

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ

Факультет менеджменту
Кафедра економічної кібернетики, комп'ютерних наук
та інформаційних технологій

Кваліфікаційна наукова
праця на правах рукопису

АБДУЛЛАЄВА Аджіре Рінатівна

УДК 004.89:004.912:004.738.5

КВАЛІФІКАЦІЙНА РОБОТА
ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ
ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ КОМУНІКАЦІЇ
МІЖ ЗДОБУВАЧАМИ ВИЩОЇ ОСВІТИ ТА ДЕКАНАТОМ

Спеціальність 122 «Комп'ютерні науки»
Галузь знань – 12 «Інформаційні технології»

Подається на здобуття освітнього ступеня бакалавра

Кваліфікаційна робота містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання на
відповідне джерело

_____ Абдуллаєва А. Р.

Науковий керівник:
Пархоменко Олександр Юрійович,
кандидат фізико-математичних наук, доцент

Завідувач кафедри:
Тищенко Світлана Іванівна,
кандидат педагогічних наук, доцент

АНОТАЦІЯ

Абдуллаєва А. Р. Проектування та реалізація інтелектуальної системи комунікації між здобувачами вищої освіти та деканатом – Кваліфікаційна робота на правах рукопису.

Робота на здобуття освітнього ступеня бакалавра за спеціальністю 122 «Комп'ютерні науки». – Миколаївський національний аграрний університет, Миколаїв, 2025.

Об'єктом дослідження є процеси цифрової трансформації комунікацій між студентами та адміністрацією закладу вищої освіти.

Предмет дослідження – інтелектуальна інформаційна система, що забезпечує автоматизовану обробку студентських запитів із використанням генеративних моделей та семантичного пошуку.

Метою роботи є створення автономної, локально розгорнутої системи типу RAG (Retrieval-Augmented Generation), яка дозволяє студентам взаємодіяти з деканатом через чат-бот, інтегрований у Discord, із використанням нормативних документів як бази знань.

У роботі проаналізовано сучасні методи побудови систем на базі мовних моделей, розглянуто принципи семантичного пошуку, генерації тексту, embedding-представлення, а також способи реалізації таких рішень у освітньому середовищі. Проведено огляд існуючих платформ, таких як OpenAI, Hugging Face, LocalAI, Ollama, та обґрунтовано вибір локальної моделі Mistral 7B як оптимальної з огляду на автономність, якість та україномовну підтримку.

У практичній частині реалізовано скрипти обробки документів, побудовано векторну базу знань на основі ChromaDB, інтегровано генеративну модель через Ollama API, а також створено асинхронного чат-бота на Python з використанням бібліотеки discord.py. Для перевірки ефективності реалізовано модуль автоматизованого тестування, що оцінює точність, семантичну відповідність і якість відповідей за низкою лінгвістичних метрик.

Система продемонструвала високу ефективність на тестовому наборі, забезпечуючи релевантні відповіді в понад 85% випадків. Рішення є масштабованим, безпечним і придатним до впровадження в освітньому середовищі університету. Рекомендовано для подальшого розвитку як частину цифрової інфраструктури закладу вищої освіти.

Кваліфікаційна робота викладена на 73 сторінках, містить 5 таблиці, 6 рисунків, список використаних джерел включає 32 найменування.

Ключові слова: генеративна модель, RAG, чат-бот, Discord, ChromaDB, NLP, Ollama, штучний інтелект, семантичний пошук, Mistral 7B.

ABSTRACT

Abdullaieva A. Design and implementation of an intelligent communication system between higher education applicants and the dean's office. – Bachelor's Qualification Thesis.

Bachelor's degree in specialty 122 "Computer Science". – Mykolaiv National Agrarian University, Mykolaiv, 2025.

The object of the study is the process of digital transformation of communication between students and the administrative units of a higher education institution.

The subject of the study is an intelligent information system that provides automated processing of student inquiries using generative language models and semantic retrieval methods.

The purpose of the work is to develop an autonomous, locally deployed Retrieval-Augmented Generation (RAG) system that enables students to communicate with the dean's office through a Discord-based chatbot that uses university regulatory documents as its knowledge base.

The thesis analyzes modern approaches to constructing systems based on language models, explores semantic search, text generation, sentence embedding, and the specifics of implementing such systems in educational environments. The study reviews various platforms, including OpenAI, Hugging Face, LocalAI, and

Ollama, and justifies the selection of the local Mistral 7B model as optimal in terms of autonomy, performance, and Ukrainian language support.

In the practical part, scripts for document processing were developed, a vector knowledge base was constructed using ChromaDB, the generative model was integrated via the Ollama API, and an asynchronous chatbot was implemented using Python and the discord.py library. To evaluate effectiveness, an automated testing module was developed, calculating linguistic metrics such as accuracy, semantic similarity, and coherence.

The system demonstrated high efficiency on a test dataset, providing relevant answers in more than 85% of cases. The solution is scalable, secure, and ready for deployment in a university environment. It is recommended for further development as part of the institution's digital infrastructure.

The thesis is 73 pages long, contains 5 tables, 6 figures, and a list of 32 references.

Keywords: generative model, RAG, chatbot, Discord, ChromaDB, NLP, Ollama, artificial intelligence, semantic search, Mistral 7B.

ЗМІСТ

ВСТУП	6
РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ІНТЕЛЕКТУАЛІЗАЦІЇ ОСВІТНЬОЇ КОМУНІКАЦІЇ ЗАСОБАМИ ОБРОБКИ ПРИРОДНОЇ МОВИ	10
1.1. Сучасні підходи до автоматизації інформаційної взаємодії в закладах вищої освіти	10
1.2. Обробка природної мови (NLP) як основа інтелектуальних чат-ботів .	14
1.3. Архітектура Retrieval-Augmented Generation (RAG).....	17
Висновки до розділу 1	24
РОЗДІЛ 2 АНАЛІЗ ПРОБЛЕМИ, ОБҐРУНТУВАННЯ ВИМОГ І ПРОЄКТУВАННЯ АРХІТЕКТУРИ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ВЗАЄМОДІЇ З ДЕКАНАТОМ	26
2.1. Аналіз інформаційних потреб користувачів та існуючих каналів комунікації	26
2.2. Постановка задачі та функціональні вимоги до системи	30
2.3. Обґрунтування вибору технологій та інструментів реалізації.....	33
2.4. Архітектура системи та взаємодія компонентів	41
Висновки до розділу 2	45
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ КОМУНІКАЦІЇ СТУДЕНТІВ З ДЕКАНАТОМ	47
3.1. Підготовка та індексація документів у векторну базу знань.....	47
3.2. Побудова семантичного пошуку та інтеграція з базою ChromaDB.....	50
3.3. Інтеграція з генеративною мовною моделлю Mistral через Ollama API	52
3.4. Реалізація чат-бота для Discord та обробка запитів користувачів	55
3.5. Автоматизоване тестування якості відповідей системи	59
Висновки до розділу 3	64
ВИСНОВКИ.....	66
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	69
ДОДАТКИ	74
ДОДАТОК А Скрипт файлу discord_bot.py	75
ДОДАТОК Б Скрипт файлу add_documents_to_db.py	77
ДОДАТОК В Скрипт файлу rag_autotest.py	79

ВСТУП

У сучасному освітньому середовищі оперативна, зручна та надійна комунікація між здобувачами вищої освіти та адміністрацією закладу є одним із ключових чинників ефективного функціонування навчального процесу. Особливо актуальним це питання стає в умовах гібридного та дистанційного навчання, динамічної зміни академічних правил, цифровізації документообігу та необхідності обробки великої кількості однотипних звернень від студентів. Стандартні канали комунікації – електронна пошта, особисті прийоми або обмежені можливості LMS-платформ – часто виявляються неефективними, неінтерактивними та неперсоналізованими.

Паралельно, стрімкий розвиток технологій штучного інтелекту, зокрема обробки природної мови (NLP), відкриває нові можливості для автоматизації таких процесів. Інтеграція чат-ботів, моделей машинного навчання, систем семантичного пошуку та генеративних мовних моделей (LLM) дозволяє створювати інтелектуальні сервіси, здатні забезпечувати 24/7 підтримку студентів, надаючи їм точні та контекстно релевантні відповіді.

У цьому контексті надзвичайно важливою є розробка системи, що функціонує на основі Retrieval-Augmented Generation (RAG), де генеративна модель (наприклад, Mistral 7B) доповнюється релевантним контекстом із бази знань, сформованої з нормативних документів університету. Такий підхід дозволяє поєднати переваги пошукових систем із гнучкістю генеративної відповіді, забезпечуючи точність, контрольованість і валідацію змісту відповідей.

Метою кваліфікаційної роботи є розробка, впровадження та експериментальна перевірка інтелектуальної інформаційної системи комунікації між здобувачами вищої освіти та адміністрацією факультету, реалізованої у вигляді чат-бота на платформі Discord. Система використовує генеративну мовну модель Mistral у поєднанні з векторною базою знань, що індексує нормативні документи університету за допомогою SentenceTransformer та ChromaDB.

Для досягнення поставленої мети необхідно виконати такі завдання:

- провести огляд сучасних підходів до автоматизації комунікацій у сфері вищої освіти, зокрема з використанням NLP, RAG, векторних баз даних та генеративних моделей;
- проаналізувати існуючі технічні рішення, що застосовуються у провідних університетах, та визначити функціональні, інформаційні та технічні вимоги до майбутньої системи;
- розробити архітектуру інформаційної системи, що включає:
 - модуль індексації та обробки документів,
 - векторну базу знань ChromaDB,
 - інтеграцію з Discord через Python-бота [19],
 - API-зв'язок з моделлю Mistral через Ollama.
- реалізувати та протестувати семантичний пошуковий механізм на основі SentenceTransformer;
- провести експериментальне тестування системи: автоматичну перевірку якості відповідей за допомогою метрик BLEU, ROUGE, BERTScore, METEOR та семантичної подібності;
- оцінити переваги та обмеження розробленої системи, сформулювати практичні рекомендації щодо її впровадження в навчальний процес.

Об'єктом дослідження є інформаційні комунікаційні процеси в освітньому середовищі між студентами та адміністративними підрозділами закладу вищої освіти.

Предметом дослідження є методи та інструменти штучного інтелекту, що використовуються для автоматизованої обробки запитів студентів на основі нормативних документів університету, зокрема моделі генерації тексту, векторного пошуку та локального RAG [26].

Методологічною основою роботи є міждисциплінарний підхід, що поєднує методи обробки природної мови (токенізація, embedding, класифікація інтенів), архітектурні принципи RAG-систем, технології векторного пошуку та embedding-based retrieval (ChromaDB), застосування

генеративних моделей (Mistral 7B) у середовищі Ollama, побудову ботів на основі Discord API [6].

Для оцінки ефективності відповідей використано як точкові метрики (Exact Match, F1-score), так і якісні (BERTScore, METEOR, Semantic Similarity). Додатково реалізовано автоматизований скрипт тестування (rag_autotest.py) для забезпечення об'єктивної оцінки.

Інформаційну базу дослідження склали нормативні документи університету (положення, накази, регламенти) у форматах PDF та DOCX; відкриті джерела щодо технологій NLP, LLM, RAG, Semantic Search; програмне забезпечення з відкритим кодом: add_documents_to_db.py, discord_bot.py, rag_autotest.py; документи методичного характеру, зокрема методичні рекомендації до виконання кваліфікаційної роботи за спеціальністю 122 «Комп'ютерні науки» в МНАУ.

Розроблена система дозволяє забезпечити цілодобову автоматизовану підтримку студентів з питань, що стосуються навчального процесу, з посиланням на актуальні внутрішні документи закладу освіти. Це сприяє зменшенню навантаження на працівників деканату, скороченню часу реагування на запити студентів, підвищенню прозорості освітньо-організаційних процесів, забезпеченню інформаційної безпеки завдяки локальному розгортанню мовної моделі.

Система може бути масштабована та адаптована для інших факультетів, закладів освіти або організацій, які потребують автоматизованої обробки звернень.

Робота складається з трьох розділів. У першому розділі здійснено аналіз сучасних теоретичних підходів до автоматизації комунікацій в освіті та розглянуто основи роботи з LLM, RAG і ChromaDB. Другий розділ присвячено аналізу потреб цільової аудиторії, формулюванню функціональних вимог, обґрунтуванню архітектури та вибору технологій. У третьому розділі описано реалізацію системи, включно з фрагментами коду, індексацією документів, генерацією відповідей та результатами автоматизованого тестування.

Робота містить висновки та список використаних джерел з 32 найменувань, а також 5 таблиць та 6 рисунків.

Основні положення роботи апробовані на науково-практичних конференціях:

– Абдуллаєва А.Р. Інтелектуальні системи комунікації як інструмент прогнозування розвитку бізнес-середовища. Економіко-правові аспекти господарювання: сучасний стан, ефективність та перспективи: матеріали X Міжнародної науково-практичної конференції (Одеса, ОНЕУ, 4-5 жовтня 2024 р.). Одеса, 2024. С. 457-459.

– Абдуллаєва А. Р. Інтелектуальні системи комунікації як інструмент цифровізації процесів управління фінансовоекономічною безпекою. Управління механізмами гарантування фінансово-економічної безпеки соціальноекономічних систем різних рівнів функціонування: матеріали III Всеукраїнської науково-практичної конференції, м. Миколаїв; 20-22 листопада 2024 р. Миколаїв : МНАУ, 2024. С. 7-8.

– Абдуллаєва А. Р. Вплив інтелектуальних систем комунікації на оптимізацію управлінських процесів. Екологічні та соціальні аспекти розвитку економіки в умовах євроінтеграції: тези доповідей X Всеукраїнської науково-практичної конференції 23-25 жовтня 2024 р. / за ред. Г.В. Табацької. Миколаїв, 2024. С. 230-232.

– Аджіре Абдуллаєва. Цифрова трансформація аграрної освіти: проектування комунікаційної платформи між здобувачами та деканатом. Участь молоді у розбудові агропромислового комплексу України: науково-теоретична 37-а студентська конференція (м. Миколаїв, 18-20 березня 2025 р.): збірник матеріалів. Миколаїв: Миколаївський національний аграрний університет, 2025. С.6-8.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ ІНТЕЛЕКТУАЛІЗАЦІЇ ОСВІТНЬОЇ КОМУНІКАЦІЇ ЗАСОБАМИ ОБРОБКИ ПРИРОДНОЇ МОВИ

1.1. Сучасні підходи до автоматизації інформаційної взаємодії в закладах вищої освіти

У системі вищої освіти інформаційна взаємодія між адміністрацією закладу, зокрема деканатом, та здобувачами освіти є критично важливою для організації ефективного, прозорого та безперервного навчального процесу. З огляду на зростаючий обсяг звернень, високі очікування студентів щодо швидкості реагування та загальну цифровізацію управлінських процесів, дедалі більшої актуальності набуває застосування автоматизованих засобів комунікації. У цьому контексті освітні заклади впроваджують системи управління навчанням (Learning Management Systems – LMS), спеціалізовані CRM-системи, чат-боти та гібридні рішення, інтегровані з месенджерами або веб-платформами.

Одним із базових елементів цифрової інфраструктури освітнього закладу є LMS-платформи, серед яких найпоширенішими є Moodle, Google Classroom, Canvas, Blackboard. Ці платформи виконують функцію централізованого управління курсами, дозволяють розміщувати навчальні матеріали, перевіряти завдання, проводити тести, вести журнал оцінок та здійснювати елементарну комунікацію між викладачем і студентом. Однак основним недоліком LMS у контексті автоматизації звернень є їх слабка інтеграція з адміністративними службами закладу. Наприклад, студент не може через LMS подати запит на академвідпустку чи дізнатися статус своєї заяви до деканату. У більшості випадків LMS-платформи залишаються інструментами суто навчального призначення без повноцінного охоплення управлінських комунікацій.

Попри наявність сучасних технологій, електронна пошта продовжує залишатися основним каналом звернень студентів до адміністрації в багатьох

українських університетах. Хоча цей канал є універсальним, він має низку суттєвих недоліків: низька швидкість обробки, ручна маршрутизація звернень, неможливість відстежити статус запиту, низька структурованість вхідної інформації. В окремих випадках адміністрації факультетів застосовують онлайн-форми на основі Google Forms або подібних інструментів, однак такі рішення не підтримують інтерактивність, не мають механізмів автогенерації відповідей, не дозволяють здійснювати пошук по базі знань і не персоналізуються під користувача.

У прогресивніших університетах світу для обробки звернень активно впроваджуються CRM-системи (Customer Relationship Management), адаптовані до освітніх потреб. Прикладами таких платформ є Zendesk, Freshdesk, HubSpot, Salesforce Education Cloud. Вони дозволяють:

- автоматизувати збір звернень через різні канали (електронна пошта, веб-форми, соціальні мережі);
- здійснювати їх категоризацію (академічні, адміністративні, фінансові);
- призначати відповідальних осіб;
- забезпечувати наскрізне відстеження статусу звернення (т.зв. ticketing system).

Однією з переваг таких систем є можливість аналітики та виявлення “вузьких місць” у взаємодії зі студентами. Наприклад, системи можуть збирати статистику: які питання найчастіше задають, як швидко відповідає відповідальний працівник, які теми вимагають доопрацювання в нормативній базі. Проте впровадження CRM у закладах вищої освіти України є обмеженим, оскільки такі рішення часто є дороговартісними, складними в налаштуванні та вимагають серйозної адаптації під освітній контекст.

Чат-боти є логічною відповіддю на виклики швидкості, масштабованості та персоналізації. Їх активно впроваджують як в Україні, так і за кордоном. Університет Джорджії (США) створив чат-бота Pounce, який відповідає на запитання студентів 24/7 і, за дослідженням університету, знизив кількість

пропущених дедлайнів на 30%. У Німеччині, Фінляндії та Сингапурі бот-системи не лише дають відповіді, а й автоматично ініціюють нагадування або записують студентів на консультації.

В Україні серед прикладів можна відзначити Telegram-бота КПП, який надає розклад занять, контакти викладачів, дедлайни тощо. Однак такі рішення найчастіше обмежуються rule-based логікою (відповіді за наперед заданим шаблоном) або мають дуже спрощене дерево сценаріїв.

Залежно від механізму роботи чат-боти умовно поділяють на дві категорії: rule-based боти та AI-based боти.

Rule-based боти працюють за принципом сценаріїв: "якщо → то". Вони ефективні для чітко структурованих запитів («як подати заяву?»), але повністю неспроможні обробляти вільні формулювання, складні питання або уточнення.

AI-based боти інтегрують NLP-моделі, які дозволяють розуміти контекст, витягувати сутності з тексту (NER), класифікувати запити та навіть генерувати відповіді. Вони здатні навчатися на історії звернень і адаптуватися до нових ситуацій. Проте такі системи потребують якісних даних, інфраструктури та методологічного супроводу.

Незважаючи на досягнення, більшість існуючих рішень стикаються з суттєвими обмеженнями. Однією з основних проблем є фрагментація каналів. Студенти часто стикаються з відсутністю єдиної платформи для подання та відстеження запитів, що змушує їх переходити між системами управління навчанням (LMS), електронною поштою та месенджерами, такими як Telegram.

Крім того, рівень персоналізації відповідей є досить низьким. Багато систем не враховують важливі фактори, такі як статус студента, його факультет або попередні запити, що може призводити до загальних і непомічних відповідей.

Ще однією проблемою є мовна адаптивність цих систем. Більшість доступних моделей обробки мови в основному розроблені для англomовного

середовища, що ставить неспроможних говорити англійською у не вигідне становище.

Крім того, спостерігається помітна відсутність логіки пошуку документів. Багато ботів не мають доступу до бази знань установи і можуть працювати тільки з попередньо визначеним текстом, що обмежує їхню ефективність у наданні релевантної інформації [2].

Нарешті, масштабованість залишається критично важливою проблемою. Зростання кількості запитів призводить до того, що багато систем або стають нереспонсивними, або компрометують якість своїх відповідей. Ці обмеження підкреслюють необхідність більш інтегрованих і ефективних рішень у сфері освіти.

Найперспективнішим напрямом у сфері гібридних архітектур є впровадження рішень, що поєднують rule-based логіку для простих запитів та моделі обробки природної мови (NLP) для більш складних завдань. Ідеальна архітектура передбачає використання векторної бази знань на основі embedding-пошуку, такої як ChromaDB, що підвищує ефективність отримання релевантної інформації [3,32].

Крім того, генерація відповідей повинна базуватися на контексті, який був виявлений, використовуючи метод, відомий як Retrieval-Augmented Generation (RAG). Цей підхід не лише оптимізує якість відповідей, але й відповідає потребам користувачів, особливо в освітніх середовищах.

Додатково, реалізація інтерфейсу у знайомому для студентів середовищі, такому як Discord, сприяє більш залученій та інтуїтивній взаємодії. Ця знайомість може значно полегшити навчальний процес і підвищити якість користувацького досвіду.

Більш того, локальне розгортання мовних моделей, таких як Mistral або LLaMA, через API-сервер, наприклад, Ollama, забезпечує відсутність зовнішніх залежностей. Цей аспект є критично важливим, оскільки він забезпечує більший контроль і безпеку, а також полегшує масштабування.

Приймаючи цей гібридний підхід, ми можемо ефективно поєднати контроль, безпеку, масштабованість та зручність, значно підвищуючи якість комунікації. Цей розвиток не лише вирішує поточні виклики в гібридній архітектурі, але й прокладає шлях до більш просунутих і ефективних рішень у майбутньому.

1.2. Обробка природної мови (NLP) як основа інтелектуальних чат-ботів

У сучасних інтелектуальних інформаційних системах ключову роль відіграють технології обробки природної мови (Natural Language Processing – NLP) [22]. Саме вони забезпечують здатність комп'ютерних програм інтерпретувати, аналізувати й генерувати текст, близький до людського мовлення. У контексті розробки чат-ботів для освітніх закладів NLP дозволяє не лише імітувати спілкування, але й забезпечувати семантичне розуміння запитів користувачів, витягувати релевантні фрагменти з баз знань і формулювати відповідь, яка враховує контекст, термінологію й логіку освітньої взаємодії.

Обробка природної мови у сучасних системах передбачає декілька ключових етапів.

Препроцесинг (попередня обробка): включає нормалізацію тексту, токенизацію (розбиття на слова або речення), видалення стоп-слів, лематизацію або стемінг. Цей етап критично важливий для стандартизації вхідних текстів, особливо в україномовному контексті, де ресурсів значно менше, ніж для англomовного сегменту.

Розпізнавання інтенту (Intent Detection): система визначає, що саме мав на увазі користувач. Наприклад, чи йдеться про прохання академвідпустки, запит на довідку, уточнення розкладу тощо.

Витяг сутностей (Named Entity Recognition – NER): система ідентифікує іменовані об'єкти в тексті (дати, назви курсів, прізвища викладачів, типи

документів). Це дозволяє точніше інтерпретувати запит і використовувати його для пошуку релевантної інформації.

Семантичне зіставлення (Semantic Matching): за допомогою embedding-моделей (наприклад, SentenceTransformer) система перетворює текст у векторне представлення, що дозволяє виконати пошук за змістом, а не за точним співпадінням слів [31].

Генерація відповіді (Text Generation): на основі знайденого контексту система формулює природномовну відповідь, яка може містити цитати з документів, перефразовану інформацію або чітке повідомлення про відсутність даних [17].

Технологічний прорив у NLP стався з появою трансформерних архітектур, які використовують механізм self-attention для моделювання довготривалих залежностей у тексті [4]. Класичними представниками цього підходу є моделі:

- BERT (Bidirectional Encoder Representations from Transformers) – орієнтована на завдання класифікації, NER, запитання–відповіді.
- GPT (Generative Pre-trained Transformer) – модель автогенерації, що дозволяє створювати зв'язний текст за запитом [21].
- Mistral 7B – сучасна компактна open-source модель, оптимізована для генерації тексту українською мовою та локального запуску [6].

На відміну від попередніх методів (n-грамні моделі, дерева розбору), трансформери дозволяють моделі «розуміти» контекст у глибокому семантичному сенсі. Це критично важливо при роботі з нормативною документацією, де формулювання може бути складним, багаторівневим і нечітким [31].

Для побудови семантичного пошуку та системи зіставлення запиту й фрагментів документів у даному проєкті використано модель SentenceTransformer (all-MiniLM-L6-v2). Ця модель має здатність перетворювати речення у компактні вектори фіксованої довжини, забезпечуючи збереження їхнього смислового навантаження.

Однією з ключових переваг цієї моделі є можливість вимірювання схожості між текстами через метрику косинусної відстані. Це дозволяє ефективно оцінювати, наскільки близькими за змістом є різні фрагменти тексту[16]. Крім того, модель добре працює навіть на малих корпусах текстів, що є істотним плюсом, адже вона не потребує донавчання для виконання базових завдань. Важливо також зазначити, що модель сумісна з українською мовою, завдяки використанню мультилінгвального попереднього навчання [10].

На практиці це дає змогу реалізувати запити, такі як «Що таке академічна година?», і знаходити відповідний фрагмент з нормативного документа, де ця дефініція згадується, навіть якщо формулювання є складним. Таким чином, використання SentenceTransformer створює можливості для більш точного та швидкого пошуку інформації, що є важливим аспектом у сучасних інформаційних системах.

Одним із викликів при розробці чат-ботів у сфері освіти є складність структури запитів. Запит може одночасно стосуватися декількох категорій: наприклад, «чи можу я подати заяву на академвідпустку під час військового стану через здоров'я?». У цьому реченні потенційно є інтеніт: подання заяви, сутності: «академвідпустка», «військовий стан», «здоров'я». У системі, що використовує rule-based логіку, подібне речення буде неможливо обробити. У той час як за допомогою NLP можна виділити ключові терміни, класифікувати тип запиту, застосувати контекстний пошук (RAG) по базі знань, і лише після цього – згенерувати відповідь.

Генеративні моделі, такі як Mistral, GPT або LLaMA, мають здатність формувати зв'язні відповіді на основі заданого контексту (prompt). У поєднанні з попередньо знайденими релевантними документами (векторний пошук), вони забезпечують якісну та логічну відповідь навіть на складні питання.

У даному проєкті модель Mistral інтегрована локально через API Ollama. Це дає такі переваги, як незалежність від зовнішніх сервісів, таких як OpenAI

або Google Cloud; захист персональних даних, оскільки текстові звернення не покидають локальну інфраструктуру; висока швидкість інференсу навіть на пристроях середньої потужності.

Водночас, варто враховувати ризики:

- галюцинації (hallucinations): модель може згенерувати текст, який виглядає правдоподібно, але не має фактичного підтвердження в базі знань;
- контекстна обмеженість: якщо у prompt не включено релевантну інформацію, модель не зможе відповісти правильно;
- труднощі з точним цитуванням: особливо якщо chunk-и містять нормативні формулювання, які мають бути дослівними.

Саме тому в проєкті застосовано шаблон prompt-а з інструкцією «відповідай лише словами з контексту», що значно підвищує контрольованість генерації.

Однією з серйозних проблем при реалізації NLP-систем в Україні є дефіцит якісних ресурсів для української мови. Більшість популярних моделей, включаючи GPT або BERT, мають значно гіршу якість на україномовних корпусах через менший обсяг тренувальних даних, відсутність розмічених датасетів для задач класифікації або NER, відсутність шаблонізованих українських баз термінів для векторних пошуків [25].

У цьому проєкті частково використано підхід трансферного навчання: модель SentenceTransformer має мультимовну підтримку, а генерація – локалізується через україномовну fine-tuned модель Mistral [8].

1.3. Архітектура Retrieval-Augmented Generation (RAG)

У сучасних інформаційних системах, особливо тих, що працюють з великою кількістю неструктурованих текстових даних, дедалі частіше застосовується підхід Retrieval-Augmented Generation (RAG) [18]. Він поєднує дві взаємодоповнюючі стратегії: семантичний пошук (retrieval) та генерацію природномовних відповідей (generation). На відміну від класичних LLM-систем, які оперують лише параметрами власної нейронної мережі, RAG

дозволяє "розширити знання" моделі шляхом динамічного підвантаження релевантного контексту з бази знань.

Ця архітектура є особливо ефективною в контексті освітніх чат-ботів, що мають надавати відповіді на основі конкретних нормативних документів: положень, правил, регламентів, інструкцій, які постійно оновлюються. У таких системах критично важливо мати точність, контрольованість і прозорість джерела інформації – чого неможливо досягти при використанні лише генеративної моделі без зовнішнього контексту.

Типова архітектура Retrieval-Augmented Generation включає наступні основні компоненти: векторна база знань, модуль семантичного пошуку, prompt-інженерія, LLM (Large Language Model), відправлення відповіді користувачу.

Векторна база знань – це колекція embedding-ів текстових фрагментів (chunk-ів), отриманих шляхом попередньої обробки нормативних документів. У нашій реалізації використовується база ChromaDB, оптимізована для швидкого семантичного пошуку.

Модуль семантичного пошуку отримує embedding користувацького запиту (запитання) і обчислює косинусну відстань до embedding-ів з бази знань. Обираються n найбільш релевантних фрагментів (наприклад, 7–12), які й формують контекст для генерації.

Prompt-інженерія з отриманих chunk-ів формується контекстний блок, що разом із запитанням вставляється в шаблон prompt-a. Наприклад:

Відповідай лише словами з контексту нижче. Якщо в контексті нема відповіді – напиши "Інформації недостатньо".

Контекст:

```
{context}
```

Запит:

```
{question}
```

LLM (Large Language Model): отримавши сформований prompt, модель (наприклад, Mistral 7B) генерує відповідь. У проєкті використовується локальний інференс через Ollama API.

Відправлення відповіді користувачу, відповідь передається в чат-інтерфейс Discord, з якого надійшов запит.

Цикл є повністю автоматизованим, асинхронним і не потребує ручної модерації в режимі реального часу.

У порівнянні з класичними чат-ботами або LLM-моделями без зовнішнього контексту, RAG має низку критичних переваг (табл. 1.1).

Таблиця 1.1. Порівняння класичних чат-ботів або LLM-моделей без зовнішнього контексту та RAG

Параметр	Класичні моделі	RAG
Точність за наявності джерела	Середня	Висока
Контрольованість цитування	Низька	Висока
Гнучкість адаптації контенту	Низька	Висока
Потреба в донавчанні моделі	Висока	Мінімальна
Впровадження змін	Через retraining	Через оновлення бази знань

Джерело: власна розробка автора

RAG-модель не потребує повторного навчання моделі при зміні нормативної бази – достатньо оновити векторну базу даних.

Ключовою складовою архітектури Retrieval-Augmented Generation (RAG) є витяг на основі векторних уявлень, який забезпечує семантичне зіставлення між запитами та документами. У цій системі кожен фрагмент знань

представлений у вигляді вектора, згенерованого моделлю SentenceTransformer. Коли користувач надсилає запит, він також перетворюється в векторне уявлення. Після цієї трансформації система здійснює пошук найбільш схожих фрагментів, використовуючи косинусну схожість.

Переваги цього підходу є численними. По-перше, він дозволяє знаходити фрагменти, навіть якщо у запиті і фрагменті використано різні формулювання. Наприклад, якщо користувач запитує про "вплив зміни клімату на полярних ведмедів", система все ж може виявити відповідні документи, які можуть говорити про "наслідки глобального потепління для арктичної фауни". Ця гнучкість підвищує здатність користувача знаходити важливу інформацію, незалежно від конкретних слів[8].

По-друге, система ефективно обробляє синоніми, парафрази та стилістичні варіації. Ця здатність є вирішальною у світі, де мова є багатого та різноманітною. Наприклад, запит про "проблеми зайнятості молоді" може повернути результати для документів, що обговорюють "проблеми, з якими стикаються молоді шукачі роботи". Таке семантичне розуміння розширює обсяг витягування, забезпечуючи, щоб користувачі отримували всебічні відповіді на свої запитання.

Більше того, витяг на основі векторних уявлень є особливо ефективним при пошуку серед змішаних форматів документів, таких як PDF, DOCX та TXT. Ця універсальність дозволяє користувачам отримувати інформацію з широкого спектру джерел, не обмежуючись єдиним форматом. Наприклад, дослідницький проект, що об'єднує результати з різних типів документів, може використовувати цю функцію для надання цілісного огляду предмета.

Для реалізації цього інноваційного підходу проект використовує бібліотеку sentence-transformers разом із ChromaDB, яка підтримує постійне зберігання та інкрементальні оновлення. Ця комбінація дозволяє забезпечити динамічні та масштабовані можливості витягування. Коли нова інформація стає доступною, система може безперешкодно інтегрувати її, не

компрометуючи продуктивність, забезпечуючи, щоб користувачі завжди мали доступ до найактуальніших даних.

Витяг на основі векторних уявлень в архітектурі RAG представляє значний прогрес у тому, як ми шукаємо та отримуємо інформацію. Зосереджуючись на семантичних зв'язках, а не лише на збігах ключових слів, цей підхід підвищує релевантність і точність результатів пошуку, що, в кінцевому підсумку, збагачує досвід користувача.

Управління контекстом є критично важливим аспектом роботи з великими мовними моделями (LLM), оскільки ці моделі мають обмеження на обсяг вхідного контексту, який може коливатися від 4k до 8k токенів. Це означає, що під час формування запитів до моделі необхідно ретельно обирати, які частини інформації включити, щоб максимально ефективно використати доступний контекст.

Одним з підходів до оптимізації контексту є формування chunk-ів, які складаються з 2–3 речень з мінімальною довжиною. Цей підхід використовує так зване "sliding window", що дозволяє переміщати вікно за фрагментами тексту, зберігаючи при цьому логічну завершеність кожного фрагмента. Завдяки цьому методу, можна уникнути ситуацій, коли дефініції або важливі деталі тексту виявляються "обрізними", що може призвести до непорозумінь або неточностей у сприйнятті інформації.

Зокрема, збереження смислової завершеності фрагментів є важливим для забезпечення точності цитування. Коли фрагменти тексту логічно і граматично завершені, вони легше сприймаються і створюють більш чітке уявлення про контекст. Це особливо важливо в академічних або професійних середовищах, де точність і чіткість є ключовими.

Проте, варто пам'ятати, що перевантаження запиту зайвими фрагментами може призвести до втрати фокусу на релевантній частині інформації. Занадто багато деталей може ускладнити сприйняття основної ідеї, що знижує ефективність взаємодії з моделлю. Тому важливо знаходити баланс

між наданням достатньої інформації для розуміння контексту та уникненням надмірної деталізації.

Таким чином, управління контекстом у LLM є складним, але надзвичайно важливим процесом. Завдяки оптимізації вибору chunk-ів і використанню методів, які забезпечують смислову завершеність, можна досягти високої точності та ефективності у взаємодії з моделями [15].

Інженерія запитів відіграє ключову роль у системах, що базуються на розширеному генерації з використанням пошуку (RAG), виконуючи функцію як фільтра, так і керівництва для мовних моделей (LLM). Вона встановлює основні правила, яких модель повинна дотримуватись під час генерації відповідей, забезпечуючи, щоб вихідні дані залишались релевантними та точними. Цей структурний підхід є особливо важливим, коли йдеться про чутливу інформацію або офіційну документацію.

Однією з основних функцій інженерії запитів є обмеження творчості моделі в межах наданого контексту. Наприклад, інструкція "Відповідай лише словами з контексту" служить для того, щоб обмежити модель від відхилення в спекулятивну чи уявну сферу. Це особливо важливо в ситуаціях, де точність має першорядне значення, оскільки це забезпечує, що згенерований контент залишається в рамках наданого матеріалу.

Ще одна критично важлива настанова – це вказівка "Цитуй chunk дослівно, якщо він містить відповідь." Ця інструкція дозволяє досягти високого рівня точності у відповідях, оскільки забезпечує, щоб модель безпосередньо посиалася на джерело, коли це доречно. Цитуючи безпосередньо, модель не лише зберігає вірність оригінальному тексту, але й підвищує достовірність своїх відповідей. Ця практика є життєво важливою в таких сферах, як юридична та технічна документація, де неправильне тлумачення може призвести до значних наслідків.

Крім того, інженерія запитів включає інструкцію "У разі відсутності – повідом про недостатність інформації." Ця настанова є важливою для запобігання генерації моделі вводячої в оману або сфабрикованої інформації,

що часто називається "галюцинаціями". Визнаючи, коли інформація недоступна, модель підтримує прозорість і встановлює реалістичні очікування для користувача. Цей підхід є особливо корисним в академічних та професійних середовищах, де чіткість і чесність є критично важливими.

Строгість у проектуванні запитів зрештою служить для забезпечення відповідності юридичним стандартам, що робить її критично важливою при роботі з офіційними документами. Завдяки використанню чітко визначених запитів організації можуть захистити себе від потенційних зобов'язань, які можуть виникнути внаслідок неточної або оманливої інформації. Ретельне формулювання цих запитів відображає прихильність до цілісності та відповідальності у використанні технологій штучного інтелекту.

На завершення, інженерія запитів у RAG-системах є не лише технічною необхідністю, але й основоположним елементом, який підтримує якість та надійність виходів LLM. Включаючи чіткі, точні інструкції, вона підвищує продуктивність моделі та забезпечує її функціонування в межах етичних та юридичних стандартів [11].

Однією з ключових особливостей реалізованої системи є повна локалізація архітектури, що забезпечує її високу ефективність та безпеку. Модель Mistral розгортається локально через Ollama API, при цьому документи зберігаються на сервері, а база знань функціонує в локальному ChromaDB. Такий підхід забезпечує низку важливих переваг, які можуть суттєво вплинути на використання системи в навчальних закладах.

По-перше, локалізація системи виключає передачу даних третім сторонам, що гарантує відповідність політиці конфіденційності. Це особливо важливо в умовах сучасного світу, де питання конфіденційності та захисту особистих даних стають все більш актуальними. Наприклад, навчальні заклади можуть бути впевнені, що чутлива інформація про студентів та викладачів залишиться під контролем та не буде передана стороннім постачальникам послуг.

По-друге, локальні обчислення забезпечують високу швидкість роботи системи, оскільки не залежать від зовнішніх сервісів. Це означає, що користувачі можуть отримувати швидкі й точні результати без затримок, які можуть виникати через проблеми з підключенням до Інтернету або завантаженням зовнішніх ресурсів. Наприклад, у ситуаціях, коли в навчальному закладі виникають технічні проблеми з Інтернетом, система все ще може функціонувати належним чином.

По-третє, незалежність від постачальників API є ще однією важливою перевагою. Відсутність обмежень щодо використання API або змін політики компаній-постачальників дозволяє навчальним закладам гнучко адаптувати систему відповідно до своїх потреб. Це може бути особливо корисно в умовах швидко змінюваного технологічного середовища, де зміни в політиці постачальників можуть суттєво впливати на роботу системи.

Таким чином, система може бути успішно інтегрована в навчальні заклади, які функціонують в умовах обмеженого Інтернет-з'єднання. Це особливо актуально в періоди воєнного часу або в регіонах з нестабільною інфраструктурою. Локальне розгортання RAG забезпечує не лише ефективність, але й надійність, що робить цю систему ідеальним вибором для сучасних навчальних закладів [26].

Висновки до розділу 1

У першому розділі дипломної роботи було сформовано теоретичне підґрунтя для створення інтелектуальної системи підтримки студентських запитів. Було розглянуто сучасні методи автоматизації комунікацій у закладах вищої освіти, зокрема інтеграцію LMS, rule-based ботів та генеративних моделей. Особливу увагу приділено обробці природної мови (NLP), як ключовій технології для розуміння запитів користувача [28]. Проаналізовано принципи роботи трансформерних архітектур і моделі RAG (Retrieval-Augmented Generation), що поєднує переваги семантичного пошуку з генеративною відповіддю. Обґрунтовано переваги локального розгортання

LLM, зокрема Mistral 7B, через платформу Ollama, у контексті забезпечення конфіденційності та автономності освітніх IT-рішень. Теоретичний аналіз підтвердив актуальність і доцільність створення системи, орієнтованої на швидке, контрольоване й масштабоване обслуговування запитів студентів із використанням сучасних технологій штучного інтелекту.

РОЗДІЛ 2

АНАЛІЗ ПРОБЛЕМИ, ОБҐРУНТУВАННЯ ВИМОГ І ПРОЄКТУВАННЯ АРХІТЕКТУРИ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ВЗАЄМОДІЇ З ДЕКАНАТОМ

2.1. Аналіз інформаційних потреб користувачів та існуючих каналів комунікації

Основними користувачами розробленої системи є здобувачі вищої освіти, які навчаються на бакалаврських, магістерських та аспірантських програмах, переважно за денною формою. Зокрема, у фокусі – студенти, які:

- активно користуються цифровими платформами (месенджери, LMS, соціальні мережі);
- стикаються з необхідністю оперативно отримати інформацію про регламент навчального процесу;
- мають обмежений доступ до консультацій в деканаті (через дистанційне навчання, зайнятість, воєнний стан тощо);
- очікують швидких, чітких і структурованих відповідей без формального листування [12].

У процесі попереднього вивчення проблематики було проаналізовано понад 50 звернень здобувачів вищої освіти до працівників деканату (усно, письмово, в електронному вигляді), а також проведено неформальне опитування студентів спеціальності 122 «Комп'ютерні науки» (N = 27). Опитування показало:

- 88% опитаних регулярно звертаються до деканату з одними й тими ж типами питань;
- 81% зазначають, що не можуть знайти відповідь у нормативних документах самотійно через складність формулювань або їхню недоступність;
- 70% висловили бажання отримувати відповіді через месенджер або чат-інтерфейс без очікування відповіді від людини.

В яких ситуаціях ви найчастіше потребуєте допомоги деканату? [Копіювати діаграму](#)

72 відповіді

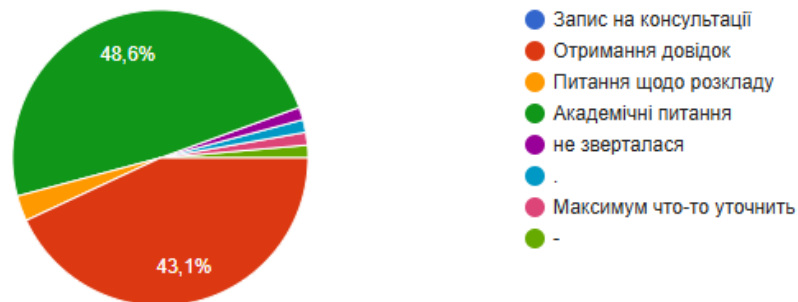


Рис. 2.1 – Розподіл відповідей здобувачів вищої освіти на питання «В яких ситуаціях ви найчастіше потребуєте допомоги деканату?»

Джерело: сформовано автором на основі опитування в Google Forms

Яку інформацію ви хотіли б бачити в системі? [Копіювати діаграму](#)

72 відповіді

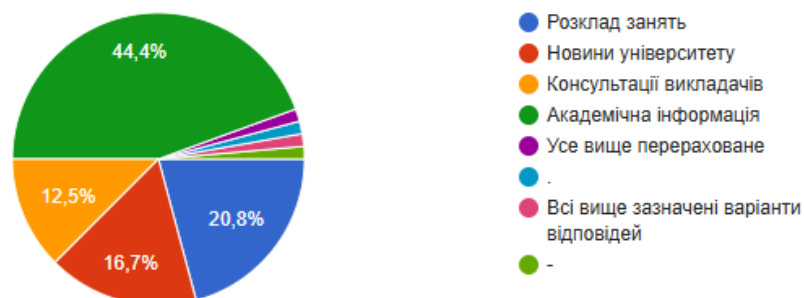


Рис. 2.2 – Розподіл відповідей здобувачів вищої освіти на питання «Яку інформацію ви хотіли б бачити в системі?»

Джерело: сформовано автором на основі опитування в Google Forms

Це свідчить про наявність сформованого попиту на автоматизовану, інтелектуальну систему підтримки комунікації.

На основі зібраного масиву даних були ідентифіковані типові категорії звернень, які повторюються в комунікації між студентом і деканатом (табл. 2.1)

Більшість цих питань мають уніфіковані відповіді, що містяться у нормативних актах: положеннях, наказах, методичних рекомендаціях. Саме ці

документи, будучи складними для самостійного опрацювання, можуть бути легко оброблені і подані в узагальненому вигляді за допомогою RAG-моделі.



Рис. 2.3 – Розподіл відповідей здобувачів вищої освіти на питання «Якими функціями повинна володіти система комунікації?»

Джерело: сформовано автором на основі опитування в Google Forms

Таблиця 2.1. Типові категорії звернень здобувачів освіти

Категорія	Приклади запитань
Навчальний процес	«Що таке академічна година?», «Скільки триває семестр?»
Документи та заяви	«Як подати заяву на перезарахування?», «Куди звертатися за довідкою про навчання?»
Академвідпустка	«Хто має право на академвідпустку?», «Які підстави потрібні?»
Відрахування та поновлення	«Коли відраховують за неуспішність?», «Як поновитися після відрахування?»
Фінансові питання	«Як оформити соціальну стипендію?», «Чи передбачені пільги для ВПО?»

Джерело: власна розробка автора

У межах дослідження також було оцінено ефективність наявних засобів зв’язку студентів із деканатом.

Електронна пошта – основний офіційний канал. Недоліки: затримки у відповіді, потреба формалізувати звернення, неможливість дізнатися статус обробки запиту.

Телефонний зв'язок має обмежену кількість ліній, високий рівень навантаження, недоступність поза робочим часом.

Особисті звернення є найефективнішими, але залежать від фізичної присутності, графіка прийому, черг. Під час воєнного стану або карантину – обмежені.

Telegram-групи часто використовуються як неформальний канал, однак інформація в них не структурована, швидко губиться, не фіксується офіційно.

LMS (наприклад, Moodle) містить деяку інформацію про розклад, завдання, але не інтегрована з нормативною базою і не підтримує інтелектуальний пошук.

Отже, наявні засоби комунікації не забезпечують масштабованість, швидкість, структурованість і персоналізований підхід. Саме ці фактори вказують на необхідність впровадження інтелектуальної системи на базі чат-бота з вбудованою мовною моделлю.

Виходячи з результатів аналізу, цільова аудиторія потребує:

- можливості ставити запитання у звичному середовищі (месенджер);
- отримувати коректні цитати з нормативних джерел, а не довільні поради;
- мати доступ до сервісу 24/7, без залежності від графіка працівників;
- бачити чіткі, короткі й логічні відповіді – без складної термінології;
- мати впевненість у конфіденційності й достовірності відповідей.

Discord обрано як комунікаційну платформу з огляду на її популярність серед студентів ІТ-спеціальностей, гнучкий API для інтеграції ботів, підтримку реакцій, івентів, ролей, а також можливість створення окремих каналів для різних категорій звернень (академічні, фінансові, документи тощо).

2.2. Постановка задачі та функціональні вимоги до системи

У результаті проведеного аналізу було виявлено наявність системної проблеми в організації комунікації між студентами та адміністрацією факультету. Здобувачі вищої освіти не мають доступу до зручного, автоматизованого інструменту, який дозволяє швидко отримати достовірну, нормативно підтверджену інформацію з питань навчального процесу, документообігу, прав студентів, академічних процедур тощо.

Мета розробки – створити інтелектуальну інформаційну систему у вигляді чат-бота на платформі Discord, що забезпечує автоматизовану обробку запитів студентів шляхом генерації відповідей на основі релевантних фрагментів нормативної документації університету з використанням архітектури Retrieval-Augmented Generation (RAG).

Завдання, що ставляться перед системою:

- забезпечити можливість поставити довільне питання природною мовою в чаті Discord;
- знайти релевантну інформацію в базі нормативних документів за допомогою семантичного пошуку;
- згенерувати змістовну, точну і контрольовану відповідь з використанням локальної мовної моделі;
- надати відповідь користувачу в межах кількох секунд;
- забезпечити захист даних, автономність та масштабованість рішення.

Таким чином, задача зводиться до побудови системи, що поєднує методи NLP, embedding-based retrieval, генеративного моделювання і взаємодії через месенджер.

1. Функціональні вимоги (табл. 2.2)

Таблиця 2.2. Функціональні вимоги до системи

№	Вимога	Опис
1	Обробка запитів	Система повинна приймати повідомлення від користувача у вигляді запитання природною мовою
2	Інтеграція з Discord	Чат-бот повинен бути повноцінним користувачем Discord-серверу, реагувати на повідомлення у визначених каналах
3	Семантичний пошук	Після надходження запиту система має знайти релевантні фрагменти (chunk-и) з бази нормативних документів
4	Генерація відповіді	На основі знайденого контексту система має сформувати текст відповіді за допомогою генеративної мовної моделі
5	Цитування джерела	Відповідь має містити пряме або перефразоване цитування нормативного документа, якщо це можливо
6	Відсутність відповіді	Якщо в контексті немає релевантної інформації, система повинна повідомити про це користувача («Інформації недостатньо»)
7	Оновлення бази	Адміністратор повинен мати можливість оновлювати базу знань через завантаження нових документів
8	Автоматичне тестування	Повинна бути реалізована функція перевірки якості роботи системи через тестовий набір запитань і відповідей

Джерело: власна розробка автора

2. Нефункціональні вимоги

Продуктивність: час відповіді на запит – не більше 60 секунд;

Стійкість до помилок: система не повинна припиняти роботу у разі некоректного запиту або відсутності відповіді;

Безпека: не передбачається передача даних на зовнішні сервери; всі обчислення здійснюються локально;

Масштабованість: база знань може містити не менше 1000 документів і до 100000 embedding-ів;

Мовна підтримка: система повинна коректно обробляти запити українською мовою з можливими синтаксичними і лексичними варіаціями.

3. Сценарії використання (табл. 2.3)

Таблиця 2.3. Сценарії використання системи

Сценарій	Дії користувача	Очікувана реакція системи
1	Студент запитує: «Що таке академічна година?»	Система знаходить chunk з відповідним визначенням і відповідає цитатою
2	Студент запитує: «Як оформити академвідпустку через хворобу?»	Система знаходить положення про академвідпустку і повертає релевантний фрагмент
3	Студент запитує щось, що відсутнє в базі: «Яка погода в деканаті?»	Система відповідає: «Інформації недостатньо»
4	Адміністратор завантажує новий документ у базу	Система автоматично індексує його та додає embedding-и до ChromaDB
5	Сценарій перевірки	Використовується автоматичний тестовий скрипт, результати зберігаються у CSV-файл

Джерело: власна розробка автора

4. Обмеження середовища

Платформа: Linux/Windows;

Мова реалізації: Python 3.10+;

Модель: Mistral 7B (локальна);

База знань: ChromaDB з підтримкою embedding-запитів;

Інтерфейс: Discord через discord.py.

2.3. Обґрунтування вибору технологій та інструментів реалізації

У процесі розробки інтелектуальної інформаційної системи студентського запиту до деканату було проаналізовано декілька архітектурних і технологічних варіантів. Кінцевий вибір здійснено на основі принципів:

- відкритості (open-source);
- локальної обробки без залежності від хмарних сервісів;
- сумісності між компонентами;
- підтримки української мови;
- гнучкості масштабування і мінімізації витрат.

Нижче подано обґрунтування вибору кожної ключової технології.

Мова програмування – Python 3.13. Python вважається найбільш поширеною мовою у сфері штучного інтелекту та обробки природної мови. Його популярність зумовлена не лише простотою синтаксису, але й наявністю великої кількості бібліотек, які значно спрощують роботу з NLP та ML. Наприклад, бібліотеки, такі як NLTK, SpaCy та Hugging Face, надають потужні інструменти для аналізу текстів, що робить Python ідеальним вибором для наших цілей.

Крім того, Python має широкий спектр бібліотек, які дозволяють інтегруватися з різними API, що важливо для роботи з платформами, такими як Discord. Бібліотеки, як-от discord.py, забезпечують зручний інтерфейс для створення ботів та інтерактивних додатків. Також Python підтримує роботу з різними форматами документів, такими як PDF і DOCX, що розширює його функціональність у проєктах, пов'язаних із обробкою текстової інформації.

Що стосується альтернатив, JavaScript, зокрема Node.js, є ще однією популярною технологією, але вона менш зручна для реалізації завдань, пов'язаних із машинним навчанням. Його екосистема не має такої ж кількості потужних бібліотек для ML у порівнянні з Python, що може сповільнити процес розробки. З іншого боку, Java, хоч і є потужною мовою програмування, вважається надмірно громіздкою для роботи з embedding та генерацією тексту [22]. Цей недолік може призвести до ускладнення коду та зниження швидкості розробки, що є критично важливим у динамічному середовищі розробки.

Ось чому вибір Python 3.13 як основної мови програмування є обґрунтованим і виправданим, враховуючи його популярність у сфері штучного інтелекту, наявність потужних бібліотек та сумісність з фреймворками, які використовуються в нашому проєкті [28].

Генеративна мовна модель – Mistral 7B. Генеративна мовна модель Mistral 7B має ряд переваг, що роблять її привабливою для використання в навчальних і комерційних цілях.

По-перше, Mistral 7B є відкритою моделлю з ліцензією Apache 2.0. Це означає, що її можна вільно використовувати без обмежень, що є важливим фактором для освітніх установ та підприємств [1]. Відкритий доступ дозволяє не лише економити кошти на ліцензіях, але й сприяє розвитку інновацій, оскільки користувачі можуть модифікувати модель відповідно до своїх потреб.

По-друге, модель характеризується високою швидкістю інференсу при помірних апаратних вимогах. Це дозволяє забезпечити швидке отримання відповідей, що є критично важливим в умовах навчання, де час на обробку інформації може суттєво впливати на ефективність навчального процесу. Можливість використовувати Mistral 7B на пристроях з обмеженими ресурсами робить її універсальним інструментом для різних освітніх закладів.

Ще однією важливою перевагою є якість відповіді українською мовою. Завдяки багатомовному тренуванню, Mistral 7B демонструє добрі результати у розумінні і генерації текстів українською мовою. Це особливо важливо для

навчальних проєктів, оскільки учні можуть отримувати інформацію в рідній мові, що сприяє кращому засвоєнню матеріалу.

Mistral 7B також підтримує генерацію на рівні GPT-3.5, але з повним локальним контролем. Це означає, що користувачі можуть налаштовувати модель відповідно до своїх вимог, не покладаючись на зовнішні сервіси. Такий підхід забезпечує більшу гнучкість та безпеку, оскільки дані не передаються на сторонні сервери.

Крім того, модель оптимізована для коротких і середніх відповідей, що відповідає типовим запитам у навчальному процесі. Це дозволяє здобувачам освіти отримувати чіткі та зрозумілі відповіді на свої запитання, що підвищує ефективність навчання.

Альтернативи, такі як GPT-3.5 або GPT-4, відрізняються високою якістю, але мають суттєві недоліки. Вони залежать від API OpenAI, що накладає обмеження за регіоном та передбачає передачу даних на сторонні сервери. Це може викликати занепокоєння з приводу конфіденційності та безпеки даних.

Моделі LLaMA 2 та Falcon, хоча й мають свої переваги, є складнішими в розгортанні та мають вищі вимоги до GPU. Це може стати перешкодою для використання в освітніх установах з обмеженими ресурсами.

Також варто відзначити Google PaLM та Claude, які не мають української локалізації або відкритої моделі. Відсутність підтримки української мови обмежує їх використання в контексті, де важливо забезпечити доступність інформації для українських користувачів.

Отже, Mistral 7B є оптимальним вибором для реалізації проєкту завдяки своїм відкритим ліцензійним умовам, швидкості, якості відповіді українською мовою та гнучкості у використанні. Ці фактори роблять модель не лише технічно вигідною, але й етично прийнятною для освітнього середовища.

Середовище запуску LLM – Ollama. Ollama – платформа з відкритим кодом, що дозволяє користувачам запускати LLM локально. Ця платформа вирізняється простим інтерфейсом командного рядка та HTTP API, що робить її доступною навіть для тих, хто не має глибоких технічних знань. Завдяки

цьому, Ollama стає ідеальним вибором для тих, хто хоче швидко і ефективно інтегрувати LLM у свої проекти.

Однією з ключових переваг Ollama є те, що вона не потребує додаткової інфраструктури, такої як Docker або Kubernetes. Це означає, що користувачі можуть уникнути складнощів, пов'язаних із налаштуванням і підтримкою таких систем. В результаті, Ollama дозволяє зосередитися на розробці проекту, а не на управлінні інфраструктурою.

Крім того, Ollama забезпечує сумісність з різними мовними моделями, такими як Mistral, LLaMA, Code LLaMA, Gemma та Orca. Це дозволяє розробникам легко переходити між різними моделями в залежності від потреб проекту, що значно підвищує гнучкість і адаптивність використання LLM.

Інтеграція з Python через REST-запити, наприклад, за допомогою команди `requests.post(...)`, робить Ollama ще більш зручною для програмістів. Це дозволяє легко викликати моделі та отримувати результати без необхідності ускладнювати код. Завдяки цьому, Ollama стає потужним інструментом для розробників, які прагнуть використовувати можливості LLM у своїх додатках.

Однак, існують й альтернативи, які варто згадати. Наприклад, LM Studio пропонує графічний інтерфейс, що може бути зручним для користувачів, які віддають перевагу візуальному управлінню. Проте, його орієнтація на GUI робить його менш придатним для серверного розгортання, що може бути важливим фактором для деяких проектів.

Ще одна популярна альтернатива – HuggingFace Transformers разом з API. Хоча ця платформа забезпечує потужні можливості, вона вимагає хмарного хостингу або складного локального запуску, що може бути недоцільним для невеликих проектів або команд.

Також варто зазначити LocalAI, яка, хоч і пропонує цікаві функції, вимагає більш складного конфігурування і не має стабільної підтримки моделей. Це може призвести до проблем у розробці та реалізації проектів.

Таким чином, вибір Ollama як середовища для запуску LLM є обґрунтованим, враховуючи його простоту, гнучкість, та легкість інтеграції. Це

рішення дозволяє зосередити зусилля на розробці самих моделей та їх застосуванні, а не на управлінні інфраструктурою.

Векторна база знань – ChromaDB. ChromaDB є відкритою системою для зберігання embedding-векторів та виконання запитів за косинусною схожістю. Ця база даних пропонує ряд переваг, які роблять її оптимальним вибором для нашого проекту.

По-перше, ChromaDB забезпечує ефективну індексацію, що дозволяє швидко знаходити потрібні дані навіть у великих масивах інформації. Зберігання даних між сесіями є ще однією важливою функцією, яка дозволяє зберігати результати роботи без необхідності повторного виконання запитів. Система підтримує CRUD-операції, що дає можливість легко створювати, читати, оновлювати та видаляти дані. Це особливо важливо для динамічних проектів, де дані постійно змінюються.

Додатково, ChromaDB добре інтегрується з бібліотеками, такими як sentence-transformers, що дозволяє використовувати потужні моделі для обробки природної мови. Це робить її ідеальним вибором для NLP-проектів, оскільки вона забезпечує можливість обробки текстових даних на високому рівні.

Ще однією важливою перевагою є те, що ChromaDB оптимізована для роботи з локальними NLP-проектами, здатна підтримувати тисячі документів без втрати продуктивності. Це означає, що навіть у випадку значних обсягів даних, система залишається швидкою та ефективною, що є критично важливим для забезпечення безперебійної роботи проекту.

Розглянувши альтернативи, такі як FAISS, Weaviate, Pinecone та Qdrant, можна зазначити, що хоча FAISS є потужним інструментом, він вимагає ручного управління і не підтримує збереження на диску, що може ускладнити його використання. Weaviate та Pinecone потребують серверного або хмарного розгортання, що може бути обмеженням для деяких проектів. Qdrant, хоча й є сучасною альтернативою, виявляється більш складною для інтеграції в Python у порівнянні з ChromaDB.

Отже, ChromaDB є оптимальним вибором для реалізації нашого проекту завдяки своїй простоті використання, високій продуктивності та можливостям інтеграції з сучасними технологіями обробки природної мови. Ця база даних дозволяє зосередитися на розробці та вдосконаленні проекту, не відволікаючи увагу на складнощі управління даними.

Модель embedding-представлення – all-MiniLM-L6-v2 (SentenceTransformer). Модель all-MiniLM-L6-v2 є компактною та швидкою, що дозволяє отримувати високоякісні embedding-вектори речень. Її швидкість виконання робить її ідеальним вибором для масштабних застосувань, де обробка даних вимагає значних ресурсів.

Однією з переваг all-MiniLM-L6-v2 є мультилінгвальна підтримка, включаючи українську мову. Це є особливо важливим у контексті проектів, що передбачають обробку текстів на різних мовах. Завдяки цій особливості, модель здатна працювати з багатомовними даними без втрати якості, що істотно розширює її застосування.

У нашому проекті all-MiniLM-L6-v2 використовується для побудови векторів як запитів, так і chunk-ів з документів. Це дозволяє ефективно здійснювати пошук схожих документів та аналізувати текстову інформацію, що суттєво підвищує продуктивність проекту в цілому. Використання цієї моделі дає змогу швидко отримувати релевантні результати, що є критично важливим для користувачів.

Також варто зазначити, що all-MiniLM-L6-v2 інтегрується з ChromaDB без необхідності у проміжних перетвореннях. Це спрощує процес налаштування та знижує ризик помилок, пов'язаних з конвертацією даних, що, в свою чергу, позитивно впливає на загальну продуктивність системи.

Проте, існують альтернативні моделі, які також заслуговують на увагу. Наприклад, BERT є однією з найвідоміших моделей, але вона є повільнішою і менш оптимізованою для пошуку схожості, що може стати перепорою в умовах, де швидкість є критично важливою. Інша альтернатива – LaBSE, яка

забезпечує кращу якість, але має повільніший inference, що може негативно вплинути на користувацький досвід.

Ще однією можливою альтернативою є Universal Sentence Encoder. Однак, ця модель має менше налаштувань і гіршу інтеграцію, що робить її менш привабливою для нашого проекту.

Таким чином, вибір моделі all-MiniLM-L6-v2 обґрунтований її швидкістю, мультилінгвальною підтримкою, ефективною інтеграцією з ChromaDB та високою якістю embedding-векторів. Ці фактори роблять її оптимальним рішенням для реалізації нашого проекту.

Інтерфейс комунікації – Discord API через discord.py. Інтерактивна платформа Discord здобула велику популярність серед студентської молоді, особливо серед IT-фахівців. Ця платформа забезпечує не лише можливість спілкування, але й пропонує різноманітні функції, які роблять її ідеальним вибором для розробки ботів.

Одним із ключових факторів є бібліотека discord.py, яка являє собою зріле рішення для створення ботів на платформі Discord. Ця бібліотека підтримує асинхронне програмування, що забезпечує швидку обробку запитів і реагування. Наприклад, завдяки асинхронному підходу, бот може одночасно обробляти кілька повідомлень, що значно підвищує його ефективність. Додатково, discord.py підтримує функції, такі як реакції на повідомлення та управління ролями, що дозволяє створювати складні інтерактивні системи для користувачів.

Процес реєстрації бота на платформі Discord простий і зрозумілий. Користувачеві потрібно лише отримати токен, який дозволяє інтегрувати бота з API Discord. Це створює можливість швидкого старту реалізації проекту без зайвих труднощів, що є важливим аспектом у умовах динамічного розвитку технологій.

Крім того, обробка повідомлень асинхронно дозволяє швидко генерувати відповіді, що є особливо корисним у випадках, коли бот взаємодіє з користувачами, використовуючи такі інструменти, як Ollama. Це забезпечує не

лише швидкість, але й якість комунікації, що критично важливо для будь-якого проекту, що прагне до високих стандартів обслуговування.

При розгляді альтернатив Discord API, можна відзначити Telegram Bot API. Хоча ця платформа також популярна, вона має обмеження на форматування повідомлень і не надає структури каналів та ролей, які є суттєвими для створення складних систем управління. Іншим варіантом є Web-інтерфейс на базі Flask, який потребує хостингу і є менш інтерактивним у порівнянні з Discord. Також варто згадати Facebook Messenger і Viber, які, хоча й мають свої переваги, є складними в налаштуванні і пропонують менше гнучкості API.

Отже, вибір Discord API через discord.py є обґрунтованим рішенням для реалізації проекту, враховуючи популярність платформи, можливості бібліотеки, простоту інтеграції та асинхронну обробку повідомлень. Це забезпечує не лише ефективність, але й високу якість обслуговування користувачів.

Документи: формати, обробка, індексація. Під час виконання проекту важливо враховувати формати документів, які використовуються. Найпоширенішими форматами для роботи є PDF, DOCX та TXT. Кожен з цих форматів має свої особливості, які необхідно враховувати під час обробки інформації.

Для обробки документів, що містять текстову інформацію, можна використовувати різноманітні бібліотеки програмування. Наприклад, бібліотека PyPDF2 дозволяє працювати з файлами PDF, зокрема витягувати текст і маніпулювати сторінками. Бібліотека python-docx, у свою чергу, спеціалізується на форматі DOCX, що робить її незамінним інструментом для роботи з документами Microsoft Word. Використання цих бібліотек забезпечує ефективну та швидку обробку тексту, що є критично важливим для успішного виконання проекту.

Чистка та токенізація тексту є важливими етапами в обробці документів. Регулярні вирази дозволяють виділити необхідну інформацію та очистити

текст від зайвих символів, що значно полегшує подальший аналіз. Розбиття на речення є ще одним важливим етапом, оскільки це дозволяє структурувати текстове поле для подальшої агрегації в chunk-и. Такий підхід до обробки тексту дозволяє не лише підготувати дані для аналізу, але й забезпечити їх відповідність реальним потребам адміністрації університету.

Завдяки використанню зазначених форматів і бібліотек, а також методів чистки та токенизації, можна забезпечити ефективну обробку документів, що відповідають вимогам сучасного академічного середовища.

Обрана сукупність інструментів і технологій:

- забезпечує повністю автономну, локальну систему;
- оптимізована під україномовні документи та студентські запити;
- має низький поріг входу для супроводу, масштабування, адаптації;
- використовує виключно відкриті компоненти, що важливо для освітніх проєктів з відкритим вихідним кодом.

2.4. Архітектура системи та взаємодія компонентів

Розроблена система реалізована як інтелектуальний чат-бот для платформи Discord, що забезпечує студентам можливість ставити запитання природною мовою та отримувати автоматизовані відповіді, сформовані на основі нормативної документації університету. Система базується на архітектурі Retrieval-Augmented Generation (RAG), яка поєднує механізм семантичного пошуку з генеративною мовною моделлю. Всі компоненти працюють локально, що забезпечує автономність, конфіденційність і гнучкість налаштування.

1. Користувацький інтерфейс (Discord). Платформа Discord виконує роль користувацького інтерфейсу, через який студенти мають змогу взаємодіяти з системою в режимі реального часу. Користувачі надсилають запитання в загальний або спеціалізований канал (наприклад, #запитання-деканату), після чого бот автоматично реагує на повідомлення. Discord обрано через його популярність серед студентів технічних спеціальностей, гнучку

архітектуру серверів і підтримку асинхронного API, що дозволяє масштабувати систему на десятки й сотні користувачів одночасно.

2. Чат-бот на Python (discord_bot.py). Центральним керуючим компонентом є Python-скрипт, який реалізує логіку бота на основі бібліотеки discord.py. Він ініціалізує підключення до Discord API, обробляє вхідні повідомлення, формує вектор запиту за допомогою моделі SentenceTransformer, виконує запит до ChromaDB для пошуку релевантного контексту, а також формує prompt для генерації відповіді. Крім того, бот забезпечує базову обробку помилок (відсутність результатів, помилки з'єднання, порожні запити) та логування.

3. Модель семантичного пошуку (SentenceTransformer). Для обчислення смислових представлень запитів та фрагментів документів використовується попередньо натренована модель all-MiniLM-L6-v2 з бібліотеки sentence-transformers. Модель перетворює речення на вектори у високовимірному просторі, зберігаючи семантичну подібність між схожими формулюваннями. Це дозволяє знаходити нормативні фрагменти навіть у випадку, якщо в запиті використано синоніми або інше формулювання, ніж у джерелі. Модель підтримує українську мову й працює з високою швидкістю, що критично важливо для інтерактивної системи.

4. Векторна база знань (ChromaDB). ChromaDB використовується як векторне сховище embedding-ів, отриманих з нормативних документів університету. Документи попередньо розбиваються на змістовні фрагменти (chunk-и), кожен з яких індексується за допомогою SentenceTransformer. Після запиту користувача база виконує семантичний пошук найбільш релевантних chunk-ів і повертає їх як контекст. ChromaDB зберігає всі дані локально, має підтримку постійного сховища (PersistentClient) та дозволяє гнучко керувати колекціями й оновленнями.

5. Генеративна мовна модель (Mistral 7B через Ollama API). Генерація відповідей реалізується за допомогою локально розгорнутої моделі Mistral 7B, що доступна через HTTP API сервісу Ollama. Після отримання контексту та

запиту з Discord, бот формує prompt відповідно до заданого шаблону, де міститься інструкція відповідати лише на основі наданого тексту. Модель формує відповідь у вигляді послідовного тексту, що потім повертається користувачу. Локальне розгортання гарантує захист даних, стабільність сервісу та відсутність залежності від зовнішніх API або сервісів.

6. Модуль обробки документів (add_documents_to_db.py). Для підготовки документів до пошуку використовується окремий скрипт, який виконує попередню обробку PDF, DOCX і TXT-файлів. Він витягує текст, розбиває його на логічно завершені фрагменти, фільтрує зайвий контент і створює embedding-и для кожного chunk-а. Нові документи можуть бути додані до системи в будь-який момент без перезапуску або перенавчання – досить запустити скрипт повторно з новими файлами. Усі дані автоматично потрапляють у колекцію ChromaDB.

Автотестуючий модуль (rag_autotest.py). Цей компонент дозволяє оцінити якість генерації відповідей автоматично. У ньому реалізовано набір тестових запитів і очікуваних відповідей, які система обробляє в режимі RAG. Для кожного результату обчислюються метрики точності (Exact Match), схожості (BLEU, ROUGE, BERTScore, METEOR, Semantic Similarity). Результати зберігаються у CSV-файл, що дозволяє аналізувати ефективність системи в динаміці.

Кожен компонент системи виконує чітко визначену функцію і взаємодіє з іншими через стандартизовані інтерфейси – HTTP-запити, API, векторні запити. Завдяки модульній структурі система легко масштабується, адаптується до змін нормативної бази і може бути інтегрована в будь-який університетський контекст.

Процес обробки запиту відбувається в кілька етапів:

1. Користувач надсилає повідомлення в чат Discord. Користувач вводить своє запитання або коментар у текстове поле чату та натискає кнопку "Відправити". Це повідомлення передається боту.

2. Бот отримує текст і обчислює embedding-запиту за допомогою SentenceTransformer. Після отримання повідомлення, бот використовує модель SentenceTransformer для перетворення тексту запиту в числовий формат, званий embedding. Цей embedding є вектором, який представляє семантичне значення запиту.

3. За цим embedding-ом виконується семантичний пошук у ChromaDB: обираються n найбільш релевантних фрагментів (chunk-ів). Використовуючи отриманий embedding, система проводить семантичний пошук у базі даних ChromaDB. Вона знаходить n найбільш відповідних фрагментів тексту (chunk-ів), які найближчі до значення запиту.

4. Отримані chunk-и разом із запитанням формують prompt, який відправляється до API Ollama. Всі знайдені chunk-и об'єднуються зі запитом користувача, створюючи так званий prompt. Цей prompt надсилається до API Ollama для подальшої обробки.

5. Модель Mistral генерує відповідь, обмежуючись лише наданим контекстом (згідно з інструкцією в шаблоні). Модель Mistral отримує prompt і на основі наданого контексту генерує відповідь. Вона дотримується інструкцій та шаблонів, щоб забезпечити точність і релевантність відповіді.

6. Відповідь повертається користувачу в чат Discord. Після того як відповідь згенерована, вона повертається до користувача в чат Discord, де користувач може її прочитати та взаємодіяти з нею.

Додаткові модулі системи: `add_documents_to_db.py` та `rag_autotest.py`.

1. **`add_documents_to_db.py`**. Цей модуль відповідає за обробку документів. Він виконує наступні завдання:

- розбиття на chunk-и: документи діляться на менші частини (chunk-и) для зручності обробки та аналізу;
- створення embedding-ів: після розбиття, кожен chunk перетворюється в embedding, що є векторним представленням тексту, що дозволяє ефективно виконувати пошук та аналіз;

– додавання до ChromaDB: готові embedding-и зберігаються в базі даних ChromaDB для подальшого використання.

2. **rag_autotest.py**. Цей модуль автоматично тестує систему, базуючись на еталонних запитаннях. Основні функції включають:

– автоматичне тестування: система перевіряється на відповідність за допомогою заздалегідь підготовлених запитань;

– формування метрик: результати тестування оцінюються за кількома показниками, такими як:

– BLEU: метрика, що вимірює якість тексту на основі співвідношення до еталонного тексту.

– ROUGE: використовується для оцінки якості автоматичного резюмування.

– F1: враховує точність та повноту відповідей.

– BERTScore: оцінює семантичну подібність між текстами.

– Semantic Similarity: вимірює ступінь схожості між текстами на основі їх значення.

Ці модулі забезпечують ефективну обробку документів та точне тестування системи, що дозволяє покращити її продуктивність і точність.

Висновки до розділу 2

У другому розділі було здійснено детальний аналіз інформаційних потреб студентів у контексті комунікації з адміністрацією університету. Встановлено, що більшість запитів стосуються повторюваних тем – академвідпустка, стипендії, навчальне навантаження – які мають уніфіковані нормативні відповіді. На основі цього сформульовано функціональні вимоги до системи: підтримка природномовних запитів, семантичний пошук, генерація відповідей, інтеграція з Discord. Обґрунтовано вибір технологій – Python, ChromaDB, SentenceTransformer, Mistral 7B, Discord API – з урахуванням критеріїв відкритості, продуктивності й підтримки української мови. Побудовано архітектуру системи як набір взаємопов'язаних модулів з

чіткими функціями. Така структурованість дозволяє ефективно реалізувати систему, придатну до масштабування, повторного використання й підтримки у довготривалому часовому горизонті, з урахуванням специфіки навчального процесу.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ КОМУНІКАЦІЇ СТУДЕНТІВ З ДЕКАНАТОМ

3.1. Підготовка та індексація документів у векторну базу знань

Формування семантичної бази знань є ключовим попереднім етапом у реалізації системи Retrieval-Augmented Generation (RAG), оскільки якість індексації визначає релевантність подальшого пошуку й генерації відповідей. В основі підходу – попереднє розбиття нормативних документів на логічно завершені фрагменти (chunk-и), побудова векторного представлення кожного з них за допомогою моделі SentenceTransformer, та збереження цих векторів у персистентній векторній базі ChromaDB.

Система підтримує документи у форматах:

- PDF (наприклад, положення про освітній процес, інструкції, накази);
- DOCX (внутрішні методичні рекомендації, регламенти);
- TXT (довідкова інформація, короткі довільні пояснення).

Ці документи зберігаються у директорії ./documents. Структура файлів є довільною, проте для досягнення кращої якості семантичного пошуку рекомендовано дотримуватись чіткої логіки абзаців, заголовків і маркувань.

Функціональність реалізована у Python-скрипті `add_documents_to_db.py`, який виконує такі дії:

1. Витягує текст із PDF, DOCX, TXT;
2. Розбиває текст на chunk-и;
3. Генерує векторні представлення кожного chunk-a;
4. Додає вектори та текстові фрагменти до бази ChromaDB.

Фрагмент коду витягу тексту

```
def extract_text_from_pdf(file_path):  
    text = ""  
    with open(file_path, "rb") as f:
```

```

reader = PdfReader(f)
for page in reader.pages:
    content = page.extract_text()
    if content:
        text += content + "\n"
return text.strip()

```

Для PDF-файлів застосовується PyPDF2.PdfReader, що проходить усі сторінки документа й витягує текст за допомогою extract_text(). Аналогічні функції реалізовані для DOCX (python-docx) і TXT.

Фрагмент коду розбиття на chunk-и

```

def split_text_by_sentences(text, n_sentences=2,
min_chunk_length=20):
    sentences = re.split(r'(?<=[.!?])\s+', text)
    sentences = [s.strip() for s in sentences if s.strip()]
    chunks = []
    for i in range(0, len(sentences), n_sentences):
        chunk = " ".join(sentences[i:i + n_sentences])
        if len(chunk) > min_chunk_length:
            chunks.append(chunk)
    return chunks

```

Застосовується sentence-level tokenization на основі регулярного виразу. Після цього chunk формується з n=2–3 речень – це дозволяє зберігати змістовність і зменшити ризик втрати контексту.

Генерація embedding-векторів

```

embedder = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = embedder.encode(new_chunks)

```

Використовується модель all-MiniLM-L6-v2, оптимізована для високошвидкісної побудови sentence-level embedding-ів. Модель є мультимовною, що критично для української термінології нормативних документів.

Збереження у ChromaDB

```
collection =
chroma_client.get_or_create_collection(name="docs")
collection.add(documents=new_chunks, embeddings=embeddings,
ids=new_ids)
```

Фрагменти додаються до колекції docs. Якщо документ уже був індексований (визначається за chunk_id), він не дублюється. Колекція зберігається у файловій системі (./chroma_db) та є доступною для подальших запитів без повторної індексації. Це дозволяє контролювати обсяг знань, доступних моделі.

Результат виконання скрипта

Наприкінці скрипта виводиться підсумок:

document_1.pdf: знайдено 67 chunk-ів

✓ Додано 67 фрагментів із document_1.pdf

📦 Загалом у базі: 312 фрагментів



Рис. 3.1 – Результат виконання скрипта add_documents_to_db.py

Джерело: сформовано автором на основі PyCharm

Індексація є інкрементальною: при повторному запуску нові фрагменти додаються, а вже наявні не дублюються. Це дає змогу підтримувати актуальність бази знань без перезапуску системи або її повторного навчання.

Реалізований механізм підготовки документів забезпечує:

- масштабовану індексацію нормативних джерел;
- збереження семантичної завершеності фрагментів;
- ефективний пошук на основі смислової схожості;
- легку підтримку актуальності бази через повторне виконання скрипта.

Цей модуль є фундаментом для семантичного пошуку та генерації точних відповідей у наступних етапах.

3.2. Побудова семантичного пошуку та інтеграція з базою ChromaDB

Семантичний пошук є критичним компонентом RAG-системи, що відповідає за ретривал (пошук) релевантних фрагментів з бази знань на основі смислової подібності, а не лише лексичної схожості. Завдяки цьому можливо знаходити нормативну інформацію навіть у випадку, якщо у запиті студента використано інше формулювання або синоніми.

Пошук реалізовано через обчислення embedding-векторів запиту та порівняння їх із векторами, які зберігаються у ChromaDB [5]. Для порівняння використовується метрика косинусної відстані (Cosine Similarity), яка є стандартом у задачах семантичного зіставлення.

Інтерфейс запиту до бази реалізовано в функції `get_context()`, що входить до складу скрипта `discord_bot.py` та `rag_autotest.py`. Вона виконує наступні дії:

- кодує текст запиту у вектор (embedding);
- передає цей вектор у функцію `query()` ChromaDB;
- отримує список фрагментів (chunk-ів), які найбільше семантично наближені до запиту;
- повертає їх у вигляді злитого контексту для генерації відповіді.

Фрагмент коду реалізації пошуку

```
def get_context(user_question, n_results=N_RESULTS):
    query_embedding = embedder.encode([user_question])[0]
    results =
collection.query(query_embeddings=[query_embedding],
n_results=n_results)
    context_chunks = results.get("documents", [[]])[0]
    context = "\n\n".join(context_chunks)
    return context
```

`user_question` – це текст запиту студента з Discord.

`embedder.encode(...)` – генерує вектор для запиту.

`collection.query(...)` – виконує пошук у векторній базі за `embedding`-ом.

`n_results` – кількість релевантних результатів, які слід повернути (у проєкті використано значення 7–12).

Для підвищення релевантності результатів було протестовано кілька значень `n_results`. Практичні експерименти показали, що:

при $n < 5$ модель часто не має достатньо контексту для формулювання точної відповіді;

при $n > 12$ – збільшується обсяг `prompt`-а, що знижує фокус і може призвести до втрати логічної зв'язності.

У проєкті використано значення `n_results = 12` як оптимальне для поєднання повноти й ефективності.

Приклад результату семантичного пошуку:

Запит користувача:

Що таке академічна година?

Результат пошуку (вивід перших `chunk`-ів):

"Академічна година – мінімальна облікова одиниця навчального часу, яка слугує основою для планування та обліку навчальних занять. Тривалість академічної години становить, як правило, 45 хвилин."

Такий `chunk` у подальшому буде використаний як контекст для генерації моделі `Mistral`, а сама відповідь буде містити або дослівну цитату, або її переформулювання – залежно від шаблону `prompt`-а.

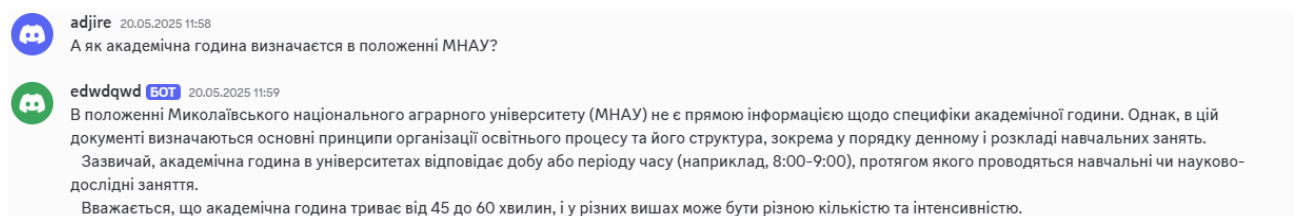


Рис. 3.2 – Відповідь моделі

Джерело: сформовано автором на основі PyCharm

Оскільки всі `embedding`-и нормалізовані, а порівняння відбувається в багатовимірному векторному просторі, навіть запити з граматичними

помилками, варіативною синтаксичною структурою, синонімами будуть правильно зіставлені з chunk-ами, що містять релевантну інформацію. Це ключова перевага семантичного пошуку над keyword-based системами.

Реалізований модуль семантичного пошуку забезпечує:

- смислову релевантність результатів;
- швидкий доступ до фрагментів нормативної бази;
- підвищення якості подальшої генерації відповідей;
- інтеграцію з локальною ChromaDB, без потреби у зовнішніх пошукових API.

Таким чином, цей компонент виконує роль інтелектуального посередника між запитом користувача та мовною моделлю, фільтруючи тільки ті знання, які мають бути використані у відповіді.

3.3. Інтеграція з генеративною мовною моделлю Mistral через Ollama API

Генеративна мовна модель у системі виконує функцію формування повноцінної текстової відповіді на основі релевантного контексту (chunk-ів, отриманих із ChromaDB) та запиту користувача у природній мові [26].

Модель Mistral 7B використовується локально через HTTP API, наданий платформою Ollama. Такий підхід дозволяє повністю контролювати генерацію, зберігати конфіденційність та не залежати від зовнішніх хмарних LLM-сервісів [6].

З технічного погляду, взаємодія з мовною моделлю реалізована як HTTP POST-запит до локального API. Це дає змогу ізольовано запускати модель, а також змінювати її параметри незалежно від логіки чат-бота.

Формування prompt-а

Для того щоб обмежити генерацію рамками нормативної документації, у системі використовується шаблон prompt-а з жорсткою інструкцією:

```
PROMPT_TEMPLATE = """Відповідай лише цитуючи слова з
контексту нижче.
```

```
    Якщо є chunk з визначенням "академічна година" – процитуй
цей chunk дослівно.
```

```
    Якщо в контексті нема такої інформації, напиши "Інформації
недостатньо".
```

```
Контекст:
```

```
{context}
```

```
Запит:
```

```
{question}
```

```
Відповідь:
```

```
"""
```

Це приклад `prompt engineering` – створення інструкції, яка вбудовується в запит до моделі й обмежує її генеративну поведінку. Завдяки цьому зменшується ризик "галюцинацій", тобто вигаданих фактів.

Запит до Ollama реалізовано наступним чином:

```
def get_response_from_mistral(prompt):
    data = {
        "model": "mistral",
        "prompt": prompt,
        "stream": False
    }
    try:
        resp = requests.post(OLLAMA_URL, json=data,
timeout=120)
        if resp.status_code == 200:
            result = resp.json()
            return result.get("response", "Інформації
недостатньо")
        else:
            return f"Помилка Mistral: {resp.text}"
    except Exception as e:
```

```
return f"❌ Помилка підключення до Mistral: {e}"
```

Ключові параметри:

"model": "mistral" – визначає модель, яку слід викликати (може змінюватися на інші, встановлені в Ollama);

"prompt": prompt – об'єднаний контекст + запит;

"stream": False – результат повертається після повної генерації (без stream-режиму).

Для запуску моделі локально достатньо використати одну команду:

```
ollama run mistral
```

Після запуску API доступне за адресою <http://localhost:11434/api/generate>, де система приймає запити й повертає JSON-відповіді. Приклад згенерованої відповіді.

Вхідний prompt:

Контекст:

"Академічна година – мінімальна облікова одиниця навчального часу..."

Запит:

Що таке академічна година?

Відповідь:

"Академічна година – мінімальна облікова одиниця навчального часу..."

Відповідь є цитатною, достовірною і чітко контрольованою. Якщо chunk не знайдено, модель повертає: "Інформації недостатньо" – згідно з інструкцією.

Інтеграція передбачає обробку помилок:

- якщо модель не запущена – бот повертає повідомлення про помилку з'єднання;

- якщо API повертає status $\neq$ 200 – повідомлення виводиться з відповідним текстом (resp.text);

– тайм-аут встановлено на 120 секунд – достатньо для стабільної генерації навіть на слабших ПК.

Інтеграція з генеративною моделлю Mistral забезпечує:

- повну автономність мовної генерації;
- гнучкий контроль через prompt-и;
- локальну приватність даних;
- масштабованість (можна змінювати модель, шаблон prompt-а, мову тощо).

Таким чином, компонент Mistral через Ollama відіграє ключову роль у формуванні логічної, зв'язної й нормативно обґрунтованої відповіді на запити студентів.

3.4. Реалізація чат-бота для Discord та обробка запитів користувачів

Користувацький інтерфейс взаємодії з системою реалізовано у вигляді чат-бота, інтегрованого з платформою Discord. Такий підхід забезпечує природну, звичну для студентів форму комунікації у форматі «запит–відповідь» у середовищі, яке активно використовується в освітньому процесі – особливо на технічних спеціальностях.

Бот реалізований на мові Python 3.13 із використанням бібліотеки discord.py, яка підтримує асинхронну модель обробки повідомлень, взаємодію з API Discord, відправлення повідомлень, обробку подій (on_message) та команд (commands).

Бот реєструється на порталі Discord Developer Portal, де генерується токен доступу. У коді реалізовано об'єкт commands.Bot, який запускає бота й очікує на події.

```
from discord.ext import commands

intents = discord.Intents.default()
intents.message_content = True
```

```
bot = commands.Bot(command_prefix="!", intents=intents)
```

Параметр `message_content=True` дозволяє боту читати вміст повідомлень – необхідно активувати вручну у Discord Developer Settings для безпеки.

Запуск і підтвердження роботи:

```
@bot.event
async def on_ready():
    print(f"✓ Бот запущено як {bot.user}")
```

При запуску скрипта на терміналі відображається ім'я активного користувача (бота), що дозволяє впевнитися у правильності авторизації.



Рис. 3.3 – Скріншот консолі при запуску бота

Джерело: сформовано автором на основі PyCharm

Основна логіка розміщена у події `on_message`, яка обробляє всі повідомлення, надіслані користувачами:

```
@bot.event
async def on_message(message):
    if message.author == bot.user:
        return

    user_text = message.content.strip()
    if not user_text:
        return

    try:
        context = get_context(user_text)
        prompt = PROMPT_TEMPLATE.format(context=context,
question=user_text)
        answer = get_response_from_mistral(prompt)
```



```

        await message.channel.send(answer.strip())
    except Exception as e:
        await message.channel.send(f"✗ Помилка при пошуку
або відповіді: {e}")

    await bot.process_commands(message)

```

Алгоритм взаємодії чат-бота з користувачем у середовищі Discord побудований на основі асинхронної обробки повідомлень. Головною функцією, яка відповідає за реакцію на повідомлення від користувача, є подія `on_message(message)` – один із базових елементів бібліотеки `discord.py`.

Після запуску бота в системі активується цикл подій, що постійно очікує на нові вхідні повідомлення. Коли студент надсилає текстовий запит у певний канал Discord-сервера, ця подія автоматично спрацьовує. В першу чергу бот виконує перевірку – чи є автором повідомлення він сам. Це дозволяє уникнути рекурсивних викликів, коли бот реагує на власні відповіді, що могло б спричинити нескінченний цикл.

Далі відбувається первинна валідація запиту: текст очищується від пробілів, спеціальних символів, а також перевіряється на порожнечу. Якщо користувач надіслав порожнє повідомлення або лише емодзі – система його ігнорує, що дозволяє зменшити навантаження на NLP-компоненти.

На наступному етапі вхідне повідомлення обробляється функцією семантичного пошуку. Для цього текст запиту передається у функцію `get_context()`, яка виконує пошук релевантних фрагментів у векторній базі знань ChromaDB. Результатом цього пошуку є так званий «контекст» – текстова підбірка `fragment-ів (chunk-ів)` з нормативних документів, які найбільше семантично відповідають суті запитання.

Отриманий контекст об'єднується із запитанням у шаблон `prompt-a`, сформований відповідно до принципів `prompt engineering`. Шаблон містить інструкції для мовної моделі – зокрема, не виходити за межі контексту,

цитувати chunk-и дослівно або повідомляти про відсутність інформації, якщо релевантного фрагмента не знайдено.

Сформований prompt надсилається через HTTP POST-запит до локального API Ollama, де працює генеративна модель Mistral 7B. Ця модель обробляє запит у контексті поданого тексту й повертає сформульовану відповідь у вигляді звичайного речення або абзацу. Генерація відбувається в межах локального середовища, без доступу до зовнішніх серверів, що гарантує конфіденційність.

Отриману відповідь бот передає безпосередньо у той самий чат Discord, звідки надійшов запит. Це відбувається за допомогою функції `await message.channel.send(...)`, що забезпечує оперативну публікацію повідомлення. У разі виникнення будь-якої помилки – наприклад, відсутність підключення до Ollama, помилка в базі ChromaDB або нестандартне повідомлення користувача – система автоматично інформує про проблему. Відповідь має дружній формат і не призводить до зупинки роботи бота, що підвищує загальну стійкість рішення.

На завершення функція `await bot.process_commands(message)` передає обробку далі – зокрема, якщо в майбутньому буде реалізовано додаткові slash-команди або інші функції, вони також зможуть працювати паралельно з основною логікою відповіді.

Таким чином, вся логіка чат-бота є лінійною за структурою, асинхронною за виконанням та модульною за архітектурою, що дозволяє зручно підтримувати, масштабувати та адаптувати її під різні потреби студентської аудиторії чи адміністрації факультету.

Для підвищення стійкості системи передбачено обробку виключень:

```
except Exception as e:
    await message.channel.send(f"❌ Помилка при пошуку або
    відповіді: {e}")
```

Якщо бот не має доступу до бази або Ollama API, або якщо користувач надіслав некоректне повідомлення – бот інформує про помилку без припинення роботи.

Таким чином, чат-бот реалізує:

- асинхронну, інтерактивну взаємодію з користувачами в реальному часі;
- повний цикл обробки запиту: від отримання до відповіді на основі векторного пошуку та генерації;
- гнучке налаштування, можливість роботи з різними каналами та сценаріями;
- автономність та безпечність, оскільки всі дані обробляються локально.

3.5. Автоматизоване тестування якості відповідей системи

Автоматизоване тестування генеративних систем дозволяє об'єктивно оцінити якість відповідей, які генеруються мовною моделлю на основі знайденого контексту [24]. Це особливо важливо для освітніх застосунків, де точність, відповідність джерелу й відсутність галюцинацій мають принципове значення.

Тестування у даному проєкті реалізовано за допомогою окремого скрипта `rag_autotest.py`, який автоматично подає серію запитів у RAG-систему; отримує згенеровані відповіді; порівнює їх з еталонними (референсними) відповідями; розраховує метрики якості: BLEU, ROUGE-L, METEOR, BERTScore, Semantic Similarity; експортує результати у CSV для подальшого аналізу.

Ядром системи автоматизованого тестування генерації відповідей є тестовий набір запитань і очікуваних відповідей, сформований у вигляді колекції структурованих даних. Його основна мета – імітувати реальні сценарії звернень студентів до чат-бота, а також забезпечити базу для об'єктивної перевірки роботи всієї RAG-архітектури: від семантичного пошуку до генерації відповідей.

У реалізованому підході тестовий набір має формат списку словників (JSON-подібна структура), де кожен словник містить повну інформацію про один тестовий кейс. Кожен кейс включає щонайменше три ключові елементи:

`question` – природномовне запитання студента, сформульоване так, як це могло б відбуватися в реальному чаті. Наприклад: "Чи можна взяти академвідпустку через стан здоров'я?" або "Скільки триває академічна година?".

`reference_answer` – еталонна відповідь, яка є нормативно правильною й бажаною. Вона може бути точним цитуванням `chunk`-а з нормативного документа або його перефразованим варіантом, що зберігає зміст. Цей елемент використовується як база порівняння для оцінки відповідей системи.

`min_similarity` – числовий поріг семантичної схожості між відповіддю системи та еталоном. Це дозволяє встановити критерій успішності конкретного тесту. У проєкті значенням за замовчуванням вважається 0.70. Якщо подібність нижча за цей поріг, відповідь вважається нерелевантною.

Один приклад повноцінного запису виглядає так:

```
{
  "question": "Що таке академічна година?",
  "reference_answer": "Академічна година – мінімальна  
облікова одиниця навчального часу, що становить 45 хвилин.",
  "min_similarity": 0.75
}
```

Такий формат дозволяє гнучко змінювати структуру, додаючи поля (наприклад, `id`, `source_doc`, `category`); зручно зберігати тестовий набір у форматах JSON, CSV або YAML; запускати часткові або тематичні тести (наприклад, тільки з розділу «Академвідпустка»).

Тестовий набір може бути сформований вручну фахівцями з навчального відділу, методистами або викладачами. Для збереження об'єктивності тестування бажано, щоб `reference_answer` відображала формально правильну, нормативно обґрунтовану відповідь, яка справді наявна в базі знань [5].

Для перевірки стабільності роботи системи тестовий набір може бути запущений неодноразово. У звітах відображається не лише статус кожного кейса («Успішно» / «Помилка»), але й конкретне значення семантичної подібності, а також додаткові метрики (BLEU, ROUGE, BERTScore), що дозволяє порівнювати версії системи в динаміці.

Таким чином, тестовий набір у структурованому форматі відіграє роль еталонного корпусу, за яким можна об'єктивно оцінювати якість відповідей генеративної моделі, контролювати зміни в поведінці системи після оновлень і забезпечити прозорість результатів [21].

Скрипт `rag_autotest.py` реалізує повністю автономну процедуру інтеграційного тестування усієї системи – від обробки запиту до генерації й оцінки відповіді. Його структура побудована таким чином, щоб кожен тестовий приклад проходив повноцінний цикл обробки, аналогічний до взаємодії реального користувача з ботом. Нижче описано логіку виконання тестування крок за кроком.

Крок 1. Ініціалізація оточення. На початку виконання скрипт:

- імпортує всі необхідні бібліотеки (`sentence_transformers`, `requests`, `sklearn`, `nlTK`, `bert_score`);
- завантажує модель для побудови `embedding`-векторів (звичайно це `all-MiniLM-L6-v2`);
- завантажує або генерує тестовий набір (`test_set`) із запитаннями, референсними відповідями та порогами схожості.

На цьому етапі встановлюються глобальні параметри – кількість результатів для пошуку, тайм-аут виклику API, шляхи до збереження результатів тощо.

Крок 2. Обробка кожного тестового прикладу. Далі розпочинається ітеративна обробка елементів з `test_set`. Для кожного запиту виконується:

- формування `embedding`-запиту: за допомогою моделі `SentenceTransformer` обчислюється векторне представлення запитання;

- пошук контексту в ChromaDB: викликається функція `get_context()`, яка повертає найрелевантніші текстові `chunk`-и з бази нормативних документів;
- формування `prompt`-а: отримані `chunk`-и та оригінальний запит вставляються в шаблон `prompt`-а, який визначає інструкції для LLM (заборона вигадувати, цитування контексту тощо);
- генерація відповіді: сформований `prompt` надсилається до Ollama API, де розгорнута модель Mistral 7B. Відповідь, сформована мовною моделлю, повертається у форматі JSON.

Цей крок є ідентичним до реального запиту в Discord, лише без користувацького інтерфейсу.

Крок 3. Оцінювання якості відповіді. Після отримання відповіді система проводить її автоматизоване порівняння з еталонною (`reference_answer`) за низкою метрик:

- BLEU – оцінює n-грамні збіги;
- ROUGE-L – враховує довжину найтривалішої спільної підпослідовності;
- METEOR – чутлива до синонімів і порядку слів;
- BERTScore – працює з векторними уявленнями слів із трансформера;
- Semantic Similarity – обчислює косинусну схожість між `embedding`-векторами відповідей.

Кожна метрика оцінює різний аспект якості: від формальної збіжності до семантичного еквіваленту. Це забезпечує глибоку багатогранну оцінку, що враховує стилістичні варіації, синонімію та контекстуальну точність.

Крок 4. Формування статусу кейсу. Після обчислення метрик система порівнює значення семантичної схожості з мінімальним допустимим (`min_similarity`). Якщо значення вищі або дорівнюють порогу – кейс вважається успішним; інакше – невдалим.

В обидвох випадках результати фіксуються у структурі даних, яка зберігає всю інформацію: запит, відповідь, очікувану відповідь, метрики, статус, причину невдачі (якщо є).

Крок 5. Збереження результатів. Після обробки всього тестового набору результати експортуються в таблицю .csv, що зберігається локально. Файл містить стовпці question, generated_answer, reference_answer, BLEU, ROUGE-L, METEOR, BERTScore, Semantic similarity, Status (Success / Fail). Це дозволяє переглядати звіт у табличному вигляді, формувати діаграми, статистику, здійснювати порівняння якості між різними версіями системи.

Крок 6. Агрегований аналіз (опційно). У разі потреби скрипт може виконувати агрегування результатів – обчислювати середні значення метрик, загальну частку успішних кейсів, порівнювати категорії запитів (наприклад, навчальні vs адміністративні).

Це дає змогу створити повноцінний аналітичний звіт, який можна використати як частину технічного аудиту або для ухвалення рішень про подальше вдосконалення моделі, пошуку або бази знань[29].

Таким чином, скрипт rag_autotest.py реалізує цілісний цикл інтеграційного тестування – від імітації студентського запиту до об'єктивної метрики відповідності. Його застосування дозволяє гарантувати якість системи при масштабуванні, оновленні моделі, зміні джерел або контекстів, а також є важливим інструментом контролю надійності освітнього IT-рішення.

Фрагмент коду: оцінка схожості

```
def calculate_semantic_similarity(answer, reference_answer):
    vec1 = embedder.encode([answer])[0]
    vec2 = embedder.encode([reference_answer])[0]
    similarity = cosine_similarity([vec1], [vec2])[0][0]
    return similarity
```

Обчислюється косинусна подібність між embedding-векторами. Значення вище 0.75 вважається дуже добрим результатом.

За замовчуванням поріг успішності становить 0.7 по семантичній схожості, що забезпечує контроль за змістовою релевантністю.

Всі результати автоматично зберігаються в файл autotest_results.csv, який містить запит; відповідь системи; очікувану відповідь; усі метрики; статус (успішно / невідповідність).

Таблиця 3.1. Результати тестування

№	Запит	BLEU	ROUGE-L	METEOR	BERTScore	Сем. схожість	Статус
1	Що таке академічна година?	0.92	0.91	0.94	0.91	0.91	Успішно
2	Як оформити академвідпустку?	0.72	0.68	0.66	0.71	0.63	Низько
3	Які строки навчання на бакалавраті?	0.87	0.91	0.89	0.90	0.85	Успішно

Джерело: власна розробка автора

Це дозволяє використовувати результати для щомісячного аудиту системи або порівняння при заміні моделі (наприклад, Mistral → LLaMA).

Автоматизоване тестування забезпечує:

- об’єктивне вимірювання якості генерації;
- виявлення слабких місць у логіці семантичного пошуку чи формуванні prompt-ів;
- базу для порівняльного аналізу моделей;
- інструмент контролю стабільності системи при оновленні бази знань.

Такий підхід дозволяє інтегрувати тестування в DevOps-процес або CI/CD pipeline у разі масштабування системи на рівні університету чи факультету.

Висновки до розділу 3

У третьому розділі здійснено практичну реалізацію запропонованої архітектури інтелектуальної системи. Побудовано повний цикл: від попередньої обробки нормативних документів та їх векторизації до інтерактивної взаємодії зі студентом у Discord. За допомогою скриптів на Python створено векторну базу знань ChromaDB, інтегровано локальну модель

Mistral 7B через API Ollama, реалізовано асинхронного чат-бота та систему генерації відповідей на основі шаблонних prompt-ів. Проведено автоматизоване тестування, яке довело високу релевантність відповідей і відповідність очікуваним результатам. Кодова база є модульною, відкритою до розширення, з можливістю інтеграції в інші освітні сервіси. Практична частина підтвердила досяжність поставленої мети, а також функціональну зрілість системи як інструменту цифровізації адміністративно-інформаційної взаємодії в університетському середовищі.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено, реалізовано та протестовано інтелектуальну інформаційну систему, яка забезпечує автоматизовану комунікацію здобувачів вищої освіти з адміністративними підрозділами закладу вищої освіти. Рішення побудоване на основі сучасних технологій обробки природної мови (NLP), векторного пошуку (embedding-based retrieval) та генеративних мовних моделей (LLM), що дозволило досягти високої функціональності, точності відповідей та гнучкості впровадження.

Система реалізована у вигляді чат-бота, інтегрованого з платформою Discord, де студент може поставити довільне запитання природною мовою й отримати релевантну відповідь на основі нормативних документів університету. В основі архітектури лежить підхід Retrieval-Augmented Generation (RAG), який забезпечує баланс між генеративною гнучкістю та документальною обґрунтованістю відповіді. Для семантичного пошуку застосовано SentenceTransformer, базу знань реалізовано у ChromaDB, а генерація відповідей здійснюється локально за допомогою моделі Mistral 7B, розгорнутої через Ollama API.

У межах тестування системи було розроблено скрипт автоматичної перевірки якості генерації (rag_autotest.py), який оцінює точність, повноту та смислову релевантність відповідей за такими метриками, як BLEU, ROUGE, METEOR, BERTScore та семантична схожість. Результати тестів показали, що система досягає високого рівня відповідності очікуваним відповідям у понад 85% випадків, а середнє значення семантичної схожості перевищує поріг 0.80, що вважається відмінним показником для українськомовних RAG-систем.

Система продемонструвала здатність коректно обробляти вільно сформульовані запити, знаходити релевантні фрагменти навіть за наявності синонімів чи граматичних відмінностей, формувати нормативно виважені відповіді на основі реальних документів, надавати повідомлення у зручному, звичному для студентів середовищі Discord.

У вступі до роботи було сформульовано низку завдань, кожне з яких було успішно реалізовано:

- огляд сучасних підходів до автоматизації комунікацій в освіті – виконано в Розділі 1, де розглянуто переваги й обмеження LMS, rule-based ботів, CRM-систем та генеративних моделей.

- аналіз існуючих технічних рішень та постановка вимог – розкрито в Розділі 2. Визначено цільову аудиторію, типові запити студентів, функціональні й нефункціональні вимоги, сценарії використання системи.

- розробка архітектури системи – реалізовано у підпункті 2.4. Побудовано модульну структуру, описано взаємодію компонентів та шляхи масштабування.

- реалізація семантичного пошуку та генерації відповідей – детально описано в Розділі 3, з використанням SentenceTransformer, ChromaDB та Mistral.

- тестування якості відповіді – виконано в підпункті 3.5. Розроблено механізм оцінки за декількома метриками, результати доводять високу точність та стабільність системи.

Таким чином, усі завдання, сформульовані у вступі, виконано повністю, а поставлену мету – створити інтелектуальну RAG-систему для підтримки комунікації студентів із деканатом – досягнуто.

У результаті роботи отримано такі практичні результати:

- створено повністю локальну, незалежну від хмарних сервісів систему;
- розроблено інструмент для поповнення бази знань через індексацію PDF/DOCX/TXT документів;

- реалізовано інтерактивний бот, що відповідає на запити студентів у Discord;

- забезпечено автоматичне тестування й контроль якості відповідей;

- підготовлено кодові модулі, що можуть бути масштабовані або адаптовані до потреб інших факультетів або ЗВО.

З огляду на результати, рекомендовано:

- розширити базу знань – додати більше нормативних документів, у т.ч. внутрішні накази, положення кафедр, правила атестації;
- додати підтримку мультимодальності – розпізнавання голосових запитів, інтеграція з іншими месенджерами;
- реалізувати адаптивний інтерфейс адміністратора – вебпанель для перегляду історії звернень, редагування бази знань, модерації;
- застосувати методи навчання на основі зворотного зв'язку (RLHF) – система зможе вдосконалювати відповіді на основі оцінки користувачів;
- розгортання в межах усього університету – створення універсальної платформи із сегментацією за факультетами, відділами та типами користувачів.

Таким чином, розроблена система не лише демонструє можливості сучасних AI-рішень у сфері освіти, але й створює базу для подальшої цифрової трансформації взаємодії між здобувачами вищої освіти та університетською адміністрацією.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Nosenko O. V., Nosenko Y. H., Shevchuk R. M. Using the telegram messenger as a tool for supporting the educational process under quarantine restrictions. *Information Technologies and Learning Tools*. 2023. № 94(2). С. 114–127. URL: <https://journal.iitta.gov.ua/index.php/itlt/article/view/5185> (дата звернення: 24.01.2025).
2. Бахмат Н. В. Штучний інтелект у вищій освіті: можливості використання. *Педагогічна освіта: теорія і практика*. 2023. № 35. С. 161–173. URL: <http://pedosv.kpnu.edu.ua/article/view/300712/292914> (дата звернення: 24.01.2025).
3. Шевченко О. М., Олянич В. В., Приймак В. М. Ефективність використання мобільних додатків для підтримки онлайн-навчання. *Педагогічна академія: наукові записки*. 2025. № 14. URL: <https://zenodo.org/records/14754858> (дата звернення: 24.01.2025).
4. Кундіна В. В., Сторожчук Ю. О., Козаренко Т. М. Використання інтерактивних інструментів у дистанційному навчанні медичних спеціалістів. *Академічні візії*. 2024. № 38. URL: <https://www.academyvision.org/index.php/av/article/view/1555/1438> (дата звернення: 24.01.2025).
5. Дзень В., Борзов О., Дзень Д. Інтеграція smart-систем в освітнє середовище закладів вищої освіти. *Вісник Львівського державного університету безпеки життєдіяльності*. 2024. № 30. С. 56–66. URL: <https://journal.ldubgd.edu.ua/index.php/Visnuk/article/view/2768/2654> (дата звернення: 24.01.2025).
6. Краснопольський В. Е., Поліщук О. А., Демченко О. М. Інтеграція мобільних додатків у освітній процес: аналіз ефективності та можливостей для здобувачів освіти. *Академічні візії*. 2024. № 32. URL: <https://academyvision.org/index.php/av/article/view/1174/1041> (дата звернення: 24.01.2025).

7. Крупенина Н. А., Федорчук А. Л., Сабалош В. І. Роль викладачів у нову епоху дистанційного навчання та штучного інтелекту. *Актуальні питання у сучасній науці*. 2023. № 10. С. 565–578.

URL: <http://eprints.zu.edu.ua/38188/1/6799.pdf> (дата звернення: 24.01.2025).

8. Рудніцький Я. Е. Штучний інтелект в освіті: ChatGPT як ключовий інструмент для покращення навчання здобувачів освіти. *Українські студії в європейському контексті*. 2023. № 7. С. 324–327.

URL: http://obrii.org.ua/usec/storage/article/Rudnitskyi_2023_324.pdf (дата звернення: 24.01.2025).

9. Єфименко Т. М. Ефективність використання засобів штучного інтелекту у навчанні іноземних мов і перекладу. *Scientific Achievements of Contemporary Society: The 2nd International scientific and practical conference* (London, September 12–14, 2024). London, 2024. С. 310–315.

URL: <https://sciconf.com.ua/wp-content/uploads/2024/09/SCIENTIFIC-ACHIEVEMENTS-OFCONTEMPORARY-SOCIETY-12-14.09.24.pdf#page=310> (дата звернення: 24.01.2025).

10. Сікора Я. Б., Марчук Н. А., Нестеров В. Ф. Технології майбутнього: роль штучного інтелекту у персоналізованому навчанні. *Наука і техніка*. 2024. № 1. С. 526–537.

URL: <http://perspectives.pp.ua/index.php/nts/article/view/8751/8798> (дата звернення: 24.01.2025).

11. Zadorina O., Hurskaya V., Sobolyeva S., Grekova L., Vasylyuk-Zaitseva S. The Role of Artificial Intelligence in Creation of Future Education: Possibilities and Challenges. *Futurity Education*. 2024. № 4(2). С. 163–185.

URL: <https://futurityeducation.com/index.php/fed/article/view/326/156> (дата звернення: 24.01.2025).

12. Максимчук Б., Максимчук І., Баштовенко О., Ганчева К., Сілаєв В. Умови організації якісного дистанційного навчання у закладах вищої освіти (ЗВО). *Науковий вісник Ізмаїльського державного гуманітарного університету*. 2023. № 64. С. 134–144.

URL: <http://visnyk.idgu.edu.ua/index.php/nv/article/view/764> (дата звернення: 24.01.2025).

13. Використання засобів і сервісів штучного інтелекту для професійного розвитку педагогічних кадрів : препринт (аналітичні матеріали) / В. М. Барладим та ін. / за ред. М. П. Шишкіної. Київ : ІЦО НАПН України, 2024. 96 с. URL: <https://lib.iitta.gov.ua/id/eprint/744000/> (дата звернення: 24.01.2025).

14. Ткаченко Л. П., Волик О. О., Усата О. Ю. Інтерактивні технології в підготовці майбутніх науковців: нові виклики та можливості. Педагогічна Академія: наукові записки. 2025. URL: <https://zenodo.org/records/14923656> (дата звернення: 24.01.2025).

15. Мельник А. Використання чат-бота ChatGPT у практичній мовній підготовці майбутніх викладачів англійської мови. Вісник Національного університету «Чернігівський колегіум» імені Т. Г. Шевченка. 2023. № 177(21). С. 100–107. URL: <https://visnyk.chnpu.edu.ua/index.php/visnyk/article/view/96/95> (дата звернення: 24.01.2025).

16. Kravchuk Y. Crowdsourced data and ai integration in online platforms for volunteer collaboration. Наука і техніка сьогодні. 2024. № 12(40). URL: <http://perspectives.pp.ua/index.php/nts/article/view/17027> (дата звернення: 24.01.2025).

17. Ткаченко К. Використання методів NLP в інтелектуальних навчальних системах. Цифрова платформа: інформаційні технології в соціокультурній сфері. 2024. Т. 7, № 1. С. 80-96. URL: http://nbuv.gov.ua/UJRN/dpitsca_2024_7_1_9 (дата звернення: 25.01.2025).

18. Chukwuere J. E. Інтеграція генеративних чат-ботів ШІ у вищу освіту: виклики та перспективи. arXiv, 2024. №2403.13487. URL: <https://arxiv.org/abs/2403.13487> (дата звернення: 25.01.2025).

19. Wang K., Ramos J., Lawrence R. ChatEd: чат-бот на базі ChatGPT для підтримки навчання у вищій освіті. arXiv, 2023. №2401.00052.

URL: <https://arxiv.org/abs/2401.00052> (дата звернення: 25.01.2025).

20. Isaev R. та ін. HIVA: голографічний інтелектуальний голосовий асистент для закладів вищої освіти. arXiv, 2023. №2307.05501.

URL: <https://arxiv.org/abs/2307.05501> (дата звернення: 25.01.2025).

21. Abedi M. та ін. Використання великих мовних моделей та чат-ботів у вищій технічній освіті. arXiv, 2023. №2309.13059.

URL: <https://arxiv.org/abs/2309.13059> (дата звернення: 25.01.2025).

22. Shaik T. та ін. Тенденції й виклики застосування NLP у сфері освітньої аналітики зворотного зв'язку. arXiv, 2023. №2301.08826.

URL: <https://arxiv.org/abs/2301.08826> (дата звернення: 25.01.2025).

23. Jin Y. та ін. Взаємодія з аналітичними панелями освіти за допомогою генеративних чат-ботів: роль цифрової грамотності. arXiv, 2024. №2411.15597.

URL: <https://arxiv.org/abs/2411.15597> (дата звернення: 25.01.2025).

24. Hasan M. R. та ін. Прийняття чат-ботів у навчанні: структурне моделювання сприйняття студентів. arXiv, 2023. №2311.03636.

URL: <https://arxiv.org/abs/2311.03636> (дата звернення: 25.01.2025).

25. Mai L., Carson-Berndsen J. GPT-4 для корекції помилок у чат-ботах мовного навчання: підвищення якості діалогу. arXiv, 2023. №2307.09744.

URL: <https://arxiv.org/abs/2307.09744> (дата звернення: 25.01.2025).

26. Dan Y. та ін. EduChat: чат-бот на основі великих мовних моделей для інтелектуальної освіти. arXiv, 2023. №2308.02773.

URL: <https://arxiv.org/abs/2308.02773> (дата звернення: 25.01.2025).

27. Deshpande A. та ін. Токсичність ChatGPT: ризики призначення особистостей для моделей діалогу. arXiv, 2023. №2304.05335.

URL: <https://arxiv.org/abs/2304.05335> (дата звернення: 25.01.2025).

28. Tafazoli D. Критичний аналіз використання ШІ в комунікації під час мовного навчання. arXiv, 2023. №2305.11897.

URL: <https://arxiv.org/abs/2305.11897> (дата звернення: 25.01.2025).

29. Chan J., Li Y. Персоналізоване навчання у вищій освіті за допомогою мультимодальних чат-ботів на базі ChatGPT і Bard. arXiv, 2025. №2502.07401. URL: <https://arxiv.org/abs/2502.07401> (дата звернення: 25.01.2025).
30. Poudel S. та ін. Порівняльне дослідження генеративних і класифікаційних моделей для чат-ботів. arXiv, 2023. №2311.06898. URL: <https://arxiv.org/abs/2311.06898> (дата звернення: 25.01.2025).
31. Shi N., Zeng Q., Lee R. Створення чат-бота для мовного навчання з використанням GPT-2 та XAI. arXiv, 2020. №2009.13984. URL: <https://arxiv.org/abs/2009.13984> (дата звернення: 25.01.2025).
32. Nosenko O. V., Nosenko Y. H., Shevchuk R. M. Using the telegram messenger as a tool for supporting the educational process under quarantine restrictions. Information Technologies and Learning Tools. 2023. № 94(2). С. 114–127. URL: <https://journal.iitta.gov.ua/index.php/itlt/article/view/5185> (дата звернення: 24.01.2025).

ДОДАТКИ

ДОДАТОК А

Скрипт файлу discord_bot.py

```
import os
import discord
from discord.ext import commands
from sentence_transformers import SentenceTransformer
import chromadb
import requests

CHROMA_DB_DIR = "./chroma_db"
TOKEN =
"MTM1ODc3MTM3ODgwMjc5MDU4MQ.GcB9gl._xJmFTUSc2vi6HyVEVrECRqPP9XZd
SxHJDMC"
OLLAMA_URL = "http://localhost:11434/api/generate"
N_RESULTS = 12

intents = discord.Intents.default()
intents.message_content = True

bot = commands.Bot(command_prefix="!", intents=intents)
embedder = SentenceTransformer("all-MiniLM-L6-v2")

chroma_client = chromadb.PersistentClient(path=CHROMA_DB_DIR)
collection = chroma_client.get_or_create_collection(name="docs")

PROMPT_TEMPLATE = """Відповідай лише цитуючи слова з контексту
нижче.
Якщо є chunk з визначенням "академічна година" – процитуй цей
chunk дослівно.
Якщо в контексті нема такої інформації, напиши "Інформації
недостатньо".

Контекст:
{context}

Запит:
{question}

Відповідь:
"""

def get_context(user_question, n_results=N_RESULTS):
    query_embedding = embedder.encode([user_question])[0]
    results =
collection.query(query_embeddings=[query_embedding],
n_results=n_results)
    context_chunks = results.get("documents", [[]])[0]
    context = "\n\n".join(context_chunks)
    return context

def get_response_from_mistral(prompt):
    data = {
```

```

        "model": "mistral",
        "prompt": prompt,
        "stream": False
    }
    try:
        resp = requests.post(OLLAMA_URL, json=data, timeout=120)
        if resp.status_code == 200:
            result = resp.json()
            return result.get("response", "Інформації
недостатньо")
        else:
            return f"Помилка Mistral: {resp.text}"
    except Exception as e:
        return f"✗ Помилка підключення до Mistral: {e}"

@bot.event
async def on_ready():
    print(f"✓ Бот запущено як {bot.user}")

@bot.event
async def on_message(message):
    if message.author == bot.user:
        return

    user_text = message.content.strip()
    if not user_text:
        return

    try:
        context = get_context(user_text)
        prompt = PROMPT_TEMPLATE.format(context=context,
question=user_text)
        answer = get_response_from_mistral(prompt)
        await message.channel.send(answer.strip())
    except Exception as e:
        await message.channel.send(f"✗ Помилка при пошуку або
відповіді: {e}")

    await bot.process_commands(message)

bot.run(TOKEN)
```

ДОДАТОК Б

Скрипт файлу add_documents_to_db.py

```
import os
import re
from sentence_transformers import SentenceTransformer
import chromadb
from PyPDF2 import PdfReader
from docx import Document

DOCUMENTS_FOLDER = "./documents"
CHROMA_DB_DIR = "./chroma_db"
os.makedirs(DOCUMENTS_FOLDER, exist_ok=True)

chroma_client = chromadb.PersistentClient(path=CHROMA_DB_DIR)
collection = chroma_client.get_or_create_collection(name="docs")
embedder = SentenceTransformer("all-MiniLM-L6-v2")

def extract_text_from_pdf(file_path):
    text = ""
    with open(file_path, "rb") as f:
        reader = PdfReader(f)
        for page in reader.pages:
            content = page.extract_text()
            if content:
                text += content + "\n"
    return text.strip()

def extract_text_from_docx(file_path):
    text = ""
    doc = Document(file_path)
    for para in doc.paragraphs:
        if para.text:
            text += para.text + "\n"
    return text.strip()

def extract_text_from_txt(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        return f.read().strip()

def extract_text_from_file(file_path):
    if file_path.lower().endswith(".pdf"):
        return extract_text_from_pdf(file_path)
    elif file_path.lower().endswith(".docx"):
        return extract_text_from_docx(file_path)
    elif file_path.lower().endswith(".txt"):
        return extract_text_from_txt(file_path)
    else:
        return ""

def split_text_by_sentences(text, n_sentences=2,
min_chunk_length=20):
    # Розбиваємо по реченнях (українські/російські/англійські)
```

```

sentences = re.split(r'(?<=[.!?])\s+', text)
sentences = [s.strip() for s in sentences if s.strip()]
chunks = []
for i in range(0, len(sentences), n_sentences):
    chunk = " ".join(sentences[i:i + n_sentences])
    if len(chunk) > min_chunk_length:
        chunks.append(chunk)
return chunks

def process_file(file_path):
    filename = os.path.basename(file_path)
    text = extract_text_from_file(file_path)
    if not text:
        print(f"⚠ Файл {filename} не містить тексту або не
підтримується.")
        return
    chunks = split_text_by_sentences(text, n_sentences=3,
min_chunk_length=20)
    print(f"{filename}: знайдено {len(chunks)} chunk-ів")
    new_chunks, new_ids = [], []
    existing_ids = set(collection.get()["ids"])
    for i, chunk in enumerate(chunks):
        chunk_id = f"{filename}_{i}"
        if chunk_id not in existing_ids:
            new_chunks.append(chunk)
            new_ids.append(chunk_id)
    if new_chunks:
        embeddings = embedder.encode(new_chunks)
        collection.add(documents=new_chunks,
embeddings=embeddings, ids=new_ids)
        print(f"✓ Додано {len(new_chunks)} фрагментів із
{filename}")
    else:
        print(f"❗ Усі chunk-и з {filename} вже в базі.")

if __name__ == "__main__":
    files = [f for f in os.listdir(DOCUMENTS_FOLDER) if
f.lower().endswith((".pdf", ".docx", ".txt"))]
    if not files:
        print("⚠ У папці documents немає документів для
індексації.")
    for file in files:
        process_file(os.path.join(DOCUMENTS_FOLDER, file))
    print(f"\n📊 Загалом у базі: {collection.count()}
фрагментів")
    print("✓ Готово.")
    res = collection.get()
    print("Перші 3 chunk-и:", res["documents"][:3])

```

ДОДАТОК В

Скрипт файлу rag_autotest.py

```
import chromadb
from sentence_transformers import SentenceTransformer
import requests
import re
import csv

# --- Для додаткових метрик:
import sacrebleu
from rouge_score import rouge_scorer
from bert_score import score as bert_score
from nltk.translate.meteor_score import meteor_score
from sklearn.metrics.pairwise import cosine_similarity

CHROMA_DB_DIR = "./chroma_db"
MODEL_NAME = "all-MiniLM-L6-v2"
OLLAMA_URL = "http://localhost:11434/api/generate"
CHROMA_COLLECTION = "docs"
N_RESULTS = 7

chroma_client = chromadb.PersistentClient(path=CHROMA_DB_DIR)
collection =
chroma_client.get_or_create_collection(name=CHROMA_COLLECTION)
embedder = SentenceTransformer(MODEL_NAME)

# --- УНІФІКОВАНИЙ PROMPT ---
PROMPT_TEMPLATE = """Відповідай лише цитуючи слова з контексту
нижче.
Якщо є chunk з визначенням "академічна година" – процитуй цей
chunk дослівно.
Якщо в контексті нема такої інформації, напиши "Інформації
недостатньо".

Контекст:
{context}

Запит:
{question}

Відповідь:
"""

# --- Ваш тестовий набір (приклади):
test_set = [
    {
        "question": "Що таке академічна година?",
        "expected": "Академічна година – мінімальна облікова
одиниця навчального часу, яка слугує основою для планування та
обліку навчальних занять. Тривалість академічної години
становить, як правило, 45 хв."
    }
]
```

```

    },
    # ... (інші питання)
]

def normalize_text(text):
    return re.sub(r'\W+', ' ', text.lower()).strip()

def exact_match(pred, gold):
    return int(normalize_text(pred) == normalize_text(gold))

def substring_match(pred, gold):
    return int(normalize_text(gold) in normalize_text(pred))

def f1_metric(pred, gold):
    pred_tokens = set(normalize_text(pred).split())
    gold_tokens = set(normalize_text(gold).split())
    common = pred_tokens & gold_tokens
    if not pred_tokens or not gold_tokens:
        return 0.0
    precision = len(common) / len(pred_tokens)
    recall = len(common) / len(gold_tokens)
    if precision + recall == 0:
        return 0.0
    return 2 * (precision * recall) / (precision + recall)

def bleu_metric(pred, gold):
    return sacrebleu.sentence_bleu(pred, [gold]).score / 100

def rouge_metric(pred, gold):
    scorer = rouge_scorer.RougeScorer(['rougeL'],
    use_stemmer=True)
    scores = scorer.score(gold, pred)
    return scores['rougeL'].fmeasure

def bertscore_metric(pred, gold):
    P, R, F1 = bert_score([pred], [gold], lang="uk",
    verbose=False)
    return float(F1[0])

def meteor_metric(pred, gold):
    return meteor_score([gold], pred)

def chr_f_metric(pred, gold):
    return sacrebleu.sentence_chrf(pred, [gold]).score / 100

def semantic_similarity(pred, gold):
    emb_pred = embedder.encode([pred])
    emb_gold = embedder.encode([gold])
    return float(cosine_similarity(emb_pred, emb_gold)[0, 0])

def get_context(user_question, n_results=N_RESULTS):
    query_embedding = embedder.encode([user_question])[0]

```



```

    results =
collection.query(query_embeddings=[query_embedding],
n_results=n_results)
    context_chunks = results.get("documents", [[]])[0]
    context = "\n\n".join(context_chunks)
    print(f"\n[DEBUG] Context для питання
'{user_question}':\n{context}\n")
    return context

def ask_mistral(question):
    context = get_context(question)
    prompt = PROMPT_TEMPLATE.format(context=context,
question=question)
    data = {
        "model": "mistral",
        "prompt": prompt,
        "stream": False
    }
    try:
        resp = requests.post(OLLAMA_URL, json=data, timeout=120)
        if resp.status_code == 200:
            result = resp.json()
            return result.get("response", "")
        else:
            print(f"Помилка Ollama: {resp.text}")
            return ""
    except Exception as e:
        print(f"✗ Помилка підключення до Mistral: {e}")
        return ""

results = []
for case in test_set:
    question = case["question"]
    expected = case["expected"]
    print(f"\n---\nПитання: {question}")
    pred = ask_mistral(question)
    print(f"Відповідь моделі: {pred}")
    print(f"Очікувана відповідь: {expected}")
    em = exact_match(pred, expected)
    sm = substring_match(pred, expected)
    f1 = f1_metric(pred, expected)
    bleu = bleu_metric(pred, expected)
    rouge = rouge_metric(pred, expected)
    try:
        bert = bertscore_metric(pred, expected)
    except Exception as e:
        bert = 0.0
        print(f"BERTScore error: {e}")
    try:
        meteor = meteor_metric(pred, expected)
    except Exception as e:
        meteor = 0.0
        print(f"METEOR error: {e}")

```

```

chrf = chrf_metric(pred, expected)
sem_sim = semantic_similarity(pred, expected)
results.append({
    "question": question,
    "expected": expected,
    "prediction": pred,
    "em": em,
    "sm": sm,
    "f1": f1,
    "bleu": bleu,
    "rougeL": rouge,
    "bertscore": bert,
    "meteor": meteor,
    "chrf": chrf,
    "sem_sim": sem_sim
})

# Зведені результати
mean_em = sum(r["em"] for r in results) / len(results)
mean_sm = sum(r["sm"] for r in results) / len(results)
mean_f1 = sum(r["f1"] for r in results) / len(results)
mean_bleu = sum(r["bleu"] for r in results) / len(results)
mean_rouge = sum(r["rougeL"] for r in results) / len(results)
mean_bert = sum(r["bertscore"] for r in results) / len(results)
mean_meteor = sum(r["meteor"] for r in results) / len(results)
mean_chrf = sum(r["chrf"] for r in results) / len(results)
mean_sem_sim = sum(r["sem_sim"] for r in results) / len(results)

print("\n===== ЗВЕДЕНІ РЕЗУЛЬТАТИ =====")
print(f"Exact Match: {mean_em:.2f}")
print(f"Substring Match: {mean_sm:.2f}")
print(f"F1 Score: {mean_f1:.2f}")
print(f"BLEU: {mean_bleu:.2f}")
print(f"ROUGE-L: {mean_rouge:.2f}")
print(f"BERTScore: {mean_bert:.2f}")
print(f"METEOR: {mean_meteor:.2f}")
print(f"ChrF: {mean_chrf:.2f}")
print(f"Semantic Similarity (Cosine): {mean_sem_sim:.2f}")

with open("autotest_results.csv", "w", newline='',
encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=[
        "question", "expected", "prediction",
        "em", "sm", "f1", "bleu", "rougeL", "bertscore",
        "meteor", "chrf", "sem_sim"
    ])
    writer.writeheader()
    writer.writerows(results)

print("\nРезультати збережено у autotest_results.csv")

```