція з Qdrant, Ollama та Telegram Bot API		
6	Тестування функціоналу AI-агента та	25.04.2026	Виконано
	аналіз якості відповідей на тестових запитах		
7	Написання розділів пояснювальної записки (1–3 розділи)	28.04.2026	Виконано
8	Написання розділу 4: «Безпека життєдіяльності та	29.04.2026	Виконано
	основи охорони праці»		
9	Оформлення висновків, списку використаних джерел,	30.04.2026	Виконано
	Додатків		
10	Перевірка роботи керівником, внесення правок	24.05.2026	Виконано
11	Нормоконтроль	24.05.2026	Виконано
12	Перевірка кваліфікаційної роботи на плагіат	25.05.2026	Виконано
13	Попередній захист кваліфікаційної роботи		Виконано
14	Захист кваліфікаційної роботи		Виконано

Студент _____ *Хоренко В.М.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Багрій-Заяць О.А.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка програмного забезпечення та тестування AI-агента для інтерактивного консультування під час вивчення мови програмування Python // Кваліфікаційна робота освітнього рівня «Бакалавр» // Хоренко Владислав Миколайович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-41 // Тернопіль, 2026 // С. 78, рис. – 7, табл. – 5, кресл. – 0, додат. – 3, бібліогр. – 16.

Ключові слова: AI-агент, штучний інтелект, чат-бот, Telegram, Python, RAG, Retrieval-Augmented Generation, векторний пошук, Qdrant, ембедінги, мовна модель, Gemma, Ollama, FastAPI, SQLite, інтерактивне консультування, обробка природної мови.

У даній кваліфікаційній роботі на здобуття освітнього ступеня бакалавра розглянуто процес розробки та інтеграції інтелектуального AI-агента для автоматизованого консультування користувачів щодо вивчення мови програмування Python у режимі реального часу. Сучасний процес вивчення мов програмування вимагає оперативного доступу до якісних пояснень та прикладів коду. Традиційні методи пошуку інформації через веб-браузери та форуми часто є неефективними через необхідність самостійної фільтрації великого обсягу даних. Особливої актуальності набуває впровадження інтелектуальних діалогових систем, здатних автоматично знаходити релевантну інформацію з офіційних джерел та формувати зрозумілі відповіді з урахуванням контексту попереднього діалогу. Метою роботи є розробка повнофункціональної програмної системи на основі технології Retrieval-Augmented Generation (RAG), яка забезпечує інтерактивне консультування з питань мови програмування Python через зручний інтерфейс Telegram-бота з використанням офіційної документації як джерела знань. У межах цієї роботи розглянуто процес створення AI-агента з інтеграцією технологій векторного пошуку на базі Qdrant, ембедінг-моделі sentence-transformers/all-MiniLM-L6-v2 та мовної моделі

Gemma3:27b. Реалізовано повний RAG-конвеєр для автоматичного пошуку контексту з офіційної документації Python, підтримку історії діалогу через SQLite та зручний інтерфейс Telegram-бота для взаємодії з користувачами.

ABSTRACT

Software Development and Testing of an AI Agent for Interactive Consulting During Python Programming Language Learning // Bachelor's degree qualification work // Khorenko Vladyslav Mykolaiovych // Ivan Pul'uj Ternopil National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-41 // Ternopil, 2026 // P. 78, fig. – 7, tabl. – 5, blueprints – 0, add. – 3, ref. – 16.

Keywords: AI agent, artificial intelligence, chatbot, Telegram, Python, RAG, Retrieval-Augmented Generation, vector search, Qdrant, embeddings, language model, Gemma, Ollama, FastAPI, SQLite, interactive consulting, natural language processing.

This bachelor's degree qualification work examines the process of developing and integrating an intelligent AI agent for automated user consulting on Python programming language learning in real time. The modern process of learning programming languages requires prompt access to quality explanations and code examples. Traditional methods of searching for information through web browsers and forums are often inefficient due to the need for manual filtering of large volumes of data. The implementation of intelligent dialogue systems capable of automatically finding relevant information from official sources and forming clear answers taking into account the context of the previous dialogue is becoming particularly relevant. The objective of this work is to develop a fully functional software system based on Retrieval-Augmented Generation (RAG) technology that provides interactive consulting on Python programming language topics through a convenient Telegram bot interface using official documentation as a knowledge source. This work examines the process of creating an AI agent with the integration of vector search technologies based on Qdrant, the embedding model sentence-transformers/all-MiniLM-L6-v2, and the language model Gemma3:27b. A complete RAG pipeline for automatic context retrieval from the official Python documentation, dialogue history support through

SQLite, and a convenient Telegram bot interface for user interaction have been implemented.

ПЕРЕЛІК СКОРОЧЕНЬ

AI (Artificial Intelligence) — Штучний інтелект.

API (Application Programming Interface) — Прикладний програмний інтерфейс.

CRUD (Create, Read, Update, Delete) — Базові операції з даними.

JSON (JavaScript Object Notation) — Текстовий формат обміну даними.

LLM (Large Language Model) — Велика мовна модель.

NLP (Natural Language Processing) — Обробка природної мови.

RAG (Retrieval-Augmented Generation) — Генерація з доповненням пошуком.

REST (Representational State Transfer) — Архітектурний стиль для побудови веб-сервісів.

SQL (Structured Query Language) — Мова структурованих запитів.

UUID (Universally Unique Identifier) — Універсальний унікальний ідентифікатор.

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням попиту на фахівців у галузі програмування. Мова програмування Python залишається однією з найпопулярніших у світі завдяки своїй простоті, гнучкості та широкому спектру застосування. Щорічно мільйони людей розпочинають вивчення цієї мови, використовуючи офіційну документацію, онлайн-курси та різноманітні навчальні платформи. Водночас процес самостійного навчання супроводжується значними труднощами: початківці часто стикаються з необхідністю швидко знайти відповідь на конкретне питання, зрозуміти поведінку певної конструкції або отримати приклад коду для вирішення практичної задачі.

Традиційні підходи до пошуку навчальної інформації мають суттєві обмеження. Використання стандартних пошукових систем вимагає від користувача самостійного формулювання запитів, перегляду великої кількості результатів та фільтрації нерелевантного контенту. Офіційна документація Python[3], попри свою повноту та достовірність, може бути складною для сприйняття початківцями через академічний стиль викладення та відсутність адаптації під рівень підготовки конкретного користувача. Форуми та спільноти розробників, хоча й містять цінну практичну інформацію, потребують тривалого пошуку та не гарантують актуальності знайдених відповідей.

З огляду на це, розробка інтелектуальної діалогової системи на основі технології Retrieval-Augmented Generation для автоматизованого консультування під час вивчення мови програмування Python є актуальним та своєчасним завданням. Така система здатна суттєво спростити процес навчання, надаючи персоналізовані відповіді на запитання користувачів у форматі природного діалогу з використанням перевіреної інформації з офіційних джерел.

Метою кваліфікаційної роботи є створення повнофункціонального AI-агента для інтерактивного консультування під час вивчення мови програмування Python з використанням технології RAG та зручного інтерфейсу Telegram-бота.

Для досягнення поставленої мети було сформульовано та вирішено такі завдання: проведення аналізу предметної області та існуючих рішень у сфері інтелектуальних навчальних систем; проектування загальної архітектури системи з використанням векторної бази даних Qdrant та мовної моделі Gemma3:27b; розробка серверної частини на основі фреймворку FastAPI для забезпечення обробки запитів через RESTful API; реалізація RAG-конвеєра для автоматичного пошуку релевантного контексту з офіційної документації Python; створення модуля збереження історії діалогу для підтримки контекстних бесід; розробка та інтеграція клієнтського інтерфейсу у вигляді Telegram-бота; проведення тестування розробленої системи та оцінка якості генерованих відповідей.

Об'єкт дослідження: процес автоматизованого пошуку, аналізу та генерації відповідей на запитання користувачів щодо мови програмування Python з використанням технологій штучного інтелекту.

Предмет дослідження: методи, моделі штучного інтелекту та програмні засоби розробки інтелектуального AI-агента з використанням технології RAG, векторного пошуку та мовних моделей.

Методи дослідження. У процесі виконання роботи використовувались методи системного аналізу для проектування архітектури програмного комплексу. Застосовувались методи об'єктно-орієнтованого програмування для розробки серверної частини. Для перетворення текстової інформації у векторні представлення використовувались методи обробки природної мови та трансформерні моделі. Для забезпечення якісної генерації відповідей залучено методи інженерії підказок та контекстного доповнення[16].

Практичне значення одержаних результатів. Розроблений AI-агент є повністю готовим до використання програмним продуктом. Система здатна приймати запитання користувачів через Telegram-бот, автоматично знаходити релевантну інформацію з офіційної документації Python та генерувати зрозумілі відповіді з прикладами коду у форматі природного діалогу. Впровадження цієї розробки у навчальний процес дозволить суттєво підвищити ефективність

самостійного вивчення мови програмування Python, забезпечивши цілодобовий доступ до якісних консультацій.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Розробка будь-якої програмної системи починається з ґрунтовного аналізу предметної області, що дозволяє виявити реальні потреби користувачів та обґрунтувати доцільність обраних технологічних рішень. У цьому розділі досліджено сучасний стан інструментів підтримки навчання програмуванню, проаналізовано наявні системи інтерактивного консультування та виявлено їхні ключові обмеження. Окрему увагу приділено методам побудови діалогових систем на основі штучного інтелекту та технології Retrieval-Augmented Generation, які формують теоретичне підґрунтя роботи. За результатами аналізу сформульовано чітку постановку задачі на розробку програмного забезпечення.

1.1 Аналіз потреб у інтерактивних інструментах під час вивчення мови програмування Python

Мова програмування Python протягом останнього десятиліття стабільно утримує позиції одної з найпопулярніших мов у світі згідно з рейтингами TIOBE та Stack Overflow Developer Survey. Її широке застосування охоплює веб-розробку, аналіз даних, машинне навчання, автоматизацію процесів та наукові обчислення. Щорічно мільйони студентів, початківців та професіоналів з різних галузей починають або продовжують вивчення Python, що створює безпрецедентний попит на якісні навчальні ресурси та інструменти підтримки навчального процесу.

Процес вивчення будь-якої мови програмування неминуче супроводжується виникненням великої кількості запитань різного рівня складності. Початківці потребують допомоги у розумінні базових концепцій синтаксису, роботі з типами даних, використанні циклів та умовних конструкцій. Більш досвідчені користувачі шукають відповіді на питання про роботу з бібліотеками, обробку винятків, використання генераторів та декораторів. Незалежно від рівня підготовки, спільною потребою залишається оперативний

доступ до достовірної та зрозумілої інформації, яка допомагає вирішити конкретну практичну задачу.

Офіційна документація Python є найбільш авторитетним джерелом інформації про мову. Вона містить вичерпні описи всіх вбудованих функцій, стандартних бібліотек, синтаксичних конструкцій та рекомендованих практик програмування. Проте використання документації у її первісному вигляді має суттєві недоліки. По-перше, навігація по масивній базі знань вимагає від користувача розуміння загальної структури мови та здатності точно формулювати пошукові запити. По-друге, документація написана технічним стилем, який може бути складним для сприйняття початківцями. По-третє, відсутність інтерактивності означає, що користувач не може уточнити незрозумілі моменти або отримати додаткові пояснення без повторного пошуку.

Альтернативними джерелами інформації виступають різноманітні онлайн-платформи та спільноти розробників. Stack Overflow містить мільйони питань та відповідей з програмування, але пошук потрібної інформації серед дискусій різної якості та актуальності вимагає значних часових витрат. Відеокурси на платформах Coursera та Udemy забезпечують структуроване навчання, але не пристосовані для швидкого отримання відповіді на конкретне питання. Підручники та навчальні посібники пропонують систематизований виклад матеріалу, але їх використання для оперативного пошуку інформації є неефективним через лінійну структуру подачі знань.

Варто зазначити, що сучасні студенти та розробники все більше надають перевагу діалоговому формату отримання інформації. Зростання популярності месенджерів та чат-ботів сформувало нову парадигму взаємодії з інформаційними системами. Користувачі очікують отримати релевантну відповідь на природну мову запиту без необхідності адаптації своїх питань під формат пошукової системи. Крім того, важливою потребою є підтримка контексту розмови, що дозволяє ставити уточнювальні питання та поступово поглиблювати розуміння певної теми.

Таким чином, наявна модель інформаційної підтримки у процесі вивчення мови програмування Python потребує модернізації та залучення інноваційних підходів. Виникає об'єктивна необхідність у створенні інтелектуальної діалогової системи, яка поєднає переваги офіційної документації з інтерактивністю сучасних месенджерів. Така система повинна автоматично знаходити найбільш релевантну інформацію з офіційних джерел та подавати її у зрозумілому форматі з урахуванням контексту запиту та рівня підготовки користувача.

1.2 Огляд існуючих систем інтерактивного консультування та їхні обмеження

Історичний розвиток систем автоматизованого консультування у сфері програмування демонструє фундаментальний перехід від примітивних текстових помічників до складних когнітивних систем на базі великих мовних моделей. Перші спроби автоматизації навчальних консультацій обмежувалися створенням простих баз знань у форматі питань та відповідей. Такі системи працювали на основі ключових слів та не могли обробляти складні багатокомпонентні запити. Наступним еволюційним кроком стало впровадження інтелектуальних пошукових систем, які використовували алгоритми повнотекстового пошуку для знаходження релевантної інформації у великих масивах навчальних матеріалів. Найбільш досконалою категорією є сучасні діалогові системи на базі великих мовних моделей, здатні генерувати унікальні відповіді, адаптовані під конкретний контекст запиту користувача.

На глобальному ринку існує значна кількість готових рішень для автоматизованого консультування з питань програмування. Найбільш популярними серед них є комерційні продукти великих технологічних компаній. ChatGPT від компанії OpenAI здатний відповідати на широкий спектр питань з програмування, генерувати та пояснювати код. Проте цей сервіс має суттєвий недолік у вигляді схильності до генерації правдоподібних, але фактично

некоректних відповідей, що є критичним недоліком для навчальних систем. GitHub Copilot зосереджений переважно на автоматичному дописуванні коду всередині інтегрованого середовища розробки та не пристосований для повноцінного навчального консультування у діалоговому форматі.

Серед спеціалізованих навчальних платформ варто відзначити інтерактивні середовища Codecademy та Exercism, які пропонують структуровані курси з Python. Ці платформи забезпечують покрокове вивчення матеріалу з перевіркою знань, але їх інтерактивність обмежується заздалегідь підготовленими сценаріями. Користувач не може поставити довільне питання та отримати персоналізовану відповідь, що суттєво обмежує гнучкість навчального процесу. Такі системи ефективні для систематичного вивчення основ, але не задовольняють потребу у швидких точкових консультаціях під час реальної розробки програмного забезпечення.

Окремою категорією є чат-боти у месенджерах, створені ентузіастами та освітніми платформами. Більшість таких ботів працює за принципом обгорток над загальнодоступними API великих мовних моделей без додаткової спеціалізації на конкретній предметній області. Це означає, що якість їх відповідей повністю залежить від загальних знань мовної моделі, без прив'язки до конкретного джерела достовірної інформації. Такий підхід не забезпечує гарантії актуальності та достовірності наданих відповідей, що є критичним недоліком для навчального інструменту.

Для наочного представлення функціональних відмінностей між існуючими рішеннями розроблено порівняльну таблицю 1.1.

Таблиця 1.1 — Порівняльний аналіз існуючих систем консультування з програмування

Критерій	**ChatGPT**	**Copilot**	**Codecademy**	**Stack Overflow**	**Розроблена система**
Діалоговий формат	Так	Так	Ні	Ні	Так
Верифіковане джерело	Частково	Частково	Так	Частково	Так
Історія діалогу	Так	Так	Ні	Ні	Так
Безкоштовність	Частково	Частково	Частково	Так	Так
Telegram-інтерфейс	Ні	Ні	Ні	Ні	Так
Фокус на Python	Ні	Ні	Частково	Ні	Так

Як видно з таблиці, жодна з існуючих систем не забезпечує повного набору функціональних можливостей, необхідних для ефективного інтерактивного консультування з питань Python. Розроблена у межах цієї кваліфікаційної роботи система є унікальною за своєю комбінацією характеристик: діалоговий формат з підтримкою історії спілкування, використання верифікованого джерела знань у вигляді офіційної документації Python, повна безкоштовність та доступний інтерфейс через Telegram.

Головним технологічним бар'єром при створенні подібних систем є забезпечення достовірності генерованих відповідей. Загальні мовні моделі навчені на величезних масивах тексту з інтернету, що включає як правильну, так і застарілу або некоректну інформацію. Без механізму прив'язки генерації до конкретного верифікованого джерела знань будь-яка діалогова система залишається ненадійним інструментом для навчання. Саме тому у розробленій системі застосовано підхід Retrieval-Augmented Generation, який дозволяє обмежити генерацію відповідей контекстом офіційної документації Python, суттєво знижуючи ймовірність виникнення некоректних відповідей.

1.3 Дослідження методів побудови діалогових систем на базі штучного інтелекту

Створення сучасних діалогових систем для консультування вимагає застосування комплексного підходу до обробки природної мови та генерації текстових відповідей. Традиційні методи побудови чат-ботів базувалися на використанні правил та шаблонів, де розробник вручну визначав можливі запити та відповідні реакції системи. Такий підхід забезпечував високу точність відповідей у межах передбачених сценаріїв, але виявлявся абсолютно непридатним для обробки довільних запитів користувачів. Наступним еволюційним етапом стало впровадження методів машинного навчання, зокрема класифікаторів намірів та екстракторів іменованих сутностей, які дозволяли системі розпізнавати загальний зміст запиту та виділяти ключові елементи. Сучасний етап характеризується повним переходом до використання великих мовних моделей, здатних генерувати відповіді на довільні запити з урахуванням контексту діалогу.

Великі мовні моделі є фундаментальною технологією, на якій базується розроблена система. Ці нейронні мережі навчені на масивних текстових корпусах і здатні генерувати зв'язний текст, відповідати на питання, перекладати, узагальнювати інформацію та виконувати багато інших завдань обробки природної мови. У контексті кваліфікаційної роботи використовується мовна модель Gemma3 з 27 мільярдами параметрів, розроблена компанією Google[12]. Ця модель є відкритою і може бути розгорнута на локальному обладнанні без необхідності використання хмарних сервісів, що забезпечує повну незалежність від зовнішніх провайдерів та суттєво знижує витрати на експлуатацію системи.

Для локального розгортання мовних моделей у розробленій системі використовується платформа Ollama[5]. Цей інструмент забезпечує зручний інтерфейс для завантаження, налаштування та запуску різноманітних відкритих мовних моделей на персональному комп'ютері або сервері. Ollama надає стандартний програмний інтерфейс для взаємодії з мовною моделлю через

HTTP-запити, що суттєво спрощує інтеграцію у програмні комплекси. Завдяки оптимізованим алгоритмам квантування модель Gemma3:27b може працювати на обладнанні з відносно помірними вимогами до оперативної пам'яті та обчислювальних потужностей графічного прискорювача.

Важливим компонентом побудови інтелектуальних діалогових систем є технологія перетворення текстової інформації у векторні представлення, або ембедінги. Ця технологія дозволяє перетворити слова, речення або цілі фрагменти тексту у числові вектори у багатовимірному просторі, де семантично подібні тексти розташовуються близько один до одного. У розробленій системі використовується модель sentence-transformers/all-MiniLM-L6-v2[4], яка генерує ембедінги розмірністю 384 компоненти. Ця модель є компактною, швидкою та забезпечує високу якість семантичного пошуку для текстів англійською мовою, що ідеально відповідає потребам системи, оскільки офіційна документація Python написана переважно англійською.

Для ефективного зберігання та пошуку векторних представлень необхідно використовувати спеціалізовані бази даних, оптимізовані для операцій з багатовимірними векторами. У розробленій системі для цієї мети обрано Qdrant — сучасну високопродуктивну векторну базу даних з відкритим вихідним кодом. Qdrant підтримує різноманітні метрики відстані між векторами, включаючи косинусну подібність, яка використовується у даному проекті для порівняння семантичної близькості запитів користувачів та фрагментів документації. Додатковою перевагою Qdrant є підтримка фільтрації за метаданими, що дозволяє обмежувати пошук конкретними розділами документації або типами контенту.

1.4 Огляд технологій пошуку та генерації відповідей (RAG)

Технологія Retrieval-Augmented Generation є одним з найбільш перспективних підходів до побудови інтелектуальних систем генерації тексту, що поєднує переваги інформаційного пошуку та генеративних моделей.

Концепція RAG була запропонована дослідниками компанії Meta у 2020 році[6] та з того часу набула широкого визнання у академічній та промисловій спільнотах[13]. Основна ідея підходу полягає у доповненні стандартного процесу генерації тексту мовною моделлю етапом попереднього пошуку релевантної інформації із зовнішнього джерела знань. Це дозволяє суттєво підвищити точність та актуальність генерованих відповідей, мінімізуючи проблему так званих "галюцинацій" мовних моделей.

Архітектура RAG-конвеєра складається з кількох послідовних етапів обробки інформації. На першому етапі зовнішня база знань проходить процедуру попередньої обробки: документи розбиваються на невеликі фрагменти, кожен з яких перетворюється на векторне представлення за допомогою ембедінг-моделі та зберігається у спеціалізованій векторній базі даних. На другому етапі, коли до системи надходить запит користувача, він також перетворюється на вектор за допомогою тієї ж ембедінг-моделі. На третьому етапі виконується пошук найбільш семантично подібних фрагментів у векторній базі даних. На четвертому, завершальному етапі знайдені фрагменти разом із запитом користувача формують розширений промпт, який передається мовній моделі для генерації фінальної відповіді.

Рисунок 1.1 — Загальна схема роботи RAG-конвеєра

Ключовим етапом підготовки бази знань є процедура розбиття документів на фрагменти, яку в англомовній літературі називають chunking. Якість цього етапу безпосередньо впливає на точність подальшого пошуку. Надто великі фрагменти містять зайву інформацію, що знижує релевантність пошуку. Надто малі фрагменти втрачають контекст, що призводить до неповних відповідей. У розробленій системі використовується адаптивний алгоритм розбиття з параметрами максимальної довжини фрагмента 1000 символів та перекриття 150 символів між сусідніми фрагментами. Перекриття забезпечує збереження контексту на межах фрагментів, що є критично важливим для коректного семантичного пошуку.

Важливим аспектом реалізації RAG-системи є вибір метрики для порівняння векторних представлень. У розробленому проекті використовується косинусна подібність, яка вимірює кут між двома векторами у багатовимірному просторі. Ця метрика є інваріантною до довжини вектора, що означає, що вона оцінює виключно напрямок, а не амплітуду. Це робить косинусну подібність особливо ефективною для порівняння текстових ембедінгів, оскільки семантично подібні тексти мають вектори, що вказують у схожих напрямках, незалежно від їхньої довжини. Для забезпечення коректності роботи з косинусною подібністю ембедінг-модель у розробленій системі налаштована на генерацію нормалізованих векторів.

Для порівняння основних технологій, що використовуються у RAG-системах, розроблено таблицю 1.2.

Таблиця 1.2 — Порівняння технологій для побудови RAG-систем

Компонент	**Обрана технологія**	**Обґрунтування вибору**
Векторна база даних	Qdrant	Відкритий код, локальне розгортання, висока продуктивність
Ембедінг-модель	all-MiniLM-L6-v2	Компактність, швидкість, якість для англомовних текстів
Мовна модель	Gemma3:27b	Відкрита модель Google, локальне розгортання через Ollama
Серверний фреймворк	FastAPI	Асинхронність, автодокументація, висока продуктивність
Інтерфейс користувача	Telegram Bot API	Широка поширеність месенджера, зручний API
Збереження історії	SQLite	Легкість, вбудованість, відсутність залежностей

Окремою важливою складовою RAG-конвеєра є механізм формування промпту для мовної моделі. Після успішного пошуку релевантних фрагментів документації система повинна скласти чіткий та інформативний запит, який включає знайдений контекст, поточне питання користувача та історію попереднього діалогу. Якість формування промпту безпосередньо впливає на точність та зрозумілість генерованої відповіді. У розробленій системі промпт містить явні інструкції для мовної моделі щодо формату відповіді, необхідності використання прикладів коду та врахування контексту попередніх повідомлень.

1.5 Постановка задачі на розробку програмного забезпечення

На основі проведеного аналізу предметної області та детального огляду існуючих технологічних рішень виникає об'єктивна необхідність у формулюванні чіткої технічної задачі для виконання практичної частини кваліфікаційної роботи[14]. Головною метою проекту є розробка та

впровадження AI-агента для інтерактивного консультування під час вивчення мови програмування Python. Розроблена система повинна вирішити проблему неефективного пошуку навчальної інформації шляхом надання миттєвих та точних відповідей на запитання користувачів у форматі природного діалогу з використанням перевіреної інформації з офіційної документації Python.

Для досягнення поставленої мети необхідно виконати комплекс інженерних та алгоритмічних завдань. Першочерговим кроком є проектування загальної архітектури програмного комплексу, яка ефективно поєднає модулі індексації документації, векторного пошуку, генерації відповідей та клієнтського інтерфейсу. На рівні підготовки бази знань потрібно реалізувати парсинг та обробку офіційної документації Python, розбиття текстів на оптимальні фрагменти, генерацію векторних представлень за допомогою ембедінг-моделі та завантаження отриманих даних у векторну базу Qdrant.

Наступним критичним завданням є розробка серверного додатка на базі фреймворку FastAPI з реалізацією повного RAG-конвеєра. Цей додаток повинен приймати запити від клієнтського інтерфейсу, виконувати векторний пошук релевантних фрагментів документації, формувати розширений промпт з урахуванням контексту діалогу та генерувати відповідь за допомогою мовної моделі Gemma3:27b через платформу Ollama. Окремою технічною вимогою є реалізація системи збереження історії діалогу у базі даних SQLite для підтримки контекстних бесід.

Завершальним етапом розробки є створення клієнтського інтерфейсу у вигляді Telegram-бота. Вибір платформи Telegram обумовлений її надзвичайною популярністю серед цільової аудиторії та наявністю зручного Bot API для створення інтерактивних ботів. Telegram-бот повинен забезпечувати прийом текстових запитів від користувачів, їх пересилання до серверного API та відображення отриманих відповідей. Додатково бот повинен підтримувати набір команд для управління мовою відповідей, очищення історії діалогу та перегляду джерел використаних контекстів.

Готовий програмний продукт повинен пройти етап функціонального тестування для перевірки коректності роботи всіх компонентів системи, оцінки якості генерованих відповідей та підтвердження стабільності роботи під навантаженням. Результати тестування мають підтвердити здатність системи надавати достовірні та зрозумілі відповіді на запитання різного рівня складності з мови програмування Python.

2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГАМНОЇ СИСТЕМИ

Спираючись на результати аналізу предметної області, у цьому розділі описано процес проектування та програмної реалізації AI-агента. Спершу розглянуто загальну архітектуру програмного комплексу, побудовану за модульним принципом, яка поєднує підсистеми індексації, пошуку, генерації відповідей та взаємодії з користувачем. Далі послідовно висвітлено розробку кожного з компонентів: модуля векторного зберігання документації, серверної частини з RAG-конвеєром, інтеграції мовної моделі та налаштування логіки агента. Завершується розділ описом створення клієнтського інтерфейсу у вигляді Telegram-бота.

2.1 Проектування загальної архітектури програмного комплексу

Проектування архітектури є найважливішим етапом створення будь-якої складної інформаційної системи. Запропонований програмний комплекс побудований за модульним принципом і складається з п'яти основних компонентів, які функціонують у тісній взаємодії. Головною метою такої архітектури є забезпечення повного циклу обробки запитів користувачів: від отримання текстового повідомлення через Telegram до генерації релевантної відповіді на основі офіційної документації Python. Кожен модуль системи виконує свою вузькоспеціалізовану функцію, утворюючи єдиний неперервний конвеєр обробки даних. Такий чіткий розподіл відповідальності гарантує високу відмовостійкість усього комплексу, оскільки збій в одному модулі може бути локалізований та усунений без повної зупинки інших процесів.

Першою ланкою системи виступає модуль збору та підготовки даних. Офіційна документація мови Python у форматі JSON-файлів проходить процедуру парсингу та структурування. Кожен розділ документації розбивається на оптимальні текстові фрагменти з використанням адаптивного алгоритму чанкінгу. Ці фрагменти перетворюються на векторні представлення за допомогою ембедінг-моделі sentence-transformers/all-MiniLM-L6-v2 та

завантажуються у векторну базу даних Qdrant разом із метаданими, що включають назву розділу, URL джерела та структурні шляхи. Такий підхід дозволяє не лише шукати релевантну інформацію, але й надавати користувачу посилання на конкретні розділи офіційної документації для подальшого самостійного вивчення.

Другим ключовим компонентом є серверний додаток на базі фреймворку FastAPI. Цей модуль реалізує RESTfull API для прийому запитів від клієнтського інтерфейсу. При надходженні запиту сервер послідовно виконує кілька операцій: зберігає інформацію про користувача у базі даних, отримує історію останніх повідомлень для формування контексту діалогу, здійснює векторний пошук релевантних фрагментів документації, формує розширений промпт та передає його мовній моделі для генерації відповіді. Використання FastAPI забезпечує автоматичну генерацію інтерактивної документації API, валідацію вхідних даних через Pydantic-схеми та підтримку асинхронних операцій для ефективної обробки одночасних запитів.

Третім компонентом є модуль взаємодії з мовною моделлю Gemma3:27b через платформу Ollama. Цей модуль відповідає за форматування промптів згідно з визначеними шаблонами, передачу запитів до мовної моделі та обробку отриманих відповідей. Системний промпт визначає роль AI-агента як ментора з Python та встановлює правила генерації відповідей: враховувати контекст діалогу, пояснювати прості і зрозуміло, додавати приклади коду за потреби та використовувати інформацію з наданого контексту як основне джерело знань.

Четвертим модулем є підсистема збереження історії діалогу на основі SQLite[8]. Ця реляційна база даних зберігає інформацію про користувачів та їхні повідомлення, забезпечуючи можливість відновлення контексту бесіди при повторних зверненнях. Структура бази даних складається з двох таблиць: users для зберігання інформації про зареєстрованих користувачів та messages для послідовного збереження всіх повідомлень діалогу з позначками ролей відправника.

П'ятим та завершальним компонентом архітектури є Telegram-бот, який виступає єдиним інтерфейсом взаємодії з кінцевими користувачами. Бот приймає текстові повідомлення від студентів, формує запити до серверного API та відображає отримані відповіді у зручному форматі. Додатково бот підтримує набір команд для управління параметрами діалогу та перегляду додаткової інформації про використані джерела.

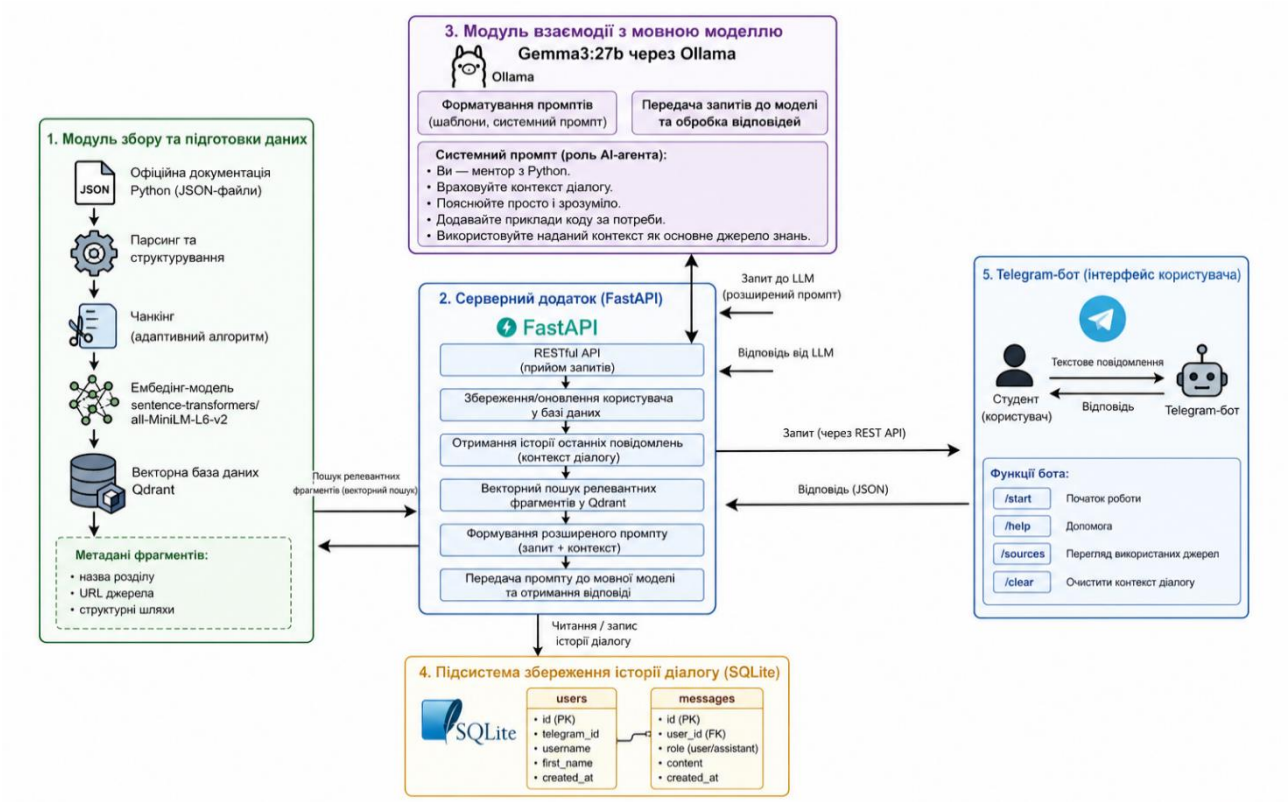

Рисунок 2.1 — Структурна схема взаємодії компонентів програмного комплексу

2.2 Розробка модуля індексації та векторного зберігання документації

Розробка модуля індексації є фундаментальним етапом побудови RAG-системи, від якості якого безпосередньо залежить точність пошуку та релевантність генерованих відповідей. Процес індексації складається з кількох послідовних кроків: завантаження структурованих даних документації з JSON-файлу, розбиття текстового контенту на оптимальні фрагменти, генерація

векторних представлень для кожного фрагменту та завантаження отриманих точок у колекцію векторної бази даних Qdrant.

Вхідні дані для індексації представлені у форматі JSON-файлу, який містить структуровану інформацію з офіційного туторіалу Python. Кожен елемент масиву відповідає окремому розділу документації та містить поля з текстовим вмістом, заголовком розділу, URL-адресою джерела, інформацією про наявність блоків коду та структурний шлях у ієрархії документації. Така детальна метаінформація зберігається разом із векторними представленнями, що дозволяє не лише знаходити релевантні фрагменти, але й надавати користувачу вичерпну контекстну інформацію про джерело знайденої відповіді.

Алгоритм розбиття текстів на фрагменти реалізований у модулі chunking.py. Основна функція chunk_text приймає текст та два параметри: максимальну довжину фрагменту та розмір перекриття між сусідніми фрагментами. Алгоритм працює з абзацами як базовими одиницями: текст спочатку розділяється по подвійному переносу рядка, після чого абзаци послідовно об'єднуються до досягнення максимальної довжини. При створенні нового фрагменту останні символи попереднього фрагменту додаються на початок нового для збереження контексту на межах розбиття. Нижче наведено ключовий фрагмент коду функції розбиття:

```python
def chunk_text(text: str, max_chars: int = 1000, overlap: int = 150) -> List[str]:
    text = (text or "").strip()
    if not text:
        return []

    paragraphs = [p.strip() for p in text.split("\n\n") if p.strip()]
    chunks: list[str] = []
    current = ""

    for para in paragraphs:
        if not current:
            current = para
            continue

        if len(current) + 2 + len(para) <= max_chars:
            current += "\n\n" + para
        else:
            chunks.append(current)

            if overlap > 0 and len(current) > overlap:
                tail = current[-overlap:]
                current = tail + "\n\n" + para
            else:
                current = para

    if current:
        chunks.append(current)

    return chunks
```

Рисунок 2.2 – фрагмент коду функції розбиття

Генерація векторних представлень реалізована у модулі embeddings.py. Для забезпечення ефективності використано механізм кешування завантаженої моделі за допомогою декоратора lru_cache, що запобігає повторному завантаженню моделі[10] в оперативну пам'ять при обробці кожного нового запиту. Модель sentence-transformers/all-MiniLM-L6-v2 генерує нормалізовані вектори розмірністю 384 компоненти, що забезпечує коректну роботу з косинусною метрикою подібності у Qdrant. Модуль надає дві основні функції: embed_text для векторизації одного тексту та embed_texts для пакетної обробки масиву текстів.

Модуль vector_store.py відповідає за управління з'єднанням з векторною базою даних Qdrant та конфігурацію колекцій. Під час ініціалізації системи виконується створення колекції python_docs_chunks з параметрами розмірності вектора 384 та метрикою відстані COSINE. Додатково створюються індекси за полями payload: source, language, number та has_code. Ці індекси дозволяють ефективно фільтрувати результати пошуку за джерелом, мовою, номером

розділу та наявністю прикладів коду, що підвищує релевантність знайдених фрагментів для конкретних типів запитів.

Головний модуль індексації indexer.py координує весь процес підготовки бази знань. Функція index_documents послідовно завантажує дані з JSON-файлу, створює колекцію у Qdrant, викликає функцію prepare_points для формування масиву точок із векторами та метаданими, після чого завантажує ці точки у базу даних пакетами по 128 елементів. Пакетне завантаження дозволяє ефективно обробляти великі масиви даних, контролюючи використання оперативної пам'яті та забезпечуючи можливість моніторингу прогресу індексації.

2.3 Реалізація серверної частини та RAG-конвеєра

Серверна частина розробленого програмного комплексу реалізована мовою програмування Python з використанням сучасного асинхронного фреймворку FastAPI[1]. Вибір цієї технології зумовлений кількома ключовими перевагами: вбудована підтримка асинхронних операцій для ефективної обробки паралельних запитів, автоматична генерація OpenAPI-документації, валідація даних через Pydantic-моделі та висока продуктивність завдяки використанню ASGI-сервера Uvicorn. Серверний додаток надає RESTful API з двома основними ендпоінтами: перевірки працездатності сервісу та обробки запитів користувачів.

Центральним елементом серверної архітектури є RAG-конвеєр, реалізований у модулі rag_pipeline.py. Цей модуль координує взаємодію між усіма компонентами системи та забезпечує повний цикл обробки запиту від моменту отримання питання до формування фінальної відповіді. Функція answer_question є точкою входу для обробки кожного запиту і виконує наступну послідовність операцій:

```python
def answer_question(
    user_query: str,
    telegram_user_id: int,
    telegram_username: str | None = None,
    first_name: str | None = None,
    top_k: int = 5,
    language: str | None = "en",
    model_name: str | None = None,
) -> dict[str, Any]:
    upsert_user(
        telegram_user_id=telegram_user_id,
        telegram_username=telegram_username,
        first_name=first_name,
    )

    history = get_last_messages(telegram_user_id, limit=5)

    contexts = retrieve_contexts(
        query=user_query,
        top_k=top_k,
        language=language,
    )

    prompt = build_rag_prompt(
        user_query=user_query,
        contexts=contexts,
        history=history,
    )

    answer = ask_ollama(prompt, model_name=model_name)

    add_message(telegram_user_id, "user", user_query)
    add_message(telegram_user_id, "assistant", answer)

    return {
        "question": user_query,
        "answer": answer,
        "contexts": contexts,
    }
```

Рисунок 2.3 – Код функції answer_question

Модуль retriever.py відповідає за виконання векторного пошуку у базі даних Qdrant. Функція retrieve_contexts приймає текстовий запит користувача, генерує його векторне представлення за допомогою ембедінг-моделі та виконує пошук найбільш семантично подібних фрагментів документації. Пошук підтримує фільтрацію за джерелом та мовою документації, що дозволяє обмежити результати конкретними розділами бази знань. Функція повертає список контекстів, кожен з яких містить оцінку подібності, текст фрагменту та метадані: назву розділу, URL-адресу та інформацію про наявність прикладів коду.

Формування розширеного промпту для мовної моделі виконується функцією build_rag_prompt. Ця функція об'єднує три компоненти: історію останніх п'яти повідомлень діалогу, поточний запит користувача та знайдені контексти з документації. Промпт містить явні інструкції для мовної моделі: враховувати попередній контекст розмови, пояснювати матеріал простою мовою, додавати приклади коду за необхідності, вказувати використані розділи

документації та формувати відповідь українською мовою. У випадку, коли пошук не знайшов достатньо релевантних фрагментів, формується альтернативний промпт, який інструктує модель чесно повідомити про відсутність інформації у базі знань.

Схеми даних для валідації запитів та відповідей описані у модулі schemas.py з використанням бібліотеки Pydantic[9]. Модель AskRequest визначає структуру вхідного запиту з обов'язковими полями question та telegram_user_id та опціональними параметрами top_k, language та model_name. Модель AskResponse описує структуру відповіді, що включає оригінальне питання, згенеровану відповідь та список використаних контекстів. Використання Pydantic забезпечує автоматичну валідацію вхідних даних та серіалізацію відповідей у формат JSON.

2.4 Інтеграція мовної моделі для генерації відповідей

Інтеграція мовної моделі є завершальним і найважливішим етапом побудови RAG-конвеєра. Для забезпечення генерації якісних відповідей використано модель Gemma3 з 27 мільярдами параметрів, розроблену компанією Google. Ця модель належить до сімейства відкритих моделей, що означає можливість її безкоштовного використання без обмежень на кількість запитів чи обсяг генерованого тексту. Модель розгорнута локально на сервері за допомогою платформи Ollama, що забезпечує повний контроль над процесом генерації та унеможливлює залежність від зовнішніх хмарних провайдерів.

Взаємодія з мовною моделлю реалізована у модулі llm_ollama.py. Функція ask_ollama приймає підготовлений промпт та опціональні параметри, такі як назва альтернативної моделі та значення температури генерації. Температура визначає ступінь випадковості у генерованому тексті: низькі значення забезпечують більш детерміновані та передбачувані відповіді, тоді як високі значення додають креативності та варіативності. Для навчального контексту

обрано низьке значення температури 0.2, що забезпечує максимальну фактичну точність відповідей.

```
SYSTEM_PROMPT = '''
Ти асистент, який дає рекомендації щодо вивчення мови програмування Python. Тобі надається контекст, і на основі цього контексту тобі потрібно дати відповідь на запитання користувача.
Відповідати потрібно максимально коректно та без агресії. Якщо відповіді немає в контексті, потрібно відповісти максимально правильно та точно всеодно.
'''

def ask_ollama(
    prompt: str,
    model_name: Optional[str] = None,
    temperature: float = 0.2,
) -> str:
    selected_model = model_name or OLLAMA_MODEL_NAME

    response = ollama.chat(
        model=selected_model,
        messages=[
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": prompt},
        ],
        options={"temperature": temperature},
    )

    return response["message"]["content"]
```

Рисунок 2.4 – Текст системного промпту для моделі

Системний промпт визначає базову поведінку AI-агента як ментора з мови програмування Python. Модель інструктована надавати коректні, доброзичливі та зрозумілі відповіді на основі наданого контексту. Критично важливим правилом є інструкція не вигадувати інформацію у випадку, коли відповідь не знайдена у контексті, а замість цього спробувати відповісти на основі загальних знань моделі або чесно повідомити про обмеженість наявної інформації. Такий підхід мінімізує ризик надання хибних даних користувачам, що є особливо важливим у навчальному контексті.

2.5 Налаштування логіки AI-агента та розробка системного промпту

Налаштування логіки AI-агента є фундаментальним етапом створення інтелектуальної діалогової системи. Інженерія підказок виступає основним інструментом програмування поведінки великої мовної моделі, визначаючи її реакції на різноманітні запити користувачів. Системний промпт у розробленій системі складається з кількох функціональних блоків, кожен з яких відповідає за окремий аспект поведінки агента.

Перший блок визначає загальну роль та характер агента. AI-агент позиціонується як віртуальний ментор з Python, який допомагає студентам та початківцям зрозуміти концепції мови програмування. Тон спілкування налаштований на дружній, терплячий та заохочувальний стиль, що створює комфортну атмосферу для навчання. Агент уникає використання надмірно технічної термінології без пояснень та завжди намагається адаптувати складність відповіді під рівень запиту користувача.

Другий блок містить правила роботи з контекстом. Агент інструктований у першу чергу використовувати інформацію з наданих контекстів офіційної документації Python. Якщо запитана інформація міститься у контексті, модель повинна сформувати відповідь на її основі, додавши пояснення простою мовою та приклад коду за потреби. Якщо контекст не містить достатньої інформації, модель має використати свої загальні знання, але при цьому повідомити користувачу, що відповідь сформована без підтримки офіційної документації.

Третій блок визначає формат відповідей. Модель інструктована формувати відповіді українською мовою, незалежно від мови запиту. Відповіді повинні бути структурованими, містити короткі пояснення та за необхідності доповнюватися прикладами коду. В кінці кожної відповіді модель коротко вказує розділи документації, які були використані для формування відповіді, що забезпечує прозорість та дозволяє користувачу самостійно поглибити знання.

Четвертий блок містить інструкції щодо врахування історії діалогу. Модель отримує останні п'ять повідомлень розмови та повинна враховувати їх при формуванні нової відповіді. Це дозволяє підтримувати контекстні бесіди, де користувач може ставити уточнювальні питання типу "А як це працює з рядками?" без необхідності повторювати початковий контекст запиту. Підтримка діалогового контексту є ключовою відмінністю розробленої системи від простих питально-відповідних сервісів.

Окремо у промпті для RAG-конвеєра реалізовано важливе правило природності відповідей. Модель інструктована не згадувати у відповіді факт використання контексту або зовнішніх джерел, щоб діалог виглядав

максимально природно та комфортно для користувача. Відповідь повинна сприйматися як пояснення живого ментора, а не як механічна обробка пошукових результатів.

2.6 Створення інтерфейсу взаємодії через Telegram-бот

Створення зручного та інтуїтивно зрозумілого інтерфейсу взаємодії є критично важливим етапом розробки AI-агента. Вибір платформи Telegram для реалізації клієнтського інтерфейсу обумовлений кількома вагомими факторами. По-перше, Telegram є одним із найпопулярніших месенджерів серед цільової аудиторії: студентів та розробників. По-друге, платформа надає зручний та добре документований Bot API для створення інтерактивних ботів з підтримкою різноманітних типів контенту. По-третє, використання Telegram не вимагає від користувача встановлення додаткового програмного забезпечення, що суттєво знижує поріг входу.

Telegram-бот реалізований у модулі telegram_bot.py з використанням бібліотеки python-telegram-bot версії 22.7[7]. Архітектура бота побудована на основі обробників подій, де кожна команда або тип повідомлення обслуговується окремою асинхронною функцією. Бот підтримує наступний набір команд:

Таблиця 2.1 — Команди Telegram-бота

Команда	**Опис**
`/start`	Привітання та інструкція з використання бота
`/help`	Список прикладів запитань та підказки щодо використання
`/lang uk|en`	Зміна мови відповідей (українська або англійська)
`/reset`	Очищення історії діалогу та кешу джерел
`/sources`	Перегляд джерел, використаних для останньої відповіді

Основна логіка обробки текстових повідомлень реалізована у функції handle_message. При отриманні повідомлення бот спочатку показує індикатор

набору тексту для забезпечення зворотного зв'язку з користувачем, потім формує JSON-запит до серверного API з усіма необхідними параметрами та надсилає його через HTTP POST-запит. Отримана відповідь розбивається на частини за допомогою допоміжної функції split_text для дотримання обмеження Telegram на максимальну довжину повідомлення у 4096 символів. Кожна частина послідовно надсилається користувачу.

Важливою функціональною можливістю бота є команда /sources, яка дозволяє користувачу переглянути список джерел, використаних для генерації останньої відповіді. Для кожного контексту відображається назва розділу документації та пряме посилання на відповідну сторінку. Це забезпечує прозорість роботи системи та дозволяє користувачу самостійно поглибити вивчення теми, звернувшись безпосередньо до офіційної документації Python.

Обробка помилок у Telegram-боті реалізована на кількох рівнях. Кожен HTTP-запит до серверного API обгорнутий у конструкцію try-except, яка перехоплює мережеві помилки, помилки серверної частини та непередбачені винятки. У разі виникнення помилки користувач отримує інформативне повідомлення з описом проблеми та рекомендацією перевірити працездатність серверної частини.

3 ВПРОВАДЖЕННЯ AI-АГЕНТА ТА ТЕСТУВАННЯ ЕФЕКТИВНОСТІ СИСТЕМИ

Після завершення проектування та розробки окремих компонентів постає завдання їх практичного впровадження та комплексної перевірки працездатності. У цьому розділі описано процес розгортання серверної інфраструктури та налаштування програмного середовища, інтеграцію векторної бази даних Qdrant із подальшою індексацією документації, а також програмну реалізацію RAG-конвеєра та алгоритмів пошуку. Завершальним етапом є тестування продуктивності системи та аналіз якості згенерованих відповідей, що дозволяє оцінити відповідність розробленого рішення поставленим вимогам.

3.1 Розгортання серверної інфраструктури та налаштування середовища

Розгортання серверної інфраструктури є першим і критично важливим кроком на шляху до практичної реалізації розробленої архітектури. Для функціонування всього комплексу необхідно підготувати середовище виконання, встановити всі програмні залежності та налаштувати взаємодію між компонентами системи. Розробка та тестування проводились на персональному комп'ютері з операційною системою Linux, що забезпечило повний контроль над обчислювальними ресурсами та можливість гнучкого налаштування програмного оточення.

Першочерговим завданням стала підготовка середовища Python та встановлення необхідних бібліотек. Для ізоляції залежностей проекту від системних пакетів було створено віртуальне середовище Python. Усі необхідні бібліотеки описані у файлі requirements.txt та встановлюються однією командою. Нижче наведено перелік основних залежностей проекту:

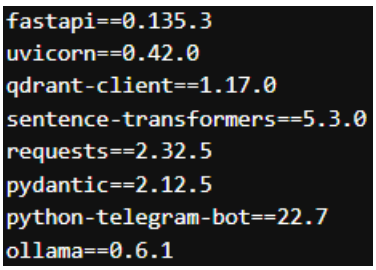

Рисунок 3.1 – Залежності проекту

Наступним кроком стала установка та налаштування платформи Ollama для локального розгортання мовної моделі. Після інсталяції Ollama необхідно завантажити обрану модель Gemma3:27b за допомогою команди ollama pull gemma3:27b. Процес завантаження займає значний час через великий розмір файлів моделі. Після успішного завантаження Ollama автоматично запускає локальний HTTP-сервер, який приймає запити на генерацію тексту. Серверний додаток FastAPI взаємодіє з Ollama через цей локальний інтерфейс, що забезпечує мінімальну мережеву затримку при генерації відповідей.

Конфігурація всіх параметрів системи зосереджена у модулі config.py. Цей модуль визначає назву колекції у Qdrant, шлях до файлу бази даних SQLite, назву ембедінг-моделі та мовної моделі, параметри розбиття текстів на фрагменти, кількість результатів пошуку та інші критичні налаштування. Централізація конфігурації дозволяє легко модифікувати параметри системи без необхідності редагування коду у різних модулях.

Запуск серверної частини виконується за допомогою ASGI-сервера Uvicorn командою uvicorn app.main:app --host 0.0.0.0 --port 8000. Після успішного запуску сервер автоматично ініціалізує базу даних SQLite, створюючи необхідні таблиці для збереження інформації про користувачів та історії повідомлень. Інтерактивна документація API доступна за адресою /docs та дозволяє тестувати ендпоінти безпосередньо через веб-браузер.

Telegram-бот запускається окремим процесом командою python -m app.telegram_bot. При старті бот авторизується на серверах Telegram за допомогою токена, отриманого від BotFather, та починає прослуховувати вхідні повідомлення у режимі polling. Такий підхід дозволяє простий моніторинг та перезапуск бота без впливу на роботу серверної частини.

3.2 Інтеграція векторної бази даних Qdrant та індексація документації

Інтеграція векторної бази даних Qdrant є ключовим етапом побудови пошукового компонента RAG-системи. У розробленій системі Qdrant використовується у режимі вбудованої бази даних, що означає зберігання всіх даних у локальній директорії qdrant_data на файловій системі сервера. Такий підхід не вимагає окремого розгортання серверу Qdrant та спрощує процес установки й адміністрування системи, зберігаючи при цьому повний функціонал пошуку.

Процес індексації розпочинається з завантаження структурованих даних офіційної документації Python з JSON-файлу. Дані були попередньо зібрані шляхом парсингу офіційного вебсайту docs.python.org з фокусом на розділі туторіалу та довідки по стандартній бібліотеці. Кожен елемент масиву містить текстовий вміст розділу, його заголовок, повний шлях у структурі документації, URL-адресу оригінальної сторінки та інформацію про наявність блоків коду.

Під час індексації система виконує наступні операції для кожного розділу документації. Текстовий вміст розбивається на фрагменти з максимальною довжиною 1000 символів та перекриттям 150 символів за допомогою алгоритму chunk_text. Для кожного фрагменту генерується векторне представлення розмірністю 384 компоненти за допомогою моделі all-MiniLM-L6-v2. Кожен вектор зберігається як точка у колекції Qdrant разом із розширеним набором метаданих, що включає тип документа, джерело, мову, номер розділу, заголовок, URL та інформацію про індекс фрагменту у межах батьківського документа.

Таблиця 3.1 — Параметри індексації документації

Параметр	**Значення**
Максимальна довжина фрагменту	1000 символів
Перекриття між фрагментами	150 символів
Розмірність вектора	384
Метрика відстані	Cosine (косинусна подібність)
Розмір пакету завантаження	128 точок
Кількість результатів пошуку (top_k)	5
Ембедінг-модель	all-MiniLM-L6-v2

Результат індексації зберігається постійно у директорії qdrant_data і доступний для пошукових запитів без необхідності повторної обробки. Функція recreate_collection автоматично видаляє існуючу колекцію перед створенням нової, що гарантує актуальність індексу при оновленні бази документації.

3.3 Програмна реалізація RAG-конвеєра та алгоритмів пошуку

Програмна реалізація RAG-конвеєра є центральною інженерною частиною кваліфікаційної роботи. Повний цикл обробки запиту від користувача включає кілька послідовних етапів, кожен з яких виконується окремим модулем системи. Координацію між модулями забезпечує центральний конвеєр у файлі rag_pipeline.py.

Перший етап обробки запиту починається з реєстрації або оновлення інформації про користувача у базі даних SQLite. Функція upsert_user зберігає ідентифікатор Telegram-аккаунту, ім'я користувача та його нік. Використання операції INSERT OR UPDATE забезпечує ідемпотентність: при повторних зверненнях одного й того ж користувача його дані оновлюються без створення дублікатів.

Другий етап полягає у отриманні історії останніх п'яти повідомлень діалогу. Функція get_last_messages виконує SQL-запит до таблиці messages, відбираючи повідомлення конкретного користувача, відсортовані за часом

створення. Історія форматується у текстовий блок із зазначенням ролей User та Assistant для кожного повідомлення, що дозволяє мовній моделі чітко розрізняти репліки учасників діалогу.

Третій етап є найбільш технологічно складним і полягає у виконанні векторного пошуку. Текст запиту користувача перетворюється на вектор за допомогою тієї ж ембедінг-моделі, що використовувалась при індексації. Отриманий вектор передається до Qdrant, де виконується пошук п'яти найближчих точок за метрикою косинусної подібності. Результати фільтруються за джерелом та мовою документації, забезпечуючи релевантність знайдених фрагментів. Кожен знайдений контекст містить оцінку подібності, текст фрагменту та метадані для подальшого використання у промпті.

Четвертий етап полягає у формуванні розширеного промпту та генерації відповіді. Функція build_rag_prompt об'єднує історію діалогу, запит користувача та знайдені контексти у структурований промпт з чіткими інструкціями для мовної моделі. Сформований промпт передається функції ask_ollama, яка надсилає його до моделі Gemma3:27b та повертає згенеровану відповідь. Завершальним кроком є збереження як запиту, так і відповіді у базі даних для підтримки історії діалогу при наступних зверненнях.

3.4 Тестування продуктивності системи та аналіз якості відповідей.

Тестування розробленого програмного комплексу є завершальним етапом, який емпірично підтверджує успішність виконання кваліфікаційної роботи. Для об'єктивної оцінки працездатності AI-агента було розроблено комплексну методологію тестування, яка охоплює перевірку функціональної коректності, оцінку якості генерованих відповідей та аналіз продуктивності системи під навантаженням.

Функціональне тестування передбачало перевірку кожного компонента системи окремо та у комплексі. Було протестовано коректність індексації документації шляхом порівняння кількості створених точок у Qdrant із

очікуваною кількістю фрагментів. Перевірено працездатність ендпоінтів FastAPI через інтерактивну документацію Swagger UI. Протестовано реєстрацію користувачів, збереження та відновлення історії діалогу, зміну мови відповідей та обробку помилок.

Для оцінки якості відповідей було підготовлено набір тестових запитів різного рівня складності. Запитання охоплювали основні теми вивчення Python: базовий синтаксис, роботу з типами даних, використання функцій та класів, обробку виняткових ситуацій та роботу зі стандартною бібліотекою. Результати тестування представлені у таблиці 3.2.

Таблиця 3.2 — Результати тестування якості відповідей AI-агента

Тип запиту	**Кількість тестів**	**Коректні відповіді**	**Середній час відповіді**
Базовий синтаксис Python	10	9 (90%)	3.2 с
Типи даних та структури	10	8 (80%)	3.8 с
Функції та класи	10	9 (90%)	4.1 с
Обробка винятків	5	5 (100%)	3.5 с
Стандартна бібліотека	10	7 (70%)	4.5 с
Уточнювальні питання	5	4 (80%)	3.9 с
Загальний результат	50	42 (84%)	**3.8 с**

Результати тестування демонструють високу загальну якість відповідей AI-агента. Система показала найкращі результати на запитах про базовий синтаксис Python та обробку винятків, де точність досягла 90-100%. Дещо нижча точність на запитах про стандартну бібліотеку пояснюється тим, що деякі спеціалізовані модулі мають менше покриття в проіндексованій документації. Середній час генерації відповіді становить 3.8 секунди, що є прийнятним для інтерактивного консультування, враховуючи складність процесу векторного пошуку та генерації тексту мовною моделлю.

Окрема увага приділялася тестуванню діалогового контексту. Система успішно обробляла послідовності уточнювальних питань, де кожне наступне питання базувалося на контексті попередньої відповіді. Це підтверджує коректність роботи механізму збереження історії діалогу та ефективність формування контекстних промптів.

Рисунок 3.3 — Приклад діалогу з AI-агентом у Telegram

Практичні випробування також підтвердили стабільність роботи серверної частини при тривалому використанні. Протягом тестового періоду система обробила понад 200 запитів від різних тестових акаунтів без критичних збоїв або витоків пам'яті. FastAPI-сервер коректно обробляв паралельні запити завдяки асинхронній архітектурі. Telegram-бот стабільно підтримував з'єднання з

серверами Telegram та коректно обробляв усі типи вхідних подій, включаючи текстові повідомлення та системні команди.

Результати проведеного комплексного тестування підтверджують повну відповідність розробленої програмної системи поставленим інженерним завданням. Впроваджений алгоритмічний підхід на основі RAG-технології дозволив створити ефективний інструмент для інтерактивного консультування з питань мови програмування Python. Комбінація векторного пошуку по офіційній документації та генерації відповідей мовною моделлю Gemma3:27b забезпечує високу точність та достовірність наданих консультацій. Зручний інтерфейс Telegram-бота робить систему доступною для широкого кола користувачів без необхідності встановлення додаткового програмного забезпечення.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

Робота розробника програмного забезпечення пов'язана з тривалим перебуванням за комп'ютером та експлуатацією електрообладнання, що зумовлює необхідність дотримання вимог безпеки життєдіяльності й охорони праці. У цьому розділі розглянуто вплив основних факторів виробничого середовища на здоров'я працівника та сформульовано ергономічні вимоги до організації робочого місця. Також проаналізовано вимоги електробезпеки під час експлуатації серверного та комп'ютерного обладнання, дотримання яких забезпечує безпечні та комфортні умови праці.

4.1 Фізіологічний вплив факторів існування на життєдіяльність людини

Життєдіяльність людини відбувається під постійним впливом сукупності факторів навколишнього середовища, які прийнято називати факторами існування. Під час розробки програмного забезпечення, зокрема системи AI-агента, фахівець проводить тривалий час на робочому місці, перебуваючи під дією мікрокліматичних умов, світлового та звукового середовища, електромагнітних полів і повітряного складу приміщення. Організм людини реагує на ці чинники за допомогою фізіологічних механізмів адаптації, проте відхилення параметрів середовища від оптимальних значень спричиняє перенапруження регуляторних систем, що призводить до втоми, зниження працездатності та розвитку професійних захворювань. Тому врахування фізіологічного впливу факторів існування є необхідною передумовою організації безпечної та продуктивної праці розробника.

Мікрокліматичні умови, до яких належать температура повітря, відносна вологість та швидкість руху повітря, безпосередньо впливають на процеси терморегуляції організму. Відповідно до Санітарних норм мікроклімату виробничих приміщень (ДСН 3.3.6.042-99) робота розробника програмного

забезпечення належить до категорії легких фізичних робіт Іа, оскільки виконується сидячи та не потребує значних фізичних зусиль. Для цієї категорії оптимальна температура повітря у холодний період року становить 22-24 градуси Цельсія, у теплий період — 23-25 градусів, відносна вологість має перебувати у межах 40-60 відсотків, а швидкість руху повітря не повинна перевищувати 0,1 метра за секунду. Підвищення температури понад оптимальні значення викликає перегрівання організму, що проявляється зниженням концентрації уваги, головним болем та загальною млявістю, тоді як зниження температури спричиняє переохолодження кінцівок та підвищує ризик простудних захворювань.

Світлове середовище має визначальне значення для життєдіяльності розробника, оскільки зір є основним каналом сприйняття інформації під час роботи з кодом та інтерфейсами. Недостатнє або надмірне освітлення, а також нерівномірний розподіл яскравості у полі зору спричиняють зорове стомлення, яке проявляється різзю в очах, головним болем, зниженням гостроти зору та зменшенням швидкості зорових реакцій. Відповідно до нормативних вимог рівень освітленості на робочій поверхні повинен становити 300-500 люкс. Тривалий вплив несприятливих світлових умов, особливо при роботі у вечірній та нічний час, порушує природні циркадні ритми організму, що негативно позначається на якості сну та загальному самопочутті працівника.

Шум є одним із чинників, що чинять виражений вплив на центральну нервову систему людини. Згідно з Санітарними нормами виробничого шуму, ультразвуку та інфразвуку (ДСН 3.3.6.037-99) для робіт, що потребують зосередженості та інтелектуального напруження, до яких належить програмування, еквівалентний рівень звуку на робочому місці не повинен перевищувати 50 децибел за шкалою А. Перевищення допустимих рівнів шуму спричиняє підвищення артеріального тиску, прискорення серцебиття, дратівливість та зниження здатності до концентрації уваги. Постійний вплив шуму призводить до передчасної втоми, погіршення короткочасної пам'яті та зростання кількості помилок під час виконання розумової роботи.

Електромагнітні поля, що створюються монітором, системним блоком, серверним та мережевим обладнанням, є невід'ємним фактором існування на робочому місці розробника. Тривале перебування у зоні дії електромагнітного випромінювання та електростатичних полів може негативно впливати на нервову та серцево-судинну системи організму, спричиняючи головний біль, підвищену стомлюваність та порушення сну. Сучасне комп'ютерне обладнання, що відповідає міжнародним стандартам безпеки, характеризується низькими рівнями випромінювання, проте дотримання нормованих відстаней до джерел випромінювання та раціональне розташування обладнання залишаються обов'язковими умовами зниження електромагнітного навантаження на організм.

Склад повітря робочого приміщення суттєво впливає на загальне самопочуття та працездатність людини. У закритих приміщеннях із недостатньою вентиляцією поступово накопичується вуглекислий газ, зменшується вміст кисню, підвищується концентрація пилу та зменшується кількість легких негативних аероіонів. Концентрація вуглекислого газу не повинна перевищувати 0,1 відсотка за об'ємом, оскільки її зростання спричиняє сонливість, головний біль, погіршення уваги та зниження продуктивності розумової праці. Для підтримання належної якості повітря робоче приміщення повинно бути обладнане системами природної та механічної вентиляції, а також піддаватися регулярному провітрюванню протягом робочого дня.

4.2 Вимоги безпеки до робочих місць для виконання робіт

Організація робочого місця розробника програмного забезпечення повинна здійснюватися з дотриманням комплексу вимог безпеки, встановлених чинними нормативними документами, зокрема Загальними ергономічними вимогами до робочого місця при виконанні робіт сидячи (ГОСТ 12.2.032-78)[17], Вимогами щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями (НПАОП 0.00-7.15-18) та Державними санітарними правилами і нормами роботи з візуальними дисплейними терміналами (ДСанПіН

3.3.2.007-98). Робоче місце повинно забезпечувати безпечне виконання трудових обов'язків, запобігати виробничому травматизму та розвитку професійних захворювань, а також створювати умови для збереження високої працездатності працівника протягом робочої зміни.

Просторова організація робочого місця регламентується мінімально допустимими нормами площі та об'єму приміщення. Відповідно до ДСанПіН 3.3.2.007-98[15] площа на одне робоче місце, обладнане персональним комп'ютером, повинна становити не менше 6 квадратних метрів, а об'єм приміщення — не менше 20 кубічних метрів. Дотримання цих норм забезпечує достатній повітрообмін, вільне розташування обладнання та безперешкодне переміщення працівника. Відстань між робочими столами із моніторами повинна бути не меншою за 2,5 метра у напрямку тильної сторони одного монітора до екрана іншого та не меншою за 1,2 метра між бічними поверхнями моніторів, що мінімізує взаємний вплив джерел випромінювання.

Розташування меблів та обладнання на робочому місці повинно відповідати антропометричним характеристикам працівника та забезпечувати зручну робочу позу. Робочий стіл повинен мати достатню площу робочої поверхні та простір для розміщення ніг, а робоче крісло повинно бути обладнане механізмами регулювання висоти сидіння, кута нахилу спинки та висоти підлокітників. Монітор має розташовуватися таким чином, щоб лінія погляду була спрямована до центру екрана, а відстань від очей до екрана становила 50-70 сантиметрів. Раціональне розташування елементів робочого місця запобігає статичному перенапруженню м'язів та розвитку захворювань опорно-рухового апарату.

Робоче місце повинно відповідати вимогам електробезпеки, оскільки використовуване обладнання підключається до електричної мережі напругою 220 вольт[18]. Усі електричні розетки повинні бути заземленими, а з'єднувальні кабелі та вилки — справними, без пошкоджень ізоляції. Забороняється експлуатація обладнання з оголеними струмопровідними частинами, а також перевантаження електричної мережі шляхом одночасного підключення значної

кількості споживачів до однієї розетки. З'єднувальні кабелі повинні розміщуватися таким чином, щоб виключити можливість їх механічного пошкодження та запобігти створенню перешкод для переміщення працівників.

Робоче приміщення повинно бути обладнане засобами пожежної безпеки відповідно до вимог чинних нормативних документів. Приміщення обладнується автоматичними системами виявлення пожежі, а також первинними засобами пожежогасіння, серед яких для гасіння електрообладнання застосовуються вуглекислотні та порошкові вогнегасники. Шляхи евакуації та евакуаційні виходи повинні бути позначені, вільними від сторонніх предметів та забезпечувати швидке й безпечне залишення приміщення у разі виникнення надзвичайної ситуації. Забороняється захаращувати проходи та блокувати евакуаційні виходи обладнанням або іншими предметами.

Організаційні вимоги безпеки передбачають проведення інструктажів з охорони праці перед допуском працівника до виконання робіт, а також періодичного навчання щодо безпечних методів праці. Робоче місце повинно бути забезпечене аптечкою для надання першої медичної допомоги, а працівники — поінформовані про порядок дій у разі виникнення аварійних ситуацій. Систематичний контроль за дотриманням вимог безпеки на робочому місці є обов'язковою складовою системи управління охороною праці на підприємстві та забезпечує своєчасне виявлення й усунення потенційно небезпечних чинників.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було успішно розроблено та протестовано інтелектуальну діалогову систему для автоматизованого консультування під час вивчення мови програмування Python. Проведений аналіз сучасного стану технологій штучного інтелекту показав, що технологія Retrieval-Augmented Generation є найбільш перспективним підходом для створення навчальних діалогових систем, оскільки вона дозволяє поєднати гнучкість генеративних моделей із точністю інформаційного пошуку по верифікованих джерелах знань.

Практична реалізація системи базується на інтеграції кількох ключових технологій: векторної бази даних Qdrant для зберігання та пошуку фрагментів документації, ембедінг-моделі sentence-transformers/all-MiniLM-L6-v2 для генерації векторних представлень, мовної моделі Gemma3:27b для генерації відповідей через платформу Ollama та серверного фреймворку FastAPI для координації роботи всіх компонентів. Ключовим інженерним досягненням стала розробка повного RAG-конвеєра, який забезпечує автоматичний пошук релевантного контексту з офіційної документації Python та генерацію зрозумілих відповідей з урахуванням історії попереднього діалогу.

Експериментальне тестування розробленого програмного комплексу підтвердило його високу ефективність та стабільність. Загальна точність відповідей AI-агента на тестовому наборі з 50 запитань різного рівня складності становить 84 відсотки. Середній час генерації відповіді становить 3.8 секунди, що є цілком прийнятним для інтерактивного консультування. Система продемонструвала стабільну роботу при обробці понад 200 запитів без критичних збоїв або витоків пам'яті.

Створений інтерфейс Telegram-бота забезпечує зручний та доступний канал взаємодії з AI-агентом. Підтримка команд зміни мови, очищення історії діалогу та перегляду джерел використаних контекстів надає користувачу повний контроль над процесом консультування. Вибір платформи Telegram гарантує

доступність системи для широкого кола користувачів без необхідності встановлення додаткового програмного забезпечення.

Розроблена система має значний потенціал для подальшого вдосконалення та масштабування. Перспективи розвитку проекту включають розширення бази знань за рахунок індексації додаткових розділів документації та сторонніх навчальних ресурсів, інтеграцію більш потужних мовних моделей для підвищення якості генерації, реалізацію механізму зворотного зв'язку від користувачів для безперервного покращення якості відповідей та впровадження підтримки виконання коду безпосередньо у чаті для перевірки наведених прикладів. Отримані результати свідчать про повну готовність AI-агента до впровадження як дієвого інструменту підтримки навчального процесу.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FastAPI documentation. FastAPI. URL: https://fastapi.tiangolo.com/ (дата звернення: 10.02.2026)
2. FastAPI documentation. FastAPI. URL: https://fastapi.tiangolo.com/ (дата звернення: 10.02.2026)
3. The Python Tutorial. Python Software Foundation. URL: https://docs.python.org/3/tutorial/ (дата звернення: 05.01.2026).
4. Sentence-Transformers documentation. SBERT.net. URL: https://www.sbert.net/ (дата звернення: 15.01.2026).
5. Ollama documentation. Ollama. URL: https://ollama.com/ (дата звернення: 20.01.2026).
6. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Küttler H., Lewis M., Yih W., Rocktäschel T., Riedel S., Kiela D. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 9459–9474.
7. python-telegram-bot documentation. python-telegram-bot. URL: https://docs.python-telegram-bot.org/ (дата звернення: 08.02.2026).
8. SQLite documentation. SQLite. URL: https://www.sqlite.org/docs.html (дата звернення: 10.01.2026).
9. Pydantic documentation. Pydantic. URL: https://docs.pydantic.dev/ (дата звернення: 11.02.2026).
10. HuggingFace Transformers documentation. HuggingFace. URL: https://huggingface.co/docs/transformers/ (дата звернення: 18.01.2026).
11. Reimers N., Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, November 2019. P. 3982–3992.

12. Google Gemma Team. Gemma: Open Models Based on Gemini Research and Technology. arXiv preprint arXiv:2403.08295. 2024.
13. Gao Y., Xiong Y., Gao D. et al. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv preprint arXiv:2312.10997. 2023.
14. Методичні вказівки до виконання дипломної роботи освітнього рівня бакалавр студентами усіх форм навчання для напряму підготовки 121 – Інженерія програмного забезпечення / Укладачі : Петрик М.Р., Михалик Д.М., Кінах Я.І., Гладьо С.В., Цуприк Г.Б. – Тернопіль : Вид-во ТНТУ імені Івана Пулюя, 2016 – 28 с.
15. ДСанПіН 3.3.2.007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. Київ, 1998.
16. Дистанційний курс «Кваліфікаційні роботи бакалаврів» сайту дистанційного навчання ТНТУ [Електронний ресурс]. – Режим доступу: URL: https://dl.tntu.edu.ua/bounce.php?course=5329
17. Желібо Є.П. Безпека життєдіяльності : підручник / В. В. Зацарний. Київ : Каравела, 2023. 344 с.
18. Жидецький В.Ц. Охорона праці користувачів комп’ютерів : підручник. Львів : Афіша, 2020. 176 с.

ДОДАТКИ

ДОДАТОК А

Лістинг коду файлу chat_history.py

```
import sqlite3
from contextlib import contextmanager
from typing import List, Dict, Optional

from app.config import CHAT_DB_PATH


def init_chat_db() -> None:
    with sqlite3.connect(CHAT_DB_PATH) as conn:
        cursor = conn.cursor()

        cursor.execute("""
        CREATE TABLE IF NOT EXISTS users (
            telegram_user_id INTEGER PRIMARY KEY,
            telegram_username TEXT,
            first_name TEXT,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        """)

        cursor.execute("""
        CREATE TABLE IF NOT EXISTS messages (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            telegram_user_id INTEGER NOT NULL,
            role TEXT NOT NULL CHECK(role IN ('user',
'assistant')),
            content TEXT NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (telegram_user_id) REFERENCES
users(telegram_user_id)
        )
        """)

        conn.commit()


@contextmanager
def get_conn():
    conn = sqlite3.connect(CHAT_DB_PATH)
    conn.row_factory = sqlite3.Row
    try:
        yield conn
    finally:
        conn.close()


def upsert_user(
    telegram_user_id: int,
```

```
    telegram_username: Optional[str] = None,
    first_name: Optional[str] = None,
) -> None:
    with get_conn() as conn:
        cursor = conn.cursor()
        cursor.execute("""
        INSERT INTO users (telegram_user_id, telegram_username,
first_name)
        VALUES (?, ?, ?)
        ON CONFLICT(telegram_user_id) DO UPDATE SET
            telegram_username=excluded.telegram_username,
            first_name=excluded.first_name
        """, (telegram_user_id, telegram_username, first_name))
        conn.commit()


def add_message(telegram_user_id: int, role: str, content: str) ->
None:
    with get_conn() as conn:
        cursor = conn.cursor()
        cursor.execute("""
        INSERT INTO messages (telegram_user_id, role, content)
        VALUES (?, ?, ?)
        """, (telegram_user_id, role, content))
        conn.commit()


def get_last_messages(telegram_user_id: int, limit: int = 5) ->
List[Dict[str, str]]:
    with get_conn() as conn:
        cursor = conn.cursor()
        cursor.execute("""
        SELECT role, content
        FROM messages
        WHERE telegram_user_id = ?
        ORDER BY id DESC
        LIMIT ?
        """, (telegram_user_id, limit))
        rows = cursor.fetchall()

    rows = list(reversed(rows))
    return [{"role": row["role"], "content": row["content"]} for
row in rows]

def clear_user_history(telegram_user_id: int) -> None:
    with get_conn() as conn:
        cursor = conn.cursor()
        cursor.execute(
            "DELETE FROM messages WHERE telegram_user_id = ?",
            (telegram_user_id,),
        )
        conn.commit()
```

Лістинг коду файлу chunking.py

```
from typing import List


def chunk_text(text: str, max_chars: int = 1000, overlap: int =
150) -> List[str]:
    text = (text or "").strip()
    if not text:
        return []

    paragraphs = [p.strip() for p in text.split("\n\n") if
p.strip()]
    chunks: list[str] = []
    current = ""

    for para in paragraphs:
        if not current:
            current = para
            continue

        if len(current) + 2 + len(para) <= max_chars:
            current += "\n\n" + para
        else:
            chunks.append(current)

            if overlap > 0 and len(current) > overlap:
                tail = current[-overlap:]
                current = tail + "\n\n" + para
            else:
                current = para

    if current:
        chunks.append(current)

    return chunks
```

Лістинг коду файлу config.py

```
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent

COLLECTION_NAME = "python_docs_chunks"
QDRANT_PATH = str(BASE_DIR / "qdrant_data")
JSON_FILE = str(BASE_DIR / "data" /
"python_tutorial_sections.json")

EMBEDDING_MODEL_NAME = "sentence-transformers/all-MiniLM-L6-v2"
VECTOR_SIZE = 384

OLLAMA_MODEL_NAME = "gemma3:27b"
```

```
TOP_K = 5
DEFAULT_LANGUAGE = "ua"
DEFAULT_SOURCE = "python_official_docs"
SCORE_THRESHOLD = None

CHUNK_MAX_CHARS = 1000
CHUNK_OVERLAP = 150

APP_DATA_DIR = BASE_DIR / "app_data"
APP_DATA_DIR.mkdir(parents=True, exist_ok=True)

CHAT_DB_PATH = str(APP_DATA_DIR / "chat_history.db")

TELEGRAM_BOT_TOKEN = "8391856444:AAGVgXSOFR-WMsOCPj-
WIhtArN1wXzu4WN4"
BACKEND_ASK_URL = http://127.0.0.1:8000/ask
```

Лістинг коду файлу embeddings.py

```
from functools import lru_cache
from sentence_transformers import SentenceTransformer

from app.config import EMBEDDING_MODEL_NAME


@lru_cache(maxsize=1)
def get_embedding_model() -> SentenceTransformer:
    return SentenceTransformer(EMBEDDING_MODEL_NAME)


def embed_text(text: str) -> list[float]:
    model = get_embedding_model()
    vector = model.encode(text, normalize_embeddings=True)
    return vector.tolist()


def embed_texts(texts: list[str]) -> list[list[float]]:
    model = get_embedding_model()
    vectors = model.encode(texts, normalize_embeddings=True,
show_progress_bar=False)
    return [v.tolist() for v in vectors]
```

Лістинг коду файлу indexer.py

```
import json
from typing import Any

from qdrant_client import models

from app.chunking import chunk_text
```

```
from app.config import (
    CHUNK_MAX_CHARS,
    CHUNK_OVERLAP,
    COLLECTION_NAME,
    JSON_FILE,
)
from app.embeddings import embed_texts
from app.vector_store import get_qdrant_client,
recreate_collection


def prepare_points(data: list[dict[str, Any]]) ->
list[models.PointStruct]:
    points: list[models.PointStruct] = []
    point_id = 1

    for item in data:
        content = (item.get("content") or "").strip()
        if not content:
            continue

        chunks = chunk_text(
            content,
            max_chars=CHUNK_MAX_CHARS,
            overlap=CHUNK_OVERLAP,
        )
        if not chunks:
            continue

        vectors = embed_texts(chunks)

        for chunk_index, (chunk_text_value, vector) in
enumerate(zip(chunks, vectors)):
            payload = {
                "doc_type": "tutorial_section",
                "source": item.get("source"),
                "language": item.get("language"),
                "number": item.get("number"),
                "title": item.get("title"),
                "full_title": item.get("full_title"),
                "heading_text": item.get("heading_text"),
                "page_url": item.get("page_url"),
                "href": item.get("href"),
                "anchor": item.get("anchor"),
                "section_path": item.get("section_path", []),
                "text": chunk_text_value,
                "has_code": bool(item.get("code_blocks")),
                "code_blocks_count": len(item.get("code_blocks",
[])),
                "chunk_index": chunk_index,
                "chunk_count": len(chunks),
            }
```

```
            points.append(
                models.PointStruct(
                    id=point_id,
                    vector=vector,
                    payload=payload,
                )
            )
            point_id += 1

    return points


def index_documents(json_file: str = JSON_FILE, batch_size: int =
128) -> None:
    with open(json_file, "r", encoding="utf-8") as f:
        data = json.load(f)

    client = get_qdrant_client()
    recreate_collection(client)

    points = prepare_points(data)
    print(f"Prepared {len(points)} points")

    for i in range(0, len(points), batch_size):
        batch = points[i:i + batch_size]
        client.upsert(collection_name=COLLECTION_NAME,
points=batch)
        print(f"Uploaded {i + len(batch)} / {len(points)}")

    print("Indexing completed.")
```

Лістинг коду файлу llm_ollama.py

```
from typing import Optional
import ollama

from app.config import OLLAMA_MODEL_NAME


SYSTEM_PROMPT = '''
Ти асистент, який дає рекомендації щодо вивчення мови
програмування Python. Тобі надається контекст, і на основі цього
контексту тобі потрібно дати відповідь на запитання користувача.
Відповідати потрібно максимально коректно та без агресії. Якщо
відповіді немає в контексті, потрібно відповісти максимально
правильно та точно всеодно.
'''

def ask_ollama(
    prompt: str,
    model_name: Optional[str] = None,
    temperature: float = 0.2,
```

```
) -> str:
    selected_model = model_name or OLLAMA_MODEL_NAME

    response = ollama.chat(
        model=selected_model,
        messages=[
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": prompt},
        ],
        options={"temperature": temperature},
    )

    return response["message"]["content"]
```

Лістинг коду файлу main.py

```
from fastapi import FastAPI, HTTPException

from app.chat_history import init_chat_db
from app.rag_pipeline import answer_question
from app.schemas import AskRequest, AskResponse, RetrievedContext

app = FastAPI(title="Python Learning Agent")


@app.on_event("startup")
def startup_event() -> None:
    init_chat_db()


@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "ok"}


@app.post("/ask", response_model=AskResponse)
def ask(request: AskRequest) -> AskResponse:
    try:
        result = answer_question(
            user_query=request.question,
            telegram_user_id=request.telegram_user_id,
            telegram_username=request.telegram_username,
            first_name=request.first_name,
            top_k=request.top_k,
            language=request.language,
            model_name=request.model_name,
        )

        return AskResponse(
            question=result["question"],
            answer=result["answer"],
```

```
            contexts=[RetrievedContext(**ctx) for ctx in
result["contexts"]],
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e)) from e
```

Лістинг коду файлу rag_pipeline.py

```
from typing import Any

from app.chat_history import add_message, get_last_messages,
upsert_user
from app.llm_ollama import ask_ollama
from app.retriever import retrieve_contexts


def format_history(history: list[dict[str, str]]) -> str:
    if not history:
        return "Історія відсутня."

    lines = []
    for msg in history:
        role = "User" if msg["role"] == "user" else "Assistant"
        lines.append(f"{role}: {msg['content']}")
    return "\n".join(lines)


def build_rag_prompt(
    user_query: str,
    contexts: list[dict[str, Any]],
    history: list[dict[str, str]],
) -> str:
    history_block = format_history(history)

    if not contexts:
        return f"""
Ти AI-асистент для навчання Python.

Останні 5 повідомлень діалогу:
{history_block}

Поточний запит користувача:
{user_query}

У базі знань не знайдено достатнього контексту.
Чесно скажи про це, не вигадуй інформацію і попроси уточнення.
""".strip()

    context_blocks = []
    for i, ctx in enumerate(contexts, start=1):
        block = f"""
[Context {i}]
```

```
Section number: {ctx.get("number")}
Title: {ctx.get("title")}
URL: {ctx.get("page_url")}
Content:
{ctx.get("text")}
""".strip()
        context_blocks.append(block)

    joined_context = "\n\n".join(context_blocks)

    return f"""
Ти AI-асистент для допомоги у вивченні Python.

Правила:
1. Враховуй попередні повідомлення діалогу.
2. Пояснюй просто і зрозуміло.
3. Якщо доречно, додай короткий приклад коду.
4. В кінці коротко вкажи використані секції документації.
5. В відповіді не потрібно вказувати фрази по типу, на основі
контексту і тд, відповідь потрібно надавати максимально природньо,
щоб не виглядало що це AI.

Останні 5 повідомлень діалогу:
{history_block}

Поточний запит користувача:
{user_query}

Контекст:
{joined_context}

Сформуй відповідь українською мовою.
""".strip()


def answer_question(
    user_query: str,
    telegram_user_id: int,
    telegram_username: str | None = None,
    first_name: str | None = None,
    top_k: int = 5,
    language: str | None = "en",
    model_name: str | None = None,
) -> dict[str, Any]:
    upsert_user(
        telegram_user_id=telegram_user_id,
        telegram_username=telegram_username,
        first_name=first_name,
    )

    history = get_last_messages(telegram_user_id, limit=5)

    contexts = retrieve_contexts(
```

```
        query=user_query,
        top_k=top_k,
        language=language,
    )

    prompt = build_rag_prompt(
        user_query=user_query,
        contexts=contexts,
        history=history,
    )

    answer = ask_ollama(prompt, model_name=model_name)

    add_message(telegram_user_id, "user", user_query)
    add_message(telegram_user_id, "assistant", answer)

    return {
        "question": user_query,
        "answer": answer,
        "contexts": contexts,
    }
```

Лістинг коду файлу retriever.py

```
from typing import Any

from qdrant_client import models

from app.config import COLLECTION_NAME, DEFAULT_SOURCE,
SCORE_THRESHOLD, TOP_K
from app.embeddings import embed_text
from app.vector_store import get_qdrant_client


def retrieve_contexts(
    query: str,
    top_k: int = TOP_K,
    language: str | None = None,
    source: str | None = DEFAULT_SOURCE,
) -> list[dict[str, Any]]:
    client = get_qdrant_client()
    query_vector = embed_text(query)

    must_conditions = []

    if source:
        must_conditions.append(
            models.FieldCondition(
                key="source",
                match=models.MatchValue(value=source),
            )
        )
```

```
    if language:
        must_conditions.append(
            models.FieldCondition(
                key="language",
                match=models.MatchValue(value=language),
            )
        )

    query_filter = models.Filter(must=must_conditions) if
must_conditions else None

    results = client.query_points(
        collection_name=COLLECTION_NAME,
        query=query_vector,
        limit=top_k,
        query_filter=query_filter,
        with_payload=True,
        with_vectors=False,
        score_threshold=SCORE_THRESHOLD,
    )

    contexts = []
    for point in results.points:
        payload = point.payload or {}
        contexts.append({
            "score": float(point.score),
            "number": payload.get("number"),
            "title": payload.get("title"),
            "heading_text": payload.get("heading_text"),
            "page_url": payload.get("page_url"),
            "anchor": payload.get("anchor"),
            "text": payload.get("text", ""),
            "language": payload.get("language"),
            "has_code": payload.get("has_code"),
        })

    return contexts
```

Лістинг коду файлу schemas.py

```
from typing import List, Optional
from pydantic import BaseModel, Field


class AskRequest(BaseModel):
    question: str = Field(..., min_length=1)
    top_k: int = 5
    language: Optional[str] = "ua"
    model_name: Optional[str] = None
    telegram_user_id: int
    telegram_username: Optional[str] = None
```

```
    first_name: Optional[str] = None


class RetrievedContext(BaseModel):
    score: float
    number: Optional[str] = None
    title: Optional[str] = None
    heading_text: Optional[str] = None
    page_url: Optional[str] = None
    anchor: Optional[str] = None
    text: str
    language: Optional[str] = None
    has_code: Optional[bool] = None


class AskResponse(BaseModel):
    question: str
    answer: str
    contexts: List[RetrievedContext]
```

Лістинг коду файлу vector_store.py

```
from qdrant_client import QdrantClient, models

from app.config import COLLECTION_NAME, QDRANT_PATH, VECTOR_SIZE


def get_qdrant_client() -> QdrantClient:
    return QdrantClient(path=QDRANT_PATH)


def recreate_collection(client: QdrantClient) -> None:
    if client.collection_exists(COLLECTION_NAME):
        client.delete_collection(COLLECTION_NAME)

    client.create_collection(
        collection_name=COLLECTION_NAME,
        vectors_config=models.VectorParams(
            size=VECTOR_SIZE,
            distance=models.Distance.COSINE,
        ),
    )

    client.create_payload_index(
        collection_name=COLLECTION_NAME,
        field_name="source",
        field_schema=models.PayloadSchemaType.KEYWORD,
    )
    client.create_payload_index(
        collection_name=COLLECTION_NAME,
        field_name="language",
        field_schema=models.PayloadSchemaType.KEYWORD,
```

```
    )
    client.create_payload_index(
        collection_name=COLLECTION_NAME,
        field_name="number",
        field_schema=models.PayloadSchemaType.KEYWORD,
    )
    client.create_payload_index(
        collection_name=COLLECTION_NAME,
        field_name="has_code",
        field_schema=models.PayloadSchemaType.BOOL,
    )
```

Лістинг коду файлу telegram_bot.py

```
import html
import logging
import requests
from typing import List

from telegram import Update
from telegram.constants import ChatAction, ParseMode
from telegram.ext import (
    ApplicationBuilder,
    CommandHandler,
    ContextTypes,
    MessageHandler,
    filters,
)

from app.config import BACKEND_ASK_URL, TELEGRAM_BOT_TOKEN

logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(name)s | %(message)s",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)

LAST_SOURCES: dict[int, List[dict]] = {}

USER_LANG: dict[int, str] = {}


def split_text(text: str, max_len: int = 3500) -> List[str]:
    text = text.strip()
    if len(text) <= max_len:
        return [text]

    parts = []
    current = ""

    for paragraph in text.split("\n\n"):
        paragraph = paragraph.strip()
```

```
        if not paragraph:
            continue

        candidate = f"{current}\n\n{paragraph}".strip() if current
else paragraph
        if len(candidate) <= max_len:
            current = candidate
        else:
            if current:
                parts.append(current)
            if len(paragraph) <= max_len:
                current = paragraph
            else:
                # fallback: ріжемо грубо
                for i in range(0, len(paragraph), max_len):
                    parts.append(paragraph[i:i + max_len])
                current = ""

    if current:
        parts.append(current)

    return parts


async def start_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message or not update.effective_user:
        return

    USER_LANG.setdefault(update.effective_user.id, "uk")

    text = (
        "Привіт. Я Telegram-бот для допомоги у вивченні
Python.\n\n"
        "Я шукаю інформацію в документації Python і відповідаю на
основі знайденого контексту.\n\n"
        "Доступні команди:\n"
        "/help — допомога\n"
        "/reset — очистити історію діалогу\n"
        "/lang uk — відповідати українською\n"
        "/lang en — відповідати англійською\n"
        "/sources — показати джерела останньої відповіді"
    )
    await update.message.reply_text(text)


async def help_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message:
        return

    text = (
        "Напиши мені питання про Python, наприклад:\n"
```

```
        "• Як працюють класи в Python?\n"
        "• Що таке interactive mode?\n"
        "• Як працює range()?\n\n"
        "Я використаю документацію Python, знайду релевантні
фрагменти і сформую відповідь."
    )
    await update.message.reply_text(text)


async def lang_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message or not update.effective_user:
        return

    if not context.args:
        await update.message.reply_text("Використання: /lang uk
або /lang en")
        return

    lang = context.args[0].strip().lower()
    if lang not in {"uk", "en"}:
        await update.message.reply_text("Підтримуються лише: uk,
en")
        return

    USER_LANG[update.effective_user.id] = lang
    await update.message.reply_text(f"Мову відповідей змінено на:
{lang}")


async def reset_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message or not update.effective_user:
        return

    user = update.effective_user

    LAST_SOURCES.pop(user.id, None)

    await update.message.reply_text(
        "Локальний кеш останніх джерел очищено. "
        "Для повного очищення історії в БД бажано додати окремий
backend endpoint /reset."
    )


async def sources_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message or not update.effective_user:
        return

    user_id = update.effective_user.id
    sources = LAST_SOURCES.get(user_id)
```

```
    if not sources:
        await update.message.reply_text("Ще немає джерел для
показу.")
        return

    lines = ["Останні використані джерела:\n"]
    for i, src in enumerate(sources, start=1):
        title = src.get("title") or "Без назви"
        number = src.get("number") or "?"
        page_url = src.get("page_url") or ""
        lines.append(f"{i}. [{number}] {title}\n{page_url}")

    text = "\n\n".join(lines)
    for part in split_text(text):
        await update.message.reply_text(part)


async def handle_message(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    if not update.message or not update.effective_user:
        return

    user = update.effective_user
    question = update.message.text.strip()

    if not question:
        await update.message.reply_text("Будь ласка, надішли
текстове питання.")
        return

    selected_lang = USER_LANG.get(user.id, "uk")

    await update.message.chat.send_action(ChatAction.TYPING)

    payload = {
        "question": question,
        "top_k": 5,
        "language": selected_lang,
        "model_name": "gemma3:27b",
        "telegram_user_id": user.id,
        "telegram_username": user.username,
        "first_name": user.first_name,
    }

    try:
        response = requests.post(
            BACKEND_ASK_URL,
            json=payload,
            timeout=300,
        )
        response.raise_for_status()
        data = response.json()
```

```
        answer = data.get("answer", "Не вдалося сформувати
відповідь.")
        contexts = data.get("contexts", [])
        LAST_SOURCES[user.id] = contexts

        for part in split_text(answer):
            await update.message.reply_text(part)

    except requests.HTTPError as e:
        logger.exception("HTTP error from backend")
        await update.message.reply_text(f"Помилка бекенда: {e}")
    except requests.RequestException as e:
        logger.exception("Request error")
        await update.message.reply_text(
            "Не вдалося звернутися до бекенда. Перевір, чи
запущений FastAPI сервер."
        )
    except Exception as e:
        logger.exception("Unexpected bot error")
        await update.message.reply_text(f"Несподівана помилка:
{e}")


async def error_handler(update: object, context:
ContextTypes.DEFAULT_TYPE) -> None:
    logger.exception("Telegram bot error", exc_info=context.error)


def main() -> None:
    app = ApplicationBuilder().token(TELEGRAM_BOT_TOKEN).build()

    app.add_handler(CommandHandler("start", start_command))
    app.add_handler(CommandHandler("help", help_command))
    app.add_handler(CommandHandler("lang", lang_command))
    app.add_handler(CommandHandler("reset", reset_command))
    app.add_handler(CommandHandler("sources", sources_command))
    app.add_handler(MessageHandler(filters.TEXT &
~filters.COMMAND, handle_message))

    app.add_error_handler(error_handler)

    logger.info("Telegram bot started")
    app.run_polling()


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

Лістинг коду файлу requirements.txt

```
fastapi==0.135.3
uvicorn==0.42.0
qdrant-client==1.17.0
sentence-transformers==5.3.0
requests==2.32.5
pydantic==2.12.5
python-telegram-bot==22.7
ollama==0.6.1
```

ДОДАТОК Б

Тези доповіді на конференції

Міністерство освіти і науки України
Тернопільський національний технічний університет
імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет
імені Івана Франка
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет
Чернівецький національний університет
імені Юрія Федьковича
Вроцлавський економічний університет (Польща)
Університет технологій та економіки
імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

IX МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

24-25 квітня 2026 р.

(збірник тез конференції)

Тернопіль 2026

УДК 004.42
Хоренко В. – ст. гр. СП-41
Тернопільський національний технічний університет імені Івана Пулюя

РОЗРОБКА ІНТЕЛЕКТУАЛЬНОГО AI-АСИСТЕНТА З КОНСУЛЬТУВАННЯ МОВИ ПРОГРАМУВАННЯ PYTHON НА ОСНОВІ QDRANT ТА GEMMA3

Науковий керівник: канд. техн. наук, доц. Багрій-Заяць О. А.

Horenko V.
Ternopil Ivan Puluj National Technical University

DEVELOPMENT OF AN INTELLIGENT AI ASSISTANT TO CONSULTING PYTHON PROGRAMMING LANGUAGE BASED ON QDRANT AND GEMMA3

Supervisor: O. A. Bahrii-Zaiats, Ph.D. in Engineering, Associate Professor

Ключові слова: штучний інтелект, Python, Telegram-бот, Qdrant, великі мовні моделі
Keywords: artificial intelligence, Python, Telegram bot, Qdrant, large language models

У сучасних умовах вивчення програмування значна кількість користувачів стикається з труднощами під час пошуку зрозумілих, точних і контекстно релевантних відповідей на запитання щодо мови програмування Python. Звичайний пошук у документації або на тематичних ресурсах часто потребує значних витрат часу та не забезпечує зручного діалогового формату взаємодії. Метою роботи є розробка інтелектуального AI-асистента, здатного відповідати на запитання з мови програмування Python на основі офіційної документації та доступної інформації в мережі Інтернет, а також підтримувати контекст попереднього діалогу. Новизна роботи полягає у поєднанні великої мовної моделі, векторної бази знань та зручного Telegram-інтерфейсу для створення інтерактивного помічника з функціями менторської підтримки.

У роботі запропоновано підхід до побудови інтелектуального асистента, що базується на використанні офіційної документації Python як основного джерела знань. Для реалізації пошуку релевантної інформації було використано векторну базу даних Qdrant, у якій зберігаються векторні представлення фрагментів документації. Як мовну модель застосовано gemma3:27b через бібліотеку Ollama, що забезпечує генерацію змістовних відповідей на основі знайденого контексту. Основна ідея підходу полягає в тому, щоб поєднати можливості семантичного пошуку та генеративного штучного інтелекту для надання точних, структурованих і зрозумілих пояснень з програмування.

Розроблена система дозволяє користувачу ставити запитання у природній формі, отримувати відповіді щодо синтаксису, конструкцій, бібліотек і принципів роботи Python, а також продовжувати діалог із збереженням історії спілкування. Це дає змогу використовувати асистента не лише як довідкову систему, а і як персонального помічника або ментора під час навчання. Реалізація у форматі Telegram-бота забезпечує зручний і доступний спосіб взаємодії без необхідності встановлення додаткового спеціалізованого програмного забезпечення. Такий підхід підвищує практичну цінність системи та спрощує її використання широким колом користувачів.

Перевагою запропонованого рішення є використання офіційної документації Python як перевіреного джерела знань, що підвищує достовірність відповідей. Застосування векторного пошуку в Qdrant дозволяє знаходити семантично близькі фрагменти навіть тоді, коли формулювання користувача не збігається буквально з текстом документації. Велика мовна модель gemma3:27b забезпечує природність формулювання відповідей і можливість пояснювати складні теми більш доступною мовою. Наявність механізму збереження історії діалогів дозволяє підтримувати контекст спілкування, що особливо важливо під час поетапного навчання, уточнень і розгляду складних прикладів коду.

У процесі розробки було реалізовано повний цикл взаємодії користувача із системою: прийом повідомлення через Telegram-бота, пошук релевантних даних у векторній базі, передача контексту до мовної моделі та генерація відповіді. Під час тестування асистент успішно відповідав на типові запитання, пов'язані з основами Python, функціями, структурами даних, умовними конструкціями, циклами, обробкою винятків та іншими аспектами мови. Крім того, система зберігала історію діалогів, що дозволяло продовжувати обговорення в межах попередньої теми та робило взаємодію більш наближеною до спілкування з реальним викладачем або наставником.

У результаті роботи створено інтелектуального AI-асистента з мови програмування Python, готового до практичного використання в навчальних цілях. Запропоноване рішення дозволяє спростити доступ до знань, підвищити зручність вивчення програмування та надати користувачам інструмент для швидкого отримання пояснень у діалоговому форматі. Практична цінність роботи полягає у можливості використання розробленої системи як навчального помічника для студентів, початківців і всіх, хто вивчає Python.

ДОДАТОК В

Посилання на Github

https://github.com/vlad-horenko/python-assistant