НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
Комп'ютерних наук
_____ Голуб Б. Л.
(підпис) (ПІБ)
"___" _____ 20__ р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

«Інформаційна технологія мовної реєстрації результатів медичних обстежень»

Спеціальність 122 – «Комп'ютерні науки»

Грант освітньої програми
Д.е.н., професор _____ Руденський Р. А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи
Д.т.н., професор _____ Семко В.В
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Чалюк В.Т
(підпис) (ПІБ студента)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____ Голуб Б. Л.
(підпис) (ПІБ)
"___" _____ 20__р.

ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи студенту
Чалюк Владислав Тарасович

Спеціальність 122 – «Комп'ютерні науки»
Тема бакалаврської кваліфікаційної роботи «Інформаційна технологія мовної реєстрації результатів медичних обстежень», затверджена наказом ректора НУБіП України від "16" грудня 2024 р. № 2246 "С".
Термін подання завершеної роботи на кафедру _____
(рік, місяць, число)

Вихідні данні до бакалаврської кваліфікаційної роботи: створення системи, яка допомагатиме керувати медичними записами та документувати результати медичних обстежень пацієнтів за допомогою мовних технологій.
Перелік питань, які потрібно розробити:
1. Аналіз предметної області
2. Моделювання предметної області
3. Проєктування програмної системи
4. Впровадження та експлуатація системи

Перелік графічних документів(за потреби)_____

Дата видачі завдання "____" _____20___р.

Керівник бакалаврської кваліфікаційної роботи _____ Семко В.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Чалюк В.Т.
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

- БД – База даних.
- СУБД – Система управління базою даних.
- ШІ – Штучний інтелект.
- NLP – Natural language processing (обробка природньої мови).
- ASR - Automatic Speech Recognition (автоматичне розпізнавання мовлення).
- АП – Альтернативний потік.
- API - Application programming interface (прикладний програмний інтерфейс)
- ППЗ – Прикладнне програмне забезпечення.

ВСТУП

У сучасних умовах розвитку цифрових технологій особливої актуальності набуває автоматизація медичних процесів, що дозволяє підвищити ефективність роботи лікарів, зменшити навантаження на медичний персонал та забезпечити точність ведення медичної документації. Одним із перспективних напрямів є використання мовних технологій для фіксації результатів обстежень пацієнтів у реальному часі. Це дає змогу лікарю зосередитись на процесі обстеження, а не витрачати час на ручне введення даних.

Актуальність розробки цієї системи полягає в необхідності підвищення ефективності ведення медичної документації та зменшення навантаження на лікарів. У той час як у світі активно впроваджуються системи з автоматичною транскрипцією прийомів пацієнтів за допомогою ШІ, в Україні подібні рішення практично відсутні. Створення локальної інформаційної технології мовної реєстрації результатів медичних обстежень є інноваційним кроком у сфері цифрової медицини.

Метою даної роботи є створення програмного забезпечення для мовної реєстрації медичних обстежень, яка дозволяє керувати інформацією про лікарів, пацієнтів та прийоми, а також здійснювати запис та автоматичну обробку мовлення для подальшого його збереження у вигляді тексту. Реалізація такого рішення дозволяє покращити якість медичних послуг, мінімізувати ризик втрати даних та знизити ймовірність помилок при документуванні.

Така система дозволяє не лише керувати лікарями, пацієнтами й прийомами, а й автоматично фіксувати розмову лікаря з пацієнтом у текстовому вигляді, що значно прискорює процес ведення медичних записів. У процесі розробки використано мови програмування **Python**, **Typescript, Rust**, а також такі технології, як **FastAPI** для створення серверної частини, **PostgreSQL** для зберігання медичних даних, **Tauri** та **NuxtJS** для реалізації графічного інтерфейсу, **WhisperRS** для транскрипції мовлення в текст. Крім того,

застосовано принципи асинхронного та багатопотокового програмування для забезпечення асинхронних запитів до серверу та стабільної обробки транскрипції мовлення у реальному часі відповідно.

Апробація програмного додатку відбувалася шляхом участі у студентській науковій конференції, де було представлено доповідь щодо реалізації системи мовної реєстрації. За результатами досліджень підготовлено та опубліковано тези на тему бакалаврської роботи.

Таким чином, дана робота демонструє можливості поєднання сучасних інформаційних технологій і медичної практики з метою автоматизації повсякденних процесів у роботі лікаря.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Сучасна медицина зазнає суттєвих змін завдяки впровадженню інформаційних технологій, що відкривають нові можливості для автоматизації процесів, оптимізацію обліку пацієнтів та підвищення точності медичної документації. Одним із ключових напрямів розвитку є автоматизація запису медичних протоколів лікарем, особливо запис анамнезу під час обстежень, прийомів пацієнтів чи оформлення медичних висновків. Створення інструментів, які дозволяють перетворювати мовлення в текст, має потенціал значно зменшити навантаження на медичний персонал, пришвидшити ведення записів та зменшити кількість помилок, пов’язаних із ручним введенням інформації.

Ключовою технологією в даній предметній області є автоматичне розпізнавання мовлення (ASR). Вона дозволяє конвертувати голос лікаря у структурований текст, який далі може бути збережений у електронній медичній картці пацієнта. Застосування ASR у медичних інформаційних системах вже активно вивчається у провідних країнах світу. Наприклад, у систематичному огляді, опублікованому в 2016 році, дослідники виявили, що використання ASR може зменшити час обробки документів (turnaround time) на 16,41% до 81,16%, залежно від дослідження, проте було відзначено збільшення кількості помилок при використанні ASR порівняно з традиційним диктуванням та транскрипцією [1].

В іншому досліджені 2022 року, де брали участь медико-хірургічні медсестри, результати показують, що використання технології розпізнавання мовлення економить від 9% до 9,7% часу на документування за зміну [2].

Предметна область цієї роботи охоплює розробку інформаційної технології, яка інтегрує ASR-модуль з локальною системою медичної документації. Такий

підхід є особливо актуальним для регіональних медичних закладів або приватних клінік, де часто відсутній доступ до складних комерційних платформ. Крім того, розробка системи, яка працює локально, без відправки аудіо-даних на сторонні сервери, дозволяє дотримуватись високого рівня конфіденційності згідно з вимогами законодавства у сфері захисту персональних даних.

Таким чином, предметна область охоплює не лише розпізнавання мовлення, але й комплексну обробку тексту, його структурування відповідно до медичних протоколів, збереження у базі даних, а також взаємодію з інтерфейсом користувача.

1.2 Опис систем розпізнавання мовлення

Головне завдання систем розпізнавання мовлення є перетворення аудіозапису людської мови в текст, який потім зможе зрозуміти інша людина або система. Функціонують такі системи завдяки складним алгоритмам машинного навчання та спеціальним моделями штучного інтелекту [3]. Нижче наведено пояснення для деяких найпоширеніших методів та алгоритмів розпізнавання мовлення:

- Обробка природної мови (NLP). Хоча NLP не є окремим алгоритмом для розпізнавання мовлення, ця галузь штучного інтелекту досліджує взаємодію між людиною та комп’ютером за допомогою мови, мовлення та тексту. Застосування NLP забезпечує інтеграцію голосового введення в мобільні пристрої (наприклад, Siri) та покращує зручність користування, зокрема при надсиланні текстових повідомлень або голосовому пошуку.
- Приховані марковські моделі (HMM). Ці моделі базуються на теорії ланцюгів Маркова, де ймовірність поточного стану залежить лише від попереднього. HMM дозволяють враховувати приховані змінні, такі як граматичні мітки, та широко використовуються для обробки послідовностей у системах розпізнавання мовлення. Вони здійснюють

ймовірнісне зіставлення між вхідним сигналом і відповідними мовними одиницями (словами, складами, реченнями).

- N-грами. Це один із найпростіших типів мовних моделей, які оцінюють ймовірність появи певних словосполучень. N-грама представляє собою послідовність із N слів. Такі моделі враховують частоту вживання словосполучень для покращення точності розпізнавання.
- Нейронні мережі. У системах глибокого навчання нейронні мережі моделюють роботу мозку за допомогою структури з багатьох взаємопов’язаних вузлів (нейронів). Кожен вузол обробляє вхідні дані з урахуванням ваг, порогу активації та передає результат далі. Процес навчання полягає в поступовому коригуванні параметрів мережі за допомогою методу градієнтного спуску для зменшення помилки. Хоча такі моделі здатні досягати високої точності, вони потребують значних обчислювальних ресурсів та часу на навчання.
- Діаризація мовця. Цей процес полягає у визначенні, хто і коли говорить. Алгоритми діаризації дозволяють сегментувати мовлення відповідно до ідентичності мовців. Такий підхід широко застосовується в практичних системах, наприклад, у кол-центрах для розділення реплік клієнтів і операторів.

1.2.1 Whisper AI

Whisper AI — це система автоматичного розпізнавання мовлення (ASR), розроблена компанією OpenAI. Вона базується на нейронній архітектурі типу Transformer і призначена для трансформування аудіосигналу в текстову форму.

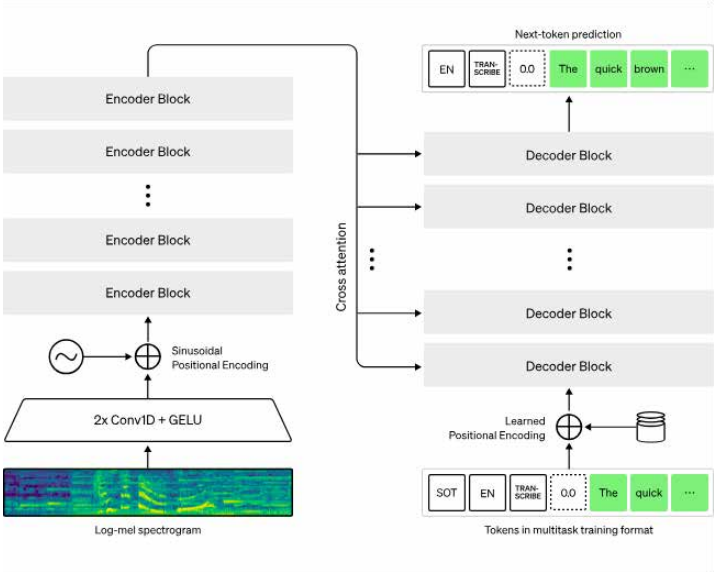

Рис. 1 Схема архітектури Whisper AI

Принцип роботи Whisper можна описати як послідовність етапів, що охоплюють перетворення аналогового аудіо у текст. На початковому етапі вхідний звуковий сигнал обробляється через процес попередньої обробки, що включає нормалізацію частоти дискретизації та обчислення мел-спектрограм. Цей тип спектрального представлення використовується для подачі у нейронні мережі, оскільки він моделює те, як людське вухо сприймає звук.

Далі обчислені спектрограми подаються на вхід моделі нейромережі encoder-decoder Transformer. На цьому етапі відбувається кодування вхідного аудіо у векторні уявлення. Потім ці представлення передаються у декодер, який формує текстову послідовність, використовуючи контекстні залежності. Декодування виконується автоперетворенням (autoregressive decoding), тобто кожне наступне слово генерується з урахуванням попередніх.

У процесі навчання Whisper було використано понад 680 000 годин багатомовних аудіоданих, частина з яких містила фоновий шум або недосконаліть запису. Завдяки цьому модель набула високої здатності до узагальнення, що дозволяє їй ефективно працювати у реальних умовах.

Whisper доступний у кількох конфігураціях (tiny, base, small, medium, large), які відрізняються розміром моделі та обчислювальними вимогами[4].

1.3 Огляд існуючих рішень

У світі стрімкого розвитку цифрової медицини автоматизація документації стала важливою складовою ефективної медичної практики. Значну роль у цьому відіграють системи, що реалізують мовну реєстрацію результатів обстежень пацієнтів. Завдяки цим технологіям медичні працівники можуть зосередитись на безпосередньому лікуванні, а не на тривалому ручному введенні інформації у медичну систему. Одними з провідних рішень на ринку, які демонструють потенціал у цьому напрямі, є платформи DeepScribe, Augmedix, та Nuance Dragon Medical One, що вже інтегруються у робочі процеси лікарів у США та інших країнах.

DeepScribe — це інтелектуальний помічник для лікарів, який використовує штучний інтелект для автоматичного створення клінічних нотаток на основі аудіозапису консультації з пацієнтом. Система записує мовлення під час візиту, і, використовуючи ASR у поєднанні з NLP-моделями, перетворює його у структуровані нотатки відповідно до медичних шаблонів. DeepScribe інтегрується з системами електронних медичних записів (EMR) і підтримує HIPAA-сумісність, що гарантує безпечне зберігання конфіденційних даних пацієнтів, що дозволяє скоротити час на документацію на 75%, що значно підвищує продуктивність лікарів [5].

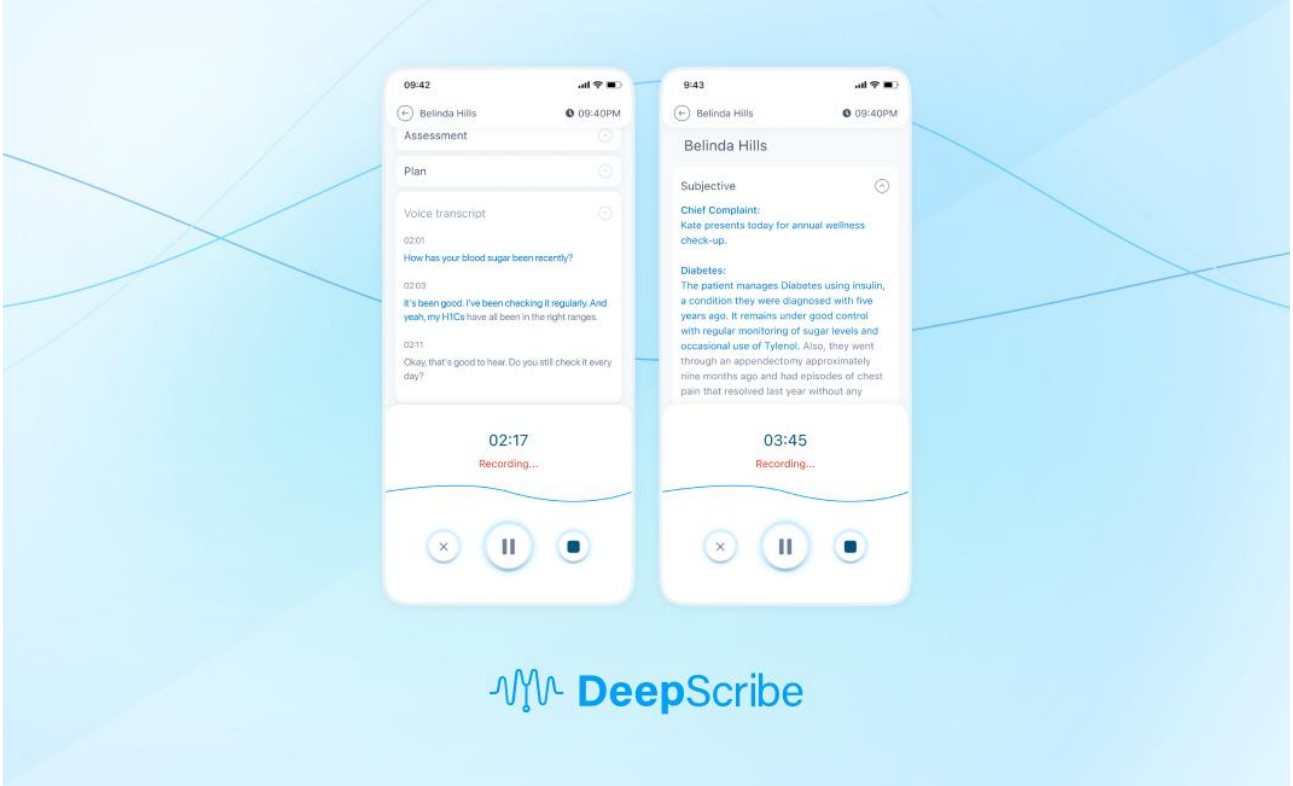

Рис. 2 Інтерфейс моб. додатку DeepScribe

Одною з особливістю DeepScribe є можливість редагувати структури нотаток, оскільки потреби в документації та документообігу змінюються в різних спеціальностях, ця функція робить дану систему гнучкою, порівняно з іншими.

Ще одним популярним рішенням є **Augmedix**, яке також забезпечує транскрипцію діалогів між лікарем і пацієнтом у режимі реального часу. Ця система функціонує як віртуальний медичний асистент, що працює через мобільний додаток або носимі пристрої, зокрема Google Glass. Основною перевагою Augmedix є поєднання живої участі скрайберів — операторів, які контролюють якість розпізнавання — з автоматичними мовними моделями. Це дозволяє досягти високої точності навіть у випадках, коли мовлення є складним або технічним. Як зазначається в матеріалах компанії їхні рішення використовуються у понад 35 медичних мережах США, включаючи Sutter Health і Dignity Health [6]. Розглянемо інше програмне забезпечення на ринку.

Nuance Dragon Medical One — це хмарне рішення для розпізнавання медичної мови, яке дозволяє лікарям диктувати клінічні нотатки прямо до електронної медичної картки за допомогою голосу. Програмне забезпечення

створене спеціально для медичного середовища та підтримує велику кількість спеціалізованих термінів, що забезпечує високу точність розпізнавання. Завдяки швидкому перетворенню мови в текст, лікарі можуть значно скоротити час на заповнення документації та зосередитися на пацієнтах.

Рішення Dragon Medical One працює на основі хмарної інфраструктури Microsoft Azure, що гарантує масштабованість, безпеку та доступ до сервісу з будь-якого пристрою. Система підтримує адаптивне навчання — вона запам’ятовує стиль мовлення кожного користувача та підлаштовується під нього, що ще більше покращує точність з часом. Інтерфейс простий у використанні, а інтеграція з популярними EHR-системами відбувається без додаткових труднощів [7].

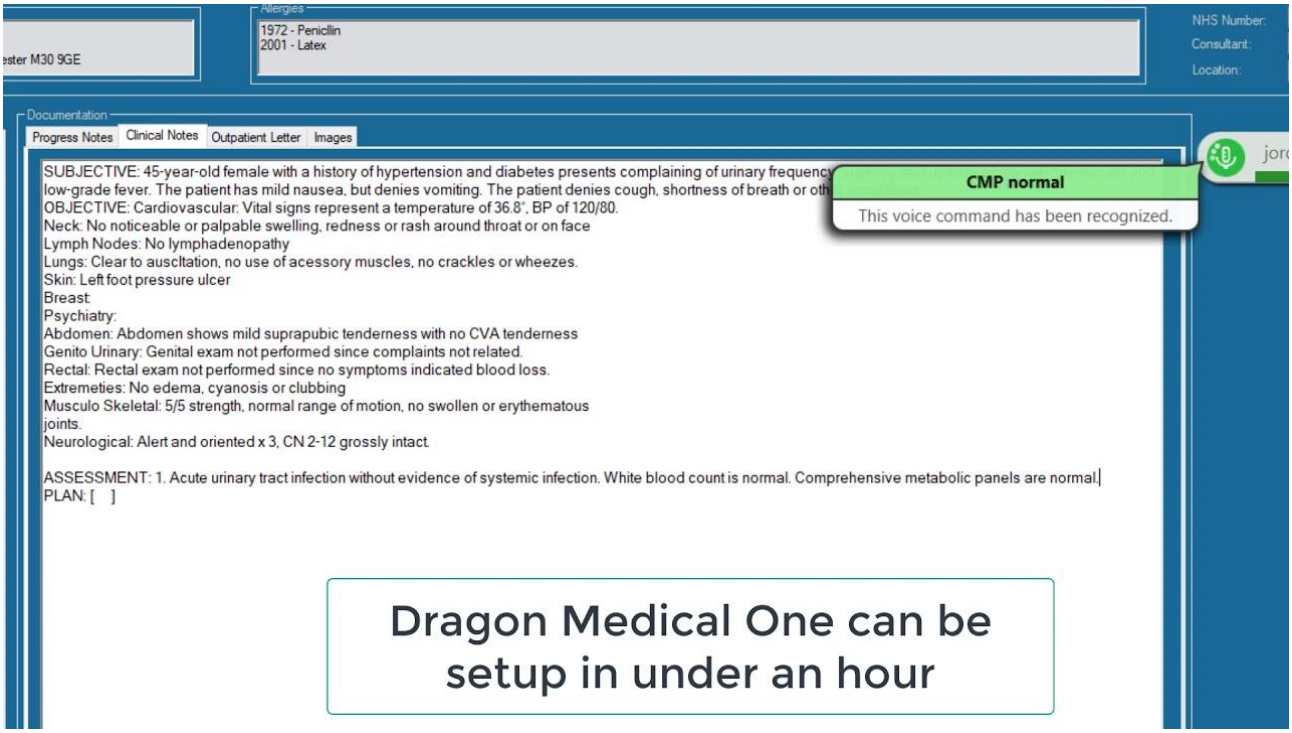

Рис. 3 Інтерфейс Dragon Medical One

Компанія Nuance Communications, що розробила Dragon Medical One, є лідером у сфері мовних технологій і з 2022 року входить до складу Microsoft. Її рішення використовуються в тисячах медичних установ по всьому світу. Dragon Medical One сертифікований для використання у сфері охорони здоров’я, відповідає вимогам HIPAA щодо конфіденційності медичних даних і визнається

як один з найбільш ефективних інструментів для зменшення адміністративного навантаження на лікарів.

Попри вражаючі результати комерційних платформ, таких як DeepScribe, Augmedix чи Dragon Medical One, вони мають певні обмеження у контексті локального використання. Ці системи залежать від постійного підключення до Інтернету, оскільки обробка мовлення виконується на хмарних серверах. Це унеможливлює або ускладнює їхнє використання в умовах, де конфіденційність відіграє критичну роль або де немає стабільного підключення. Крім того, вони орієнтовані переважно на англомовне середовище, що створює виклики для країн з іншими мовами та медичними стандартами.

В українській медичній практиці подібні рішення майже не представлені, а потреба в автоматизованих технологіях мовної документації лише зростає. Наявні рішення, як правило, мають обмежену функціональність, зосереджену на зберіганні даних без глибокої мовної інтерпретації. В умовах переходу до електронних медичних записів та цифрової трансформації охорони здоров’я в Україні, створення автономної, багатомовної системи ASR для медичних закладів є актуальним завданням.

1.4 Постановка завдання

У межах цієї роботи ставиться завдання створити інформаційну систему, призначену для мовної реєстрації результатів медичних обстежень у реальному часі. Система має забезпечити можливість автоматичної транскрипції усного мовлення, яке виникає під час взаємодії лікаря та пацієнта, із подальшою обробкою цієї інформації в структуровану форму та збереженням. Основна мова взаємодії — українська, отже, система повинна бути адаптована до специфіки української фонетики, лексики та медичної термінології. Такий підхід дозволить автоматизувати рутинні процеси медичного документування, зменшити кількість ручного введення даних та підвищити точність медичних записів.

Завданням є розробити локальне рішення, що працюватиме автономно, без потреби у надсиланні аудіо чи текстових даних на зовнішні сервери. Це

дозволить дотримуватись законодавчих норм у сфері збереження персональних даних, зокрема положень Закону України «Про захист персональних даних» [8], та уникнути ризиків, пов’язаних з витоком конфіденційної інформації. Система повинна працювати на звичайному персональному комп’ютері або локальному сервері медичної установи, забезпечуючи зручний користувацький інтерфейс для лікаря, медсестри чи адміністратора закладу.

Крім транскрипції мовлення, система повинна дозволяти керування медичними картками пацієнтів: створення, редагування, зберігання та пошук записів. Також має бути реалізоване керування персоналом — із можливістю додавання профілів лікарів, призначення пацієнтів, ведення графіків прийомів, та безпекою системи – тобто реалізоване налаштування ролей та доступу до дій в системі. У перспективі система має підтримувати можливість експорту результатів обстеження у форматах PDF або DOCX, а також формування статистичних звітів для адміністрації медичного закладу.

Технічна реалізація передбачає використання сучасних інструментів та бібліотек для обробки мовлення, таких як Whisper, які підтримують розпізнавання української мови в режимі реального часу. Для обробки тексту буде застосовано технології великих мовних моделей(LLM) — для структуризації розмови в медичний документ з виокремленням ключових фрагментів: скарги, анамнез, попередній діагноз, тощо. База даних має бути реалізована із використанням реляційного підходу (наприклад, PostgreSQL), з дотриманням нормального формування структури таблиць для зберігання медичних записів, історій прийомів та обліку пацієнтів.

Таким чином, кінцевим результатом цієї роботи має стати прототип локальної інформаційної системи, здатної у реальному часі розпізнавати мовлення між лікарем і пацієнтом українською мовою, перетворювати його на структурований текст, зберігати у базі даних і забезпечувати доступ до медичної інформації через інтерфейс, зручний для медичного персоналу. Система має бути

масштабованою, захищеною та відкритою до подальшої інтеграції з іншими медичними сервісами.

1.5 Функціональні та нефункціональні вимоги

Функціональні вимоги:

1. Розпізнавання мовлення в реальному часі українською мовою з високою точністю з використанням ASR технології.
2. Транскрипція діалогу між лікарем і пацієнтом у структуровану форму з використанням LLM моделей.
3. Збереження розпізнаного тексту до бази даних з прив’язкою до конкретного лікаря та пацієнта.
4. Управління медичними картками пацієнтів (створення, перегляд, редагування, пошук).
5. Управління медичним персоналом (додавання, редагування, перегляд лікарів).
6. Управління медичними записами (створення, редагування, призначення відвідування)
7. Управління доступом та безпекою (створення, редагування, призначення ролей та дозволів)
8. Генерація та експорт медичних документів у форматах PDF/DOCX.

Не функціональні вимоги:

1. Підтримка автономної роботи без підключення до Інтернету, або по локальній мережі медичної установи.
2. Високий рівень безпеки персональних даних відповідно до локального законодавства.
3. Сумісність із сучасними ОС (Windows, Linux).
4. Інтерфейс, адаптований для медичного персоналу з мінімальними технічними навичками.
5. Можливість масштабування системи для використання у великих медичних установах.

6. Час розпізнавання мовлення не повинен перевищувати 2 секунду на 1 секунду аудіо.
7. Легкість у розгортанні та оновленні програмного забезпечення.

2 МОДЕЛЮВАННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Загальні відомості

Моделювання інформаційних систем є критично важливим етапом у процесі їх розробки, оскільки дозволяє формалізувати структуру, поведінку та взаємодію компонентів до початку реалізації. Одним з найпоширеніших підходів до моделювання складних програмних систем є використання мови UML (Unified Modeling Language). UML — це стандартизована мова візуального моделювання, яка підтримується об’єктно-орієнтованим підходом і широко використовується в промисловій розробці ПЗ [9]. Загалом UML включає в себе набір діаграм, які можна умовно поділити на дві групи: структурні та поведінкові.

Структурні діаграми дозволяють описати статичну частину системи, вибудовуючи ієрархію її компонентів та зв’язків між ними. Однією з ключових структурних діаграм у UML є діаграма класів. Вона дозволяє представити статичну структуру системи: класи, їхні атрибути, методи, а також зв’язки між класами — асоціації, агрегації, композиції та наслідування. Ця діаграма допомагає зрозуміти логічну структуру даних і полегшує побудову бази даних і реалізацію моделі предметної області у коді [10]. Діаграми пакетів використовуються для моделювання модульної структури програмної системи, що дозволяють групувати класи, інтерфейси та інші компоненти в логічні підсистеми або модулі. Це особливо корисно при побудові масштабованої архітектури, де важливо розділити систему на функціональні блоки. Діаграми розгортання демонструють фізичний розподіл компонентів системи по апаратних засобах, такі як сервери та пристрої.

Поведінкові діаграми відображають динамічні аспекти системи, тобто поведінку об’єктів і взаємодію між ними протягом часу. Одною з головних поведінкових діаграм є діаграма прецедентів (Use Case), вона описує що повинна робити система з точки зору кінцевого користувача, при цьому зосереджуючись

на функціональних вимогах[11]. Діаграма послідовностей – поведінкова діаграма, яка показує як об’єкти взаємодіють між собою через виклики методів, ці діаграми особливо корисна для моделювання складних бізнес-процесів.

Загалом UML є гнучкою та потужною мовою, яку можна застосовувати на різних етапах розробки — від аналізу вимог до реалізації та тестування. У контексті дипломної роботи UML використовується для представлення ключових аспектів системи, сприяє структурованому підходу до проектування і дозволяє полегшити комунікацію між розробниками, замовниками та іншими зацікавленими сторонами. Створені діаграми у подальшому допоможуть також при технічній документації до програмного продукту.

Враховуючи складність проєкту, що поєднує елементи розпізнавання мовлення, обробки природної мови та керування медичними даними, використання UML-моделей є доцільним і необхідним. Саме через візуалізацію взаємозв’язків між сутностями та процесами можна забезпечити цілісне розуміння архітектури системи та ефективно організувати процес її реалізації.

2.2 Об’єктне моделювання

2.2.1 Діаграма прецедентів. Діаграма прецедентів (use case diagram) є одним із базових інструментів об’єктно-орієнтованого моделювання в UML і використовується для опису функціональних можливостей системи з точки зору її зовнішніх користувачів — акторів. Основна мета цієї діаграми — виявити, які дії (прецеденти) може виконати кожен тип користувача, а також які взаємозв’язки існують між цими діями. Це дозволяє формалізувати вимоги до системи у вигляді сценаріїв взаємодії, які є зрозумілими як для розробників, так і для замовників або доменних експертів.

На діаграмі прецедентів кожен актор представлений у вигляді фігури людини, а прецедент — овальної фігурою з назвою дії, яку актор може ініціювати. Між актором і прецедентом проводиться лінія зв’язку, що відображає взаємодію [11]. Існує кілька типів зв’язків між прецедентами:

- «include» — використовується, коли один прецедент завжди включає поведінку іншого (наприклад, «Додати пацієнта» завжди включає «Перевірити наявність у базі»)
- «extend» — застосовується, коли один прецедент може розширити поведінку іншого за певних умов (наприклад, «Завантажити результати обстеження» може розширювати «Переглянути картку пацієнта»);
- генералізація — дозволяє описати спадкування ролей акторів або варіацій прецедентів.

Для цієї системи була розроблена діаграма прецедентів, яка містить двох акторів – «Лікар» та «Пацієнт» (Див. Рис. 4).

Актор «Пацієнт» має лише один прецедент:

- Аудіозапис анамнезу та бесіди з лікарем

Актор «Лікар» має декілька прецедентів:

- CRUD Транскрипцій для медичного запису
- CRUD Медичних записів
- CRUD Мед карти пацієнта
- CRUD Лікарів
- CRUD Відділень
- Генерація медичних виписок та звітів
- Налаштування доступу до системи та її функцій

Прецедент «аудіозапис анамнезу та бесіди з лікарем» пов'язаний з прецедентом «Транскрипція голосу в текст» через зв’язок «include», цей зв’язок вказує на те що під час аудіозапису розмови завжди йде транскрипція голосу в текст, тобто пряме застосування ASR технології.

Прецедент «CRUD Транскрипцій для мед. Запису» пов’язаний з прецедентом «Підсумовування розмови за допомогою LLM» через зв’язок «extend», оскільки після закінчення аудіозапису, та перед збереженням транскрибованого тексту в БД, Лікар може вручну відредагувати текст, або використати LLM, щоб коротко підсумувати розмову, та отримати попередній діагноз з рекомендаціями.

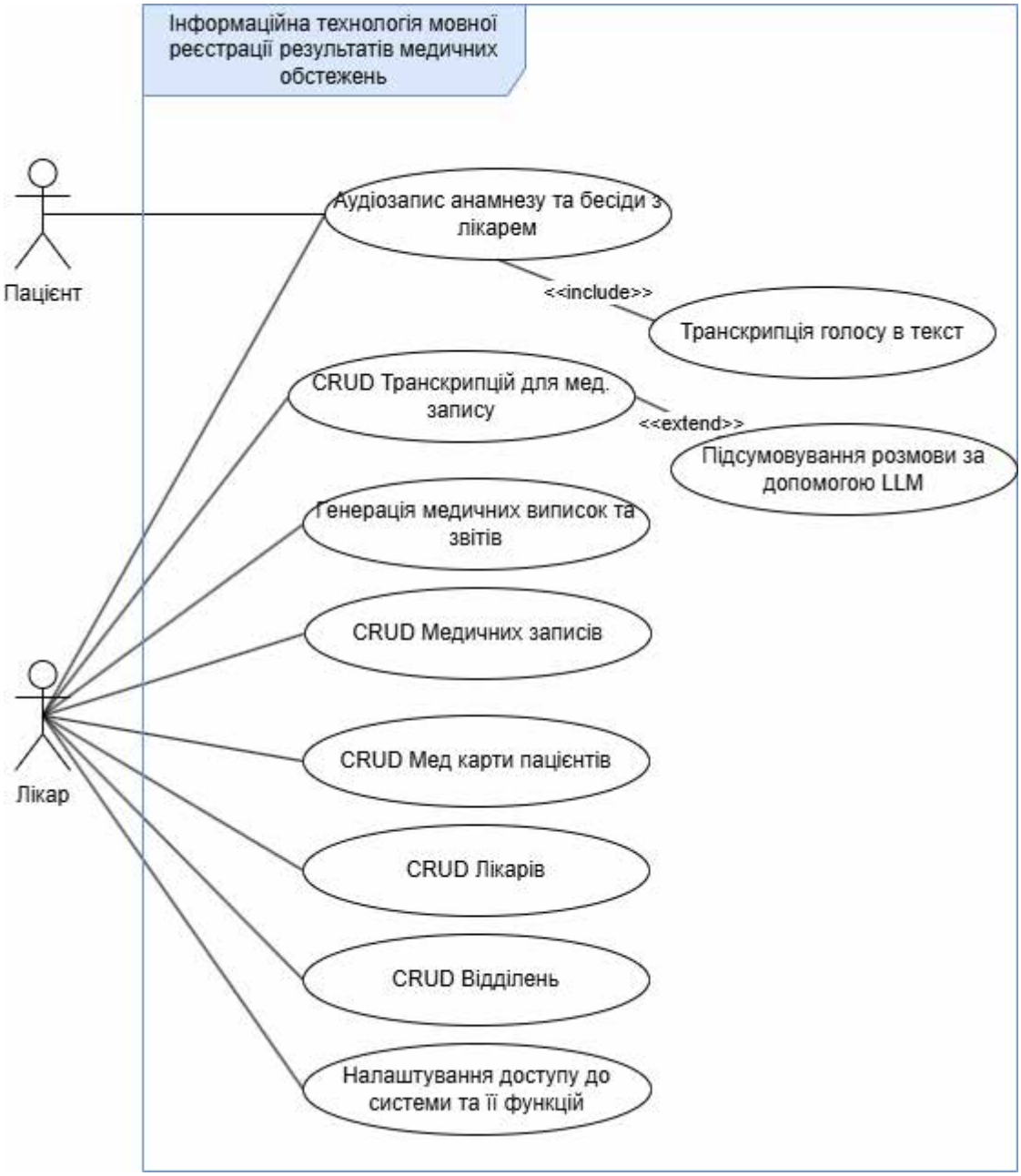

Рис. 4 Діаграма прецедентів

Розглянемо детальніше вищеописані прецеденти за допомогою сценаріїв використання.

Прецедент	Аудіозапис анамнезу та бесіди з лікарем
Актори	Пацієнт, Лікар
Передумови	1. Лікар авторизований та має доступ до системи. 2. Лікар вже створив медичний запис(прийом) у системі для конкретного пацієнта у системі.

	3. Пацієнт вже прийшов на прийом, та очікує початок розмови.
Основний потік	1. Лікар отримав медичний запис для конкретного пацієнта. 2. Лікар відкриває налаштування для ASR модуля, та вибирає мовну модель, мову спілкування та інші налаштування під конкретного пацієнта. 3. Система виводить повідомлення про успішну ініціалізацію ASR модуля з відповідними налаштуваннями, інакше(А1). 4. Лікар починає аудіозапис розмови з пацієнтом. 5. Пацієнт веде розмову з лікарем, надаючи усний анамнез паралельно записуючи це в мікрофон. 6. Система транскрибує розмову в реальному часі та виводить текст на екран. 7. Після отримання всієї необхідної інформації лікар зупиняє запис. 8. Лікар редагує транскрибований текст, та зберігає його в системі.
Альтернативний потік	А1. Система не може ініціалізувати ASR модуль з відповідними параметрами А2. Система виводить повідомлення про помилку. Б1. Лікар перед збереженням натиснув “Підсумувати”, та запустив LLM, для виокремлення з розмови основних даних. Б2. Після завершення обробки, лікар зберігає трансрипцію

Таблиця 1 Сценарій для прецеденту "Аудіозапис анамнезу та бесіди з лікарем"

2.2.2 Діаграма послідовності. Діаграма послідовності (sequence diagram) в UML моделює динамічний аспект системи, зосереджуючись на впорядкованій взаємодії об’єктів у часі. Вона відображає, які повідомлення надсилаються між об’єктами та в якій послідовності, щоб реалізувати певний сценарій або прецедент. Діаграма показує час по вертикалі (зверху вниз), а об’єкти, що взаємодіють, розташовані горизонтально. Основна її мета — проілюструвати, як відбувається обмін повідомленнями в межах певної функціональності системи.

Кожен об’єкт на діаграмі представлений прямокутником із назвою, під яким проходить пунктирна лінія — «лінія життя». Повідомлення між об’єктами передаються у вигляді стрілок, що йдуть від одного об’єкта до іншого, позначаючи виклик методу або відповідь. Важливим аспектом є підтримка синхронних та асинхронних викликів, а також можливість показати створення й завершення об’єктів, блоки альтернатив чи циклів.

У межах розробки системи мовної реєстрації результатів медичних обстежень діаграми послідовності будуть корисними для моделювання сценаріїв: початку сеансу розпізнавання мовлення, надсилання аудіо потоку до модуля ASR, отримання текстової транскрипції, підсумовування розмови за допомогою модуля LLM, збереження запису у базу даних, та подальшого доступу лікаря до результатів.

На рис. 5, представлення діаграма послідовності, яка включає кілька ключових об’єктів:

- «Лікар»
- «Пацієнт»
- «Медичний запис»
- «Модуль розпізнавання мовлення»
- «Модуль LLM»

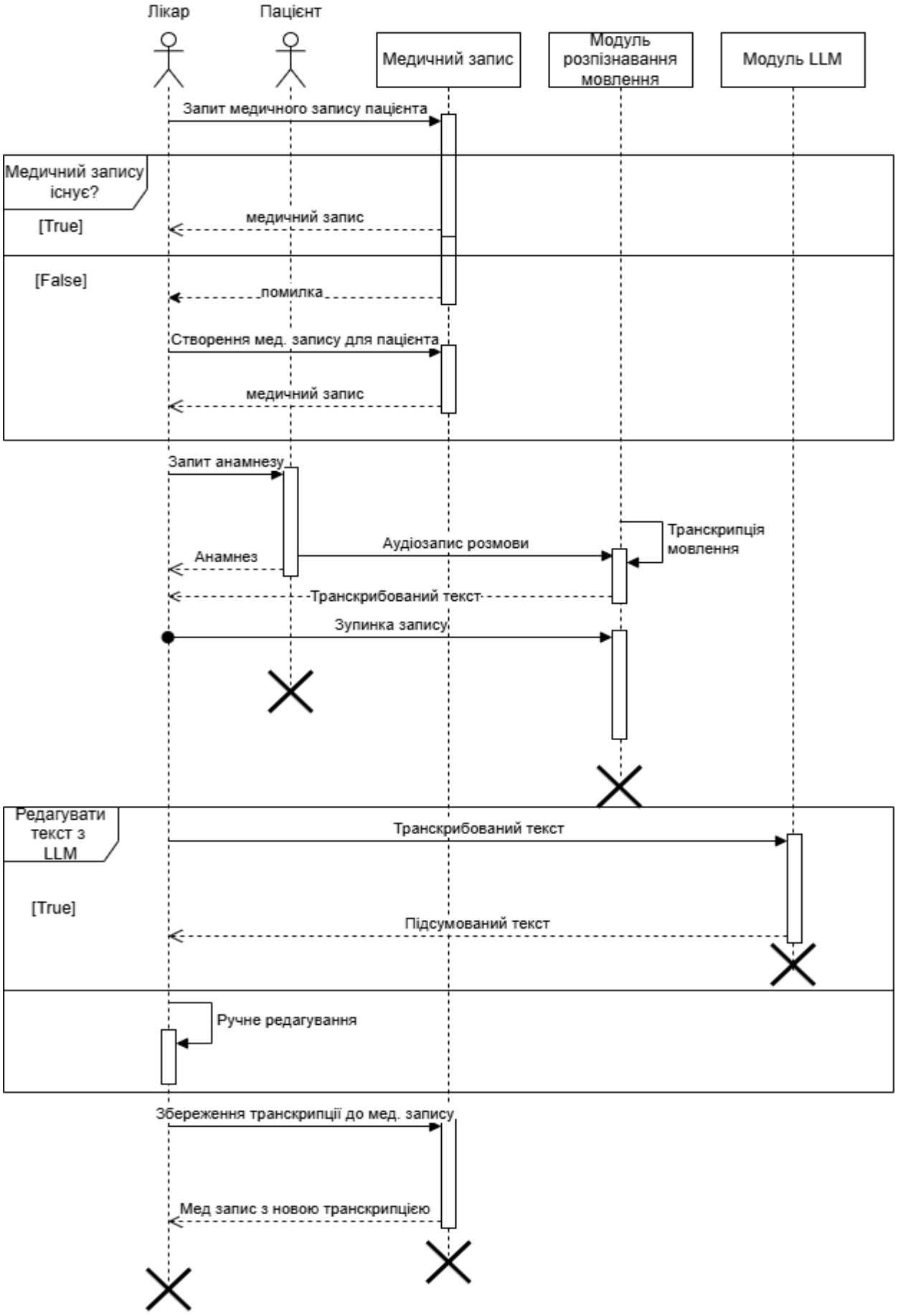

Рис. 5 Діаграма послідовності

У цьому сценарії актор «Лікар» запитує в об’єкта «Медичний запис» дані щодо конкретного пацієнта, якщо запису для конкретного пацієнта но конкретну дату та час немає, то об’єкт повертає помилку, і «Лікар» повинен створити новий запис. Після цього «Лікар» починає запис розмови з пацієнтом, передаючи мовлення до «Модуль розпізнавання мовлення», на діаграмі це зображено, як асинхронний виклик. Після завершення розмови, лікар надсилає сигнал про зупинку транскрипції до ASR модуля, і на цьому “life time” об’єкту завершується.

Після завершення розмови, та маючи транскрибований текст, «Лікар» може надіслати транскрибований текст до об’єкта «Модуль LLM», і у відповідь отримати підсумовану розмову з ключовими деталями, попереднім діагнозом та рекомендаціями, в іншому ж випадку об’єкт «Лікар» проводить редагування в ручну.

В завершенні «Лікар» ініціює збереження транскрипції надіславши дані до об’єкту «Медичний запис».

2.2.3 Діаграма активності. Діаграма активності (activity diagram) є частиною стандарту UML і використовується для моделювання поведінки системи у вигляді потоку робіт (workflow) або алгоритмів. Вона відображає послідовність дій, умовних розгалужень, циклів, а також паралельних процесів, що дає змогу наочно представити логіку виконання певної задачі. Діаграми активності зручні для опису як бізнес-процесів, так і внутрішніх механізмів програмної системи, що робить їх особливо актуальними при розробці інформаційних технологій, які мають багато гілок прийняття рішень чи паралельного виконання.

Діаграма діяльності містить ключові елементи, такі як дії, переходи, початкові та кінцеві вузли, вузли прийняття рішень, а також розгалуження, з’єднання та доріжки. Дії відображають окремі завдання процесу і позначаються округленими прямокутниками, а переходи — це стрілки між ними, які демонструють напрям руху від одного завдання до іншого і можуть містити умови для визначення наступного кроку. Початковий вузол, зображений як заповнене чорне коло,

позначає старт процесу, тоді як кінцевий вузол (чорне коло в кільці) вказує на його завершення. Вузли прийняття рішень у формі ромбів показують, де в процесі необхідно зробити вибір, що веде до різних сценаріїв дій.

Розгалуження й об’єднання дозволяють відображати паралельне виконання дій: перше розділяє процес на кілька паралельних шляхів, а друге — зводить їх у єдину лінію виконання. Ще одним важливим елементом є доріжки, які поділяють діаграму на горизонтальні або вертикальні секції. Вони дозволяють чітко розмежувати, яка роль або компонент відповідає за виконання певної дії, що допомагає краще зрозуміти розподіл відповідальностей у процесі. Таке представлення дозволяє виявити «вузькі місця» або потенційні точки відмови ще на етапі проектування. Діаграма активності, як і інші типи UML-діаграм, сприяє підвищенню узгодженості між технічною реалізацією та очікуваннями замовника.

На рис.6 зображена діаграма активностей процесу транскрипції мовлення під час медичного обстеження пацієнта у лікаря.

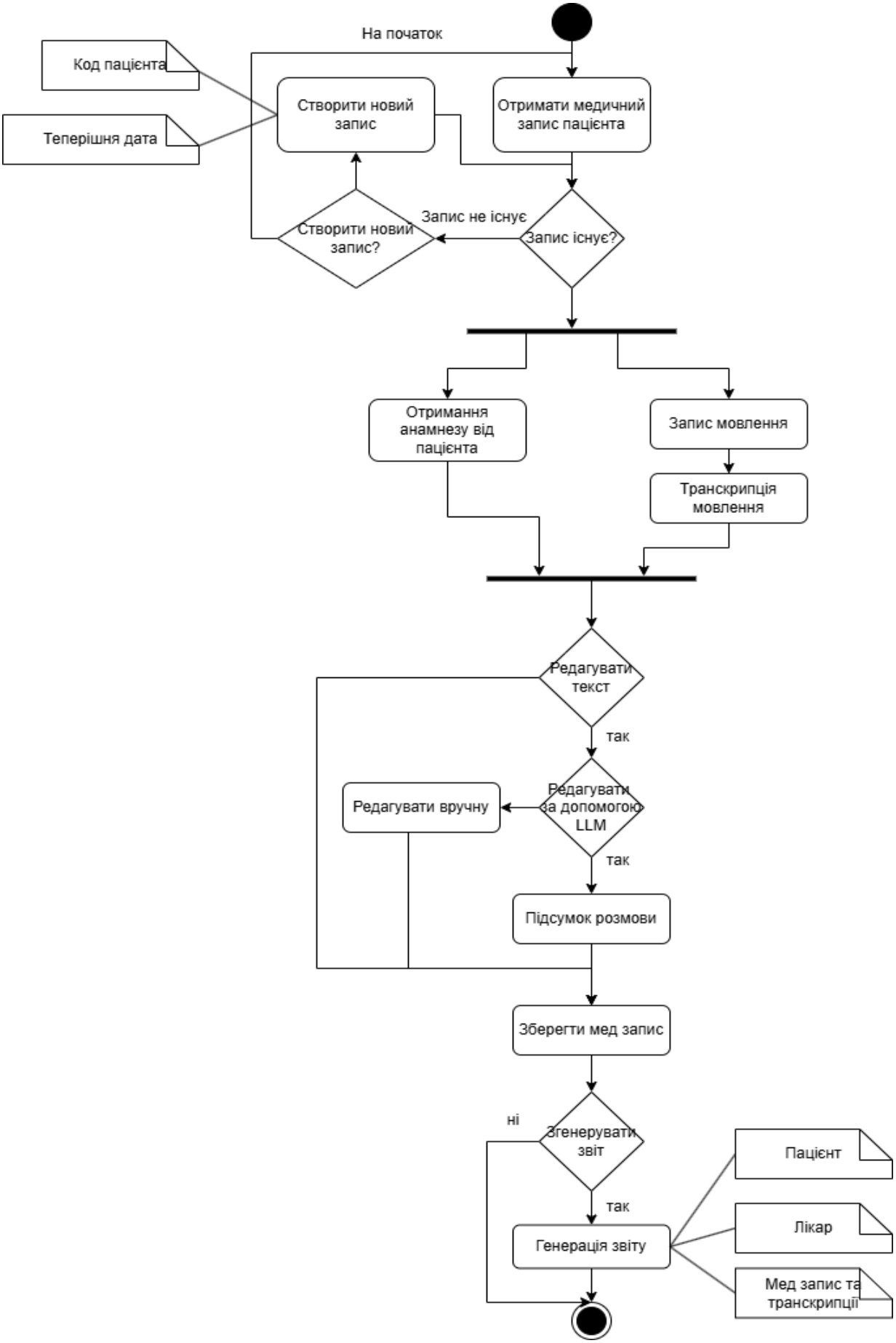

Рис. 6 Діаграма активності

2.3 Діаграма класів

Діаграма класів (class diagram) є однією з ключових структурних діаграм UML, яка відображає статичну структуру системи. Вона показує класи, їхні атрибути, методи, а також зв’язки між ними: асоціації, наслідування, агрегацію чи композицію. Це дозволяє зрозуміти, як саме організовані основні об’єкти системи, як вони пов’язані між собою та як взаємодіють. У контексті програмної інженерії діаграма класів виконує функцію "архітектурної мапи", на основі якої реалізуються класи у коді .

Кожен клас у діаграмі представлений прямокутником, який складається з трьох частин: назва класу, перелік атрибутів і перелік методів. Атрибути описують властивості класу, а методи — функціональність, яку він надає. Діаграма також показує модифікатори доступу: «+» (public), «–» (private), «#» (protected), що сприяє правильному проектуванню інтерфейсів і забезпеченню інкапсуляції.

Зв’язки між класами відображаються стрілками або лініями з певною семантикою. Наприклад, стрілка з порожнім трикутником означає наслідування, ромб — агрегацію (коли один клас логічно містить інший), а заповнений ромб — композицію (жорсткий зв’язок, де життєвий цикл залежного об’єкта прив’язаний до головного). Додатково можуть вказуватися кратності (наприклад, «1..*»), які показують скільки об’єктів може бути пов’язано з іншим класом.

У випадку системи мовної реєстрації результатів медичних обстежень діаграма класів охоплює сутності на кшталт Patient, Doctor, MedRecord, MedRecordTranscription, тощо. Вона дозволяє наочно представити, як зберігається інформація про пацієнтів, як працює транскрипція мовлення, які сервіси обробляють дані та які є зв’язки між цими компонентами. Це є важливим етапом у підготовці до реалізації логіки системи та побудови взаємодії між її модулями.

На основі діаграми класів будується шарова архітектура, що є загальноприйнятим патерном у сучасній розробці. Одним із популярних підходів

є патерн Service-Repository, де бізнес-логіка виноситься у сервіси (наприклад, DoctorService, AuthService), а доступ до даних — в окремі репозиторії (DoctorRepository, PatientRepository). Такий підхід дозволяє зменшити зв’язаність компонентів, покращити тестування, спрощує масштабування та повторне використання коду [10].

На рис. 7, нижче, зображена діаграма класів, яка була створена з використанням Service-Repository патерну, для серверної частини бакалаврської роботи. Як бачимо, діаграма побудована навколо класів-сутностей(для зручності позначені оранжевим кольором), “Doctor”, “Patient”, “MedRecord”, тощо.

Кожна сутність поєднана з власним репозиторієм, кожен з яких імплементує власний інтерфейс-репозиторій, який унаслідується від головного інтерфейсу “ICRUDRepository”, який імплементує клас “ICRUDRepository”, якого наслідують ті ж імплементації інтерфейсів-репозиторіїв.

Далі ми використовуємо класи-сервіси, для кожного репозиторію, в яких зберігається наша бізнес-логіка. Кожен сервіс унаслідується від “CRUDService”.

З таким підходом, ми зменшили зв’язаність між компонентами, і якщо, наприклад, потрібно змінити метод доступу до наших сутностей, потрібно лише імплементувати інтерфейс відповідного репозиторію. В результаті ми отримали, що завдяки такому чіткому розділенню обов’язків кожен шар системи виконує строго визначену функцію, що відповідає принципам чистої архітектури.

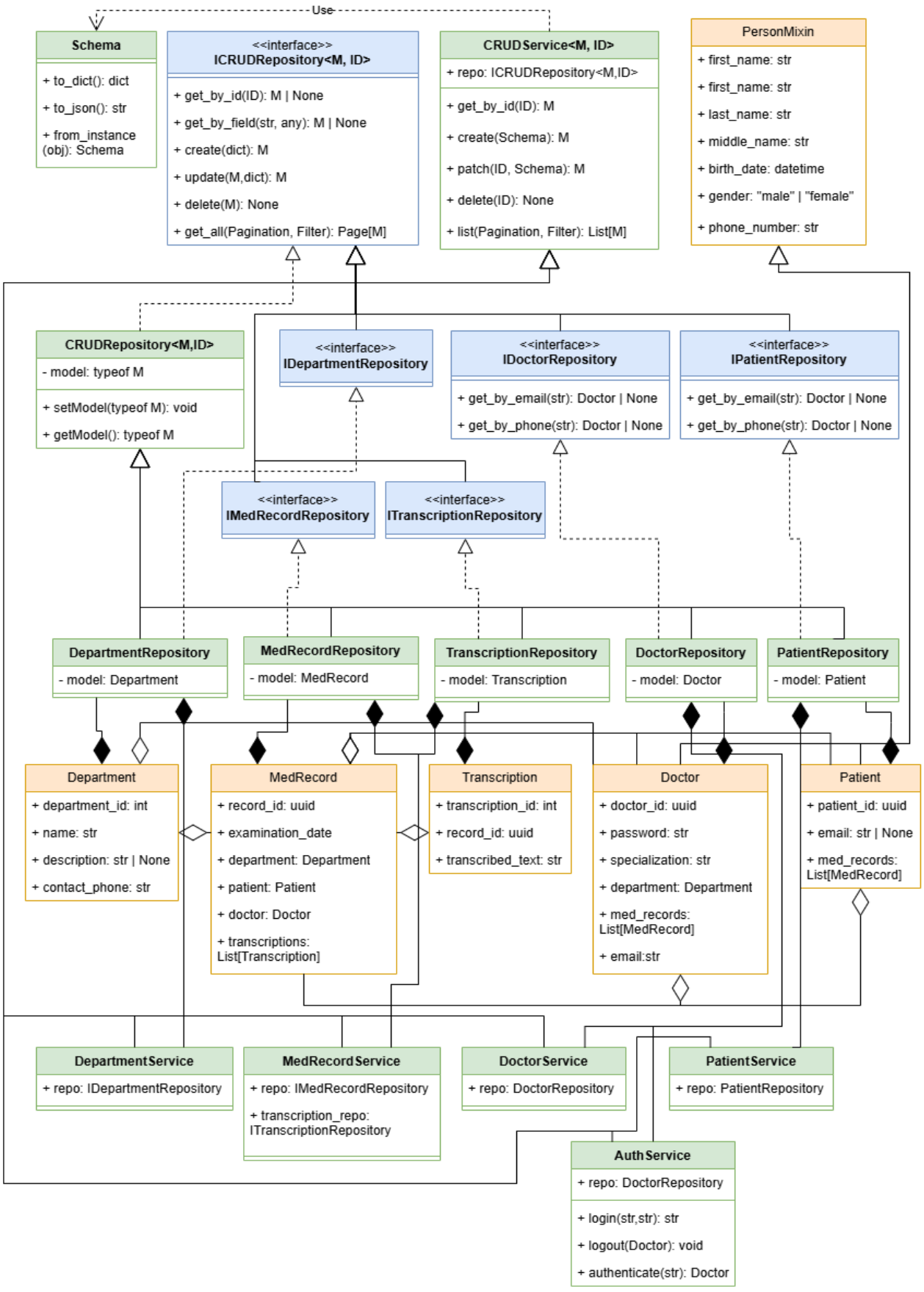

Рис. 7 Діаграма класів

2.4 Логічна модель даних

Логічна модель даних — це формалізоване уявлення про інформаційну структуру предметної області, яка відображає логічні зв’язки між даними, незалежно від конкретної системи управління базами даних (СУБД). На відміну від фізичної моделі, логічна модель не визначає специфічні параметри зберігання даних, індексів або типів полів у конкретній СУБД, а зосереджується на семантичній суті об'єктів, їх атрибутів та взаємозв’язків [13].

Основним інструментом побудови логічної моделі є ER-діаграми (Entity-Relationship diagrams), які дозволяють графічно зобразити сутності, атрибути та зв’язки між ними. Наприклад, у контексті інформаційної технології мовної реєстрації медичних обстежень можна виділити сутності Пацієнт, Лікар, Мед-Запис, Транскрипція, між якими встановлюються зв’язки типу «один до багатьох» або «багато до багатьох». Такі моделі є фундаментом для побудови реляційних баз даних.

ER-діаграми складаються з прямокутників (сутності), овалів (атрибути) та ромбів (зв’язки). Атрибути можуть бути простими або складеними, ключовими чи зовнішніми. Зв’язки можуть мати різну кратність — наприклад, «один до одного», «один до багатьох», «багато до багатьох». Це дозволяє точно описати логіку відношень у системі без надлишкової деталізації реалізації.

Для побудови логічної моделі даних розробники часто застосовують спеціалізовані програмні засоби, зокрема ERWin Data Modeler. Цей інструмент дозволяє створювати, візуалізувати та документувати складні моделі баз даних, підтримує реверс-інжиніринг та синхронізацію з фізичною моделлю. Завдяки йому можна автоматично генерувати SQL-скрипти для створення структури бази даних на основі логічної моделі, що суттєво спрощує процес розробки [14].

У випадку технології з мовної транскрипції медичних записів логічна модель дозволяє точно зафіксувати вимоги до зберігання медичної інформації: зв’язки між лікарями та пацієнтами, історію прийомів, результатами розпізнавання

мовлення, метаданими аудіофайлів тощо. Така модель забезпечує консистентність даних і дозволяє ефективно реалізовувати запити до бази даних.

Логічна модель системи (рис. 8), складається з таких сутностей:

- “Doctor”
- “Patient”
- “Department”
- “MedicalRecord”
- “MedicalRecordTranscription”
- “Role”
- “Permission”

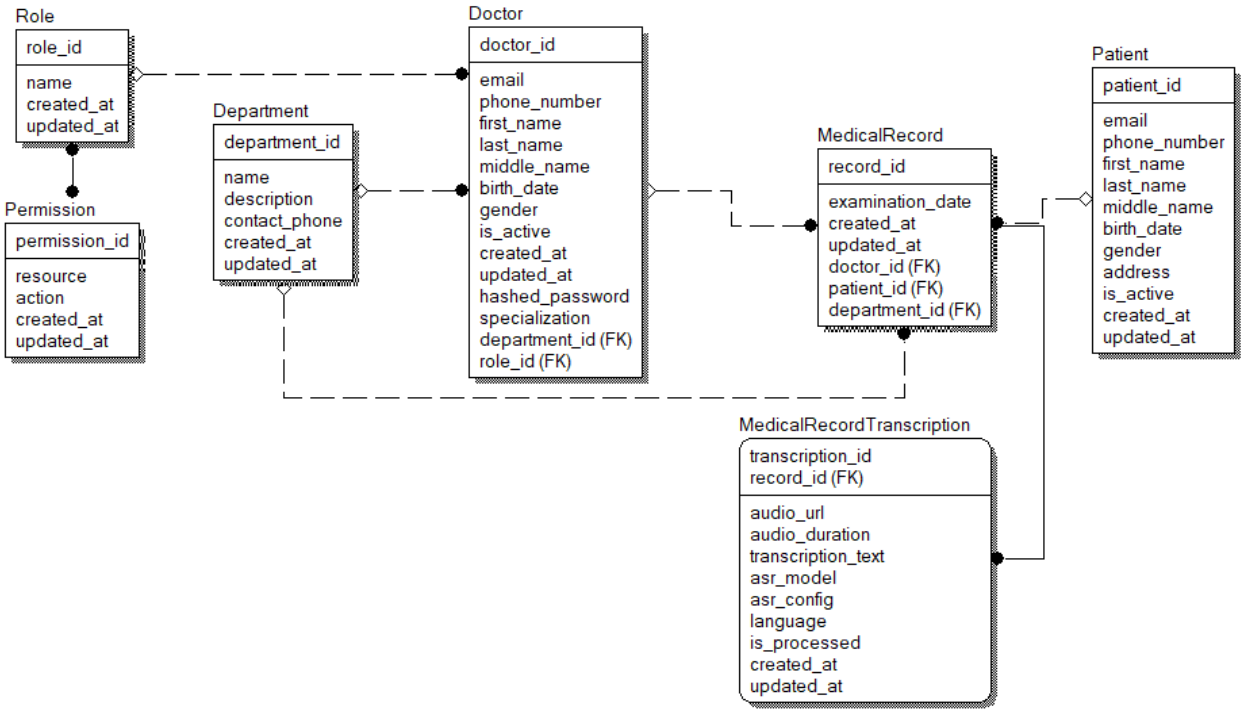

Рис. 8 ER-діаграма

Сутність «MedicalRecord» пов’язана з сутностями «Doctor» та «Patient» зв’язком «many-to-one», і відповідно сутності «Doctor» та «Patient» пов’язані між собою через зв’язко «many-to-many» через цю ж сутність. Також ця сутність пов’язана з «MedicalRecordTranscription» та «Department» зв’язком «many-to-one» та «one-to-many» відповідно. Сутність «Doctor» пов’язана з сутностями

«Department» та «Role» через зв’язок «one-to-many». Сутності «Role» та «Permission» пов’язані зв’язком «many-to-many».

3 ПРОЄКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

3.1 Вибір СУБД

Під час створення інформаційної технології мовної реєстрації результатів медичних обстежень надзвичайно важливо обрати відповідну систему управління базами даних (СУБД). Вона має гарантувати стабільність, масштабованість, безпечне збереження медичної інформації, а також гнучкість при роботі з різноманітними запитами. З огляду на потреби в транзакційності, надійності та сумісності з сучасними фреймворками, найкращим рішенням у межах цього проєкту стала об’єктно-реляційна СУБД - PostgreSQL.

Реляційні бази даних базуються на математичній моделі зберігання інформації у вигляді таблиць. Кожна таблиця представляє певну сутність, а зв'язки між таблицями реалізуються за допомогою зовнішніх ключів. Ця модель була вперше запропонована Едгаром Коддом ще у 1970-х роках [15] та й досі залишається стандартом де-факто у більшості систем. Вона потрібна там, де важлива структурованість, цілісність даних та підтримка складних транзакцій.

СУБД — це програмне забезпечення, яке забезпечує створення, зміну, збереження та захищений доступ до даних у базах. Існують різноманітні види СУБД: реляційні (PostgreSQL, MySQL, Oracle), документоорієнтовані (MongoDB), графові (Neo4j) тощо. Для систем, де переважає чітко визначена структура даних і необхідна підтримка складних зав’язків між таблицями, реляційні СУБД залишаються оптимальним вибором [16].

Серед реляційних СУБД PostgreSQL вирізняється своєю відкритістю, активною спільнотою розробників, відповідністю стандартам SQL, підтримкою транзакційності (ACID), надійною реплікацією та великою кількістю розширень. PostgreSQL – це об’єктно-реляційна СУБД, тобто вона поєднує реляційну модель з принципами ООП, вона має підтримку складних, кастомних, типів даних, таких як JSONB або Enum, що дозволяє працювати з частково структурованими

даними. Це може бути корисним при збереженні неформалізованих частин медичних записів.

Необхідно також зазначити, що PostgreSQL має велику кількість інтеграцій з популярними мовами програмування (Python, Java, Node.js тощо), фреймворками, інструментами для роботи з даними і хмарними платформами. Завдяки відкритій архітектурі та підтримці розширень, PostgreSQL легко інтегрується в сучасні інфраструктури та підходить для розробки як веб застосунків, так і аналітичних систем.

Інструменти розробника, такі як pgAdmin, DBeaver, а також можливість інтеграції з Docker-контейнерами, роблять PostgreSQL зручним у налаштуванні, тестуванні та масштабуванні. Це дозволяє зберігати цілісність даних на різних етапах життєвого циклу системи, включаючи розробку, тестування, та реліз.

В цілому, PostgreSQL є повнофункціональною, потужною та досвідченою СУБД, яка успішно застосовується у фінансових, медичних, наукових та державних проектах у всьому світі. Її стабільність, продуктивність і активна спільнота роблять її ідеальним рішенням для побудови відповідальних і масштабованих рішень, зокрема інформаційної системи для голосової реєстрації результатів медичних обстежень [17].

Отже, вибір PostgreSQL у цьому проєкті обумовлений поєднанням гнучкості, продуктивності та широкої екосистеми інструментів. Вона забезпечує надійну основу для створення критично важливої інфраструктури, що оперує чутливою інформацією про здоров'я пацієнтів та потребує максимальної точності у роботі з даними.

Під час проектування таблиць у фізичній моделі, основою слугували сутності з логічної моделі. Атрибути цих сутностей були використані для розробки структури таблиці. Отриману схему бази даних зображено на рис. 9

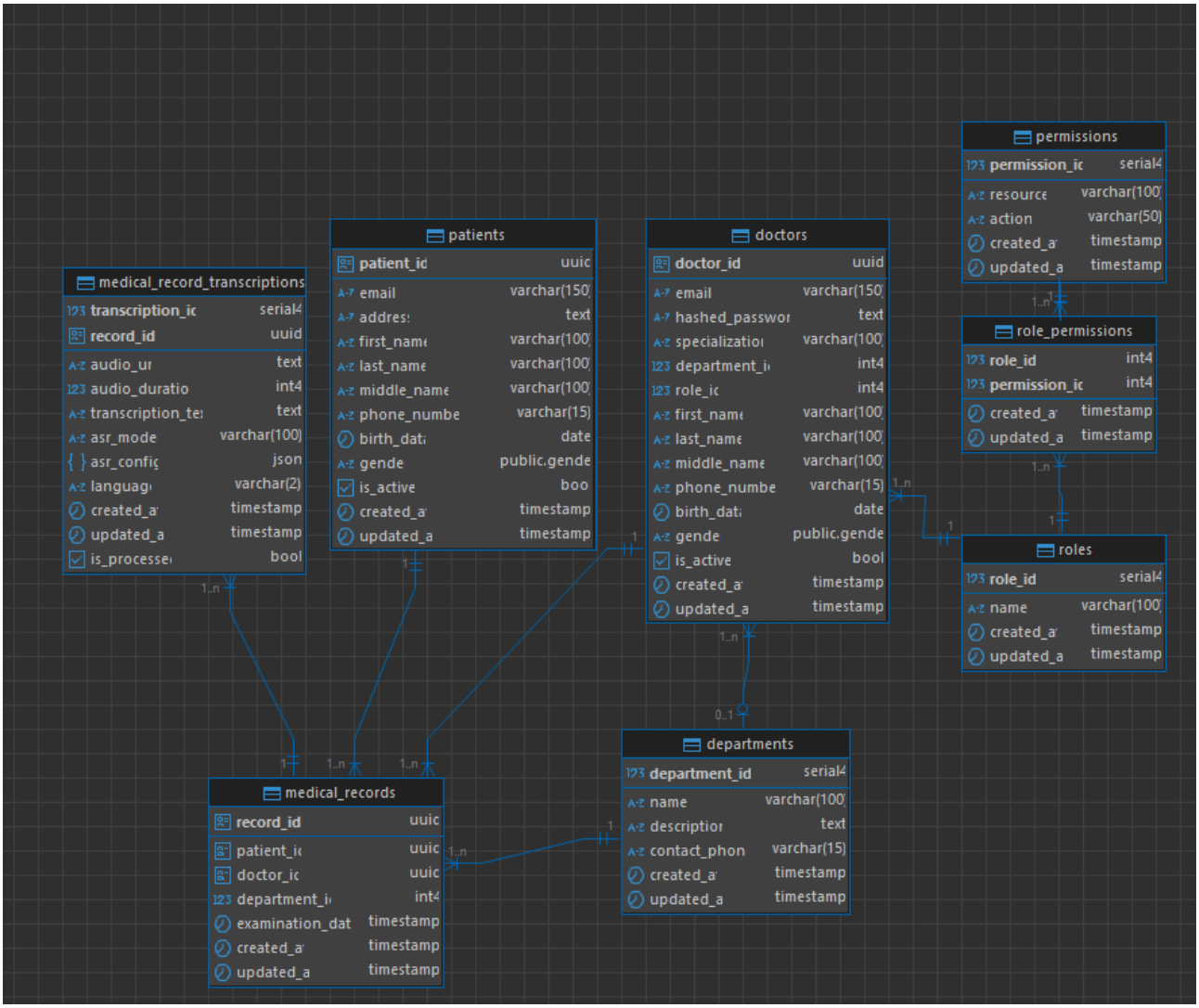

Рис. 9 Фізична схема БД

База-даних для інформаційної технології мовної реєстрації результатів медичних обстежень складається з наступних таблиць: **doctors, patients, departments, medical_records, medical_record_transcriptions, roles, permission, role_permissions**. Кожна таблиця відповідає сутності в логічній моделі системи.

Таблиця doctors та patients пов’язані зв’язком багато-до-багатьох через таблицю medical_record, це дає змогу лікарю переглядати пацієнтів, медичні записи та транскрипції, пов’язані з ним(якщо doctor_id в medical_record співпадає з doctor_id лікаря). Він має змогу створювати нові медичні записи та додавати до них транскрипції. Таблиця doctors пов’язана з таблицею roles, кожна роль має свій набір дозволів, це дозволяє мінімізувати доступ до тих даних та

дій, які не повинні робити деякі лікарі, наприклад лікар з роллю інтерн не може видаляти пацієнта, тощо.

SQL код створення таблиць для PostgreSQL наведений у ДОДАТКУ А.

3.2 Архітектура та організаційна структура програмного забезпечення

Архітектура програмного забезпечення (ПЗ) — це структура системи, що визначає її основні компоненти, способи їх взаємодії, правила обміну даними та технологічні обмеження. Архітектура ПЗ є основою для подальшої розробки, тестування, масштабування та підтримки проєкту. Вона описує як високорівневу організацію системи, так і деталізовані взаємозв’язки між її модулями. Грамотно спроєктована архітектура дозволяє досягти гнучкості, надійності та ефективності функціонування системи [18].

У даному проєкті було обрано клієнт-серверну архітектура, яка є класичним підходом до побудови сучасних веб-орієнтованих інформаційних систем. У цій моделі клієнт (графічний інтерфейс користувача) надсилає запити до сервера, а сервер обробляє їх, виконує бізнес-логіку, звертається до бази даних та повертає клієнту відповідь. Такий підхід забезпечує розділення відповідальності між інтерфейсом та логікою обробки даних, що полегшує супровід і масштабування системи.

Особливістю проєкту є те, що серверна частина на Python, виконує лише функції зберігання та обробки даних, надаючи API до бази даних PostgreSQL. Уся обчислювальна логіка, пов’язана з транскрипцією мовлення та генерацією тексту або висновків, виконується на клієнтській стороні — у настільному застосунку, створеному за допомогою Tauri та NuxtJS. Це забезпечує кращу продуктивність при роботі з великими аудіоданими та дозволяє уникнути передачі конфіденційної медичної інформації через мережу.

Використання фронтенду на базі NuxtJS дозволяє реалізувати сучасний, інтерактивний інтерфейс користувача з підтримкою клієнтської логіки обробки аудіо. NuxtJS взаємодіє з бекендом через REST API, а з Tauri — через спеціальні виклики (invoke) для передачі аудіо та отримання результатів транскрипції через канали (channel). Це забезпечує асинхронну, масштабовану і розділену архітектуру, в якій кожен компонент виконує строго визначену роль.

Таким чином, обрана архітектура дозволяє створити повністю автономну систему, яка може працювати як на окремому комп'ютері лікаря, так і бути розгорнутою на локальному сервері медичного закладу без потреби підключення до Інтернету. Це відповідає вимогам безпеки, конфіденційності та зручності використання у сфері охорони здоров'я.

Під час моделювання архітектури програмного забезпечення особливу роль відіграє організація коду на структурному рівні. Одним із ключових інструментів, що дозволяє візуально описати внутрішню архітектуру системи з погляду її логічного поділу на модулі, є діаграма пакетів (package diagram) у нотації UML. Вона дає змогу представити високорівневу структуру системи, поділивши її на пакети, що відображають простори імен, логічні або фізичні компоненти ПЗ, та зв'язки між ними.

Кожен пакет у такій діаграмі відображає окрему частину функціональності системи або її логічний шар, наприклад: представлення даних (DTO), сервіси, репозиторії, контролери, інтерфейси для зовнішніх API тощо. Завдяки цьому можна наочно продемонструвати модульність та ізоляцію окремих компонентів, що особливо важливо при реалізації принципів розробки на основі чистої архітектури або шарованої архітектури.

Діаграми пакетів не тільки дозволяють візуалізувати залежності між частинами системи, але й полегшують розуміння напрямку імпортів і контролюють впровадження правил залежностей, згідно з якими, наприклад, зовнішні шари не повинні знати про внутрішні. UML також дозволяє показувати імпорт між пакетами у вигляді стрілок, що відображають напрямок звернення

одного модуля до іншого, що особливо корисно для попередження циклічних залежностей.

У випадку створення програмного забезпечення для мовної реєстрації медичних обстежень діаграма пакетів допомагає структурувати систему на логічні блоки, наприклад: модулі транскрипції, моделі системи, модуль сервісів, модулі графічного інтерфейсу, а також взаємодії з базою даних. Такий підхід не лише спрощує підтримку системи, а й дозволяє розподіляти розробку між різними командами або мікросервісами.

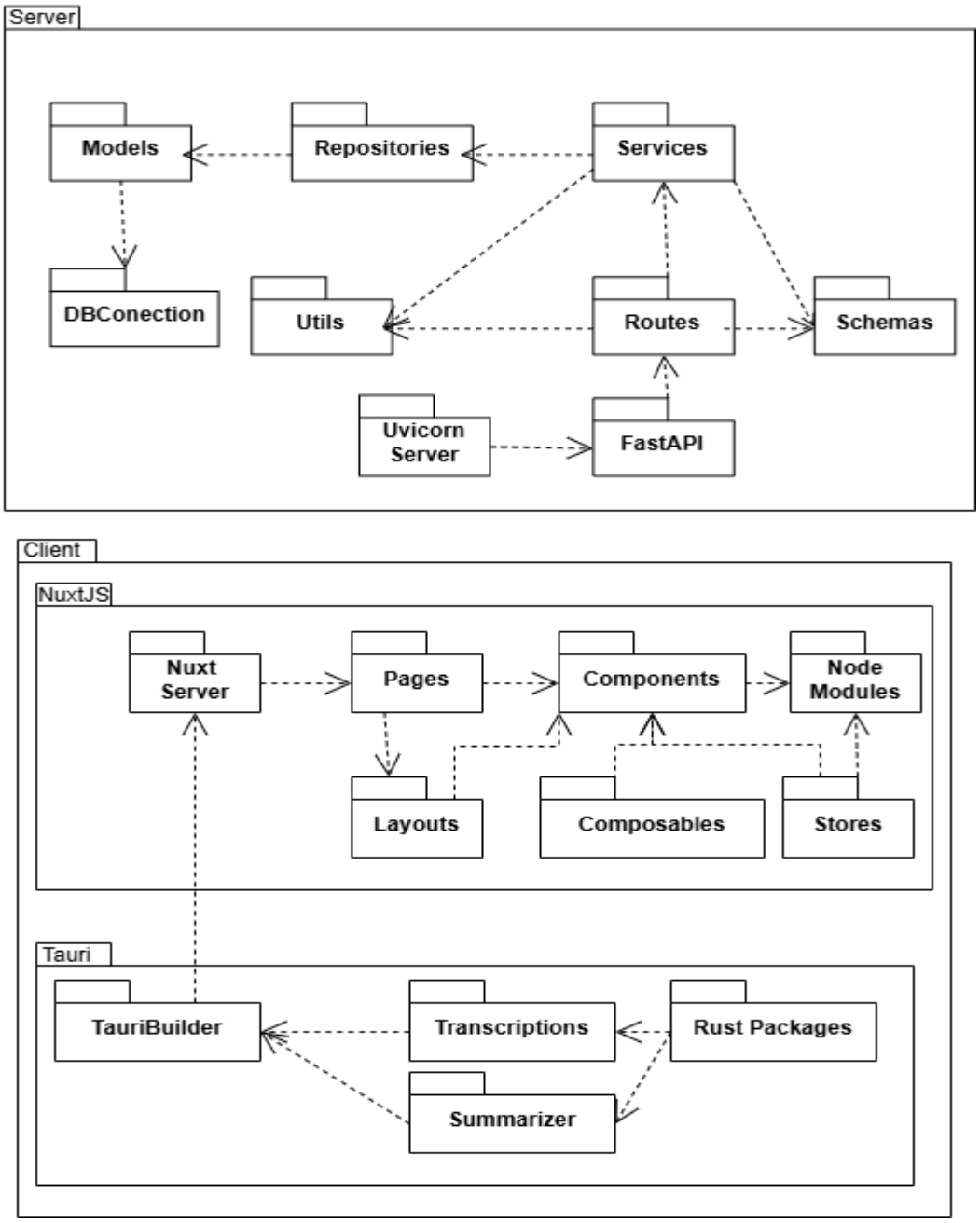

Рис. 10 Діаграма пакетів

3.3 Вибір інструментарію для створення програмного забезпечення

Вибір засобів розробки інформаційної технології мовної реєстрації результатів медичних обстежень здійснювався з урахуванням вимог проекту. Вибрані інструменти включають Python та його бібліотеки FastAPI, SQLAlchemy, NuxtJS, Tauri, jsPDF, Whisper-rs, Ollama-rs.

Основною мовою програмування для серверної частини системи було обрано **Python**. Це інтерпретована, високорівнева мова з чітким синтаксисом, яка активно використовується в області штучного інтелекту, обробки мови та створення веб-сервісів. Python має потужну екосистему бібліотек і фреймворків, що дозволяє швидко розгортати прототипи та підтримувати масштабовані проєкти. Завдяки своїй зрозумілості та великій спільноті, Python є оптимальним вибором для науково-технічних систем, зокрема в галузі медичних технологій [19].

Для побудови API-сервісу була використана бібліотека **FastAPI**, яка на сьогодні є однією з найсучасніших асинхронних фреймворків для Python. Вона дозволяє будувати швидкі та безпечні RESTful-сервіси з мінімальними зусиллями. FastAPI підтримує автоматичну генерацію документації (Swagger/OpenAPI), що спрощує інтеграцію з клієнтською частиною. Крім того, вона відмінно поєднується з асинхронними веб-серверами (наприклад, Uvicorn) і чудово масштабується для роботи з потоковими задачами [20].

Для взаємодії з базою даних обрано ORM-бібліотеку **SQLAlchemy**, яка дозволяє зручно працювати з реляційними базами даних, використовуючи об'єктно-орієнтовану модель. SQLAlchemy надає потужний інструментарій для створення складних запитів, міграцій, зв’язків між таблицями та lazy-loading механізмів. У поєднанні з FastAPI, SQLAlchemy дозволяє створювати повноцінну backend-архітектуру за принципом service-repository з чіткою декомпозицією шарів логіки [21].

Як основну СУБД було обрано **PostgreSQL** — надійну, масштабовану та відкриту реляційну систему керування базами даних. Вона має багатий набір функцій, підтримує транзакції, ACID-властивості, зберігання JSON-структур та розширення для повнотекстового пошуку — що важливо для зберігання та пошуку медичних записів. PostgreSQL також чудово працює у хмарному середовищі та має широке ком'юніті, що забезпечує довгострокову підтримку.

Для клієнтської частини, що виконує роль десктопного GUI, був обраний **NuxtJS** — фреймворк на базі Vue.js, який підтримує рендерінг на стороні клієнта і сервера, модульну архітектуру та гнучке налаштування. Nuxt дозволяє створювати продуктивні інтерфейси з сучасною UX-логікою, що критично важливо для роботи лікарів. Його модульна структура дозволяє легко інтегрувати зовнішні API та забезпечує хорошу оптимізацію для SPA/MPA-архітектури [22]. Для побудови UI інтерфейсу використовується **Nuxt UI** у поєднанні з **TailwindCSS** — сучасним CSS-фреймворком, що дозволяє створювати адаптивні інтерфейси з гнучкою системою класів. Це забезпечує швидку розробку при збереженні єдиного стилю у всіх компонентах додатку. TailwindCSS полегшує створення кастомного дизайну без необхідності писати власні CSS-файли, а Nuxt UI надає набір готових компонентів для ефективної інтеграції.

Оскільки система має бути локальною десктопною програмою, для упаковки веб-клієнта в нативний застосунок використовується **Tauri** — фреймворк для створення настільних застосунків з використанням вебтехнологій (HTML, CSS, JavaScript/TypeScript) і нативного бекенду на Rust. Tauri працює швидше за Electron і має менший розмір, оскільки не потребує вбудованого браузера, використовуючи нативні ресурси операційної системи. Це дозволяє створити легку, швидку і безпечну десктопну версію клієнта [23].

Для реалізації автоматичного розпізнавання мовлення (ASR) застосовується **WhisperRS** — високопродуктивна Rust-реалізація моделі Whisper від OpenAI, яка оптимізована для локального виконання та має підтримку потокової обробки.

Вона забезпечує розпізнавання української мови в реальному часі, що є критично важливим для ведення медичних діалогів без затримок. WhisperRS може працювати навіть на слабших комп'ютерах без потреби у GPU, що забезпечує автономність системи [24].

Для зведень та інтелектуальної обробки результатів обстежень було використано **Ollama-rs** — клієнтська бібліотека на Rust для взаємодії з локальним LLM-сервером Ollama. Це дозволяє запускати мовні моделі без підключення до Інтернету, забезпечуючи повну конфіденційність обробки медичних даних. Ollama-rs інтегрується з Rust-компонентами, що спрощує виклик моделей з десктопного застосунку [25].

Для генерації текстових, PDF, звітів була використана бібліотека jsPDF - це JavaScript-бібліотека, яка дозволяє генерувати PDF-файли безпосередньо у браузері. Вона особливо корисна для веб-додатків, які потребують створення та завантаження PDF-документів без використання серверної логіки. Головною особливістю є перетворення HTML-елементів у PDF, що дає нам змогу генерувати інформативні звіти з простих HTML-компонентів [26].

3.4 Алгоритмізація та програмування ППЗ

У побудові інформаційної системи мовної реєстрації результатів медичних обстежень велика увага була приділена поділу логіки між клієнтською та серверною частинами. У цій архітектурі Tauri-додаток, що об'єднує NuxtJS та Rust-ядро, виконує всі ресурсоємні обчислення локально на клієнті: транскрипцію мовлення за допомогою Whisper-rs, а також семантичну обробку тексту за допомогою Ollama-rs. Таким чином, бекенд, написаний на Python з FastAPI, виконує лише функції збереження даних у PostgreSQL, автентифікації та надання REST API для взаємодії з базою.

Ключовим алгоритмом є передача аудіофрагментів від NuxtJS до Tauri для транскрипції. Під час діалогу лікаря й пацієнта NuxtJS-інтерфейс використовує Web Audio API для запису звуку. Записані аудіофрагменти передаються до Rust-частини Tauri через виклик *invoke('process_chunk',{chunk:Array<number>})* — це

ініціює буферизацію на стороні Tauri. У момент досягнення певного об'єму або закінчення запису, Tauri запускає локальну модель Whisper-rs, яка виконує транскрипцію української мови. Результати обробки передаються назад у NuxtJS через Tauri Chanel API, що дозволяє клієнту асинхронно підписатися на події отримання тексту. Нижче наведений приклад коду на Rust, який в окремому потоці оброблює аудіо фрагменти в текст.

```
#[tauri::command]
pub fn start_transcription<'a, 'b>(state: State<'_, TranscriptionState<'a, 'b>>,
on_transcript: Channel<TranscriptedSegment> ) -> Result<(), String> {
    let mut recording = state.is_recording.lock().unwrap();
    if *recording {
        return Err("Already recording".to_string());
    }
    *recording = true;
    let recording_flag = Arc::clone(&state.is_recording);
    let buffer = Arc::clone(&state.buffer);
    let whisper= Arc::clone(&state.whisper);
    let params_lock = Arc::clone(&state.params);
    std::thread::spawn(move || {
        let whisper = whisper.lock().unwrap();
        let whisper_ctx = whisper.as_ref().expect("Whisper Model not initialized");
        let mut whisper_state = whisper_ctx.create_state().expect("Failed to create
whisper state");
        while *recording_flag.lock().unwrap() {
            std::thread::sleep(std::time::Duration::from_millis(100));
            let mut audio_buffer = buffer.lock().unwrap();
            if audio_buffer.len() >= BUFFER_SIZE {

                let params = params_lock.lock().unwrap();
                whisper_state.full(params.clone(), &audio_buffer).expect("Failed to
process audio");
                let num_segments = whisper_state
                    .full_n_segments()
                    .expect("failed to get number of segments");
                for i in 0..num_segments {
                    let segment = whisper_state
                        .full_get_segment_text(i)
                        .expect("failed to get segment");
                    let start_timestamp = whisper_state
                        .full_get_segment_t0(i)
                        .expect("failed to get segment start timestamp");
                    let end_timestamp = whisper_state
                        .full_get_segment_t1(i)
                        .expect("failed to get segment end timestamp");
```

```
                    let result = TranscriptedSegment {
                        text: segment,
                        start: start_timestamp,
                        end: end_timestamp,
                    };
                    on_transcript.send(result).unwrap();
                }
                audio_buffer.clear();
            }
        }
    });
    Ok(())
}
```

Наступний важливий процес — семантична обробка тексту. Після завершення діалогу, лікар має транскрибований текст, який він може надіслати на локальну LLM-модель через Ollama-rs. Клієнт викликає команду *invoke('process_conversation',{conversation:string, on_process: Channel<string>})* після чого в Tauri використовується локальний інтерфейс до клієнту Ollama для обробки розмови та генерації попереднього діагнозу. Цей підхід гарантує повну автономність і конфіденційність обробки даних — жодна приватна інформація не покидає комп'ютер користувача, що особливо важливо в умовах медичних стандартів безпеки. Ключовим при використанні LLM є написання коректного промпту (prompt) - текстового запиту або інструкції, яку ми передаємо LLM, щоб вона сформувала відповідь.

```
#[tauri::command]
pub async fn process_conversation(conversation: String, on_process:
Channel<String>) -> Result<(), String>  {
    let model = "llama3.2".to_string();
    let prompt = format!(
        "Ти — медичний асистент. На основі транскрибованого діалогу між лікарем і
пацієнтом, структуруй інформацію у вигляді медичного звіту. Витягни ключову
інформацію та запиши її в наступному форматі:

    - Скарги: {{основні скарги пацієнта}}
    - Анамнез: {{історія хвороби або симптомів}}
    - Об'єктивні дані: {{вимірювання, спостереження лікаря}}
    - Діагноз (попередній): {{якщо згадано}}
    - Рекомендації: {{лікування, обстеження, поради}}

    Текст діалогу:
```

```
    \"\"\"{}\"\"\"

    Сформуй відповідь у вигляді структурованого звіту українською мовою. Не додавай
зайвих пояснень.", conversation);

    let ollama = Ollama::default();
    let mut stream = ollama.generate_stream(GenerationRequest::new(model,
prompt)).await.expect("Failed to find Ollama, is it started on this device?");

    while let Some(res) = stream.next().await {
        let responses = res.unwrap();
        for resp in responses{
            on_process.send(resp.response.to_string());
            println!("{}", resp.response.to_string());
        }
    }

    Ok(())
}
```

Передача результатів на бекенд здійснюється окремими запитами: NuxtJS надсилає транскрипцію, медичний запис, дані пацієнта, тощо через REST API до серверної частини. Тут обробка вже реалізована через патерн Service-Repository. Наприклад, ми передаємо HTTP запит на endpoint для створення транскрипції, FastAPI викликає відповідний метод з MedRecordService сервісу, який в свою чергу викликає метод *create()* з TranscriptionRepository, передаючи провалідовані дані з запиту. Репозиторій з'єднується з БД, та виконує INSERT запит, після чого повертає відповідний ORM клас з даними, який знову валідується і повертається на клієнт через HTTP протокол. Більш детальний процес по обробці запиту на бекенд-сервері зображено на рис. 11.

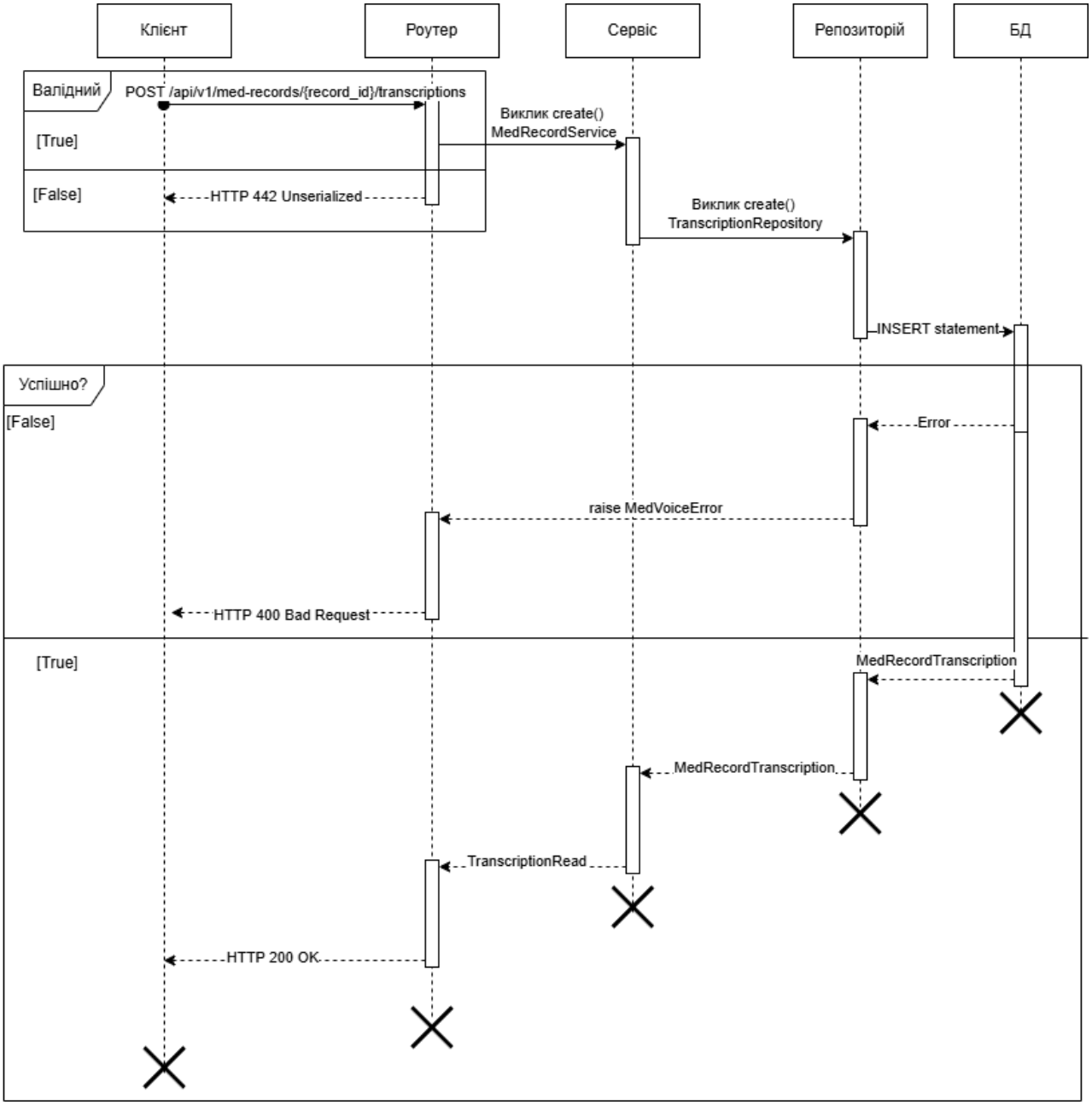

Рис. 11 Діаграма послідовності для запиту на збереження транскрипції

У клієнтській частині розроблено окремі алгоритми для зручної інтеграції інтерфейсу: useAudioRecorder.ts відповідає за запис мовлення та показ транскрипції в реальному часі, useWhisper.ts — за ініціалізацію моделі Whisper на стороні Tauri, а useOllama — за передачу транскрибованого тексту до LLM, та отримання форматованого тексту. Ці компоненти взаємодіють з Tauri через API виклики та події каналів, дозволяючи отримати максимально плавний UX. Нижче наведений фрагмент коду useAudioRecorder.ts

```
export const useAudioRecorder = () => {
  const toast = useToast()
```

```
  const isRecording = useState('isRecording', ()=> false)
  const onTranscript = new Channel<TranscriptionFragment>()
  const fragments = useState("fragments", () => [] as Array<TranscriptionFragment>)
  const audioBlob = useState("audioBlob", ()=> undefined)
  const audioURL = ref<string>()

  let mediaRecorder: RecordRTC

  onTranscript.onmessage = (fragment) => {
    if (!fragment.text.includes("[BLANK_AUDIO]") ||
!fragment.text.includes("[музика]")){
      fragments.value.push(fragment)
    }
  }

  const initRecorder = async () => {
    console.log('Recorder inited')
    const stream = await navigator.mediaDevices.getUserMedia({ audio: true });

    mediaRecorder = new RecordRTC(stream, {
      type: "audio",
      mimeType: "audio/wav",
      recorderType: RecordRTC.StereoAudioRecorder,
      numberOfAudioChannels: 1,
      timeSlice: 1000,
      desiredSampRate: 16000,
      bufferSize: 4096,
      audioBitsPerSecond: 128000,
      ondataavailable: async (blob) => {
        const audioBuffer = await blob.arrayBuffer()
        const audioContext = new AudioContext({sampleRate: 16000})
        const audioData = await audioContext.decodeAudioData(audioBuffer)
        const chunk = new Float32Array(audioData.getChannelData(0))
        invoke("process_chunk", {chunk: Array.from(chunk)}).catch((error) => {
          useToast().add({
            title: "Error while processing chunk",
            description: error.toString(),
            color: "error"
          })
        })
      }
    })
  }

  const startRecording = async () => {
    invoke("start_transcription", {onTranscript: onTranscript})
    .then(()=>{
      mediaRecorder.startRecording()
      isRecording.value = true
```

```
    })
    .catch((error) => {
      toast.add({
        title: "Error while starting transcription",
        description: error.toString(),
        color: "error"
      })
    })
  }

  const stopRecording = async () => {
    if(isRecording){
      mediaRecorder.stopRecording(function () {
        const blob = mediaRecorder.getBlob()
        audioBlob.value = blob
        audioURL.value = mediaRecorder.toURL()
      })
      invoke('stop_transcription').then(()=>{
        isRecording.value = false
      }).catch(error =>{
        isRecording.value = false
        toast.add({
          title: 'Error when stop recording',
          description: error.toString(),
          color: 'error'
        })
      })
    }
  }

  const resetRecorder = () => {
    audioBlob.value = undefined as Blob | undefined
    audioURL.value = undefined as string | undefined
    fragments.value = []
  }

  return {initRecorder, startRecording, stopRecording, resetRecorder, audioBlob,
audioURL, isRecording, fragments}

}
```

У цілому така організація — з переміщенням інтелектуальної обробки на клієнт — дозволяє створити офлайн-сумісне, безпечне й легке для розгортання рішення без потреби у потужному серверному кластері. Завдяки цьому архітектурному підходу система може функціонувати навіть у віддалених

регіонах без стабільного інтернету, що особливо актуально для первинної медичної допомоги.

4 ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ

4.1 Тестування

Тестування — це невід'ємна частина процесу розробки програмного забезпечення, що дозволяє перевірити коректність, надійність та стабільність роботи системи. Основна мета тестування — виявити помилки на ранніх етапах, зменшити витрати на підтримку та забезпечити високу якість продукту. Сучасна практика розробки ПЗ передбачає використання автоматизованих тестів, які дозволяють перевіряти поведінку окремих компонентів системи та взаємодію між ними. Одним з підходів до організації тестування є розділення його на рівні: модульне, інтеграційне, системне та приймальне.

Модульне тестування (unit testing) спрямоване на перевірку окремих функцій, класів або методів без врахування їх залежностей. Тестується лише логіка окремого компонента в ізоляції. Це дозволяє швидко виявляти помилки саме в реалізації окремих одиниць коду. Для прикладу, у даній системі доцільно створити юніт-тести для сервісів, репозиторіїв, моделей. У таких тестах зазвичай замінюються зовнішні залежності, як-от база даних чи зовнішній API, що дозволяє зосередитися на логіці самого сервісу [27].

Інтеграційне тестування (integration testing) перевіряє, як різні компоненти системи взаємодіють між собою. У контексті веб-додатків це, зокрема, означає перевірку повноцінних HTTP-запитів до API. У прикладі даної системи тестування маршруту "*/api/v1/auth/login"* є інтеграційним, оскільки перевіряє, чи правильно працює взаємодія між веб-маршрутом, сервісним шаром, базою даних та схемами валідації. Такий тип тестів дозволяє виявити проблеми, які не видно на рівні окремих модулів[28].

Mocking (мокінг) — це техніка, яка дозволяє створювати штучні замінники (mocks) для залежностей, які зазвичай взаємодіють із зовнішніми ресурсами, наприклад з базою даних, файловою системою або API. Моки дозволяють

ізолювати тестований об’єкт, забезпечуючи повний контроль над поведінкою залежностей. Наприклад, під час тестування “AuthService”, замість реальної бази даних може використовується мок об’єкта репозиторію користувачів, що дозволяє точно моделювати різні сценарії (успішний логін, помилкові дані, тощо). Приклад коду тестування “AuthService” див. ДОДАТОК Б.

Для реалізації тестування в Python широко використовується бібліотека **pytest** — популярний інструмент, який підтримує як юніт, так і інтеграційні тести. pytest надає зручний синтаксис, можливість використання фікстур, параметризації тестів та потужну систему плагінів. Також pytest добре інтегрується з FastAPI, дозволяючи створювати тестові клієнти та виконувати запити до API. Це особливо важливо у системах, побудованих на сервісній архітектурі, де важливо тестувати взаємодію між шарами [29].

Таким чином, у даному проєкті було реалізовано два підходи до тестування: юніт-тести для бізнес-логіки (“AuthService”, “CRUDRepository” тощо), та інтеграційні тести для перевірки API-ендпойнтів. Поєднання цих підходів дозволяє досягти високої надійності системи та швидко локалізовувати помилки в разі їх виникнення на етапі експлуатації.

4.1.1 Тестування Whisper моделей

Тестування автоматичного розпізнавання мовлення є ключовим етапом перевірки якості роботи мовної технології. Для цього використовуються метрики точності, зокрема Word Error Rate (WER) — загальноприйнятий показник, який показує відсоток помилок у результатах розпізнавання мовлення [30]. Метрика WER визначається за формулою:

$$WER = \frac{S + D + I}{N}$$

- S – к-ть замінених слів
- D – к-ть пропущених слів
- I – к-ть зайвих слів
- N – Загальна кількість слів у еталонному тексті

Окрім WER метрики, також необхідно порівнювати час, за який було оброблено фрагмент тексту. Для даної системи було проведено ряд тестів, для відповідних Whisper моделей.

Спочатку було записано аудіо-фрагмент тексту (Див. ДОДАТОК В), який для кожної моделі був однаковий, та за допомогою секундоміра, вимірювався час обробки аудіо. Потім, отриманий текст від моделі порівнювався до еталонного за допомогою WER метрики, для цього була створена утиліта на Python за допомогою бібліотеки «jiwer» [31].

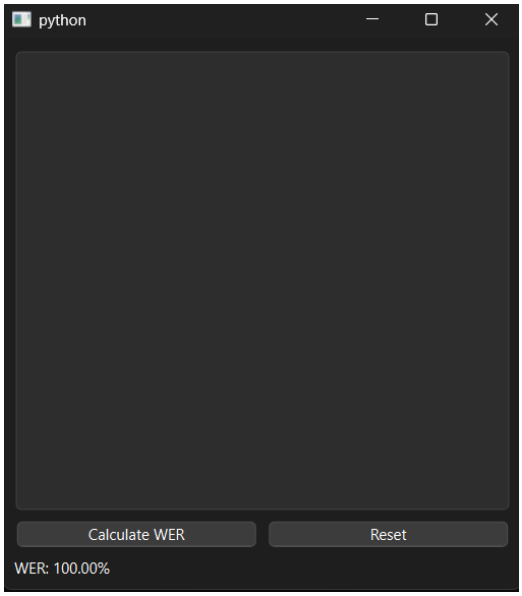

Рис. 12 Утиліта для отримання значення WER

В результаті отримали порівняльну таблицю

Модель	Розмір моделі	Час обробки	Час аудіо-фрагменту	Різниця	WER
Tiny	75 МБ	72 с.	72 с.	0 с.	89,33%
Base	143 МБ	100 с.	72 с.	28 с.	73,03%
Small	466 МБ	82 с.	72 с.	10 с.	62,92%
Medium	1,5 ГБ	172 с.	72 с.	100 с.	50,00%
Large V1	2,9 ГБ	310 с.	72 с.	238 с.	33,71%
Large V2	2,9 ГБ	300 с.	72 с.	228 с.	29,11%
Large V3	2,9 ГБ	286 с.	72 с.	214 с.	28,65%
Large V3 Turbo	1.62 ГБ	180 с.	72 с.	108 с.	30,34%

Таблиця 2 WER-метрика Whisper моделей

Як бачимо, найвища точність була досягнута при використанні моделі Large V3, проте вона потребує більше часу на обробку. Для реального використання, оптимальним є компроміс між швидкістю і точністю, середній час обробки між всіма моделями 187 с., а середня точність 44,12 %, тоді найоптимальнішим варіантом буде модель Large V3 Turbo. Всі тести проводились на локальній системі з процесором AMD Ryzen 7 4700U, 16 ГБ ОЗП без використання GPU.

4.2 Вимоги до апаратного та програмного забезпечення

Система підтримує два основні режими розгортання: локальний автономний та серверно-клієнтський. У першому випадку всі елементи — інтерфейс, транскрипція, сумаризація, база даних та API — працюють на одному пристрої. Такий підхід особливо зручний у невеликих клініках або в умовах виїзного прийому. У другому варіанті API-сервер та БД можуть бути винесені на локальний сервер медичної установи. Клієнтські пристрої лікарів у такому випадку підключаються через внутрішню мережу, отримуючи доступ до загальної бази пацієнтів і звітів.

Для ефективної роботи системи мовної реєстрації результатів медичних обстежень необхідно визначити мінімальні вимоги до апаратного та програмного забезпечення. Враховуючи використання локального розпізнавання мовлення (через Whisper-rs) та локальної LLM для генерації медичних звітів (через Ollama-rs), система має певні ресурсоємні компоненти. З цієї причини рекомендується використовувати сучасні комп’ютери з достатньою обчислювальною потужністю, особливо при локальному використанні.

Апаратні вимоги у випадку повністю автономного використання на одному комп’ютері такі: процесор із принаймні 4 фізичними ядрами (рекомендовано — 6 або більше), оперативна пам’ять від 16 ГБ, наявність SSD-диску з вільним простором не менше 50 ГБ для зберігання локальних моделей та бази даних, а також бажано наявність GPU (NVIDIA з підтримкою CUDA), що значно прискорює розпізнавання мовлення та сумаризацію.

У випадку розгортання на сервері, клієнтські пристрої можуть мати менші характеристики щодо постійної пам’яті, оскільки БД та API-сервер будуть на окремому сервері. Сам сервер повинен мати підключення до локальної мережі, процесор з 4 фізичними ядрами, оперативна пам’ять від 8 ГБ, наявність SDD-диску не менше як 100ГБ.

З програмного забезпечення, для повноцінної роботи клієнту, необхідно завантажити Ollama-клієнт(https://ollama.com/), який відповідає за використання локальної LLM. Для серверу, необхідно встановити, Docker, систему контейнеризації.

Для візуалізації розгортання системи доцільно використовувати діаграму розгортання (Deployment Diagram) з UML, яка демонструє взаємозв’язки між вузлами — наприклад, між локальним комп’ютером лікаря, сервером установи, та базою даних.

Завдяки гнучкій архітектурі, система легко адаптується під різні масштаби впровадження: від індивідуального кабінету лікаря до цілої поліклініки. Усі компоненти працюють офлайн, без необхідності підключення до хмарних сервісів, але бажано мати доступ до інтернету, для завантаження клієнту Ollama та моделей Whisper.

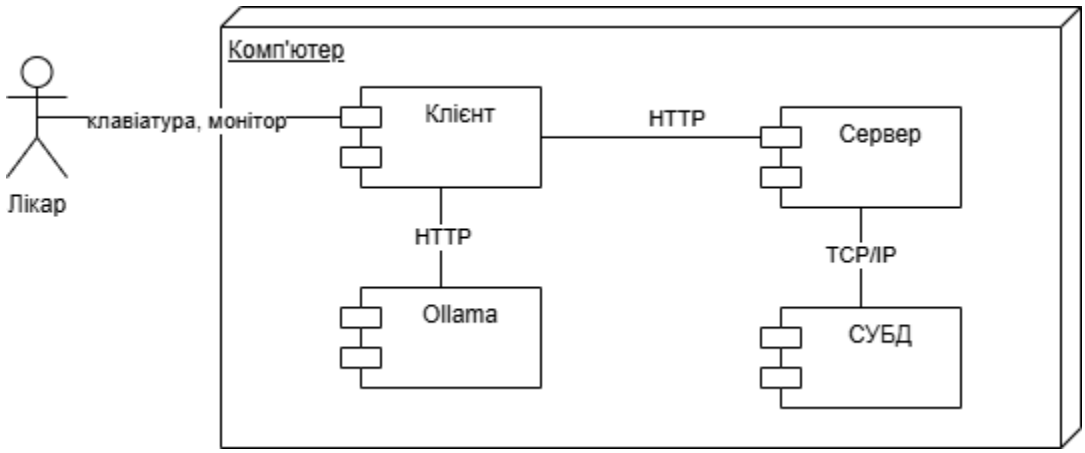

Рис. 13 Діаграма розгортання(локально-автономний режим)

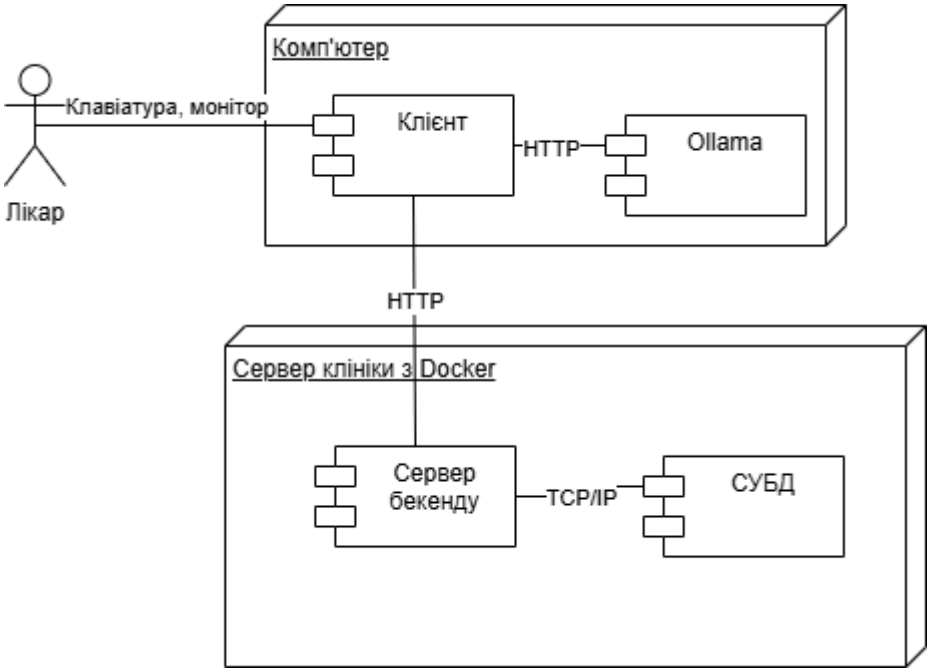

Рис. 14 Діаграма розгортання(серверний режим)

4.3 Перевірка якості програмного продукту

При запуску додатку, ми потрапляємо на екран авторизації зображений на рисунку 13, де потрібно ввести Email чи номер телефону та пароль.

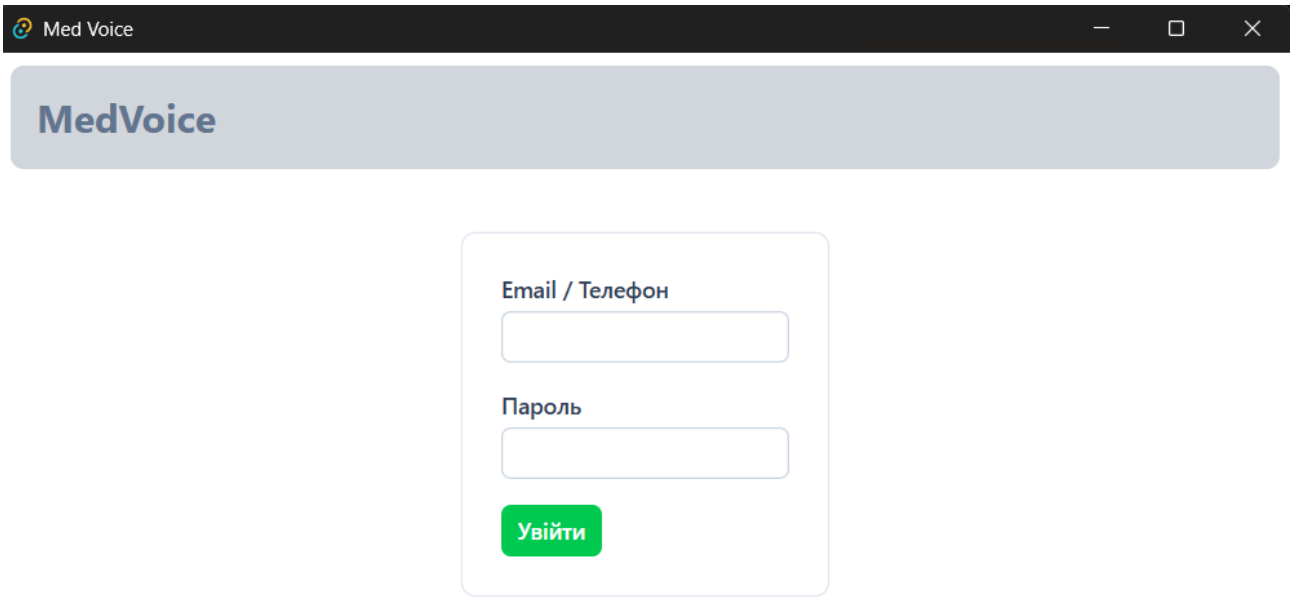

Рис. 15 Екран входу в систему

Після входу в систему, ми потрапляємо на головний екран зображений на рисунку 14, тут ми можемо бачити меню навігації та календар лікаря, з

відміченими записами, якщо натиснути на запис, то ми перейдемо на відповідну сторінку.

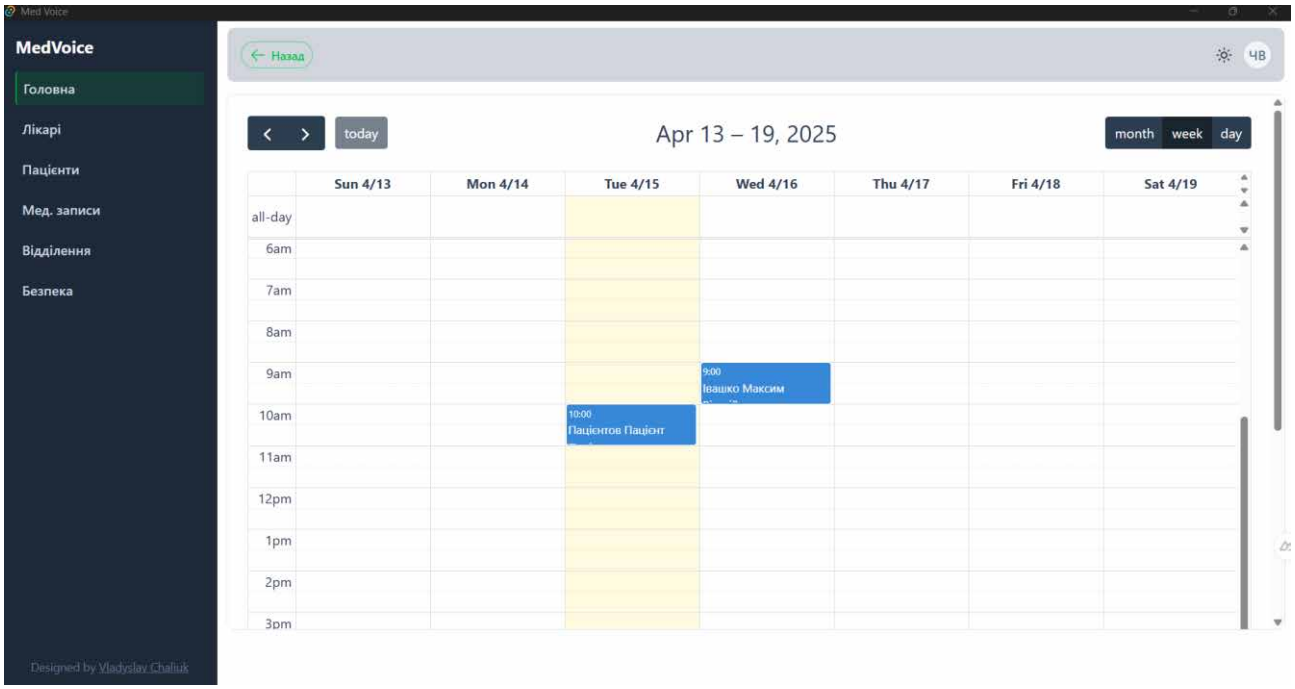

Рис. 16 Головний екран

На екрані «Лікарі» ми бачимо список лікарів у системі, та кнопку додати лікаря.

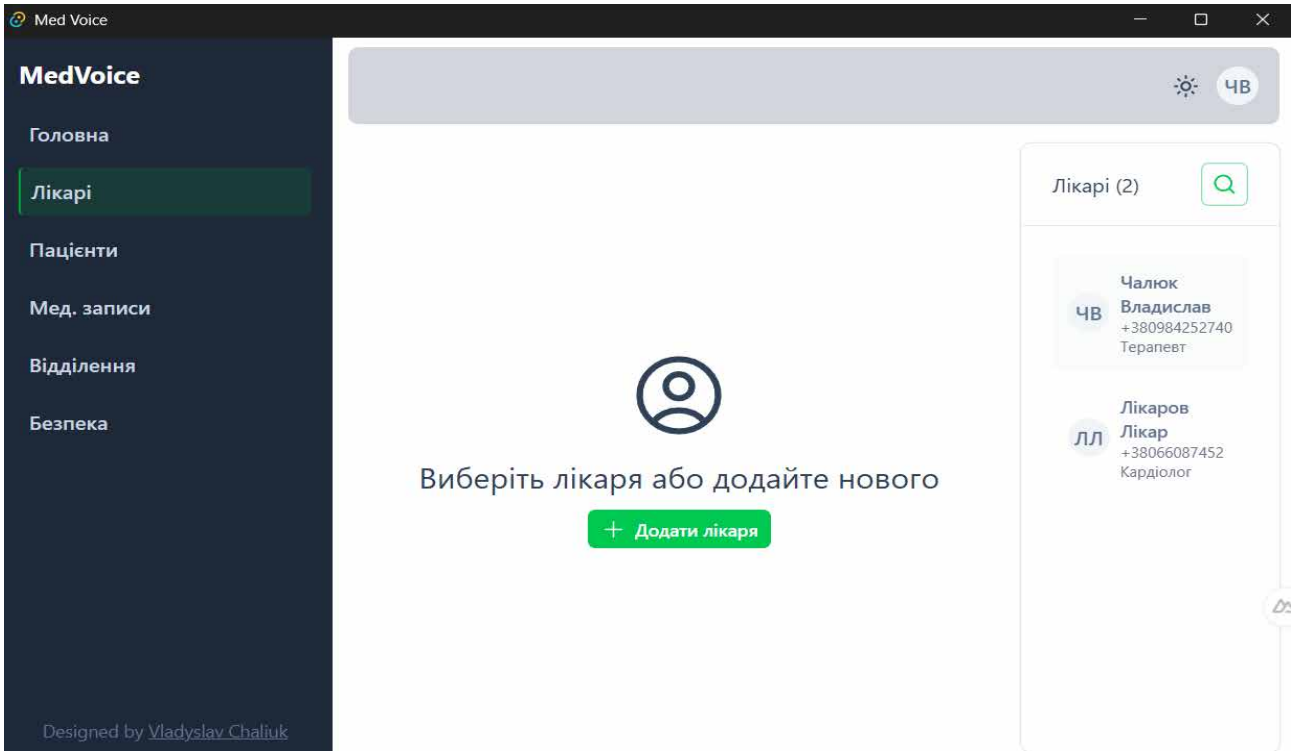

Рис. 17 Екран лікарів

Якщо натиснути кнопку додати лікаря, то відкриється модальне вікно з формою, де ми можемо ввести дані нового лікаря, та додати його до системи.

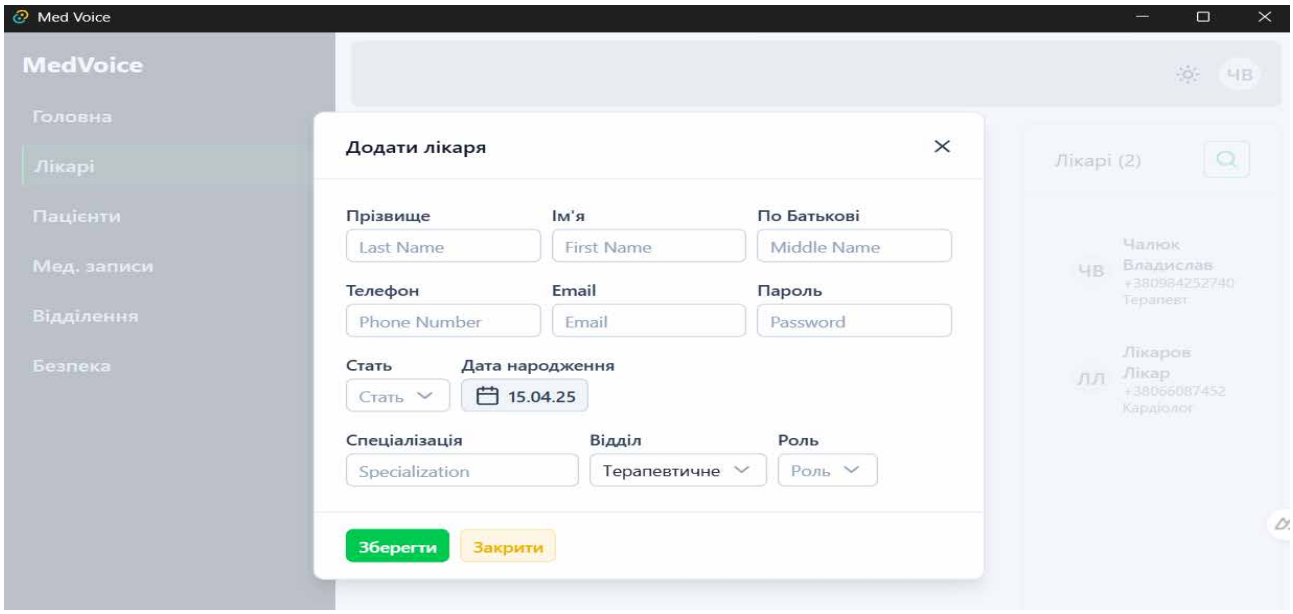

Рис. 18 Додати лікаря

Натиснувши на «Лікаря» зі списку, ми перейдемо на його детальну сторінку. На цій сторінці ми можемо бачити повну інформації про лікаря, його ПІБ, контактні дані, відділення, роль та дозволи в системі, а також вкладки з його пацієнтами, та медичними записами. Також ми бачимо кнопки які дозволяють нам редагувати чи видалити лікаря, в залежності від наших дозволів у системі.

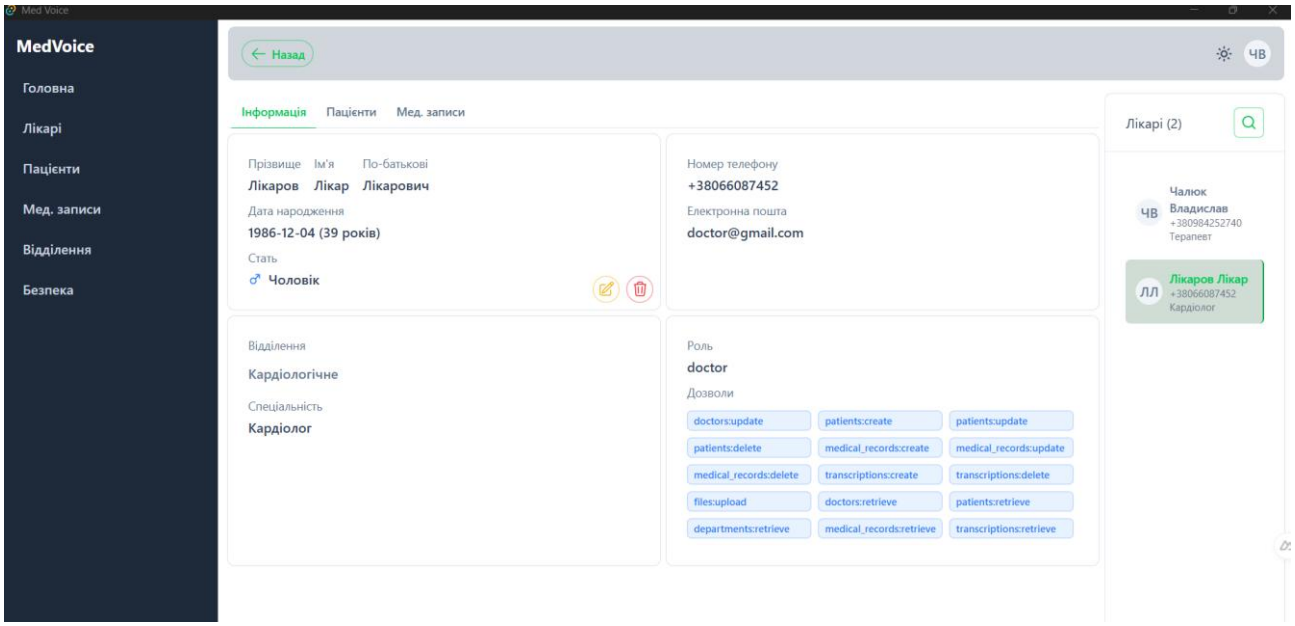

Рис. 19 Екран інформації про конкретного лікаря

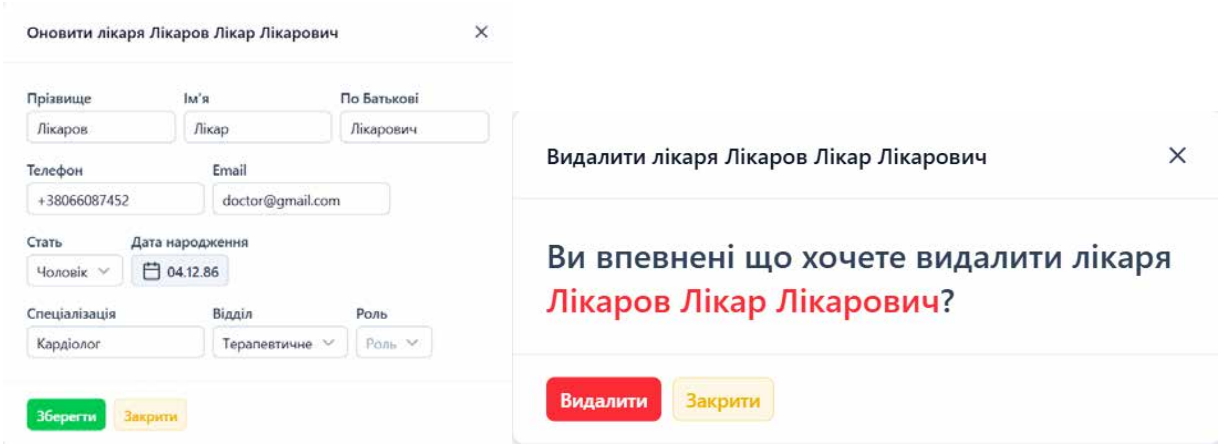

Рис. 20 Форми редагування та видалення лікаря

Натиснувши вкладку «Пацієнти» в лівому головному меню, ми перейдемо на аналогічний екран, але з списком пацієнтів, де ми можемо також відкрити модальне вікно, щоб додати нового пацієнта, або перейти на його сторінку.

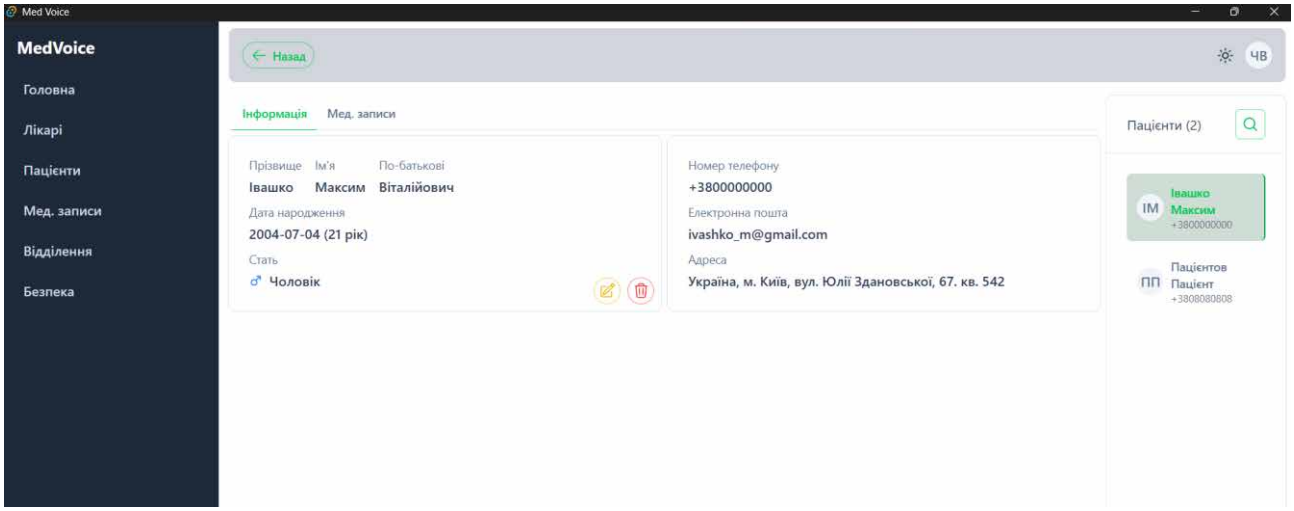

Рис. 21 Екран з даними конкретного пацієнта

Натиснувши на вкладку «Мед. Записи», ми перейдемо до таблиць зі списком медичних записів лікаря, та всіх записів в системі. Тут ми можемо призначити, редагувати, видалити, чи перейти до конкретного мед. запису

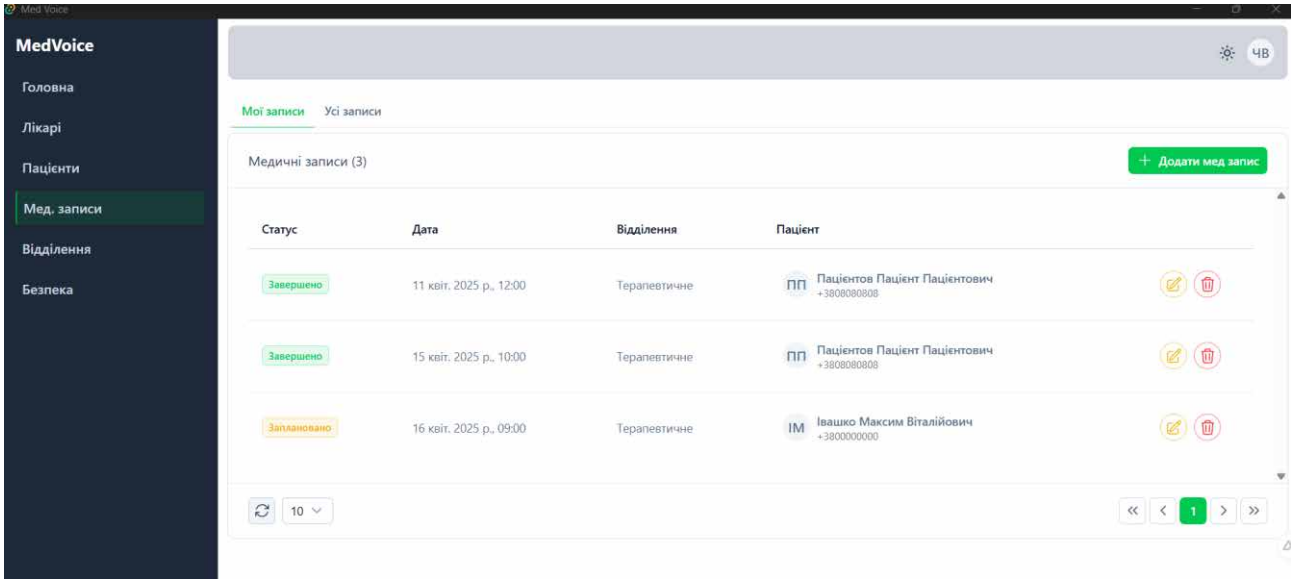

Рис. 22 Екран медичних записів

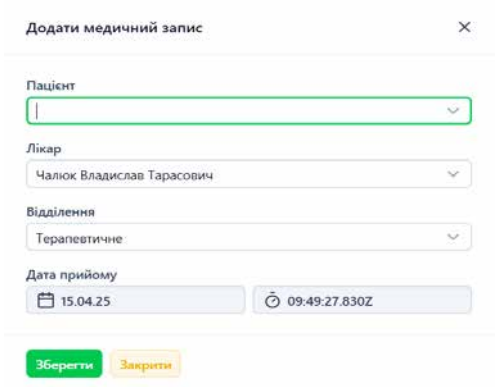

Рис. 23 Форма призначення запису

Натиснувши на конкретний запис, ми перейдемо на його сторінку, де можемо бачити більше інформації, а також переглядати та створювати транскрипції, генерувати медичний звіт.

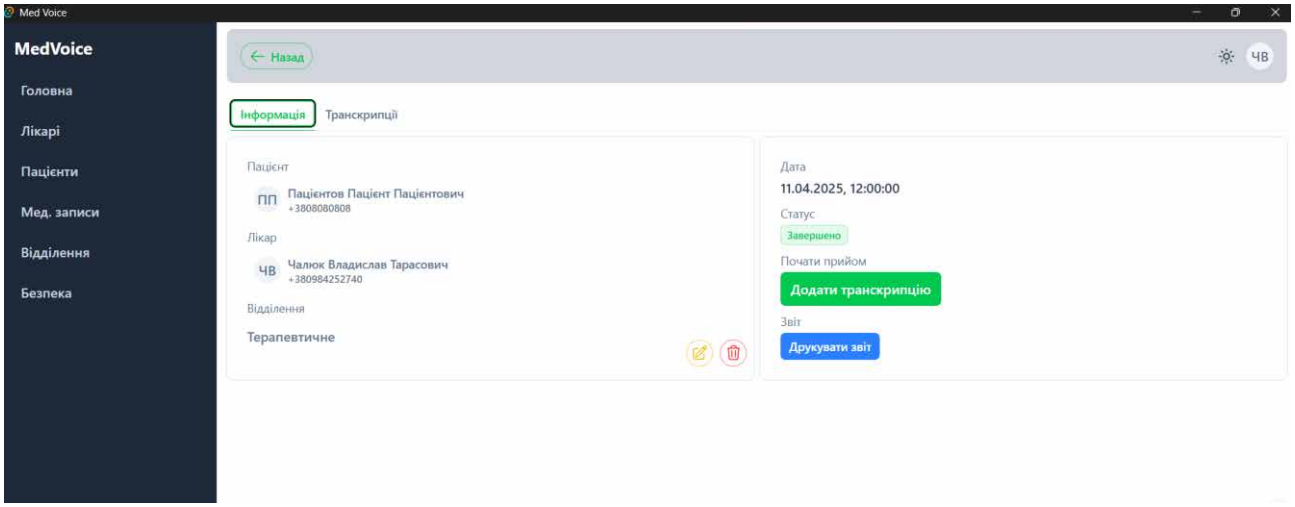

Рис. 24 Екран конкретного мед. Запису

Натиснувши на кнопку «Додати транскрипцію» ми перейдем на екран транскрипції. Де ми можемо вибрати Whisper модель, мову, к-ть потоків для обробки мовлення в реальному часі, також ми можемо почати самий запис, та бачити транскрибований текст, який можемо редагувати в ручну, або за допомогою LLM, натиснувши кнопку підсумувати.

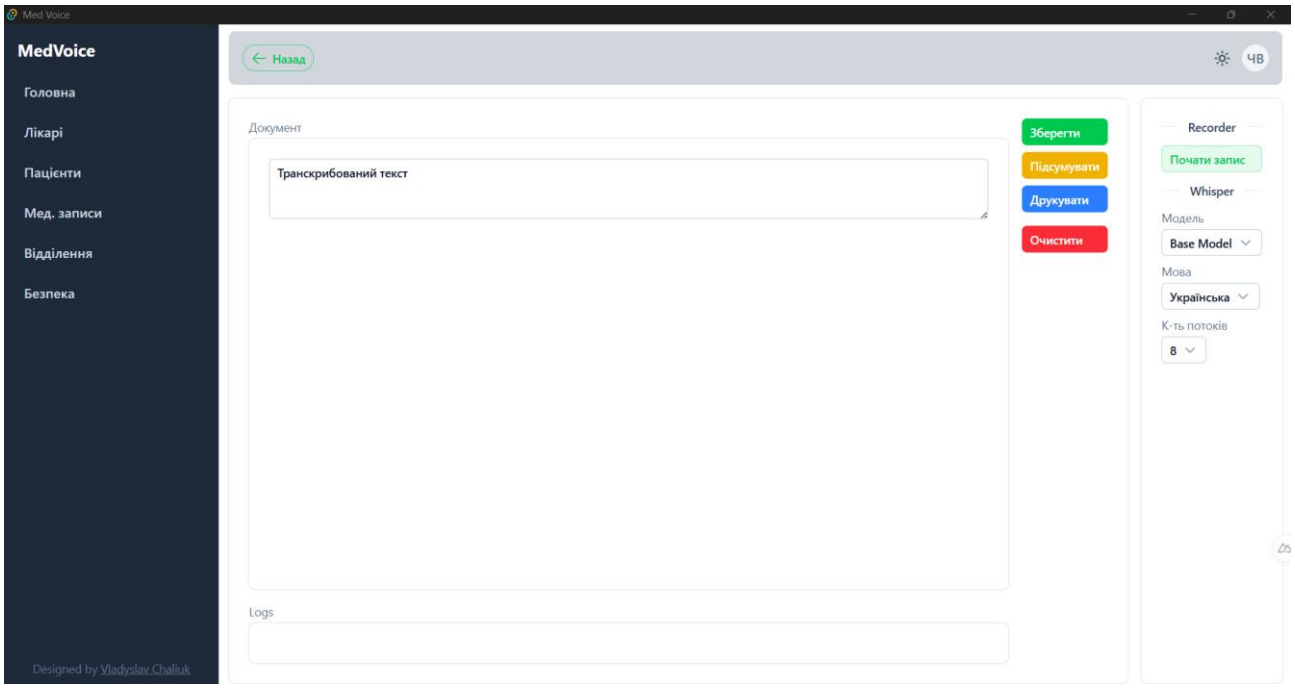

Рис. 25 Екран створення транскрипції

Вибравши вкладку транскрипції на сторінці медичного запису, ми можемо бачити таблицю з транскрипціями, де можемо як прослухати аудіозапис розмови, так і переглянути транскрибований текст.

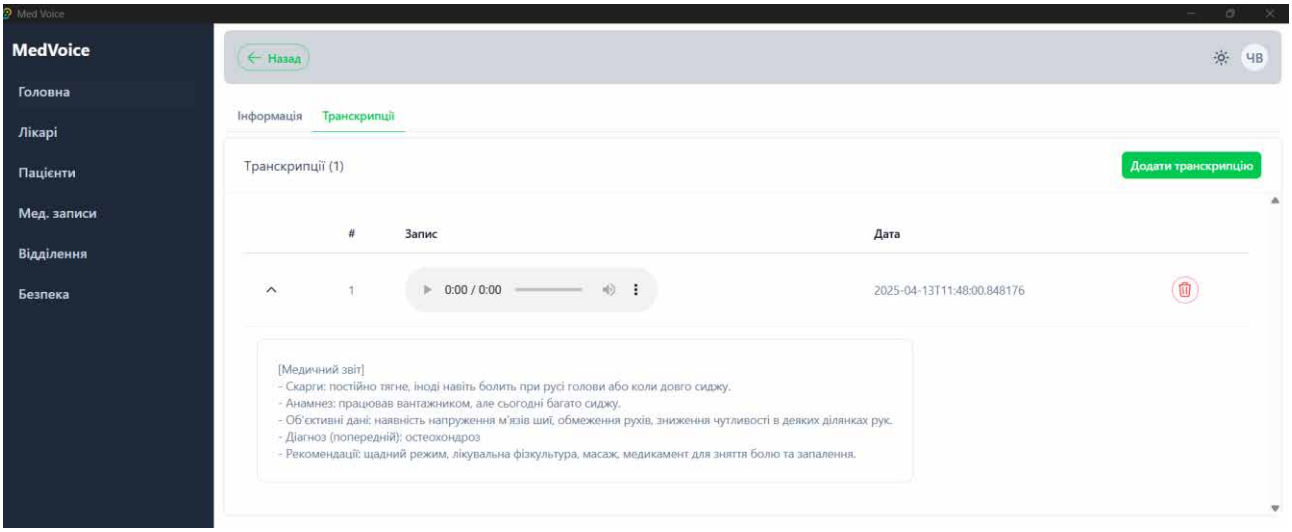

Рис. 26 Таблиця транскрипцій мед. Запису

Натиснувши «Друкувати звіт», ми отримаємо PDF файл з даними лікаря, пацієнта та транскрибованим текстом.

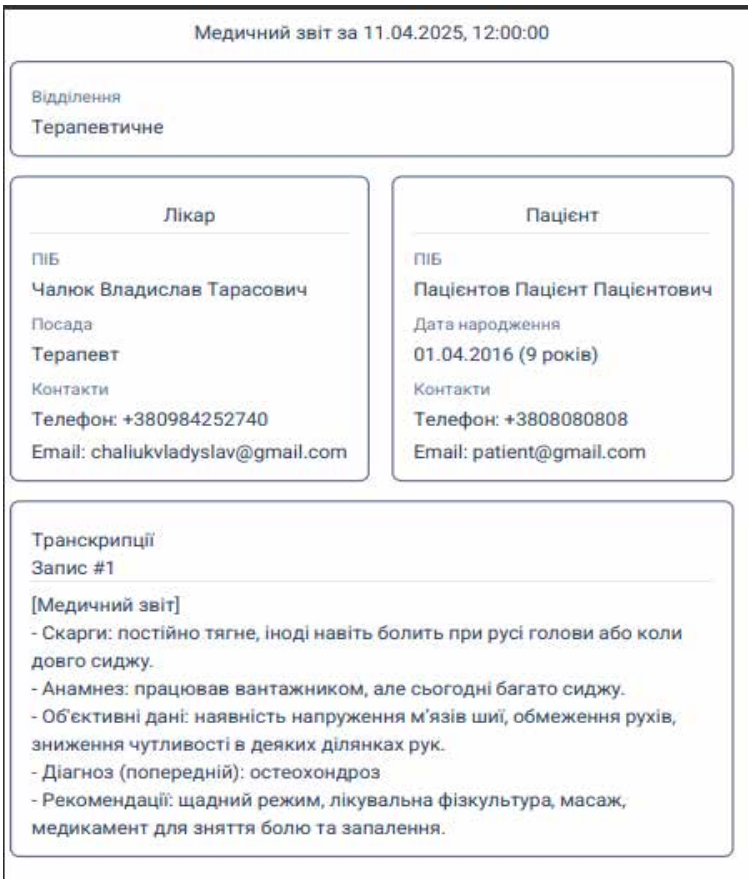

Рис. 27 Приклад згенерованого звіту

Також в нас є вкладка з списком відділень клініки, тут ми можемо переглянути відділення, та які лікарі прив’язані до конкретного відділення.

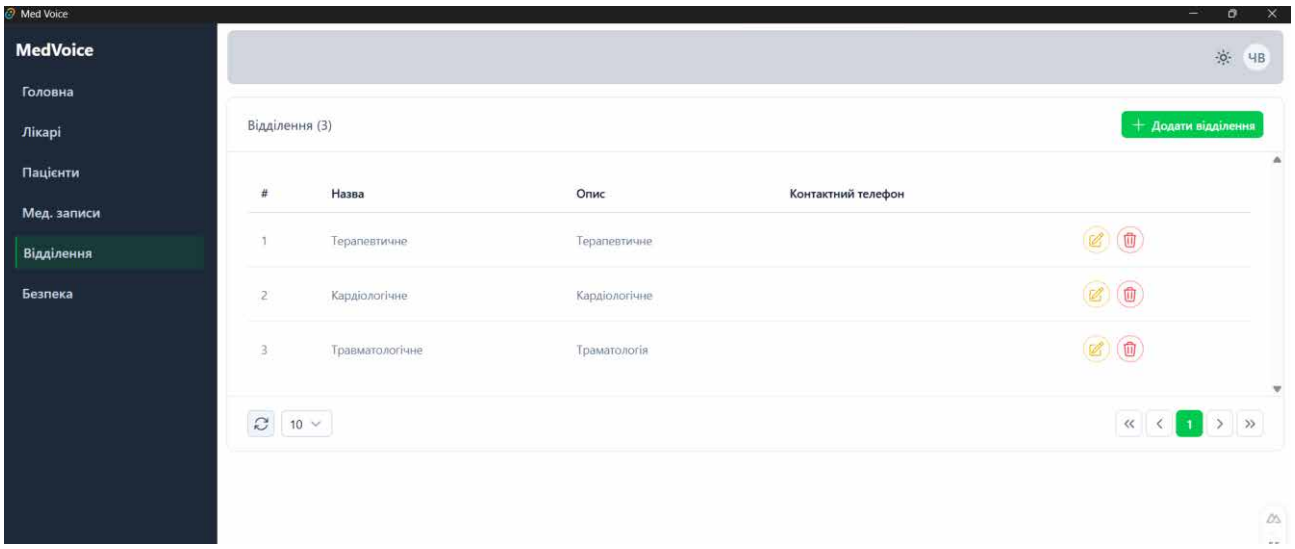

Рис. 28 Екран зі списком відділень

У вкладці безпека, яку можуть відкрити лише ті, хто мають доступ, має вкладки ролі та дозволи, з відповідними таблицями. В цій вкладці, можна створювати нові ролі, та дозволи для системи, щоб мати більш точне управління доступом.

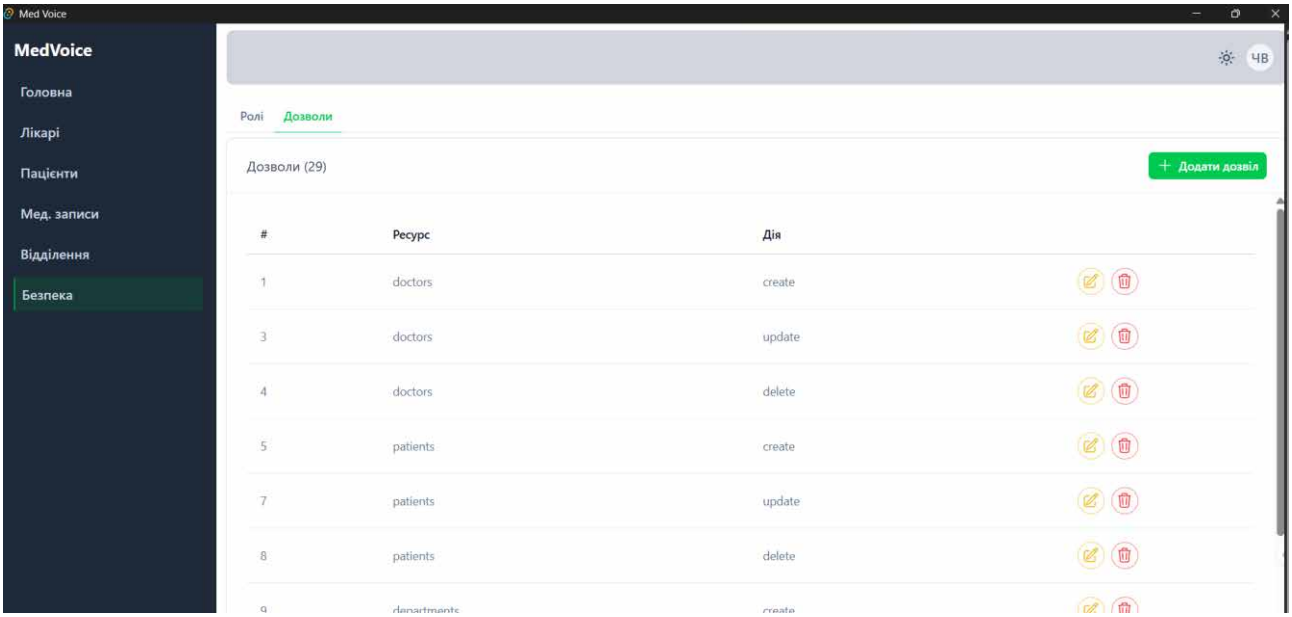

Рис. 29 Екран зі списком дозволів

4.4 Інсталяційний пакет

У рамках розробки програмного забезпечення для інформаційної технології було створено інсталяційні пакети, які дозволятимуть користувачеві швидко встановити систему на локальний комп'ютер або в межах локальної мережі медичної установи. Інсталяційний пакет містить два основні компоненти: бекенд (серверну частину) та клієнтську частину з графічним інтерфейсом, побудовану на основі фреймворку Tauri і NuxtJS.

Оскільки API-сервер реалізовано мовою програмування Python з використанням FastAPI як веб-фреймворку, то для створення виконуваного файлу та подальшого розповсюдження серверної частини було використано утиліту PyInstaller. Вона дозволяє зібрати Python-додаток у вигляді самостійного .exe-файлу або іншого виконуваного пакету, що не потребує попередньої інсталяції інтерпретатора Python чи залежностей [32].

Клієнтська частина, яка побудована на Tauri, має власний компілятор що створює інсталяційний файл(Tauri Builder). Цей інсталятор дозволяє встановити графічний інтерфейс разом з усіма вбудованими модулями з мови rust, такими як WhisperRS для транскрипції аудіо та Ollama-rs для локального з'єднання з LLM.

Для автоматизації збирання, тестування та публікації нових версій було використано систему контролю версій GitHub, та їх CI/CD платформу GitHub Actions [33].

CI/CD – набір практик і автоматизованих процесів, що застосовуються у розробці програмного забезпечення для забезпечення постійного та надійного постачання оновлень.

CI (безперервна інтеграція) означає, що кожна зміна в коді автоматично перевіряється за допомогою тестів і збирається в єдину гілку репозиторію, що дозволяє розробникам швидко виявляти помилки.

CD (безперервна доставка або розгортання) — це наступний етап після CI. У варіанті Continuous Delivery готові до релізу версії автоматично проходять усі

стадії збірки, тестування та потрапляють у репозиторій релізів, звідки можуть бути розгорнуті вручну [34].

У репозиторії проєкту налаштовано робочі процеси, що автоматично виконують тестування коду, створення PyInstaller-збірки для бекенду, Tauri-збірки для клієнтської частини, а також створення релізів та публікацію інсталяційних пакетів. Це дозволяє забезпечити стабільну поставку програмного забезпечення кінцевому користувачу без помилок та з збереженням повної історії змін.

Завдяки такому підходу користувач отримує готові інсталятори, які можуть бути встановлені на звичайний комп’ютер без складної конфігурації. Це особливо важливо в медичних установах, де обмежено доступ до технічної підтримки чи зовнішніх онлайн-ресурсів.

Рис. 30 Сторінка з інсталяційними пакетами в GitHub

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи було спроєктовано та реалізовано інформаційну технологію мовної реєстрації результатів медичних обстежень, здатну працювати повністю локально без підключення до зовнішніх сервісів. Система дозволяє лікарю записувати розмову з пацієнтом за допомогою голосу, отримувати транскрибований текст у реальному часі, автоматично генерувати медичні записи за допомогою вбудованої мовної моделі, а також зберігати ці дані у локальну базу даних. Усі етапи проєктування, включаючи побудову діаграм UML, логічне моделювання бази даних, реалізацію API та створення настільного клієнта, були реалізовані відповідно до вимог технічного завдання. Негативні моменти, такі як складність у налаштуванні взаємодії між клієнтом Tauri та мовними модулями, були вирішені в процесі тестування та оптимізації структури інтерфейсів.

Запропонована система може використовуватися в закладах охорони здоров’я для автоматизації процесу заповнення медичних карток, зменшення навантаження на лікарів, а також підвищення точності ведення записів. Автономність системи дозволяє застосовувати її у лікарнях, де відсутній стабільний доступ до Інтернету або де є підвищені вимоги до безпеки персональних даних пацієнтів. Завдяки використанню сучасних мовних моделей та ефективної взаємодії між клієнтом і сервером, система демонструє швидкий відгук, достатню точність розпізнавання та зручність у роботі.

Усі цілі, поставлені у технічному завданні, були досягнуті. Порівняння результатів із сучасними аналогічними системами, зокрема хмарними рішеннями, показує перевагу системи у локальності, гнучкості налаштування, а також у відсутності потреби в зовнішніх API, що скорочує витрати на обслуговування. Техніко-економічна ефективність полягає у скороченні часу на заповнення медичних документів та зниженні потреби в додатковому ПЗ.

Основне нове технічне рішення, покладене в основу роботи, — це побудова повністю локальної мовної системи обробки медичних даних із застосуванням Whisper-rs та Ollama-rs на стороні клієнта, без використання сторонніх API. Запропонована взаємодія між NuxtJS і Tauri через `invoke` та `channel` дозволила реалізувати реальний потік аудіоданих та обробку без затримок. Інтеграція FastAPI та SQLAlchemy забезпечила зручний API-доступ до бази даних, а PostgreSQL виступає надійною СУБД для зберігання даних.

Серед авторських рішень варто виділити власну логіку передачі аудіофрагментів між фронтендом і ядром, реалізацію ізольованої обробки кожного запиту в клієнті. Окрім того, створено структуру розгортання, пакетування ПЗ та інсталятор для кінцевих користувачів.

У перспективі доцільно розширити функціонал системи — наприклад, реалізувати підтримку медичних шаблонів, підключити електронний підпис лікаря. Також варто дослідити можливість портативного використання системи на планшетах або вбудованих медичних терміналах. Усе це дозволить ще більше підвищити ефективність та практичну користь створеної системи для закладів охорони здоров’я.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hodson, Tobias, і Enrico Coiera. «Risks and benefits of speech recognition for clinical documentation: a systematic review». *Journal of the American Medical Informatics Association* 23, вип. 1 (2016): 169–179. https://doi.org/10.1093/jamia/ocv152.
2. Everett, M, J Redner, A Kalenscher, D Durso, і S Nguyen. «Speech recognition technology for increasing nursing documentation efficiency». *Online Journal of Nursing Informatics (OJNI)* 2, вип. 26 (2022). https://www.himss.org/resources/online-journal-nursing-informatics.
3. IBM – Speech Recognition. URL: https://www.ibm.com/think/topics/speech-recognition
4. OpenAI. «Whisper: Robust Speech Recognition via Large-Scale Weak Supervision.», б. д. https://openai.com/index/whisper/.
5. «AI Medical Scribe». DeepScribe: Сан-Франциско, Каліфорнія, б. д. https://www.deepscribe.ai/.
6. «Augmedix Live». Augmedix, б. д. https://www.augmedix.com/.
7. «Dragon Medical One». Microsoft, б. д. https://www.microsoft.com/en-us/health-solutions/clinical-workflow/dragon-medical-one.
8. Про захист персональних даних : Закон України від 01.06.2010. № 2297–VI. https://www.president.gov.ua/documents/2297vi-11567.
9. Booch, G, J Rumbaugh, і I Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley., 2005.
10. Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley., 2004.
11. «UML», 2017. https://www.omg.org/spec/UML.
12. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley., 2003.
13. Teorey, T. J., Lightstone, S. S., Nadeau, T. *Database Modeling and Design: Logical Design*. Morgan Kaufmann., 2011.
14. «ERWin Data Modeler». Quest Software, б. д. https://www.erwin.com/products/erwin-data-modeler/.
15. Codd, E. F. *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM., 1970.
16. Coronel, C., Morris, S., і Rob, P. *Database Systems: Design, Implementation, & Management*. Cengage Learning., 2016.

17. «PostgreSQL». PostgreSQL Global Development Group, б. д. https://www.postgresql.org/about/.
18. Bass, Len, Paul Clements, і Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2012.
19. «Python». Python Software Foundation, б. д. https://www.python.org/.
20. Sebastián, Ramírez. «FastAPI», б. д. https://fastapi.tiangolo.com/.
21. Bayer, Michael. «SQLAlchemy», б. д. http://sqlalchemy.org/.
22. «NuxtJS», б. д. https://nuxt.com/.
23. «Tauri», б. д. https://v2.tauri.app/.
24. Tazz4843. «whisper-rs», б. д. https://github.com/tazz4843.
25. pepperoni21. «ollama-rs», б. д. https://github.com/pepperoni21/ollama-rs.
26. «jsPDF». Parallax Agency Ltd, б. д. https://github.com/parallax/jsPDF
27. Martin Fowler. Unit Testing, 2014. https://martinfowler.com/bliki/UnitTest.html
28. Дмитренко, Євген. «Інтеграційні тести на Python з використанням pytest та FastAPI». *dou.ua*, б. д. https://dou.ua/forums/topic/47260/.
29. «pytest», б. д. https://docs.pytest.org/en/stable/.
30. Wikipedia. «Word error rate», б. д. https://en.wikipedia.org/wiki/Word_error_rate.
31. «JiWER». Jitsi, б. д. https://jitsi.github.io/jiwer/.
32. David Cortesi. «PyInstaller», б. д. https://pyinstaller.org/en/stable/.
33. «GitHub Actions». GitHub Inc., б. д. https://github.com/features/actions.
34. Sten Pittet. «Continuous integration vs. delivery vs. deployment», б. д. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

ДОДАТОК А

SQL код, для створення БД системи.

```
CREATE TABLE public.departments (
department_id serial4 NOT NULL,
"name" varchar(100) NOT NULL,
description text NULL,
contact_phone varchar(15) NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT departments_pkey PRIMARY KEY (department_id)
);
CREATE UNIQUE INDEX ix_departments_name ON public.departments USING btree (name);
CREATE TABLE public.patients (
patient_id uuid NOT NULL,
email varchar(150) NULL,
address text NOT NULL,
first_name varchar(100) NOT NULL,
last_name varchar(100) NOT NULL,
middle_name varchar(100) NOT NULL,
phone_number varchar(15) NOT NULL,
birth_data date NOT NULL,
"gender" public."gender" NOT NULL,
is_active bool NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT patients_pkey PRIMARY KEY (patient_id)
);
CREATE UNIQUE INDEX ix_patients_email ON public.patients USING btree (email);
CREATE UNIQUE INDEX ix_patients_phone_number ON public.patients USING btree
(phone_number);
CREATE TABLE public.permissions (
permission_id serial4 NOT NULL,
resource varchar(100) NOT NULL,
"action" varchar(50) NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT permissions_pkey PRIMARY KEY (permission_id),
CONSTRAINT uq_resource_action UNIQUE (resource, action)
);
CREATE TABLE public.roles (
role_id serial4 NOT NULL,
"name" varchar(100) NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT roles_pkey PRIMARY KEY (role_id)
);
CREATE UNIQUE INDEX ix_roles_name ON public.roles USING btree (name);
CREATE TABLE public.doctors (
doctor_id uuid NOT NULL,
email varchar(150) NOT NULL,
hashed_password text NOT NULL,
specialization varchar(100) NOT NULL,
department_id int4 NULL,
role_id int4 NOT NULL,
first_name varchar(100) NOT NULL,
last_name varchar(100) NOT NULL,
```

```
middle_name varchar(100) NOT NULL,
phone_number varchar(15) NOT NULL,
birth_data date NOT NULL,
"gender" public."gender" NOT NULL,
is_active bool NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT doctors_pkey PRIMARY KEY (doctor_id),
CONSTRAINT doctors_department_id_fkey FOREIGN KEY (department_id) REFERENCES
public.departments(department_id),
CONSTRAINT doctors_role_id_fkey FOREIGN KEY (role_id) REFERENCES public.roles(role_id)
);
CREATE UNIQUE INDEX ix_doctors_email ON public.doctors USING btree (email);
CREATE UNIQUE INDEX ix_doctors_phone_number ON public.doctors USING btree
(phone_number);
CREATE TABLE public.medical_records (
record_id uuid NOT NULL,
patient_id uuid NOT NULL,
doctor_id uuid NOT NULL,
department_id int4 NOT NULL,
examination_date timestamp NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT medical_records_pkey PRIMARY KEY (record_id),
CONSTRAINT medical_records_department_id_fkey FOREIGN KEY (department_id) REFERENCES
public.departments(department_id),
CONSTRAINT medical_records_doctor_id_fkey FOREIGN KEY (doctor_id) REFERENCES
public.doctors(doctor_id),
CONSTRAINT medical_records_patient_id_fkey FOREIGN KEY (patient_id) REFERENCES
public.patients(patient_id)
);
CREATE TABLE public.role_permissions (
role_id int4 NOT NULL,
permission_id int4 NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
CONSTRAINT role_permissions_pkey PRIMARY KEY (role_id, permission_id),
CONSTRAINT role_permissions_permission_id_fkey FOREIGN KEY (permission_id) REFERENCES
public.permissions(permission_id) ON DELETE CASCADE,
CONSTRAINT role_permissions_role_id_fkey FOREIGN KEY (role_id) REFERENCES
public.roles(role_id) ON DELETE CASCADE
);
CREATE TABLE public.medical_record_transcriptions (
transcription_id serial4 NOT NULL,
record_id uuid NOT NULL,
audio_url text NOT NULL,
audio_duration int4 NOT NULL,
transcription_text text NOT NULL,
asr_model varchar(100) NOT NULL,
asr_config json NOT NULL,
"language" varchar(2) NOT NULL,
created_at timestamp DEFAULT now() NOT NULL,
updated_at timestamp DEFAULT now() NOT NULL,
is_processed bool NOT NULL,
CONSTRAINT medical_record_transcriptions_pkey PRIMARY KEY (transcription_id,
record_id),
CONSTRAINT medical_record_transcriptions_record_id_fkey FOREIGN KEY (record_id)
REFERENCES public.medical_records(record_id)
);
```

ДОДАТОК Б

Unit-тест AuthService за допомогою бібліотеки pytest та техніки moking

```python
import pytest
from unittest.mock import AsyncMock, MagicMock, patch
import uuid
from datetime import datetime, timedelta, UTC
import jwt
from fastapi import Response, Depends
from fastapi.security import APIKeyCookie
from http.cookies import SimpleCookie

# Import the necessary classes and functions
from backend.core import MedServiceException, status
from backend.models.users import Doctor
from backend.schemas.users import DoctorUpdate
from backend.utils.password_helper import PasswordHelperProtocol
from backend.repositories.doctors import IDoctorRepository
from backend.services.auth import AuthService, authenticate, get_auth_service
from backend.config import config

# Create test fixtures
@pytest.fixture
def mock_doctor_repository():
    return AsyncMock(spec=IDoctorRepository)

@pytest.fixture
def mock_password_helper():
    password_helper = MagicMock(spec=PasswordHelperProtocol)
    # Set up default behavior
    password_helper.hash.return_value = "hashed_password"
    password_helper.verify_and_update.return_value = (True, None)
    return password_helper

@pytest.fixture
def auth_service(mock_doctor_repository, mock_password_helper):
    service = AuthService(mock_doctor_repository)
    service.password_helper = mock_password_helper
    return service

@pytest.fixture
def mock_doctor():
    return MagicMock(
        spec=Doctor,
        doctor_id=uuid.uuid4(),
        email="doctor@example.com",
        phone_number="1234567890",
```

```
        hashed_password="hashed_password",
        is_active=True,
    )

@pytest.fixture
def mock_token():
    return "mock_jwt_token"

# Test the AuthService class methods
@pytest.mark.asyncio
async def test_login_successful(auth_service, mock_doctor_repository, mock_doctor):
    # Arrange
    username = "doctor@example.com"
    password = "password123"
    mock_doctor_repository.get_doctor_for_login.return_value = mock_doctor

    # Setup mock for _create_login_response
    with patch.object(auth_service, '_create_login_response') as
mock_create_response:
        mock_response = Response(status_code=status.HTTP_204_NO_CONTENT)
        mock_create_response.return_value = mock_response

        # Act
        response = await auth_service.login(username, password)

        # Assert
        mock_doctor_repository.get_doctor_for_login.assert_called_once_with(usernam
e)
        auth_service.password_helper.verify_and_update.assert_called_once_with(
            password, mock_doctor.hashed_password
        )
        mock_create_response.assert_called_once_with(mock_doctor)
        assert response == mock_response

@pytest.mark.asyncio
async def test_login_invalid_credentials(auth_service, mock_doctor_repository):
    # Arrange
    username = "nonexistent@example.com"
    password = "wrong_password"
    mock_doctor_repository.get_doctor_for_login.return_value = None

    # Act & Assert
    with pytest.raises(MedServiceException) as exc_info:
        await auth_service.login(username, password)

    assert exc_info.value.code == status.HTTP_401_UNAUTHORIZED
    assert "Invalid credentials" in exc_info.value.title

@pytest.mark.asyncio
```

```python
async def test_login_inactive_user(auth_service, mock_doctor_repository,
mock_doctor):
    # Arrange
    username = "inactive@example.com"
    password = "password123"
    mock_doctor.is_active = False
    mock_doctor_repository.get_doctor_for_login.return_value = mock_doctor

    # Act & Assert
    with pytest.raises(MedServiceException) as exc_info:
        await auth_service.login(username, password)

    assert exc_info.value.code == status.HTTP_401_UNAUTHORIZED
    assert "Invalid credentials" in exc_info.value.title

@pytest.mark.asyncio
async def test_login_invalid_password(auth_service, mock_doctor_repository,
mock_doctor, mock_password_helper):
    # Arrange
    username = "doctor@example.com"
    password = "wrong_password"
    mock_doctor_repository.get_doctor_for_login.return_value = mock_doctor
    auth_service.password_helper.verify_and_update.return_value = (False, None)

    # Act & Assert
    with pytest.raises(MedServiceException) as exc_info:
        await auth_service.login(username, password)

    assert exc_info.value.code == status.HTTP_401_UNAUTHORIZED
    assert "Invalid credentials" in exc_info.value.title
```

ДОДАТОК В

Фрагмент тексту для перевірки ASR на WER

Доброго дня, лікарю. Маю останнім часом неприємні відчуття в шиї та спині. Постійно тягне, іноді навіть болить при русі голови або коли довго сиджу.

Доброго дня. Зрозуміло. З якою періодичністю виникає біль? Постійно чи епізодично?

Часто — майже щодня. Найбільше — під кінець робочого дня, коли багато сиджу за комп’ютером.

А є ще якісь симптоми? Наприклад, головний біль, запаморочення, шум у вухах?

Так, буває головний біль у потилиці. А ще іноді трошки німіють пальці рук, особливо вночі або зранку.

Добре. Це важлива інформація. Ви колись травмували шию чи спину? Можливо, займалися важкою фізичною працею або спортом?

Ну, нічого серйозного. Але давно працював вантажником, а зараз просто багато сиджу.

Розумію. Давайте огляну вашу шию та спину, перевірю рухливість, чутливість і рефлекси.

У вас є напруження м’язів шиї, обмеження рухів, зниження чутливості в деяких ділянках рук. Це вказує на можливі дегенеративні зміни в шийному відділі хребта. Я підозрюю остеохондроз.

Це щось серйозне?

Це не критично, але потребує лікування. Головне — вчасно зайнятися терапією. Ми зробимо рентген або МРТ, щоб підтвердити діагноз. А поки що я порекомендую вам щадний режим, лікувальну фізкультуру, масаж і, можливо, медикаменти для зняття болю та запалення.