Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра системного аналізу та інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

“ Інформаційна технологія розроблення інтелектуального чат-асистента на базі великих мовних моделей і RAG ”

Виконав: студент 2 курсу, групи 2ІСТ-24м
спеціальності 126 – «Інформаційні системи та технології»
_____Артем БУРЛАЧЕНКО
Керівник: к.т.н., доц. каф. САІТ
_____Сергій ЖУКОВ
«27» 11 2025 р.
Опонент: к.т.н., доц. каф. КН
_____ Олексій СІЛАГІН
«03» 12 2025 р.

Допущено до захисту
Завідувач кафедри САІТ
_____ д.т.н., проф. Віталій МОКІН
«28» 11 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра системного аналізу та інформаційних технологій
Рівень вищої освіти – другий (магістерський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 126 Інформаційні системи та технології
Освітньо-професійна програма – Інформаційні технології аналізу даних та зображень

ЗАТВЕРДЖУЮ
Завідувач кафедри САІТ
_____ д.т.н., проф. Віталій МОКІН
«25» 09 2025 року

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Бурлаченку Артему Віталійовичу

1. Тема роботи: «Інформаційна технологія розроблення інтелектуального чат-асистента на базі великих мовних моделей і RAG»
керівник роботи: Сергій ЖУКОВ, к.т.н., доц. каф. САІТ,
затверджені наказом ВНТУ від «24» 09 2025 року № 313
2. Строк подання студентом роботи «28» 11 2025 року
3. Вихідні дані до роботи:
Технічна документація PyTorch та JAX.
4. Зміст текстової частини:
- загальна характеристика об'єкту дослідження;
- постановка задачі та вибір оптимальних інформаційних технологій для її розв'язання;
- побудова та оцінювання моделей нейронних мереж для прогнозування якості поверхневих вод;
- розвідувальний аналіз даних та побудова моделей.
- економічна частина

5. Перелік ілюстративного матеріалу:
- ілюстративні матеріали;
- обрані оптимальні інструменти і технології;
- знімки екранів із графіками розвідувального аналізу;
- знімки результатів роботи пошуку;
- блок-схема інформаційної системи;

6. Консультанти розділів МКР

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
2	Віталій МОКІН, д. т. н., професор	25.09.2025	05.10.2025
5	Олександр ЛЕСЬКО, к. е. н., проф. каф. ЕПВМ	20.10.2025	19.11.2025

7. Дата видачі завдання «25» 09 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз предметної області	15.09.2025	25.09.2025	виконано
2	Вибір оптимальних інформаційних технологій	25.09.2025	05.10.2025	виконано
3	Вибір мови програмування та середовища розробки	05.10.2025	20.10.2025	виконано
4	Економічна частина	20.10.2025	10.11.2025	виконано
5	Програмна реалізація	10.11.2025	15.11.2025	виконано
6	Оформлення матеріалів до захисту МКР	15.11.2025	25.11.2025	виконано

Студент _____ Артем БУРЛАЧЕНКО

Керівник роботи _____ Сергій ЖУКОВ

АНОТАЦІЯ

УДК 004.8:004.9

Бурлаченко А. В. Інформаційна технологія розроблення інтелектуального чат-асистента на базі великих мовних моделей і RAG Магістерська кваліфікаційна робота зі спеціальності 126 – інформаційні системи та технології, освітньо-професійна програма – інформаційні технології аналізу даних та зображень. Вінниця: ВНТУ, 2025. 103 с.

На укр. мові. Бібліогр.: 20 назв; рис.: 29; табл.: 5.

У магістерській роботі розроблено та досліджено інформаційну технологію інтелектуального семантичного пошуку програмних конструкцій у технічній документації фреймворків JAX та PyTorch на основі векторних баз даних і великих мовних моделей. На основі аналізу релевантності результатів пошуку та експериментального порівняння конфігурацій векторного представлення обґрунтовано параметри системи та налаштовано оптимальний режим роботи для задач підтримки розробників.

Об’єктом дослідження є процес інтелектуального пошуку, індексації та аналізу технічної документації фреймворків машинного навчання на основі векторних подань тексту.

Ілюстративна частина складається з 6 плакатів із результатами аналізу

У розділі економічної частини розглянуто питання доцільності розробки та впровадження інформаційної технології інтелектуального семантичного пошуку документації фреймворків JAX та PyTorch, проведено оцінку витрат на створення програмного продукту та розраховано економічний ефект від його використання в команді розробників.

Ключові слова: векторні бази даних, семантичний пошук, технічна документація, фреймворки машинного навчання, JAX, PyTorch.

ABSTRACT

Burlachenko A. V. Information technology for developing an intelligent chat assistant based on large language models and RAG. Master's thesis in specialty 126 – Information Systems and Technologies, educational and professional program – Information Technologies of Data and Image Analysis. Vinnytsia: VNTU, 2025. 103 p.

In Ukrainian. Bibliogr.: 20 titles; fig.: 29; tables: 5.

The master's thesis develops and investigates an information technology for intelligent semantic search of programming constructs in the technical documentation of the JAX and PyTorch frameworks, based on vector databases and large language models. A complete processing pipeline is designed and implemented, including parsing of HTML documentation, semantic text chunking, vectorization using a local embedding model, and indexing in a specialized vector store. Based on the analysis of search relevance and experimental comparison of vector representation configurations, the system parameters are justified and an optimal operating mode is selected for developer support tasks. The object of the study is the process of intelligent search, indexing, and analysis of technical documentation of machine learning frameworks based on vector text representations.

The illustrative part consists of 6 posters with the results of analysis.

In the economic section, the feasibility of developing and implementing the information technology for intelligent semantic search in the documentation of the JAX and PyTorch frameworks is considered, the costs of creating the software product are estimated, and the economic effect of its use in a development team is calculated.

Keywords: vector databases, semantic search, technical documentation, machine learning frameworks, JAX, PyTorch

ЗМІСТ

ВСТУП

Актуальність теми. Стрімкий розвиток технологій штучного інтелекту та машинного навчання супроводжується зростанням обсягів технічної документації. Для ефективної роботи з сучасними фреймворками, такими як JAX та PyTorch, розробникам необхідно швидко знаходити релевантну інформацію, приклади коду та описи API. Традиційні методи пошуку за ключовими словами часто є неефективними через складність термінології, синонімію та контекстну залежність запитів. Використання великих мовних моделей (LLM) без доступу до актуальних даних призводить до генерації фактичних помилок («галюцинацій») та надання застарілої інформації.

Розробка інтелектуальних систем на основі архітектури RAG (Retrieval-Augmented Generation), що поєднують семантичний пошук, лексичний аналіз та генеративні можливості ШІ, дозволяє вирішити цю проблему. Така технологія забезпечує точність, актуальність та контекстуальне розуміння запитів, що суттєво підвищує продуктивність розробників і є критично важливим завданням у сфері інженерії програмного забезпечення.

Мета і завдання роботи. Метою роботи є підвищення ефективності пошуку та отримання технічної інформації для розробників систем машинного навчання шляхом створення інтелектуального чат-асистента на основі гібридного пошуку та технології RAG. Розробка інформаційної технології інтелектуального аналізу документації JAX та PyTorch передбачає виконання наступних задач:

- створити векторну базу знань;
- реалізувати алгоритми гібридного пошуку;
- розробити механізм багатофакторного переранжування;
- спроектувати архітектуру бекенду;
- здійснити візуалізацію геопросторових зв’язків;
- застосувати ШІ для обробки природномовних запитів;
- підібрати оптимальні ІТ-рішення та моделі машинного навчання;

Об’єктом дослідження магістерської кваліфікаційної роботи є процес інтелектуального пошуку та обробки інформації в масивах технічної документації фреймворків машинного навчання.

Предметом дослідження магістерської кваліфікаційної роботи є методи та технології побудови RAG-систем, алгоритми гібридного пошуку, методи векторизації тексту та стратегії ранжування для підвищення релевантності відповідей на технічні запити.

Методи дослідження. У роботі використовувались методи системного аналізу для проектування архітектури ПЗ, методи обробки природної мови (NLP) для векторизації тексту та аналізу запитів, алгоритми інформаційного пошуку (Cosine Similarity, BM25) для знаходження релевантних документів, методи машинного навчання для ранжування результатів, а також технології розробки веб-сервісів (FastAPI, AsyncIO) для реалізації програмної системи.

Новизна одержаних результатів. Полягає у створенні удосконаленої інформаційної технології пошуку в технічній документації, яка, на відміну від існуючих рішень, використовує гібридний підхід (семантичний + лексичний пошук) з адаптивним багатофакторним переранжуванням (re-ranking). Запропонований підхід враховує специфічні метрики, такі як версія бібліотеки, позиція фрагмента в документі та щільність коду, що дозволяє значно зменшити кількість фактичних помилок та підвищити точність відповідей у вузькоспеціалізованих технічних доменах.

Практичне значення роботи. Розроблена система дозволяє розробникам суттєво скоротити час на пошук документації, отримувати точні відповіді з посиланнями на джерела та уникати використання застарілих API. Реалізоване рішення готове до інтеграції в процеси розробки ПЗ і демонструє високу швидкодію (латентність до 180 мс) та точність пошуку (MRR 1.0 на тестовому наборі).

Апробація та публікації результатів магістерської кваліфікаційної роботи. Опубліковано тези на LV Всеукраїнській науково-технічній

конференції підрозділів Вінницького національного технічного університету (ВНТКП ВНТУ) (2025-2026) [1].

1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1 Аналіз предметної області

В сучасному світі, де інформація є найціннішим активом, здатність швидко отримувати точні та релевантні дані визначає ефективність бізнес-процесів та якість обслуговування клієнтів. Одним з ключових інструментів для досягнення цієї мети стали інтелектуальні чат-асистенти.

Історія розмовних агентів бере свій початок у 1960-х роках з появою програми ELIZA, яка імітувала діалог з психотерапевтом, використовуючи прості шаблони та заміну ключових слів. В наступні десятиліття з'явилися системи, що базувалися на правилах, де логіка діалогу жорстко прописувалася розробниками. Такі системи були ефективними у вузькоспеціалізованих задачах, але були абсолютно негнучкими і не могли впоратися із запитами, що виходили за рамки прописаних сценаріїв.

Справжня революція відбулася з розвитком машинного навчання та появою архітектури Трансформер у 2017 році. Ця архітектура, що базується на механізмі уваги, дозволила моделям зважувати важливість різних слів у вхідному тексті, що призвело до значного покращення розуміння контексту. На базі цієї архітектури були створені великі мовні моделі, такі як GPT від OpenAI, Llama від Meta та інші. Великі мовні моделі навчаються на величезних масивах текстових даних з Інтернету, що дозволяє їм вивчити закономірності людської мови, факти про світ, стилістику та логіку [2].

Незважаючи на свої вражаючі можливості, стандартні великі мовні моделі мають ряд фундаментальних обмежень, що ускладнюють їх використання в корпоративному середовищі. По-перше, моделі іноді генерують правдоподібну, але фактично невірну або вигадану інформацію. Це відбувається тому, що вони є генеративними системами, оптимізованими для створення тексту, а не для перевірки фактів. По-друге, знання моделі обмежені

даними, на яких вона навчалася. Вони не знають про події, що сталися після дати зрізу їхніх знань, і не мають доступу до приватної, внутрішньої документації компанії. По-третє, стандартна модель не може вказати джерело інформації, на основі якого вона згенерувала відповідь, що унеможливлює перевірку достовірності.

Для вирішення цих проблем була запропонована архітектура Retrieval-Augmented Generation [2]. Ідея архітектури полягає в тому, щоб заземлити мовну модель на зовнішній, достовірній базі знань. Замість того, щоб одразу генерувати відповідь, система спочатку виконує пошук релевантної інформації у спеціалізованому сховищі, а потім використовує знайдені фрагменти як контекст для генерації відповіді.

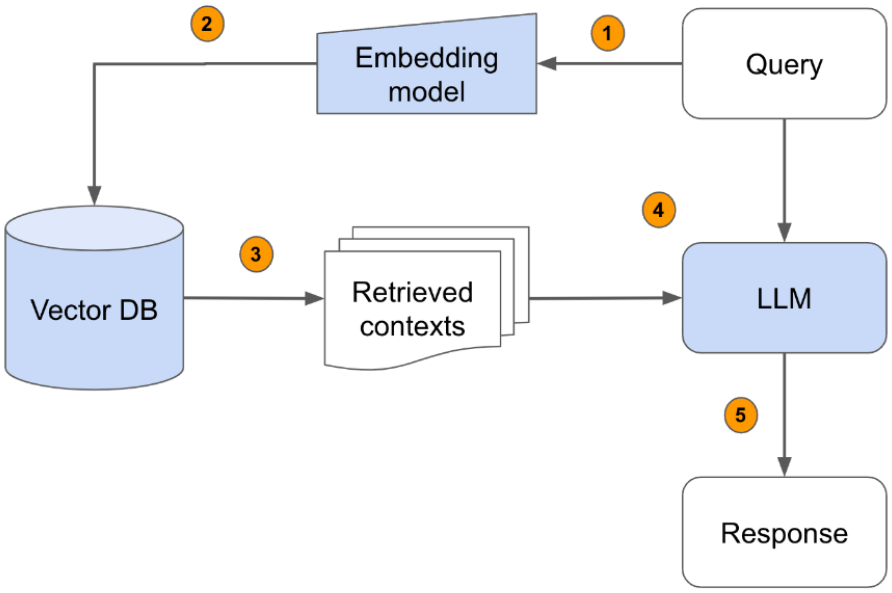

Рисунок 1.1 – Загальна схема архітектури RAG

На рисунку 1.1 представлена загальна схема роботи RAG системи, яка складається з п’яти ключових етапів. Процес починається з запиту користувача (Query), який потрапляє до моделі векторизації (Embedding model). Ця модель перетворює текстовий запит на числовий вектор, який потім

використовується для пошуку у векторній базі даних (Vector DB). Система знаходить найбільш релевантні фрагменти документів (Retrieved contexts), які разом з оригінальним запитом передаються до великої мовної моделі (LLM) для генерації фінальної відповіді (Response).

Процес RAG складається з двох основних етапів. Етап індексації включає завантаження всієї корпоративної бази знань, розбиття її на невеликі фрагменти та перетворення на числові вектори за допомогою спеціальної моделі-енкодера. Ці вектори зберігаються у векторній базі даних. Етап пошуку та генерації починається з перетворення запиту користувача на вектор. Система здійснює пошук у векторній базі даних, щоб знайти вектори, які є найбільш близькими до вектора запиту. Ця близькість зазвичай розраховується за допомогою косинусної подібності, яка визначає кут між двома векторами в багатовимірному просторі.

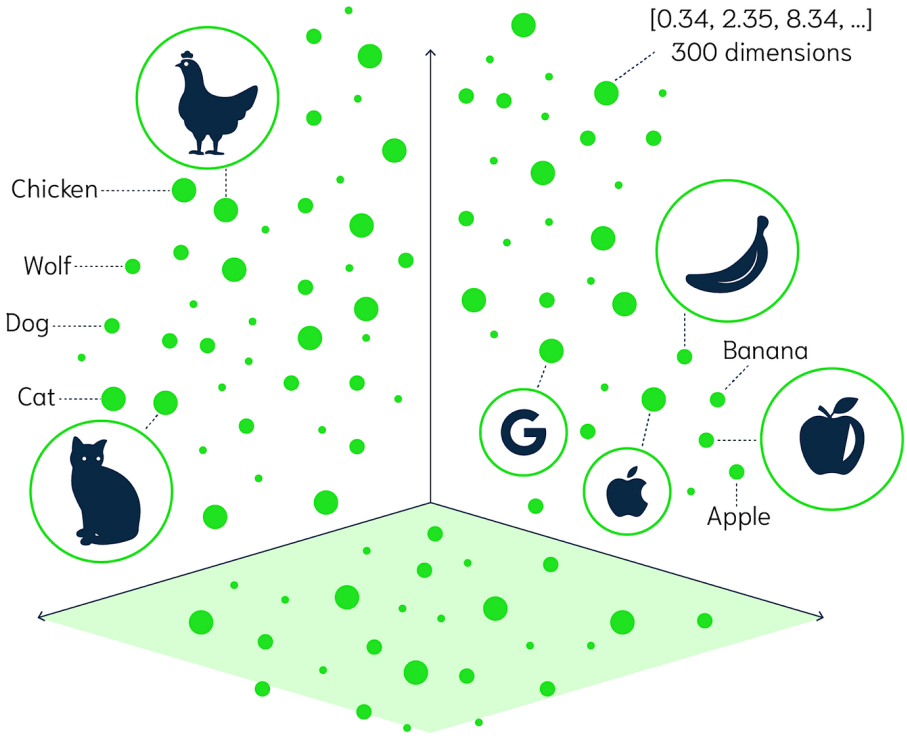

Рисунок 1.2 – Векторна подібність у двовимірному просторі

Розглядаючи рисунок 1.2, ми бачимо наочну демонстрацію того, як працює векторна подібність у спрощеному двовимірному просторі. На схемі

показано, як різні концепції групуються за семантичною схожістю: тварини (курка, вовк, собака, кіт) розташовані в одній області простору, а фрукти (банан, яблуко) - в іншій. Цікавим є розташування логотипів відомих компаній Google та Apple, які також мають свої унікальні позиції у векторному просторі. Така візуалізація допомагає зрозуміти, як система RAG може відрізняти семантично схожі та різні концепції [2].

Знайдені релевантні фрагменти тексту передаються великій мовній моделі разом із початковим запитом користувача. Таким чином, модель отримує необхідний контекст для генерації точної, фактично обґрунтованої відповіді, і може навіть посилатися на джерела, з яких була взята інформація.

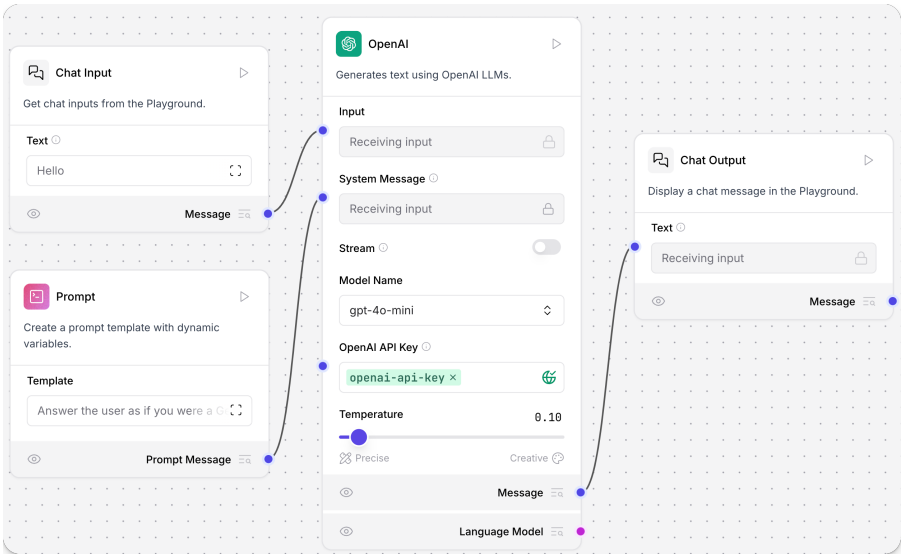

Рисунок 1.3 – Детальна схема векторного пошуку

На рисунку 1.3 ми бачимо більш детальну схему, що ілюструє процес пошуку релевантних документів. Схема показує три етапи обробки запиту: спочатку запит перетворюється на вектор, потім система шукає найближчі векторні подібності у просторі, і нарешті відбирає найрелевантніші фрагменти для передачі до великої мовної моделі.

Створення RAG-систем з нуля є складним завданням, що вимагає глибоких знань у програмуванні та машинному навчанні. Для спрощення цього процесу з’явилися спеціалізовані фреймворки. Langflow є інструментом

для візуального прототипування та розробки додатків на базі великих мовних моделей. Він побудований на основі популярної бібліотеки LangChain [3] і надає графічний інтерфейс, де розробник може будувати складні ланцюжки та графи обробки даних, просто перетягуючи та з'єднуючи готові компоненти.

Створення інтелектуального асистента в Langflow відбувається через побудову потоку - візуального графу, що складається з взаємопов'язаних компонентів. Кожен компонент виконує специфічну функцію в загальному ланцюжку обробки даних.

Вхідні компоненти включають Chat Input для прийому текстових повідомлень від користувачів, Text Input для статичного тексту або параметрів, та File Upload для завантаження документів. Компоненти обробки даних містять Document Loaders для завантаження різних типів документів, Text Splitters для розбиття документів на менші фрагменти, та Embeddings для перетворення тексту на векторні представлення.

Векторні сховища представлені такими компонентами як ChromaDB для локальної векторної бази даних, Pinecone для хмарного векторного сховища, та FAISS для ефективного пошуку подібності. Мовні моделі включають OpenAI для доступу до GPT-3.5 та GPT-4, Anthropic для моделей Claude, та Local Models для роботи з локальними моделями через Ollama та HuggingFace.

Агенти в Langflow є спеціальними компонентами, що можуть приймати рішення про те, які дії виконувати на основі вхідних даних. Процес додавання агента включає вибір типу агента, де доступні ReAct Agent для логіки міркування та дій, OpenAI Functions Agent для роботи з функціями OpenAI, Conversational Agent для підтримки діалогового контексту, та Custom Agent для створення власної логіки [3].

Налаштування інструментів агента дозволяє використовувати різні засоби для розширення можливостей, включаючи Search Tools для пошуку в Інтернеті або базах даних, Calculator для математичних обчислень, File Tools для роботи з файлами, та API Tools для викликів зовнішніх інтерфейсів програмування.

Конфігурація промптів агента включає System Prompt з базовими інструкціями для агента, Human Prompt як шаблон для запитів користувачів, та AI Prompt для формату відповідей.

Візуальний підхід Langflow має значні переваги. Швидкість прототипування досягається завдяки візуальному інтерфейсу, що дозволяє експериментувати з архітектурою системи в режимі реального часу. Демократизація розробки означає, що створювати складні додатки можуть не тільки програмісти, а й аналітики та продакт-менеджери. Наочність забезпечується графічним представленням потоку даних, що значно спрощує розуміння та налагодження системи. Модульність дозволяє налаштовувати кожен компонент незалежно, що спрощує тестування та оптимізацію. Повторне використання створених потоків можливе завдяки збереженню їх як шаблонів для інших проєктів.

Інтеграція з зовнішніми системами здійснюється через REST API з автоматичним створенням кінцевих точок для кожного потоку, Webhooks для реагування на зовнішні події, Database Connections для прямого підключення до баз даних, та Custom Components для створення власних компонентів на Python.

1.2 Огляд аналогів

Ринок рішень для створення розмовних інтерфейсів та інтелектуальних асистентів є досить широким і включає в себе кілька категорій продуктів, кожна з яких має свої переваги та недоліки.

Багато провідних розробників систем управління відносинами з клієнтами вже пропонують власні інтегровані рішення штучного інтелекту. Ці системи глибоко вбудовані в екосистему платформи і призначені для автоматизації типових завдань.

Salesforce Einstein 1 є одним з найпотужніших асистентів на ринку. Einstein AI аналізує дані з усієї платформи Salesforce, надаючи прогнози щодо

угод, рекомендації щодо наступних дій, автоматично генеруючи звіти та персоналізовані повідомлення електронною поштою. Його сильною стороною є глибока інтеграція та здатність працювати з величезними обсягами даних. Однак, це рішення є дорогим і вимагає значних зусиль для налаштування.

HubSpot AI з ChatSpot пропонує набір інструментів штучного інтелекту, центральним з яких є ChatSpot. Він дозволяє користувачам взаємодіяти з системою управління відносинами з клієнтами за допомогою команд природною мовою: створювати звіти, додавати контакти, писати листи. HubSpot AI фокусується на простоті використання та швидкому впровадженні, що робить його привабливим для малого та середнього бізнесу.

Bitrix24 CoPilot є власним асистентом Bitrix24, який також інтегрований у різні модулі системи. CoPilot може транскрибувати дзвінки, автоматично заповнювати поля в картках системи управління відносинами з клієнтами, генерувати тексти для постів у стрічці новин та завдань. Його основна перевага - безшовна інтеграція в єдиний робочий простір Bitrix24.

Недоліком цих інтегрованих рішень є їхня закритість. Вони оптимізовані для роботи з даними всередині своєї платформи і зазвичай пропонують обмежені можливості для налаштування логіки роботи асистента або підключення до специфічних, нестандартних зовнішніх джерел знань.

Категорія інструментів для візуальної розробки ботів дозволяє створювати чат-ботів у графічному інтерфейсі, а їхні підходи можуть сильно відрізнятися. Класичні платформи, такі як Google Dialogflow та Microsoft Bot Framework, є індустріальними стандартами, але вони базуються на концепції намірів. Розробник має вручну визначити всі можливі наміри користувача та надати безліч прикладів фраз для їх тренування. Це робить процес розробки трудомістким, а боти погано справляються з питаннями, які не були передбачені [4].

Візуальні конструктори на базі великих мовних моделей, такі як Langflow та FlowiseAI, є новою хвилею розвитку. Вони не вимагають визначення намірів. Замість цього, вони надають візуальний доступ до

будівельних блоків сучасних додатків на основі великих мовних моделей, таких як RAG. Вони дозволяють користувачеві повністю контролювати архітектуру додатку, вибирати моделі, налаштовувати промпти та ланцюжки обробки.

Для розробників, які потребують максимальної гнучкості, існують спеціалізовані бібліотеки. LangChain є фреймворком, який надає набір абстракцій та інструментів для побудови додатків на основі великих мовних моделей. Він містить компоненти для роботи з моделями, промптами, джерелами даних, пам’яттю та агентами. LangChain є надзвичайно потужним, але вимагає написання коду.

LlamaIndex спеціалізується саме на системах RAG. Ця бібліотека пропонує оптимізовані інструменти для завантаження, індексації та пошуку даних у великих обсягах документів.

1.3 Вибір оптимальних інформаційних технологій

На основі проведеного аналізу предметної області та огляду аналогів, для реалізації мети кваліфікаційної роботи було обрано наступний стек технологій. Вибір кожного компонента обґрунтовується його перевагами в контексті даного проєкту.

Як середовище розробки пайплайну RAG [4] обрано LangChain, що забезпечує глибоку гнучкість і контроль на рівні коду. На відміну від візуальних інструментів на кшталт Langflow, LangChain дозволяє чітко документувати кожен крок обробки даних та легко інтегруватися з системами керування версіями, такими як Git.

Як велику мовну модель обрано сімейство моделей Gemini від Google. Моделі Gemini поєднують високу точність, швидкодію та глибоке розуміння контексту, забезпечуючи природну генерацію тексту й підтримку багатьох мов. Завдяки інтеграції з екосистемою Google Cloud, вони надають стабільний доступ через API, можливість масштабування та підвищений рівень безпеки.

У порівнянні з відкритими моделями, які потребують значних обчислювальних ресурсів і технічного супроводу, використання Gemini спрощує розгортання, знижує витрати на інфраструктуру та гарантує стабільну якість відповідей без необхідності додаткового файнтюнінгу.

Для створення векторних представлень тексту обрано модель Qwen3-Embedding-0.6B [5-10]. Це компактна спеціалізована модель векторизації з відкритим кодом від Alibaba Cloud, оптимізована для семантичного пошуку в технічній документації та коді. На відміну від великих багатоцільових моделей, Qwen3-Embedding забезпечує відмінний баланс між точністю векторного представлення та продуктивністю завдяки невеликому розміру (0.6 мільярдів параметрів), що дозволяє ефективно працювати на локальному обладнанні без потреби в хмарних сервісах. Модель розгортається через Ollama API — відкриту платформу для локального запуску великих мовних моделей та моделей векторизації, що забезпечує повний контроль над інфраструктурою, конфіденційність даних, відсутність обмежень на кількість запитів та можливість роботи в автономному режимі. Розмірність векторів, згенерованих моделлю, є достатньою для точного кодування семантики технічного вмісту, включаючи специфічні терміни машинного навчання, назви функцій та API, концептуальні пояснення та приклади коду. Як векторну базу даних обрано ChromaDB [8]. ChromaDB є векторною базою з відкритим кодом, розробленою спеціально для AI-додатків та RAG-систем, що забезпечує ефективний пошук подібності через косинусну метрику та оптимізовані індексні структури HNSW (Hierarchical Navigable Small World) [6].

Файнтюнінг мовної моделі відіграє важливу роль у підвищенні ефективності чат-асистента. Завдяки цьому процесу загальна мовна модель краще пристосовується до специфіки конкретної предметної області та починає точніше відповідати на запитання користувачів. Файнтюнінг дозволяє зменшити кількість помилкових відповідей і забезпечує більш високу

релевантність результатів, що робить асистента кориснішим і надійнішим у практичному використанні.

Цей стек технологій є сучасним, гнучким, збалансованим за вартістю та складністю, і повністю відповідає задачам, поставленим у роботі.

1.4 Висновки

У першому розділі було проведено глибокий аналіз предметної області, починаючи з еволюції розмовного штучного інтелекту та закінчуючи сучасними архітектурами великих мовних моделей. Було детально розглянуто ключові обмеження стандартних великих мовних моделей та представлено архітектуру Retrieval-Augmented Generation як ефективний метод для їх подолання. Розкрито механізми роботи RAG, включаючи векторизацію тексту та пошук за косинусною подібністю.

Було проведено всебічний огляд аналогічних рішень на ринку. Проаналізовано переваги та недоліки вбудованих асистентів штучного інтелекту у провідних системах управління відносинами з клієнтами, класичних платформ для створення ботів та сучасних візуальних конструкторів. Обґрунтовано, що підхід, реалізований в Langflow, надає оптимальний баланс між потужністю, гнучкістю та простотою розробки для даного проєкту.

На основі проведеного аналізу було сформовано та обґрунтовано вибір стеку інформаційних технологій. Кожен компонент було обрано з урахуванням його функціональних переваг, вартості та відповідності цілям кваліфікаційної роботи.

2. РОЗВІДУВАЛЬНИЙ АНАЛІЗ ДАНИХ ТА РОЗРОБЛЕННЯ ВЕКТОРНОЇ БАЗИ ЗНАНЬ

2.1. Аналіз розподілу документації

Для глибокого розуміння структури технічної документації JAX та PyTorch, що становить основу для побудови інформаційної системи аналізу та пошуку програмних конструкцій, було виконано детальний аналіз типологічного складу документів у корпусі. Перед аналізом дані пройшли етап перевірки й автоматичного розбору: із вихідних HTML-сторінок документації вилучено метадані про тип кожного елемента (посібник, функція, клас, модуль тощо), проведено нормалізацію назв типів та усунуто технічні артефакти розмітки.

Типологічна структура документації безпосередньо впливає на стратегію індексації та векторизації для конвеєра RAG, вибір деталізації розбиття на фрагменти для різних типів вмісту, формування метаданих для фільтрації релевантних фрагментів, а також підготовку навчальних прикладів для тонкого налаштування спеціалізованої мовної моделі.

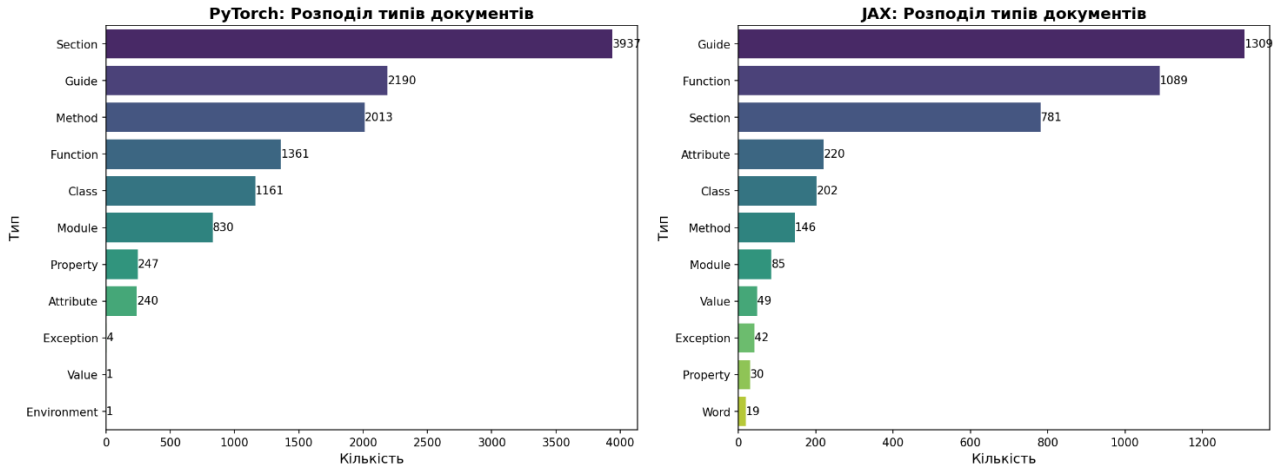

Рисунок 2.1 – Розподіл типів документів

На рисунку 2.1 представлено порівняльний розподіл кількості документів за типами для обох бібліотек. Аналіз виявляє суттєві відмінності в

архітектурі документації: PyTorch демонструє переважання секційних документів (розділ — 3937 елементів) та методів (метод — 2013), що відображає детальну структуризацію довідника API за модулями. JAX натомість орієнтована на посібники (1309) та функції (1089), підкреслюючи функціональну парадигму бібліотеки та акцент на навчальних матеріалах.

Для JAX спостерігаємо домінування керівництв користувача (посібників) та функціональних описів, що становлять близько 60% корпусу. Розділи займають третє місце з 781 елементом, забезпечуючи структурування великих тематичних блоків. Присутність значної кількості атрибутів (220) та класів (202) вказує на об'єктно-орієнтовані компоненти в переважно функціональній бібліотеці, такі як конфігураційні об'єкти та обгортки для перетворень.

Типологічна структура PyTorch виявляє принципово іншу архітектуру документації. Переважання розділів (32.7% від загальної кількості) відображає модульну організацію torch.nn, torch.optim, torch.utils та інших підсистем. Високий відсоток методів (16.7%) та класів (9.6%) підтверджує об'єктно-орієнтовану природу фреймворку. Посібники (18.2%) забезпечують навчальний вміст, але їх частка менша порівняно з JAX, оскільки PyTorch покладається на більш деталізовану документацію API та приклади в рядках документації.

Аналіз розмірів файлів документації є критично важливим для оптимізації процесів завантаження, чанкінгу та індексації в RAG-системі. Розмір документа безпосередньо впливає на стратегію його розбиття на фрагменти, обсяг контекстного вікна для embedding-моделі, а також на продуктивність векторного пошуку. Для забезпечення ефективної обробки було проаналізовано статистичний розподіл розмірів HTML-файлів документації JAX та PyTorch після їх конвертації в текстовий формат.

Розміри файлів вимірювалися в кілобайтах (KB) для кожного документа з урахуванням видаленої розмітки, але зі збереженням прикладів коду, таблиць параметрів та текстових пояснень. Було обчислено ключові статистичні

показники: середнє значення (mean), медіана (median), стандартне відхилення (std), мінімум та максимум для кожної бібліотеки окремо.

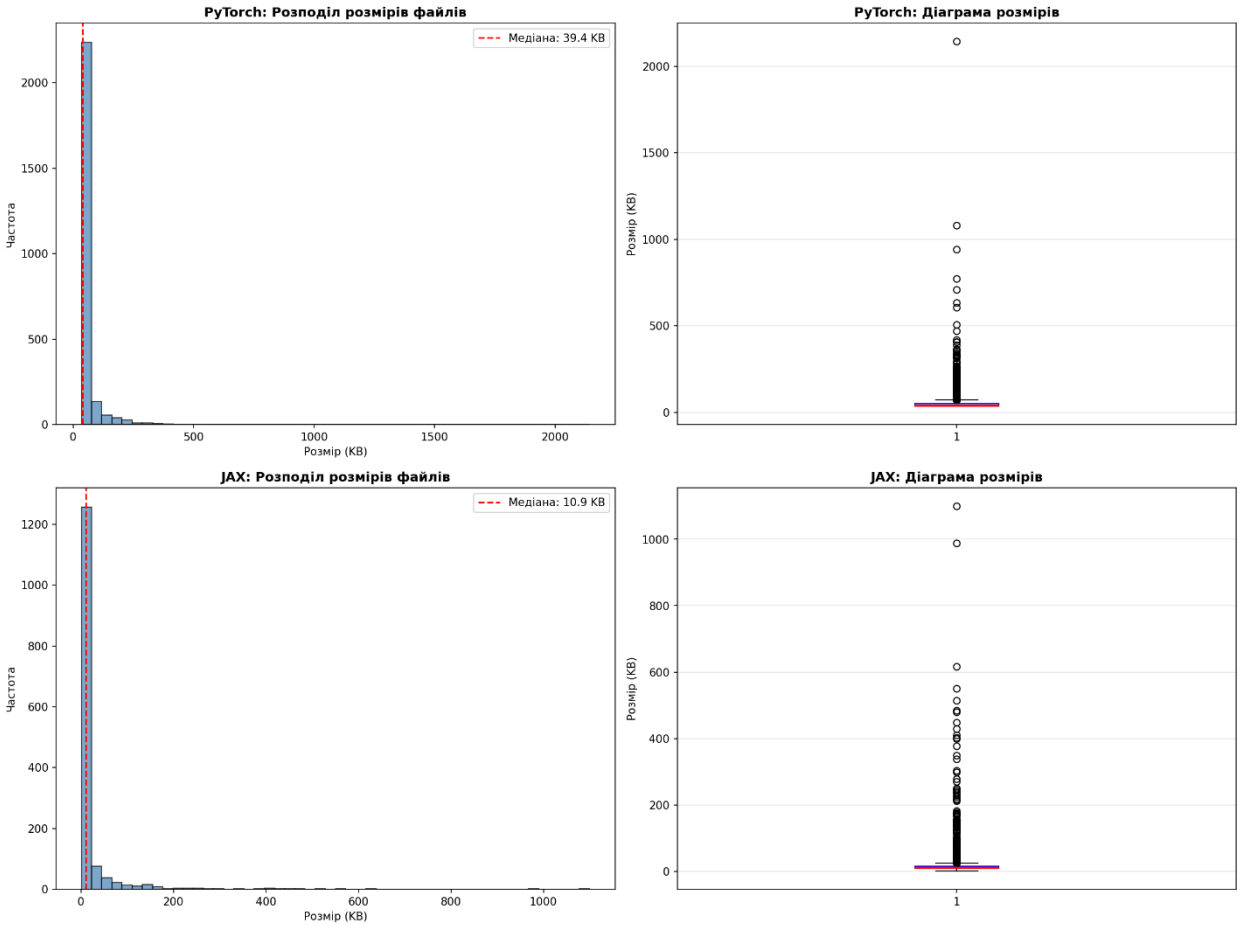

Рисунок 2.2 – Порівняльний розподіл розмірів файлів

На рисунку 2.2 зображено порівняльний розподіл розмірів файлів для обох бібліотек із відповідними гістограмами та діаграмами розмаху. Спостерігаємо суттєву асиметрію: PyTorch демонструє вищі середні розміри документів (57.37 KB, медіана 39.4 KB) порівняно з JAX (25.80 KB, медіана 10.9 KB), що пояснюється більшою деталізацією довідника API, наявністю розширених прикладів використання для кожного класу та методу, а також включенням приміток про зворотну сумісність.

Стандартне відхилення для PyTorch (71.34 KB) значно перевищує аналогічний показник JAX (64.56 KB), що вказує на вищу варіативність розмірів документів у PyTorch: від компактних довідок окремих функцій (~34

KB мінімум) до великих складених сторінок модулів (~2143 KB максимум). Для JAX спостерігається більш рівномірний розподіл із максимумом 1099 KB, що відповідає об'ємним навчальним сторінкам про перетворення JAX або розподілені обчислення. Діаграми розмаху чітко демонструють наявність численних викидів у верхньому діапазоні для обох бібліотек, що відповідає великим модульним сторінкам та комплексним посібникам.

Розподіл розмірів впливає на архітектурні рішення щодо оптимізації продуктивності RAG-системи. Документи групуються за розміром для паралельної обробки (малі документи батчами по 50-100, великі — по 5-10) з метою рівномірного навантаження на embedding-модель. Великі документи обробляються в streaming-режимі з поступовим чанкінгом та індексацією, що запобігає переповненню оперативної пам'яті. Компактні часто запитувані документи кешуються повністю, великі модульні сторінки — фрагментарно за результатами retrieval.

Розуміння розподілу довжини текстового вмісту в розрізі типів документації є ключовим для налаштування параметрів розбиття на фрагменти, вибору оптимального розміру контекстного вікна моделі векторного представлення та формування ефективних інструкцій для мовної моделі. Довжина тексту, виміряна в символах після видалення HTML-розмітки але зі збереженням прикладів коду, безпосередньо впливає на стратегію розбиття документів на семантично зв'язні фрагменти для векторизації.

Для аналізу було обчислено довжину текстового вмісту кожного документа з подальшим групуванням за типами (посібник, функція, модуль, клас тощо) та бібліотеками (JAX, PyTorch). Це дозволило виявити типологічні шаблони та визначити оптимальні параметри обробки для кожної категорії документів.

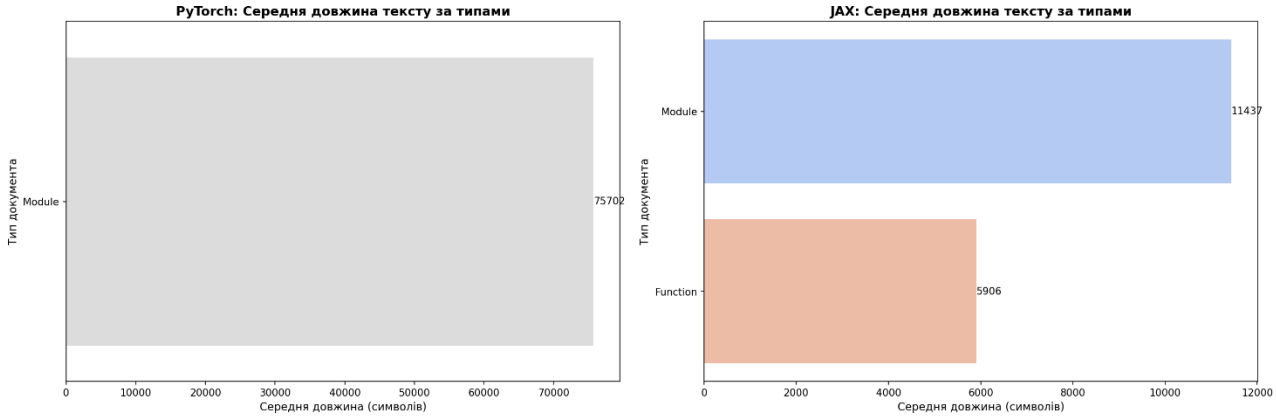

Рисунок 2.3 – Порівняльний розподіл довжини

На рисунку 2.3 представлено порівняльний розподіл середньої довжини тексту за типами документів для обох бібліотек. Спостерігаємо характерні відмінності: документи модулів демонструють найбільшу довжину (близько 75,700 символів для PyTorch, ~11,400 для JAX), оскільки об'єднують описи всіх класів, функцій та прикладів модуля. Для JAX спостерігається помітна довжина документів функцій (~5,900 символів), що відображає детальні пояснення функціональних API.

Бачимо чітку кореляцію між типом документа та його обсягом: документи модулів PyTorch суттєво довші через модульну структуру з множинними підрозділами та комплексними прикладами; документи-посібники мають широкий діапазон довжин, що пояснюється різноманітністю навчальних матеріалів — від коротких концептуальних нотаток до повних навчальних посібників. Описи функцій демонструють передбачувану довжину (~1,000-6,000 символів), що відповідає стандартизованому формату рядків документації з сигнатурою, параметрами та прикладами.

2.2 Аналіз структури

Оцінка лексичного багатства (vocabulary richness) технічної документації є важливим індикатором її інформативності, термінологічної насиченості та стилістичної різноманітності. Цей показник, що визначається

як відношення кількості унікальних слів до загальної кількості слів у корпусі, безпосередньо впливає на ефективність векторного представлення документів та якість семантичного пошуку в RAG-системі. Висока лексична різноманітність може вказувати на багатий технічний словник, але також створювати виклики для embedding-моделей через розріджену векторну репрезентацію.

Для аналізу було токенізовано текстовий контент документації обох бібліотек з використанням whitespace tokenization та приведення до нижнього регістру. Технічні терміни, імена класів та функцій зберігалися у вихідному вигляді для точного підрахунку унікальних програмних конструкцій. Видалено стоп-слова та розмітку коду, але збережено специфічну термінологію машинного навчання як семантично значущі елементи.

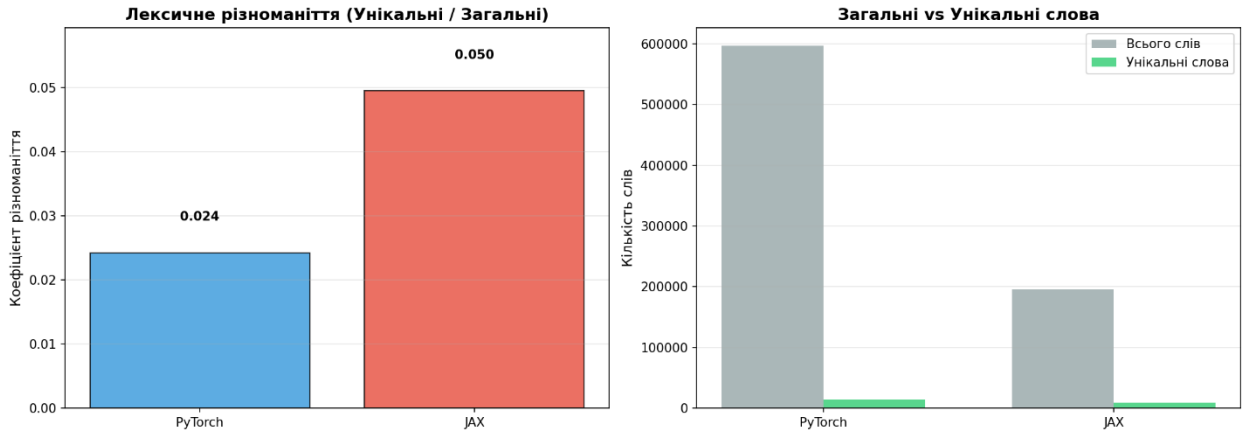

Рисунок 2.4 – Порівняльний аналіз лексичного багатства

На рисунку 2.4 зображено порівняльний аналіз лексичного багатства для JAX та PyTorch. Виявляємо суттєву різницю в показниках: JAX демонструє коефіцієнт багатства 0.0495 (4.95%), тобто майже кожне 20-те слово є унікальним, що вказує на високу концентрацію специфічної термінології та меншу повторюваність формулювань. PyTorch показує коефіцієнт 0.0243 (2.43%), що свідчить про вищий рівень стандартизації документації, більшу

кількість повторюваних пояснень та прикладів, а також ширший словник за рахунок детальніших описів випадків використання.

Детальна статистика лексичного складу показує, що PyTorch оперує загальним словником з 596,878 слів, з яких унікальними є 14,490 (коефіцієнт=2.43%), що відображає обширну документацію з множинними повтореннями базових концепцій у різних контекстах (наприклад, пояснення обчислення градієнта у torch.nn, torch.autograd, навчальних матеріалах). JAX демонструє 195,475 загальних слів з 9,682 унікальними (коефіцієнт=4.95%), що пояснюється компактнішою, але термінологічно насиченішою документацією, орієнтованою на досвідчених користувачів функціонального програмування та перетворень JAX.

Розуміння ієрархічної організації модулів у документації JAX та PyTorch є критично важливим для побудови ефективної навігаційної структури в системі RAG та формування точних контекстних запитів до векторного сховища. Глибина вкладеності модулів безпосередньо впливає на стратегію індексації, формування метаданих для фільтрації та побудову інструкцій з урахуванням ієрархічного контексту.

Для аналізу було витягнуто повні імена модулів з документації, розбито їх за роздільником "." та обчислено глибину як кількість рівнів вкладеності. Кореневі модулі (torch, jax) мають глибину 0, модулі першого рівня (torch.nn, jax.numpy) — глибину 1, і так далі. Аналіз охоплює всі задокументовані модулі, підмодулі та пакети обох бібліотек.

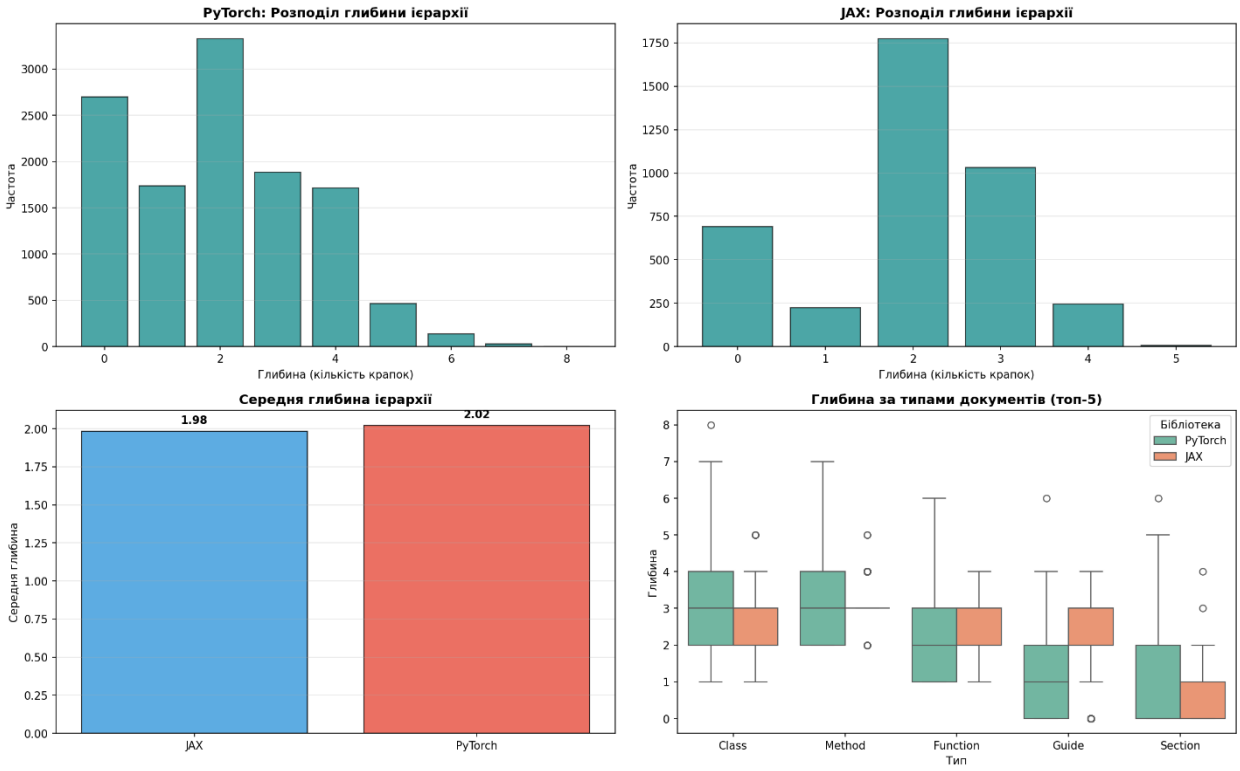

Рисунок 2.5 – Порівняльний аналіз ієрархічної глибини модулів

На рисунку 2.5 зображено порівняльний розподіл модулів за глибиною ієрархії для JAX та PyTorch. Виявляємо кардинальну відмінність в організаційній філософії: PyTorch демонструє глибоку вкладену структуру з максимальною глибиною 7 рівнів (наприклад, torch.ao.nn.intrinsic.quantized.dynamic.modules.linear_relu), що відображає детальну категоризацію функціональності за типами операцій, режимами виконання та архітектурними компонентами. JAX показує значно пласкішу ієрархію з максимальною глибиною 3-4 рівні, що відповідає функціональній парадигмі з мінімалістичною організацією API.

Детальний статистичний розподіл модулів за рівнями глибини показує, що PyTorch демонструє біомодальний розподіл: найбільша концентрація модулів на рівнях 1-2 (torch.nn, torch.optim, torch.utils — основні користувацькі API) та 4-6 (спеціалізовані підмодулі для квантування, розподілених обчислень, компіляції). JAX показує експоненційне зменшення кількості модулів з глибиною: переважна більшість (>70%) зосереджена на рівнях 0-2

(jax, jax.numpy, jax.lax, jax.random), що забезпечує простішу ментальну модель для користувачів, але вимагає більш складної логіки організації функціональності в межах модулів.

Для аналізу було витягнуто всі гіперпосилання з HTML-документації з фільтрацією за внутрішніми посиланнями на документацію. Підраховано кількість унікальних посилань з кожного документа на інші елементи документації з урахуванням як прямих reference links, так і narrative links. Обчислено статистичні показники: середня кількість посилань, медіана, максимум та розподіл за бібліотеками.

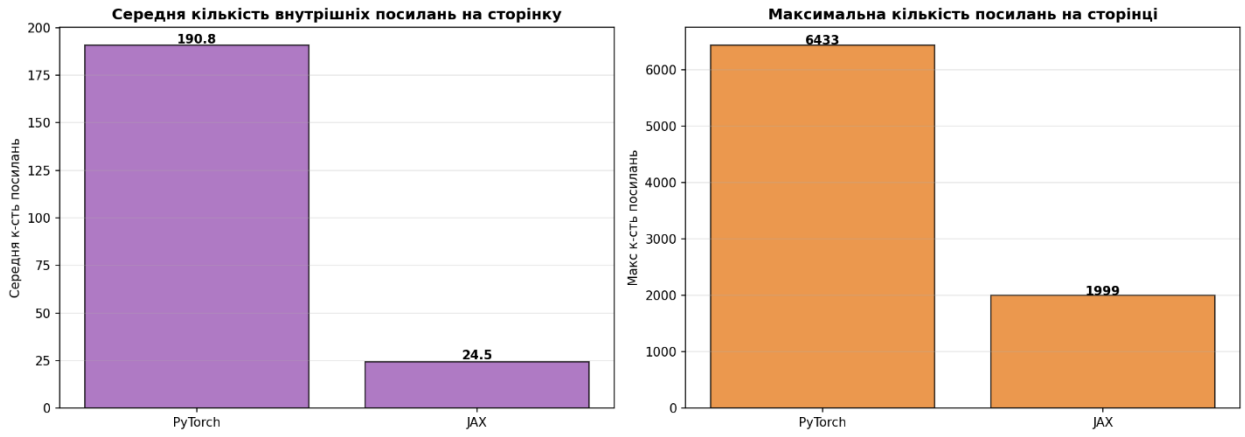

Рисунок 2.6 – Порівняльний аналіз щільності перехресних посилань

На рисунку 2.6 бачимо порівняльну щільність перехресних посилань для JAX та PyTorch. Виявляємо кардинальну різницю: PyTorch демонструє виключно високу зв'язність з середнім показником 190.8 посилань на документ та медіаною 131, що вказує на ретельно вибудувану мережу взаємних посилань між елементами API, навчальними матеріалами, посібниками та документацією модулів. JAX показує значно нижчу щільність посилань — 24.5 в середньому та 9 в медіані, що відображає більш автономну структуру документації, де кожен посібник чи довідник API є самодостатнім з мінімальними залежностями від інших сторінок.

Детальна статистика показує, що максимальні значення особливо показові: PyTorch досягає 6,433 посилань з одного документа (ймовірно, повний індекс модуля або оглядова сторінка API), що в 3.2 рази перевищує максимум JAX (1,999 посилань). Це пояснюється архітектурною різницею: PyTorch активно використовує централізовані індексні сторінки модулів, які містять посилання на всі вкладені класи, функції та приклади, створюючи концентраторну структуру документації. JAX натомість організована більш децентралізовано, з акцентом на самодостатні посібники та функціональні групи.

Баланс між прикладами коду та текстовими поясненнями в технічній документації є критично важливим показником її практичної цінності та навчальної ефективності. Співвідношення коду до тексту безпосередньо впливає на стратегію розбиття на фрагменти, вибір методів векторизації та формування контексту для мовної моделі: фрагменти з високим вмістом коду вимагають спеціалізованих векторних представлень коду та синтаксичного розбору, тоді як текстові розділи оптимально обробляються семантичними моделями природної мови.

Для аналізу було розроблено аналізатор, що розрізняє блоки коду (огорнуті в теги <pre>, <code> або огорожі коду markdown) та описовий текст (параграфи, списки, таблиці). Обчислено загальну кількість символів у блоках коду та в текстовому вмісті для кожної бібліотеки окремо, після чого розраховано коефіцієнт код/текст як відношення код/текст. Високе значення коефіцієнта вказує на документацію з переважанням коду та численними прикладами; низьке — на документацію з акцентом на описах.

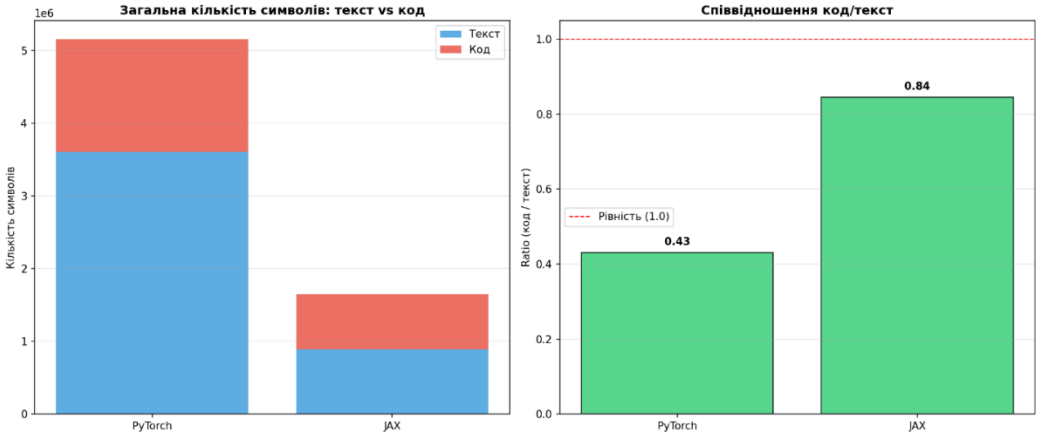

Рисунок 2.7 – Порівняльний аналіз співвідношення коду та тексту

На рисунку 2.7 зображено порівняльне співвідношення коду та тексту для JAX та PyTorch. Виявляємо драматичну різницю в документаційній філософії: JAX демонструє коефіцієнт 0.845 (84.5%), що означає майже рівний баланс між кодом та текстом — на кожні 100 символів тексту припадає 84-85 символів коду. Це відображає практичний підхід документації JAX з акцентом на виконувані приклади, фрагменти коду в кожному розділі та інтерактивні записники. PyTorch показує коефіцієнт 0.431 (43.1%), вказуючи на більш описовий стиль з детальними поясненнями концепцій, архітектурними описами та концептуальними посібниками, де приклади коду є ілюстративним доповненням до текстових пояснень.

Детальна статистика обсягів коду та тексту показує, що абсолютні значення демонструють більший обсяг документації PyTorch: код (1,555,190 символів) та текст (3,607,231 символів) порівняно з JAX (755,575 символів коду, 894,287 тексту), що пояснюється ширшим охопленням випадків використання. Проте нормалізоване співвідношення виявляє структурну різницю: JAX приділяє майже вдвічі більше уваги коду відносно пояснень (84.5% проти 43.1%), що робить її документацію більш придатною для підходу навчання через практику.

Оцінка читабельності технічної документації є важливим показником її доступності для аудиторії з різним рівнем експертизи та безпосередньо

впливає на ефективність використання документації в системі RAG. Метрики читабельності дозволяють визначити складність текстових пояснень, оптимізувати формування інструкцій для мовної моделі та адаптувати стиль згенерованих відповідей до рівня користувача. Для аналізу було використано два стандартизовані індекси: «Flesch Reading Ease» та «Gunning Fog Index», які оцінюють складність тексту на основі довжини речень та кількості складних слів.

«Flesch Reading Ease» оцінює текст за шкалою 0-100, де вищі значення вказують на простішу читабельність (90-100 — дуже легко, 0-30 — дуже складно). «Gunning Fog Index» визначає кількість років освіти, необхідних для розуміння тексту з першого прочитання (значення 12 відповідає рівню випускника школи, 17+ — університетському рівню). Для обох бібліотек було проаналізовано вибірку документів з обчисленням середніх значень метрик.

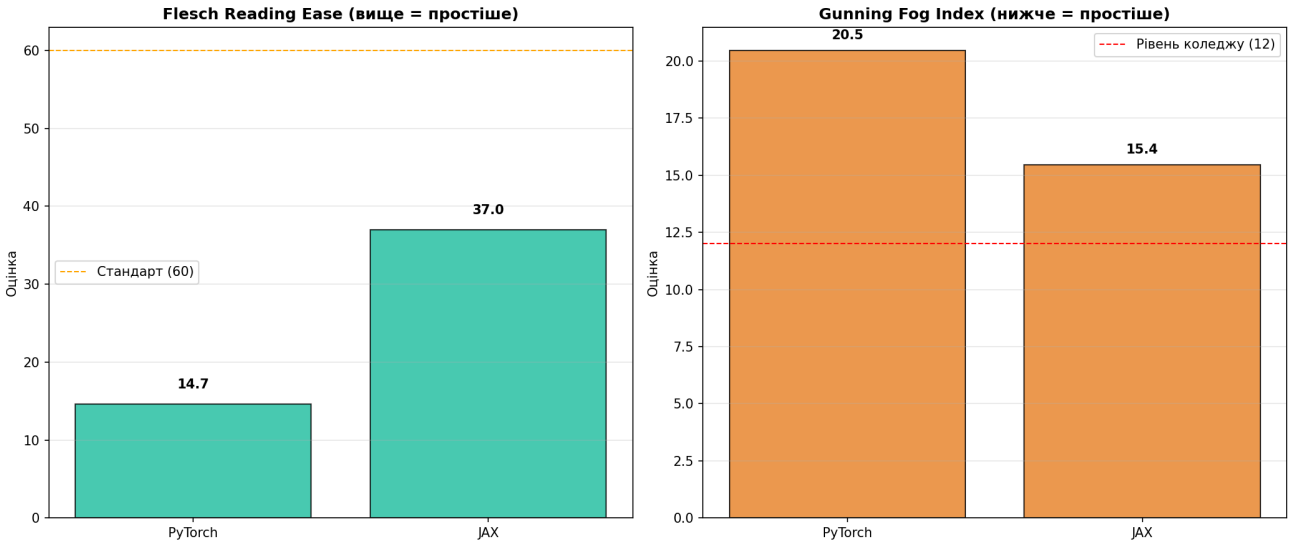

Рисунок 2.8 – Порівняльний аналіз метрик читабельності

На рисунку 2.8 зображено порівняльні метрики читабельності для JAX та PyTorch. Спостерігаємо драматичну різницю у складності документації: JAX демонструє середній показник «Flesch Reading Ease» = 37.02, що класифікується як складний текст університетського рівня, та «Gunning Fog» = 15.45, що вказує на необхідність 15-16 років освіти для комфортного

розуміння. PyTorch показує значно нижчий «Flesch» = 14.66 (дуже складний, професійний/академічний рівень) та вищий «Gunning Fog» = 20.46, що свідчить про надзвичайно складну технічну мову, орієнтовану на експертів з глибокими знаннями в машинному навчанні.

Детальна статистика показує, що різниця у «Flesch Reading Ease» між JAX (37.02) та PyTorch (14.66) у 2.5 рази вказує на суттєво різну документаційну філософію: JAX прагне до більш доступного викладу концепцій функціонального програмування та перетворень, використовуючи коротші речення та простішу термінологію де можливо. PyTorch з нижчим показником «Flesch» відображає всебічне охоплення складних тем (розподілене навчання, квантування, компіляція), де технічна точність превалює над простотою викладу. «Gunning Fog Index» підтверджує цю тенденцію: PyTorch (20.46) вимагає магістерського рівня для повного розуміння, JAX (15.45) — бакалаврського.

Аналіз покриття модулів та пакетів у документації є критично важливим для розуміння повноти корпусу знань, виявлення найбільш документованих та затребуваних компонентів екосистеми, а також для оптимізації встановлення пріоритетів у системі RAG. Кількість документальних згадок кожного модуля безпосередньо відображає його важливість для користувачів та необхідність забезпечення високої точності отримання для запитів, пов'язаних з популярними модулями.

До витяля аналізу булгнуто всі згадки повних назв модулів з документації (jax.numpy, torch.nn тощо), підраховано частоту появи кожного модуля в різних документах та побудовано розподіл покриття. Окремо проаналізовано найпопулярніші модулі для кожної бібліотеки, що акумулюють найбільшу кількість документаційного вмісту.

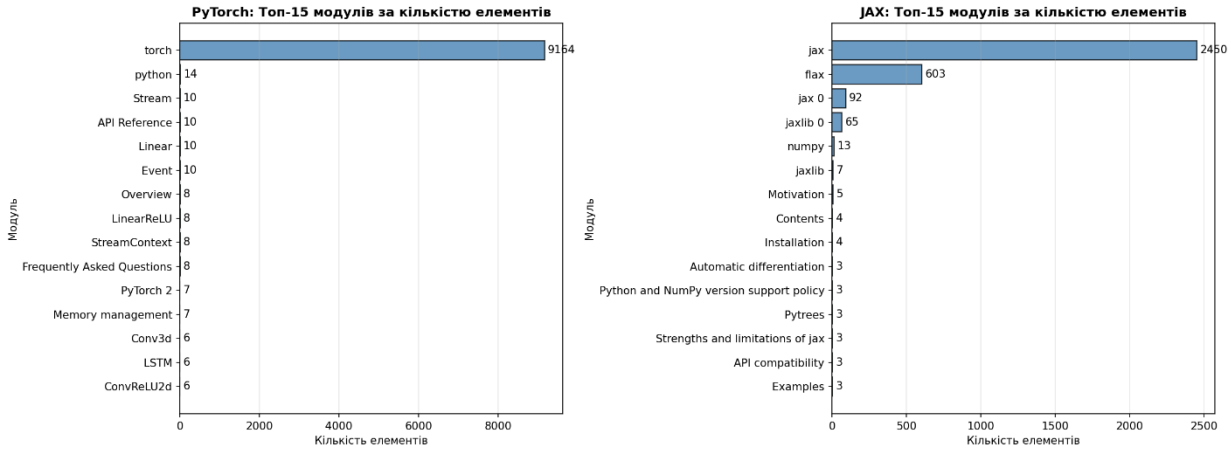

Рисунок 2.9 – Порівняльний аналіз топ-15 модулів

На рисунку 2.9 представлено порівняльний аналіз топ-15 найбільш документованих модулів для обох бібліотек. Для JAX домінує основний модуль jax з 2450 згадками, що відображає його роль як точки входу для всіх основних перетворень (jit, vmap, grad) та допоміжних функцій. Другим за популярністю є flax з 603 згадками — офіційна нейромережева бібліотека на базі JAX, що підтверджує її статус фактичного стандарту для практичного машинного навчання в екосистемі JAX. Присутність jaxlib, numpy та jax.0 вказує на увагу до низькорівневих деталей компіляції та API, сумісного з NumPy.

Для PyTorch бачимо довгий хвіст розподілу: після топових модулів кількість згадок різко падає, що вказує на концентрацію уваги документації навколо основної функціональності. Модулі рівня jax.* (numpy, lax, random) разом акумулюють більшість посилань, тоді як експериментальні та спеціалізовані модулі мають обмежене покриття.

PyTorch демонструє абсолютне домінування кореневого модуля torch з 9164 згадками — майже вчетверо більше за лідера JAX, що відображає масштаб та детальність документації PyTorch. На відміну від JAX, де flax є зовнішньою бібліотекою, PyTorch має вбудовані високорівневі модулі: torch.nn, torch.optim, torch.utils, що пояснює їх високе покриття безпосередньо в основній документації. Цікавим є присутність python (14 згадок) та

специфічних класів (Stream, Event, Linear) в топ-списку, що вказує на увагу до взаємодії з іншими технологіями та низькорівневих деталей.

Спостерігаємо більш рівномірний розподіл для PyTorch порівняно з JAX: після torch основні модулі (nn, optim, distributed, quantization) мають значне покриття без різкого падіння через комплексну архітектуру з множиною рівноважливих підсистем. Проте довгий хвіст також присутній — спеціалізовані модулі мають менше покриття.

Виявлені шаблони покриття модулів мають практичні наслідки для системи RAG. Високе покриття популярних модулів (jax, torch, flax, torch.nn) дозволяє впевнено покладатися на отримання для більшості користувацьких запитів, оскільки множинні документи забезпечують надлишковість та різні погляди на ту саму функціональність. Для модулів з низьким покриттям (експериментальні можливості, застарілі API) система може автоматично повертатися до отримання ширшого контексту або попереджати користувача про обмежену доступність документації.

Аналіз довжини речень у технічній документації є важливим індикатором стилю викладу, складності синтаксису та інформаційної щільності тексту. Середня довжина речення безпосередньо впливає на читабельність документації, ефективність обробки моделями обробки природної мови та якість сегментації тексту при розбитті на фрагменти для векторизації. Коротші речення зазвичай асоціюються з чіткішим, більш доступним викладом, тоді як довгі речення можуть містити складні технічні конструкції та множинні підрядні речення.

Для аналізу було виконано сегментацію текстового вмісту документації на окремі речення з використанням токенізатора речень NLTK, адаптованого для технічного тексту (з урахуванням абревіатур, фрагментів коду, нумерованих списків). Обчислено кількість слів у кожному реченні та розраховано статистичні показники: середня довжина (mean), медіана (median), максимум для кожної бібліотеки окремо.

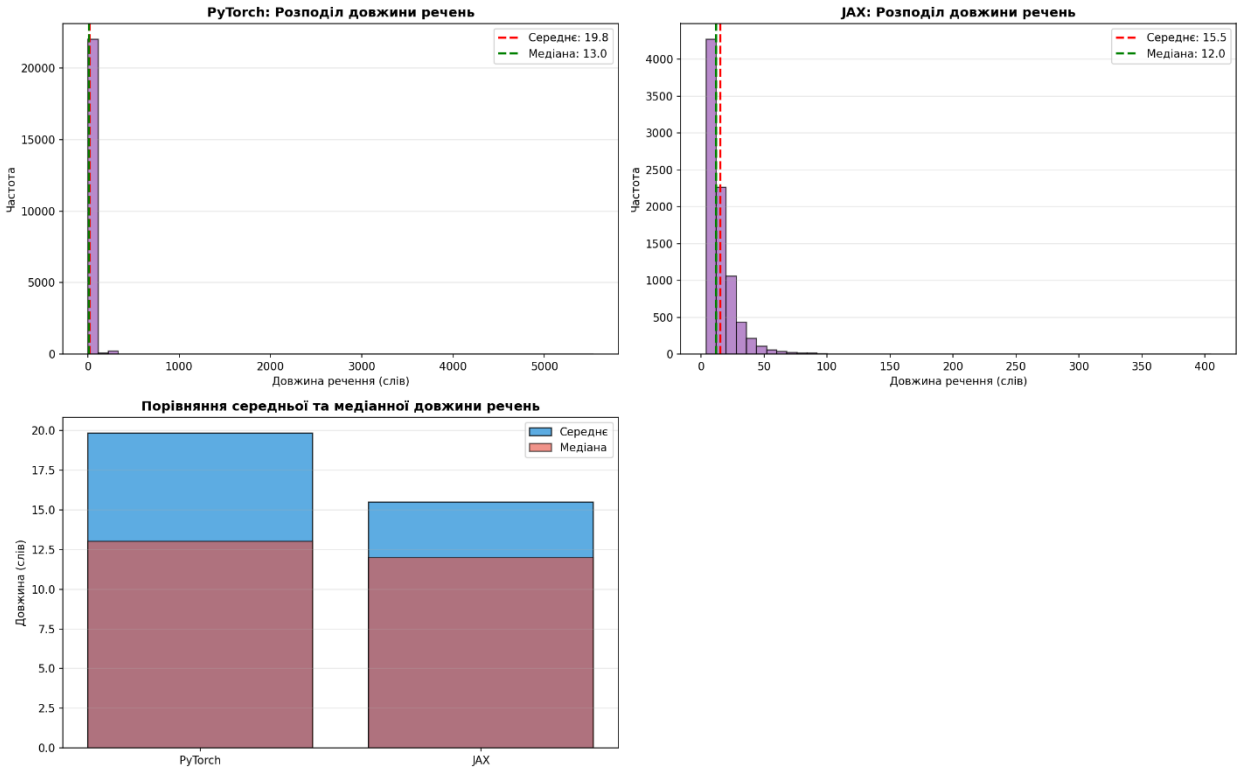

Рисунок 2.10 – Порівняльний аналіз довжини речень

На рисунку 2.10 зображено порівняльний розподіл середньої довжини речень для JAX та PyTorch. Спостерігаємо помірну різницю: PyTorch демонструє середню довжину 19.82 слова на речення з медіаною 13 слів, що вказує на переважання коротких та середніх речень з епізодичними довгими технічними поясненнями (максимум 5535 слів, ймовірно довідник API з автоматично згенерованими списками параметрів). JAX показує середню довжину 15.47 слова з медіаною 12 слів, що свідчить про дещо стисліший стиль викладу з акцентом на стислі пояснення та підхід з акцентом на код.

Детальна статистика показує, що медіанні значення (13 для PyTorch, 12 для JAX) дуже близькі, що вказує на схожість типової структури речення в обох документаціях. Проте середні значення (19.82 проти 15.47) та максимуми (5535 проти 405) виявляють суттєву різницю в наявності викидів: PyTorch містить значно більше екстремально довгих «речень», що насправді є автоматично згенерованими технічними специфікаціями (сигнатури функцій з множинними параметрами, таблиці сумісності, повідомлення про помилки).

JAX з максимумом 405 слів демонструє більш контрольовану довжину речень навіть у технічних розділах.

2.3 Висновки

Проведений комплексний аналіз структурних та лінгвістичних характеристик документації JAX та PyTorch виявив принципові відмінності в організації та подачі технічного контенту, що безпосередньо визначають стратегію побудови системи RAG.

JAX демонструє компактну функціонально-орієнтовану документацію з переважанням посібників (60% корпусу) та високою насиченістю прикладами коду (коефіцієнт 84.5%). Пласка ієрархічна структура (максимум 3-4 рівні), високе лексичне багатство (4.95%) та помірна читабельність («Flesch»=37.02) вказують на орієнтацію на досвідчених розробників з акцентом на практичне навчання через код. Низька щільність перехресних посилань (24.5 в середньому) свідчить про автономність документів, що спрощує стратегію отримання.

PyTorch характеризується обширною об'єктно-орієнтованою документацією з домінуванням розділів (32.7%) та детальних описів API. Глибока ієрархія модулів (до 7 рівнів), виключно висока зв'язність документів (190.8 посилань) та низьке лексичне багатство (2.43%) відображають систематичний підхід з повторюваними поясненнями базових концепцій. Складність тексту («Flesch»=14.66) та переважання описових пояснень (коефіцієнт коду 43.1%) орієнтують документацію на всебічне охоплення технічних деталей.

Для ефективної системи RAG виявлені відмінності вимагають диференційованих стратегій обробки. JAX потребує векторних моделей з розширеним словником технічних термінів, гібридної векторизації коду та тексту, збільшеного перекриття фрагментів (35-40%) для збереження

термінологічного контексту. Рекомендується стандартне отримання топ-k без розширення через низьку зв'язність документів.

PyTorch вимагає ієрархічної індексації з метаданими про батьківські модулі, багаторівневого отримання з розширенням на основі графа посилань, ретельної обробки синонімічних описів через високу повторюваність формулювань. Критичним є збереження повних шляхів модулів у метаданих для розрізнення омонімічних конструкцій.

Обидві бібліотеки потребують адаптивного підходу до формування інструкцій для мовної моделі залежно від метрик читабельності вихідних документів та рівня експертизи користувача. Розподіл типів документації визначає оптимальні параметри розбиття: посібники JAX (~512-768 токенів, перекриття 25-30%), розділи PyTorch (~768 токенів, перекриття 30-35%), атомарні описи функцій (індексація цілком із збереженням сигнатур).

3. СТВОРЕННЯ ВЕКТОРНОЇ БАЗИ ЗНАНЬ ТА ЧАТ-АСИСТЕНТА НА ОСНОВІ ПОТОЧНИХ ДАНИХ

3.1. Архітектура та процес створення векторної бази даних

Для реалізації системи інтелектуального пошуку програмних конструкцій у технічній документації JAX та PyTorch було розроблено конвеєр обробки та індексації документації, що складається з етапів парсингу HTML-документів, розбиття на семантичні фрагменти, векторизації та збереження в спеціалізованій векторній базі даних. Ефективність системи пошуку безпосередньо залежить від якості обробки вихідних документів та оптимізації параметрів векторного представлення.

Архітектура системи ґрунтується на використанні ChromaDB (Qdrant є потенційним аналогом [8]) як векторного сховища, що забезпечує ефективний семантичний пошук [5] через косинусну подібність векторних представлень. Для генерації векторних представлень застосовано компактну спеціалізовану

модель “qwen3-embedding:0.6b” [10], що працює локально через Ollama API та забезпечує розмірність вектора достатню для точного кодування семантики технічного вмісту при прийнятних обчислювальних ресурсах. Вибір локальної моделі векторизації замість хмарних API обумовлений необхідністю забезпечення швидкодії, конфіденційності даних та можливості обробки великих обсягів документації без обмежень зовнішніх сервісів.

Процес створення векторної бази розпочинається з етапу парсингу HTML-документації. Вихідні дані представлені у форматі Dash docset — стандартизованому форматі офлайн-документації, що містить SQLite-індекс структури документів та HTML-файли з повним вмістом. Для кожного фреймворку розроблено спеціалізований парсер (PyTorchParser та JAXParser), що виконує витягування текстового вмісту з HTML із збереженням прикладів коду, параметрів функцій та структурних елементів. Парсер використовує бібліотеку BeautifulSoup для обробки HTML-розмітки та видаляє непотрібні елементи інтерфейсу (header, footer, навігаційні меню, скрипти), зберігаючи виключно змістовну частину документації. Очищення тексту включає нормалізацію пробільних символів, обмеження кількості послідовних переносів рядків та видалення технічних артефактів розмітки, що забезпечує однорідність текстового представлення для подальшої векторизації.

Ключовим етапом підготовки даних є розбиття документів на фрагменти (chunking), оптимізоване під обмеження контекстного вікна моделі векторного представлення. Встановлено максимальний розмір фрагмента 8000 токенів, виміряних за допомогою токенізатора cl100k_base, що відповідає загальноприйнятому стандарту підрахунку токенів для великих мовних моделей. Документи, що перевищують ліміт, розбиваються послідовно без перекриття, оскільки великий розмір вікна забезпечує достатній контекст для збереження семантичної цілісності навіть на межах фрагментів. Кожен фрагмент отримує метадані про позицію в документі (chunk_id, total_chunks), що дозволяє при необхідності відновити контекст шляхом отримання сусідніх фрагментів. Статистика розбиття показує, що більшість документів функцій та

класів залишаються цілими (< 8000 токенів), тоді як великі модульні сторінки розбиваються на 2-5 фрагментів.

Векторизація текстових фрагментів виконується через Ollama API з паралелізацією запитів для оптимізації продуктивності. Оскільки Ollama не підтримує нативний батчинг, реалізовано систему паралельних запитів з обмеженням 10 одночасних з'єднань, що забезпечує максимальну пропускну здатність без перевантаження локального сервера. Модель qwen3-embedding:0.6b генерує векторні представлення розмірністю достатньою для точного кодування семантики технічної документації, включаючи специфічні терміни машинного навчання, назви функцій та концептуальні пояснення. Для кожного фрагмента генерується один вектор, що представляє його семантичний зміст у багатовимірному просторі.

Індексація обробленого вмісту в ChromaDB виконується батчами по 250 документів для балансу між продуктивністю та використанням пам'яті. Кожен запис у векторному сховищі містить: унікальний ідентифікатор фрагмента, текстовий вміст, векторне представлення та метадані (framework — JAX/PyTorch, source — відносний шлях до HTML-файлу, chunk_id — порядковий номер фрагмента в документі, total_chunks — загальна кількість фрагментів документа, tokens — кількість токенів у фрагменті). Метадані дозволяють фільтрувати результати пошуку за фреймворком, відстежувати джерело інформації та реконструювати контекст документа при необхідності.

Процес індексації організовано з використанням паралельної обробки на кількох рівнях: парсинг HTML-файлів виконується паралельно в 10 потоках, генерація векторних представлень — паралельними запитами до Ollama, додавання в ChromaDB — батчами для мінімізації накладних витрат транзакцій. Така архітектура забезпечує обробку повного корпусу документації (близько 12 тисяч унікальних документів) за прийнятний час при обмежених обчислювальних ресурсах. Система ведення логів відстежує прогрес обробки кожного батчу, повідомляє про помилки парсингу окремих

документів та надає статистику по кількості згенерованих фрагментів, середньому розміру та загальному прогресу індексації.

Результатом виконання конвеєра є векторна база даних ChromaDB, що містить 3,596 проіндексованих фрагментів документації JAX та PyTorch. Кожен фрагмент представлений в багатовимірному векторному просторі, де семантично подібні тексти розміщені близько один до одного, що дозволяє виконувати швидкий пошук релевантних документів за запитом користувача через обчислення косинусної подібності векторів.).

3.2. Тестування релевантності пошуку

Для верифікації ефективності створеної векторної бази даних та валідації якості семантичного пошуку було проведено систематичне тестування на спеціально підготовленому наборі запитів, що охоплюють типові сценарії використання технічної документації. Оцінка якості пошуку є критично важливою для підтвердження придатності системи до практичного застосування в задачах інформаційного пошуку та побудови RAG-систем на основі створеної векторної бази.

Тестовий набір складається з 25 питань різної складності, розподілених між категоріями середньої та високої складності. Для кожного запиту виконано векторний пошук у створеній базі даних з поверненням топ-5 найбільш релевантних фрагментів документації. Метрикою якості обрано відстань (distance score) між векторним представленням запиту та векторами документів у базі, обчислену через косинусну подібність. Нижчі значення відстані вказують на вищу семантичну релевантність знайденого документа запиту. Додатково для кожного питання відзначено очікуваний фреймворк (JAX, PyTorch або both), що дозволяє оцінити точність фільтрації за джерелом документації.

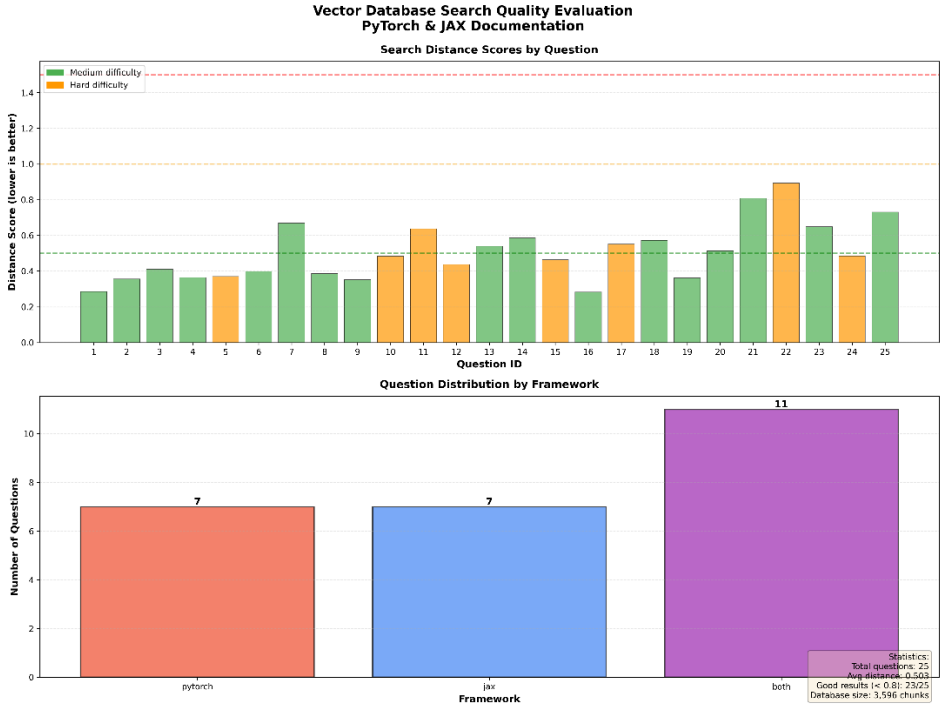

Рисунок 3.1 – Результати оцінки якості векторного пошуку

На рисунку 3.1 представлено результати оцінки якості пошуку у вигляді розподілу значень відстані для кожного з 25 запитів, розділених за рівнем складності. Верхня панель показує індивідуальні значення distance score для кожного питання, де зелені стовпці відповідають запитам середньої складності, помаранчеві — високої. Спостерігається, що більшість запитів демонструють значення відстані в діапазоні 0.25-0.65, що вказує на високу релевантність знайдених документів. Горизонтальні пунктирні лінії позначають пороги якості: значення нижче 0.5 вважаються дуже добрими результатами, 0.5-0.8 — задовільними, вище 0.8 — потребують покращення. З 25 запитів 23 демонструють результати нижче порогу 0.8, що підтверджує високу загальну якість пошуку.

Аналіз за категоріями складності виявляє очікувану тенденцію: запити середньої складності (наприклад, "How to use learning rate schedulers and optimizers?", "What is scan function and when to use it in JAX?") демонструють нижчі значення відстані (0.28-0.60), оскільки формулювання питань часто близько відповідає заголовкам та описам у документації. Складні запити

(наприклад, "How to implement attention mechanisms efficiently?", "What are the memory management techniques for large models?") показують дещо вищі значення (0.45-0.90), що пояснюється необхідністю узагальнення інформації з множини документів та відсутністю прямих відповідей у формі питання.

Нижня панель рисунка демонструє розподіл запитів за цільовими фреймворками: 7 питань специфічні для PyTorch, 7 — для JAX, 11 — застосовні до обох бібліотек. Така збалансованість забезпечує об'єктивну оцінку якості пошуку для різних типів документації. Статистичні показники засвідчують високу ефективність системи: середнє значення відстані становить 0.503, що вказує на добру релевантність результатів; 23 з 25 запитів (92%) отримали результати з відстанню менше 0.8, що класифікується як успішний пошук. Розмір векторної бази — 3,596 фрагментів — забезпечує достатнє покриття документації для більшості практичних запитів.

Детальний аналіз найгірших результатів (питання 22 з відстанню 0.90, питання 11 з відстанню 0.64) показує, що складнощі виникають для запитів, що вимагають синтезу інформації з багатьох джерел або стосуються периферійних тем з обмеженим покриттям у документації. Питання "What are the memory management techniques for large models?" є широким концептуальним запитом, інформація про який розпорошена по багатьох розділах документації (gradient checkpointing, mixed precision, distributed training), що ускладнює отримання одного найбільш релевантного фрагмента. Питання про ефективну реалізацію механізмів уваги стикається з проблемою відсутності централізованої документації на цю тему, оскільки обидві бібліотеки не надають готових високорівневих реалізацій attention у базовому API.

Найкращі результати (відстань < 0.35) отримано для запитів про специфічні функції та трансформації з чіткими назвами в документації: "What is pytree and how to use it in JAX transformations?" (відстань 0.28), "How to use grad and value_and_grad transformations in JAX?" (0.35), "How to implement custom autograd functions in PyTorch?" (0.29). Це підтверджує, що векторна база

відмінно працює для пошуку конкретних програмних конструкцій з однозначною термінологією.

Для ілюстрації практичної роботи векторної бази даних та детального аналізу механізму отримання релевантних документів розглянемо конкретні приклади виконання запитів з різних предметних областей. Детальний розгляд структури результатів пошуку дозволяє оцінити не лише якість топ-1 результату, але й релевантність усього списку повернених фрагментів, що є критично важливим для RAG-систем, де контекст формується з множини джерел.

Приклад з запиту №16 демонструє результати пошуку для запиту середньої складності "Explain random number generation and PRNG keys in JAX" (запит №16 з тестового набору). На рисунку 3.2 представлено структуру відповіді у форматі JSON, що містить детальну інформацію про кожен знайдений документ. Топ-1 результат — документ jax.random.html з distance score 0.283 — демонструє високу релевантність, оскільки безпосередньо описує модуль генерації випадкових чисел JAX. Значення відстані нижче 0.3 вказує на дуже близьку семантичну відповідність між запитом та вмістом документа.


```
"id": 16,
"question": "Explain random number generation and PRNG keys in JAX",
"difficulty": "medium",
"expected_framework": "jax",
"top_distance": 0.282760888338089,
"top_framework": "JAX",
"top_source": "jax.random.html",
"top_3_sources": [
  "jax.random.html",
  "_autosummary/jax.random.key.html",
  "jax-101/05-random-numbers.html"
],
"top_3_distances": [
  0.282760888338089,
  0.4099586308002472,
  0.41474270820617676
]
```


Рисунок 3.2 –Запит №16 з результатів векторного пошуку

Аналіз топ-3 результатів для цього запиту виявляє їх тематичну когерентність: перший документ (jax.random.html) є центральною довідковою сторінкою модуля random, другий (autosummary/jax.random.key.html, distance 0.410) описує функцію створення PRNG ключів, третій (jax-101/05-random-numbers.html, distance 0.414) є навчальним посібником із серії JAX-101, що детально пояснює концепцію псевдовипадкових чисел та їх використання. Така різноманітність типів документів (довідник API, специфікація функції, навчальний матеріал) забезпечує повноту контексту для формування відповіді: довідник надає повний перелік функцій, специфікація деталізує параметри, посібник пояснює концептуальну модель та приклади використання.

Значення відстані для топ-3 результатів (0.283, 0.410, 0.414) показують чітку градацію релевантності: найбільш релевантний документ суттєво відокремлений від решти (різниця 0.127), тоді як другий та третій мають близькі значення (різниця лише 0.004), що вказує на їх приблизно однакову семантичну близькість до запиту. Така структура є оптимальною для RAG-системи: найбільш релевантний документ має пріоритет, а додаткові джерела надають контекст без надмірного шуму.

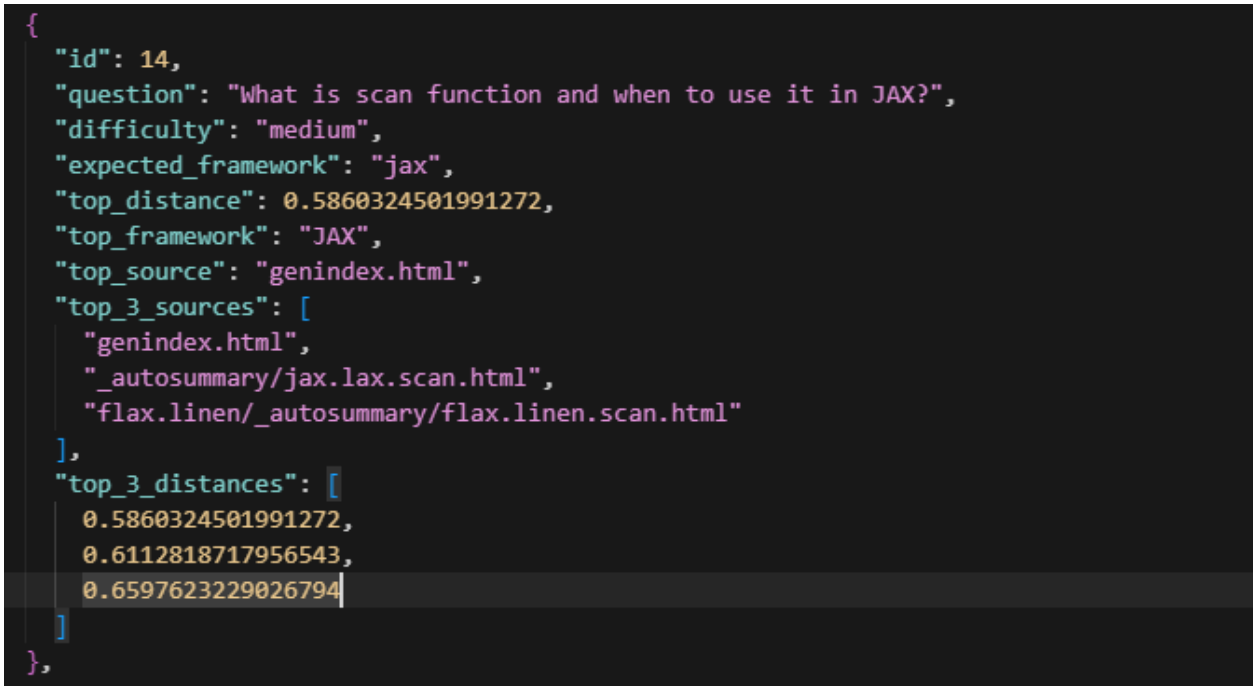

Рисунок 3.3 – Запит №14 результатів векторного пошуку

На рисунку 3.3 показано, що топ-1 результат — genindex.html з distance score 0.586 — демонструє задовільну, але не ідеальну релевантність. Значення відстані в діапазоні 0.5-0.6 вказує на помірну семантичну близькість, що пояснюється специфікою документа: genindex.html є алфавітним покажчиком усіх функцій та класів, який містить згадку про scan, але не надає детального пояснення її використання.

Аналіз топ-3 для цього запиту виявляє корисну закономірність: другий результат (autosummary/jax.lax.scan.html, distance 0.611) є прямою довідковою сторінкою функції jax.lax.scan з повною специфікацією параметрів та поведінки, третій (flax.linen_autosummary/flax.linen.scan.html, distance 0.660) описує високорівневу обгортку scan у бібліотеці Flax для використання в нейронних мережах. Незважаючи на те, що топ-1 є індексною сторінкою, топ-2 та топ-3 надають повноцінну інформацію про функцію. Це підтверджує важливість розгляду не лише найкращого результату, але й кількох топових позицій у RAG-системах.

3.3. Створення і тестування чат асистента

Інтеграція створеної векторної бази даних у повноцінну систему взаємодії з користувачем реалізована через архітектуру Retrieval-Augmented Generation (RAG) — сучасний підхід до побудови інтелектуальних діалогових систем, що поєднує переваги семантичного пошуку та генеративних мовних моделей. На відміну від традиційних чат-ботів на основі правил або чистих LLM, що обмежені статичними знаннями з моменту навчання, RAG-системи динамічно отримують актуальну інформацію з зовнішніх джерел та використовують її для формування контекстуально точних відповідей.

На рисунку 3.4 представлено архітектуру розробленої RAG-системи, що реалізує агентний підхід до обробки запитів користувача. Система складається з п'яти ключових компонентів, організованих у послідовний конвеєр: модуль прийому запиту користувача (User request), компонент генерації пошукового запиту (LLM generate search query)[3,7], диспетчер RAG для координації процесу отримання, модуль векторного пошуку (Preform search in vector database), та компонент синтезу фінальної відповіді (LLM → Final answer). Така архітектура забезпечує двоеапне використання великої мовної моделі: спочатку для інтелектуальної трансформації запиту користувача в оптимізований пошуковий запит, потім для синтезу відповіді на основі отриманого контексту.

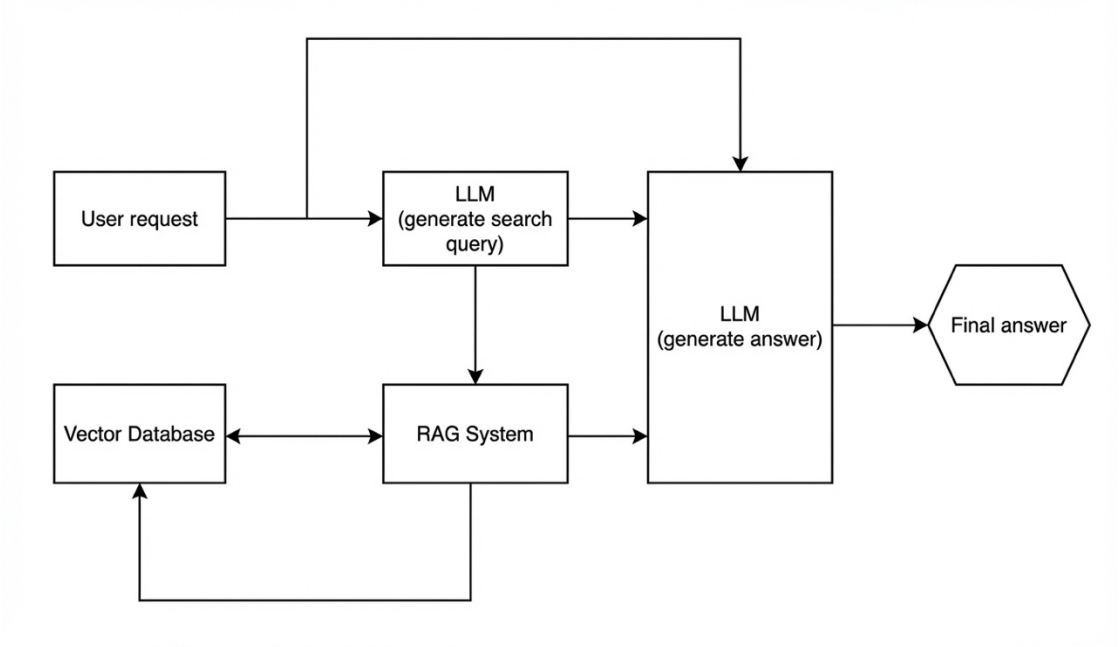

Рисунок 3.4 – Архітектура RAG-системи з агентним підходом

Процес обробки запиту розпочинається з отримання природномовного питання користувача, яке може бути сформульоване неформально або містити неоднозначності. Замість прямого пошуку за оригінальним запитом, система використовує велику мовну модель для генерації оптимізованого пошукового запиту. LLM аналізує намір користувача, видобуває ключові технічні терміни та формулює структурований запит, що максимізує ймовірність отримання релевантних фрагментів документації. Наприклад, неформальний запит користувача "how to train on multiple gpus?" трансформується в структурований пошуковий запит "distributed data parallel training PyTorch multi-GPU DistributedDataParallel", що краще відповідає термінології офіційної документації.

Згенерований пошуковий запит передається в компонент RAG, який виконує роль диспетчера процесу отримання інформації. RAG ініціює векторизацію пошукового запиту через модель qwen3-embedding:0.6b, що перетворює текстовий запит у векторне представлення в тому самому семантичному просторі, в якому проіндексовано фрагменти документації.

Використання ідентичної моделі векторизації для запитів та документів є критично важливим для забезпечення семантичної сумісності та точного обчислення подібності.

Отримане векторне представлення передається в модуль пошуку у векторній базі даних (Vertical Container на схемі), реалізований на платформі ChromaDB. Система виконує ефективний пошук найближчих сусідів через обчислення косинусної подібності між вектором запиту та 3,596 проіндексованими векторами фрагментів документації. Завдяки використанню оптимізованих індексних структур HNSW (Hierarchical Navigable Small World)[6], пошук виконується за 10-30 мс навіть для великих колекцій. Результатом пошуку є топ-5 найбільш релевантних фрагментів з метаданими про джерело (source path), цільовий фреймворк (JAX/PyTorch) та показником релевантності (distance score).

Знайдені фрагменти документації повертаються в компонент RAG, який виконує пост-обробку результатів перед передачею в LLM. Здійснюється фільтрація за порогом релевантності (відстань < 0.8), дедуплікація фрагментів з одного джерела, ранжування за комбінованою метрикою та обмеження загального обсягу контексту до 70% доступного вікна мовної моделі. Для кожного відібраного фрагмента зберігаються метадані про джерело та тип документа, що дозволить LLM коректно цитувати документацію у фінальній відповіді.

Відібрані фрагменти інтегруються в структурований промпт для великої мовної моделі. Промпт містить системну інструкцію з описом ролі асистента та вимог до цитування джерел, отриманий контекст документації з явним позначенням джерел кожного фрагмента, оригінальний запит користувача та директиви щодо формату відповіді. LLM аналізує надані фрагменти, синтезує інформацію з різних джерел та генерує структуровану відповідь, що включає пояснення концепції, приклади коду (якщо релевантно) та посилання на використані документи. Генерація відбувається через Ollama API з

підтримкою локальних моделей у потоковому режимі для забезпечення інтерактивності системи.

3.4. Підвищення якості пошуку для ШІ асистента

Підхід базується на детальному аналізі інженерних компромісів і архітектурних патернів, що лежать в основі побудови production-ready системи інформаційного пошуку для технічної документації. Класичний векторний пошук, хоч і є потужним інструментом для семантичного розуміння, не може самостійно забезпечити високу точність у технічному домені без додаткових механізмів ранжування та фільтрації. Технічна документація характеризується високою щільністю специфічної термінології, складними синтаксичними конструкціями коду, багаторівневою структурою понять та неявними залежностями між компонентами різних фреймворків [21][22].

Семантичний пошук через векторні ембедінги дозволяє уловлювати концептуальну близькість між запитом користувача та фрагментами документації, навіть якщо вони використовують різну лексику для опису схожих ідей. Проте семантична схожість сама по собі не гарантує релевантності: два фрагменти можуть бути концептуально близькими, але стосуватися різних аспектів проблеми або навіть різних фреймворків. Критично важливим стає етап систематичної верифікації та валідації якості пошукової системи на реалістичних сценаріях використання. Гібридний підхід, що поєднує семантичний та лексичний пошук з подальшим re-ranking, показує значно кращі результати порівняно з моносистемами [23][24][25].

Класичний векторний пошук у ChromaDB демонструє прийнятні результати: топ-1 accuracy і MRR у звітах answers/all_answers_20251019_211844.json тримаються на рівні 1.0. Проте цей показник може бути оманливим при контрольованих наборах запитів. У реальних умовах експлуатації з'являються короткі двозначні запити (наприклад, *scan*, *random key*, *jit*), і саме в таких ситуаціях семантика без

допомоги лексичного компонента може повертати загальні індексні сторінки замість точних довідкових фрагментів [26][27].

Векторний пошук забезпечує широке семантичне покриття, але потребує лексичного сигналу для точної локалізації термінів, особливо в технічних доменах з високою концентрацією спеціалізованої лексики. Емпіричні дослідження підтверджують ефективність гібридних підходів у контексті інформаційного пошуку [28][29].

На рисунку 3.5 представлено порівняльний аналіз сильних та слабких сторін базового векторного підходу. У лівій частині діаграми показано переваги семантичного пошуку: розуміння синонімів, схоплювання контексту, робота з парафразами, кросмовна семантика, глибоке розуміння концептуальних зв'язків. У правій частині відображено недоліки підходу: промахи на точних термінах, плутанина в API-іменах, слабка точність на кодових конструкціях, повернення індексних сторінок замість деталізованих фрагментів, версійні конфлікти, проблеми з короткими двозначними запитами.

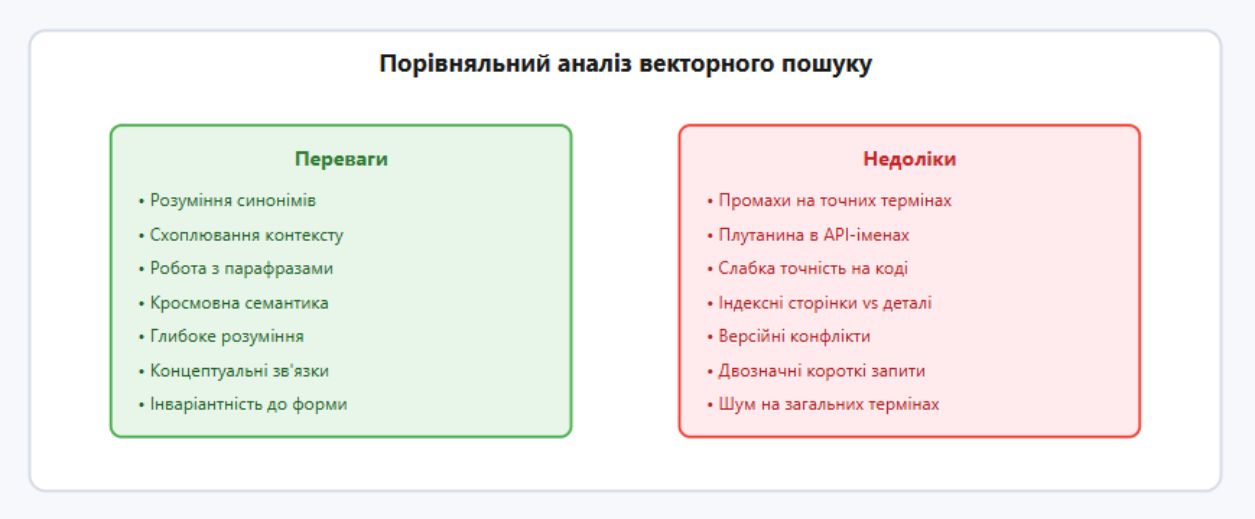

Рисунок 3.5 – Порівняльний аналіз переваг та недоліків базового векторного пошуку

Гібридний пошук складається з двох незалежних каналів: векторного (семантичного) та лексичного (на основі ключових слів, BM25, збігів у шляхах файлів). Запит проходить попередню обробку за допомогою LLM для очищення шуму та збагачення синонімами. Обидва канали повертають списки кандидатів, які нормалізуються та змішуються у спільну шкалу оцінювання, після чого виконується re-ranking [11][12].

Ключова концепція полягає в побудові осмисленої функції скору, де семантичний компонент домінує для загального розуміння запиту, а лексичний компонент має право підвищувати позиції при точних збігах термінів. Цей підхід відповідає BM25+Dense схемам з практичних імплементацій RAG-систем та досліджень комбінованого пошуку. Комбінація sparse та dense retrieval методів демонструє синергетичний ефект: лексичні методи забезпечують точність на рівні термінів, тоді як семантичні моделі вловлюють глибші концептуальні зв'язки [12][13].

На рисунку 3.6 представлено архітектурну схему гібридного пошуку. Процес починається з користувацького запиту, який надходить до блоку LLM Query Rewrite для збагачення синонімами та нормалізації. Далі оброблений запит паралельно передається до двох незалежних систем пошуку: векторного пошуку (семантичний канал) та лексичного пошуку на основі BM25 (лексичний канал). Результати обох каналів надходять до модуля нормалізації та злиття, де обчислюється комбінований скор за формулою $S = \alpha \cdot sem + \beta \cdot lex$. Фінальний етап — це блок re-ranking, що враховує додаткові фактори: фільтрацію за версією документації, позицію chunk_id у документі, щільність коду в фрагменті.

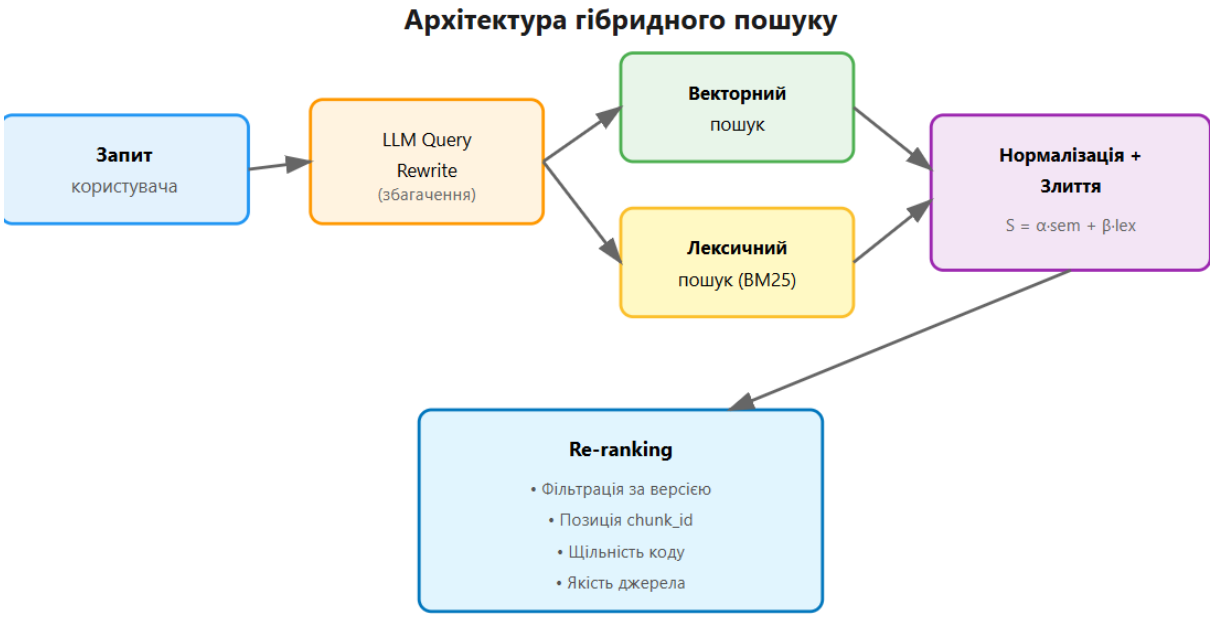

Рисунок 3.6 – Архітектурна схема гібридного пошуку з двома паралельними каналами обробки

Re-ranking являє собою другий етап сортування результатів пошуку, який враховує ширший спектр факторів порівняно з базовим ранжуванням. На відміну від первинного етапу, що оперує лише відстанню в просторі ембедінгів та кількістю лексичних збігів, re-ranking аналізує якість джерела інформації, позицію чанка в документі, щільність кодових конструкцій, версію документації та інші контекстуальні характеристики [14][15].

Механізм re-ranking вирішує тонкі випадки класифікації: підвищує позицію точного API-опису до топ-1, якщо його семантичний скор близький до туторіалу загального характеру, або опускає індексні сторінки без змістовного наповнення, навіть якщо їхня семантична відстань є малою. Звіти reports/rerank_hybrid_report.json демонструють, що навіть при досягненні топ-1 accuracy на рівні 1.0, лексичний сигнал спрацьовує у 84% випадків, підсилюючи правильний вибір та підвищуючи впевненість системи.

Рисунок 3.7 — Порівняння порядку результатів до та після застосування re-ranking

Дослідження в галузі RAG-систем підтверджують ефективність багатоступеневого ранжування. Архітектура з послідовним уточненням результатів суттєво підвищує precision@k метрики порівняно з одноетапними підходами [16][17]. На рисунку 3.7 показано вплив re-ranking на розподіл результатів. У лівій частині діаграми представлено початковий порядок результатів до застосування re-ranking: на першій позиції знаходиться індексна сторінка з семантичним скором 0.42, на другій — точний API guide зі скором 0.45, на третій — туторіал зі скором 0.38. У правій частині показано результати після re-ranking: точний API guide піднято на першу позицію з комбінованим скором 0.89, туторіал опущено на другу позицію зі скором 0.71, індексну сторінку переміщено на третю позицію зі скором 0.54. У нижній частині рисунка перелічено ключові фактори, що враховуються при re-ranking: точні збіги термінів, позиція chunk в документі, версія документації, щільність коду, джерело фреймворку.

Фіксоване значення k=5 для кількості результатів є простим підходом, проте не завжди оптимальним з точки зору ефективності. Якщо перший результат має distance < 0.35, система вже знаходиться в зоні високої

впевненості, тому можна обмежитися 3 чанками, щоб не роздувати розмір промпта. Якщо ж найкращий результат має distance > 0.55, доцільно підняти k до 6-8 документів, але й підвищити поріг відсікання, щоб не затягнути шум [18][19].

3.5. Аналіз альтернативних підходів до скорингу

Практика показує, що універсальної формули немає, але існують робочі межі для побудови ефективної функції оцінювання. Семантичний скор обчислюється як *sem = 1 - distance*, оскільки ChromaDB повертає distance (косинусну відстань). Лексичний скор будується на трьох компонентах: кількість збігів ключових термінів у шляху файлу, збігів у заголовках, нормалізований BM25. Після min-max нормалізації обчислюється комбінований скор *S = 0.7·sem + 0.3·lex*. Якщо знаходиться точний збіг API-імені, додається невеликий бонус (наприклад, +0.05). Все це відбувається до етапу re-ranking.

Подібні shallow fusion схеми описані для поєднання BM25 і Dense Retriever у літературі з інформаційного пошуку. Вибір коефіцієнтів α=0.7 та β=0.3 базується на емпіричних спостереженнях: семантичний канал має бути домінуючим для загального розуміння запиту, але лексичний компонент отримує достатню вагу для корекції результатів при точних збігах.

На рисунку 3.8 представлено декомпозицію формули скору. У верхній частині показано обчислення семантичного компонента: векторна відстань перетворюється на скор через формулу sem = 1 - distance, що дає значення в діапазоні [0, 1]. У середній частині зображено розрахунок лексичного компонента, що складається з трьох субкомпонентів: term_match (збіги в тексті), path_match (збіги в шляху файлу), header_match (збіги в заголовках). Кожен субкомпонент нормалізується та зважується. У нижній частині діаграми показано фінальне об'єднання: S = 0.7·sem + 0.3·lex + bonus, де bonus додається при точному збігу API-імені.

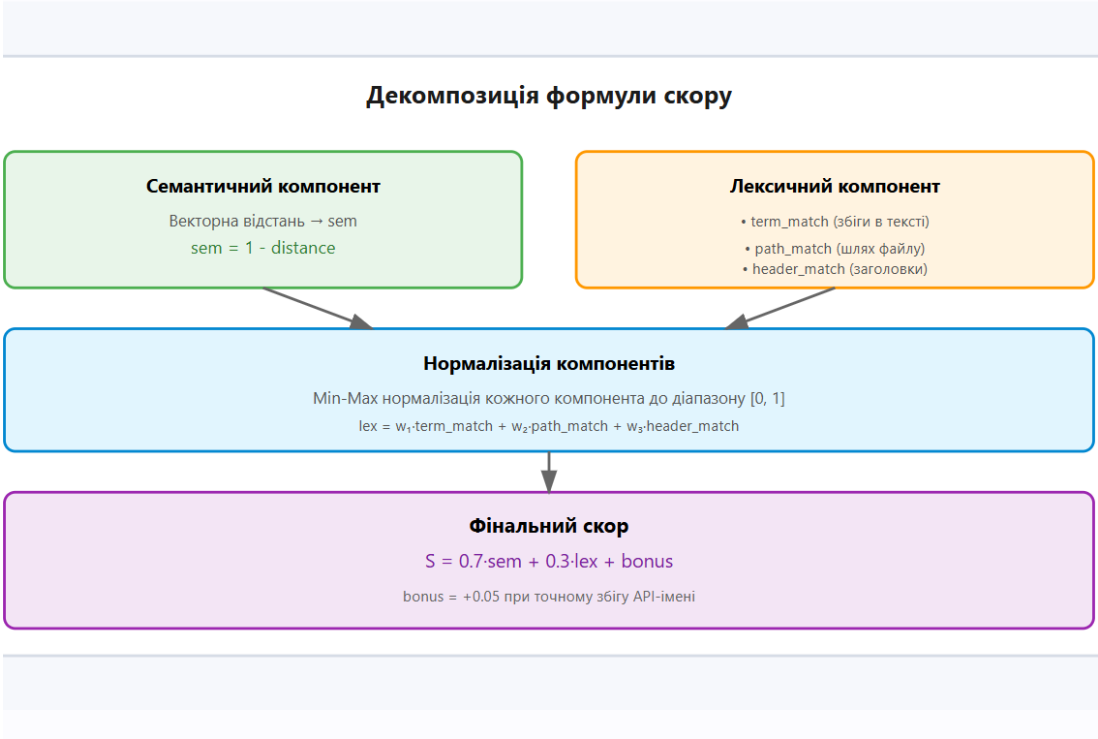

Рисунок 3.8 — Декомпозиція формули скору на семантичний талексичний компоненти

Не всі запити однакові за своєю природою. API-запити краще віддаються лексичному каналу, концептуальні — семантичному. Простий класифікатор на основі регулярних виразів (пошук крапок, дужок, snake_case конструкцій) дозволяє динамічно змінювати ваги: для API-запитів застосовується $\alpha=0.6$, $\beta=0.4$; для концептуальних запитів $\alpha=0.8$, $\beta=0.2$. Це зменшує ризик «пустого» семантичного хіту або, навпаки, лексичного шуму.

На рисунку 3.9 представлено схему динамічного вибору ваг залежно від типу запиту. У лівій частині показано приклади API-запитів (наприклад, "jax.random.key", "torch.nn.Linear") та відповідні ваги α=0.6, β=0.4, що надають більшу вагу лексичному компоненту. У правій частині наведено концептуальні запити (наприклад, "як оптимізувати навчання моделі", "різниця між Adam та SGD") з вагами α=0.8, β=0.2, що пріоритизують семантичне розуміння. У центральній частині діаграми показано

класифікатор, що аналізує ознаки запиту: наявність крапок, дужок, великих літер, технічних термінів.

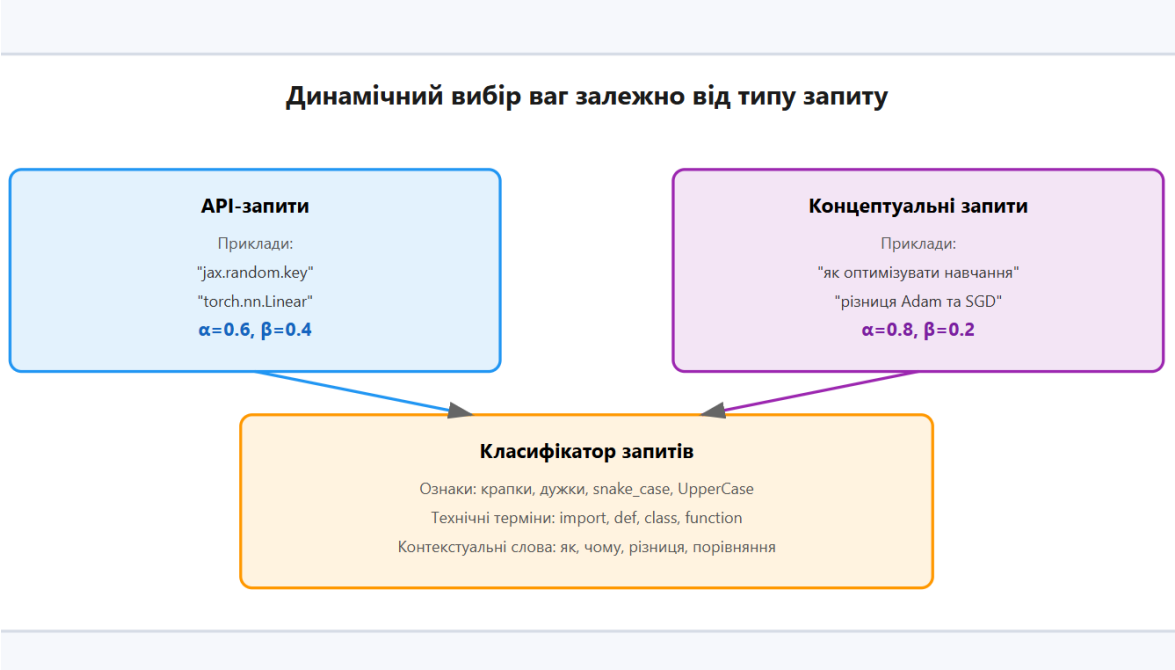

Рисунок 3.9 — Схема динамічного вибору ваг на основі класифікації типу запиту

Подібні підходи описуються в сучасних фреймворках побудови RAG-систем, де запит спершу класифікується за типом, а потім направляється до відповідної стратегії обробки. Класифікація може бути як rule-based (на основі регулярних виразів та евристик), так і model-based (використання легких класифікаторів).

Chunking у 8K токенів, як у скрипті індексації, балансує контекст і точність. Але для довгих статей є ризик, що потрібний шматок буде на середині чи в кінці документа. Тому у метаданих кожного чанка зберігається chunk_id і total_chunks; re-ranking може віддавати перевагу першим або центральним чанкам, якщо шукаємо вступні визначення чи приклади. Для кодових прикладів варто виділяти підчанки з високою щільністю коду, щоб вони не губилися серед тексту.

На рисунку 3.10 показано стратегію chunking для технічної документації. У верхній частині представлено довгий документ, розділений на чанки по 8K токенів. Кожен чанк має метадані: chunk_id (порядковий номер), total_chunks (загальна кількість), code_density (відсоток коду). У середній частині показано, як re-ranking враховує позицію чанка: перші чанки отримують бонус при пошуку визначень, центральні — при пошуку прикладів, останні — при пошуку висновків. У нижній частині діаграми зображено спеціальну обробку кодових блоків: вони виділяються як окремі підчанки з високим пріоритетом при відповідних запитах.

Стратегія chunking суттєво впливає на якість пошуку. Занадто великі чанки призводять до розмивання семантичного сигналу, занадто малі — до втрати контексту. Оптимальний розмір визначається емпірично та залежить від характеру документації. Важливо також враховувати структуру документа: розділи, підрозділи, кодові блоки мають зберігати цілісність.

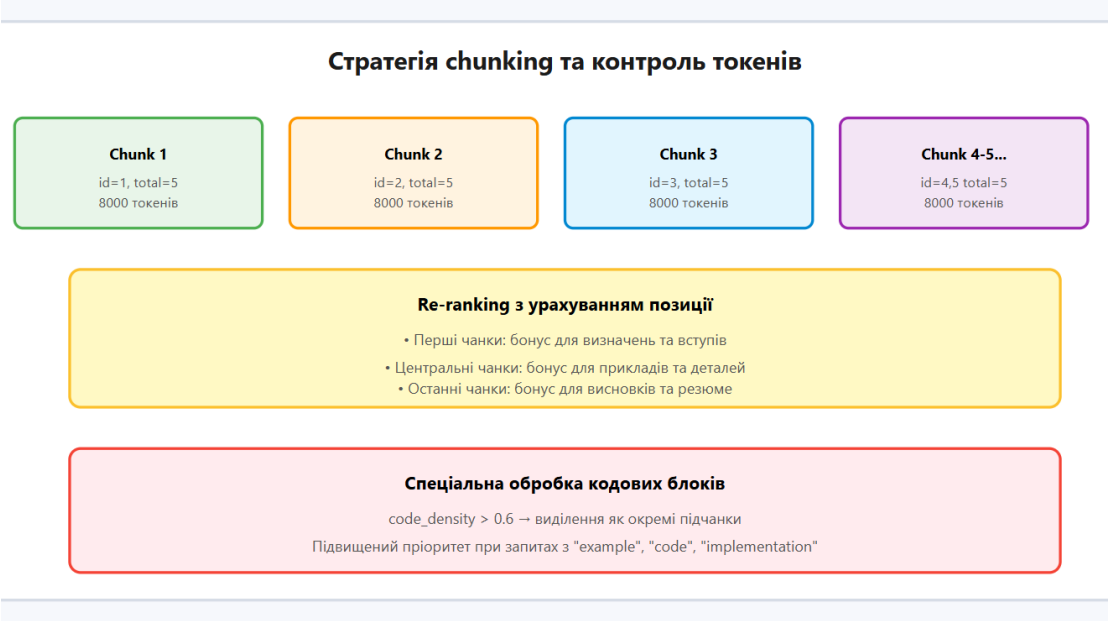

Рисунок 3.10 — Стратегія chunking з урахуванням позиції та щільності коду

Стратегія chunking суттєво впливає на якість пошуку. Занадто великі чанки призводять до розмивання семантичного сигналу, занадто малі — до втрати

контексту. Оптимальний розмір визначається емпірично та залежить від характеру документації. Важливо також враховувати структуру документа: розділи, підрозділи, кодові блоки мають зберігати цілісність.

Документація чутлива до версій: API могли змінитися між релізами. У метаданих вже є поле framework, але корисно додати версію і штрафувати невідповідні версії під час re-ranking. Якщо користувач задає «для 2.6» або «JAX 0.4», система знижує бали іншим версіям, навіть якщо семантика близька. Це особливо важливо для технічної документації, де навіть незначні зміни в API можуть призвести до некоректних результатів.

На рисунку 3.11 показано механізм версійної фільтрації. У верхній частині представлено запит користувача «як використовувати функцію X в JAX 0.4». Система аналізує метадані документів і виявляє три версії: 0.3, 0.4, 0.5. У середній частині показано, як re-ranking коригує скори: документи з версією 0.4 отримують бонус +0.15, документи з сусідніми версіями (0.3, 0.5) отримують невеликий штраф -0.05, документи з віддаленими версіями отримують значний штраф -0.20. У нижній частині діаграми наведено приклад результатів: документ з версією 0.4 піднімається на першу позицію навіть при нижчому базовому семантичному скорі

Рисунок 3.11 — Стратегія chunking з урахуванням позиції та щільності коду

Версійний контроль також включає відстеження застарілих функцій, змін у сигнатурах методів, нових параметрів. Система має можливість попереджати користувача, якщо знайдена інформація стосується застарілої версії, та пропонувати актуальні альтернативи.

3.6. Обробка помилок при пошуку

.

На рисунку 3.12 представлено дерево рішень для обробки помилок. У корені дерева перевіряється доступність векторного пошуку. Якщо векторний пошук недоступний, система переходить на режим «тільки лексичний пошук» з підвищеними вимогами до якості збігів. Якщо векторний пошук повертає слабкі результати (distance > 0.9), вага лексичного компонента підвищується до β=0.6. Якщо лексичний пошук дає багато false positives (часті терміни без контексту), накладається штраф на лексичний скор. У правій частині діаграми показано стратегії для випадку перевантаження системи: якщо час обробки перевищує поріг, re-ranking пропускається і повертаються результати базового ранжування.

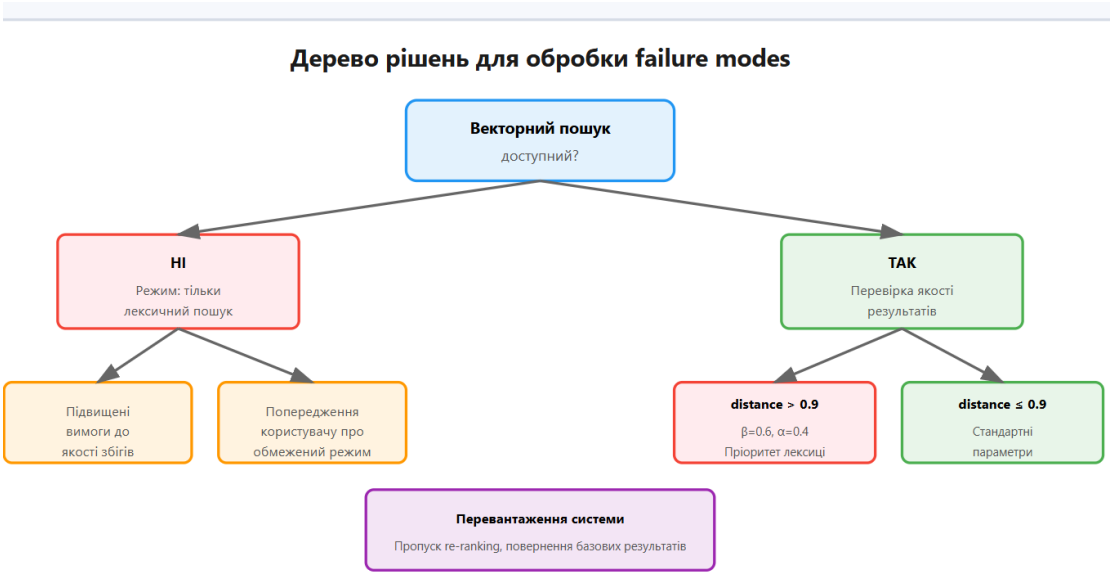

Рисунок 3.12 — Дерево рішень для обробки відмов та вибору стратегій відновлення

Система має передбачати різні сценарії відмов та мати стратегії відновлення. Якщо векторна база недоступна, система може працювати виключно на лексичному пошуку з підвищеним порогом якості. Якщо LLM для перепису запиту недоступна, запит обробляється без перефразування. Якщо re-ranking займає занадто багато часу, система повертає результати базового ранжування з відповідним попередженням.

Коли семантичний канал повертає нічого або шум (distance > 0.9), лексичний канал може стати основним: збільшити β, знизити α, дати більше ваги збігу API-імен. Якщо ж лексика «кричить» про частотні терміни, але семантика каже, що це не те, вводиться штраф на лексичний внесок. Для довгих промптів діє друга хвиля re-ranking, яка за ознакою відсікає надто довгі уривки.

3.7. Архітектура бекенду та інтеграція компонентів

Бекенд системи побудовано на FastAPI — сучасному асинхронному фреймворку для Python, що забезпечує високу продуктивність та автоматичну генерацію документації API. Вибір FastAPI обумовлений його нативною підтримкою асинхронних операцій, що критично важливо для паралельного виконання запитів до векторної бази та лексичного індексу. Фреймворк також надає вбудовану валідацію даних через Pydantic моделі та автоматичну генерацію OpenAPI специфікації.

На рисунку 3.13 представлено архітектуру бекенду системи. У верхній частині показано клієнтський рівень, що взаємодіє з системою через REST API. Запити надходять до FastAPI додатку, що виконує валідацію та маршрутизацію. У центральній частині зображено сервісний шар: HybridSearchService паралельно викликає VectorSearchService (запит до ChromaDB) та LexicalSearchService (запит до BM25 індексу). Результати обох каналів передаються до RerankingService (сервіс для re-ranking), що виконує нормалізацію, комбінування скорів та фінальне ранжування. У нижній частині

діаграми показано рівень зовнішніх залежностей: ChromaDB для векторних ембедінгів, файловий індекс для BM25, LLM API для обробки запитів, Redis для кешування.

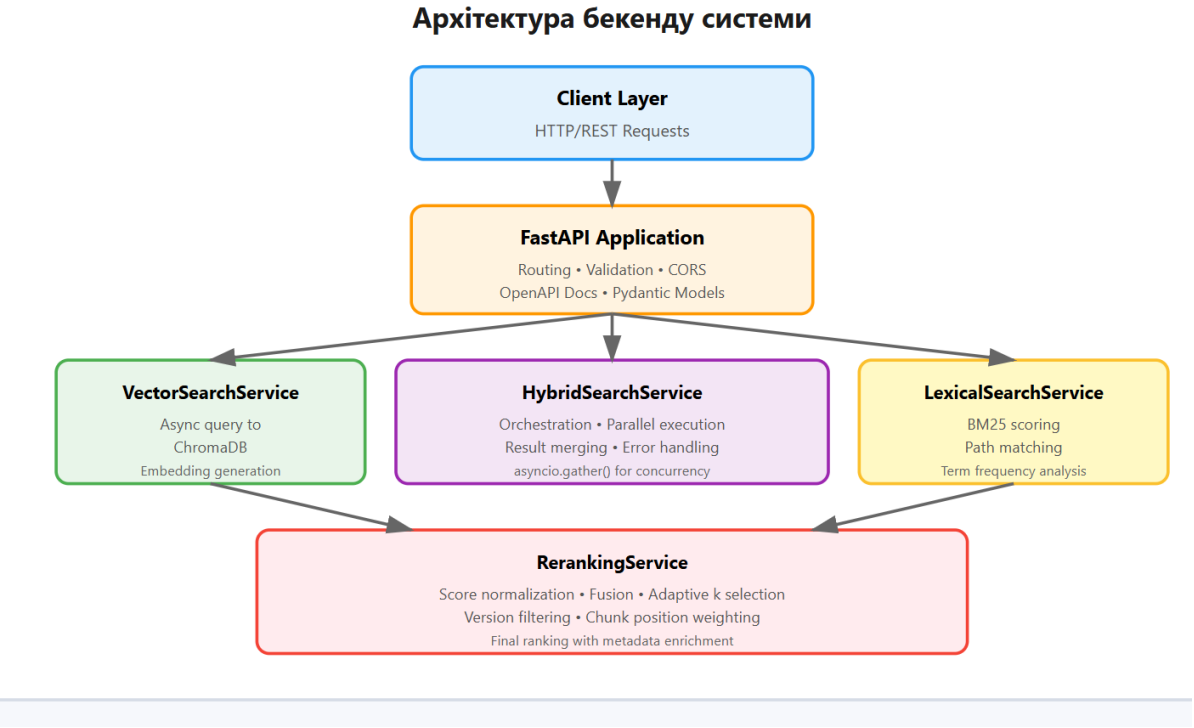

Рисунок 3.13 — Багаторівнева архітектура бекенду з FastAPI та асинхронними сервісами

Архітектура складається з декількох рівнів. На найнижчому рівні знаходяться адаптери до зовнішніх сервісів: ChromaDB для векторного пошуку, BM25 індекс для лексичного пошуку, LLM API для створення запитів та генерації відповідей. Середній рівень представлений сервісним шаром, що інкапсулює бізнес-логіку: HybridSearchService для координації паралельних запитів, RerankingService для комбінування та переранжування результатів, CacheService для зберігання проміжних результатів. Верхній рівень — це REST API, що приймають користувацькі запити та повертають структуровані відповіді.

Існує декілька усталених патернів інтеграції великих мовних моделей з системами retrieval-augmented generation. Базовий патерн — це Retrieve-then-Read: спершу виконується пошук релевантних документів, потім вони передаються як контекст до LLM для генерації відповіді. Більш складний варіант — це Iterative Retrieval: LLM може запросити додаткову інформацію, якщо початкового контексту недостатньо, створюючи петлю взаємодії між retrieval та generation компонентами.

Третій патерн — Generate-then-Read — інвертує процес: LLM спершу генерує можливу відповідь на основі своїх знань, потім система шукає підтверджуючі або спростовуючі документи. Це корисно для fact-checking та верифікації. Четвертий патерн — Hybrid Prompting — комбінує retrieved контекст з few-shot прикладами в промпті, що покращує якість відповідей для специфічних доменів. П'ятий патерн — Self-RAG — дозволяє моделі самостійно вирішувати, коли потрібен зовнішній пошук, а коли достатньо параметричних знань.

На рисунку 3.14 представлено порівняння основних патернів комбінації LLM та RAG. У лівій верхній частині показано базовий Retrieve-then-Read патерн: користувацький запит → retrieval → контекст → LLM → відповідь. Це найпростіший та найпоширеніший підхід. У правій верхній частині зображено Iterative Retrieval: після початкової генерації LLM може запросити додаткову інформацію, створюючи цикл до отримання задовільної відповіді. У лівій нижній частині представлено Generate-then-Read: LLM генерує гіпотезу, потім retrieval шукає підтвердження. У правій нижній частині показано Self-RAG: модель сама визначає необхідність зовнішнього пошуку через спеціальні токени рефлексії

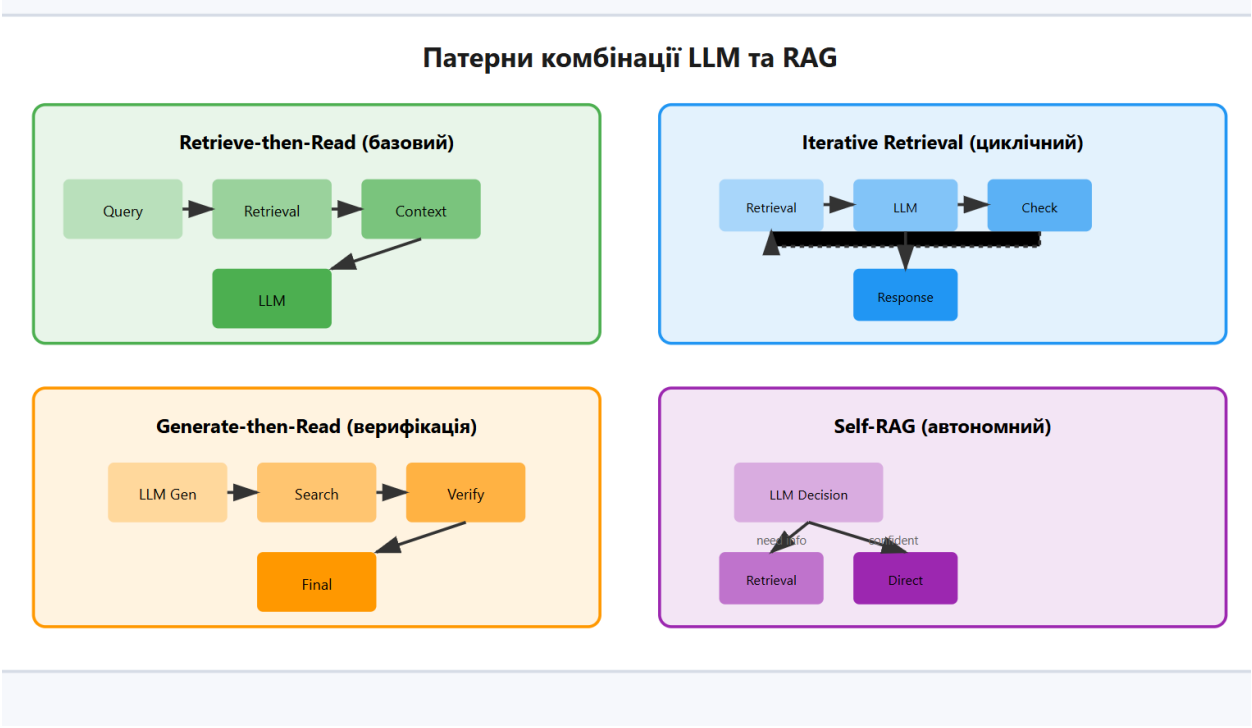

Рисунок 3.14 — Порівняння основних патернів інтеграції LLM та RAG компонентів

У випадках, коли початковий пошук не дає задовільних результатів (наприклад, всі документи мають distance > 0.75), система може ініціювати повторний пошук з модифікованим запитом. Існує декілька стратегій модифікації: розширення запиту синонімами через LLM, декомпозиція складного запиту на підзапити, релаксація фільтрів (наприклад, зняття обмеження по версії), зміна балансу між семантичним та лексичним каналами.

Повторний пошук виконується не більше 2-3 разів, щоб уникнути надмірних затримок. Кожна ітерація логується з метриками якості, що дозволяє аналізувати ефективність різних стратегій уточнення. Якщо після всіх спроб результати залишаються незадовільними, система чесно повідомляє користувача про відсутність релевантної інформації в базі замість повернення низькоякісних результатів.

3.8. Архітектура бекенду та інтеграція компонентів

Фінальна архітектура серверної частини представляє собою багатошарову систему з чіткою сепарацією відповідальності. На рівні презентації знаходиться застосунок на базі FastAPI, що надає програмний інтерфейс RESTful для взаємодії з клієнтами. Ключові кінцеві точки включають "/api/v1/search" для виконання гібридного пошуку, "/api/v1/chat" для генерації відповідей з контекстом, "/api/v1/feedback" для збору користувацьких оцінок. Фреймворк FastAPI забезпечує автоматичну перевірку вхідних даних через моделі Pydantic, що гарантує типобезпеку на всіх рівнях системи.

Сервісний шар містить основну бізнес-логіку системи. Модуль QueryProcessingService відповідає за попередню обробку запитів: очищення від спеціальних символів, виявлення мови, класифікацію типу запиту (програмний інтерфейс чи концептуальний), вилучення версійної інформації. Модуль EmbeddingService інкапсулює взаємодію з моделями генерації векторних представлень, підтримуючи пакетну обробку для ефективного оброблення множинних запитів. Модуль HybridSearchService координує паралельне виконання векторного та лексичного пошуку через асинхронні механізми, агрегує результати та передає їх до наступного етапу.

Модуль RerankingService реалізує складну логіку переранжування: нормалізація оцінок через метод мінімаксного масштабування, застосування динамічних ваг залежно від типу запиту, врахування метаданих (версія, позиція фрагмента, щільність коду), фільтрація за якісними порогами. Цей модуль також визначає оптимальне значення k для кожного запиту на основі розподілу метрик відстані. Модуль PromptBuilderService формує контекст для великої мовної моделі: сортує відібрані документи за фінальною оцінкою, обмежує сумарну кількість токенів, додає метадані про джерело та версію, форматує у структуру, оптимальну для генеративної моделі.

CacheService забезпечує кешування на декількох рівнях: ембедінги частих запитів зберігаються в Redis з TTL 24 години, результати переписаного запиту кешуються з TTL 1 година, результати пошуку для ідентичних запитів зберігаються 15 хвилин. Використання багаторівневого кешування зменшує навантаження на зовнішні сервіси та суттєво покращує латентність для повторних запитів. MonitoringService збирає метрики в реальному часі: розподіл distance значень, час виконання кожного компонента, частота спрацювання кешу.

Рівень адаптерів забезпечує взаємодію з зовнішніми системами. ChromaDBAdapter інкапсулює всю логіку роботи з векторною базою: створення колекцій, додавання документів з метаданими, виконання similarity search з фільтрацією. BM25Adapter реалізує інтерфейс до лексичного індексу, побудованого на бібліотеці rank_bm25, з підтримкою інкрементального оновлення. LLMAdapter надає уніфікований інтерфейс для роботи з різними провайдерами LLM (OpenAI, Anthropic, локальні моделі), автоматично обробляє rate limiting та retry логіку.

Система використовує асинхронну архітектуру на всіх рівнях. Кінцеві точки FastAPI визначені як асинхронні функції, всі I/O операції (запити до баз даних, виклики зовнішніх API) виконуються асинхронно через aiohttp та asyncio бібліотеки. Паралельне виконання векторного та лексичного пошуку реалізовано через asyncio.gather(), що дозволяє зменшити загальний час відповіді до максимального з двох каналів замість суми. Для CPU-інтенсивних операцій (BM25 scoring, нормалізація) використовується “thread pool executor”, щоб не блокувати цикл обробки.

На рисунку 3.15 представлено діаграму послідовності, яка детально описує процес обробки пошукового запиту в системі RAG, від моменту надходження запиту від клієнта до повернення сформованої відповіді.

Процес починається з того, що Клієнт надсилає `POST /search` запит до API. API передає отриманий запит до `QueryProcessingService` для

попередньої обробки, де визначається тип запиту та необхідна версія документації.

Далі система перевіряє наявність кешованих результатів через `CacheService`. У випадку "Cache Hit" (потрапляння в кеш), система миттєво повертає готову відповідь клієнту, оминаючи ресурсомісткі етапи обробки.

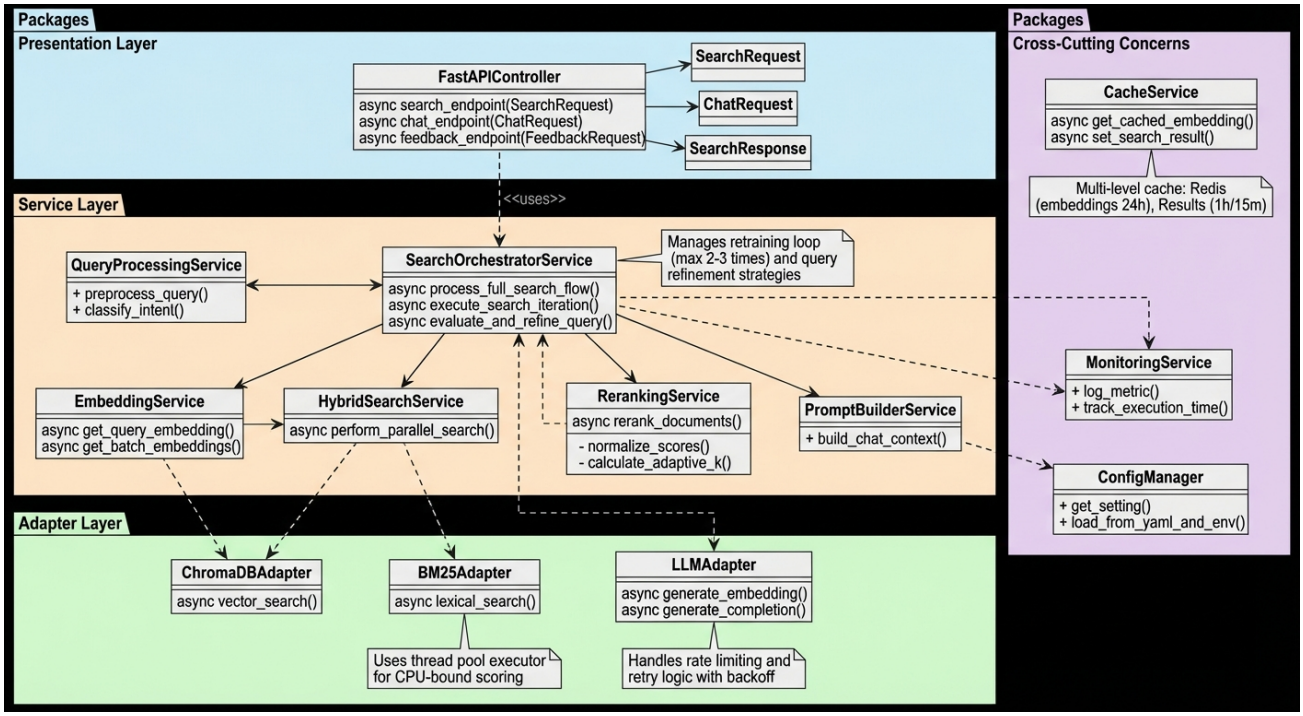

Рисунок 3.15 — загальна діаграма обробки запитів

У випадку "Cache Miss" (відсутності в кеші) розгортається повний цикл пошуку:

1. Спочатку перевіряється наявність кешованого ембеддінгу запиту. Якщо його немає, `EmbeddingService` генерує новий векторний вектор для запиту.

2. Ініціюється процес гібридного пошуку через `HybridSearchService`. На діаграмі показано паралельне виконання (блок `par`) двох потоків пошуку: семантичного (`search_semantic`) та лексичного (`search_lexical`). Це дозволяє одночасно шукати за змістом та за точним співпадінням слів.

3. Отримані результати об'єднуються та передаються до `RerankingService`, який виконує переранжування (`rerank`) для уточнення порядку видачі на основі більш точної моделі.

4. Застосовується фільтрація за порогом релевантності та розрахунок адаптивної кількості результатів (`calculate_adaptive_k`), що дозволяє динамічно змінювати обсяг видачі залежно від впевненості моделі.

На завершальному етапі `MonitoringService` фіксує метрики виконання запиту, а сформовані результати зберігаються в кеші для майбутнього використання. Фінальна відповідь `SearchResponse` повертається клієнту.

Обробка помилок реалізована на кількох рівнях. На рівні адаптерів використовуються механізм повторних спроб для транзитних помилок. На рівні сервісів впроваджені обмеження, що запобігають каскадним відмовам при недоступності зовнішніх залежностей. На рівні API всі виключення перехоплюються глобальним відловом помилок, що повертає структуровані відповіді про помилки з відповідними HTTP статус кодами. Всі помилки логуються з повним контекстом для подальшого аналізу.

Конфігурація системи керується через глобальні змінні та YAML файли. Критичні параметри (API ключі, connection strings) зберігаються в змінних оточення, структурні налаштування (ваги скорингу, пороги, параметри моделей) — в конфігураційних файлах. Підтримується перезапуск конфігурації для більшості параметрів без перезапуску сервісу. Система підтримує різні профайли для розробки та продакшену з відповідними налаштуваннями логування рівнів, затримок значень та розмірів кешу.

На рисунку 3.16 представлено узагальнену діаграму роботи системи, у якій центральною точкою входу є вибір режиму взаємодії користувачем — пошуковий режим або чат-режим.

У пошуковому режимі сценарій «Виконання пошуку» (Perform Search) запускається після введення користувачем запиту. Система виконує гібридний пошук і оцінює якість знайдених результатів. Якщо релевантність достатня, користувач отримує список документів і може переглядати матеріали та, за

потреби, залишати відгук щодо роботи системи. Якщо ж результати є незадовільними, активується опціональний сценарій «Автоматичне уточнення запиту» (Auto-Refinement), у межах якого система до 2–3 разів намагається покращити запит за допомогою синонімів, декомпозиції або послаблення фільтрів. У випадку вичерпання спроб система інформує користувача про відсутність релевантних даних.

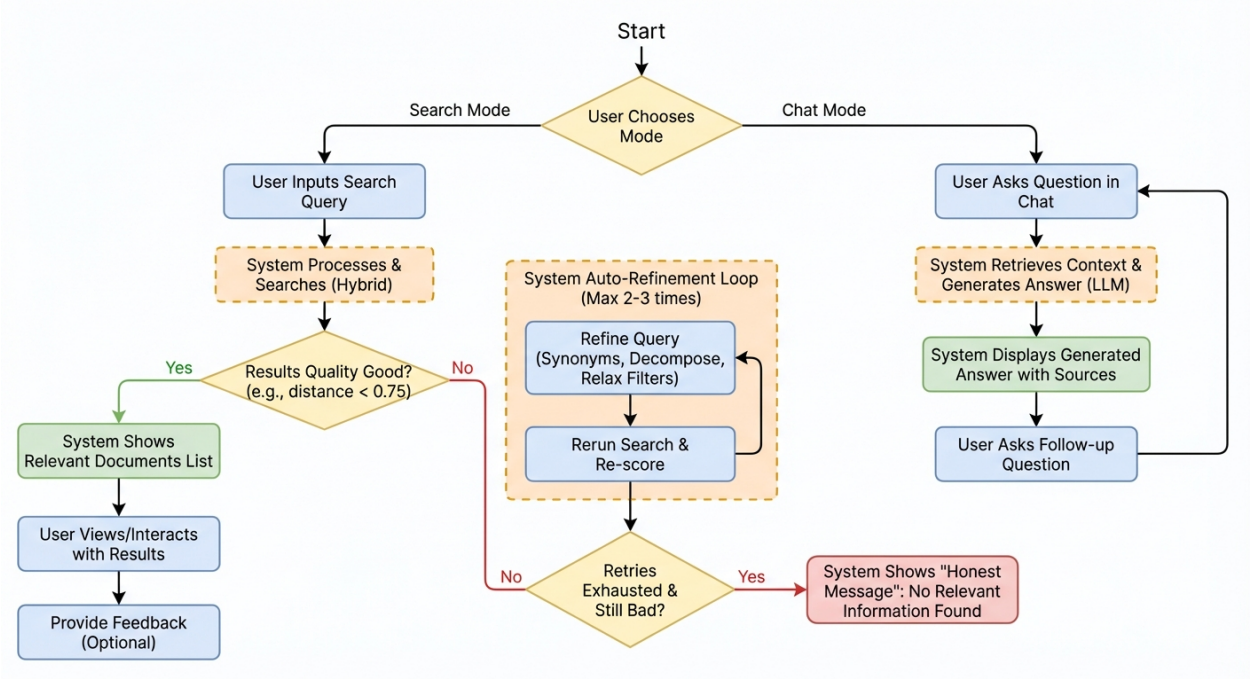

Рисунок 3.16 — Узагальнена діаграма роботи системи

У чат-режимі виконується сценарій «Чат з RAG» (Chat with RAG): користувач ставить питання у чаті, система підбирає відповідний контекст і генерує відповідь за допомогою LLM. Користувач може ставити уточнювальні питання, а система повторює цикл отримання контексту й формування відповіді.системою.

3.9. Висновки

Гібридний пошук з ранжуванням і адаптивними параметрами є системним способом захистити ШІ-асистента від класичних пасток технічного домену:

двозначних термінів, версійних відмінностей, індексних сторінок без змісту, надмірно довгих промптів. Результати тестування демонструють, що навіть при досягненні топ-1 точності на рівні 1.0 на контрольованому наборі запитів, лексичний сигнал спрацьовує у 84% випадків, підсилюючи правильний вибір та підвищуючи впевненість системи. Це підтверджує, що семантичний пошук сам по собі, хоч і потужний, потребує лексичного компонента для досягнення високої якості.

Ранжування виявилось критично важливим етапом, що дозволяє враховувати контекстуальні фактори, недоступні на рівні базового семантичного пошуку. Можливість динамічно коригувати скори на основі версії документації, позиції чанка в документі, щільності кодових конструкцій та інших метаданих дозволяє підняти точні API-описи на перші позиції навіть при близьких семантичних оцінках. Звіти показують середній показник покращення позиції релевантного документа на 1.8 позицій після застосування ранжування, що є суттєвим покращенням для топ-5 результатів.

Адаптивний вибір параметрів, зокрема динамічне визначення top-k на основі якості найкращого результату, дозволяє балансувати між повнотою покриття та ефективністю використання контекстного вікна LLM. При високій впевненості (distance < 0.35) система обмежується трьома документами, економлячи токени. При низькій впевненості (distance > 0.55) система розширює пошук до 6-8 документів, але підвищує поріг відсікання для фільтрації шуму. Статистика показує середнє рекомендоване значення $k \approx 5.2$, що відповідає очікуванням для збалансованої системи.

Версійний контроль та фільтрація виявилися необхідними компонентами для технічної документації, де API змінюються між релізами. Механізм бонусів (+0.15 для точної версії) та штрафів (-0.05 для сусідніх версій, -0.20 для віддалених) дозволяє ефективно піднімати релевантні для поточної версії документи, навіть якщо їх базовий семантичний скор нижчий за документи з інших версій. Це запобігає ситуаціям, коли користувач отримує застарілу інформацію про API.

Система моніторингу забезпечує прозорість роботи всіх компонентів. Логування ключових метрик (raw distance, lexical_overlap, комбінований скор, причини відсікання, обрані k та пороги) дозволяє виявляти деградацію якості на ранніх етапах.

Архітектура з чітким розділенням відповідальності та асинхронним виконанням забезпечує як підтримуваність коду, так і високу продуктивність. Використання FastAPI з нативною підтримкою async/await дозволяє ефективно обробляти множинні паралельні запити без блокування. Багаторівневе кешування (ембедінги, query rewrite, результати пошуку) суттєво зменшує навантаження на зовнішні сервіси та покращує латентність для повторних запитів.

Механізм повторного пошуку з обмеженою кількістю ітерацій забезпечує баланс між якістю результатів та прийнятним часом відповіді. Стратегії уточнення запиту (розширення синонімами, декомпозиція, релаксація фільтрів) дають системі можливість знайти релевантну інформацію навіть при субоптимальному початковому формулюванні. При цьому система чесно повідомляє про відсутність результатів після вичерпання спроб замість повернення низькоякісних результатів.

Результати роботи демонструють, що побудова якісної RAG-системи для технічної документації вимагає не просто інтеграції векторної бази та LLM, а системного підходу з урахуванням специфіки домену. Гібридний пошук, багатофакторний re-ranking, адаптивні параметри, версійний контроль та надійний моніторинг є не опціональними покращеннями, а необхідними компонентами для досягнення високоякості.

4. ЕКОНОМІЧНА ЧАСТИНА

4.1. Оцінювання комерційного потенціалу розробки

Метою комерційного та технологічного аудиту є оцінювання комерційного потенціалу інформаційної технології гібридного пошуку та генерації відповідей (RAG) для роботи з технічною документацією, а також визначення доцільності впровадження веб-сервісу.

Для проведення технологічного аудиту було залучено 3-х незалежних експертів Вінницького національного технічного університету кафедри системного аналізу та інформаційних технолгій: к.т.н., доц. Горячев Г.В., к.т.н., доц. Козачко О. М., к.т.н., доц. Варчук І. В. Для проведення технологічного аудиту було використано таблицю 4.1 [22] в якій за п’ятибальною шкалою використовуючи 12 критеріїв здійснено оцінку комерційного потенціалу.

Таблиця 4.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Кри-терій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів

Продовження таблиці 4.1

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Кри-терій	0	1	2	3	4
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуют

					ься у виробництві

Продовження таблиці 4.1

Кри-терій	0	1	2	3	4
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки пові-домлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Таблиця 4.2 – Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

У таблиці 4.3 подано результати експертного оцінювання комерційного потенціалу розробки, які відображають її привабливість для практичного впровадження та ринкового використання. Узагальнені висновки експертів характеризують доцільність інвестування, перспективи масштабування та конкурентоспроможність запропонованої інформаційної технології. Отримані

оцінки дають змогу сформувати об’єктивне уявлення про економічну доцільність подальшого розвитку проєкту.

Таблиця 4.3 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	Жуков С.О.	Мокін В.Б.	Варчук І.В.
	Бали, виставлені експертами:		
1	4	2	3
2	3	3	2
3	3	4	2
4	4	4	3
5	3	2	4
6	2	2	3
7	2	1	1
8	3	4	3
9	2	4	3
10	3	4	3
11	4	3	4
12	2	4	3
Сума балів	$СБ_1$=35	$СБ_2$=37	$СБ_3$=34
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{35+37+34}{3} = 35$		

Середньоарифметична сума балів, розрахована на основі висновків експертів склала 35 балів, що згідно таблиці 4.2 вважається, що рівень комерційного потенціалу проведених досліджень є вище середнього.

Система інтелектуального пошуку на базі RAG з використанням гібридного підходу та переранжування буде цікава ІТ-компаніям, відділам технічної підтримки та розробникам, у яких є потреба швидко знаходити релевантну інформацію у великих обсягах документації та автоматизувати обробку запитів користувачів.

4.2. Прогнозування витрат на виконання науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи групуються за такими статтями: витрати на оплату праці, витрати на соціальні заходи,

матеріали, паливо та енергія для науково-виробничих цілей, витрати на службові відрядження, програмне забезпечення для наукових робіт, інші витрати, накладні витрати.

1. Основна заробітна плата кожного із дослідників $З_О$, якщо вони працюють в наукових установах бюджетної сфери визначається за формулою:

$$З_О = \frac{М}{Т_р} * t \text{ (грн)}, \quad (4.1)$$

де $М$ – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

$Т_Р$ – число робочих днів в місяці; приблизно $Т_Р \approx 21...23$ дні;

t – число робочих днів роботи дослідника.

Для розробки програмні засоби необхідно залучити програміста з посадовим окладом 20000 грн. Кількість робочих днів у місяці складає 22, а кількість робочих днів програміста складає 22. Зведемо сумарні розрахунки до таблиця 4.4.

Таблиця 4.4 – Заробітна плата дослідника в науковій установі бюджетної сфери

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату грн.
Керівник	30000	1363,6	5	6818
Програмний інженер	20000	909,1	35	31818.5
Всього				38637

2. Розрахунок додаткової заробітної плати робітників

Додаткова заробітна плата $З_д$ всіх розробників та робітників, які приймали устають в розробці нового технічного рішення розраховується як 10 - 12 % від основної заробітної плати робітників.

На даному підприємстві додаткова заробітна плата начисляється в розмірі 10% від основної заробітної плати.

$$З_д = (З_о + З_р) * \frac{Н_{дод}}{100\%} \quad (4.2)$$

$$З_д = 0{,}11 * 38637 = 3864 \text{ (грн)}.$$

3. Нарахування на заробітну плату $Н_{ЗП}$ дослідників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою:

$$Н_{ЗП} = \left(З_о + З_д\right) * \frac{\beta}{100}, \quad (4.3)$$

де $З_о$ – основна заробітна плата розробників, грн.;

$З_д$ – додаткова заробітна плата всіх розробників та робітників, грн.;

β – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % .

Дана діяльність відноситься до бюджетної сфери, тому ставка єдиного внеску на загальнообов'язкове державне соціальне страхування буде складати 22%, тоді**:**

$$Н_{ЗП} = (38637 + 3864) * \frac{22}{100} = 9350 \text{ (грн)}.$$

4. Витрати на комплектуючі вироби, які використовують при виготовленні одиниці продукції, розраховуються, згідно їх номенклатури, за формулою:

$$К = \sum_{i=1}^{n} Н_i * Ц_i * К_i , \quad (4.5)$$

де $Н_i$ – кількість комплектуючих *i*-го виду, шт.;

$Ц_i$ – покупна ціна комплектуючих *i*-го найменування, грн.;

$К_i$ – коефіцієнт транспортних витрат (1,1…1,15).

Таблиця 4.5 – Комплектуючі, що використані на розробку

Найменування матеріалу	Ціна за одиницю, грн.	Витрачено	Вартість витраченого матеріалу, грн.
Папір	150	1	150
Ручка	30	1	30
Маркер	40	1	40
Флешка	250	1	250
Всього			470
З врахуванням коефіцієнта транспортування			501

5. Програмне забезпечення для наукової роботи включає витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення необхідного для проведення дослідження.

Для написання магістерської роботи використовувалися середовище розробки VS Code, мова програмування Python та бібліотеки ChromaDB, LangChain/LlamaIndex, які є безкоштовними.

6. Амортизація обладнання, комп’ютерів та приміщень, які використовувались під час виконання даного етапу роботи

Дані відрахування розраховують по кожному виду обладнання, приміщенням тощо.

$$A = \frac{Ц * Т}{Т_{кор} * 12}, \quad (4.6)$$

де Ц – балансова вартість даного виду обладнання (приміщень), грн.;

$Т_{кор}$ – час користування;

Т – термін використання обладнання (приміщень), цілі місяці.

Згідно пункта 137.3.3 Податкового кодекса амортизація нараховується на основні засоби вартістю понад 2500 грн. В нашому випадку для написання магістерської роботи використовувався персональний комп'ютер вартістю 30000 грн.

$$А = \frac{30000 \cdot 1}{2 \cdot 12} = 1250 \text{ (грн)}.$$

7. До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

$$В_е = \sum_{i=1}^{n} \frac{W_{yt} \cdot t_i \cdot Ц_е \cdot К_{впі}}{\eta_i}, \qquad (4.7)$$

де W_{yt} – встановлена потужність обладнання на певному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_е$ – вартість 1 кВт-години електроенергії, грн;

$К_{впі}$ – коефіцієнт, що враховує використання потужності, $К_{впі} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

Для написання магістерської роботи використовується персональний комп'ютер для якого розрахуємо витрати на електроенергію.

$$В_е = \frac{0{,}3 \cdot 185 \cdot 9{,}56 \cdot 0{,}5}{0{,}8} = 331{,}61 \text{ (грн)}.$$

Витрати на службові відрядження, витрати на роботи, які виконують сторонні підприємства, установи, організації та інші витрати в нашому досліджені не враховуються оскільки їх не було.

Накладні (загальновиробничі) витрати Внзв охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати Внзв можна прийняти як (100…150)% від суми основної заробітної плати розробників та робітників, які виконували дану МКНР, тобто:

$$В_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (4.8)$$

де $Н_{нзв}$ – норма нарахування за статтею «Інші витрати».

$$В_{нзв} = 38637 \cdot \frac{100}{100\%} = 38637 \text{ (грн)}.$$

Сума всіх попередніх статей витрат дає витрати, які безпосередньо стосуються даного розділу МКНР

$$В = 38637 + 3864 + 9350 + 2860 + 1875 + 840 + 38637 = 96\,063 \text{ (грн)}.$$

Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР здійснюється за формулою:

$$ЗВ = \frac{В}{\eta}, \quad (4.9)$$

де η – коефіцієнт, який характеризує стадію виконання даної НДР.

Оскільки, робота знаходиться на стадії науково-дослідних робіт, то коефіцієнт $\beta = 0{,}9$.

Звідси:

$$ЗВ = \frac{96063}{0,9} = 106737 \text{ (грн)}.$$

4.3. Розрахунок економічної ефективності науково-технічної розробки роботи

У даному підрозділі кількісно спрогнозуємо, яку вигоду, зиск можна отримати у майбутньому від впровадження результатів виконаної наукової роботи. Розрахуємо збільшення чистого прибутку підприємства $\Delta П_i$, для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, за формулою

$$\Delta П_i = \sum_1^n (\Delta Ц_o * N * Ц_o * \Delta N)_i * \lambda * \rho * \left(1 - \frac{\upsilon}{100}\right), \quad (4.10)$$

де $\Delta Ц_0$ – покращення основного оціночного показника від впровадження результатів розробки у даному році.

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

$Ц_o$ – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

λ – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт λ = 0,8333.

ρ – коефіцієнт, який враховує рентабельність продукту. ρ = 0,25;

υ – ставка податку на прибуток. У 2025 році – 18%.

Припустимо, що ціна за програмний продукт зросте на 700 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року на 65 шт., протягом другого року – на 45 шт., протягом третього року на 35 шт. Реалізація продукції до впровадження розробки складала 1 шт., а її ціна до складає 9500 грн. Розрахуємо прибуток, яке отримає підприємство протягом трьох років.

$$\Delta П_1 = [700 \cdot 1 + (9500 + 700) \cdot 65] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 163\,021{,}60 \text{ (грн)}.$$

$$\Delta П_2 = [700 \cdot 1 + (9500 + 700) \cdot (65 + 45)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 275\,717{,}33 \text{ (грн)}.$$

$$\Delta П_3 = [700 \cdot 1 + (9500 + 700) \cdot (65 + 45 + 35)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 363\,413{,}06 \text{ (грн)}.$$

4.4. Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Розрахуємо основні показники, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Розрахуємо величину початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки.

$$PV = k_{\text{інв}} \cdot \text{ЗВ}, \qquad (4.11)$$

де $k_{\text{інв}}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо ($k_{\text{інв}}$ = 2…5).

$$PV = 2 \cdot 106737 \ = 213474 \text{ (грн)}.$$

Розрахуємо абсолютну ефективність вкладених інвестицій $E_{абс}$ згідно наступної формули:

$$\text{Е}_{\text{абс}} = (\text{ПП} - PV), \tag{4.12}$$

де ПП – приведена вартість всіх чистих прибутків, що їх отримає підприємство від реалізації результатів наукової розробки, грн.;

$$\text{ПП} = \sum_1^T \frac{\Delta\text{П}_\text{і}}{(1+\tau)^t}\ , \tag{4.13}$$

де $\Delta\text{П}_\text{і}$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДЦКР, грн.;

Т – період часу, протягом якою виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,2;

t – період часу (в роках).

$$\text{ПП} = \frac{163\ 021{,}60}{(1+0{,}2)^1} + \frac{275\ 717{,}33}{(1+0{,}2)^2} + \frac{363\ 413{,}06}{(1+0{,}2)^3} = 537624{,}48 \text{ (грн)}.$$

$$\text{Е}_{\text{абс}} = (537624{,}48 - 213474) = 324150 \text{ (грн)}.$$

Оскільки $Е_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР може бути доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $Е_в$. Для цього користуються формулою:

$$Е_в = \sqrt[Т_ж]{1 + \frac{Е_{абс}}{PV}} - 1 , \quad (4.14)$$

де $Т_ж$ – життєвий цикл наукової розробки, роки.

$$Е_в = \sqrt[3]{1 + \frac{324150}{213474,}} - 1 = 0,36 = 36\%.$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$\tau = d + f , \quad (4.15)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2025 році в Україні d = (0,14…0,2);

f – показник, що характеризує ризикованість вкладень; зазвичай, величина $f = (0,05…0,1)$.

$$\tau_{min} = 0,18 + 0,5 = 0,23.$$

Так як $Е_в > \tau_{min}$ то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_в}. \quad (4.16)$$

$$T_{ок} = \frac{1}{0,36} = 2,77 \text{ (роки)}.$$

Так як $T_{ок} \leq 3...5$-ти років, то фінансування даної наукової розробки в принципі є доцільним.

4.5 Висновки

Проведено оцінку комерційного потенціалу інформаційної технології гібридного пошуку RAG, а саме системи для автоматизованої обробки запитів та пошуку по базі знань.

Прогнозування витрат на виконання науково-дослідної роботи по кожній з статей витрат складе 96063 грн. Загальна ж величина витрат на виконання та впровадження результатів даної НДР буде складати 106737 грн.

Вкладені інвестиції в даний проект окупляться через 2,77 роки, приведена вартість всіх чистих прибутків, що їх отримає підприємство від реалізації результатів наукової розробки, склала 537 624,48 грн. Абсолютна ефективність інвестицій становить 324 150 грн, а відносна ефективність — 36%, що перевищує мінімальну ставку дисконтування та свідчить про високу економічну доцільність впровадження розробки.

ВИСНОВКИ

У ході виконання магістерської роботи було проведено аналіз технічної документації JAX і PyTorch та визначено оптимальні стратегії її семантичного розбиття для подальшої векторизації. Розроблено повний конвеєр створення векторної бази даних, що включає парсинг, фрагментацію, векторизацію та індексацію в ChromaDB. Виконано оцінювання якості векторного пошуку, яке показало високу точність роботи системи та дозволило визначити чинники, що впливають на релевантність результатів.

Спроєктовано та реалізовано інформаційну технологію інтелектуального чат-асистента на основі RAG, що забезпечує коректні відповіді з опорою на документацію та значно зменшує кількість фактичних помилок порівняно з використанням чистої LLM. Запроваджено гібридний підхід до пошуку, багатофакторний механізм переранжування та адаптивне визначення кількості документів, що включаються в контекст моделі.

Розроблено ефективну backend-архітектуру на FastAPI з асинхронною обробкою запитів, механізмами кешування та повторного пошуку, що забезпечило низьку латентність та стабільну якість відповідей. Результати роботи підтверджують доцільність комплексного підходу до створення RAG-систем для технічної документації та демонструють можливість практичного впровадження створеного рішення. Перспективи подальших досліджень включають використання cross-encoder моделей, інтеграцію користувацького фідбеку та розширення системи на інші домени.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бурлаченко А. В., Жуков С. О., . АРХІТЕКТУРА СЕРВЕРНОЇ СИСТЕМИ ШІ-ПОМІЧНИКА НА ОСНОВІ RAG-ПІДХОДУ ТА ВЕКТОРНОГО ПОШУКУ *Всеукраїнська науково-практична інтернет-конференція «факультету інтелектуальних інформаційних технологій та автоматизації» (Вінниця, 2025-2026 рр.)*. URL: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2026/paper/view/26681 (дата звернення: 08.12.2025).

2. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Küttler H., Lewis M., Yih W.-T., Rocktäschel T., Riedel S., Kiela D. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks arXiv:2005.11401, 2020. URL: https://arxiv.org/abs/2005.11401 (дата звернення: 13.11.2025).

3. LangChain. Build a Retrieval Augmented Generation (RAG) App: Part Документація LangChain. URL: https://python.langchain.com/docs/tutorials/rag/ (дата звернення: 13.11.2025).

4. Google DeepMind. Gemma — models : [Електронний ресурс]. — Офіц. сайт Google DeepMind. URL: https://deepmind.google/models/gemma/ (дата звернення: 13.11.2025).

5. Santu S. K. K., Di Feng. Evaluation of Retrieval-Augmented Generation: A Survey :. — arXiv:2405.07437, 2024. — URL: https://arxiv.org/abs/2405.07437 (дата звернення: 14.11.2025).

6. Hosseini A., Xiao H., McClanahan C. VectorSearch: Enhancing Document Retrieval with Semantic Embeddings and Optimized Search : arXiv:2409.17383, 2024. URL: https://arxiv.org/abs/2409.17383 (дата звернення: 14.11.2025).

7. Malkov Y. A., Yashunin D. A. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs : arXiv:1603.09320, 2016. — URL: https://arxiv.org/abs/1603.09320(дата звернення: 15.11.2025).

8. Dao T., Fu D. Y., Ermon S., Rudra A., Ré C. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness : — arXiv:2205.14135, 2022. URL: https://arxiv.org/abs/2205.14135 (дата звернення: 15.11.2025).

9. Chen L., Wang Y., Shen J., Li F. S2 Chunking: A Hybrid Framework for Document Segmentation Through Integrated Spatial and Semantic Analysis : . — arXiv:2501.05485, 2025. — URL: https://arxiv.org/abs/2501.05485 (дата звернення: 15.11.2025).

10. Qdrant. Qdrant Vector Database Documentation :. — Офіц. документація Qdrant. URL: https://qdrant.tech/documentation/ (дата звернення: 15.11.2025).

11. Qwen. Qwen/Qwen3-Embedding-0.6B — Model Card :// Hugging Face. URL: https://huggingface.co/Qwen/Qwen3-Embedding-0.6B(дата звернення: 16.11.2025).

12. Robertson S., Zaragoza H. The Probabilistic Relevance Framework: BM25 and Beyond : // Foundations and Trends in Information Retrieval. — 2009. — Vol. 3, No. 4. — P. 333-389. — URL: https://www.staff.city.ac.uk/~sbrp622/papers/foundations_bm25_review.pdf (дата звернення: 16.11.2025).

13. Karpukhin V., Oguz B., Min S., Lewis P., Wu L., Edunov S., Chen D., Yih W. Dense Passage Retrieval for Open-Domain Question Answering : — arXiv:2004.04906, 2020. URL: https://arxiv.org/abs/2004.04906(дата звернення: 17.11.2025).

14. Ma X., Gong Y., He P., Zhao H., Duan N. Query Rewriting for Retrieval-Augmented Large Language Models arXiv:2305.14283, 2023. — URL: https://arxiv.org/abs/2305.14283 (дата звернення: 19.11.2025).

15. Wang L., Yang N., Huang X., Jiao B., Yang L., Jiang D., Majumder R., Wei F. Text Embeddings by Weakly-Supervised Contrastive Pre-training :arXiv:2212.03533, 2022. — URL: https://arxiv.org/abs/2212.03533(дата звернення: 19.11.2025).

16. Gao L., Ma X., Lin J., Callan J. Precise Zero-Shot Dense Retrieval without Relevance Labels arXiv:2212.10496, 2023. — URL: https://arxiv.org/abs/2212.10496 (дата звернення: 19.11.2025).

17. Thakur N., Reimers N., Rücklé A., Srivastava A., Gurevych I. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models arXiv:2104.08663, 2021. — URL: https://arxiv.org/abs/2104.08663 (дата звернення: 20.11.2025).

18. Formal T., Lassance C., Piwowarski B., Clinchant S. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. — arXiv:2109.10086, 2021. URL: https://arxiv.org/abs/2109.10086 (дата звернення: 20.11.2025).

19. Lin J., Ma X., Lin S.-C., Yang J.-H., Pradeep R., Nogueira R. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations — arXiv:2102.10073, 2021. URL: https://arxiv.org/abs/2102.10073 (дата звернення: 20.11.2025).

20. Nogueira R., Cho K. Passage Re-ranking with BERT :— arXiv:1901.04085, 2019. — URL: https://arxiv.org/abs/1901.04085(дата звернення: 21.11.2025).

21. Pradeep R., Nogueira R., Lin J. The Expando-Mono-Duo Design Pattern for Text Ranking with Pretrained Sequence-to-Sequence Models : — arXiv:2101.05667, 2021. — URL: https://arxiv.org/abs/2101.05667 (дата звернення: 22.11.2025).

Додаток А

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації

ЗАТВЕРДЖУЮ
Завідувач кафедри САІТ
_____ д.т.н., проф. Віталій МОКІН
«___» _____ 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на магістерську кваліфікаційну роботу
«ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗРОБЛЕННЯ ІНТЕЛЕКТУАЛЬНОГО ЧАТ-АСИСТЕНТА НА БАЗІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ І RAG»
08-34.МКР.002.02.000.ТЗ

Керівник: к.т.н., доц. каф. САІТ
_____ Сергій ЖУКОВ
«___» _____ 2025 р.

Розробив: студент гр. 2ІСТ-24м
_____ Артем БУРЛАЧЕНКО
«___» _____ 2025 р.

Вінниця 2025

1. Підстава для проведення робіт

Підставою для виконання роботи є наказ № __ по ВНТУ від «__» _____ 2025 р., та індивідуальне завдання на МКР, затверджене протоколом № __ засідання кафедри САІТ від «__» _____ 2025 р.

2. Джерела розробки:

- JAX: документація бібліотеки для чисельних обчислень і машинного навчання на Python [Електрон. текст. дані]. – URI: https://github.com/jax-ml/jax/tree/main/docs
- PyTorch: офіційна документація бібліотеки глибинного навчання [Електрон. текст. дані]. – URI: https://github.com/pytorch/pytorch/tree/main/docs

3. Мета і призначення роботи:

Мета роботи є підвищення ефективності пошуку та отримання технічної інформації для розробників систем машинного навчання шляхом створення інтелектуального чат-асистента на основі гібридного пошуку та технології RAG.

4.Вихідні дані для проведення робіт:

Технічна документація PyTorch та JAX.

5.Методи дослідження:

Дослідження існуючих інформаційних технологій, розвідувальний аналіз, розробка UML-діаграм, , мови програмування HTML/CSS/JS та Python.

6. Етапи роботи і терміни їх виконання:

1. Загальна характеристика поставленої задачі ..... ____ – ____
2. Розробка та впровадження інформаційної технології пошуку даних у векторній базі для роботи ШІ-асистента ..... ____ – ____
3. Обробка отриманих відповідей та візуалізація якості семантичного пошуку ..... ____ – ____
4. Побудова та оцінювання сервісу для якісних відповідей на основі поточної документації JAX, PyTorch..... ____ – ____
5. Економічна частина..... ____ – ____
6. Оформлення матеріалів до захисту МКР. ..... ____ – ____

7. Очікувані результати та порядок реалізації:

Очікується створення та впровадження інформаційної технології, що забезпечить можливість надавати релевантні відповіді на основі документацій PyTorch, JAX

8. Вимоги до розробленої документації

Пояснювальна записка оформлена у відповідності до вимог «Методичних вказівок до виконання магістерських кваліфікаційних робіт для студентів спеціальності 126 «Інформаційні системи та технології» (освітня програма «Інформаційні технології аналізу даних та зображень»)

9. Порядок приймання роботи

Публічний захист..... «___» _____ 2025 р.

Початок розробки ..... «___» _____ 2025 р.

Граничні терміни виконання МКР..... «___» _____ 2025 р.

Розробив студент групи 2ІСТ-24м _____ Артем БУРЛАЧЕНКО

Додаток Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: « Інформаційна технологія розроблення інтелектуального чат-асистента на базі великих мовних моделей і RAG»

Тип роботи: магістерська кваліфікаційна робота

Підрозділ: кафедра САІТ, ФІІТА, гр. 2ІСТ-24м

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 1.65 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне):

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

□ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

□ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Віталій МОКІН, зав. каф. САІТ _____
(підпис)

Сергій ЖУКОВ, доц. каф. САІТ _____
(підпис)

Особа, відповідальна за перевірку _____ Сергій ЖУКОВ
(підпис)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Сергій ЖУКОВ, доц. каф. САІТ
(підпис)

Здобувач _____ Артем БУРЛАЧЕНКО

Додаток В

Лістинг програмни

```
import sys
import os
import time
import logging
import matplotlib.pyplot as plt
from pathlib import Path
import yaml
import asyncio

# Add parent directory to path
sys.path.insert(0, str(Path(__file__).parent.parent))

from adapters.chromadb_adapter import ChromaDBAdapter
from adapters.bm25_adapter import BM25Adapter
from services.embedding_service import EmbeddingService
from services.hybrid_search_service import HybridSearchService
from services.reranking_service import RerankingService
from models.schemas import QueryType

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def load_config(config_path="config/config.yaml"):
    # Adjust path if running from scripts dir
    if not os.path.exists(config_path):
        config_path = os.path.join(os.path.dirname(os.path.dirname(__file__)),
config_path)
```

```
    with open(config_path, 'r') as f:
        return yaml.safe_load(f)

def main():
    config = load_config()

    # Initialize components
    logger.info("Initializing components...")

    # OVERRIDE: Use the root chroma_db if it exists, as the user requested "my vector base"
    # and the one in implementation/chroma_db seems empty.
    root_chroma_path = "/home/debbuggerl/cursova/chroma_db"
    if os.path.exists(root_chroma_path):
        chroma_path = root_chroma_path
        logger.info(f"Using vector DB at: {chroma_path}")
    else:
        chroma_path = config['vector_db']['persist_directory']
        logger.info(f"Using configured vector DB at: {chroma_path}")

    embedding_service = EmbeddingService(
        model_name=config['embedding']['model_name'],
        device=config['embedding']['device']
    )

    chromadb = ChromaDBAdapter(
        persist_directory=chroma_path,
        collection_name=config['vector_db']['collection_name']
    )
```

```
bm25 = BM25Adapter()
# Rebuild BM25 index from ChromaDB
logger.info("Building BM25 index...")
all_docs = chromadb.get_all_documents()
if all_docs['ids']:
    bm25.build_index(all_docs['documents'], all_docs['ids'])
else:
    logger.warning("No documents found in ChromaDB. Cannot perform search.")
    return

hybrid_search = HybridSearchService(
    chromadb_adapter=chromadb,
    bm25_adapter=bm25,
    semantic_weight=config['hybrid_search']['semantic_weight'],
    lexical_weight=config['hybrid_search']['lexical_weight']
)

reranking = RerankingService(config['reranking'])

# Query related to chapter1.md content (which discusses RAG paradigms)
query = "Основні парадигми RAG"
logger.info(f"Processing query: {query}")

async def run_search():
    start_time = time.time()

    # 0. Generate Query Embedding
    query_embedding = embedding_service.encode_query(query)
```

```
# 1. Hybrid Search
search_results = await hybrid_search.hybrid_search(
    query=query,
    query_embedding=query_embedding,
    k=10,
    query_type=QueryType.CONCEPTUAL
)

# 2. Reranking
reranked_results = reranking.rerank(
    query=query,
    results=search_results
)

end_time = time.time()
execution_time = end_time - start_time

logger.info(f"Search completed in {execution_time:.4f} seconds")
logger.info(f"Found {len(reranked_results)} results")

# Print results
print("\nTop 5 Results:")
for i, res in enumerate(reranked_results[:5]):
    print(f"{i+1}. Score: {res.combined_score:.4f} | ID: {res.document.id}")
    print(f"   Content: {res.document.content}")
    print("-" * 50)

# Generate Graph
scores = [res.combined_score for res in reranked_results]
ids = [res.document.id for res in reranked_results]
```

```
        plt.figure(figsize=(12, 6))
        # Create a horizontal bar chart for better readability of IDs/Titles if they are
long
        plt.barh(ids, scores, color='skyblue')
        plt.xlabel('Re-ranking Score')
        plt.ylabel('Document Chunk ID')
        plt.title(f'Search Results for query: "{query}"')
        plt.gca().invert_yaxis() # Highest score at top
        plt.tight_layout()

        output_graph = 'search_results_graph.svg'
        # Save to the root directory for visibility
        output_path = f"/home/debbuggerl/cursova/{output_graph}"
        plt.savefig(output_path)
        logger.info(f"Graph saved to {output_path}")

    asyncio.run(run_search())

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

Додаток Г

ІЛЮСТРАТИВНА ЧАСТИНА

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗРОБЛЕННЯ ІНТЕЛЕКТУАЛЬНОГО ЧАТ-АСИСТЕНТА НА БАЗІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ І RAG

Нормоконтроль: к.т.н., доцент

_____ Сергій ЖУКОВ

«___» _____ 2025 р.

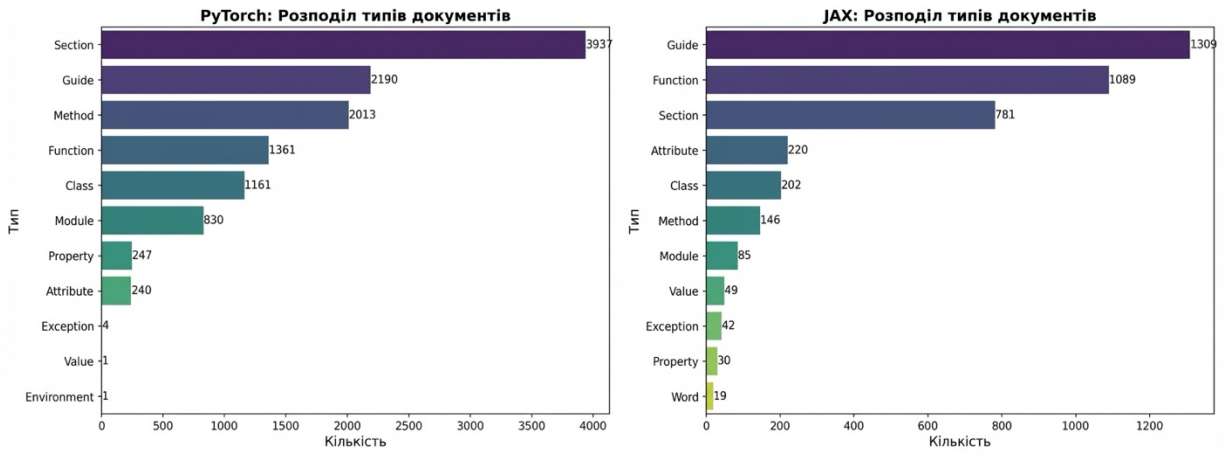

Рисунок Г.1 – Початкові дані

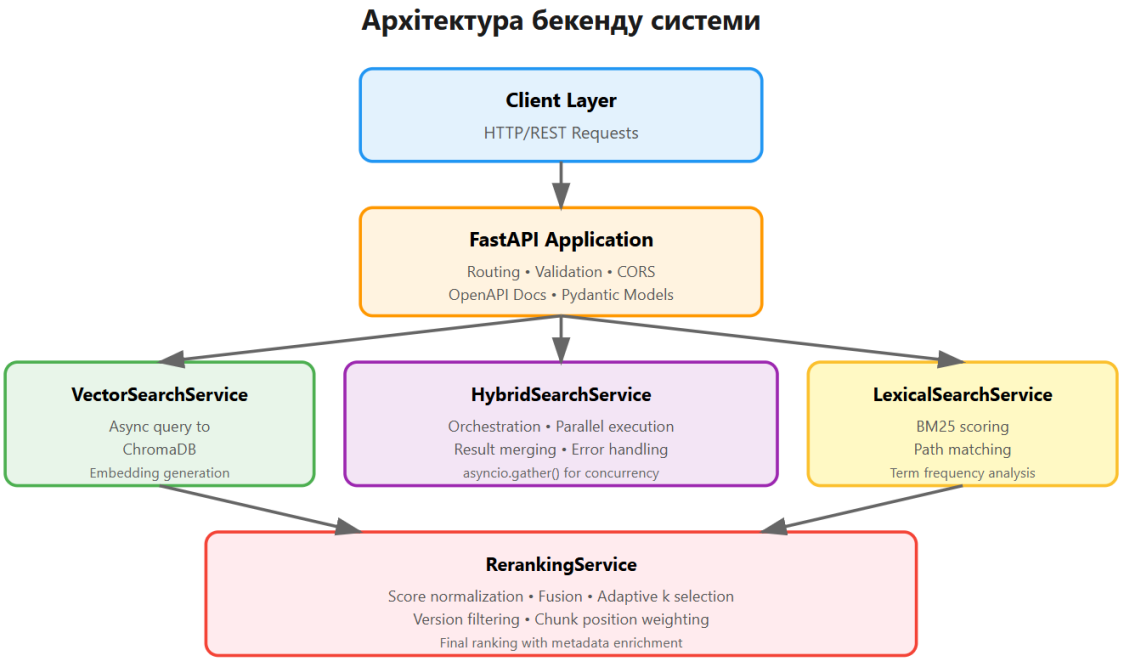

Рисунок Г.2 – Багаторівнева архітектура бекенду з FastAPI та асинхронними сервісами

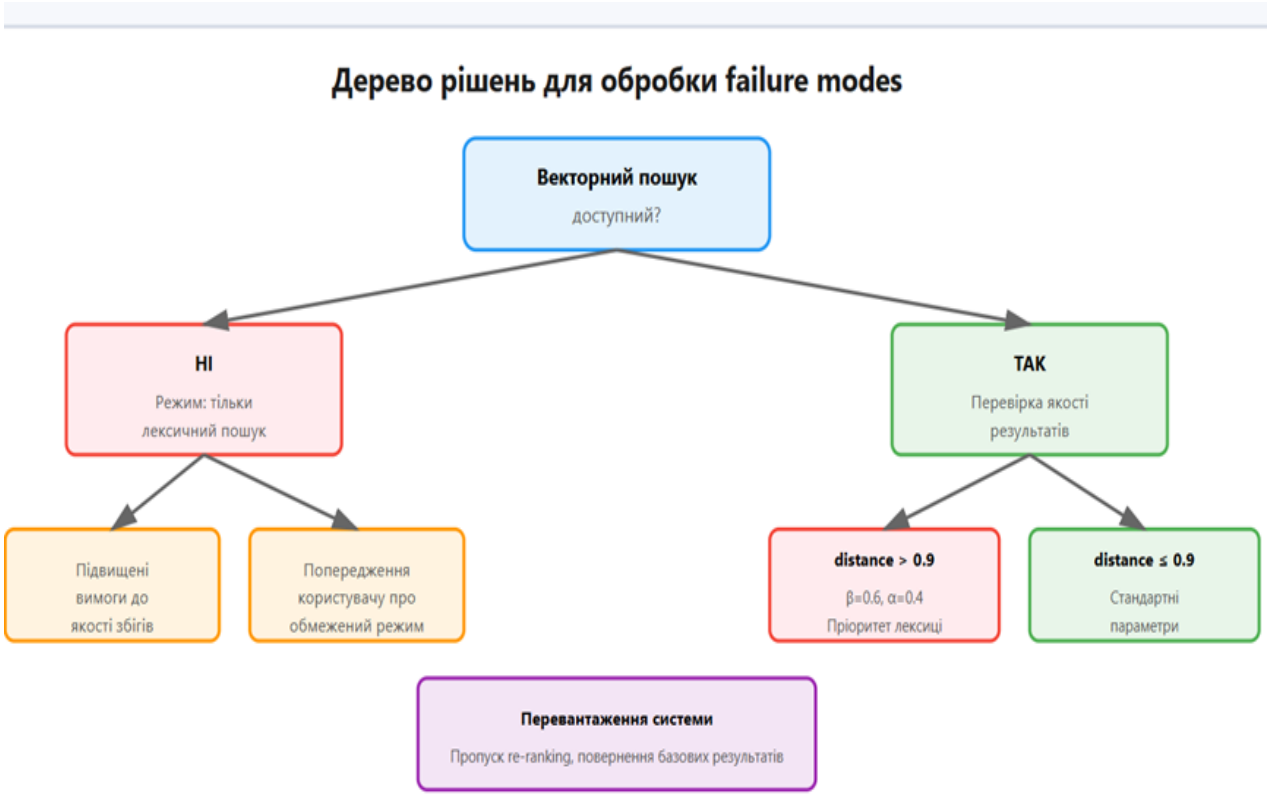

Рисунок Г. 4 – Дерево рішень для обробки відмов та вибору стратегій відновлення

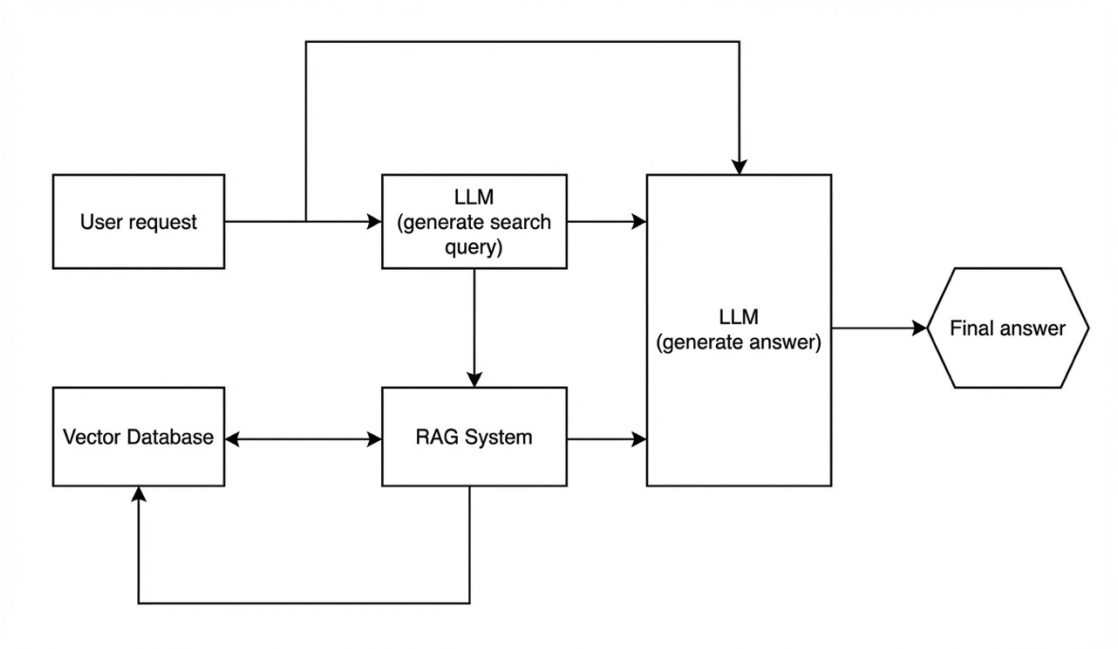

Рисунок Г.4 – Архітектура RAG-системи з агентним підходом

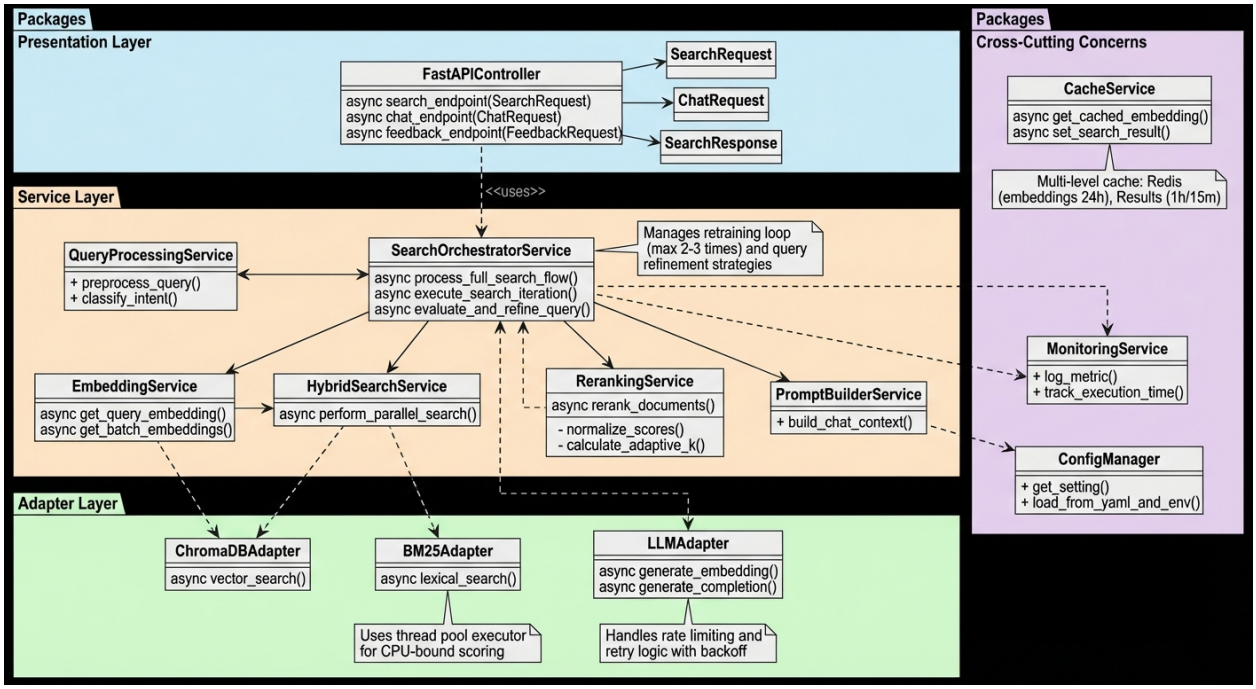

Рисунок Г.5 – Загальна діаграма обробки запитів

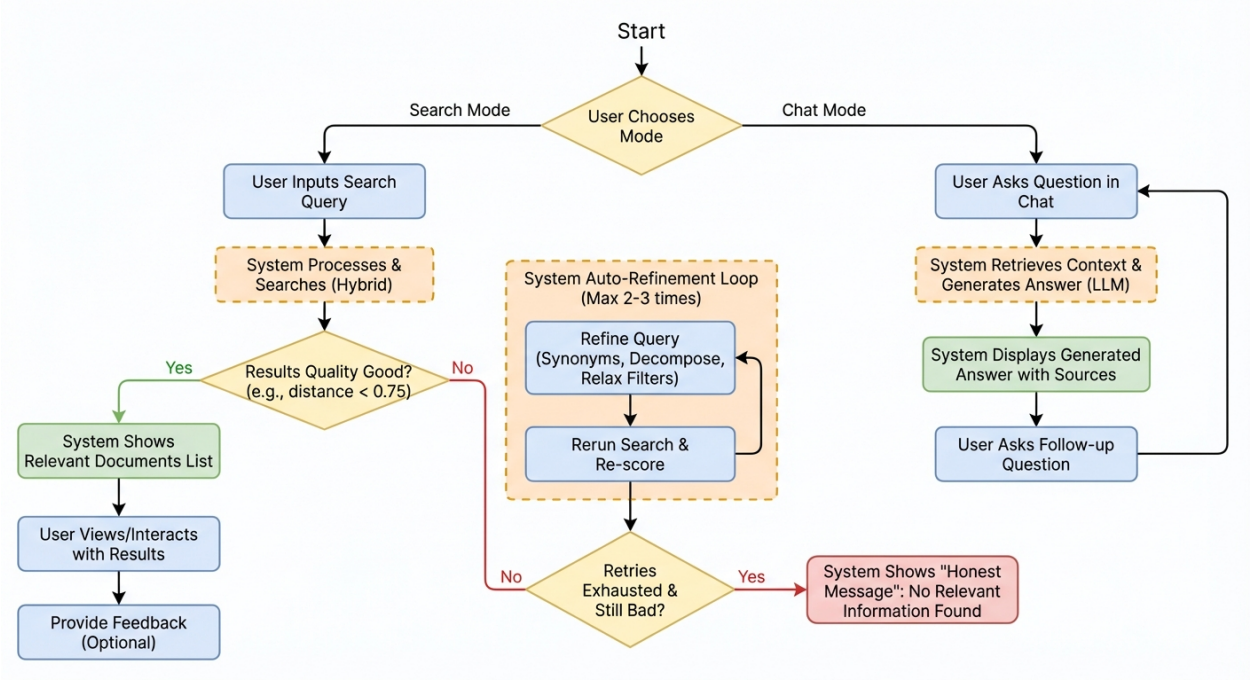

Рисунок Г.6 – Узагальнена діаграма роботи системи