НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри

_____ Наталія Аушева

Дипломна робота

на здобуття ступеня бакалавра

За освітньою програмою «Цифрові технології в енергетиці»
Спеціальності 122 «Комп'ютерні науки»
на тему: «Система розпізнавання та структуризації усного мовлення»

Виконав: студент 4 курсу, групи ТР-22

Лакін Дмитро Андрійович _____
(прізвище ім'я по батькові) (підпис)

Керівник асистент каф. цифрових технологій в енергетиці,
Владислав ТІТОВ _____
(посада, науковий ступінь, вчене звання, ім'я ПРІЗВИЩЕ) (підпис)

Рецензент професор каф. теплової та альтернативної
енергетики, д.т.н., Ольга ЧЕРНОУСЕНКО _____
(посада, науковий ступінь, вчене звання, ім'я ПРІЗВИЩЕ) (підпис)

Н.Контроль доцент каф. цифрових технологій в енергетиці,
д.ф. Артем ГУРІН _____
(посада, науковий ступінь, вчене звання, ім'я ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

КИЇВ – 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти — перший (бакалаврський)
спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
"___" _____ 2026 р.

ЗАВДАННЯ
на дипломну роботу студенту

Лакіну Дмитру Андрійовичу
(прізвище ім'я по батькові)

1. Тема роботи «Система розпізнавання та структуризації усного мовлення»
Науковий керівник Тітов Владислав Миколайович
(прізвище ім'я по батькові)

затверджені наказом по університету від "28" травня 2026 року № 1992-с
2. Термін подання студентом роботи: 08.06.2026
3. Вихідні дані до роботи: мова програмування Python; бібліотеки Faster-Whisper, Transformers, Sentence-Transformers, SpaCy, BM25S, YAKE!, RapidFuzz; середовище розробки VS Code.
4. Перелік питань, які потрібно розробити:
   1) провести аналіз серед наявних аналогів;
   2) визначити головні функції застосунку, що виконує структуризацію тексту;
   3) аргументувати вибрані інструменти, алгоритми та технології для реалізації локальної системи;
   4) побудувати архітектуру програмного забезпечення та роботи алгоритмів;
   5) створити прикладний програмний інтерфейс;
   6) представити процес застосування інтерфейсу.

5. Орієнтований перелік ілюстративного матеріалу: зображення інтерфейсів альтернативних рішень; схематики та діаграми модулів та алгоритмів системи; приклади використання програмного забезпечення.

6. Дата видачі завдання 01.12.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	01.10.2025	
2.	Аналіз методів та засобів розв’язання задачі	20.04.26 – 26.04.26	
3.	Розробка архітектури та загальної структури системи	27.04.26 – 03.05.26	
4.	Розробка окремих підсистем	04.05.26 – 10.05.26	
5.	Програмна реалізація системи	11.05.26 – 17.05.26	
6.	Оформлення пояснювальної записки	14.05.26 – 16.05.26	
7.	Захист програмного забезпечення	14.05.2026	
8.	Передзахист	04.06.2026	
9.	Захист	17.06.2026	

Студент _____ Лакін Д. А.
(підпис) (прізвище та ініціали)

Керівник _____ Тітов В. М.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 61 сторінці, містить 33 розділи, має 16 рисунків та 55 використаних джерел.

Мета роботи: розробка програмної системи для автоматичного розпізнавання та структуризації усного мовлення, з першочерговим використанням на локальному пристрої.

Методи та засоби: мова програмування Python; бібліотеки Faster-Whisper, Transformers, Sentence-Transformers, SpaCy, BM25S, YAKE!, RapidFuzz; PySide6 для побудови графічного інтерфейсу.

Основний зміст роботи: технічна постановка задачі, формалізація вимог до програмної системи; аналіз наявних рішень; огляд технічних рішень впровадження ASR, NLP, Word Embeddings, SLM, Hybrid Search у прикладну задачу; інструкція з встановлення та використання проекту.

Результат: локальна система для транскрипції та структуризація медіаконтенту у текстовому представленні.

Ключові слова: ASR, NLP, Word Embeddings, Semantic Search, Hybrid Search, Bi-encoder, Cross-encoder, RAG, Reranker, BM25, RRF, YAKE!, Fuzzy Mathing, LLM/SLM, Transformer Model, CUDA, MoE.

ABSTRACT

The thesis is completed on 61 pages, contains 33 sections, has 16 figures and 55 sources used.

Goal: development of a software system for automatic recognition and structuring of oral speech, with primary use on a local device.

Methods and tools: Python programming language; Faster-Whisper, Transformers, Sentence-Transformers, SpaCy, BM25S, YAKE!, RapidFuzz libraries; PySide6 for building a graphical interface.

The main content of the paper: technical formulation of the problem, formalization of requirements for the software system; analysis of existing solutions; review of technical solutions for implementing ASR, NLP, Word Embeddings, SLM, Hybrid Search in an applied problem; instructions for installing and using the project.

Result: a local system for transcription and structuring of media content in text representation.

Keywords: ASR, NLP, Word Embeddings, Semantic Search, Hybrid Search, Bi-encoder, Cross-encoder, RAG, Reranker, BM25, RRF, YAKE!, Fuzzy Mathing, LLM/SLM, Transformer Model, CUDA, MoE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ASR (Automatic Speech Recognition) — технологія автоматичного розпізнавання мовлення, яка перетворює аудіосигнал на текст. Сучасні ASR-системи використовують глибокі нейронні мережі (наприклад, Whisper від OpenAI) для транскрибування усного мовлення з високою точністю, враховуючи акценти, шум та інші перешкоди.

NLP (Natural Language Processing) — галузь штучного інтелекту, що займається обробкою та розумінням природної людської мови. NLP охоплює широкий спектр задач: класифікацію тексту, машинний переклад, аналіз тональності, вилучення сутностей тощо — і є основою для більшості сучасних мовних застосунків.

Word Embeddings — спосіб представлення слів у вигляді щільних числових векторів у багатовимірному просторі, де семантично близькі слова розташовані поруч. Класичні моделі (Word2Vec, GloVe) навчаються на великих корпусах тексту, фіксуючи контекстуальні зв’язки між словами та дозволяючи моделям «розуміти» значення.

Semantic Search — пошук, що спирається не на точний збіг ключових слів, а на змістову близькість запиту та документів. Запит і тексти кодуються у векторному просторі, після чого знаходяться найближчі за косинусною відстанню результати, що дозволяє знаходити релевантні відповіді навіть при відмінному формулюванні.

Hybrid Search — підхід, що поєднує лексичний (keyword-based) пошук із семантичним векторним пошуком для досягнення кращої якості результатів. Зазвичай результати обох методів об’єднуються за допомогою алгоритмів злиття (наприклад, RRF), компенсуючи слабкі місця кожного з підходів окремо.

Bi-encoder — архітектура, де запит і документ кодуються двома окремими (але схожими) енкодерами незалежно один від одного у вектори фіксованої розмірності. Це дозволяє заздалегідь обчислити та зберегти вектори документів,

забезпечуючи надшвидкий пошук у великих колекціях, хоча й з дещо нижчою точністю порівняно з cross-encoder.

Cross-encoder — модель, яка отримує на вхід одночасно запит і документ, аналізуючи їхню взаємодію на рівні self-attention для точнішої оцінки релевантності. На відміну від bi-encoder, cross-encoder не дозволяє попередньо кешувати вектори, тому застосовується переважно для переранжування невеликого набору кандидатів, а не для первинного пошуку.

RAG (Retrieval-Augmented Generation) — архітектура, що поєднує пошук релевантних документів із генерацією відповіді мовною моделлю. Замість того щоб покладатися виключно на знання, «зашиті» в параметри LLM, RAG динамічно підтягує актуальну інформацію з зовнішньої бази знань і передає її у контекст моделі, знижуючи галюцинації та підвищуючи точність.

Reranker — компонент пошукового або RAG-пайплайну, що отримує попередньо відібраний список кандидатів і перевпорядковує їх за більш точним критерієм релевантності. Як правило, реалізується на базі cross-encoder і слугує «другим фільтром» після швидкого первинного відбору, суттєво підвищуючи якість фінальних результатів.

BM25 (Best Match 25) — класичний статистичний алгоритм ранжування документів на основі частоти термінів (TF-IDF-подібна логіка) з нормалізацією за довжиною документа. Незважаючи на відносну простоту, BM25 залишається надзвичайно ефективним лексичним baseline-методом і широко використовується у гібридних пошукових системах.

RRF (Reciprocal Rank Fusion) — алгоритм злиття результатів із кількох ранжованих списків, де кожному документу присвоюється оцінка, обернено пропорційна його позиції в кожному списку. RRF не потребує нормалізації різнорідних скорів і на практиці показує стабільно високу якість при об’єднанні лексичного та семантичного пошуку.

YAKE! (Yet Another Keyword Extractor) — легковагий статистичний алгоритм вилучення ключових слів із тексту без потреби у навчальних даних чи зовнішніх ресурсах. YAKE! спирається на статистичні ознаки слів (позиція в

тексті, частота, розсіювання) і добре працює для швидкої індексації та тегування документів у реальному часі.

Fuzzy Matching — техніка нечіткого порівняння рядків, що знаходить приблизні збіги на основі метрик редакційної відстані (наприклад, Levenshtein). Застосовується для виправлення друкарських помилок, нормалізації варіантів написання та пошуку в умовах неточних або зашумлених запитів.

LLM / SLM (Large / Small Language Model) — великі та малі мовні моделі, навчені на масивних текстових корпусах для генерації, розуміння та трансформації тексту. LLM (GPT-4, Claude, Gemini) вирізняються широкими можливостями за рахунок мільярдів параметрів, тоді як SLM (Phi, Mistral 7B) оптимізовані для ефективної роботи на обмежених ресурсах — на пристрої або у латентно-чутливих сценаріях.

Transformer Model — революційна архітектура нейронних мереж, запропонована у статті «Attention is All You Need» (2017), що базується на механізмі самоуваги (self-attention) замість рекурентних зв’язків. Трансформери паралельно обробляють усі токени послідовності й стали основою для переважної більшості сучасних NLP-моделей — від BERT до GPT і далі.

CUDA (Compute Unified Device Architecture) — паралельна обчислювальна платформа та API від NVIDIA, що дозволяє використовувати GPU для задач загального призначення, зокрема для тренування та інференсу нейронних мереж. Завдяки тисячам паралельних ядер CUDA значно прискорює матричні операції, що є основою глибокого навчання.

MoE (Mixture of Experts) — архітектурний підхід, при якому модель складається з набору спеціалізованих підмереж («експертів»), а маршрутизатор (gating network) для кожного токена активує лише частину з них. Це дозволяє масштабувати загальну кількість параметрів моделі (і тим самим її здатності) без пропорційного зростання обчислювальних витрат під час інференсу — як у моделях Mixtral чи GPT-4.

ВСТУП

Стрімкий розвиток інформаційних технологій і хмарних обчислювальних платформ зумовив суттєву трансформацію форм подання цифрового контенту. Дедалі більша частка інформації виробляється та поширюється у відео- та аудіоформатах, призначених для наочного перегляду або фонового прослуховування. Разом з тим, попри очевидні переваги мультимедійного формату з точки зору сприйняття, він створює низку практичних обмежень у ситуаціях, що вимагають детального аналізу, систематичного пошуку або точного цитування інформації. У таких випадках текстове представлення змісту медіаматеріалів є значно зручнішим та ефективнішим засобом опрацювання — воно дозволяє здійснювати повнотекстовий пошук, структурований аналіз і автоматизовану обробку за допомогою сучасних інструментів.

Системи автоматичного перетворення медіаформатів у текст пройшли значний шлях розвитку, який умовно можна розподілити на два ключові етапи — «до» та «після» широкого впровадження сучасних моделей машинного навчання. Системи першого покоління спираються на відносно прості архітектурні рішення та детерміновані алгоритми обробки мовлення, що суттєво обмежує якість вихідного тексту, зокрема в умовах зашумленого акустичного середовища, при наявності акцентів або варіативного темпу мовлення. Системи другого покоління, засновані на глибоких нейронних мережах, демонструють надзвичайно динамічний темп розвитку [1] та планомірно впроваджують найновіші досягнення у сфері обробки природної мови й акустичного моделювання.

Водночас ситуація, що склалася внаслідок прискорення технологічної конкуренції у сфері штучного інтелекту, породила певну структурну неоднорідність ринку програмного забезпечення. Більшість рішень, що функціонують локально на пристрої користувача, продовжують спиратися на застарілі механізми обробки, тоді як переважна частина новітніх розробок, адаптованих до сучасних реалій, орієнтована на модель надання послуг (Software-as-a-Service) і не передбачає можливості розгортання на власній апаратній

інфраструктурі. При цьому обчислювальні потужності споживчих персональних пристроїв невпинно зростають протягом останніх років. Сучасні конфігурації кінцевих пристроїв вже є достатніми не лише для виконання інференсу нейромережевих моделей, а й для їх навчання, що відкриває перед розробниками програмного забезпечення можливість реалізовувати складніші алгоритми безпосередньо у призначених для користувача застосунках без залучення зовнішніх хмарних ресурсів.

Метою даної роботи є розробка системи, що реалізує сучасний підхід до обробки медіаконтенту на пристрої кінцевого користувача. Система охоплює повний цикл опрацювання аудіоматеріалів: автоматичне перетворення мовлення у текст, виявлення семантичних меж і структурована сегментація контенту, узагальнення змісту за допомогою малих мовних моделей (SLM), а також семантичний пошук з підтримкою природномовних запитів. Архітектурно система складається з таких взаємопов’язаних модулів: транскрипції, сегментації, інтерфейсу взаємодії з мовними моделями, оркестрації процесів та підсистеми індексації й пошуку. Принциповою відмінністю запропонованого рішення від наявних хмарних альтернатив є повна локальна обробка даних, що забезпечує конфіденційність користувацького контенту та незалежність від мережевого підключення.

Описане програмне забезпечення орієнтоване на широке коло користувачів, чия професійна або дослідницька діяльність пов’язана з систематичним опрацюванням значних обсягів медіаконтенту — зокрема журналістів, науковців, аналітиків і студентів, — та які потребують зручного та ефективного інструментарію для його структурованого аналізу й пошуку.

1 ЗАДАЧІ ПОБУДОВИ ТА АНАЛІЗ НАЯВНИХ РІШЕНЬ

Стрімкий розвиток методів обробки природної мови та нейромережевих архітектур відкриває нові можливості для автоматизації роботи з усним мовленням. Незважаючи на значний прогрес у галузі автоматичного розпізнавання мовлення, задача перетворення неструктурованого аудіоконтенту на семантично організований текст залишається актуальною та недостатньо вирішеною з практичної точки зору. У цьому контексті виникає потреба у розробці інтегрованої системи, що поєднує розпізнавання мовлення, структуризацію тексту та інтелектуальний пошук в єдиному пайплайні, орієнтованому на реальні сценарії використання.

Перш ніж формулювати конкретні вимоги до розроблюваної системи та окреслювати підходи до їх реалізації, доцільно проаналізувати наявний ландшафт існуючих рішень. Такий аналіз дозволяє виявити незаповнені ніші та обґрунтувати архітектурні рішення, прийняті в межах цієї роботи. Як показує огляд, представлений у підрозділі 1.2, наявні інструменти або охоплюють лише окремі складові необхідного функціоналу, або реалізують його в повному обсязі, але виключно у хмарному середовищі, що суперечить вимогам конфіденційності та автономності. Саме ця прогалина визначає практичну мотивацію розробки та задає систему вимог, викладену в підрозділі 1.1.

1.1 Постановка прикладної задачі

Метою розробки системи розпізнавання та структуризації усного мовлення є автоматизоване отримання текстової інформації у структурованому вигляді, що забезпечує полегшене сприйняття контенту, а також реалізація функціоналу підведення підсумків і семантичного пошуку для ефективного аналізу великих масивів інформації.

Для досягнення високої точності розпізнавання мовлення система має спиратися на сучасні нейромережеві моделі, побудовані на архітектурі Transformers [2] або її похідних — GPT та BERT [3, 4, 5]. Ключовою особливістю зазначених архітектур є механізм само-уваги (self-attention), який здійснює динамічну переоцінку вагових коефіцієнтів залежно від контексту, що суттєво підвищує якість розуміння мовленнєвих послідовностей порівняно з рекурентними підходами. Апаратне прискорення на базі GPU з використанням інтерфейсу CUDA дозволяє досягти значного скорочення часу інференції, що є критично важливим для обробки тривалих аудіозаписів у режимі, наближеному до реального часу. Додатковою перевагою трансформерних архітектур є вбудована здатність до відновлення пунктуації [6], що позитивно впливає на читабельність вихідного тексту та якість його подальшої обробки. Результатом етапу розпізнавання є набір пар «текст — часова мітка», що формує основу для подальшої структуризації.

З метою підвищення зручності сприйняття отриманого тексту необхідно здійснити його розподіл на логічні структурні одиниці: параграфи та заголовки. Це формулює задачу автоматичного виявлення семантичних меж у неструктурованому тексті. Для її розв’язання пропонується використання речення як базової семантичної одиниці, що дозволяє досягти збалансованої гранулярності при подальшому сегментуванні: на відміну від окремих токенів, речення зберігають достатній семантичний контекст, а на відміну від абзаців — не вносять надмірного узагальнення. Отримані текстові фрагменти перетворюються на щільні векторні представлення (ембедингі) за допомогою моделей сенс-кодування, що дозволяє виконувати над ними математичні операції у векторному просторі. Різкий спад косинусної подібності між ембедингами суміжних речень слугує сигналом семантичного переходу — зміни теми або зсуву думки, — що й визначає межі між структурними блоками тексту.

Виявлені семантичні розриви визначають межі між тематичними блоками, однак для змістовної інтерпретації кожного блоку необхідно виокремити ключову інформацію та сформувати відповідний заголовок. Традиційні підходи до вилучення ключових слів, зокрема TF-IDF та BM25 [7], ґрунтуються на

статистичному аналізі частотності термінів і не враховують семантичних зв’язків між ними. Сучасніші альтернативи — KeyBERT, YAKE та RAKE [8, 9, 10] — дещо розширюють можливості екстракції, проте також зводяться до виведення окремих слів або словосполучень, що є недостатнім для автоматичного формування граматично зв’язних назв секцій. Для розв’язання цієї задачі доцільнішим є застосування малої мовної моделі (Small Language Model, SLM): попри вищу обчислювальну вартість порівняно зі статистичними методами, SLM здатна до абстрактного розуміння тексту та генерації природних формулювань. Крім того, та сама модель може використовуватися для автоматичного генерування коротких підсумків окремих секцій, що робить її застосування функціонально виправданим.

Реалізація пошуку по отриманому тексту можлива у двох принципово різних варіантах: канонічне зіставлення за ключовими словами або семантичний пошук на основі векторних представлень. Перший підхід відзначається високою швидкодією та підтримкою часткового збігу рядків, однак позбавлений здатності до змістовної фільтрації результатів у випадках, коли пошуковий термін часто зустрічається в різних контекстах. Семантичний пошук, натомість, дозволяє формулювати запити природною мовою та отримувати контекстуально релевантні фрагменти тексту, проте схильний до хибнопозитивних результатів у ситуаціях семантичної близькості без тематичної відповідності. Оскільки обидва підходи ґрунтуються на принципово різних механізмах індексації та пошуку, повноцінна паралельна реалізація обох у межах єдиного пайплайну вимагає суттєвих архітектурних компромісів.

Основним сценарієм використання системи є обробка медіаконтенту в умовах редакційного або журналістського робочого процесу. Користувач завантажує аудіо- або відеоматеріал і отримує структуровану текстову транскрипцію, готову до редагування, а також інструменти для пошуку конкретних фрагментів за допомогою запитів природною мовою. Такий підхід суттєво скорочує час, необхідний для первинного опрацювання матеріалу, і знижує когнітивне навантаження на фахівця.

Запропонована система орієнтована на розгортання на споживчому обладнанні без необхідності хмарної інфраструктури. За наявності GPU із

підтримкою CUDA забезпечується висока швидкодія всього пайплайну; при використанні виключно CPU час обробки зростає, проте функціональність системи зберігається в повному обсязі, що робить її доступною для широкого кола користувачів.

1.2 Аналіз наявних рішень

Для систематизації огляду існуючих програмних рішень, що реалізують функціонал, споріднений із запропонованою системою, доцільно виокремити дві узагальнені категорії: «відповідні» — системи, що переважно реалізують увесь визначений функціонал, та «релевантні» — системи, що реалізують його частково або в обмеженому обсязі.

1.2.1 Відповідні системи

Системи першого типу здебільшого є хмарними сервісами, які, попри спільну спрямованість, суттєво різняться за цільовою аудиторією та прикладними задачами. Характерними представниками цієї категорії є Otter.ai та Fireflies.ai — інтелектуальні асистенти для автоматизованої обробки нарад на базі штучного інтелекту. Обидва сервіси виконують транскрипцію аудіоматеріалу, його структуризацію із залученням часових міток, а також формування підсумкових зведень. Разом із тим вони орієнтовані на дещо відмінні сценарії застосування.

Otter.ai переважно призначений для обробки онлайн-нарад, лекцій, інтерв’ю та інших аудіозаписів із залученням кількох мовців. За функціональним наповненням це рішення є найближчим аналогом до розроблюваної системи та найбільше підходить для персонального використання. Приклад робочого інтерфейсу сервісу наведено на рис. 1.1.

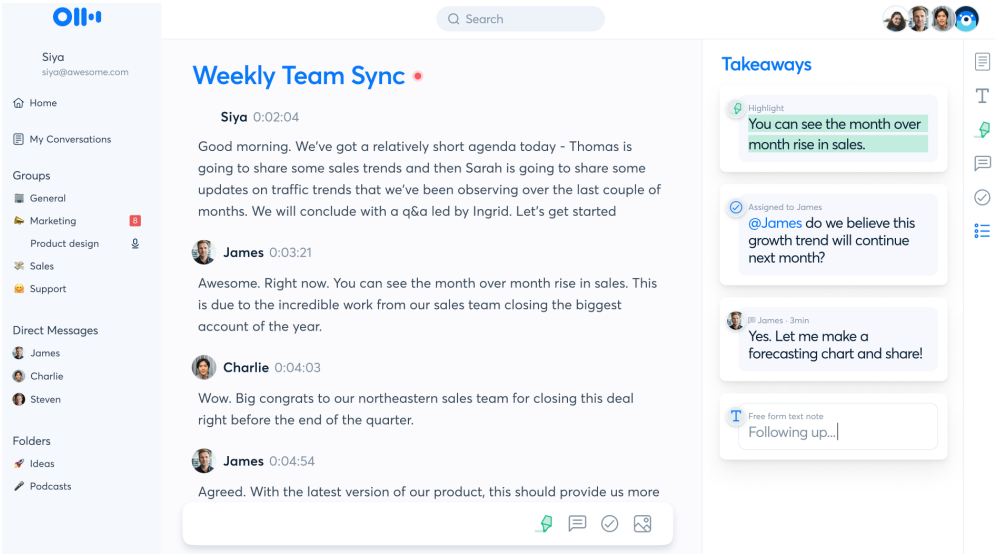

Рисунок 1.1 — Інтерфейс Otter.ai

Fireflies.ai, порівняно з попереднім, пропонує розширений статистичний інструментарій і позиціонується як корпоративний інструмент для відстеження результатів роботи відділів продажів, управлінської аналітики та координації командної діяльності. Попри те що цей сервіс технічно може бути застосований до поставлених задач, його функціональний акцент зміщений у бік корпоративного сегменту, що знижує його доцільність у контексті цієї роботи. Приклад робочого інтерфейсу системи наведено на рис. 1.2.

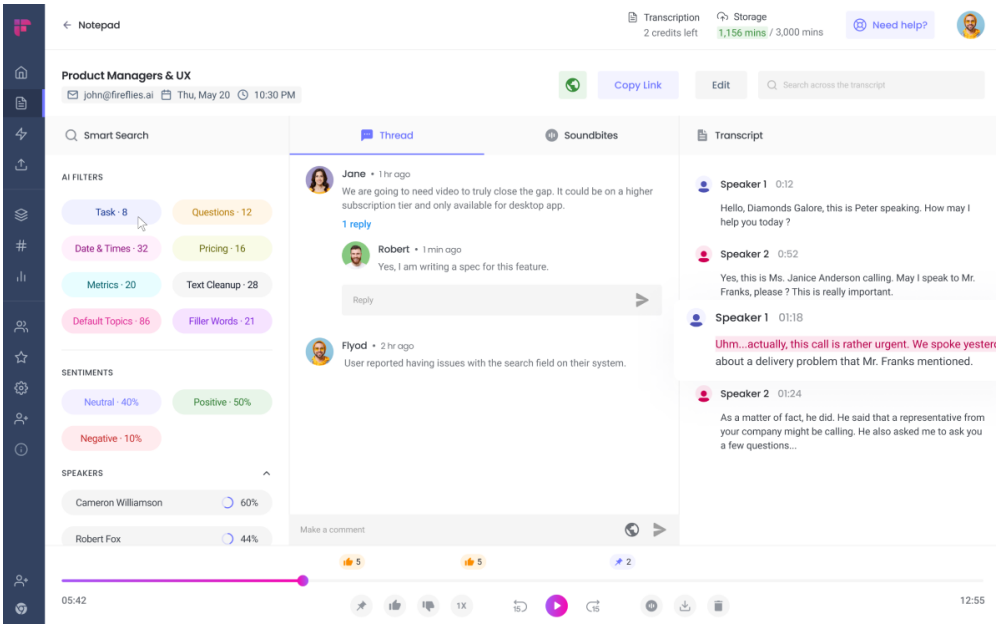

Рисунок 1.2 — Інтерфейс Fireflies.ai

Слід зазначити, що обидва сервіси орієнтовані переважно на обробку полілогу — онлайн-нарад із множиною учасників. Для сценаріїв монологу (персональні аудіозаписи) або діалогу (розмова з одним співрозмовником) їхнє використання може виявитися економічно недоцільним з огляду на вартість підписки. Окрім того, принципово важливим обмеженням є виключно хмарна архітектура цих рішень: жодне з них не підтримує локального розгортання. Це унеможливлює їх застосування в умовах обмеженого доступу до мережі або суворих вимог щодо конфіденційності даних, оскільки вся обробка відбувається на серверах «стороннього постачальника послуг».

До цієї ж категорії належать сервіси Notta, Rev та Fathom, які також певною мірою різняться за функціональністю та сферою використання. Спільним і визначальним обмеженням для всіх перелічених рішень залишається відсутність можливості локального розгортання та пов’язані з цим ризики щодо захисту персональних даних і незалежності від інфраструктури постачальника.

1.2.2 Релевантні системи

Системи другого типу здебільшого представлені локальними програмами або скриптами. Вони або базуються на детермінованих алгоритмах обробки природної мови, або безпосередньо є реалізацією відповідних моделей машинного навчання. Розглянемо два характерні приклади — WhisperCPP CLI та PrivateGPT, кожен із яких покриває окрему підзадачу запропонованої системи.

WhisperCPP є оптимізованою реалізацією ASR-моделі (Automatic Speech Recognition), орієнтованою насамперед на розробників програмного забезпечення. Разом із тим проєкт також надає утиліту командного рядка whisper-cli, придатну для безпосереднього використання без написання коду. Суттєвим обмеженням є те, що програма виконує виключно транскрипцію аудіосигналу у текст, не здійснюючи подальшої структуризації або семантичного аналізу отриманого матеріалу. Приклад виклику утиліти наведено на рис. 1.3.

```
./build/bin/whisper-cli -h

usage: ./build/bin/whisper-cli [options] file0 file1 ...
supported audio formats: flac, mp3, ogg, wav

options:
  -h,        --help              [default] show this help message and exit
  -t N,      --threads N         [4      ] number of threads to use during computation
  -p N,      --processors N      [1      ] number of processors to use during
                                           computation
  -ot N,     --offset-t N        [0      ] time offset in milliseconds
  -on N,     --offset-n N        [0      ] segment index offset
  ...
```

Рисунок 1.3 — Командна утиліта whisper-cli

PrivateGPT є локальною системою для роботи з великими мовними моделями (LLM), що забезпечує можливість діалогової взаємодії з документами без передачі даних зовнішнім сервісам. Принциповим обмеженням цього рішення є відсутність підтримки транскрипції аудіофайлів та автоматичної структуризації текстового матеріалу — задач, що є ключовими для розроблюваної системи. Приклад робочого інтерфейсу наведено на рис. 1.4.

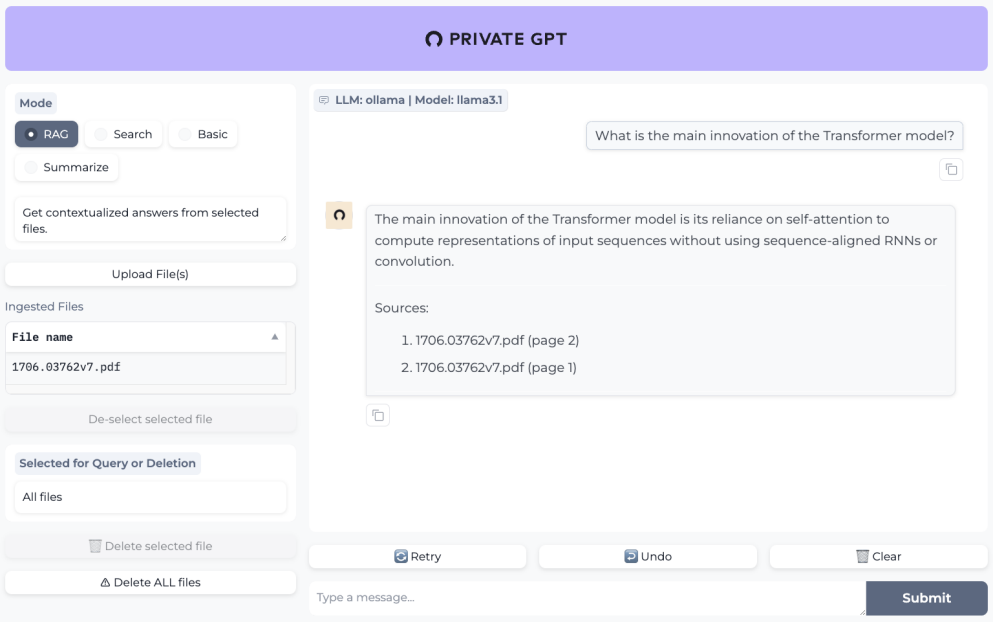

Рисунок 1.4 — Інтерфейс PrivateGPT

1.2.3 Висновки з аналізу

Проведений огляд засвідчує, що наявні рішення розподіляються на два полюси: хмарні сервіси з розвиненим функціоналом, але з обмеженнями щодо конфіденційності та відсутністю локального режиму роботи, і локальні інструменти, що забезпечують повний контроль над даними, проте реалізують лише окремі компоненти необхідного функціоналу. Система, що пропонується в цій роботі, спрямована на усунення зазначеної прогалини шляхом поєднання повноти функціональності з можливістю локального розгортання та збереженням конфіденційності оброблюваних даних.

2 ЗАСОБИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ

Цей розділ присвячений огляду та обґрунтуванню технологічного стеку, обраного для реалізації системи. Послідовно розглядаються ключові складові: технічні обмеження локального середовища розробки, особливості Python як основної мови реалізації, а також інструменти для розпізнавання мовлення, обробки природної мови, векторного представлення тексту, мовного моделювання, пошуку та побудови графічного інтерфейсу. Для кожного з напрямків наводиться порівняльний аналіз наявних рішень із подальшим обґрунтуванням вибору конкретної бібліотеки чи моделі.

Особливу увагу приділено питанню ефективності в умовах обмежених обчислювальних ресурсів, що є визначальним чинником при локальному розгортанні системи. Саме ця вимога стала наскрізним критерієм відбору: перевага надавалася рішенням, що поєднують прийнятну якість результатів із мінімальним споживанням пам’яті та обчислювальних потужностей — зокрема, квантизованим моделям, оптимізованим рушіям інференції та компактним архітектурам із доведеною практичною придатністю.

2.1 Технічні обмеження проекту

Багато сучасних програм ефективно використовують моделі машинного навчання на серверах з потужним залізом та груповими операціями (batching) [11]. Це дозволяє максимально завантажити GPU, амортизувати накладні витрати на запуск ядер CUDA та досягти високої пропускної здатності. Однак локальна розробка накладає певні обмеження.

Оперативна пам’ять стає першою перешкодою. Споживча відеокарта з 8-16 ГБ VRAM кардинально відрізняється від серверної A100/H100 з 40-80 ГБ. Це означає, що навіть завантаження моделі може виявитися проблемою, не кажучи

вже про обробку даних. Доводиться вдаватися до компромісів: квантизація (int8, int4), offloading частини ваг у RAM або на диск, обрізка батчів до розміру 1.

Відсутність batching, по суті, змінює режим роботи відеокарти. На сервері GPU постійно зайнятий потоком запитів; локально ж кожен одиничний запит запускає ядра, які оптимізовані під паралельну обробку, але виконуються послідовно. Через це реальна затримка на запит може бути непропорційно вищою, ніж можна було б очікувати.

Ці обмеженням локальних пристрої були враховані при розробці та лягли в основу побудови системи.

2.2 Особливості мови програмування Python

Мова програмування Python [12] належить до категорії інтерпретованих та динамічно типізованих мов і використовує механізм автоматичного управління пам’яттю — збирач сміття (Garbage Collector). Еталонна реалізація мови — CPython — написана мовою C, тоді як Cython [13] слугує мостом між Python та C, забезпечуючи компіляцію Python-коду до C-розширень і суттєво підвищуючи продуктивність обчислювально інтенсивних задач.

Однією з ключових переваг Python є лаконічний та читабельний синтаксис у поєднанні з багатою стандартною бібліотекою, що охоплює широкий спектр задач — від роботи з файловою системою та мережевими протоколами до засобів тестування та паралельних обчислень. Це дозволяє скоротити обсяг шаблонного коду та зосередитися на предметній логіці застосунку.

Суттєвою перевагою мови є також простота інтеграції зі стороннім кодом через механізм прив’язок (bindings). Використання пакетів (packages) та пакетів простору імен (namespace packages) значно полегшує створення інтерфейсів до бібліотек, реалізованих іншими мовами програмування. Завдяки цьому сформувалась розвинена екосистема розширень, написаних низькорівневими або високопродуктивними мовами — C/C++, Cython та Rust, — що надають доступ до

критично важливих для продуктивності функцій безпосередньо з Python-середовища.

Разом із тим слід зазначити, що при такому підході Python не здійснює повноцінного контролю над оперативною пам’яттю (RAM/VRAM): підключені бібліотеки, як правило, мають власне середовище виконання (runtime) і керують пам’яттю самостійно, поза механізмами збирача сміття CPython. Це може ускладнювати діагностику витоків пам’яті та налагодження продуктивності в гетерогенних стеках.

Незважаючи на зазначені обмеження, Python залишається де-факто стандартом у галузі машинного навчання та штучного інтелекту. Переважна більшість провідних бібліотек мають зрілі та надійні реалізації, підкріплені вичерпною документацією та активною спільнотою розробників.

2.3 Автоматичне розпізнавання мовлення (ASR)

На сьогоднішній день існує значна кількість провідних систем автоматичного розпізнавання мовлення (Automatic Speech Recognition, ASR): OpenAI Whisper, NVIDIA Parakeet, NVIDIA Canary, IBM Granite, Qwen3-ASR та інші. У межах даного проекту було обрано реалізацію на основі моделі Whisper [14, 15], яка ґрунтується на архітектурі енкодер-декодер трансформерів (encoder-decoder Transformer).

Модель Whisper пройшла навчання на корпусі обсягом 680,000 годин «слабко розміченого» багатомовного аудіоматеріалу, зібраного з відкритих інтернет-ресурсів. Такий підхід до навчання зумовлює ключову перевагу моделі: вона не потребує додаткового донавчання для виконання конкретних задач і здатна розпізнавати різноманітні акценти, фонові шуми та розмовну лексику без жодних модифікацій. Широка функціональність і висока якість розпізнавання забезпечили моделі значну популярність у науковій та інженерній спільнотах, що, своєю чергою, стимулювало появу низки альтернативних реалізацій:

а) Faster-Whisper [16];

б) WhisperCPP [17];

в) WhisperX [18].

Faster-Whisper (реалізація 2.3, а)) перебудовує оригінальну модель на основі бібліотеки CTranslate2 [19] — оптимізованого механізму інференції, реалізованого на C++, що орієнтований на роботу з трансформерними архітектурами. Ключовою особливістю є підтримка 8-бітного квантування як для центрального, так і для графічного процесора, що дозволяє суттєво знизити обсяг споживаної оперативної пам’яті без значної втрати якості розпізнавання.

WhisperCPP (реалізація 2.3, б)) є нативним портом Whisper мовою C/C++, розробленим з акцентом на високопродуктивну інференцію на широкому спектрі апаратних платформ. Серед підтримуваних архітектур — Apple Silicon з апаратним прискоренням через Metal API, процесорні розширення ARM NEON, а також вбудовані системи типу System-on-Chip (SoC), що функціонують виключно на центральному процесорі.

WhisperX (реалізація 2.3, в)) побудований поверх Faster-Whisper та розширює його функціональність у напрямку більш точного визначення часових міток і розмежування мовців (speaker diarization), що робить його особливо придатним для транскрибування багатоосібних діалогів.

Для реалізації проекту було обрано бібліотеку Faster-Whisper з огляду на її переваги перед альтернативними варіантами. Оригінальна реалізація Whisper орієнтована насамперед на відтворення результатів академічного дослідження і свідомо позбавлена будь-яких оптимізацій продуктивності чи ефективного управління пам’яттю. WhisperCPP, попри широку кросплатформену сумісність, не надає нативних прив’язок до Python, що суттєво ускладнює інтеграцію у середовища розробки, побудовані на цій мові. WhisperX зміщує фокус у бік обробки багатоосібного аудіо та визначення часових міток, що виходить за межі вимог даного проекту. Faster-Whisper, натомість, поєднує в собі оптимізовану продуктивність інференції та повноцінну нативну інтеграцію з екосистемою Python, що відповідає технічним вимогам проекту.

2.4 Обробка природної мови (NLP)

Обробка природної мови (Natural Language Processing, NLP) є однією з ключових підгалузей машинного навчання, що зосереджується на розробці методів та алгоритмів, які дозволяють обчислювальним системам розуміти, інтерпретувати та генерувати людську мову в різних її формах. Розуміння меж цього терміну залишаються дискусійними у сучасній науковій літературі: поряд із класичними лінгвістичними підходами NLP дедалі частіше охоплює методи глибокого навчання [20], що суттєво розширює традиційне розуміння цієї галузі та ускладнює її однозначне визначення.

Для реалізації базових алгоритмів обробки природної мови у цьому проекті розглядаються дві провідні бібліотеки — SpaCy [21] та NLTK [22]. Попри спільну предметну область, зазначені інструменти суттєво різняться за архітектурними рішеннями, продуктивністю та типовими сценаріями застосування, що обумовлює необхідність їх порівняльного аналізу.

SpaCy є промисловою бібліотекою, орієнтованою на розгортання у виробничому середовищі. Ключовою перевагою цього інструменту є висока обчислювальна ефективність, досягнута завдяки реалізації на мові Cython — надмножині Python, що компілюється у машинний код. Бібліотека інтегрує найбільш ефективні серед відомих алгоритми для кожного з підтримуваних завдань: токенізації, морфологічного аналізу, розпізнавання іменованих сутностей та синтаксичного розбору. Такий підхід — одна оптимальна реалізація замість множини альтернатив — забезпечує передбачуваність та відтворюваність результатів у реальних застосунках.

NLTK (Natural Language Toolkit), натомість, позиціонується передусім як освітній та дослідницький інструмент. Бібліотека надає широкий спектр алгоритмів для кожного типу задач, включно з класичними та експериментальними підходами, що робить її незамінною для прототипування та порівняльного аналізу методів. Реалізація на чистому Python спрощує читання та

модифікацію вихідного коду, однак значно поступається SpaCy у швидкодії, особливо при обробці великих текстових корпусів.

З огляду на вимоги проекту — зокрема, відсутність потреби у специфічних чи експериментальних алгоритмах — у якості основного інструменту обрано SpaCy. Бібліотека повністю покриває запланований спектр завдань, забезпечуючи при цьому необхідну продуктивність для розгортання фінального програмного продукту.

2.5 Кодування слів у векторний простір (Word Embeddings)

Представлення слів та тексту у вигляді багатовимірних векторних масивів (Word Embeddings) [23] є одним із фундаментальних підходів сучасної обробки природної мови, що забезпечує можливість машинного розуміння людського тексту. Завдяки тому, що векторні представлення слугують базисом для текстових генеративних моделей, цей підхід набув широкого поширення в академічних і промислових застосуваннях.

Переважна більшість алгоритмів машинного навчання та штучного інтелекту не здатні безпосередньо обробляти необроблений текст. Традиційні методи векторизації, зокрема one-hot encoding, присвоювали кожному слову унікальний числовий ідентифікатор без урахування його семантичного значення. Наслідком цього підходу було формування надмірно розріджених і обчислювально неефективних представлень, позбавлених контекстуальної інформації.

Принципова перевага векторних ембедінгів полягає у кодуванні семантичного змісту слів у неперервному математичному просторі, що одночасно вирішує кілька ключових проблем. По-перше, слова зі схожими значеннями або вживаними контекстами отримують вектори, геометрично близькі між собою в багатовимірному просторі, що дозволяє моделювати семантичну подібність через метрики відстані. По-друге, замість розріджених і високовимірних векторів, характерних для one-hot encoding, ембедінги оперують у «просторах нижчої

вимірності» — зазвичай від кількох десятків до кількох сотень вимірів, — ефективно стискаючи лексичні та граматичні нюанси слова в компактне щільне представлення.

Першим проривним алгоритмом побудови сучасних щільних ембедінгів став Word2Vec [24]. У сучасній термінології він належить до категорії статичних вбудовувань: слово отримує фіксований вектор незалежно від контексту речення, в якому воно вжите. Натомість сучасні архітектури штучного інтелекту спираються на контекстні вбудовування, що генеруються великими мовними моделями та «кодерами на основі механізму уваги» — зокрема BERT і GPT, — які динамічно адаптують векторне представлення слова відповідно до оточуючих слів у реченні. Це принципово розширює виражальні можливості моделі та дозволяє розрізняти полісемічні слова залежно від контексту.

У рамках цього проекту розглядається реалізація моделі Nomic Embed v2 MoE [25, 26] — першої універсальної моделі векторного вбудовування тексту, побудованої на архітектурі Mixture of Experts (MoE) [27]. Із загальної кількості 475 мільйонів параметрів під час інференції активується лише 305 мільйонів, що суттєво скорочує затримку та обчислювальні витрати, зберігаючи при цьому точність, порівнянну з повністю щільними моделями вдвічі більшого розміру. Модель навчена на корпусі з 1,6 мільярда пар прикладів приблизно у 100 мовах, що забезпечує високу ефективність як у багатомовних, так і в одномовних задачах. Додатково модель навчена з використанням Matryoshka Representation Learning [28], що дозволяє скорочувати розмір векторів до трьох разів із мінімальними втратами у якості представлення. Сукупність цих властивостей робить модель особливо привабливою для розгортання в умовах локального обчислювального середовища з обмеженими ресурсами.

Для роботи з моделями векторного вбудовування та переранжування (Reranker), необхідність і принцип роботи якої буде розглянуто далі у розділі 2.7, використовуються бібліотеки Transformers [29] та Sentence-Transformers [30]. За своєю концептуальною основою бібліотеки тісно пов’язані: Sentence-Transformers є надбудовою над Transformers, що спирається на її базову інфраструктуру. Водночас вона пропонує спрощений і розширений програмний інтерфейс,

орієнтований на практичне застосування та навчання сучасних моделей векторного вбудовування і переранжування текстів.

2.6 Глибоке розуміння контексту (SLM/LLM)

Виконання окремих завдань, передбачених системою, вимагає глибшого розуміння контексту оброблюваних даних. Зокрема, задачі формування змістовних заголовків секцій і автоматичного підведення підсумків не піддаються вирішенню засобами традиційних статистичних алгоритмів — їхні підходи, засновані на частотному аналізі чи ранжуванні ключових слів, виявляються недостатніми для забезпечення семантично повноцінних результатів. Єдиним ефективним рішенням у даному контексті є застосування великої або малої мовної моделі (LLM/SLM) [31, 32], здатної до глибокого семантичного аналізу природної мови.

На сьогоднішній день існує низка провідних мовних моделей із відкритим вихідним кодом, придатних для локального розгортання без необхідності звернення до хмарних сервісів. Серед найбільш поширених і активно підтримуваних: DeepSeek, Qwen, Llama, Gemma, GPT-oss та інші. Порівняльний аналіз сучасних відкритих моделей демонструє їхні загалом високі показники якості [33], що суттєво нівелює різницю між ними та переносить вибір переважно у площину персональних уподобань або специфіки конкретного застосування. Оскільки в межах розробленого проєкту мовна модель залучається виключно для опрацювання текстових даних, мультимодальні можливості не є визначальними. Натомість ключовою та єдиною суттєвою вимогою залишається компактний розмір моделі — з огляду на необхідність її ефективного виконання на ресурсно обмежених пристроях.

З урахуванням зазначених вимог у проєкті було обрано модель Qwen3.5 [34] з кількістю параметрів 4 мільярди. Попри те, що еталонною версією у даному сімействі вважається модель на 9 мільярдів параметрів, обрана конфігурація демонструє цілком задовільну якість генерації тексту відносно свого розміру. Це

робить її обґрунтованим компромісом між обчислювальною ефективністю та функціональною придатністю в умовах реального розгортання.

Для розгортання та виконання мовної моделі існує кілька платформ із різними характеристиками. Перша група — Ollama [35, 36] та LM Studio [37] — орієнтована на простоту використання: обидва інструменти забезпечують зручний інтерфейс для локального запуску моделей і пропонують REST API, що спрощує інтеграцію у прикладні застосунки. Друга група — Llama.cpp [38] та vLLM [39, 40] — представляє низькорівневі рушії виконання, розраховані на виробниче середовище. Вони надають розширені можливості оптимізації та тонкого налаштування, зокрема квантизацію, керування пам’яттю та паралельну обробку запитів, однак потребують значних зусиль із конфігурації та відповідної технічної експертизи.

У рамках проєкту як платформу виконання обрано Ollama, оскільки специфіка застосування не передбачає необхідності у нестандартних архітектурних рішеннях чи низькорівневому системному налаштуванні. Ollama забезпечує достатній рівень функціональності при мінімальних витратах на розгортання, що відповідає загальним вимогам проєкту до простоти інтеграції та відтворюваності середовища виконання.

2.7 Архітектура алгоритму пошуку

Реалізація пошукового алгоритму є нетривіальним завданням, яке суттєво варіюється залежно від типу даних і конкретної прикладної задачі. У випадку роботи з текстовими даними існує широкий клас алгоритмів, що реалізують зіставлення зі зразком (pattern matching) [41, 42]. Проте такий підхід накладає на користувача жорстку вимогу — чітке та однозначне формулювання пошукового запиту, що в умовах реальної взаємодії не завжди є можливим або зручним.

Альтернативним рішенням є семантичний пошук (Semantic Search) [20, 32] — метод, що ґрунтується на пошуку семантичної схожості у векторному просторі

ембедингів. На відміну від класичного лексичного зіставлення, семантичний пошук оперує не буквальним збігом символів, а семантичним навантаженням слів і фраз — їхнім контекстним значенням у векторному представленні. Це дозволяє отримувати результати, що є змістовно спорідненими із запитом, навіть за відсутності прямого лексичного збігу.

Разом з тим, семантичний пошук має низку суттєвих обмежень: якість результатів безпосередньо залежить від способу формування векторних представлень, розміру текстових фрагментів (чанків), ступеня інформативності самих векторів, а також від жанрово-стилістичних характеристик тексту (розмовний, технічний, художній тощо). Першочерговим і найбільш ефективним способом поліпшення якості пошуку є використання якісних механізмів побудови ембедингів. Подолання інших зазначених обмежень потребує застосування різноманітних додаткових підходів.

Одним із поширених і практично доцільних способів підвищення якості є гібридний пошук — метод, що поєднує кілька пошукових стратегій для формування кінцевого результату. Типовою конфігурацією є комбінація RAG та BM25, що засвідчила помітне покращення якості пошуку на різноманітних наборах даних. Утім, алгоритм BM25 є статистичним за своєю природою та потребує достатнього за обсягом текстового корпусу для ефективного функціонування, що є його принциповим обмеженням.

Слід також зазначити, що існує широкий спектр більш складних і ресурсномістких підходів до вдосконалення пошуку. Проте в контексті даного проєкту необхідно враховувати архітектурне обмеження: система орієнтована на роботу лише з одним файлом. За таких умов жоден із запропонованих алгоритмів не може продемонструвати той рівень ефективності, що спостерігається при роботі з корпусами обсягом у тисячі або сотні тисяч документів. Відтак, реалізація має бути узгоджена з реалістичними очікуваннями щодо якості результатів при малому обсязі вхідних даних.

У рамках проєкту пропонується використовувати алгоритму пошуку, схему якого зображену на рис. 2.1.

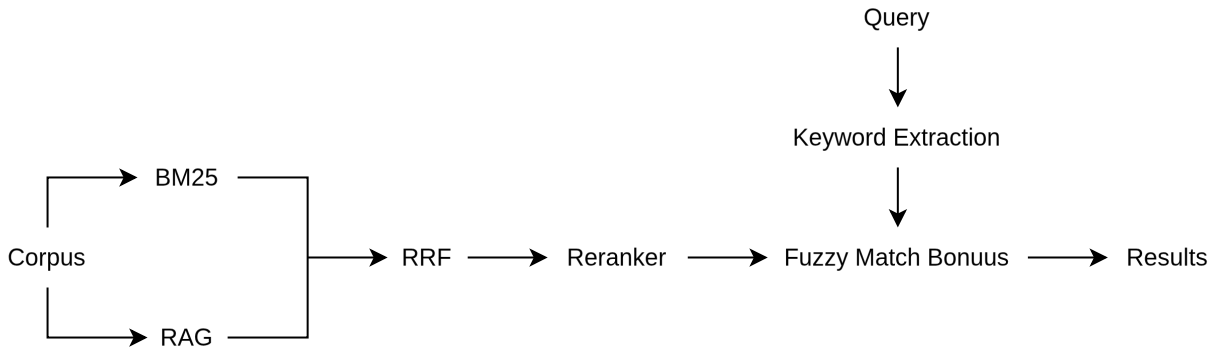

Рисунок 2.1 — Схема пошуку системи

Запропонована схема функціонує таким чином: пошуковий запит паралельно обробляється двома методами — Retrieval-Augmented Generation (RAG) та Best Match 25 (BM25); отримані ранжовані списки результатів об’єднуються за допомогою методу Reciprocal Rank Fusion (RRF); зведений результат переоцінюється крос-енкодером (Cross-Encoder) [43], після чого відкидаються результати з від’ємними значеннями релевантності; паралельно із запиту користувача алгоритмом YAKE! виокремлюються ключові слова, на основі яких до переоцінених результатів додається коефіцієнт нечіткого збігу (Fuzzy Match Bonus); фінальний список сортується за сукупним балом і передається як результат пошуку.

Необхідно підкреслити, що запропонований алгоритм не є оптимальним для малих документів і не забезпечуватиме суттєвого приросту якості при невеликих обсягах тексту. Водночас ефективність системи зростатиме пропорційно до збільшення обсягу документа та різноманітності його змістового наповнення.

Для реалізації описаної архітектури системи гібридного пошуку та обробки інформації обрано такі бібліотеки:

- RAG: Nomic-Embed-v2-MoE;
- BM25: bm25s [44, 45];
- RRF: власна реалізація;
- Reranker: jina-reranker-v3 [46, 47];
- Keyword Extraction: YAKE! [48, 49].

Векторне кодування корпусу та запиту, здійснення семантичного пошуку виконується на основі ембедингів Nomic-Embed-v2, представлених та детально описаних у розділі 2.5.

Високопродуктивною реалізацією алгоритму BM25 мовою Python є BM25S, що залежить виключно від бібліотек NumPy та SciPy. Принципова відмінність BM25S від аналогів полягає у стратегії eager-обчислення: BM25-оцінки обчислюються не в момент пошукового запиту, а завчасно — на етапі індексування, — і зберігаються у вигляді розріджених матриць. Завдяки цьому бібліотека досягає до 500-кратного прискорення порівняно з найпоширенішою Python-реалізацією BM25. Крім того, BM25S відтворює точні реалізації п'яти варіантів алгоритму BM25, задокументованих у академічній роботі [45], розширюючи цей підхід на нерозріджені варіанти за допомогою методу зсуву оцінок (score shifting).

Метод агрегації результатів пошуку з кількох незалежних ранжуючих систем Reciprocal Rank Fusion (RRF), що не вимагає нормалізації вихідних оцінок і є стійким до відмінностей у шкалах різних ранжирів. Кожному документу присвоюється оцінка за формулою:

$$\sum_r \frac{1}{k+\mathrm{rank}_r(d)}, \tag{2.1}$$

де k — константа згладжування,

$\mathrm{rank}_r(d)$ — позиція документа d у ранжуванні r.

У даній роботі RRF реалізовано власноруч з метою об'єднання результатів семантичного (RAG) та лексичного (BM25) пошуку в єдиний ранжований список.

jina-reranker-v3 — мультимовна модель переранжування документів, що реалізує новий механізм last but not late interaction (LBNL). На відміну від моделей із пізньою взаємодією (зокрема ColBERT), в яких взаємодія між запитом і документами відбувається лише після завершення незалежного кодування кожної сторони, механізм LBNL виконує причинно-наслідкову само-увагу між запитом і документами в межах єдиного контекстного вікна трансформера. Це дозволяє документам формувати семантичні зв'язки ще до вилучення контекстних

ембедингів, що підвищує точність переранжування. Модель побудована на основі Qwen3-0.6B з 28 трансформерними шарами та здатна обробляти до 64 документів одночасно у контексті 131K токенів, досягаючи значення 61,94 nDCG@10 на еталонному наборі даних BEIR [46].

Легковагий несупервізований метод автоматичного вилучення ключових слів YAKE! (Yet Another Keyword Extractor), що ґрунтується виключно на статистичних характеристиках тексту, здобутих із окремих документів. Система не потребує попереднього навчання на спеціалізованих наборах даних і не залежить від зовнішніх словників, корпусів, обсягу тексту, мови або предметної галузі, що робить її придатною до застосування в умовах обмеженої доступності лінгвістичних ресурсів. Порівняльне дослідження на двадцяти наборах даних різного обсягу, мови та домену підтвердило, що YAKE! суттєво перевершує інші несупервізовані підходи до вилучення ключових слів [48].

2.8 Графічне відображення (GUI)

З метою забезпечення зручного та інтуїтивно зрозумілого користування розробленою системою оптимальним рішенням є створення графічного інтерфейсу користувача (GUI — Graphical User Interface). Мова програмування Python пропонує широкий спектр бібліотек для розробки GUI, серед яких найбільш поширеними є: Tkinter, PyQt6 / PySide6, Kivy, Flet, Dear PyGui та інші.

Відповідно до функціональних вимог проекту, графічний інтерфейс повинен забезпечувати такі можливості: завантаження файлів у підтримуваних форматах, відображення тексту транскрипції, візуалізацію поточного стану обробки, а також виконання текстового пошуку з підсвіткою знайдених збігів. Аналіз наведених бібліотек свідчить про те, що більшість із них тією чи іншою мірою задовольняє зазначеним вимогам. Єдиним очевидним винятком є бібліотеки класу immediate mode GUI, зокрема Dear PyGui: вони функціонують у режимі реального часу з покадровою перерисовкою інтерфейсу та, як правило, не передбачають розвиненої

підтримки складних текстових компонентів. Це суттєво ускладнює реалізацію функцій відображення та інтерактивного пошуку у великих текстових масивах, що робить подібні бібліотеки малопридатними для цілей даного проекту.

За результатами порівняльного аналізу для реалізації графічного інтерфейсу було обрано фреймворк PySide6 [50]. Ця бібліотека є офіційним Python-прив’язуванням до кросплатформенного фреймворку Qt і надає доступ до розгалуженого набору візуальних компонентів (QWidget), що охоплюють більшість типових елементів інтерфейсу. Крім того, PySide6 підтримує використання декларативної мови розмітки Qt Meta-object Language (QML) — мови, заснованої на синтаксисі ECMAScript/JavaScript і призначеної для опису структури та поведінки інтерфейсу користувача у декларативному стилі. Це дозволяє чітко розмежувати логіку застосунку й представлення інтерфейсу, що позитивно позначається на супроводжуваності коду.

Визначальною перевагою PySide6 є ґрунтовна та систематизована офіційна документація, що суттєво скорочує час на освоєння бібліотеки та налагодження окремих компонентів. Оскільки PySide6, як і PyQt6, є прив’язуванням до Qt, реалізованого на мові C++, інтерфейс, побудований на її основі, успадковує високу продуктивність рідного C++-рушія, що особливо важливо при роботі зі значними обсягами тексту та складними операціями відображення.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У сучасних системах обробки природної мови якість кінцевого результату визначається не лише потужністю окремих моделей, а й тим, наскільки злагоджено взаємодіють усі компоненти системи. Ефективна архітектура передбачає ретельне налаштування кожного модуля — від розпізнавання мовлення до семантичного аналізу та пошуку інформації — з урахуванням як обчислювальних обмежень, так і вимог до якості та швидкодії.

У цьому розділі детально розглядаються основоположні технічні рішення, прийняті в процесі розробки системи. Спочатку розглядається основний принцип роботи системи: її основний пайплайн та алгоритми, етапи обробки та принципи міжмодульної системи. Далі слідує детальний розгляд ключових технічних рішень, прийнятих в процесі розробки системи: конфігурація модуля автоматичного розпізнавання мовлення на основі Faster-Whisper, алгоритм семантичного розподілу тексту з використанням ковзного вікна, підходи до налаштування великої мовної моделі через промпт-інжиніринг, а також реалізація гібридного пошукового механізму, що поєднує лексичні та семантичні методи.

Обґрунтування кожного з цих рішень спирається на результати емпіричного тестування та порівняльного аналізу альтернативних підходів.

3.1 Огляд основного пайплайну

У цьому розділі наведено загальний опис архітектури програмного пайплайну системи та детально розглянуто його ключові етапи: приймання вхідних даних, сегментація та структуризація тексту, генерація заголовків і підсумків, а також індексація й виконання пошукових запитів. Узагальнену схему алгоритму функціонування системи зображено на рис. 3.1.

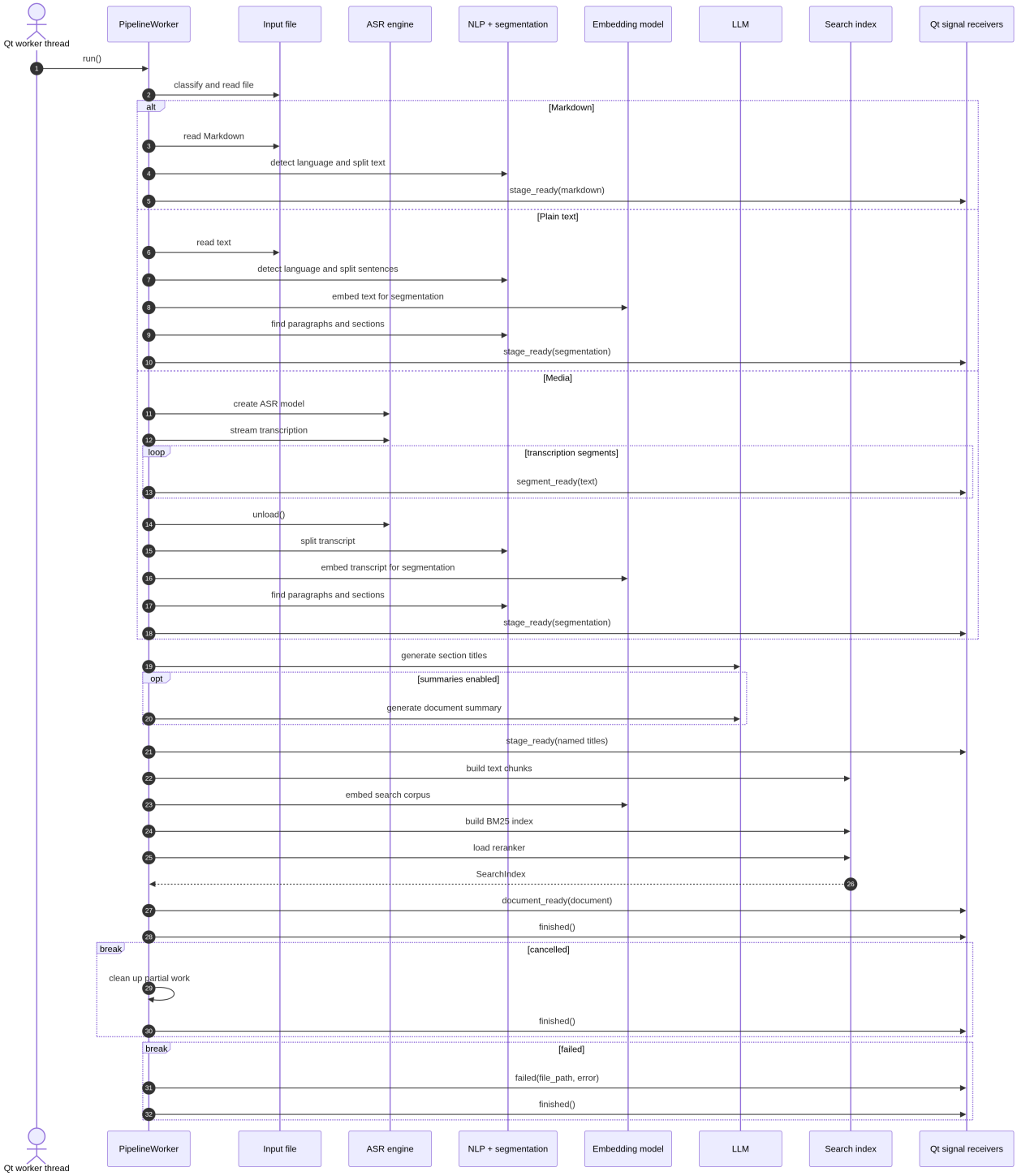

Рисунок 3.1 — Діаграма послідовності основного пайплайну

Діаграма відображає розгалужену структуру пайплайну, де маршрут обробки визначається типом вхідного файлу. Незалежно від обраної гілки, всі шляхи сходяться на спільних етапах генерації заголовків та побудови пошукового індексу.

3.1.1 Вхідні дані та класифікація файлів

На вхід системи подається файл довільного формату. Відповідно до характеру вмісту та розширення, система розпізнає три архетипи вхідних даних:

а) файл у форматі Markdown (*.md, *.markdown);

б) текстовий файл із небінарним вмістом (будь-яке розширення);

в) бінарний файл (будь-яке розширення).

Файл формату Markdown (пункт 3.1.1, а)) ідентифікується виключно за розширенням *.md або *.markdown. Оскільки такий файл розглядається як вже структурований і відформатований документ з наявним логічним розподілом на розділи, основний пайплайн обробки для нього не застосовується. Натомість виконується лише підготовка індексу для подальшого пошуку.

Текстовий файл (пункт 3.1.1, б)) — це будь-який файл, вміст якого є небінарним і доступним для прямого читання. Такий вміст вважається неструктурованим, тому для нього виконується повний пайплайн сегментації та структуризації тексту. Перед цим здійснюється базовий препроцесинг: нормалізація «пробільних символів», метою якої є усунення зайвих пробілів, табуляцій та нерегулярних переносів рядків, що можуть перешкоджати коректному синтаксичному аналізу. Після структуризації виконується індексація для пошуку.

Бінарний файл (пункт 3.1.1, в)) — це файл із бінарним вмістом, який не може бути безпосередньо прочитаний як текст; цей тип вважається аудіо або відеофайлом. Для таких файлів спочатку виконується транскрипція: перетворення мультимедійного або нетекстового вмісту на текстове представлення. Після цього застосовується стандартний пайплайн сегментації та структуризації, а завершальним кроком є індексація.

На цьому етапі всі підтримувані формати файлів опрацьовуються з підтримкою української та англійської мов — як для транскрипції аудіоданих, так і для задач обробки природної мови (NLP). Зазначимо, що через технічні обмеження використовуваних NLP-моделей, прив’язаних до конкретних мов,

підтримка справжньої багатомовності в межах одного документа наразі недоступна.

3.1.2 Сегментація та структуризація тексту

Етап сегментації та структуризації передбачає формування граматично й семантично цілісних речень із вхідного тексту, а також розподіл цих речень на параграфи та абзаци на підставі семантичних розривів — різких змін у тематичній схожості між суміжними фрагментами тексту. Метрика подібності обчислюється на основі векторних представлень речень, що дає змогу виявити логічні межі між тематичними блоками без залучення наперед заданих правил.

3.1.3 Генерація заголовків і підсумків

Для кожного отриманого параграфа система автоматично генерує заголовок за допомогою малої мовної моделі (SLM). Заголовки формуються на підставі змісту параграфа та відображають його ключову тему в стислій і інформативній формі. Додатково, якщо відповідний параметр конфігурації активовано користувачем, система генерує короткий підсумок змісту кожного параграфа, що може бути використано для швидкого перегляду документа або подальшого опрацювання.

3.1.4 Гібридний пошук та індексація

Система реалізує гібридний пошук, що поєднує кілька взаємодоповнюючих методів для досягнення високої точності та повноти результатів. Етап індексації передбачає підготовку чанків (фрагментів тексту) для двох механізмів:

семантичного пошуку на основі векторних представлень (RAG — Retrieval-Augmented Generation) та лексичного пошуку за алгоритмом BM25.

Безпосередньо під час виконання пошукового запиту застосовується комплексний алгоритм, описаний у розділі 2.7, який включає: RAG, BM25, злиття результатів за методом RRF (Reciprocal Rank Fusion), ранжування за допомогою Reranker-моделі, а також добування ключових слів із застосуванням нечіткого зіставлення з бонусним коефіцієнтом (Fuzzy Match Bonus). Схему алгоритму пошуку наведено на рис. 3.2.

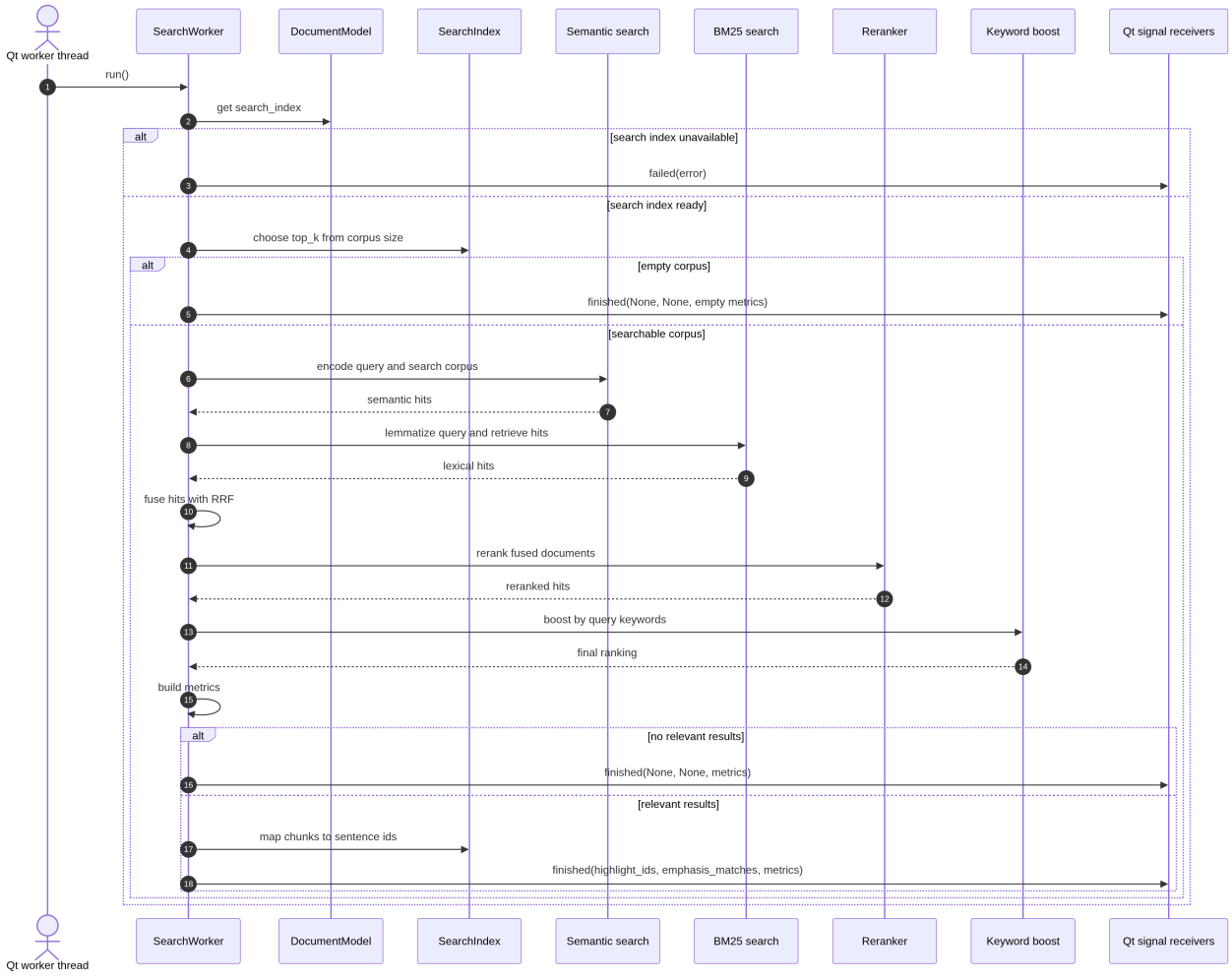

Рисунок 3.2 — Діаграма послідовності алгоритму пошуку

Діаграма ілюструє послідовне виконання пошукового конвеєра — від паралельного отримання результатів семантичного пошуку та BM25 до їх злиття, перевпорядкування та фінального підсилення перед поверненням до інтерфейсу.

3.1.5 Архітектура інтерфейсу та міжмодульна комунікація

Графічний інтерфейс користувача реалізовано з використанням фреймворку Qt. Для забезпечення коректного оновлення стану застосунку та обміну даними між модулями системи в умовах асинхронного виконання застосовуються потоки (threads) і механізм сигналів та слотів Qt — стандартна парадигма міжкомпонентної комунікації в Qt-застосунках, що гарантує потокобезпечність і слабку зв’язаність між компонентами.

3.2 Налаштування модуля ASR

Модуль автоматичного розпізнавання мовлення (ASR) побудовано на основі абстрактних інтерфейсів, що забезпечує архітектурну гнучкість та можливість підключення альтернативних ASR-рушіїв без модифікації суміжних компонентів системи. Для керування екземплярами рушіїв застосовується патерн проєктування «Фабричний метод» (Factory Method), який дозволяє виконувати динамічну заміну моделі під час виконання програми без необхідності перезапуску або реструктуризації коду. Структуру модуля зображено на рис. 3.3.

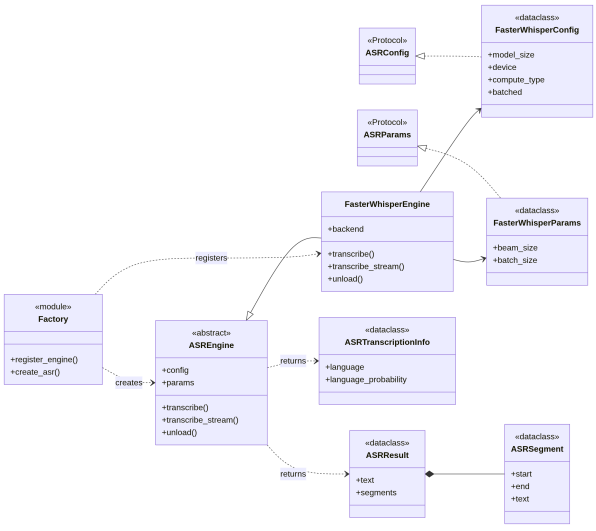

Рисунок 3.3 — Класова діаграма модуля ASR

Розширення модуля новим рушієм є технічно нескладним завданням: достатньо розмістити відповідну реалізацію у директорії asr/engines/ та зареєструвати її у реєстрі фабрики. Така архітектура відповідає принципу відкритості/закритості (Open/Closed Principle) та суттєво знижує поріг введення нових компонентів у систему.

У якості основної моделі розпізнавання мовлення у проєкті використовується large-v3-turbo [51] — дистильований варіант моделі large-v3, оптимізований для суттєво вищої швидкості інференсу при мінімальних втратах точності. Дистиляція реалізована шляхом скорочення кількості декодерних шарів з 32 до 4, що забезпечило зниження споживання відеопам'яті приблизно з 10 до 6 ГБ та збільшення відносної швидкості обробки до ~8x у порівнянні з базовою моделлю large-v2. При цьому показник Word Error Rate (WER) погіршується незначно: з 2.4% до 2.5% для англійської мови та з 3.5% до 3.7% для багатомовного режиму.

Порівняльний аналіз усіх доступних варіантів моделей сімейства Whisper за ключовими характеристиками наведено у табл. 3.1. Вибір моделі large-v3-turbo обґрунтовується оптимальним співвідношенням між обчислювальною вартістю та якістю розпізнавання стосовно вимог розроблюваної системи.

Таблиця 3.1 — Порівняння розміру моделей Whisper [52]

Модель	Параметри, млн	Шари, шт	VRAM, GB	RAM, ~MB	Швидкість, ~	English WER, ~%	Multilingual WER, ~%
tiny	39	4	1	273	10x	7.6	12
base	74	6	1	388	7x	5	10
small	244	12	2	852	4x	3.4	7
medium	769	24	5	2100	2x	2.9	5
large-v2	1550	32	10	3900	1x	2.7	4
large-v3	1550	32	10	3900	1x	2.4	3.5
large-v3-turbo	809	4	6	2300	8x	2.5	3.7

Реалізація ASR-рушія базується на бібліотеці Faster-Whisper, яка використовує рушій CTranslate2 для ефективного інференсу. Конфігурація моделі передбачає автоматичний вибір рівня квантизації залежно від апаратного забезпечення, що дозволяє мінімізувати споживання оперативної та відеопам'яті

без ручного налаштування. Відповідність пристроїв і типів обчислень (Compute Type) наведено у табл. 3.2.

Таблиця 3.2 — Рівні квантизації ASR залежно від пристрою

Пристрій	Тип обчислення
CUDA	int8_float16
MPS	float16
CPU	int8

З метою зниження затримки та рівномірного завантаження обчислювальних ресурсів параметр beam_size встановлено рівним 1 (жадібний декодинг), що зменшує кількість паралельних гіпотез при декодуванні. Значення batch_size диференційовано: 16 для GPU-прискорення та 1 для обробки на CPU, що відповідає різниці у пропускній здатності цих пристроїв.

Для забезпечення потокової обробки аудіоданих у режимі реального часу задіяно механізм ітераторів та відкладеного обчислення (lazy evaluation), який надає бібліотека Faster-Whisper. Це дозволяє починати передачу частково розпізнаних результатів до клієнта ще до завершення обробки всього аудіофрагменту, що суттєво зменшує суб’єктивну затримку відповіді системи.

3.3 Семантичний розподіл речень

Семантичний розподіл речень ґрунтується на представленні текстових одиниць у вигляді векторних ембедінгів та подальшому аналізі їх розташування у багатовимірному семантичному просторі. Для отримання таких представлень використовується бібліотека Sentence-Transformers, яка побудована на основі фреймворку PyTorch [53, 54]. Оскільки PyTorch підтримує різні режими числових обчислень залежно від апаратної конфігурації, у роботі застосовуються типи даних, відмінні від тих, що використовуються в CTranslate2. Відповідні конфігурації наведено у табл. 3.3.

Таблиця 3.3 — Рівні квантизації PyTorch залежно від пристрою

Пристрій	Тип обчислення
CUDA	float16
MPS / CPU	float32

Вибір типу обчислення зумовлений як продуктивністю відповідного апаратного забезпечення, так і підтримуваними операціями: відеокарти з підтримкою CUDA ефективно виконують операції у форматі половинної точності (float16), тоді як процесори та пристрої з архітектурою MPS потребують стандартної одинарної точності (float32). Це пов’язано з тим, що більшість сучасних процесорів не мають нативної апаратної підтримки формату float16, внаслідок чого операції з таким типом даних емулюються програмно. Подібна емуляція не лише не дає жодних переваг у швидкодії, але й може призводити до збільшення часу обчислень та зниження числової точності результатів, що робить використання float32 єдиним обґрунтованим вибором для цих пристроїв.

3.3.1 Алгоритм виявлення семантичних розривів

Основним алгоритмом для виявлення меж між семантично відмінними фрагментами тексту є пошук за подібністю (similarity search), реалізований у вигляді авторської модифікації з використанням методу ковзного вікна (Sliding Window). Суть методу полягає у послідовному обчисленні косинусної подібності між ембедінгами сусідніх речень у межах вікна фіксованого розміру, що дозволяє виявляти локальні зміни тематичної спрямованості тексту. Принцип алгоритму зображено на рис. 3.4.

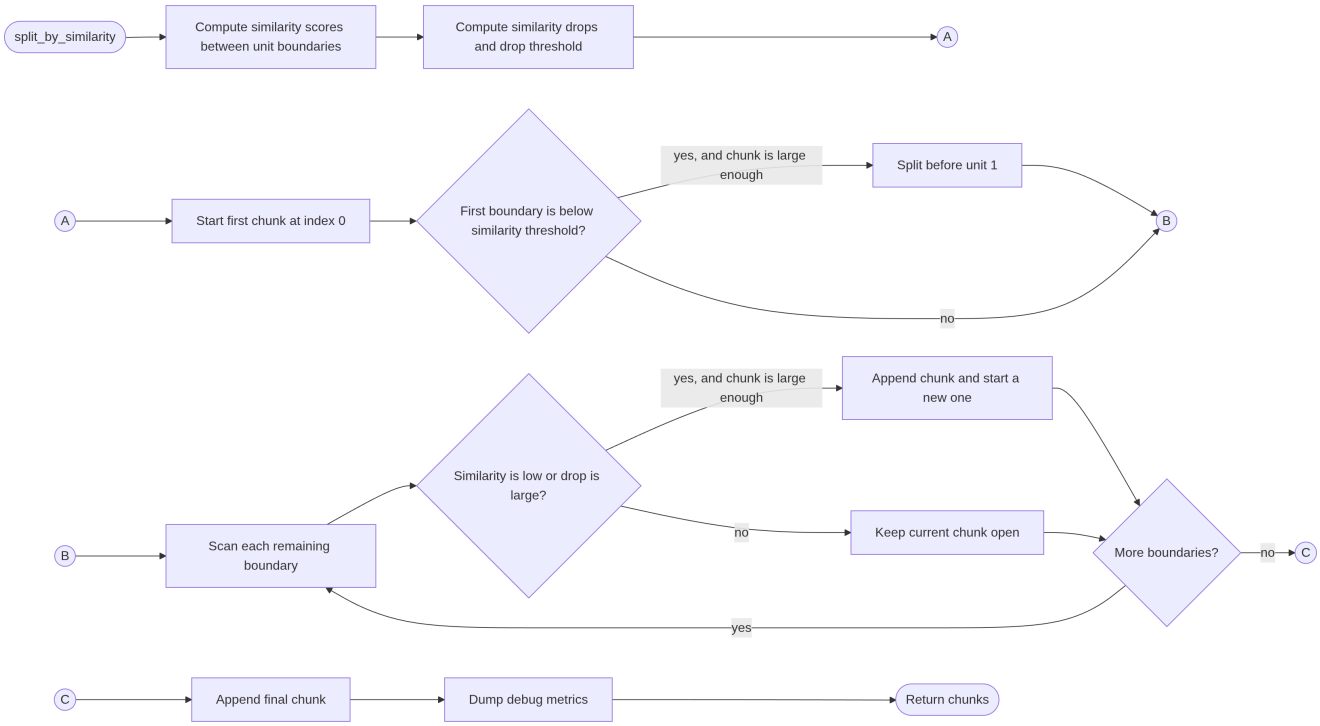

Рисунок 3.4 — Схема послідовності виявлення семантичних розривів

На основі отриманих значень подібності додатково обчислюються спади (drops) — різниці між поточним та попереднім значеннями подібності, які відображають різкість семантичного переходу. З метою нормалізації та приведення спадів до порівнянного масштабу застосовується стандартизація: від кожного значення віднімається середнє арифметичне, а результат ділиться на стандартне відхилення вибірки.

Прийняття рішення про наявність семантичного розриву здійснюється шляхом одночасної перевірки двох незалежних умов: перевищення порогу подібності (similarity threshold) та перевищення порогу нормалізованого спаду (drops threshold). Такий двокритеріальний підхід дозволяє усунути принципові недоліки кожного із критеріїв окремо: використання виключно порогу подібності не дозволяє виявити різкі тематичні зміни в тих випадках, коли абсолютні значення подібності між реченнями залишаються вище встановленого порогу — тобто коли перехід є різким, але обидва суміжні фрагменти мають достатньо спільних семантичних компонентів; використання виключно порогу спаду призводить до помилкового об’єднання семантично віддалених речень у єдиний

сегмент у випадках поступового тематичного дрейфу, коли кожен окремий перехід є плавним, однак накопичений семантичний зсув є суттєвим.

Поєднання обох критеріїв дозволяє охопити обидва типи меж — як різкі тематичні розриви, так і поступові переходи — забезпечуючи тим самим більш повне та точне сегментування тексту.

3.3.2 Параметри порогів та їх налаштування

Для задач розподілу тексту на абзаци та параграфи пропонуються наступні значення порогу подібності (sim_threshold): 0.63 для абзаців та 0.75 для параграфів. Менше значення для абзаців відображає очікування щодо допустимого рівня семантичної відмінності між суміжними реченнями всередині одного смислового блоку, тоді як вище значення для параграфів відповідає вимозі більшої тематичної однорідності у межах великих структурних одиниць.

Слід зазначити, що запропоновані значення порогів є результатом емпіричного підбору та оптимізовані для застосування до широкого спектру типів текстів. Разом із тим, текстові корпуси з вираженою доменною специфікою, нестандартною стилістикою або особливою структурою можуть потребувати індивідуального налаштування зазначених параметрів. Таким чином, запропоновані значення слід розглядати як науково обґрунтовані початкові значення (baseline), що забезпечують прийнятну якість сегментування за замовчуванням, але не виключають можливості подальшої адаптації під конкретні умови застосування.

3.4 Налаштування SLM

Взаємодія з малою мовною моделлю здійснюється через платформу Ollama, яка забезпечує оптимізовані базові параметри виконання моделі. Визначальним

аспектом налаштування є інженерія промптів (prompt engineering) — методологія формулювання інструкцій для моделі засобами природної мови. В рамках розробленої системи необхідно визначити два чітко регламентовані набори правил: для генерації заголовків параграфів та для побудови узагальнювального резюме вхідного тексту.

За результатами практичного тестування встановлено, що модель Qwen3.5-4B демонструє задовільну якість для свого класу параметрів, проте виявляє суттєві обмеження при виконанні окремих завдань. Зокрема, генерація лаконічних заголовків параграфів є відносно складною задачею для моделей такого масштабу, тоді як побудова текстових резюме дається значно краще завдяки менш жорстким формальним обмеженням на вихідний текст.

Для досягнення оптимальної якості результатів найкраще використовувати модель Qwen3.5-4B. Однак, для середовищ з меншим обсягом оперативної пам’яті (VRAM/RAM) система автоматично обирає модель меншого розміру відповідно до наявних апаратних ресурсів. Відповідні значення наведено у табл. 3.4.

Таблиця 3.4 — Розмір моделі від кількості пам’яті

Розмір моделі	Необхідно пам’яті, МіВ	Типовий еквівалент GPU, GB
Qwen3.5-4B	≥ 7192	~8
Qwen3.5-2B	≥ 5120	~6
Qwen3.5-0.8B	< 5120	~4

Для обох завдань застосовується обрізання вхідного контексту до 2000 символів. Ця міра є необхідною для мінімізації ризику галюцинацій — феномену генерації семантично некоректного або фактично хибного тексту, що виникає, зокрема, при перевищенні ефективного контекстного вікна моделі. Обмеження контексту також сприяє стабільності та відтворюваності результатів між різними запусками системи.

3.4.1 Генерація заголовків параграфів

Ключовим обмежувальним чинником при генерації заголовків є ліміт довжини вихідного тексту. Для української мови ця проблема набуває особливої гостроти: речення в українській мові в середньому довші, ніж в англійській, через що моделі значно складніше компресувати розгорнутий зміст абзацу в стислу номінативну конструкцію. Додаткову складність становить робота з художніми або публіцистичними текстами, де семантика визначається переважно через метафору, алюзію або імпліцитний підтекст, що утруднює однозначну тематичну інтерпретацію.

Емпіричним шляхом встановлено, що ліміт у ~10 слів для заголовку є практично прийнятним як для української, так і для англійської мови: він достатньо широкий для передачі основної теми і водночас забезпечує лаконічність заголовку.

3.4.2 Генерація підсумків

Задача підведення підсумків є структурно простішою, оскільки вимоги до формату вихідного тексту менш обмежувальні порівняно з генерацією заголовків. У розробленій системі реалізовано двохпрохідну стратегію підсумовування: перший прохід — незалежне підсумовування кожного параграфу тексту (ліміт: 1 параграф, до 6 речень); другий прохід — об’єднання отриманих проміжних резюме та їх компресія у фінальне підсумування (ліміт: 3 параграфи по 6 речень кожен).

Такий ієрархічний підхід дозволяє зберегти семантичну зв’язність підсумку навіть при роботі з довгими документами, де одноразова обробка повного тексту неможлива через обмеження контекстного вікна.

3.5 Реалізація гібридного пошуку

З метою забезпечення ефективного пошуку релевантної інформації в системі реалізовано механізм гібридного пошуку, що поєднує декілька взаємодоповнювальних методів: розріджений лексичний пошук на основі алгоритму BM25, щільний семантичний пошук із застосуванням підходу RAG (Retrieval-Augmented Generation), злиття рейтингів за методом RRF (Reciprocal Rank Fusion), переранжування результатів за допомогою крос-енкодерної моделі (Reranker), а також евристичний бонусний механізм часткового збігу ключових слів (Fuzzy Match).

З метою підвищення якості роботи лексичних алгоритмів — зокрема BM25 та YAKE — застосовується попередня лематизація тексту засобами бібліотеки SpaCy. Алгоритм YAKE вже налаштовано на роботу зі SpaCy, тоді як для BM25S реалізовано власний мехінзм токенізації, що також використовує SpaCy як базовий інструмент морфологічного аналізу. Такий підхід забезпечує уніфіковану нормалізацію термінів для обох алгоритмів і сприяє покращенню точності лексичного зіставлення.

Для підвищення релевантності семантичного пошуку в алгоритмі RAG застосовується спеціалізована стратегія формування текстових фрагментів (chunking). Речення в межах одного абзацу формують чанки з перекриттям в одне речення, що забезпечує контекстуальну неперервність у межах абзацу. Водночас між різними абзацами перекриття відсутнє, що підвищує локальність пошуку та знижує семантичний шум. Для алгоритму BM25 використовується аналогічна стратегія розбиття, однак без включення перекривних речень, оскільки лексичне зіставлення менш чутливе до розриву контексту між суміжними реченнями.

Обидва алгоритми — BM25 та RAG — застосовують механізм динамічного визначення кількості релевантних фрагментів (top_k) залежно від загального обсягу корпусу. Така адаптація дозволяє уникнути надмірного або недостатнього охоплення пошукового простору. Відповідні значення наведено у табл. 3.5.

Таблиця 3.5 — Значення top_k від кількості речень

Кількість речень у корпусі	Кількість релевантних речень пошуку
≤ 45	5
≤ 90	10
≤ 150	15
> 150	20

Злиття результатів двох алгоритмів виконується за допомогою методу RRF у його класичній реалізації. Метод полягає у присвоєнні кожному документу рейтингового балу, оберненого до зміщеної позиції в ранжованому списку, що дозволяє ефективно агрегувати результати з принципово різних за природою алгоритмів пошуку. Як значення параметра k, що регулює чутливість до позиції в рейтингу, використовується загальноприйняте стандартне значення $k = 60$.

Для додаткового уточнення результатів пошуку введено механізм бонусної оцінки на основі часткового збігу ключових слів із запиту користувача. З кожного запиту витягується до 8 ключових слів, які використовуються для обчислення оцінки часткового збігу (fuzzy match score) із текстом знайдених фрагментів. Отримані бали масштабуються за допомогою вагового коефіцієнта $w = 0{,}05$ та множаться на відповідний бал RRF, формуючи таким чином фінальну оцінку ранжування. Слід зауважити, що цей механізм відіграє роль вирішника неоднозначних ситуацій (tie-breaker) і суттєво впливає на результат лише у випадках, коли декілька фрагментів мають близькі значення RRF-балів.

4 ВЗАЄМОДІЯ КОРИСТУВАЧА З ЗАСТОСУНКОМ

У цьому розділі розглядається практична сторона використання розробленого застосунку: від встановлення необхідного програмного забезпечення до детального опису взаємодії користувача з інтерфейсом на кожному етапі роботи. Наведені інструкції охоплюють усі підтримувані платформи та конфігурації, а покроковий опис функціональних можливостей супроводжується ілюстраціями відповідних екранів програми.

4.1 Інструкція зі встановлення та запуску

Розроблений програмний застосунок є кросплатформним і підтримує роботу в операційних системах Linux, macOS та Windows. Обов’язковою передумовою для його функціонування є наявність встановленого інтерпретатора мови програмування Python версії не нижче 3.13.

Для користувачів, які мають намір задіяти апаратне прискорення обчислень за допомогою графічного процесора з підтримкою архітектури CUDA, необхідною є попередня інсталяція відповідних драйверів NVIDIA. Користувачам операційної системи Linux передбачено альтернативний спосіб розгортання середовища CUDA — через встановлення рантайму як окремого пакету Python, що спрощує процес налаштування та усуває необхідність у системній інсталяції драйверів. У складі репозиторію проєкту міститься скрипт export.sh, призначений для ініціалізації змінної середовища оболонки LD_LIBRARY_PATH. Зазначена змінна є необхідною для коректної роботи динамічного компонувальника, оскільки визначає шляхи до розташування бібліотек CUDA у файловій системі. Виконання цього скрипту є обов’язковим перед кожним запуском застосунку в сесіях, де відповідне середовище не було попередньо налаштовано.

Управління залежностями та запуск проєкту здійснюються засобами пакетного менеджера і менеджера проєктів uv [55]. Перед першим запуском

необхідно виконати синхронізацію залежностей за допомогою команди “uv sync” або “uv sunc --extra cuda” (опціонально для Linux), яка автоматично створює ізольоване віртуальне середовище та встановлює всі необхідні пакети відповідно до специфікації, зафіксованої у файлі блокування uv.lock. Після успішного завершення синхронізації застосунок готовий до використання.

4.2 Опис взаємодії

При запуску програми користувачу відображається початковий екран завантаження файлу, зображений на рис. 4.1. Програма підтримує три категорії вхідних даних: попередньо експортований файл у форматі Markdown, довільний текстовий файл, а також бінарний аудіо- або відеофайл, сумісний із бібліотекою FFmpeg. Кожна з категорій обробляється відповідно до своєї природи: для файлів Markdown виконується виключно побудова індексу семантичного пошуку без додаткової структуризації; для текстових файлів запускається повний пайплайн обробки; для медіафайлів перед структуризацією виконується автоматична транскрипція аудіодоріжки.

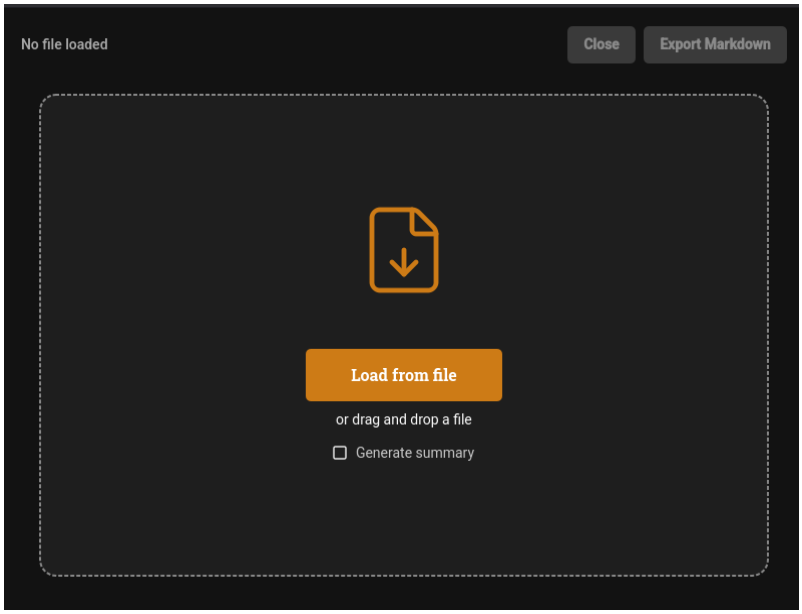

Рисунок 4.1 — Екран завантаження файлу

Завантаження файлу здійснюється двома способами: натисканням відповідної кнопки вибору файлу або перетягуванням файлу у визначену зону інтерфейсу (Drag & Drop), що продемонстровано на рис. 4.2. На цьому ж екрані користувач має можливість активувати опцію автоматичного генерування підсумку. Оскільки дана операція є обчислювально затратною та суттєво збільшує загальний час обробки, вона відключена за замовчуванням і виконується лише за явним запитом.

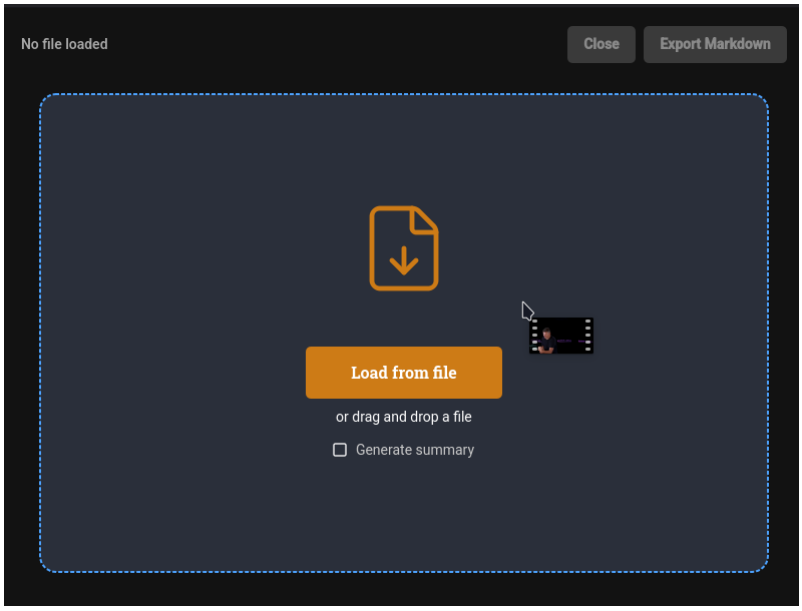

Рисунок 4.2 — Завантаження файлу методом Drag & Drop

Після успішного завантаження файлу програма автоматично переходить до головного робочого вікна, зображеного на рис. 4.3. На цьому етапі пайплайн обробки вже розпочав виконання: у відповідній панелі поступово відображається текст транскрипції в міру його формування. Над панеллю вкладок розташований індикатор поточного завдання, що дозволяє користувачу відстежувати прогрес виконання. Процес обробки відбувається у фоновому режимі, і інтерфейс залишається чутливим до введення. У разі необхідності передчасного завершення роботи передбачена кнопка «Close», що ініціює коректне переривання виконання пайплайну.

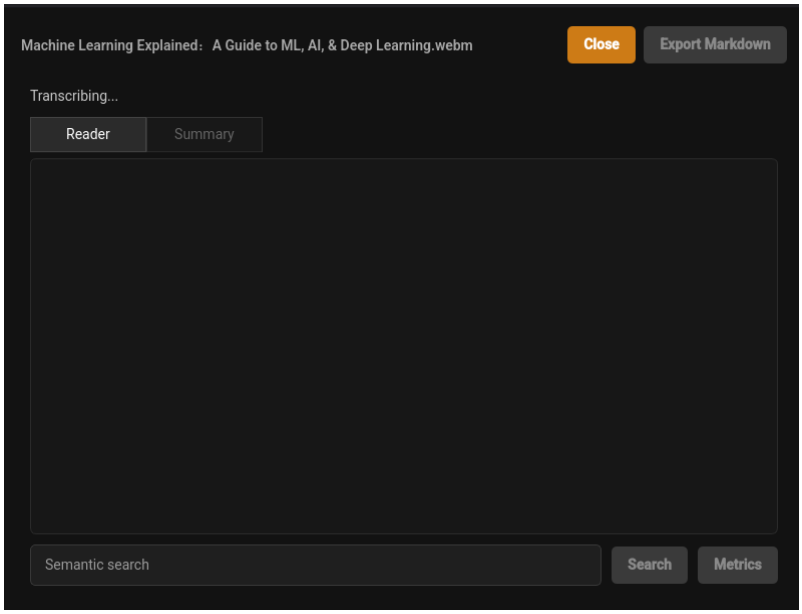

Рисунок 4.3 — Головне вікно програми під час обробки

Після завершення роботи пайплайну в головному вікні відображається фінальний результат структуризації тексту, що проілюстровано на рис. 4.4. На цьому етапі користувач може розпочати безпосередню роботу з отриманим структурованим текстом, або зберегти результат у зовнішній файл для подальшого використання. Експортований файл може бути повторно завантажений у програму, минаючи етап повторної обробки.

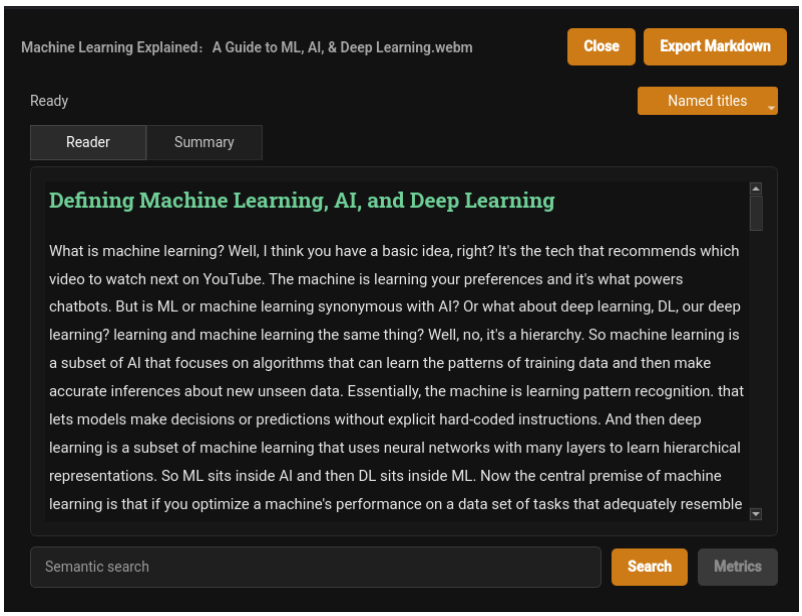

Рисунок 4.4 — Результат структуризації тексту

Для детального аналізу результатів програма надає можливість перегляду проміжних станів пайплайну через розкривний список, зображений на рис. 4.5. Передбачено три стадії відображення: початковий необроблений текст, сегментований текст із виділеними структурними межами, та текст з іменованими секціями після фінального розмічення. Зазначені пункти доступні для текстових і бінарних файлів, проте не відображаються для файлів формату Markdown, оскільки їх структура вважається заздалегідь визначеною.

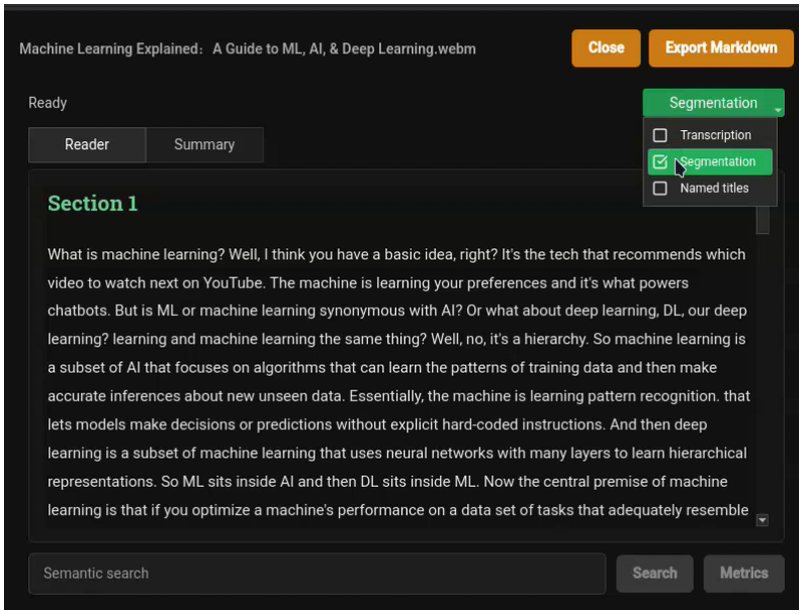

Рисунок 4.5 — Вибір стадії відображення результатів

Якщо під час завантаження файлу було активовано опцію генерування підсумку, його зміст стає доступним у вкладці «Summary», що продемонстровано на рис. 4.6. Підсумок формується алгоритмом автоматичної реферації та відображає стислий виклад ключових тез документа.

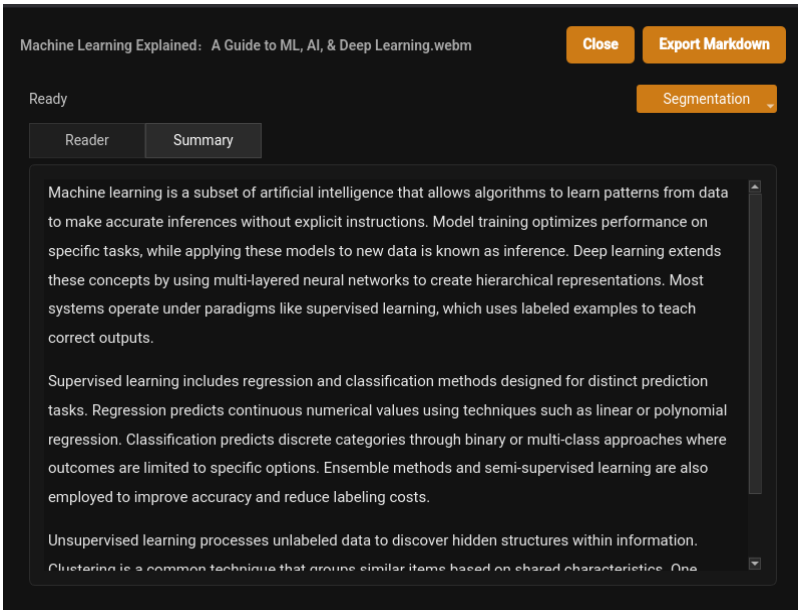

Рисунок 4.6 — Вкладка підведення підсумків

Повернувшись до вкладки «Reader», користувач отримує доступ до рядка пошуку, призначеного для виконання семантичного пошуку за змістом документа. На відміну від традиційного повнотекстового пошуку, семантичний підхід дозволяє формулювати запити природною мовою — у вигляді питань або тверджень — і знаходити змістово релевантні фрагменти незалежно від точного збігу лексики. Результати пошуку відображаються у вигляді виділених речень безпосередньо в тексті документа, що продемонстровано на рис. 4.7.

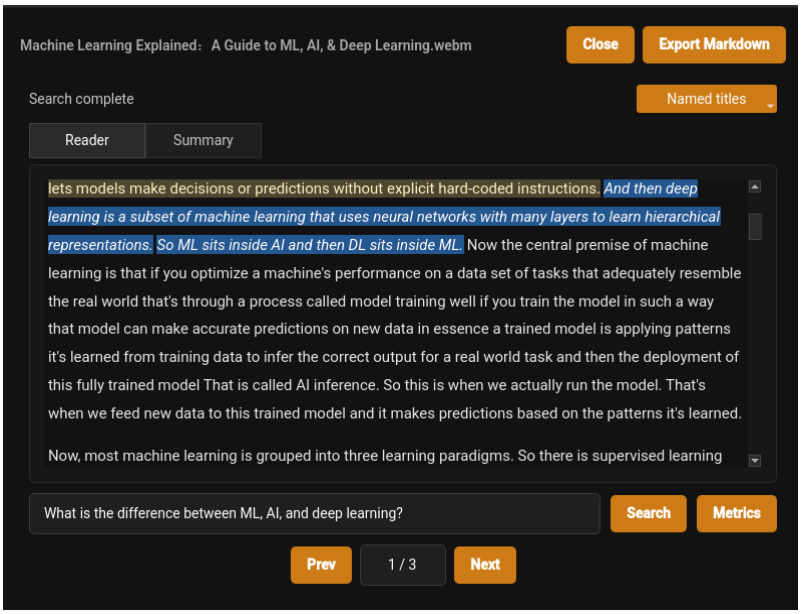

Рисунок 4.7 — Результат семантичного пошуку за запитом

За потреби користувач може ознайомитися з кількісними метриками пошукового алгоритму, натиснувши кнопку «Metrics». Відкриється діалогове вікно з детальними числовими показниками роботи алгоритмів пошуку для кожного результату, що зображено на рис. 4.8.

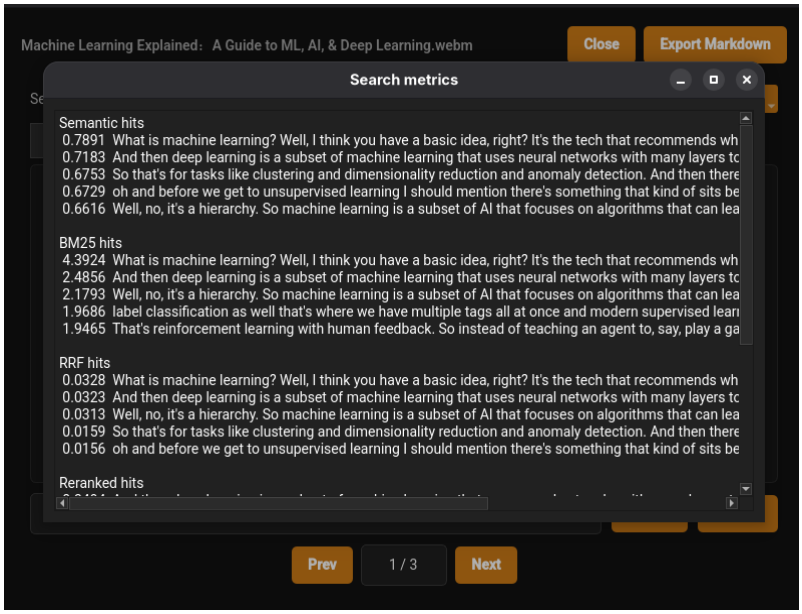

Рисунок 4.8 — Діалогове вікно метрик пошуку

Наведені метрики мають суто технічний характер і не призначені для інтерпретації кінцевим користувачем у звичайному режимі роботи. Проте, зважаючи на притаманну семантичному пошуку ймовірність хибнопозитивних результатів, можливість перегляду метрик надає досвідченому користувачу інструмент для самостійної оцінки релевантності та точності отриманих збігів.

ВИСНОВКИ

У рамках виконання дипломної роботи було сформульовано мету дослідження та визначено комплекс завдань, спрямованих на розробку інструменту для локального використання, який заповнює існуючу нішу на ринку систем транскрипції та семантичної структуризації тексту. У процесі роботи проведено порівняльний аналіз систем-аналогів, що надають суміжний функціонал, а також досліджено й систематизовано можливі архітектурні та технічні підходи до реалізації системи.

За підсумками проведеного дослідження прийнято низку обґрунтованих архітектурних і технічних рішень, що лягли в основу розробленої системи автоматичного розпізнавання мовлення з медіаконтенту та подальшої семантичної структуризації отриманого тексту на параграфи й абзаци. Цільовим апаратним середовищем виконання системи визначено персональний комп’ютер кінцевого користувача з актуальним апаратним забезпеченням. При цьому особливу увагу приділено мінімізації вимог до обчислювальних ресурсів: мінімально необхідний обсяг оперативної пам’яті становить близько 3 ГБ, тоді як для досягнення оптимальної якості результатів рекомендується не менше 7 ГБ. Саме ці обмеження ресурсів локальних пристроїв визначили ключові проєктні рішення та пріоритети при реалізації системи.

Упродовж розробки окремих модулів системи проводилося систематичне тестування з метою забезпечення достатнього рівня якості вихідних результатів та загальної стабільності роботи застосунку. Виявлені в ході тестування недоліки усувалися ітеративно, що дозволило досягти прийнятного рівня надійності програмного продукту.

Усі поставлені цілі й завдання дипломної роботи успішно виконані в повному обсязі. Розроблений програмний продукт є готовим до практичного використання та водночас має чіткий фундамент для подальшого розширення функціональних можливостей, масштабування та адаптації до нових вимог у майбутньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Stanford HAI. AI Index Report 2026: Technical Performance. – Stanford University, 2026. – URL: https://hai.stanford.edu/ai-index/2026-ai-index-report/technical-performance (дата звернення: 14.05.2026).
2. Vaswani A. та ін. Attention Is All You Need. – 2017. – URL: https://arxiv.org/abs/1706.03762 (дата звернення: 14.05.2026).
3. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. – 2018. – URL: https://arxiv.org/abs/1810.04805 (дата звернення: 14.05.2026).
4. Ray P. P. Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. – 2023. – URL: https://arxiv.org/abs/2305.10435 (дата звернення: 14.05.2026).
5. Salazar J., Liang D., Nguyen T. Q., Kirchhoff K. Adapting GPT, GPT-2 and BERT Language Models for Speech Recognition. – 2021. – URL: https://arxiv.org/abs/2108.07789 (дата звернення: 14.05.2026).
6. Alam T., Khan A., Alam F. Punctuation Restoration using Transformer Models for High-and Low-Resource Languages. Proceedings of the Sixth Workshop on Noisy User-generated Text (W-NUT 2020). – Online : Association for Computational Linguistics, 2020. – С. 132–142. – URL: https://aclanthology.org/2020.wnut-1.18/ (дата звернення: 14.05.2026).
7. Manning C. D., Raghavan P., Schütze H. Introduction to Information Retrieval. – Cambridge : Cambridge University Press, 2008. – 544 с.
8. Grootendorst M. KeyBERT. – 2020. – URL: https://github.com/MaartenGr/KeyBERT (дата звернення: 14.05.2026).
9. Campos R., Mangaravite V., Pasquali A., Jorge A., Nunes C., Jatowt A. YAKE! Keyword Extraction from Single Documents using Multiple Local Features. Advances in Information Retrieval. Lecture Notes in Computer Science. – Cham : Springer, 2018. – С. 797–804. – URL:

https://link.springer.com/chapter/10.1007/978-3-319-76941-7_80 (дата звернення: 14.05.2026).

10. Rose S. та ін. Automatic Keyword Extraction from Individual Documents. Text Mining: Applications and Theory / ed. M. W. Berry, J. Kogan. – Wiley, 2010. – URL: https://www.researchgate.net/publication/227988510_Automatic_Keyword_Extraction_from_Individual_Documents (дата звернення: 14.05.2026).
11. NVIDIA Developer Blog. Inference: The Next Step in GPU-Accelerated Deep Learning. – 2017. – URL: https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/ (дата звернення: 14.05.2026).
12. Python Software Foundation. Python : мова програмування. – 2025. – URL: https://www.python.org/ (дата звернення: 14.05.2026).
13. Behnel S., Bradshaw R., Citro C., Dalcin L., Seljebotn D. S., Smith K. Cython: The Best of Both Worlds. Computing in Science & Engineering. – 2011. – Vol. 13, No. 2. – C. 31–39. – URL: https://www.researchgate.net/publication/224176310_Cython_The_Best_of_Both_Worlds (дата звернення: 14.05.2026).
14. OpenAI. Whisper. – 2022. – URL: https://github.com/openai/whisper (дата звернення: 14.05.2026).
15. Radford A. та ін. Robust Speech Recognition via Large-Scale Weak Supervision. – 2022. – URL: https://arxiv.org/pdf/2212.04356 (дата звернення: 14.05.2026).
16. SYSTRAN. Faster-Whisper. – 2023. – URL: https://github.com/SYSTRAN/faster-whisper (дата звернення: 14.05.2026).
17. Gerganov G. whisper.cpp. – 2022. – URL: https://github.com/ggml-org/whisper.cpp (дата звернення: 14.05.2026).
18. Bain M. WhisperX. – 2023. – URL: https://github.com/m-bain/whisperx (дата звернення: 14.05.2026).
19. Klein G. та ін. Efficient and High-Quality Neural Machine Translation with OpenNMT. Proceedings of the 4th Workshop on Neural Generation and Translation (WNGT 2020). – Online : Association for Computational Linguistics, 2020. – C.

211–217. – URL: https://aclanthology.org/2020.ngt-1.25/ (дата звернення: 14.05.2026).
20. Lane H., Howard C., Hapke H. Natural Language Processing in Action. – Shelter Island : Manning Publications, 2019. – 544 с.
21. Honnibal M., Montani I., Van Landeghem S., Boyd A. spaCy: Industrial-strength Natural Language Processing in Python. – 2020. – URL: https://github.com/explosion/spaCy (дата звернення: 14.05.2026).
22. Bird S., Klein E., Loper E. Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit. – Beijing : O'Reilly Media, 2009. – 512 с.
23. Jurafsky D., Martin J. H. Speech and Language Processing. – 3rd ed. – Upper Saddle River : Prentice Hall, 2024. – URL: https://web.stanford.edu/~jurafsky/slp3/ (дата звернення: 14.05.2026).
24. Mikolov T., Chen K., Corrado G., Dean J. Efficient Estimation of Word Representations in Vector Space. – 2013. – URL: https://arxiv.org/abs/1301.3781 (дата звернення: 14.05.2026).
25. Nomic AI. nomic-embed-text-v2-moe : модель вбудовування тексту. – 2025. – URL: https://huggingface.co/nomic-ai/nomic-embed-text-v2-moe (дата звернення: 14.05.2026).
26. Nussbaum Z. та ін. Training Sparse Mixture Of Experts Text Embedding Models. – 2025. – URL: https://arxiv.org/abs/2502.07972 (дата звернення: 14.05.2026).
27. Jacobs R. A., Jordan M. I., Nowlan S. J., Hinton G. E. Adaptive Mixtures of Local Experts. Neural Computation. – 1991. – Vol. 3, No. 1. – С. 79–87. – URL: https://www.researchgate.net/publication/233806999_Adaptive_Mixtures_of_Local_Experts (дата звернення: 14.05.2026).
28. Kusupati A. та ін. Matryoshka Representation Learning. – 2022. – URL: https://arxiv.org/abs/2205.13147 (дата звернення: 14.05.2026).
29. Hugging Face. Transformers : документація бібліотеки. – 2025. – URL: https://huggingface.co/docs/transformers/en/index (дата звернення: 14.05.2026).
30. Reimers N. Sentence-Transformers : документація бібліотеки. – 2025. – URL: https://sbert.net/ (дата звернення: 14.05.2026).

31. Burkov A. The Hundred-Page Language Models Book. – Quebec City : Andriy Burkov, 2025. – 272 с.
32. Alammar J., Grootendorst M. Hands-On Large Language Models. – Sebastopol : O'Reilly Media, 2024. – 372 с.
33. Hugging Face. Open LLM Leaderboard. – 2025. – URL: https://huggingface.co/open-llm-leaderboard (дата звернення: 14.05.2026).
34. Qwen Team. Qwen3 Technical Report. – 2025. – URL: https://arxiv.org/abs/2505.09388 (дата звернення: 14.05.2026).
35. Ollama. Ollama : застосунок для локального запуску мовних моделей. – 2023. – URL: https://github.com/ollama/ollama (дата звернення: 14.05.2026).
36. Ollama. Ollama Documentation. – 2025. – URL: https://docs.ollama.com/ (дата звернення: 14.05.2026).
37. LM Studio. LM Studio Developer Documentation. – 2025. – URL: https://lmstudio.ai/docs/developer (дата звернення: 14.05.2026).
38. Gerganov G. llama.cpp. – 2023. – URL: https://github.com/ggml-org/llama.cpp (дата звернення: 14.05.2026).
39. vLLM Project. vLLM : бібліотека для швидкого виводу LLM. – 2023. – URL: https://github.com/vllm-project/vllm (дата звернення: 14.05.2026).
40. vLLM Project. vLLM Documentation. – 2025. – URL: https://docs.vllm.ai/en/latest/ (дата звернення: 14.05.2026).
41. Crochemore M., Lecroq T. Handbook of Exact String Matching Algorithms. – London : King's College Publications, 2004. – 544 с.
42. Navarro G., Raffinot M. Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. – Cambridge : Cambridge University Press, 2002. – 328 с.
43. Reimers N. Cross-Encoder Applications : документація Sentence-Transformers. – 2025. – URL: https://www.sbert.net/examples/cross_encoder/applications/README.html (дата звернення: 14.05.2026).
44. Luca X. H. bm25s. – 2024. – URL: https://github.com/xhluca/bm25s (дата звернення: 14.05.2026).

45. Luca X. H. BM25S: Orders of Magnitude Faster Lexical Search via Eager Sparse Scoring. – 2024. – URL: https://arxiv.org/abs/2407.03618 (дата звернення: 14.05.2026).
46. Jina AI. jina-reranker-v3 : модель реранжування. – 2025. – URL: https://huggingface.co/jinaai/jina-reranker-v3 (дата звернення: 14.05.2026).
47. Günther M., Mohr I., Wang B., Xiao H. jina-reranker-v3: Last but Not Late Interaction for Listwise Document Reranking. – 2025. – URL: https://arxiv.org/abs/2509.25085 (дата звернення: 14.05.2026).
48. INESCTEC. YAKE : бібліотека вилучення ключових слів. – 2019. – URL: https://github.com/INESCTEC/yake (дата звернення: 14.05.2026).
49. Campos R., Mangaravite V., Pasquali A., Jorge A., Nunes C., Jatowt A. YAKE! Keyword Extraction from Single Documents using Multiple Local Features. Information Processing & Management. – 2020. – URL: https://www.researchgate.net/publication/335766438_YAKE_Keyword_Extraction_from_Single_Documents_using_Multiple_Local_Features (дата звернення: 14.05.2026).
50. The Qt Company. Qt for Python (PySide6) Documentation. – 2025. – URL: https://doc.qt.io/qtforpython-6/ (дата звернення: 14.05.2026).
51. deepdml. whisper-large-v3-turbo : модель розпізнавання мовлення. – 2024. – URL: https://huggingface.co/deepdml/whisper-large-v3-turbo (дата звернення: 14.05.2026).
52. OpenWhispr. Whisper Model Sizes Explained. – 2024. – URL: https://openwhispr.com/blog/whisper-model-sizes-explained (дата звернення: 14.05.2026).
53. Paszke A. та ін. PyTorch: An Imperative Style, High-Performance Deep Learning Library. – 2019. – URL: https://arxiv.org/abs/1912.01703 (дата звернення: 14.05.2026).
54. PyTorch Foundation. PyTorch Documentation. – Version 2.12. – 2025. – URL: https://docs.pytorch.org/docs/2.12/index.html (дата звернення: 14.05.2026).
55. Astral. uv : менеджер пакетів Python. – 2025. – URL: https://docs.astral.sh/uv/ (дата звернення: 14.05.2026).

ДОДАТОК А

Реалізація основного пайплайну

```
class PipelineWorker(QtCore.QObject):
  status_changed = QtCore.Signal(object, str)
  segment_ready  = QtCore.Signal(object, str)
  stage_ready    = QtCore.Signal(object, object)
  document_ready = QtCore.Signal(object)
  failed         = QtCore.Signal(object, str)
  finished       = QtCore.Signal()

  def __init__(self, file_path: Path, generate_summary: bool = False):
    super().__init__()
    self.file_path = file_path
    self.generate_summary = generate_summary
    self._cancel_requested = False

  @QtCore.Slot()
  def cancel(self):
    self._cancel_requested = True

  def _raise_if_cancelled(self):
    if (
      self._cancel_requested or
      QtCore.QThread.currentThread().isInterruptionRequested()
    ):
      raise PipelineCancelled()

  @QtCore.Slot()
  def run(self):
```

```
  document = None
  try:
    self._raise_if_cancelled()

    suffix = self.file_path.suffix.lower()

    if suffix in {".md", ".markdown"}:
      document = self._load_markdown_file()
    elif not is_binary(self.file_path):
      document = self._load_text_file()
    else:
      document = self._load_media_file()

    self._raise_if_cancelled()

    self.document_ready.emit(document)
  except PipelineCancelled:
    if document is not None:
      document.release()

    cuda_cleanup()
  except Exception as exc:
    self.failed.emit(self.file_path, str(exc))
  finally:
    self.finished.emit()

def _load_media_file(self) -> DocumentModel:
  self._raise_if_cancelled()

  self.status_changed.emit(self.file_path, "Loading ASR model...")
  config = FasterWhisperConfig(
```

```
  model_size="large-v3-turbo",
  device=_DEVICE,
  compute_type=_COMPUTE_TYPE,
  batched=_IS_GPU,
)
params = FasterWhisperParams(
  beam_size=1,
  batch_size=_BATCH_SIZE,
)

asr_model = create_asr(config, params)

self._raise_if_cancelled()

transcript_parts: list[str] = []
language = None

try:
  self.status_changed.emit(self.file_path, "Transcribing...")
  stream, info = asr_model.transcribe_stream(
    audio_path=str(self.file_path),
    language=language,
  )
  language = info.language

  for segment in stream:
    self._raise_if_cancelled()

    text = segment.text.strip()

    if text:
```

```
        transcript_parts.append(text)
        self.segment_ready.emit(self.file_path, text)
    finally:
      asr_model.unload()

    transcript = " ".join(transcript_parts).strip()

    self._raise_if_cancelled()

    if not transcript:
      raise ValueError("ASR did not return any transcription text.")

    transcription_stage = DocumentStage(
      key="transcription",
      label="Transcription",
      sentences=[transcript],
      paragraphs_splits={0},
      sections_splits={0},
      sections_titles=[self.file_path.stem],
      markdown=transcript + "\n",
    )
    self.stage_ready.emit(self.file_path, transcription_stage)

    self._raise_if_cancelled()

    self.status_changed.emit(self.file_path, "Segmenting transcript...")
    nlp_language = normalize_language(language)
    if nlp_language == "en":
      nlp_language = detect_text_language(transcript)
    nlp_model = load_nlp(nlp_language)
    sentences = split_sentences(
```

```
  transcript,
  nlp_model,
  trim=True,
)

self._raise_if_cancelled()

with warnings.catch_warnings():
  warnings.simplefilter("ignore", UserWarning)
  embed_model = SentenceTransformer(
    "nomic-ai/nomic-embed-text-v2-moe",
    device=_DEVICE,
    trust_remote_code=True,
    model_kwargs={"torch_dtype": _DTYPE}
  )

self._raise_if_cancelled()

self.status_changed.emit(self.file_path, "Embedding sentences...")
embeddings_sentences = embed_model.encode(
  sentences,
  prompt_name="passage",
  show_progress_bar=False,
  convert_to_tensor=True,
  normalize_embeddings=True,
)

self._raise_if_cancelled()

self.status_changed.emit(self.file_path, "Finding paragraphs...")
paragraphs_chunks = split_by_similarity(
```

```
  len(sentences),
  embed_model,
  embeddings_sentences,
  sim_threshold=0.63,
  task_title="paragraphs",
)
paragraphs_splits = {
  start
  for start, _ in
  paragraphs_chunks
}

self._raise_if_cancelled()

self.status_changed.emit(self.file_path, "Finding sections...")
paragraphs_text = [
  " ".join(sentences[start:end])
  for start, end in
  paragraphs_chunks
]
embeddings_paragraphs = embed_model.encode(
  paragraphs_text,
  prompt_name="passage",
  show_progress_bar=False,
  convert_to_tensor=True,
  normalize_embeddings=True,
)

self._raise_if_cancelled()

sections_chunks = split_by_similarity(
```

```
  len(paragraphs_chunks),
  embed_model,
  embeddings_paragraphs,
  sim_threshold=0.75,
  task_title="sections",
)
sections_splits = {
  paragraphs_chunks[start][0]
  for start, _ in
  sections_chunks
}

self._raise_if_cancelled()

del embed_model
cuda_cleanup()

segmentation_stage = DocumentStage(
  key="segmentation",
  label="Segmentation",
  sentences=sentences,
  paragraphs_splits=paragraphs_splits,
  sections_splits=sections_splits,
  sections_titles=_section_placeholders(len(sections_splits)),
  markdown=_sentences_to_markdown(
    sentences,
    paragraphs_splits,
    sections_splits,
    _section_placeholders(len(sections_splits)),
  ),
)
```

```
            self.stage_ready.emit(self.file_path, segmentation_stage)

            self._raise_if_cancelled()

            sections_titles = self._build_section_titles(
              sentences,
              paragraphs_chunks,
              sections_chunks,
            )

            self._raise_if_cancelled()

            titles_stage = DocumentStage(
              key="named",
              label="Named titles",
              sentences=sentences,
              paragraphs_splits=paragraphs_splits,
              sections_splits=sections_splits,
              sections_titles=sections_titles,
              markdown=_sentences_to_markdown(
                sentences,
                paragraphs_splits,
                sections_splits,
                sections_titles,
              ),
            )
              summary = self._build_summary(titles_stage) if self.generate_summary else
None

            self._raise_if_cancelled()
```

```
return self._prepare_search(
  [transcription_stage, segmentation_stage, titles_stage],
  summary=summary,
  bm25_nlp_model=nlp_model,
  bm25_language=nlp_language,
)
```