МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В.Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки та інформаційні технології в бізнесі**

Група: АК-41б **денна форма навчання**

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

на тему:

РОЗРОБКА МОБІЛЬНОГО РОЗУМНОГО АСИСТЕНТУ НА БАЗІ ФРЕЙМВОРКУ RASA

ЗА НАКАЗОМ № 4601-5/335 ВІД 07 ЛЮТОГО 2025 РОКУ

здобувача вищої освіти **Сусідка Давида Юрійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ №13 від 31.05.2025р.

Завідувач кафедри ІТММ
к.п.н., доцент
_____ **Н.І. Стяглик**

Науковий керівник
к. т. н.
_____ **О.В. Соболєв**

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки та інформаційні технології в бізнесі

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Н. І. Стяглик**
Підпис ініціали, прізвище

"08" лютого 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Сусідка Давида Юрійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи «Розробка мобільного розумного асистенту на базі фреймворку Rasa»

керівник роботи Соболєв Олександр Вікторович, к.т.н
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "08" лютого 2025 року № 4601-5/335

2. Строк подання студентом роботи 15 травня 2025 року

3. Перелік питань, які потрібно розробити:
У розділі 1:
Дізнатися про поняття та еволюцію розумних асистентів, класифікувати розумних помічників, проаналізувати сучасні підходи до їх розробки, розглянути фреймворк Rasa як інструмент для створення діалогового розумного асистента, та познайомитись із методологією дослідження та аналізу застосованих технологій.
У розділі 2:
Проаналізувати сучасні приклади інтелектуальних помічників, обґрунтувати вибір клієнт-серверної архітектури, порівняти варіанти комунікації між мобільним додатком та сервером, розглянути інструменти розробки Xcode, Swift та SwiftUI, визначити принцип взаємодії Rasa із ChatGPT за допомогою кастомних екшенів.
У розділі 3:
Розробити розумного помічника як мобільний додаток, описати його функціональні можливості, детально розглянути побудовану архітектуру серверної частини асистента, налаштувати веб-сервіси за допомогою Render, натренувати модель Rasa для подальшої роботи, протестувати готовий застосунок, визначити його сильні та слабкі сторони, порівняти готовий результат із бажаним.

4. План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної бакалаврської роботи
2	Затвердження плану і завдання кваліфікаційної бакалаврської роботи
3	Здача кваліфікаційної бакалаврської роботи керівнику
4	Підпис кваліфікаційної бакалаврської роботи керівника
5	Підпис кваліфікаційної бакалаврської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної бакалаврської роботи
7	Захист кваліфікаційної бакалаврської роботи

5. Дата видачі завдання 08 лютого 2025 року

Студент _____ Сусідко Д.Ю.
підпис ініціали, прізвище

Керівник роботи _____ Соболєв О.В.
підпис ініціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ
«РОЗРОБКА МОБІЛЬНОГО РОЗУМНОГО АСИСТЕНТУ НА БАЗІ ФРЕЙМВОРКУ RASA»
Сусідка Давида Юрійовича

Кваліфікаційна бакалаврська робота містить 56 сторінок, 2 таблиці, 12 рисунків, список літератури з 30 найменувань.

Об'єктом дослідження є концепція системи інтелектуального розумного мобільного асистента з використанням обробки природної мови.

Предметом дослідження є технології розробки мобільного інтелектуального помічника з використанням фреймворку Rasa, SwiftUI та OpenAI API.

Мета кваліфікаційної бакалаврської роботи полягає у створенні повноцінного власного рішення мобільного розумного асистента, використовуючи безкоштовний та гнучкий до налаштувань фреймворк Rasa та безкоштовний фреймворк для створення мобільних додатків SwiftUI у середовищі розробки Xcode.

Завданнями кваліфікаційної бакалаврської роботи є:

- познайомитись із сучасними рішеннями розумних асистентів та проаналізувати їх сильні та слабкі сторони.
- детально розглянути безкоштовний фреймворк Rasa та визначити його можливості з технічної точки зору.
- розробити архітектуру помічника за принципом клієнт-серверної взаємодії.
- створити свій власний мобільний додаток, використовуючи розроблену серверну частину асистента на базі фреймворку Rasa з використанням ChatGPT.
- протестувати готовий додаток, виправити критичні помилки та максимально наблизити його до мінімально бажаного результату.

Актуальність дослідження:

Оскільки сучасні рішення розумних асистентів з використанням штучного інтелекту давно стали невід'ємною частиною будь-кого, розробка власного гнучкого до кастомізації рішення, враховуючи локальні та глобальні потреби користувачів, є дуже актуальною темою на сьогоднішній день.

За результатами дослідження:

Було визначено найоптимальніший та найефективніший спосіб створити свого власного розумного помічника, та, власне, був розроблений прототип інтелектуального мобільного асистента на базі фреймворка Rasa з використанням мовної моделі штучного інтелекту OpenAI ChatGPT.

Практична новизна:

Створено унікальний мобільний застосунок, який поєднує у собі цікаву мову програмування Swift та фреймворк SwiftUI, технічні можливості Rasa та незвичну інтеграцію OpenAI ChatGPT.

Одержані результати можуть бути використані у навчальних та комерційних проектах, пов’язаних у впровадженні мобільних інтелектуальних рішень, а також як основа для подальших досліджень в області NLP NLU.

КЛЮЧОВІ СЛОВА:

Мобільний асистент, Rasa, Swift, SwiftUI, Web-Socket, Render, API, інтент, сутність, GPT, клієнт-серверна архітектура.

ABSTRACT

AT QUALIFICATION BACHELOR WORK

«DEVELOPMENT OF A MOBILE INTELLIGENT ASSISTANT BASED ON THE RASA FRAMEWORK»

The bachelor's thesis contains 56 pages, 2 tables, 12 drawings, a list of references of 30 titles.

The object of the research is the concept of a mobile intelligent assistant system based on natural language processing.

The subject of the research is technologies for developing a mobile intelligent assistant using the Rasa framework, SwiftUI, and the OpenAI API.

The purpose of a bachelor's thesis is to create a complete, custom mobile assistant solution using the open and flexible Rasa framework and the free SwiftUI framework in the Xcode development environment.

The tasks of a bachelor's degree are:

- to explore modern intelligent assistant solutions and analyze their strengths and weaknesses;
- to examine the Rasa open-source framework and assess its capabilities from a technical perspective;
- to design the assistant's architecture based on the client-server interaction principle;
- to develop a mobile application using the Rasa-based backend integrated with ChatGPT;
- to test the final application, fix critical bugs and optimize it for the minimum viable result.

The relevance of the Intelligent assistants with artificial intelligence have become an integral part of digital life. Developing a customizable and locally adapted solution is highly relevant in today's context.

According to the results of the research:

the most efficient method for developing a custom intelligent assistant was identified. A prototype of a mobile intelligent assistant based on the Rasa framework and OpenAI ChatGPT was implemented.

Main theoretical provisions on the topic of the practical relevance of the research:

A unique mobile application was created that combines the Swift programming language and SwiftUI framework, the technical potential of Rasa, and an unconventional integration with OpenAI ChatGPT.

The results obtained can be used in educational or commercial projects related to mobile NLP-based solutions, as well as for further research in the field of NLP and NLU.

KEYWORDS:

Mobile assistant, Rasa, Swift, SwiftUI, WebSocket, Render, API, intent, entity, GPT, client-server architecture.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

AI – Artificial Intelligence (штучний інтелект) – галузь комп'ютерних наук, що займається створенням систем, здатних до навчання, логічного мислення і самостійного прийняття рішень.

API – Application Programming Interface (інтерфейс прикладного програмування) – набір правил, що дозволяє взаємодіяти між програмними компонентами.

GitHub – веб-сервіс для зберігання, керування та контролю версій коду, побудований на основі системи Git.

IDE – Integrated Development Environment (інтегроване середовище розробки) – програмне забезпечення для розробки, тестування та налагодження коду.

LLM – Large Language Model (велика мовна модель) – нейронна мережа, здатна обробляти природну мову на основі великих обсягів тексту.

ML – Machine Learning (машинне навчання) – підгалузь AI, яка вивчає методи навчання машин на основі даних.

NLP – Natural Language Processing (обробка природної мови) – галузь AI, що фокусується на взаємодії між комп'ютером та людською мовою.

NLU – Natural Language Understanding (розуміння природної мови) – складова NLP, яка відповідає за розпізнавання інтенцій і сутностей у введеному тексті.

NLG – Natural Language Generation (генерація природної мови) – компонент NLP, який формує відповіді у вигляді людської мови.

OS – Operating System (операційна система) – базове програмне забезпечення, що керує апаратними ресурсами комп'ютера або мобільного пристрою.

Rasa – фреймворк з відкритим кодом для створення розумних асистентів та чат-ботів з підтримкою NLU і Dialogue Management.

Rasa NLU – підсистема Rasa, що відповідає за розпізнавання намірів користувача та витяг сутностей.

Rasa Core – підсистема Rasa, що реалізує логіку діалогу, керує діалоговим потоком.

Rasa X – графічний інтерфейс для тестування, вдосконалення і тренування моделей Rasa.

Render – хмарна платформа для деплою веб-додатків і серверів.

REST API – архітектурний стиль обміну даними, що використовує протокол HTTP і дозволяє організовувати взаємодію між клієнтом і сервером.

Socket.IO – бібліотека JavaScript для двосторонньої комунікації між клієнтом і сервером у режимі реального часу.

Swift – сучасна мова програмування для iOS/macOS від Apple.

SwiftUI – фреймворк для створення інтерфейсів у iOS-додатках на мові Swift.

UI – User Interface (інтерфейс користувача) – зовнішній вигляд і спосіб взаємодії користувача з програмним забезпеченням.

UX – User Experience (досвід користувача) – загальні враження користувача від використання продукту або послуги.

WebSocket – протокол для встановлення постійного з’єднання між клієнтом і сервером з можливістю двосторонньої передачі даних у реальному часі.

ВСТУП

Історія людства багатогранна та несподівана. Та її частина що нам відома охоплює надвеликий проміжок часу, за який встигло статись дуже багато еволюційних змін, покращень, вдосконалень, якісних змін – від перших знань про спосіб добування вогню до створення повністю автоматизованих систем виробництва матеріальних благ. Однак сьогодення зовсім не відрізняється від всієї історії людства з точки зору розвитку. Здавалось би – ще 5 років тому ми навіть не здогадувались що зовсім скоро з'явиться те, що зможе повністю змінити життя кожного. З'явиться те, що стане більш невід'ємною частиною нашого світу. Мова йде про штучний інтелект. Саме він став однією з найперспективніших напрямів IT індустрії, адже створення такої комп'ютерної системи, яка буде здатна самостійно навчатись, адаптуватись та вирішувати будь-які поставлені задачі без втручання людини в цей процес – є основною ідеєю розвитку сучасних інформаційних технологій. Звідси ми маємо логічний висновок – тема створення розумних асистентів з використанням штучного інтелекту стає дедалі актуальнішою. Проте мова йде не стільки про штучний інтелект в цілому, скільки про фреймворк Rasa, який на мій погляд мало де використовується. І дарма – адже завдяки ньому будь-який розробник має можливість створити власного розумного мобільного асистента та спрощувати життя як собі, так і іншим користувачам мобільних пристроїв.

Я більш ніж переконаний в тому, що майже кожен хто користується сучасною технікою, і немає різниці чи то комп'ютер, телефон або планшет, знає або хоча б чув такі слова як Google Assistant, Siri, Alexa або щось подібне. Так, йдеться про розумних мобільних голосових та текстових асистентів. Ці помічники дуже добре виконують поставлені задачі, але вони мають суттєвий недолік – це комерційний продукт, який майже неможливо «кастомізувати» або дуже глибоко налаштувати під себе. Безкоштовний фреймворк Rasa дозволяє в свою чергу будь-якому бажаючому створити свого власного

розумного мобільного асистента з нуля, підлаштовуючи його під власні конкретні задачі а також використовуючи сучасні методи обробки природної мови та машинного навчання. Насправді дослідження в цій темі проводяться вже багато років, зокрема багато хто аналізував архітектуру діалогових систем та інтеграції їх з мобільними додатками. Проте досить часто бракує прикладів реалізації подібних додатків з використанням саме фреймворку Rasa в контексті мобільних платформ.

Саме тому метою моєї роботи є створення мобільного розумного асистенту, який буде базуватись на роботі фреймворку Rasa, який давав би змогу користувачу досить легко та швидко виконувати різні поставлені задачі з можливістю масштабування проекту в майбутньому.

Я вважаю, що виконання моєї мети не зможе обійтись одним чи двома кроками. Саме тому я би хотів розробити чітку стратегію дій та, власне, описати її наступним чином:

- ознайомитись із можливостями фреймворку Rasa та подібних технологій
- визначити архітектуру та принцип роботи мобільного асистента
- дослідити підходи до побудови діалогових систем
- зробити додаток, використовуючи власне розуміння принципу роботи сучасного зручного інтерфейсу
- знайти спосіб грамотно інтегрувати фреймворк Rasa у мій мобільний додаток та реалізувати знайдений підхід
- детально протестувати готове рішення розумного асистента та зрозуміти, які сильні та слабкі сторони він має та чи достатньо добре він виконує свою роботу

Об’єктом дослідження в цій роботі є процес побудови діалогових систем на базі фреймворку Rasa. Предметом дослідження є методи та технології реалізації мобільного асистенту з використанням обробки природної мови та штучного інтелекту.

У ході роботи використовувалися такі методи дослідження як вивчення та аналіз літературних джерел, порівняння наявних рішень, експериментальна розробка, проектування та реалізація архітектурного рішення, виявлення помилок, недоліків програмного продукту, що розроблено, глобальне тестування мобільного продукту та його покращення.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ МОБІЛЬНОГО РОЗУМНОГО АСИСТЕНТУ НА БАЗІ ФРЕЙМВОРКУ RASA

1.1. Що таке розумний асистент та які їх особливості

З розвитком сучасних комп'ютерних технологій дедалі популярнішим стають так звані «розумні» помічники – асистенти. Наразі вони існують майже у будь-якому пристрої, яким користується людина – в комп'ютері, телефоні, планшеті або, наприклад, в умовному холодильнику. Проте далеко не кожен користувач цього «дива техніки» знає що таке «розумний» асистент за своєю суттю. В дійсності, це програма або пристрій , яка здатна допомагати людині виконувати різноманітні задачі за допомогою тексту, голосу або навіть жестів. Серед найвідоміших звичайному користувачу помічників є такі як Siri, Google Assistant, Amazon Alexa і т.д. Вони можуть робити усе – від простої дії налаштувати для людини будильник або замовити їжу з доставкою на дім до розв'язування будь-яких математичних задач та аналізу графіків.

З технічної точки зору такий асистент є програмним забезпеченням, основа роботи якого є використання штучного інтелекту, а саме – обробку природної мови (NLP), машинне навчання або навіть синтез мовлення та розпізнавання зображень. Завдяки усім цим технологіям асистент здатен «зрозуміти» суть поставленої задачі, її контекст, проаналізувати вхідні дані та видати користувачу бажаний результат. Це може бути або звичайна відповідь на задане питання або виконання якоїсь дії, яку теоретично очікує користувач.

Існує багато різновидів розумних асистентів, тож їх можна умовно класифікувати за такими ознаками:

- за способом взаємодії: текстові асистенти(такі як чат-боти в Telegram або на сайтах) та голосові(як-от Alexa або Siri). Є також гібридний вид асистентів, які одночасно здатні і говорити, і писати текст.

- за платформою: асистенти можуть бути мобільними(тобто вбудованими в телефон як інтегрований фреймворк або додаток), веб-асистентом(які розміщені на сайті або інтегровані в браузер), десктопні (наприклад Cortana в операційній системі Microsoft Windows), або бути інтегрованими в окремі пристрої(такі як музикальна колонка, Smart-TV тощо.)
- за функціональністю: далеко не кожен розумний помічник має змогу допомогти людині в усіх питаннях будь-якої сфери діяльності. Тобто якщо, наприклад, якійсь лікарні потрібен інтелектуальний асистент, то немає сенсу створювати для неї такого помічника, який знається в усьому, але по трохи. У такому випадку варто прийти до рішення, де асистент буде максимально корисний у контексті медицини, і неважливо, наскільки він буде корисний у будь-якій іншій сфері діяльності людини.

Останнім часом я помічаю таку тенденцію, що подібні розумні асистенти здебільшого використовуються саме в мобільних пристроях, оскільки переважна більшість користувачів інтернету відвідує його саме з мобільних пристроїв, що робить мобільних розумних асистентів досить пріоритетними у розробці та розвитку. На мою думку, саме ці причини найяскравішим чином будують розуміння того, наскільки важливо розробляти свого власного розумного мобільного асистента.

Безумно до таких асистентів існують певні вимоги. Вони мають бути не просто ботами, які здатні виконувати конкретну поставлену задачу. Такі асистенти мають «вміти» та підтримувати діалог з користувачем, визначати та розуміти контекст бесіди, аналізувати потреби користувача, його манеру розмови та підлаштовувати свою поведінку саме під нього. Це насамперед створює дуже позитивний досвід роботи з асистентом та комфортні умови для подальшої взаємодії людини з ним. Окрім того, існує також вимога інтеграції таких асистентів в різноманітні сервіси – календар, пошта, погода, месенджери тощо. Це допоможе асистенту більше знати та розуміти, а значить набагато

швидше та чіткіше виконувати поставлені задачі, що в свою чергу значно покращить загальну працездатність та ефективність асистента.

1.2. Огляд компонентів сучасних інтелектуальних помічників

Для звичайного користувача розумний асистент виглядає як «чудо техніки». Більшість навіть не замислюється над тим, що насправді розумний асистент – це дуже потужний програмний продукт, в основі якого лежить досить складна «технічна начинка». Тому варто розуміти, що усі можливості таких асистентів – це не просто якась «магія». Це велика множина різноманітних логічних алгоритмів, чітких підходів розробки та реалізацій певних функцій, а також вибір правильних інструментів.

За моїми спостереженнями, будь-який розумний асистент складається з декількох основних компонентів. Оскільки більшість з таких помічників має схожу структуру, я би виділив серед основних компонентів такі:

- NLU (Natural Language Understanding) – це компонент, який потрібен розумному асистенту для того, щоб розуміти, що пише йому користувач. Наприклад, якщо користувач напише до асистента щось схоже на «Хочу замовити їжу», то в такому випадку асистент має зорієнтуватись та зрозуміти, що мова йде про замовлення їжі через інтернет доставкою на дім користувача. В цій ситуації асистент також може дізнатись додаткову інформацію, таку як де замовляти, яку саме їжу хоче замовити користувач, на котру годину і т.д.
- Dialogue Management (менеджер діалогу) – цей компонент допомагає асистенту продовжувати діалог з користувачем залежно від контексту [18, 21]. У контексті прикладу, який наведено в попередньому пункті, асистент має продовжити діалог, запитавши у користувача, наприклад, яку саме піцу він хоче замовити.
- NLG (Natural Language Generation) – це компонент, який допомагає асистенту видавати більш підходящі відповіді користувачу у даному

діалозі [12]. Інколи це може бути якась шаблонна, заздалегідь запрограмована відповідь, а іноді більш гнучке формування відповіді.

Сучасні розумні асистенти будуються з використанням різноманітних технологій залежно від того, з якими задачами матиме справу даний асистент. Серед основних підходів можна визначити наступні:

1) Чат-боти з правилами. Тут підхід дуже простий – достатньо лише прописати усі необхідні «якщо-то» правила, орієнтуючись на які асистент буде мати справу з користувачем. Цей підхід має суттєвий недолік – з ним майже неможливо створити справжнього розумного асистента, оскільки в такому випадку асистент вийде більше як алгоритм з «жорсткою» логікою. В якості яскравого прикладу такого можна навести авто-відповідача у Facebook Messenger.
2) Асистенти на базі NLP + ML. Цей підхід вже суттєво складніше в реалізації – асистент навчається на прикладах, здатен розуміти різноманітні фрази, витягувати дані, адаптувати свої відповіді [11, 22]. Саме такий підхід ми використовуватимемо разом із фреймворком Rasa у цій роботі.
3) Асистенти з генеративними моделями (LLM). Ні для кого не секрет, що сучасна тенденція буквально «крутиться» навколо повноцінних потужних мовних моделей, таких як ChatGPT або DeepSeek, які здатні генерувати максимально унікальну відповідь в поточному контексті прямо тут і зараз під час розмови з таким асистентом [13]. На практиці це виглядає майже як повноцінна розмова однієї людини і іншою, але існують декілька недоліків, таких як складність реалізації, публічна підтримка, проблеми з безпекою персональних даних, потреба в ресурсах та техніці тощо.

Вважаю за потрібне розглянути таблицю нижче, за допомогою якої ми розглянемо декілька популярних фреймворків штучного інтелекту та розглянемо їх основні позитивні та негативні сторони.

Таблиця 1.1

Порівняння популярних ШІ фреймворків

Фреймворк	Переваги	Недоліки
Rasa	Відкритий безкоштовний код, абсолютна кастомізація, підтримка ML	Потрібно набагато більше ручних налаштувань
DialogFlow (Google)	Проста інтеграція, інтуїтивний легкий UI, безпроблемний швидкий старт	Обмежена кастомізація, строга залежність від Google
Microsoft Bot Framework	Потужна інфраструктура, глибока інтеграція з Azure	Занадто складна екосистема, особливо для початківців
Wit.ai (Facebook)	Простий, безкоштовний, легка інтеграція з Messenger	Не гнучкий, обмежений набір функцій

У нашому випадку саме фреймворк Rasa був обраний для розробки свого власного мобільного розумного асистенту, оскільки нам потрібно мати повний та чіткий контроль над асистентом та абсолютну відсутність залежностей від будь-яких сервісів. Я погоджусь із думкою, що такий вибір змусить мене отримати досить велику кількість незручних моментів під час розробки, однак я впевнений, що це буде досить цікаво, корисно та круто!

Сучасні тенденції вимагають від розумних асистентів значно більшого ніж «зробити, аби працювало». Підвищення комфорту взаємодії з асистентом, його здатність пам’ятати контекст і усе те що було до цього, адаптувати свою поведінку під даного користувача, інтеграція асистента у різноманітні

мобільні додатки та смарт-пристрої – усе це безумовна базова складова сучасного потужного розумного асистента. Окрім того, варто не забувати про безпеку та етичність – асистент ні в якому випадку не може мати можливості казати щось неприємне, образливе для користувача, а також не може мати можливості порушувати приватність користувача або зберігати якусь приватну інформацію, яку асистенту точно не слід пам’ятати та зберігати десь у себе в «пам’яті».

1.3. Методологія дослідження

Для того щоб створити свого власного розумного мобільного асистента, недостатньо просто взяти, знайти фреймворк та почати щось налаштовувати та щось кодувати. Для початку, необхідно визначити план дій, що саме ми хочемо створити, які ресурси для цього нам необхідні, які технології ми використовуватимемо, яким чином буде проходити етап тестування для того, щоб виправити усі поточні проблеми та «баги» та вдосконалити готовий додаток. Така сукупність дій дозволить нам ефективно почати розробку, і не менш ефективно її закінчити.

Головна мета цієї роботи – розробити мобільного розумного асистента на базі фреймворку Rasa, який міг би вести діалог з користувачем, розуміти мову людини, і при цьому легко інтегруватись у мобільний додаток. Також метою роботи було створення дійсно корисного та практичного асистента яким можна користуватись у повсякденному житті.

Для досягнення нашої мети необхідно виконати певний список дій, а саме:

- дослідити теоретичні основи розумних асистентів та їхню класифікацію;
- проаналізувати поточні підходи реалізації подібних систем;
- ознайомитися з можливостями фреймворку Rasa і обґрунтувати вибір саме його;

- створити власного асистента з базовим набором функцій;
- забезпечити його інтеграцію у застосунок за допомогою Web Requests;
- оцінити ефективність обраної архітектури створеного помічника, його практичність, зручність, гнучкість.

У цій роботі я використовую декілька різних методів, які допоможуть мені опанувати як теоретичні основи, так і реалізувати практичну складову. Серед цих методів я виділю наступні: (по зразку попереднього списку)

- Аналіз літературних джерел – тобто я вивчаю, що пишуть люди у книжках, форумах стосовно теми штучного інтелекту, розумних асистентів, фреймворку Rasa, середовищ розробки (IDE), NLP, NLU, машинного навчання тощо.
- Порівняльний аналіз – я розглядаю різноманітні фреймворки штучного інтелекту, їхні технології та підходи до реалізації і оцінюю, що з цього найкращим чином підходить для реалізації поставленої мети.
- Проектування системи – визначення та створення архітектури майбутнього асистента – починаючи від Back-end складової закінчуючи Front-end складовою, тобто безпосередньо розробкою інтерфейсу мобільного додатку.
- Прототипування та розробка – написання коду додатку, налаштування моделі Rasa, її запуск та розгортання, створення кастомних інтентів та дій, тестування функціональності моделі та додатку.
- Тестування та валідація – процес, необхідний для того, щоб переконатися в тому, що створений асистент дійсно працює, відповідає головній меті роботи та коректно виконує різноманітні поставлені йому задачі.

Виникає головне питання – чому я обрав саме фреймворк Rasa? У дійсності, відповідь на це питання досить легке. Я вибрав саме цей фреймворк

перш за все тому, що ми маємо справу з відкритим кодом – тобто я можу змінювати логіку роботи фреймворку та підлаштовувати усе під себе. По-друге, Rasa чітко ділиться на компоненти, що відповідають за розпізнавання мови та компоненти, що відповідають за ведення діалогу з користувачем. Це дозволяє мені будувати дійсно розумну та гнучку логіку спілкування асистента з майбутнім користувачем. По-третє, Rasa не потребує якоїсь важкої інтеграції у проект – достатньо розвернути її натреновану модель локально або на сервері, та спілкуватись з нею за допомогою POST Web Requests. Окрім того, модель Rasa можна розгорнути і запустити локально, що дає повну свободу дій для власних кастомних рішень.

Таким чином моя дипломна робота базується на поєднанні аналізу, проектування, розробки та тестування – з упором на те, щоб створити не просто звичайного бота, а дійсно корисного та практичного розумного асистента, який зможе спілкуватися з користувачем в досить зручному та інтуїтивному форматі, допомагати вирішувати різноманітні задачі і просто бути гарним помічником людині.

1.4. Детальний огляд фреймворку Rasa

Оскільки ми визначились із вибором фреймворку, тепер слід більш детально його розглянути, зрозуміти що це таке, як він працює і як нам працювати з ним. Отже, Rasa – це Python-базований фреймворк, за допомогою якого ми маємо можливість створювати своїх власних розумних асистентів, які розуміють людську мову, запам’ятовують контекст розмови, оброблюють інформацію та видають адаптовану відповідь користувачу. Цей фреймворк складається з двох основних компонентів:

- Rasa NLU (Natural Language Understanding) – компонент, який відповідає за розуміння людської мови та того, що написав користувач нашому асистенту. Він визначає інтенції(тобто наміри), та витягує сутності(наприклад дати, імена, місця тощо).

- Rasa Core (Dialogue Management) – це ніби «мозок» нашого асистента. Він приймає рішення щодо того, яким чином реагувати на те, що написав користувач, враховуючи контекст, попередню розмову та ті сценарії, які ми самостійно визначимо.
- Rasa X – досить зручний інтерфейс для тестування та вдосконалення асистента. Там можна тренувати саму модель, переглядати приклади діалогів та налаштовувати інтенції без взаємодії з кодом.

Виникає логічне питання – а яким чином працює асистент на базі фреймворку Rasa? Насправді так званий «життєвий цикл» такого асистента виглядатиме приблизно наступним чином:

- Користувач надсилає повідомлення асистенту. Наприклад це буде «хочу замовити піцу».
- Rasa NLU аналізує отримане повідомлення та розпізнає інтенцію «замовлення_їжі» і сутність «піца».
- Rasa Core аналізує історію діалогу з цим користувачем та вирішує, що робити далі. Наприклад, уточнити розмір або адресу доставки
- Асистент дає відповідь або викликає певну дію.

Увесь цей процес легко кастомізується та налаштовується саме так, як того ми забажаємо. Для реалізації Back-end складової нашого застосунку на практиці нам необхідно буде виконати наступні дії:

- Завантажити фреймворк локально на комп’ютер.
- Налаштувати конфігураційні файли для тренування моделі Rasa.
- За допомогою терміналу виконати команди для тренування моделі.
- Отриману модель розгорнути за запустити на віддаленому сервері, до якого ми матимемо доступ через посилання.

Оскільки Rasa не є мовною моделлю, ми можемо стикнутись з деякими труднощами щодо очікуваних результатів. Для вирішення цих труднощів ми вдосконалимо логіку Back-end складової за допомогою інтеграції OpenAI ChatGPT в процес обробки інформації. Інакше кажучи, якщо Rasa не зрозуміє те, що написав користувач, то в такому випадку ми налаштуємо її поведінку

таким чином, щоб вона надсилала запит користувача до ChatGPT, отримувала від нього повідомлення та передавала користувачу. Таким чином ми досягнемо максимальної гнучкості в логіці нашого розумного асистента, і він буде адаптований під будь-які інтенти, що надає користувач.

РОЗДІЛ 2
АНАЛІЗ СУЧАСНИХ ПІДХОДІВ РОЗРОБКИ МОБІЛЬНИХ РОЗУМНИХ АСИСТЕНТІВ

2.1. Обговорення сучасного стану розвитку інтелектуальних помічників в Україні та світі

Інтелектуальні асистенти як технологічне явище вже досить давно перестало бути чимось особливим у колі IT-спеціалістів та «євангелістів» технологій. Сьогодні це явище максимально поширене по всьому світу. Розумні асистенти активно інтегруються в будь-яку техніку, операційні системи, віртуальні еко-системи, хмарні сервіси тощо. Відповідно і попит на цих асистентів досить суттєвий – майже кожна людина, яка має доступ до смартфона та інтернету(іноді опціонально) використовує якщо не мінімум, то точно суттєвий об’єм функцій та можливостей такого розумного асистенту. І основна ідея в тому, що ця технологія дедалі стрімко розвивається, подібні асистенти стають більш розумними, вправними, краще починають розуміти людську мову, контекст, а також вони стають більш етичними та «дружніми» до користувача.

Якщо ми розглянемо питання розвитку мобільних асистентів на більш глобальному рівні, то можемо дуже швидко і легко переконатись в одному – темп розвитку цих асистентів можна назвати буквально шаленим. Адже за даними міжнародних аналітичних компаній кількість активних користувачів розумних асистентів до 2026 року може перевищити відмітку в 8 мільярдів осіб. Тобто це означає, що приблизно до того часу в середньому майже на кожну першу людину припадатиме хоча б один такий розумний асистент.

Я досить часто користуюсь мобільними пристроями на базі операційної системи як Android, так і iOS, і найчастіше я бачу, чую та користуюсь такими розумними асистентами:

- Google Assistant – розумний асистент, який інтегровано у мільйони мобільних пристроїв на базі операційної системи Android та Google Home. Основна особливість цього асистенту – тісна взаємодія з екосистемою Google. Цей асистент легко, швидко та досить вправно працює з такими сервісами як календар, пошук, пошта, мапи тощо.

- Apple Siri – найпопулярніший асистент у світі користувачів техніки Apple. Його вважають однією із перших спроб створити масового розумного помічника. Досить зручно інтегрований в усю екосистему Apple, підтримується на iPhone, Mac, має досить зручний голосовий інтерфейс і достатньо високу якість розпізнавання людської мови, що робить його дуже практичним та корисним у повсякденному використанні як власного «співбесідника».

- Amazon Alexa – досить популярна модель штучного інтелекту, особливо у США. Її можна підключити до розумного будинку, налаштувати керування технікою, створювати нагадування, включати музику і навіть робити замовлення в Amazon. Прекрасний варіант для виконання звичайних життєвих задач разом з приємним шопінгом.

- Samsung Bixby – розумний помічник, який є розробкою компанії Samsung. Він має досить розширений набір функцій, а особливо сильно розкриває свої позитивні сторони коли йдеться про його можливості, пов’язані з функціями поточного пристрою.

Як ми бачимо, подібні сучасні розумні асистенти встигли зробити багато кроків вперед від звичайної текстової відповіді на поставлене питання чи просто запит. Вони розвиваються в більш глобальному напрямку: виконання будь-яких реальних дії починаючи від увімкнення світла в домі до бронювання квитків на готель чи літак. Усі ці системи активно використовують машинне навчання, штучний інтелект, обробку природної мови та синтез мовлення для досягнення максимально наближеного результату до бажаного.

Варто зауважити, що сучасна тенденція розвитку розумних асистентів концентрується саме на інтеграції їх в мобільні пристрої. І це дивно, адже саме

смартфон є найбільш часто вживаною технікою будь-якої людини. Вони з нами завжди поруч, вони компактні, легкі, зручні, дуже потужні та здатні виконувати майже увесь спектр потрібних користувачу цілей, таких як замовлення таксі або їжі, прослуховування музики, моніторинг новин, менеджмент повідомлень та нагадувань тощо. Саме тому усі найбільш крупні технологічні компанії (так звані техно-гіганти) інтегрують своїх розумних асистентів саме у мобільні пристрої. При цьому найбільш корисним для користувача є надзвичайна зручність – при купівлі нового смартфона йому не потрібно нічого встановлювати або налаштовувати, адже потужний розумний асистент вже існує, і він вже готовий до плідної співпраці.

Така легка доступність супроводжує відсутність потреби користувача у зацікавленості в тому, яким саме чином працює розумний асистент «під капотом». Достатньо сказати «окей Google», і асистент миттєво приготувався допомагати в будь-яких справах та питаннях. І ця простота взаємодії з асистентом – ключова причина успіху розвитку цієї технології.

Однак виникає найголовніше питання: то ж як зараз виглядає ситуація з розвитком розумних асистентів в Україні? Насправді, до недавнього часу ситуація була не настільки яскравою, оскільки основною проблемою стала локалізація – найбільш поширені по всьому світу розумні асистенти просто не підтримували українську мову. Це стало причиною використання українцями асистентів російською або англійською мовами. Проте ситуація почала змінюватись. Наприклад Google Assistant вже частково підтримує українську мову, хоча поки що без офіційного запуску. Операційна система Android OS повноцінно підтримує українську локалізацію, хоча сам асистент досить часто дає відповідь англійською. Це ж стосується і Apple Siri – вона вже підтримує українську мову на базовому рівні, через що її функціонал поки що обмежений. Однак розвиток розумних асистентів в Україні не зупиняється на використанні зарубіжних технологічних рішень. Наприклад популярні українські банки, такі як Monobank та Privat24, створюють своїх власних асистентів чат-ботів, які частково використовують функції розумних

асистентів. Вони вміють вести бесіду з користувачем, підказувати йому та допомагати з проведенням різноманітних операцій. Також варто відмітити Кейс «Дія», який хоч і не є повноцінним розумним асистентом в класичному розумінні цього слова, проте має певні функції автоматизованої взаємодії з користувачем. Насправді з точки зору перспективи туди можна інтегрувати додаткові елементи розумного помічника, що суттєво зможе комплексно вдосконалити цю технологію та досить сильно наблизити її до повноцінного інтелектуального асистента.

Варто пам’ятати про досить важливий показник, який стосується інформаційної безпеки, приватності та контролю даних користувачів. У багатьох випадках користувачі або організації не бажають надавати якусь конфіденційну або персональну інформацію стороннім сервісам. У таких ситуаціях на допомогу приходять відкриті платформи, які можна розгорнути локально або на віддаленому довіреному сервері(наприклад, Rasa), які мають очевидні переваги. І однією з таких є абсолютний контроль процесу обробки інформації і даних та адаптація системи до власних потреб без створення зовнішньої залежності.

Однією з досить суттєвих проблем на шляху розвитку розумних асистентів є сам користувач. Далеко не всі активно користуються мобільними асистентами, і ось чому:

- досить часта недовіра людини до асистента, страх «підслуховування».
- елементарне нерозуміння функцій
- слабка локалізація, особливо мовна
- просте «я не знаю, навіщо він мені»

Це і є одна з найголовніших проблем розвитку розумних асистентів, і щоб її вирішити, необхідно розробляти розумних асистентів таким чином, щоб вони були максимально зрозумілими, простими у використанні та корисними у виконанні реальних задач. Тут і підключається базова психологія людини –

якщо асистент працює, і він дійсно виконує задачу, і результат користувачу подобається, тоді людина буде цим користуватись.

Отже, сучасна тенденція розвитку розумних асистентів концентрується по всьому світу в більшості галузей діяльності людини. Існують як великі технологічні лідери ринку, які пропонують надпотужні рішення розумних асистентів, так і малі компанії або «IT-ентузіасти», які прагнуть зробити свій власний вклад в розвиток цієї технології, запропонувати свою інтерпретацію або ж винайти щось зовсім нове, більше корисне та більш потужне. І найголовнішим середовищем розвитку є саме мобільні пристрої. Український розвиток розумних асистентів дійсно не настільки стрімкий в порівнянні зі світом, проте має досить великий попит та багатообіцяючі перспективи розвитку. Достатньо розвинути поточні власні рішення, вирішити питання української локалізації і цілком можливо Україна стане на світові рейки лідерів розвитку мобільних інтелектуальних асистентів.

2.2. Огляд та детальний аналіз сучасних розумних помічників

Як уже згадувалось раніше, більшість користувачів сучасних технологій, які мають інтегрованого розумного асистента, не замислюються над тим, яким саме чином працює цей самий асистент. Для користувача важлива зручність, швидкість та якісний отриманий результат. Проте досить важливо все ж таки більше заглибитись у принцип роботи таких інтелектуальних помічників, проаналізувати існуючі рішення на світовому ринку, визначити їх фунціонал, сильні та слабкі сторони. До того ж подібний аналіз послужить нам відмінним джерелом натхнення під час власних розробок, адже детально розглянувши існуючі найпоширеніші приклади реалізації розумних асистентів можна у своїх використати щось позитивне з цих рішень і при цьому відмовлятись від негативного. Таким чином, пропоную проаналізувати більш детально найпотужніші світові рішення і створити на

основі цього відповідну порівняльну таблицю, яка зможе чітко та коротко підсумувати наш аналіз.

І першим прикладом у нас буде Google Assistant – один із найпотужніших розумних асистентів на сьогоднішній день. Він інтегрований майже у кожний Android смартфон і вже давно став невід'ємною частиною повсякденного користувацького досвіду. Серед його функцій основними вважатимемо наступні:

- вміння виконувати команди(увімкнення музики, телефонування, налаштування будильника).
- вміння давати відповідь на поставлене запитання, пошук інформації в мережі інтернет.
- тісна інтеграція з Google сервісами, через що присутність вміння працювати з календарем, Google Maps, Gmail тощо.
- вміння керувати розумним будинком(Google Home).
- праця в бесіді з контекстом, здатність пам'ятати що було раніше, здатність продовжувати розмову на основі поточного контексту.

Серед позитивних сторін цього асистенту варто виділити досить високу якість розпізнавання людської мови, сильну та зручну інтеграцію в сервіси Google а також достатньо широкий фунціонал, який створює приємний досвід співпраці з цим асистентом та надає дуже багато можливостей для підвищення своєї продуктивності впродовж дня. Однак без негативних сторін не обійшлося. Серед таких варто виділити дуже обмежені можливості кастомізації під окремого користувача або навіть цілий бізнес, що як правило унеможливлює забезпечення максимальної ефективності та реалізацію власних потреб та цілей. Також варто пам'ятати, що такий асистент не може працювати без інтернет підключення. Цей недолік дуже сильно заважає, адже незважаючи на те що зараз більшість має в своєму домі інтернет, досить велика кількість людей все ще або не мають його взагалі, або мають тільки вдома. Проте найбільшою негативною стороною я вважаю відсутність української локалізації, а точніше часткову її підтримку, і навіть не на офіційному рівні.

Наступним не менш цікавим прикладом я би хотів розглянути такого розумного асистента як Replika. Replika – це інтелектуальний помічник, робота якого зосереджена не стільки на виконанні задач скільки на емоційній взаємодії з користувачем. Цей бот, який створений на основі нейромереж, буквально «вчиться» бути твоїм другом. Він доступний в якості мобільного застосунку і має мільйони завантажень по всьому світу, що робить його досить популярним та затребуваним у суспільстві. Серед його основних функцій можна навести наступні:

- ведення діалогів з користувачем у форматі чату.
- емоційне реагування, «підтримка» розмови, здатність пам’ятати події.
- має певну особистість, яку можна налаштовувати.
- доступність на операційних системах Android та iOS.

Позитивними сторонами цього асистенту я би назвав дуже високий рівень персоналізації, який очевидно суттєво впливає на приємний користувацький досвід, приємний інтерфейс та UX, що є дуже важливою складовою для комфортного використання, а також здатність вправно реагувати на неочікувані фрази та якісно їх обробляти завдяки присутності нейромереж. Але такий асистент теж має досить велику купу значних мінусів. Наприклад, він не має якоїсь конкретної функціональності. Це означає, що він не зможе користувачу налаштувати будильник або, скажімо, замовити їжу доставкою на дім. Також він досить часто «галюцинує» - тобто видає не зовсім логічні або надумані відповіді, що досить негативно може вплинути на загальний досвід роботи з ним. Більше того, даного помічника не вийде налаштувати під себе. Але окрім звичайних користувацьких мінусів, цей асистент ще не підійде і для розробників завдяки наявності закритого коду.

А тепер проаналізуємо вітчизняний приклад – чат-бот Monobank. Як ми вже знаємо, Monobank, один із найпопулярніших українських банків, запустив свого власного чат-бота, який частково здатен виконувати функції інтелектуального помічника. Тобто його можна умовно вважати

спеціалізованою версією мобільного асистента. Відповідно серед його основних функцій варто назвати наступні:

- вміння давати відповіді на питання клієнтів щодо їхніх банківських справ.
- здатність допомагати клієнту вирішувати питання стосовно своїх справ з банком.
- вміння зв’язати користувача із оператором у випадку необхідності.

У такому випадку серед очевидних сильних сторін цього асистента варто виділити здатність вирішувати чітко поставлені задачі в рамках справ клієнта з банком, а це вказує на досить високу практичність помічника у використанні. Також він вбудований прямо в мобільний додаток, що дозволяє дуже швидко почати з ним працювати, і дуже легко продовжувати цю співпрацю. Однак із його практичності випливає найголовніший недолік – це його спектр тем для обговорення. Оскільки цей чат-бот створений для праці з банківськими справами, його навряд чи вийде використовувати як повноцінного асистента у будь-яких життєвих справах клієнта, на відміну від того ж самого Google Assistant. Також даний помічник дуже погано може підтримувати діалог в прямому сенсі цього слова. Тобто він більше виступає не як твій співрозмовник, а як помічник формату «питання-відповідь». Ці недоліки остаточно підкріплюються тим, що такий чат-бот не навчається самостійно, адже усі його відповіді заздалегідь вшиті в його базовий функціонал, що робить його обмеженим у використанні.

А зараз я проаналізую приклад, про який я дізнався зовсім нещодавно. Мова йде про Mycroft – розумний асистент з відкритим кодом, який в дійсності позиціонується як альтернатива комерційним рішенням, таким як Alexa чи Siri. Цей асистент має підтримку голосу, може спілкуватись звичайним текстом, його можна запустити локально або на віддаленому сервері, і до того ж він кастомізується. Цікаво те, що він має повністю відкритий код, а це значить що він є прекрасним вибором для розробників, працює на різних платформах, а також підтримує створення власних навичок (skills).

Позитивними сторонами цього асистенту безумовно можна назвати повну кастомізацію під користувача або бізнес та можливість його запустити без зовнішніх сервісів. Якщо казати про розробників, то цей асистент є гарною базою для створення власних кастомізованих рішень. Однак такий асистент має далеко не найкращий користувацький інтерфейс, його дуже довго потрібно налаштовувати вручну, а також з ним ми маємо слабку підтримку за замовчуванням і мало готових рішень, на відміну від великих платформ. Тобто цей асистент більше підходить більше як фундамент для розробника для створення чогось абсолютно нового і «свіжого», ніж як уже готове рішення для користувача.

Беручи до уваги усе вище сказане, наведу порівняльну таблицю, яка допоможе нам швидко, легко та чітко визначити, який серед цих розумних асистентів є найбільш «вдалим» прикладом реалізації інтелектуального помічника.

Таблиця 2.1

Порівняльна таблиця найвпливовіших розумних асистентів

1	2	3	4	5
Асистент	Google Assistant	Replika	Monobank Bot	Mycroft
Тип	Загальний	Емоційний	Спеціалізований	Загальний
Мобільна платформа	Так	Так	Так	Частково

Продовження таблиці 2.1

1	2	3	4	5
Основна функція	Команди, пошук, інтеграція	Спілкування, підтримка	Банківські операції	Розширені дії, open-source
Відкритість	Ні	Ні	Ні	Так
Підтримка укр. мови	Частково	Ні	Так	Ні
Рівень кастомізації	Низький	Середній	Низький	Високий

Внаслідок цього аналізу можна зробити декілька цікавих висновків, навіть неочевидних. По-перше, варто розуміти, що мобільні асистенти – це не завжди про голосову взаємодію з користувачем. Досить часто текстовий помічник буде кращим і більш зручним варіантом, особливо тоді, коли йдеться про якісь специфічні задачі (як у банку). По-друге, далеко не кожен інтелектуальний помічник універсальний. Наприклад, чат-бот Monobank дуже добре справляється зі своєю вузькою задачею, проте явно не зможе замінити хоча б приблизно такого асистента як Google Assistant. Також слід зауважити, що таке явище як кастомізація – рідкість серед вже існуючих рішень. Як ми бачимо, подібні існуючі рішення, як правило, це повністю або частково закриті системи, які або складно або взагалі неможливо якимось чином адаптувати під власні задачі та потреби. Якщо ми кажемо про розробку нових застосунків з повного нуля, то в такому випадку краще роздивлятись відкриті рішення, такі як Rasa або Mycroft. Вони є дуже перспективними і мають великий потенціал створення абсолютно нових, більш потужних рішень на світовому ринку, враховуючи більш гнучку та розширену кастомізацію. Але головною проблемою подібних підходів досі залишається потреба у великій кількості ресурсів та довгого ручного налаштування великої кількості параметрів та функцій.

2.3. Нюанси розробки розумних помічників з технічної точки зору

Коли ми кажемо про взаємодію зі штучним інтелектом, у нашій голові зазвичай випливає така картина: користувач надає запит, наприклад, у вигляді текстового повідомлення, а розумний асистент дає текстову відповідь. Або користувач щось каже, а асистент в свою чергу виконує якусь певну дію. На перший погляд, такий принцип роботи виглядає дуже просто, проте «під капотом» в ці моменти працює досить важка система, яка з логічної точки зору має виконати велику купу дій задля видачі бажаного результату. Їй необхідно розпізнати та обробити мову, визначити логіку поведінки залежно від запиту, у випадку необхідності взаємодіяти належним чином з певними сторонніми сервісами тощо.

Я вважаю, що найбільш ефективною архітектурою сучасного мобільного розумного помічника є так звана клієнт-серверна архітектура [14, 26]. Вона дозволяє розділити зону відповідальності на два основних компоненти – Frontend та Backend, де Frontend – інтерфейсна частина асистента, основна задача якої – це надати можливість користувачу комфортно співпрацювати з асистентом, а Backend – це вже логічна частина, яка власне приймає усі необхідні дані, обробляє їх, та видає готовий результат до інтерфейсної частини для її показу на екрані. Я вважаю, що необхідно намалювати і наглядно показати, як схематично може виглядати подібна архітектура, оскільки одне діло описати її текстом, зовсім інше показати візуально. На рисунку 2.1 можна побачити, яким чином вона виглядає:

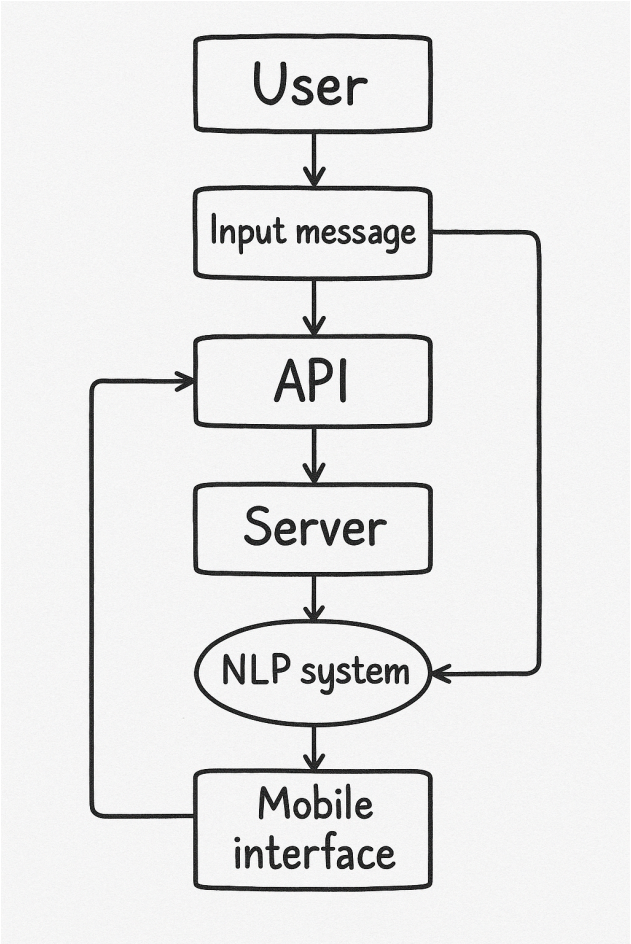

Рисунок 2.1 Клієнт-серверна архітектура

В дійсності подібний архітектурний підхід дозволяє досить суттєво оптимізувати роботу асистента, дозволяючи Frontend частині не просто виконувати конкретно поставлену їй задачу, а й робити це ефективно без зайвої «напруги», адже уся логіка обробки інформації та генерації результату працює у відповідній Backend частині нашого асистента.

Отже, ми визначили що для реалізації мобільного розумного асистента необхідно поділити усю його логіку роботи на два основних компоненти – Frontend та Backend складові. Тепер варто обговорити, які саме технології зазвичай використовуються для реалізації кожних цих складових окремо. Якщо йдеться про Frontend складову, то зазвичай для операційної системи Android використовуються такі технології як Flutter або певне «нативне» рішення на базі Kotlin. Flutter – це потужний кросплатформовий фреймворк від Google, який дозволяє збирати додаток водночас під дві найпопулярніші

операційні системи – Android та iOS. Також існує «нативний» iOS фреймворк Xcode, де в основному код додатку пишеться на Swift мові програмування з використанням фреймворку SwiftUI для створення інтерфейсу додатку. Щодо Backend складової, насправді тут може бути абсолютно будь-що. Усе залежить від вподобань розробника, специфіки поставленої задачі та конкретизації фінального результату. Проте найчастіше використовують, наприклад, Python, як у випадку з фреймворком Rasa (який власне повністю написаний на Python). Можуть використовувати Node.js, якщо мова йде про інтеграцію з зовнішніми сервісами або про створення окремої бізнес-логіки. Також існують такі популярні фреймворки як FastAPI, Flask, Express.js, які використовуються для створення своїх власних API(Application Programming Interface).

Однак цього не достатньо, для повноцінної реалізації свого мобільного розумного асистента. Також необхідно замислитись над реалізацією NLP-платформи. Це «основний мозок» асистента, у якому і зосереджена уся «магія» роботи асистента, визначення та обробки мови, процесу підбору відповіді, її генерація тощо. Rasa в цьому випадку може використовуватись для гнучкого налаштування, детальної кастомізації та локального або серверного розгортання. Щодо хмарних рішень, можуть використовуватись DialogFlow, Wit.ai або Lex. Також зазвичай необхідно забезпечити базою даних нашого асистента. Для цього відмінно підійдуть такі сервіси як PostgreSQL, MongoDB, Firebase тощо.

Зазвичай, у мобільні додатки взаємодія з сервером інтегрується за допомогою використання REST API, адже це зручно, надійно, а також підтримується усіма фреймворками. REST (Representational State Transfer) – це такий архітектурний стиль, який використовується для побудови веб-сервісів. У контексті мобільного розумного асистента REST API – це визначений набір HTTP запитів, за допомогою якого здійснюється комунікація між клієнтом та сервером, або дати можливість клієнту надіслати свій запит до сервера, а серверу дати можливість отримувати запити, обробляти їх, і віддавати згенеровану відповідь безпосередньо клієнту. Це все працює приблизно

наступним чином: наприклад клієнт пише у чат «Привіт». Додаток формує HTTP POST-запит до, умовно кажучи, https://server.com/webhooks/rest/webhook. Сервер в свою чергу обробляє цей запит та повертає клієнту відповідь «Привіт! Чим можу допомогти?».

Але ми знаємо, що будь-який запит має якусь певну затримку – час, який необхідно почекати, поки запит дійде куди треба, і поки звідти не прийде до нас відповідь. У таких випадках задля реалізації функціоналу отримання миттєвої відповіді(жива комунікація), можуть використовувати WebSocket або push-повідомлення. У випадку фреймворку Rasa, наприклад, існує вбудований REST-інтерфейс, за допомогою якого можна дуже легко спілкуватись з мобільним клієнтом. WebSocket – це такий протокол, що дозволяє після своєї ініціалізації один раз встановити повноцінне з'єднання між клієнтом та сервером задля того, щоб без зайвої потреби відкривати нове з'єднання мати можливість надсилати дані в обидві сторони. Використання подібного протоколу дуже зручне у випадках, коли потрібно реалізувати швидку відповідь на запити. В основному це актуально для чатів, ігор та будь-яких застосунків, де потрібна швидка реакція на запит у реальному часі.

Проте задля реалізації більш «живого» спілкування з ботом, особливо коли мова йде про додаток у вигляді чату, краще слід використати Rasa Socket.IO канал. Це специфічна бібліотека, яка поверх WebSocket може надавати додаткові можливості, такі як автоматичне перепідключення у випадку розриву з'єднання, підтримка fallback-механізмів(наприклад на HTTP polling, якщо WebSocket в поточному випадку не підтримується) а також обробка подій (event-based модель). Socket.IO в Rasa здатна зробити бота більш «живим» - він здатен в такому випадку миттєво реагувати на запити користувача, підтримувати діалог базовими фразами, надсилати повідомлення без очікування запиту, відображати свій поточний статус роботи («друкує…», «очікуйте…», тощо).

Як ми вже знаємо, розумні асистенти бувають як голосові, так і текстові. У мобільному ж середовищі використовуються обидва варіанти, оскільки

текстовий інтерфейс набагато простіший в реалізації та завжди актуальний незалежно від обставин, а голосовий суттєво додає зручності у побутовому використанні. Так як Rasa «з коробки» не підтримує спілкування голосом, це можна виправити за допомогою створення взаємодії Rasa з будь-яким голосовим асистентом через API. Таким чином, Rasa дає можливість детально налагодити розпізнавання мови голосом, звідки маємо великий потенціал у створення практичного, корисного та оптимізованого асистента.

У контексті реалізації мобільного розумного асистенту розробнику критично важливо враховувати певні особливості, оскільки навіть найпотужніший мобільний пристрій зараз все ще дуже слабкий для того, щоб повноцінно підтримувати можливість обробки інформації на рівні «мозку» розумного асистента. Тобто обмеженість ресурсів смартфона напряму впливає на його обчислювальні потужності, що не скажеш про комп'ютери або повноцінні сервери. Саме з цієї причини, на мою думку, розробнику варто переносити усю Backend складову на спеціальний віддалений сервер. Пам'ятаємо, що не менш важливою є Frontend частина. Якісний UI суттєво потрібен для кінцевого користувача, якщо він грамотно побудований, він дасть можливість додатку бути дуже простим у розумінні, швидким для взаємодії та «ненав'язливим». Зазвичай, в таких випадках вдаються до «чатоподібного» інтерфейсу, оскільки він дуже простий, легкий, до того ж усім давно знайомий. Оскільки взаємодія з сервером неможлива за відсутності інтернет підключення, варто заздалегідь передбачити, яким чином буде себе поводити додаток за відсутності інтернет підключення. Для цього може бути реалізований певний офлайн-функціонал, показ повідомлень тощо. Також у випадку, якщо додаток має можливість працювати з конфіденційними даними користувача, слід пам'ятати про інформаційну безпеку при обробці таких персональних даних – важливо використовувати протокол запиту HTTPS, авторизацію, шифрування та дешифрування даних(як на стороні додатку, так і на стороні сервера), і бажано не зберігати зайву інформацію на стороні клієнта, оскільки це в майбутньому може завдати великої шкоди. Куди

простіше доглядати за своїм власним сервером, аніж за пристроєм користувача.

2.4. Сучасна проблематика та виклики, що виникають при розробці мобільних інтелектуальних асистентів

Сучасні мобільні розумні помічники для пересічного користувача виглядають досить просто. Здавалось би, достатньо лише навчити бота відповідати на поставлені запитання потрібним чином, і все працює як потрібно. Однак далеко не кожен задумується над тим, наскільки насправді це важко реалізувати. Оскільки розробка сучасного мобільного розумного асистента важка та змушує розробника звертати свою увагу на велику кількість можливих проблем та потенційних викликів, створити дійсно якісний та робочий продукт стає все більш складною задачею.

Однією із перших проблем на шляху створення розумного асистента є процес розпізнавання природної мови (Natural Language Understanding, або NLU). Як ми знаємо, усі люди спілкуються між собою не ідеальним чином. Ця властивість спілкування переходить не тільки на людей, а і на інтелектуальних помічників. Тобто користувач може писати слова з помилками, або плутати слова та їхній порядок у реченні, або навіть використовувати скорочення, «сленги» та діалектизми. І будь-який сучасний розумний помічник має бути готовим до цього. У цьому випадку не допоможе звичайне порівняння тексту один до одного. Замість цього використовують більш поглиблені алгоритми обробки тексту, виявлення сутностей, виділення інтенцій із заданого, як правило, неідеального вводу. Також варто розуміти, що не усі мови, які відомі людству, мають настільки широку базу для навчання, як наприклад англійська мова. І йдеться про відсутність підтримки не стільки стародавніх або маловідомих мов, скільки про досить поширені та багато кому відомі мови, як-от українська, польська, чеська і т.д.

У свою чергу достатньо впливовим викликом є контекстна обробка. Будь-яка бесіда, запит, діалог, розмова завжди матиме якийсь контекст. Фактично, контекст – це сукупність деяких обставин, за допомогою яких визначається значення поточної розмови і будь-якого сказаного речення або слова в ньому. Розумний асистент обов'язково має пам'ятати поточний контекст і адаптувати свої відповіді на запитання згідно ньому. Класичний приклад: користувач хоче замовити піцу. Наприклад, він пише простенький короткий запит: «Я хочу піцу». Помічник у свою чергу питає: «Яку саме піцу Ви хочете замовити?». У свою чергу користувач відповідає максимально коротко, але чітко: «Пепероні». Цей простий приклад наглядно показує, наскільки важливо розумному помічнику розуміти контекст. У цьому випадку, оскільки у контексті фігурує замовлення їжі, а конкретно піци, то відповідь «Гавайську» означає додаткову відповідь на перше поставлене питання асистентом. І помічник має зрозуміти, що ця відповідь все ще стосується замовлення піци, і ця відповідь фактично доповнює собою перший запит користувача. Розуміння контексту дозволяє уникнути «казусних» та неприємних ситуацій під час співпраці з асистентом, дозволяючи таким чином значно покращити користувацький досвід та ефективність роботи самого розумного помічника. Інакше кажучи, контекстна обробка – дуже серйозна і важка задача, яка вимагає правильної побудови діалогових сценаріїв або використання моделей пам'яті в діалозі.

Серед найважливіших критеріїв сучасного мобільного розумного асистента є «мультимовність», тобто підтримка різноманітних людських мов. Ні для кого не є секретом той факт, що більшість людей у сучасному глобалізованому світі спілкується хоча б на двох мовах. Хочу зауважити, що у більшості сучасних країн дуже велика кількість населення може спілкуватися далеко не однією мовою. Україна у цьому випадку не є виключенням. У нашій країні зазвичай користувач надає запити українською мовою та очікує від розумного асистента відповіді саме цією мовою, однак більшість українського населення також активно може спілкуватись англійською або будь-якою

іншою мовою. Багатомовність як параметр застосунку має бути реалізований не через просте дублювання сценаріїв діалогу, а за допомогою використання гнучких механізмів багатомовної обробки.

Сучасні розумні асистенти дуже часто співпрацюють з персональними даними користувача: ім’я, контактна інформація, місцезнаходження, історія покупок тощо. Це створює велику відповідальність перед користувачами, адже інформаційна безпека та захист конфіденційної інформації користувачів є одним із ключових аспектів функціювання, який має бути реалізовано належним чином. Основними ризиками у цьому випадку може стати створення погано захищеного API (Application Programming Interface), через який може статись витік даних. Також досить важливим ризиком є несанкціоноване збереження конфіденційних даних або передача персональної інформації третім особам. Недостатня прозорість у тому, що саме зберігається та використовується також може призвести до великих проблем у питанні інформаційної безпеки інтелектуального асистента під час роботи з персональними даними користувача. Серед популярних рішень подібних ризиків може бути використання протоколу HTTPS для захисту даних під час їхньої передачі. Також досить важливим способом рішення подібних викликів є шифрування та дешифрування важливої інформації, що також зможе додатково захистити конфіденційну інформацію від третіх осіб. Також варто пам’ятати, що далеко не уся персональна інформація дійсно потрібна асистенту для роботи – достатньо мінімізувати кількість такої інформації, щоб використовувати тільки те, що критично необхідно для роботи асистента. Таким чином ми зможемо мінімізувати кількість потенційної шкоди, якщо навіть і станеться витік інформації або передача її «не в ті руки».

Попри все це, досить часто розробники забувають про один із найважливіших критеріїв якісно зробленого та повноцінного розумного асистента. Мова йде про UI/UX інтерфейс. Саме від нього залежить, наскільки користувачу буде усе зрозуміло з першого разу, наскільки користувач зможе

легко та швидко орієнтуватись в інтерфейсі додатку, чи буде бачити користувач активний статус бота під час співпраці з ним аби розуміти поточний його стан та розуміти, чи можна зараз надіслати запит, чи варто почекати на відповідь. Дуже «розумний» асистент – не означає дуже якісний. У цьому випадку погано спроектований та реалізований UI/UX інтерфейс може зіпсувати будь-які позитивні враження навіть від найрозумнішого асистента у світі. Інтерфейс – це візуальна оболонка будь-чого, що здатна бачити людина. Адже ми спочатку бачимо, оцінюємо побачене а тільки потім починаємо користуватись тим, що бачимо. Саме тому дуже важливо правильно реалізувати не тільки технічні аспекти розумного асистента, а й те, яким саме чином користувач буде взаємодіяти з ним. Існують декілька дійсно корисних практик, які допоможуть значно покращити візуальний компонент розумного асистенту. Зокрема, варто показувати візуально індикатори завантаження або «друкування відповіді». Також необхідно давати змогу користувачу відредагувати свій попередній запит та перенадіслати його. Для цього необхідно створювати додаткові кнопки із зрозумілими іконками або текстом у потрібному зручному для використання місці. Також можна реалізувати основний візуальний компонент додатку у простій, зрозумілій та ненав’язливій формі, яка буде одразу інтуїтивно зрозумілою для кожного як нового, так і вже досвідченого користувача [3].

Окрім того, слід також замислитись про технічні та організаційні обмеження мобільних пристроїв. Незважаючи на те, що будь-який сучасний смартфон, як правило, забезпечений потужним процесором та має потужний комплексний набір обчислювальних можливостей, сучасні розумні асистенти дедалі потребують все більш таких можливостей від пристроїв. Саме тому варто розумного асистента спочатку запустити на віддаленому сервері та надати цьому потужному серверу право на усю логічну обробку інформації, а самому мобільному пристрою у свою чергу дати можливість лише спілкуватись із сервером та візуально відображати необхідну інформацію. Це допоможе зекономити ресурси і без того недостатньо потужного мобільного

пристрою та підвищити ефективність взаємодії з розумним асистентом. Також варто пам’ятати про мережеве з’єднання, адже далеко не кожен користувач має стабільний інтернет. Також існує вимога до швидкої реакції – на своєму досвіді можу сказати, що це дуже не зручно чекати від асистента відповідь протягом декількох хвилин. Розумний помічник має вміти давати відповідь на поставлене питання якомога швидко, бажано в рамках декількох секунд. Складність масштабування також є серйозним викликом на шляху створення розумного асистента, і особливо гострим цей виклик стає тоді, коли кількість користувачів досягає значної позначки. Варто пам’ятати і про необхідність постійного оновлення даних та моделей для того, щоб завжди підтримувати актуальність наданих відповідей.

Таким чином, створення розумного мобільного асистента є досить важкою, відповідальною справою. Без чіткого плану дій, проектування, розуміння усіх перелічених проблем та викликів, розробник не зможе створити дійсно якісний, безпечний та корисний продукт, який зможуть користуватись усі без проблем, з комфортом та високою ефективністю. Забезпечення рішення під кожну таку проблему та виклик, на перший погляд, виглядає неймовірно важко. Однак в дійсності достатньо лише приділити трохи часу, терпіння та ініціативи, і необхідні рішення прийдуть самі по собі. А звідти і асистент зможе бути створений так, як цього потребує сучасне середовище користувачів.

2.5. Узагальнення проведеного аналізу

Проведений у цьому розділі аналіз сучасного стану розробки мобільних інтелектуальних асистентів, прикладів реалізації таких помічників, технічних особливостей створення та основних проблем та викликів дозволяє сформувати цілісне розуміння та уявлення того, яка існує специфіка розробки подібного асистента у сучасних умовах. Таким чином ми маємо можливість узагальнити усе вище сказане, підсумувати результати аналізу, а також з

певністю обрати правильний підхід до своєї власної реалізації мобільного розумного асистента.

Отже, сучасні мобільні інтелектуальні помічники дедалі стрімко набирають обороти. Світовий ринок розумних асистентів показує значну тенденцію у розвитку цього напрямку комп'ютерних наук. І основним пристроєм для реалізації подібних помічників є сучасні смартфони, у які можна легко інтегрувати помічника та не менш легко ним користуватись. Простий та зручний інтерфейс, легка інтеграція у різноманітні сервіси, мобільність та доступність – основні чинники реалізації розумного помічника саме для мобільних пристроїв.

Сьогоднішні найпопулярніші розумні асистенти, такі як Google Assistant, Apple Siri, Amazon Alexa, дійсно є потужними рішеннями у цій сфері. Однак вони мають досить велику кількість недоліків, які впливають як на звичайних користувачів, так і на розробників, які б хотіли використовувати ці рішення для своїх потреб та цілей. Серед перших таких недоліків є повна або часткова закритість системи, з якої ми маємо дуже обмежену кастомізацію під власні потреби. Відкриті рішення, на кшталт Mycroft, дають більше свободи дій, проте вимагають велику кількість зусиль та ручних налаштувань, що робить їх не дуже зручним варіантом вибору основи для створення розумного асистенту серед сучасних рішень.

Технічна архітектура будь-якого сучасного мобільного розумного асистента базується на клієнт-серверній архітектурі. Такий підхід дозволяє отримувати швидку зручну запит-відповідь, спілкуватись з асистентом та отримувати від нього «живі» відповіді у режимі реального часу, а також значно економити обчислювальні можливості поточного мобільного пристрою, перекидаючи основну логіку розмірковувань асистента на потужний сервер, на якому він власне і запущений.

Однак розробка мобільного інтелектуального помічника має свій набір проблем та викликів, з якими стикається кожен розробник, бажаючий реалізувати своє власне рішення. Серед таких однозначно варто виділити:

- технічні можливості розпізнавання природної мови, зокрема врахування мовних помилок, сленгів та неоднозначностей.
- контекстну обробку діалогів для забезпечення природності спілкування.
- підтримку мультимовності та якісну локалізацію.
- забезпечення безпеки персональних даних користувачів під час спілкування з асистентом.
- розробку зручного, якісного, інтуїтивного інтерфейсу та оптимізацію взаємодії користувача з асистентом.

Також варто пам’ятати, що сучасний розумний мобільний асистент обов’язково має бути адаптивним та гнучким, якщо ми кажемо про успішне рішення. Такий помічник має вміти адаптуватись під різні сценарії використання, підлаштовуватись під особливості локального ринку та індивідуально потреби кожного користувача. Таким чином, беручи до уваги результати аналізу, стає досить очевидним те, що створення нового, «свіжого», мобільного розумного асистента на базі відкритого фреймворку Rasa – цілком виправдане і добре рішення. Такий вибір дозволить нам під час розробки взяти під повний контроль логіку та поведінку нашого асистента, забезпечити захист і приватність даних користувачів, адаптувати систему під конкретні завдання і навіть аудиторію, а також забезпечити підтримку української мови та врахувати локальні особливості використання. У наступному, третьому розділі, буде розглянуто сам процес розробки мобільного розумного асистента на базі фреймворку Rasa, його архітектуру, функціональні можливості, технічну реалізацію та оцінку ефективності готового помічника.

РОЗДІЛ 3
РОЗРОБКА МОБІЛЬНОГО ІНТЕЛЕКТУАЛЬНОГО АСИСТЕНТА НА БАЗІ ФРЕЙМВОРКУ RASA

3.1. Вибір інструментів та технологій розробки

З точки зору розробника початком реалізації розумного мобільного асистента є власне вибір технології, завдяки якій буде створюватись додаток. Оскільки мова йде про мобільного асистента, тобто додаток, який має бути випущений на мобільні платформи(здебільшого це Android та iOS), потрібно розглянути такі технології, які ідеально підходять для створення додатку саме під ці найпопулярніші операційні системи. Існує досить велика кількість сучасних популярних технологій, кожна з яких відмінно виконує свою функцію. Однак на мою думку, як на думку починаючого розробника мобільних додатків, варто розглянути два варіанти – або готовий ігровий рушій або якийсь фреймворк, за допомогою якого буде зручно працювати над логікою та інтерфейсом додатку. Перше, що приходить в голову у цьому випадку, Unity – це дуже популярний та потужний ігровий рушій, за допомогою якого можна створювати повноцінні ігри будь-якого масштабу та об’єму на велику кількість різноманітних платформ. Однак цей рушій краще підходить саме для розробки якоїсь потужної гри, а суть нашої дипломної роботи в тому, щоб створити корисний та легкий «софт» (від англ. «software»). Тобто на мою думку Unity у цьому випадку буде грати роль більше як інструмент, основний набір функцій якого ми навіть не будемо використовувати. Таким чином нам потрібен такий фреймворк, який дозволить нам створити легкий за фізичним розміром мобільний додаток, але при цьому щоб була можливість мати потужний потенціал як у розробці логіки застосунку, так і у створенні візуального інтерфейсу нашого майбутнього розумного помічника.

Відносно нещодавно я познайомився з таким фреймворком як Xcode. Xcode – це потужний інструмент для розробки додатків будь-якої важкості від компанії Apple, який дозволяє створювати програми під усю екосистему Apple [30]. Оскільки Apple встигла за останні роки буквально захопити сучасний світовий ринок технології, кількість їхніх пристроїв по всьому світу вражає. Це стосується у першу чергу їхніх мобільних телефонів iPhone. Так як вони попадають під категорію «мобільний пристрій», вважаю за потрібне скористатись саме застосунком Xcode. Він фактично є IDE (Integrated Development Environment), тобто він одночасно є і редактором коду, і програмою для «паблішингу» нашого розумного асистента в AppStore – повноцінний каталог усеможливих застосунків, починаючи від ігор закінчуючи корисним «софтом».

Однак тоді виникає питання – а яким чином саме буде писатись код нашого додатку? Відповідь на це питання досить проста. Apple має свою власну мову програмування, Swift, і на ньому вона має два готових потужних UI фреймворки – SwiftUI та UIKit [1]. Різниця між ними у тому, що UIKit з’явився раніше за SwiftUI, і вони відрізняються за набором доступних об’єктів та їхніх методів для розробки. Тоді ви можете мене запитати – але чому моїм вибором стало саме це? У дійсності, Swift як мова програмування досить легка для освоєння. Вона трішки схожа на Python з точки зору синтаксису, тому код на ній пишеться легко, швидко та без проблем. Проте основний фокус у цьому питанні я би хотів поставити саме на фреймворк SwiftUI, який я буду використовувати для розробки нашого застосунку. Основна причина, з якої я вибрав саме цей фреймворк закладається у тому, що він дозволяє «верстати» (тобто реалізовувати UI частину додатку) за принципом «менше написаного коду – максимум отриманого результату» [2, 20, 25]. Це означає, що для того щоб створити хоча б мінімальне віконце з елементарними кнопочками та картинками, ми можемо написати буквально декілька строчок коду, і усе це з’явиться на екрані. Такий «шикарний результат» був досягнутий створенням різноманітних власних UI контейнерів,

які чудово поєднуються один з одним та комплексно створюють прекрасну, приємну картинку на екрані смартфона. Легка підтримка під iPhone дозволить нам не просто зробити додаток з легкістю прямо тут і зараз, а також дозволить з не меншою легкістю підтримувати наш додаток у майбутньому. Саме тому основною технологією розробки я вибираю саме Xcode на мові програмування Swift з використанням UI фреймворку SwiftUI.

Проте наш розумний асистент все ж таки має бути розумним, і має бути повноцінним асистентом. Основна суть моє дипломної роботи полягає у тому, щоб створити такого помічника на базі фреймворку Rasa [4]. Я вважаю не менш цікавим дізнатись, чому я обрав саме цей фреймворк для нашої роботи. Його я обрав для обробки природної мови (NLP) та побудови логіки діалогу між асистентом та користувачем. Відповідаючи на питання, чому саме його я обрав для реалізації свого власного розумного мобільного асистента, хочу визначити декілька суттєвих переваг, які допоможуть нам переконатися у ефективності мого вибору. Серед них, перш за все, варто відмітити відкритість коду та повний контроль над логікою роботи асистента, що надає розробнику саме фреймворк Rasa. Це дозволить нам максимально кастомізувати наше рішення під свої потреби та цілі, та максимально наблизитись до бажаного результату. Цей фреймворк також дозволяє підтримувати кастомні інтенти, сутності та складні сценарії діалогу. Усе це можна налаштовувати власноруч так, як нам це буде зручно. Також цей фреймворк дозволяє нам самостійно тренувати готову модель Rasa таким чином, як нам це буде потрібно, що однозначно є суттєвою перевагою цього фреймворку. У разі потреби у нас присутня можливість локально розгорнути нашу натреновану Rasa модель, що також відкриває додаткові можливості під час розробки [10]. Ну і фінальною перевагою цього фреймворку я би хотів відзначити готову інтеграцію через API (REST, Socket.IO), що ідеально підходить у випадку створення мобільного розумного помічника, оскільки таким чином ми зможемо позбавити наш додаток від важких обчислювальних процесів, делегуючи право цим займатись серверу, на якому ми будемо розгортати нашу Rasa модель, що

зробить додаток компактним, швидким і ефективним під час роботи. Саме завдяки вибору саме цього фреймворка я зміг створити повноцінного розумного асистента, який не просто відповідає на окремі повідомлення, а веде логічно зв’язаний діалог, враховуючи контекст попередніх реплік.

Оскільки Rasa як фреймворк не дає можливості створювати повноцінну мовну модель, яка зможе надавати максимально «природні» відповіді по аналогії з сучасними рішеннями, такими як ChatGPT, Alexa, Copilot, DeepSeek та інші, необхідно також врахувати потребу реалізації додаткового «розуму» нашому асистенту, щоб отримати ще більш добрий ефект та з точки зору ефективності наблизитись до рівня світових рішень. Для цього я вирішив реалізувати інтеграцію нашої моделі Rasa з ChatGPT. Щоб успішно створити взаємодію Rasa з ChatGPT, я написав скрипт на Python, який дозволяє Rasa спілкуватись з ChatGPT, а точніше передавати повідомлення нашого користувача останньому для того, щоб отримати від нього відповідь на переслати її користувачу. Використання інтеграції нашого асистента з ChatGPT дозволить розширити його базу знань та здатність відповідати навіть на ті запитання, які не були прописані у сценаріях.

Для того щоб реалізувати подібне Backend рішення, я використав GitHub для створення свого власного репозитарію, який буде розділено на дві гілки. Перша зберігатиме у собі натреновану Rasa модель, а друга – матиме скрипт роботи з ChatGPT. GitHub у цьому контексті є відмінним рішенням, оскільки він не просто дозволяє зручно керувати версіями на надійно зберігати нашу Backend базу, а й дає можливість розгорнути окремі гілки нашого репозитарію як окремі Web-Service, що дозволить звертатись до цих сервісів за допомогою REST API, що є для нас пріоритетною задачею. Тобто у цьому випадку, GitHub грає роль основної платформи серверної частини нашого розумного асистента.

Тепер необхідно визначити, де саме ми будемо розгортати гілки нашого репозитарію. Оскільки я не маю свого власного повноцінного сервера, потрібно використати якийсь готовий сервіс, який надає подібні послуги як безкоштовно, так і за певну оплату. Власне для розміщення нашої серверної

частини я обрав сервіс Render [6]. Але чому саме Render? По-перше, цей сервіс пропонує безкоштовний тарифний план для невеликих проектів. На старті розробки та власного тестування нашого помічника це було дуже важливим критерієм мого вибору. Оскільки ми використовуємо GitHub та гілки нашого репозитарію, потрібен такий сервіс, який дозволяє легко та швидко його розгорнути як Web-Service. І Render відмінно впорався з цією задачею. Більше того, цей сервіс дозволяє підтримку запуску декількох окремих веб-сервісів, що дуже доречно у нашому випадку, так як один веб-сервіс ми використовуватимемо для гілки з Rasa моделлю, а другий веб-сервіс нам потрібен для запуску нашого скрипта взаємодії з ChatGPT. Також Render для окремих веб-сервісів дозволяє нам додатково налаштовувати параметри запуску веб-сервісу, у тому числі можливість налаштовувати так звані Environment Variables. Це дозволить зберігати Access Token OpenAI, за допомогою якого Rasa буде спілкуватись з ChatGPT. Саме завдяки сервісу Render я зміг зручно, легко та швидко розгорнути свій GitHub репозитарій як окремі два веб-сервіси, завдяки яким мені відкрився доступ до повноцінно робочого Backend-у нашого розумного асистента.

Оскільки взаємодія з відкритим API OpenAI неможлива без Access Token, необхідно було знайти рішення якось його зберігати. Access Token – це унікальний ключ, який ідентифікує застосунок і дає йому право надсилати запити до ChatGPT [19, 23]. Render у цьому випадку ідеально пасує для зберігання нашого Access Token, оскільки його можна легко вказати для веб-сервісу з скриптом взаємодії з ChatGPT як Environment Variable (змінна середовища) [15, 29]. Налаштування Access Token в Render дозволяє нам не вказувати його у коді нашого застосунку, що автоматично захищає його від «витоку» та потрапляння у руки потенційних зловмисників, що у свою чергу робить нашого асистента захисним з точки зору кібербезпеки.

Отже, таким чином ми дізнались, які технології розробки та інструменти я використав для реалізації свого власного розумного мобільного асистента. Усе це дозволило мені створити дійсно зручного, інтуїтивно зрозумілого,

корисного, сучасного, практичного у використанні розумного мобільного асистента, який я можу підтримувати у майбутньому, розширяти його фунціонал, покращувати Backend складову, вдосконалювати усі основні компоненти додатку та завжди тримати «на плаву» свого помічника як сучасний аналог реалізації інтелектуального асистента.

3.2. Опис архітектурного рішення системи розумного асистента

Більш детальний повний опис свого готового мобільного розумного асистента я би хотів почати з опису мого архітектурного рішення внутрішньої його системи, тобто Backend складової. Цю архітектуру я побудував завдяки грамотному розділенню обов’язків між клієнтською частиною та серверною логікою, де кожна частина відповідає за свій етап роботи асистента. У дійсності, дану архітектуру можна описати наступним чином:

- користувач пише свій запит у мобільному додатку та натискає на кнопку для відправки свого повідомлення асистенту.
- додаток надсилає це повідомлення за допомогою REST API на Web-Service, на якому розгорнута наша натренована модель Rasa.
- Rasa отримує повідомлення, обробляє його, і у випадку якщо вона не може розпізнати повідомлення, надсилає його за допомогою REST API на Web-Service, на якому розгорнутий скрипт взаємодії з ChatGPT.
- Далі надсилається це повідомлення за допомогою скрипта до ChatGPT, він генерує відповідь, яку потім повертає до Rasa.
- Rasa віддає отриману відповідь назад користувачу у додаток.
- Додаток отримує це повідомлення як Web Request Response (відповідь), і показує його в інтерфейсі.

Такий архітектурний підхід дозволив мені досить ефективно та зручно спілкуватись з мобільний асистентом, який здатен генерувати відповідь на поставлене питання у реальному часі. Завдяки подібному розподіленню зон

відповідальності у результаті я отримав повноцінну Backend частину, яку можна вдосконалювати та покращувати навіть без повторної відправки нової версії додатку на реліз. Це особливо зручно, коли необхідно оперативно виправити декілька незначних помилок у роботі серверної частини, оскільки у такому випадку будь-яке втручання у сам застосунок необов'язкове.

Отже, першою складовою архітектури серверної частини нашого асистента є взаємодія користувача з мобільним додатком, де застосунок – це перша точка взаємодії. Вона має досить простий та зрозумілий інтерфейс: чат, куди користувач має змогу писати своє повідомлення, відправляти його і бачити усі свої повідомлення та повідомлення асистента у поточному екрані. Ця складова не містить у собі важкої логіки, оскільки основне її завдання – забезпечити швидкий, стабільний та зручний спосіб спілкування користувача із асистентом.

Наступна ключова складова розділяється на два основні компоненти – два веб-сервіси, запущені за допомогою сервісу Render, на одному з яких розгорнута натренована Rasa модель, а на іншому – скрипт, за допомогою якого здійснюється взаємодія з ChatGPT. Саме тут виконується основна логіка нашого мобільного розумного асистента. Тут Rasa здатна отримувати повідомлення від користувача, обробляти його, і у випадку якщо вона не може розпізнати його, надсилає далі до ChatGPT за допомогою «кастомного екшена».

Власне останньою ключовою складовою архітектури серверної частини нашого розумного мобільного асистента є ChatGPT. ChatGPT – це мовна модель від компанії OpenAI, яка дозволяє зробити більш розумним нашого асистента та допомагає генерувати більш «природну» відповідь у випадку, якщо Rasa не змогла розпізнати повідомлення користувача серед своїх інтентів, на основі яких вона тренувалась [5].

Закінчуючи свій опис, я би хотів додатково зазначити декілька сильних сторін подібного рішення. Перш за все, такий підхід дозволив мені розділити усю серверну частину на декілька окремих блоків, кожен з яких виконує свою

конкретно задану функцію. Сильною стороною тут є те, що у випадку якщо мені необхідно, наприклад, оновити модель Rasa, я просто можу «закинути» нову модель на GitHub репозитарій у відповідну гілку, і Render, побачивши зміни у репозитарії, автоматично зробить redeploy нашого веб-сервісу з моделлю Rasa [7]. Також я можу окремо додатково налаштовувати скрипт взаємодії з ChatGPT, не роблячи ніяк змін ні у веб-сервісі з Rasa моделлю, ні у самому додатку. Інакше кажучи, будь-які потенційні маніпуляції з боку серверної частини взагалі не потребують втручання у додаток як проект або у його код. Але навіть якщо я захочу додатково вдосконалити сам додаток, я зможу додатково над ним працювати і буду переконаний у тому, що точно не зустрінусь з проблемами після реалізації нового функціоналу, оскільки серверна частина працює максимально прозоро для мене, як для розробника, і випадково її «зламати» буде дуже важко. У світі розробників подібні випадки не поодинокі, тому моє рішення одночасно допомагає уникнути навіть таких потенційних проблем.

3.3. Розробка додатку за допомогою Xcode мовою програмування Swift

Після детального опису логіки роботи серверної частини нашого мобільного розумного асистента варто розглянути, яким чином я створив сам додаток, його базовий функціонал та інтерфейс. Оскільки мій застосунок – це простий, але дуже зручний та швидкий асистент, варто створити стартовий екран, який буде вітати користувача. Його інтерфейс має бути простий, ненав’язливим, та він має мати легкий спосіб переходу до нового чату з асистентом та до екрану з історією чатів. Пропоную більш детально розглянути інтерфейс початкового екрана на малюнку 2:

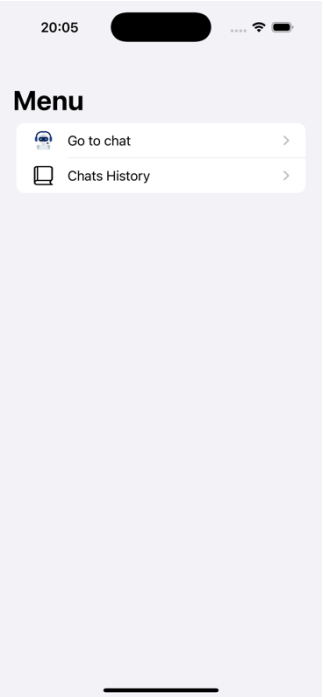

Рисунок 3.1 Початковий екран застосунку

Отже, на даний момент часу мій застосунок має поки що дві кнопки, кожна з яких веде на певний екран – або екран самого чату, або екран історії чатів. Так як наш асистент має бути зручним, мною було прийнято рішення створити інтерфейс чату стандартним способом, за допомогою зручних панелей, кожна з яких містить у собі текст, та за допомогою відповідного позиціювання цих панелей залежно від того, чиє це повідомлення. Тобто якщо повідомлення надіслав користувач, воно буде прив’язано до правого краю екрана, а якщо це повідомлення від асистента – воно буде орієнтуватись на ліву частину екрана відповідно [16]. Наглядний приклад візуального вигляду чату мого застосунку показано на рисунку 3.2:

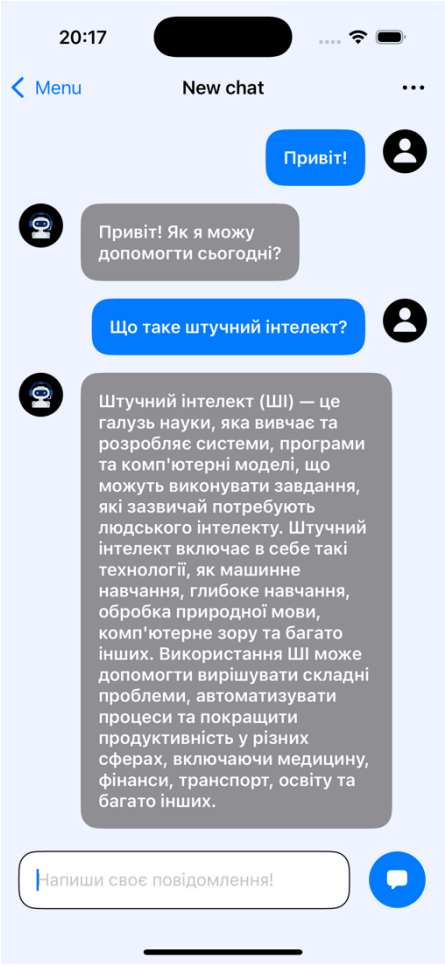

Рисунок 3.2 Зовнішній вигляд вікна чату з асистентом

Як ми бачимо, такий інтерфейс зустрічається досить часто у сучасних месенджерах оскільки він є зручним та інтуїтивно зрозумілим. Мій асистент у цьому випадку не є виключенням. Також я додав до цього екрану меню, за допомогою якого можна досить детально та зручно керувати поточним вікном. Серед базових функцій я вирішив реалізувати можливість збереження даного чату, можливість зберегти чат та одразу почистити поточне вікно від усіх повідомлень а також можливість видалити цей чат. Меню у цьому випадку створюється дуже легко за допомогою фреймворку SwiftUI, а зовнішній його вигляд можна бачити на рисунку 3.3:

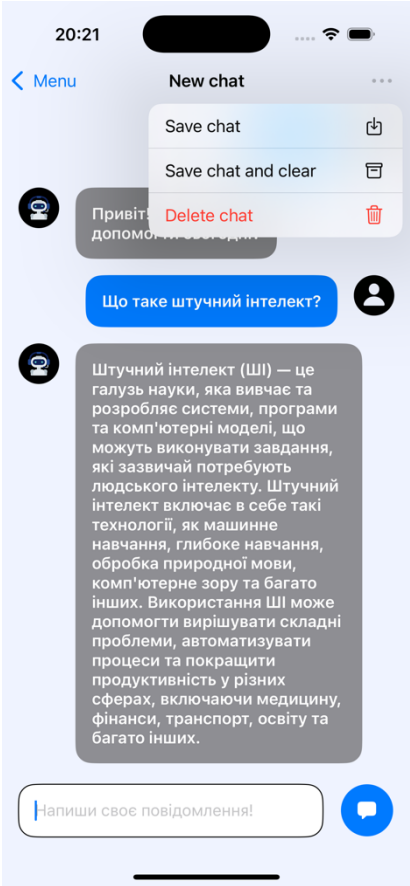

Рисунок 3.3 Зовнішній вигляд меню чату

У свою чергу вікно історії усіх чатів я вирішив розробити мінімалістичним, але максимально інформативним для користувача. Кожен чат в моєму випадку виглядає як кнопка, яка показує усю необхідну інформацію – назву чату та його дату. На мій погляд такий підхід дозволяє створити приємний інтерфейс, за допомогою якого можна буде дуже легко та швидко зрозуміти, який саме чат нас цікавить у даний момент часу. Відповідно кожна кнопка відкриває вікно чату, де ми можемо заново подивитись абсолютно усі повідомлення, які були між користувачем та асистентом. Така історія чатів дозволяє зручно зберігати усю цю інформацію та за необхідності її переглядати безліч разів у будь-який момент часу. Також такий підхід дозволяє різноманітним чином маніпулювати чатами, створюючи, наприклад, один великий або декілька маленьких, іменуючи їх певним чином як зручно користувачу. Саме вікно історії чатів можна розглянути на рисунку 3.4:

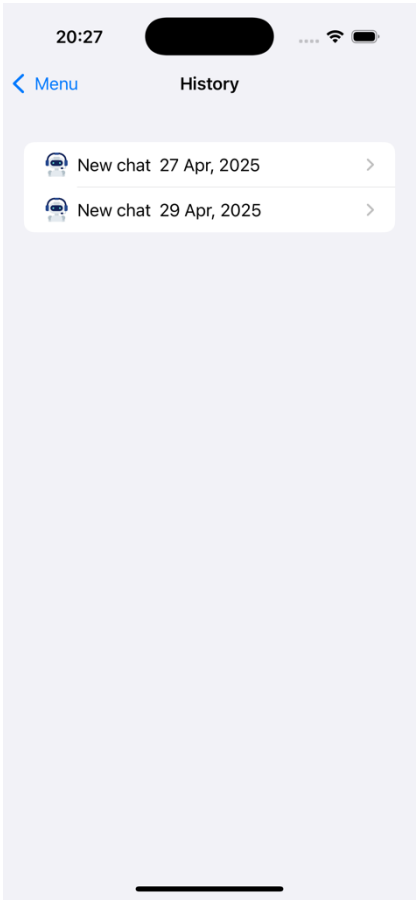

Рисунок 3.4 Вікно історії чатів

Таким чином мій розумний мобільний асистент виглядає як звичайний зручний чат-бот, за допомогою якого можна вести досить комфортну бесіду, зберігати безлімітну кількість чатів з безлімітною кількістю повідомлень, видаляти за якихось причин вже непотрібні чати, а також легко та зручно створювати нові бесіди. Подібний підхід до реалізації візуальної частини нашого асистента також є дуже потужним, оскільки він дозволяє у майбутній перспективі нескінченно покращувати та доповнювати функціонал застосунку. У будь-який момент часу можна додавати нові кнопки, нові вікна, не втручаючись у те, що вже є у додатку. Тобто таке рішення цілком впевнено можна назвати грамотним з точки зору проектування, оскільки воно як мінімум відповідає одному з принципів об’єктно-орієнтованого програмування SOLID, а конкретно принципу відкритості/закритості (англ. Open Closed Principle, OCP).

3.4. Огляд реалізації серверного компоненту розумного асистента

Оскільки ми вже розглянули принцип роботи Backend складової нашого розумного асистента, а також дізнались яким чином виглядає та працює наш додаток, на цьому моменті слід більш детально обговорити мій спосіб реалізації серверної частини нашого розумного мобільного асистента. Нагадаю, що для самої реалізації нам необхідно мати GitHub репозитарій, натреновану модель Rasa, скрипт, написаний на Python, за допомогою якого Rasa «спілкуватиметься» з ChatGPT, а також налаштування веб-сервісів на Render [17].

У такому випадку я би хотів почати з пояснення того, яким чином нам необхідно налаштувати GitHub репозитарій. У дійсності, це можна зробити дуже простим способом. Достатньо зареєструвати свій новий акаунт на офіційному сайті GitHub, використовуючи свою поштову скриньку, і нам відкривається увесь простір цього сервісу для роботи. Для того, щоб створити новий репозитарій, нам необхідно зайти на вкладку Repositories та натиснути на зелену кнопочку «New», що показано на рисунку 3.5:

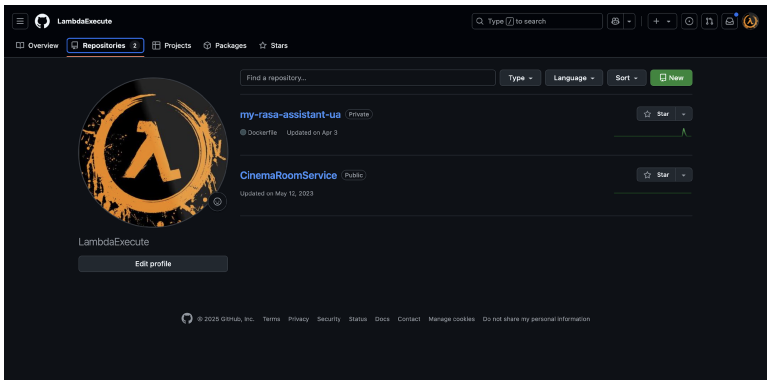

Рисунок 3.5 Початок створення GitHub репозитарію

Після чого перед нами відкриється віконце з більш менш детальним першочерговим налаштуванням нашого нового репозитарію. Тут в основному ми прописуємо його ім’я, що є дуже важливим параметром репозитарію, адже саме за ним взагалі відбуватиметься будь-яка взаємодія з репозитарієм,

додаткове налаштування файлів для першої гілки репозитарію а також «видимість» репозитарію. Детально екран налаштувань я показав на рисунку 3.6:

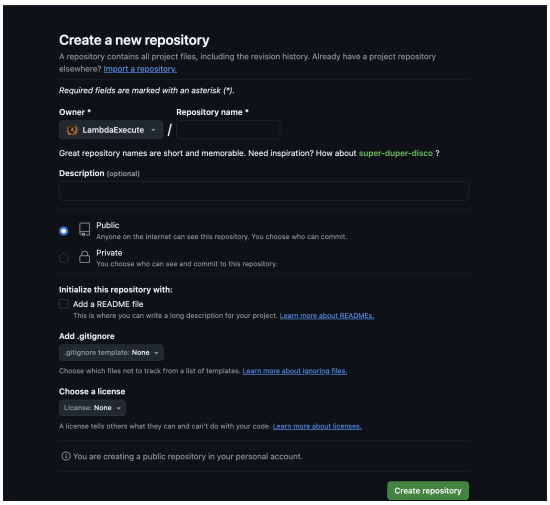

Рисунок 3.6 Налаштування репозитарію

Після створення репозитарію ми можемо працювати з ним в повній мірі. Існують два способи взаємодіяти з хмарним GitHub репозитарієм на своєму комп’ютері локально – або використовувати для цього консоль (залежно від операційної системи консоль може називатись по-різному, наприклад у Windows вона називається PowerShell, а в MacOS - Terminal), або спробувати працювати з більш простим, особливо для початківців, застосунком під назвою «GitHub Desktop». Оскільки ми використовуємо саме GitHub як сервіс надання хмарних репозитаріїв, нам необхідно буде працювати з ним за допомогою відповідної системи контролю версій. Найпростіше у такому випадку, на мою думку, використовувати систему контролю версій Git, яка по своїй суті виглядає як консольний набір текстових команд для роботи з репозитарієм як локально на комп’ютері, так і глобально у безпосередньо GitHub. Оскільки я використовую MacOS, я буду надалі показувати приклади консолі саме з цієї операційної системи, тобто працювати я буду у Terminal. Важливо уточнити, що принцип роботи з Git є одним і тим самим незалежно від того, яку операційну систему і яку саме консоль ви використовуєте.

Git – це надпотужний інструмент для контролю версій [8]. Він має велику кількість команд, і запам’ятати усі з них справедливо можна назвати випробовуванням. Однак існує офіційний сайт Git, який допомагає початківцям налаштувати його у себе на комп’ютері, а також дає увесь набір команд, які потрібні розробнику як для початкової роботи, так і для виконання просунутих дій. Наша задача полягає у тому, щоб на базовому рівні працювати з репозитарієм, оскільки якихось складних дій ми не робитимемо. Саме тому наша праця з Git буде легкою та швидкою.

Ми вже розібрались з тим, яким чином ми створюємо новий репозитарій а також з тим, яким саме чином ми будемо працювати з ним. На мій погляд, настав час поговорити про налаштування окремих гілок нашого репозитарію. Нагадаю, що у нашому випадку я використовую дві окремі гілки мого репозитарію, кожна з яких виконує свою окрему функцію. Перелік гілок мого репозитарію я візуально позначив на рисунку 3.7:

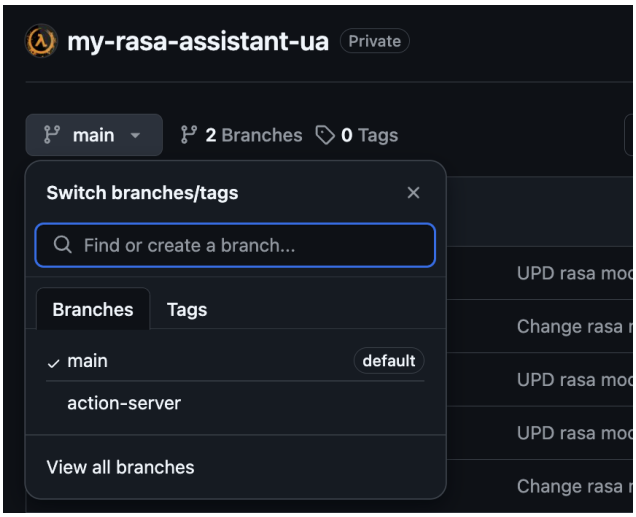

Рисунок 3.7 Структура репозитарію

Тобто, ми маємо дві гілки – main та action-server. Перша гілка відповідає за розгортку натренованої Rasa моделі, а друга – має у собі скрипт взаємодії Rasa з ChatGPT. Звідси виникає логічне питання – а як натренувати Rasa модель? Насправді, зробити це можна декількома способами, проте я обрав, на мою думку, найефективніший з можливих – це тренування моделі локально у

себе на комп’ютері. Оскільки ми маємо репозитарій, з яким ми можемо працювати також локально за допомогою Git команд через Terminal, нам ніщо не заважає створити необхідний набір файлів локально, і тільки потім, після ретельного тестування отриманої моделі, ми маємо можливість безпосередньо завантажити усі ці файли на потрібну нам гілку.

Отже, для того щоб отримати повністю натреновану модель Rasa, для початку нам необхідно завантажити фреймворк Rasa собі на комп’ютер. Оскільки Terminal в MacOS є потужним інструментом для роботи, за допомогою нього можна дуже легко встановити цей фреймворк. Достатньо ввести команду «pip3 install rasa», і вона завантажить фреймворк та налаштує його на роботи. Однак перед цим необхідно встановити таку версію Python, яка підходитиме для роботи з Rasa, так як цей фреймворк написаний саме на цій мові програмування. У моєму випадку, я використовував версію Python 3.10. Для завантаження цієї версії Python у Terminal достатньо виконати команду «brew install python@3.10». Brew, скорочення від HomeBrew, це менеджер пакетів операційної системи MacOS. За допомогою нього ось таким простим чином ми можемо встановити потрібну версію Python локально на комп’ютер.

Коли ми встановили потрібний Python та Rasa фреймворк, необхідно розібратись із тим, яким саме чином натренувати Rasa модель. Для цього нам потрібно виконати два кроки. Перший крок полягає у тому, що нам необхідно створити нову папку, і в ній створити усі необхідні файли конфігурації, на

основі яких фреймворк натренує нову модель. Повну структуру файлів ми можемо побачити на рисунку 3.8:

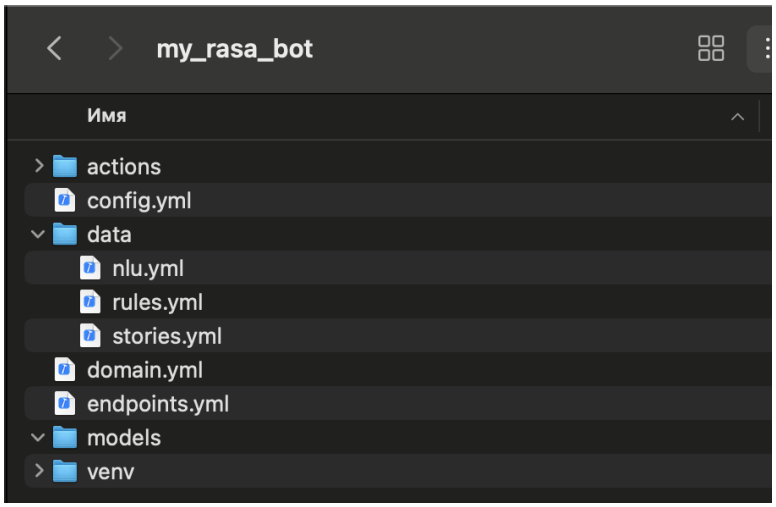

Рисунок 3.8 Конфігурація Rasa моделі

У цій конфігурації кожен файл виконує свою окрему функцію. Config.yml глобально налаштовує модель, тобто встановлює для неї ряд підтримуваних мов для розпізнавання, політики, назви необхідних компонентів моделі, її версію, а також додаткові поля налаштувань моделі. Файл nlu.yml контролює власне набір інтентів, тобто набір фраз, які може розпізнавати модель. У свою чергу кожен «субнабір» інтентів може класифікуватись по групах. Чим більше наведених таких груп, тим краще модель зможе розпізнавати інтенти. У файлі rules.yml розробнику необхідно прописати правила поведінки Rasa моделі у будь-яких ситуаціях. Тобто тут ми можемо конкретно та чітко вказати, як повинен реагувати наш бот при певних обставинах. У файлі stories.yml опціонально (тобто необов'язково) можна прописувати історію правил нашої моделі. У свою чергу файл domain.yml чітко прописує усі інтенти, усі дії нашого бота, а також можливі початкові відповіді асистента під час того, як він виконує якусь дію. Ну і у файлі endpoints.yml, як можна досить легко здогадатись за назвою файлу, розробнику необхідно прописати усі HTTPS адреси з певними їх назвами, до

яких буде «стучатись», тобто відправляти запит, наша натренована Rasa модель.

Після налаштування усих цих файлів, можна переходити у консоль. Для цього відкриваємо нове віконце консолі у нашій папочці, і починаємо працювати. По-перше, необхідно активувати Python Environment, тобто таке середовище, у якому буде здійснюватися тренування моделі [28]. Нам потрібно створити це середовище та запустити його. Після чого необхідно встановити у це середовище фреймворк Rasa для того, щоб почати тренування у рамках цього середовища. Ось так виглядає початок роботи у Terminal (рисунок 3.9):

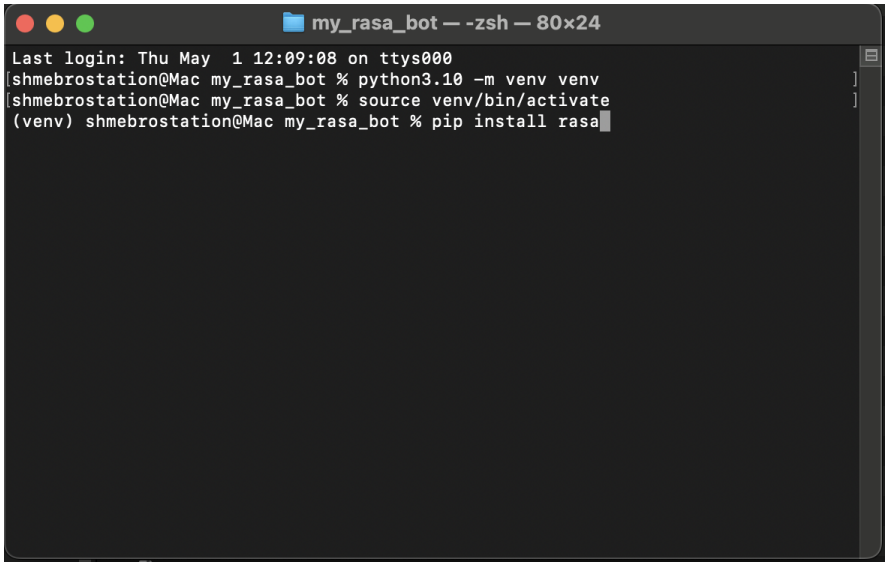

Рисунок 3.9 Початок тренування Rasa моделі

Після налаштування середовища та установки Rasa фреймворку, усе що нам необхідно зробити, це виконати команду «rasa train», щоб почати тренування моделі [27]. Результат тренування – це файл розширення .gz, у якому Rasa фреймворк налаштувала усі необхідні файли та компоненти нашого бота. Цей файл – і є натренована модель Rasa, яку ми уже можемо використовувати. Як видно на рисунку 9, існує спеціальна папка «models», яка зберігає у собі абсолютно усі натреновані Rasa моделі. Rasa фреймворк автоматично створює цю папку у разі її відсутності у поточній директиві, тому

за це перейматись на моменті налаштування файлів конфігурації майбутньої моделі не варто. Як було згадано вище: спосіб тренувати модель локально у себе на комп'ютері – відмінне рішення. І не дарма, адже на цьому етапі ми маємо можливість локально протестувати роботу нашого бота одразу після отримання нової натренованої моделі. Для цього, у випадку якщо у поточній директиві є папка «models», у якій присутня Rasa модель, у Terminal можна виконати команду «rasa shell». Ця команда у поточному Python середовищі запустить нашу натреновану модель, і відкриє для нас доступ до написання повідомлення до нашого бота, таким чином повністю емалюючи його справжню роботу.

Після того, як ми отримали та протестували нашу модель Rasa, варто її відправити на гілку main нашого репозитарію. У моєму випадку, структура файлів гілки main виглядає так, як на рисунку 3.10:

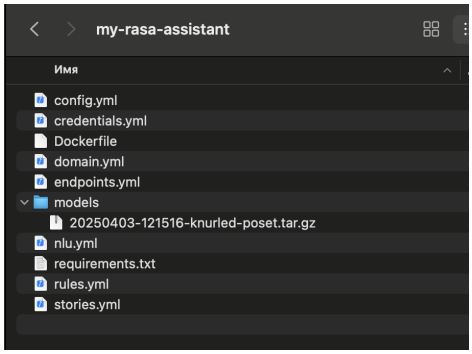

Рисунок 3.10 Файлова структура гілки main

Тут ми вже бачимо знайомі нам файли конфігурації нашої Rasa моделі. Окрім них, присутня папка «models» з натренованою Rasa моделлю, а також нові файли – Dockerfile, credentials.yml та requirenemnts.txt. Усі ці файли існують не дарма, адже кожен з них також виконує свою важливу функцію. Оскільки цю гілку ми будемо «розгортати» на Render як окремий веб-сервіс, нам необхідно зробити такий файл, який дасть фактично повну інструкцію, що необхідно зробити веб-сервісу при своєму запуску, щоб коректно запустити Rasa модель та усі необхідні для роботи компоненти. Цим займається файл

Dockerfile. Файл requirenments.txt має у собі набір назв фреймворків для встановлення та вказівку версії для окремого фреймворку. У файлі credentials.yml ми вказуємо, яким чином Rasa має спілкуватись з іншими веб-сервісами, тобто нам необхідно вказати ключове слово «rest», тобто за спілкування за допомогою REST API [9].

Для того щоб завантажити наші зміни на репозитарії, у консолі нам необхідно по черзі виконати наступні команді:

- git add .
- git commit -m “commit name”
- git push

Перша команда додає у відслідковування усі нові та поточні файли гілки, друга робить так званий «коміт», що фактично можна вважати сукупністю змін, а третя надсилає наш «коміт» на репозитарій, оновлюючи усі файли поточної гілки. Подібним чином ми також налаштовуємо гілку action-server, на яку нам необхідно «закинути» Dockerfile, requirenments.txt та сам скрипт взаємодії з ChatGPT.

Для того, щоб «задеплоїти», тобто запустити наші веб-сервіси, на Render створюємо новий акаунт [24]. Тут бажано виконати зв’язування з проектом на GitHub, оскільки буде простіше налаштовувати веб-сервіс через взаємодію з нашим GitHub акаунтом, а значить і взаємодію з репозитарієм. У нашому випадку ми налаштовуємо для веб-сервіси – перший під гілку main, другий під гілку action-server. Дуже важливо пам’ятати, що другий веб-сервіс повинен мати OpenAI access token. Оскільки Render надає можливість налаштовувати Environment Variables, скористаємось цією можливістю, створюючи для другого веб-сервісу таку змінну середовища, у яку ми пропишемо наш access token. Тоді у скрипті взаємодії з ChatGPT обов’язково потрібно прописати,

звідки ми беремо цей access token та як він називається. У результаті екран на Render виглядатиме подібно до того, як це виглядає на рисунку 3.11:

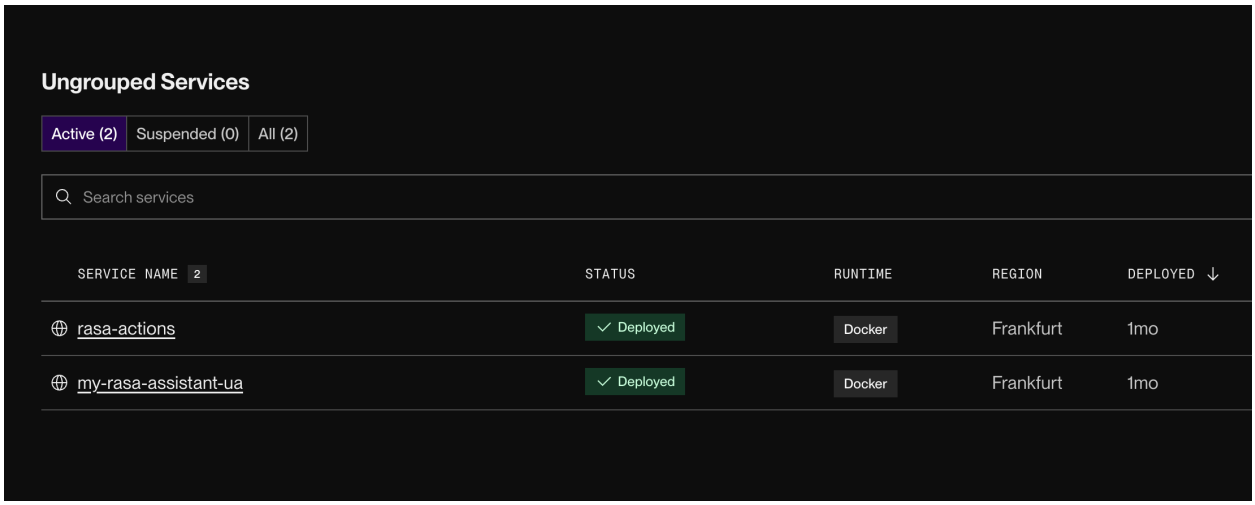

Рисунок 3.11 Готові веб-сервіси Render

ВИСНОВКИ

Під час виконання цієї роботи мною було помічено, що за останні 5 років технології штучного інтелекту суттєво розвинулись, зміни набули глобального масштабу. І це досить цікаве спостереження, адже перші згадки штучного інтелекту з’явились ще у минулому столітті, та попри цей факт, гігантський скачок у розвитку відбувся, на мій погляд, саме в останні роки. Я пам’ятаю, як ще на початку свого навчання у вищому навчальному закладі більшість з нас розуміли поняття штучного інтелекту як певний набір команд та поведінки бота, який дуже легко зрозуміти та прорахувати самостійно. Однак зараз це поняття набагато ширше, оскільки сучасне поняття штучного інтелекту базується більше на автоматизованому навчанні розумного бота без втручання людини у цей процес, якщо не враховувати додаткову розробку політик безпеки та захисту даних.

Сучасні рішення реалізації штучного інтелекту вражають. Компанії-гіганти, такі як Google, Apple та Microsoft, які фактично керують більшою долею сучасного ринку технологій, пропонують дійсно цікаві розробки штучного інтелекту. Вони потужні, кмітливі, дуже органічно інтегровані у сучасні мобільні та не тільки пристрої. Такі рішення дуже зручно співпрацюють із більшістю сучасних сервісів. На виході ми отримуємо відмінного помічника, який здатен допомагати людині виконувати рутинні задачі, підвищувати комфорт роботи та спрощувати життя у цілому. Проте такі рішення, як правило, абсолютно не піддаються серйозному рівню кастомізації та підлаштовуванню під себе як мінімум через абсолютну закритість системи.

Моя задача полягала у тому, щоб поринути у світ сучасних технологій та систем штучного інтелекту, проаналізувати поточні рішення реалізації штучного інтелекту, виявити їх сильні та слабкі сторони, і на основі цього аналізу створити власного розумного мобільного асистента. Цей процес – дуже довгий та важкий шлях, оскільки мені, як розробнику, потрібно для початку виконати чіткий аналіз ситуації, створити конкретний план дій,

визначити бажаний результат, а тільки на основі цих теоретичних даних почати діяти. Я зрозумів, що створювати сучасне рішення реалізації розумного асистента потрібно з урахуванням багатьох аспектів, серед яких сучасні проблеми та виклики під час розробки, вибір інструментів розробки, технологій, фреймворків, вподобання сучасних користувачів і так далі. Я зміг визначити, що для цього ідеально підходить поєднання гнучкого до налаштування та кастомізації фреймворку Rasa з абсолютним лідером у сфері створення інтерфейсних мобільних застосунків середовищем розробки Xcode з використанням мови програмування Swift та готового фреймворку SwiftUI. Окрім того, я дійшов до висновку, що найкращим способом реалізації подібного розумного асистента є клієнт-серверна архітектура, яка дозволяє усі складні та потужні обчислювальні операції надавати відповідному серверу, а додатку надати функціонал взаємодії з цим сервером та візуалізації процесу співпраці з асистентом. Таким підхід дозволив мені раціонально розподілити зони відповідальності, написати простий, але дуже гнучкий код та створити такий серверний компонент, який налаштовувати та покращувати одне задоволення для розробника.

У результаті цієї роботи ми дізналися про поняття та еволюцію розумних асистентів, класифікували розумних помічників, проаналізували сучасні підходи до їх розробки, розглянули фреймворк Rasa як інструмент для створення діалогового розумного асистента, познайомились із методологією дослідження та аналізу застосованих технологій. Також було проаналізовано сучасні приклади інтелектуальних помічників, обґрунтувано вибір клієнт-серверної архітектури, були порівняні варіанти комунікації між мобільним додатком та сервером, було розглянуто інструменти розробки Xcode, Swift та SwiftUI, визначено принцип взаємодії Rasa із ChatGPT за допомогою кастомних екшенів.

Нами успішно було розроблено свого власного розумного мобільного асистента, який вміє повноцінно спілкуватись із користувачем а також запам’ятовувати історію спілкування. Так, на виході ми отримали досить

простого асистента, який має цілком маленький набір функцій. Проте це не проблема, адже наведене рішення реалізації розумного асистента дозволяє досить просто, гнучко та ефективно налаштовувати та покращувати кожен окремий компонент нашого інтелектуального помічника у майбутній розробці.

ПЕРЕЛІК ПОСИЛАНЬ

1. Swift Language Guide. – Apple Developer Documentation. – Режим доступу: https://developer.apple.com/documentation/swift
2. SwiftUI Documentation. – Apple Developer Documentation. – Режим доступу: https://developer.apple.com/documentation/swiftui
3. Apple Inc. Human Interface Guidelines. – Режим доступу: https://developer.apple.com/design/human-interface-guidelines/
4. Rasa Open Source Documentation. – Режим доступу: https://rasa.com/docs/rasa/
5. OpenAI API Reference. – Режим доступу: https://platform.openai.com/docs
6. Render Documentation. – Режим доступу: https://render.com/docs
7. GitHub Docs – Introduction to GitHub. – Режим доступу: https://docs.github.com/en/get-started
8. Git Version Control. Pro Git Book by Scott Chacon and Ben Straub. – Режим доступу: https://git-scm.com/book/en/v2
9. RESTful Web Services. Richardson, L., & Ruby, S. (2008). RESTful Web Services. O’Reilly Media. – 454 с.
10. WebSocket API (MDN Web Docs). – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
11. NLP Natural Language Processing. Bird S., Klein E., Loper E. Natural Language Processing with Python. O’Reilly, 2009. – 504 ст.
12. Jurafsky D., Martin J.H. Speech and Language Processing. 3rd ed. Pearson, 2023. – 1264 ст.
13. Large Language Models (LLMs). OpenAI Technical Guide. – Режим доступу: https://platform.openai.com/docs/guides/gpt
14. Client-Server Architecture Overview. IBM Developer Docs. – Режим доступу: https://developer.ibm.com/articles/cl-client-server/

15. Environment Variables. Twelve-Factor App Methodology. – Режим доступу: https://12factor.net/config
16. Chat Interface Design. Nielsen Norman Group. – Режим доступу: https://www.nngroup.com/articles/chatbots/
17. Rasa Custom Actions Docs. – Режим доступу: https://rasa.com/docs/rasa/custom-actions/
18. Dialog Manager Design. Gnewuch, U., Morana, S., Maedche, A. (2017).
19. OpenAI Authentication: Managing API Keys. – Режим доступу: https://platform.openai.com/account/api-keys
20. SwiftUI Architecture Guide. Raywenderlich.com. – Режим доступу: https://www.raywenderlich.com/4161005-swiftui-tutorial-for-ios
21. Jurafsky D., Martin J. H. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. 3rd ed. Draft. – Stanford University, 2025. – 1264 ст. – Режим доступу: https://web.stanford.edu/~jurafsky/slp3/
22. Bird S., Klein E., Loper E. Natural Language Processing with Python. – O'Reilly, 2009. – 504 ст.
23. OpenAI. API Reference. – Режим доступу: https://platform.openai.com/docs
24. GitHub. GitHub Docs. – Режим доступу: https://docs.github.com/
25. Raywenderlich.com. SwiftUI Tutorial for iOS. – Режим доступу: https://www.raywenderlich.com/
26. IBM. Client/server architecture. – IBM Documentation, 2025. – Режим доступу: https://www.ibm.com/docs/en/informix-servers/14.10?topic=communication-clientserver-architecture
27. Rasa Technologies GmbH. Training Data Format. – Rasa Documentation, 2025. – Режим доступу: https://rasa.com/docs/rasa/nlu-training-data/
28. Python Software Foundation. Python Docs. – Режим доступу: https://docs.python.org/

29. Stack Overflow. Як працюють environment variables. – Режим доступу: https://stackoverflow.com/questions/4906977/
30. Apple. Xcode Overview. – Режим доступу: https://developer.apple.com/xcode/

ДОДАТКИ

Додаток А

Код початкового екрану додатку

```swift
import SwiftUI

struct ContentView: View {

    @State var chatHistory: ChatHistory = ChatHistory()

    var body: some View {
        NavigationView {
            List {
                NavigationLink(destination: ChatTab()) {
                    Label {
                        Text("Go to chat")
                    } icon: {
                        ResizableImage(image: "assistant")
                    }
                }
                NavigationLink(destination: HistoryTab()) {
                    Label {
                        Text("Chats History")
                    } icon: {
                        ResizableImage(image: "book.closed", isSystemName: true)
                            .foregroundColor(.black)
                    }
                }
            }
            .navigationTitle("Menu")
        }
        .environmentObject(chatHistory)
    }
}

#Preview {
    ContentView()
}

```

Додаток Б

Код вікна чату з ботом

```swift
import SwiftUI

struct ChatTab: View {

    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var chatHistory: ChatHistory
    @State var rasaClient: RasaClient = RasaClient()

    @State var currentChat: Chat = Chat(name: "New chat", date: Date(), messages: [])
    @State var showInput: Bool = true
    @State var userMessage: String = ""
    @State var isLoading: Bool = false
    @State var showNameAlert: Bool = false

    @State var isTextFieldFocused: Bool = false

    var body: some View {
        VStack {
            ChatScroll(currentChat: $currentChat, isTextFieldFocused: $isTextFieldFocused)

            if showInput {
                InputField(isLoading: $isLoading, isTextFieldFocused: $isTextFieldFocused) {
                    sendMessage()
                }
            }
        }
        .padding()
        .customBackground()
        .navigationBarItems(trailing: Menu(content: {
            if showInput {
                Button(action: {
                    showNameAlert.toggle()
                }) {
                    Label(title: { Text("Save chat") }, icon: { ResizableImage(image: "square.and.arrow.down", isSystemName: true) })
                }
                Button(action: {
                    chatHistory.saveChat(currentChat)
                    withAnimation(.spring()) {
                        currentChat.messages = []
                    }
                }) {
                    Label(title: { Text("Save chat and clear") }, icon: { ResizableImage(image: "archivebox", isSystemName: true) })
                }
            }
            Button(role: .destructive, action: {
                chatHistory.deleteChat(with: currentChat.id)
                dismiss()
            }) {
                Label(title: { Text("Delete chat") }, icon: { ResizableImage(image: "trash", isSystemName: true) })
            }
        }, label: {
            Image(systemName: "ellipsis")
                .foregroundColor(.black)
        }))
        .navigationTitle(currentChat.name)
        .navigationBarTitleDisplayMode(.inline)
        .alert("Chat Name", isPresented: $showNameAlert) {
            TextField("", text: $currentChat.name)
                .onChange(of: currentChat.name) { newValue in
                    currentChat.name = String(newValue.trimmingCharacters(in: .newlines).prefix(15))
                }
            Button("Cancel") { }
            Button("Save") {
                chatHistory.saveChat(currentChat)
            }
        } message: {
            Text("Input new chat name here to save it.")
        }
    }

    func sendMessage() {
        guard !isLoading else { return }

        setLoading(true)
        let message = userMessage
        withAnimation(.spring()) {
            currentChat.messages.append(ChatMessage(message: userMessage, owner: .user))
            userMessage = ""
        }
        rasaClient.sendMessage(message) { response in
            let defaultResponse = "Вибачте, але щось пішло не так. Будь ласка, спробуйте ще раз написати мені."
            DispatchQueue.main.async {
                setLoading(false)
                withAnimation(.spring()) {
                    currentChat.messages.append(
                        ChatMessage(
                            message: response ?? defaultResponse,
                            owner: .rasa))
                }
            }
        }
    }

    func setLoading(_ status: Bool) {
        withAnimation(.spring()) {
            isLoading = status
        }
    }
}

#Preview {
    NavigationView {
        ChatTab()
            .environmentObject(ChatHistory())
    }
}
```

Додаток В

Код вікна історії чатів

```swift
import SwiftUI

struct HistoryTab: View {

    @EnvironmentObject var chatHistory: ChatHistory

    var body: some View {
        VStack {
            if chatHistory.history.isEmpty {
                Spacer()

                Image("nochat")
                    .resizable()
                    .scaledToFit()
                    .padding()

                Spacer()
            } else {
                List {
                    ForEach(chatHistory.history) { chat in
                        NavigationLink(destination: ChatTab(currentChat: chat, showInput: false)) {
                            HStack {
                                ResizableImage(image: "assistant")
                                Text(chat.name)
                                Text(Tools.formatDateToShortString(chat.date))
                            }
                        }
                    }
                }
            }
        }
        .customBackground()
        .navigationTitle(Text("History"))
        .navigationBarTitleDisplayMode(.inline)
    }
}

#Preview {
    NavigationView {
        HistoryTab()
    }
    .environmentObject(ChatHistory())
}

```

Додаток Г

Код тла додатку

```swift
import SwiftUI

struct Background: View {
    var body: some View {
        Color.bgWhite
            .ignoresSafeArea()
            .tapCloseKeyboard()
    }
}

#Preview {
    Background()
}
```

Додаток Г

Код візуального контейнера повідомлення користувача та бота

```swift
import SwiftUI

struct MessagePanel: View {

    let message: ChatMessage

    var body: some View {
        HStack(alignment: .top, spacing: 16) {
            if message.owner == .user {
                Spacer()
            } else {
                ResizableImage(image: "assistant", size: 24)
                    .padding(8)
                    .background() {
                        Circle()
                            .fill(.black)
                    }
            }

            Text(message.message)
                .foregroundColor(.white)
                .font(.system(size: 16))
                .fontWeight(.semibold)
                .padding()
                .multilineTextAlignment(message.owner == .user ? .trailing : .leading)
                .background() {
                    RoundedRectangle(cornerRadius: 16)
                        .foregroundColor(message.owner == .user ? .accentColor : .gray)
                }

            if message.owner == .rasa {
                Spacer()
            } else {
                ResizableImage(image: "person.circle.fill", isSystemName: true, size: 40)
                    .foregroundColor(.black)
            }
        }
        .tapCloseKeyboard()
    }
}

#Preview {
    MessagePanel(message: ChatMessage(message: "Some preview text here", owner: .user))
}

```

Додаток Д

Код допоміжного візуального контейнера для картинок

```swift
import SwiftUI

struct ResizableImage: View {

    let image: String
    var isSystemName: Bool = false
    var size: CGFloat = 24

    var body: some View {
        finalImage
            .resizable()
            .frame(width: size, height: size)
    }

    var finalImage: Image {
        isSystemName ? Image(systemName: image) : Image(image)
    }
}

#Preview {
    ResizableImage(image: "nochat")
        .foregroundColor(.black)
}

```

Додаток Е

Код внутрішніх допоміжних інструментів

```swift
import Foundation

final class Tools {
    static func formatDateToShortString(_ date: Date) -> String {
        let formatter = DateFormatter()
        formatter.dateFormat = "d MMM, yyyy"
        formatter.locale = Locale(identifier: "en_US_POSIX")
        return formatter.string(from: date)
    }
}

```

Додаток Є

Код розширення для розробки інтерфейсу

```swift
import Foundation
import SwiftUI

extension View {
    func customBackground() -> some View {
        self.background {
            Background()
        }
    }

    func tapCloseKeyboard() -> some View {
        self.onTapGesture {
            UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder), to: nil, from: nil, for: nil)
        }
    }
}

```

Додаток Ж

Код логіки роботи з чатом

```swift
import Foundation

enum MessageOwner: Codable {
    case user, rasa
}

struct ChatMessage: Identifiable, Codable {
    var id: UUID = UUID()
    let message: String
    let owner: MessageOwner
}

struct Chat: Identifiable, Codable {
    var id: UUID = UUID()
    var name: String
    var date: Date
    var messages: [ChatMessage]
}

final class ChatHistory: ObservableObject {
    @Published var history: [Chat] = []

    private let userDefaults = UserDefaults.standard
    private let defaultsKey = "chatHistory"

    init () {
        if let savedData = userDefaults.data(forKey: defaultsKey) {
            do {
                history = try JSONDecoder().decode([Chat].self, from: savedData)
            } catch {
                print("Error loading chat history: \(error)")
            }
        }
    }

    func saveChat(_ chat: Chat) {
        if let chatIndex = history.firstIndex(where: { $0.id == chat.id }) {
            history[chatIndex] = chat
        } else {
            history.append(chat)
        }
        saveHistory()
    }

    func deleteChat(with id: UUID) {
        if let chatIndex = history.firstIndex(where: { $0.id == id }) {
            history.remove(at: chatIndex)
        }
        saveHistory()
    }

    private func saveHistory() {
        do {
            let data = try JSONEncoder().encode(history)
            userDefaults.set(data, forKey: defaultsKey)
        } catch {
            print("Error saving chat history: \(error)")
        }
    }
}
```

Додаток З

Код взаємодії додатку з Rasa

```swift
import Foundation

final class RasaClient {

    private let endpoint = URL(string: "https://my-rasa-assistant-ua.onrender.com/webhooks/rest/webhook")!

    func sendMessage(_ message: String, sender: String = "ios_user", completion: @escaping (String?) -> Void) {
        var request = URLRequest(url: endpoint)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")

        let payload: [String: Any] = [
            "sender": sender,
            "message": message
        ]

        do {
            request.httpBody = try JSONSerialization.data(withJSONObject: payload, options: [])
        } catch {
            print("❌ Ошибка при кодировании JSON: \(error)")
            completion(nil)
            return
        }

        let task = URLSession.shared.dataTask(with: request) { data, response, error in

            if let error = error {
                print("❌ Ошибка запроса: \(error)")
                completion(nil)
                return
            }

            guard let data = data else {
                print("❌ Нет данных в ответе")
                completion(nil)
                return
            }

            do {
                if let json = try JSONSerialization.jsonObject(with: data) as? [[String: Any]],
                   let firstResponse = json.first,
                   let text = firstResponse["text"] as? String {
                    completion(text)
                } else {
                    print("⚠️ Неожиданный формат ответа: \(String(data: data, encoding: .utf8) ?? "nil")")
                    completion(nil)
                }
            } catch {
                print("❌ Ошибка парсинга JSON: \(error)")
                completion(nil)
            }
        }

        task.resume()
    }
}

```

Додаток И

Код візуального компоненту чату

```swift
import SwiftUI

struct ChatScroll: View {

    @Binding var currentChat: Chat
    @Binding var isTextFieldFocused: Bool

    var body: some View {
        ScrollViewReader { scrollProxy in
            ScrollView {
                VStack(spacing: 16) {
                    ForEach(currentChat.messages) { message in
                        MessagePanel(message: message)
                            .id(message.id)
                    }
                    Color.clear
                        .frame(height: 5)
                        .id("bottom")
                }
            }
            .onChange(of: isTextFieldFocused) { newValue in
                if newValue {
                    withAnimation {
                        scrollProxy.scrollTo("bottom", anchor: .bottom)
                    }
                }
            }
            .onChange(of: currentChat.messages.count) { _ in
                withAnimation {
                    scrollProxy.scrollTo("bottom", anchor: .bottom)
                }
            }
        }
    }
}

#Preview {
    ChatScroll(currentChat: .constant(Chat(name: "", date: Date(), messages: [])), isTextFieldFocused: .constant(false))
}
```

Додаток І

Код компоненту вводу повідомлення та кнопки його відправки

```swift
import SwiftUI

struct InputField: View {

    @Binding var isLoading: Bool
    @Binding var isTextFieldFocused: Bool
    let sendMessage: () -> Void

    @FocusState var isTextFieldFocusedState: Bool
    @State var userMessage: String = ""

    var body: some View {
        HStack(alignment: .top, spacing: 16) {
            TextField("", text: $userMessage, prompt: Text("Напиши своє повідомлення!"), axis: .vertical)
                .font(.system(size: 16))
                .focused($isTextFieldFocusedState)
                .foregroundColor(.black)
                .padding()
                .background() {
                    RoundedRectangle(cornerRadius: 16)
                        .fill(.white)
                        .overlay {
                            RoundedRectangle(cornerRadius: 16)
                                .stroke()
                        }
                }

            Button(action: sendMessage) {
                Group {
                    if isLoading {
                        ProgressView()
                            .tint(.white)
                    } else {
                        Image(systemName: "bubble.fill")
                            .foregroundColor(.white)
                    }
                }
                .padding()
                .background {
                    Circle()
                }
            }
        }
        .onChange(of: isTextFieldFocusedState) { newValue in
            isTextFieldFocused = newValue
        }
    }
}
```