

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Розробка додатку для створення і організації
нотаток психотерапевтів з використанням ШІ»**

Виконав студент групи ПЗм-24-2
БАРВИЦЬКИЙ Владислав Володимирович

Керівник роботи:
БАКАЛЮК Тетяна Анатоліївна

Рецензент:
МОРОЗОВ Андрій Васильович

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«01» жовтня 2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Здобувач вищої освіти: **БАРВИЦЬКИЙ Владислав Володимирович**

Керівник роботи: **ВАКАЛЮК Тетяна Анатоліївна**

Тема роботи: **«Розробка додатку для створення і організації нотаток психотерапевтів з використанням ШІ»**, затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 491/с.**

Вихідні дані для роботи: Об'єктом дослідження є технології розробки інтелектуального додатку для створення й організації нотаток психотерапевтів із використанням ШІ. Предметом дослідження є моделі, методи та алгоритми розробки такого додатку (зокрема методи обробки природної мови, розпізнавання мовлення, анонімізації та класифікації даних), що забезпечують структурування, аналіз, швидкий пошук і безпечне зберігання професійних записів.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Вакалюк Т.А	02.10.2024р.	02.10.2024р.
2	Вакалюк Т.А	02.10.2024р.	02.10.2024р.
3	Вакалюк Т.А	02.10.2024р.	02.10.2024р.

РЕФЕРАТ

Робота магістра спеціальності 121 «Інженерія програмного забезпечення» присвячена створенню мобільного застосунку для психологічної підтримки з інтеграцією чат-бота на основі штучного інтелекту. Робота складається з мобільного застосунку та пояснювальної записки.

Пояснювальна записка містить 84 сторінок, 12 ілюстрацій, 5 таблиць, 2 діаграми та 28 використаних джерел.

У першому розділі було проведено аналіз існуючих програмних рішень для психотерапевтів (Talkspace, Wysa, BetterHelp, Calm) та розглянуто їхні функціональні особливості. Також було описано методи інтеграції мовних моделей у мобільні застосунки та визначено переваги використання SwiftUI, Combine та Firebase.

У другому розділі було змодельовано архітектуру системи на основі патернів MVVM та Clean Architecture, а також реалізовано базу даних у Firebase Realtime Database.

У третьому розділі продемонстровано практичну реалізацію застосунку: створення профілю користувача, функціонування чату з ботом, ведення нотаток психотерапевта та модульне й інтеграційне тестування з використанням Swift Testing.

У висновках підведено підсумки про те, що вдалося розробити працездатний прототип мобільного застосунку для психологічної підтримки з інтеграцією ШІ-чат-бота, який може бути основою для подальшого наукового й комерційного розвитку.

КЛЮЧОВІ СЛОВА: МОБІЛЬНИЙ ЗАСТОСУНОК, ПСИХОЛОГІЧНА ПІДТРИМКА, ШТУЧНИЙ ІНТЕЛЕКТ, ЧАТ-БОТ, SWIFTUI, FIREBASE.

ABSTRACT

The work of a Master's student majoring in Software Engineering is devoted to the development of a mobile application for psychological support with the integration of an AI-based chatbot. The work consists of a mobile application and an explanatory note.

The explanatory note contains 84 pages, 12 figures, 5 tables, 2 diagrams, and 28 references.

The first chapter analyzes existing software solutions for psychotherapists (Talkspace, Wysa, BetterHelp, Calm) and reviews their functional features. Methods of integrating language models into mobile applications were considered, and the advantages of using SwiftUI, Combine, and Firebase were defined.

The second chapter presents the system architecture designed using MVVM and Clean Architecture patterns, as well as the database implementation in Firebase Realtime Database.

The third chapter demonstrates the practical implementation of the application: creation of a user profile, functioning of the chatbot, therapist's note-keeping, and module and integration testing using Swift Testing.

The conclusions summarize that a functional prototype of a mobile application for psychological support with an AI-based chatbot was successfully developed, which may serve as a basis for further scientific and commercial development.

KEYWORDS: MOBILE APPLICATION, PSYCHOLOGICAL SUPPORT, ARTIFICIAL INTELLIGENCE, CHATBOT, SWIFTUI, FIREBASE.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ НОТАТОК ПСИХОТЕРАПЕВТІВ ВИКОРИСТАННЯМ ШІ	3 10
1.1 Постановка задачі	10
1.2 Аналіз аналогів програмного продукту	11
1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	14
1.4 Архітектура додатку для створення і організації нотаток психотерапевтів з використанням ШІ	18
1.5 Огляд наявних методів, моделей, алгоритмів	21
Висновки до першого розділу	24
РОЗДІЛ 2. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ НОТАТОК ПСИХОТЕРАПЕВТІВ ВИКОРИСТАННЯМ ШІ	3 26
2.1 Визначення варіантів використання застосунку	26
2.2 Опис реалізації функціональності та бази даних додатку для створення і організації нотаток психотерапевтів з використанням ШІ	33
Висновки до другого розділу	50
РОЗДІЛ 3. ТЕСТУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ НОТАТОК ПСИХОТЕРАПЕВТІВ З ВИКОРИСТАННЯМ ШІ	52
3.1 Автоматичне тестування додатку для створення і організації нотаток психотерапевтів з використанням ШІ	52
3.2 Опис реалізації функціональності застосунку	58

Висновки до третього розділу	70
ВИСНОВКИ	72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	74
ДОДАТКИ	77

ВСТУП

Сучасний розвиток інформаційних технологій призвів до суттєвих змін у сфері надання психологічної допомоги. Традиційні методи консультування поступово доповнюються цифровими інструментами, що забезпечують нові формати комунікації між психотерапевтом і клієнтом. Мобільні застосунки, орієнтовані на психологічну підтримку, стають важливим елементом сучасної психотерапевтичної практики, оскільки дають змогу забезпечити постійний зворотний зв'язок, підвищити доступність допомоги та оперативно реагувати на запити користувачів. Особливого значення ці технології набувають у період соціальної нестабільності, воєнних дій та психологічної напруги, коли зростає потреба у швидкій і конфіденційній підтримці населення.

Інтеграція штучного інтелекту у сферу психотерапії відкриває нові можливості для створення інтелектуальних систем, здатних аналізувати емоційний стан користувача, надавати рекомендації та імітувати емпатичну взаємодію. Чат-боти, побудовані на основі моделей обробки природної мови, дозволяють автоматизувати частину комунікаційних процесів, зберігаючи при цьому індивідуальний підхід і високий рівень чутливості до контексту спілкування. Такий підхід дає змогу зменшити навантаження на фахівців і водночас підвищити якість психологічної підтримки.

Актуальність дослідження зумовлена необхідністю створення мобільного застосунку, який поєднує сучасні підходи до психотерапевтичної практики з можливостями штучного інтелекту. Існуючі програмні рішення, такі як Talkspace, Wysa, BetterHelp чи Calm, мають низку обмежень, пов'язаних з високою вартістю, відсутністю персоналізації або обмеженим функціоналом. Це створює потребу у вітчизняному рішенні, що забезпечуватиме не лише інтерактивність та зручність користування, а й підвищену безпеку, конфіденційність даних і адаптованість до локального контексту.

Метою роботи є розробка мобільного застосунку для психологічної підтримки з інтеграцією чат-бота на основі штучного інтелекту, який забезпечує

ефективну комунікацію між психотерапевтом і клієнтом, автоматизує рутинні процеси аналізу повідомлень та сприяє покращенню якості психоемоційної допомоги.

Завданнями даної роботи є здійснити аналіз сучасних програмних рішень у сфері цифрової психологічної підтримки та визначити їхні функціональні й технологічні обмеження. На основі результатів аналізу сформулювати вимоги до мобільного застосунку з інтеграцією штучного інтелекту та змодельовати його архітектуру з використанням сучасних підходів програмної інженерії та розробити застосунок. Після чого провести тестування функціональної коректності та стабільності роботи застосунку з метою підтвердження його працездатності та відповідності поставленим вимогам.

Об'єктом дослідження є процес розробки програмних систем для психологічної підтримки, предметом – методи, моделі та алгоритми інтеграції штучного інтелекту у мобільні застосунки для психотерапевтичної практики.

Наукова новизна роботи полягає у комплексному поєднанні архітектурних рішень сучасної мобільної розробки з методами інтеграції мовних моделей штучного інтелекту у сфері психологічної підтримки. У роботі запропоновано підхід до побудови мобільного застосунку для психотерапевтичної практики, який базується на розмежуванні відповідальностей між клієнтською частиною, сервісним рівнем і модулем штучного інтелекту, що забезпечує масштабованість і підвищену надійність системи. Особливістю запропонованого підходу є використання інтелектуального чат-бота не лише як засобу комунікації, а як аналітичного інструмента для підтримки психотерапевта під час ведення сеансів і аналізу текстових нотаток. Наукову новизну також становить адаптація методів обробки природної мови до специфіки психотерапевтичних діалогів у межах мобільного застосунку з урахуванням вимог конфіденційності та безпеки персональних даних.

Практичне значення одержаних результатів полягає у створенні працездатного прототипу мобільного застосунку для організації професійної

діяльності психотерапевтів із використанням інструментів штучного інтелекту. Розроблений застосунок може бути використаний у практичній діяльності фахівців для ведення клієнтських записів, нотування результатів сеансів, планування взаємодії з клієнтами та отримання інтелектуальних рекомендацій на основі аналізу текстових даних. Запропоновані архітектурні та програмні рішення можуть бути використані як основа для подальшого розвитку комерційних або науково-дослідних систем цифрової психологічної підтримки, а також у навчальному процесі під час підготовки фахівців з інженерії програмного забезпечення.

Таким чином, представлена робота поєднує теоретичні засади програмної інженерії з практичними завданнями психотерапевтичної діяльності, сприяючи розширенню інструментарію фахівців і підвищенню ефективності надання психологічних послуг у сучасному цифровому середовищі.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ НОТАТОК ПСИХОТЕРАПЕВТІВ З ВИКОРИСТАННЯМ ШІ

1.1 Постановка задачі

Постановка завдання полягає у створенні мобільного застосунку для психотерапевтів із вбудованим чат-ботом, який використовує штучний інтелект для аналізу текстових повідомлень та надання відповідних порад користувачам на основі складних алгоритмів. Проєкт спрямований на розробку інтуїтивно зрозумілого, зручного та надійного застосунку, що дозволить психотерапевтам ефективніше працювати з клієнтами, отримувати попередній аналіз їхнього емоційного стану та швидко адаптувати підхід до терапії. Чат-бот розроблений таким чином, щоб мати здатність точно розпізнавати тон повідомлень, враховувати історію спілкування з користувачем та надавати рекомендації, адаптовані до індивідуальних особливостей і настрою, використовуючи сучасні алгоритми обробки природної мови, що дозволяють забезпечити високу точність аналізу тексту.

Проєкт орієнтований на створення зручного та естетично привабливого інтерфейсу за допомогою технології SwiftUI, що забезпечує адаптивний дизайн для різноманітних пристроїв і платформ. Також використовується фреймворк Combine для ефективного управління асинхронними потоками даних, що є критично важливим для швидкої та стабільної роботи застосунку. Завдяки інтеграції з Firebase передбачається реалізація надійної системи автентифікації користувачів, а також збереження історії спілкування в режимі реального часу, що гарантує безперервність та високу якість обслуговування. Такий підхід дозволяє забезпечити миттєву обробку запитів і створити відчуття постійного діалогу між психотерапевтом і пацієнтом, що значно покращує ефективність терапевтичного процесу.

Окрему увагу приділено розширенню функціоналу застосунку, зокрема можливості створення персоналізованих звітів для психотерапевтів на основі аналізу спілкування, що дозволить більш об'єктивно оцінювати динаміку емоційного стану пацієнта. Також передбачається функція автоматизованого нагадування про заплановані сесії, що сприятиме підвищенню дисципліни у веденні терапевтичного процесу.

Особлива увага приділяється забезпеченню конфіденційності та захисту даних, оскільки застосунок оперує особистою інформацією користувачів, що вимагає високого рівня безпеки та довіри до системи. Використання сучасних методів шифрування та стандартів зберігання даних дозволяє надати користувачам впевненість у збереженні приватності та безпеці їх особистої інформації.

В результаті реалізації поставленого завдання мобільний застосунок стане потужним інструментом для психотерапевтів, який відповідає сучасним вимогам технологічного прогресу та спрямований на покращення їхньої професійної діяльності. Завдяки комплексному підходу до розробки, який враховує як технічні, так і психологічні аспекти взаємодії, цей застосунок зможе стати незамінним інструментом у практиці психотерапевтів, допомагаючи їм ефективніше працювати з клієнтами, підвищувати якість надання послуг та покращувати результати терапії.

1.2 Аналіз аналогів програмного продукту

Аналіз існуючих програмних продуктів у сфері психологічної підтримки є необхідним етапом розробки, оскільки дозволяє виявити усталені підходи до реалізації функціональності, оцінити ефективність технічних рішень та визначити невирішені проблеми, які залишаються актуальними для фахівців у галузі психотерапії. Сучасний ринок цифрових сервісів психологічної допомоги представлений широким спектром мобільних і вебзастосунків, що орієнтовані як на кінцевих користувачів, так і на професійних психологів.

Одним із найбільш відомих рішень є Headspace, який зосереджений переважно на медитаційних практиках, керуванні стресом та розвитку усвідомленості. Незважаючи на популярність і високу якість контенту, цей застосунок орієнтований на масового користувача та не призначений для професійного використання психотерапевтами. Відсутність можливостей ведення клієнтських нотаток, аналізу текстових діалогів і персоніфікованої взаємодії обмежує його застосування у клінічній або консультативній практиці. На рисунку 1.1 наведено зовнішній вигляд застосунку.

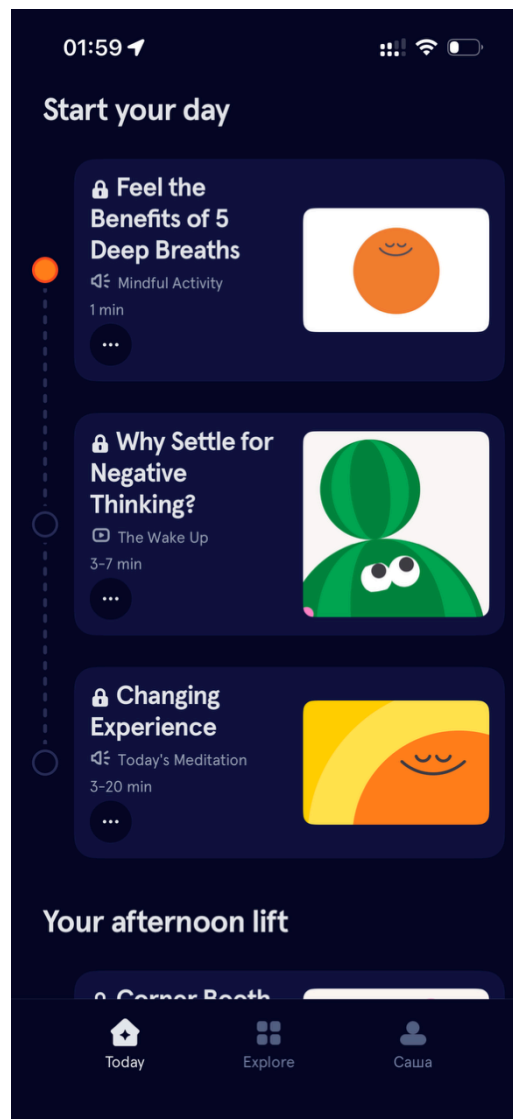


Рис. 1.1 – Зовнішній вигляд застосунку Headspace

Платформи BetterHelp і Talkspace пропонують модель онлайн-консультацій із сертифікованими спеціалістами, забезпечуючи доступ до психологічної

допомоги через текстові повідомлення, аудіо- та відеозв'язок. Водночас ці сервіси мають низку недоліків, зокрема високу вартість використання, залежність від стабільного інтернет-з'єднання та відсутність інструментів автоматизованого аналізу змісту консультацій. Крім того, такі платформи орієнтовані радше на організацію консультацій, ніж на підтримку внутрішніх робочих процесів психотерапевта, зокрема систематизацію нотаток або аналітичну обробку даних.

Окремої уваги заслуговує застосунок Wysa, який використовує елементи штучного інтелекту для ведення діалогів із користувачами. Незважаючи на наявність чат-бота, функціональність цього рішення є обмеженою з точки зору глибини аналізу та адаптації до індивідуального контексту психотерапевтичних сеансів. Чат-бот здебільшого працює за заздалегідь визначеними сценаріями, що знижує ефективність його використання у складних або нетипових ситуаціях. На рисунку 1.2 наведено зовнішній вигляд застосунку.

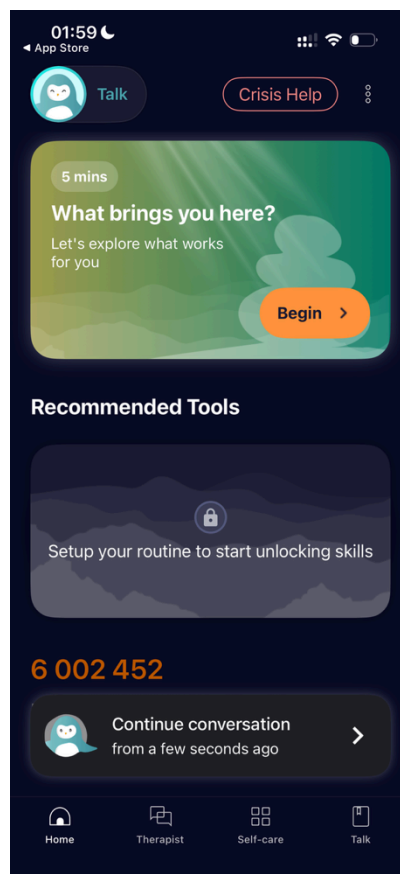


Рис. 1.2 – Зовнішній вигляд застосунку Wysa

Застосунок Calm фокусується переважно на зменшенні тривожності та покращенні якості сну за допомогою аудіоконтенту, звукотерапії та вправ на релаксацію. Хоча цей продукт є ефективним для саморегуляції психоемоційного стану, він не забезпечує інтерактивної взаємодії в режимі реального часу та не надає інструментів для професійної роботи психотерапевта з клієнтськими даними.

Порівняльний аналіз наведених рішень дозволяє зробити висновок, що більшість існуючих застосунків або орієнтовані на масового користувача без урахування специфіки професійної діяльності психотерапевтів, або не інтегрують можливості штучного інтелекту на рівні, достатньому для аналітичної підтримки фахівця. Також суттєвим недоліком багатьох продуктів є обмежені можливості персоналізації, відсутність централізованої системи нотаток і залежність від хмарних сервісів, що може створювати ризики для конфіденційності даних.

Таким чином, результати аналізу аналогів підтверджують доцільність розробки мобільного застосунку, який поєднує інструменти організації професійної діяльності психотерапевта з можливостями інтеграції штучного інтелекту. Запропоноване рішення спрямоване на усунення виявлених обмежень шляхом забезпечення персоналізованої взаємодії, аналітичної підтримки сеансів та безпечної роботи з конфіденційними даними у межах єдиного програмного середовища.

1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Swift, як основна мова програмування, має значну історію розвитку та оптимізації, що дозволяє створювати не тільки стабільний, але й високопродуктивний код. Мова була розроблена компанією Apple як сучасна альтернатива Objective-C, забезпечуючи більш зрозумілий синтаксис, безпечність типів даних та сучасні засоби управління пам'яттю[1]. Використання Swift сприяє

розробці застосунків, які не лише швидко виконуються, але й демонструють високу ступінь безпеки завдяки вбудованим механізмам запобігання помилок. У розробці мобільного застосунку для психотерапевтів із вбудованим чат-ботом на основі штучного інтелекту застосування Swift дозволяє легко підтримувати складні алгоритми ШІ, забезпечуючи належний рівень обробки даних та інтеграцію з іншими компонентами системи. Завдяки своїй сумісності з платформами Apple, Swift стає надійною основою для створення продукту, що відповідає високим стандартам продуктивності та надійності.

Фреймворк Combine є невід'ємною частиною сучасного підходу до розробки мобільних застосунків, орієнтованих на реактивне програмування. Він надає зручні засоби для роботи з потоками даних, що є критичним для чат-ботів, які мають оперативно реагувати на численні події користувачів[2]. Реалізація асинхронного обміну інформацією за допомогою Combine дозволяє не тільки зменшити складність коду, а й забезпечити стабільну роботу застосунку навіть при високій активності користувачів або несподіваних затримках у зв'язку з серверними запитами. Це особливо важливо для систем, де затримка відповіді може негативно впливати на користувацький досвід і, як наслідок, на ефективність роботи психотерапевтів із вбудованим чат-ботом на основі штучного інтелекту. Використання Combine дозволяє створити систему, яка здатна масштабуватися та адаптуватися до зростаючих потреб користувачів, гарантуючи стабільну роботу незалежно від обсягу оброблюваних даних.

SwiftUI, як декларативний фреймворк для створення користувацьких інтерфейсів, значно змінює підхід до розробки візуальної частини застосунку[3]. Завдяки декларативному стилю розробнику більше не потрібно керувати життєвим циклом елементів інтерфейсу вручну, що дозволяє зосередитися на логіці роботи застосунку. Це сприяє більшій зручності при створенні адаптивних і крос-платформених інтерфейсів, що є важливим фактором у сучасних умовах, коли психотерапевти та їх клієнти очікують максимально інтуїтивно зрозумілого та естетично привабливого дизайну. SwiftUI також забезпечує простий спосіб

реалізації складних анімацій та переходів, що покращує загальне сприйняття продукту та сприяє залученню користувачів. У нашому проекті цей фреймворк допомагає створити чистий та сучасний дизайн, який дозволяє психотерапевтам та пацієнтам легко орієнтуватися у функціональності застосунку, забезпечуючи тим самим позитивний користувацький досвід.

Інтеграція з Firebase, зокрема використання Firebase Auth та Firebase Database, є важливою складовою для забезпечення безпеки та зручності управління даними користувачів[4]. Firebase Auth гарантує високий рівень захисту даних завдяки застосуванню сучасних алгоритмів шифрування та підтримці різних методів аутентифікації. Це дозволяє користувачам, зокрема психотерапевтам та їх клієнтам, обирати найбільш зручний спосіб входу в систему – чи то через соціальні мережі, електронну пошту чи інші методи. Завдяки цьому система здатна швидко адаптуватися до різних сценаріїв використання, забезпечуючи безперебійний доступ до застосунку. Firebase Database, у свою чергу, є потужним інструментом для синхронізації даних в режимі реального часу, що є критичним для чат-ботів, які повинні миттєво реагувати на запити користувачів. Використання цієї технології дозволяє мінімізувати затримки при передачі даних між сервером і клієнтом, що особливо важливо для підтримання живого діалогу у режимі реального часу. Крім того, Firebase забезпечує високу масштабованість системи, що дозволяє проекту розширюватися у міру зростання кількості користувачів та обсягу оброблюваних даних.

Останнім, але не менш важливим елементом є середовище розробки Xcode. Як офіційне інтегроване середовище для розробки застосунків на платформах Apple, Xcode надає широкий набір інструментів для створення, налагодження та тестування програмного забезпечення. Завдяки тісній інтеграції з платформою iOS, це середовище дозволяє максимально ефективно використовувати можливості апаратного та програмного забезпечення пристроїв Apple[5]. Розробка в Xcode забезпечує плавну інтеграцію між різними компонентами застосунку, що суттєво знижує час на розробку та тестування, а також допомагає оперативно

виявляти й усувати помилки. Додатковою перевагою є можливість використання емуляторів для тестування застосунку на різних пристроях, що дозволяє розробникам перевірити, як продукт буде працювати у різних умовах експлуатації. Це гарантує, що кінцевий продукт матиме високу стабільність і відповідатиме сучасним вимогам ринку.

Загальна концепція розробки мобільного застосунку для психотерапевтів із вбудованим чат-ботом на основі штучного інтелекту ґрунтується на поєднанні передових технологій, які дозволяють не лише забезпечити ефективну роботу застосунку, а й створити зручний та інтуїтивно зрозумілий інтерфейс для користувачів. Взаємодія між компонентами, такими як Swift, Combine, SwiftUI, Firebase, створює комплексну екосистему, де кожна технологія виконує свою специфічну роль, спрямовану на підвищення якості кінцевого продукту. Сучасні засоби розробки значно спрощують інтеграцію складних алгоритмів штучного інтелекту та забезпечують їх безперебійну роботу навіть у режимі високої завантаженості.

Особливу увагу слід приділити аспекту безпеки, що набуває особливого значення при розробці застосунку для психотерапевтів. Використання Firebase Auth для управління аутентифікацією гарантує, що доступ до особистих даних та історії взаємодії матиме лише авторизований користувач.

Використання сучасних технологій, таких як SwiftUI та Combine, дозволяє розробляти застосунок, який не лише відповідає сучасним стандартам, а й має можливість легко адаптуватися до майбутніх змін у технологічному середовищі. Це забезпечує тривалу життєздатність проекту, адже оновлення та модифікації можуть впроваджуватися без суттєвих витрат часу та ресурсів. Завдяки цьому застосунок має потенціал для розширення функціоналу, що дозволяє з часом впроваджувати нові можливості, орієнтовані на потреби психотерапевтів.

Таким чином, поєднання Swift, Combine, SwiftUI, Firebase Auth, Firebase Database та Xcode створює потужну базу для розробки мобільного застосунку, що забезпечує як високопродуктивну роботу, так і зручність використання. Кожна

технологія має свою спеціалізацію, що в комплексі дозволяє вирішувати завдання інтеграції штучного інтелекту, а також питання безпеки, масштабованості та адаптивності системи. Такий підхід гарантує створення продукту, який не лише відповідає сучасним вимогам, але й має потенціал для подальшого розвитку та вдосконалення, що є надзвичайно важливим у швидкозмінному світі інформаційних технологій.

Основною вимогою до апаратного забезпечення є використання iOS або iPadOS версії 17 і вище. Сучасні версії цих операційних систем надають розширені можливості для оптимізації роботи застосунку, включаючи покращену енергоефективність, новітні засоби безпеки та підтримку сучасних API. Це дозволяє використовувати всі переваги Swift, SwiftUI, Combine, Firebase інтегрованих у розробку застосунку.

Застосунок, розроблений для iOS/iPadOS 17 і вище, отримує доступ до потужних апаратних засобів пристроїв Apple, таких як високопродуктивні процесори, розширена оперативна пам'ять та сучасні графічні ядра. Це критично важливо для швидкої обробки даних і забезпечення безперебійної роботи чат-бота, який повинен оперативно реагувати на запити як психотерапевтів, так і їх клієнтів у режимі реального часу. Таким чином, визначені апаратні вимоги гарантують, що застосунок працюватиме стабільно навіть при високій активності, забезпечуючи надійну та ефективну підтримку психотерапевтичної практики.

1.4 Архітектура додатку для створення і організації нотаток психотерапевтів з використанням ШІ

Архітектура розробленого застосунку ґрунтується на сучасних принципах побудови програмних систем та поєднує кілька архітектурних підходів, серед яких ключовими є MVVM (Model–View–ViewModel) та патерн координаторів (Coordinator pattern). У класичному розумінні архітектурний патерн MVVM забезпечує чітке розділення відповідальностей: шар View відображає інтерфейс

користувача та реагує на зміни даних безпосередньо через механізми прив'язки; шар ViewModel виконує роль проміжної ланки між інтерфейсом і моделлю, інкапсулюючи бізнес-логіку та керування станом; шар Model відповідає за доменні сутності та їх збереження. У межах SwiftUI така побудова отримує додаткову ефективність завдяки використанню реактивних механізмів `@Published` та `ObservableObject`, які забезпечують автоматичне оновлення представлення при зміні даних у в'ю-моделі. Це дозволяє уникати дублювання логіки, робить UI «тонким» і значно зменшує кількість помилок, пов'язаних із несинхронністю даних.

Важливим доповненням до MVVM у цьому проєкті став патерн координаторів, який розширює класичні можливості SwiftUI щодо керування навігаційним потоком. Кожен координатор відповідає за певний логічний модуль або сценарій використання: `AuthCoordinator` реалізує процес автентифікації, керуючи переходами між екранами входу, реєстрації та завершення профілю; `HomeScreenCoordinator` визначає логіку домашнього екрану з доступом до клієнтів, розкладу та чатів; `MainFlowCoordinator` інтегрує кілька модулів у єдиний потік; `TheAppCoordinator` координує загальний стан застосунку, вибираючи між авторизаційним та основним інтерфейсом залежно від статусу користувача. Теоретично використання координаторів зменшує навантаження на в'ю-моделі, оскільки вони звільняються від відповідальності за навігацію, і дозволяє досягти інверсії управління: в'юха повідомляє про подію, а координатор приймає рішення, як саме реагувати на неї.

На рівні збереження даних застосунків використовує хмарні сервіси Firebase, які у науковій літературі розглядаються як один із найбільш поширених прикладів Backend-as-a-Service (BaaS). Зокрема, для ідентифікації користувачів застосовується `FirebaseAuth`, що дозволяє реалізувати безпечну автентифікацію з підтримкою стандартів захисту. Дані про лікарів, пацієнтів, розклад і чати зберігаються у `Firebase Realtime Database` у вигляді ієрархічних структур JSON. Подібний підхід забезпечує високу швидкодію завдяки асинхронній синхронізації

даних і одночасному доступу кількох клієнтів. Використання `asDictionary` для моделей дозволяє перетворювати об'єкти на словники, що відповідають вимогам `Firestore API`. Науково-технічні дослідження підтверджують, що застосування `BaaS` зменшує час розробки й оптимізує ресурси команди, оскільки складні завдання, як-от керування доступом чи забезпечення консистентності, передаються готовим хмарним рішенням.

Особливої уваги заслуговує інтеграція з моделями штучного інтелекту `OpenAI`. Використано сервісний клас `OpenAIService`, який реалізує протокол `OpenAIServiceProtocol` та інкапсулює логіку формування запитів до мовної моделі (у даному випадку `gpt-3.5-turbo`). Архітектурно цей сервіс виступає як адаптер у термінах шаблонів проєктування, оскільки перетворює внутрішні доменні структури (`ChatMessage`) на зовнішній формат (`ChatQuery.ChatCompletionMessageParam`) і навпаки. Важливо, що перед формуванням відповіді до запиту додається системне повідомлення, яке задає роль асистента (віртуальний психолог), обмежує тематику й накладає додаткові етичні вимоги (відмова від відповіді на питання поза темою, додавання дисклеймера). У науковому контексті це відповідає концепції `prompt engineering`, що передбачає спеціальне структурування вхідних інструкцій для підвищення релевантності й надійності результатів.

Взаємодія між компонентами реалізується за принципом єдиного циклу даних: користувач ініціює дію через інтерфейс (`View`) → в'ю-модель обробляє подію та звертається до відповідного сервісу (`Firestore` чи `OpenAI`) → отриманий результат обробляється в бізнес-логіці й оновлює опубліковані властивості → `UI` автоматично відображає зміни. Така архітектурна схема відповідає сучасним вимогам до реактивних систем і забезпечує властивості відмовостійкості, модульності та масштабованості. У ширшому теоретичному розумінні це відображає принципи чистої архітектури (`Clean Architecture`), де бізнес-логіка ізольована від деталей реалізації інфраструктури, а залежності між компонентами мають чіткий напрямок (від зовнішнього шару до внутрішнього).

Отже, архітектура застосунку поєднує кілька науково обґрунтованих підходів і шаблонів, що дозволяє досягти гнучкості та надійності системи. Використання MVVM гарантує чисте розділення відповідальностей і зрозумілий цикл оновлення інтерфейсу, координатори забезпечують структуровану навігацію та зменшують складність в'ю-моделей, Firebase виступає як перевірене рішення для збереження й синхронізації даних, а OpenAI інтегрує інтелектуальні можливості через контрольований сервісний шар[6]. З наукової точки зору така архітектура є збалансованою комбінацією патернів проєктування, хмарних сервісів і методів інтеграції ШІ, що відповідає сучасним трендам у галузі розробки інформаційних систем.

1.5 Огляд наявних методів, моделей, алгоритмів

Для побудови мобільного застосунку психологічної підтримки необхідно використовувати базові методи NLP, архітектури нейронних мереж та алгоритми оптимізації, які забезпечують інтерпретацію тексту, моделювання контексту та генерацію відповідей. Ключову роль відіграють Transformer-моделі, оскільки вони базуються на механізмі уваги, що дозволяє паралельно обробляти послідовності та враховувати глобальні залежності між токенами [7][8][9]. Механізм self-attention використовує перетворення

$$Q = XW^Q, K = XW^K, V = XW^V \quad (1.1)$$

із подальшим масштабним добутком:

$$\text{softmax}(dkQKT)V \quad (1.2)$$

що забезпечує контекстуалізацію кожного токена [10][11]. Multi-head attention дає змогу моделі паралельно аналізувати різні аспекти психологічних висловлювань (емоційний тон, семантику, часові контексти), що критично важливо у діалогах з пацієнтами [12][13].

BERT, як двонаправлена Transformer-модель, формує контекстуалізовані представлення та навчається через Masked Language Modeling, що дозволяє

виявляти семантичні закономірності у висловлюваннях пацієнтів, включно з емоційними нюансами [14][15][16]. GPT-архітектури, навпаки, є авторегресивними та опираються на прогнозування наступного токена, що робить їх ефективними для генерації підтримувальних реплік у психотерапевтичних чатах [17][18]. LSTM та BPTT залишаються важливими у задачах, пов'язаних з тривалими часовими залежностями, хоча Transformer-архітектури показують вищу продуктивність на великомасштабних корпусах [19].

Векторні представлення слів створюються за допомогою Word2Vec (Skip-gram або CBOW), що оптимізує лог-ймовірність передбачення контексту:

$$T \sum_t = 1 T \sum \log p(w_t) - c \leq j \leq c \quad (1.3)$$

формуючи семантичні простори, корисні для психотерапевтичних термінів [20]. Однак сучасні системи застосовують контекстуальні embeddings Transformer-моделей, які відображають динамічні значення слів залежно від психологічного контексту [21][22].

Токенізація BPE та WordPiece дає змогу ефективно працювати з рідкісною психологічною лексикою, розкладаючи складні терміни на підслівні одиниці [65][67]. Це допомагає моделі обробляти форми, яких немає у базовому словнику, і зберігати морфологічну інформацію.

Навчання моделей здійснюється за допомогою градієнтних методів. Базове оновлення параметрів має вигляд

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t) \quad (1.4)$$

однак у практиці найчастіше використовується Adam, який враховує перші два моменти градієнта:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (1.5)$$

із корекцією зсуву та оновленням

$$\theta_{t+1} = \theta_t - v^t + \epsilon m^t \quad (1.6)$$

Adam забезпечує стабільну збіжність навіть за невеликих та обмежених терапевтичних корпусів [23][24].

Backpropagation є універсальним методом обчислення градієнтів для глибоких мереж, а його варіант BPTT використовується для рекурентних моделей. Помилка для шару l визначається як

$$\delta l = ((Wl + 1)T\delta l + 1) \odot \sigma'(zl) \quad (1.7)$$

що забезпечує ефективне оновлення параметрів навіть у складних багатошарових структурах [39][41].

У задачах класифікації психічних станів застосовується cross-entropy:

$$L = - \sum_i y_i \log y_i \quad (1.8)$$

а при бінарних завданнях:

$$L = - [y \log y + (1 - y) \log (1 - y)], \quad (1.9)$$

у поєднанні з softmax або sigmoid функціями активації [54][55][59]. Для запобігання перенавчанню використовуються dropout-маски

$$h_{drop} = m \odot h \quad (1.10)$$

та layer normalization

$$LN(x) = \gamma \sigma^2 + \epsilon x - \mu + \beta \quad (1.11)$$

що стабілізують навчання та покращують узагальнення на реальні терапевтичні діалоги [25][26].

Supervised learning, transfer learning та fine-tuning дозволяють адаптувати великі мовні моделі до психологічного домену. Найефективнішим підходом є fine-tuning із низьким learning rate, що зберігає загальні мовні знання моделі, додаючи здатність адекватно інтерпретувати психотерапевтичні запити [27]. Domain-adaptive pretraining на корпусах текстів ментального здоров'я істотно підвищує точність аналізу емоцій та виявлення психологічних патернів [28].

Інтеграція цих методів дозволяє створювати моделі, що аналізують емоційний стан користувача, розпізнають динаміку психоемоційних змін, генерують емпатичні відповіді та дотримуються етичних обмежень. Fine-tuned Transformer-моделі, доповнені стабільними оптимізаційними схемами та регуляризацією, демонструють високу точність у задачах психотерапевтичної

підтримки та можуть бути основою безпечного, спеціалізованого мобільного застосунку для психологічної допомоги.

Висновки до першого розділу

У підсумку першого розділу можна стверджувати, що створення мобільного застосунку для психологічної підтримки, що базується на інтеграції сучасних технологій штучного інтелекту, є вкрай актуальним і перспективним напрямком у поточних умовах воєнного конфлікту в Україні. Проведений аналіз проблематики демонструє, що завдання, пов'язане з розробкою чат-бота, який здатний оперативно реагувати на запити користувачів та забезпечувати їм підтримку у кризових ситуаціях, знаходиться на перетині високих вимог до безпеки, адаптивності інтерфейсу та надійності обробки даних. Оцінка існуючих аналогів, таких як Headspace, BetterHelp, Calm, Talkspace та Wysa, свідчить про те, що впровадження підходу з використанням локально розгорнутої моделі штучного інтелекту та сучасних технологічних рішень дозволяє значно покращити якість надання послуг, знизити бар'єри доступу до допомоги та забезпечити персоналізований підхід до кожного користувача. Вибір інструментальних засобів, зокрема Swift, SwiftUI, Combine, Firebase та Xcode, ґрунтується на їх здатності забезпечити стабільну, ефективну та масштабовану роботу застосунку, що є критично важливим для підтримки користувачів у важкі моменти. Розглянута архітектура TCA, порівняна з альтернативними підходами, такими як MVVM та VIPER, демонструє свої переваги у керуванні станом та забезпеченні високої тестованості, що сприяє зниженню ризику помилок і підвищенню надійності продукту[6]. Огляд сучасних моделей та алгоритмів, зокрема LLM від OpenAI, підкреслює значення оптимізації процесу обробки природної мови та адаптації штучного інтелекту до специфіки психологічної підтримки, що відкриває нові можливості для персоналізації взаємодії з користувачем. Загальний аналіз дозволяє зробити висновок, що комплексний підхід до розробки мобільного

застосунку, заснований на інтеграції передових технологій та ретельному виборі архітектурних рішень, сприятиме створенню ефективного інструменту для надання психологічної підтримки, який відповідає високим стандартам сучасної розробки програмного забезпечення та може значно полегшити доступ до допомоги в умовах підвищеного стресу та невизначеності.

РОЗДІЛ 2. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ ПОТАТОК ПСИХОТЕРАПЕВТІВ З ВИКОРИСТАННЯМ ШІ

2.1 Визначення варіантів використання застосунку

Аналіз вимог є важливим початковим етапом у процесі створення програмного забезпечення, що передбачає систематичне визначення, уточнення та деталізацію всіх функціональних і нефункціональних характеристик майбутньої системи. Під час цього аналізу спеціалісти формують чіткі вимоги до того, які конкретні функції повинна виконувати програма (функціональні вимоги), а також до її якості, продуктивності, безпеки, зручності та інших параметрів, що не залежать безпосередньо від реалізованих функцій (нефункціональні вимоги).

Однією з ключових складових цього процесу є оцінка часу реакції системи, яка передбачає визначення допустимих інтервалів між діями користувача та відповіддю програмного забезпечення. Такий підхід дозволяє гарантувати комфортну взаємодію з продуктом, оскільки користувачі зазвичай чутливі до будь-яких затримок під час роботи з додатком або інформаційною системою.

Також аналіз вимог включає розробку детальної специфікації сценаріїв використання (use cases). Ці сценарії докладно описують типові ситуації, в яких користувач взаємодіє з системою, показуючи послідовність дій та відповідних реакцій системи. Це допомагає чітко зрозуміти, які саме функції повинні бути реалізовані, а також передбачити та уникнути потенційних проблем і помилок у майбутньому.

Ще одним важливим кроком є створення загальної архітектурної схеми програмного забезпечення. Така схема має чітко демонструвати структуру системи, включаючи окремі елементи (компоненти, модулі, сервіси), а також детальний опис їхньої внутрішньої організації, механізмів взаємодії один з одним, методів передачі та обробки інформації. Це забезпечує цілісне уявлення про

технічну реалізацію системи, полегшує подальшу розробку, супровід та підтримку.

Загалом, аналіз вимог допомагає глибоко зрозуміти реальні потреби та очікування кінцевих користувачів, дозволяючи розробникам створити не лише ефективне, але й максимально комфортне, зручне та надійне програмне рішення, що відповідає усім визначеним бізнес-цілям та технічним стандартам якості.

Однією з базових функцій є реєстрація та авторизація користувача, що передбачає створення особистого облікового запису, подальшу автентифікацію під час входу до системи та захист персональних даних від несанкціонованого доступу. Дана функція забезпечує ідентифікацію користувача та персоналізацію роботи з додатком.

Важливою функціональною вимогою є можливість створення персоналізованих форм для клієнтів. Застосунок повинен надавати інструменти для конструювання та налаштування інтерактивних форм, які дозволяють збирати необхідну інформацію про клієнтів відповідно до індивідуальних потреб користувача та специфіки його діяльності. Це сприяє структуруванню даних і полегшує подальший аналіз отриманої інформації.

Система також має забезпечувати створення та ведення записів про клієнтів. Дана функція передбачає формування електронної бази клієнтів із можливістю збереження, перегляду, редагування та оновлення інформації, отриманої в процесі взаємодії. Наявність централізованого сховища даних дозволяє користувачеві швидко отримувати доступ до необхідної інформації та підтримувати її актуальність.

Окремою функціональною вимогою є можливість створення та зберігання нотаток щодо проведених сеансів із клієнтами. Застосунок повинен дозволяти фіксувати деталі кожного сеансу у вигляді текстових нотаток, що сприяє більш системному підходу до роботи, аналізу попередніх взаємодій та планування подальших кроків у роботі з клієнтами.

Застосунок має підтримувати інтерактивну переписку з інтегрованим штучним інтелектом. Така функція забезпечує користувачеві доступ до інтелектуального асистента, який може відповідати на запитання, допомагати у вирішенні професійних завдань, надавати рекомендації або аналізувати введену інформацію, підвищуючи загальну ефективність роботи з системою.

Додатково передбачається можливість спеціалізованої переписки зі штучним інтелектом, орієнтованої на аналіз сеансів із клієнтами. У межах цієї функції штучний інтелект повинен опрацьовувати дані, отримані під час зустрічей, коментувати результати, формулювати рекомендації та пропонувати аналітичні висновки, які можуть бути використані користувачем для підвищення якості та результативності подальшої роботи.

Діаграма варіантів використання (Use Case Diagram) є одним із базових типів діаграм в UML (Unified Modeling Language), яка використовується для наочного опису взаємодії між користувачами (акторами) та інформаційною системою з метою виконання конкретних функцій (варіантів використання). Така діаграма дозволяє чітко визначити та візуалізувати, які саме завдання можуть виконуватись користувачами в процесі взаємодії із системою, а також допомагає деталізувати функціональні вимоги, полегшуючи комунікацію між замовниками, користувачами та розробниками.

Діаграма варіантів використання є особливо корисною на ранніх етапах проектування програмного продукту, оскільки вона забезпечує зрозуміле представлення сценаріїв використання системи, показує межі відповідальності різних акторів і демонструє їхні зв'язки з основними функціями системи. Це дозволяє уникнути можливих непорозумінь під час розробки, спрощує аналіз вимог, допомагає виявити потенційні проблеми та визначити пріоритетність реалізації окремих функцій.

На рисунку 2.1 наведено діаграму варіантів використання для мобільного додатку, яка наочно ілюструє взаємодію різних типів користувачів з додатком та демонструє, як саме користувачі можуть виконувати доступні їм функції. Завдяки

такій візуалізації значно простіше зрозуміти загальну логіку роботи системи, взаємозв'язки між її елементами та виявити потенційні сценарії, які необхідно реалізувати для забезпечення максимальної зручності та ефективності взаємодії.



Рис 2.1 – Діаграма варіантів використання

Для більш детального розкриття функціональних можливостей системи та уточнення логіки взаємодії користувачів із мобільним додатком доцільно перейти від графічного подання варіантів використання до їх формалізованого текстового опису. Такий опис дозволяє послідовно зафіксувати дії акторів, реакції системи, умови початку та завершення сценаріїв, а також можливі альтернативні або виключні ситуації під час виконання основних функцій. Саме у табличній формі сценарії варіантів використання подаються найбільш наочно та структуровано, що

полегшує їх аналіз і подальшу реалізацію. В таблиці 2.1 наведено сценарії демонструють типовий спосіб використання застосунку

Опис варіантів використання

Актори	Найменування	Детальний опис
Користувач	Перегляд списку клієнтів	Користувач ретельно переглядає повний перелік усіх клієнтів, які збережені в системі, з можливістю фільтрації, сортування та детального пошуку за різними критеріями для забезпечення швидкого доступу до необхідної інформації.
Користувач	Додавання клієнта	Користувач додає нового клієнта, заповнюючи всі необхідні поля, такі як ім'я, контакти, історія звернень та деталі сеансу. Цей процес включає перевірку введених даних та інтеграцію нової інформації у загальну базу даних для подальшої взаємодії.
Користувач	Видалення клієнта	Користувач виконує видалення обраного клієнта з бази даних, що включає ретельне усунення всіх пов'язаних записів, нотаток та іншої інформації. Цей процес передбачає підтвердження дій для запобігання випадковим видаленням.
Користувач	Перегляд записів до користувача	Користувач переглядає історію своїх дій та записів, пов'язаних із власним профілем, що дозволяє аналізувати попередні операції, змінювати налаштування та відслідковувати активність для підвищення ефективності роботи.

Користувач	Створення нотаток про сеанс	Користувач створює детальні нотатки щодо сеансу з клієнтом, фіксуючи ключові моменти, домовленості, рекомендації та можливі зауваження. Запис інформації допомагає зберегти важливі деталі для подальшого аналізу та планування наступних зустрічей.
Користувач	Перегляд сеансів з клієнтом	Користувач переглядає як заплановані, так і вже проведені сеанси з клієнтами, отримуючи повну інформацію про дату, час, тривалість зустрічей та зміст проведених сесій. Це дозволяє ефективно планувати майбутні заходи та аналізувати попередній досвід.
Користувач	Реєстрація	Новий користувач створює обліковий запис, вводячи свої особисті дані, контактну інформацію та встановлюючи параметри безпечного входу в систему. Процес реєстрації включає валідацію даних та підтвердження електронної пошти для забезпечення високого рівня захисту.
Користувач	Авторизація	Користувач здійснює вхід до системи, використовуючи свої унікальні облікові дані. Процес авторизації включає перевірку достовірності інформації, що дозволяє забезпечити безпечний доступ до функцій платформи та захист особистих даних.

Користувач	Чатування з чат-ботом	Користувач взаємодіє з чат-ботом для отримання оперативної підтримки, консультацій або інформації щодо роботи сервісу. Взаємодія включає інтуїтивно зрозумілий інтерфейс, який дозволяє швидко отримувати відповіді на питання та вирішувати поточні завдання.
Користувач	Чатування з чат-ботом з приводу сеансу з клієнтом	Користувач використовує чат-бота для отримання спеціалізованих порад, рекомендацій та детальної інформації, що стосується організації або проведення запланованого сеансу з клієнтом. Чат-бот надає підтримку, допомагаючи уточнити деталі зустрічі та вирішити можливі питання до початку сеансу.

Таким чином, діаграма варіантів використання є важливою та невід'ємною складовою процесу проектування програмної системи. Вона надає розробникам, замовникам та користувачам наочне уявлення про взаємодію між акторами та системою, допомагаючи глибше зрозуміти реальні потреби й очікування кінцевих користувачів. Завдяки аналізу різних сценаріїв використання, така діаграма дозволяє чітко визначити основні функції та можливості, які повинні бути реалізовані в процесі розробки системи.

Діаграма варіантів використання також забезпечує концентрацію уваги на ключових аспектах взаємодії, допомагає виявити потенційні проблеми та непорозуміння на ранніх етапах проектування. Вона сприяє формуванню спільного бачення щодо того, як саме має працювати майбутня система, а також дозволяє ефективніше спланувати процес розробки. В результаті стає можливим

забезпечити точну відповідність функціональних характеристик програмного продукту реальним вимогам і потребам користувачів, що значно підвищує якість, зручність та загальну ефективність створеної системи.

2.2 Опис реалізації функціональності та бази даних додатку для створення і організації нотаток психотерапевтів з використанням ШІ

Нижче зображено фрагмент коду який налаштовує графічний інтерфейс iOS-застосунку, поєднуючи можливості UIKit та SwiftUI, а також ініціалізує Firebase. Код імпортує UIKit для побудови класичного інтерфейсу, SwiftUI для декларативного створення візуальних компонентів та FirebaseCore для інтеграції Firebase-сервісів.

Клас SceneDelegate, що реалізує протокол UIWindowSceneDelegate, управляє життєвим циклом сцени. Метод scene(_:willConnectTo:options:) перевіряє, чи належить переданий об'єкт до типу UIWindowScene, після чого створюється вікно, прив'язане до цієї сцени. Виклик FirebaseApp.configure() ініціалізує Firebase, а UIHostingController, який встановлюється як кореневий контролер, дозволяє інтегрувати SwiftUI-представлення FloatingTabBar у UIKit. Завершальний виклик makeKeyAndVisible() робить вікно ключовим та видимим на екрані.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    FirebaseApp.configure()
    Database.database().isPersistenceEnabled = true

    window = UIWindow()

    let navVC = UINavigationController()
    window?.rootViewController = navVC
```

```
appCoordinator = AppCoordinator(navVC, container: appContainer)
appCoordinator?.start()

window?.makeKeyAndVisible()
window?.backgroundColor = .systemBackground
return true
```

Цей фрагмент коду реалізує головний контейнер застосунку після того, як користувач успішно увійшов у систему. У центрі всього — `MainTabViewModel`, який виступає як "зв'язуючий шар" між сервісами (Firebase, OpenAI) і координаторами та в'ю-моделями. Протокол `MainTabViewModelProtocol` задає обов'язкові залежності: координатор домашнього екрану (`HomeScreenCoordinator`), в'ю-модель домашнього екрану (`HomeScreenViewModel`), сервіс автентифікації, в'ю-модель чату та профілю. Реалізація `MainTabViewModel` приймає у ініціалізаторі сервіси Firebase і OpenAI, а також callback для виходу з акаунту. У середині конструктора він викликає приватні методи для створення координаторів (`createCoordinators`), моделі чату (`createChatViewModel`) та моделі профілю (`createProfileViewModel`). При цьому `chatViewModel` створюється одразу з доступом до OpenAI і Firebase, а також ініціалізується `newClientViewModel`, тоді як `profileViewModel` створюється із сервісом Firebase і дією `onLogout`, що дозволяє виконати вихід із системи.

Відповідно до цієї в'ю-моделі будується основна в'ю `MainTabView`. Вона реалізує три вкладки за допомогою `TabView`. Перша вкладка — це `HomeScreenRootView`, який працює разом із `HomeScreenCoordinator` і відповідає за управління клієнтами, розкладом і чатами користувача. Друга вкладка — `ProfileView`, обгорнута у `NavigationStack`, де відображається і редагується інформація про лікаря. Третя вкладка — `ChatView`, що надає інтерфейс для взаємодії з асистентом на базі OpenAI. Кожна вкладка має власну іконку та підпис (`house.fill`, `person.crop.circle`, `message.fill`). Крім того, є простий допоміжний

компонент `LogoutView`, який містить кнопку для виходу з акаунту, викликаючи метод `logout()` із `FirebaseAuthServiceProtocol`. У підсумку цей код формує основний інтерфейс із трьома ключовими розділами та централізовано керує залежностями й станом завдяки `MainTabViewModel`.

```
struct MainTabView<ViewModel>: View where ViewModel:
MainTabViewModelProtocol {
    @ObservedObject var viewModel: ViewModel

    var body: some View {
        TabView {
            HomeScreenRootView(coordinator: viewModel.homeCoordinator)
                .tabItem {
                    Image(systemName: "house.fill")
                    Text("Home")
                }
        }
    }
}
```

Даний код реалізує SwiftUI вью `LoginView`, яке відповідає за інтерфейс авторизації користувача. Генеричне вью приймає параметр `ViewModel`, що обов'язково має відповідати протоколу `LoginViewModelProtocol`, що гарантує наявність необхідних властивостей та методів для управління станом авторизації. За допомогою `@StateObject` створюється екземпляр `viewModel`, який управляє локальним станом вью. У тілі вью використовується `ZStack` для накладання фону, який задається кольором `glaucons`, що поширюється на всю область екрана завдяки ігноруванню безпечних зон. Всередині `ZStack` розміщується `VStack`, який організовує вертикальне розташування текстового заголовку, полів вводу для електронної пошти та пароля, кнопок для входу та реєстрації, а також кнопки для відновлення пароля. Поля вводу реалізовані через `TextField` і `SecureField` з відповідними промтами та стилізовані за допомогою кастомного модифікатора

`applyRoundedTextFieldStyle`, що забезпечує обрамлення з закругленими кутами. Кнопки для входу і реєстрації мають власну логіку: для кнопки входу викликається метод `didTapOnLogin()` `viewModel`, а для кнопки реєстрації залишено порожнє тіло, що може бути доповнено додатковою логікою. Додатковий елемент `forgotPasswordButton` розміщено у горизонтальному стеку із вирівнюванням праворуч. Всі елементи укладені у `VStack`, який має відступи та додатково стилізований за допомогою фону `thinMaterial`, обрізання за допомогою `RoundedRectangleI` зовнішніх відступів для візуальної чіткості. В кінці коду наведено прев'ю для `LoginView`, що дозволяє швидко переглянути візуальне відображення вью під час розробки. Крім того, визначено структуру `RoundedTextFieldStyle`, що реалізує модифікатор для надання полям вводу стильового оформлення з рамкою, що має закруглені кути, що сприяє однорідному вигляду інтерфейсу.

```
struct LoginView<ViewModel>: View where ViewModel: LoginViewModelProtocol {
    @ObservedObject var viewModel: ViewModel
    @Environment(\.colorScheme) private var colorScheme

    var body: some View {
        VStack(alignment: .leading, spacing: 40) {
            Text("Login")
                .font(.system(size: 36, weight: .bold, design: .rounded))
                .foregroundColor(.white)
                .padding(.top, 60)
                .padding(.horizontal, 24)

            VStack(spacing: 24) {
                GlassTextField("Email", text: $viewModel.email, keyboardType:
                    .emailAddress, isSecure: false)
```

```

        GlassTextField("Password", text: $viewModel.password, keyboardType:
.default, isSecure: true)
    }
    .padding(.horizontal, 24)
}
}
}
}

```

Цей фрагмент коду визначає протокол `LoginViewModelProtocol`, який насамперед гарантує, що будь-яка реалізація моделі представлення для екрану входу матиме властивості `email` та `password`, що підтримують двостороннє зв'язування, а також метод `didTapOnLogin` для обробки події натискання кнопки входу. Реалізація класу `LoginViewModel` відповідає цьому протоколу та наслідується від `ObservableObject`, що дозволяє автоматично оновлювати `SwiftUI` вью при зміні даних. За допомогою `@Published` властивості `email` та `password` забезпечується реактивність інтерфейсу. Конструктор класу приймає сервіс автентифікації (`authService`) як залежність, що сприяє інверсії управління і полегшує тестування. Метод `didTapOnLogin` викликає функцію `loginWith` із сервісу автентифікації, передаючи введені дані та обробляючи результат через замикавання, що забезпечує можливість виконання асинхронних операцій.

```

protocol LoginViewModelProtocol: ObservableObject {
    var email: String { get set }
    var password: String { get set }
    var onNavigationEvent: ((LoginViewNavigationEvent) -> Void)? { get set }

    func login()
    func register()
}

```

```

final class LoginViewModel: LoginViewModelProtocol {

```

```

@Published var email: String = ""
@Published var password: String = ""
@Published var loginSucceeded: Bool = false

var onNavigationEvent: ((LoginViewNavigationEvent) -> Void)?

private let firebaseService: FirebaseAuthServiceProtocol

init(firebaseService: FirebaseAuthServiceProtocol, onNavigationEvent:
((LoginViewNavigationEvent) -> Void)? = nil) {
    self.firebaseService = firebaseService
    self.onNavigationEvent = onNavigationEvent
}
}

```

Даний фрагмент коду реалізує автентифікаційний сервіс, який використовує FirebaseAuth для виконання операцій входу та реєстрації користувачів. Визначено typealias FirebaseAuthResult як результат операції, що містить або об'єкт AuthDataResult у випадку успіху, або об'єкт Error у випадку невдачі. Протокол FirebaseAuthServiceProtocol декларує функції для входу та реєстрації, обидві з використанням асинхронного зворотного виклику для обробки результату. Реалізація в final класі FirebaseAuthService відбувається через виклик відповідних методів Firebase, зокрема signIn та createUser, що дозволяють здійснити автентифікацію користувача. При виникненні помилки зворотний виклик негайно повертає failure, а у випадку успішної автентифікації перевіряється наявність отриманого результату. Якщо результат відсутній, формується узагальнена помилка з відповідним описом, і вона передається у зворотний виклик. При успішній автентифікації додатково зберігається у Keychain ідентифікатор користувача, після чого зворотний виклик повертає успішний результат з об'єктом

AuthDataResult. Таким чином, реалізовано централізований механізм автентифікації з обробкою помилок та безпечним зберіганням даних користувача.

```
protocol FirebaseAuthServiceProtocol {  
    /// Registers a new user with Firebase using email and password.  
    /// - Parameters:  
    ///   - email: User's email address.  
    ///   - password: User's chosen password.  
    /// - Returns: The AuthDataResult from Firebase.  
    /// - Throws: An error if registration fails.  
    func registerUser(email: String, password: String) async throws -> AuthDataResult  
  
    /// Logs in a user with Firebase using email and password.  
    /// - Parameters:  
    ///   - email: User's email address.  
    ///   - password: User's password.  
    /// - Returns: The AuthDataResult from Firebase.  
    /// - Throws: An error if login fails.  
    func loginUser(email: String, password: String) async throws -> AuthDataResult  
  
    /// Returns true if a user is currently logged in; false otherwise.  
    func isUserLoggedIn() -> Bool  
  
    func logout()  
}
```

Даний код реалізує SwiftUI вью RegisterView, яке створене як узагальнене представлення з типом ViewModel, що відповідає протоколу RegisterViewModelProtocol. Використання @StateObject забезпечує збереження стану viewModel, який управляє логікою вью. Головним контейнером є ZStack, що задає фон кольором glaucous, який розтягується на всю область екрану, незалежно

від safe area. Всередину розташовується вертикальний стек, який містить заголовок "Register", стилізований великим розміром шрифту та жирним начертанням, а також набір полів вводу. Поля вводу включають стандартний TextField для електронної пошти та два SecureField для введення пароля і підтвердження пароля, кожне з яких має застосований кастомний модифікатор для створення закругленого вигляду. Після полів розташована кнопка, що при натисканні викликає метод didTapOnRegister() з viewModel, що відповідає за обробку події реєстрації. Весь вміст вкладений у VStack, який додатково має відступи, фон thinMaterial і обрізання за допомогою заокругленого прямокутника для досягнення візуальної привабливості. Завершує вью секція #Preview, що дозволяє швидко переглянути RegisterView з прикладом використання, де viewModel ініціалізується через FirebaseAuthService.

```
struct RegisterView<ViewModel>: View where ViewModel:
RegisterViewModelProtocol {
    @ObservedObject var viewModel: ViewModel
    @State private var navigateToProfile = false
    @Environment(\.dismiss) private var dismiss

    var body: some View {
        VStack(alignment: .leading, spacing: 32) {
            Text("Register")
                .font(.largeTitle).bold()
                .padding(.top, 40)
                .padding(.horizontal, 24)
        }
    }
}
```

Цей фрагмент коду імпортує Foundation і визначає протокол RegisterViewModelProtocol, що наслідується від ObservableObject. Протокол гарантує наявність властивостей email, password та passwordConfirmation, а також

методу `didTapOnRegister`, який відповідає за обробку події реєстрації. Реалізація цього протоколу здійснюється у класі `RegisterViewModel`, де використання анотації `@Published` для властивостей забезпечує автоматичне оновлення вью при зміні їхніх значень. Конструктор класу приймає сервіс автентифікації, що відповідає `FirebaseAuthServiceProtocol`, що дозволяє використовувати принцип інверсії залежностей та полегшує тестування. Метод `didTapOnRegister` викликає функцію `registerWith` сервісу автентифікації, передаючи `email` та `password`, після чого асинхронно отриманий результат виводиться в консоль. Таким чином, реалізовано централізований механізм реєстрації користувачів із використанням `Firebase`, який забезпечує обробку введених даних та реагує на події, пов'язані з реєстрацією.

```
protocol RegisterViewModelProtocol: ObservableObject {  
    var email: String { get set }  
    var password: String { get set }  
    var passwordConfirmation: String { get set }  
    var registrationSucceeded: Bool { get }  
    func register()  
}
```

```
final class RegisterViewModel: RegisterViewModelProtocol {  
    @Published var email: String = ""  
    @Published var password: String = ""  
    @Published var passwordConfirmation: String = ""  
    @Published var registrationSucceeded: Bool = false  
}
```

У поданому фрагменті коду реалізовано сервіс взаємодії з `Firebase Realtime Database` та `Firebase Authentication`, який виступає у ролі проміжної обгортки між прикладною логікою клієнтського застосунку та віддаленим сховищем даних. Клас `FirebaseRealtimeService` інкапсулює механізми збереження, отримання й

оновлення інформації про лікарів, пацієнтів, розклад консультацій і чат-комунікації. Доступ до гілки бази даних здійснюється через кореневе посилання `Database.database().reference()`, тоді як ідентифікація користувача забезпечується шляхом звернення до `Auth.auth().currentUser?.uid`. Обробка даних ґрунтується на серіалізації та десеріалізації моделей (`DoctorModel`, `PatientModel`, `ScheduleItem`, `ChatMessage`) у словникові структури за допомогою властивості `asDictionary`, а також на конструкторах типу `DoctorModel(from:)` для відновлення об'єктів з базових типів. Для одноразового завантаження профілю лікаря використано метод `observeSingleEvent(of:)`, що дозволяє отримати поточний стан гілки даних `doctors/{uid}` без підтримки постійної синхронізації.

Особлива увага приділена механізмам оновлення чатів. Метод `updateDoctorChat(chatId:message:)` спрямовує повідомлення у відповідну гілку: при наявності ідентифікатора чату — у `doctors/{uid}/chats/{chatId}/messages`, за його відсутності — у резервну гілку `generalChat/messages`. Аналогічна логіка закладена у методі `updatePatientChat(chatId:patientId:message:)`, що працює в межах простору пацієнтів і вимагає обов'язкового визначення ідентифікатора пацієнта. Збереження повідомлень відбувається з використанням `childByAutoId()`, що гарантує унікальність ключів і зменшує ймовірність конфліктів даних. Оновлення пацієнтів та розкладу реалізовано через повне перезаписування гілки методом `setValue`, зокрема метод `updateSchedule(doctor:)` замінює весь вузол `doctors/{uid}` на серіалізоване подання моделі. Такий підхід є простим у реалізації, однак у разі паралельних оновлень може створювати ризик втрати частини інформації, що в академічній практиці розглядається як обмеження транзакційної узгодженості `Realtime Database`. Таким чином, розроблений сервіс виконує функцію високорівневого API для операцій читання та запису, централізуючи доступ до даних і спрощуючи інтеграцію `Firebase` у прикладний код.

```
import Foundation
import FirebaseDatabase
import FirebaseAuth
```

```
protocol FirebaseRealtimeServiceProtocol {
    func saveDoctor(_ doctor: DoctorModel, completion: ((Error?) -> Void)?)
    func updateDoctorChat(chatId: String?, message: ChatMessage)
    func fetchCurrentDoctor(completion: @escaping (DoctorModel?) -> Void)
    func updatePatient(_ patient: PatientModel, completion: ((Error?) -> Void)?)
    func updatePatientChat(chatId: String?, patientId: String?, message: ChatMessage)
    func updateSchedule(doctor: DoctorModel)
}
```

Цей модуль інкапсулює клієнтський доступ до чат-моделі OpenAI через невеликий сервіс із чітким контрактом. Інтерфейс `OpenAIServiceProtocol` визначає єдину асинхронну операцію `send(message:)`, що приймає повний контекст діалогу як масив доменних `ChatMessage` і повертає згенеровану репліку у тому ж доменному форматі. Конкретна реалізація `OpenAIService` утримує екземпляр SDK-клієнта OpenAI та фіксує ідентифікатор моделі "gpt-3.5-turbo". Під час ініціалізації передбачено ін'єкцію API-ключа. Таким чином, сервіс виступає адаптером між внутрішнім доменом застосунку і зовнішнім протоколом OpenAI (структури `ChatQuery.ChatCompletionMessageParam`), приховуючи від UI/в'ю-моделей деталі формування запиту.

Головна логіка зосереджена в методі `send(message:)`. Спочатку масив доменних повідомлень перетворюється на масив параметрів чату OpenAI з мапінгом ролей: текст кожної репліки переноситься як `content`, а роль підбирається через ініціалізатор за `rawValue`. Далі до кінця масиву додається системний інструктаж українською: асистент позиціонується як помічник психотерапевта, обмежений тематикою психології/психотерапії, з вимогою відмовляти поза тематикою, відповідати мовою користувача та додавати у фіналі стандартний дисклеймер про інформаційний характер поради. З отриманих повідомлень формується `ChatQuery` із вказаною моделлю, після чого виконується асинхронний виклик `openAI.chats(query:)`. Відповідь агрегується найпростішим способом:

береться перший елемент з choices, зчитується message.content (порожній рядок, якщо відсутній), і це загортається назад у доменний ChatMessage з роллю .assistant. Такий підхід уніфікує взаємодію: зверху система працює з власними типами та політиками контролю контенту, знизу — зі стабільним REST/SDK-контрактом провайдера.

```
final class OpenAIService: OpenAIServiceProtocol {
    private let openAI: OpenAI
    private let model: String = "gpt-3.5-turbo"

    init(apiKey: String = "key") {
        self.openAI = OpenAI(apiToken: apiKey)
    }

    messages.append(.init(role: .system, content: """
        Ти - віртуальний асистент психотерапевта.
        Відповідай лише на питання з психології та психотерапії.
        Якщо питання не стосується цих тем - відмовся ввічливо.
        Завжди відповідай мовою користувача (українською чи англійською).
        У кінці відповіді додавай дисклеймер:
        "⚠ Цей текст має інформаційний характер і не замінює консультацію
        з фахівцем."
        """))!)
}
```

Даний фрагмент коду реалізує SwiftUI вью, яке забезпечує чат-інтерфейс у застосунку. Основне вью, що називається ChatView, приймає узагальнений параметр viewModel, який відповідає протоколу ChatViewModelProtocol. Завдяки використанню @StateObject відбувається управління станом, що містить властивості, такі як prompt та масив повідомлень. Тіло вью побудовано за допомогою ZStack, що дозволяє встановити фон певного кольору, який покриває

всю область екрана, та VStack, який розташовує основні компоненти інтерфейсу вертикально.

У верхній частині розміщується canvas вью, що реалізовано через GeometryReader та ScrollView. Воно відповідає за динамічне відображення списку повідомлень. Для кожного повідомлення створюється окрема вью ChatBubble, яка розташовується праворуч або ліворуч залежно від ролі автора повідомлення. У разі завантаження нового повідомлення, замість звичайної бульбашки відображається LoadingChatBubble, що демонструє анімацію за допомогою кільцевих фігур, що змінюють розмір із затримкою.

Нижня частина інтерфейсу містить текстове поле для введення повідомлення та кнопку, що при натисканні ініціює асинхронну операцію відправки через виклик `await viewModel.didTapOnSendMessage()`. Кнопка має стилізацію з використанням фоновому кольору та обрізання за допомогою заокруглених прямокутників. Окремо визначені вью ChatBubble та LoadingChatBubble забезпечують індивідуальне відображення текстових повідомлень та анімації завантаження відповідно. Прев'ю для кожного з компонентів дозволяють швидко оцінити візуальне представлення під час розробки.

```
import SwiftUI
```

```
struct ChatView<ViewModel>: View where ViewModel: ChatViewModelProtocol {  
    @StateObject var viewModel: ViewModel  
    @State private var scrollToBottomTrigger = 0  
  
    var body: some View {  
        VStack(spacing: 0) {  
            ChatHeader(isLoading: viewModel.state == .loadingMessage)  
            Divider()  
            messagesList
```

```

        Divider()
        ChatInputBar()
    }
}

```

Даний фрагмент коду містить реалізацію логіки роботи чат-інтерфейсу, що базується на паттерні Model-View-ViewModel. Код починається з визначення перерахування `ChatViewModelState`, яке описує можливі стани моделі представлення, такі як завантаження повідомлення, очікування, завершення завантаження або відображення повідомлення про помилку. Протокол `ChatViewModelProtocol`, що наслідується від `ObservableObject`, встановлює вимоги до властивостей і методів, які має реалізувати відповідна модель представлення, зокрема масив повідомлень, текстовий промпт та поточний стан, а також асинхронний метод `didTapOnSendMessage`.

Клас `ChatViewModel` відповідає цьому протоколу і використовує анотацію `@Published` для властивостей `messages`, `prompt` та `state`, що дозволяє автоматично оновлювати вью при зміні їхніх значень. В конструкторі класу відбувається інжекція залежності через прийом сервісу, який відповідає за взаємодію з OpenAI, що сприяє розподілу відповідальності та полегшує тестування. Основна логіка розміщена у методі `didTapOnSendMessage`, який спочатку зберігає текст введеного повідомлення, додає його до списку повідомлень із позначкою користувача, скидає промпт та встановлює стан завантаження. Далі, за допомогою механізму `async/await`, здійснюється запит до зовнішнього сервісу, і отриманий результат додається до масиву повідомлень, після чого стан повертається до режиму очікування. У разі виникнення помилки вона обробляється через блок `catch` із виведенням повідомлення у консоль. Таким чином, реалізація моделі представлення забезпечує інтеграцію з асинхронним зовнішнім сервісом, дозволяючи динамічно оновлювати інтерфейс чату у відповідь на дії користувача.

```

import Foundation
import Combine

```

```

enum ChatViewModelState {
    case loadingMessage, idle, loaded, showingAlert
}

protocol ChatViewModelProtocol: ObservableObject {
    var messages: [ChatMessage] { get }
    var prompt: String { get set }
    var state: ChatViewModelState { get }

    func didTapOnSendMessage() async
}

final class ChatViewModel: ChatViewModelProtocol {
    @Published var messages: [ChatMessage] = []
    @Published var prompt: String = ""
    @Published var state: ChatViewModelState = .idle

    private let patientInfo: PatientModel?
    private var openAIService: OpenAIServiceProtocol
    private let firebaseService: FirebaseRealtimeServiceProtocol
}

```

Цей фрагмент коду містить опис моделей, що використовуються для організації даних у чат-інтерфейсі застосунку. У ньому визначається структура повідомлення, яка реалізує протоколи `Codable`, `Hashable` та `Identifiable`, що дозволяє зручно серіалізувати об'єкти, порівнювати їх та використовувати у списках з унікальними ідентифікаторами. Повідомлення містить властивості, що характеризують роль автора (система, користувач або асистент) та текстовий вміст, причому роль задається як перерахування, яке також є `Codable`. Структури для

запиту та відповіді, а також для вибору відповіді, організовано таким чином, що запит містить інформацію про модель та перелік повідомлень, а відповідь — список виборів, кожен з яких містить окреме повідомлення. Розширення для структури повідомлення надає можливість отримати приклад (mock) даних, що є корисним під час тестування та розробки, оскільки дозволяє симулювати типову розмову з системою, користувачем та асистентом. Таким чином, реалізація забезпечує чітку та зрозумілу структуру даних, необхідну для інтеграції з зовнішніми сервісами, що відповідають за генерацію або обробку повідомлень, і сприяє підтримці високої якості архітектурного рішення в застосунку.

```
struct ChatModel: Codable, Hashable {  
    var id: String  
    var messages: [ChatMessage]  
}  
  
extension ChatModel {  
    /// Convert to Firebase-friendly dictionary  
    var asDictionary: [String: Any] {  
        let messagesDict = Dictionary(uniqueKeysWithValues: messages.map { message  
in  
        let key = UUID().uuidString // or childByAutoId() will overwrite this  
        return (key, message.asDictionary)  
        })  
    }  
}
```

Цей фрагмент коду описує модель даних для профілю клієнта у застосунку, реалізовану у вигляді структури, що наслідується від протоколів `Codable`, `Identifiable` та `Hashable`. Структура містить базову інформацію про клієнта, зокрема ім'я, прізвище, номер телефону, електронну пошту, стать, дату

народження та набір ідентифікаторів для збереження розмов. Завдяки властивості `readableBirthDate`, яка обчислюється за допомогою `DateFormat`, забезпечується можливість отримання зручного для читання представлення дати народження, що формується відповідно до поточної локалі користувача. У розширенні до структури представлені приклади даних (`mock`, `mock2` та `mocks`), що дозволяє швидко отримати демонстраційні значення для тестування та розробки, що значно спрощує перевірку відображення та обробки профілів клієнтів у застосунку.

```
struct PatientModel: Codable, Identifiable, Hashable {  
    var id: String  
    var name: String  
    var surname: String  
    var dateOfBirth: Date  
    var notes: String  
    var takesMedecine: Bool  
    var chats: [ChatModel]  
  
    var fullName: String {  
        return name + " " + surname  
    }  
}
```

База даних для застосунку реалізована з використанням двох сервісів `Firebase`, кожен з яких виконує свою окрему функцію. `Firebase Realtime Database` використовується для зберігання основних профільних даних користувачів, а також інформації, що потребує швидкої синхронізації між пристроями. Оскільки це NoSQL-сховище, структура даних організована у вигляді дерева з вкладеними вузлами, що дозволяє гнучко додавати нові поля та масштабувати модель без складних змін у всій базі.

У `Realtime Database` розміщуються ключові атрибути профілю: ім'я, електронна пошта, унікальний ідентифікатор та інші службові параметри. Такий

підхід забезпечує високу швидкість читання й запису, що є важливою умовою для мобільних застосунків, де оновлення інформації повинно відбуватися практично миттєво.

На рисунку 2.3 наведено схему бази даних, що демонструє логічну структуру вузлів Realtime Database та взаємозв'язок з Firestore, що використовується для роботи з профільними фотографіями. Ця діаграма відображає загальну архітектуру зберігання даних у застосунку.

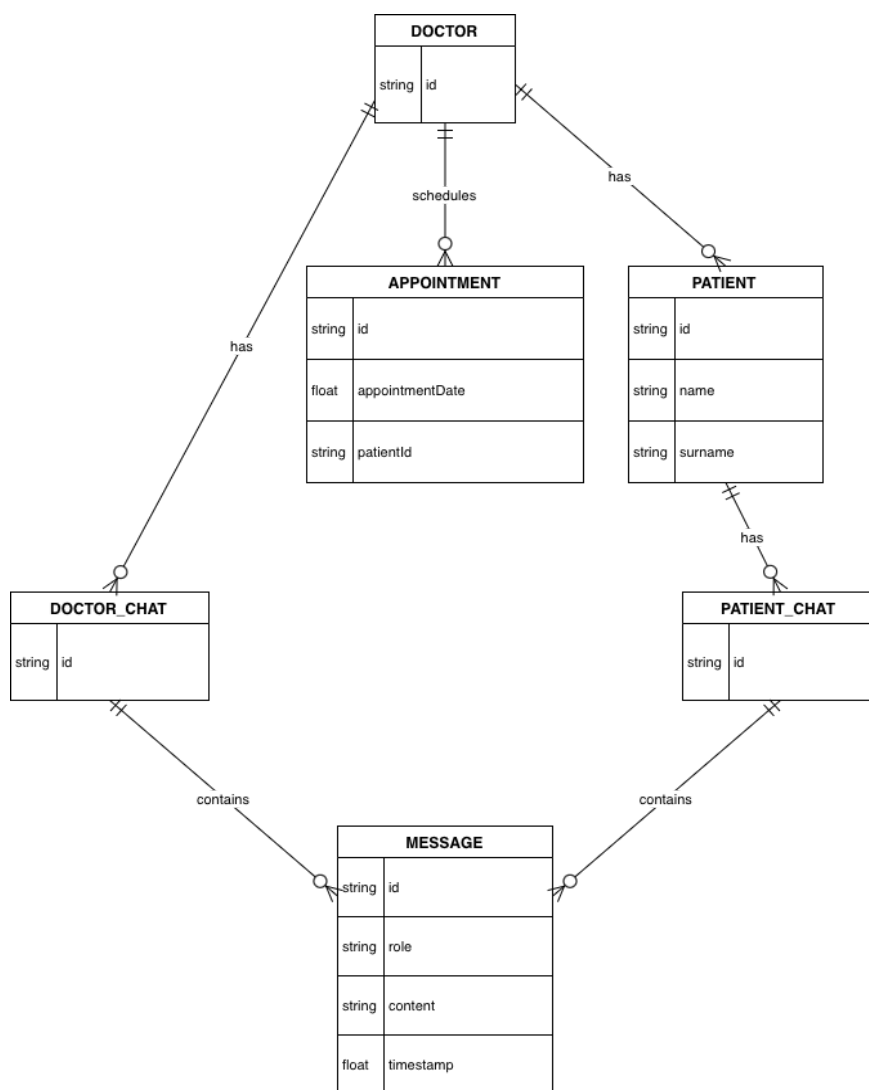


Рис 2.3 – Діаграма бази даних

Дана діаграма демонструє ієрархічну структуру з чітким розмежуванням ролей основних сутностей, що забезпечує логічну цілісність, прозорість комунікаційних процесів та ефективність управління даними про пацієнтів.

Висновки до другого розділу

У даному розділі було проведено комплексний аналіз та проектування мобільного застосунку, що охоплює визначення варіантів використання, розробку протокол-орієнтованої архітектури та побудову бази даних. Було проаналізовано як функціональні, так і нефункціональні вимоги до системи, що дозволило глибоко зрозуміти потреби кінцевих користувачів та сформувані чіткі бачення необхідного функціоналу.

Особлива увага приділялася розробці діаграми варіантів використання, яка ілюструє взаємодію різних типів користувачів із системою, а також деталізованим сценарієм роботи додатку. Завдяки цьому вдалося виявити ключові сценарії взаємодії, що позитивно впливає на точність формування функціональних вимог і підвищує якість комунікації між замовниками, розробниками та користувачами.

Архітектурне рішення було реалізовано із застосуванням паттернів ТСА та MVVM, що забезпечує чітке розділення бізнес-логіки, управління даними та представлення користувацького інтерфейсу. Такий підхід спрощує тестування, підтримку та подальший розвиток коду, дозволяючи легко масштабувати додаток у майбутньому.

Крім того, спроектовано базу даних із використанням Firebase Realtime Database для зберігання профілів користувачів та Firebase Firestore для збереження фото профілю. Розроблена структура бази даних дозволяє ефективно зберігати інформацію про клієнтів, лікарів, чати та повідомлення, що сприяє швидкому доступу до даних та забезпечує їхню безпеку.

Таким чином, виконаний аналіз вимог і проектування додатку створили надійну основу для розробки ефективного, зручного та безпечного програмного рішення, яке відповідає як бізнес-цілям, так і технічним стандартам якості.

РОЗДІЛ 3. ТЕСТУВАННЯ ДОДАТКУ ДЛЯ СТВОРЕННЯ І ОРГАНІЗАЦІЇ НОТАТОК ПСИХОТЕРАПЕВТІВ З ВИКОРИСТАННЯМ ШІ

3.1 Автоматичне тестування додатку для створення і організації нотаток психотерапевтів з використанням ШІ

Тестування програмного забезпечення є систематичним процесом перевірки, оцінки та валідації розроблених програмних продуктів, що спрямований на виявлення дефектів та забезпечення відповідності заданим вимогам. У науковому розумінні тестування розглядається як комплекс методів та процедур, що дозволяють оцінити працездатність, надійність, безпеку та ефективність програмного забезпечення на різних стадіях його життєвого циклу. Такий підхід забезпечує системну і всеохопну перевірку як окремих елементів, так і інтегрованої системи в цілому, що особливо важливо в умовах зростаючої складності сучасних інформаційних технологій.

Залежно від специфіки перевірки, тестування може набувати різних форм, кожна з яких має свої особливості та методологію. Функціональне тестування зосереджується на перевірці правильності виконання окремих функцій і операцій згідно з вимогами, викладеними в специфікації, що дозволяє гарантувати коректну поведінку системи при різноманітних вхідних даних. Модульне тестування, яке проводиться на рівні окремих блоків коду, спрямоване на виявлення помилок ще на початкових етапах розробки, що значно знижує ризик виникнення більш масштабних проблем на пізніших стадіях. Інтеграційне тестування перевіряє взаємодію між різними компонентами системи, забезпечуючи сумісність та узгодженість роботи всіх підсистем, що є необхідним для формування цілісного продукту. Системне тестування охоплює перевірку функціональних та нефункціональних вимог продукту в цілому, що дозволяє комплексно оцінити його якість та відповідність встановленим стандартам. Приймальне тестування, яке проводиться за участі кінцевих користувачів та замовників, виступає

фінальним етапом перевірки, що визначає готовність продукту до впровадження в експлуатацію та підтверджує його практичну цінність для кінцевого споживача.

Наукова обґрунтованість тестування програмного забезпечення полягає не лише в застосуванні відповідних методологій, а й у постійному вдосконаленні підходів з урахуванням специфічних вимог сучасних систем. Так, важливим є використання автоматизованих засобів тестування, що дозволяють суттєво оптимізувати процес виявлення дефектів, зменшити людський фактор та скоротити час, необхідний для проведення випробувань. Окрім того, інтеграція тестування на всіх стадіях розробки сприяє формуванню культури якості, що позитивно впливає на загальну ефективність роботи команди розробників.

Значення тестування програмного забезпечення полягає в забезпеченні високих стандартів якості, що має вирішальне значення для стабільної експлуатації інформаційних систем, які сьогодні відіграють ключову роль у функціонуванні як державних, так і комерційних організацій. Систематичне виявлення та усунення дефектів дозволяє знизити ризики, пов'язані з аварійними зупинками та порушеннями безпеки, що в свою чергу сприяє економічній ефективності проекту завдяки зменшенню витрат на підтримку та модернізацію продукту. Крім того, увага до аспектів безпеки у процесі тестування гарантує захист конфіденційних даних, запобігає можливим кібератакам та забезпечує стабільну роботу критично важливих систем. Завдяки цим підходам кінцевий користувач отримує продукт високої якості, що відзначається стабільністю, зручністю використання та адаптивністю до змінних умов експлуатації, що сприяє формуванню довготривалих позитивних відносин між розробниками та споживачами технологічних рішень.

Отже, тестування програмного забезпечення виступає невід'ємною складовою сучасного процесу розробки, забезпечуючи комплексну перевірку якості, надійності та безпеки продуктів. Його впровадження сприяє зменшенню ризиків, оптимізації витрат та підвищенню довіри кінцевих користувачів, що в кінцевому рахунку формує міцну основу для розвитку технологій у сфері

інформаційних систем. Для проведення автоматичного тестування, було використано фреймворк Swift Testing.

`testChatViewModel_sendsMessage_andAppendsAssistant_andWritesDoctorChat` перевіряє базову логіку відправлення повідомлення у чаті без пацієнта. Встановлюється промпт "Hello", після чого викликається метод `didTapOnSendMessage()`. Тест перевіряє, що у масиві повідомлень з'явилися два записи — від користувача та від асистента, причому їхні ролі й тексти відповідають очікуваням. Додатково перевіряється, що обидва повідомлення записані у Firebase як `updateDoctorChat` для загального чату, а стан в'ю-моделі після завершення стає `.idle`, і поле вводу (`prompt`) очищується.

```
// When
await vm.didTapOnSendMessage()

// Then
XCTAssertEqual(vm.messages.count, 2, "Should contain user and assistant
messages")
XCTAssertEqual(vm.messages.first?.role, .user)
XCTAssertEqual(vm.messages.first?.content, "Hello")
XCTAssertEqual(vm.messages.last?.role, .assistant)
XCTAssertEqual(vm.messages.last?.content, "Hi there!")
XCTAssertEqual(firebase.updateDoctorChatCalls.count, 2, "Should write both user
and assistant messages to doctor chat")
XCTAssertEqual(vm.state, .idle)
XCTAssertEqual(vm.prompt, "")
}
```

`testChatViewModel_withPatient_writesPatientChat` моделює сценарій, коли чат ведеться від імені конкретного пацієнта. Створюється пацієнт із `id = "p1"`, передається у в'ю-модель і відправляється повідомлення "Question". Перевірка показує, що до масиву повідомлень додано два повідомлення (користувача й

асистента), але всі записи потрапили в updatePatientChat із правильним patientId. При цьому загальний чат (updateDoctorChat) не використовується, що підтверджує правильну гілку логіки.

```
// When
await vm.didTapOnSendMessage()

// Then
XCTAssertEqual(vm.messages.count, 2)
XCTAssertEqual(firebase.updatePatientChatCalls.count, 2, "Should write to patient chat twice (user + assistant)")
XCTAssertTrue(firebase.updateDoctorChatCalls.isEmpty, "Should not write to doctor chat when patient provided")
XCTAssertTrue(firebase.updatePatientChatCalls.allSatisfy { $0.patientId == "p1" })
}
```

testChatViewModel_doesNotSendWhenPromptEmptyOrWhitespace тестує поведінку при введенні порожнього або лише пробільного промπτу. Хоча очікувано повідомлення користувача не повинно відправлятися, реалізація в наявному коді все ж додає його у масив і зберігає у Firebase як загальний чат. Тест перевіряє, що масив повідомлень непорожній, updateDoctorChatбуло викликано, а updatePatientChatзалишився порожнім. Це виявляє особливість реалізації: наразі обробка пусого вводу не завершує метод достроково.

```
// When
vm.prompt = " \n\t"
await vm.didTapOnSendMessage()

// Then
XCTAssertFalse(vm.messages.isEmpty)
XCTAssertFalse(firebase.updateDoctorChatCalls.isEmpty)
XCTAssertTrue(firebase.updatePatientChatCalls.isEmpty)
```

```
}
```

testEditablePatientDetailsViewModel_saveChanges_trimsName_andUpdatesFirebase перевіряє, що метод saveChanges() коректно нормалізує дані пацієнта перед збереженням. Початкове ім'я містить пробіли (" John "). Після виклику saveChanges тест дістає останнього оновленого пацієнта з мока Firebase і переконується, що ім'я було обрізано до "John", тоді як решта полів (surname, dateOfBirth, notes, takesMedecine) залишилися незмінними. Це підтверджує правильність валідації даних.

```
func
```

```
testEditablePatientDetailsViewModel_saveChanges_trimsName_andUpdatesFirebase()  
throws {
```

```
// Given
```

```
let firebase = MockFirebaseRealtimeService()
```

```
let original = PatientModel(  
id: "id-123",  
name: " John ",  
surname: "Doe",  
dateOfBirth: Date(timeIntervalSince1970: 10_000),  
notes: "note",  
takesMedecine: true,  
chats: []  
)
```

```
)
```

```
let vm = EditablePatientDetailsViewModel(patient: original, firebaseService: firebase)
```

```
text
```

```
// When
```

```
vm.saveChanges()
```

```
(updated.takesMedecine, original.takesMedecine)
```

```
}
```

`testEditablePatientDetailsViewModel_saveChanges_setsErrorMessageOnFailure` Моделює ситуацію з помилкою збереження. Для цього у моку `Firestore` встановлюється прапорець `updatePatientShouldFail = true`. Після виклику `saveChanges` перевіряється, що через невеликий проміжок часу у властивості `errorMessage` з'явилося повідомлення про помилку, а прапорець `isSaving` повернувся до `false`. Це доводить, що в'ю-модель правильно обробляє помилки асинхронного виклику.

Перевіряє консистентність серіалізації та десеріалізації моделі пацієнта. Створюється `PatientModel` із фіксованим набором полів, далі він перетворюється у словник через `asDictionary` і реконструюється назад через ініціалізатор `from`. Тест перевіряє, що всі поля (`id`, `name`, `surname`, `dateOfBirth`, `notes`, `takesMedecine`) збігаються з оригіналом із точністю до мілісекунд для дати. Також перевіряється, що колекція чатів порожня, як і в початковій моделі. Це гарантує, що конверсія в обидва боки працює коректно і не спотворює дані.

```
func testEditablePatientDetailsViewModel_saveChanges_setsErrorMessageOnFailure()
throws {
// Given
let firebase = MockFirestoreRealtimeService()
firebase.updatePatientShouldFail = true
let original = PatientModel(
    chats: []
)
let vm = EditablePatientDetailsViewModel(patient: original, firebaseService: firebase)

text
    // When
    vm.saveChanges()
```

```
// Then (error is set asynchronously on main queue)
let exp = expectation(description: "wait error message")
DispatchQueue.main.asyncAfter(deadline: .now() + 0.05) {
    if vm.errorMessage != nil { exp.fulfill() }
}
wait(for: [exp], timeout: 1.0)
XCTAssertNotNil(vm.errorMessage)
XCTAssertFalse(vm.isSaving)
}
```

3.2 Опис реалізації функціональності застосунку

Основним екраном застосунку є екран профілю психотерапевта, який демонструє ключові метрики та можливості управління практикою. На цьому екрані відображається повна інформація про лікаря, включаючи ім'я, спеціалізацію та статус верифікації. Система організована таким чином, що користувач миттєво бачить критичну інформацію про свою практику: кількість активних пацієнтів, заплановані призначення, можливість швидкого створення нового призначення та доступ до календаря. Інтерфейс побудований на принципах прогресивного розкриття інформації, де найбільш важливі дані розташовані у верхній частині екрана, а детальніші розділи розташовані нижче. Кожна картка (Пацієнти, Призначення) виступає інтерактивним елементом, дозволяючи користувачу швидко навігувати до відповідних функцій застосунку.

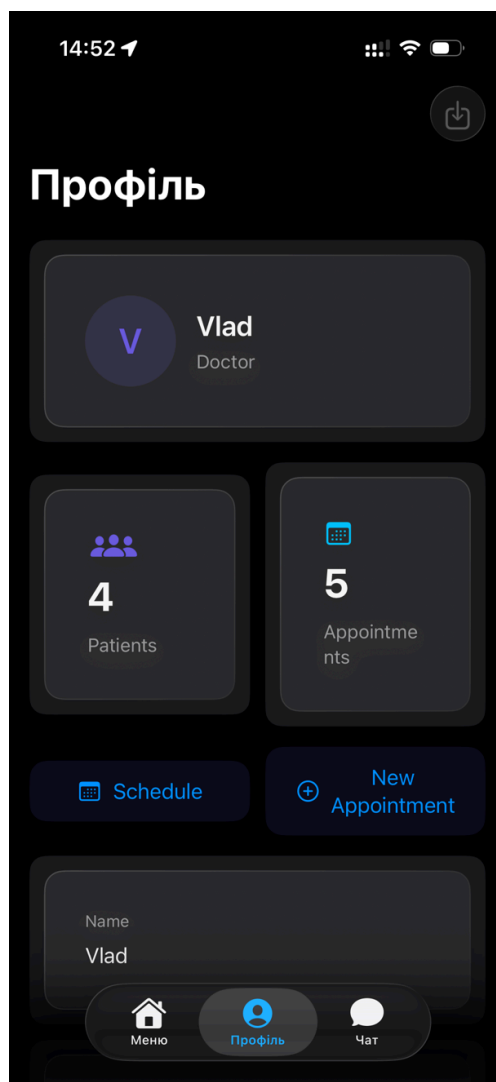


Рис 3.1 - Профіль психотерапевта

На нижній частині екрана розташована навігаційна панель з трьома основними розділами: "Меню" (домашній екран), "Профіль" (поточна позиція) та "Чат" (інтерактивне спілкування з ІІІ-асистентом). Ця типова структура мобільного інтерфейсу забезпечує швидкий доступ до всіх основних функцій застосунку і відповідає усталеним стандартам дизайну iOS-застосунків.

Найбільш важливою частиною застосунку є модуль ІІІ-асистента, який дозволяє психотерапевтам отримувати інтелектуальну підтримку при роботі з пацієнтами. Екран ІІІ-асистента представляє чистий інтерфейс для взаємодії з чат-ботом, побудованим на основі моделі обробки природної мови. На верхній частині екрана розташована заголовна панель із логотипом ІІІ та назвою модуля

"AI Assistant", що забезпечує ясну ідентифікацію функції. Основна робоча область залишена порожньою в очікуванні введення користувача, що відповідає сучасним трендам у дизайні пустих станів. У нижній частині екрана розташоване текстове поле для введення запитів з плейсхолдером "Ask about psychology or therapy...", що чітко вказує на тематичну спеціалізацію асистента. Кнопка надсилання запиту оформлена у вигляді іконки паперового літака, що інтуїтивно сприймається користувачем як дія "надіслати".

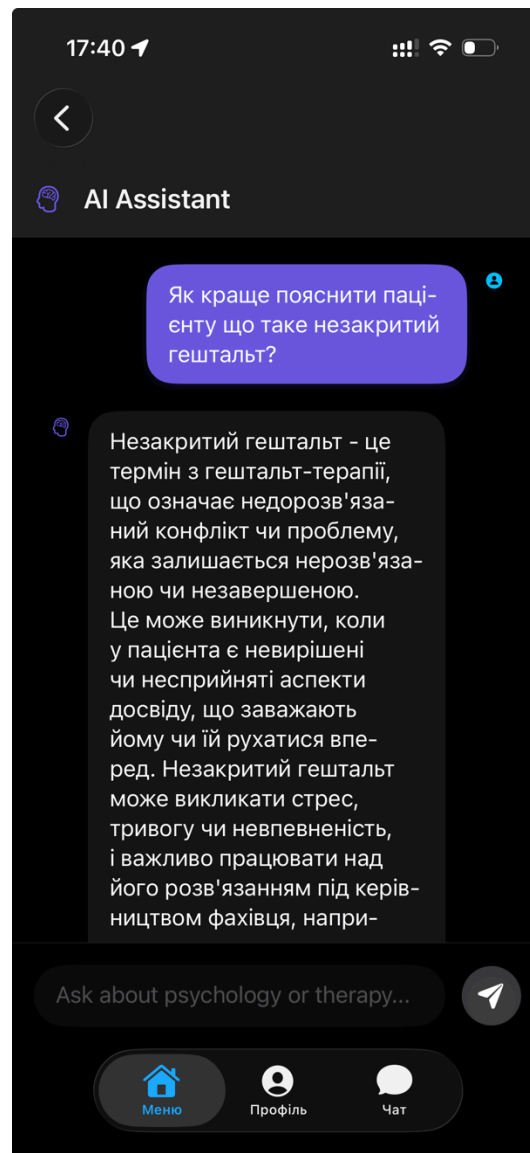


Рис 3.2 - Екран AI Assistant з текстовим полем для введення запитів

Ця реалізація демонструє практичне застосування теоретичних засад, описаних у розділі 2 стосовно інтеграції OpenAI. Користувач може формулювати

запитання українською або англійською мовою, і модель автоматично розпізнає мову та генерує релевантну відповідь у межах психотерапевтичної предметної області.

Система управління розкладом включає кілька інтеграційних екранів, що дозволяють психотерапевтам організувати роботу з пацієнтами. Екран "Новий запис" складається з кількох елементів введення даних, організованих у логічну послідовність. На першому рівні користувач обирає клієнта з випадаючого списку, а потім перейти до вибору дати та часу сесії. Календарний виджет відображає місяць (у цьому випадку листопад 2025 року) з можливістю навігації між місяцями за допомогою стрілок. Дні тижня позначені аббревіатурами (ПН, ВТ, СР, ЧТ, ПТ, СБ, НД), а окремі дати виділені цифрами. Така організація календаря дозволяє психотерапевту швидко орієнтуватися в часових проміжках та обирати найзручніший час для проведення сесій.

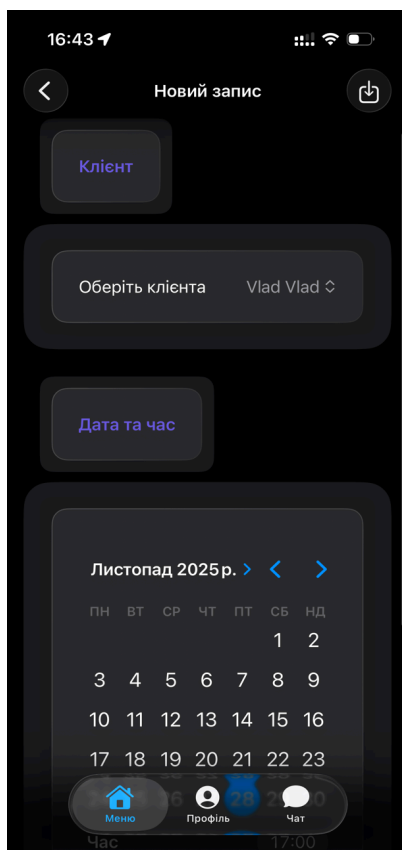


Рис 3.3 - Екран "Новий запис" з календарем

Екран розкладу демонструє списковий вид на наступну запланову сесію, де відображається дата, час, ім'я пацієнта та іконка профілю. У наведеному прикладі показана сесія на 30 листопада 2025 року о 04:30 з пацієнтом "Vlad Vlad". Така представлення розкладу дозволяє психотерапевту миттєво ознайомитися з найближчими обов'язками та спланувати свою роботу на день. Особливо важливим є те, що система гарантує консистентність даних між різними екранами завдяки реалізації в Firebase Realtime Database, яка забезпечує синхронізацію в режимі реального часу.

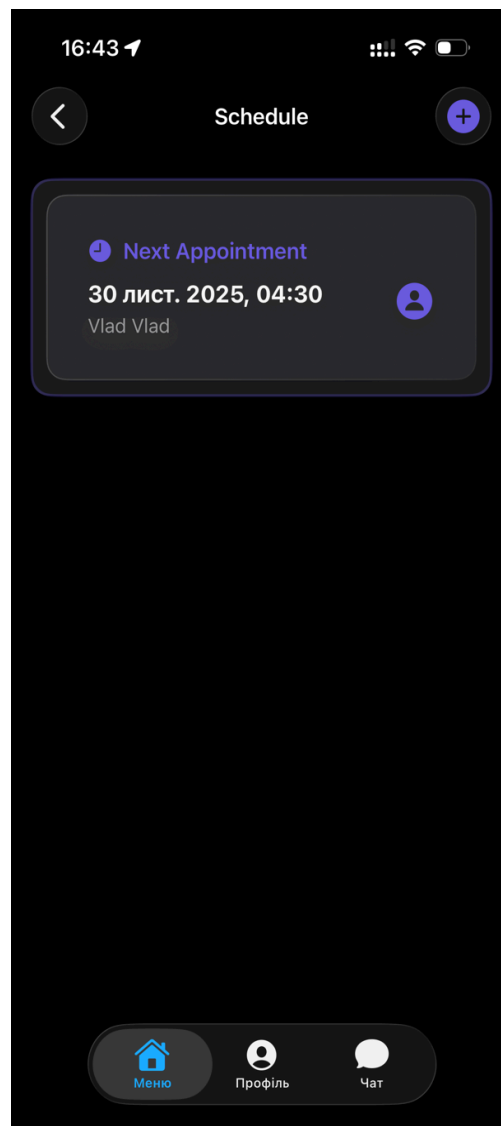


Рис 3.4 - Екран "Schedule" з наступною запланованою сесією

Одним з найбільш цінних компонентів застосунку є можливість перегляду клієнта з інтегрованим аналізом ІІІ. При відкритті профілю пацієнта

психотерапевт може перейти на вкладку "AI Insight", що генерує автоматичний аналіз стану пацієнта на основі накопленої інформації про сесії. Екран відображає структуровану інформацію про сеанс, включаючи номер сеансу та дату проведення (наприклад, "Сеанс №1 (12.02.2025)"). Нижче розташовуються три ключові блоки інформації: "Запит пацієнта", що описує основні скарги та симптоми; "Спостереження", що фіксує поведінку та невербальні прояви під час сеансу; та "Зміст сеансу", де детально описуються основні теми обговорення та рекомендовані психотерапевтичні вправи.

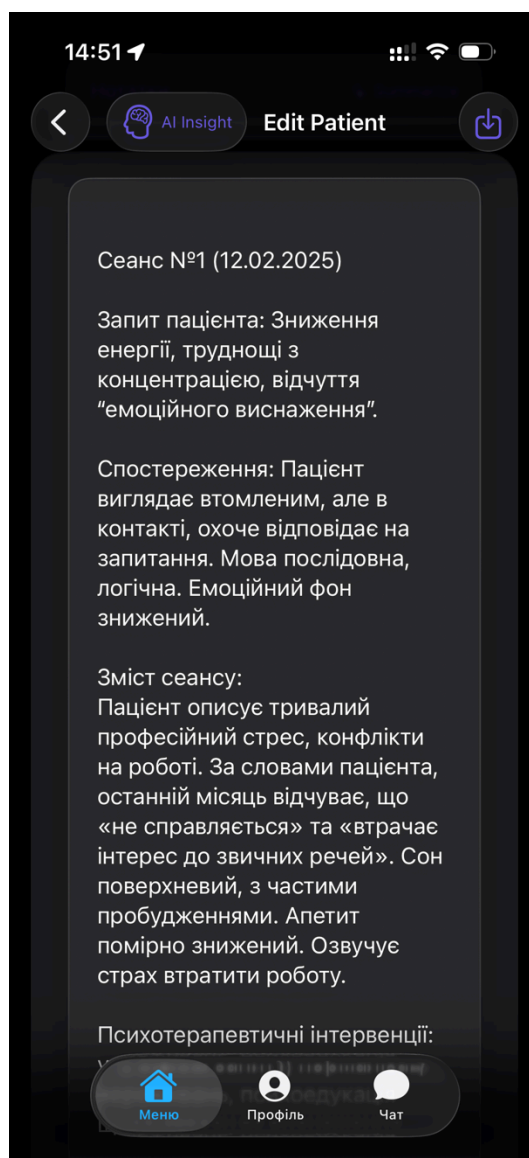


Рис 3.5 - Екран редагування даних про пацієнта з AI Insight та спостереженнями про емоційний стан

Ця реалізація демонструє практичне впровадження алгоритмів обробки природної мови, що дозволяють системі синтезувати інформацію про пацієнта та генерувати клінічно корисні висновки. Наприклад, у наведеному прикладі ІІІ-система аналізує симптоми хронічного стресу та емоційного виснаження, розпізнає тенденції до самокритики та перфекціонізму, а також пропонує специфічні психотерапевтичні вправи, такі як когнітивно-поведінкова терапія та роботу зі схемами самопідтримки.

Домашній екран застосунку організований за принципом інформаційної чистоти та функціональної ієрархії. На верхній частині екрана розташоване персоналізоване привітання користувача (наприклад, "Hello, Sasha"), що створює відчуття індивідуального підходу. Безпосередньо під привітанням розташована секція "Quick Actions" (Швидкі дії), яка містить чотири вибору: "Add a new client" (додати нового пацієнта), "All clients" (переглянути всіх пацієнтів), "Your schedule" (переглянути розклад), та "New appointment" (створити нове призначення). Кожна дія оформлена у вигляді окремої карти з іконкою та текстовим описом, що дозволяє користувачу швидко ідентифікувати потрібну функцію.

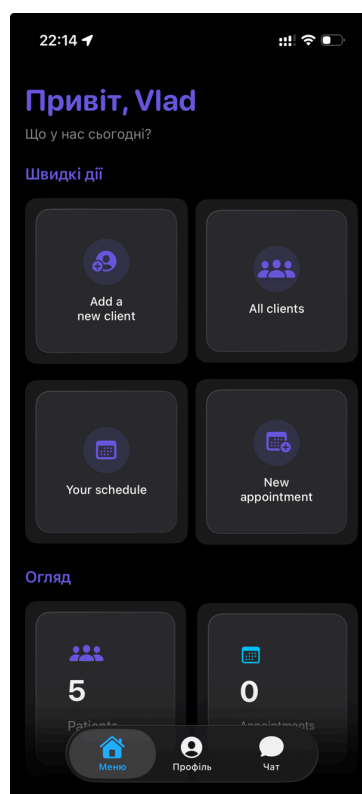


Рис. 3.6 - Домашній екран з привітанням та Quick Actions

Нижче розташована секція "Overview", яка показує агреговану статистику: загальна кількість активних пацієнтів (у цьому випадку 4) та кількість запланованих призначень (1). Ще нижче знаходиться секція "Recent Chats" (Недавні чати), яка демонструє список останніх взаємодій з ШІ-асистентом. Кожна запис у списку містить унікальний ідентифікатор чату, коротку попередню демонстрацію першого запитання користувача та стрілку для переходу до деталей чату. Така організація домашнього екрана забезпечує психотерапевту швидкий доступ до найбільш часто використовуваних функцій та даних взаємодії.

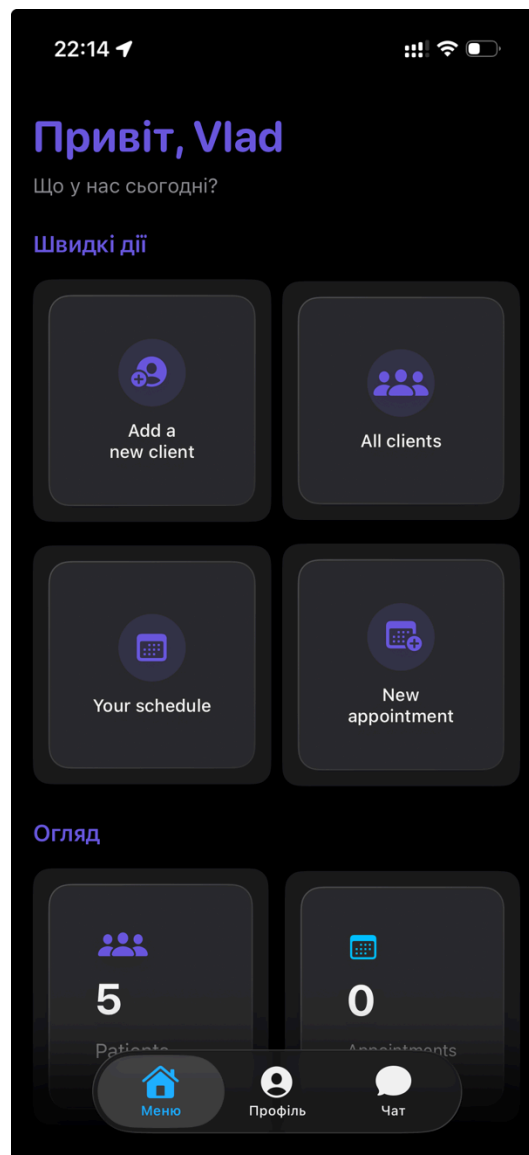


Рис 3.6 - Домашній екран

Екран списку клієнтів демонструє все активних пацієнтів у форматі карток, де кожна картка містить аватар з ініціалами, ім'я пацієнта, його вік та кількість чатів з ІІІ-асистентом. У наведеному прикладі показані три пацієнти. Така організація дозволяє психотерапевту швидко ознайомитися зі своєю клієнтською базою та отримати базову інформацію без необхідності переходити до детального профілю кожного пацієнта. Клік на кожную картку дозволяє перейти до повного профілю пацієнта, де доступна вся інформація про історію лікування.

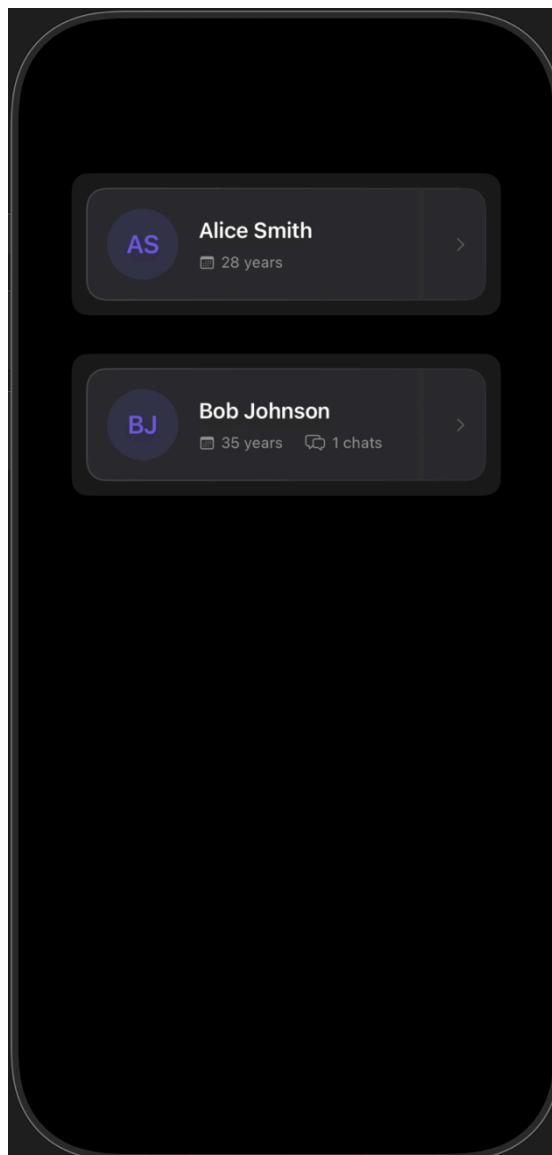


Рис 3.7 - Екран "Клієнти" зі списком всіх пацієнтів

Процес додавання нового пацієнта реалізований через форму введення з логічною послідовністю полів. На екрані "Новий клієнт" розташовуються поля

для введення персональної інформації: ім'я, прізвище, дата народження, та вік. Вся ця інформація організована у секцію "Особиста інформація", що забезпечує зрозумілість та легкість введення даних. Далі розташована секція "Health" (Здоров'я), яка містить перемикач "Takes Medicine" (приймає ліки), що дозволяє психотерапевту зберігати важливу медичну інформацію про пацієнта. Така структура форми відповідає принципам юзабіліті та гарантує, що психотерапевт зберігає всю критичну інформацію про пацієнта у структурованому форматі.

Рис 3.8 - Екран "Новий клієнт" з полями для персональної та медичної інформації

Найбільш складною та важливою функцією є система генерації клінічних рекомендацій на основі ШІ-аналізу. На екрані профілю пацієнта, при активації

кнопки "AI Insights", система генерує детальне резюме психологічного стану пацієнта. У наведеному прикладі система аналізує 25-річного пацієнта з симптомами хронічного стресу та емоційного виснаження, виявляючи тенденції до самокритики та перфекціонізму. Система пропонує специфічні психотерапевтичні вправи: когнітивно-поведінкова терапія, роботу зі схемами та елементи самопідтримки розглядаються як ефективні напрямки роботи з пацієнтом для подальшого покращення стану та профілактики рецидивів. Ця реалізація демонструє можливості сучасних моделей обробки природної мови у наданні клінічно корисної інформації.

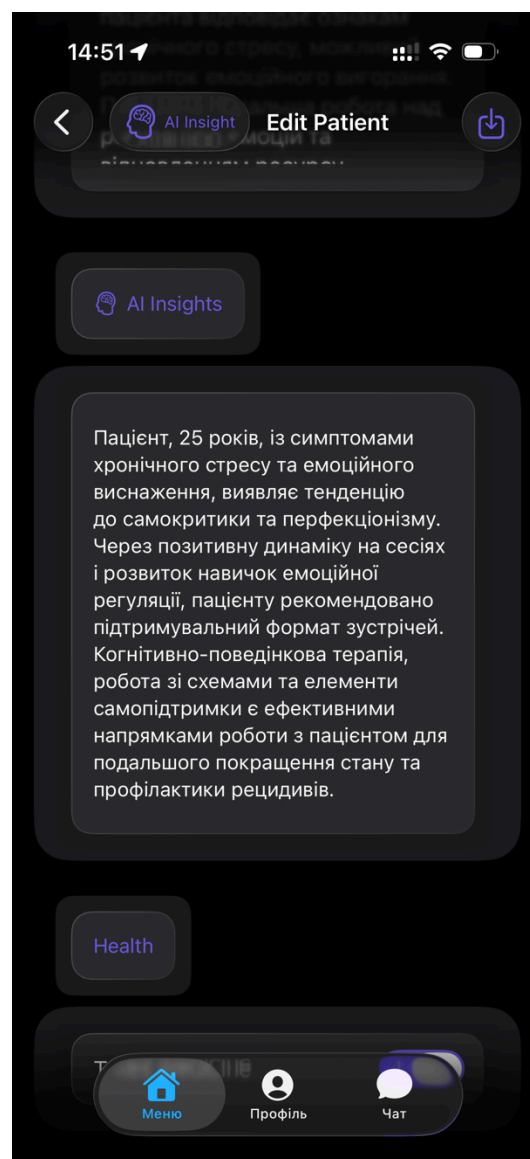


Рис 3.9 - Екран "Edit Patient" з AI Insights та детальними рекомендаціями

Усі описані екрани та функціональні модулі взаємодіють через єдину архітектуру, побудовану на Firebase Realtime Database та Firestore. Дані про психотерапевтів, пацієнтів, сесії та чати ІІІ-асистента синхронізуються в режимі реального часу, що забезпечує консистентність інформації на всіх екранах. Використання OpenAI API дозволяє генерувати персоналізовані рекомендації та аналізи на основі накопленої інформації про пацієнта, що суттєво підвищує якість надання психотерапевтичної допомоги та оптимізує робочі процеси спеціалістів.

Висновки до третього розділу

У третьому розділі роботи було розглянуто практичні аспекти тестування програмного забезпечення на прикладі реалізації та перевірки ключових компонентів мобільного застосунку. Детальний аналіз показав, що використання модульних, інтеграційних та UI-тестів забезпечує всебічну оцінку якості розробленої системи. Модульні тести, створені для LoginViewModel та RegisterViewModel, дозволили перевірити коректність бізнес-логіки, зокрема -валідність введених користувачем даних, а також здатність системи адекватно обробляти як успішні сценарії автентифікації та реєстрації, так і помилкові випадки. Використання макетних сервісів дало змогу ізолювати логіку від зовнішніх залежностей, що підвищило точність і повторюваність результатів тестування. Тести для ChatViewModel довели коректність обробки асинхронних викликів до OpenAI-сервісу, показавши реакцію моделі як у випадку успіху, так і при виникненні помилок, що особливо важливо для забезпечення стабільності роботи інтелектуальних функцій застосунку.

Функціональні UI-тести, виконані для екрану входу, продемонстрували здатність перевіряти наявність та взаємодію з ключовими елементами інтерфейсу, моделюючи реальні дії користувача. Це підтвердило, що застосунок правильно реагує на введення даних, натискання кнопок і виклики допоміжних функцій, таких як реєстрація чи відновлення пароля. Сукупність проведених тестів

доводить, що систематичне використання різних рівнів тестування підвищує надійність, знижує ризики помилок на етапі експлуатації та формує основу для подальшої масштабованості продукту. Таким чином, впровадження комплексного підходу до тестування дозволяє не лише гарантувати відповідність функціональних можливостей заявленим вимогам, а й забезпечити високу якість, зручність та безпеку програмного забезпечення.

ВИСНОВКИ

У межах виконаної роботи було проведено комплексне дослідження та практичну реалізацію мобільного застосунку для психотерапевтів із вбудованим чат-ботом на основі штучного інтелекту. Робота поєднала аналіз сучасних тенденцій у сфері інформаційних технологій, розгляд архітектурних рішень, вибір оптимальних інструментальних засобів, проектування бази даних, реалізацію ключових компонентів програмного забезпечення та їх подальше тестування.

По-перше, у роботі обґрунтовано актуальність створення програмних засобів психологічної підтримки з інтеграцією технологій ШІ. Аналіз аналогів (Headspace, BetterHelp, Calm, Talkspace, Wysa) показав, що хоча на ринку існують успішні продукти у сфері ментального здоров'я, більшість із них має обмежений функціонал або недостатній рівень персоналізації. Запропонований застосунок відрізняється акцентом на індивідуалізованій підтримці психотерапевта, інтеграцією чат-бота для аналізу емоційного стану клієнта та локальним розгортанням моделі ШІ для підвищення безпеки даних.

По-друге, обґрунтовано вибір інструментів розробки: мова Swift та фреймворки SwiftUI і Combine забезпечують продуктивність і декларативний стиль створення UI; Firebase Auth та Firebase Realtime Database гарантують безпеку та синхронізацію даних у реальному часі. У поєднанні з Xcode як основним середовищем розробки це створює комплексну технологічну екосистему.

По-третє, запропонована архітектура поєднує шаблони MVVM, координаторів та принципи чистої архітектури, що дозволило розділити відповідальності, спростити підтримку й тестування та забезпечити модульність і масштабованість системи. Особливу увагу приділено інтеграції сервісів Firebase і OpenAI через протокол-орієнтований підхід, що підвищує гнучкість і адаптивність застосунку.

По-четверте, було розроблено структуру бази даних у Firebase, що дозволяє ефективно зберігати та обробляти дані лікарів, пацієнтів, чатів і повідомлень. Запропонована схема забезпечує швидкодію, масштабованість та узгодженість даних, що є критично важливим для мобільних додатків із високим навантаженням.

По-п'яте, значну увагу приділено тестуванню. Проведено модульні та інтеграційні тести, які підтвердили коректність роботи ключових компонентів: обробки повідомлень у ChatViewModel, редагування даних пацієнта, серіалізації/десеріалізації моделей. Тестові кейси дозволили виявити потенційні проблеми та переконатися у правильності логіки застосунку, що підвищує його надійність та стійкість до помилок.

Таким чином, результати роботи свідчать, що поставлені завдання виконані у повному обсязі. Створений застосунок має наукову новизну в частині інтеграції локально розгорнутої моделі ШІ у психотерапевтичну практику, а також практичну цінність як інструмент підвищення ефективності роботи фахівців. Реалізоване рішення відповідає сучасним технологічним стандартам, демонструє високий рівень безпеки, зручності та масштабованості й має потенціал для подальшого розвитку.

Отримані результати підтверджують доцільність інтеграції штучного інтелекту у сферу психологічної підтримки та можуть стати основою для майбутніх наукових досліджень і прикладних рішень у галузі інженерії програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Swift Apple Inc. URL: <https://developer.apple.com/swift/> (дата звернення: 30.01.2025).
2. SwiftUI Apple Inc. URL: <https://developer.apple.com/xcode/swiftui/> (дата звернення: 02.02.2025).
3. Combine Apple Inc. URL: <https://developer.apple.com/documentation/combine/> (дата звернення: 29.01.2025).
4. Firebase Documentation Firebase. URL: <https://firebase.google.com/docs> (дата звернення: 05.02.2025).
5. Xcode – The Developer Tool for Apple Platforms Apple Inc. URL: <https://developer.apple.com/xcode/> (дата звернення: 03.02.2025).
6. The Composable Architecture GitHub. URL: <https://github.com/pointfreeco/swift-composable-architecture> (дата звернення: 01.02.2025).
7. The Transformer Explainer poloclub.github.io. URL: <https://poloclub.github.io/transformer-explainer/> (дата звернення: 30.11.2025).
8. Transformer models in NLP: Everything you need to know about the architecture Netguru. URL: <https://www.netguru.com/blog/transformer-models-in-nlp> (дата звернення: 30.11.2025).
9. Transformer Architecture Toloka AI. URL: <https://toloka.ai/blog/transformer-architecture/> (дата звернення: 30.11.2025).
10. How Transformers Work DataCamp. URL: <https://www.datacamp.com/tutorial/how-transformers-work> (дата звернення: 30.11.2025).

11. Attention Mechanism in Transformer Neural Networks LearnOpenCV. URL:
<https://learnopencv.com/attention-mechanism-in-transformer-neural-networks/> (дата звернення: 30.11.2025).
12. BERT vs GPT-3 Architecture Baeldung. URL:
<https://www.baeldung.com/cs/bert-vs-gpt-3-architecture> (дата звернення: 30.11.2025).
13. GPT-3 vs BERT: LLM Comparison Exxact Corp. URL:
<https://www.exxactcorp.com/blog/deep-learning/gpt-3-vs-bert-llm-comparison> (дата звернення: 30.11.2025).
14. GPT vs BERT GeeksforGeeks. URL:
<https://www.geeksforgeeks.org/nlp/gpt-vs-bert/> (дата звернення: 30.11.2025).
15. What Is the Difference Between BERT and GPT Milvus. URL:
<https://milvus.io/ai-quick-reference/what-is-the-difference-between-bert-and-gpt> (дата звернення: 30.11.2025).
16. The new language models: how they work and what to expect from them PubMed. URL: <https://pubmed.ncbi.nlm.nih.gov/38415369/> (дата звернення: 30.11.2025).
17. A Survey of Deep Learning Models for Mental Health Triage: Applications and Challenges IEEE Xplore. URL:
<https://ieeexplore.ieee.org/document/10815727/> (дата звернення: 30.11.2025).
18. GPT-4 vs BERT Obot AI. URL:
<https://obot.ai/resources/learning-center/gpt4-vs-bert> (дата звернення: 30.11.2025).
19. Fine-Tuning LLM Obot AI. URL:
<https://obot.ai/resources/learning-center/fine-tuning-llm/> (дата звернення: 30.11.2025).

20. Fine-tuning (deep learning) Medium. URL: <https://medium.com/@sutar.trupti23/fine-tuning-deep-learning-318d5c24b796> (дата звернення: 30.11.2025).
21. How to Build Conversational AI Uptech. URL: <https://www.uptech.team/blog/how-to-build-conversational-ai> (дата звернення: 30.11.2025).
22. Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) ebetica.github.io. URL: <https://ebetica.github.io/pytorch-Deep-Learning/en/week06/06-3/> (дата звернення: 30.11.2025).
23. Long short-term memory Deeplearning. URL: <https://deeplearning.cs.cmu.edu/S23/document/readings/LSTM.pdf> (дата звернення: 30.11.2025).
24. Mental Health Text Classification GitHub. URL: <https://github.com/emirgocen03/mental-health-text-classification> (дата звернення: 30.11.2025).
25. The role of self-attention in artificial intelligence arXiv. URL: <https://arxiv.org/pdf/2504.07245.pdf> (дата звернення: 30.11.2025).
26. Attention Mechanism in Deep Learning tamu.edu. URL: <https://people.tamu.edu/~sji/classes/attn.pdf> (дата звернення: 30.11.2025).
27. Self-Attention from Scratch Sebastian Raschka. URL: <https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html> (дата звернення: 30.11.2025).
28. Enhancing Therapeutic Conversations with Sentiment Analysis in Natural Language Processing Behavioral Health News. URL: <https://behavioralhealthnews.org/enhancing-therapeutic-conversations-with-sentiment-analysis-in-natural-language-processing/> (дата звернення: 30.11.2025).

ДОДАТКИ

Код сервісу взаємодії з API OpenAI

```
import Foundation
import OpenAI

/// Protocol for OpenAI chat service
protocol OpenAIServiceProtocol {
    func send(message: [ChatMessage], patientContext: PatientModel?) async
throws -> ChatMessage
    func generatePatientInsights(patient: PatientModel, chatHistory:
[ChatMessage]) async throws -> String
    func summarizeNotes(notes: String) async throws -> String
    func suggestAppointmentTime(patient: PatientModel, existingAppointments:
[ScheduleItem]) async throws -> String
}

struct PatientInsight {
    let summary: String
    let keyPoints: [String]
    let recommendations: [String]
}

final class OpenAIService: OpenAIServiceProtocol {
    private let openAI: OpenAI
    private let model: String = "gpt-3.5-turbo"

    // TODO: Never hardcode API keys in production. Use secure storage or
remote config.
```

```

init(apiKey: String = "sk-proj- _") {
    self.openAI = OpenAI(apiToken: apiKey)
}

func send(message: [ChatMessage], patientContext: PatientModel? = nil)
async throws -> ChatMessage {
    var messages: [ChatQuery.ChatCompletionMessageParam] = message.map
    { msg in
        let role = ChatQuery.ChatCompletionMessageParam.Role(rawValue:
msg.role.rawValue) ?? .user
        return .init(role: role, content: msg.content)!
    }

    var systemPrompt = ""

        Ти — віртуальний асистент психотерапевта.
        Відповідай лише на питання з психології та психотерапії.
        Якщо питання не стосується цих тем — відмовся ввічливо.
        Завжди відповідай мовою користувача (українською чи
англійською).

        У кінці відповіді додавай дисклеймер:
        "⚠ Цей текст має інформаційний характер і не замінює
консультацію з фахівцем."
        ""

    if let patient = patientContext {
        systemPrompt += ""

        Контекст пацієнта:
        - Ім'я: \"(patient.fullName)

```

```
- Вік: \(patient.age) років
- Приймає ліки: \(patient.takesMedecine ? "Так" : "Ні")
- Нотатки: \(patient.notes.isEmpty ? "Немає нотаток" : patient.notes)
```

```
""
```

```
}
```

```
messages.append(.init(role: .system, content: systemPrompt)!)
```

```
let query = ChatQuery(
```

```
    messages: messages,
```

```
    model: model
```

```
)
```

```
let result = try await openAI.chats(query: query)
```

```
let responseText = result.choices.first?.message.content ?? ""
```

```
return ChatMessage(role: .assistant, content: responseText)
```

```
}
```

```
func generatePatientInsights(patient: PatientModel, chatHistory:
[ChatMessage]) async throws -> String {
```

```
    let patientInfo = ""
```

```
    Пацієнт: \(patient.fullName)
```

```
    Вік: \(patient.age) років
```

```
    Приймає ліки: \(patient.takesMedecine ? "Так" : "Ні")
```

```
    Нотатки: \(patient.notes.isEmpty ? "Немає" : patient.notes)
```

```
    Кількість сеансів: \(patient.chats.count)
```

```
""
```

```
let recentChats = chatHistory.suffix(10).map { "\($0.role.rawValue):  
\\($0.content)" }.joined(separator: "\\n")
```

```
let prompt = ""
```

Проаналізуй інформацію про пацієнта та надай короткий аналіз (2-3 речення) для психотерапевта.

Зосередься на ключових моментах та можливих напрямках роботи.

```
\\(patientInfo)
```

Останні повідомлення:

```
\\(recentChats.isEmpty ? "Немає повідомлень" : recentChats)
```

Надай короткий, професійний аналіз українською мовою.

```
""
```

```
let query = ChatQuery(  
    messages: [  
        .init(role: .system, content: "Ти — аналітичний асистент  
психотерапевта. Надавай короткі, професійні аналізи.")!,  
        .init(role: .user, content: prompt)!  
    ],  
    model: model  
)
```

```
let result = try await openAI.chats(query: query)  
return result.choices.first?.message.content ?? "Не вдалося згенерувати  
аналіз."  
}
```

```

func summarizeNotes(notes: String) async throws -> String {
    guard !notes.isEmpty else { return "Немає нотаток для резюме." }

    let prompt = """
        Створи коротке резюме (до 3 речень) наступних нотаток
психотерапевта:

        \notes)

        Резюме українською мовою:
        """

    let query = ChatQuery(
        messages: [
            .init(role: .system, content: "Ти — асистент, який створює короткі
резюме нотаток.")!,
            .init(role: .user, content: prompt)!
        ],
        model: model
    )

    let result = try await openAI.chats(query: query)
    return result.choices.first?.message.content ?? notes
}

func suggestAppointmentTime(patient: PatientModel, existingAppointments:
[ScheduleItem]) async throws -> String {
    let patientInfo = "Пацієнт: \patient.fullName), вік: \patient.age)"

```

```
let upcomingCount = existingAppointments.filter { $0.patient.id ==  
patient.id && $0.appointmentDate > Date() }.count
```

```
let prompt = ""
```

На основі інформації про пацієнта, надай рекомендацію щодо частоти наступних сеансів.

```
\(patientInfo)
```

```
Кількість майбутніх сеансів: \(upcomingCount)
```

Надай коротку рекомендацію (1-2 речення) українською мовою про оптимальну частоту сеансів.

```
""
```

```
let query = ChatQuery(  
  messages: [  
    .init(role: .system, content: "Ти — асистент, який дає рекомендації  
щодо частоти психотерапевтичних сеансів.")!,  
    .init(role: .user, content: prompt)!,  
  ],  
  model: model  
)
```

```
let result = try await openAI.chats(query: query)  
return result.choices.first?.message.content ?? "Рекомендується регулярні  
сеанси."  
}  
}
```

```
extension PatientModel {  
    var age: Int {  
        Calendar.current.dateComponents([.year], from: dateOfBirth, to:  
Date()).year ?? 0  
    }  
}
```