РЕФЕРАТ

ОБРОБКА ТЕКСТОВИХ ДАНИХ, RAG-МЕТОДОЛОГІЯ, ВЕКТОРНА БАЗА ДАНИХ, ГІБРИДНИЙ ПОШУК, ВЕКТОРНІ МОДЕЛІ, ЩІЛЬНІ ВЕКТОРИ (E5, JINA, MINILM, MPNET), РОЗРІДЖЕНІ ВЕКТОРИ (BM25, BM42, MINICOIL), ВЕЛИКІ МОВНІ МОДЕЛІ, LLM, КРОС-МОВНИЙ ПОШУК, АПАРАТНЕ НАВАНТАЖЕННЯ.

Пояснювальна записка: 80 с., 4 табл., 52 рис., 24 джерела.

Метою кваліфікаційної роботи є створення програмного забезпечення на базі архітектури RAG для автоматизованого пошуку, обробки та генерації контекстно-залежних відповідей на основі наданих текстових файлів, а також дослідження впливу векторизаційних моделей та вибору LLM на ресурсоємність, швидкість, релевантність пошуку та точність генерації відповідей.

У ході виконання кваліфікаційної роботи спроєктовано та реалізовано програму мовою Python. Розроблено архітектуру, що включає модулі семантичного розбиття тексту із збереженням метаданих (назви файлу походження, ієрархії заголовків, сторінки), асинхронної взаємодії з векторною базою даних та генерації відповідей через локальну LLM.

Проведено серію експериментальних досліджень. Виконано порівняльний аналіз різних векторизаційних моделей щодо їх здатності знаходити релевантний контекст. Досліджено роботу кількох локальних LLM, за результатами чого оцінено рівень їх ресурсоємності (споживання оперативної пам’яті, навантаження на процесор), швидкодію (час відповіді) та схильність до галюцинацій при генерації відповідей на основі знайденого тексту.

Основні положення кваліфікаційної роботи опубліковано у вигляді тез доповіді на XIX Всеукраїнській науково-практичній WEB конференції аспірантів, студентів та молодих вчених «Комп’ютерні інтелектуальні системи та мережі» КІСМ 2026.

ABSTRACT

TEXT DATA PROCESSING, RAG METHODOLOGY, VECTOR DATABASE, HYBRID SEARCH, VECTOR MODELS, DENSE VECTORS (E5, JINA, MINILM, MPNET), SPARSE VECTORS (BM25, BM42, MINICOIL), LARGE LANGUAGE MODELS, LLM, CROSS-LINGUAL SEARCH, HARDWARE LOAD.

Thesis in 80 p., 4 tab., 52 fig., 24 references.

The goal of the thesis is to create software based on the RAG architecture for automated search, processing and generation of context-aware answers based on the provided text files, as well as to study the impact of vectorization models and LLM selection on resource consumption, speed, relevance of search and accuracy of answer generation.

During the qualification work, a program in Python was designed and implemented. An architecture was developed that includes modules for semantic text splitting with metadata preservation (origin file name, heading hierarchy, page numbers), asynchronous interaction with a vector database and response generation via a local LLM.

A series of experimental studies was conducted. A comparative analysis of different vectorization models was performed regarding their ability to find relevant context. The operation of several local LLMs was studied, based on which the level of their resource consumption (RAM usage, CPU load), speed (response time) and susceptibility to hallucinations when generating answers based on the found text were assessed.

Main ideas of the thesis were published in the XIX All-Ukrainian Scientific and Practical WEB Conference of postgraduates, students, and young scientists “Computer Intelligent Systems and Networks” CISN-2026.

ЗМІСТ

ВСТУП

В умовах стрімкого зростання обсягів цифрової інформації критично важливим завданням для сучасних підприємств стає ефективна обробка та аналіз неструктурованих текстових даних. Сучасні компанії акумулюють величезні масиви внутрішньої документації, інструкцій та регламентів. Класичні методи навігації та пошуку, що базуються виключно на прямому збігу ключових слів, вичерпали свій потенціал. Вони часто виявляються неефективними, оскільки не здатні розуміти семантичний контекст та синонімію у текстах.

Новим етапом у вирішенні завдань обробки природної мови стала поява технологій генеративного штучного інтелекту (ШІ) та великих мовних моделей (Large Language Model, LLM). Вони демонструють великі можливості в аналізі та синтезі тексту. Проте використання популярних публічних хмарних ШІ-рішень у корпоративному секторі стикається з низкою фундаментальних бар'єрів. До них належать неприпустимі ризики витоку конфіденційної комерційної інформації, схильність моделей до генерації правдоподібних, але хибних фактів (через відсутність доступу до внутрішніх знань, а також внаслідок притаманної їм архітектурної проблеми «галюцинацій»), залежність від сторонніх провайдерів та значні фінансові витрати на їх використання.

З огляду на це, виникає гостра необхідність у розробці автономних, повністю локальних систем інтелектуального пошуку. Такі рішення мають поєднувати аналітичну потужність нейромереж із закритими базами знань підприємства без необхідності постійного та ресурсоємного донавчання самих моделей. Система повинна «розуміти» специфіку корпоративних даних і формувати відповіді, спираючись лише на перевірені внутрішні першоджерела.

Саме дослідженню проблематики безпечної інтеграції мовних моделей із локальними масивами документів та розробці програмної системи семантичного пошуку і генерації відповідей присвячена дана кваліфікаційна робота.

1 СУЧАСНИЙ СТАН І АКТУАЛЬНІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1 Критичний аналіз літературних джерел за темою

В умовах стрімкого зростання обсягів цифрової інформації критично важливим завданням для сучасних підприємств стає ефективна обробка та аналіз неструктурованих текстових даних. Сучасні компанії акумулюють величезні масиви внутрішньої документації, інструкцій та регламентів, проте часто стикаються з парадоксом: інформація існує, але доступ до неї надзвичайно ускладнений. Згідно з дослідженнями компанії McKinsey, працівники витрачають близько 1,8 години щодня лише на пошук необхідної інформації замість безпосереднього виконання своїх обов’язків [1].

Класичні методи навігації та пошуку, що базуються виключно на прямому збігу ключових слів, вичерпали свій потенціал. Вони часто виявляються неефективними, оскільки не здатні розуміти семантичний контекст та синонімію у текстах: якщо користувач не вводить точну фразу, передбачену системою, він не отримує релевантного результату. Як наслідок, згідно з дослідженнями, опублікованими у Harvard Business Review, традиційна система корпоративного управління знаннями часто зазнає краху через те, що працівники починають покладатися на колег замість використання документації. Фахівці з найбільшим обсягом знань змушені витрачати значну частину свого робочого часу на відповіді на одні й ті самі рутинні запитання [1].

Новим етапом у вирішенні завдань обробки природної мови стала поява технологій ШІ та великої кількості різних LLM. Вони стали каталізатором переходу до концепції «автоматизації знань» (Knowledge Automation) – підходу, що дозволяє об’єднати існуючі розрізнені джерела інформації підприємства та надавати точні відповіді природною мовою за допомогою LLM, усуваючи

необхідність ручного пошуку інформації. Вони демонструють великі можливості в аналізі та синтезі тексту, здатні розуміти контекст і наміри користувача [1], [2].

Проте використання популярних публічних хмарних ШІ-рішень у корпоративному секторі стикається з низкою фундаментальних бар'єрів. До них належать неприпустимі ризики витоку конфіденційної комерційної інформації, залежність від сторонніх провайдерів, значні фінансові витрати, а також схильність моделей до «галюцинацій» – генерації правдоподібних, але хибних фактів, що виникають через відсутність у базових моделей доступу до внутрішніх знань компанії.

Оптимальним рішенням є використання локальних LLM з RAG (Retrieval Augmented Generation – доповнене пошуком генерування) – це процес отримання контенту для розширення запиту до LLM перед генерацією відповіді. Він використовується для надання моделі доступу до контексту предметної області для вирішення завдання. RAG – це неймовірно цінний інструмент для підвищення точності та узгодженості LLM [3].

Використання RAG гарантує, що система формує відповіді, спираючись виключно на перевірені внутрішні першоджерела компанії, що нівелює ризик фактологічних помилок. Важливість цього підтверджується звітом «2026 State of AI in Knowledge Management», який показує, що 62% співробітників цінують в ШІ-асистентах саме точність відповідей, тоді як швидкість є пріоритетом лише для 26% (повний аналіз опитування наведено на рисунку 1.1) [1].

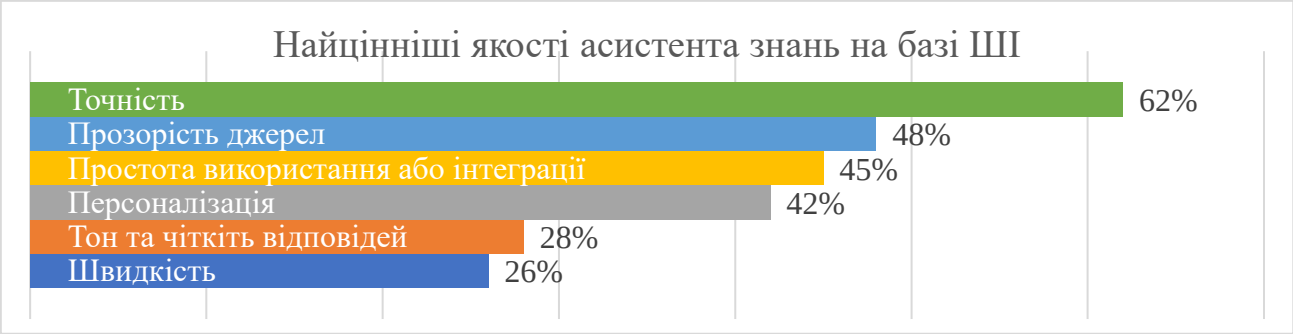

Рисунок 1.1 – Результати опитування користувачів [1]

1.2 Актуальність теми кваліфікаційної роботи

На основі проведеного аналізу літературних джерел можна зробити висновок про високу актуальність розробки автономних, повністю локальних систем інтелектуального пошуку. Безпечна інтеграція локальних LLM та векторних баз даних із закритими корпоративними документами дозволяє уникнути ризиків використання хмарних API. Таким чином, розробка та дослідження програмних засобів автоматизованої обробки текстових даних є своєчасним і науково обґрунтованим завданням, вирішення якого забезпечить ефективну навігацію в корпоративній документації без компромісів у сфері інформаційної безпеки.

Для обґрунтування доцільності розробки власної програмної системи необхідно провести порівняльний аналіз існуючих аналогів, орієнтованих на корпоративне використання. Як еталонні комерційні SaaS-рішення обрано корпоративні версії популярних хмарних систем: ChatGPT (тариф Enterprise/Business) та Gemini (тариф Enterprise/Business). У якості передового відкритого (open-source) рішення розглянуто платформу RAGFlow. Такий підхід дозволяє всебічно зіставити розроблювану архітектуру як із комерційними хмарними продуктами, так і з існуючими безкоштовними локальними системами.

За результати порівняльного аналізу наведеного у таблиці 1.1 видно, що кожен із варіантів має свої переваги та недоліки, але власна розробка містить в собі найбільшу кількість переваг для корпоративного використання.

Таблиця 1.1 – Порівняльний аналіз аналогів [4], [5], [6]

Критерій порівняння	Gemini	ChatGPT	RAGFlow	Розробка
Вартість впровадження	В залежності від підписки та терміну: 6,12 € - 25,30 € за користувача/місяць. Також є персоналізована вартість з підпискою Enterprise.	В залежності від підписки: щорічна/щомісячна відповідно $25/30 за користувача/місяць. Також є персоналізована вартість з підпискою Enterprise.	Безкоштовне ПЗ, але присутні витрати на обладнання	Разове придбання, присутні витрати на обладнання
Модель розгортання	Хмарна (SaaS)	Хмарна (SaaS)	Локальна	Локальна
Конфіденційність даних	Висока (захищено політиками Google Workspace), але дані в хмарі	Висока (дані не йдуть у загальне навчання), але фізично знаходяться на серверах OpenAI	Максимальна (дані на власних серверах)	Максимальна (дані на власних серверах)
Складність розгортання та архітектури	Відсутня (готовий інтерфейс)	Відсутня (готовий інтерфейс)	Дуже висока (потрібно власноруч розбиратися та впроваджувати)	Оптимізована
Гнучкість налаштування логіки та використовуємих технологій	Відсутня (чорний ящик)	Відсутня (чорний ящик)	Абсолютна (білий ящик)	Абсолютна (білий ящик)
Точність відповідей	Висока, але може галюцинувати	Висока, але може галюцинувати	Дуже висока	Дуже висока
Підтримка та обслуговування	Офіційна підтримка 24/7 від Google, автоматичні оновлення	Офіційна підтримка 24/7 від OpenAI, автоматичні оновлення	Підтримка спільноти. Усі проблеми вирішуються власноруч	Індивідуальна / Внутрішня (здійснюється розробником (за контрактом) або штатним IT-відділом підприємства)

1.3 Цілі та завдання кваліфікаційної роботи

Зважаючи на вище зазначене можна сформувати наступну мету кваліфікаційної роботи: розробити систему за RAG методологією та дослідити вплив на її якість, швидкість та ресурсоємність в залежності від векторизаційної та мовної моделей.

Для досягнення мети треба виконати наступні завдання:

1) ознайомитися з проблематикою по галузі;

2) ознайомитися з варіантами програмного рішення даної проблеми;

3) проаналізувати отримані варіанти;

4) розробити прототип своєї системи:

a) розробити та описати діаграму варіантів використання програми;

b) розробити та описати функціональну схему програми;

c) розробити та описати алгоритм роботи програми;

d) розробити та описати діаграму потоків даних;

e) розробити структуру векторної бази даних;

5) реалізувати свою систему;

6) дослідити моделі генерації векторів та LLM: їх вплив на якість відповідей, швидкість та навантаження апаратного забезпечення;

7) зробити висновки.

2 ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ КОМПОНЕНТІВ

2.1 Діаграма варіантів використання системи

На рисунку 2.1 зображена діаграма варіантів використання системи, яка наочно демонструю хто і для чого може використовувати проектуєму програму.

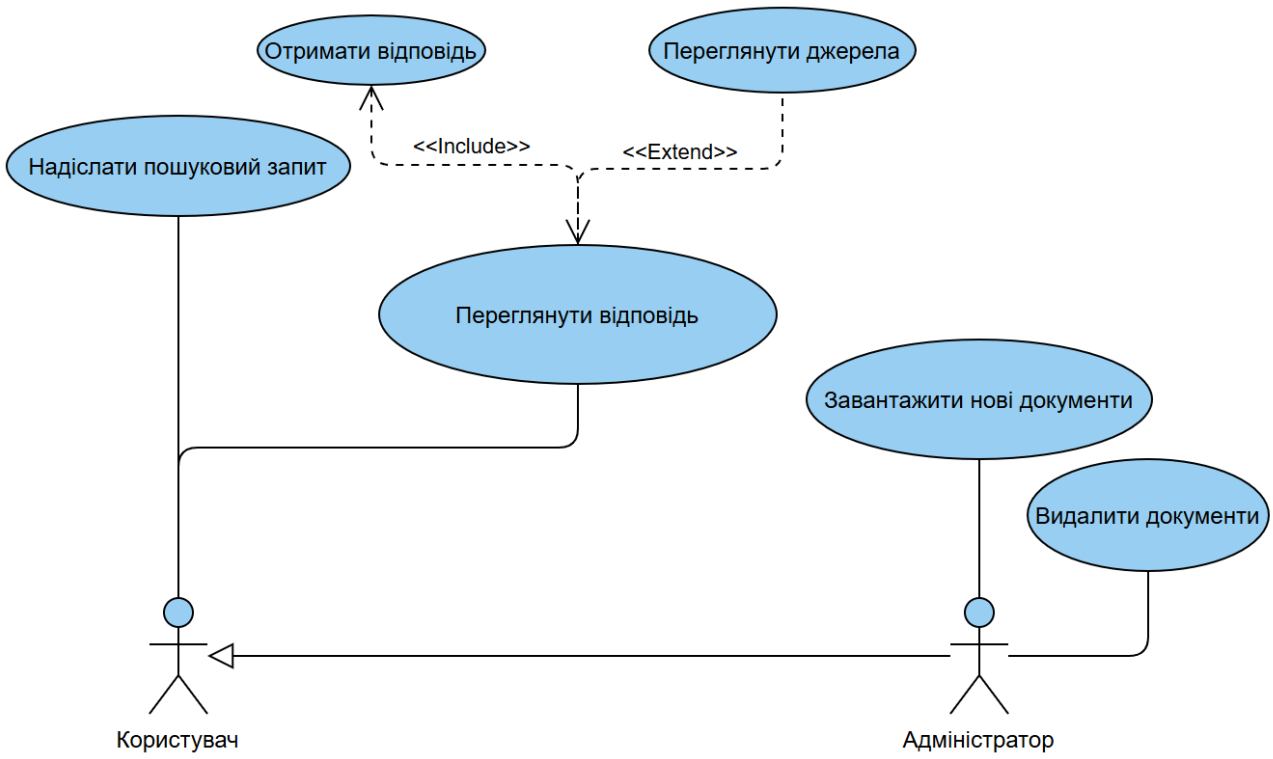

Рисунок 2.1 – Use case діаграма

З діаграми видно, що системою можуть користуватися 2 типи користувачів:

1. користувач: може надсилати пошукові запити до програми та переглядати відповіді, що безпосередньо в себе включає саме отримання відповіді, а також додаткову можливість перегляду джерел, на основі яких надано відповідь;
2. адміністратор: може все те саме, що і звичайний користувач, але ще додатково може керувати документами (завантажувати нові та видаляти старі), на основі яких надаються відповіді.

2.2 Розробка та докладний опис алгоритму роботи програми

Згідно зі статтею від компанії OpenAI всі програми, які є реалізаціями RAG системи, виконують алгоритм зазначений на рисунку 2.2.

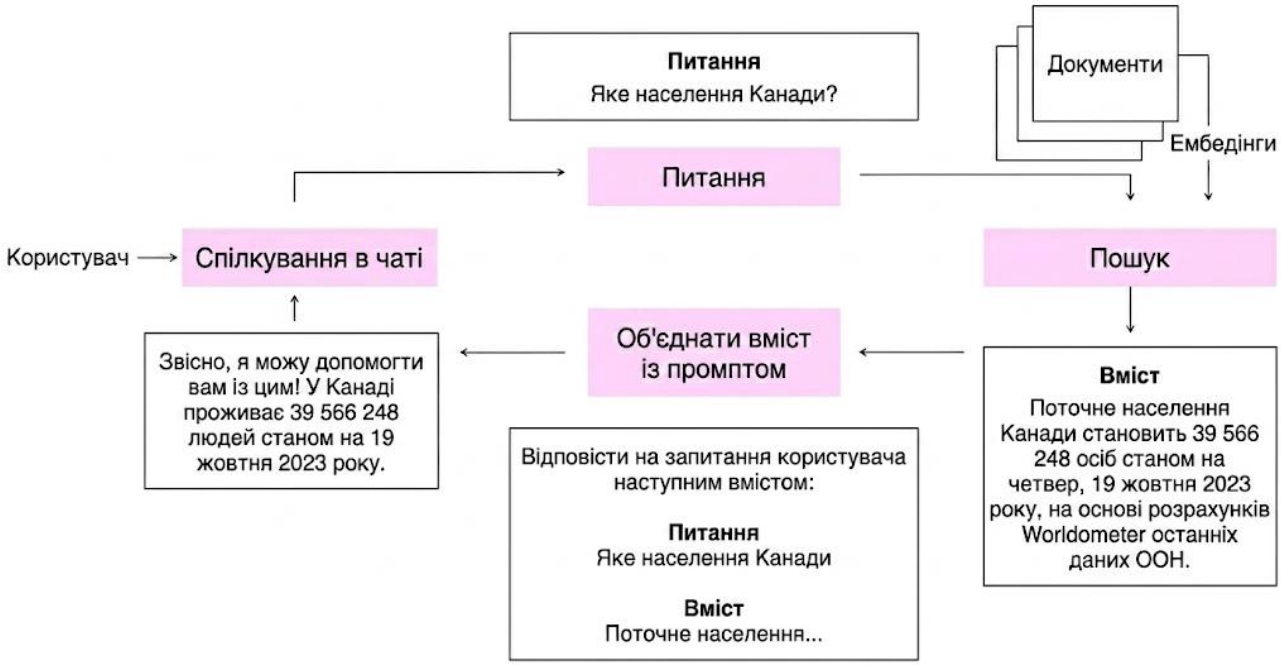

Рисунок 2.2 – Алгоритм роботи RAG систем [3]

Таким чином видно, що в даного типу програмах користувач через чат програми ставить питання, потім виконується пошук по векторній базі даних, результатом якого стає текстовий вміст результуючих ембедінгів (векторного представлення (в даному випадку) тексту), після чого йде об’єднання результату пошуку та питання в промпті (структурований запит) до LLM, відповідь якої вже безпосередньо надається користувачу в тому ж чаті.

2.3 Розробка функціональної схеми програми

На наступних діаграмах (Рисунок 2.3 - Рисунок 2.6) зображено функціональну діаграму програми (розроблену за нотацією IDEF0), яка наочно описує поведінку системи під час роботи програми.

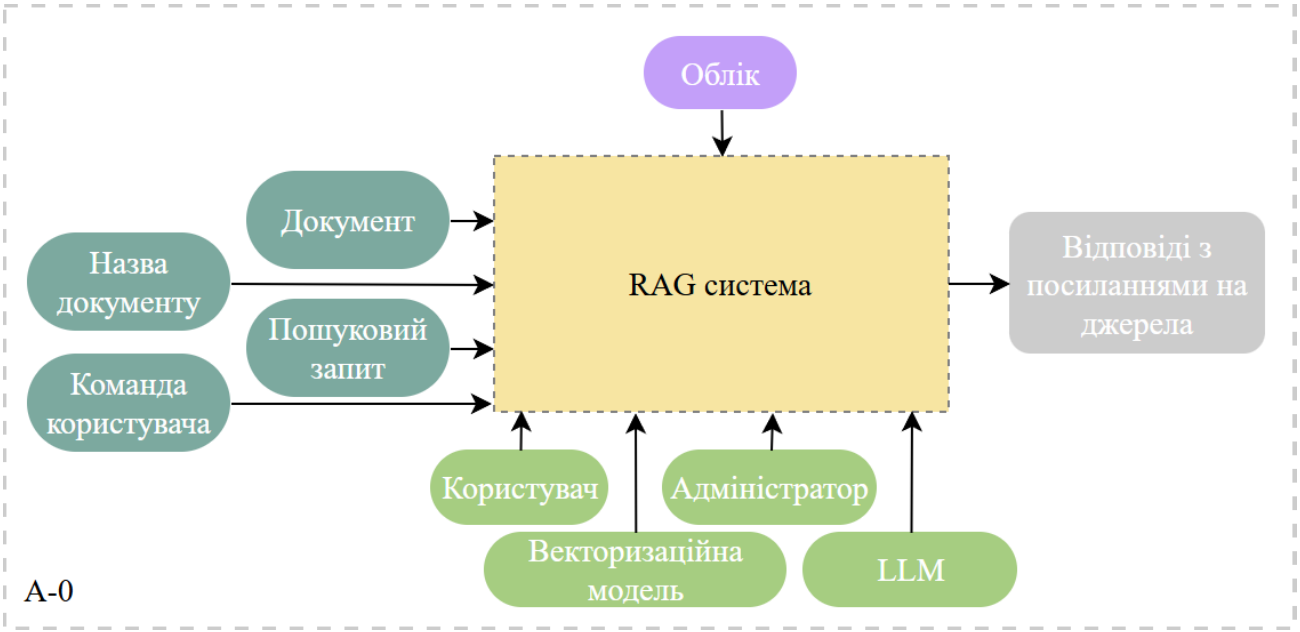

Рисунок 2.3 – Функціональна діаграма програми – Композиція

На контекстній діаграмі (A-0, див. Рисунок 2.3) функціональної моделі система відображається як єдиний головний процес – «RAG система», який взаємодіє із зовнішнім середовищем:

– вхідними даними (Input) є: текстові документи, їхні назви, пошукові запити користувачів, а також керуючі команди;

– вихідними даними (Output) є згенеровані системою відповіді, які обов’язково містять посилання на першоджерела;

– механізмами виконання (Mechanism) виступають як людські ресурси (Користувач, Адміністратор), так і програмні компоненти: векторизаційна модель (для створення ембедингів) та LLM (для генерації тексту-відповіді);

– керуванням (Control) системи є «Облік» (правила логування, збереження цілісності бази або системні політики).

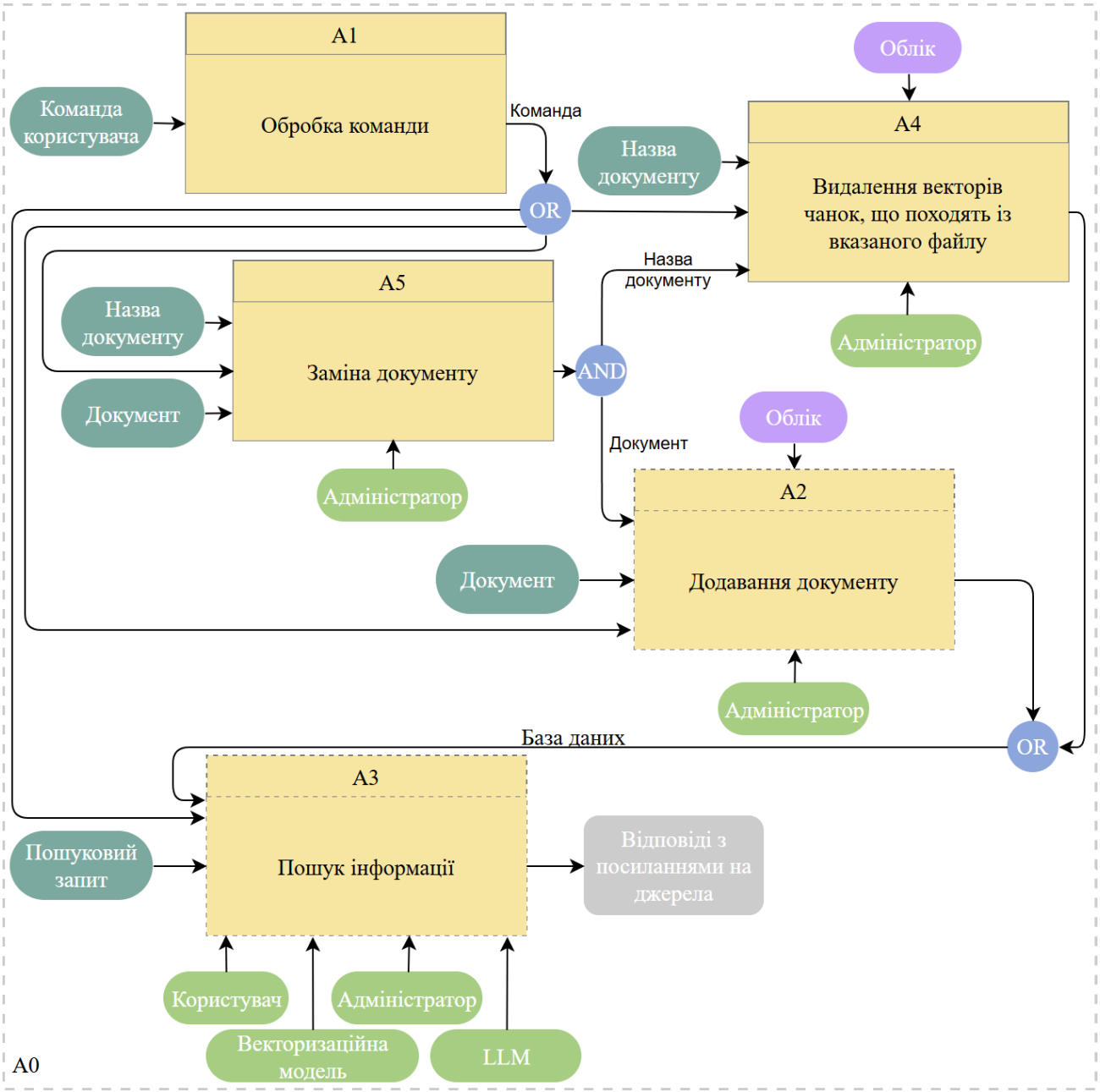

Рисунок 2.4 – Функціональна діаграма програми – Декомпозиція

Для деталізації внутрішньої логіки головний процес було декомпозовано на 5 базових підпроцесів (див. Рисунок 2.4):

– A1 «Обробка команди»: виконує роль маршрутизатора, який приймає команди від користувача/адміністратора та перенаправляє потік управління до відповідного блоку;

– A2 «Додавання документу»: відповідає за первинну обробку та додавання нових файлів у базу знань;

– A3 «Пошук інформації»: ядро системи, що реалізує безпосередньо процеси RAG;

– A4 «Видалення векторів...»: забезпечує очищення бази знань від неактуальної інформації за вказівкою адміністратора;

– A5 «Заміна документу»: реалізує оновлення файлу шляхом послідовного виклику процесів видалення (A4) та додавання нового матеріалу (A2).

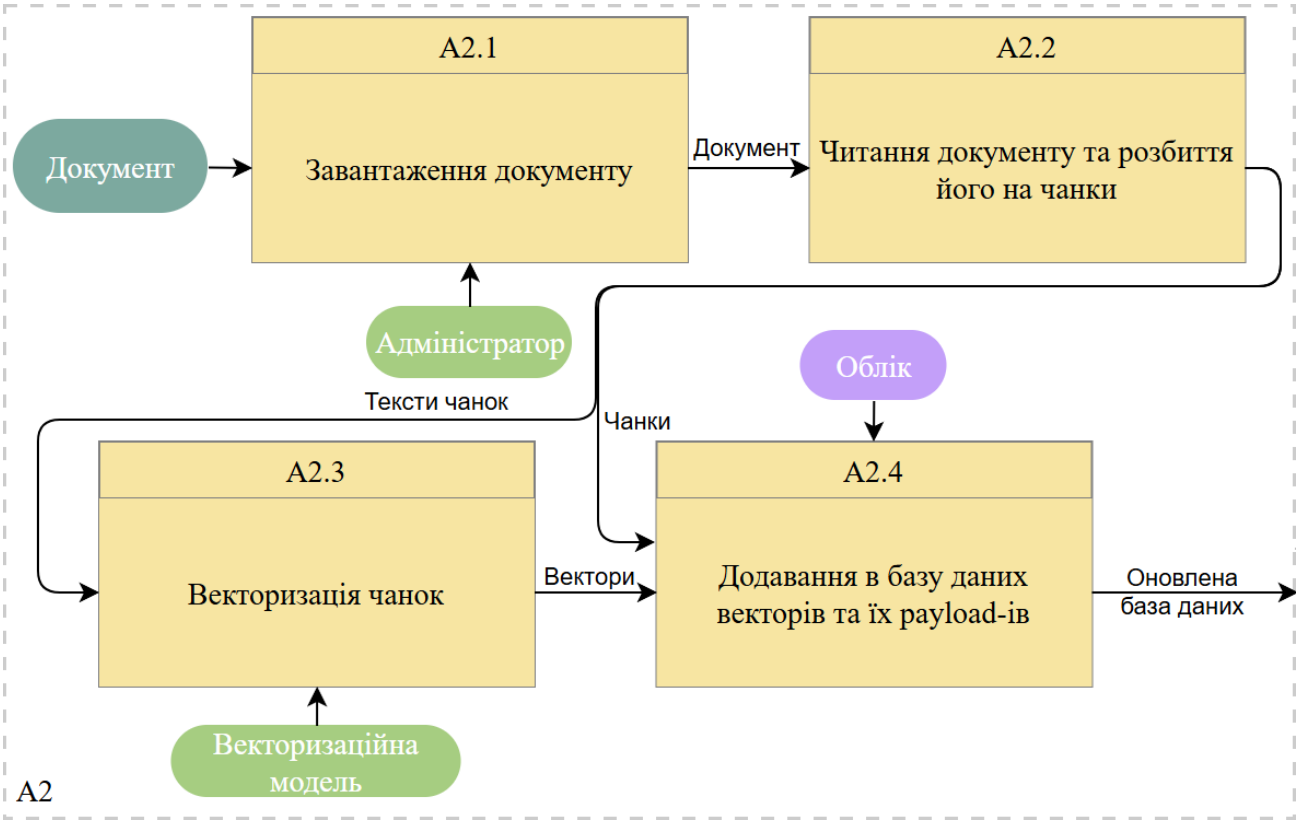

Рисунок 2.5 – Функціональна діаграма програми – Декомпозиція блоку A2

На діаграмі декомпозиції 2-го рівня: Додавання документу (A2, див. Рисунок 2.5) зображено деталізацію процесу підготовки бази знань. Процес має лінійну конвеєрну структуру:

– A2.1 «Завантаження документу»: отримання файлу системою;

– A2.2 «Читання документу та розбиття його на чанки»: синтаксичний аналіз файлу та його сегментація на текстові блоки (чанки), що є критично важливим для якості подальшого пошуку;

– A2.3 «Векторизація чанків»: перетворення чанків у математичні вектори (ембедінги) за допомогою локальної векторизаційної моделі;

– A2.4 «Додавання в базу даних...»: збереження згенерованих векторів разом із їхнім текстовим наповненням та певними метаданими (payload) до векторної бази даних.

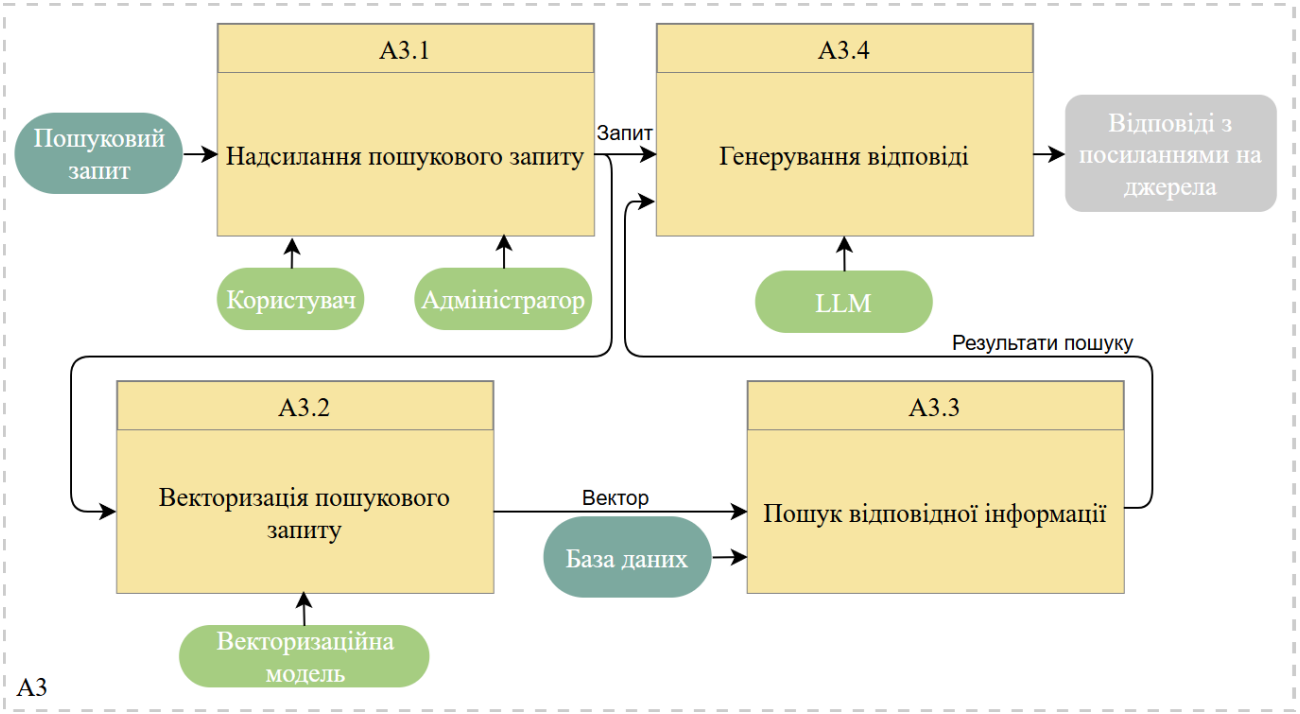

Рисунок 2.6 – Функціональна діаграма програми – Декомпозиція блоку A3

Діаграма декомпозиції 2-го рівня: Пошук інформації (A3, див. Рисунок 2.6) демонструє архітектуру обробки пошукового запиту користувача:

– A3.1 «Надсилання пошукового запиту»: ініціалізація сеансу пошуку;

– A3.2 «Векторизація пошукового запиту»: перетворення запиту у вектор за допомогою тієї ж моделі, що використовувалася для додавання документів (задля збереження спільного семантичного простору);

– A3.3 «Пошук відповідної інформації»: математичне порівняння вектора запиту з векторами бази знань для знаходження найбільш релевантних текстових чанків (контексту);

– A3.4 «Генерування відповіді»: фінальний етап, на якому сирий пошуковий запит та знайдений контекст передаються до локальної LLM, яка формує кінцеву, зв'язну та точну відповідь, яка разом з окремо згрупованими посиланнями на першоджерела (наприклад назви файлів та сторінки, з яких походить контекст) надсилається користувачу.

2.4 Розробка діаграми потоків даних

На наступних діаграмах (Рисунок 2.7 - Рисунок 2.8) зображено діаграму потоків даних (DFD), яка наочно демонструє рух даних по проектованій системі.

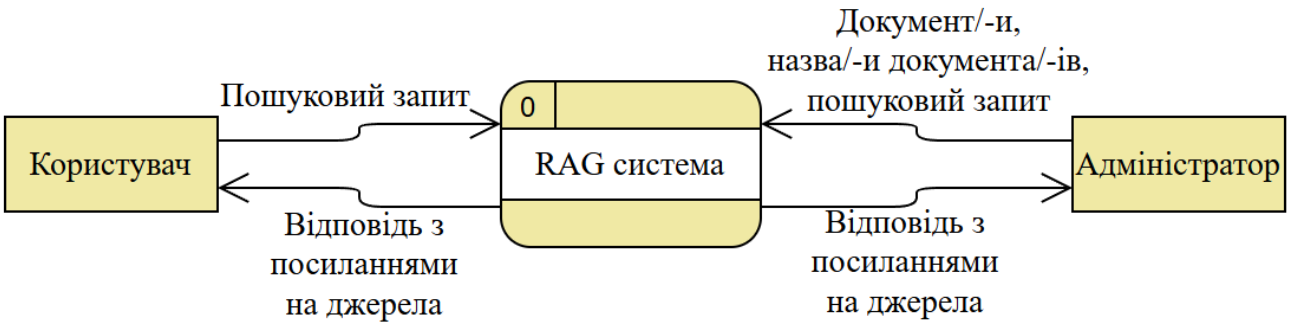

Рисунок 2.7 – Діаграма потоків даних – Композиція

Контекстна діаграма DFD (див. Рисунок 2.7) ілюструє межі розроблюваної програмної системи та її інформаційні зв'язки із зовнішніми сутностями. У системі виділено дві ключові зовнішні сутності: «Користувач» (ініціює пошукові запити та отримує відповіді) та «Адміністратор» (все те саме, що і «Користувач» + здійснює управління інформаційним наповненням системи: передає документи, ініціює їх видалення або оновлення).

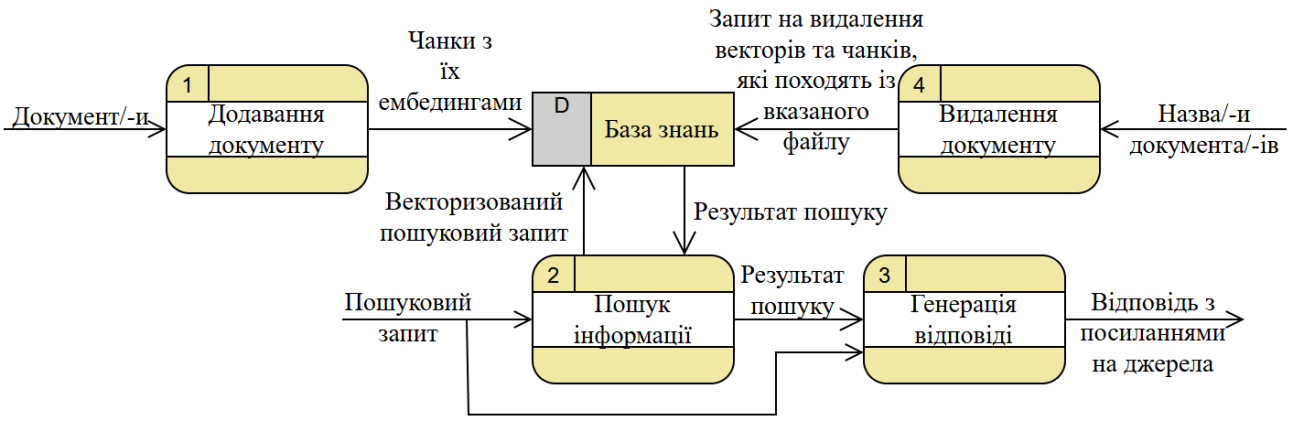

Рисунок 2.8 – Діаграма потоків даних – Декомпозиція

Для розкриття внутрішньої маршрутизації інформації розроблено декомпозицію 1-го рівня DFD (див. рис. Рисунок 2.8). Ключовим елементом даної діаграми є сховище даних «D: База знань» (векторна БД), яке виступає центральним вузлом архітектури. Діаграма описує 4 головні процеси обробки даних:

1) додавання документу: приймає сирі документи, трансформує їх і спрямовує потік підготовлених даних (чанки з їхніми ембедингами) до сховища *D* для постійного зберігання;

2) пошук інформації: ілюструє механізм взаємодії із базою. На вхід надходить сирий текстовий запит. Процес конвертує його у «Векторизований пошуковий запит», здійснює пошук по сховищу *D* і повертає знайдений «Результат пошуку» (релевантний контекст);

3) генерація відповіді: наочно демонструє парадигму RAG-методології. Цей процес агрегує два потоки даних: оригінальний «Пошуковий запит» (від користувача) та «Результат пошуку». На основі цих об’єднаних даних генерується фінальна «Відповідь з посиланнями на джерела»;

4) видалення документу: направляє запит до сховища *D* на знищення конкретних векторів та метаданих, що відповідають зазначеному адміністратором файлу.

2.5 Розробка бази даних

Для даної програми є доречним використання векторних баз даних. Структуру даних для зберігання в сучасних векторних базах даних наведено на рисунку 2.9.

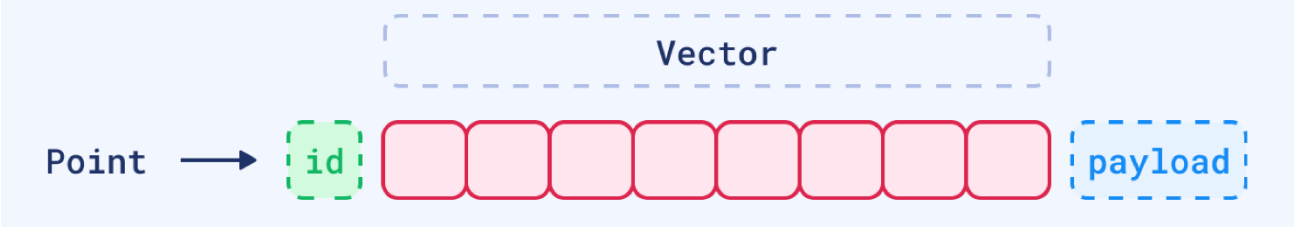

Рисунок 2.9 – Структура векторної бази даних [7]

Точки (points) – це центральна сутність, з якою оперують векторні бази такі як Qdrant. Точка – це запис, що складається з трьох компонентів:

– Унікальний ID (64-бітове без знакове ціле число або UUID), якщо ідентифікатори не надано, клієнт Qdrant автоматично згенерує їх як випадкові UUID;

– вектор (щільний, розріджений або багатовекторний);

– необов'язковий Payload – додаткове корисне навантаження (метадані).

Для даної робити достатньо наступного Payload-у:

– text: str – безпосередня частина тексту (чанк);

– source_file: str – назва файлу, з якого походить даний чанк; за цим полем також йде індексація для пришвидшення пошуку чанків, що відносяться до визначеного файлу, для маніпуляцій з самим файлом (видалення/замінна);

– headings: list[str] – список ієрархія заголовків (розділ -> підрозділ -> …) частини документу, до якої відноситься чанк;

– page_numbers: list[int] – список сторінок документа, на яких розташовується чанк;

– chunk_index: int – індекс чанка, для можливості додаткового отримання попереднього та наступного чанку за потреби.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз обраної середи програмування

Для розробки програми на базі RAG потрібні наступні інструменти:

1) чанекр – інструмент для обробки документів та розбиття тексту на частини (чанки);

2) векторизатор – інструмент для перетворення чанок на вектор;

3) векторизаційна модель (для генерації ембедінгів);

4) векторна база даних (для збереження векторів та їх оригінального вигляду (чанок), та подальшого пошуку потрібних);

5) LLM;

6) інструмент для роботи з LLM;

7) інструмент, за допомогою якого буде реалізована вся робота програми.

3.1.1 Чанкер

Як інструмент для чанкінгу було обрано Docling, який є одним із найнадійніших фреймворків для обробки складних бізнес-документів. Він пропонує високу точність вилучення тексту, високий рівень збереження структури таблиць та ефективну реконструкцію змісту, що поєднується зі стабільною та передбачуваною швидкістю обробки (6,28 с для 1 сторінки і лінійно масштабується до 65,12 с для 50 сторінок). Використання передових моделей, таких як DocLayNet та TableFormer, забезпечує надійну обробку різноманітних елементів документа лише з незначними пропусками [8].

Такий баланс точності, структурної достовірності та продуктивності є критично важливим для RAG-систем. Коректне розпізнавання та збереження структури таблиць і параграфів на етапі чанкінгу дозволяє передавати LLM максимально чистий та семантично зв’язний контекст, що безпосередньо підвищує релевантність та точність кінцевих відповідей системи.

3.1.2 Векторна база даних

Як векторну базу даних було обрано Qdrant. Основними причини вибору Qdrant стали:

– рекомендації спільноти розробників в обговорені на Reddit на тему обрання кращої векторної бази даних [9];

– інтегрується з провідними інструментами та фреймворками ШІ [10];

– підтримує векторизацію текстів за допомогою опціонального розширення FastEmbed [7];

– високопродуктивна векторна пошукова система (збудовано повністю на Rust з SIMD та власним механізмом сховища (Gridstore) – жодних обгорток, жодних доповнень; швидкий, масштабований векторний пошук) [10];

– підтримка гібридного пошуку (підтримує як щільні (dense), так і розріджені (sparse) вектори, що дозволяє комбінувати семантичний пошук із класичним пошуком за ключовими словами, що підвищує точність відповідей у системах RAG) [10];

– підтримка фільтрації та метаданих (дозволяє зберігати разом із векторами довільні метадані (payload) та виконувати пошук із урахуванням фільтрів) [7];

– є повністю open-source рішенням, що дозволяє уникати залежності від сторонніх хмарних сервісів та обмежень на комерційне використання ліцензією [10];

– активно розвивається;

– ефективне використання ресурсів: завдяки вбудованим механізмам квантизації та зберіганню даних на диску, може значно зменшувати споживання оперативної пам’яті [10];

– зрозуміла документація та наявність навчальних курсів від розробників.

Ці характеристики роблять Qdrant оптимальним рішенням для створення системи, що відповідає на запити користувача на основі наданих документів.

3.1.3 Векторизація даних

Щодо векторизації, то тут беззаперечний лідер FastEmbed завдяки своїй інтеграції з Qdrant, що зменшує кількість залежностей та складність розробки, а також рекомендовано до використання самими розробниками Qdrant за наступних умов:

– локальне виконання для чутливих до конфіденційності програм [7];

– висока швидкість роботи на основі процесора (CPU) без важких залежностей, таких як PyTorch [7];

– масштабоване, недороге рішення для генерації векторів (векторизації), тісно інтегроване з Qdrant [7].

І всі перелічені умови ідеально співпадають з реальними поточними потребами. Крім того ще є окрема версія цього розширення, яке оптимізовано для роботи на відеокарті (GPU).

3.1.4 Векторизаційні моделі

Для реалізації ефективного інформаційного пошуку у системі використовується архітектура Universal Query API (API універсального запиту) від Qdrant, яка забезпечує виконання складних пошукових патернів через простий декларативний інтерфейс. Ця архітектура дозволяє уникнути необхідності виконання множинних викликів API, складного злиття результатів на стороні клієнта та надмірної логіки управління – всі операції оптимізовано виконуються на стороні сервера в рамках одного запиту [11].

Основними етапами обробки комплексного запиту в такій архітектурі є:

– попередня вибірка: паралельне виконання декількох пошукових запитів до різних векторних полів. Кожна вибірка може мати власні фільтри, ліміти та використовувати різні типи векторів [11];

– злиття: об’єднання результатів з декількох попередніх вибірок за допомогою алгоритмів, таких як Reciprocal Rank Fusion (RRF) або Distribution-Based Score Fusion (DBSF) [11];

– переранжування: альтернативний кінцевий етап, що передбачає переранжування кандидатів з попередньої вибірки за допомогою більш потужної моделі оцінювання (наприклад, ColBERT) [11];

– фільтрація: застосування фільтрів глобально на рівні запиту (з автоматичним поширенням на всі вибірки) або додавання специфічних фільтрів для створення додаткових обмежень для окремих пошуків [11].

В рамках такого підходу часто поєднують 2 типи пошуку:

1) щільний пошук (dense search): знаходить семантично схожі документи на основі концепцій та загального змісту;

2) розріджений пошук (sparse search): забезпечує точний збіг технічних або специфічних термінів.

3.1.4.1 Щільні векторизаційні моделі

Для реалізації семантичної частини гібридного пошуку було обрано моделі, які підтримуються FastEmbed і є для багатомовного використання. Їх короткий опис наведено у Таблиця 3.1.

Таблиця 3.1 - Опис щільних векторизаційних моделей

Модель	Багатомовність	Усічення вхідних токенів	Префікс для запитів/ документів	Розмір в ГБ	Розмірність	Тип
sentence-transformers/ paraphrase-multilingual-MiniLM-L12-v2	~50	До 512	Не обов’язково	0.22	384	Швидка
sentence-transformers/ paraphrase-multilingual-mpnet-base-v2	~50	До 384	Не обов’язково	1.00	768	Збалансована
intfloat/multilingual-e5-large	~100	До 512	Обов’язково	2.24	1024	Висока продуктивність
jinaai/jina-embeddings-v3	~100	До 1024 і довжина послідовності 8192	Не обов’язково	2.29	1024	Висока продуктивність

3.1.4.2 Розрідженні векторизаційні моделі

Для реалізації другої частини гібридного пошуку – пошуку за точними ключовими словами – базовим рішенням є використання моделі Qdrant/bm25. Вона є реалізацією класичного алгоритму BM25 від Qdrant для генерації розріджених векторів. Дана модель базується на концепції «мішка слів», завдяки чому вона не прив’язана до словника конкретної мови і чудово підходить для багатомовного пошуку [12].

Проте традиційні методи, такі як BM25, мають суттєвий недолік при роботі зі складними запитами – повну відсутність семантичного розуміння. Наприклад, класичний BM25 не здатен відрізнити слово «apple» у запиті «apple charger»(зарядний пристрій від компанії Apple) від того ж слова в «apple cutter»(яблуко-різка), що має зовсім інше значення [12].

Для вирішення цієї проблеми розробники Qdrant створили та зробили відкритими 2 власні моделі для роботи з розрідженими векторами, кожна з яких базується на формулі BM25 [12]:

- BM42 (для роботи через FastEmbed – Qdrant/bm42-all-minilm-l6-v2-attentions) [12];
- miniCOIL (для роботи через FastEmbed – Qdrant/minicoil-v1) [12].

Також розробники надають можливість використання моделі SPLADE, а саме одну з її останніх реалізацій SPLADE++. SPLADE не лише намагається закодувати значення ключових слів, які вже присутні в тексті, він також розширює документи та запити контекстуально відповідними словами [12].

Детальний опис та порівняння моделей наведено у Таблиця 3.2 і Таблиця 3.3.

Таблиця 3.2 - Опис моделей для генерації розріджених векторів [12]

	BM25	miniCOIL	SPLADE
Зіставлення	Лише точний збіг термінів	Лише точний збіг термінів	Точні терміни + розширення синонімами
Підтримка доменів	Висока, статистична модель, працює «з коробки» на будь-якому домені	Від середньої до високої, завдяки неконтрольованом у навчанню на вебданих	Низька, рекомендується донавчання (fine-tuning)
Підтримка мов	Будь-яка (залежить від токенізатора)	Англійська (v1); словник легко розширити для окремих слів	Фіксується на навчальному словнику (зазвичай на базі BERT і англійська)
Невідомі терміни (поза словником)	Зіставляються «як є» (залежить від токенізатора)	Перемикається на BM25 як резервного варіанта	Зіставлюється з [UNK], ймовірно незнаходження відповідності
Інтерпре-тованість	Висока	Від середньої до високої	Нижча без донавчання; механізм розширення може додавати непов’язані токени
Швидкість кодування	Майже миттєва: лише токенізація, без нейронних моделей	Середня: легка модель (lightweight)	Середня - низька: через внутрішні механізми розширення
Щільність розріджених векторів	Розріджені	У ~4 рази менш розріджені, ніж BM25	У ~8+ разів менш розріджені, ніж BM25

Продовження таблиці 3.2

	BM25	miniCOIL	SPLADE
Витрати на пошук	Низькі: компактні розріджені вектори, швидкий пошук за індексом	Середні: менш розріджені вектори вимагають більше обчислень під час пошуку	Високі: розширені вектори є щільнішими, що збільшує вимоги до пам’яті та вартість пошуку
Коли використовувати	Сильне базове рішення; швидке, легке, не потребує налаштувань	Як покращення BM25 із контекстним ранжуванням; усе ще легке зіставлення точних ключових слів	Коли важлива повнота пошуку за синонімами, і наявні ресурси для донавчання (fine-tuning)
Приклад: запит «apple slicer» (яблукорізка)	Оцінює «apple charger» (зарядка Apple) та «apple cutter» (різак для яблук) однаково, оскільки обидва містять слово «apple»	Оцінює «apple cutter» вище, оскільки слово «apple» розуміється в контексті як фрукт, а не як бренд	Може знайти «fruit peeler» (овочечистка для фруктів), навіть без збігу ключових слів, якщо воно охоплюється розширенням

Таблиця 3.3 - Порівняльна таблиця BM25 vs SPLADE vs BM42 [13]

	BM25	SPLADE	BM42
Інтерпретованість	Висока ☑	Нормальна 🆗	Висока ☑
Швидкість обробки документів	Дуже висока ☑	Повільна 🐌	Висока ☑
Швидкість обробки пошукових запитів	Дуже висока ☑	Повільна 🐌	Дуже висока ☑
Використання пам’яті	Низьке ☑	Високе ❌	Низьке ☑
Точність у межах домену	Нормальна 🆗	Висока ☑	Висока ☑
Точність поза межами домену	Нормальна 🆗	Низька ❌	Нормальна 🆗
Точність для малих документів	Низька ❌	Висока ☑	Висока ☑
Точність для великих документів	Висока ☑	Низька ❌	Нормальна 🆗
Обробка невідомих токенів	Так ☑	Погано ❌	Так ☑
Багатомовна підтримка	Так ☑	Ні ❌	Так ☑
Найкращій збіг	Так ☑	Ні ❌	Так ☑

З таблиць видно, що для дослідження варто взяти моделі BM25, BM42 та miniCOIL, бо кожна з них має свої переваги та недоліки. Натомість модель SPLADE не варто брати, бо вона майже у всьому гірша або на рівні, і додатково потребує ресурсоємного донавчання для демонстрації більш-менш гарних результатів.

3.1.5 LLM

Для проведення експериментального дослідження було обрано набір відкритих великих мовних моделей різних виробників та розмірів: від 8 до 32 мільярдів параметрів. Моделі згруповано за трьома класами розміру:

1) 8B – базовий клас, орієнтований на високу швидкість:
   a) llama3-chatqa (8b) [14];
   b) llama3.1 (8b) [15];
   c) deepseek-r1 (8b) [16];
   d) qwen3 (8b) [17];
2) 12-14B – середній клас із кращим балансом якості та ресурсів:
   a) mistral-nemo (12b) [18];
   b) gemma3 (12b) [19];
   c) deepseek-r1 (14b) [16];
   d) qwen3 (14b) [17];
3) 20-32B – високопродуктивний клас, орієнтований на високу якість:
   a) gpt-oss (20b) [20];
   b) mistral-small3.2 (24b) [21];
   c) gemma3 (27b) [19];
   d) deepseek-r1 (32b) [16];
   e) qwen3 (32b) [17].

Для забезпечення всебічного аналізу до вибірки включено моделі від провідних розробників (Meta, Alibaba, DeepSeek, Google, Mistral, OpenAI), які репрезентують три ключові підходи до побудови LLM, спираючись на їхні офіційні архітектурні специфікації:

1) моделі з поглибленим логічним мисленням та агентними можливостями (GPT-OSS від OpenAI, серія R1 від DeepSeek, Qwen3 від Alibaba) – архітектури, що цілеспрямовано спроєктовані для складного логічного аналізу, автономного виконання агентних задач та вирішення комплексних проблем. Навчання таких

моделей орієнтоване на глибоке розуміння контексту та багатомовність, що є критично важливим для аналітичних запитів [20], [16], [17];

2) універсальні та ресурсоефективні архітектури (Llama 3.1 від Meta, Gemma 3 від Google, Mistral Small 3.2 та Mistral-NeMo від Mistral AI) – гнучкі оптимізовані моделі загального призначення з великим контекстним вікном (до 128 тисяч токенів). Вони демонструють високу продуктивність за реальних обчислювальних обмежень (зокрема Mistral-NeMo, створена у співпраці з NVIDIA, використовує високоефективний токенізатор) [15], [19], [21], [18];

3) вузькоспеціалізовані рішення для RAG (Llama3-ChatQA) – моделі, алгоритм навчання яких був спеціально модифікований для вилучення інформації. Зокрема, ChatQA – це модель, донавчена (fine-tuned) дослідниками NVIDIA спеціально для задач Question-Answering та RAG [14].

Таке архітектурне різноманіття дозволяє комплексно дослідити фактори, що впливають на якість генерації у системах RAG. Зокрема, це дасть змогу:

– оцінити кореляцію між масштабом моделі (від 8B до 32B) та точністю вилучення інформації;

– порівняти ефективність універсальних базових моделей зі спеціалізованими RAG-рішеннями;

– перевірити на практиці, наскільки архітектурний фокус на поглиблене логічне мислення та агентні можливості покращує здатність моделі аналізувати складний або об’ємний контекст.

Таким чином, обрана вибірка формує репрезентативну експериментальну базу для оцінки сучасних відкритих LLM у задачах RAG.

3.1.6 Інструмент для взаємодії з LLM

Як інструмент взаємодії з LLM найкращим варіантом для локального тестування є Ollama. Також для зручного подальшого масштабування буде використана її додаткова можливість – OpenAI-сумісна API, бо сама по собі Ollama ідеально підходить для одноосібного користування, що ідеально для дослідження, але для готового продукту для багатокористувацького використання вже буде недуже, і ось тут якраз і спрацює OpenAI-сумісна API адже можна буде замінити лише посилання хосту та ключ API для роботи з тою ж vLLM, яка теж має OpenAI-сумісну API, але ідеально підходить для багатокористувацького використання і розміщення на сервері [22].

3.1.7 Інструмент для взаємодії з програмою

Для оформлення програми в готовий продукт буде використано FastAPI для перетворення на повноцінний back-end застосунок. Вибір на ньому зупинився завдяки його простоті, підтримці асинхронного використання, швидкій роботі, автоматичній генерації документації API, а також наявності попереднього досвіду роботи з ним.

Для наочної демонстрації роботи фінального результату, розроблений back-end буде підключено до Telegram бота.

3.1.8 Підсумок використовуємих технологій

Таким чином стек використовуємих технологій виглядає наступним чином:

– Docling;

– Qdrant + FastEmbed;

– Ollama + OpenAI-compatible API;

– FastAPI + Telegram bot.

3.2 Програмна реалізація основних функцій

Архітектура системи побудована на взаємодії трьох основних компонентів, які забезпечують повний цикл обробки інформації: підготовка вхідних документів -> пошук -> генерація відповіді на запити за RAG методологією:

1) модуль обробки та фрагментації текстів (клас Chunker):
   – клас Chunker відповідає за первинну обробку вхідних документів та їх розбиття на чанки для подальшої векторизації;
   – технологічний стек: бібліотека docling для конвертації документів та transformers (HuggingFace) для токенізації тексту;
   – особливості реалізації: модуль використовує гібридний підхід до розбиття тексту (HybridChunker), який враховує структуру документа;
   – основний функціонал: метод chunk_document приймає джерело даних, конвертує його та повертає ітерований об’єкт із фрагментами тексту та їхніми метаданими, які необхідні для збереження контексту джерела;
   – програмна реалізація модуля:

```
from docling.document_converter import DocumentConverter
from docling_core.transforms.chunker.tokenizer.huggingface import
HuggingFaceTokenizer
from transformers import AutoTokenizer
from docling.chunking import HybridChunker
from docling.datamodel.base_models import DocumentStream
from pathlib import Path


class Chunker:
    def __init__(self, EMBED_MODEL_ID="sentence-
transformers/paraphrase-multilingual-MiniLM-L12-v2",
MAX_TOKENS=256):
        tokenizer = HuggingFaceTokenizer(tokenizer =
AutoTokenizer.from_pretrained(EMBED_MODEL_ID), max_tokens =
MAX_TOKENS)
        self.chunker = HybridChunker(tokenizer = tokenizer,
merge_peers = True)

    def chunk_document(self, source: Path | str | DocumentStream):
        doc = DocumentConverter().convert(source=source).document
        return self.chunker.chunk(dl_doc=doc)
```

2) модуль взаємодії з векторною базою даних (клас AsyncVectorDB):

– клас AsyncVectorDB є асинхронним клієнтом для взаємодії з векторною системою управління базами даних Qdrant. Він забезпечує збереження, оновлення та ефективний семантичний пошук інформації;

– технологічний стек: бібліотека qdrant_client, моделі даних pydantic;

– особливості реалізації: клас реалізує механізм гібридного пошуку. Він одночасно використовує щільні вектори (Dense vectors) та розріджені вектори (Sparse vectors) для підвищення точності результатів. Для об’єднання результатів обох типів пошуку застосовується алгоритм взаємного злиття рангів (Reciprocal Rank Fusion, RRF);

– основний функціонал: Ініціалізація та налаштування колекцій із заданими моделями (create). Асинхронне додавання фрагментів разом із розширеними метаданими (текст, назва файлу, заголовки, номери сторінок) через метод add_chunks. Виконання пошукових запитів (query) та керування даними конкретних файлів (update_file, delete_file);

– програмна реалізація модуля:

```
from qdrant_client import AsyncQdrantClient, models
from pydantic import BaseModel
from typing import Iterable

class QdrantPayload(BaseModel):
    text: str
    source_file: str
    headings: list[str]
    page_numbers: list[int]
    file_chunk_index: int | None = None

class AsyncVectorDB:
    def __init__(self, url: str,: int, gpr_port: int,
collection_name: str, model_name: str, sparse_model_name: str):
        self.collection_name = collection_name
        self.client = AsyncQdrantClient(url=url, port=port,
grpc_port=gpr_port, prefer_grpc=True)
        self.client.set_model(model_name)
        self.client.set_sparse_model(sparse_model_name)
```

```
    @classmethod
    async def create(cls, url: str = "http://localhost",port: int =
6333, gpr_port: int = 6334, collection_name: str = "collection",
model_name: str = "sentence-transformers/paraphrase-multilingual-
MiniLM-L12-v2", sparse_model_name: str = "Qdrant/bm25"):
        self=cls(url, port, gpr_port, collection_name, model_name,
sparse_model_name)

        if not await
self.client.collection_exists(self.collection_name):
            await self.client.create_collection(collection_name =
self.collection_name, vectors_config =
self.client.get_fastembed_vector_params(),
sparse_vectors_config=self.client.get_fastembed_sparse_vector_param
s(modifier=models.Modifier.IDF))
            await self.client.create_payload_index(collection_name=
self.collection_name, field_name="source_file", field_schema=
models.PayloadSchemaType.KEYWORD)
        return self

    async def add_chunks(self, chunks: Iterable[QdrantPayload]):
        documents, metadata = [], []
        for i, chunk in enumerate(chunks):
            chunk.file_chunk_index = i
            documents.append(chunk.text)
            metadata.append(chunk.model_dump(exclude={"text"}))
        await self.client.add(collection_name=self.collection_name,
documents=documents, metadata=metadata)

    async def query(self, query_text: str, limit: int = 10,
prefetch_limit_coeff: int = 2):
        results = await self.client.query_points(collection_name=
self.collection_name, prefetch=[
                models.Prefetch(query = models.Document(text =
query_text, model=self.client.sparse_embedding_model_name),
using=self.client.get_sparse_vector_field_name(),
limit=limit*prefetch_limit_coeff),
                models.Prefetch(query = models.Document(text =
query_text, model=self.client.embedding_model_name),
using=self.client.get_vector_field_name(),limit =
limit*prefetch_limit_coeff)
            ], query=models.RrfQuery(rrf=models.Rrf()),limit=limit)
        return results.points

    async def update_file(self, filename: str, new_chunks:
Iterable[QdrantPayload]):
        await self.delete_file(filename)
        await self.add_chunks(new_chunks)
```

```
    async def delete_file(self, filename: str):
        await self.client.delete(collection_name=
self.collection_name, points_selector = models.Filter(must =
[models.FieldCondition(key="source_file", match =
models.MatchValue(value = filename))]))
```

3) модуль генерації відповідей (клас AsyncLLM):

– клас AsyncLLM відповідає за формування фінальної відповіді на запит користувача на основі знайденого релевантного контексту, використовуючи велику мовну модель (LLM);

– технологічний стек: асинхронний клієнт openai для підключення до локального сервера генерації (Ollama);

– особливості реалізації: модуль формує спеціалізований структурований промпт (підказку) для мовної моделі. Системні інструкції жорстко обмежують модель: вона повинна відповідати мовою запиту, опиратися виключно на наданий контекст, не використовувати власні знання та повідомляти про відсутність інформації, якщо контекст не містить відповіді. Температура генерації встановлена на 0, бо креативність у відповідях непотрібна.

– основний функціонал: метод generate приймає текстовий запит і список релевантних фрагментів (ContextChunk), форматує їх у єдиний блок контексту та асинхронно повертає згенеровану відповідь;

– програмна реалізація модуля:

```
from openai import AsyncOpenAI
from pydantic import BaseModel


class ContextChunk(BaseModel):
    text: str
    source_file: str
    headings: list[str]
    page_numbers: list[int]
    file_chunk_index: int
```

```
class AsyncLLM:
    def __init__(self, base_url: str = "http://localhost:11434/v1",
model_name: str = "llama3-chatqa:8b", api_key: str = "ollama"):
        self.client=AsyncOpenAI(base_url=base_url, api_key=api_key)
        self.model_name = model_name

    async def generate(self, query: str, context_chunks:
list[ContextChunk]) -> str:
        context_parts = [f"[{i+1}] {' > '.join(chunk.headings)}\n
{chunk.text}" for i, chunk in enumerate(context_chunks)]
        context = "\n\n".join(context_parts)
        response = await self.client.chat.completions.create(
            model=self.model_name,
            messages=[
                {"role": "system",
                 "content": """You are a helpful assistant.
Your ONLY task is to answer the user's query based EXACTLY on the
provided context.

CORE RULES:
1. Language: Answer in EXACTLY the same language as the query text.
2. Grounding:
    - DO NOT use your internal knowledge.
    - DO NOT add any information that is not present in the
context.
    - ONLY use the text provided inside the context.
    - Be factual and avoid fluff.
3. Fallback: If the context is "Not found" or does not contain the
answer, say that you don't know."""},
                {"role": "user",
                 "content": f"""Use the following context to answer
the question.

=== CONTEXT START ===
{context if context else "Not found"}
=== CONTEXT END ===

=== QUERY START ===
{query}
=== QUERY END ==="""}
            ], temperature=0.0)
        return response.choices[0].message.content

    async def close(self):
        await self.client.close()
```

4 ДОСЛІДЖЕННЯ ЯКОСТІ ТА ЕФЕКТИВНОСТІ МЕТОДІВ

Проведення досліджень виконувалося майже повністю локально на ноутбуці ASUS Zenbook S 16 з процесором AMD Ryzen Al 9 HX 370 w, інтегрованою відеокартою Radeon 890M та оперативною пам’яттю на 32ГБ, ОС: Windows 11 Pro версії 25H2.

Дослідження проводилися в декілька етапів:

1) дослідження швидкості векторизації і додавання чанків у векторну БД;

2) дослідження точності пошуку;

3) дослідження якості генерації відповіді.

Під час кожного з етапів дослідження також аналізувалося апаратне навантаження (навантаження на процесор та використання оперативної пам’яті).

Перші 2 етапи дослідження були організовані наступним чином: спочатку вони обидва проводилися на всіх обраних моделях генерації щільних (dense) векторів + базова модель генерації розріджених (sparse) векторів – «BM25», після чого була обрана найкраща модель генерації щільних векторів і разом із моделями генерації розріджених векторів, що лишилися («BM42» та «MINICOIL») було проведено повторно перші 2 етапи досліджень. Це дозволило значно зекономити час на дослідженнях, адже не довелося досліджувати всі комбінації моделей генерації щільних та розріджених векторів.

Для спрощення позначення моделей генерації щільних векторів було запроваджено наступні скорочення:

– sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 -> MINILM;

– sentence-transformers/paraphrase-multilingual-mpnet-base-v2 -> MPNET;

– intfloat/multilingual-e5-large -> E5;

– jinaai/jina-embeddings-v3 -> JINA.

4.1 Дослідження швидкості векторизації і додавання чанків у векторну БД: моделі генерації щільних векторів + BM25

Даний етап дослідження проводився на 2-х наборах даних: оригінальному англомовному та його україномовному перекладі. Вони мали незначну розбіжність у кількості екземплярів, що зумовлено специфікою початкового датасету [23], який містив дублікати. Під час створення україномовної версії деякі з цих повторень були перекладені з використанням різних синонімів, що призвело до зміни загальної кількості унікальних записів. Оскільки наступним етапом була оцінка якості пошуку, обидва набори даних попередньо було очищено від дублікатів. Це дозволило уникнути повторень серед вибірки результатів пошуку. Також з метою оптимізації обчислювальних ресурсів та часу на проведення експериментів, з початкового датасету було сформовано випадкову вибірку обсягом 5000 записів (кожен запис містить еквівалентні дані обома мовами).

Результати проведеного дослідження наведено на рисунках 3.1 - 3.4.

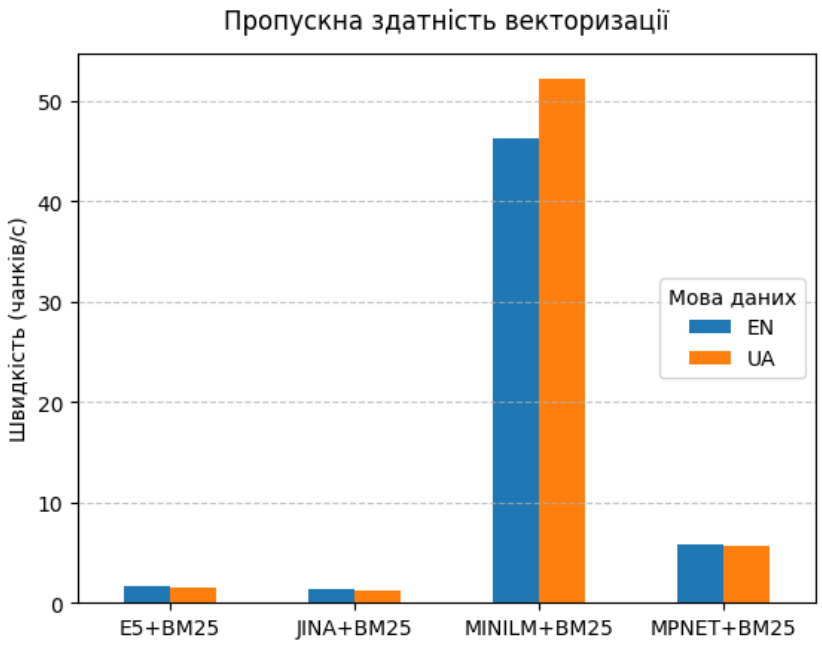

Рисунок 4.1 - Швидкості векторизації чанків моделями генерації щільних векторів та BM25

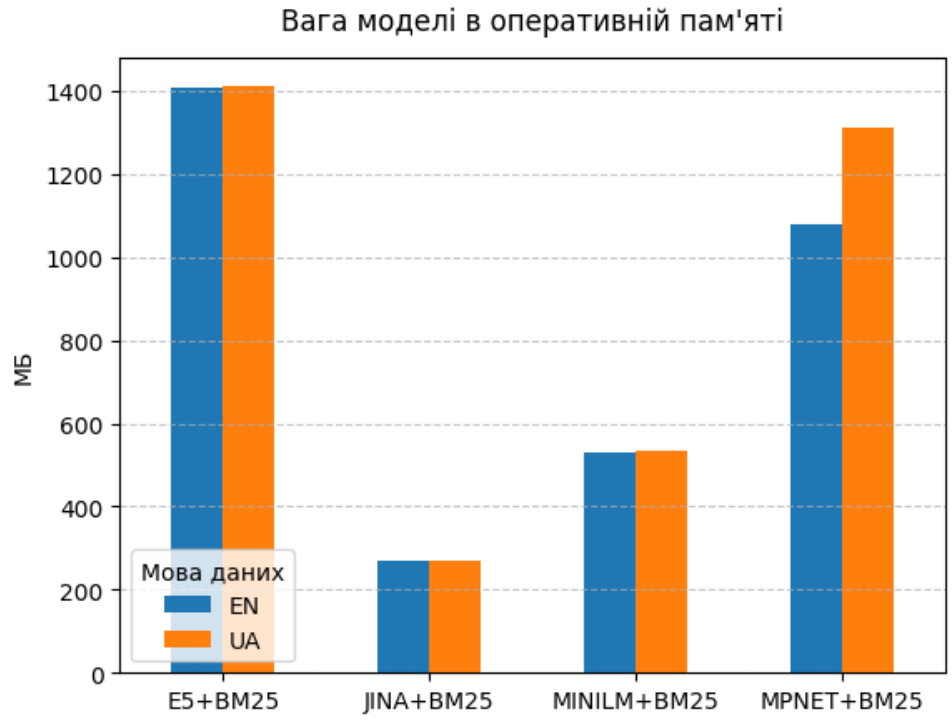

Рисунок 4.2 - Вага в оперативній пам’яті моделей генерації щільних векторів та BM25

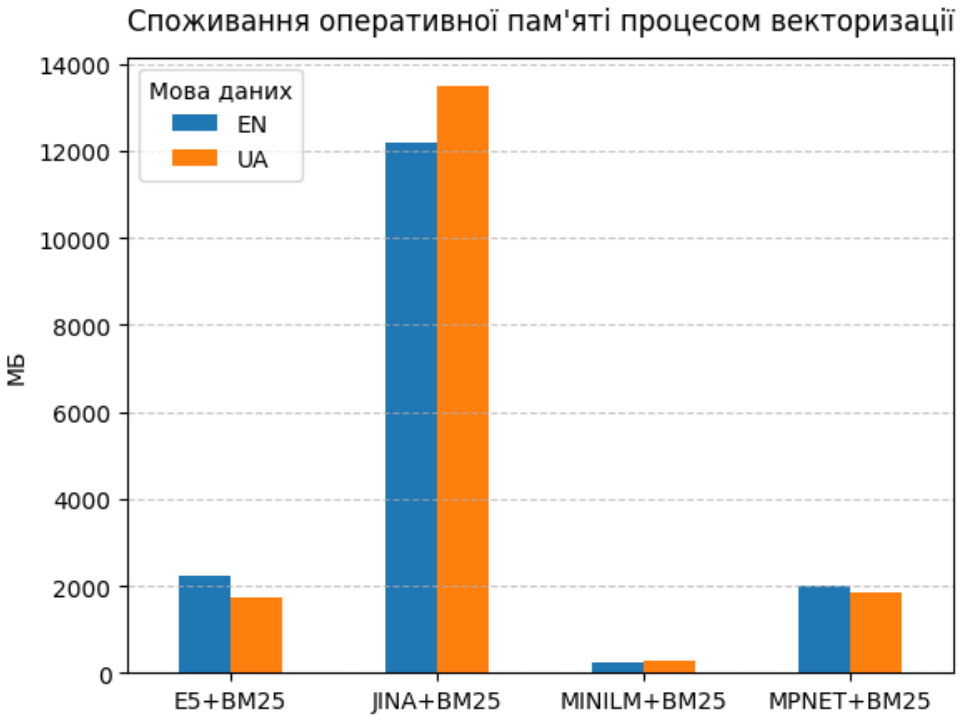

Рисунок 4.3 - Споживання оперативної пам’яті процесом векторизації моделями генерації щільних векторів та BM25

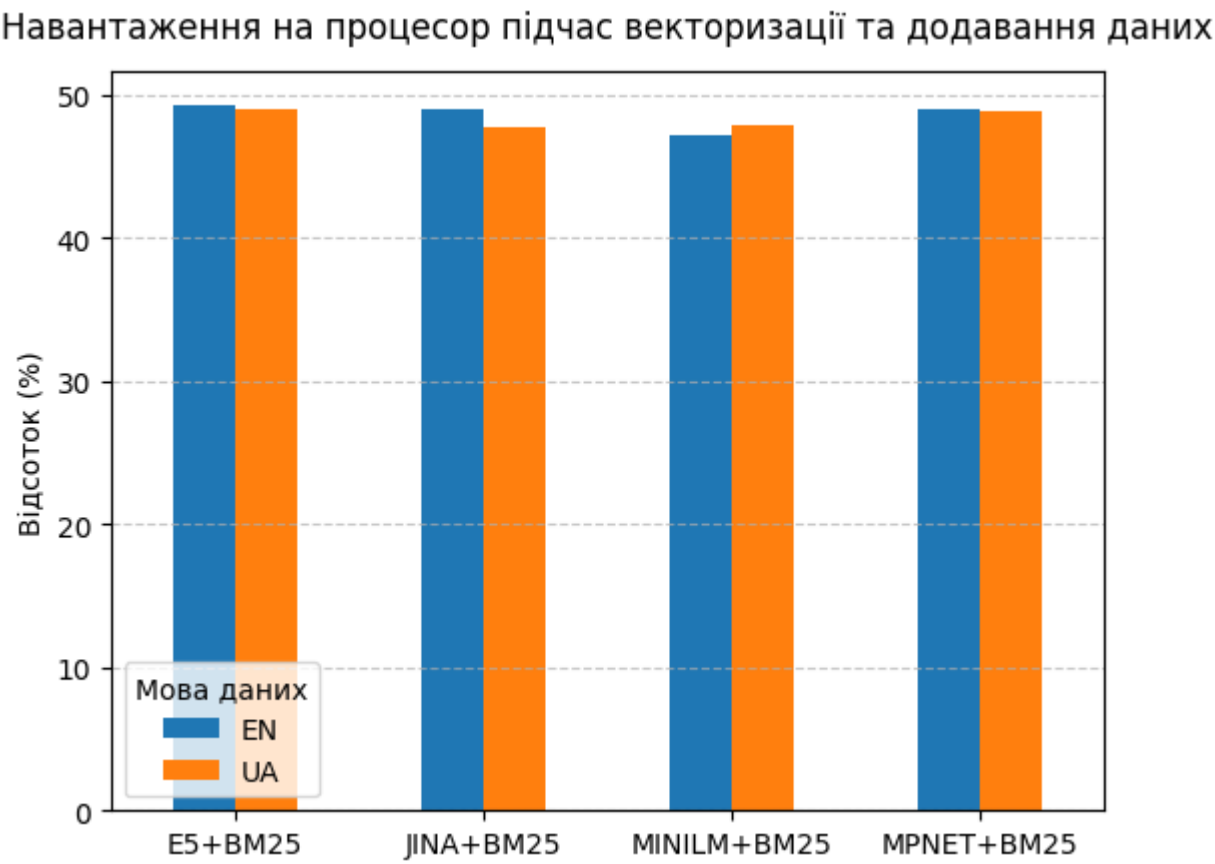

Рисунок 4.4 - Навантаження на процесор підчас векторизації моделями генерації щільних векторів та BM25, та додавання даних

Отримані результати є доволі передбачуваними: чим легшою є модель для генерації щільних векторів, тим меншим є апаратне навантаження і вищою – швидкість процесу векторизації. Незначні розбіжності в показниках під час обробки україномовних та англомовних текстів можна пояснити впливом фонових процесів системи, який під час експерименту було максимально мінімізовано. Проте неочікуваним результатом стала поведінка моделі JINA. Попри те, що вона є найбільшою за розміром, у стані спокою ця модель виявилася найбільш компактною і займає найменше місця в оперативній пам’яті. Натомість безпосередній етап векторизації даних за її допомогою є найбільш ресурсомістким, а споживання оперативної пам’яті перевищує показники інших моделей щонайменше в 6 разів.

4.2 Дослідження точності пошуку: моделі генерації щільних векторів

На даному етапі дослідження оцінювалася здатність моделей генерації щільних векторів (MINILM, MPNET, E5, JINA) самостійно знаходити релевантний контекст.

Для об’єктивної оцінки якості інформаційного пошуку використовувалися дві ключові метрики:

– Hit Rate (%) – відсоток запитів, для яких хоча б один релевантний документ потрапив у топ видачі. Ця метрика відображає загальну здатність моделі знаходити правильну відповідь;

– MRR (Mean Reciprocal Rank) – середній обернений ранг першого релевантного документа. Цей показник (на графіках помножений на 100 для зручності) демонструє, наскільки високо в списку результатів знаходиться правильна відповідь (чим ближче до 100, тим частіше правильна відповідь є 1-ю).

Тестування проводилося у двох режимах: мономовному (мова запиту збігається з мовою бази знань), результати якого наведено на рисунках *4.5 - 4.7*, та крос-мовному (запит англійською до української бази знань, і навпаки), результати якого – рисунки *4.8 - 4.10*.

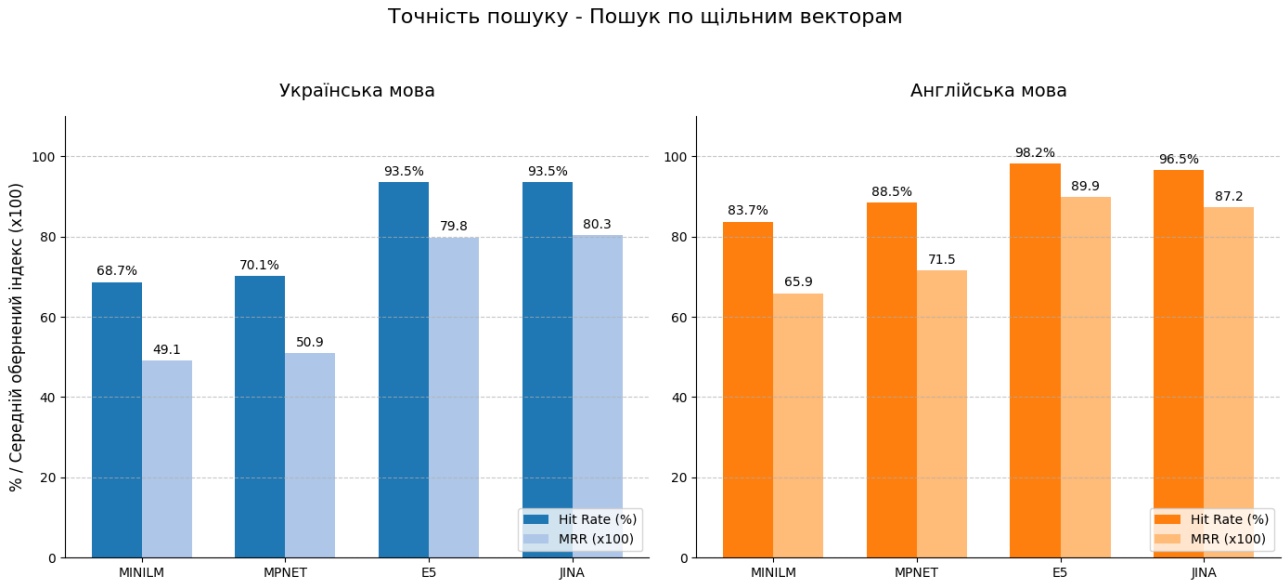

Рисунок 4.5 - Точність пошуку по щільним векторам

Результати одномовного пошуку (див. Рисунок 4.5) демонструють чіткий поділ моделей на дві ліги. Беззаперечними лідерами стали важкі моделі E5 та JINA. На україномовному наборі вони показали ідентичний Hit Rate – 93,5%, при цьому E5 (MRR 79,8) та JINA (MRR 80,3) забезпечили дуже високе ранжування знайдених документів. На англомовному наборі лідерство закріпила модель E5 із видатним результатом Hit Rate 98,2% (проти 96,5% у JINA).

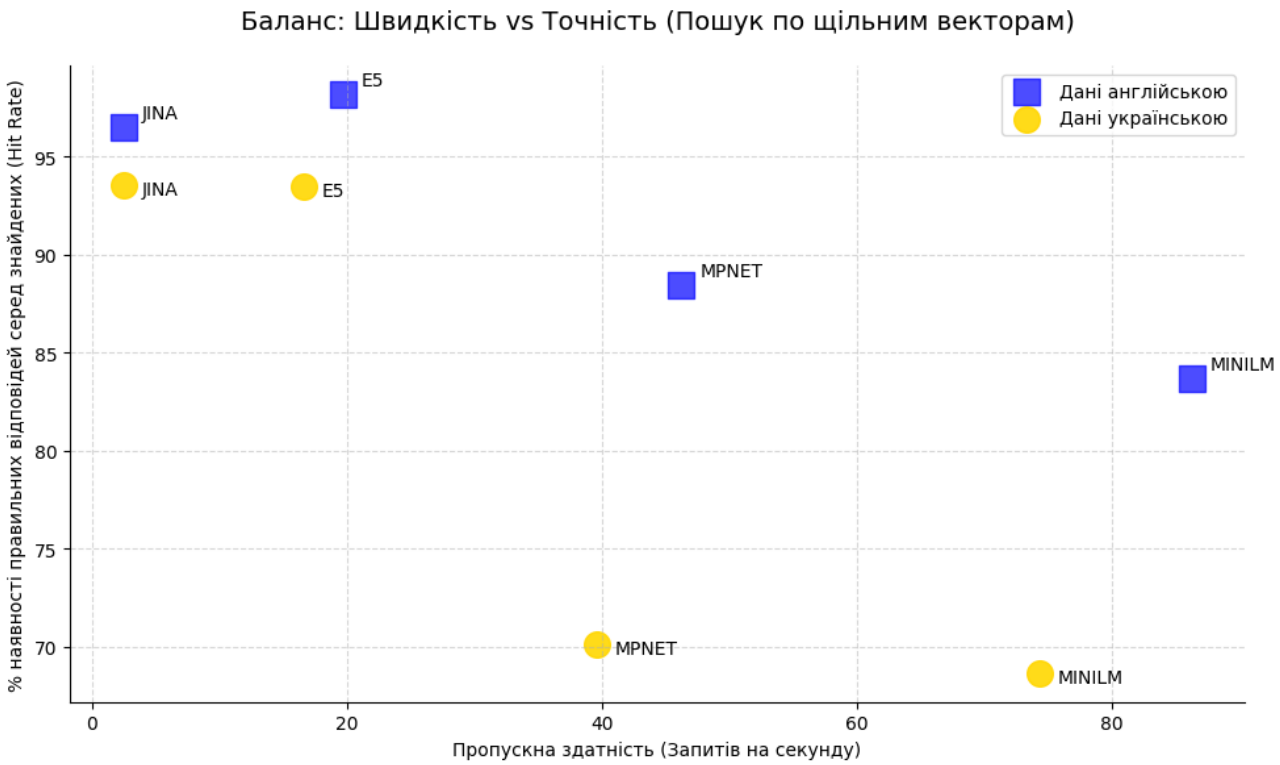

Рисунок 4.6 - Баланс між швидкістю та точністю пошуку по щільним векторам

Графік розсіювання (див. Рисунок 4.6) наочно доводить, що модель E5 є найкращим рішенням, оскільки гарантує найвищу якість пошуку для обох мов без критичного падіння пропускної здатності системи.

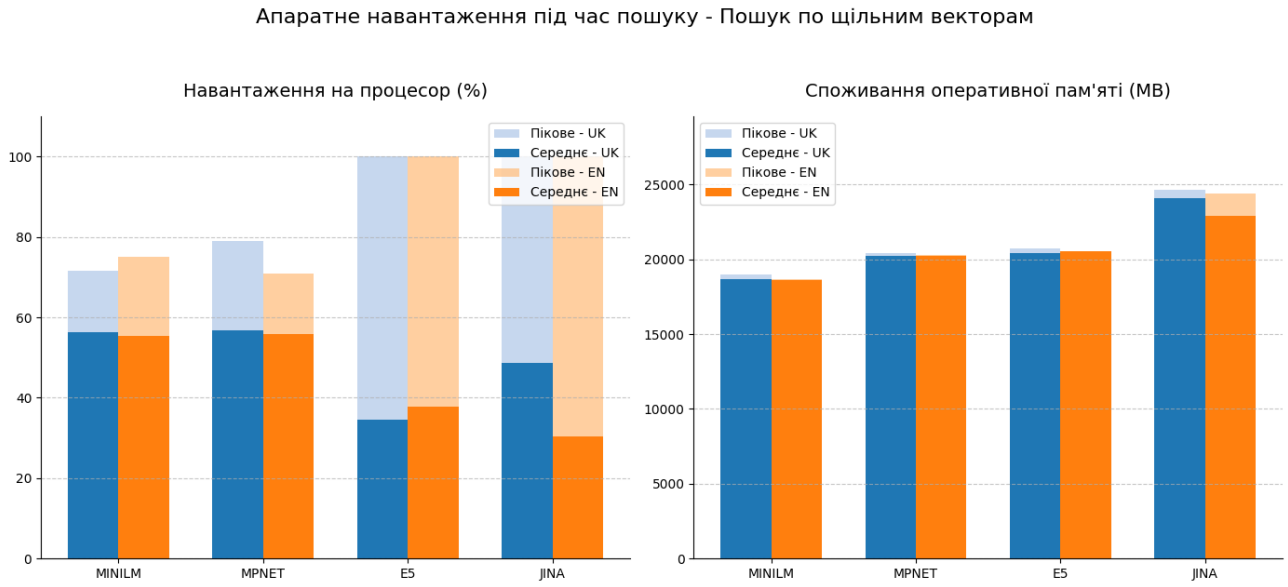

Рисунок 4.7 - Апаратне навантаження підчас пошуку по щільним векторам

Аналіз апаратного навантаження (див. Рисунок 4.7) виявив наступне:

– споживання пам’яті прямо пропорційне розміру моделі. Найбільш економною є MINILM (~18-19 ГБ), E5 та MPNET демонструють помірне та стабільне споживання (близько 20 ГБ), тоді як JINA вимагає найбільше ресурсів, наближаючись до позначки 25 ГБ;

– по навантаженню на процесор спостерігається нетипова динаміка навантаження. Важкі моделі (E5, JINA) мають порівняно низьке середнє навантаження (30-50%), проте під час розрахунку векторних відстаней генерують екстремальні пікові стрибки до 100%. Натомість легші моделі (MINILM, MPNET) підтримують вище середнє навантаження (близько 55-60%), але працюють більш рівномірно, без 100% піків.

Таким чином, модель E5 підтверджує свій статус оптимального компромісу: вона потребує помірного обсягу оперативної пам'яті (на рівні з MPNET), і хоча створює пікові навантаження на процесор, її середнє споживання CPU залишається найнижчим серед усіх досліджуваних моделей.

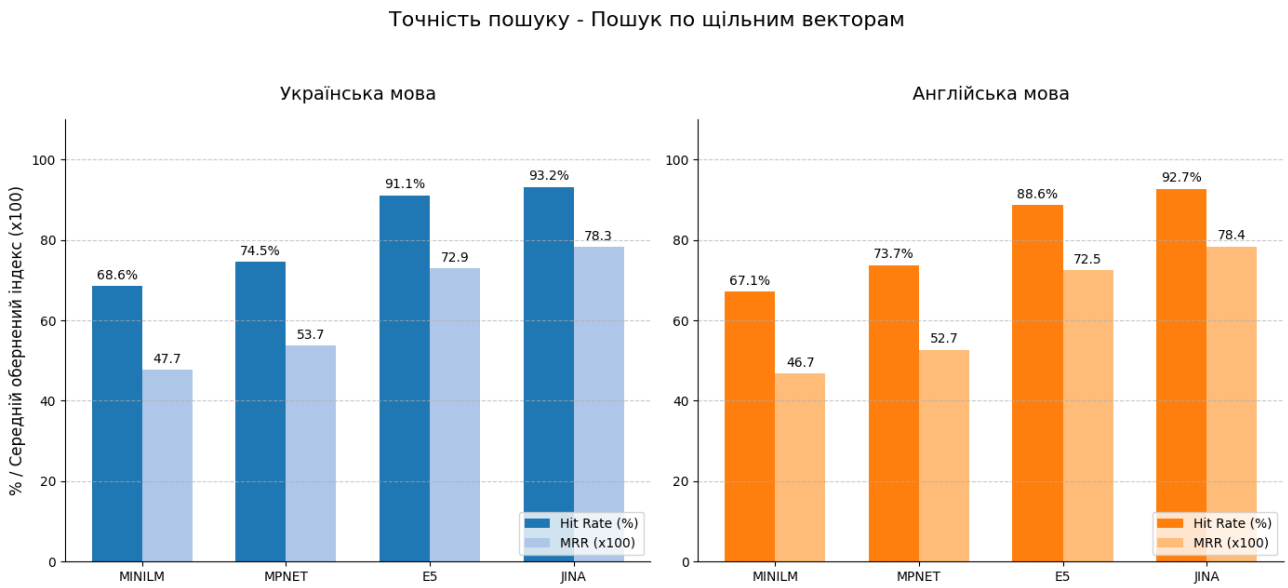

Рисунок 4.8 - Точність крос-мовного пошуку по щільним векторам

Крос-мовний пошук є найскладнішим випробуванням для векторних моделей, оскільки вимагає від моделі глибокого розуміння семантики незалежно від мови. За результатами тестування (див. Рисунок 4.8), моделі E5 та JINA чудово впоралися з цим завданням. При пошуку англійським запитом по українській базі JINA показала Hit Rate 93,2%, а E5 – 91,1%. У зворотному напрямку (запит українською по англійській базі) знову лідером стала JINA (92,7%), а E5 знову трохи відстала (88,6%). По значенням MRR у моделей E5 та JINA майже нема різниці (72,9/72,5 та 78,3/78,4 відповідно) відносно мови, на відміну від легших моделей, де різниця становить рівно 1, що пояснюється кращім розумінням семантики важчими моделями.

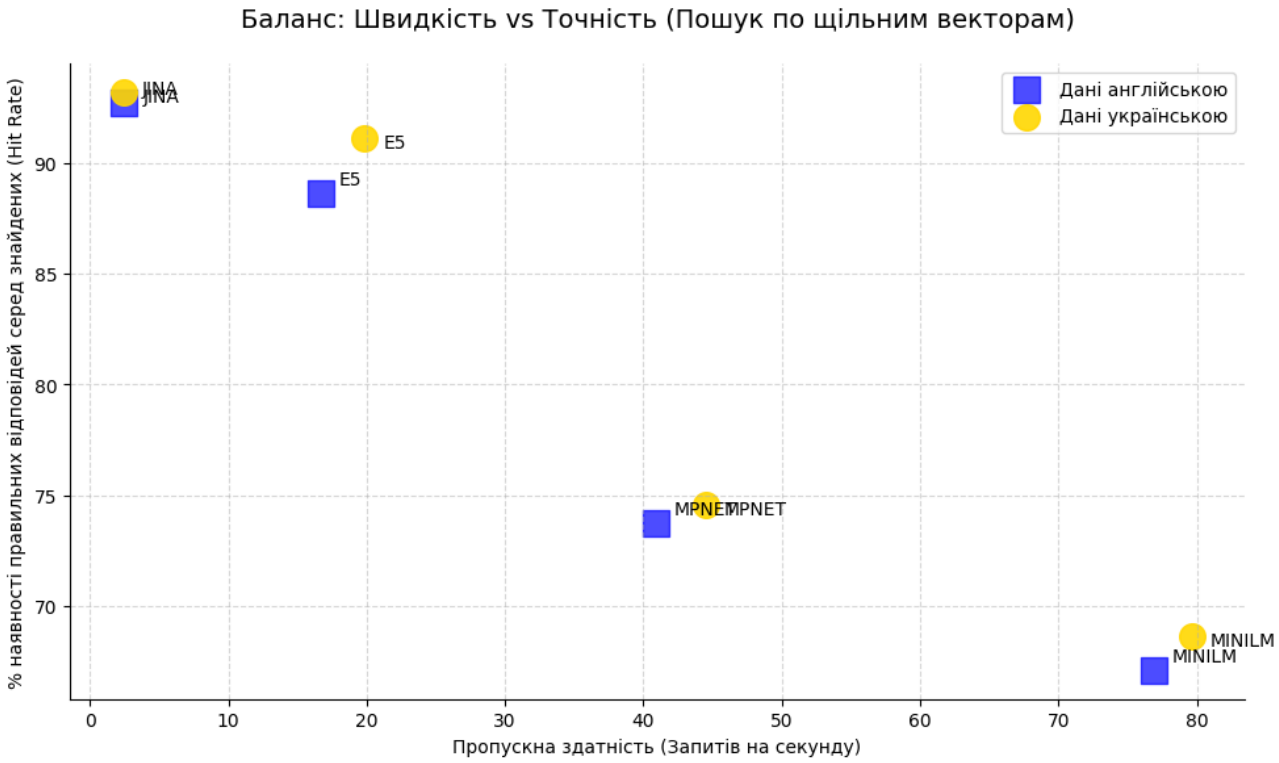

Рисунок 4.9 - Баланс між швидкістю та точністю крос-мовного пошуку по щільним векторам

Аналіз діаграми розсіювання (див. Рисунок 4.9) для крос-мовних запитів повністю підтверджує тенденції, виявлені під час мономовного тестування. Здатність долати мовний бар’єр прямо залежить від складності моделі:

– JINA утримує лідерство за точністю (понад 92%), але її продуктивність залишається критично низькою (менше 5 запитів на секунду);

– E5 демонструє статус ідеального компромісу: вона зберігає високу точність крос-мовного розуміння (88 - 91%) при стабільній швидкості близько 15 - 20 запитів на секунду, що є найкращим показником для реального впровадження;

– легкі моделі MINILM та MPNET мають як і раніше найбільшу пропускну здатність, але в даному їх точність впала нижче 75%, що вже є неприйнятним.

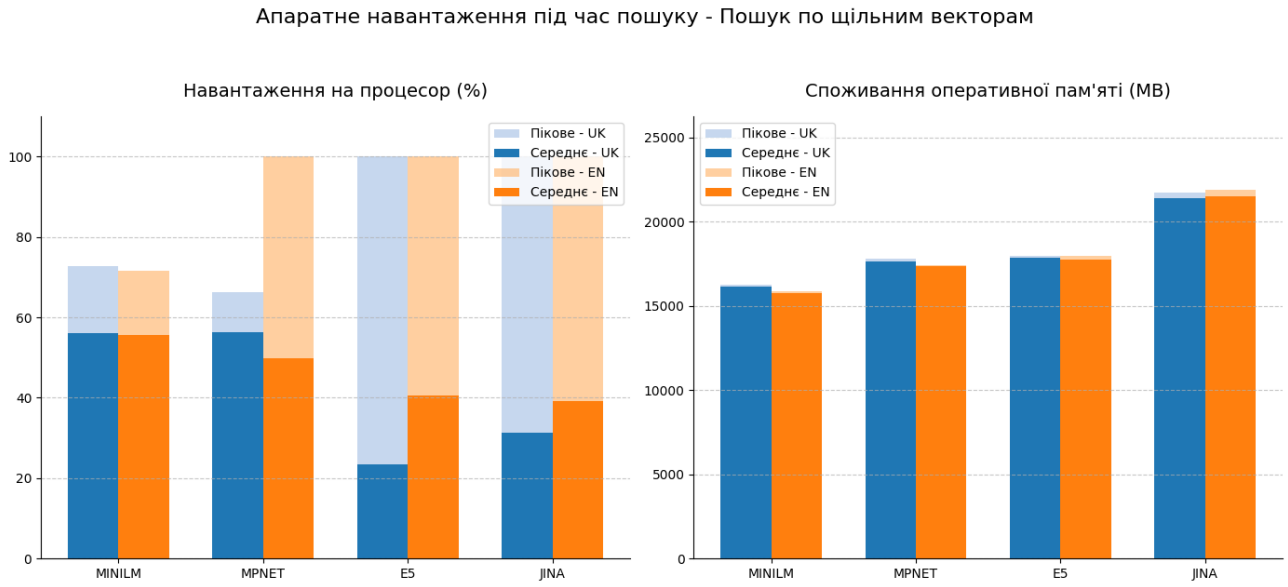

Рисунок 4.10 - Апаратне навантаження підчас крос-мовного пошуку по щільним векторам

За аналізом апаратного навантаження в умовах крос-мовного пошуку (див. Рисунок 4.10) споживання ресурсів зберігають стабільність і відповідають мономовним тестам:

– обсяг споживання оперативної пам’ять чітко корелює з вагою моделей: від найлегшої MINILM (~16 ГБ) до найважчої JINA (понад 21.5 ГБ);

– навантаження на процесор у крос-мовному середовищі зберігається нетиповим. Компактні моделі (MINILM, MPNET) мають вище середнє навантаження (50 - 55%). Натомість важкі архітектури (E5, JINA) демонструють значно менше середнє споживання (25 - 40%), однак із максимальними піковими навантаженнями до 100%.

4.3 Дослідження точності пошуку: BM25

У цьому дослідженні проведена оцінка пошуку виключно за допомогою моделі генерації розріджених векторів BM25.

Для об’єктивної оцінки якості інформаційного пошуку використовувалися ті самі метрики (Hit Rate (%) та MRR×100), що і в попередньому дослідженні пошуку.

Це дослідження як і попереднє дослідження пошуку має 2 частини: мономовну (рисунки 4.11 - 4.13) та крос-мовну (рисунки 4.14 - 4.16).

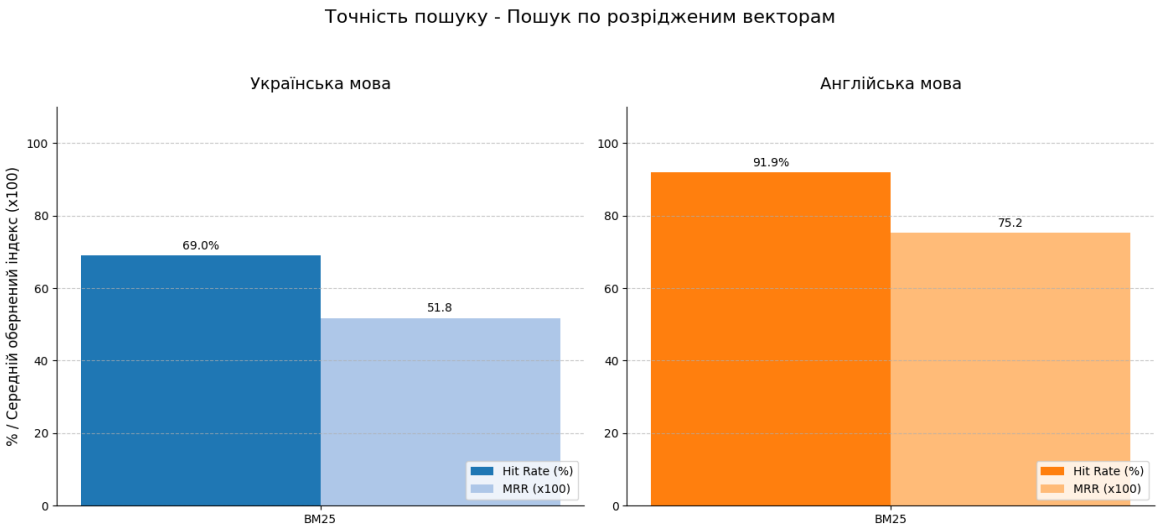

Рисунок 4.11 - Точність пошуку BM25

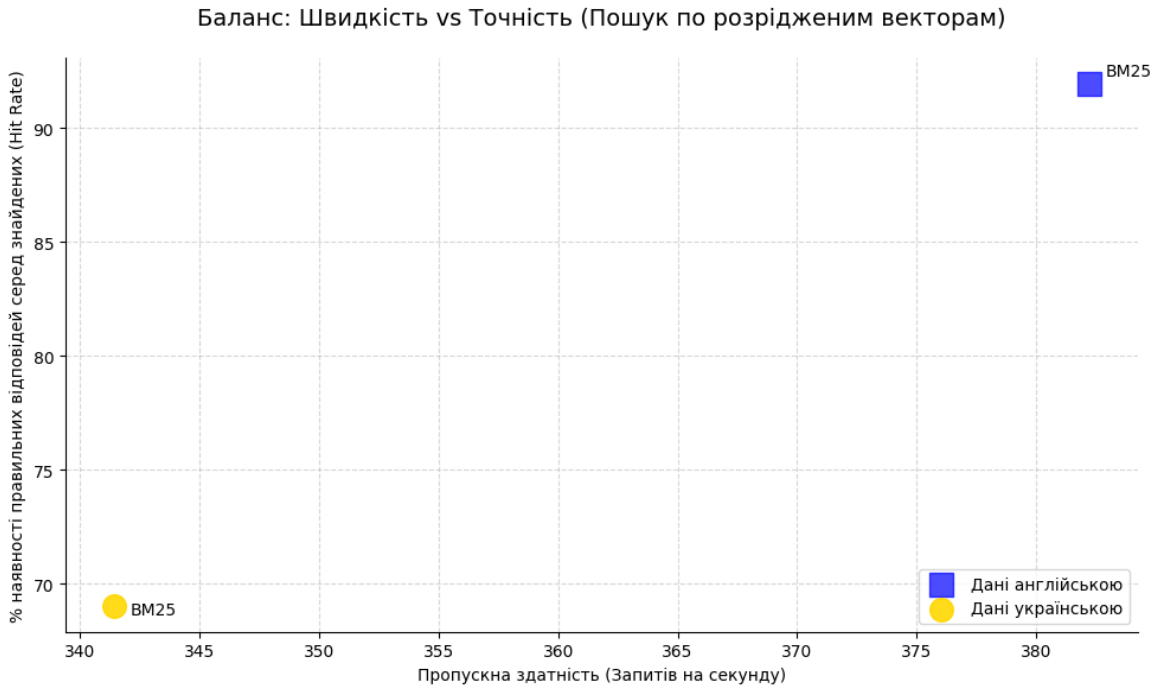

Рисунок 4.12 - Баланс між швидкістю та точністю пошуку BM25

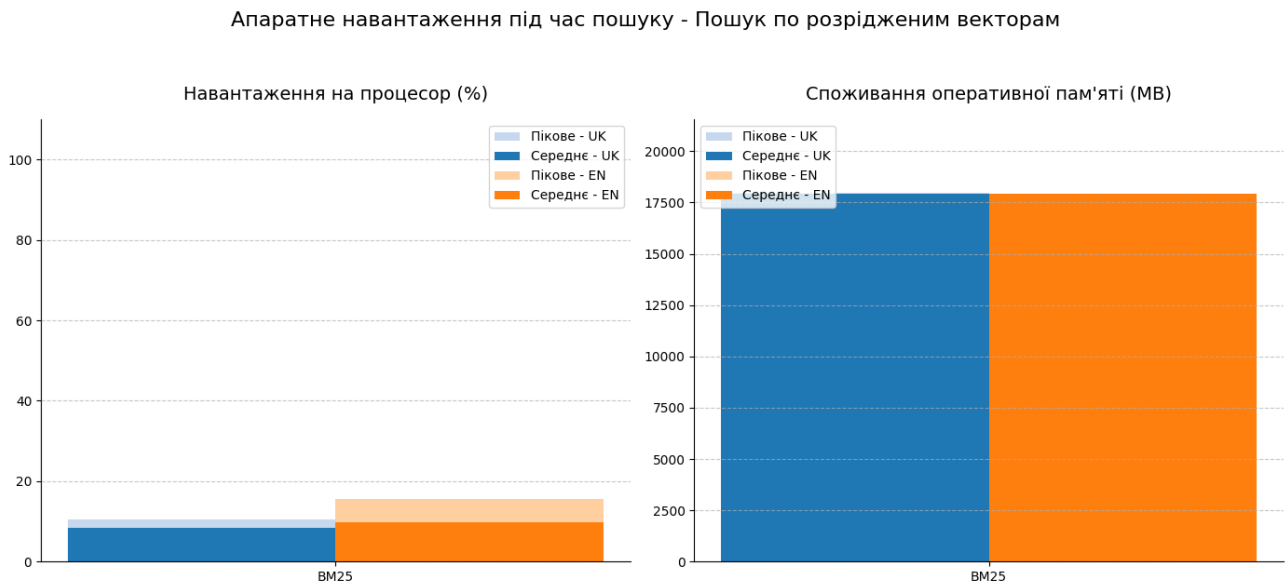

Рисунок 4.13 - Апаратне навантаження підчас пошуку BM25

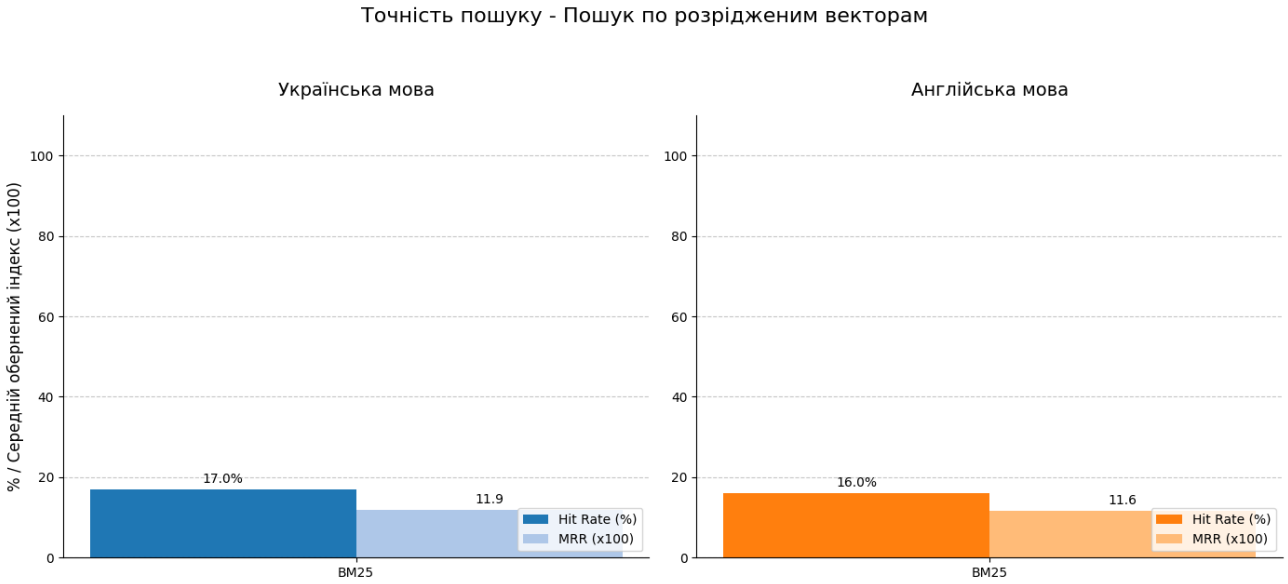

Рисунок 4.14 - Точність крос-мовного пошуку BM25

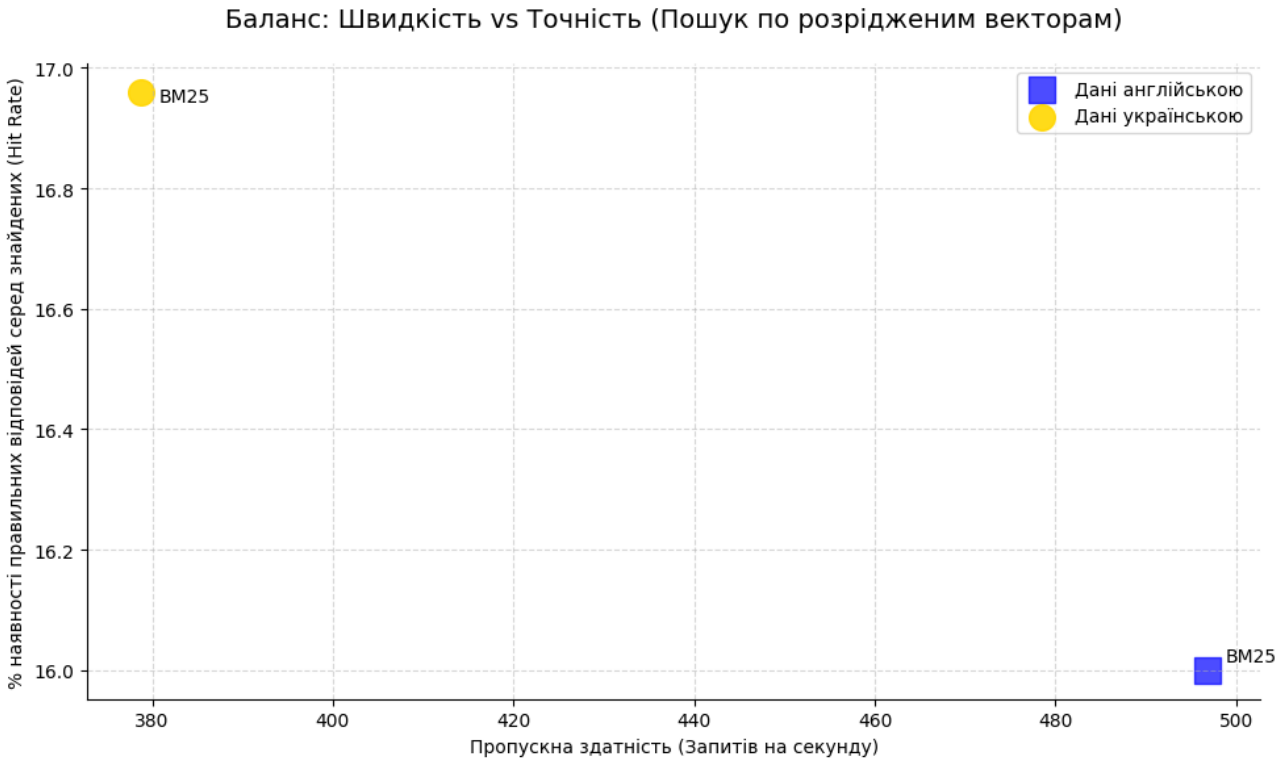

Рисунок 4.15 - Баланс між швидкістю та точністю крос-мовного пошуку BM25

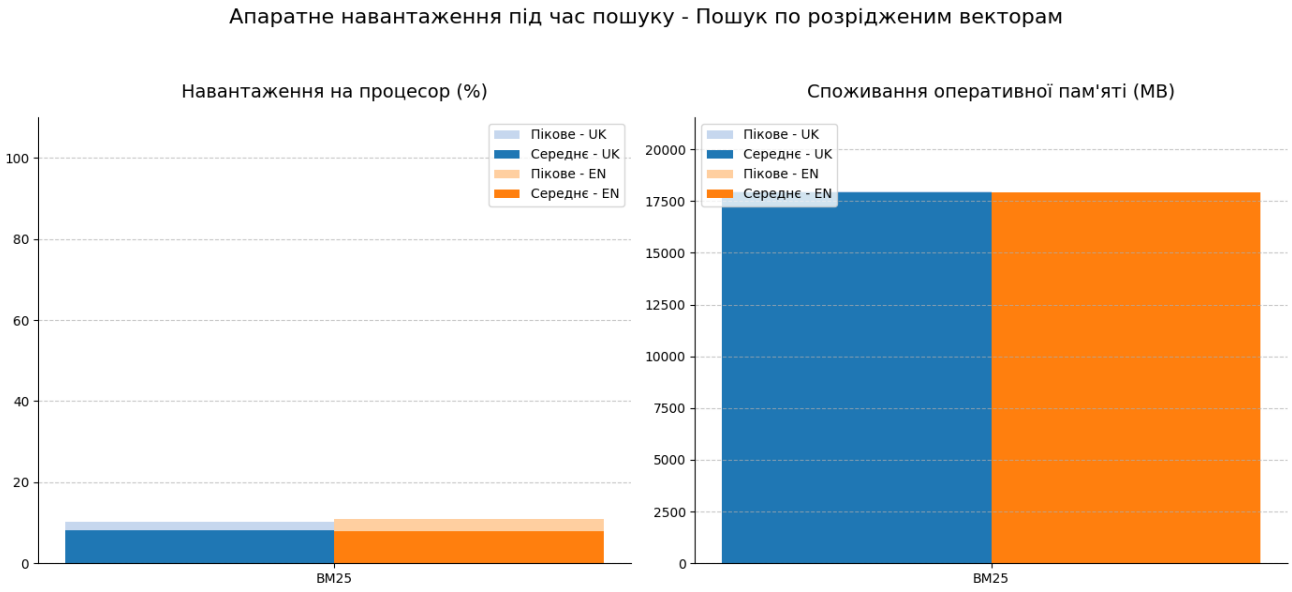

Рисунок 4.16 - Апаратне навантаження підчас крос-мовного пошуку BM25

Дослідження класичного алгоритму BM25 наочно продемонструвала його сильні та слабкі сторони:

– мономовний пошук (див. рис. 4.11 - 4.13): у середовищі, де мова запиту збігається з мовою документів, BM25 показав відмінну точність для англійської мови (Hit Rate 91,9%). Однак на україномовних текстах цей показник впав до 69%, що пояснюється багатшою морфологією української мови та нездатністю алгоритму розуміти синонімічні ряди (пошук відбувається лише за точним збігом);

– тестування крос-мовних запитів (див. рис. 4.14 - 4.16) виявило абсолютну «сліпу зону» алгоритму. Як видно з результатів, точність (Hit Rate) обвалилася до катастрофічних 16 - 17%. Цей результат є закономірним: оскільки алгоритм шукає статистичні збіги конкретних слів, він фізично не здатен співставити англійський запит з українським текстом без попереднього машинного перекладу;

– продуктивність: незважаючи на проблеми з точністю, BM25 залишається абсолютним еталоном продуктивності. Алгоритм створює мінімальне навантаження на процесор (до 10%), споживає найменше оперативної пам’яті (~18 ГБ) і забезпечує безпрецедентну пропускну здатність від 340 до майже 500 запитів на секунду.

Незважаючи на неперевершену швидкість та ресурсоефективність, результати крос-мовного тестування беззаперечно доводять: використання суто лексичного пошуку (BM25) як самостійного інструмента в сучасних багатомовних RAG-системах є неможливим. Це повністю обґрунтовує необхідність використання семантичних (щільних) моделей та переходу до архітектури гібридного пошуку.

4.4 Дослідження точності гібридного пошуку: моделі генерації щільних векторів + BM25

У даному дослідженні проведена оцінка гібридного пошуку, який об’єднує результати семантичних (щільних) моделей та класичного лексичного алгоритму (BM25) за допомогою методів злиття рангів. Метою було перевірити, чи здатна синергія точного збігу ключових слів та глибинного розуміння контексту перевершити результати ізольованих методів.

Для об’єктивної оцінки якості інформаційного пошуку використовувалися ті самі метрики (Hit Rate (%) та MRR×100), що і в попередніх дослідженнях пошуку.

Це дослідження як і попередні дослідження пошуку має 2 частини: мономовну (рисунки 4.17 - 4.19) та крос-мовну (рисунки 4.20 - 4.22).

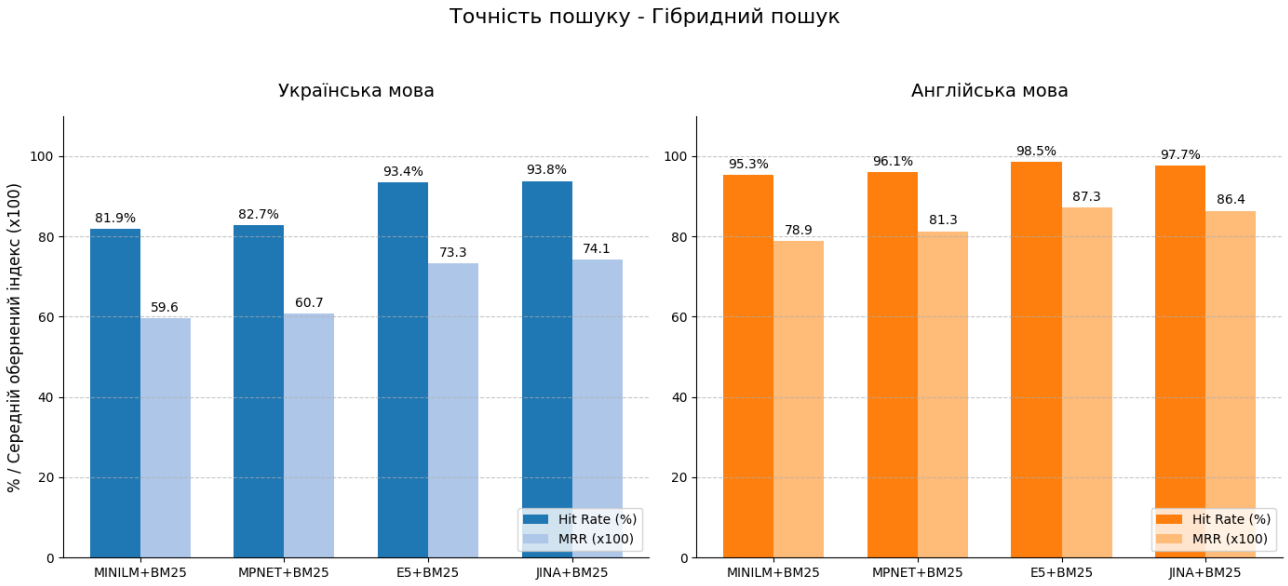

Рисунок 4.17 - Точність гібридного пошуку по щільним векторам та BM25

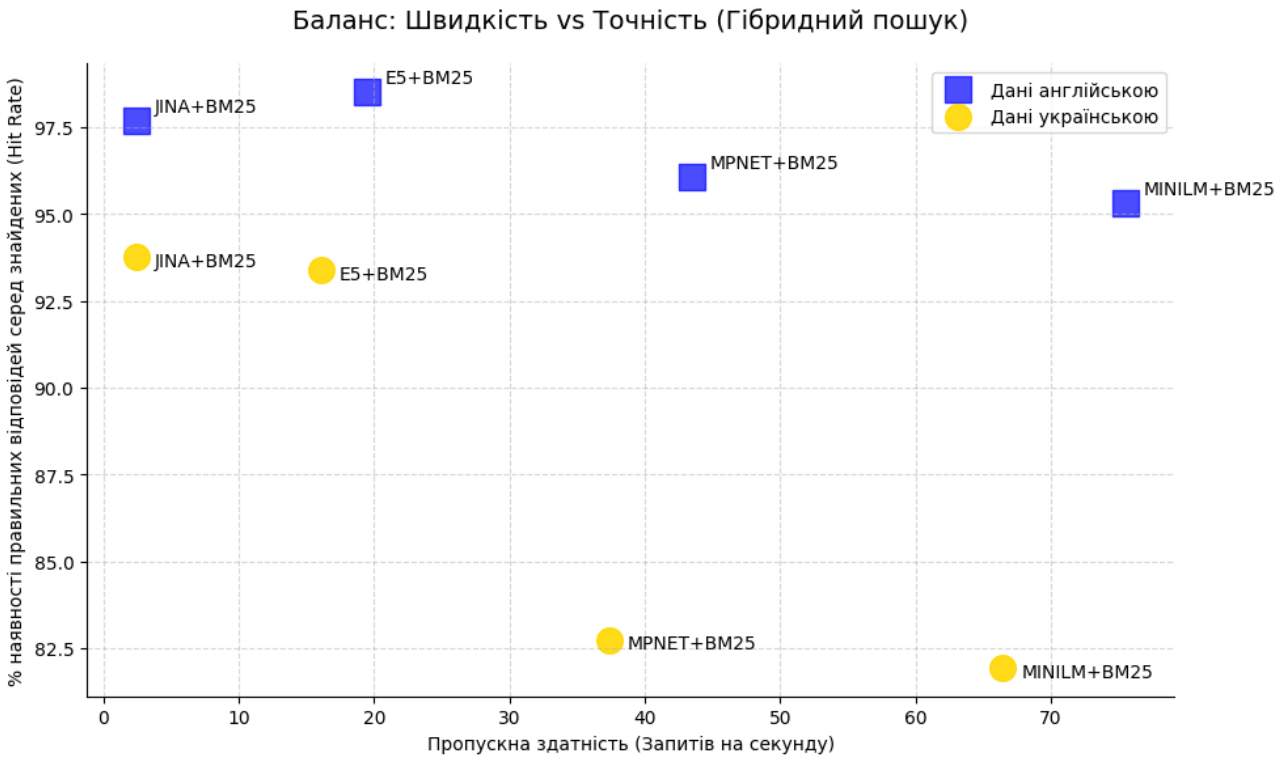

Рисунок 4.18 - Баланс між швидкістю та точністю гібридного пошуку по щільним векторам та BM25

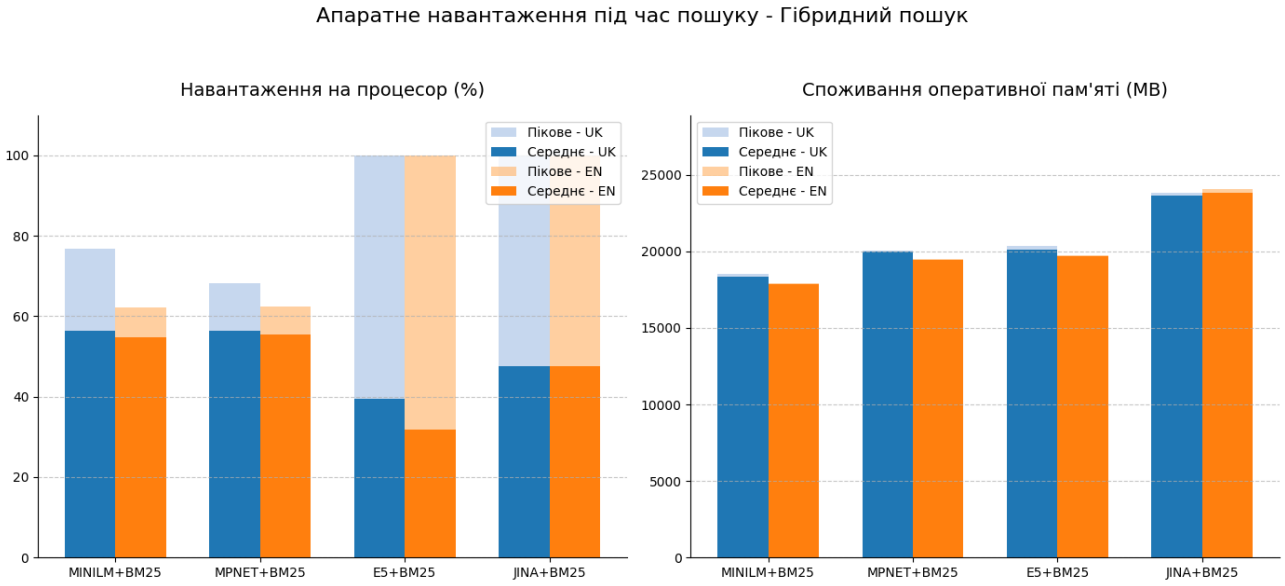

Рисунок 4.19 - Апаратне навантаження підчас гібридного пошуку по щільним векторам та BM25

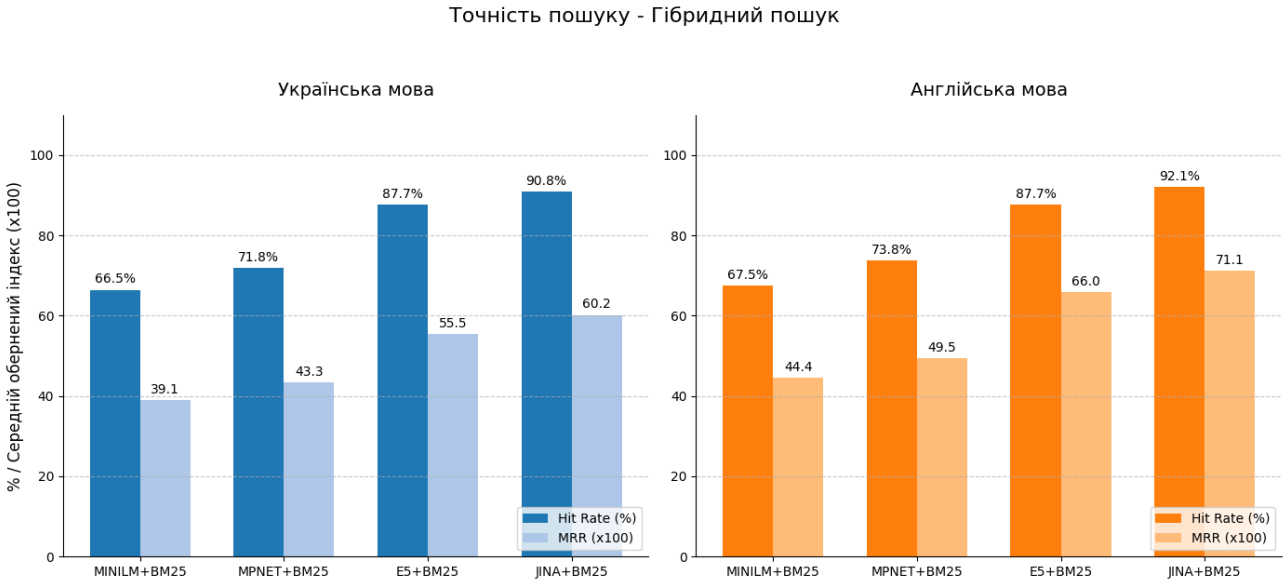

Рисунок 4.20 - Точність крос-мовного гібридного пошуку по щільним векторам та BM25

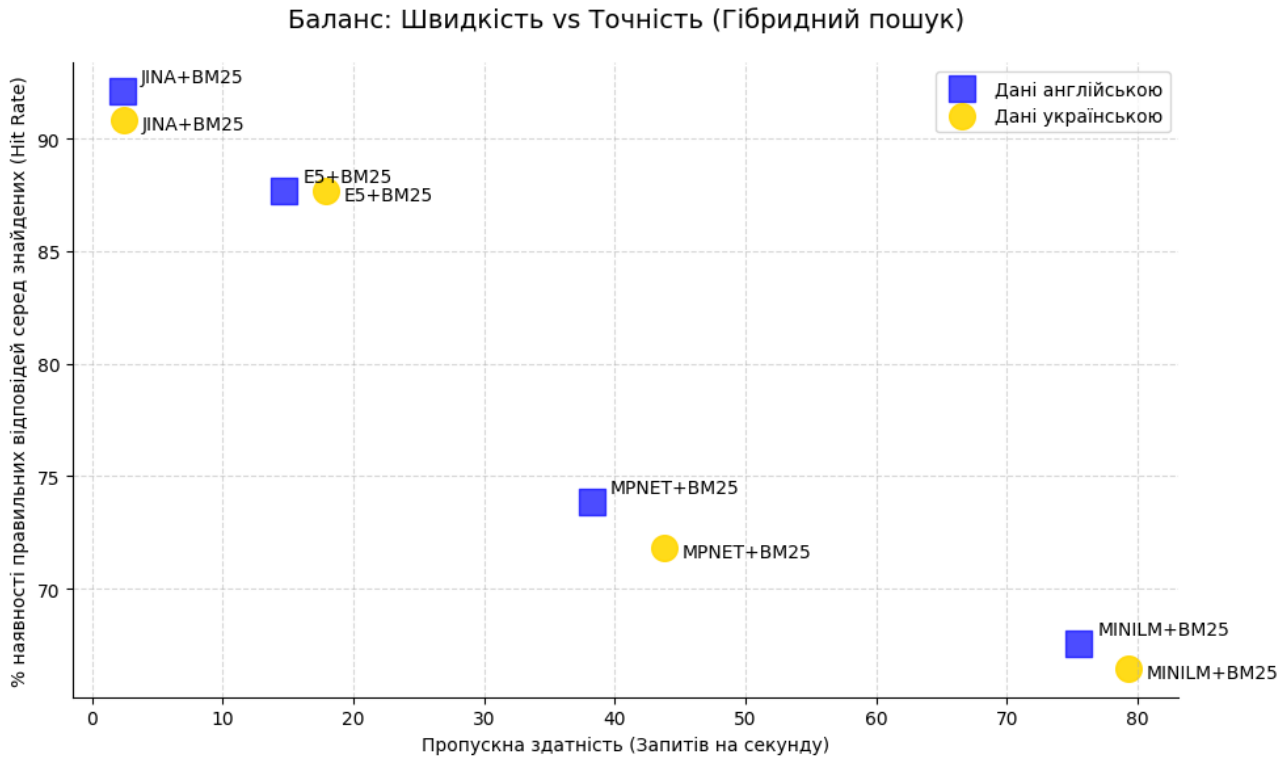

Рисунок 4.21 - Баланс між швидкістю та точністю крос-мовного гібридного пошуку по щільним векторам та BM25

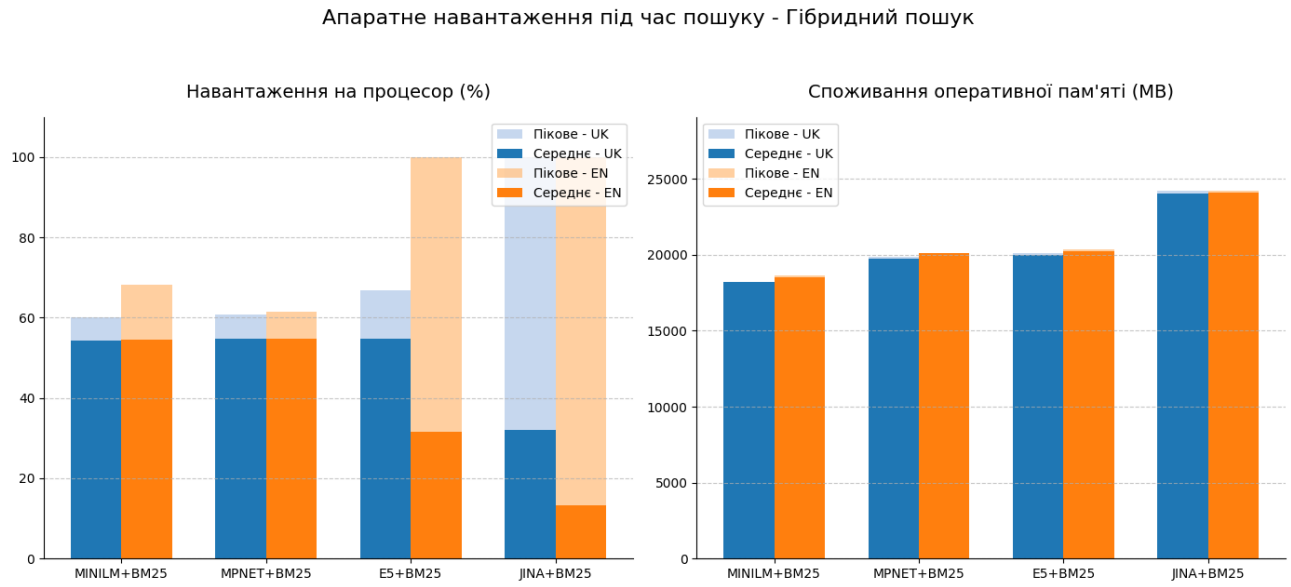

Рисунок 4.22 - Апаратне навантаження підчас крос-мовного гібридного пошуку по щільним векторам та BM25

Порівняння результатів гібридного пошуку (див. Рисунок 4.17, Рисунок 4.18, Рисунок 4.20, Рисунок 4.21) з попередніми тестами з однотипними моделями (див. Рисунок 4.5, Рисунок 4.6, Рисунок 4.8, Рисунок 4.9, Рисунок 4.11, Рисунок 4.12, Рисунок 4.14, Рисунок 4.15) демонструє яскраво виражений позитивний ефект, проте його прояв кардинально залежить від мовного середовища:

– у порівняння із суто щільними векторами:

1) у мономовному сценарії (див. Рисунок 4.5, Рисунок 4.6) додавання лексичного компонента суттєво покращує результати слабших моделей (мономовний Hit Rate MINILM для української мови зріс із 68,7% до 81,9%). Для потужних моделей гібридний підхід фіксує результати на високому рівні: гібрид E5+BM25 досяг безпрецедентних 98,5% для англійської мови (проти 98,2%);
2) у крос-мовному сценарії (див. Рисунок 4.8, Рисунок 4.9) показники гібридного пошуку виявилися майже ідентичними до результатів суто щільного пошуку (наприклад, Hit Rate комбінації E5+BM25

склав 87,7%, що в межах похибки відповідає рівню ізольованої E5). Цей феномен має чітке пояснення: оскільки класичний BM25 шукає лише точні збіги і не здатен співставляти слова різними мовами (що підтвердив його результат у 16%), вага його видачі під час злиття рангів для крос-мовного запиту є нульовою. Відповідно, гібридна система автоматично і повністю делегує відповідальність за пошук щільній моделі;

3) швидкість не залежно від сценарію (мономовного чи крос-мовного) майже не змінилася. Це пояснюється тим, що швидкість пошуку по розрідженим векторам BM25 дуже висока і переранжування результатів виконується майже миттєво, тим самим залишаючи майже абсолютний вплив на швидкість на найповільніший процес – пошук по щільним векторам;

– порівняння із суто розрідженим пошуком (BM25) (див. Рисунок 4.11, Рисунок 4.14,) гібридний підхід повністю нівелює критичні недоліки лексичного пошуку. Якщо чистий BM25 показав катастрофічне падіння точності до 16% у крос-мовному сценарії, то в гібриді, саме завдяки семантичному компоненту, цей показник відновлюється до 87 - 92% (залежно від обраної моделі). Також гібрид вирішує проблему розуміння синонімів та морфології в україномовному пошуку, піднімаючи базові 69% алгоритму BM25 до понад 93% у комбінації з E5.

Аналізуючи графіки апаратного навантаження (див. Рисунок 4.7, Рисунок 4.19) стає зрозуміло, що додавання BM25 до щільних векторів збільшує використання пам’яті на ~3ГБ.

За результатами проведених досліджень (див. Рисунок 4.6, Рисунок 4.9, Рисунок 4.12, Рисунок 4.15, Рисунок 4.18, Рисунок 4.21) фаворитом лишається комбінація E5+BM25, адже вона разом з JINA+BM25 мають найвищі показники в точності (на різних тестах то одна трохи випереджає, то інша) значно випереджаючи інші, але при цьому E5 (чисто або в комбінації) завжди зберігає

значно вищу швидкість пошуку. Спираючись на це, наступні досліди розріджених моделей, що лишилися недослідженими, будуть проведені в комбінації з моделлю генерації щільних векторів E5.

4.5 Дослідження швидкості векторизації і додавання чанків у векторну БД: краща модель генерації щільних векторів + моделі генерації розріджених векторів

В даному дослідженні використовувались ті самі набори даних, які були створенні для проведення попереднього дослідження додавання даних. Для проведення експерименту найкращу базову щільну модель (E5) було об'єднано з розрідженими моделями – BM42 та MINICOIL. З метою забезпечення спадковості експерименту та об'єктивності порівняння, до аналізу також було включено еталонну комбінацію-лідера попереднього етапу (E5+BM25), а також комбінацію, яка час від часу трохи випереджала лідера за точністю, але завжди програвала за швидкістю (JINA+BM25). Основна мета полягає в оцінці апаратної вартості ускладнення розрідженого компонента гібридної архітектури та його подальшого впливу на точність та швидкість відповідей. Результати цього дослідження наведено на рисунках 4.23 - 4.26.

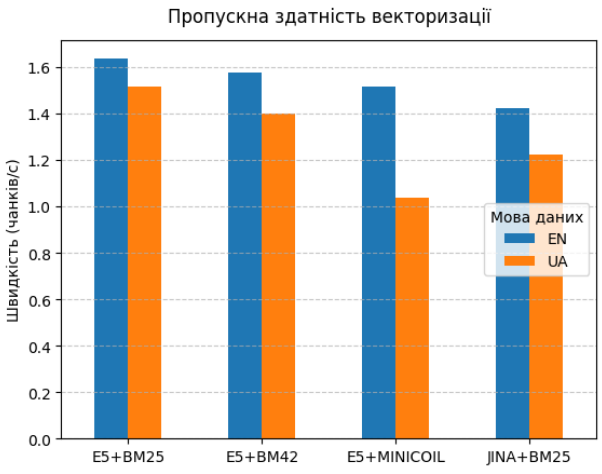

Рисунок 4.23 - Швидкості векторизації чанків моделями генерації розріджених векторів та E5 (додатково JINA+BM25)

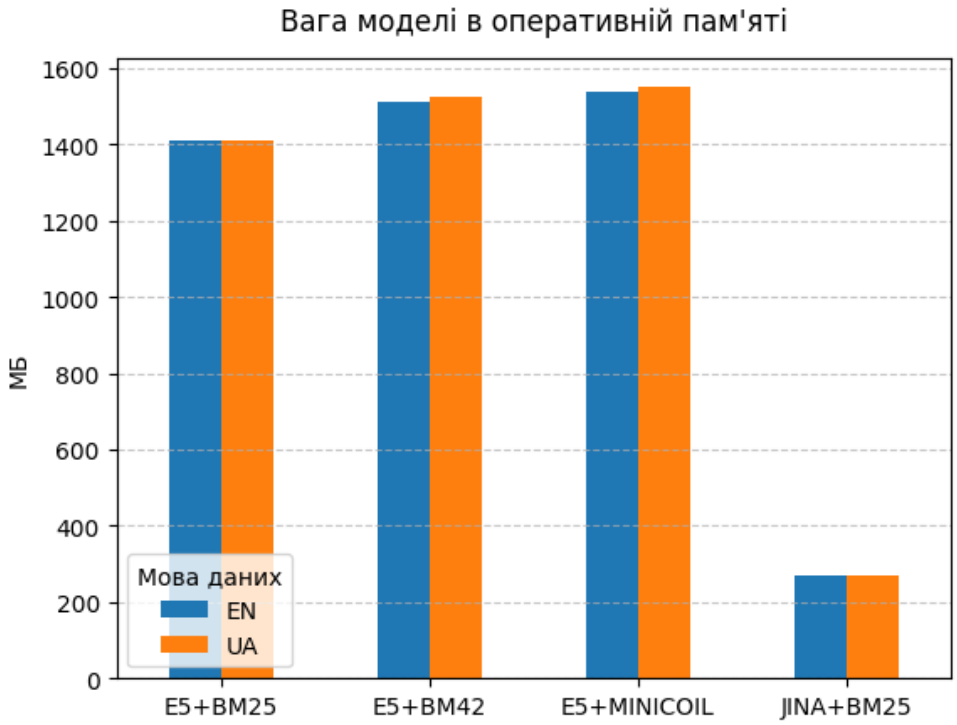

Рисунок 4.24 - Вага в оперативній пам’яті моделей генерації розріджених векторів та E5 (додатково JINA+BM25)

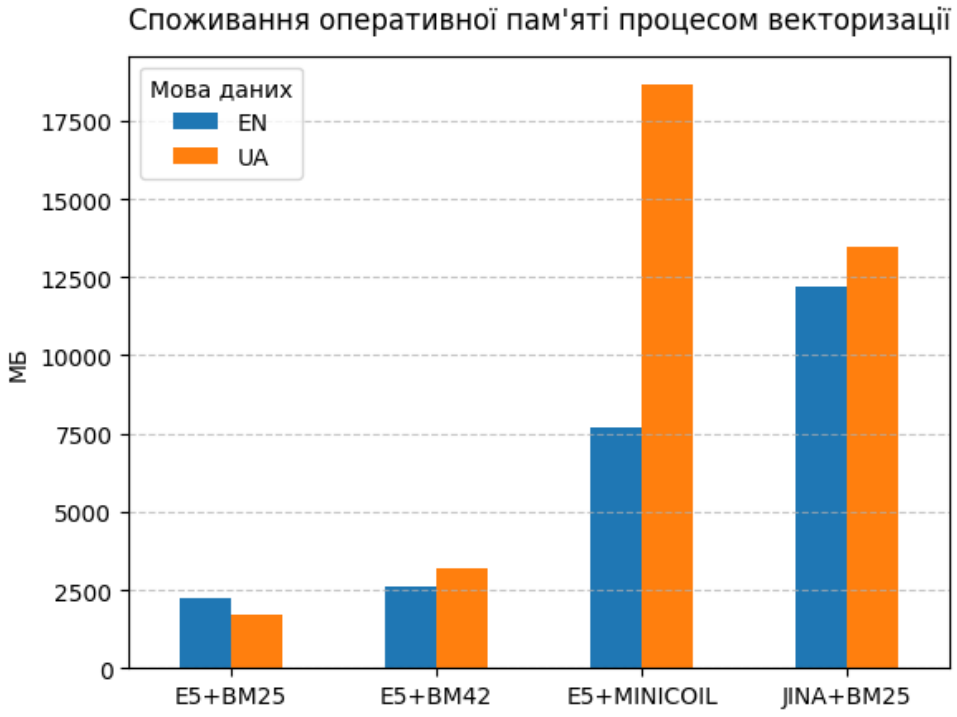

Рисунок 4.25 - Споживання оперативної пам’яті процесом векторизації моделями генерації розріджених векторів та E5 (додатково JINA+BM25)

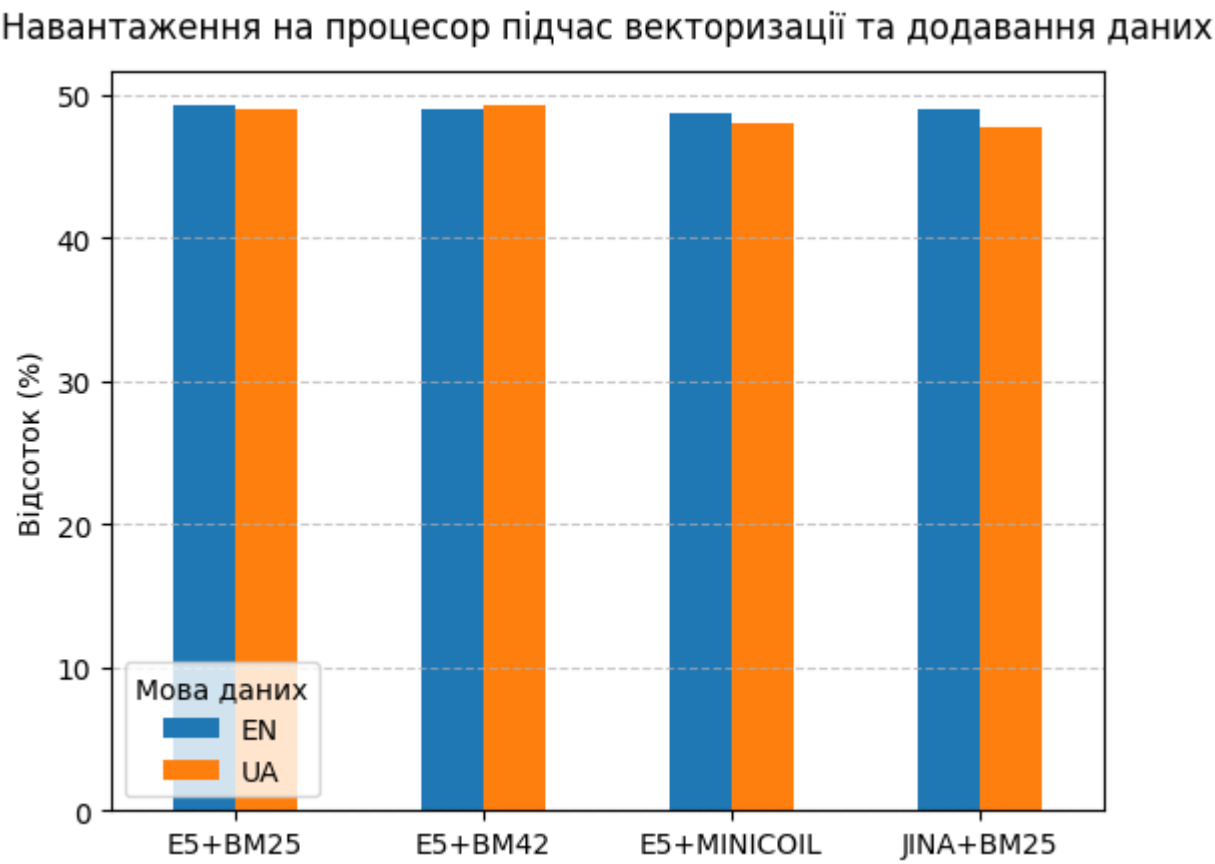

Рисунок 4.26 - Навантаження на процесор підчас векторизації моделями генерації розріджених векторів та E5 (додатково JINA+BM25), та додавання даних

Аналіз отриманих даних засвідчив, що заміна класичної статистики на альтернативні розріджені моделі неминуче підвищує апаратні вимоги:

– E5+BM42 продемонструвала оптимальний баланс. Комбінація зберігає високу пропускну здатність (трохи нижчу за BM25) та демонструє адекватне і стабільне споживання оперативної пам’яті (до 3 ГБ) незалежно від мови;

– E5+MINICOIL продемонструвала критичну архітектурну аномалію. Незважаючи на стабільне навантаження на процесор, при векторизації україномовних текстів споживання оперативної пам’яті цією комбінацією екстремально зростає до ~18 ГБ (перевершуючи навіть JINA), що супроводжується найнижчою швидкістю обробки (близько 1 чанка/с).

4.6 Дослідження точності пошуку: моделі генерації розріджених векторів

В даному дослідженні було проведено порівняльний аналіз роботи всіх обраних моделей генерації розріджених векторів (BM25, BM42 і MINICOIL). Метою було визначити, яка з альтернатив є кращою.

Для об’єктивної оцінки якості інформаційного пошуку використовувалися ті самі метрики (Hit Rate (%) та MRR×100), що і в попередніх дослідженнях пошуку.

Тестування проводилося як в одномовному, результати якого наведено на рисунках 4.27 - 4.29, так і в крос-мовному сценаріях, результат якого наведено на рисунках 4.30 - 4.32.

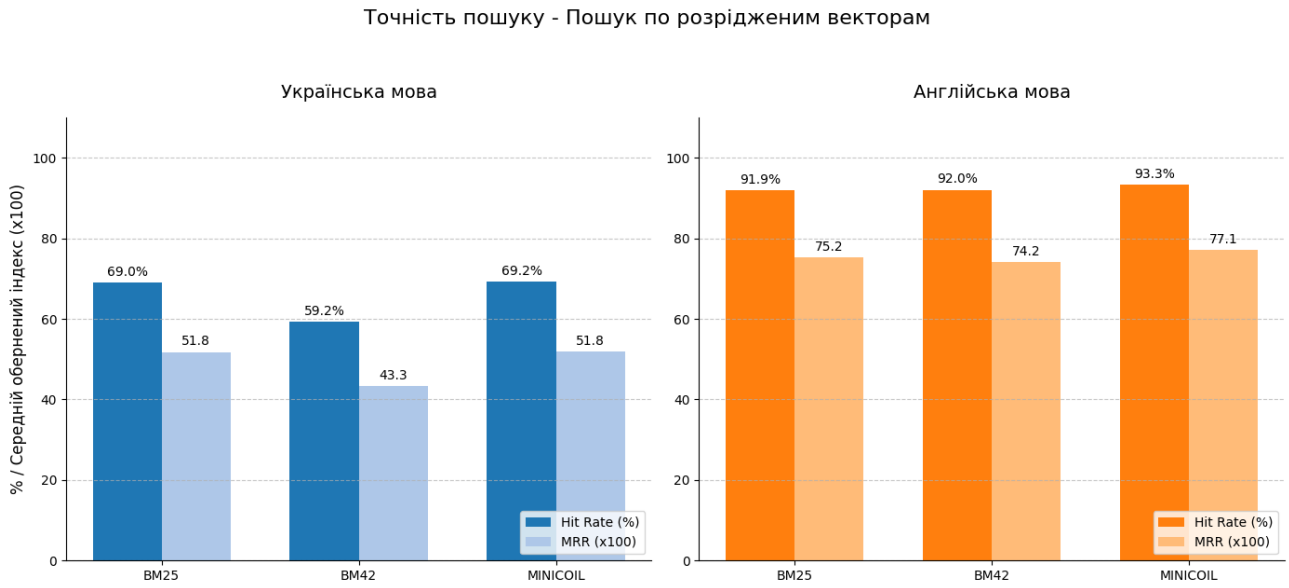

Рисунок 4.27 - Точність пошуку по розрідженим векторам

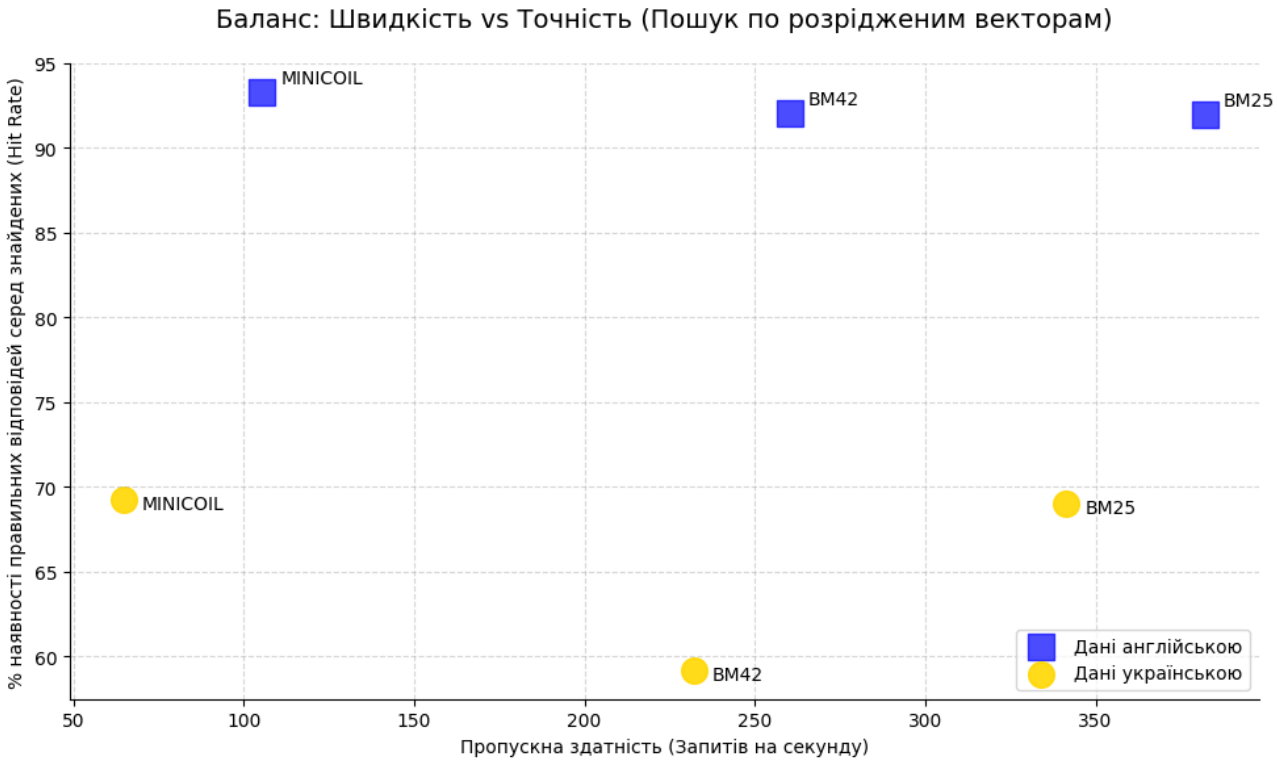

Рисунок 4.28- Баланс між швидкістю та точністю пошуку по розрідженим векторам

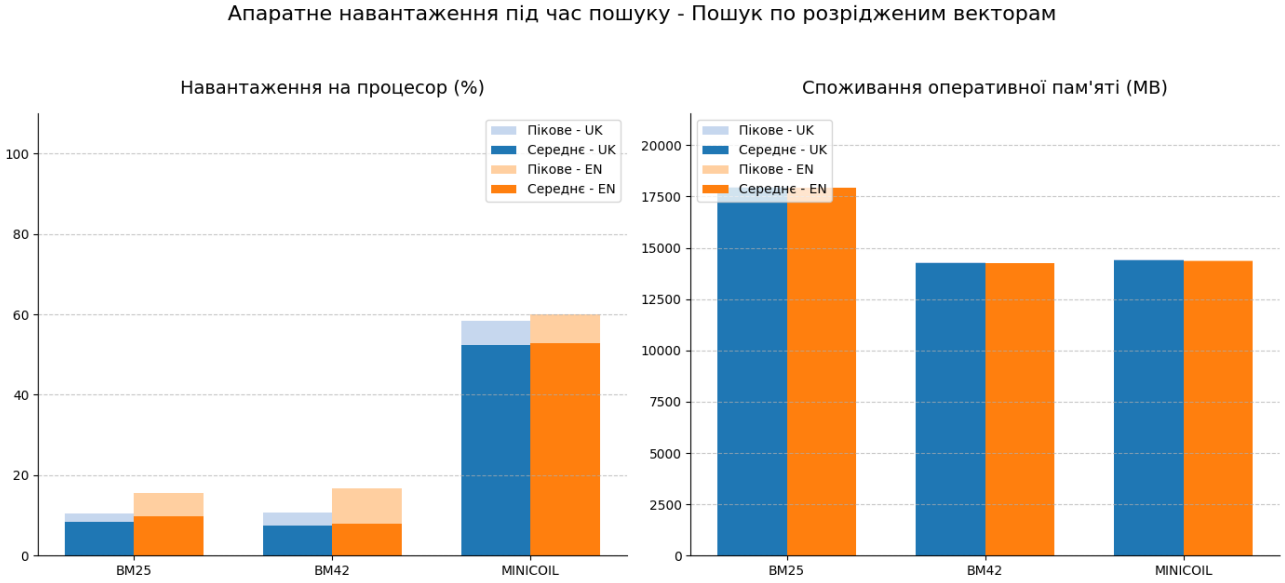

Рисунок 4.29 - Апаратне навантаження підчас пошуку по розрідженим векторам

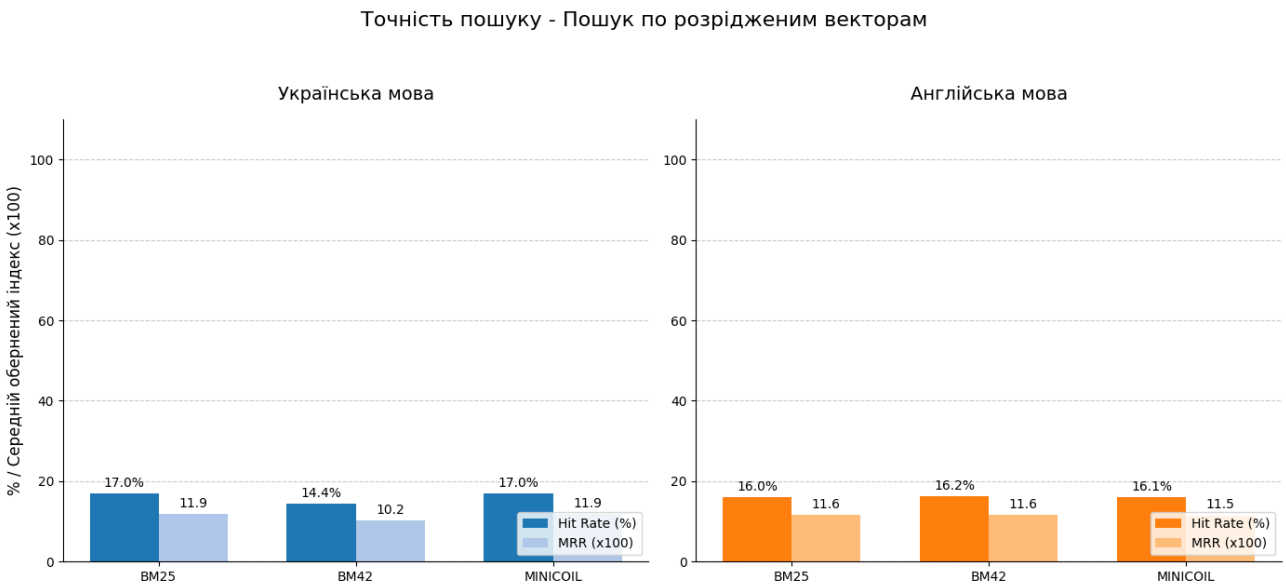

Рисунок 4.30 - Точність крос-мовного пошуку по розрідженим векторам

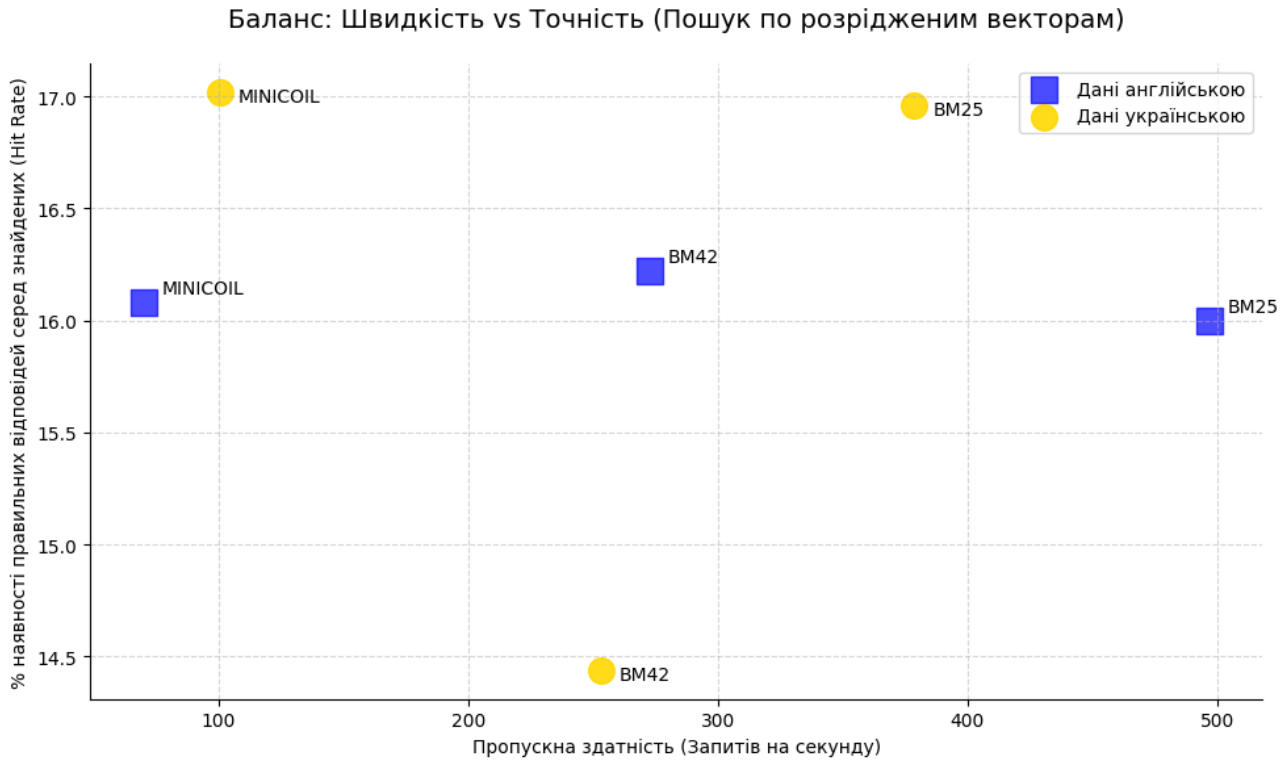

Рисунок 4.31 - Баланс між швидкістю та точністю крос-мовного пошуку по розрідженим векторам

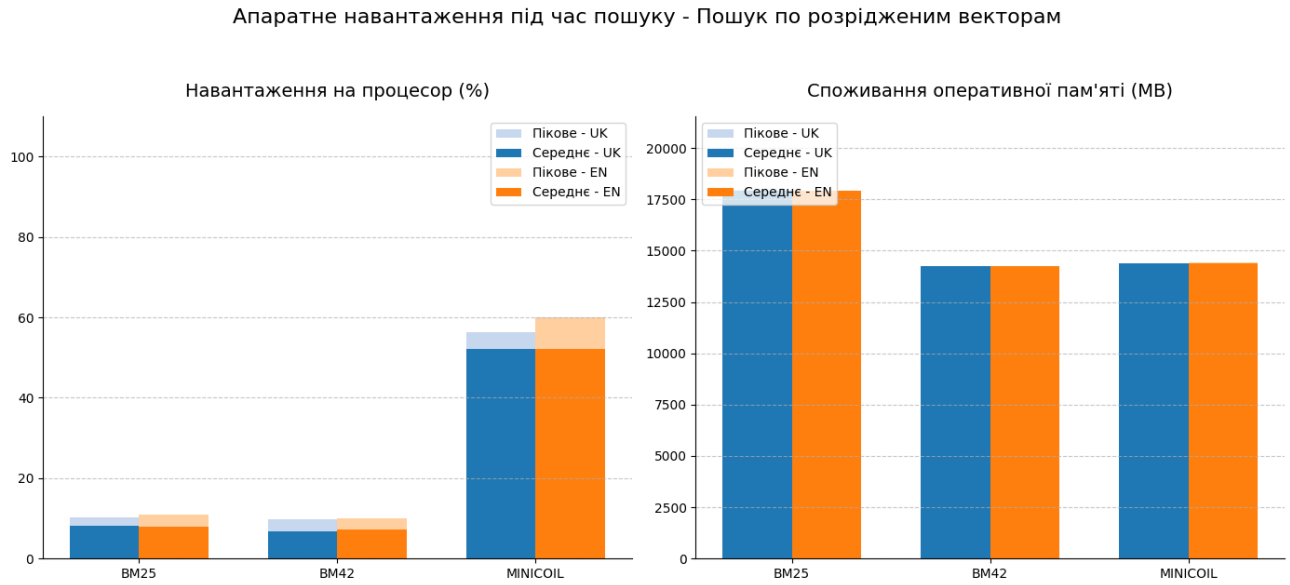

Рисунок 4.32 - Апаратне навантаження підчас крос-мовного пошуку по розрідженим векторам

Аналіз графіків точності (див. Рисунок 4.27, Рисунок 4.30) демонструє дві кардинально різні картини залежно від складності завдання:

– одномовний пошук в англомовному середовищі всі моделі показали стабільно високий результат (Hit Rate >91%), однак на україномовному наборі даних ситуація змінилася: MINICOIL (69,2%) показав результат, ідентичний класичному BM25 (69%), тоді як модель BM42 продемонструвала значне падіння точності – лише 59,2%. Це свідчить про те, що BM42 гірше адаптований до специфіки української мови;

– крос-мовний пошук довів ключове обмеження всіх без винятку розріджених моделей. Всі вони продемонстрували катастрофічне падіння Hit Rate до рівня 14 - 17%. Це доводить, що жодна з обраних розріджених моделей не здатна самостійно долати мовний бар’єр, оскільки їхній механізм все одно базується на співставленні токенів (ваг слів), а не на єдиному семантичному просторі.

Для оцінки доцільності впровадження альтернативних моделей було проаналізовано також їхню апаратну ефективність та пропускну здатність під час виконання пошуку (див. Рисунок 4.28, Рисунок 4.29, Рисунок 4.31, Рисунок 4.32):

– пропускна здатність: BM25 залишається недосяжним еталоном швидкості (340 - 400+ запитів/с), модель BM42 демонструє прийнятне падіння продуктивності (230 - 260 запитів/с), а MINICOIL працює найповільніше (лише 70 - 110 запитів/с);

– споживання ресурсів: модель BM42 виявилася несподівано економною щодо оперативної пам’яті (~14 ГБ), обійшовши навіть BM25 (~18 ГБ), при цьому обидві моделі майже не навантажують процесор (<10%), натомість MINICOIL, споживаючи близько 14,5 ГБ пам’яті, генерує постійне високе навантаження на процесор (понад 50%), що пояснює її низьку пропускну здатність.

Аналіз ізольованої роботи розріджених моделей доводить, що перехід на аналоги (BM42, MINICOIL) не виправдовує себе за відсутності гібридизації.

4.7 Дослідження точності гібридного пошуку: краща модель генерації щільних векторів + моделі генерації розріджених векторів

В даному дослідженні було проведено оцінку гібридного пошуку з комбінаціями моделей E5 та аналогами BM25: BM42 та MINICOIL. З метою забезпечення спадковості експерименту, до аналізу також було включено еталонну комбінацію E5+BM25 та JINA+BM25.

Для об’єктивної оцінки якості інформаційного пошуку використовувалися ті самі метрики (Hit Rate (%) та MRR×100), що і в попередніх дослідженнях пошуку.

Результати мономовного та крос-мовного тестування наведено на рисунках 4.33 - 4.38.

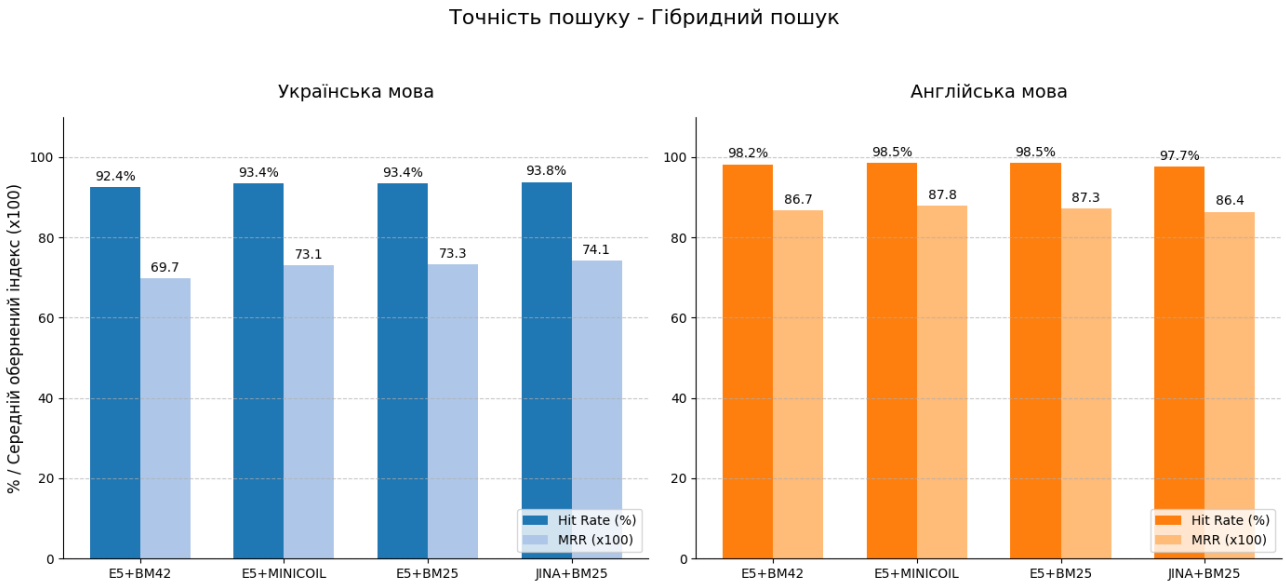

Рисунок 4.33 - Точність гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

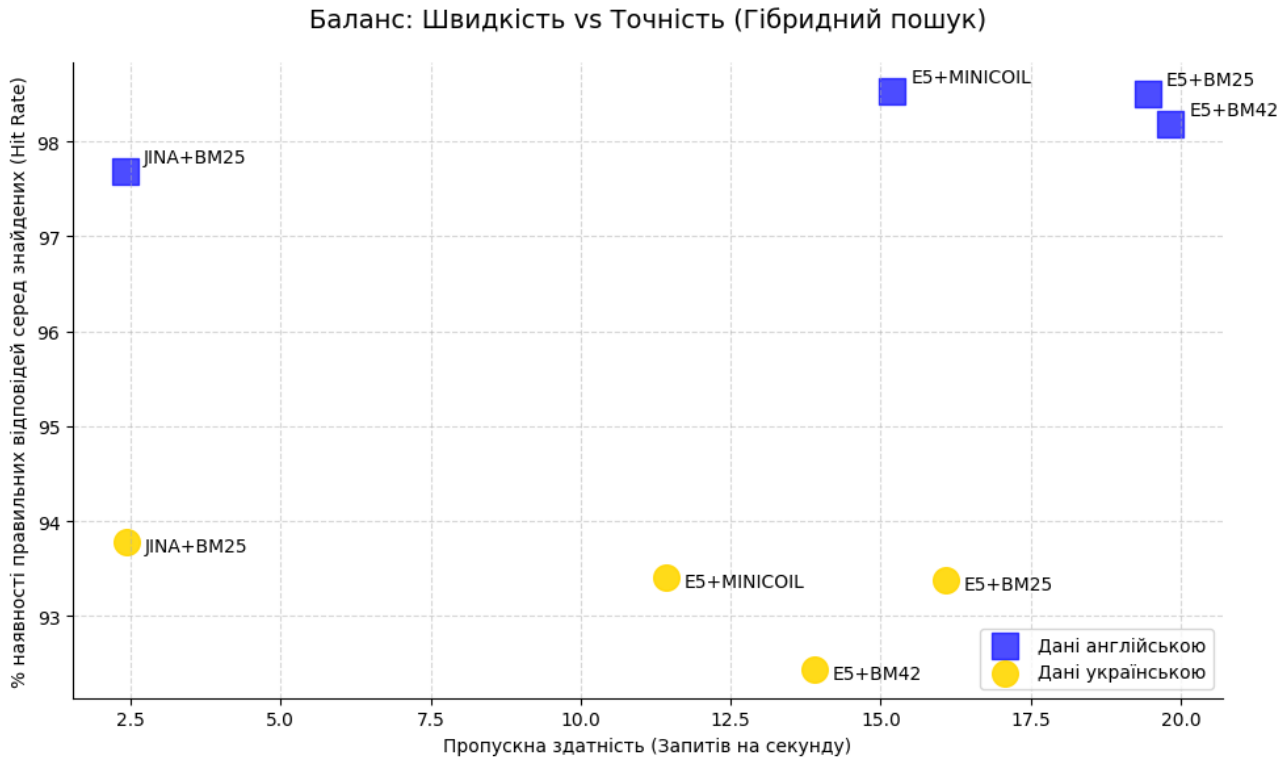

Рисунок 4.34 - Баланс між швидкістю та точністю гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

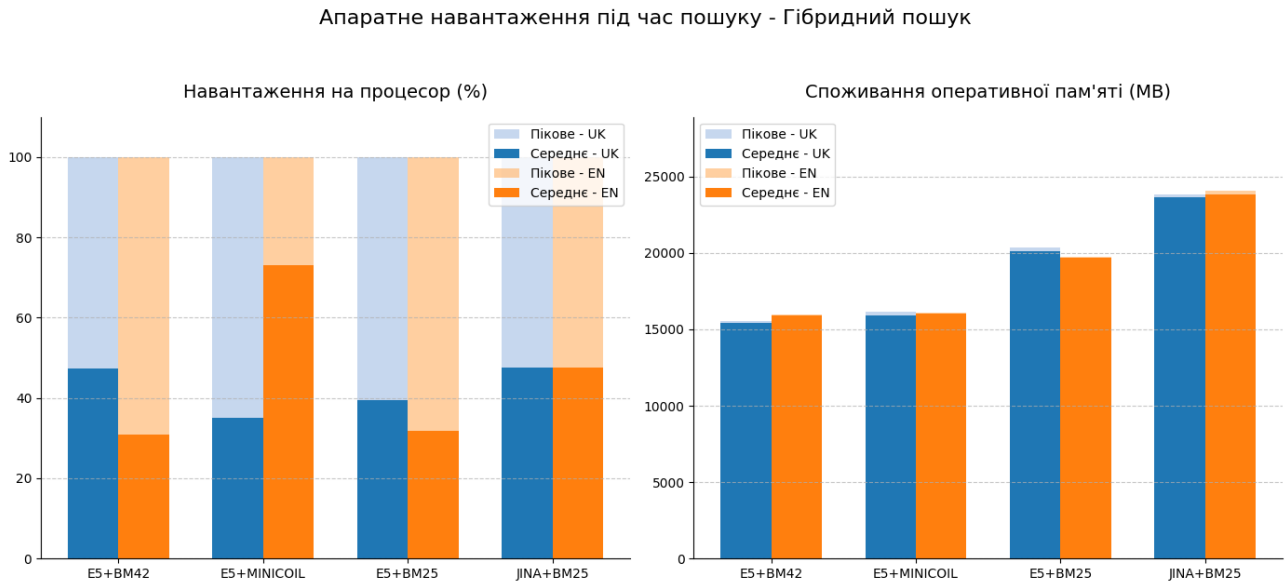

Рисунок 4.35 - Апаратне навантаження підчас гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

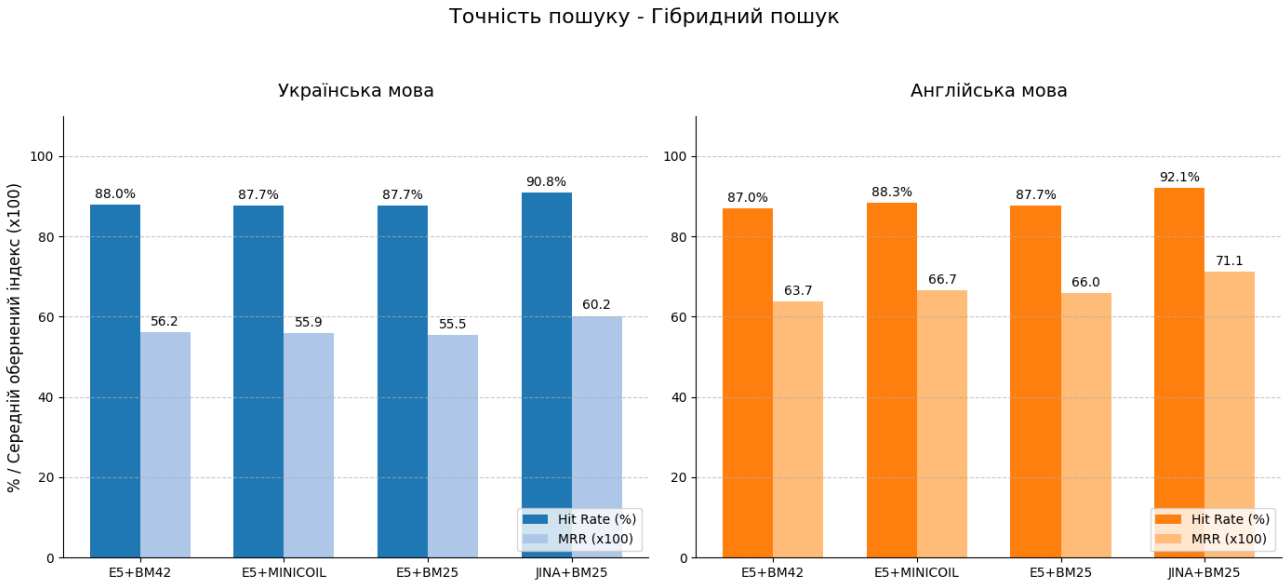

Рисунок 4.36 - Точність крос-мовного гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

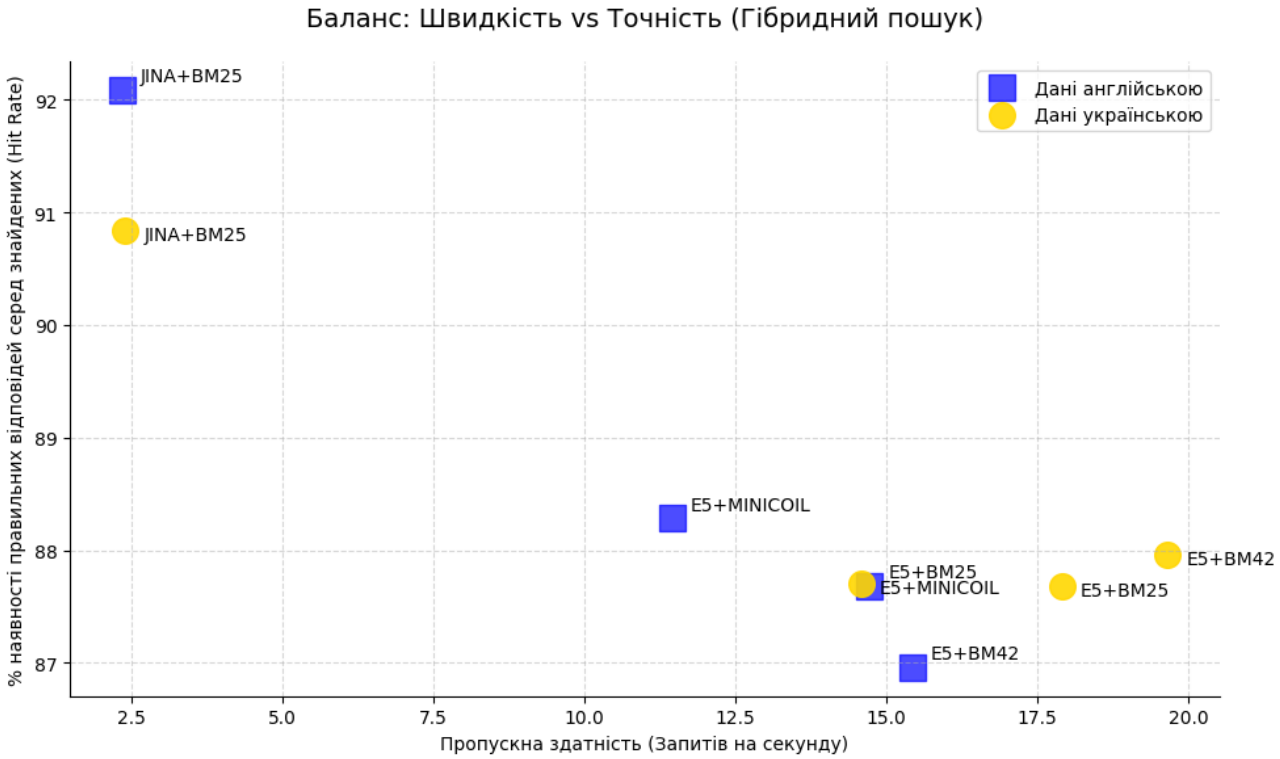

Рисунок 4.37 - Баланс між швидкістю та точністю крос-мовного гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

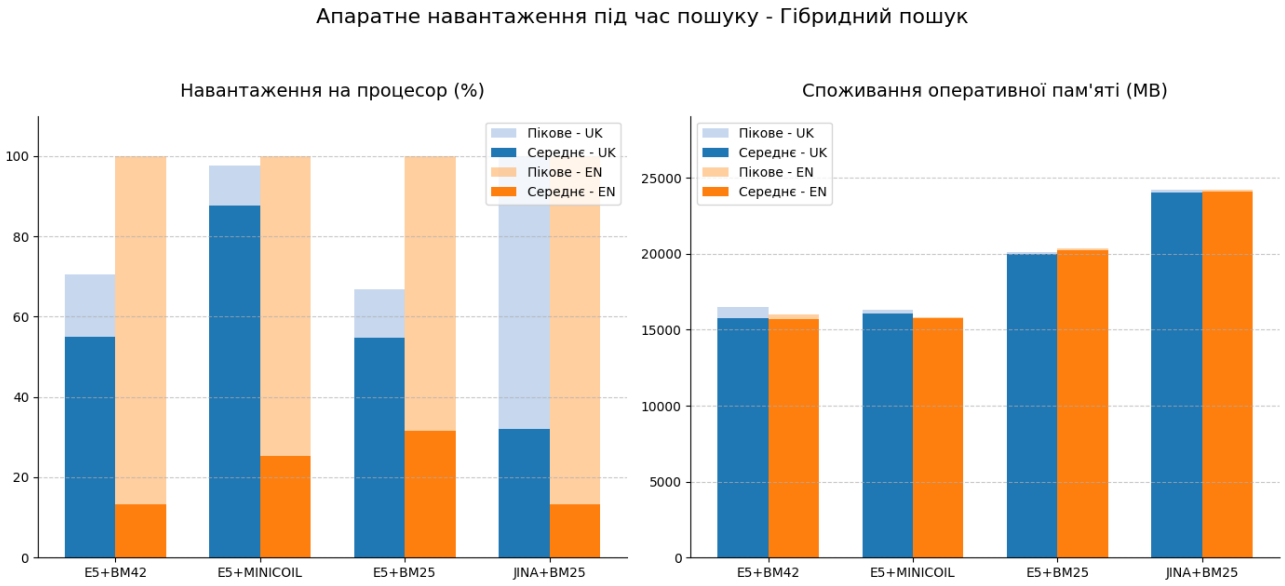

Рисунок 4.38 - Апаратне навантаження підчас крос-мовного гібридного пошуку по розрідженим векторам та E5 (додатково JINA+BM25)

Порівняльний аналіз виявив, що використання альтернативних розріджених векторів не дає приросту точності або скорочення швидкості:

– мономовний пошук (див. Рисунок 4.33): комбінація E5+MINICOIL продемонструвала результати, абсолютно ідентичні до класичного гібрида E5+BM25 (Hit Rate 98,5% для англійської та 93,4% для української мов). Використання E5+BM42 показало мінімальне зниження точності на 1% (до 92,4% в україномовному та 98,2% в англомовному наборах);

– крос-мовний пошук (див. Рисунок 4.36): у багатомовному середовищі результати всіх гібридів на базі E5 зрівнялися в межах статистичної похибки (87 - 88,3%). Це вкотре підтверджує раніше виявлену закономірність: під час крос-мовних запитів вага розріджених моделей під час злиття рангів (Rank Fusion) нівелюється, і пошук фактично спирається виключно на щільні вектори E5. JINA+BM25 утримує лідерство (90,8% - 92,1%), але залишається поза конкуренцією через малу швидкість та велике споживання ресурсів.

Найцікавіші результати були зафіксовані під час моніторингу апаратного навантаження та заміру пропускної здатності:

– пропускна здатність (див. Рисунок 4.34, Рисунок 4.37): швидкість пошуку для всіх комбінацій на базі E5 залишилася практично незмінною (~15-20 запитів на секунду). Це пояснюється тим, що «вузьким місцем» гібридного пошуку є час обчислення щільних векторів моделлю E5, тому заміна розрідженого компонента не сильно впливає на загальний час відповіді;

– економія оперативної пам’яті (див. Рисунок 4.35, Рисунок 4.38): альтернативні розріджені моделі виявилися значно ефективнішими за класичний BM25 саме на етапі пошуку. Якщо еталонний гібрид E5+BM25 споживає близько 20 ГБ оперативної пам’яті, то комбінації E5+BM42 та E5+MINICOIL стабільно використовують лише ~15,5 - 16 ГБ пам’яті під час обробки як мономовних, так і крос-мовних запитів.

Таким чином найкращім варіантом використання є комбінація E5+BM25, що забезпечу високу точність пошуку, помірну ресурсоефективність та задовільну швидкість як додавання чанків так і самого пошуку.

4.8 Дослідження якості генерації відповіді

Оцінка якості генерації відповідей у контексті RAG-системи проводилася на україномовному наборі даних [24], з якого випадковим чином було сформовано вибірку обсягом 50 записів. Для автоматизації процесу перевірки було застосовано сучасну методологію «LLM-as-a-Judge» (модель у ролі судді).

Кожна з досліджуваних локальних моделей, запущених через Ollama, пройшла ітеративне тестування, згенерувавши по 50 відповідей на тестові запити. Оцінювання цих відповідей виконувалося через хмарний API Google AI Studio, що і зумовило використання гібридного («майже повністю локального») підходу до тестування (локальна генерація та хмарна валідація). З метою балансування навантаження та уникнення лімітів квот було налаштовано ротацію хмарних моделей-суддів (gemini-3.1-flash-lite, gemma-4-31b-it та gemma-4-26b-a4b-it), хоча основний обсяг перевірок успішно опрацювала модель gemini-3.1-flash-lite.

Результати проведеного дослідження наведено на рисунках 4.39 - 4.42.

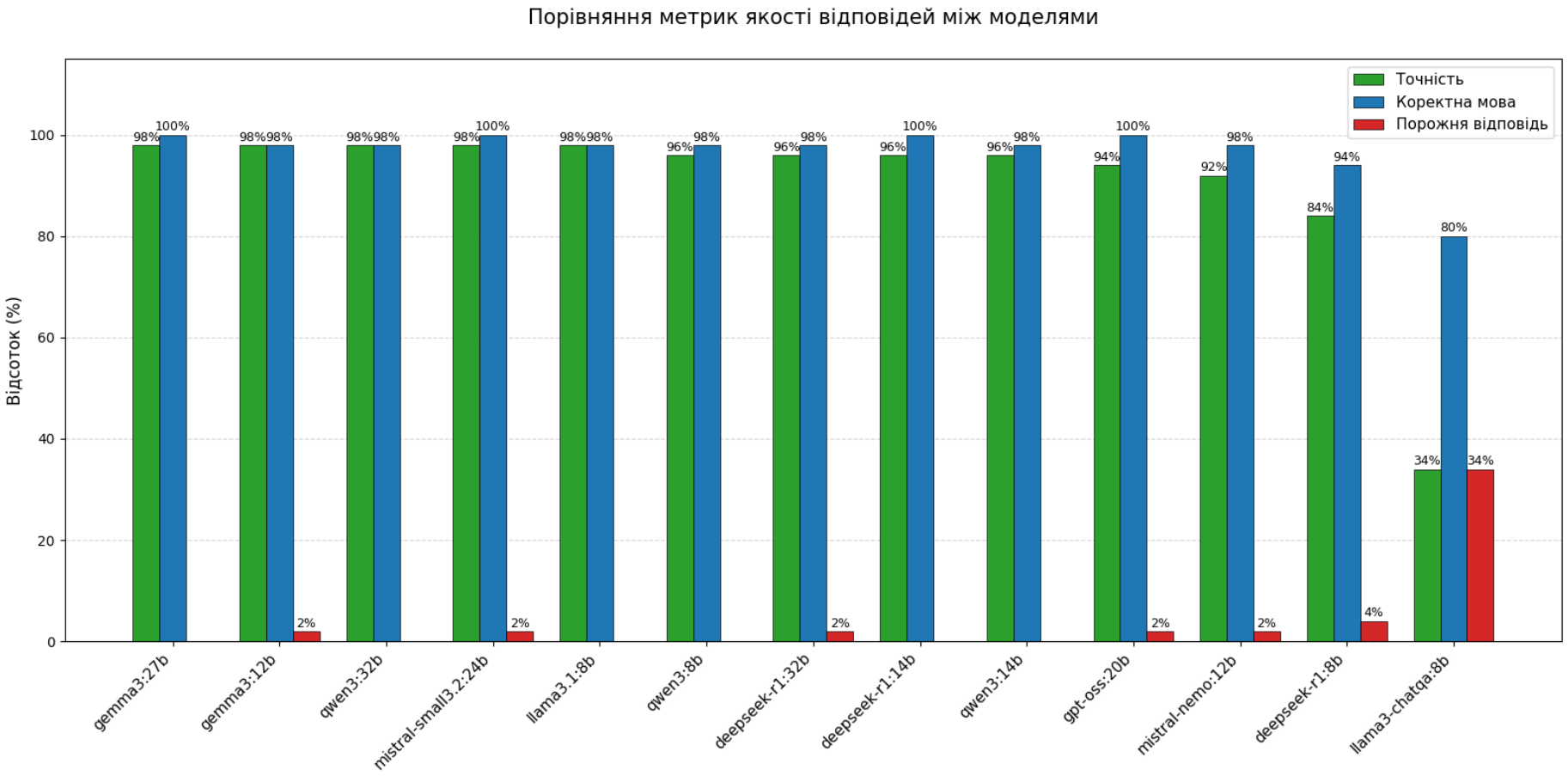

Рисунок 4.39 - Аналіз точності та надійності генерації

Результати перевірки (див. Рисунок 4.39) продемонстрували, що більшість сучасних архітектур добре адаптовані до україномовного контексту. Лідерами за показником точності (98%) стали моделі gemma3:27b, gemma3:12b, qwen3:32b, mistral-small3.2:24b та llama3.1:8b.

Проте під час детального аналізу надійності (див. Рисунок 4.39) було виявлено критичний недолік у деяких моделей: повернення порожньої відповіді. Так, незважаючи на високу загальну точність, модель gemma3:12b у 2% випадків не змогла згенерувати відповідь. Аномально низький результат продемонструвала спеціалізована llama3-chatqa:8b – рівень її точності склав лише 34%, а відсоток порожніх відповідей сягнув 34%, що робить її абсолютно непридатною для поточного завдання. Альтернативні моделі з родин Mistral та GPT показали загальне зниження точності відповідей (до 92-94%) порівняно з лідерами.

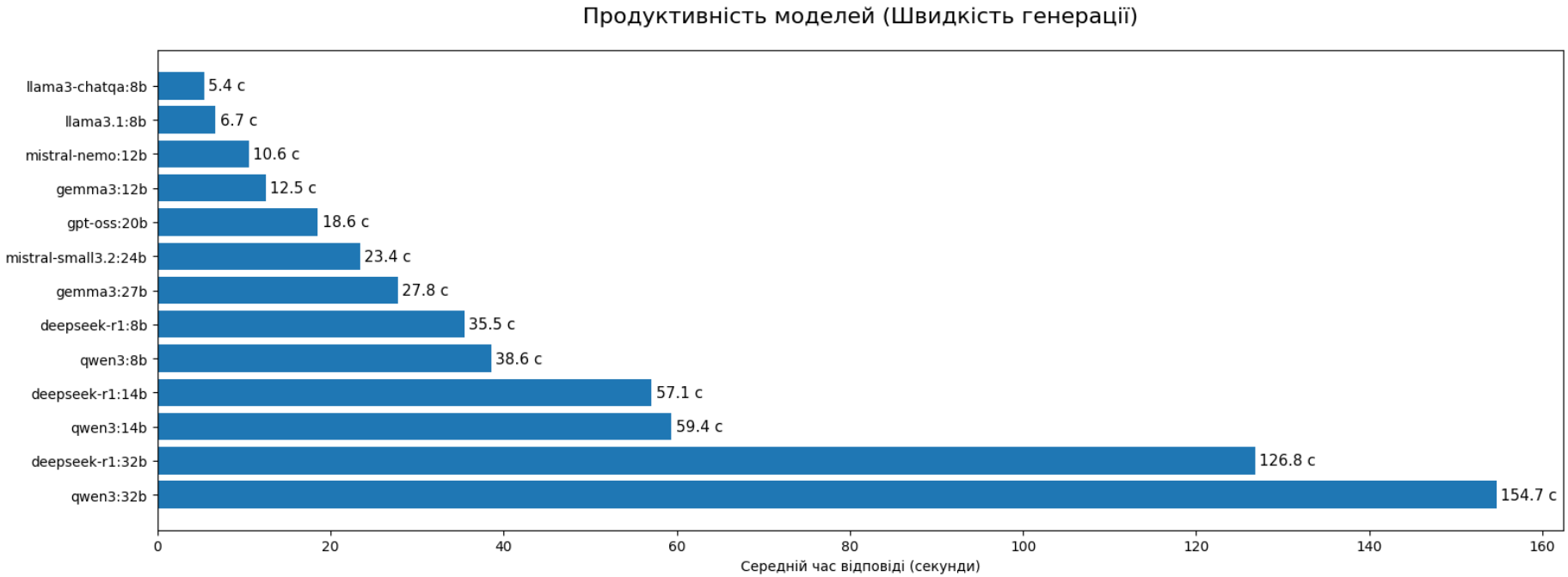

Рисунок 4.40 - Швидкість генерації відповіді (у середньому)

Швидкість генерації (див. Рисунок 4.40) загалом корелює з розміром моделі. Найважчі моделі qwen3:32b та deepseek-r1:32b очікувано показали найнижчу швидкість – 154,7 та 126,8 секунди відповідно. Також результати

тестування виявили цікаву архітектурну особливість: деякі компактні моделі поступаються за швидкістю генерації значно важчим. Наприклад, порівняно легкі моделі deepseek-r1:8b та 14b (35,5 с і 57,1 с відповідно), а також qwen3:8b та 14b (38,6 с і 59,4 с) відпрацьовують запити повільніше, ніж масивні gemma3:27b (27,8 с) та mistral-small3.2:24b (23,4 с). Це свідчить про те, що архітектура родин Qwen3 та DeepSeek-R1 рідше застосовується для швидкої потокової видачі прямого тексту, яка є типовою для RAG-систем. Таку затримку можна пояснити тим, що ці моделі спроєктовані з акцентом на процеси глибокого міркування та покрокового вирішення складних завдань.

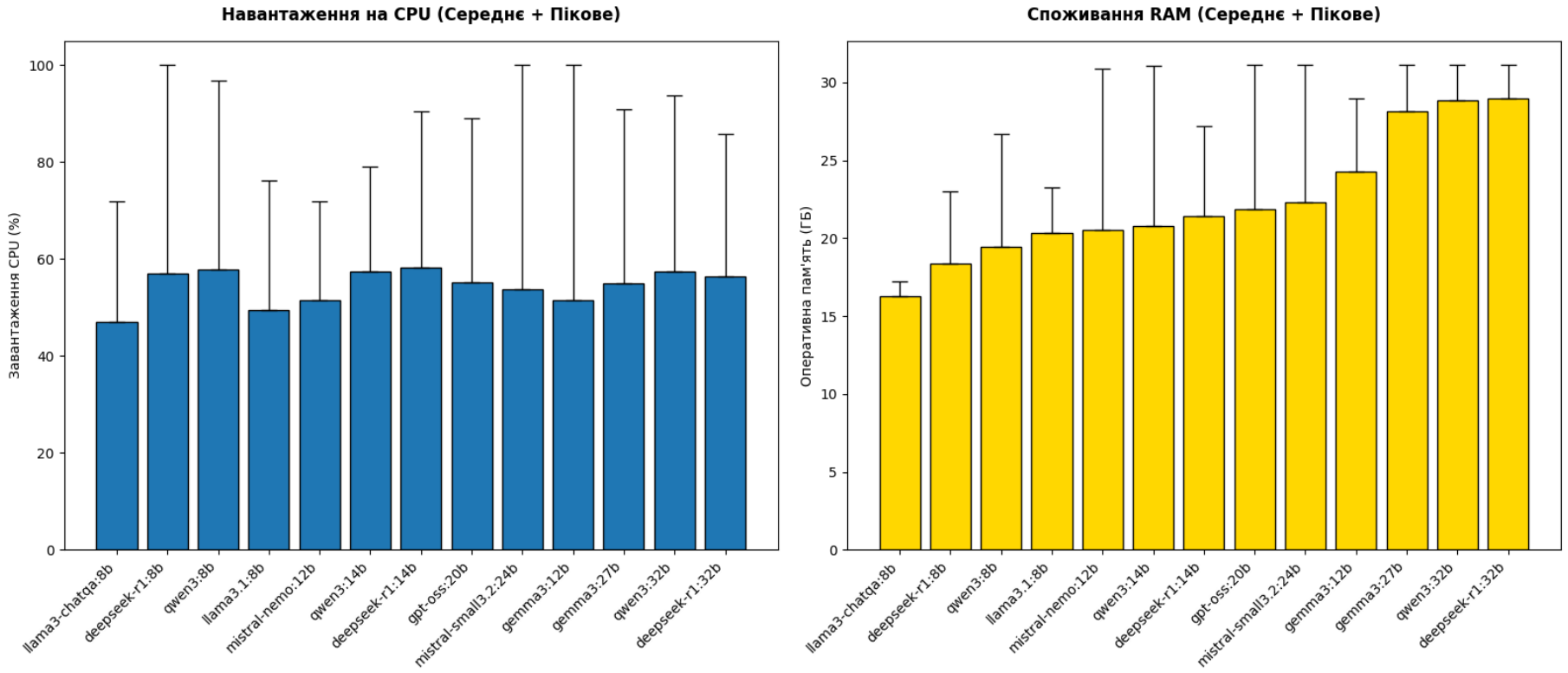

Рисунок 4.41 - Апаратне навантаження генерації відповіді (у середньому)

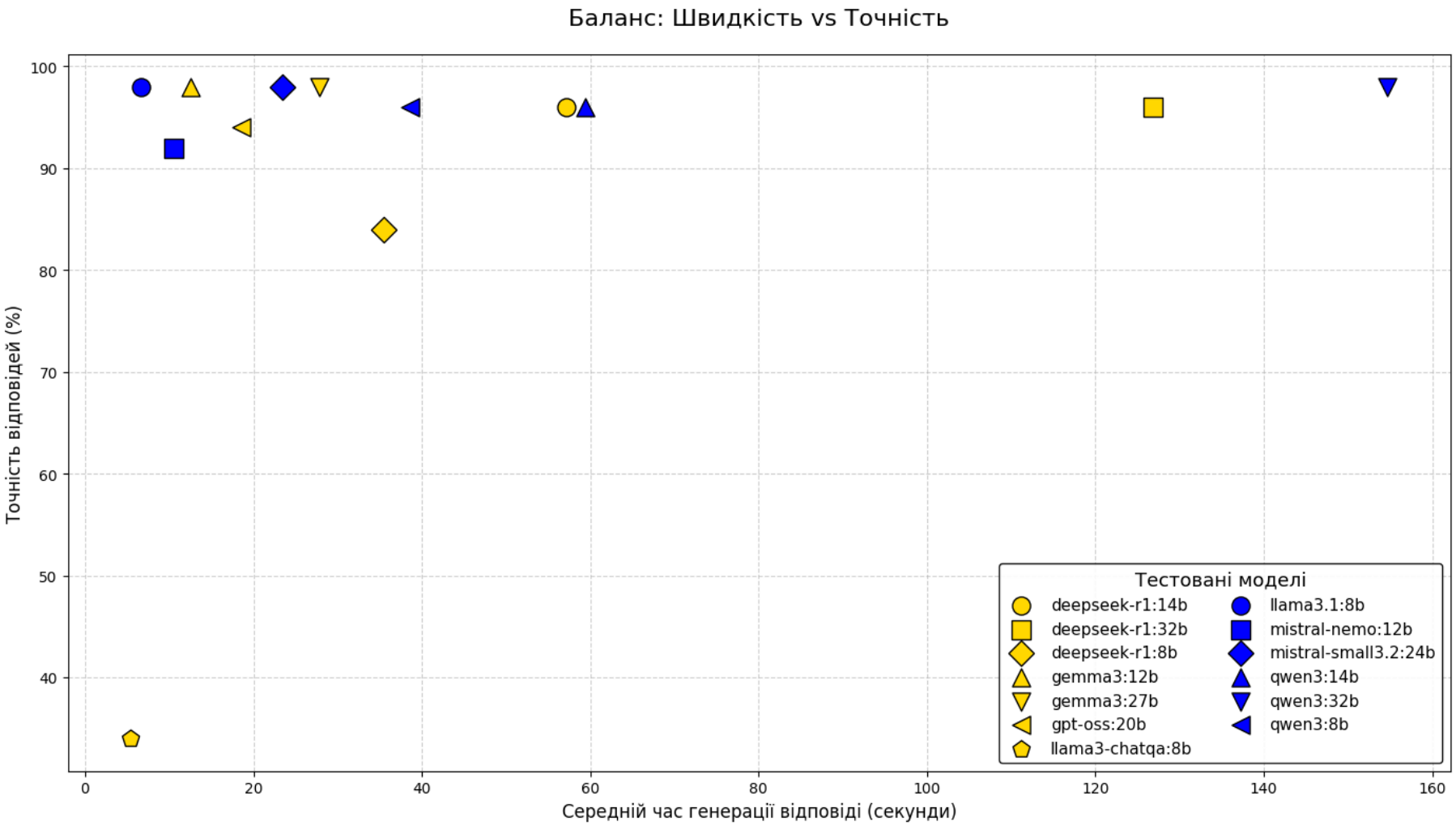

Рисунок 4.42 - Баланс швидкості і точності генерації відповіді

Моніторинг апаратного забезпечення (див. Рисунок 4.41) дозволяє пояснити причини затримок під час генерації: споживання оперативної пам’яті має пряму залежність від кількості параметрів моделі. Якщо компактні моделі класу 8b споживають у середньому 16-20 ГБ, залишаючи необхідний резерв для фонових процесів ОС, то масивні 32b-моделі вичерпують апаратний ліміт системи (32 ГБ), що неминуче призводить до використання файлу підкачки та критичного падіння швидкості обміну даними. Водночас середнє навантаження на центральний процесор для більшості моделей залишається відносно стабільним на рівні 50-60%, проте фіксуються короткочасні пікові стрибки до 96-100%.

Для визначення найкращого кандидата було проаналізовано діаграму розсіювання (див. Рисунок 4.42), що візуалізує компроміс між часом генерації та точністю. За сукупністю метрик було виділено двох фаворитів:

1) llama3.1:8b (абсолютний лідер): Забезпечує ідеальний баланс продуктивності та якості. Модель досягла точності 98% при відсутності порожніх відповідей (0%), маючи при цьому найменший середній час генерації – 6,7 с;

2) gemma3:27b (надійна потужна альтернатива): На відміну від своєї меншої версії 12b, ця модель продемонструвала 100% стабільність (жодної порожньої відповіді) при аналогічній точності 98%. Хоча час її генерації становить 27,8 секунди, це є цілком прийнятним компромісом для архітектур, де потрібна більша глибина параметрів для складного логічного висновку.

ВИСНОВКИ

У кваліфікаційній роботі було успішно вирішено актуальну науково-практичну задачу – розроблено та досліджено програмні засоби для автоматизованого оброблення текстових даних на базі RAG-методології. Відповідно до поставленої мети, було послідовно виконано всі заплановані завдання.

На початковому етапі було проведено ґрунтовний аналіз проблематики в галузі автоматизованої обробки текстових даних. Ознайомлення з існуючими варіантами програмних рішень дозволило об’єктивно проаналізувати їхні сильні та слабкі сторони, що стало надійним теоретичним базисом для проєктування власної архітектури.

На етапі проєктування було розроблено комплексний прототип системи. Для забезпечення чіткого розуміння архітектури та бізнес-логіки розроблено діаграму варіантів використання, детальну функціональну схему та діаграму потоків даних. Було адаптовано базовий алгоритм роботи програми на основі передових галузевих практик побудови RAG-систем, а також імплементовано структуру векторної бази даних із налаштуванням розширеного набору метаданих (payload) для точного вказання джерела фрагменту текста, ефективної фільтрації та пошуку інформації по базі знань.

Після успішної програмної реалізації розробленої системи було проведено ключовий етап роботи – масштабне дослідження базових технологій. Було комплексно досліджено різноманітні моделі генерації векторів (як щільних, так і розріджених) та їхнє функціонування в архітектурі гібридного пошуку, а також проаналізовано роботу локальних LLM. Встановлено вплив як самих моделей так і їх розмірності на якість та релевантність згенерованих відповідей, пропускну здатність системи та апаратне навантаження (споживання оперативної пам’яті та навантаження на процесора).

Розроблена система забезпечує ґрунтовний технічний базис і демонструє високий потенціал для подальшого масштабування. До найбільш перспективних напрямків розвитку, реалізацію яких наразі залишено для майбутніх досліджень, належить розробка механізму формування точних посилань на першоджерела шляхом генерації структурованих JSON-відповідей мовною моделлю. Ще одним важливим кроком є впровадження модуля обробки графічної інформації за допомогою мультимодальних LLM для автоматичного створення текстових описів зображень (графіків, схем, ілюстрацій) та їх інтеграції в текстові чанки. Такі архітектурні рішення дозволять векторній базі даних ефективно працювати з візуальним контекстом документів, а самій системі – гарантувати фактологічну достовірність відповідей.

ПЕРЕЛІК ПОСИЛАНЬ

[1] 1up, "Your Enterprise Knowledge Base Is Failing - Here's Why," 1up, 6 March 2026. [Online]. Available: https://www.linkedin.com/pulse/your-enterprise-knowledge-base-failing-heres-why-1up-ai-vn2xc. [Accessed 10 May 2026].

[2] S. Sarangdhar, "Knowledge Automation: How Top Companies Deliver Answers at Scale," 1up, 29 April 2026. [Online]. Available: https://1up.ai/blog/knowledge-automation. [Accessed 10 May 2026].

[3] OpenAI, "Optimizing LLM Accuracy," OpenAI, [Online]. Available: https://platform.openai.com/docs/guides/optimizing-llm-accuracy#retrieval-augmented-generation-rag. [Accessed 20 January 2026].

[4] Google, «Порівняння цін на тарифні плани | Google Workspace,» Google, [Онлайновий]. Available: https://workspace.google.com/pricing.html. [Дата звернення: 20 березень 2026].

[5] OpanAI, «Плани ChatGPT | Free, Go, Plus, Pro, Business та Enterprise,» ChatGPT, [Онлайновий]. Available: https://chatgpt.com/uk-UA/pricing/. [Дата звернення: 20 березень 2026].

[6] InfiniFlow, "infiniflow/ragflow: RAGFlow is a leading open-source Retrieval-Augmented Generation (RAG) engine that fuses cutting-edge RAG with Agent capabilities to create a superior context layer for LLMs," InfiniFlow, [Online]. Available: https://github.com/infiniflow/ragflow. [Accessed 20 March 2026].

[7] Qdrant, "Points, Vectors and Payloads - Qdrant," Qdrant, [Online]. Available: https://qdrant.tech/course/essentials/day-1/embedding-models/. [Accessed 1 May 2026].

[8] A. Javanmard, "PDF Data Extraction Benchmark 2025: Comparing Docling, Unstructured, and LlamaParse for Document Processing Pipelines," Procycons, 24 March 2025. [Online]. Available: https://procycons.com/en/blogs/pdf-data-extraction-benchmark/. [Accessed 18 February 2026].

[9] "What is the best vector database?," reddit, 2024. [Online]. Available: https://www.reddit.com/r/vectordatabase/comments/1d4kz2p/what_is_the_best_vector_database/. [Accessed 20 February 2026].

[10] Qdrant, "Qdrant - Vector Search Engine," Qdrant, [Online]. Available: https://qdrant.tech/. [Accessed 20 February 2026].

[11] Qdrant, "The Universal Query API - Qdrant," Qdrant, [Online]. Available: https://qdrant.tech/course/essentials/day-5/universal-query-api/. [Accessed 10 May 2026].

[12] Qdrant, "Demo: Keyword Search with Sparse Vectors - Qdrant," Qdrant, [Online]. Available: https://qdrant.tech/course/essentials/day-3/sparse-retrieval-demo/. [Accessed 10 May 2026].

[13] A. Vasnetsov, "BM42: New Baseline for Hybrid Search - Qdrant," Qdrant, 1 July 2024. [Online]. Available: https://qdrant.tech/articles/bm42/. [Accessed 20 May 2026].

[14] "llama3-chatqa," Ollama, 10 May 2024. [Online]. Available: https://ollama.com/library/llama3-chatqa. [Accessed 25 February 2026].

[15] "llama3.1," Ollama, 30 November 2024. [Online]. Available: https://ollama.com/library/llama3.1. [Accessed 25 February 2026].

[16] "deepseek-r1," Ollama, 2 July 2025. [Online]. Available: https://ollama.com/library/deepseek-r1. [Accessed 25 February 2026].

[17] "qwen3," Ollama, 10 October 2025. [Online]. Available: https://ollama.com/library/qwen3. [Accessed 25 February 2026].

[18] "mistral-nemo," Ollama, 23 July 2025. [Online]. Available: https://ollama.com/library/mistral-nemo. [Accessed 25 February 2026].

[19] "gemma3," Ollama, 5 December 2025. [Online]. Available: https://ollama.com/library/gemma3. [Accessed 25 February 2026].

[20] "gpt-oss," Ollama, 9 October 2025. [Online]. Available: https://ollama.com/library/gpt-oss. [Accessed 25 February 2026].

[21] "mistral-small3.2," Ollama, 20 June 2025. [Online]. Available: https://ollama.com/library/mistral-small3.2. [Accessed 25 February 2026].

[22] M. B. Gürbüz, "8 Best Tools to Run LLMs Locally, Ranked [2026] | TECHSY," TECHSY, 29 April 2026. [Online]. Available: https://techsy.io/en/blog/best-tools-run-llms-locally. [Accessed 25 May 2026].

[23] GAddicts, "G37A/ua-retrieval-80k-silver · Datasets at Hugging Face," Hugging Face, 23 March 2026. [Online]. Available: https://huggingface.co/datasets/G37A/ua-retrieval-80k-silver. [Accessed 24 May 2026].

[24] HPLT, "HPLT/ua-squad · Datasets at Hugging Face," HuggingFace, 24 April 2025. [Online]. Available: https://huggingface.co/datasets/HPLT/ua-squad. [Accessed 25 May 2026].