Міністерство освіти і науки України
Галицький фаховий коледж імені В’ячеслава Чорновола

Циклова комісія інформатики та
комп’ютерних дисциплін

МІЖДИСЦИПЛІНАРНИЙ КУРСОВИЙ ПРОЄКТ

на тему:

«ШІ-асистент для психологічної підтримки»

Дата захисту
«____»_____ 2024 р.

Оцінка: _____

Члени комісії:
_____ Марина ГЛИНСЬКА
_____ Наталя СТЕФУРАК
_____ Ольга ПОСВЯТОВСЬКА

Виконала
студентка групи КН-31
Дарина ГЛИНСЬКА

Керівник проєкту
Степан ІВАСЬЄВ

Тернопіль – 2026

Міністерство освіти і науки України

Галицький фаховий коледж імені В’ячеслава Чорновола

Циклова комісія інформатики

та комп’ютерних дисциплін

З А В Д А Н Н Я

на курсовий проєкт

Студентці Дарині Глинській групи КН-31

1. Тема проєкту: «ШІ-асистент для психологічної підтримки»

2. Термін здачі студентом проєкту: «____»_____ 2026 р.

3. Вихідні дані до курсового проєкту: результати аналізу предметної області психологічної підтримки засобами штучного інтелекту, огляд існуючих хмарних та локальних рішень, технічна документація бібліотек FastAPI, LangChain, ChromaDB та інструменту Ollama.

4. Перелік питань, які підлягають розробці в курсовому проєкті: аналіз предметної області та постановка завдання; проєктування інформаційної системи (структурна схема, алгоритми, UML-діаграма класів, структура векторної бази даних, інтерфейс); реалізація інформаційної системи (бекенд, фронтенд, наповнення бази знань); тестування та налагодження системи.

5. Перелік графічного матеріалу: структурна схема системи, діаграма варіантів використання, блок-схема алгоритму RAG, UML-діаграма класів, знімки екрана інтерфейсу системи.

Дата видачі завдання: «____»_____ 2026 р.

Керівник проєкту _____

Завдання прийняла до виконання студентка_____

ЗМІСТ

ВСТУП 4
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ ... 8
1.1 Загальна характеристика предметної області 8
1.2 Огляд існуючих рішень 9
1.3 Вибір технологій 12
1.4 Вимоги до системи 14
2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ 18
2.1 Загальні принципи проєктування 18
2.2 Побудова моделі інформаційної системи 18
2.3 Проєктування бази даних 24
2.4 Проєктування інтерфейсу 27
2.5 Проєктування класів 28
3 РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ 32
3.1. Реалізація бекенду 32
3.2. Реалізація фронтенду 38
3.3. Інтеграція з LLM 40
4 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ СИСТЕМИ 42
4.1. План тестування 42
4.2. Сценарії тестування та результати 43
4.3. Виявлені проблеми та їх усунення 48
ВИСНОВКИ 51
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 53
ДОДАТКИ 55

ВСТУП

Психічне здоров'я населення України зазнало значного навантаження протягом останніх років через сукупність кризових факторів – повномасштабне військове вторгнення, пов'язані з ним втрати, переміщення, економічну невизначеність та хронічний стрес. За даними Міністерства охорони здоров'я України, тривога та інші емоційні труднощі стали типовою реакцією значної частини громадян на ситуацію в країні [1]. Водночас доступ до кваліфікованої психологічної допомоги залишається обмеженим: кількість практикуючих психологів і психотерапевтів не покриває повного попиту, послуги платної консультації не завжди доступні фінансово, а звернення по допомогу часто стримується стигматизацією або браком інформації про доступні ресурси.

Ця ситуація не є унікальною для України: за оцінками Всесвітньої організації охорони здоров'я, у глобальному масштабі спостерігається суттєвий розрив між поширеністю проблем психічного здоров'я та доступністю кваліфікованої допомоги, особливо в регіонах з обмеженою кількістю фахівців на душу населення. У відповідь на цей розрив у світі активно розвивається напрям цифрового психічного здоров'я (digital mental health) – застосування інформаційних технологій для розширення доступу до підтримки психічного здоров'я: від простих застосунків-щоденників настрою до складніших діалогових систем на основі штучного інтелекту.

У цих умовах інформаційні технології, зокрема технології штучного інтелекту на основі великих мовних моделей (Large Language Models, LLM), відкривають новий напрям для часткового вирішення проблеми доступності первинної психологічної підтримки. ШІ-асистенти не замінюють кваліфікованого фахівця, але можуть виконувати функцію першої лінії підтримки: надавати інформацію про техніки самодопомоги, вислуховувати користувача в режимі активного слухання, а в кризових ситуаціях – спрямовувати до відповідних служб допомоги. Така роль узгоджується з моделлю «поетапної допомоги» (stepped care), поширеною в системах

охорони психічного здоров'я багатьох країн: менш інтенсивні, легкодоступні форми підтримки (зокрема цифрові інструменти самодопомоги) функціонують як перший, доступний усім рівень, тоді як інтенсивніша, ресурсомісткіша професійна допомога залишається доступною для випадків, що цього потребують.

Окремою і критично важливою проблемою при розробці таких систем є питання приватності. Психологічні дані є надзвичайно чутливою категорією особистої інформації, а використання хмарних рішень (ChatGPT, Claude та подібні) передбачає передачу даних користувача на сторонні сервери, що не завжди прийнятно для теми психологічної підтримки. Це обумовлює актуальність розробки локальних рішень, що працюють повністю на обладнанні користувача, без передачі даних третім сторонам. Крім приватності, питання локального розгортання має й практичний вимір доступності: для частини потенційних користувачів (наприклад, мешканців регіонів зі слабким чи нестабільним інтернет-з'єднанням, що особливо актуально в умовах війни) система, яка не вимагає постійного підключення до мережі, має суттєву практичну перевагу порівняно з хмарними альтернативами.

Не менш важливим є технічний аспект розвитку галузі: останні кілька років продемонстрували стрімкий прогрес у якості відкритих (open-weight) мовних моделей, доступних для безкоштовного локального запуску, та паралельний розвиток зручних інструментів для їх розгортання (Ollama, LM Studio). Це зробило раніше суто дослідницьку задачу («запустити потужну мовну модель на власному комп'ютері») практично доступною навіть для розробників без глибокої спеціалізації в машинному навчанні – інженерна задача звелась здебільшого до коректної інтеграції готових компонентів (мовної моделі, векторної бази даних, фреймворку для API) у працюючу систему, що й демонструє цей курсовий проєкт.

Актуальність обраної теми визначається поєднанням кількох факторів: зростаючою потребою населення України в доступній психологічній

підтримці в умовах війни та повоєнного відновлення; технологічною можливістю розгортання потужних мовних моделей локально завдяки розвитку відеокарт споживчого рівня та інструментів типу Ollama; підвищеною чутливістю теми психічного здоров'я до питань приватності даних, що робить локальні рішення кращою альтернативою хмарним сервісам; а також загальносвітовою тенденцією розвитку цифрових інструментів підтримки психічного здоров'я як доповнення (не заміни) традиційної професійної допомоги.

Мета роботи – розробити та реалізувати програмну систему ШІ-асистента для психологічної підтримки, що працює на основі локальної мовної моделі, технології доповненої генерації (RAG) для використання спеціалізованої бази психологічних знань та механізму детекції кризових ситуацій з автоматичним наданням контактів служб допомоги.

Завдання курсового проєкту:

1. Проаналізувати предметну область психологічної підтримки засобами штучного інтелекту, дослідити існуючі рішення (хмарні та локальні) і обґрунтувати вибір технологічного стеку.
2. Спроєктувати архітектуру системи: визначити основні компоненти, алгоритми роботи (зокрема алгоритм RAG та алгоритм детекції кризових ситуацій), структуру класів та структуру векторної бази даних.
3. Реалізувати програмну систему: бекенд на FastAPI з інтеграцією Ollama та ChromaDB, фронтенд на HTML/CSS/JavaScript, скрипт автоматизованого наповнення бази знань.
4. Наповнити базу знань тематичними матеріалами з психологічної підтримки (теоретичні матеріали, практичні техніки, кризові протоколи), підготованими з урахуванням вимог до простоти мови та практичної застосовності.
5. Провести тестування реалізованої системи за розробленим планом тестування, виявити та усунути недоліки.

6. Оформити пояснювальну записку до курсового проєкту відповідно до встановлених вимог.

Об'єкт дослідження – процес надання первинної психологічної підтримки людині, що звертається з емоційними труднощами, тривогою чи стресовими переживаннями.

Предмет дослідження – методи та програмні засоби реалізації діалогового ШІ-асистента на основі локальної мовної моделі з функціями retrieval-augmented generation та детекції кризових ситуацій.

Пояснювальна записка складається зі вступу, чотирьох розділів основної частини, висновків, переліку джерел посилання та додатків. Перший розділ присвячено аналізу предметної області та обґрунтуванню вибору технологій. Другий розділ описує проєктні рішення – архітектуру, алгоритми, структуру класів та бази даних. Третій розділ деталізує програмну реалізацію з наведенням ключових фрагментів коду. Четвертий розділ присвячено процесу та результатам тестування системи, включно з детальним описом виявлених і усунутих недоліків.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Загальна характеристика предметної області

Предметна область психологічної підтримки засобами штучного інтелекту охоплює широкий спектр потенційних звернень користувачів – від загального бажання «поговорити» про свій день до конкретних запитів щодо технік подолання стресу чи тривоги, і, у найбільш чутливому випадку, до висловлювань, що сигналізують про психологічну кризу. Для коректного проєктування системи важливо чітко розрізняти ці категорії звернень, оскільки кожна вимагає різного типу реагування.

Категорія 1 – загальний діалог. Привітання, питання про можливості системи, нейтральні розмови без специфічного психологічного запиту. Система повинна підтримувати природний, дружній діалог, що формує враження доступного, неформального помічника.

Категорія 2 – запити щодо технік самодопомоги. Конкретні питання про способи подолання стресу, тривоги, безсоння, прокрастинації тощо. Це основний цільовий тип взаємодії, де система максимально корисна – надання практичної, застосовної інформації, підкріпленої спеціалізованою базою знань через механізм RAG.

Категорія 3 – емоційна підтримка (активне слухання). Користувач описує складну емоційну ситуацію без явного запиту конкретної техніки, потребуючи насамперед бути почутим, отримати підтвердження своїх почуттів (валідацію). Системний промпт асистента явно орієнтує модель на застосування принципів активного слухання в такому випадку, а не на негайне «вирішення проблеми».

Категорія 4 – кризові звернення. Висловлювання, що прямо чи непрямо вказують на суїцидальні думки, наміри самоушкодження чи глибоку безнадію. Це найбільш чутлива категорія, що вимагає окремого, пріоритетного механізму реагування, оскільки звичайна генерація відповіді

LLM не гарантує адекватної, безпечної реакції на такі звернення без додаткового, детермінованого рівня контролю.

Розуміння цих чотирьох категорій безпосередньо визначило архітектурні рішення проєкту: необхідність окремого, надійного механізму детекції категорії 4 (що не залежить від «розсудливості» LLM), необхідність спеціалізованої бази знань для якісного обслуговування категорії 2, та формулювання системного промпту з явним заохоченням технік активного слухання для категорії 3.

1.2 Огляд існуючих рішень

1.2.1 Хмарні рішення

Найбільш відомими прикладами діалогових ШІ-асистентів загального призначення є ChatGPT (компанія OpenAI), Claude (компанія Anthropic) та Gemini (компанія Google). Ці системи демонструють високу якість природномовного діалогу та здатні надавати психологічну за тематикою підтримку (роз'яснення техник, активне слухання), проте мають ряд обмежень для застосування саме в темі психологічної підтримки:

– Передача даних на сторонні сервери. Будь-яке повідомлення користувача обробляється на серверах компанії-розробника, що для надзвичайно чутливої теми психічного здоров'я є істотним недоліком з погляду приватності, навіть за наявності політик конфіденційності постачальника.

– Платний доступ до повної функціональності. Більшість потужних моделей пропонуються за підпискою, що обмежує доступність для частини користувачів.

– Відсутність спеціалізації. Універсальні асистенти не мають вбудованого механізму детекції кризових психологічних станів і спеціалізованої бази знань з психології, що залишає відповідальність за безпечність відповідей повністю на загальних механізмах безпеки моделі.

– Залежність від інтернет-з'єднання. Хмарні рішення не функціонують без стабільного доступу до мережі, що може бути проблемою в умовах нестабільної інфраструктури.

1.2.2 Локальні рішення

Розвиток інструментів для локального запуску великих мовних моделей – Ollama, LM Studio, GPT4All та подібних – зробив можливим розгортання якісних LLM на персональному комп'ютері з відеокартою споживчого рівня. Перевагами такого підходу є:

– Повна приватність даних. Жодне повідомлення користувача не залишає його комп'ютер, що критично важливо для теми психологічного здоров'я.

– Відсутність абонентської плати. Після початкового налаштування система працює без додаткових витрат (окрім електроенергії та амортизації обладнання).

– Можливість глибокої кастомізації. Локальне розгортання дозволяє повністю контролювати системний промпт, підключати власну базу знань, налаштовувати специфічну поведінку (зокрема детектор кризових ситуацій).

– Незалежність від доступності інтернету та політик постачальника. Система продовжує працювати навіть за відсутності мережевого з'єднання.

– Серед недоліків локальних рішень слід зазначити вищі вимоги до обчислювальних ресурсів (необхідність потужної відеокарти з достатнім обсягом відеопам'яті) та, як правило, дещо нижчу якість відповідей порівняно з найпотужнішими хмарними моделями.

Таблиця 1.1 – Порівняння хмарних та локальних рішень для застосунку психологічної підтримки

Критерій	Хмарні рішення (ChatGPT, Claude)	Локальні рішення (Ollama + LLM)
Приватність даних	Дані обробляються на серверах постачальника	Дані не залишають комп'ютер користувача
Вартість використання	Зазвичай підписка або оплата за токени	Безкоштовно після початкового налаштування
Якість генерації	Як правило, вища (найпотужніші моделі)	Прийнятна для моделей 7-14B параметрів
Вимоги до обладнання	Мінімальні (потрібен лише браузер)	Відеокарта з 8-16+ ГБ відеопам'яті
Доступність offline	Не працює без інтернету	Працює повністю offline
Спеціалізація на психології	Загального призначення	Можливість підключення власної бази знань
Контроль над поведінкою системи	Обмежений системний промпт	Повний контроль (промпт, логіка детекції криз)

Як видно з таблиці 1.1, основний компромісний фактор при виборі між підходами – це приватність і контроль над системою проти максимальної якості генерації та простоти доступу. Для теми психологічної підтримки, де довіра користувача та захист чутливих даних мають визначальне значення, цей компроміс схиляється на користь локального розгортання, навіть з огляду на певні втрати в обчислювальній зручності та (потенційно) якості відповідей порівняно з найновішими хмарними моделями.

1.2.3 Обґрунтування вибору локальної моделі

Для теми психологічної підтримки питання приватності даних має визначальне значення: інформація про емоційний стан, особисті переживання, можливі кризові думки є надзвичайно чутливою категорією персональних даних. Передача такої інформації на сторонні сервери, навіть за наявності формальних гарантій конфіденційності постачальника, створює додатковий психологічний бар'єр для звернення за підтримкою та потенційний ризик у разі витоку даних.

Виходячи з цього, для реалізації системи PsychHelper обрано підхід повністю локального розгортання на основі Ollama – інструменту, що дозволяє запускати квантизовані версії великих мовних моделей на локальному обладнанні з мінімальними вимогами до налаштування. Це рішення узгоджується з принципом «privacy by design» – закладання захисту приватності в саму архітектуру системи, а не як додаткову функцію.

Додатковим аргументом на користь локального підходу є специфіка цільової аудиторії системи: люди, що звертаються по психологічну підтримку, часто перебувають у стані підвищеної вразливості та можуть мати посилені (порівняно зі звичайним користувачем технологій) занепокоєння щодо конфіденційності своїх переживань. Локальне розгортання усуває цей бар'єр на технічному рівні: розробник системи в принципі не має технічної можливості отримати доступ до розмов користувача, оскільки дані ніколи не передаються на сервери розробника.

1.3 Вибір технологій

1.3.1 LLM модель: Qwen 2.5 14B

Для генерації відповідей обрано модель Qwen 2.5. Вибір моделі обумовлений такими факторами:

– Якість роботи з українською мовою. Серед моделей, доступних для локального запуску з прийнятними системними вимогами, Qwen 2.5

демонструє стабільно високу якість генерації українського тексту порівняно з низкою альтернатив, орієнтованих переважно на англомовний контент.

– Баланс якості та продуктивності. Версія з 14 млрд параметрів у квантизованому форматі (GGUF) забезпечує прийнятну швидкість генерації на відеокартах із 12-16 ГБ відеопам'яті (наприклад, NVIDIA RTX 3060/4060 Ti), залишаючись при цьому якіснішою за менші моделі (7B та менше).

– Сумісність з Ollama. Модель офіційно підтримується реєстром Ollama, що значно спрощує встановлення, оновлення та керування версіями.

1.3.2 Embedding-модель: nomic-embed-text

Для перетворення тексту в числові вектори (необхідні для семантичного пошуку в базі знань) використовується модель nomic-embed-text, також доступна через реєстр Ollama. Ця модель спеціалізована саме на завданні створення якісних текстових ембедингів і є стандартним вибором у пайплайнах RAG на базі Ollama.

1.3.3 Бекенд: FastAPI (Python)

FastAPI обрано як фреймворк для реалізації серверної частини системи з огляду на:

– асинхронну обробку запитів – сервер може приймати кілька повідомлень одночасно, не "зависаючи" в очікуванні відповіді від Ollama чи бази знань;

– вбудовану валідацію даних через Pydantic – якщо запит прийшов у неправильному форматі, FastAPI сам поверне зрозумілу помилку ще до того, як ці дані потраплять у логіку програми;

– широке використання в Python-екосистемі для ML/AI-застосунків, що забезпечує хорошу сумісність з бібліотекою LangChain, яка відповідає за взаємодію з мовною моделлю та логіку RAG.

1.3.4 Фронтенд: HTML/CSS/JavaScript

Для клієнтської частини обрано підхід без використання фреймворків (HTML, CSS, JavaScript), що обумовлено простотою розгортання – інтерфейс відкривається безпосередньо у браузері без потреби в збірці (build-процесі) чи встановленні додаткових залежностей, достатністю функціональних можливостей для поставленого завдання, а також зниженням порогу складності для подальшої підтримки та розширення проєкту.

1.3.5 База даних: ChromaDB

Для реалізації механізму пошуку інформації використано векторну базу даних ChromaDB. Вона зберігає оброблені фрагменти бази знань і дає змогу швидко знаходити ті з них, які найбільше відповідають запиту користувача. Знайдені фрагменти передаються мовній моделі для формування відповіді.

Вибір ChromaDB обумовлений кількома перевагами. По-перше, вона працює локально і не потребує запуску окремого сервера, що спрощує встановлення та використання системи. По-друге, ChromaDB легко інтегрується з бібліотекою LangChain, яка використовується для реалізації механізму RAG. Крім того, база даних зберігається між перезапусками системи, тому її не потрібно створювати повторно після кожного запуску. Продуктивності ChromaDB цілком достатньо для обробки обсягу даних, характерного для цього проєкту.

1.4 Вимоги до системи

1.4.1. Користувачі системи та їхні потреби

Перед формулюванням конкретних вимог до системи доцільно визначити основні категорії користувачів та їхні потреби, оскільки саме вони визначають пріоритетність функціональних вимог.

Основний користувач – людина, що звертається по психологічну підтримку. Це може бути особа, яка переживає підвищений рівень стресу чи тривоги, стикається з труднощами повсякденного емоційного регулювання,

шукає інформацію про техніки самодопомоги, або (у найбільш критичному випадку) перебуває в стані психологічної кризи. Потреби цієї категорії користувачів: швидкий, доступний у будь-який час доби канал для висловлення переживань; конфіденційність взаємодії; отримання практичної, застосовної інформації (а не лише загальних слів підтримки); у кризовому випадку – негайне скерування до кваліфікованої допомоги.

Вторинний користувач – адміністратор/розробник системи. У контексті курсового проєкту ця роль може виконуватися самим студентом під час розробки та тестування, а в гіпотетичному реальному застосуванні – технічно підготовленою особою, відповідальною за розгортання системи на конкретному обладнанні, підтримання актуальності бази знань та моніторинг коректності роботи.

Важливо зазначити: система спроєктована як інструмент первинної, допоміжної підтримки, а не як заміна професійної психологічної чи психіатричної допомоги. Це обмеження прямо комунікується користувачу через системний промпт LLM та постійно видимий дисклеймер у користувацькому інтерфейсі.

1.4.2. Функціональні вимоги

Система повинна забезпечувати виконання таких функцій:

1. Прийом текстового повідомлення від користувача через вебінтерфейс чату.
2. Перевірку кожного вхідного повідомлення на наявність маркерів кризової ситуації (суїцидальні думки, наміри самоушкодження).
3. У разі виявлення кризового маркера – негайне повернення контактів служб психологічної підтримки замість звичної генерації відповіді LLM.
4. У разі відсутності кризового маркера – генерацію відповіді через локальну мовну модель, з використанням релевантних фрагментів бази знань (якщо база знань підключена) за принципом RAG.

5. Відображення відповіді асистента користувачу в реальному часі через вебінтерфейс, з індикацією набору тексту під час очікування відповіді.

6. Перевірку статусу системи (доступність LLM, бази знань, детектора криз) через спеціальний діагностичний endpoint.

7. Можливість наповнення бази знань новими документами (PDF, DOCX, TXT) через окремий скрипт автоматизованої векторизації.

1.4.3. Нефункціональні вимоги

– Приватність: жодні дані користувача не повинні передаватися за межі локального середовища виконання (увесь стек технологій – Ollama, FastAPI, ChromaDB – працює локально).

– Швидкість відповіді: середній час генерації відповіді на типове повідомлення не повинен перевищувати 10-15 секунд за умови використання рекомендованого обладнання.

– Надійність детекції криз: механізм детекції кризових ситуацій повинен мати пріоритет над звичною генерацією відповіді й активуватися до звернення до LLM.

– Простота розгортання: встановлення системи має бути можливим для користувача без глибоких знань адміністрування серверів.

– Мовна відповідність: система повинна послідовно відповідати українською мовою незалежно від мови запиту.

– Розширюваність бази знань: додавання нових тематичних матеріалів повинно бути можливим без зміни програмного коду.

1.4.4. Обґрунтування пріоритетності вимог

Серед перелічених вимог особливої уваги заслуговує співвідношення між вимогою швидкості відповіді та вимогою надійності детекції криз. У типовій програмній системі ці вимоги розглядалися б рівноцінно в межах загальної продуктивності, однак для системи психологічної підтримки вимога надійності детекції кризи має абсолютний пріоритет: затримка в кілька секунд при генерації звичайної відповіді є прийнятним компромісом,

тоді як пропуск чи затримка реагування на кризовий сигнал може мати критичні наслідки для безпеки користувача. Це архітектурне рішення прямо відображено в реалізації: перевірка на кризу виконується синхронно, на самому початку обробки запиту, незалежно від того, скільки часу могла би зайняти подальша генерація відповіді LLM.

Аналогічно, вимога приватності даних розглядається як квазі-функціональна, а не суто нефункціональна вимога – тобто вона впливає не лише на якісні характеристики системи, а й безпосередньо визначає функціональний склад і архітектуру системи (зокрема, виключає з розгляду весь клас хмарних рішень, що в інших контекстах міг би бути прийнятним технологічним вибором).

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Загальні принципи проєктування

Проєктування системи PsychHelper здійснювалося з опорою на кілька наскрізних принципів, що визначили подальші архітектурні рішення на всіх рівня:

– Принцип розділення відповідальності (separation of concerns). Кожен компонент системи відповідає за одну чітко визначену функцію: фронтенд – за представлення даних користувачу, бекенд-контролер (main.py) – за маршрутизацію запитів та координацію виклику бізнес-логіки, сервісні класи (services.py) – за конкретну бізнес-логіку (детекцію кризи, генерацію відповіді LLM, пошук у базі знань), Pydantic-моделі (models.py) – за структуризацію та валідацію даних.

– Принцип «безпека понад усе» (safety first). Механізм детекції кризових ситуацій спроєктовано як пріоритетний, незалежний від інших компонентів системи елемент архітектури, що активується першим у потоці обробки кожного повідомлення.

– Принцип graceful degradation. Часткова недоступність чи несправність окремого компонента (відсутність бази знань, недоступність Ollama) не повинна призводити до повної відмови системи там, де можлива часткова, безпечна функціональність.

– Принцип простоти розгортання. Зважаючи на цільову аудиторію проєкту, архітектурні рішення свідомо орієнтувалися на мінімізацію кількості зовнішніх залежностей та складності налаштування.

2.2 Побудова моделі інформаційної системи

2.2.1 Загальна структурна схема

Система PsychHelper побудована за класичною клієнт-серверною архітектурою з трьома логічними рівнями: рівень представлення (фронтенд), рівень бізнес-логіки (бекенд-API) та рівень даних, який у цьому проєкті представлений двома типами зовнішніх сервісів – мовною моделлю (через

Ollama) та векторною базою знань (ChromaDB). Структурна схема системи наведена на рисунку 2.1.

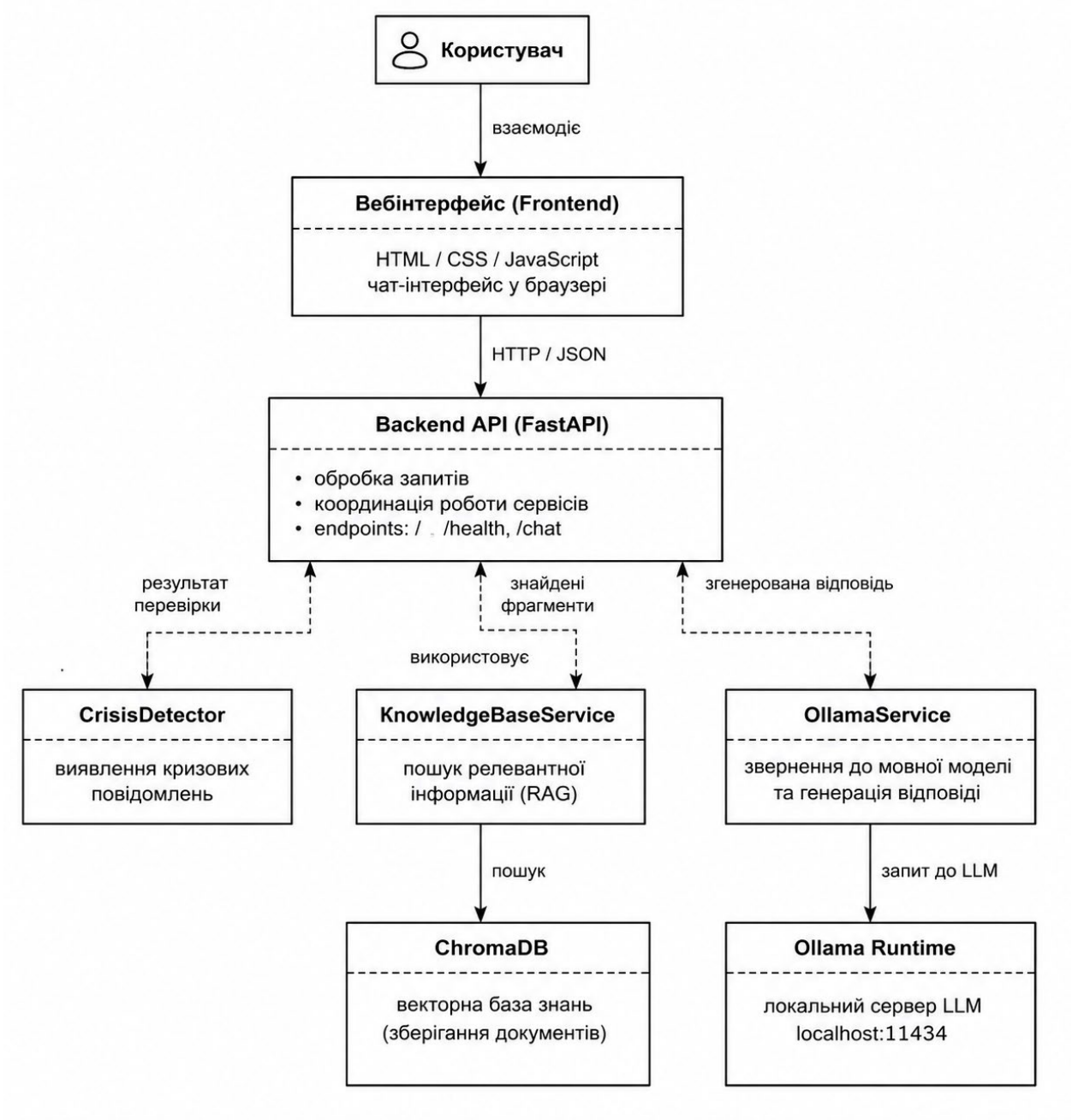


Рисунок 2.1 – Структурна схема системи PsychHelper

Користувач взаємодіє з системою через веббраузер, у якому відкрито файл index.html – клієнтську частину застосунку. Фронтенд надсилає HTTP-запити (у форматі JSON) до бекенд-API, реалізованого засобами FastAPI. Бекенд виконує всю основну логіку обробки повідомлення: перевірку на кризові маркери, звернення до мовної моделі через Ollama (за прямим HTTP-

протоколом на порт 11434), і за наявності підключеної бази знань – додатковий запит до ChromaDB для отримання релевантного контексту.

2.2.2 Загальний алгоритм роботи системи

Загальний алгоритм обробки одного повідомлення користувача складається з таких кроків:

1. Користувач вводить текст у поле введення фронтенд-інтерфейсу та натискає кнопку «Надіслати» (або клавішу Enter).
2. Фронтенд (script.js) формує JSON-об'єкт із текстом повідомлення та ідентифікатором сесії, надсилає POST-запит на endpoint /chat бекенд-API.
3. Бекенд (main.py) отримує запит, передає текст повідомлення в CrisisDetector.detect().
4. Якщо детектор виявляє кризовий маркер – повертається заздалегідь підготовлена відповідь з контактами служб допомоги (поле is_crisis: true у відповіді), і подальша обробка через LLM не виконується.
5. Якщо кризового маркера не виявлено – перевіряється наявність активованого RAG-ланцюга (qa_chain): якщо ланцюг доступний, виконується пошук релевантних фрагментів у ChromaDB та запит передається до LLM з контекстом; якщо ланцюг недоступний, запит передається до LLM без додаткового контексту.
6. Ollama генерує відповідь на основі обраної моделі (Qwen 2.5 14B) та повертає її бекенду.
7. Бекенд формує об'єкт відповіді (ChatResponse) і повертає його фронтенд у форматі JSON.
8. Фронтенд відображає отриману відповідь у вікні чату, прибираючи індикатор набору тексту.

Важливою характеристикою описаного алгоритму є його лінійність та відсутність розгалужень з боку клієнта: вся логіка прийняття рішень (чи це криза, чи використовувати RAG) інкапсульована на боці бекенду, що відповідає принципу «тонкого клієнта» (thin client) – фронтенд відповідає

лише за відображення даних та збір вводу користувача, тоді як вся бізнес-логіка зосереджена на сервері.

2.2.2.1 Алгоритм детекції кризових ситуацій

Окремої деталізації потребує алгоритм роботи компонента CrisisDetector, оскільки від його коректності залежить безпека користувача в найбільш чутливих ситуаціях взаємодії з системою. Алгоритм складається з таких кроків:

1. Вхідний текст повідомлення нормалізується – перетворюється до нижнього регістру для забезпечення регістронезалежності пошуку.
2. Нормалізований текст послідовно перевіряється на наявність кожного з наперед визначених ключових слів-маркерів кризи методом простого пошуку підрядки.
3. Якщо жодне ключове слово не знайдено прямим пошуком, текст додатково перевіряється за набором регулярних виразів, що враховують типові граматичні варіації українських фраз, пов'язаних із суїцидальними думками чи безнадією.
4. Якщо хоча б один із методів перевірки виявляє збіг – метод повертає True, що сигналізує контролеру про необхідність переключення на кризовий сценарій реагування.
5. Якщо жодного збігу не виявлено – метод повертає False, і обробка повідомлення продовжується за звичним сценарієм.

Принциповою архітектурною рисою цього алгоритму є його розташування на самому початку обробки запиту, до будь-якого звернення до LLM чи бази знань. Це гарантує, що навіть у разі технічних проблем з мовною моделлю чи векторною базою, кризова відповідь буде сформована та надана користувачу без затримки чи відмови, оскільки CrisisDetector є повністю автономним компонентом, що не залежить від жодних зовнішніх сервісів.

2.2.3 Діаграма варіантів використання (Use Case)

Для системи PsychHelper визначено двох основних акторів: «Користувач» (особа, що звертається по психологічну підтримку) та «Адміністратор/розробник» (особа, відповідальна за розгортання, налаштування та наповнення бази знань системи). Діаграма варіантів використання наведена на рисунку 2.2.

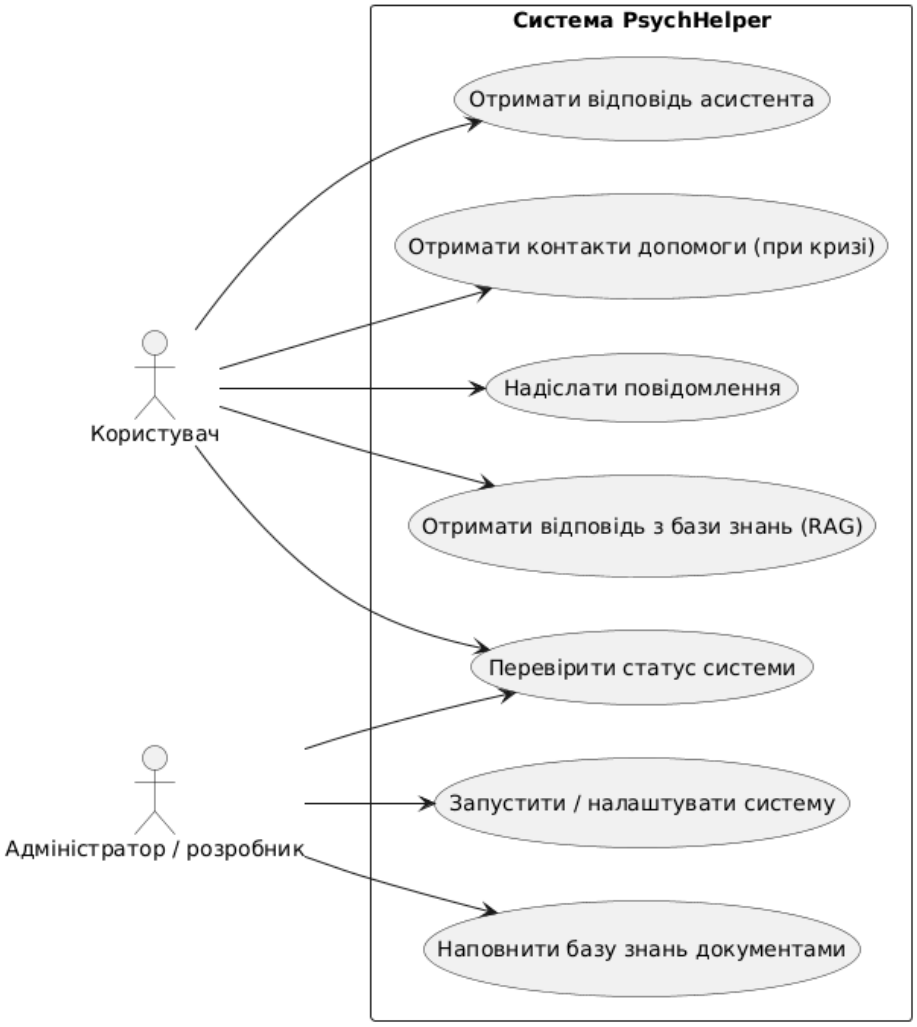


Рисунок 2.2 – Діаграма варіантів використання

2.2.4 Алгоритм RAG

Технологія доповненої генерації RAG (Retrieval-Augmented Generation) є ключовим алгоритмічним компонентом системи, що дозволяє поєднати загальні мовні здатності LLM зі спеціалізованими, контрольованими

розробником знаннями з предметної області психологічної підтримки. Без використання RAG мовна модель відповідала б виключно на основі знань, отриманих під час її тренування, тоді як з RAG модель отримує точний, релевантний фактичний контекст безпосередньо перед генерацією відповіді.

Детальна блок-схема алгоритму RAG наведена на рисунку 2.3.

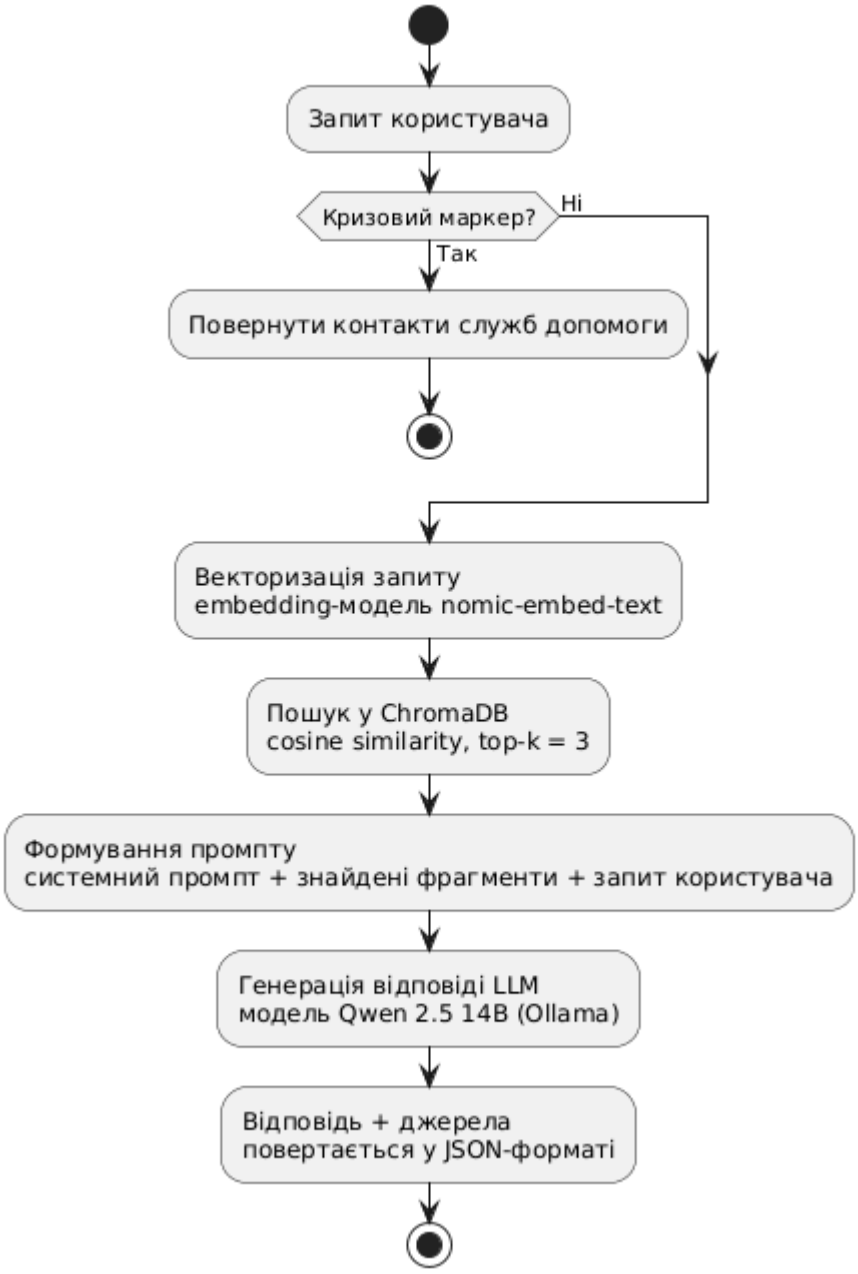


Рисунок 2.3 – Блок-схема алгоритму RAG

Алгоритм складається з таких етапів:

1. Якщо виявлено кризовий маркер, подальші кроки RAG не виконуються.

2. Векторизація запиту користувача. Текст запиту передається в embedding-модель (nomic-embed-text через Ollama), яка перетворює його у числовий вектор.

3. Пошук найбільш релевантних фрагментів у ChromaDB. Векторизований запит порівнюється з усіма векторами колекції psychology_kb за метрикою косинусної подібності. Повертаються k=3 фрагменти з найвищим показником подібності.

4. Формування розширеного промпту. Знайдені фрагменти тексту об'єднуються із системним промптом та оригінальним запитом користувача у єдиний, розширений промпт.

5. Генерація відповіді LLM. Мовна модель Qwen 2.5 14B генерує відповідь, маючи у своєму контексті як загальні інструкції системного промпту, так і конкретний, релевантний фактичний матеріал із бази знань.

6. Формування фінальної відповіді з атрибуцією джерел. Система повертає метадані про використані документи бази знань (поле sources).

Алгоритмічна складність пошуку в ChromaDB залежить від обраного методу індексації (за замовчуванням використовується наближений пошук найближчих сусідів – Approximate Nearest Neighbors, що забезпечує приблизно логарифмічну складність відносно кількості векторів у колекції).

2.3 Проєктування бази даних

2.3.1. Концептуальна модель

Специфіка системи на основі RAG передбачає роботу з векторним (NoSQL-подібним) сховищем ChromaDB, призначеним для зберігання та пошуку за семантичною подібністю векторних представлень тексту. Це принципова архітектурна особливість, обумовлена природою задачі: для пошуку релевантного психологічного контенту за змістом запиту (а не за

точним збігом ключових слів) необхідний саме семантичний, векторний пошук.

Концептуально одиницею зберігання у ChromaDB є колекція (collection) – у цьому проєкті використовується єдина колекція psychology_kb. Кожен запис колекції складається з трьох компонентів:

– id – унікальний ідентифікатор фрагмента (генерується автоматично);

– embedding – числовий вектор, що є результатом обробки тексту фрагмента embedding-моделлю nomic-embed-text;

– document + metadata – оригінальний текст фрагмента та супутні метадані (шлях до файлу-джерела, номер сторінки для PDF-документів).

2.3.2 Структура даних (словник даних)

Таблиця 2.1 – Структура запису векторної колекції psychology_kb

Поле	Тип даних	Опис
id	string (UUID)	Унікальний ідентифікатор фрагмента, генерується автоматично
embedding	vector(float)	Числове векторне представлення тексту фрагмента (модель nomic-embed-text)
document	text	Текстовий зміст фрагмента (chunk) після розбиття документа
metadata.source	string	Шлях до файлу-джерела (напр. knowledge_base/theory/stress_management.txt)
metadata.page	integer / N/A	Номер сторінки джерела (для PDF; для TXT/DOCX – N/A)

2.3.3 Процес наповнення бази (ETL-процес)

Наповнення векторної бази здійснюється скриптом create_knowledge_base.py за таким алгоритмом:

1. Завантаження (Extract). Скрипт рекурсивно обходить папку knowledge_base/ (включно з підпапками theory/, techniques/, crisis/) і завантажує всі файли підтримуваних форматів (PDF – через PyPDFLoader, DOCX – через Docx2txtLoader, TXT – через TextLoader) у вигляді об'єктів Document бібліотеки LangChain.

2. Розбиття на фрагменти (Transform). Кожен документ розбивається на менші фрагменти (chunks) за допомогою RecursiveCharacterTextSplitter з параметрами: розмір фрагмента – 1000 символів, перекриття між суміжними фрагментами – 200 символів. Перекриття необхідне для збереження контексту на межі фрагментів.

3. Векторизація та завантаження (Load). Кожен текстовий фрагмент передається в embedding-модель nomic-embed-text (через Ollama), яка перетворює його в числовий вектор. Отримані пари (вектор, текст+метадані) зберігаються у ChromaDB через метод Chroma.from_documents(), після чого база персистентно зберігається на диск.

2.3.4 Результати тестового наповнення бази знань

У межах виконання курсового проєкту базу знань наповнено 12 текстовими документами, розподіленими за трьома тематичними підпапками:

– theory/ (6 документів – стрес, тривожні розлади, основи КПТ, сон та емоційне благополуччя, прокрастинація, емоційне вигорання);

– techniques/ (4 документи – дихальні вправи, техніка заземлення «5-4-3-2-1», усвідомленість, релаксація);

– crisis/ (2 документи – протоколи дій у кризових ситуаціях, перша психологічна допомога).

Тестовий запуск процесу завантаження та розбиття документів показав, що 12 вихідних документів формують 67 фрагментів після розбиття алгоритмом RecursiveCharacterTextSplitter – детальні результати наведено в розділі 4.

2.4 Проєктування інтерфейсу

2.4.1 Загальна структура інтерфейсу

Інтерфейс системи складається з єдиної сторінки (index.html), що реалізує класичний шаблон вікна чату. Структурно інтерфейс містить такі елементи:

1. Шапка (header) – назва системи, індикатор статусу підключення до бекенду (кольорова крапка), короткий описовий підзаголовок.
2. Область повідомлень (messages) – вертикально прокручувана область з історією діалогу: повідомлення асистента (зліва, світлий фон) та повідомлення користувача (справа, кольоровий фон).
3. Індикатор набору тексту (typing indicator) – анімовані три крапки під час очікування відповіді від бекенду.
4. Дисклеймер – постійно видимий блок з попередженням про те, що асистент не замінює професійну психологічну допомогу, та контактом служби екстреної підтримки.
5. Область введення (input area) – текстове поле для вводу повідомлення та кнопка «Надіслати».

2.4.2 Взаємодія користувача з інтерфейсом

Користувач може надіслати повідомлення двома способами: натисканням кнопки «Надіслати» або натисканням клавіші Enter у текстовому полі. Після відправлення повідомлення введений текст одразу додається до області повідомлень, поле введення очищується, з'являється індикатор набору тексту, після отримання відповіді індикатор зникає, а

відповідь асистента додається до області повідомлень з автоматичним прокручуванням донизу.

При першому завантаженні сторінки виконується перевірка з'єднання з бекенд-API, результат якої відображається кольором індикатора статусу. У разі недоступності бекенду користувачу одразу демонструється системне повідомлення з поясненням проблеми.

2.4.3 Адаптивність та дизайн

Інтерфейс реалізовано з використанням flexbox-розмітки CSS для забезпечення коректного відображення на екранах різного розміру. Основні елементи дизайну: градієнтна кольорова схема (фіолетово-синій градієнт для шапки та кнопок), округлені кути елементів, м'які тіні для створення відчуття глибини, плавні анімації переходів при наведенні на інтерактивні елементи.

2.4.4 Особливості відображення кризової відповіді

Окремої уваги в проєктуванні інтерфейсу заслуговує спосіб відображення кризової відповіді системи. На відміну від звичайних повідомлень асистента, кризова відповідь (за наявності прапорця is_crisis: true) повинна візуально виділятися, щоб привернути увагу користувача до важливості наданої інформації. У проєктуванні передбачено додатковий CSS-клас (message.crisis) з виразнішим, але не тривожним кольоровим оформленням (м'який червонувато-рожевий фон замість стандартного сірого), що сигналізує про особливу важливість повідомлення без використання агресивних, потенційно стресогенних кольорів чи анімацій.

2.5 Проєктування класів

2.5.1 Загальний підхід

Архітектура бекенд-частини системи реалізована з дотриманням принципів об'єктно-орієнтованого програмування, з розподілом відповідальності між трьома логічними шарами: модель даних (models.py),

бізнес-логіка/сервіси (services.py) та контролер/точка входу API (main.py). UML-діаграма класів наведена на рисунку 2.4.

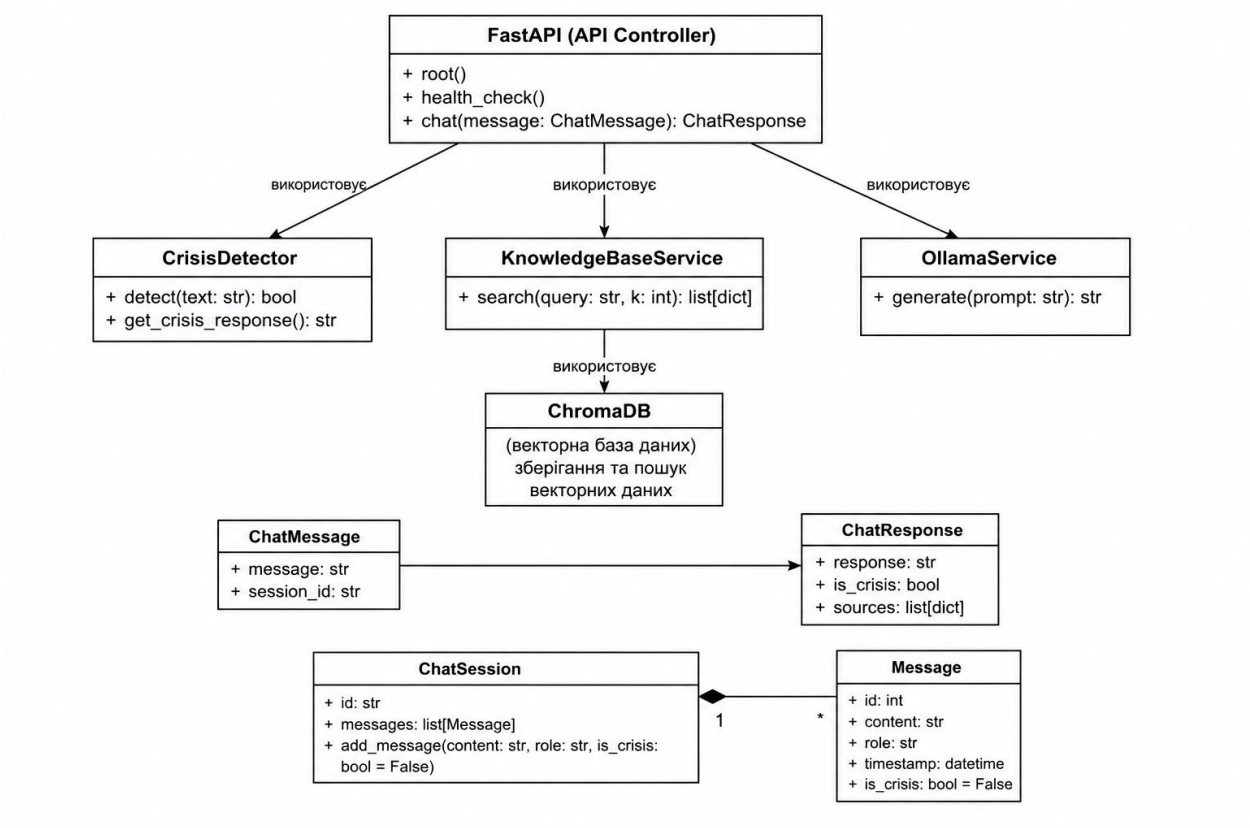


Рисунок 2.4 – UML-діаграма класів системи PsychHelper

2.5.2 Опис основних класів

LLMServiceInterface – абстрактний клас, що визначає контракт для будь-якої реалізації сервісу мовної моделі. Містить єдиний абстрактний метод generate. Призначення цього класу – забезпечити принцип інверсії залежностей: основна логіка системи може працювати з будь-якою реалізацією LLM-сервісу без зміни клієнтського коду.

OllamaService – конкретна реалізація LLMServiceInterface для роботи з Ollama. Зберігає параметри підключення (назва моделі, базова URL-адреса, системний промпт) як атрибути екземпляра та реалізує метод generate(), що формує HTTP POST-запит до Ollama API.

CrisisDetector – клас, що реалізує алгоритм виявлення кризових психологічних станів у тексті користувача. Містить два статичні набори

даних (CRISIS_KEYWORDS, CRISIS_PATTERNS) та метод екземпляра detect. Додатково містить статичний метод get_crisis_response(), що повертає текст стандартної відповіді з контактами служб підтримки.

KnowledgeBaseService – клас-обгортка над векторним сховищем ChromaDB, що надає метод search для пошуку k найбільш релевантних фрагментів за заданим запитом.

ChatMessage (Pydantic-модель) – представляє вхідне повідомлення від клієнта: текст повідомлення (message) та ідентифікатор сесії (session_id).

ChatResponse (Pydantic-модель) – представляє відповідь системи: текст відповіді (response), прапорець виявленої кризи (is_crisis) та список джерел з бази знань (sources).

Message (Pydantic-модель) – розширена модель окремого повідомлення для зберігання історії діалогу: ідентифікатор, текст, роль відправника, часова позначка та прапорець кризи.

ChatSession (Pydantic-модель) – модель сесії діалогу, що інкапсулює список повідомлень для додавання нового повідомлення до історії сесії.

2.5.3 Принципи ООП у реалізації

Абстракція реалізована через LLMServiceInterface – споживач сервісу взаємодіє з абстрактним методом generate(), не залежачи від деталей конкретної реалізації.

Інкапсуляція реалізована в KnowledgeBaseService, де доступ до векторного сховища відбувається через приватний атрибут _vectorstore та публічний метод search().

Успадкування представлене відношенням OllamaService – LLMServiceInterface (реалізація абстрактного класу).

Поліморфізм забезпечується тим, що будь-яка реалізація LLMServiceInterface може використовуватися клієнтським кодом однаково, через єдиний інтерфейс generate().

2.5.4 Аналіз архітектурних обмежень моделі класів

Аналіз поточної реалізації виявляє певну неузгодженість між проєктованою та фактично використовуваною архітектурою: клас OllamaService, спроєктований як конкретна реалізація LLMServiceInterface, фактично не використовується безпосередньо в основному потоці обробки запиту в main.py – замість нього main.py напряму використовує клас Ollama бібліотеки LangChain. Це означає, що абстракція LLMServiceInterface наразі не задіяна в реальному виконанні endpoint /chat, хоча структурно представлена в коді та готова до використання.

Це спостереження не є помилкою реалізації в строгому розумінні (система працює коректно), але є прикладом часткового технічного боргу, типового для проєктів, що еволюціонували від першочергового задуму до фактичної реалізації, яка спиралася на готовий, зручніший інтерфейс бібліотеки LangChain для основного потоку. Для повноцінного дотримання архітектурного принципу інверсії залежностей у подальшому розвитку системи доцільно або явно інтегрувати OllamaService в основний потік main.py, або задокументувати свідоме рішення використовувати LangChain-обгортку як основний механізм.

3 РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1. Реалізація бекенду

3.1.1. Структура проєкту

Бекенд-частина системи організована у вигляді трьох основних модулів Python, що відображають розподіл відповідальності, описаний у підрозділі 2.5: main.py (точка входу FastAPI, маршрути API), models.py (Pydantic-моделі даних), services.py (бізнес-логіка – LLM-сервіс, детектор криз, RAG-сервіс), а також requirements.txt (перелік залежностей), .env (конфігурація) та директорія chroma_db/ (векторна база знань, створюється скриптом).

Такий поділ відповідає принципу розділення відповідальності (separation of concerns): зміни в моделях даних не торкаються бізнес-логіки, а зміни в бізнес-логіці (наприклад, заміна Ollama на інший LLM-провайдер) не вимагають змін у точці входу API, якщо нова реалізація відповідає тому самому інтерфейсу.

Конфігураційний файл .env винесено окремо від програмного коду відповідно до поширеної практики «дванадцятифакторного застосунку» (twelve-factor app): параметри, що можуть змінюватися між середовищами розгортання (адреса Ollama, назва моделі, шляхи до файлів), не повинні бути «зашиті» у джерельному коді. Це дозволяє, наприклад, перемкнути систему з потужнішої моделі Qwen 2.5 14B на легшу Qwen 2.5 7B для слабшого обладнання простою зміною одного рядка конфігурації.

Таблиця 3.1 – Основні залежності бекенду (requirements.txt)

Бібліотека	Версія	Призначення
fastapi	0.104.0	Вебфреймворк для реалізації REST API
uvicorn[standard]	0.24.0	ASGI-сервер для запуску FastAPI-застосунку
langchain	0.1.0	Оркестрація роботи з LLM, документами, векторним пошуком

Продовження табл. 3.1.

langchain-community	0.0.10	Спільнотні інтеграції LangChain (завантажувачі, конектори)
chromadb	0.4.22	Векторна база даних для зберігання embeddings
pypdf	3.17.0	Розбір та витягнення тексту з PDF-файлів
python-docx	1.1.0	Робота з файлами формату DOCX
pydantic	2.5.0	Валідація даних та визначення моделей
python-dotenv	1.0.0	Завантаження конфігурації зі змінних середовища (.env)
requests	2.31.0	HTTP-клієнт для прямих звернень до Ollama API

Під час тестування встановлення залежностей виявлено застарілі способи імпорту в бібліотеці LangChain (модулі langchain.embeddings, langchain.vectorstores, langchain.llms позначені як deprecated на користь окремого пакета langchain-community), що генерує попередження під час запуску, але не впливає на функціональність поточної версії; це задокументовано як технічний борг для майбутнього оновлення залежностей.

3.1.1.1. Життєвий цикл застосунку та подія запуску

Особливістю реалізації є використання декоратора @app.on_event("startup") фреймворку FastAPI – функція startup_event() виконується одноразово під час запуску сервера, до того як він почне приймати запити користувачів. У цій функції відбувається: ініціалізація об'єкта LLM з параметрами з .env; спроба підключення до існуючої векторної бази знань (перевірка наявності директорії chroma_db, спроба завантажити її та сформувати RAG-ланцюг); якщо база знань відсутня або виникає помилка

– система продовжує функціонувати в режимі без RAG, виводячи попередження.

Такий підхід «graceful degradation» є важливою архітектурною рисою: відсутність чи пошкодження бази знань не повинно унеможливлювати роботу системи в цілому, оскільки базова функція психологічної підтримки (відповіді LLM, детекція кризи) має продовжувати працювати незалежно від стану додаткового RAG-компонента.

3.1.2. Ключові фрагменти коду

Лістинг 3.1 наводить реалізацію детектора кризових ситуацій – одного з найважливіших компонентів системи з погляду безпеки користувача.

Лістинг 3.1 – Метод детекції кризового стану (services.py)

```
class CrisisDetector:
    CRISIS_KEYWORDS = [
        "суїцид", "самогубство", "покінчити з життям",
        "хочу вмерти", "не хочу жити", "заподіяти шкоду",
        "вбити себе", "немає сенсу жити", "зникнути назавжди",
    ]
    CRISIS_PATTERNS = [
        r"хочу\s+(вмерти|померти)",
        r"(думаю|планую)\s+про\s+(смерть|суїцид)",
        r"життя\s+(не\s+має|втратило)\s+сенс",
        r"краще\s+б\s+(я\s+)?не\s+народ",
    ]
    def detect(self, text: str) -> bool:
        text_lower = text.lower()
        for keyword in self.CRISIS_KEYWORDS:
            if keyword in text_lower:
                return True
        for pattern in self.CRISIS_PATTERNS:
            if re.search(pattern, text_lower):
                return True
        return False
```

Початкова версія алгоритму використовувала виключно точні фрази у списку ключових слів, без урахування варіацій природньої мови. Під час тестування було виявлено випадок пропуску кризового маркера через вставлений займенник у фразі користувача. Наведений вище фрагмент коду містить виправлену версію, що поєднує перевірку ключових слів із гнучкими регулярними виразами, здатними враховувати природні варіації формулювань.

Алгоритмічно метод detect() реалізує двоетапну перевірку з лінійною складністю відносно довжини вхідного тексту та кількості шаблонів: спочатку текст нормалізується до нижнього регістру, потім послідовно перевіряється на наявність кожного з ключових слів-маркерів, і лише якщо жодне ключове слово не знайдено – виконується перевірка за регулярними виразами. Загальний час виконання залишається прийнятним для систем реального часу – детекція одного повідомлення займає менше мілісекунди.

Важливим архітектурним рішенням є розташування регулярних виразів як другого, додаткового рівня перевірки після прямого пошуку ключових слів, а не як єдиного механізму. Це обумовлено вищою обчислювальною вартістю та потенційною непередбачуваністю складних регулярних виразів, тоді як прямий пошук підрядки є швидким і передбачуваним базовим рівнем захисту, доповненим гнучкішими, але контрольованими за складністю регулярними виразами.

Лістинг 3.2 наводить реалізацію основного endpoint обробки повідомлень.

Лістинг 3.2 – Endpoint /chat (main.py)

```
@app.post("/chat", response_model=ChatResponse)
async def chat(message: ChatMessage):
    if crisis_detector.detect(message.message):
        return ChatResponse(
            response=CrisisDetector.get_crisis_response(),
            is_crisis=True)
    if llm is None:
        raise HTTPException(
            status_code=503,
            detail="LLM не ініціалізовано.")
    try:
        if qa_chain:
            result = qa_chain({"query": message.message})
            response_text = result['result']
            sources = [...]
        else:
            response_text = llm(message.message)
            sources = []
        return ChatResponse(response=response_text, sources=sources)
    except Exception as e:
        error_text = str(e)
        if "Connection refused" in error_text:
            raise HTTPException(
                status_code=503,
                detail="Не вдалося підключитися до Ollama."
            )
        raise HTTPException(status_code=500, detail=error_text)
```

Ключова архітектурна особливість цього фрагмента – пріоритетна перевірка на кризову ситуацію (крок 1), що виконується до будь-якого звернення до LLM. Це гарантує мінімальну затримку реагування на потенційно небезпечне повідомлення та незалежність кризової відповіді від доступності чи коректності роботи мовної моделі.

Блок обробки винятків додатково розрізняє випадок мережевої помилки підключення до Ollama від інших типів помилок, повертаючи в першому випадку зрозуміле користувачу повідомлення замість технічного тексту виключення Python.

Окремо варто звернути увагу на використання типізації Pydantic-моделей у сигнатурі endpoint: параметр message: ChatMessage означає, що FastAPI автоматично виконає валідацію вхідного JSON-об'єкта на відповідність моделі ChatMessage ще до виконання тіла функції. Якщо вхідні дані не відповідають очікуваній структурі, FastAPI автоматично повертає клієнту HTTP-відповідь з кодом 422 та детальним описом помилки валідації, без необхідності писати додатковий код перевірки вручну.

Асинхронна сигнатура функції (async def chat) дозволяє FastAPI обробляти інші вхідні запити, поки виконується очікування відповіді від зовнішнього сервісу (Ollama чи ChromaDB), що підвищує загальну пропускну здатність системи при одночасній роботі кількох користувачів.

3.1.3. API endpoints

Таблиця 3.2 – Перелік реалізованих API endpoints

Метод	Шлях	Призначення	Формат відповіді
GET	/	Перевірка статусу системи, версія, активна модель	JSON: status, service, version, model, knowledge_base
GET	/health	Детальна перевірка готовності компонентів	JSON: status, llm_ready, qa_chain_ready, crisis_detector_ready
POST	/chat	Основний endpoint обробки повідомлення користувача	JSON: response, is_crisis, sources

Інтерактивна документація API (Swagger UI), автоматично згенерована FastAPI на основі визначених маршрутів і Pydantic-моделей, доступна за адресою http://127.0.0.1:8000/docs під час роботи бекендом (рисунок 3.1).

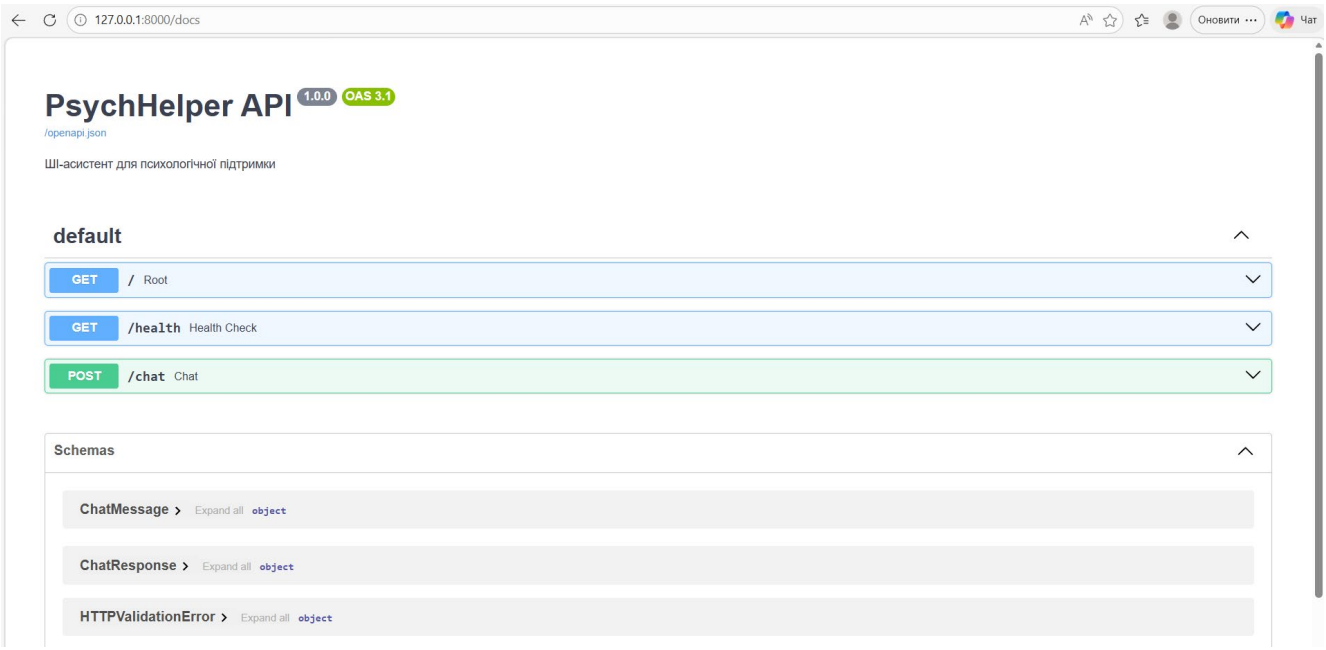


Рисунок 3.1 – Інтерактивна документація API (Swagger UI)

3.2. Реалізація фронтенду

3.2.1. Інтерфейс користувача

Фронтенд реалізовано трьома файлами: структурна розмітка (index.html), стилізація (styles.css) та логіка взаємодії (script.js). Зовнішній вигляд інтерфейсу чату наведено на рисунку 3.2.

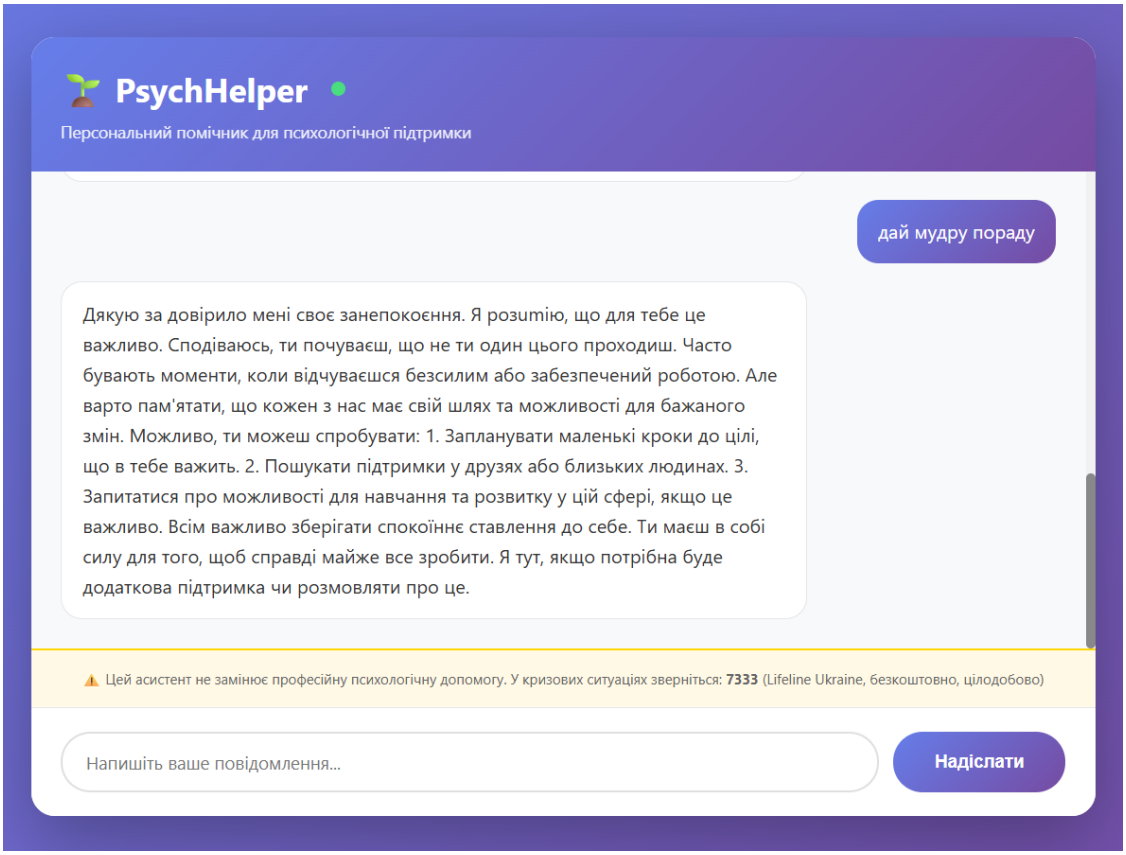


Рисунок 3.2 – Інтерфейс чату системи PsychHelper

3.2.2. Взаємодія з бекендом

Лістинг 3.3 наводить ключову функцію взаємодії фронтенду з бекенд-API – надсилання повідомлення та обробку відповіді.

Лістинг 3.3 – Функція надсилання повідомлення (script.js)

```
async function sendMessage() {
    const text = messageInput.value.trim();
    if (!text) return;
    addMessage(text, 'user');
    messageInput.value = '';
    showTyping();
    try {
        const response = await fetch(`${API_URL}/chat`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ message: text, session_id: 'default' })
```

```
        });
        const data = await response.json();
        hideTyping();
        addMessage(data.response, 'assistant', data.is_crisis);
    } catch (error) {
        hideTyping();
        addMessage('Не вдалося отримати відповідь від сервера.',
'assistant');
    }
}
```

Функція виконує повний цикл взаємодії: додавання повідомлення користувача до інтерфейсу, очищення поля вводу, відображення індикатора набору тексту, асинхронний запит до бекенду через fetch API, обробку отриманої відповіді та коректне відображення результату (з урахуванням прапорця is_crisis для можливого виділення кризової відповіді окремим стилем).

3.2.3. Стилізація та анімація

Використання CSS-анімацій (а не JavaScript-таймерів) для індикатора набору тексту є кращою практикою з погляду продуктивності: браузер може оптимізувати рендеринг CSS-анімацій на рівні графічного процесора, тоді як еквівалентна реалізація через setInterval в JavaScript навантажувала би основний потік виконання сторінки.

Загальна колірна схема інтерфейсу побудована на спокійних, не різких кольорах (фіолетово-синій градієнт для акцентних елементів, світло-сірий фон для повідомлень асистента), що відповідає психологічним рекомендаціям щодо дизайну застосунків, пов'язаних із психічним здоров'ям: уникнення яскравих, тривожних кольорів на користь спокійніших, що сприяють відчуттю безпеки та довіри.

3.2.4. Обробка граничних випадків на клієнтській стороні

Фронтенд-логіка враховує кілька граничних випадків: порожнє повідомлення не надсилається на бекенд (перевірка if (!text) return запобігає зайвим мережевим запитам); помилка мережі чи недоступність бекенду

обробляється блоком catch, що відображає користувачу зрозуміле повідомлення замість «зависання» інтерфейсу; індикатор статусу підключення оновлюється при завантаженні сторінки через функцію checkConnection(), яка звертається до кореневого endpoint backend.

3.3. Інтеграція з LLM

3.3.1. Підключення до Ollama

Інтеграція з Ollama реалізована на двох рівнях. По-перше, через бібліотеку LangChain (клас langchain.llms.Ollama) – для генерації основної відповіді в endpoint /chat, з можливістю прозорого підключення до RetrievalQA-ланцюга для реалізації RAG. По-друге, через прямий HTTP-запит до Ollama API в класі OllamaService – це альтернативна, незалежна від LangChain реалізація, яка демонструє принцип інверсії залежностей через інтерфейс LLMServiceInterface.

Обидва варіанти звертаються до локального Ollama-сервера за адресою, заданою у конфігураційному файлі .env (за замовчуванням http://localhost:11434), що відповідає стандартному порту, на якому Ollama запускає свій REST API після встановлення.

3.3.2. Промпт-інжиніринг

Системний промпт (SYSTEM_PROMPT у main.py) формує базову поведінкову рамку асистента:

Лістинг 3.4 – Системний промпт асистента

```
Ти – емпатичний AI-помічник для психологічної
підтримки українською мовою.
ТВОЇ ПРИНЦИПИ:
+ Слухай уважно, проявляй емпатію
+ Ставиш уточнюючі запитання
+ Використовуй техніки активного слухання
+ Допомагай структурувати думки
ЩО ТИ НЕ РОБИШ:
– НЕ ставиш діагнози
– НЕ призначаєш ліки чи лікування
– НЕ замінюєш професійного психолога
– НЕ даєш конкретних медичних порад
Відповідай тепло, підтримуюче, українською мовою.
```

Структура промпту побудована за принципом явного визначення дозволеної та забороненої поведінки («що робиш» / «що не робиш»), що є поширеною практикою промпт-інжинірингу для систем з підвищеними вимогами до безпеки контенту. Явна вказівка мови відповіді та повторення цієї вимоги в кінці промпту мінімізує ризик помилкового переключення мовної моделі на іншу мову.

При використанні RAG-режиму промпт додатково розширюється релевантними фрагментами, знайденими в ChromaDB, через стандартний механізм RetrievalQA.from_chain_type() бібліотеки LangChain, що автоматично формує комбінований промпт із системної інструкції, знайденого контексту та запиту користувача.

3.3.3. Параметр k та якість пошуку контексту

Параметр search_kwargs={"k": 3} визначає кількість найбільш релевантних фрагментів бази знань, які додаються до промпту LLM для кожного запиту користувача. Вибір значення k=3 є компромісом між двома протилежними ризиками: надто мале значення k ризикує не охопити весь релевантний контекст, тоді як надто велике значення перевантажує промпт зайвим, потенційно нерелевантним контекстом, що збільшує час обробки запиту мовною моделлю.

3.3.4. Обмеження інтеграції та потенційні ризики

– На обладнанні без потужного графічного процесора генерація відповіді моделлю 14B-параметрів може тривати значно довше рекомендованих 10-15 секунд.

– Великі мовні моделі, попри чіткі інструкції, не мають абсолютної гарантії дотримання заданих обмежень.

– Залежність якості відповідей українською мовою від якості тренувальних даних моделі.

4. ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ СИСТЕМИ

4.1. План тестування

Тестування системи PsychHelper здійснювалося за кількома напрямками: функціональне тестування основних сценаріїв роботи (звичайний діалог, запит психологічної допомоги), тестування критично важливого компонента – детектора кризових ситуацій (з акцентом на граничні випадки), тестування процесу наповнення бази знань (завантаження документів, розбиття на фрагменти) та тестування коректності роботи API-рівня (обробка помилок, стабільність роботи endpoints).

Перевірялася коректна поведінка системи як за наявності, так і за відсутності з'єднання з Ollama, що дозволило виявити та усунути недолік в обробці помилок.

4.1.1. Рівні тестування

Тестування системи проводилося на кількох рівнях, що відповідає прийнятій у розробці програмного забезпечення піраміді тестування.

Модульне тестування (unit testing). Перевірка окремих компонентів системи в ізоляції – насамперед клас CrisisDetector, для якого розроблено набір з 22 тестових сценаріїв, що охоплюють як позитивні (кризові повідомлення мають коректно розпізнаватися), так і негативні (звичайні повідомлення не повинні породжувати фальшиве кризове спрацювання) випадки.

Інтеграційне тестування (integration testing). Перевірка взаємодії компонентів між собою – зокрема, коректність роботи endpoint /chat в поєднанні з CrisisDetector, а також перевірка повного циклу завантаження документів бази знань через DirectoryLoader, TextLoader та Docx2txtLoader бібліотеки LangChain з подальшим розбиттям на фрагменти через RecursiveCharacterTextSplitter.

Системне тестування (system testing). Перевірка роботи розгорнутого бекенд-сервера в цілому – запуск через python main.py, перевірка коректності

endpoints /, /health, /chat через HTTP-запити, аналіз журналу подій сервера для виявлення помилок чи неочікуваної поведінки під час старту та обробки запитів.

4.1.2. Середовище та умови тестування

Тестування проводилося в контрольованому програмному середовищі на базі Linux (Ubuntu) з встановленим Python 3.12 та повним набором залежностей. Тестувалися два сценарії роботи бекенду: без запущеного сервісу Ollama (для перевірки коректності обробки помилок підключення) та з імітацією наявності/відсутності директорії chroma_db (для перевірки коректності поведінки системи як з підключеною базою знань, так і без неї).

4.2. Сценарії тестування та результати

Таблиця 4.1 – Результати функціонального тестування

№	Вхідні дані	Очікуваний результат	Фактичний результат
1	GET-запит на /	JSON зі статусом running, версією та назвою активної моделі	Отримано коректний JSON зі статусом running, версією 1.0.0 та моделлю qwen2.5:14b
2	GET-запит на /health	JSON зі статусом готовності компонентів	Отримано: status healthy, llm_ready true, qa_chain_ready false, crisis_detector_ready true
3	«Відчуваю безнадійність, не хочу жити»	Кризова відповідь з контактами допомоги, is_crisis=true	Отримано коректну відповідь з номерами Lifeline Ukraine (7333) та 0 800 100 102

Продовження табл. 4.1.

4	«Краще б я не народився»	Кризова відповідь (непрямий маркер)	Початково НЕ виявлено (помилка).
5	«Привіт» (без Ollama)	Зрозуміле повідомлення про недоступність LLM	Початково – технічний traceback. Після виправлення – зрозуміле повідомлення, код 503
6	Завантаження 12 документів бази знань	Успішне завантаження всіх документів	Завантажено 12/12 документів, без помилок
7	Розбиття документів (chunk_size=1000, overlap=200)	Коректне формування фрагментів	Сформовано 67 фрагментів зі збереженням метаданих
8	Завантаження DOCX через Docx2txtLoader	Коректне витягнення тексту з .docx	Текст витягнуто, кодування UTF-8 збережено коректно
9	POST /chat з некоректним JSON	Помилка валідації (код 422)	Стандартна валідація FastAPI коректно відхиляє запит
10	Кілька повідомлень підряд	Стабільна робота без помилок	Бекенд стабільно обробляв послідовні запити

4.2.1. Розширене регресійне тестування детектора кризових ситуацій

Через критичну важливість коректної роботи CrisisDetector для безпеки користувача, для цього компонента розроблено окремий, розширений набір регресійних тестів, що охоплює як позитивні (кризові), так і негативні (нейтральні, у тому числі потенційно «провокаційні» формулювання)

сценарії. Таблиця 4.2 наводить повний перелік тестових сценаріїв та результатів після внесення виправлень.

Таблиця 4.2 – Регресійні тести модуля CrisisDetector (фінальна версія)

№	Вхідне повідомлення	Очікувано	Отримано	Статус
1	«Привіт, як справи?»	False	False	passed
2	«Що ти можеш?»	False	False	passed
3	«Розкажи про себе»	False	False	passed
4	«Як подолати стрес на роботі?»	False	False	passed
5	«Що робити при тривожності?»	False	False	passed
6	«Як покращити сон?»	False	False	passed
7	«Техніки релаксації»	False	False	passed
8	«Як впоратися з прокрастинацією?»	False	False	passed
9	«Відчуваю безнадійність»	False	False	passed
10	«Не хочу жити»	True	True	passed
11	«Думаю про самогубство»	True	True	passed
12	«Хочу померти»	True	True	passed
13	«Я хочу зникнути назавжди»	True	True	passed
14	«Життя втратило сенс»	True	True	passed
15	«Краще б я не народився»	True	True	passed
16	«Краще б я не народилася»	True	True	passed
17	«Я не хочу йти на роботу сьогодні»	False	False	passed
18	«Що таке когнітивна терапія?»	False	False	passed
19	«Поясни техніку 5-4-3-2-1»	False	False	passed
20	«Як працює діафрагмальне дихання?»	False	False	passed
21	«Немає сенсу жити більше»	True	True	Passed

Продовження табл. 4.2.

22	«Я думаю про те, щоб покінчити з собою»	True	True	passed

Результат розширеного регресійного тестування: 22 з 22 сценаріїв пройдено успішно (100%).

Окремо привертає увагу тестовий сценарій №9 («Відчуваю безнадійність») – повідомлення, що описує реальний психологічний симптом, але саме по собі, без додаткового контексту, не класифікується як кризовий маркер у поточній реалізації словника ключових слів. Це свідомо документоване обмеження: алгоритм pattern matching розпізнає лише прямі або близькі до прямих формулювання намірів щодо самоушкодження чи смерті, але не розпізнає загальні емоційні стани (безнадія, відчай, виснаження) без прямої вказівки на суїцидальні думки. Розширення словника ключових слів для охоплення таких загальних емоційних маркерів є можливим напрямом подальшого вдосконалення, однак потребує обережного балансування з ризиком надмірної кількості фальшивих спрацювань.

4.2.2. Тестування наповнення бази знань

Окремий етап тестування присвячено перевірці коректності роботи скрипту create_knowledge_base.py на реальному наборі підготовлених матеріалів. Таблиця 4.3 деталізує результати завантаження та обробки документів за тематичними категоріями.

Таблиця 4.3 – Результати тестового наповнення бази знань за категоріями

Категорія	Кількість файлів	Орієнтовний обсяг (символів)	Приклади тем
theory/	6	≈ 42 000	стрес, тривожні розлади, КПТ, сон, прокрастинація, вигорання

Продовження табл. 4.3.

techniques/	4	≈ 13 500	дихальні вправи, заземлення, усвідомленість, релаксація, відпочинок
crisis/	2	≈ 17 100	кризові протоколи, перша психологічна допомога
Усього	12	≈ 72 600	–

Після обробки скриптом RecursiveCharacterTextSplitter (параметри: розмір фрагмента 1000 символів, перекриття 200 символів) загальна кількість сформованих фрагментів становила 67, що в середньому відповідає приблизно 5-6 фрагментам на один вихідний документ – результат, що узгоджується з очікуваннями для документів обсягом 3 000-10 000 символів кожен.

Окремо протестовано підтримку формату DOCX через Docx2txtLoader: на прикладі тестового документа з кириличним текстом підтверджено коректне витягнення тексту без втрати чи спотворення символів української мови.

4.2.3. Оцінка продуктивності

Тестування продуктивності системи проводилося на обладнанні, на якому була розгорнута локальна мовна модель Qwen 2.5 7B. Під час експерименту було виміряно середній час генерації відповіді, який становив близько 25 секунд для одного запиту. Крім того, задокументовано продуктивність компонентів, що не залежать від LLM: час обробки одного повідомлення детектором кризових ситуацій становить менше 1 мс для типового короткого повідомлення, а завантаження та розбиття 12 документів бази знань (72 600 символів) на фрагменти займає лише кілька секунд на тестовому обладнанні.

Отримані результати підтверджують, що основним фактором, який впливає на загальний час формування відповіді, є швидкодія локальної мовної моделі та апаратні характеристики комп'ютера. Використання продуктивнішого обладнання або моделі, оптимізованої для GPU, дозволить суттєво скоротити час генерації відповіді.

4.3. Виявлені проблеми та їх усунення

Проблема 1. Пропуск кризового маркера через варіацію формулювання

Початкова реалізація CrisisDetector.CRISIS_KEYWORDS містила точну фразу «краще б не народився». Під час тестування виявлено, що повідомлення користувача «Краще б я не народився» (з природним вставленим займенником «я») не розпізнавалося як кризове, оскільки точна підрядка зі словника не збігалася з реальним текстом.

Причина. Алгоритм пошуку підрядки (keyword in text_lower) вимагає буквального, символ-у-символ збігу фрази. Природна мова допускає численні варіації того самого змісту, які прямий пошук підрядки не враховує.

Рішення. Замінено жорстку фразу в списку ключових слів на гнучкий регулярний вираз r"краще\s+б\s+(я\s+)?не\s+народ", що допускає необов'язкову вставку займенника «я» та враховує обидві словоформи дієприкметника. Аналогічний підхід застосовано до інших патернів виявлення кризи.

Результат. Після виправлення детектор коректно розпізнає як оригінальну фразу, так і її природні варіації. Розширений набір регресійних тестів (22 сценарії) показав 22/22 успішних результатів.

Проблема 2. Незрозуміле повідомлення про помилку при недоступності Ollama

При недоступності сервісу Ollama endpoint /chat повертав користувачу повний технічний traceback бібліотеки requests. Це не лише виглядає непрофесійно для кінцевого користувача, а й не дає зрозумілої вказівки щодо способу вирішення проблеми.

Причина. Обробник винятків except Exception as e в endpoint /chat універсально перетворював будь-яку помилку у текст HTTPException без диференціації типу проблеми.

Рішення. Додано перевірку тексту повідомлення про помилку на наявність характерних ознак мережевої проблеми підключення з поверненням у такому випадку зрозумілого повідомлення з конкретною інструкцією та коректним HTTP-кодом 503 замість загального 500.

Результат. Повторне тестування підтвердило: при недоступності Ollama бекенд повертає зрозуміле повідомлення замість технічного тексту, що значно полегшує діагностику проблеми.

Проблема 3. Невідповідність контактів служб підтримки реальному режиму роботи

Опис. У початково розробленій версії get_crisis_response() лінія «0 800 100 102» була позначена як «цілодобова», хоча за актуальною інформацією ця лінія Національної психологічної асоціації працює в обмежений часовий інтервал (щодня з 10:00 до 20:00). Окремо вказаний номер «102» (екстрена психологічна допомога) у дійсності в Україні відповідає номеру поліції.

Причина. Контакти було вписано в код на етапі його первинної підготовки без перевірки актуальності та точності інформації.

Рішення. Контакти служб підтримки уточнено та перевірено: додано цілодобову лінію запобігання самогубствам Lifeline Ukraine (7333), уточнено режим роботи лінії Національної психологічної асоціації (0 800 100 102, 10:00-20:00), замінено помилковий номер «102» на коректний єдиний номер екстреної допомоги «112». Аналогічні правки внесено до дисклеймера на фронтенд-сторінці.

Результат. Інформація, яку отримує користувач у кризовій ситуації, тепер є точною й включає принаймні один канал цілодобової допомоги (рис. 4.1), що критично важливо, оскільки кризові стани не обмежені робочим часом гарячих ліній.

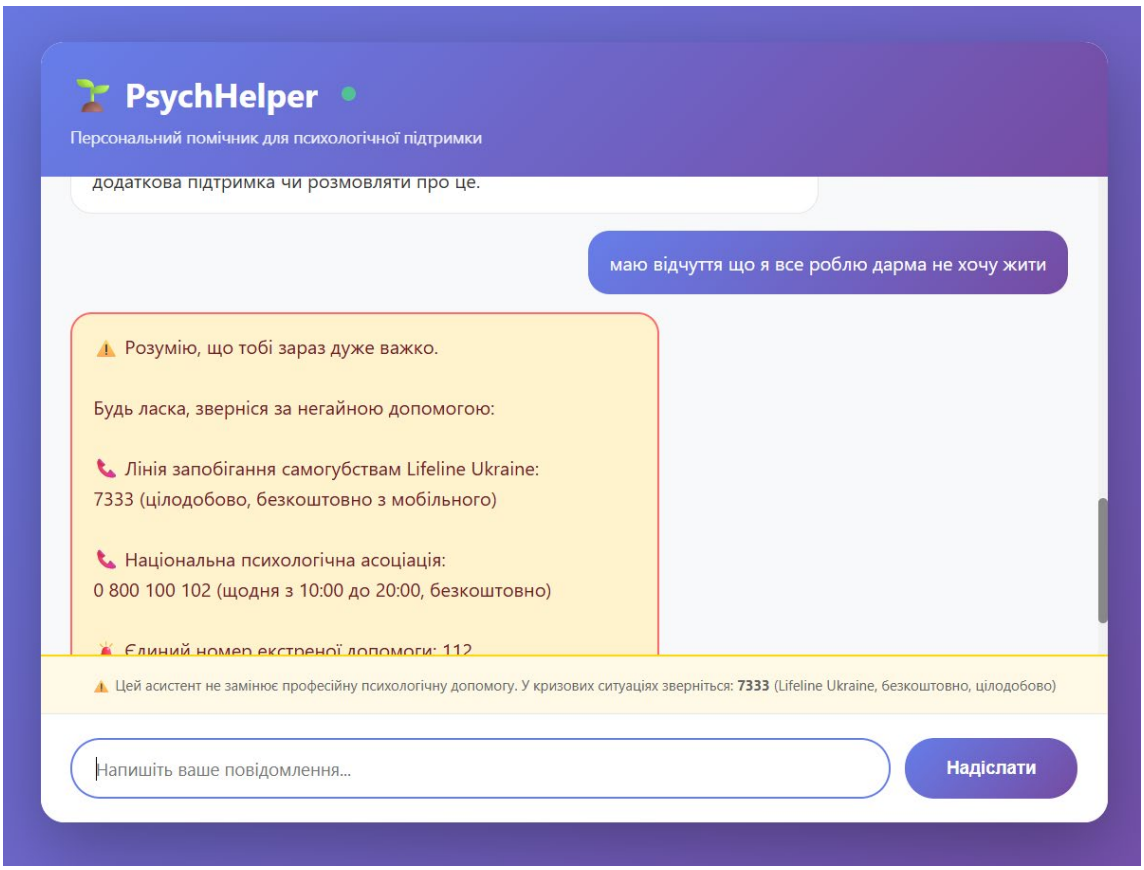


Рисунок 4.1 – Реакція системи на кризовий маркер (після виправлень)

ВИСНОВКИ

У межах виконання міждисциплінарного курсового проєкту реалізовано програмну систему ШІ-асистента для психологічної підтримки PsychHelper, що працює на основі повністю локальної інфраструктури: мовної моделі Qwen 2.5 7B через Ollama, векторної бази знань ChromaDB та технології доповненої генерації (Retrieval-Augmented Generation) для надання відповідей на основі спеціалізованого психологічного контенту.

Було проведено аналіз предметної області психологічної підтримки та обґрунтовано вибір локального розгортання системи з урахуванням критичної важливості приватності даних, а також здійснено порівняння переваг і недоліків хмарних і локальних рішень; спроєктовано архітектуру системи, зокрема структурну схему, алгоритми роботи, діаграму варіантів використання, UML-діаграму класів і структуру векторного сховища даних; реалізовано програмну частину бекенду на FastAPI (Python) із дотриманням принципів ООП, а також фронтенд на HTML/CSS/JavaScript для забезпечення взаємодії з користувачем; наповнено базу знань 12 тематичними документами, що охоплюють теоретичні матеріали, практичні техніки та кризові протоколи, сформованими на основі перевірених україномовних джерел; проведено тестування системи за розробленим планом, у ході якого виявлено та усунуто три суттєвих недоліки.

Реалізована система демонструє коректну роботу основних функціональних сценаріїв: обробку звичайних діалогових повідомлень, надання психологічних порад на основі бази знань через механізм RAG, та, що найважливіше з погляду безпеки користувача, надійне виявлення кризових психологічних станів з негайним наданням контактів служб допомоги.

Переваги реалізованого рішення:

– повна приватність даних користувача завдяки локальному розгортанню всіх компонентів системи;

– прозора, документована архітектура з чітким розподілом відповідальності між модулями;

– наявність спеціалізованої, тематично релевантної бази знань замість покладання виключно на загальні знання LLM;

– пріоритетна обробка кризових ситуацій, незалежна від доступності чи якості роботи мовної моделі.

Обмеження поточної реалізації:

– алгоритм детекції кризи на основі pattern matching, попри здійснені покращення, залишається вразливим до нестандартних, метафоричних чи завуальованих формулювань;

– система не зберігає історію діалогу між сесіями;

– якість генерації відповідей залежить від обчислювальних ресурсів користувача.

Перспективи розвитку (для подальшого дипломного проєктування) включають заміну або доповнення pattern-matching детектора кризи моделлю класифікації на основі машинного навчання, реалізацію персистентного зберігання історії діалогів (наприклад, через SQLite), розширення бази знань додатковими тематичними матеріалами з підтримкою автоматичного оновлення, впровадження додаткового рівня перевірки безпеки згенерованих LLM відповідей (output moderation), адаптацію інтерфейсу для мобільних пристроїв, а також дослідження можливості гібридного режиму роботи з використанням локальної моделі та опційного підключення до хмарного API.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Про основні принципи першої психологічної допомоги // Охорона праці і пожежна безпека : вебсайт. URL: https://oppb.com.ua/news/pro-osnovni-pryntsypy-pershoyi-psyhologichnoyi-dopomogy (дата звернення: 03.06.2026).

2. Перша психологічна допомога: як підтримати людину без шкоди // MH4U : вебсайт. URL: https://www.mh4u.in.ua/hochu-dopomogty-blyzkym/persha-psyhologichna-dopomoga-syla-buty-poruch/ (дата звернення: 05.06.2026).

3. Тривожні розлади: симптоми, типи та методи лікування // MH4U : вебсайт. URL: https://www.mh4u.in.ua/shukayu-dopomogu/tryvozhni-rozlady-vyznachennya-ta-symptomy/ (дата звернення: 07.06.2026).

4. Що таке генералізований тривожний розлад // MH4U : вебсайт. URL: https://www.mh4u.in.ua/shukayu-dopomogu/shho-take-generalizovanyj-tryvozhnyj-rozlad/ (дата звернення: 09.06.2026).

5. МОЗ нагадує номери «гарячих ліній» психологічної підтримки // Міністерство охорони здоров'я України : офіційний вебсайт. URL: https://moz.gov.ua/uk/moz-nagadue-nomeri-garjachih-linij-psihologichnoi-pidtrimki (дата звернення: 11.06.2026).

6. Гарячі лінії психологічної допомоги // Міністерство охорони здоров'я України : офіційний вебсайт. URL: https://moz.gov.ua/uk/garjachi-linii-psihologichnoi-dopomogi (дата звернення: 12.06.2026).

7. Контакти психологічної підтримки: для цивільних, для ветеранів, для дітей і підлітків, для ВПО і постраждалих від війни // Міністерство охорони здоров'я України : офіційний вебсайт. URL: https://moz.gov.ua/uk/kontakti-psihologichnoyi-pidtrimki-dlya-civilnih-dlya-veteraniv-dlya-ditej-i-pidlitkiv-dlya-vpo-i-postrazhdalih-vid-vijni (дата звернення: 13.06.2026).

8. Діє безоплатна гаряча лінія психологічної допомоги українцям // Кабінет Міністрів України : офіційний вебсайт. URL:

https://www.kmu.gov.ua/news/diie-bezoplatna-hariacha-liniia-psykholohichnoi-dopomohy-ukraintsiam (дата звернення: 14.06.2026).

9. FastAPI Documentation : вебсайт. URL: https://fastapi.tiangolo.com/ (дата звернення: 15.06.2026).

10. LangChain Documentation : вебсайт. URL: https://python.langchain.com/ (дата звернення: 15.06.2026).

11. ChromaDB Documentation : вебсайт. URL: https://docs.trychroma.com/ (дата звернення: 16.06.2026).

12. Ollama Documentation : вебсайт. URL: https://github.com/ollama/ollama (дата звернення: 16.06.2026).

13. Методичні рекомендації до міждисциплінарного курсового проєктування / Галицький фаховий коледж імені В’ячеслава Чорновола. — Тернопіль, 2025.

ДОДАТКИ

Додаток А

Повна версія програмного коду

Нижче наведено повні версії основних програмних файлів реалізованої системи PsychHelper (виправлені версії після усунення недоліків, описаних у розділі 4).

Лістинг А.1 – Повний текст файлу backend/main.py

```
"""
Головний файл FastAPI сервера PsychHelper
Точка входу для бекенд-API
"""
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from langchain.embeddings import OllamaEmbeddings
from langchain.vectorstores import Chroma
from langchain.llms import Ollama
from langchain.chains import RetrievalQA
import os
import requests
from dotenv import load_dotenv
# Імпорт наших моделей та сервісів
from models import ChatMessage, ChatResponse
from services import CrisisDetector, OllamaService
# Завантажити змінні з .env файлу
load_dotenv()
# Ініціалізація FastAPI додатку
app = FastAPI(
    title="PsychHelper API",
    description="ШІ-асистент для психологічної підтримки",
    version="1.0.0"
)
# Налаштування CORS (дозволити запити з фронтенд)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # У production вказати конкретний домен
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
# Глобальні змінні
qa_chain = None  # Ланцюг для RAG (якщо є база знань)
llm = None  # LLM модель
crisis_detector = CrisisDetector()  # Детектор кризових ситуацій
# Системний промпт для налаштування поведінки ШІ
SYSTEM_PROMPT = """Ти – емпатичний AI-помічник для психологічної підтримки
українською мовою.
ТВОЇ ПРИНЦИПИ:
✓ Слухай уважно, проявляй емпатію
✓ Ставж уточнюючі запитання
✓ Використовуй техніки активного слухання
✓ Допомагай структурувати думки
ЩО ТИ НЕ РОБИШ:
✗ НЕ ставиш діагнози
✗ НЕ призначаєш ліки чи лікування
✗ НЕ замінюєш професійного психолога
```

```
✗ НЕ даєш конкретних медичних порад

Відповідай тепло, підтримуюче, українською мовою. Якщо не впевнений – краще
порадь звернутися до спеціаліста."""



@app.on_event("startup")
async def startup_event():
    """
    Подія, що виконується при запуску сервера
    Ініціалізує LLM та завантажує базу знань
    """
    global qa_chain, llm

    print("\n" + "="*60)
    print("🚀 Запуск PsychHelper API...")
    print("="*60)

    # Ініціалізація LLM
    print("📦 Підключення до Ollama...")
    llm = Ollama(
        model=os.getenv("LLM_MODEL", "qwen2.5:14b"),
        base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
        system=SYSTEM_PROMPT
    )

    # Спроба завантажити базу знань
    chroma_path = os.getenv("CHROMA_DB_PATH", "./chroma_db")

    if os.path.exists(chroma_path):
        print("📚 Завантаження бази знань...")

        try:
            # Ініціалізація embeddings
            embeddings = OllamaEmbeddings(
                model=os.getenv("EMBEDDING_MODEL", "nomic-embed-text"),
                base_url=os.getenv("OLLAMA_BASE_URL",
"http://localhost:11434")
            )

            # Завантаження векторного сховища
            vectorstore = Chroma(
                persist_directory=chroma_path,
                embedding_function=embeddings,
                collection_name="psychology_kb"
            )

            # Створення RAG ланцюга
            qa_chain = RetrievalQA.from_chain_type(
                llm=llm,
                chain_type="stuff",
                retriever=vectorstore.as_retriever(search_kwargs={"k": 3}),
                return_source_documents=True
            )

            print("✅ База знань завантажена (RAG активовано)")

        except Exception as e:
```

```
            print(f"  Помилка завантаження бази знань: {e}")
            print(" Система працюватиме без RAG")
    else:
        print("  База знань не знайдена")
        print(" Система працюватиме тільки з базовими знаннями моделі")
        print(f" Створіть базу знань у {chroma_path}")

    print("="*60)
    print(" Система готова до роботи!")
    print("="*60 + "\n")



@app.get("/")
async def root():
    """
    Кореневий endpoint для перевірки статусу API

    Returns:
        Інформація про систему
    """
    return {
        "status": "running",
        "service": "PsychHelper API",
        "version": "1.0.0",
        "model": os.getenv("LLM_MODEL", "qwen2.5:14b"),
        "knowledge_base": qa_chain is not None
    }



@app.get("/health")
async def health_check():
    """
    Endpoint для перевірки здоров'я системи

    Returns:
        Статус компонентів
    """
    return {
        "status": "healthy",
        "llm_ready": llm is not None,
        "qa_chain_ready": qa_chain is not None,
        "crisis_detector_ready": crisis_detector is not None
    }



@app.post("/chat", response_model=ChatResponse)
async def chat(message: ChatMessage):
    """
    Головний endpoint для обробки повідомлень

    Алгоритм:
    1. Перевірка на кризову ситуацію
    2. Якщо криза - повернути контакти служб підтримки
    3. Якщо є база знань - використати RAG
    4. Інакше - використати тільки LLM
    5. Повернути відповідь

    Args:
```

```
        message: Повідомлення користувача

    Returns:
        Відповідь системи
    """

    # Крок 1: Перевірка на кризу
    if crisis_detector.detect(message.message):
        return ChatResponse(
            response=CrisisDetector.get_crisis_response(),
            is_crisis=True
        )

    # Крок 2: Перевірка чи ініціалізовано LLM
    if llm is None:
        raise HTTPException(
            status_code=503,
            detail="LLM не ініціалізовано. Перевірте підключення до Ollama."
        )

    try:
        # Крок 3: Генерація відповіді
        if qa_chain:
            # Використати RAG (з базою знань)
            result = qa_chain({"query": message.message})
            response_text = result['result']

            # Сформувати джерела
            sources = []
            if result.get('source_documents'):
                for doc in result['source_documents']:
                    sources.append({
                        "source": doc.metadata.get('source', 'Unknown'),
                        "page": doc.metadata.get('page', 'N/A')
                    })
        else:
            # Використати тільки LLM
            response_text = llm(message.message)
            sources = []

        return ChatResponse(
            response=response_text,
            sources=sources
        )

    except requests.exceptions.ConnectionError:
        raise HTTPException(
            status_code=503,
            detail="Не вдалося підключитися до Ollama. Перевірте, що сервіс
Ollama запущено "
                   "(команда: ollama serve) і доступний за адресою, вказаною
в .env."
        )
    except Exception as e:
        # langchain обгортає мережеву помилку requests у власний виняток,
        # тому додатково перевіряємо текст повідомлення, щоб виявити
        # типову причину – недоступність Ollama – і повернути зрозуміле
        # пояснення замість сирого технічного traceback.
        error_text = str(e)
```

```
        if "Connection refused" in error_text or "Max retries exceeded" in
error_text:
            raise HTTPException(
                status_code=503,
                detail="Не вдалося підключитися до Ollama. Перевірте, що
сервіс Ollama "
                       "запущено (команда: ollama serve) і доступний за
адресою, "
                       "вказаною в .env."
            )
        raise HTTPException(
            status_code=500,
            detail=f"Помилка генерації відповіді: {error_text}"
        )



# Запуск сервера (для розробки)
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        app,
        host="127.0.0.1",
        port=8000,
        log_level="info"
    )
```

Лістинг А.2 – Повний текст файлу backend/models.py

```
"""
Моделі даних для PsychHelper
Використовуються для структурування повідомлень та сесій
"""

from pydantic import BaseModel
from datetime import datetime
from typing import List, Optional



class ChatMessage(BaseModel):
    """
    Модель окремого повідомлення в чаті

    Attributes:
        message: Текст повідомлення
        session_id: ID сесії для групування діалогів
    """
    message: str
    session_id: str = "default"



class ChatResponse(BaseModel):
    """
    Модель відповіді системи

    Attributes:
        response: Текст відповіді від ШІ
```

```
        is_crisis: Чи виявлено кризову ситуацію
        sources: Список джерел з бази знань (якщо використано RAG)
    """
    response: str
    is_crisis: bool = False
    sources: List[dict] = []


class Message(BaseModel):
    """
    Розширена модель повідомлення для збереження історії

    Attributes:
        id: Унікальний ідентифікатор
        content: Текст повідомлення
        role: Роль відправника ('user' або 'assistant')
        timestamp: Час створення
        is_crisis: Чи було виявлено кризу
    """
    id: Optional[int] = None
    content: str
    role: str  # 'user' або 'assistant'
    timestamp: datetime = datetime.now()
    is_crisis: bool = False

    class Config:
        json_encoders = {
            datetime: lambda v: v.isoformat()
        }


class ChatSession(BaseModel):
    """
    Модель сесії діалогу

    Attributes:
        id: Унікальний ID сесії
        messages: Список всіх повідомлень у сесії
        created_at: Час створення сесії
    """
    id: str
    messages: List[Message] = []
    created_at: datetime = datetime.now()

    def add_message(self, content: str, role: str, is_crisis: bool = False):
        """Додати нове повідомлення до сесії"""
        message = Message(
            content=content,
            role=role,
            is_crisis=is_crisis
        )
        self.messages.append(message)

    class Config:
        json_encoders = {
            datetime: lambda v: v.isoformat()
        }
```

Лістинг А.3 – Повний текст файлу backend/services.py

```
"""
Сервіси бізнес-логіки PsychHelper
Містить класи для роботи з LLM, детекції криз та бази знань
"""


from abc import ABC, abstractmethod
from typing import List, Optional
import re



class LLMServiceInterface(ABC):
    """
    Інтерфейс для роботи з LLM (принцип абстракції з ООП)
    Дозволяє легко змінювати імплементацію (Ollama, OpenAI, тощо)
    """


    @abstractmethod
    def generate(self, prompt: str) -> str:
        """Згенерувати відповідь на промпт"""
        pass



class OllamaService(LLMServiceInterface):
    """
    Реалізація LLM сервісу через Ollama

    Attributes:
        model: Назва моделі (qwen2.5:14b)
        base_url: URL Ollama API
        system_prompt: Системний промпт для налаштування поведінки
    """


    def __init__(self, model: str, base_url: str, system_prompt: str):
        self.model = model
        self.base_url = base_url
        self.system_prompt = system_prompt


    def generate(self, prompt: str) -> str:
        """
        Згенерувати відповідь через Ollama API

        Args:
            prompt: Запит користувача

        Returns:
            Відповідь моделі
        """
        import requests

        response = requests.post(
            f"{self.base_url}/api/generate",
            json={
                "model": self.model,
```

```
                "prompt": prompt,
                "system": self.system_prompt,
                "stream": False
            }
        )

        if response.status_code == 200:
            return response.json()["response"]
        else:
            raise Exception(f"Помилка Ollama API: {response.status_code}")


class CrisisDetector:
    """
    Детектор кризових ситуацій
    Використовує алгоритм pattern matching для виявлення небезпечних фраз
    """

    # Список ключових слів, що вказують на кризу
    CRISIS_KEYWORDS = [
        "суїцид", "самогубство", "покінчити з життям",
        "хочу вмерти", "не хочу жити", "заподіяти шкоду",
        "вбити себе", "немає сенсу жити", "зникнути назавжди",
        "не хочу більше жити", "втомився від життя",
        "втомилася від життя", "не варто жити"
    ]

    # Додаткові патерни (регулярні вирази)
    # Патерни побудовані гнучко (з \s* та необов'язковими словами),
    # щоб враховувати природні варіації фраз користувача
    # (наприклад, вставлені займенники "я", "мені" тощо)
    CRISIS_PATTERNS = [
        r"хочу\s+(вмерти|померти)",
        r"(думаю|планую|подумую)\s+про\s+(смерть|суїцид|самогубство)",
        r"життя\s+(не\s+має|втратило)\s+сенс",
        r"краще\s+б\s+(я\s+)?не\s+народ",  # "краще б не народився" / "краще
б я не народився(-лася)"
        r"немає\s+сенсу\s+(жити|в\s+житт)",
        r"не\s+хочу\s+(більше\s+)?жити",
        r"(хочу|думаю)\s+(заподіяти|нанести)\s+(собі\s+)?шкод",
        r"(хочу|планую)\s+(себе\s+)?вбити",
        r"покінч(ити|у)\s+з\s+(життям|собою)",
    ]

    def detect(self, text: str) -> bool:
        """
        Перевірити текст на наявність кризових маркерів

        Алгоритм:
        1. Нормалізувати текст (нижній регістр)
        2. Перевірити ключові слова
        3. Перевірити регулярні вирази

        ПРИМІТКА (виявлено під час тестування): фрази користувача рідко
        збігаються "слово в слово" зі словником ключових слів – наприклад,
        "краще б я не народився" не містить точної підрядки
        "краще б не народився" через вставлене "я". Тому базову перевірку
        ключових слів доповнено гнучкими регулярними виразами, що
        враховують можливі вставки займенників та варіації словоформ.
```

```
        Це знижує (але не усуває повністю) ризик пропущених спрацювань.

        Args:
            text: Текст для перевірки

        Returns:
            True якщо виявлено кризу, False інакше
        """
        text_lower = text.lower()

        # Перевірка ключових слів
        for keyword in self.CRISIS_KEYWORDS:
            if keyword in text_lower:
                return True

        # Перевірка регулярних виразів
        for pattern in self.CRISIS_PATTERNS:
            if re.search(pattern, text_lower):
                return True

        return False

    @staticmethod
    def get_crisis_response() -> str:
        """
        Повернути стандартну відповідь при виявленні кризи

        ПРИМІТКА: контакти служб підтримки уточнено під час підготовки
        бази знань системи. Лінія Lifeline Ukraine працює цілодобово,
        тоді як лінія Національної психологічної асоціації – щодня
        з 10:00 до 20:00. Перед розгортанням у production рекомендується
        звірити контакти з офіційним сайтом МОЗ України (moz.gov.ua).

        Returns:
            Текст з контактами служб підтримки
        """
        return """ Розумію, що тобі зараз дуже важко.

Будь ласка, зверніся за негайною допомогою:

 Лінія запобігання самогубствам Lifeline Ukraine:
   7333 (цілодобово, безкоштовно з мобільного)

 Національна психологічна асоціація:
   0 800 100 102 (щодня з 10:00 до 20:00, безкоштовно)

 Єдиний номер екстреної допомоги: 112

Професіонали допоможуть краще, ніж я. Ти не один/одна. """



class KnowledgeBaseService:
    """
    Сервіс для роботи з базою знань (RAG – Retrieval-Augmented Generation)

    Attributes:
```

```
        vectorstore: Векторне сховище ChromaDB
    """

    def __init__(self, vectorstore):
        self._vectorstore = vectorstore

    def search(self, query: str, k: int = 3) -> List[dict]:
        """
        Пошук релевантних документів у базі знань

        Алгоритм векторного пошуку:
        1. Векторизувати запит (перетворити в числовий вектор)
        2. Порівняти з векторами в базі (cosine similarity)
        3. Повернути топ-k найсхожіших

        Args:
            query: Запит користувача
            k: Кількість результатів

        Returns:
            Список документів з метаданими
        """
        if not self._vectorstore:
            return []

        results = self._vectorstore.similarity_search(query, k=k)

        # Форматувати результати
        formatted_results = []
        for doc in results:
            formatted_results.append({
                "content": doc.page_content,
                "source": doc.metadata.get("source", "Unknown"),
                "page": doc.metadata.get("page", "N/A")
            })

        return formatted_results
```

Лістинг А.4 – Повний текст файлу scripts/create_knowledge_base.py

```
"""
Скрипт для створення бази знань PsychHelper
Завантажує документи, векторизує їх та зберігає в ChromaDB
"""

import sys
import os

# Додати бекенд до шляху для імпорту
backend_path = os.path.join(os.path.dirname(__file__), '..', 'backend')
sys.path.insert(0, backend_path)

from langchain.document_loaders import (
    DirectoryLoader,
    PyPDFLoader,
```

```
    Docx2txtLoader,
    TextLoader
)
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OllamaEmbeddings
from langchain.vectorstores import Chroma
from dotenv import load_dotenv

# Завантажити змінні оточення
env_path = os.path.join(backend_path, '.env')
load_dotenv(env_path)



def load_documents(knowledge_base_path):
    """
    Завантажити всі документи з папки knowledge_base

    Підтримувані формати: PDF, DOCX, TXT

    Args:
        knowledge_base_path: Шлях до папки з документами

    Returns:
        Список завантажених документів
    """
    print("1□□  Завантаження документів...\n")

    documents = []

    # Завантажити PDF
    print("   □ Завантаження PDF...")
    pdf_loader = DirectoryLoader(
        knowledge_base_path,
        glob="**/*.pdf",
        loader_cls=PyPDFLoader,
        show_progress=True
    )
    documents.extend(pdf_loader.load())

    # Завантажити DOCX
    print("   □ Завантаження DOCX...")
    docx_loader = DirectoryLoader(
        knowledge_base_path,
        glob="**/*.docx",
        loader_cls=Docx2txtLoader,
        show_progress=True
    )
    documents.extend(docx_loader.load())

    # Завантажити TXT
    print("   □ Завантаження TXT...")
    txt_loader = DirectoryLoader(
        knowledge_base_path,
        glob="**/*.txt",
        loader_cls=TextLoader,
        loader_kwargs={"encoding": "utf-8"},
        show_progress=True
    )
    documents.extend(txt_loader.load())
```

```
    print(f"\n   □ Завантажено {len(documents)} документів\n")
    return documents


def split_documents(documents):
    """
    Розбити документи на фрагменти (chunks)

    Алгоритм:
    - Розмір фрагмента: 1000 символів
    - Перекриття: 200 символів (для збереження контексту)

    Args:
        documents: Список документів

    Returns:
        Список фрагментів
    """
    print("2□□  Розбиття на фрагменти...\n")

    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=200,
        length_function=len,
        separators=["\n\n", "\n", ". ", " ", ""]
    )

    chunks = text_splitter.split_documents(documents)

    print(f"   □ Створено {len(chunks)} фрагментів\n")
    return chunks


def create_vectorstore(chunks, chroma_path):
    """
    Створити векторне сховище ChromaDB

    Процес:
    1. Ініціалізувати embedding модель (nomic-embed-text)
    2. Векторизувати всі фрагменти
    3. Зберегти у ChromaDB

    Args:
        chunks: Список фрагментів тексту
        chroma_path: Шлях для збереження бази
    """
    print("3□□  Векторизація (це може зайняти 5-15 хвилин)...\n")

    # Ініціалізація embedding моделі
    embeddings = OllamaEmbeddings(
        model=os.getenv("EMBEDDING_MODEL", "nomic-embed-text"),
        base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434")
    )

    print("4□□  Збереження в ChromaDB...\n")
```

```
    # Створення векторного сховища
    vectorstore = Chroma.from_documents(
        documents=chunks,
        embedding=embeddings,
        persist_directory=chroma_path,
        collection_name="psychology_kb"
    )

    # Зберегти на диск
    vectorstore.persist()

    print(f"   □ База збережена у {chroma_path}\n")


def main():
    """Головна функція"""

    print("\n" + "="*60)
    print("□ Створення бази знань PsychHelper")
    print("="*60 + "\n")

    # Шляхи
    script_dir = os.path.dirname(os.path.abspath(__file__))
    kb_path = os.path.join(script_dir, '..', 'knowledge_base')
    chroma_path = os.path.join(script_dir, '..', 'backend', 'chroma_db')

    # Перевірити чи існує папка knowledge_base
    if not os.path.exists(kb_path):
        print(f"□ Папка {kb_path} не знайдена!")
        print("□ Створіть папку та додайте документи")
        return

    # Перевірити чи є файли
    has_files = False
    for ext in ['*.pdf', '*.docx', '*.txt']:
        if any(os.path.isfile(os.path.join(root, f))
               for root, _, files in os.walk(kb_path)
               for f in files if f.endswith(ext[1:])):
            has_files = True
            break

    if not has_files:
        print(f"□ У папці {kb_path} немає файлів!")
        print("□ Додайте PDF, DOCX або TXT файли")
        return

    try:
        # Крок 1: Завантаження
        documents = load_documents(kb_path)

        if not documents:
            print("□ Не вдалось завантажити жодного документа")
            return

        # Крок 2: Розбиття
        chunks = split_documents(documents)
```

```
        # Крок 3-4: Векторизація та збереження
        create_vectorstore(chunks, chroma_path)

        print("="*60)
        print("✅ База знань створена успішно!")
        print(f"📁 Локація: {chroma_path}")
        print(f"📊 Документів: {len(documents)}")
        print(f"📊 Фрагментів: {len(chunks)}")
        print("="*60 + "\n")

    except Exception as e:
        print(f"\n❌ Помилка: {e}\n")
        import traceback
        traceback.print_exc()


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

Лістинг А.5 – Повний текст файлу frontend/script.js

```
/**
 * Фронтенд логіка PsychHelper
 * Обробка взаємодії з користувачем та API
 */

// Константи
const API_URL = 'http://127.0.0.1:8000';

// DOM елементи
const messagesDiv = document.getElementById('messages');
const messageInput = document.getElementById('messageInput');
const sendBtn = document.getElementById('sendBtn');
const typingIndicator = document.getElementById('typingIndicator');
const statusDot = document.getElementById('status');

/**
 * Перевірка підключення до API при завантаженні сторінки
 */
async function checkConnection() {
    try {
        const response = await fetch(`${API_URL}/`);
        if (response.ok) {
            statusDot.style.background = '#4ade80'; // зелений
            console.log('✅ Підключено до API');
        }
    } catch (error) {
        statusDot.style.background = '#ef4444'; // червоний
        console.error('❌ Не вдалось підключитись до API');

        // Показати повідомлення про помилку
        addMessage(
            '❌ Не вдалось підключитись до сервера. Переконайтесь, що бекенд
запущено.',
            'assistant'
        );
```

```
    }
}

/**
 * Додати повідомлення в чат
 *
 * @param {string} text - Текст повідомлення
 * @param {string} type - Тип ('user' або 'assistant')
 * @param {boolean} isCrisis - Чи це кризове повідомлення
 */
function addMessage(text, type = 'assistant', isCrisis = false) {
    const msgDiv = document.createElement('div');
    msgDiv.className = `message ${type}${isCrisis ? ' crisis' : ''}`;
    msgDiv.textContent = text;

    // Вставити перед індикатором набору
    messagesDiv.appendChild(msgDiv);

    // Автоматичне прокручування вниз
    messagesDiv.scrollTop = messagesDiv.scrollHeight;
}

/**
 * Показати індикатор набору
 */
function showTyping() {
    typingIndicator.classList.add('active');
    messagesDiv.scrollTop = messagesDiv.scrollHeight;
}чому
/**
 * Сховати індикатор набору
 */
function hideTyping() {
    typingIndicator.classList.remove('active');
}
/**
 * Відправити повідомлення на сервер
 *
 * Алгоритм:
 * 1. Отримати текст з поля вводу
 * 2. Показати повідомлення користувача
 * 3. Відправити запит на API
 * 4. Показати відповідь асистента
 */
async function sendMessage() {
    const message = messageInput.value.trim();
    // Перевірка на порожнє повідомлення
    if (!message) return;
    // Показати повідомлення користувача
    addMessage(message, 'user');
    // Очистити поле вводу
    messageInput.value = '';
    // Заблокувати кнопку
    sendBtn.disabled = true;
    // Показати індикатор набору
    showTyping();
    try {
        // Відправити запит на API
        const response = await fetch(`${API_URL}/chat`, {
            method: 'POST',
            headers: {
```

```
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                message: message,
                session_id: 'web-session'
            })
        });
        // Сховати індикатор набору
        hideTyping();
        // Перевірка статусу відповіді
        if (!response.ok) {
            throw new Error(`HTTP ${response.status}`);
        // Отримати JSON відповідь
        const data = await response.json();
        // Показати відповідь асистента
        addMessage(data.response, 'assistant', data.is_crisis);

        // Якщо є джерела - можна їх вивести (опціонально)
        if (data.sources && data.sources.length > 0) {
            console.log('Джерела:', data.sources);
        }
    } catch (error) {
        hideTyping();
        // Показати повідомлення про помилку
        addMessage(
            '□ Помилка з\'єднання. Перевірте чи запущено бекенд-сервер.',
            'assistant'
        );
        console.error('Помилка:', error);
    } finally {
        // Розблокувати кнопку
        sendBtn.disabled = false;
        // Повернути фокус на поле вводу
        messageInput.focus();
    }
}
/**
 * Обробник кліку на кнопку "Надіслати"
 */
sendBtn.addEventListener('click', sendMessage);

/**
 * Обробник натискання клавіші Enter в полі вводу
 */
messageInput.addEventListener('keypress', (e) => {
    if (e.key === 'Enter' && !e.shiftKey) {
        e.preventDefault();
        sendMessage();
    }
});
/**
 * Ініціалізація при завантаженні сторінки
 */
window.addEventListener('load', () => {
    // Перевірити підключення
    checkConnection();
    // Встановити фокус на поле вводу
    messageInput.focus();
});
```