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
“___”_____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “ Система реферування та семантичного аналізу відеоконтенту на основі мовних моделей ”

Виконав: студент 4 курсу, групи ТР-23
Дениско Віталій Миколайович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Тітов Владислав Миколайович _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент д. т. н. професор Черноусенко Ольга Юріївна _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент кафедри Артем Мельниченко _____
(посада, 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

Дениско Віталію Миколайовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Система реферування та семантичного аналізу відеоконтенту на основі мовних моделей”
Науковий керівник Тітов Владислав Миколайович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “_28_” _травня__ 2026 року № 1992 -с

2. Термін подання студентом роботи 04.06.2026

3. Вихідні дані до роботи мова програмування Python, фреймворк FastAPI для серверної частини, бібліотеки Pydantic, httpx, ffmpeg, whisper.cpp; React, Vite та TypeScript; СУБД SQLite; розгортання засобами Docker Compose; середовище розробки VS Code.

4. Перелік питань, які потрібно розробити _____

1) проаналізувати сучасні підходи, методи та програмні засоби для автоматичної транскрипції мовлення та семантичного реферування текстового контенту на основі мовних моделей;

2) визначити архітектуру та реалізувати програмну систему, яка забезпечує транскрипцію аудіо- та відеоматеріалів з таймкодами і генерацію структурованого конспекту засобами великих мовних моделей;

3) провести тестування розробленої системи для оцінки якості транскрипції та генерованих конспектів і визначити основні напрямки для її подальшого вдосконалення.

5. Орієнтований перелік ілюстративного матеріалу діаграма варіантів використання, діаграма компонентів системи, структура проєкту, схема взаємодії мікросервісів, блок-схема алгоритму транскрипції з поділом на частини, блок-схема генерації конспекту, графічний інтерфейс застосунку.

6. Дата видачі завдання 13.09.2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	15.01.26	виконано
2.	Аналіз методів та засобів розв’язання задачі	17-20.04.26	виконано
3.	Розробка архітектури та загальної структури системи	21-25.04.26	виконано
4.	Розробка окремих підсистем	26-02.05.26	виконано
5.	Програмна реалізація системи	03-07.05.26	виконано
6.	Оформлення пояснювальної записки	08-12.05.26	виконано
7.	Захист програмного забезпечення	14.05.26	виконано
8.	Передзахист	04.06.26	виконано
9.	Захист	17.06.26	виконано

Студент _____ Віталій ДЕНИСКО
(підпис) (прізвище та ініціали)

Керівник _____ Владислав ТІТОВ
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломну роботу виконано на 44 сторінках. Робота містить 10 ілюстрацій, 5 таблиць, 20 джерел у переліку посилань, 1 додаток.

Мета роботи — розробка веб-застосунку для автоматизованої обробки аудіо- та відеоматеріалів, який забезпечує транскрипцію мовлення з таймкодами та генерацію структурованого конспекту українською мовою на основі великих мовних моделей.

Підходи і засоби розробки: метод автоматичного розпізнавання мовлення (ASR) на основі моделей сімейства Whisper у реалізації whisper.cpp; поділ аудіо на частини з перекриттям (overlap) для обробки довгих файлів; генерація структурованого конспекту за допомогою великих мовних моделей (LLM) через Chat Completions API; мова програмування Python; фреймворк FastAPI для серверної частини та REST API; бібліотеки Pydantic для валідації схем, httpx для HTTP-взаємодії, ffmpeg і ffprobe для медіаобробки; клієнтська частина на React із використанням Vite та TypeScript; СУБД SQLite для збереження стану задач; розгортання за допомогою Docker Compose.

Ключові слова: автоматична транскрипція, розпізнавання мовлення, ASR, Whisper, whisper.cpp, великі мовні моделі, LLM, генерація конспектів, веб-застосунок, FastAPI, React, Docker, Python, ffmpeg, SQLite.

ANNOTATION

The thesis consists of 44 pages. The work contains 10 illustrations, 5 tables, 20 sources in the list of references, and 1 appendix.

The purpose of the work is to develop a web application for automated processing of audio and video materials, providing speech transcription with timestamps and generation of a structured summary in Ukrainian using large language models.

Development approaches and tools: automatic speech recognition (ASR) based on the Whisper model family via whisper.cpp; audio chunking with overlap for long-file processing; structured summary generation using large language models (LLM) through the Chat Completions API; Python programming language; FastAPI framework for the backend and REST API; Pydantic for schema validation, httpx for HTTP communication, ffmpeg and ffprobe for media processing; React with Vite and TypeScript for the frontend; SQLite for task state persistence; deployment with Docker Compose.

Keywords: automatic transcription, speech recognition, ASR, Whisper, whisper.cpp, large language models, LLM, summary generation, web application, FastAPI, React, Docker, Python, ffmpeg, SQLite.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

ASR (Automatic Speech Recognition) – Автоматичне розпізнавання мовлення.

LLM (Large Language Model) – Велика мовна модель.

API (Application Programming Interface) – Програмний інтерфейс застосунку.

REST (Representational State Transfer) – Архітектурний стиль побудови веб-API.

UI (User Interface) – Інтерфейс користувача.

FastAPI – Веб-фреймворк Python для побудови REST API з підтримкою асинхронного програмування.

React – JavaScript-бібліотека для побудови користувацьких інтерфейсів.

Vite – Інструмент збірки та середовище розробки для сучасних веб-застосунків.

Whisper – Багатомовна модель автоматичного розпізнавання мовлення, розроблена компанією OpenAI.

whisper.cpp – Ефективна C++-реалізація моделей Whisper з HTTP-сервером для локального запуску.

ffmpeg – Кросплатформний інструмент для конвертації, витягнення та обробки аудіо- і відеофайлів.

ffprobe – Утиліта з пакету ffmpeg для аналізу метаданих медіафайлів (тривалість, кодеки, формат).

SQLite – Легковага вбудована реляційна система управління базами даних.

Docker – Платформа контейнеризації для пакування та розгортання застосунків у ізольованих середовищах.

Docker Compose – Інструмент для визначення та запуску багатоконтейнерних застосунків.

Pydantic – Python-бібліотека для валідації та серіалізації даних на основі анотацій типів.

httpx – Асинхронний HTTP-клієнт для Python.

JSON (JavaScript Object Notation) – Текстовий формат обміну даними.

WAV (Waveform Audio File Format) – Формат аудіофайлу для зберігання нестисненого звуку.

UUID (Universally Unique Identifier) – Стандартизований унікальний ідентифікатор.

CSS (Cascading Style Sheets) – Каскадні таблиці стилів для оформлення веб-сторінок.

IDE (Integrated Development Environment) – Інтегроване середовище розробки.

ВСТУП

У сучасному інформаційному середовищі значна частина знань, навчальних матеріалів і професійних комунікацій існує у форматі аудіо- та відеозаписів. Лекції, семінари, наради, інтерв'ю, презентації, навчальні курси та робочі зустрічі часто зберігаються як довгі медіафайли. Такий формат є зручним для запису та повторного перегляду, однак має суттєвий недолік: для швидкого пошуку інформації, виділення головних тез або підготовки письмового матеріалу користувач змушений витрачати значний час на ручне прослуховування запису. Особливо помітною ця проблема стає для матеріалів тривалістю декілька годин, де потрібна не лише текстова розшифровка, а й логічно впорядкований конспект.

Автоматичне розпізнавання мовлення та генерація текстових узагальнень є одними з практичних напрямків застосування штучного інтелекту. Сучасні моделі ASR (Automatic Speech Recognition) дозволяють перетворювати мовлення на текст із достатньою точністю для навчальних і робочих сценаріїв. Водночас великі мовні моделі можуть аналізувати отриманий текст, виділяти ключові положення, групувати матеріал за темами, формувати висновки, списки термінів і практичні дії. Поєднання цих двох класів технологій дає змогу створити систему, яка не просто розшифровує запис, а перетворює його на структурований навчальний або робочий матеріал.

Метою дипломної роботи є розробка веб-застосунку для обробки довгих аудіо- та відеофайлів, який забезпечує завантаження медіафайлу через браузер, фонову обробку, автоматичну транскрипцію з таймкодами та генерацію структурованого конспекту українською мовою. Система орієнтована на практичне використання в освітньому та професійному середовищі, де важливо швидко отримати придатний до читання текстовий результат із довгого запису.

Для досягнення поставленої мети були сформульовані такі основні завдання:

1. Проаналізувати предметну область автоматичного розпізнавання мовлення, обробки довгих медіафайлів і генерації конспектів за допомогою LLM.

2. Визначити функціональні та нефункціональні вимоги до веб-застосунку, зокрема вимоги до завантаження файлів, фонового виконання, відстеження прогресу, збереження результатів і відновлення незавершених задач.

3. Спроектувати архітектуру системи, що складається з клієнтського React/Vite-інтерфейсу, FastAPI-серверу, SQLite-сховища задач, фонового worker-а, сервісу медіа-обробки, сервісу whisper.cpp та сервісу генерації конспекту.

4. Реалізувати серверний API для створення задач обробки, отримання статусу, читання транскрипту та конспекту.

5. Реалізувати медіа-пайплайн, який перевіряє тривалість файлу, витягує аудіодоріжку, нарізає її на частини, виконує транскрипцію кожного фрагмента та об’єднує сегменти без дублювання overlap-ділянок.

6. Реалізувати механізм генерації україномовного конспекту на основі транскрипту з валідацією очікуваної JSON-структури.

7. Розробити користувацький інтерфейс для завантаження файлу, перегляду прогресу, перемикання між режимами “Conspect” і “Transcripting”, а також відображення отриманих результатів.

8. Налаштувати контейнеризований запуск системи за допомогою Docker Compose.

9. Проаналізувати якість отриманого рішення, його обмеження та можливі напрямки подальшого розвитку.

Об’єктом дослідження є процес автоматизованої обробки аудіо- та відеоматеріалів. Предметом дослідження є методи та програмні засоби побудови веб-системи, яка поєднує медіаобробку, автоматичне розпізнавання мовлення та генерацію структурованого конспекту. Практичним результатом роботи є програмний прототип, який реалізує повний цикл обробки: від завантаження медіафайлу до отримання транскрипту і конспекту.

1 АВТОМАТИЗОВАНА ТРАНСКРИПЦІЯ ТА ГЕНЕРАЦІЯ КОНСПЕКТІВ ЯК ПРИКЛАД ШІ-СИСТЕМИ

Цей розділ присвячений теоретичним і прикладним аспектам задачі, що розв’язується в межах проекту. Розглядається актуальність автоматичної обробки довгих аудіо- та відеоматеріалів, описуються типові етапи перетворення медіафайлу на текстовий результат, а також аналізуються наявні підходи до транскрипції та генерації конспектів. Особливу увагу приділено практичним обмеженням, які виникають під час роботи з довгими файлами: розмір завантаження, тривалість обробки, необхідність фонового виконання, поділ аудіо на частини та узгодження результатів між різними компонентами системи.

1.1 Задача обробки довгих аудіо- та відеоматеріалів і її актуальність

Довгі аудіо- та відеозаписи є типовим джерелом інформації в освітньому, науковому та корпоративному середовищі. Лекційний курс може містити десятки годин записів, робоча зустріч може тривати декілька годин, а інтерв’ю або технічна консультація часто містить важливі деталі, які складно швидко знайти без текстового представлення. Ручне конспектування таких матеріалів потребує значного часу, уваги та повторного прослуховування окремих фрагментів. У результаті користувачі або витрачають багато ресурсів на обробку запису, або взагалі відмовляються від повноцінного аналізу матеріалу.

Автоматична транскрипція вирішує першу частину цієї проблеми: вона створює текстову версію мовлення. Наявність тексту спрощує пошук, цитування, редагування та подальшу обробку. Проте сам по собі повний транскрипт не завжди є зручним кінцевим результатом. Якщо запис триває дві або три години, текстова

розшифровка може містити десятки сторінок непідготовленого матеріалу, повтори, вставні слова, паузи, неповні речення та тематичні переходи. Для практичного використання потрібна друга стадія — змістове узагальнення.

Генерація конспекту дозволяє перетворити повний транскрипт на структурований документ. Такий документ може містити короткий огляд, ключові думки, тематичні розділи, перелік термінів і список практичних дій. Саме це робить систему корисною для користувача: він отримує не лише “сирий” текст, а підготовлений матеріал, придатний для навчання, повторення або прийняття рішень.

Актуальність розробки також пов’язана з поширенням гібридного навчання та дистанційної роботи. У таких умовах відеозаписи стають основним способом збереження знань. Водночас користувачі потребують інструментів, які дозволяють швидко опрацювати велику кількість матеріалів. Автоматизований веб-застосунок є зручним форматом для такої задачі, оскільки не вимагає від користувача знання командного рядка, ручного запуску ASR-моделей або самостійного налаштування складного пайплайну.

Окремою вимогою є підтримка довгих файлів. Багато простих демонстраційних ASR-рішень добре працюють з короткими аудіофрагментами, але погано масштабуються на записи тривалістю декілька годин. Для довгих матеріалів необхідно організувати чергу задач, фонову обробку, контроль максимальної тривалості, тимчасові робочі директорії, збереження проміжних результатів і можливість відновлення після перезапуску серверу. Саме ці особливості роблять задачу інженерно змістовною.

1.2 Етапи автоматичної обробки медіафайлів

Процес автоматичного перетворення медіафайлу на конспект можна подати як послідовність взаємопов’язаних етапів. Кожен етап має власні вхідні та вихідні дані, а також набір технічних ризиків.

Першим етапом є завантаження файлу користувачем. На цьому етапі система повинна прийняти файл через веб-інтерфейс, передати його на сервер, перевірити наявність імені, безпечно сформувати локальне ім’я файлу та проконтролювати максимальний розмір завантаження. Важливо, щоб сервер не блокувався на час подальшої обробки. Тому після збереження файлу створюється задача, яка ставиться в чергу, а користувач одразу отримує ідентифікатор задачі та початковий статус [20].

Другим етапом є підготовка медіафайлу до транскрипції. Оскільки ASR-модель зазвичай очікує аудіофайл певного формату, відеозапис необхідно перетворити на аудіо. У цьому проекті використовується mono WAV із частотою дискретизації 16 kHz і кодуванням PCM [7]. Такий формат є типовим для систем розпізнавання мовлення, оскільки зменшує надлишковість і забезпечує передбачувану якість вхідного сигналу.

Третім етапом є нарізка аудіо на частини. Довгий файл неефективно передавати в ASR-систему одним запитом: це може призвести до таймаутів, перевищення лімітів пам’яті або нестабільної якості результату. Тому аудіо розбивається на чанки фіксованої тривалості. Для зменшення ризику втрати слів на межі фрагментів використовується overlap, тобто невелике перекриття між сусідніми частинами. У проекті overlap становить 5 секунд.

Четвертим етапом є транскрипція кожного чанка. Сервер whisper.cpp приймає WAV-файл через HTTP-запит і повертає verbose JSON із сегментами. Кожен сегмент містить початок, кінець і текст. Оскільки модель повертає час відносно початку конкретного чанка, система додає зміщення chunk_start і

перетворює локальні таймкоди на абсолютні таймкоди в межах усього запису [1, 8].

П’ятим етапом є об’єднання сегментів. Через overlap на межах чанків можуть виникати дублікати. Наприклад, остання фраза попереднього фрагмента може повторитися на початку наступного. Для цього реалізується логіка порівняння сусідніх сегментів: текст нормалізується, після чого перевіряється повний збіг, входження одного тексту в кінець або початок іншого. Якщо сегмент визначено як дубль overlap-ділянки, він не додається повторно.

Шостим етапом є генерація конспекту. Повний транскрипт може бути занадто великим для одного запиту до LLM, тому система ділить текст на частини за максимальною кількістю символів. Межі чанків шукаються за природними розділювачами: подвійним переносом рядка, крапкою, знаком оклику, знаком питання, переносом або пробілом. Кожна частина обробляється за prompt-шаблоном, а якщо частин декілька, часткові конспекти об’єднуються в один фінальний JSON.

Сьомим етапом є відображення результатів користувачу. Після завершення обробки фронтенд отримує транскрипт і конспект окремими API-запитами. Користувач може перемикатися між двома режимами перегляду: детальним транскриптом із сегментами та компактним структурованим конспектом.

1.3 Огляд наявних підходів і програмних рішень для транскрипції та узагальнення

Існує декілька підходів до побудови систем автоматичної транскрипції. Перший підхід полягає у використанні хмарних сервісів розпізнавання мовлення. Такі сервіси зазвичай мають високу якість, масштабовану інфраструктуру та готові API. Їх недоліками можуть бути вартість, залежність від зовнішнього провайдера,

обмеження щодо конфіденційності даних і складність контролю над внутрішніми параметрами моделі.

Другий підхід полягає у локальному запуску моделей розпізнавання мовлення. Це дає більший контроль над даними та дозволяє розгортати систему в ізольованому середовищі. Одним із популярних напрямків є використання моделей сімейства Whisper [9]. Вони добре підходять для багатомовної транскрипції, а whisper.cpp надає ефективну реалізацію, яку можна запускати як окремий сервер. Такий підхід зручний для архітектури мікросервісного типу: основний бекенд не виконує ASR-логіку безпосередньо, а звертається до окремого HTTP-сервісу.

Третій підхід стосується узагальнення. Конспект можна створювати простими алгоритмами extractive summarization, які вибирають найважливіші речення з тексту, або за допомогою abstractive summarization, коли модель формує новий текст на основі змісту. У цьому проекті використовується другий підхід через LLM API, оскільки завдання полягає не лише в скороченні тексту, а у створенні структурованого JSON-результату з полями title, short_overview, key_points, sections, action_items і terms.

Таблиця 1.1 — Порівняння підходів до автоматичної транскрипції та конспектування.

Підхід	Переваги	Недоліки	Доцільність для проекту
Хмарний ASR API	Висока якість, простий старт, масштабованість	Вартість, зовнішня залежність, питання конфіденційності	Може бути альтернативою, але не дає повного контролю над ASR-сервісом

Продовження таблиці 1.1

Локальний whisper.cpp server	Контроль над даними, окремий сервіс, HTTP-інтеграція	Потрібні ресурси для запуску моделі, необхідне налаштування контейнера	Обрано для проекту як практичний баланс між якістю і контрольованістю
Просте текстове скорочення	Швидкість, передбачуваність	Не формує глибокої структури, може втрачати контекст	Недостатньо для цілі проекту
LLM-конспектування	Гнучка структура, україномовний результат, виділення тез і термінів	Залежність від API, потреба у валідації JSON, ризик помилок моделі	Обрано для генерації фінального конспекту
Готові SaaS-продукти	Зручність для кінцевого користувача	Обмежений контроль, залежність від сервісу, не демонструє власну реалізацію	Можуть бути референсом, але не замінюють розробку власної системи

Проведений аналіз показує, що для цілей переддипломної практики доцільним є створення власної системи, яка поєднує локальний або контейнеризований ASR-сервіс із керованим серверним пайплайном і LLM-компонентом для конспектування. Такий підхід дозволяє продемонструвати як навички веб-розробки, так і розуміння особливостей ШІ-інтеграції.

2 ПРОЕКТУВАННЯ СИСТЕМИ ТРАНСКРИПЦІЇ ТА УКРАЇНОМОВНОГО КОНСПЕКТУВАННЯ

У цьому розділі розглядається проектування програмної системи. Основну увагу приділено вимогам, архітектурі, життєвому циклу задачі та форматам даних. Система розроблялася як веб-застосунок, у якому користувач взаємодіє з браузерним інтерфейсом, а складна обробка виконується на сервері у фоновому режимі. Такий підхід дозволяє приховати від користувача технічну складність пайплайну та надати йому простий сценарій: завантажити файл, дочекатися завершення і переглянути результат.

2.1 Аналіз та формулювання вимог до системи

Функціональні вимоги визначають, які дії система повинна виконувати. Для даного проекту ключовими функціональними вимогами є такі:

1. Користувач повинен мати можливість завантажити аудіо- або відеофайл через веб-інтерфейс.
2. Система повинна створити окрему задачу обробки для кожного завантаження.
3. Користувач повинен бачити статус задачі: queued, processing, done або failed.
4. Користувач повинен бачити поточну стадію обробки: uploaded, extracting_audio, transcribing, summarizing, finished або failed.
5. Система повинна показувати приблизний прогрес виконання у відсотках.
6. Після завершення обробки система повинна надати повний транскрипт із таймкодами.

7. Після завершення обробки система повинна надати структурований конспект українською мовою.

8. Інтерфейс повинен дозволяти перемикатися між режимами перегляду транскрипту і конспекту.

9. Сервер повинен зберігати результати у файловій системі, щоб їх можна було отримати окремими API-запитами.

10. Після перезапуску бекенда система повинна відновлювати незавершені задачі, переводячи їх назад у чергу.

Нефункціональні вимоги визначають якісні характеристики системи. До них належать продуктивність, надійність, безпека, підтримуваність і зручність розгортання. Для даного проекту важливими є такі нефункціональні вимоги:

1. Система повинна обробляти великі файли без блокування HTTP-запиту на весь час транскрипції.

2. Максимальний розмір завантаження має бути обмежений налаштуванням, щоб уникнути неконтрольованого використання диску.

3. Максимальна тривалість відео має перевірятися перед повною обробкою.

4. Вхідне ім’я файлу повинно нормалізуватися, щоб уникнути небезпечних шляхів і некоректних символів.

5. Компоненти системи повинні бути розділені за відповідальністю: API, worker, repository, media service, whisper service, conspect service, frontend.

6. Система повинна мати конфігурацію через змінні середовища та `.env`-файли.

7. Розгортання має бути відтворюваним через Docker Compose.

8. Формати результатів мають бути валідовані через Pydantic-схеми.

Важливою проектною вимогою є асинхронність з точки зору користувацького сценарію. Транскрипція довгого файлу може тривати довго, тому API не повинен очікувати завершення всієї обробки в межах одного запиту. Замість цього використовується модель job-ів: запит POST /api/jobs створює задачу і

повертає її ідентифікатор, а фронтенд періодично опитує GET /api/jobs/{job_id} для отримання оновленого статусу [1].

2.2 Загальна архітектура програмного комплексу

Система складається з трьох основних частин: frontend, backend і whisper-сервіс. Такий поділ дозволяє незалежно розвивати користувацький інтерфейс, серверну бізнес-логіку та компонент розпізнавання мовлення.

Frontend реалізовано на React із використанням Vite. Він відповідає за взаємодію з користувачем: завантаження файлу, відображення поточного стану задачі, показ помилок, перемикання між режимами результату та рендеринг транскрипту або конспекту. Клієнтська частина не виконує медіаобробку, оскільки такі операції є ресурсомісткими та залежать від серверних інструментів [4, 5, 6].

Backend реалізовано на FastAPI. Він надає REST API, зберігає завантажені файли, створює записи задач у SQLite, запускає фоновий worker і керує повним пайплайном обробки. Backend містить декілька логічних шарів: API routes, dependencies, repository, domain models, schemas, services і workers. Така структура спрощує підтримку коду, оскільки кожен модуль має чітку відповідальність [1, 3].

Whisper-сервіс працює як окремий контейнер із whisper.cpp server. Backend надсилає йому WAV-чанки через HTTP-запит /inference, а у відповідь отримує JSON із сегментами. Винесення ASR-компонента в окремий сервіс має декілька переваг: його можна масштабувати окремо, замінити іншою реалізацією або запускати на іншій машині з потрібними обчислювальними ресурсами [9].

Загальний потік даних має такий вигляд:

1. Користувач відкриває веб-інтерфейс і вибирає файл.
2. Frontend надсилає файл на POST /api/jobs у форматі multipart/form-data.
3. Backend зберігає файл у data/uploads/<job_id>/, створює запис у SQLite та ставить job у чергу.

4. Worker отримує job_id і запускає PipelineService.process_job.

5. MediaService визначає тривалість файлу, витягує аудіо і нарізає його на чанки.

6. WhisperService надсилає чанки до whisper.cpp server і повертає список сегментів.

7. PipelineService об’єднує сегменти, формує transcript.json і зберігає його у data/results/<job_id>/.

8. ConspectService генерує структурований конспект через LLM API і зберігає conspect.json.

9. Backend оновлює статус задачі на done.

10. Frontend отримує результати через GET /api/jobs/{job_id}/transcript і GET /api/jobs/{job_id}/conspect.

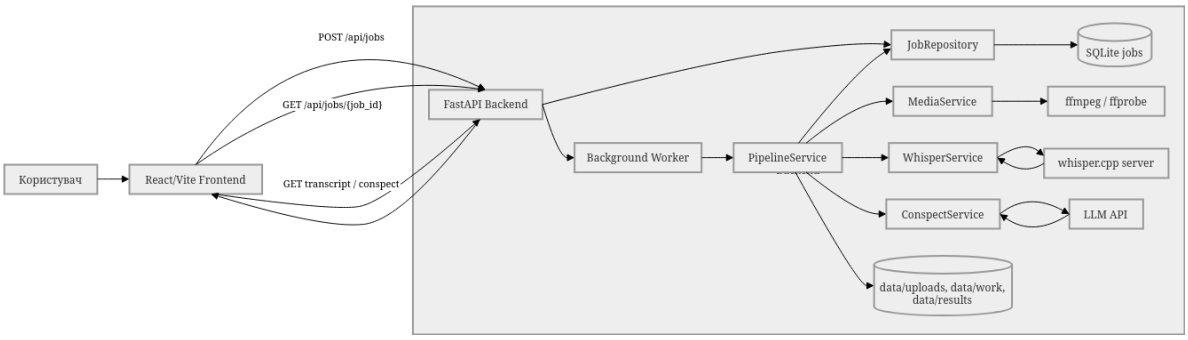

Рисунок 2.1 — Архітектура системи

Архітектура орієнтована на послідовну обробку задач у поточному прототипі. Worker використовує чергу і обробляє job-и у фоновому режимі. Такий підхід достатній для демонстраційного та навчального сценарію. У майбутньому його можна розширити до багатопроцесної або розподіленої обробки, якщо виникне потреба обробляти багато файлів одночасно.

2.3 Модель даних, життєвий цикл задачі та формати результатів

Центральною сутністю системи є задача обробки, або job. Кожна задача має унікальний ідентифікатор UUID, оригінальне ім’я файлу, шлях до завантаженого файлу, статус, стадію, прогрес, можливе повідомлення про помилку, час створення, час оновлення, а також шляхи до результатів. Збереження задач у SQLite дозволяє відокремити довготривалий стан системи від оперативної пам’яті процесу.

Життєвий цикл задачі починається зі статусу queued і стадії uploaded. Після старту пайплайну задача переходить у статус processing, а стадія змінюється відповідно до поточного етапу. Під час витягнення аудіо встановлюється стадія extracting_audio, під час транскрипції — transcribing, під час генерації конспекту — summarizing. Після успішного завершення статус змінюється на done, стадія — на finished, а прогрес — на 100. Якщо на будь-якому етапі виникає виняток, задача переходить у статус failed, стадію failed, а текст помилки зберігається в полі error.

Таблиця 2.1 — Статуси та стадії задачі.

Параметр	Можливі значення	Призначення
status	queued, processing, done, failed	Загальний стан задачі
stage	uploaded, extracting_audio, transcribing, summarizing, finished, failed	Поточний етап пайплайну

Продовження таблиці 2.1

progress	0-100	Орієнтовний відсоток виконання
error	null або текст помилки	Діагностика невдалого виконання

Формат транскрипту містить ідентифікатор задачі, мову, повний текст і масив сегментів. Кожен сегмент має початковий час, кінцевий час і текст. Такий формат зручний для двох сценаріїв: читання повного тексту та відображення транскрипту як часової послідовності.

Формат конспекту містить назву, короткий огляд, ключові думки, тематичні секції, практичні дії та терміни. Важливо, що результат LLM не використовується як довільний текст, а проходить валідацію через Pydantic-схему. Це зменшує ризик того, що фронтенд отримає структуру, яку не зможе коректно відобразити.

Приклад всього життєвого циклу програми можна побачити на рисунку 2.2.

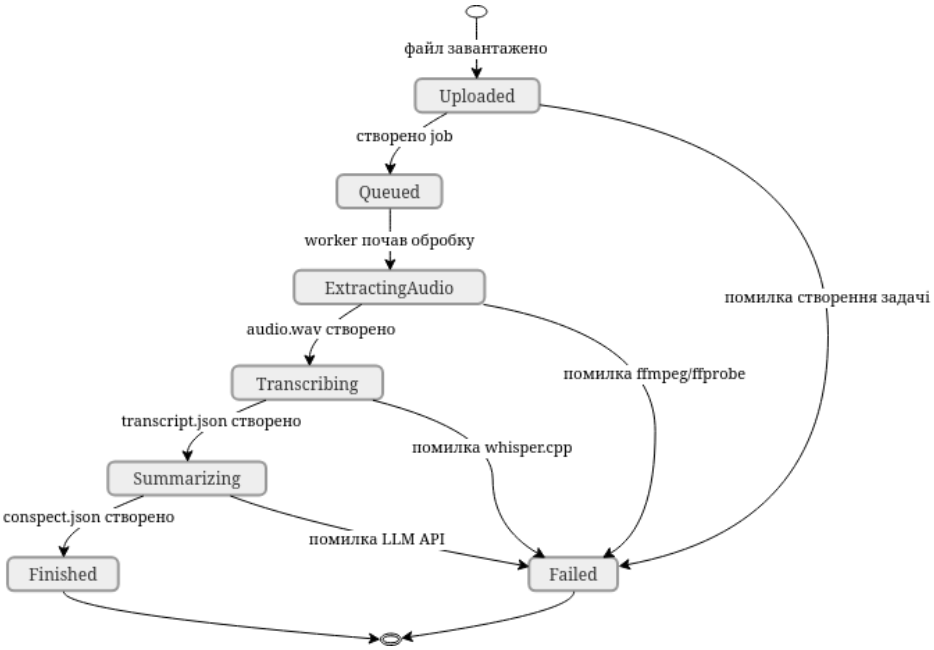

Рисунок 2.2 – Життєвий цикл програми

Обрані формати результатів забезпечують баланс між гнучкістю і передбачуваністю. Транскрипт залишається достатньо детальним, щоб користувач міг звернутися до конкретного моменту запису, а конспект надає узагальнений зміст без необхідності читати весь текст.

2.4 Вибір серверної платформи та мови програмування

Для кращого розуміння структури програми та вибір певних бібліотек було сформовано блок-схему, зображену на рисунку 3.1.

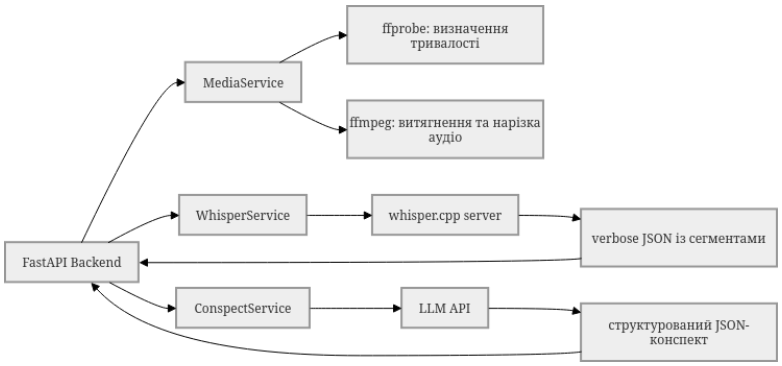

Рисунок 2.3 – Блок-схема архітектури програми

Для реалізації серверної частини було обрано мову Python і фреймворк FastAPI. Python є зручним вибором для систем, пов’язаних з обробкою даних, інтеграцією з ШІ-сервісами та швидкою розробкою прототипів. Його екосистема містить велику кількість бібліотек для веб-розробки, роботи з HTTP, валідації даних, медіаобробки та взаємодії з моделями машинного навчання [3].

FastAPI було обрано через його продуктивність, підтримку асинхронного програмування, інтеграцію з Pydantic і зручний опис REST API. У проекті FastAPI використовується для реалізації маршрутів створення задачі, отримання статусу, отримання транскрипту та отримання конспекту. Важливою перевагою є можливість явно описувати response_model, що підвищує передбачуваність API та полегшує інтеграцію з фронтендом [2].

Асинхронність у FastAPI використовується на рівні HTTP-обробників і взаємодії з фоновою чергою. Водночас ресурсомісткі або блокуючі операції, такі як робота з файлами, запуск ffmpeg або синхронний HTTP-запит до whisper-сервісу, виконуються через asyncio.to_thread. Це дозволяє не блокувати event loop і підтримувати коректну роботу веб-серверу.

Для збереження стану задач використовується SQLite. У межах прототипу SQLite є доцільним вибором, оскільки не потребує окремого серверу баз даних, легко розгортається разом із застосунком і достатньо добре підходить для збереження метаданих job-ів. У майбутньому, якщо система буде масштабуватися

на багато одночасних користувачів, SQLite можна замінити на PostgreSQL без зміни загальної архітектурної ідеї.

2.5 Засоби медіаобробки та автоматичного розпізнавання мовлення

Медіаобробка є ключовим етапом системи, оскільки якість підготовки аудіо безпосередньо впливає на якість транскрипції. Для роботи з аудіо- та відеофайлами обрано ffmpeg і ffprobe. Це стандартні інструменти, які широко використовуються для конвертації, аналізу та обробки медіа.

ffprobe використовується для визначення тривалості файлу. Це потрібно для перевірки обмеження max_video_minutes, яке захищає систему від надто довгих файлів. ffmpeg використовується для витягнення аудіодоріжки у формат mono WAV 16 kHz PCM, а також для нарізки аудіо на частини [8].

Для автоматичного розпізнавання мовлення використовується whisper.cpp server. Він приймає аудіофайли через HTTP і повертає JSON-результат. Такий формат інтеграції добре вписується в серверну архітектуру, оскільки основний бекенд не залежить від внутрішньої реалізації ASR-моделі. Йому достатньо знати URL whisper-сервісу і формат запиту.

Для генерації конспекту використовується LLM API. У конфігурації передбачено базовий URL, шлях chat completions, модель, API-ключ, timeout, максимальну кількість символів у chunk-у та кількість повторних спроб. Така конфігурація робить компонент гнучким: за потреби можна змінити модель або провайдера без переписування основної логіки [15, 16].

2.5.1 Бібліотеки FastAPI, Pydantic і httpx

FastAPI відповідає за серверні маршрути та життєвий цикл застосунку. У проекті використовується lifespan-функція, яка під час старту налаштовує логування, створює необхідні директорії, запускає worker і відновлює незавершені задачі. Під час завершення застосунку worker коректно скасовується.

Pydantic використовується для опису схем відповідей. Це важливо для API, оскільки фронтенд очікує конкретні поля: id, filename, status, stage, progress, error, created_at, updated_at, а також структури TranscriptResponse і ConspectResponse. Для конспекту Pydantic виконує додаткову роль: перевіряє, що LLM повернула результат у потрібному форматі.

httpx використовується для HTTP-взаємодії з whisper-сервісом і LLM API. Для whisper використовується синхронний клієнт із великим timeout, оскільки транскрипція чанка може бути тривалою. Для LLM використовується AsyncClient, що відповідає загальній асинхронній структурі ConspectService [14].

2.5.2 Інструменти ffmpeg та ffprobe

ffprobe викликається з параметрами, які повертають тривалість файлу у JSON-форматі. Сервер читає format.duration і перетворює його на число секунд. Це дозволяє прийняти рішення, чи можна обробляти файл відповідно до налаштування максимальної тривалості.

ffmpeg використовується у двох сценаріях. Перший — витягнення аудіо з відеофайлу. Команда встановлює один аудіоканал (-ac 1), частоту дискретизації 16000 Hz (-ar 16000), відключає відео (-vn) і задає PCM-кодек. Другий сценарій — нарізка аудіо на чанки. Для цього використовується початкова позиція -ss, тривалість -t і копіювання аудіопотоку [17].

Перевагою ffmpeg є стабільність і підтримка великої кількості форматів. Це дозволяє системі приймати не лише один конкретний тип файлу, а різні аудіо- та відеоформати, які ffmpeg може прочитати.

2.5.3 Сервер whisper.cpp

whisper.cpp є ефективною реалізацією моделей Whisper, орієнтованою на продуктивний локальний запуск. У проекті використовується режим серверу, що відкриває HTTP endpoint /inference. Backend передає файл як multipart/form-data, вказує response_format=verbose_json і мову з конфігурації. Значення whisper_language може бути auto, що дозволяє моделі самостійно визначати мову.

Отриманий verbose JSON містить масив сегментів. Кожен сегмент обробляється сервером: якщо в ньому є поля start і end, вони перетворюються на absolute_start і absolute_end з урахуванням позиції чанка в повному аудіо. Текст сегмента обрізається від зайвих пробілів. Порожні сегменти не додаються до фінального списку.

Такий підхід забезпечує збереження часової структури запису. Користувач отримує не просто суцільний текст, а набір фрагментів, які можна відобразити у вигляді транскрипту з таймкодами.

2.5.4 Великі мовні моделі для генерації конспекту

Великі мовні моделі використовуються для створення змістовного конспекту на основі транскрипту. На відміну від простого скорочення тексту, LLM може групувати матеріал за темами, переформульовувати речення, відокремлювати головне від другорядного та повертати структурований результат [19].

У проекті застосовується prompt-шаблон conspect_uk.txt, у який підставляється текст транскрипту та контекст частини. Якщо транскрипт не перевищує ліміт символів, LLM отримує його одним запитом. Якщо текст довший, він розбивається на декілька частин. Після цього кожна частина конспектується окремо, а фінальний запит об’єднує часткові конспекти в один JSON.

Важливою особливістю є вимога повертати лише валідний JSON без пояснень. Додатково реалізовано очищення відповіді від code fences, оскільки мовні моделі іноді повертають JSON у Markdown-блоці. Після цього відповідь парситься як JSON і передається на валідацію.

2.6 Вибір фронтенд-технологій, сховища даних і засобів розгортання

Клієнтська частина реалізована на React із використанням Vite. React зручний для побудови інтерактивного інтерфейсу, де стан залежить від асинхронних API-запитів. Vite забезпечує швидку розробку, просту конфігурацію та ефективну збірку.

У фронтенді основний компонент App керує станом задачі, транскрипту, конспекту, помилки, активного режиму перегляду та стану завантаження. Після відправлення файлу через uploadVideo застосунок отримує job і починає періодично опитувати статус через fetchJob. Коли статус стає done, фронтенд паралельно запитує transcript і conspect.

Інтерфейс розділений на декілька компонентів: UploadForm, JobStatus, TranscriptView і ConspectView. Такий поділ відповідає принципу єдиної відповідальності. Форма завантаження відповідає лише за вибір і відправлення файлу, картка статусу — за відображення прогресу, а компоненти результатів — за представлення отриманих даних.

Для розгортання використовується Docker Compose. Він описує три сервіси: whisper, backend і frontend. Сервіс backend залежить від whisper і отримує змінну WHISPER_SERVER_URL=http://whisper:8080, що дозволяє звертатися до ASR-сервісу за внутрішнім DNS-іменем Docker Compose. Директорія data монтується як volume, щоб завантаження, результати, база даних і логи зберігалися між перезапусками контейнерів. Frontend збирається з аргументом VITE_API_BASE_URL, щоб браузер міг звертатися до API за адресою http://localhost:8000.

Таблиця 2.2 — Основні технології проекту.

Компонент	Технологія	Призначення
Frontend	React, Vite, TypeScript	Інтерфейс завантаження, статусу та перегляду результатів
Backend	Python, FastAPI	REST API, запуск worker-а, керування пайплайном
Сховище	SQLite	Збереження стану задач
Медіаобробка	ffmpeg, ffprobe	Витягнення аудіо, визначення тривалості, нарізка чанків
ASR	whisper.cpp server	Автоматична транскрипція аудіо
LLM	Chat Completions API	Генерація україномовного конспекту
Розгортання	Docker Compose	Запуск frontend, backend і whisper як пов’язаної системи

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

Цей розділ присвячений безпосередній реалізації програмного продукту. Описується структура проекту, серверна логіка, фоновий пайплайн, інтеграція з whisper.cpp і LLM API, а також клієнтський інтерфейс. Реалізація побудована так, щоб основні частини системи були відокремлені одна від одної та могли розвиватися незалежно.

3.1 Структура програмного проекту

У backend-частині модуль api містить маршрути для роботи із задачами та healthcheck. Модуль core відповідає за конфігурацію, логування та створення необхідних директорій. Модуль domain описує внутрішні моделі та enum-и статусів. Модуль repositories ізолює доступ до SQLite. Модуль schemas описує зовнішні API-формати. Модуль services містить основну бізнес-логіку: медіаобробку, транскрипцію, конспектування і повний пайплайн. Модуль workers реалізує фонове виконання задач [11].

У frontend-частині api/client.ts інкапсулює HTTP-запити до backend. Це дозволяє компонентам не знати деталей URL-ів і форматів запитів. Компоненти UploadForm, JobStatus, TranscriptView і ConspectView відповідають за окремі частини інтерфейсу. Головний компонент App об’єднує їх і керує станом застосунку.

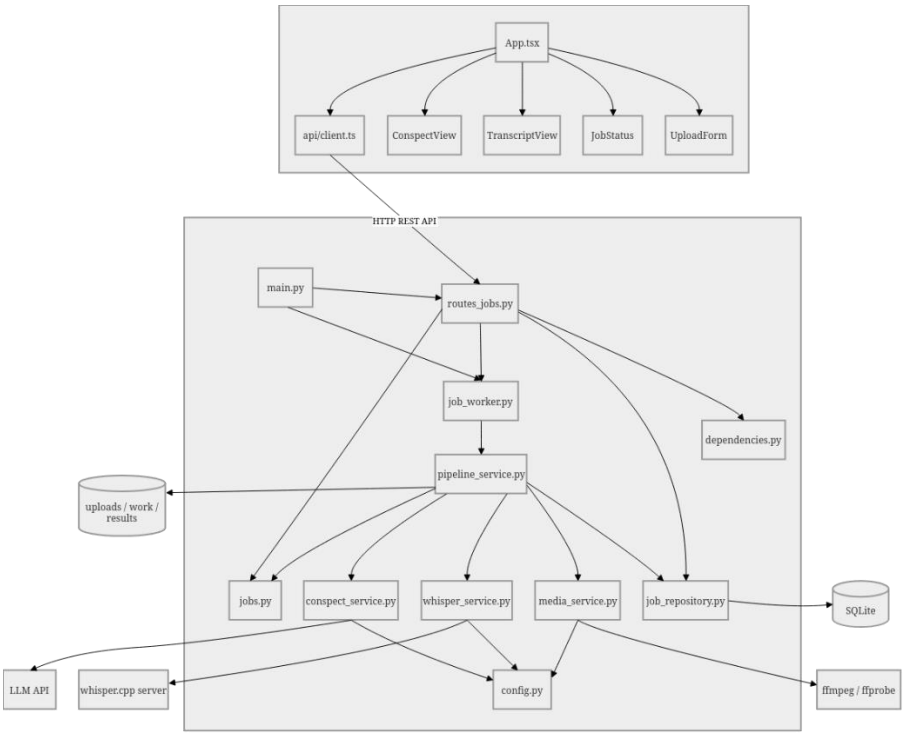

Рисунок 3.1 — Структура програмної системи та взаємодія модулів backend і frontend.

Саме ця структура і зображена на рисунку 3.1. Завдяки ній можна детальніше зрозуміти всю взаємодію між різними модулями внутрішньої та зовнішньої частини.

3.2 Реалізація серверної частини та фонового пайплайну

Серверна частина починає роботу з ініціалізації FastAPI-застосунку. У файлі main.py визначено lifespan-функцію, яка виконується під час старту і завершення застосунку. На старті налаштовується логування, створюються директорії, запускається worker і виконується відновлення незавершених задач. Якщо в SQLite є задачі, які не були завершені до перезапуску, вони переводяться у статус queued, стадію uploaded, прогрес 0 і повторно додаються в чергу.

API для задач реалізовано в routes_jobs.py. Маршрут POST /api/jobs приймає файл, перевіряє його ім’я, створює UUID, формує директорію завантаження, зберігає файл, створює запис у repository і додає задачу в worker. Відповідь повертається зі статусом HTTP 202 Accepted, що логічно відповідає асинхронній природі обробки: запит прийнято, але результат буде готовий пізніше.

Безпечна обробка імені файлу реалізована функцією _safe_upload_name. Вона нормалізує розділювачі шляху, бере лише leaf name, видаляє null-символи, замінює небезпечні символи на підкреслення та гарантує fallback upload.bin, якщо ім’я стало порожнім. Це важлива деталь, оскільки користувацьке ім’я файлу не можна безпосередньо використовувати як шлях у файловій системі.

Збереження файлу виконується потоково частинами по 1 МБ. Під час запису контролюється сумарний розмір. Якщо він перевищує max_upload_mb, частково записаний файл видаляється, а сервер повертає HTTP 413. Такий підхід захищає систему від надмірно великих завантажень.

Маршрут GET /api/jobs/{job_id} повертає актуальний стан задачі. Маршрути GET /api/jobs/{job_id}/transcript і GET /api/jobs/{job_id}/conspect читають відповідні JSON-файли після завершення обробки. Якщо шлях до результату ще не встановлено, API повертає 404 з повідомленням, що результат не готовий.

Основна логіка обробки зосереджена в PipelineService. Метод process_job виконує послідовність дій:

1. Отримує job з repository.
2. Формує робочу директорію та директорію результатів.
3. Оновлює статус на processing і стадію extracting_audio.
4. Визначає тривалість файлу через MediaService.probe_duration_seconds.
5. Перевіряє, чи не перевищує тривалість налаштований максимум.
6. Витягує WAV-аудіо через MediaService.extract_audio.
7. Нарізає аудіо на чанки через MediaService.split_audio.
8. Переводить задачу на стадію transcribing.
9. Виконує транскрипцію кожного чанка.

10. Формує і зберігає transcript.json.
11. Переводить задачу на стадію summarizing.
12. Генерує конспект через ConspectService.
13. Валідує конспект через ConspectPayload.
14. Зберігає conspect.json.
15. Оновлює шляхи результатів і переводить задачу в done.

Якщо на будь-якому кроці виникає помилка, вона перехоплюється, логуються деталі, а задача переводиться в failed. Такий механізм гарантує, що користувач не залишиться з “завислою” задачею без фінального стану.

Медіаобробка реалізована у MediaService. Метод probe_duration_seconds запускає ffprobe і читає JSON-відповідь. Метод extract_audio запускає ffmpeg для створення WAV-файлу. Метод split_audio визначає тривалість аудіо і в циклі створює чанки. Початок кожного наступного чанка збільшується на chunk_seconds - overlap_seconds, що забезпечує перекриття між фрагментами.

Транскрипція реалізована у WhisperService. Метод transcribe_chunk відкриває WAV-файл, формує multipart-запит до whisper server і передає параметри response_format=verbose_json та language. Отримана відповідь зберігається як проміжний JSON у робочій директорії. Це корисно для діагностики: якщо фінальний результат має проблему, можна перевірити відповідь ASR для конкретного чанка.

Після отримання segments сервіс перетворює локальні таймкоди на абсолютні. Наприклад, якщо chunk починається на 600-й секунді, а whisper повернув сегмент від 2.5 до 6.0 секунд, фінальний сегмент матиме start 602.5 і end 606.0. Це дозволяє коректно відображати час у межах усього запису.

Злиття сегментів виконується в PipelineService._merge_segments. Для кожного нового сегмента перевіряється, чи не дублює він останній уже доданий сегмент. Метод _is_duplicate_overlap порівнює нормалізований текст: застосовується casefold і розбиття/з’єднання пробілів. Якщо тексти збігаються або один є початком чи кінцем іншого, сегмент вважається дублем overlap-ділянки.

Генерація конспекту реалізована у ConspectService. Під час створення сервіс читає prompt-шаблон з файлу conspect_uk.txt. Метод build_conspect ділить транскрипт на частини відповідно до llm_chunk_chars. Якщо частина одна, результат LLM одразу повертається. Якщо частин декілька, кожна обробляється окремо, а потім формується merge prompt для об’єднання часткових результатів.

Взаємодія з LLM містить повторні спроби. Змінна llm_max_retries визначає, скільки разів система може повторити запит у разі помилки. Якщо всі спроби невдалі, сервіс генерує RuntimeError з описом останньої помилки. Такий підхід підвищує надійність при тимчасових мережевих збоях або нестабільності зовнішнього API [16].

Таблиця 3.1 — Стадії серверного пайплайну та відповідальні модулі.

Стадія	Модуль	Основна дія	Результат
Завантаження	routes_jobs.py	Прийом файлу, створення job	Запис у SQLite і файл у uploads
Витягнення аудіо	MediaService	ffprobe, ffmpeg	audio.wav
Нарізка	MediaService	Поділ на чанки з overlap	chunk_0000.wav, chunk_0001.wav тощо
Транскрипція	WhisperService	HTTP-запити до whisper.cpp	Список TranscriptSegment
Злиття	PipelineService	Видалення overlap-дублікатів	transcript.json
Конспект	ConspectService	Запити до LLM API	conspect.json
Завершення	JobRepository	Оновлення статусу і шляхів	Job у стані done

3.3 Реалізація клієнтського інтерфейсу

Блок-схему клієнтського інтерфейсу можна побачити на рисунку 4.2

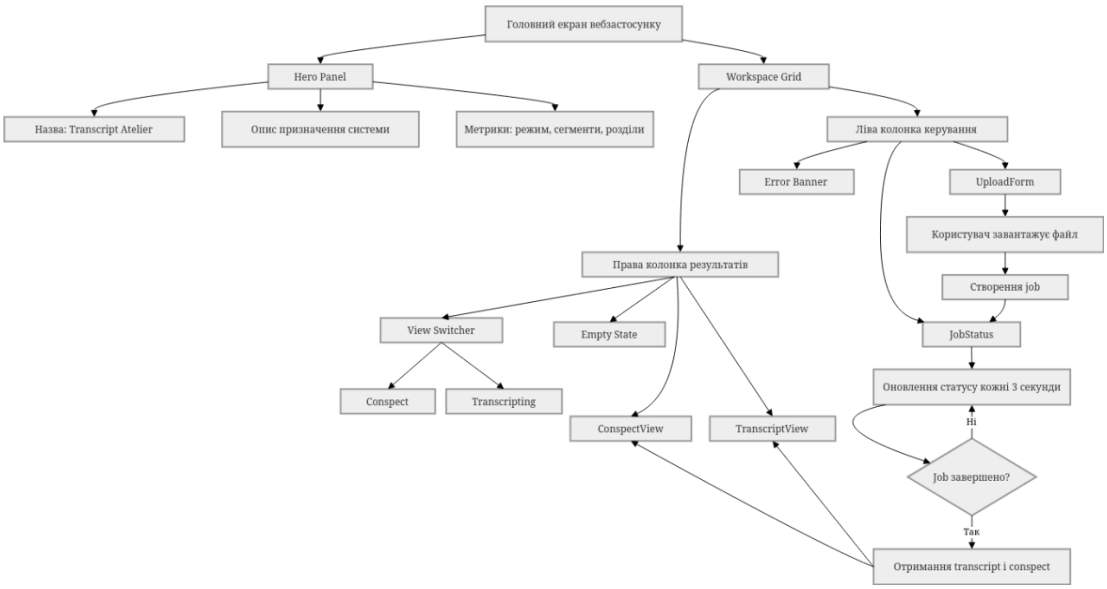

Рисунок 3.2 — Блок-схема головного екрану веб-застосунку

Клієнтський інтерфейс побудовано як односторінковий застосунок. Головний компонент App зберігає стан поточної задачі, транскрипту, конспекту, помилки, прапорця busy та активного режиму перегляду. Початковим режимом є conspect, але після завантаження файлу застосунок автоматично перемикається на transcripting, оскільки користувач очікує побачити процес і майбутній детальний результат.

Функція handleUpload відповідає за сценарій завантаження. Вона встановлює busy, очищає попередні помилки та результати, викликає uploadVideo, а після успішної відповіді зберігає job у стані компонента. Якщо виникає помилка, вона відображається в error banner.

Періодичне оновлення статусу реалізоване через useEffect і setInterval. Якщо job існує і ще не має статусу done або failed, фронтенд кожні 3 секунди викликає fetchJob. Це простий і достатній для прототипу механізм polling. У майбутньому його можна замінити на WebSocket або Server-Sent Events, якщо потрібне миттєве оновлення.

Другий useEffect реагує на перехід job у стан done. Після цього фронтенд паралельно викликає fetchTranscript і fetchConspect через Promise.all. Це оптимізує час очікування, оскільки обидва результати вже збережені на сервері й можуть бути отримані незалежно.

Інтерфейс має декілька логічних зон. Hero-panel містить короткий опис продукту та метрики: активний режим, кількість сегментів транскрипту і кількість розділів конспекту. Workspace-grid розділяє екран на колонку керування та колонку перегляду результатів. У першій колонці розташовані форма завантаження, повідомлення про помилку та статус задачі. У другій колонці розташований перемикач режимів і область результатів.

Перемикач режимів містить два варіанти: “Conspect” і “Transcripting”. Режим “Conspect” показує стислий підсумок, ключові думки, тематичні секції, action items і terms. Режим “Transcripting” показує повний перебіг запису по сегментах. Такий поділ відповідає різним потребам користувача: швидко зрозуміти зміст або детально перевірити конкретні фрагменти.

Компонентний підхід спрощує подальше розширення інтерфейсу. Наприклад, у майбутньому можна додати пошук по транскрипту, копіювання окремих блоків, експорт у Markdown або DOCX, фільтрацію action items, прив’язку сегментів до аудіоплеєра або показ confidence-метрик ASR.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

У цьому розділі описується практичне використання системи: вимоги до запуску, процедура розгортання, типові сценарії роботи користувача та перспективи вдосконалення. Оскільки система складається з декількох сервісів, особлива увага приділяється контейнерному запуску та конфігурації.

4.1 Системні вимоги та розгортання

Для запуску системи потрібне середовище, здатне виконувати Docker-контейнери. Рекомендовано використовувати сучасний багатоядерний процесор, достатній обсяг оперативної пам’яті та вільне місце на диску для збереження завантажених файлів, робочих аудіочанків і результатів. Конкретні вимоги залежать від розміру ASR-моделі, тривалості файлів і кількості одночасних задач.

Основні програмні вимоги:

1. Docker і Docker Compose для запуску сервісів.

2. Доступ до моделі або контейнера whisper.cpp server.

3. Наявність .env-файлу або змінних середовища для LLM API, зокрема LLM_API_KEY.

4. Відкриті локальні порти 5173 для frontend, 8000 для backend і 8080 для whisper server.

У Docker Compose описано три сервіси. Сервіс whisper збирається з whisper.Dockerfile і відкриває порт 8080. Сервіс backend збирається з backend/Dockerfile, читає .env.example і .env, отримує адресу whisper-сервісу через WHISPER_SERVER_URL=http://whisper:8080, відкриває порт 8000 і монтує директорію ./data:/app/data. Сервіс frontend збирається з frontend/Dockerfile,

отримує build argument VITE_API_BASE_URL=http://localhost:8000 і відкриває порт 5173.

Типовий порядок запуску:

1. Перевірити наявність Docker і Docker Compose.
2. Створити або оновити .env з ключем LLM API.
3. Запустити систему командою docker compose up --build.
4. Відкрити браузер за адресою http://localhost:5173.
5. Завантажити тестовий аудіо- або відеофайл.
6. Дочекатися завершення обробки та переглянути результати.

Успішний запуск програми можна побачити на рисунку 4.1.

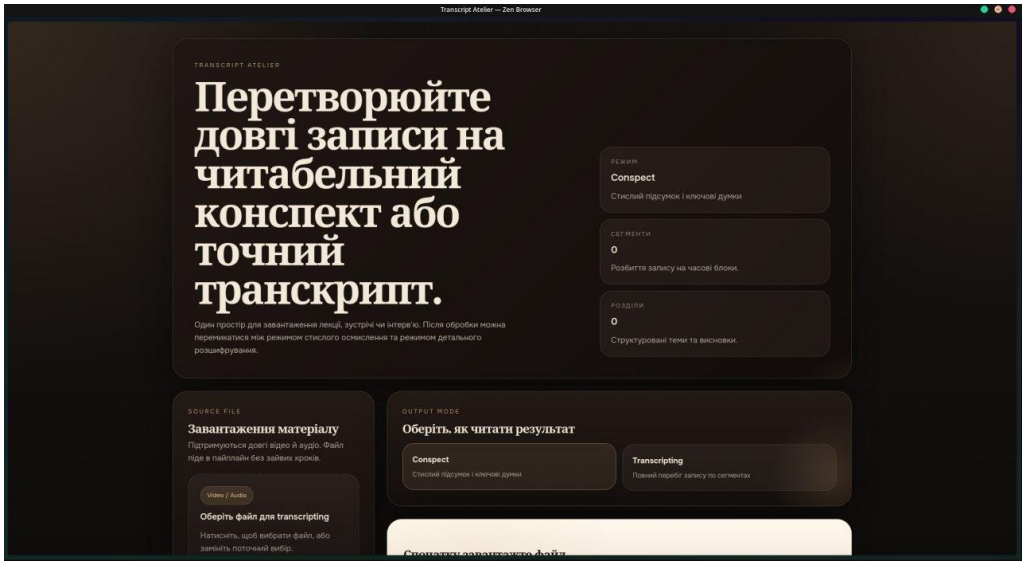

Рисунок 4.1 – Головна сторінка веб-застосунку

Для локальної розробки без Docker можна запускати frontend і backend окремо, але в такому випадку необхідно самостійно забезпечити доступність ffmpeg, ffprobe, whisper server і потрібних змінних середовища. Docker Compose є рекомендованим способом, оскільки він зменшує кількість ручних кроків і забезпечує відтворюваність середовища.

4.2 Основні сценарії використання

Сценарій 1: Завантаження файлу та створення задачі.

Користувач відкриває веб-інтерфейс, бачить головну сторінку із коротким описом системи та форму завантаження. Він вибирає аудіо- або відеофайл і натискає кнопку відправлення. Frontend формує FormData і надсилає файл на POST /api/jobs. Backend зберігає файл, створює job і повертає початковий стан. На екрані з’являється інформація про задачу: ідентифікатор, ім’я файлу, статус, стадія і прогрес.

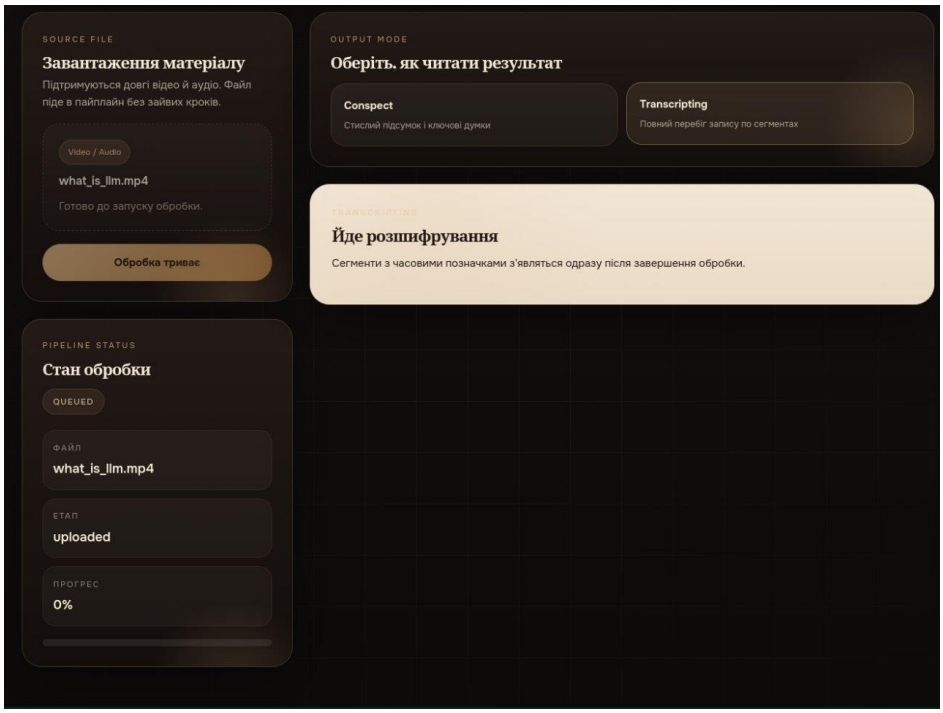

Рисунок 4.2 – Перший сценарій

Сценарій 2: Відстеження процесу обробки.

Після створення задачі frontend автоматично починає опитувати backend. Користувач бачить зміну стадій: спочатку файл завантажено, потім витягується аудіо, далі виконується транскрипція, після цього генерується конспект. Прогрес

оновлюється відповідно до етапів пайплайну. Якщо виникає помилка, статус змінюється на failed, а повідомлення про помилку відображається в інтерфейсі.

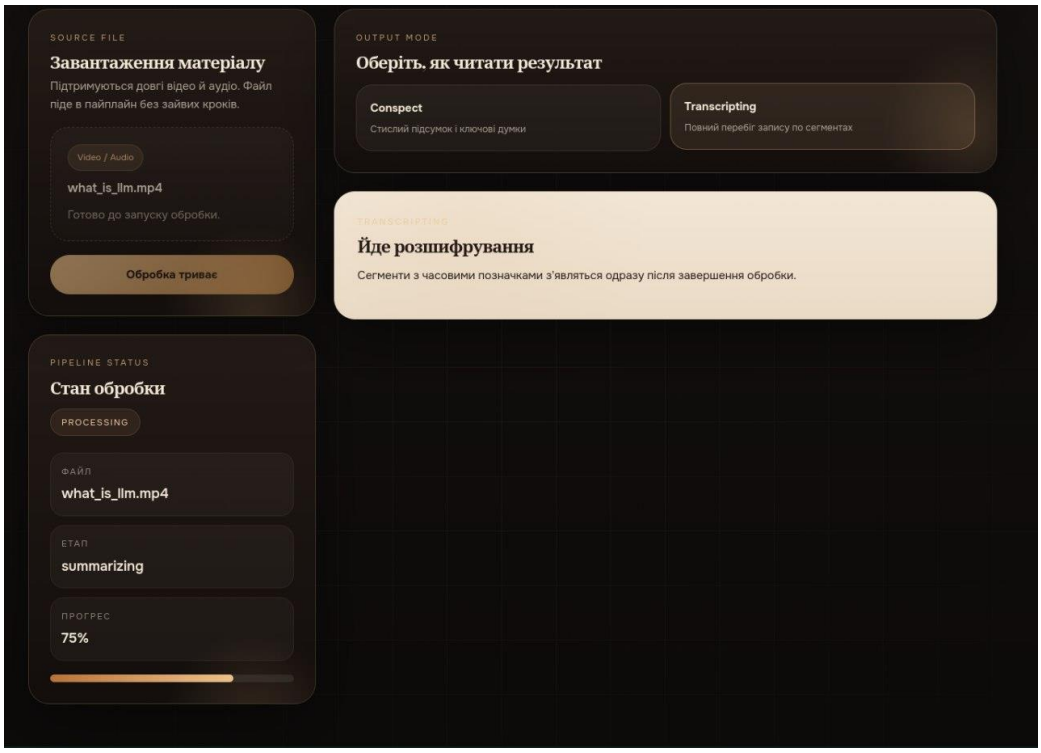

Рисунок 4.3 – Другий сценарій

Сценарій 3: Перегляд транскрипту.

Після завершення задачі користувач може перейти в режим “Transcripting”. У цьому режимі відображається повний перебіг запису по сегментах. Кожен сегмент має часові межі та текст. Такий перегляд корисний, якщо потрібно знайти конкретний момент у записі або перевірити, як саме модель розпізнала певну фразу.

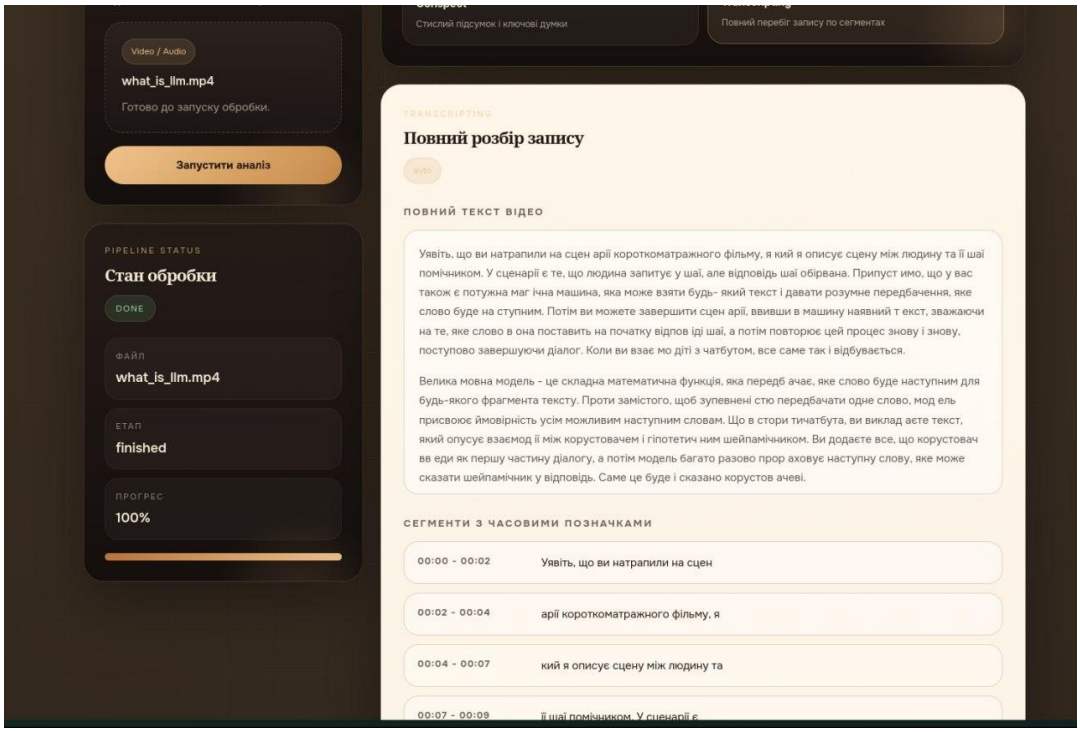

Рисунок 4.4 – Третій сценарій

Сценарій 4: Перегляд конспекту.

У режимі “Conspect” користувач бачить підготовлений структурований матеріал. Він містить назву, короткий огляд, ключові думки, тематичні розділи, практичні дії та терміни. Цей режим корисний для швидкого ознайомлення зі змістом лекції, зустрічі або інтерв’ю без читання повного транскрипту.

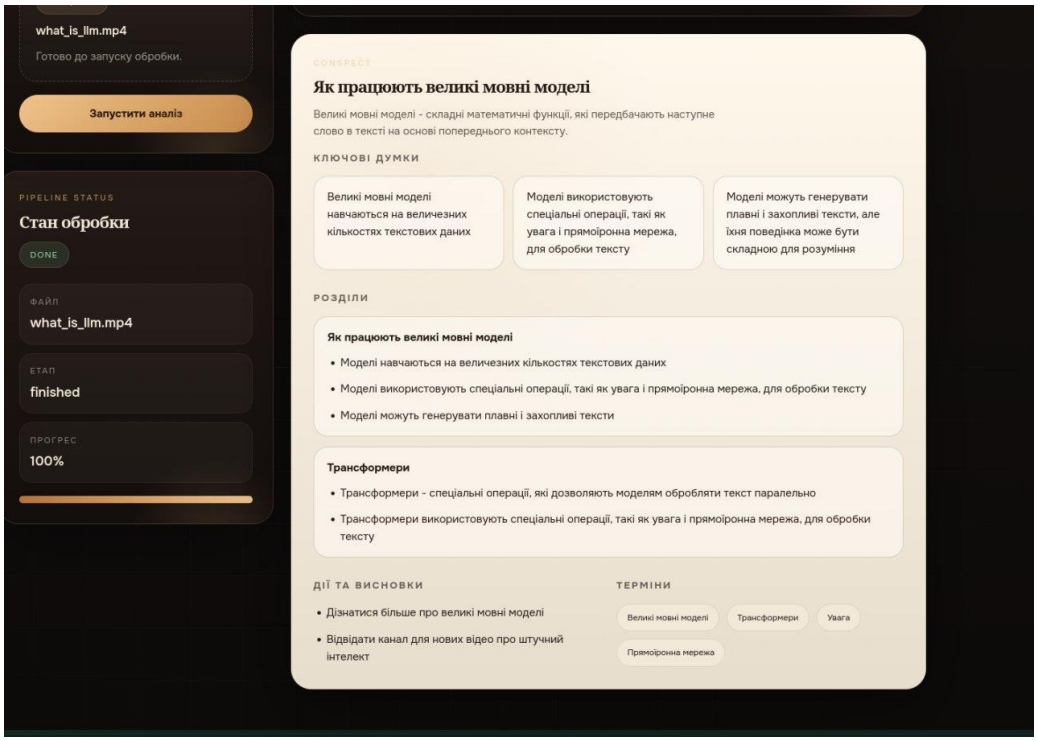

Рисунок 4.5 – Четвертий сценарій

Сценарій 5: Відновлення після перезапуску backend.

Якщо backend було перезапущено під час обробки, при наступному старті система знаходить незавершені задачі в SQLite, переводить їх у чергу та запускає повторно. Це не є повноцінним продовженням з останнього чанка, але забезпечує, що задача не залишиться назавжди у проміжному стані. Для прототипу такий механізм є достатнім і значно підвищує надійність.

Таблиця 4.1 — Типові сценарії та очікуваний результат.

Сценарій	Дія користувача	Очікуваний результат
Завантаження	Вибір файлу і відправлення	Створено job зі статусом queued
Очікування	Автоматичне оновлення статусу	Видно стадію і прогрес
Транскрипт	Перемикання на Transcripting	Відображено сегменти з таймкодами

Продовження таблиці 4.1

Конспект	Перемикання на Conspect	Відображено структурований конспект
Помилка	Некоректний файл або збій API	Показано статус failed і текст помилки

4.3 Напрямки підвищення якості та подальшого розвитку системи

Розроблена система є функціональним прототипом, але має низку напрямків для подальшого вдосконалення. Першим напрямком є покращення механізму черги. Поточний worker обробляє задачі послідовно в межах процесу backend. Для production-сценарію доцільно використати окрему систему черг, наприклад Celery, RQ або Arq, а також винести worker в окремий процес або контейнер. Це дозволить масштабувати обробку незалежно від API [20].

Другим напрямком є покращення відновлення після збоїв. Зараз незавершені задачі після перезапуску ставляться в чергу з початку. У майбутньому можна зберігати прогрес на рівні чанків і продовжувати обробку з останнього успішного фрагмента. Це особливо важливо для дуже довгих файлів, де повторна обробка з початку може бути дорогою.

Третім напрямком є підвищення якості транскрипції. Можна додати вибір мови, вибір моделі, попередню нормалізацію аудіо, шумозаглушення, voice activity detection і постобробку пунктуації. Також доцільно зберігати додаткові метрики, якщо ASR-сервіс їх надає, наприклад confidence або ймовірність мови.

Четвертим напрямком є покращення якості конспекту. Можна реалізувати декілька типів конспектів: короткий, детальний, навчальний, протокол зустрічі,

action-oriented summary. Також можна додати вибір мови конспекту, налаштування стилю, експорт у Markdown, DOCX або PDF. Важливим покращенням є автоматична перевірка того, що LLM не додала тверджень, яких не було у транскрипті.

П’ятим напрямком є розвиток інтерфейсу. Корисними функціями можуть бути пошук по транскрипту, клік по сегменту з переходом до відповідного часу в аудіоплеєрі, копіювання окремих блоків, редагування конспекту, історія задач і видалення старих результатів. Для великих транскриптів доцільно додати віртуалізацію списку сегментів, щоб інтерфейс залишався швидким.

Шостим напрямком є безпека та контроль доступу. Поточний прототип не містить автентифікації. Для реального використання слід додати користувачів, обмеження доступу до задач, політику зберігання файлів, автоматичне очищення старих даних і захист API від надмірної кількості запитів.

Сьомим напрямком є тестування та моніторинг. Доцільно розширити unit-тести для медіасервісу, логіки злиття сегментів, chunking-алгоритму для LLM і API-маршрутів. Також варто додати інтеграційні тести для повного сценарію з невеликим аудіофайлом. У production-середовищі потрібні метрики тривалості обробки, кількості помилок, середнього часу транскрипції чанка і вартості LLM-запитів.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено та реалізовано програмний веб-застосунку для автоматичної транскрипції довгих аудіо- та відеофайлів і генерації структурованого україномовного конспекту. Актуальність роботи зумовлена зростанням кількості навчальних, робочих і комунікаційних матеріалів у форматі довгих записів, які складно швидко опрацьовувати вручну.

На першому етапі було проаналізовано предметну область автоматичного розпізнавання мовлення та генерації текстових узагальнень. Було визначено, що практично корисна система повинна надавати не лише повний транскрипт, а й структурований конспект, який дозволяє швидко зрозуміти зміст запису. Також було розглянуто особливості обробки довгих файлів: необхідність фонового виконання, нарізки аудіо на частини, збереження проміжних результатів і контролю стану задачі.

На другому етапі було спроектовано архітектуру системи. Вона складається з React/Vite frontend, FastAPI backend, SQLite-сховища задач, фонового worker-а, сервісу медіаобробки, whisper.cpp server і LLM-компонента для конспектування. Такий поділ дозволив відокремити користувацький інтерфейс, серверну бізнес-логіку, ASR-компонент і генерацію конспекту. Також було обґрунтовано вибір технологій. Python і FastAPI були обрані для серверної частини через швидкість розробки, підтримку асинхронності та зручну інтеграцію з Pydantic. ffmpeg і ffprobe використано для медіаобробки. whisper.cpp server застосовано для транскрипції аудіочанків. LLM API використано для формування структурованого конспекту українською мовою. React і Vite забезпечили створення інтерактивного клієнтського інтерфейсу, а Docker Compose — відтворюване розгортання всіх сервісів.

На третьому етапі було реалізовано серверну частину. API підтримує створення задачі, отримання статусу, отримання транскрипту та отримання конспекту. Реалізовано безпечне збереження завантажених файлів, контроль

максимального розміру, нормалізацію імені файлу, фонову чергу, відновлення незавершених задач після перезапуску та оновлення прогресу. Пайплайн обробки виконує перевірку тривалості, витягнення WAV-аудіо, нарізку на чанки з overlap, транскрипцію через whisper.cpp, злиття сегментів і генерацію конспекту через LLM.

На четвертому етапі було реалізовано клієнтський інтерфейс. Користувач може завантажити файл, спостерігати за статусом обробки, бачити помилки та після завершення переглядати результати у двох режимах: “Transcripting” для повного транскрипту з сегментами та “Conspect” для стислого структурованого конспекту. Інтерфейс побудовано компонентно, що спрощує подальший розвиток.

У результаті було створено працездатний прототип системи, який демонструє повний цикл автоматизованої обробки медіафайлу: від завантаження до отримання структурованого текстового результату. Всі основні завдання практики були виконані. Розроблене рішення має практичне значення для навчальних і робочих сценаріїв, де потрібно швидко перетворювати довгі записи на зручні для читання матеріали.

Подальший розвиток системи може включати масштабовану чергу задач, продовження обробки з останнього успішного чанка, покращення якості транскрипції, різні режими конспектування, експорт результатів у DOCX/PDF, пошук по транскрипту, автентифікацію користувачів, політики зберігання даних і розширене тестове покриття. Отриманий під час практики досвід є цінним для подальшої професійної діяльності у сфері веб-розробки, інтеграції ШІ-сервісів та побудови прикладних інформаційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 25.05.2026).
2. Pydantic Documentation. URL: https://docs.pydantic.dev/ (дата звернення: 25.05.2026).
3. Python Language Reference / Python Software Foundation. URL: https://docs.python.org/3/reference/ (дата звернення: 25.05.2026).
4. React Documentation. URL: https://react.dev/ (дата звернення: 25.05.2026).
5. Vite Documentation. URL: https://vite.dev/ (дата звернення: 25.05.2026).
6. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 25.05.2026).
7. FFmpeg Documentation. URL: https://ffmpeg.org/documentation.html (дата звернення: 25.05.2026).
8. FFprobe Documentation. URL: https://ffmpeg.org/ffprobe.html (дата звернення: 25.05.2026).
9. whisper.cpp repository / Georgi Gerganov. URL: https://github.com/ggerganov/whisper.cpp (дата звернення: 25.05.2026).
10. Radford A., Kim J. W., Xu T., Brockman G., McLeavey C., Sutskever I. Robust Speech Recognition via Large-Scale Weak Supervision. 2022.
11. SQLite Documentation. URL: https://www.sqlite.org/docs.html (дата звернення: 25.05.2026).
12. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 25.05.2026).
13. Docker Compose Documentation. URL: https://docs.docker.com/compose/ (дата звернення: 25.05.2026).
14. HTTPX Documentation. URL: https://www.python-httpx.org/ (дата звернення: 25.05.2026).

15. OpenAI. Chat Completions API Documentation. URL: https://platform.openai.com/docs/ (дата звернення: 25.05.2026).

16. NVIDIA API Catalog Documentation. URL: https://docs.api.nvidia.com/ (дата звернення: 25.05.2026).

17. Jurafsky D., Martin J. H. Speech and Language Processing. 3rd ed.

18. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016.

19. Vaswani A. et al. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017.

20. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine, 2000.

ДОДАТОК А

СЕРВЕРНА ЛОГІКА ТРАНСКРИПЦІЇ ТА АНАЛІЗУ

Текст програми

УКР.НТУУ”КПІ ім. Ігоря Сікорського”

Аркушів 16

Київ 2026

Програмні засоби

— мова програмування – Python;
— фреймворк для розробки серверної частини та REST API – FastAPI;
— бібліотека для валідації та серіалізації даних – Pydantic;
— бібліотека для асинхронних HTTP-запитів – httpx;
— рушій автоматичного розпізнавання мовлення – whisper.cpp;
— інструмент для обробки та конвертації медіафайлів – ffmpeg;
— система управління базами даних – SQLite;
— бібліотека для роботи з базою даних – aiosqlite;
— середовище розробки – VS Code;

```
import json
from pathlib import Path

import httpx

from backend.app.core.config import settings



class ConspectService:
    def __init__(self) -> None:
        self._prompt_template = (
            Path(__file__).resolve().parent.parent / "prompts" / "conspect_uk.txt"
        ).read_text(encoding="utf-8")

    async def build_conspect(self, transcript_text: str) -> dict:
        chunks = self._chunk_text(transcript_text, settings.llm_chunk_chars)
        chunk_summaries: list[dict] = []
        for idx, chunk in enumerate(chunks, start=1):
```

```
            prompt = self._prompt_template.replace("{{TRANSCRIPT}}", chunk)
            prompt = prompt.replace("{{CONTEXT}}", f"Частина {idx} з {len(chunks)}")
            chunk_summaries.append(await self._ask_llm(prompt))

        if len(chunk_summaries) == 1:
            return chunk_summaries[0]

        merge_prompt = (
            "Ти створюєш фінальний конспект українською мовою. "
            "Об'єднай часткові конспекти у єдиний фінальний JSON. "
            "Відповідь має бути лише одним JSON об'єктом (не масивом), що містить поля: "
            "title, short_overview, key_points, sections, action_items, terms.\n\n"
            f"Часткові конспекти: {json.dumps(chunk_summaries, ensure_ascii=False)}"
        )
        return await self._ask_llm(merge_prompt)

    async def _ask_llm(self, prompt: str) -> dict:
        if not settings.llm_api_key:
            raise RuntimeError("LLM_API_KEY is not configured")

        payload = {
            "model": settings.llm_model,
            "messages": [
                {"role": "system", "content": "Поверни лише валідний JSON без пояснень."},
                {"role": "user", "content": prompt},
```

```
            ],
            "temperature": 0.2,
        }
        headers = {
            "Authorization": f"Bearer {settings.llm_api_key}",
            "Content-Type": "application/json",
        }
        url = f"{settings.llm_base_url.rstrip('/')}{settings.llm_chat_path}"

        last_error = "unknown error"
        for _ in range(settings.llm_max_retries + 1):
            try:
                async with httpx.AsyncClient(timeout=settings.llm_timeout_seconds) as client:
                    response = await client.post(url, headers=headers, json=payload)
                    response.raise_for_status()
                content = response.json()["choices"][0]["message"]["content"]
                return json.loads(self._strip_code_fences(content))
            except Exception as exc:  # noqa: BLE001
                last_error = self._format_llm_error(exc)
        raise RuntimeError(f"LLM request failed: {last_error}")

    def _format_llm_error(self, error: Exception) -> str:
        if isinstance(error, httpx.HTTPStatusError):
            response_text = error.response.text.strip()
            if len(response_text) > 1000:
                response_text = f"{response_text[:1000]}..."
            return f"HTTP {error.response.status_code}: {response_text or error.response.reason_phrase}"
```

```
        return f"{type(error).__name__}: {error}"

    def _chunk_text(self, text: str, max_chars: int) -> list[str]:
        if len(text) <= max_chars:
            return [text]
        chunks: list[str] = []
        start = 0
        while start < len(text):
            hard_end = min(start + max_chars, len(text))
            end = self._find_chunk_boundary(text, start, hard_end)
            chunks.append(text[start:end].strip())
            start = end
            while start < len(text) and text[start].isspace():
                start += 1
        return chunks

    def _find_chunk_boundary(self, text: str, start: int, hard_end: int) -> int:
        if hard_end >= len(text):
            return len(text)

        search_start = min(len(text), start + max(1, int((hard_end - start) * 0.5)))
        for delimiter in ("\n\n", ". ", "! ", "? ", "\n", " "):
            boundary = text.rfind(delimiter, search_start, hard_end)
            if boundary != -1 and boundary > start:
                return boundary + len(delimiter)

        return hard_end

    def _strip_code_fences(self, text: str) -> str:
```

```
        cleaned = text.strip()
        if cleaned.startswith("```"):
            cleaned = cleaned.split("\n", 1)[1]
        if cleaned.endswith("```"):
            cleaned = cleaned.rsplit("\n", 1)[0]
        return cleaned.strip()
import json
import subprocess
from pathlib import Path

from backend.app.core.config import settings


class MediaService:
    def probe_duration_seconds(self, video_path: Path) -> float:
        cmd = [
            settings.ffprobe_bin,
            "-v",
            "error",
            "-show_entries",
            "format=duration",
            "-of",
            "json",
            str(video_path),
        ]
        result = subprocess.run(cmd, capture_output=True, text=True, check=True)
        payload = json.loads(result.stdout)
        return float(payload["format"]["duration"])
```

```
    def extract_audio(self, video_path: Path, output_wav: Path) -> None:
        output_wav.parent.mkdir(parents=True, exist_ok=True)
        cmd = [
            settings.ffmpeg_bin,
            "-y",
            "-i",
            str(video_path),
            "-ac",
            "1",
            "-ar",
            "16000",
            "-vn",
            "-c:a",
            "pcm_s16le",
            str(output_wav),
        ]
        subprocess.run(cmd, capture_output=True, text=True, check=True)

    def split_audio(self, audio_path: Path, output_dir: Path, chunk_seconds: int, overlap_seconds: int = 5) -> list[Path]:
        output_dir.mkdir(parents=True, exist_ok=True)
        duration = self.probe_duration_seconds(audio_path)

        chunks = []
        current_start = 0.0
        idx = 0

        while current_start < duration:
            # Add a small epsilon to chunk_seconds so we definitely get all audio
```

```
            out_file = output_dir / f"chunk_{idx:04d}.wav"
            cmd = [
                settings.ffmpeg_bin,
                "-y",
                "-ss", str(current_start),
                "-t", str(chunk_seconds),
                "-i", str(audio_path),
                "-c", "copy",
                str(out_file),
            ]
            subprocess.run(cmd, capture_output=True, text=True, check=True)
            chunks.append(out_file)

            # If this chunk reaches or exceeds the end, stop
            if current_start + chunk_seconds >= duration:
                break

            current_start += (chunk_seconds - overlap_seconds)
            idx += 1

        return chunks
import asyncio
import json
from pathlib import Path

from backend.app.core.config import settings
from backend.app.core.logging import get_logger
from backend.app.domain.enums import JobStage, JobStatus
from backend.app.domain.models import TranscriptSegment
```

```
from backend.app.repositories.job_repository import JobRepository
from backend.app.schemas.jobs import ConspectPayload
from backend.app.services.conspect_service import ConspectService
from backend.app.services.media_service import MediaService
from backend.app.services.whisper_service import WhisperService


class PipelineService:
    def __init__(
        self,
        job_repository: JobRepository,
        media_service: MediaService,
        whisper_service: WhisperService,
        conspect_service: ConspectService,
    ) -> None:
        self.job_repository = job_repository
        self.media_service = media_service
        self.whisper_service = whisper_service
        self.conspect_service = conspect_service

    async def process_job(self, job_id: str) -> None:
        job = self.job_repository.get(job_id)
        log = get_logger(job_id)
        work_dir = settings.work_dir / job_id
        result_dir = settings.results_dir / job_id
        audio_path = work_dir / "audio.wav"
        chunks_dir = work_dir / "chunks"
        transcript_path = result_dir / "transcript.json"
        conspect_path = result_dir / "conspect.json"
```

```
work_dir.mkdir(parents=True, exist_ok=True)
result_dir.mkdir(parents=True, exist_ok=True)

try:
    self.job_repository.update_status(
        job_id,
        status=JobStatus.PROCESSING,
        stage=JobStage.EXTRACTING_AUDIO,
        progress=5,
    )
    duration = await asyncio.to_thread(
        self.media_service.probe_duration_seconds,
        Path(job.uploaded_path),
    )
    if duration > settings.max_video_minutes * 60:
        raise ValueError("Video duration exceeds configured limit")

    log.info("Extracting audio")
    await asyncio.to_thread(
        self.media_service.extract_audio,
        Path(job.uploaded_path),
        audio_path,
    )

    log.info("Splitting audio into chunks")
    # Overlap by 5 seconds
    chunk_paths = await asyncio.to_thread(
        self.media_service.split_audio,
```

```
                audio_path,
                chunks_dir,
                settings.chunk_seconds,
                5,
            )
            if not chunk_paths:
                raise RuntimeError("No audio chunks were generated")

            self.job_repository.update_status(
                job_id,
                status=JobStatus.PROCESSING,
                stage=JobStage.TRANSCRIBING,
                progress=15,
            )
            segments = await self._transcribe_chunks(job_id, chunk_paths)
            transcript_payload = self._build_transcript_payload(job_id, segments)
            await asyncio.to_thread(self._write_json_file, transcript_path, transcript_payload)

            self.job_repository.update_status(
                job_id,
                status=JobStatus.PROCESSING,
                stage=JobStage.SUMMARIZING,
                progress=75,
            )
            log.info("Generating conspect via LLM")
            transcript_text = "\n\n".join(segment.text for segment in segments if segment.text)
```

```
            conspect_payload = await self.conspect_service.build_conspect(transcript_text)
            validated_conspect = ConspectPayload.model_validate(conspect_payload)
            final_payload = {"job_id": job_id, **validated_conspect.model_dump()}

            await asyncio.to_thread(self._write_json_file, conspect_path, final_payload)

            self.job_repository.set_outputs(job_id, str(transcript_path), str(conspect_path))
            self.job_repository.update_status(
                job_id,
                status=JobStatus.DONE,
                stage=JobStage.FINISHED,
                progress=100,
            )
            log.info("Job completed")
        except Exception as exc:  # noqa: BLE001
            log.exception("Job failed: {}", exc)
            self.job_repository.update_status(
                job_id,
                status=JobStatus.FAILED,
                stage=JobStage.FAILED,
                progress=100,
                error=str(exc),
            )

    async def _transcribe_chunks(self, job_id: str, chunk_paths: list[Path]) -> list[TranscriptSegment]:
```

```
log = get_logger(job_id)
segments: list[TranscriptSegment] = []
overlap_seconds = 5

for index, chunk_path in enumerate(chunk_paths):
    # Calculate actual start based on chunk index and overlap
    start_second = index * (settings.chunk_seconds - overlap_seconds)
    output_prefix = settings.work_dir / job_id / f"transcript_{index:04d}"
    log.info("Transcribing chunk {}/{}", index + 1, len(chunk_paths))

    chunk_segments = await asyncio.to_thread(
        self.whisper_service.transcribe_chunk,
        chunk_path,
        output_prefix,
        float(start_second),
    )

    segments = self._merge_segments(segments, chunk_segments)

    progress = 15 + int(((index + 1) / len(chunk_paths)) * 55)
    self.job_repository.update_status(
        job_id,
        status=JobStatus.PROCESSING,
        stage=JobStage.TRANSCRIBING,
        progress=progress,
    )
return segments
```

```
        def _build_transcript_payload(self, job_id: str, segments: list[TranscriptSegment]) -> dict:
            return {
                "job_id": job_id,
                "language": settings.whisper_language,
                "text": "\n\n".join(segment.text for segment in segments if segment.text),
                "segments": [segment.__dict__ for segment in segments],
            }

        def _merge_segments(
            self,
            existing_segments: list[TranscriptSegment],
            new_segments: list[TranscriptSegment],
        ) -> list[TranscriptSegment]:
            merged = list(existing_segments)
            for segment in new_segments:
                if merged and self._is_duplicate_overlap(merged[-1], segment):
                    continue
                merged.append(segment)
            return merged

        def _is_duplicate_overlap(self, previous: TranscriptSegment, current: TranscriptSegment) -> bool:
            if current.start >= previous.end:
                return False

            previous_text = self._normalize_segment_text(previous.text)
            current_text = self._normalize_segment_text(current.text)
            if not previous_text or not current_text:
```

```
            return False

        return (
            previous_text == current_text
            or previous_text.endswith(current_text)
            or current_text.startswith(previous_text)
        )

    def _normalize_segment_text(self, text: str) -> str:
        return " ".join(text.casefold().split())

    def _write_json_file(self, output_path: Path, payload: dict) -> None:
        output_path.write_text(json.dumps(payload, ensure_ascii=False, indent=2), encoding="utf-8")
import json
from pathlib import Path

import httpx

from backend.app.core.config import settings
from backend.app.domain.models import TranscriptSegment


class WhisperService:
    def transcribe_chunk(self, chunk_path: Path, output_prefix: Path, chunk_start: float) -> list[TranscriptSegment]:
        output_prefix.parent.mkdir(parents=True, exist_ok=True)

        with chunk_path.open("rb") as f:
```

```
                files = {"file": (chunk_path.name, f, "audio/wav")}
                data = {
                    "response_format": "verbose_json",
                    "language": settings.whisper_language,
                }

                with httpx.Client(timeout=3600.0) as client:
                    response = client.post(f"{settings.whisper_server_url}/inference", files=files, data=data)
                    response.raise_for_status()

                    resp_data = response.json()

            json_path = output_prefix.with_suffix(".json")
            json_path.write_text(json.dumps(resp_data, ensure_ascii=False, indent=2), encoding="utf-8")

            segments = []
            for seg in resp_data.get("segments", []):
                if "start" in seg and "end" in seg:
                    start_offset = float(seg["start"])
                    end_offset = float(seg["end"])
                else:
                    continue

                absolute_start = chunk_start + start_offset
                absolute_end = chunk_start + end_offset

                text = seg.get("text", "").strip()
```

```
            if text:
                segments.append(TranscriptSegment(
                    start=absolute_start,
                    end=absolute_end,
                    text=text
                ))
        return segments
class JobWorker:
    def __init__(self, pipeline_service: PipelineService) -> None:
        self.pipeline_service = pipeline_service
        self.queue: asyncio.Queue[str] = asyncio.Queue()

    async def start(self) -> None:
        log = get_logger()
        log.info("Background worker started")
        try:
            while True:
                job_id = await self.queue.get()
                try:
                    await self.pipeline_service.process_job(job_id)
                finally:
                    self.queue.task_done()
        except asyncio.CancelledError:
            log.info("Background worker stopped")
            raise

    async def enqueue(self, job_id: str) -> None:
        await self.queue.put(job_id)
```