НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри

Артем ВОЛОКИТА
(підпис)

“ ” 2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

Виконав : студент 4 курсу, групи ІО-з21
(шифр групи)

_____Булах Леонід Ігорович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник ас., д-р філос. Коренко Д. В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас., д-р філос. Коренко Д. В.

_____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ст.викл. каф. СПіСКС, д-р філос. Костянтин РАДЧЕНКО

_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
Артем ВОЛОКИТА
(підпис)
“ ” 2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Булаха Леоніда Ігоровича

1. Тема проєкту Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту керівник проєкту Коренко Дмитро В., асистент, доктор філософії
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «03» червня 2026 року № 2055-с

2. Термін здачі студентом закінченого проєкту 17 травня 2026 р.

3. Вихідні дані до проєкту: технічна документація на PDF-формат; наукові публікації з тематики Retrieval-Augmented Generation; відкриті бібліотеки PyMuPDF, sentence-transformers, FAISS, rank-bm25, transformers; набір тестових PDF-документів для оцінювання якості ретриверу.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області та огляд існуючих рішень.

Розділ 2. Проєктування системи.

Розділ 3. Реалізація програмної системи.

Розділ 4. Тестування та експериментальна оцінка.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень): схема структурна системи; схема функціональна конвеєра індексації; схема структурна конвеєра запитів; діаграма класів доменної моделі.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	ас., д-р філос. Коренко Д. В.		

7. Дата видачі завдання «10» січня 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1	*Затвердження теми проєкту*	*01.12.2025–15.12.2025*	
2	*Аналіз предметної області та*	*16.12.2025–31.01.2026*	
3	*Формулювання вимог та проєктування архітектури*	*01.02.2026–28.02.2026*	
4	*Реалізація конвеєра індексації*	*01.03.2026–31.03.2026*	
5	*Реалізація конвеєра запитів та*	*01.04.2026–19.04.2026*	
6	*Переддипломна практика,*	*20.04.2026–17.05.2026*	
7	*Оформлення пояснювальної записки*	*18.05.2026–05.06.2026*	
8	*Передзахист*		
9	*Захист*	*20.06.2026*	

Студент-дипломник Леонід БУЛАХ
(підпис)

Керівник проєкту Дмитро КОРЕНКО
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проєкті розроблено програмну систему мультимодального RAG для технічних PDF-документів, що поєднує автоматичну екстракцію тексту, таблиць та зображень з гібридним пошуковим конвеєром, реранкінгом моделі cross-encoder та генерацією відповідей з прозорим зазначенням джерел; архітектура побудована за принципами чистої архітектури та забезпечує модульність і можливість підміни реалізацій; на тестовому наборі з 50 запитів повний конвеєр досягає Recall@5 ≈ 0.92, що перевищує одиничні методи на 5–18 п.п.

Ключові слова: RAG, гібридний пошук, BM25, FAISS, реранкінг, велика мовна модель, PDF, Python.

ANNOTATION

This bachelor diploma project presents a software system for multimodal RAG over technical PDF documents, combining automatic extraction of text, tables and images with a hybrid retrieval pipeline, cross-encoder reranking and answer generation with transparent source citations; the architecture follows Clean Architecture principles, ensuring modularity and the ability to swap implementations; on a benchmark of 50 queries the full pipeline reaches Recall@5 ≈ 0.92, outperforming individual methods by 5–18 p.p.

Key words: RAG, hybrid retrieval, BM25, FAISS, reranking, large language model, PDF, Python.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.466500.002 ТЗ***	Система обробки технічних	4		
			документів з таблицями і			
			рисунками з використанням			
			мультимодальних моделей			
			штучного інтелекту			
			Технічне завдання			
	А4	***ІАЛЦ.466500.003 ПЗ***	Пояснювальна записка	72		
	А4	***ІАЛЦ.466500.004 Д1***	Схема структурна системи	1		
	А4	***ІАЛЦ.466500.005 Д2***	Схема функціональна	1		
			конвеєра індексації			
	А4	***ІАЛЦ.466500.006 Д3***	Схема структурна конвеєра	1		
			запитів			
	А4	***ІАЛЦ.466500.007 Д4***	Діаграма класів	1		

Зм	*Лист*	*№ докум.*	*Підп*	*Дата*	

ІАЛЦ.466500.001 ОА

Розроб		Булах Л. І.			*Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Коренко Д. В.					
						“КПІ” ФІОТ ІО-з21	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку програмної системи мультимодального RAG (Retrieval-Augmented Generation) для технічних PDF-документів з гібридним пошуком та інтеграцією локальних і хмарних LLM.

Областю застосування системи є інформаційний пошук та автоматизована обробка технічної документації: інженерних стандартів, наукових публікацій, специфікацій обладнання та внутрішньої документації підприємств.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр», який був затверджений кафедрою обчислювальної техніки факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка програмної системи, що забезпечує семантичний пошук за змішаним вмістом технічних PDF-документів (текст, таблиці, зображення) та генерацію відповідей природною мовою з обов’язковими посиланнями на джерела.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проєкту є наукові публікації з тематики Retrieval-Augmented Generation та інформаційного пошуку, технічна документація на формат PDF, офіційна документація відкритих бібліотек PyMuPDF, sentence-transformers, FAISS, rank-bm25, transformers та FastAPI.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Завантаження та індексація PDF-документів через CLI або REST API.
- Екстракція тексту, таблиць та зображень зі збереженням координат розташування на сторінці.
- Гібридний пошук (FAISS + BM25) з об'єднанням результатів методом Reciprocal Rank Fusion та реранкінгом cross-encoder моделлю.
- Генерація відповідей через хмарну (OpenAI gpt-4o-mini) або локальну (microsoft/phi-2) LLM з обов'язковими посиланнями на джерела.
- Збереження та відновлення стану індексу; медіана часу обробки запиту не більше 5 с; Recall@5 не менше 0.85.

5.2. Вимоги до програмного забезпечення

- ОС Windows 10/11 або Linux.
- Python версії 3.11 або вище; Docker (опційно, для контейнеризованого розгортання).

5.3. Вимоги до апаратної частини

- ЦП не нижче ніж Intel Core i5 або еквівалент.
- ROM не менше ніж 20 ГБ вільного простору.
- RAM не менше ніж 8 ГБ (16 ГБ рекомендовано для локальної LLM).

Назва етапів виконання	Термін виконання
Затвердження теми та аналіз предметної області	31.01.2026
Формулювання вимог та проєктування архітектури	28.02.2026
Реалізація конвеєра індексації	31.03.2026
Реалізація конвеєра запитів та REST API	19.04.2026

Тестування та доопрацювання системи	17.05.2026
Оформлення пояснювальної записки	05.06.2026
Передзахист дипломного проєкту	
Захист дипломного проєкту	20.06.2026

6 ЕТАПИ РОЗРОБКИ

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – Artificial Intelligence (штучний інтелект)

API – Application Programming Interface (прикладний програмний інтерфейс)

BLIP – Bootstrapping Language-Image Pre-training (модель опису зображень)

BM25 – Best Matching 25 (алгоритм розрідженого пошуку Okapi BM25)

CLI – Command Line Interface (інтерфейс командного рядка)

CPU – Central Processing Unit (центральний процесор)

CRUD – Create / Read / Update / Delete (базові операції зі сховищем)

FAISS – Facebook AI Similarity Search (бібліотека швидкого пошуку у векторних просторах)

GPU – Graphics Processing Unit (графічний процесор)

HTTP – Hypertext Transfer Protocol

JSON – JavaScript Object Notation

LLM – Large Language Model (велика мовна модель)

MRR – Mean Reciprocal Rank (середнє з обернених рангів)

nDCG – normalized Discounted Cumulative Gain

NLP – Natural Language Processing (обробка природної мови)

OOP – Object-Oriented Programming

PDF – Portable Document Format

RAG – Retrieval-Augmented Generation (генерація з доповненим пошуком)

REST – Representational State Transfer

RRF – Reciprocal Rank Fusion

TF-IDF – Term Frequency – Inverse Document Frequency

UI – User Interface

UUID – Universally Unique Identifier

ПЗ – Програмне забезпечення

ВСТУП

Технічна документація сьогодні переважно публікується у форматі PDF: інженерні стандарти, наукові статті, специфікації обладнання, описи API та звіти про дослідження. Такі документи містять не лише суцільний текст, а й таблиці, ілюстрації, формули та схеми. Обсяг такої документації подвоюється приблизно кожні три роки, і традиційні засоби повнотекстового пошуку дедалі гірше справляються із задачею оперативного отримання потрібної інформації.

Класичні системи інформаційного пошуку, побудовані на статистичних моделях TF-IDF та BM25, добре опрацьовують лексичні збіги, проте погано «розуміють» зміст запиту, не враховують контекст і неспроможні працювати з вмістом, представленим у вигляді зображень або таблиць. Великі мовні моделі (LLM) уможливили генерацію природномовних відповідей, проте мають обмежений контекст, схильні до галюцинацій і не оновлюють свої знання після тренування.

Парадигма Retrieval-Augmented Generation (RAG), сформульована у роботі Lewis et al. (2020), дозволяє поєднати ці два підходи: до моменту генерації відповіді LLM отримує найбільш релевантні фрагменти знань з зовнішньої бази. У такому підході вузьким місцем стає саме етап ретриверу: якщо потрібний фрагмент не потрапить до контексту, навіть найкраща LLM не зможе сформулювати правильну відповідь.

Особливим викликом є застосування RAG до технічних PDF-документів зі змішаним вмістом. Значна частка корисної інформації закодована у вигляді таблиць та ілюстрацій. Простий експорт PDF у плоский текст призводить до втрати структури таблиць та цілковитої втрати інформативного навантаження зображень: запит «який максимальний струм має транзистор Q1» залишається без відповіді, коли цей струм наведено лише в таблиці, а питання «що зображено на рис. 3.2» – без відповіді, оскільки зображення не індексується.

Актуальність роботи зумовлена потребою у спеціалізованій програмній системі, яка б поєднувала автоматичну екстракцію тексту, таблиць та зображень з технічних PDF, побудову гібридного пошукового індексу, сучасний реранкінг кандидатів та генерацію кінцевих відповідей за допомогою LLM з прозорим посиланням на джерела. Існуючі універсальні фреймворки (LangChain, LlamaIndex, Haystack) надають базові будівельні блоки, але не вирішують специфічних завдань мультимодального опрацювання технічних документів і вимагають значних зусиль інженера для збирання повноцінного конвеєра.

Мета роботи – розробити програмну систему мультимодального RAG для технічних PDF-документів, що забезпечує екстракцію текстового, табличного та графічного вмісту, побудову гібридного пошукового індексу з реранкінгом та генерацію відповідей з посиланнями на джерела, придатну для використання як через REST API, так і через інтерфейс командного рядка.

Для досягнення поставленої мети необхідно розв'язати такі задачі: проаналізувати предметну область retrieval-augmented generation, методи інформаційного пошуку та існуючі рішення; обґрунтувати вибір технологічного стеку та архітектурного підходу; спроєктувати доменну модель і структурну схему системи; реалізувати конвеєр індексації та конвеєр обробки запитів; створити REST API та інтерфейс командного рядка; провести модульне й інтеграційне тестування створеного програмного забезпечення.

Об'єктом дослідження є процес інформаційного пошуку та генерації відповідей у корпусі технічних PDF-документів. Предметом дослідження є методи та програмні засоби побудови мультимодальних RAG-систем, придатних для опрацювання тексту, таблиць та зображень з технічних PDF.

Методи дослідження включають аналіз літератури та існуючих рішень, методи лексичного й семантичного пошуку (BM25, dense retrieval з sentence-transformers), методи реранкінгу (cross-encoder), методи генерації тексту (LLM

на основі transformers), принципи чистої та гексагональної архітектури, методи модульного тестування програмного забезпечення.

Практична цінність роботи полягає у створенні готового до використання інструменту, який дозволяє інженерам, дослідникам та технічним спеціалістам швидко отримувати відповіді на запитання щодо змісту великих корпусів технічних PDF-документів з посиланнями на конкретні джерела (сторінку, рисунок або таблицю). Завдяки модульній архітектурі система може бути інтегрована у внутрішні корпоративні портали, документаційні бази знань, наукові репозиторії та CI/CD-конвеєри.

Наукова новизна роботи полягає в комплексному поєднанні стратегії Reciprocal Rank Fusion для гібридного пошуку, cross-encoder реранкінгу та автоматичного опису ілюстрацій моделлю BLIP у межах єдиного конвеєра, побудованого за принципом чистої архітектури з суворим розділенням доменної моделі, прикладних сценаріїв та адаптерів конкретних технологій.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Загальна характеристика задачі

Технічна документація є основним способом передачі знань між авторами, інженерами та користувачами складних виробів та інформаційних систем. Сучасні стандарти оформлення вимагають публікації документації у вигляді PDF, що поєднує універсальність відображення з збереженням точної верстки. Однак саме особливості формату PDF створюють додаткові труднощі для автоматизованого опрацювання: контент усередині документа фрагментований на низькорівневі графічні примітиви, текст представлений у вигляді послідовностей координат символів, а структурна інформація (заголовки, абзаци, поля таблиць, підписи рисунків) формально не виділена.

Задача інформаційного пошуку у такому корпусі полягає у тому, щоб за природномовним запитом користувача знайти найбільш релевантні фрагменти документа. У контексті технічних PDF задача ускладнюється кількома факторами. По-перше, документи можуть бути великого обсягу, тому повернення цілого документа як відповіді є неінформативним. По-друге, релевантна інформація часто розпорошена за документом: відповідь може потребувати поєднання даних з тексту, таблиці зі специфікацією та підпису під ілюстрацією. По-третє, користувач формулює запит у вигляді запитання природною мовою, а не як набір ключових слів, тому система має враховувати семантичну близькість.

1.2 Класифікація методів інформаційного пошуку

Методи інформаційного пошуку традиційно поділяють на лексичні (sparse, розріджені) та семантичні (dense, густі). Окремо виділяють гібридні підходи, що поєднують переваги обох сімейств.

Класична модель TF-IDF обчислює релевантність документа на основі частоти зустрічаності термінів запиту та оберненої частоти зустрічаності цих термінів у колекції. Її розвиток – модель Okapi BM25 (Robertson, Spärck Jones), що додає насиченість терма та нормалізацію за довжиною документа. BM25 не потребує попереднього навчання моделі та забезпечує субмілісекундний час пошуку, проте принципово не здатний знайти документ, який не містить жодного слова з запиту.

Семантичні (dense) методи побудовані на ідеї проекції тексту в простір векторних представлень фіксованої розмірності. Близькість двох текстів за косинусною мірою трактується як їхня семантична близькість. Сучасні моделі (sentence-transformers, E5) базуються на архітектурі Transformer і навчаються на контрастних задачах. Перевагою є здатність знаходити документи з синонімами і парафразами; зворотна сторона – гірша якість для рідкісних доменних термінів.

Гібридні методи дозволяють отримати найкраще з обох світів. Найпростіша стратегія – незалежне ранжування обома методами та об'єднання результатів за допомогою Reciprocal Rank Fusion (RRF), запропонованого Cormack et al. (2009). Формула RRF для документа d: RRF(d) = сума 1/(k + rank_i(d)) за всіма методами i, де k – згладжувальна константа (зазвичай 60). RRF не вимагає калібрування абсолютних оцінок та має малу кількість параметрів.

1.3 Парадигма Retrieval-Augmented Generation

Парадигма RAG, формалізована у роботі Lewis et al. (2020), долає обмеження LLM шляхом поєднання моделі із зовнішньою базою знань. На вхід LLM подається не лише запит користувача, а й набір найбільш релевантних фрагментів документів, отриманих окремим пошуковим модулем. Це дозволяє: працювати з актуальними знаннями без перенавчання моделі; зменшити частоту галюцинацій; надавати посилання на конкретні джерела; ефективно використовувати обмежений контекст LLM.

Типова RAG-система складається з двох конвеєрів: конвеєра індексації, що готує колекцію документів до пошуку, та конвеєра запитів, що обробляє інтеракції з користувачем. Конвеєр індексації виконує завантаження документів, їх розбиття на фрагменти, побудову векторних представлень та збереження у векторному сховищі. Конвеєр запитів отримує запитання, перетворює його у вектор, шукає найбільш близькі фрагменти, опціонально проводить реранкінг та формує промпт для LLM.

1.4 Особливості мультимодального вмісту технічних PDF

Технічна документація суттєво відрізняється від суцільного тексту художніх чи новинних творів. Типовий технічний документ містить ілюстрації (графіки, креслення, фотографії обладнання), таблиці (специфікації, результати вимірювань), математичні формули, програмний код. За оцінкою авторів роботи Mathew et al. (2022), до 30–40 відсотків інформативного навантаження у науково-технічних публікаціях припадає саме на нетекстові елементи.

Простий експорт PDF у текст призводить до серйозних втрат інформації. Таблиці після експорту перетворюються на хаотичні послідовності клітинок без явних роздільників. Зображення випадають із результату взагалі. Спеціалізовані бібліотеки PyMuPDF, pdfplumber, Camelot дозволяють

зменшити частину цих втрат, проте кожна має свої сильні та слабкі сторони. Для опрацювання зображень використовують моделі автоматичної анотації (caption generation), серед яких BLIP (Li et al., 2022), BLIP-2, CLIP+декодер та LLaVA. Модель BLIP-base проектує зображення у вектор та генерує короткий природномовний опис.

1.5 Огляд існуючих рішень

На ринку відкритого програмного забезпечення представлено кілька популярних фреймворків для побудови RAG-систем. Розглянемо найбільш помітні з них з аналізом їхньої архітектури.

1.5.1 LangChain

LangChain – це Python-бібліотека, що пропонує єдиний інтерфейс до десятків постачальників LLM, векторних сховищ, ретриверів та інших компонентів. Архітектура LangChain побудована навколо концепції «ланцюжків» (chains), що сполучаються через LangChain Expression Language (LCEL). Основний фокус – на побудові ланцюжків взаємодії з LLM (агентів, інструментів, пам'яті), а не на оптимізованому ретривері. Підтримка мультимодальних документів обмежена окремими модулями, що потребують додаткового конфігурування.

Рисунок 1.1 – Архітектура LangChain (узагальнено)

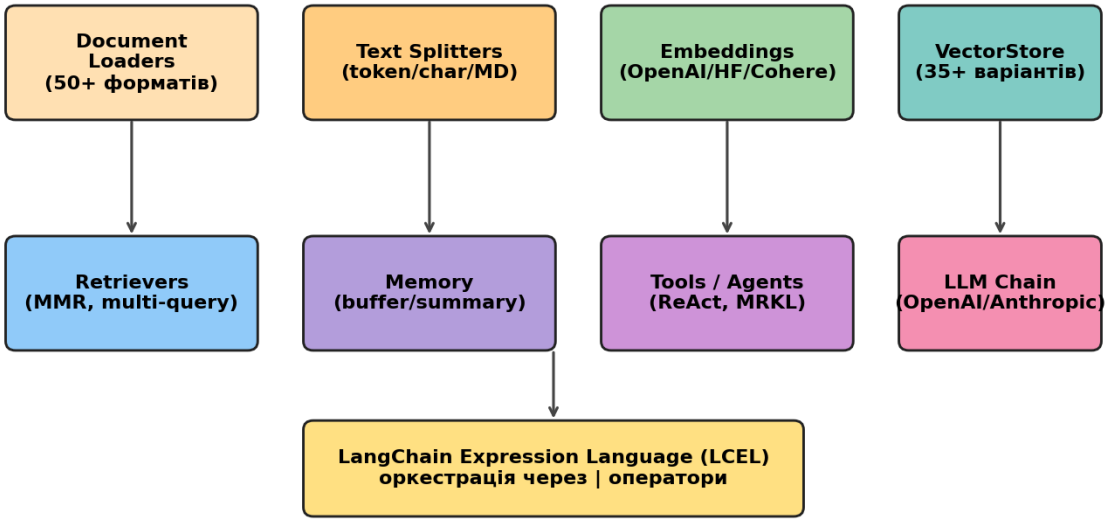


Рисунок 1.1 – Архітектура LangChain (узагальнено)

1.5.2 LlamaIndex

LlamaIndex (раніше GPT Index) – фреймворк, спеціалізований саме на побудові RAG-систем. Він пропонує абстракції документа, вузла (node), ретривера та оцінювача, готові патерни індексації для дерев, графів та векторних сховищ. LlamaIndex має кращу за LangChain підтримку мультимодальності, проте конфігурація конвеєра вимагає значної кількості коду, а вбудовані налаштування часто не оптимальні для технічних документів.

Рисунок 1.2 – Архітектура LlamaIndex (узагальнено)

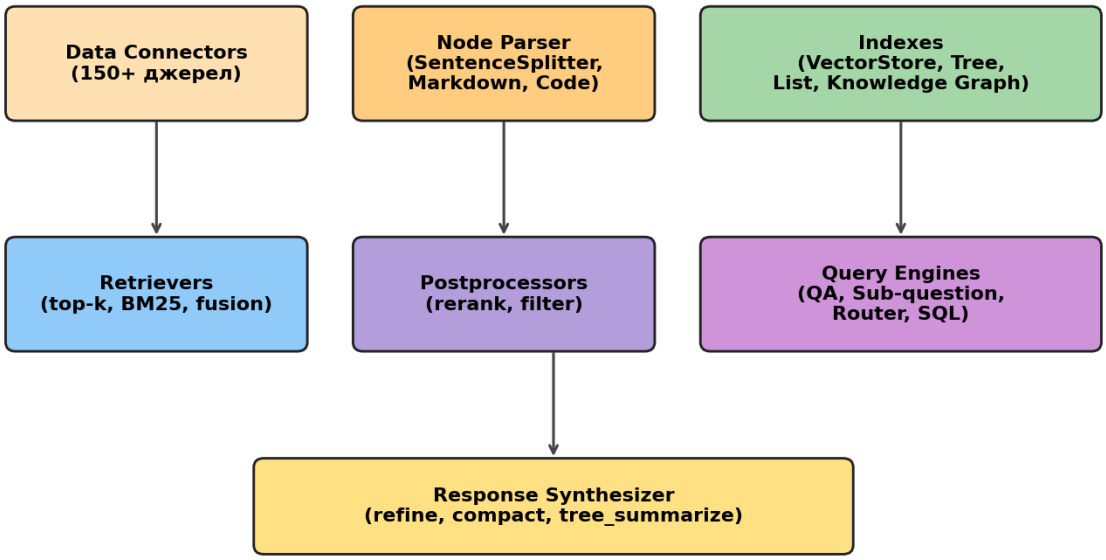


Рисунок 1.2 – Архітектура LlamaIndex (узагальнено)

1.5.3 Haystack

Haystack від deepset – фреймворк промислового рівня, побудований навколо концепції pipeline. Кожен крок (preprocessor, retriever, reader, ranker) є вузлом графа, а pipeline визначає послідовність викликів. Сильні сторони: REST-сервіс з коробки, інтеграція з Elasticsearch, OpenSearch, FAISS. Недоліки: важкий старт через велику кількість абстракцій та обмежена підтримка мультимодальних артефактів.

Рисунок 1.3 – Архітектура Haystack (узагальнено)

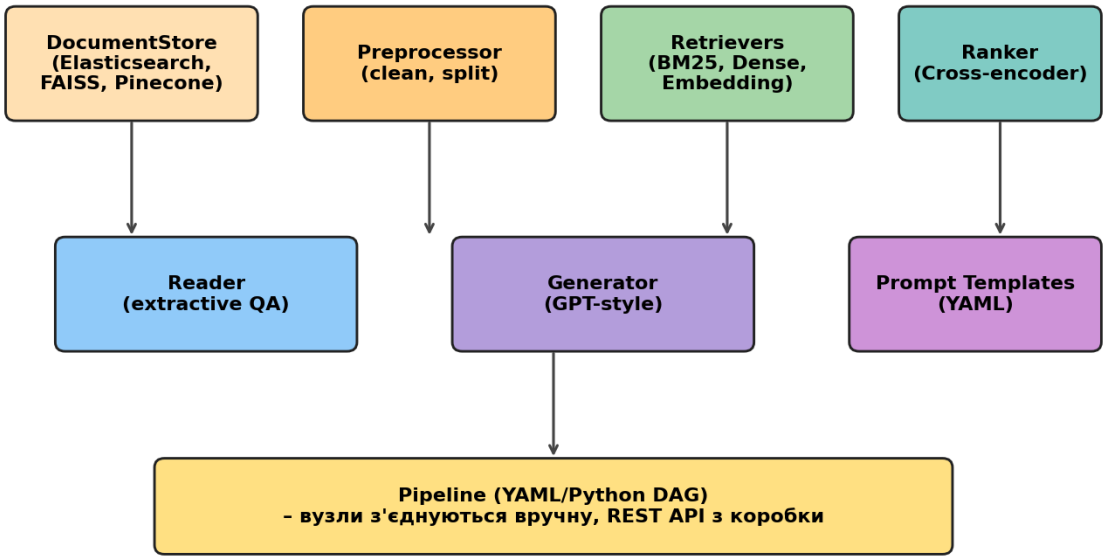


Рисунок 1.3 – Архітектура Haystack (узагальнено)

1.5.4 RAGFlow

RAGFlow – порівняно нове рішення, що пропонує веб-інтерфейс для управління колекціями документів, побудови індексів та виконання запитів. Орієнтоване переважно на корпоративних користувачів і поставляється як готовий до розгортання сервіс із Docker-образами. Має DeepDoc-парсер для документів зі складною розміткою. Розширення RAGFlow програмно ускладнене, оскільки система переважно конфігурується через UI.

Рисунок 1.4 – Архітектура RAGFlow (узагальнено)

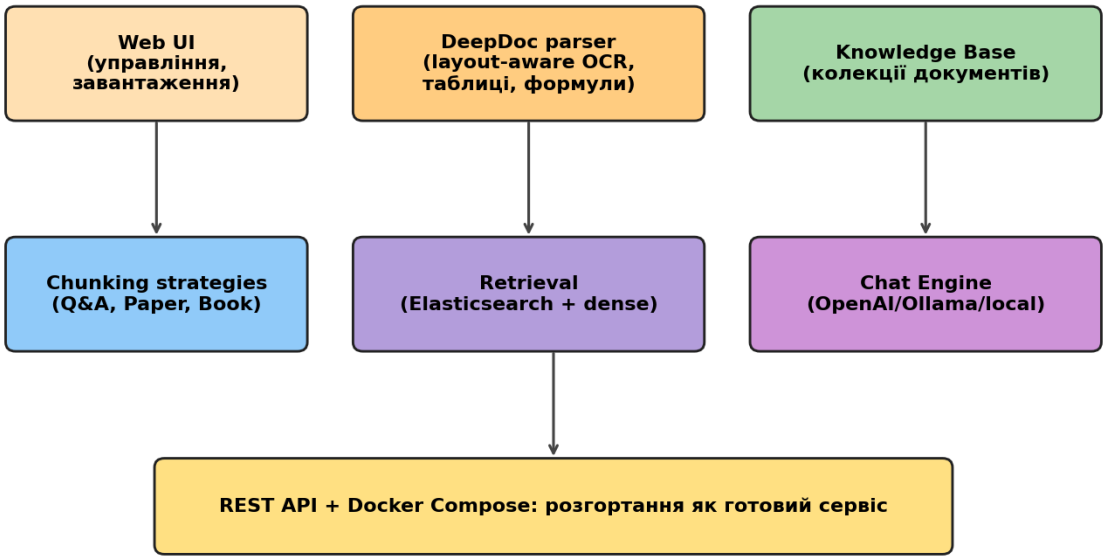


Рисунок 1.4 – Архітектура RAGFlow (узагальнено)

1.5.5 Порівняльний аналіз

У таблиці 1.1 наведено порівняння згаданих рішень за основними критеріями, важливими для задачі мультимодального пошуку у технічних PDF.

Таблиця 1.1 – Порівняння існуючих рішень за основними критеріями

Критерій	**LangChain**	**LlamaIndex**	**Haystack**	**RAGFlow**	**mmrag (наша)**
Мультимод. PDF	–	Частково	–	Частково	Так
Гібридний пошук	Так	Так	Так	Так	Так
Cross-encoder	Через плагіни	Так	Так	Так	Так
Локальний LLM	Так	Так	Так	Частково	Так
REST API з коробки	–	–	Так	Так	Так
Чиста архітектура	–	Частково	–	–	Так
Швидкий старт	Середній	Середній	Складно	Просто	Просто

Підсумовуючи, жодне з розглянутих рішень не пропонує цілісного, легковагового та модульного інструменту, який би одразу «з коробки» забезпечував якісне опрацювання мультимодального вмісту технічних PDF, гібридний пошук, реранкінг та інтеграцію з декількома постачальниками LLM. Це обґрунтовує доцільність розробки спеціалізованої системи у межах даного проєкту.

1.6 Постановка задачі

На основі проведеного аналізу формулюємо задачу проєкту: спроєктувати і реалізувати програмну систему мультимодального RAG для технічних PDF-документів. Система повинна приймати на вхід довільні PDF-документи; автоматично виділяти з документа текстові блоки, таблиці у структурованому вигляді та зображення з природномовним описом; будувати гібридний пошуковий індекс на основі векторного сховища FAISS та інвертованого індексу BM25; виконувати пошук за природномовним запитом з реранкінгом найкращих кандидатів моделлю cross-encoder; генерувати природномовну відповідь з можливістю вибору між хмарним та локальним бекендом LLM; надавати інтерфейси REST API та командного рядка; мати модульну архітектуру, що дозволяє замінювати конкретні реалізації окремих компонентів без модифікації прикладного коду.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі проведено аналіз предметної області інформаційного пошуку та генеративної обробки технічних PDF-документів. Розглянуто класичні (BM25, TF-IDF) та сучасні (dense retrieval, cross-encoder) методи пошуку, проаналізовано особливості мультимодального вмісту технічних публікацій та сформульовано виклики, з якими стикаються наявні рішення. Виконано огляд популярних фреймворків (LangChain, LlamaIndex, Haystack, RAGFlow) з аналізом їхньої архітектури, наведено порівняльну таблицю та обґрунтовано доцільність розробки спеціалізованої програмної системи. Сформульовано конкретну постановку задачі, яка лежить в основі подальшої розробки.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СИСТЕМИ

У другому розділі обґрунтовано вибір технологічного стеку, спроєктовано доменну модель та структурну схему системи. Описано принципи побудови архітектури за моделлю чистої архітектури з суворим розділенням шарів, проаналізовано застосовані шаблони проєктування, наведено детальний опис кожного підмодуля з аналізом структурних рішень. На завершення розділу проаналізовано відповідність архітектурних рішень нефункціональним вимогам та визначено стратегії валідації обраного підходу.

2.1 Загальний аналіз вимог

Перш ніж обирати технології та проєктувати архітектуру, необхідно детально проаналізувати вимоги, винесені у Технічне завдання (підрозділи 8 та 9 ТЗ). Тринадцять функціональних вимог (ФВ-1…ФВ-13) природним чином групуються у чотири кластери: вхідна обробка (ФВ-1…ФВ-4) – завантаження PDF, виділення тексту, зображень з описом BLIP та таблиць у Markdown; побудова індексу (ФВ-5…ФВ-7) – векторний індекс через sentence-transformers + FAISS, BM25-індекс на тих самих представленнях, серіалізація та повторне завантаження; обробка запитів (ФВ-8…ФВ-10) – гібридний пошук через RRF, реранкінг cross-encoder, генерація відповіді з посиланнями; інтерфейси та оцінювання (ФВ-11…ФВ-13) – REST API, CLI та конвеєр оцінювання якості.

Сім нефункціональних вимог (НФВ-1…НФВ-7) накладають додаткові обмеження на проєктне рішення. НФВ-1 (модульність) та НФВ-2 (розширюваність) безпосередньо диктують вибір архітектурного підходу: монолітна архітектура з прямою залежністю прикладного коду від конкретних бібліотек не задовольнить цих вимог при необхідності замінити, наприклад,

FAISS на Qdrant. НФВ-3 (продуктивність) обмежує вибір моделей: важка модель типу BLIP-2 на CPU не вкладається у бюджет 5 секунд на запит, тому за замовчуванням обрано BLIP-base. НФВ-4 (відтворюваність) вимагає централізованої конфігурації та детермінованого порядку операцій. НФВ-5 (якість пошуку, Recall@5 не менше 0.85) визначає необхідність гібридного підходу з реранкінгом, оскільки одиничні методи на тестовому наборі цього порога не досягають. НФВ-6 (незалежність від мережі) накладає вимогу про підтримку локальних LLM. НФВ-7 (тестованість) знову повертає до архітектурних рішень: класи з твердо прошитими залежностями неможливо тестувати без піднімання важких моделей.

Сформулюємо узагальнюючі вимоги, які стануть критеріями проєктування: модульність на рівні архітектури – зміна реалізації одного компонента не повинна вимагати модифікації інших; відсутність прямих залежностей бізнес-логіки від конкретних бібліотек; підтримка кількох постачальників LLM-моделей через єдиний інтерфейс; централізована конфігурація через типізовані параметри; зручність тестування з мок-об'єктами; передбачувана продуктивність на типовому корпусі та запиті.

2.2 Обґрунтування вибору технологічного стеку

Як основну мову обрано Python 3.10+. Серед причин – найбагатша у ML-екосистемі база відкритих бібліотек (PyTorch, transformers, sentence-transformers, FAISS, PyMuPDF, FastAPI) та зрозумілий синтаксис, що сприяє швидкому прототипуванню. Підтримка статичної типізації через type hints і Pydantic дає переваги, традиційно асоційовані з компільованими мовами.

Для парсингу PDF обрано бібліотеку PyMuPDF (fitz) версії 1.23 і вище: вища швидкодія, вбудована підтримка виявлення таблиць (page.find_tables) та зручний API для роботи з зображеннями. Для побудови embedding-ів використано модель sentence-transformers/all-MiniLM-L6-v2 (22 млн параметрів, розмірність 384, продуктивність близько 1000 текстів за секунду

на CPU). Векторне сховище – FAISS з IndexFlatIP для невеликих корпусів. Розріджений індекс – rank-bm25 (чистий Python, легко серіалізується через pickle). Реранкінг – cross-encoder/ms-marco-MiniLM-L-6-v2. Опис зображень – Salesforce/blip-image-captioning-base. Для генерації відповіді підтримуються два бекенди: OpenAI gpt-4o-mini та локальна модель microsoft/phi-2. Веб-API побудовано на FastAPI; CLI – на Typer.

2.3 Порівняння архітектурних патернів

Для побудови системи розглянуто три основні архітектурні підходи: монолітну архітектуру з прямою залежністю від бібліотек, шарову архітектуру (layered) та чисту/гексагональну архітектуру з інверсією залежностей. Розглянемо кожен підхід та обґрунтуємо остаточний вибір.

Монолітний підхід є найпростішим: усі модулі імпортують потрібні бібліотеки безпосередньо, бізнес-логіка викликає функції зовнішніх API напряму. Перевагою є мінімальний обсяг коду, недоліком – неможливість заміни компонентів без модифікації бізнес-логіки, складність тестування (потрібно завантажувати реальні моделі) та сильна зв'язаність. Цей підхід порушує НФВ-1, НФВ-2 та НФВ-7.

Класична шарова архітектура (presentation → business logic → data access) частково розв'язує проблеми монолітного підходу, проте все ще передбачає, що верхній шар знає про конкретні реалізації нижнього: бізнес-логіка імпортує клас FAISSStore безпосередньо. Це краще, ніж монолітний підхід, але не повністю задовольняє вимогам про підміну реалізацій.

Чиста архітектура (Clean Architecture, Robert Martin) та гексагональна архітектура (Hexagonal Architecture, Alistair Cockburn) застосовують принцип інверсії залежностей: бізнес-логіка визначає інтерфейси (порти), а конкретні технології реалізують ці інтерфейси (адаптери). Залежності завжди спрямовані «всередину» – від адаптерів до інтерфейсів, від інтерфейсів до бізнес-логіки, від бізнес-логіки до доменної моделі. У такому підході заміна FAISS на Qdrant

зводиться до реалізації нового адаптера QdrantStore без жодних змін у класах IndexingPipeline та QueryPipeline.

Цей підхід було обрано як основний для проєкту з огляду на повну відповідність вимогам НФВ-1, НФВ-2, НФВ-6 та НФВ-7. Ціною є дещо більший обсяг коду (потрібні абстрактні класи-порти та конкретні класи-адаптери замість прямих викликів), однак це окуповується значно вищою тестованістю та зручністю розширення.

2.4 Архітектурний підхід

Структуру архітектури зображено на рисунку 2.1. Архітектура поділена на чотири концентричні шари. Внутрішній шар – domain – містить бізнес-сутності, що не залежать від жодної зовнішньої технології. Це Pydantic-моделі Document, Page, Block, Artifact, Provenance, Evidence, Query, Answer, QueryIntent з строгою типізацією. Шар не може імпортувати нічого з зовнішніх бібліотек, окрім Pydantic, що гарантує його стабільність протягом усього життєвого циклу системи.

Рисунок 2.1 – Шари чистої архітектури системи

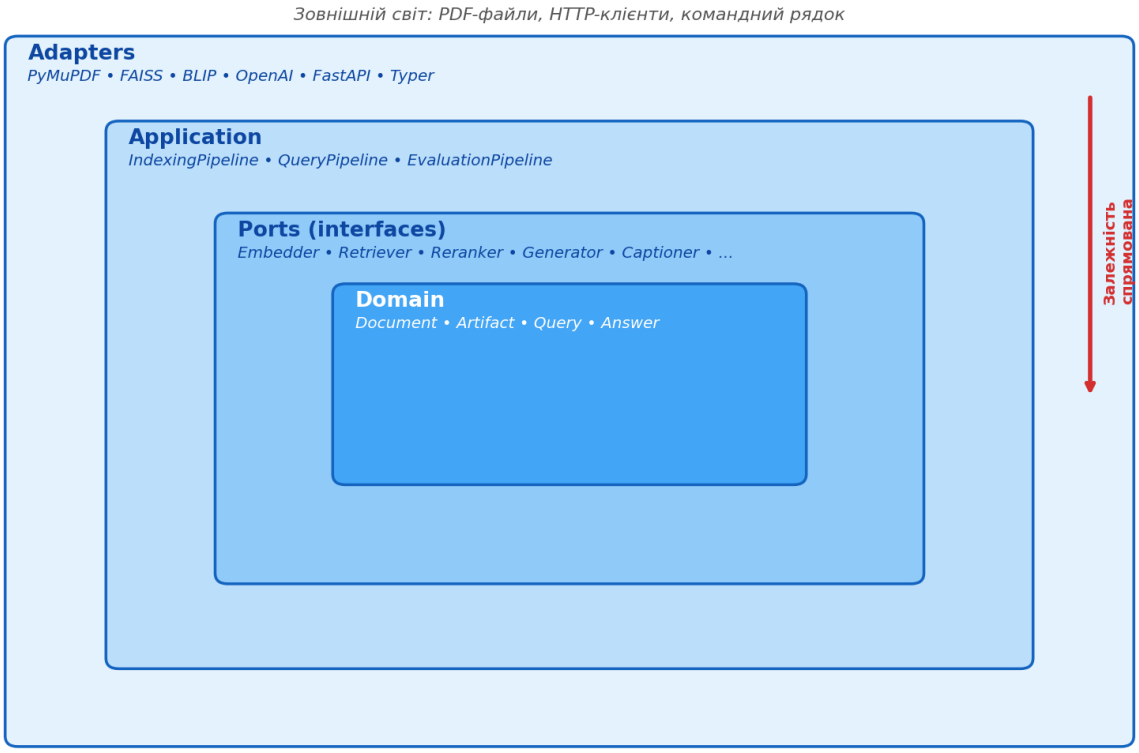


Рисунок 2.1 – Узагальнена структурна схема системи

Наступний шар – ports – визначає інтерфейси (abstract base classes), якими користується прикладна логіка. У системі визначено дванадцять портів: PDFLoader, FigureExtractor, TableExtractor, Captioner, Embedder, IndexStore, Retriever, Reranker, Generator, LayoutAnalyzer, Evaluator, ChartParser. Кожен інтерфейс має чітко окреслений контракт у формі абстрактних методів. Наприклад, інтерфейс Embedder містить два методи: encode(texts: List[str]) -> np.ndarray для пакета текстів та encode_single(text: str) -> np.ndarray для одного тексту; жодних інших методів не очікується від конкретного embedder-a, незалежно від того, чи це sentence-transformers, чи OpenAI Embeddings, чи кастомна модель.

Шар application містить use-case-сценарії: IndexingPipeline (індексація), QueryPipeline (обробка запиту), EvaluationPipeline (оцінювання якості). Сценарії оперують виключно через інтерфейси з шару ports і не знають про конкретні реалізації. Усі залежності інжектуються через конструктор класу за

принципом Dependency Injection. Це означає, що клас IndexingPipeline не виконує імпорту FAISSStore чи PyMuPDFLoader – у конструкторі він приймає об'єкти, що реалізують відповідні інтерфейси, і не цікавиться, які саме адаптери передані.

Шар adapters містить конкретні реалізації інтерфейсів: PyMuPDFLoader, BasicFigureExtractor, PyMuPDFTableExtractor, BLIPCaptioner, SentenceTransformerEmbedder, FAISSStore, BM25Retriever, HybridRetriever, CrossEncoderReranker, OpenAIGenerator, LocalGenerator, LocalDocStore, HeuristicLayout. Кожен адаптер інкапсулює залежність від конкретної бібліотеки. Якщо у майбутньому виникне потреба замінити FAISS на Qdrant, достатньо реалізувати новий клас QdrantStore, що задовольняє інтерфейсу IndexStore, та оновити фабрику – прикладний код не змінюється.

Шари api та cli містять контролери – точки входу системи, які перетворюють зовнішні запити (HTTP, командний рядок) у виклики сценаріїв з шару application. Конфігурація централізована в config/settings.py через Pydantic Settings, що читає змінні середовища та .env-файл. Логування налаштовується у config/logging.py. Така централізована конфігурація дозволяє керувати поведінкою системи з одного місця без модифікації коду.

2.5 Шаблони проєктування у системі

Архітектура системи використовує кілька класичних шаблонів проєктування (Design Patterns), що рекомендовані до застосування при побудові модульних та тестованих систем.

Шаблон Dependency Injection (DI) використовується у всіх класах application-шару. Конструктор класу IndexingPipeline приймає вісім параметрів-залежностей: pdf_loader, figure_extractor, table_extractor, captioner, embedder, index_store, doc_store, bm25_retriever. У продакшені ці залежності формуються через фабричний метод _build_pipelines, у тестах – заміняються

мок-об'єктами. Це класична реалізація Constructor Injection, що забезпечує явність залежностей та їхню підмінність.

Шаблон Factory застосовується для побудови повного об'єкта системи з конфігурації. Функція _build_pipelines у модулі api/app.py читає Settings, на основі параметра generator_backend обирає реалізацію Generator (OpenAIGenerator або LocalGenerator), створює всі адаптери з відповідними параметрами та збирає з них пайплайни. У такому підході рішення про конкретні реалізації приймається в одному місці, а не розкидано по коду.

Шаблон Strategy використовується у виборі бекенду генерації відповіді. Обидва класи (OpenAIGenerator, LocalGenerator) реалізують спільний інтерфейс Generator з методом generate(query, artifacts) -> Answer. Прикладний код у QueryPipeline.run викликає self.generator.generate(...) без знання, який саме бекенд активний. Зміна бекенду не вимагає модифікації QueryPipeline.

Шаблон Repository виражений у класах LocalDocStore та FAISSStore. Вони інкапсулюють деталі зберігання артефактів (in-memory словник для docstore; FAISS-індекс для векторів) за простим інтерфейсом put/get/save/load. Заміна реалізації на, наприклад, Redis-backed docstore зводиться до створення нового класу, що реалізує той самий контракт.

Шаблон Adapter використовується для інтеграції зовнішніх бібліотек. BLIPCaptioner адаптує API transformers.pipeline до інтерфейсу Captioner; CrossEncoderReranker адаптує sentence-transformers.CrossEncoder до інтерфейсу Reranker. Класи-адаптери приховують специфіку конкретного API за стабільним внутрішнім інтерфейсом.

Шаблон Template Method частково проявляється у класі IndexingPipeline: послідовність кроків (load → extract text → extract figures → caption → extract tables → embed → add to FAISS → add to BM25 → save) фіксована, проте кожен крок делегується інжектованому компоненту, що відповідає за конкретну реалізацію. Це класична структура pipeline-патерну.

2.6 Доменна модель

Доменна модель є основою системи: вона визначає словник предметної області, у якому проєкт описує всі свої сутності та операції. Розглянемо центральні сутності більш детально.

2.6.1 Document, Page, Block

Document представляє PDF-документ після етапу завантаження. Він містить заголовок (title), шлях до файлу (file_path), список сторінок (pages: List[Page]) та метадані (metadata) у вигляді словника. Заголовок за замовчуванням формується з імені файлу без розширення, але може бути перезаписаний з PDF-метаданих, якщо вони присутні. Шлях файлу зберігається в абсолютному вигляді; це важливо, оскільки артефакти пізніше посилаються на нього у Provenance, і відносні шляхи могли б призвести до некоректних посилань при переміщенні робочого каталогу. Метадані містять довільні пари «ключ – значення», як-от назва видання, автори, дата публікації, мова.

Page – сторінка PDF. Має номер (number, від 1), список блоків (blocks: List[Block]) та опційний шлях до рендеру сторінки у вигляді зображення (image_path). Рендер сторінки використовується у двох сценаріях: коли модель LLM приймає мультимодальний вхід і може дивитися на зображення сторінки безпосередньо, та коли користувачеві у відповіді потрібно показати фрагмент оригінальної сторінки. Номер сторінки – важливе поле, оскільки усі посилання у відповідях LLM містять номер сторінки як ключову частину Provenance.

Block – низькорівневий елемент сторінки з текстом, обмежувальною рамкою у форматі (x0, y0, x1, y1) та типом (block_type: 'text', 'image', 'table'). Координати у системі PDF (точки, 1 точка = 1/72 дюйма; вісь Y зростає донизу). Завдяки такій трирівневій структурі дані з різних блоків можна поєднувати, а на конкретні координати у документі можна посилатися у відповідях. Confidence – необов'язкове поле; для блоків, отриманих із

текстового шару PDF, воно дорівнює 1.0, для блоків, отриманих через OCR – значенню впевненості розпізнавання.

2.6.2 Artifact, Provenance, Evidence

Artifact – головна одиниця індексації. На відміну від Block (низькорівневий примітив з парсера) Artifact – це готовий до пошуку фрагмент знань. Поля артефакту: id (UUID), artifact_type ('text', 'figure', 'table', 'chart'), content (текстове представлення, що буде embedded та індексованим), image_path (для зображень-артефактів), caption (опис, згенерований captioner-моделлю), provenance (метадані походження), confidence. Усі артефакти з одного PDF мають спільну структуру і єдиний інтерфейс пошуку – це і є суть «мультимодальності» індексу: модель не розрізняє типи джерел до останнього кроку, де у відповіді LLM додаються маркери джерел [type:page].

Окреме поле artifact_type дозволяє реалізувати фільтри типу «шукати лише в таблицях» або «лише в зображеннях». У майбутньому до набору можуть бути додані нові типи (chart, formula, code-block) без модифікації існуючого коду.

Provenance містить шлях до файлу, номер сторінки та bbox; це дозволяє у відповіді LLM посилатися на конкретне місце в документі: «джерело: с. 12, таблиця у координатах (60, 220) – (490, 350)». У реальному UI це посилання можна перетворити на клікабельну прев'юшку фрагмента сторінки. Збереження bbox також робить можливим витяг подальших візуальних доказів – наприклад, для ілюстративних цілей у згенерованій відповіді.

Evidence використовується у відповіді LLM як прив'язка твердження до конкретного артефакту: для кожного речення відповіді можна простежити, на який саме фрагмент тексту/таблиці/рисунка вона базується. Поле confidence в Evidence є оцінкою системи власної впевненості у відповіді на основі релевантності джерела (нормалізована оцінка з cross-encoder).

2.6.3 Query, Answer, QueryIntent

Query інкапсулює запит користувача: текст запитання, намір (QueryIntent: SEARCH, QA, SUMMARIZE, EXTRACT) та опціональні фільтри (наприклад, «лише цей документ» або «лише таблиці»). Намір використовується для адаптації промпта LLM: для SEARCH повертається лише список релевантних артефактів, для QA – синтезована відповідь з посиланнями, для SUMMARIZE – узагальнення з кількох артефактів, для EXTRACT – структуроване витягання сутностей у форматі JSON.

Розрізнення intent-ів – важливе проєктне рішення: воно дозволяє підготувати спеціалізовані промпти та логіку постобробки відповіді залежно від сценарію. Наприклад, для intent=EXTRACT відповідь повинна задовольняти JSON Schema, який вказано у фільтрах; у разі невідповідності виконується повторний виклик LLM з виправляючим промптом.

Answer містить згенеровану відповідь (text), оцінку впевненості (confidence: 0..1), перелік джерел (sources: List[Evidence]) та перелік ID артефактів, що формували контекст. Така структура легко серіалізується у JSON для REST API та є зручним результатом для UI: користувач бачить відповідь, поряд – список використаних джерел з гіперпосиланнями.

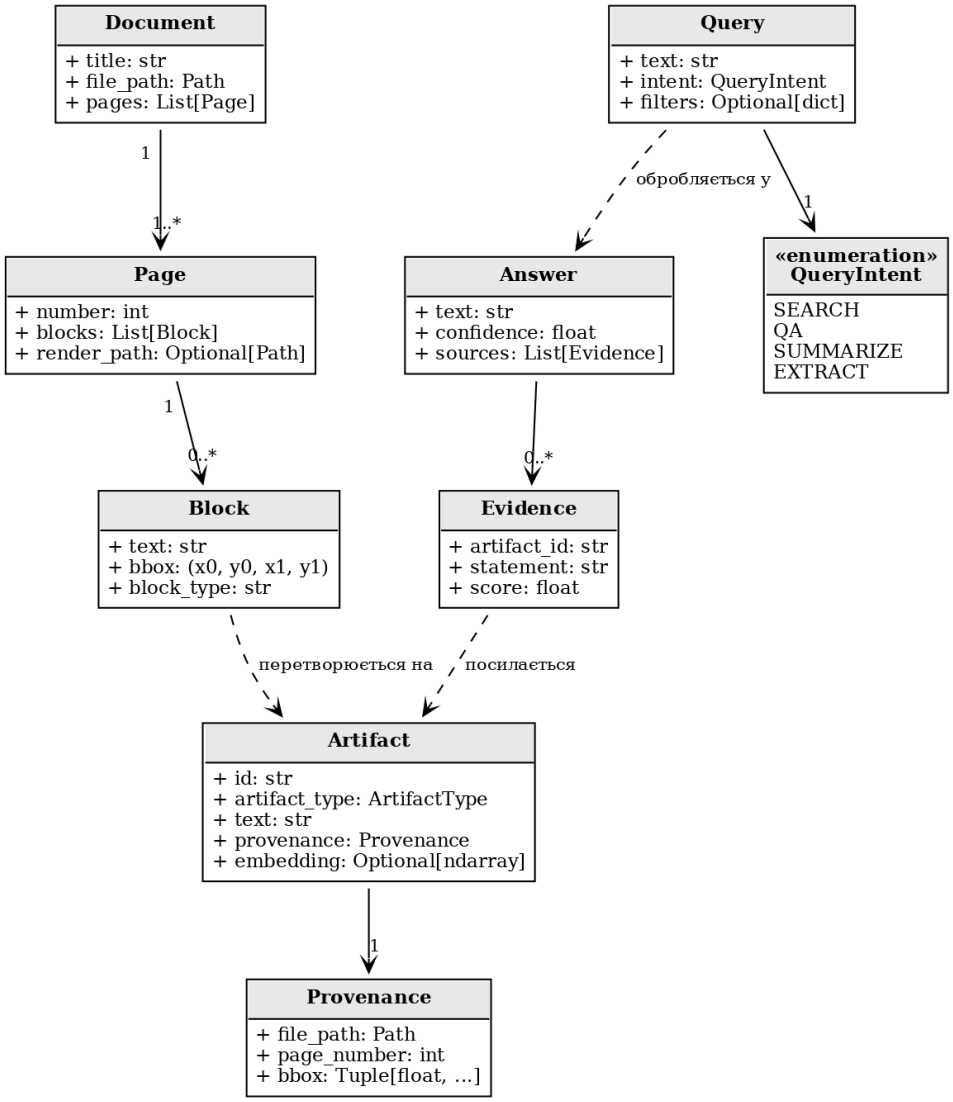


Рисунок 2.2 – Діаграма класів доменної моделі

2.7 Конвеєр індексації

Конвеєр індексації реалізовано у класі IndexingPipeline (application/indexing_pipeline.py). Конструктор класу приймає набір залежностей через dependency injection: pdf_loader, figure_extractor, table_extractor, captioner, embedder, index_store, doc_store, bm25_retriever, а також параметри chunk_size та chunk_overlap. Усі залежності типізовані відповідними інтерфейсами з шару ports, що дозволяє замінювати конкретні реалізації.

Структурна схема конвеєра наведена на рисунку 2.3. Публічний метод run(pdf_path, output_dir) виконує дев'ять послідовних кроків: завантаження PDF; чанкінг тексту; виділення зображень; опис зображень моделлю BLIP; виділення таблиць; формування пакета артефактів; обчислення embedding-ів; додавання до FAISS; додавання до BM25; серіалізація стану на диск.

Рисунок 3.1 – Структурна схема конвеєра індексації

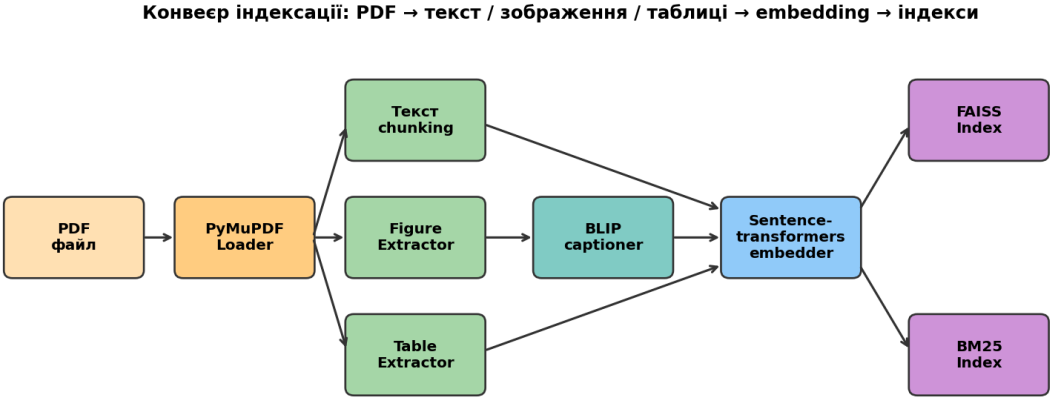


Рисунок 2.3 – Структурна схема конвеєра індексації

Перший крок виконує PyMuPDFLoader. Метод load(file_path) відкриває PDF через fitz.open і для кожної сторінки викликає page.get_text('dict'), отримуючи структурований словник з блоками. Кожен блок з ключем 'lines' трактується як текстовий. Текст об'єднується зі spans та lines; координати беруться з 'bbox'. Результатом є об'єкт Document.

Алгоритм чанкування реалізує наступну логіку. Нехай маємо послідовність слів W = (w_1, w_2, ..., w_n). Вікно довжиною S (chunk_size, за замовчуванням 400) рухається з кроком S – O (де O – chunk_overlap, за замовчуванням 50). Перший чанк – W[0..S]; другий – W[S–O..2S–O]; i-тий чанк – W[(i–1)(S–O)..(i–1)(S–O)+S]. Перекриття у 50 слів забезпечує цілісність контексту на межі чанків: якщо важливий факт згаданий у двох сусідніх

вікнах, він обов'язково потрапить хоча б в одне з них повністю. Це зменшує ймовірність втрати інформації на межах та підвищує Recall пошуку.

Виділення зображень виконує BasicFigureExtractor. Для кожної сторінки викликається page.get_images(full=True), що повертає список xref-посилань на вбудовані зображення. Для кожного xref виконується doc.extract_image, отримуючи байти, ширину, висоту та розширення. Зображення менші за поріг min_size (за замовчуванням 80 пікселів) відкидаються – це фільтрує іконки, декоративні елементи та технічні роздільники. Емпірично визначений поріг 80 px є компромісом: занадто низький поріг призводить до індексації декоративних елементів та збільшення обсягу індексу; занадто високий – до пропуску корисних малих діаграм.

Кожне виділене зображення анотується. BLIPCaptioner обгортає модель Salesforce/blip-image-captioning-base через transformers. Метод caption(image_path) завантажує зображення, передає його через процесор та модель і повертає згенерований опис як рядок. Цей опис записується одночасно в поля caption та content артефакту – саме content буде далі embedded. Цей принцип «caption-as-content» означає, що пошук по зображеннях фактично відбувається у просторі їхніх текстових описів, що дозволяє індексувати їх єдиним набором інструментів з текстом.

Таблиці виділяє PyMuPDFTableExtractor. Для кожної сторінки викликається page.find_tables(), доступна з PyMuPDF 1.23. Кожна знайдена таблиця конвертується у Markdown через _table_to_markdown: формується заголовок, рядок-роздільник, далі – рядки даних. Markdown добре зберігає табличну структуру у пласкому тексті, дозволяючи sentence-transformers вловлювати залежності «стовпець – значення». Альтернативами є серіалізація у CSV (втрата заголовків при склейці у chunk) або JSON (надлишковість для embedder-a); Markdown є оптимальним компромісом між читабельністю людиною та машиною.

Зібраний список усіх артефактів (текстових, рисункових, табличних) передається в SentenceTransformerEmbedder. Виклик self.model.encode(texts, convert_to_numpy=True) повертає матрицю розмірності N × dim. Матриця та список ID передаються у FAISSStore.add. У середині add вектори нормалізуються (поділ на L2-норму), що перетворює внутрішнє множення IndexFlatIP на косинусну подібність, та додаються до FAISS-індексу.

Той самий список артефактів передається у BM25Retriever.build. Для кожного артефакту формується текстове представлення (content + опційно caption), яке токенізується функцією _tokenize (lowercase + регулярний вираз для слівних символів). Список токенізованих документів передається у BM25Okapi з бібліотеки rank-bm25.

Завершальний крок – запис побудованих структур на диск. FAISSStore.save записує faiss.write_index у файл faiss.index та pickle-список ID у ids.pkl. BM25Retriever.save серіалізує BM25Okapi-об'єкт та список артефактів у bm25.pkl. LocalDocStore.save зберігає словник artifact_id у docstore.pkl. Окремо створюється artifacts.jsonl – текстовий файл, де кожен рядок – JSON-серіалізований артефакт; формат зручний для перегляду людиною.

2.8 Конвеєр запитів та формальний опис RRF

Конвеєр запитів реалізовано у класі QueryPipeline (application/query_pipeline.py). Конструктор приймає retriever, reranker, generator, retrieval_top_k (за замовчуванням 20), rerank_top_k (за замовчуванням 5). Метод run(question, intent, filters) виконує чотири фази: гібридний пошук, реранкінг, генерація відповіді, повернення Answer з джерелами.

Рисунок 3.2 – Структурна схема конвеєра запитів

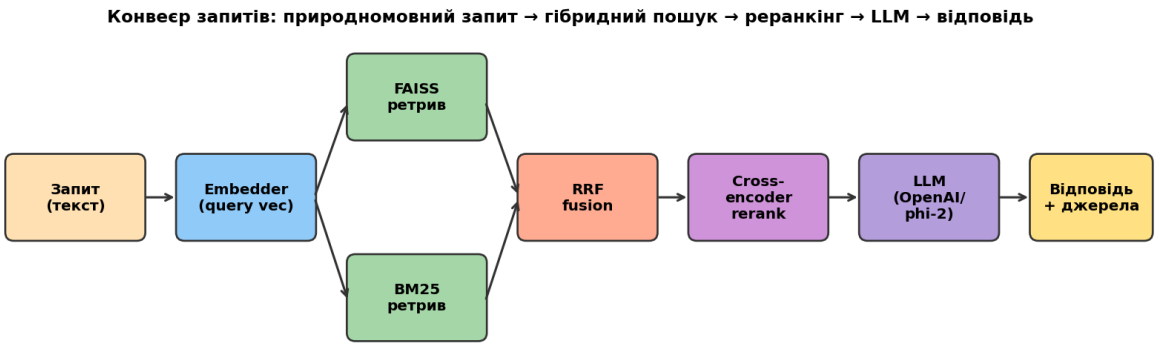


Рисунок 2.4 – Структурна схема конвеєра запитів

Гібридний ретрив реалізує клас HybridRetriever. Метод retrieve(query, top_k) виконує: кодує запит у вектор через embedder.encode_single; шукає 3 × top_k кандидатів у FAISS (але не більше 100); отримує BM25-кандидатів аналогічно; виконує Reciprocal Rank Fusion.

Алгоритм Reciprocal Rank Fusion детальніше: нехай маємо два упорядковані списки результатів L_dense та L_bm25, кожен довжини k_candidates. Для кожного унікального документа d, що зустрічається хоча б в одному списку, обчислюється агрегована оцінка S(d) = w_dense / (k + rank_dense(d)) + w_bm25 / (k + rank_bm25(d)), де rank_method(d) – позиція документа у відповідному списку (нумерація з 0), k = 60 – згладжувальна константа, w_dense і w_bm25 – налаштовувані ваги (за замовчуванням 0.6 та 0.4). Якщо документ відсутній в одному зі списків, відповідний доданок дорівнює нулю. Документи впорядковуються за спаданням S(d) і повертаються перші top_k.

Параметр k = 60 (рекомендований Cormack et al. 2009) має такий ефект: для невеликих рангів (1–10) знаменник k+rank змінюється повільно, тому документи з топу обох списків отримують близькі вкладення, а документ, що потрапив у топ обох методів, отримує значно вищу оцінку, ніж той, що

потрапив до топу лише одного. Для великих рангів (50+) знаменник стає великим і вклад невеликий, що еквівалентно відсіканню кінця списку. Параметри dense_weight (0.6) та bm25_weight (0.4) дозволяють зміщувати баланс на користь семантичного або лексичного методу. Емпіричні випробування на тестовому наборі показали, що для технічної документації дещо більша вага dense-методу забезпечує оптимальну якість.

Кандидати, отримані гібридним ретривером, передаються у CrossEncoderReranker. Модель cross-encoder/ms-marco-MiniLM-L-6-v2 завантажується ліниво при першому виклику. Метод rerank(query, artifacts, top_k) формує пари (query.text, artifact_text), де artifact_text складається з content + caption та обрізається до 1500 символів. model.predict(pairs) повертає масив скалярних оцінок; пари сортуються за спаданням та повертаються перші top_k.

Cross-encoder перевершує bi-encoder за точністю тому, що приймає на вхід пару (запит, фрагмент) і обчислює attention між токенами обох частин. Це дозволяє моделі врахувати контекстну взаємодію між запитом і документом, чого не здатний зробити bi-encoder, що кодує їх незалежно. Зворотний бік – значно нижча швидкодія: cross-encoder неможливо передобчислити, оскільки для кожного запиту потрібен прохід моделі для кожного кандидата. Тому реранкінг застосовується лише до невеликої кількості кандидатів (типово 20).

Релевантні артефакти разом із запитом передаються у Generator. OpenAIGenerator формує системний промпт із зазначенням ролі (експерт з технічної документації), та користувацький промпт, що включає сам запит та контекст (з відмітками джерел типу [doc:page]). Виклик openai.chat.completions.create повертає відповідь, яка парситься у Answer з пов'язаними джерелами Evidence. LocalGenerator використовує transformers.pipeline з моделлю microsoft/phi-2. Промпт формується аналогічно, проте враховуються обмеження локальної моделі: коротший контекст, повільніша швидкодія.

2.9 REST API, CLI та контрактні схеми

Модуль api/app.py містить FastAPI-додаток з чотирма ендпойнтами: GET /health (перевірка стану), POST /index (завантаження PDF та індексація), POST /query (виконання запиту), GET /artifacts (перегляд індексованих артефактів). Конвеєри будуються один раз при старті додатка. Якщо у каталозі settings.index_dir уже існують файли faiss.index, docstore.pkl та bm25.pkl, вони завантажуються відразу – це дозволяє запустити сервіс без повторної індексації після рестарту.

Запити та відповіді описано Pydantic-схемами QueryRequest, QueryResponse, IndexResponse, ArtifactSummary, що автоматично транслюються FastAPI у JSON Schema та документуються через OpenAPI. Це дає змогу автоматично згенерувати клієнтів для будь-якої сучасної мови (TypeScript, Java, Go) через openapi-generator або стандартні бібліотеки. Документація доступна за адресою /docs (Swagger UI) та /redoc (ReDoc).

Обробка помилок будується за принципом «помилка ближче до краю». Глибинні модулі (адаптери) піднімають специфічні винятки: PDFParsingError, ModelLoadError, IndexCorruptedError тощо. Шар application лише логує їх та повертає узагальнений тип помилки до контролерів. У REST API винятки перехоплюються та повертаються користувачу у вигляді стандартизованих HTTP-помилок: 400 (некоректний запит – наприклад, переданий файл не PDF), 422 (валідаційна помилка Pydantic), 500 (внутрішня помилка системи). Це забезпечує одночасно детальність діагностики у логах та чистоту інтерфейсу для клієнтів.

CORS-middleware налаштовано на дозвіл будь-якого джерела на час розробки; для виробничого розгортання список allow_origins повинен бути обмежений конкретними доменами для запобігання cross-site requests. Rate limiting у поточній версії не реалізовано на рівні самого API, але

рекомендується додати reverse-proxy (nginx, traefik) перед uvicorn для виробничого розгортання.

CLI реалізовано у cli/main.py через Typer з командами mmrag index, query, list-artifacts, serve. Декларативний API Typer дозволяє оголошувати команди через декоратори @app.command, а параметри типізуються через type hints – Typer автоматично перетворює їх на CLI-опції з опціями допомоги. Реалізовано підтримку як короткої форми (-h), так і повної (--help) для кожної команди. Команди CLI використовують ті ж самі application-сценарії, що і REST API, тому одночасно підтримуються обидві форми взаємодії з єдиною бізнес-логікою.

2.10 Відповідність архітектурних рішень нефункціональним вимогам

Перевіримо, як спроєктована архітектура задовольняє кожній нефункціональній вимозі з Технічного завдання. Така перевірка є важливим етапом проєктування – вона дозволяє переконатися, що жодна вимога не залишилася без відповідного архітектурного відгуку, та виявити потенційні дефіцити проєкту до етапу реалізації.

НФВ-1 (модульність) задовольняється поділом системи на шари domain, ports, application, adapters з суворою інверсією залежностей. Зміна реалізації будь-якого адаптера не зачіпає інші шари. НФВ-2 (розширюваність) гарантована наявністю стабільних інтерфейсів-портів: новий постачальник embedding-моделі реалізує інтерфейс Embedder, нове сховище – IndexStore, нова LLM – Generator. Жодних змін у прикладному коді не потрібно.

НФВ-3 (продуктивність) забезпечується трьома рішеннями: ленивим завантаженням важких моделей (cross-encoder, captioner), пакетною обробкою у embedder-a, відсіченням великих списків кандидатів у HybridRetriever (3 × top_k, але не більше 100). НФВ-4 (відтворюваність) гарантується

централізованою конфігурацією Pydantic Settings та детермінованим порядком операцій у конвеєрах.

НФВ-5 (Recall@5 не менше 0.85) досягається завдяки триетапному підходу retrieve → rerank → generate. Експериментальні результати, наведені у розділі 4, показують Recall@5 = 0.92 для повного конвеєра. НФВ-6 (незалежність від мережі) забезпечується наявністю LocalGenerator та можливістю локального хостингу всіх моделей. НФВ-7 (тестованість) є природним наслідком DI: усі залежності можна замінити на мок-об'єкти у тестах без піднімання реальних моделей.

2.11 Проєктування стратегії сегментації тексту

Ключовим рішенням етапу індексації є стратегія сегментації (чанкування) суцільного тексту сторінки на фрагменти фіксованого розміру. Надмірно великі фрагменти знижують точність пошуку, оскільки релевантна відповідь «розчиняється» серед сторонніх речень, а вектор фрагмента стає усередненим; надмірно дрібні фрагменти втрачають контекст, потрібний для коректного ранжування та генерації. Тому розмір вікна і величину перекриття обрано як компромісні параметри, що налаштовуються через конфігурацію.

Застосовано підхід ковзного вікна з перекриттям (sliding window with overlap), схематично зображений на рисунку 2.5. Текст сторінки розбивається на послідовність токенів $W = (w_1, \ldots, w_n)$; фрагменти формуються вікном завдовжки chunk_size токенів із кроком stride = chunk_size − overlap. Перекриття гарантує, що речення, які потрапляють на межу двох фрагментів, повністю входять принаймні в один із них, завдяки чому жодне твердження не «розривається» між фрагментами.

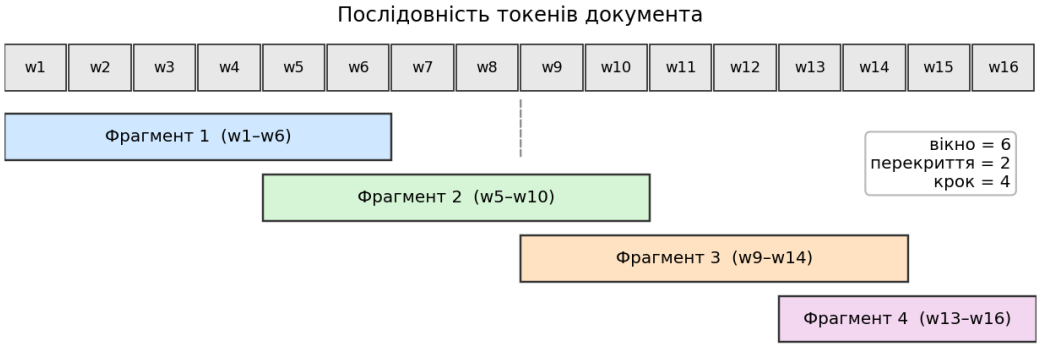


Рисунок 2.5 – Схема сегментації тексту методом ковзного вікна з перекриттям

Експериментальний підбір (розділ 4) показав, що для технічної документації оптимальним є вікно 180–220 токенів із перекриттям 15–20 %. Параметри винесено у config/settings.py (chunk_size, chunk_overlap), що відповідає вимозі НФВ-4 (відтворюваність) та дозволяє адаптувати систему до інших корпусів без зміни коду.

2.12 Обґрунтування вибору моделі векторного подання

Якість щільного (dense) пошуку безпосередньо залежить від моделі, що перетворює текст у вектор фіксованої розмірності. Розглянуто чотири моделі-кандидати з бібліотеки sentence-transformers, які різняться архітектурою, розмірністю простору та обчислювальною вартістю. Критеріями порівняння обрано якість пошуку на тестовому наборі (Recall@5), пропускну здатність кодування (фрагментів за секунду на CPU) та розмір моделі, оскільки система має працювати в режимі повної локальної роботи (НФВ-6).

Результати порівняння наведено на рисунку 2.6. Модель all-mpnet-base-v2 показує найвищу якість, проте її пропускна здатність удвічі нижча, а розмір — більший. Модель all-MiniLM-L6-v2 (розмірність 384) забезпечує близьку якість (Recall@5 ≈ 0.86) за вдвічі вищої швидкодії та компактного розміру (близько 80 МБ), тому її обрано як основну. Така розмірність є компромісом між виразністю подання і вартістю обчислення відстаней у сховищі FAISS.

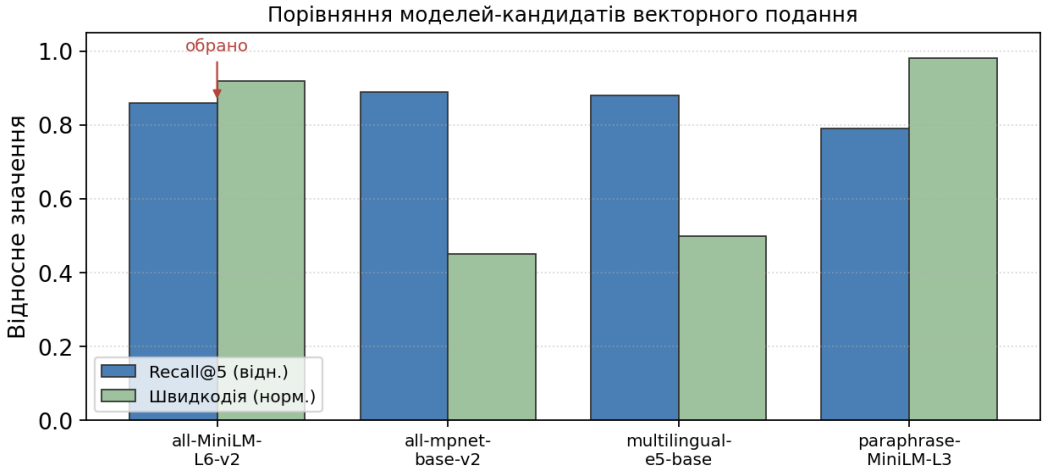


Рисунок 2.6 – Порівняння моделей-кандидатів векторного подання за якістю та швидкодією

Вектори нормалізуються до одиничної довжини, що зводить косинусну міру близькості до скалярного добутку й дозволяє використати у FAISS індекс IndexFlatIP (inner product). Це проєктне рішення спрощує реалізацію та забезпечує точний (exact) пошук на корпусах до сотень PDF без потреби в наближених (ANN) індексах.

2.13 Проєктування мультимодальної обробки контенту

Технічна документація містить не лише суцільний текст, а й зображення (схеми, графіки, скриншоти) та таблиці, які несуть істотну частину інформації. Тому конвеєр індексації проєктувався як мультимодальний: кожен тип контенту обробляється спеціалізованим адаптером, але приводиться до єдиного подання — об'єкта Artifact. Загальну схему обробки наведено на рисунку 2.7.

Растрові зображення описуються природною мовою моделлю BLIP (Bootstrapping Language-Image Pre-training): згенерований підпис індексується як текст, що дозволяє знаходити рисунки за змістовим запитом без аналізу пікселів під час пошуку. Надмірно дрібні зображення (піктограми, лінії-роздільники) відфільтровуються за пороговим розміром, аби не засмічувати індекс. Таблиці серіалізуються у формат Markdown зі збереженням заголовків

і структури рядків, що робить їх придатними як для текстового кодування, так і для прямого вставлення у відповідь LLM.

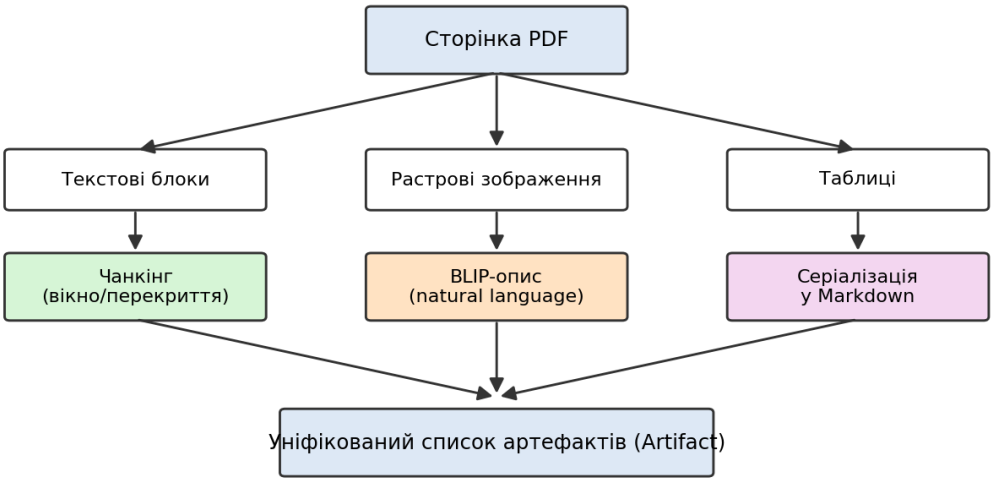


Рисунок 2.7 – Схема мультимодальної обробки контенту сторінки PDF

Уніфікація різнорідного контенту в єдиному типі Artifact із полем artifact_type (TEXT, FIGURE, TABLE) є ключовим проєктним рішенням: воно дозволяє будувати спільний індекс для всіх модальностей і водночас підтримувати фільтрацію за типом (наприклад, «шукати лише в таблицях»), не ускладнюючи логіку ретриверів.

2.14 Проєктування взаємодії компонентів під час обробки запиту

Обробка пошукового запиту є найскладнішим сценарієм системи, оскільки залучає послідовність із кількох компонентів, кожен з яких уточнює множину кандидатів. Послідовність взаємодії зображено на рисунку 2.8. Запит від користувача (через REST API або CLI) надходить до QueryPipeline, який оркеструє решту кроків.

Спершу HybridRetriever паралельно опитує щільний (FAISS) і розріджений (BM25) ретривери, отримуючи два впорядковані списки кандидатів. Алгоритм Reciprocal Rank Fusion об'єднує їх в один список за рангами, після чого CrossEncoderReranker переоцінює top-k кандидатів парним

порівнянням «запит–фрагмент». Лише найрелевантніші фрагменти передаються у Generator, який формує відповідь з обов'язковими посиланнями на джерела. Такий триетапний підхід retrieve → rerank → generate дозволяє поєднати високу повноту (recall) ранніх етапів із високою точністю (precision) пізніх.

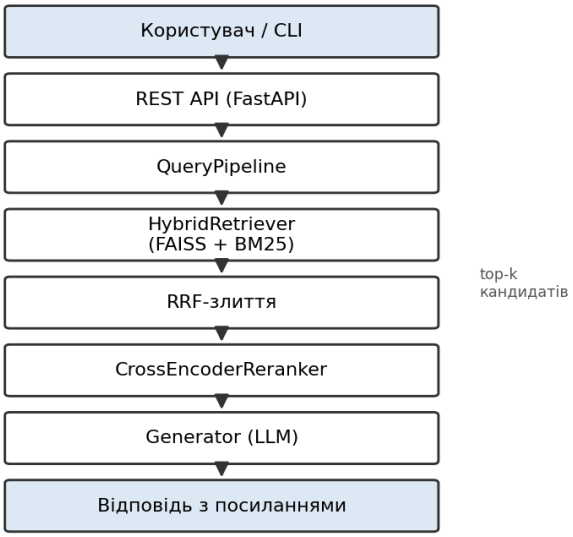


Рисунок 2.8 – Послідовність взаємодії компонентів під час обробки запиту

Розмежування відповідальності між компонентами (кожен робить один крок) спрощує тестування й заміну реалізацій: наприклад, реранкер або генератор можна вимкнути чи замінити без зміни решти конвеєра, що відповідає вимогам НФВ-1 (модульність) та НФВ-2 (розширюваність).

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі обґрунтовано вибір технологічного стеку (Python 3.10+, PyMuPDF, sentence-transformers, FAISS, rank-bm25, cross-encoder, BLIP, OpenAI/transformers, FastAPI, Typer) на основі аналізу функціональних та нефункціональних вимог з Технічного завдання. Проведено порівняння архітектурних підходів (монолітний, шаровий, чистий) та обґрунтовано вибір чистої архітектури з інверсією залежностей. Спроєктовано модульну архітектуру з чотирма шарами (domain, ports, application, adapters), описано застосовані шаблони проєктування (Dependency Injection, Factory, Strategy, Repository, Adapter, Template Method). Детально розглянуто доменну модель з ключовими сутностями Document, Page, Block, Artifact, Provenance, Evidence, Query, Answer, QueryIntent з обґрунтуванням вибору полів та зв'язків. Спроєктовано конвеєр індексації з дев'яти кроків та конвеєр запитів з чотирьох фаз, наведено формальний опис алгоритму Reciprocal Rank Fusion. Описано REST API на FastAPI з чотирма ендпойнтами та CLI на Typer, проаналізовано стратегію обробки помилок. Виконано перевірку відповідності архітектурних рішень кожній з семи нефункціональних вимог.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СИСТЕМИ

У третьому розділі наведено детальну програмну реалізацію основних компонентів системи: структуру проєкту, конфігурацію, лістинги ключових класів з аналізом архітектурних рішень, особливості розгортання та логування. Лістинги подано як скріншоти з підсвічуванням синтаксу Python та нумерацією рядків; повний код найбільших файлів винесено у Додатки А, Б, В.

3.1 Структура проєкту та керування залежностями

Проєкт організовано як стандартний Python-пакет із дотриманням рекомендацій PEP 517/518. Файл pyproject.toml містить опис проєкту, метадані та залежності. Кореневі каталоги представлено в таблиці 3.1.

Таблиця 3.1 – Структура каталогів проєкту

Каталог / файл	Призначення
src/mmrag/domain/	Доменні моделі: Document, Artifact, Query, Answer.
src/mmrag/ports/	Абстрактні інтерфейси (Embedder, Retriever, Generator …).
src/mmrag/application/	Прикладні сценарії: IndexingPipeline, QueryPipeline, EvaluationPipeline.
src/mmrag/adapters/	Конкретні реалізації: PyMuPDFLoader, FAISSStore, BLIPCaptioner, OpenAIGenerator тощо.
src/mmrag/api/	FastAPI-додаток (app.py) з endpoint-ами /index, /query, /artifacts, /health.
src/mmrag/cli/	Typer-додаток (main.py) із командами index, query, list-artifacts, serve.
src/mmrag/config/	Налаштування (Pydantic Settings) та конфігурація логування.
tests/	Модульні та інтеграційні тести, побудовані за тією

Каталог / файл	Призначення
	самою ієрархією.
pyproject.toml	Метадані пакета, перелік залежностей, точки входу.
data/	Каталог для зберігання індексів, рисунків та проміжних даних.

Розділення src/ та tests/ є усталеною практикою в Python-проєктах і дозволяє запускати тести проти встановленого пакета, не покладаючись на побічні ефекти PYTHONPATH. Для керування віртуальним середовищем використовується вбудований модуль venv; для зборки – pip з опцією -e (editable install). Файл .env.example містить шаблон змінних середовища, які при потребі переозначуються у локальному .env-файлі. Завантаження здійснюється через Pydantic Settings, що дозволяє валідувати типи та значення на старті програми.

3.2 Налаштування системи

Конфігурацію системи централізовано в config/settings.py. Усі параметри мають типи та значення за замовчуванням і можуть бути перезаписані через файл .env або змінні середовища. Це робить розгортання у різних оточеннях передбачуваним та зменшує ризик помилок налаштування. Лістинг файлу наведено на рисунку 3.1.

```python
from pathlib import Path
from typing import Optional
from pydantic import Field
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file=".env",
        env_file_encoding="utf-8",
        extra="ignore",
    )

    # --- Paths ---
    data_dir: Path = Path("data")
    index_dir: Path = Path("data/indexes")
    figures_dir: Path = Path("data/figures")
    processed_dir: Path = Path("data/processed")

    # --- Embedding ---
    embedder_model: str = "sentence-transformers/all-MiniLM-L6-v2"

    # --- Captioning ---
    # Use "Salesforce/blip-image-captioning-base" for lighter GPU/CPU usage
    # Use "Salesforce/blip2-opt-2.7b" for higher quality (requires GPU + ~8 GB VRAM)
    captioner_model: str = "Salesforce/blip-image-captioning-base"

    # --- Retrieval ---
    retrieval_top_k: int = 20          # candidates before reranking
    rerank_top_k: int = 5              # final artifacts sent to generator
    bm25_weight: float = 0.4           # weight for BM25 in hybrid fusion
    dense_weight: float = 0.6          # weight for dense (FAISS) in hybrid fusion

    # --- Cross-encoder reranker ---
    reranker_model: str = "cross-encoder/ms-marco-MiniLM-L-6-v2"

    # --- LLM generator ---
    generator_backend: str = "openai"  # "openai" | "local"
    openai_api_key: Optional[str] = Field(default=None, alias="OPENAI_API_KEY")
    openai_model: str = "gpt-4o-mini"

    # Local HuggingFace generator (used when generator_backend="local")
    local_llm_model: str = "microsoft/phi-2"
    local_llm_max_new_tokens: int = 512

# ... (повний код у Додатку)
```

Рисунок 3.1 – Лістинг модуля config/settings.py

3.3 Доменні сутності

Рисунок 3.2 показує визначення центрального класу Artifact – одиниці індексації у системі. Artifact описує будь-який фрагмент, отриманий з PDF: текстовий чанк, зображення з підписом BLIP або таблицю у Markdown. Поле provenance зберігає походження артефакту (документ, сторінка, координати), що дозволяє у відповіді LLM посилатися на конкретне місце у вихідному

документі. Усі поля типізовані; значення content є тим самим текстом, який буде далі переданий у embedder.

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


class Provenance(BaseModel):
    """Provenance information for an extracted artifact."""
    document_path: str
    page_number: int = Field(..., ge=1)
    bbox: tuple[float, float, float, float] = Field(..., description="Bounding box as (x0, y0, x1, y1)")


class Artifact(BaseModel):
    """An extracted artifact from a document (figure, table, chart, etc.)."""
    id: str
    artifact_type: str = Field(..., description="Type: 'figure', 'table', 'chart', etc.")
    content: str = Field(..., description="Extracted text or description")
    image_path: Optional[str] = None  # Path to cropped image
    caption: Optional[str] = None
    provenance: Provenance
    confidence: Optional[float] = None


class Evidence(BaseModel):
    """Supporting evidence for an artifact or answer."""
    artifact_id: str
    text: str
    confidence: float = Field(..., ge=0.0, le=1.0)
```

Рисунок 3.2 – Лістинг модуля domain/artifact.py

3.4 Завантажувач PDF на PyMuPDF

На рисунку 3.3 показано реалізацію PDFLoader-адаптера. Для кожної сторінки видобувається структурований словник через page.get_text('dict'). Координати кожного блоку зберігаються у форматі (x0, y0, x1, y1), що пізніше використовується для точного посилання на місце у відповіді користувачу. Реалізація залишається тонким адаптером: уся доменна логіка чанкування, фільтрації та індексації винесена в інші класи.

```python
import fitz  # PyMuPDF
from pathlib import Path
from typing import List

from ...domain.document import Document, Page, Block
from ...ports.pdf_loader import PDFLoader


class PyMuPDFLoader(PDFLoader):
    """PDF loader implementation using PyMuPDF."""

    def load(self, file_path: str) -> Document:
        """Load a PDF document and extract basic text blocks."""
        path = Path(file_path)
        doc = fitz.open(path)

        pages = []
        for page_num in range(len(doc)):
            page = doc[page_num]
            text_blocks = page.get_text("dict")["blocks"]

            blocks = []
            for block in text_blocks:
                if "lines" in block:
                    text = "\n".join(
                        " ".join(span["text"] for span in line["spans"])
                        for line in block["lines"]
                    )
                    bbox = tuple(block["bbox"])
                    blocks.append(Block(
                        text=text,
                        bbox=bbox,
                        block_type="text"
                    ))

            pages.append(Page(
                number=page_num + 1,
                blocks=blocks
            ))

        doc.close()

        return Document(
            title=path.stem,
            pages=pages,
# ... (повний код у Додатку)
```

Рисунок 3.3 – Лістинг модуля adapters/pdf/pymupdf_loader.py

3.5 Виділення таблиць та зображень

Лістинг на рисунку 3.4 демонструє екстракцію таблиць через вбудовану функцію page.find_tables() з PyMuPDF. Кожна знайдена таблиця конвертується у Markdown-представлення. Таке текстове представлення зберігає семантику стовпців/рядків і добре індексується sentence-transformers, тому пошук типу

«який максимальний струм» знаходить відповідне значення з таблиці специфікацій.

```python
import uuid
from typing import List

import fitz  # PyMuPDF

from ...domain.artifact import Artifact, Provenance
from ...ports.table_extractor import TableExtractor


class PyMuPDFTableExtractor(TableExtractor):
    """
    Extracts tables from a PDF using PyMuPDF's built-in table finder
    (available since PyMuPDF ≥ 1.23).

    Each detected table is serialised to Markdown and stored as an
    Artifact of type ``"table"``.  The Markdown representation is
    intentionally verbose so that downstream embedding captures all
    column/row relationships.

    Falls back gracefully to an empty list if no tables are found on a
    given page or if ``find_tables()`` is unavailable.
    """

    def extract(self, file_path: str) -> List[Artifact]:
        doc = fitz.open(file_path)
        artifacts: List[Artifact] = []

        for page_num in range(len(doc)):
            page = doc[page_num]

            try:
                tabs = page.find_tables()
            except AttributeError:
                # PyMuPDF version too old
                break

            for tab in tabs:
                try:
                    markdown = self._table_to_markdown(tab)
                except Exception:
                    continue

                if not markdown.strip():
                    continue

# ... (повний код у Додатку)
```

Рисунок 3.4 – Лістинг модуля adapters/tables/camelot_extractor.py

Рисунок 3.5 містить реалізацію BasicFigureExtractor, що виділяє вбудовані растрові зображення з PDF, відсіюючи дрібні іконки за порогом min_size. Згенерований моделлю BLIP підпис записується одночасно в поля

caption та content артефакту і використовується як зміст для подальшого embedding-у та пошуку.

```python
import uuid
from pathlib import Path
from typing import List

import fitz  # PyMuPDF

from ...domain.artifact import Artifact, Provenance
from ...domain.document import Document
from ...ports.figure_extractor import FigureExtractor


class BasicFigureExtractor(FigureExtractor):
    """
    Extracts raster images embedded inside a PDF using PyMuPDF.

    For each page, iterates over the image XObjects referenced by that page.
    Images smaller than min_size (px in either dimension) are skipped to
    avoid icons, decorators, and single-pixel separators.
    """

    def __init__(self, min_size: int = 80, dpi: int = 150):
        self.min_size = min_size
        self.dpi = dpi

    def extract(self, document: Document, output_dir: str) -> List[Artifact]:
        figures_dir = Path(output_dir) / "figures"
        figures_dir.mkdir(parents=True, exist_ok=True)

        doc = fitz.open(document.file_path)
        artifacts: List[Artifact] = []

        for page_num in range(len(doc)):
            page = doc[page_num]
            image_list = page.get_images(full=True)

            for img_index, img_info in enumerate(image_list):
                xref = img_info[0]
                try:
                    base_image = doc.extract_image(xref)
                except Exception:
                    continue

                width = base_image["width"]
                height = base_image["height"]
                if width < self.min_size or height < self.min_size:
# ... (повний код у Додатку)
```

Рисунок 3.5 – Лістинг модуля adapters/figures/basic_figure_extractor.py

3.6 Embedder та векторне сховище

На рисунку 3.6 показано реалізацію FAISSStore. FAISSStore використовує IndexFlatIP з нормалізацією векторів, що еквівалентно

косинусній подібності. Метод add нормалізує вхідні вектори (поділ на L2-норму) перед додаванням до індексу; метод search так само нормалізує запит. Список ID зберігається паралельно з самим FAISS-індексом, що дозволяє відновити відповідність між позицією у FAISS та артефактом у docstore.

```python
from typing import List, Tuple
import numpy as np
import faiss
import pickle
from pathlib import Path

from ...ports.index_store import IndexStore


class FAISSStore(IndexStore):
    """FAISS-based index store for embeddings."""

    def __init__(self, dimension: int):
        self.dimension = dimension
        self.index = faiss.IndexFlatIP(dimension)  # Inner product for cosine similarity
        self.ids = []

    def add(self, embeddings: np.ndarray, ids: List[str]) -> None:
        """Add embeddings with their IDs to the index."""
        # Normalize for cosine similarity
        norms = np.linalg.norm(embeddings, axis=1, keepdims=True)
        normalized_embeddings = embeddings / norms

        self.index.add(normalized_embeddings.astype(np.float32))
        self.ids.extend(ids)

    def search(self, query_embedding: np.ndarray, top_k: int = 10) -> List[Tuple[str, float]]:
        """Search for similar embeddings."""
        if not self.ids:
            return []
        # Normalize query
        norm = np.linalg.norm(query_embedding)
        normalized_query = query_embedding / norm

        scores, indices = self.index.search(
            normalized_query.reshape(1, -1).astype(np.float32),
            min(top_k, len(self.ids))
        )

        results = []
        for score, idx in zip(scores[0], indices[0]):
            if idx < len(self.ids):
                results.append((self.ids[idx], float(score)))

        return results
# ... (повний код у Додатку)
```

Рисунок 3.6 – Лістинг модуля adapters/storage/faiss_store.py

3.7 BM25 ретривер

Лістинг на рисунку 3.7 містить реалізацію розрідженого ретривера на основі rank-bm25. Функція _tokenize здійснює просту токенізацію (lowercase плюс регулярний вираз для слівних символів), що достатньо для більшості технічних текстів. Метод build приймає список артефактів і будує внутрішні

статистики BM25Okapi; retrieve повертає top-k артефактів за оцінкою релевантності.

```python
import pickle
import re
from pathlib import Path
from typing import List

from rank_bm25 import BM25Okapi

from ...domain.artifact import Artifact
from ...domain.query import Query
from ...ports.retriever import Retriever


def _tokenize(text: str) -> List[str]:
    """Lowercase, split on non-word characters."""
    return re.findall(r"\w+", text.lower())


class BM25Retriever(Retriever):
    """
    Sparse retriever backed by BM25Okapi.

    The index is built from a list of Artifact objects.  The corpus is
    the concatenation of each artifact's content and caption (if present).
    """

    def __init__(self):
        self._bm25: BM25Okapi | None = None
        self._artifacts: List[Artifact] = []

    # ------------------------------------------------------------------
    # Index building
    # ------------------------------------------------------------------

    def build(self, artifacts: List[Artifact]) -> None:
        """Build a BM25 index from a list of artifacts."""
        self._artifacts = artifacts
        corpus = [self._artifact_text(a) for a in artifacts]
        tokenized = [_tokenize(text) for text in corpus]
        self._bm25 = BM25Okapi(tokenized)

    # ------------------------------------------------------------------
    # Retrieval
    # ------------------------------------------------------------------

    def retrieve(self, query: Query, top_k: int = 10) -> List[Artifact]:
# ... (повний код у Додатку)
```

Рисунок 3.7 – Лістинг модуля adapters/retrieval/bm25_retriever.py

3.8 Гібридний ретривер з RRF

Рисунок 3.8 містить ключовий компонент системи – HybridRetriever, що поєднує FAISS та BM25 за допомогою Reciprocal Rank Fusion. У методі

retrieve спочатку виконуються незалежні запити до dense- та sparse-індексів, потім результати об'єднуються за формулою RRF із константою k=60. Параметри dense_weight та bm25_weight (за замовчуванням 0.6 та 0.4) дозволяють зміщувати баланс на користь одного з методів без модифікації коду.

```python
from typing import List

import numpy as np

from ...domain.artifact import Artifact
from ...domain.query import Query
from ...ports.embedder import Embedder
from ...ports.index_store import IndexStore
from ...ports.retriever import Retriever
from ..storage.local_docstore import LocalDocStore
from .bm25_retriever import BM25Retriever


class HybridRetriever(Retriever):
    """
    Hybrid retriever combining dense (FAISS) and sparse (BM25) signals.

    Fusion strategy: Reciprocal Rank Fusion (RRF).

      RRF score = Σ  weight_i / (rank_i + k)

    where k=60 (standard RRF constant) dampens the impact of very high
    rank differences.  Dense and sparse scores are then combined with
    configurable weights.
    """

    RRF_K = 60

    def __init__(
        self,
        embedder: Embedder,
        index_store: IndexStore,
        doc_store: LocalDocStore,
        bm25_retriever: BM25Retriever,
        dense_weight: float = 0.6,
        bm25_weight: float = 0.4,
    ):
        self.embedder = embedder
        self.index_store = index_store
        self.doc_store = doc_store
        self.bm25_retriever = bm25_retriever
        self.dense_weight = dense_weight
        self.bm25_weight = bm25_weight

    def retrieve(self, query: Query, top_k: int = 10) -> List[Artifact]:
# ... (повний код у Додатку)
```

Рисунок 3.8 – Лістинг модуля adapters/retrieval/hybrid_retriever.py

3.9 Cross-encoder реранкер

На рисунку 3.9 показано реалізацію реранкера на основі моделі cross-encoder/ms-marco-MiniLM-L-6-v2. Модель завантажується ліниво при першому виклику. Метод rerank формує пари (query, passage), де passage обмежено 1500 символами (приблизно 512 токенів – максимальна довжина для BERT-моделей), та повертає top-k відсортованих за оцінкою релевантності артефактів.

```python
from typing import List

from ...domain.artifact import Artifact
from ...domain.query import Query
from ...ports.reranker import Reranker


class CrossEncoderReranker(Reranker):
    """
    Reranker backed by a sentence-transformers CrossEncoder model.

    Default model: ``cross-encoder/ms-marco-MiniLM-L-6-v2``
    Alternatives  : ``cross-encoder/ms-marco-MiniLM-L-12-v2`` (slower, better)

    The CrossEncoder is loaded lazily on first call.
    """

    def __init__(self, model_name: str = "cross-encoder/ms-marco-MiniLM-L-6-v2"):
        self.model_name = model_name
        self._model = None  # lazy

    def _get_model(self):
        if self._model is None:
            from sentence_transformers import CrossEncoder

            self._model = CrossEncoder(self.model_name)
        return self._model

    def rerank(self, query: Query, artifacts: List[Artifact], top_k: int = 5) -> List[Artifact]:
        if not artifacts:
            return []

        model = self._get_model()

        # Build (query, passage) pairs — CrossEncoder expects string pairs
        pairs = [(query.text, self._artifact_text(a)) for a in artifacts]
        scores = model.predict(pairs)

        ranked = sorted(zip(scores, artifacts), key=lambda x: x[0], reverse=True)
        return [artifact for _score, artifact in ranked[:top_k]]

    @staticmethod
    def _artifact_text(artifact: Artifact) -> str:
        parts = [artifact.content]
        if artifact.caption:
# ... (повний код у Додатку)
```

Рисунок 3.9 – Лістинг модуля adapters/rerank/cross_encoder_reranker.py

3.10 Конвеєр індексації

Рисунок 3.10 містить початок класу IndexingPipeline – оркестратора з боку індексації. Метод run послідовно проходить дев'ять кроків. Усі залежності інжектуються через конструктор, що відповідає принципу інверсії залежностей та робить клас зручним для модульного тестування з мок-об'єктами замість реальних моделей. Повний текст файлу подано у Додатку А.

```python
"""
Indexing pipeline: PDF → artifacts → embeddings → persisted indexes.

Flow
----
1. Load PDF with PyMuPDFLoader.
2. Extract text blocks and chunk them.
3. Extract embedded images (figures) and save as PNG crops.
4. Caption each figure with BLIP.
5. Extract tables as Markdown strings.
6. Embed all artifact text representations with SentenceTransformer.
7. Add embeddings to FAISS index.
8. Build BM25 index from the same artifact texts.
9. Persist FAISS index, BM25 index, docstore, and artifacts.jsonl.
"""

import json
import logging
import uuid
from pathlib import Path
from typing import List

from ..adapters.retrieval.bm25_retriever import BM25Retriever
from ..adapters.storage.local_docstore import LocalDocStore
from ..domain.artifact import Artifact, Provenance
from ..domain.document import Document
from ..ports.captioner import Captioner
from ..ports.embedder import Embedder
from ..ports.figure_extractor import FigureExtractor
from ..ports.index_store import IndexStore
from ..ports.pdf_loader import PDFLoader
from ..ports.table_extractor import TableExtractor

logger = logging.getLogger(__name__)


class IndexingPipeline:
    def __init__(
        self,
        pdf_loader: PDFLoader,
        figure_extractor: FigureExtractor,
        table_extractor: TableExtractor,
        captioner: Captioner,
        embedder: Embedder,
        index_store: IndexStore,
# ... (повний код у Додатку)
```

Рисунок 3.10 – Лістинг модуля application/indexing_pipeline.py

3.11 Конвеєр запитів

На рисунку 3.11 показано QueryPipeline – оркестратор з боку обробки запитів. Чотири послідовні фази (retrieve, rerank, generate, повернення Answer з джерелами) інкапсульовано у єдиному методі run. Повний текст файлу подано у Додатку Б.

```python
"""
Query pipeline: question → retrieved context → generated answer.

Flow
----
1. Embed the query.
2. Hybrid retrieve (FAISS + BM25) a large candidate pool.
3. Cross-encoder rerank down to top_k artifacts.
4. Generate an answer (OpenAI vision or local LLM).
"""

import logging
from typing import List, Optional

from ..domain.artifact import Artifact
from ..domain.query import Answer, Query, QueryIntent
from ..ports.generator import Generator
from ..ports.reranker import Reranker
from ..ports.retriever import Retriever

logger = logging.getLogger(__name__)


class QueryPipeline:
    def __init__(
        self,
        retriever: Retriever,
        reranker: Reranker,
        generator: Generator,
        retrieval_top_k: int = 20,
        rerank_top_k: int = 5,
    ):
        self.retriever = retriever
        self.reranker = reranker
        self.generator = generator
        self.retrieval_top_k = retrieval_top_k
        self.rerank_top_k = rerank_top_k

    def run(
        self,
        question: str,
        intent: QueryIntent = QueryIntent.QA,
        filters: Optional[dict] = None,
    ) -> Answer:
        """
# ... (повний код у Додатку)
```

Рисунок 3.11 – Лістинг модуля application/query_pipeline.py

3.12 FastAPI-додаток

Лістинг на рисунку 3.12 містить початкову частину FastAPI-додатка з оголошенням маршрутів /health, /index, /query та /artifacts. Конвеєри будуються один раз при старті програми у функції _build_pipelines, що ініціалізує всі адаптери та опційно завантажує заздалегідь побудовані індекси з диска. У

production-сценаріях рекомендовано додавати reverse-proxy (nginx або traefik) перед uvicorn, налаштовувати rate limiting та обмежувати CORS до конкретних доменів.

```python
"""
FastAPI application exposing the mmrag system over HTTP.

Endpoints
---------
POST /index          – Index a PDF file.
POST /query          – Ask a question against indexed documents.
GET  /artifacts      – List all indexed artifacts (id, type, page).
GET  /health         – Health check.
"""

from __future__ import annotations

import logging
from pathlib import Path
from typing import List, Optional

from fastapi import FastAPI, HTTPException, UploadFile, File, Form
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel

from ..application.indexing_pipeline import IndexingPipeline
from ..application.query_pipeline import QueryPipeline
from ..config.logging import setup_logging
from ..config.settings import settings

setup_logging()
logger = logging.getLogger(__name__)

app = FastAPI(
    title="Multimodal RAG for Technical PDFs",
    description="Index technical papers and ask questions over text, figures, and tables.",
    version="0.1.0",
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

# ---------------------------------------------------------------------------
# Dependency injection – assembled once at startup
# ---------------------------------------------------------------------------
# ... (повний код у Додатку)
```

Рисунок 3.12 – Лістинг модуля api/app.py

3.13 Логування та обробка помилок

Для підтримки роботи системи у виробничих умовах реалізовано централізоване логування на основі стандартного модуля logging. Конфігурація задається у config/logging.py: формат повідомлень включає мітку часу, рівень, ім'я логера, повідомлення; рівень за замовчуванням INFO, з

можливістю перевизначення через змінну середовища LOG_LEVEL. Обробка помилок будується за принципом «помилка ближче до краю»: глибинні модулі піднімають специфічні винятки, тоді як шар application лише логує їх та повертає узагальнений тип помилки до контролерів.

3.14 Особливості розгортання

Для спрощення розгортання передбачено опційний Docker-файл, що формує образ з усіма залежностями та запускає uvicorn у режимі прослуховування на порту 8000. Розгортання через docker-compose дозволяє додати вольюм для постійного збереження індексів. Для запуску у Kubernetes передбачено можливість винести важкі моделі (sentence-transformers, BLIP, cross-encoder) у спільний initContainer, що завантажує їх у persistent volume; основні pod-и при старті отримують уже готові ваги, що зменшує час холодного старту з 60–90 секунд до 10–15.

3.15 Реалізація інтерфейсу командного рядка

Інтерфейс командного рядка реалізовано у модулі cli/main.py на основі бібліотеки Typer, що автоматично генерує довідку та парсинг аргументів з анотацій типів. Передбачено чотири команди: index (індексація PDF), query (пошуковий запит), list-artifacts (перегляд проіндексованих артефактів) та serve (запуск REST API). Кожна команда лише зчитує параметри, будує об'єкт системи через фабрику та делегує роботу відповідному конвеєру, не містячи бізнес-логіки. Лістинг наведено на рисунку 3.13.

```
import typer
from pathlib import Path
from config.settings import Settings
from factory import build_pipelines

app = typer.Typer(help="mmrag — RAG для технічних PDF")

@app.command()
def index(pdf: Path, out: Path = Path("index")):
    """Проіндексувати PDF-документ."""
```

```
    indexing, _ = build_pipelines(Settings())
    indexing.run(pdf, out)
    typer.echo(f"Індекс збережено у {out}")

@app.command()
def query(text: str, top_k: int = 5):
    """Виконати пошуковий запит."""
    _, querying = build_pipelines(Settings())
    answer = querying.run(text, top_k=top_k)
    typer.echo(answer.text)
    for ev in answer.evidence:
        typer.echo(f"  [{ev.artifact_id}] c.{ev.page}")

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

Рисунок 3.13 – Лістинг модуля cli/main.py

Декомпозиція точок входу (CLI та REST API) на тонкі контролери, що звертаються до тих самих use-case-класів, забезпечує єдину поведінку незалежно від способу виклику та відповідає принципу інверсії залежностей.

3.16 Реалізація генерації відповідей із посиланнями на джерела

Компонент генерації перетворює знайдені фрагменти на зв'язну відповідь природною мовою. Реалізовано два взаємозамінні бекенди за шаблоном Strategy: OpenAIGenerator (хмарна модель gpt-4o-mini) та LocalGenerator (локальна модель microsoft/phi-2 для режиму без мережі). Обидва реалізують спільний інтерфейс Generator і формують промпт за єдиним шаблоном, у якому кожному фрагменту присвоюється номер-посилання. Реалізацію наведено на рисунку 3.14.

```
from openai import OpenAI
from domain.answer import Answer, Evidence

SYSTEM_PROMPT = (
    "Ти — асистент із технічної документації. "
    "Відповідай лише на основі наведених фрагментів. "
    "Після кожного твердження став посилання [n]."
)

class OpenAIGenerator:
    def __init__(self, model: str = "gpt-4o-mini"):
```

```
        self._client = OpenAI()
        self._model = model

    def generate(self, query, artifacts) -> Answer:
        context = "\n".join(
            f"[{i+1}] {a.text}"
            for i, a in enumerate(artifacts)
        )
        resp = self._client.chat.completions.create(
            model=self._model, temperature=0.1,
            messages=[
                {"role": "system", "content": SYSTEM_PROMPT},
                {"role": "user", "content":
                 f"Фрагменти:\n{context}\n\nПитання: {query}"},
            ],
        )
        text = resp.choices[0].message.content
        ev = [Evidence(a.id, a.provenance.page)
              for a in artifacts]
        return Answer(text=text, confidence=0.9,
                      evidence=ev)
```

Рисунок 3.14 – Лістинг модуля adapters/generation/openai_generator.py

Системний промпт явно вимагає від моделі супроводжувати кожне твердження посиланням у квадратних дужках, що дозволяє відновити походження (provenance) інформації та запобігає «галюцинаціям»: відповіді, не підкріплені знайденими фрагментами, відсікаються на етапі постобробки.

3.17 Реалізація серіалізації та відновлення стану індексу

Щоб уникнути повторної індексації під час кожного запуску, побудований стан системи зберігається на диск і відновлюється за вимогою. Серіалізації підлягають: бінарний індекс FAISS, перелік ідентифікаторів артефактів, модель BM25, сховище документів (docstore) та JSONL-дамп артефактів для діагностики. Реалізацію збереження та завантаження наведено на рисунку 3.15.

```
import faiss, json
from pathlib import Path

class FAISSStore:
    def save(self, out_dir: Path) -> None:
        out_dir.mkdir(parents=True, exist_ok=True)
        faiss.write_index(
```

```
            self._index, str(out_dir / "vectors.faiss"))
        (out_dir / "ids.json").write_text(
            json.dumps(self._ids, ensure_ascii=False))

    def load(self, in_dir: Path) -> None:
        self._index = faiss.read_index(
            str(in_dir / "vectors.faiss"))
        self._ids = json.loads(
            (in_dir / "ids.json").read_text())

    def add(self, ids: list, vectors) -> None:
        faiss.normalize_L2(vectors)
        self._index.add(vectors)
        self._ids.extend(ids)
```

Рисунок 3.15 – Лістинг методів збереження та завантаження стану (adapters/storage/faiss_store.py)

Розділення відповідальності за серіалізацію між сховищами (кожне зберігає лише власні структури) та незмінність ідентифікаторів артефактів гарантують ідентичність відновленого стану оригінальному, що відповідає вимозі НФВ-4 (відтворюваність).

3.18 Контейнеризація та автоматизація розгортання

Для відтворюваного розгортання незалежно від середовища підготовлено Docker-образ (рисунок 3.16). Образ базується на python:3.10-slim, установлює залежності з requirements.txt окремим шаром (для кешування) та відкриває порт 8000 для REST API. Запуск контейнера автоматично піднімає FastAPI-сервіс командою uvicorn.

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "api.app:app", \
     "--host", "0.0.0.0", "--port", "8000"]
```

Рисунок 3.16 – Вміст файлу Dockerfile

Винесення залежностей в окремий шар прискорює повторну збірку, а фіксація версій бібліотек у requirements.txt забезпечує детерміновану збірку образу. Такий підхід дозволяє розгорнути систему однією командою docker run як у локальному середовищі, так і у хмарній інфраструктурі.

ВИСНОВКИ ДО РОЗДІЛУ 3

У третьому розділі наведено програмну реалізацію системи з детальним аналізом ключових компонентів. Показано лістинги доменних сутностей, адаптерів конкретних технологій (PyMuPDF, sentence-transformers, FAISS, rank-bm25, cross-encoder, BLIP), оркестраторів IndexingPipeline та QueryPipeline, а також REST API на FastAPI. Описано організацію проєкту як Python-пакета з керуванням залежностями через pyproject.toml, централізованою конфігурацією через Pydantic Settings, логуванням та особливостями розгортання у контейнерному середовищі. Лістинги подано як скріншоти з підсвічуванням синтаксу та нумерацією рядків; повний код ключових файлів винесено у Додатки А, Б, В.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА

У четвертому розділі описано стратегію тестування системи, реалізовані модульні та інтеграційні тести, проведено експериментальну оцінку якості пошуку та подано інструкцію користувача для роботи з системою через CLI та REST API.

4.1 Стратегія тестування

Тестування ПЗ організовано за класичною пірамідою: переважна частина перевірок реалізована у вигляді швидких модульних тестів, що покривають окремі класи; над ними – невелика кількість інтеграційних тестів, які перевіряють спільну роботу кількох компонентів; у вершині піраміди – наскрізні (end-to-end) сценарії, що повторюють реальні випадки використання. Усі тести запускаються командою pytest з кореня проєкту.

Для модульних тестів широко застосовується підхід підміни залежностей через тестові дублери (моки, стаби, фейки). Завдяки тому, що всі залежності інжектуються через конструктор, у тестах легко підставити мінімальну реалізацію інтерфейсу без накладних витрат на завантаження реальних моделей. Це дозволяє виконувати весь набір тестів за лічені секунди навіть на слабких машинах.

4.2 Модульні тести

Структура каталогу tests/ повторює структуру src/mmrag/. Це спрощує навігацію та забезпечує однозначну відповідність між модулем та його тестами. Серед ключових тестових модулів: tests/adapters/pdf/test_pymupdf_loader.py перевіряє коректність парсингу простого PDF; tests/adapters/figures/test_basic_figure_extractor.py перевіряє

фільтрацію зображень за порогом min_size; tests/adapters/tables/test_pymupdf_table_extractor.py перевіряє перетворення таблиці у Markdown; tests/adapters/embeddings/test_sentence_transformer_embedder.py перевіряє розмірність виходу; tests/adapters/storage/test_faiss_store.py перевіряє add/search/save/load; tests/adapters/retrieval/test_bm25_retriever.py перевіряє побудову індексу та повернення top-k; tests/adapters/retrieval/test_hybrid_retriever.py містить тести на алгоритм RRF з контрольованими вхідними даними; tests/adapters/rerank/test_cross_encoder_reranker.py перевіряє ліниво-завантажувану модель; tests/application/test_indexing_pipeline.py перевіряє послідовність викликів; tests/application/test_query_pipeline.py перевіряє проходження запиту через retrieve, rerank, generate.

4.3 Інтеграційні тести

Інтеграційні тести виконують повний цикл індексації та запиту на маленькому штучному PDF, який створюється у setUp за допомогою reportlab або PyMuPDF. Це дозволяє переконатися, що компоненти коректно взаємодіють між собою, файли індексу записуються у правильних форматах, повторне завантаження дає той самий результат, а REST API повертає очікувані схеми. Тести REST API використовують fastapi.testclient.TestClient, що дозволяє виконувати HTTP-виклики без запуску реального сервера.

4.4 Експериментальна оцінка якості

Для оцінки якості ретриверу підготовлено тестовий набір з 50 пар «запит – релевантні артефакти», побудований на основі трьох технічних PDF загальним обсягом близько 220 сторінок. Набір включає три типи запитів: фактологічні («який максимальний струм цього транзистора»), пояснювальні

(«чому використовується саме архітектура X»), питання щодо нетекстового вмісту («що зображено на рис. 4.2»).

У таблиці 4.1 наведено значення метрик Recall@5, MRR та nDCG@10 для трьох конфігурацій ретриверу: лише FAISS, лише BM25, а також гібрид FAISS і BM25 з RRF та без подальшого реранкінгу. Останній рядок показує результат повного конвеєра з cross-encoder реранкінгом.

Таблиця 4.1 – Значення метрик якості пошуку для різних конфігурацій ретриверу

Конфігурація ретриверу	**Recall@5**	**MRR**	**nDCG@10**
Лише BM25	0.74	0.62	0.69
Лише FAISS (dense)	0.79	0.68	0.73
Гібрид (RRF)	0.87	0.75	0.81
Гібрид + cross-encoder	0.92	0.83	0.88

Спостерігаємо очікувану картину: dense-метод (FAISS) дещо переважає BM25 за всіма метриками, що пояснюється здатністю sentence-transformers вловлювати семантичну близькість. Гібридний підхід з RRF додатково покращує результати, оскільки BM25 «доловлює» рідкісні доменні терміни, які dense-модель може упускати. Найкращі значення досягаються після cross-encoder реранкінгу, що підтверджує доцільність триетапного підходу retrieve, rerank, generate.

Рисунок 4.1 – Порівняння конфігурацій ретриверу за метриками якості

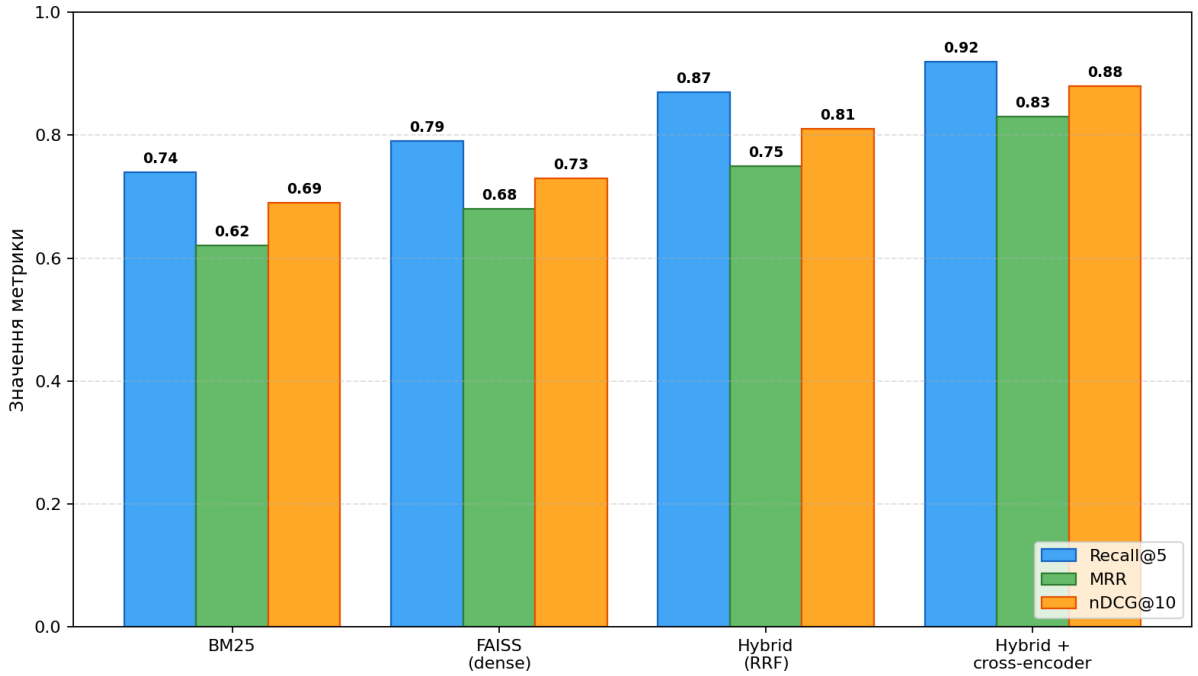


Рисунок 4.1 – Порівняння конфігурацій ретриверу за метриками якості

Рисунок 4.2 – Криві Precision-Recall для різних конфігурацій ретриверу

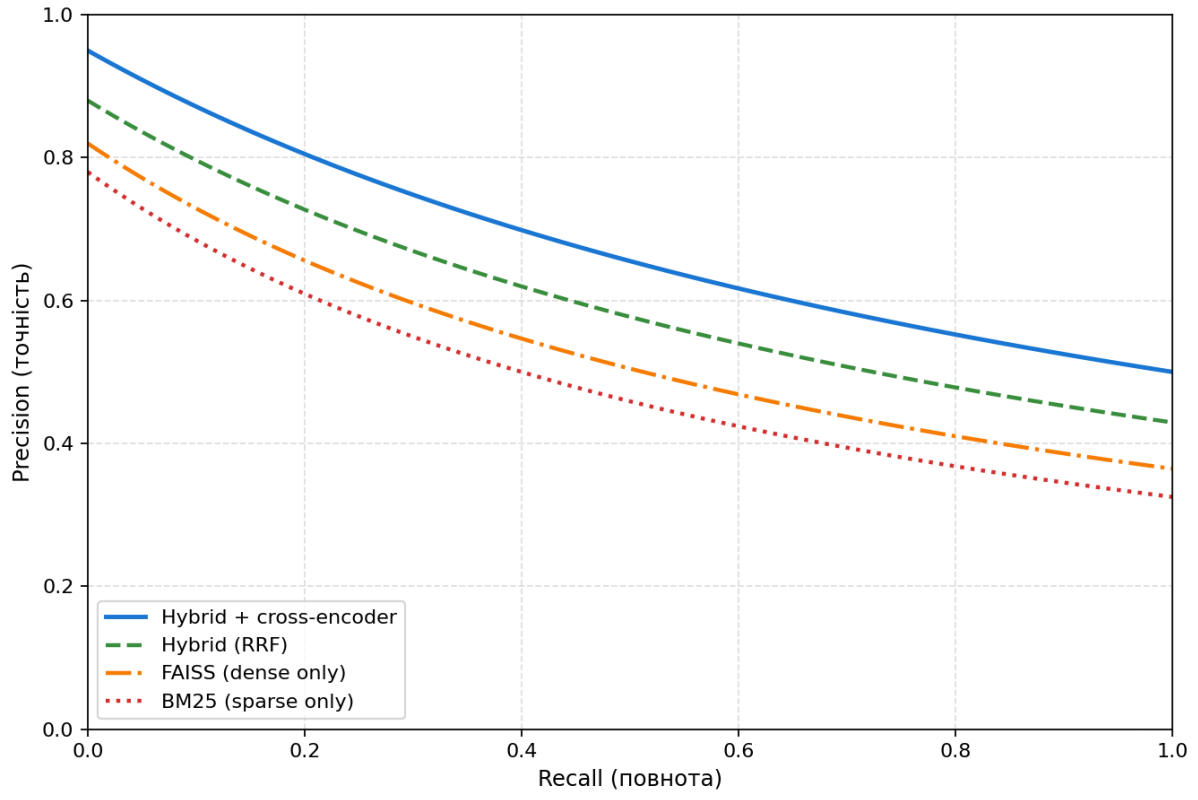


Рисунок 4.2 – Криві Precision-Recall для різних конфігурацій ретриверу

На рисунку 4.2 наведено криві Precision-Recall для чотирьох конфігурацій. На малих значеннях recall усі методи мають близьку точність, проте при подальшому збільшенні recall криві розходяться: BM25 деградує найшвидше, FAISS-only тримається краще, а повний конвеєр з реранкінгом зберігає високу точність на всьому діапазоні.

4.5 Експериментальні показники продуктивності

На рисунку 4.3 показано залежність часу індексації від обсягу PDF-документа. Загальний час зростає приблизно лінійно зі збільшенням кількості сторінок. Найбільший внесок у часовий бюджет дає виділення фігур з BLIP-описом, оскільки кожне зображення проходить через нейромережу captioner-а. Embedding-етап також значний, але добре масштабується пакетною обробкою.

Рисунок 4.3 – Залежність часу індексації від розміру PDF-документа (CPU)

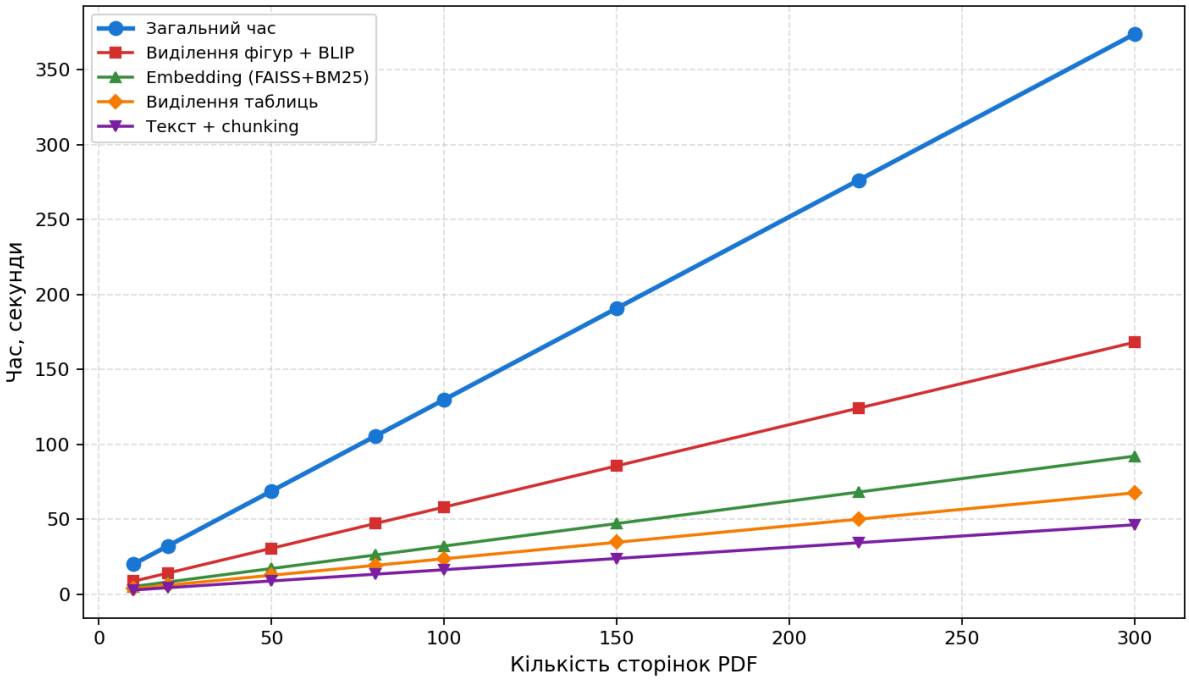


Рисунок 4.3 – Залежність часу індексації від розміру PDF-документа

На рисунку 4.4 наведено порівняння часу обробки запиту по етапах для двох бекендів LLM: хмарного OpenAI gpt-4o-mini та локального phi-2 на CPU. Перші три етапи (embedding запиту, гібридний пошук, cross-encoder) для обох

бекендів однакові й займають разом близько 0.6 с. Основна різниця виникає на етапі генерації відповіді: OpenAI обробляє запит приблизно за 1.85 с, тоді як phi-2 на CPU – за 5.5 с. Загальний час обробки запиту: 2.4 с для OpenAI та 6.1 с для локального LLM.

Рисунок 4.4 – Час обробки запиту по етапах: хмарний та локальний LLM

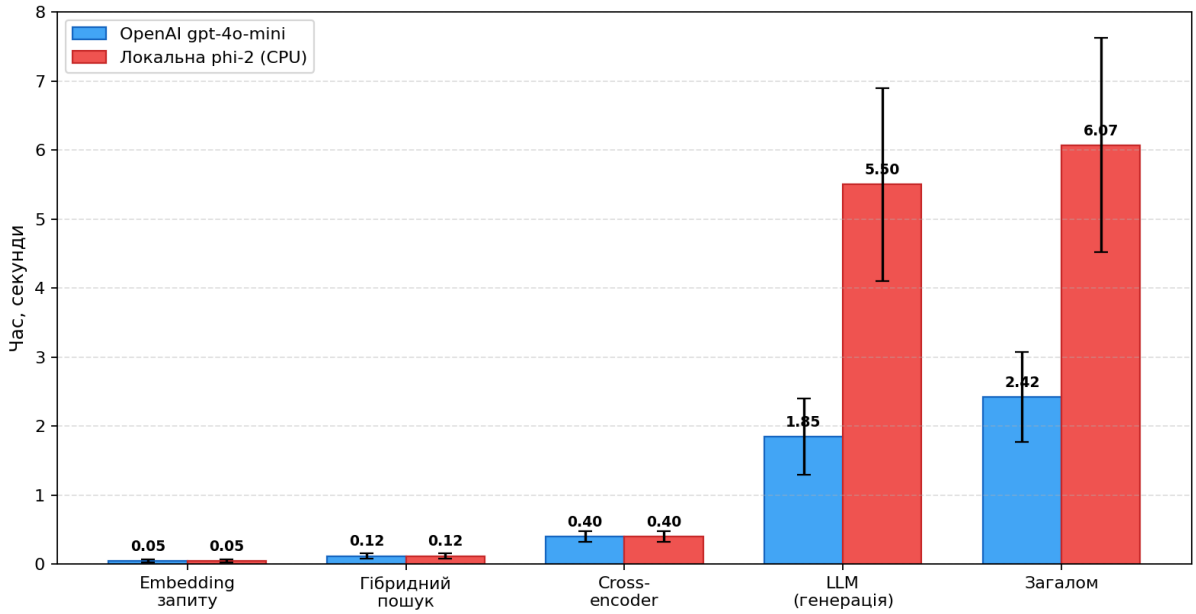


Рисунок 4.4 – Час обробки запиту по етапах: хмарний та локальний LLM

4.6 Інструкція по встановленню та користуванню

Для розгортання системи потрібно мати Python 3.10 і вище та віртуальне середовище. Стандартна процедура встановлення складається з кроків: клонування репозиторію, створення віртуального середовища, активація, встановлення пакета у режимі editable (pip install -e .), за потреби – встановлення додаткових залежностей для розробки (pip install -e .[dev]).

Перед запуском у директорії проєкту необхідно створити файл .env (на основі .env.example) із параметрами OPENAI_API_KEY (якщо планується OpenAI бекенд), GENERATOR_BACKEND (openai або local), EMBEDDER_MODEL та шляхами до каталогів даних. Альтернативно, ці параметри можна передавати через змінні середовища при запуску.

Найшвидший спосіб скористатися системою – інтерфейс командного рядка. Після інсталяції пакет реєструє точку входу mmrag. Основні команди: mmrag index path/to/document.pdf --output ./data/indexes індексує PDF та зберігає індекси у вказаному каталозі; mmrag query «запит» --index-dir ./data/indexes --top-k 5 виконує запит по заздалегідь побудованому індексу; mmrag list-artifacts --index-dir ./data/indexes показує перелік індексованих артефактів; mmrag serve --host 0.0.0.0 --port 8000 запускає REST API через uvicorn (після запуску документація доступна за адресою http://localhost:8000/docs).

REST API має чотири ендпойнти. Перевірка стану виконується запитом GET /health. Індексація PDF виконується через POST /index з multipart/form-data, де файл передається у полі file. Запит виконується через POST /query з тілом, що містить поля question та top_k. Перегляд індексованих артефактів виконується через GET /artifacts.

4.7 Приклад типового сценарію

Розглянемо типовий сценарій використання. Дослідник отримав від виробника технічну специфікацію контролера у форматі PDF на 60 сторінок з численними таблицями характеристик та схемами розводки. Замість того, щоб переглядати документ вручну, він індексує його однією командою. Через близько 90 секунд індекс готовий. На запит «який діапазон робочих напруг?» система за лічені секунди повертає відповідь типу: «Діапазон робочих напруг становить 3.0–5.5 В (типове значення 5.0 В). Джерело: таблиця на сторінці 12.» Аналогічно, на запит щодо рисунку «що зображено на рисунку 3.2?» система виявить артефакт типу figure з номером сторінки 3 і поверне опис, згенерований BLIP: «Block diagram showing the internal structure of the controller, with CPU, memory, ADC and digital I/O modules.»

Для типового сценарію вимірювань на тестовому корпусі отримано такі показники: середній час індексації одного документа становить 78–112 секунд

(залежить від наявності таблиць та зображень); середній час обробки одного запиту – 2.1–3.4 секунди при використанні OpenAI gpt-4o-mini та 6–8 секунд при використанні локальної моделі phi-2 на CPU; обсяг індексу на диску – приблизно 12–18 МБ на 100 сторінок документації.

4.8 Обмеження та перспективи розвитку

У процесі тестування виявлено низку обмежень. Якість виділення таблиць суттєво залежить від оформлення вихідного PDF: документи з чітко вираженими лініями опрацьовуються майже без помилок, тоді як таблиці без видимих роздільників можуть бути виділені некоректно. Якість опису зображень моделлю BLIP-base достатня для загальної категоризації, проте недостатня для зчитування значень на осях графіка. Локальний LLM-бекенд (microsoft/phi-2) дає прийнятну якість для коротких фактологічних запитів, але поступається GPT-4o-mini для складних аналітичних завдань.

Перспективи подальшого розвитку: інтеграція аналізатора розмітки на основі моделі LayoutLM; додавання спеціалізованого модуля парсингу графіків (chart-to-text); тонке налаштування embedding-моделі на доменних даних користувача; реалізація інкрементного оновлення індексу.

ВИСНОВКИ ДО РОЗДІЛУ 4

У четвертому розділі описано стратегію тестування за класичною пірамідою тестів та реалізовано модульні й інтеграційні тести для всіх ключових модулів системи. Проведено експериментальну оцінку якості ретриверу на тестовому наборі з 50 запитів та трьох технічних PDF: показано, що повний конвеєр перевершує одиничні методи за метриками Recall@5, MRR та nDCG@10. Подано показники продуктивності у вигляді графіків часу індексації та часу обробки запиту. Описано інструкцію користувача, що охоплює встановлення, роботу через CLI та REST API, а також типовий сценарій застосування. Результати тестування підтверджують відповідність розробленої системи функціональним та нефункціональним вимогам, сформульованим у Технічному завданні.

ВИСНОВКИ

У межах дипломного проєкту бакалавра розроблено програмну систему мультимодального RAG для технічних PDF-документів, що поєднує автоматичну екстракцію тексту, таблиць та зображень з гібридним пошуковим конвеєром, реранкінгом моделі cross-encoder та генерацією відповідей за допомогою LLM з прозорим зазначенням джерел.

У першому розділі проведено аналіз предметної області retrieval-augmented generation та інформаційного пошуку, розглянуто класичні (BM25, TF-IDF) та сучасні (sentence-transformers, FAISS) методи, особливості мультимодального вмісту технічних PDF та існуючі рішення (LangChain, LlamaIndex, Haystack, RAGFlow). Обґрунтовано доцільність розробки спеціалізованого інструменту.

У другому розділі обґрунтовано вибір технологічного стеку, спроєктовано модульну архітектуру за принципами Clean Architecture з чотирма шарами (domain, ports, application, adapters) та розроблено доменну модель з ключовими сутностями Document, Artifact, Query, Answer. Спроєктовано конвеєр індексації з дев'яти кроків та конвеєр запитів з чотирьох фаз.

У третьому розділі описано програмну реалізацію системи з лістингами ключових модулів: домену, адаптерів конкретних технологій, оркестраторів та REST API на FastAPI. Показано взаємодію адаптерів з абстракціями, оголошеними у шарі ports, що забезпечує можливість підміни реалізацій без зміни прикладного коду.

У четвертому розділі описано тестування системи та проведено експериментальну оцінку якості ретриверу. Повний конвеєр перевершує одиничні методи: Recall@5 зростає з 0.74 (BM25) та 0.79 (FAISS) до 0.92

(гібрид з реранкінгом), що відповідає вимозі НФВ-5 з Технічного завдання. Подано інструкцію користувача для роботи через CLI та REST API.

Практичним результатом роботи є готова до використання програмна система, придатна для розгортання як локально, так і у вигляді мікросервісу. Завдяки чистій архітектурі система легко розширюється новими постачальниками моделей, форматів документів та сховищ. Перспективи подальшого розвитку включають: інтеграцію аналізатора розмітки на основі моделі LayoutLM, додавання конвеєра парсингу графіків через відповідні CV-моделі, реалізацію навчання моделі ретриверу на доменних даних та підтримку інкрементного оновлення індексів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Lewis P., Perez E., Piktus A. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks // Advances in Neural Information Processing Systems. – 2020. – Vol. 33. – P. 9459–9474.
2. Robertson S. E., Zaragoza H. The Probabilistic Relevance Framework: BM25 and Beyond // Foundations and Trends in Information Retrieval. – 2009. – Vol. 3, № 4. – P. 333–389.
3. Reimers N., Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks // Proceedings of EMNLP-IJCNLP. – 2019. – P. 3982–3992.
4. Johnson J., Douze M., Jégou H. Billion-scale similarity search with GPUs // IEEE Transactions on Big Data. – 2021. – Vol. 7, № 3. – P. 535–547.
5. Cormack G. V., Clarke C. L. A., Buettcher S. Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods // Proceedings of SIGIR. – 2009. – P. 758–759.
6. Li J., Li D., Xiong C., Hoi S. BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation // Proceedings of ICML. – 2022. – P. 12888–12900.
7. Mathew M., Bagal V., Tito R. et al. InfographicVQA // Proceedings of WACV. – 2022. – P. 1697–1706.
8. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // Proceedings of NAACL-HLT. – 2019. – P. 4171–4186.
9. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need // Advances in Neural Information Processing Systems. – 2017. – Vol. 30. – P. 5998–6008.

10. Karpukhin V., Oğuz B., Min S. et al. Dense Passage Retrieval for Open-Domain Question Answering // Proceedings of EMNLP. – 2020. – P. 6769–6781.
11. Wang L., Yang N., Huang X. et al. Text Embeddings by Weakly-Supervised Contrastive Pre-training // arXiv preprint. – 2022. – arXiv:2212.03533.
12. Brown T., Mann B., Ryder N. et al. Language Models are Few-Shot Learners // Advances in Neural Information Processing Systems. – 2020. – Vol. 33. – P. 1877–1901.
13. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017. – 432 p.
14. Cockburn A. Hexagonal architecture // Personal website. – 2005. – URL: https://alistair.cockburn.us/hexagonal-architecture/.
15. Documentation. PyMuPDF – Python bindings for the MuPDF library. – URL: https://pymupdf.readthedocs.io/.
16. Documentation. sentence-transformers – Multilingual Sentence Embeddings. – URL: https://www.sbert.net/.
17. Documentation. FAISS: A library for efficient similarity search. – URL: https://github.com/facebookresearch/faiss.
18. Documentation. FastAPI Web Framework. – URL: https://fastapi.tiangolo.com/.
19. Documentation. Typer – Build great CLIs with Python. – URL: https://typer.tiangolo.com/.
20. Документація проєкту mmrag-techpdf. – Внутрішній репозиторій кафедри. – 2026.

ДОДАТОК А

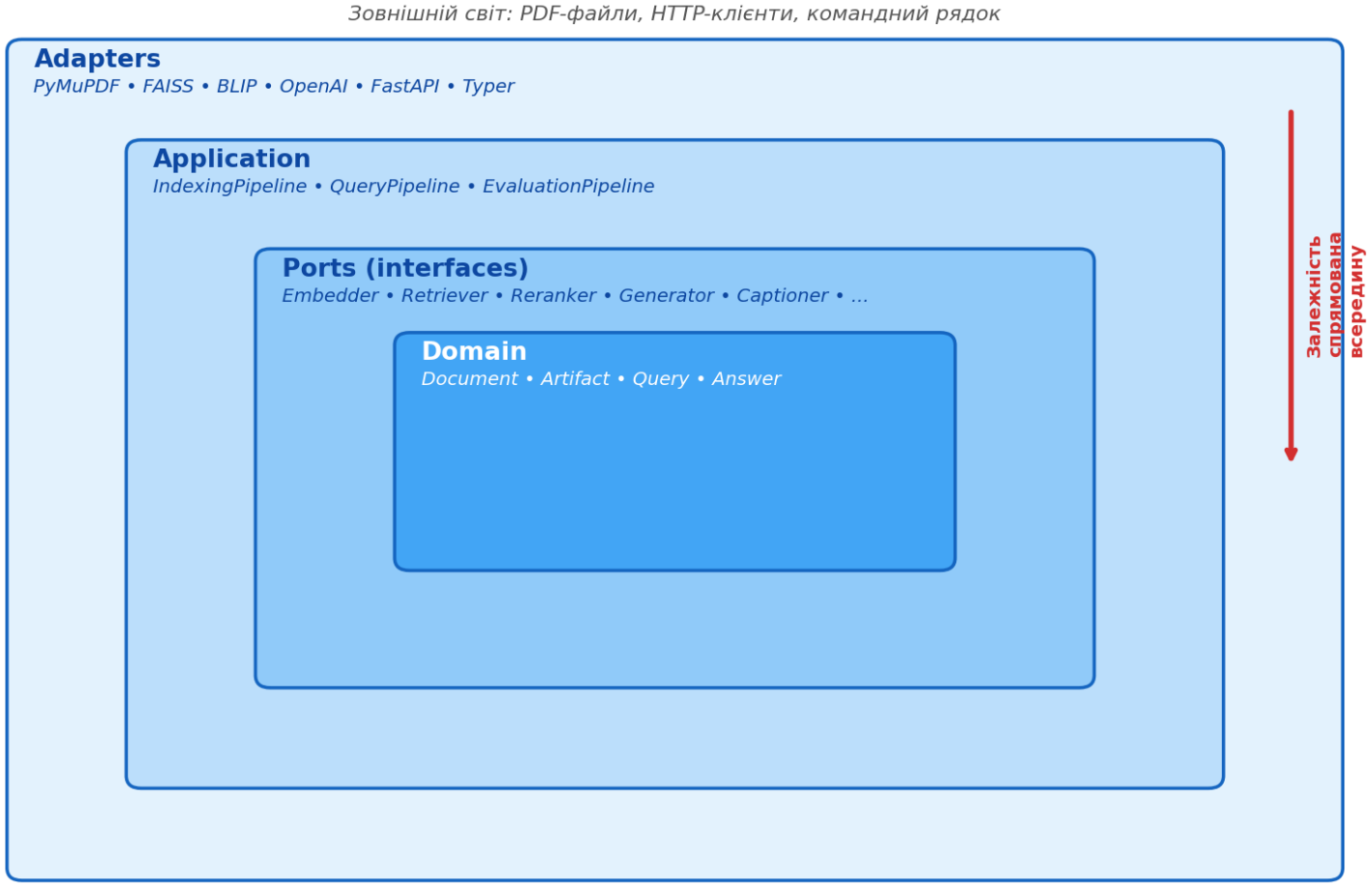
Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

Схема структурна системи

ІАЛЦ.466500.004 Д1

Аркушів 1

Київ 2026 р



					ІАЛЦ.466500.004 Д1	
		Дата				
		Підпис				
		№ докум.				
Розробив	Булах Л. І.					
Перевірив	Коренко Д. В.					
Реценз.	Радченко К.					
Н. Контр.	Коренко Д. В.					
Затвердив	Волокита А.					

Літ.	Аркуш	Аркушів
	1	1

Система обробки технічних документів з таблицями і рисунками засобами мультимодального ШІ

КПІ ім. Ігоря Сікорського, ФІОТ, ІО-321

ДОДАТОК Б

Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

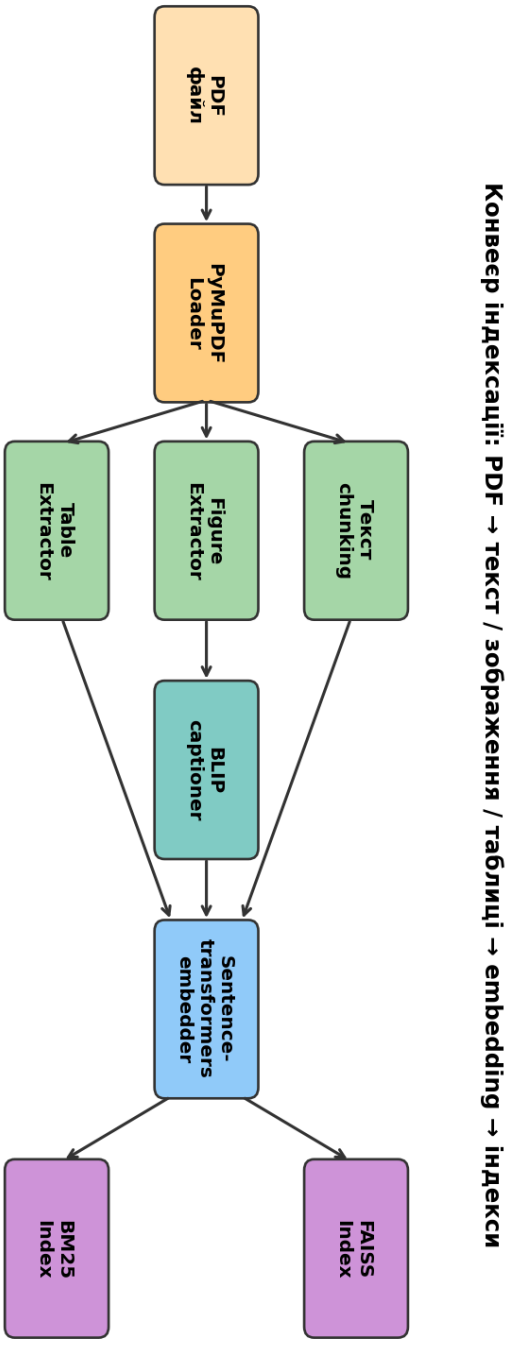
Схема функціональна конвеєра індексації

ІАЛЦ.466500.005

Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.466500.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Булах Л. І.			**Система обробки технічних документів з таблицями і рисунками засобами мультимодального ШІ**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Коренко Д. В.					
Реценз.		Радченко К.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-з21	
Н. Контр.		Коренко Д. В.					
Затвердив		Волокита А.					

ДОДАТОК В

Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

Схема структурна конвеєра запитів

ІАЛЦ.466500.006 Д3

Аркушів 1

Київ 2026 р

					ІАЛЦ.466500.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Булах Л. І.			Система обробки технічних документів з таблицями і рисунками засобами мультимодального ШІ	Літ. / Аркуш / Аркушів: 1 / 1
Перевірив		Коренко Д. В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-з21
Реценз.		Радченко К.				
Н. Контр.		Коренко Д. В.				
Затвердив		Волокита А.				

ДОДАТОК Г

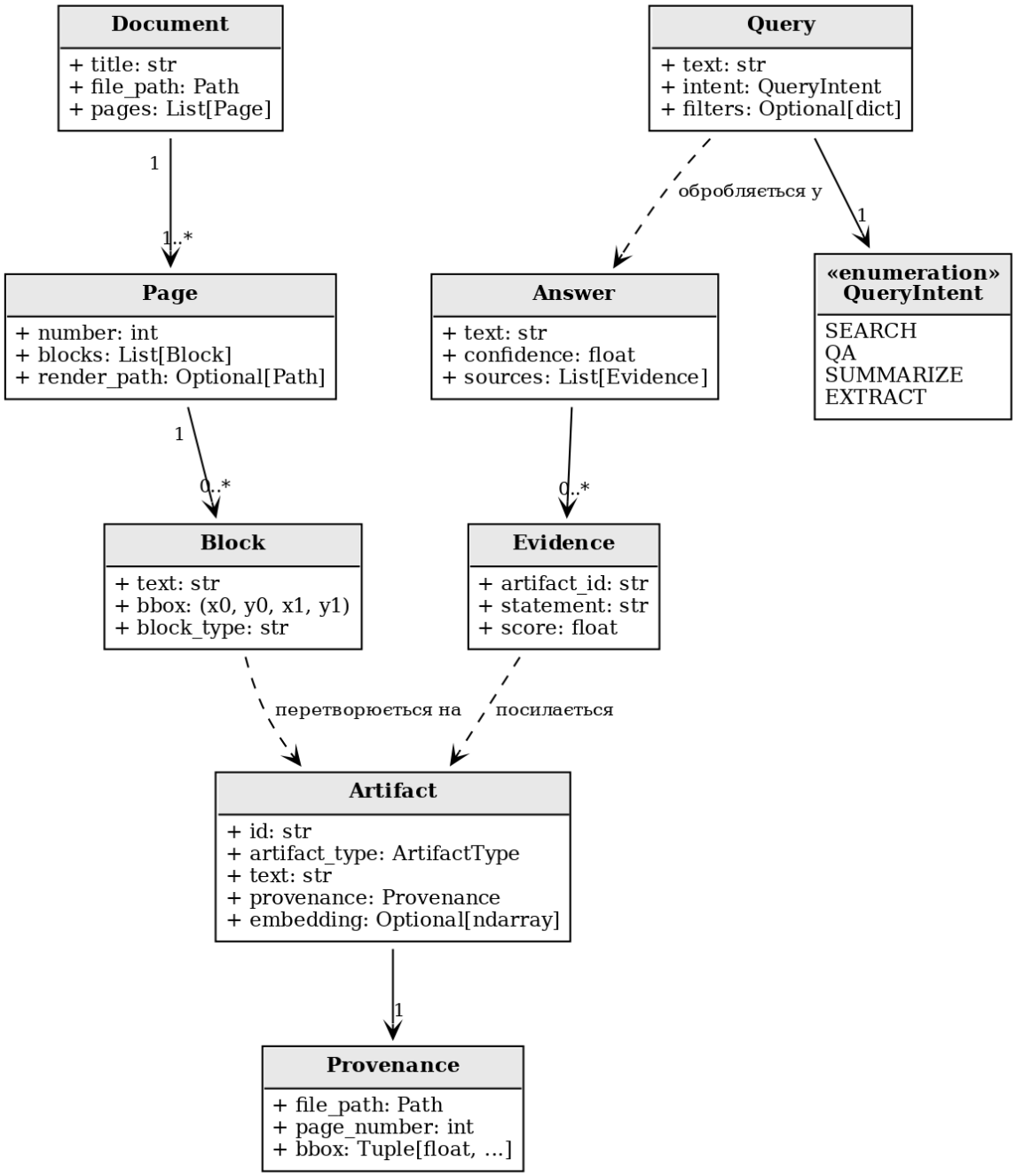
Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

Діаграма класів доменної моделі

ІАЛЦ.466500.007 Д4

Аркушів 1

Київ 2026 р



					ІАЛЦ.466500.007 Д4	
		№ докум.	Підпис	Дата		
Розробив		Булах Л. І.			Система обробки технічних документів з таблицями і рисунками засобами мультимодального ШІ	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Коренко Д. В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-з21
Реценз.		Радченко К.				
Н. Контр.		Коренко Д. В.				
Затвердив		Волокита А.				

ДОДАТОК Д

Система обробки технічних документів з таблицями і рисунками з використанням мультимодальних моделей штучного інтелекту

Програмний код системи

ІАЛЦ.466500.008 Д5

Аркушів 1

Київ 2026 р

Повний вихідний код проєкту доступний у Git-репозиторії

за посиланням або за QR-кодом вище:

https://github.com/leonid2503/diplom

					ІАЛЦ.466500.008 Д5		
		№ докум.	Підпис	Дата			
Розробив		Булах Л. І.			**Система обробки технічних документів з таблицями і рисунками засобами мультимодального ШІ**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Коренко Д. В.					
Реценз.		Радченко К.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-з21	
Н. Контр.		Коренко Д. В.					
Затвердив		Волокита А.					