Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ___ від ____ ____ 2025 р.
Завідувач кафедри ММтаАД
_____ Струков В.М.
(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему Створення персонального Telegram-бот помічника

Захищено на засіданні ЕК № _____
протокол № ___ від ___ ___ 2025 р.
Оцінка ___ / _____
Голова ЕК
_____ Фесенко Г.В.
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КС-42
спеціальності:
122 Комп'ютерні науки
(шифр і назва спеціальності підготовки)
Кошовий Максим Вячеславович
(прізвище, ім'я та по батькові, підпис)
Керівник к.т.н., доцент Лисицький К.Є.
(ступень, звання, прізвище та ініціали, підпис)
Рецензент к.т.н., доцент Нарєжній О.П.
(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра математичного моделювання та аналізу даних
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Освітня програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.О. завідувача кафедри ММтаАД

_____ Володимир СТРУКОВ
підпис ім'я, прізвище

«____» _____20___ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЕКТ)

Кошового Максима Вячеславовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи Створення персонального Telegram-бот помічника

керівник роботи Лисицький Костянтин Євгенійович, к.т.н, доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "16" квітня 2025року № 4101-5/962

2. Строк подання студентом роботи 2 червня 2025 року

3. Перелік питань, які потрібно розробити
ознайомитись з особливостями створення чат-ботів в Telegram; провести огляд існуючих методів розробки цифрових асистентів, познайомитися з алгоритмом роботи і методиками розробки ботів, що розуміють природню мову; створити прототип Telegram-бота помічника; провести програмну реалізацію та експериментальне дослідження використання створеного прототипу.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).	лютий-березень
2	Пошук та аналіз інформаційних джерел за темою КР.	середина березня
3	Визначення особливостей ботів що використовують великі мовні моделі.	кінець березня
4	Розробка прототипу бота помічника.	квітень
5	Програмна реалізація Telegram-бота з використанням великої мовної моделі для обробки запитів.	кінець квітня
6	Перевірка та тестування можливостей розробленого прототипу.	квітень-травень
7	Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ.	травень

5. Дата видачі завдання 10 жовтня 2024 року

Студент М.В. Кошовий _____
ініціали, прізвище підпис

Керівник роботи К.Є. Лисицький _____
ініціали, прізвище підпис

АНОТАЦІЯ

Пояснювальна записка до дипломної роботи бакалавра: 69с., 42 рис., 1 додаток, 27 джерел.

В роботі досліджується та виконується розробка персонального Telegram-бот помічника, що використовує велику мовну модель для інтелектуальної обробки запитів користувача та ефективної взаємодії з інтегрованими сервісами Gmail, Google Calendar, Notion, для підвищення особистої продуктивності та автоматизації рутинних завдань.

У процесі дослідження застосовано методи побудови оркестрованих LLM-агентів, модульне архітектурне проєктування, а також інструменти інтеграції зовнішніх сервісів через їх програмні інтерфейси. Основними використаними технологіями є мова програмування Python, фреймворки LangChain і LangGraph, мовна модель Gemini 2.0 Flash, а також платформи Telegram Bot API, Google Cloud API, Notion API, Tavily Search API.

У результаті реалізовано функціональний прототип Telegram-бот помічника, здатного розуміти запити користувача написані природною мовою, виконувати завдання з управління електронною поштою, подіями календаря, нотатками та завданнями, а також надавати відповіді на питання загального характеру. Новизна роботи полягає в поєднанні оркестрації агентів через LangGraph із реальними прикладними сценаріями для щоденного планування та комунікації.

Розроблений бот-помічник може бути рекомендований для персонального використання з метою підвищення ефективності управління особистими справами та інформацією. Результати роботи можуть також слугувати основою для подальших наукових досліджень та розробки більш досконалих інтелектуальних асистентів, розширення їх функціональності шляхом додавання інтеграцій з новими сервісами та можливостями.

Ключові слова: БОТ, ПЕРСОНАЛЬНИЙ ПОМІЧНИК, ШТУЧНИЙ ІНТЕЛЕКТ, ВЕЛИКА МОВНА МОДЕЛЬ, АГЕНТ, АВТОМАТИЗАЦІЯ.

ABSTRACT

The qualification thesis comprises: 69p., 42 fig., 1 appendix, 27 references.

The objective of this work is to develop a personal Telegram assistant bot that leverages a large language model for intelligent processing of user queries and effective interaction with integrated services such as Gmail, Google Calendar, and Notion, aiming to enhance personal productivity and automate routine tasks.

The research applies methods of orchestrated LLM-agent construction, modular architectural design, and tools for integration with external services programming interfaces. The primary technologies used include the Python programming language, the LangChain and LangGraph frameworks, the Gemini 2.0 Flash language model, as well as the Telegram Bot API, Google Cloud API, Notion API, and Tavily Search API.

As a result, a functional prototype of the Telegram assistant bot has been implemented. It is capable of understanding user queries expressed in natural language, performing tasks related to email management, calendar events, notes and task tracking, and responding to general knowledge questions. The novelty of this work lies in combining LangGraph-based agent orchestration with real-world application scenarios for daily planning and communication.

The developed assistant bot can be recommended for personal use to improve the efficiency of managing personal affairs and information. The outcomes of this project may serve as a foundation for further academic research and the development of more advanced intelligent assistants, with extended functionality through the integration of additional services and capabilities.

Keywords: BOT, PERSONAL ASSISTANT, ARTIFICIAL INTELLIGENCE, LANGUAGE MODEL, AGENT, AUTOMATION.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

БД – База Даних

МН – Машинне Навчання

ШІ – Штучний Інтелект

API – Application Programming Interface

LLM – Large Language Model

MCP – Model Context Protocol

NLP – Natural Language Processing

ВСТУП

У сучасному цифровому світі, що характеризується стрімким зростанням обсягів інформації та цифрових взаємодій, ефективне управління особистими даними, завданнями та комунікаціями стає ключовим фактором продуктивності та успішності. Інформаційне перевантаження та необхідність оперувати численними програмними засобами й сервісами створюють значні виклики для користувачів. У відповідь на ці виклики активно розвивається напрямок інтелектуальних персональних асистентів, покликаних спростити повсякденні завдання та забезпечити швидкий доступ до необхідної інформації. Активний розвиток великих мовних моделей (LLM) та інтеграційних платформ відкрив нові можливості для створення персоналізованих ботів, що можуть не лише взаємодіяти з користувачем природною мовою, а й здійснювати обробку запитів у реальному часі, працюючи із зовнішніми сервісами.

Предметною областю даної дипломної роботи є розробка програмного забезпечення, зокрема бота, з використанням технологій штучного інтелекту. Сучасний стан розвитку цих технологій, особливо в галузі обробки природної мови (NLP) та LLM, відкрив нові горизонти для створення значно більш досконалих та функціональних чат-ботів і віртуальних помічників. Великі мовні моделі, такі як GPT, Gemini та аналогічні, здатні розуміти контекст, генерувати людиноподібний текст, виконувати складні інструкції та навіть навчатися в процесі взаємодії. Це дозволяє переходити від простих скриптованих ботів до справді інтелектуальних систем, здатних надавати персоналізовану допомогу. Telegram, як однин з найпопулярніших месенджерів у світі, надає зручний та широкодоступний інтерфейс для взаємодії з такими помічниками.

Незважаючи на значний прогрес, необхідність ефективно оперувати електронною поштою, планувати свій час у календарі, структурувати знання та нотатки, а також швидко знаходити відповіді на різноманітні питання,

спонукає до пошуку інтелектуальних інструментів, здатних автоматизувати рутинні операції та надавати комплексну підтримку. Існуючі програмні рішення, такі як окремі клієнти для пошти, календарі, системи нотаток та загальнодоступні пошукові системи, хоча й виконують свої функції, часто залишаються розрізненими. Багато популярних віртуальних асистентів, таких як Siri та Google Assistant, хоч і є потужними, проте можуть бути недостатньо гнучкими для інтеграції з усім спектром специфічних інструментів, якими користується конкретна людина. Спеціалізовані боти, з іншого боку, часто обмежені вузьким набором функцій.

Таким чином, існує виражена потреба у створенні персоналізованих помічників, які б глибоко інтегрувалися в робочий процес користувача. Серед проблеми, що потребують вирішення, є фрагментація інструментів, інформаційне перенавантаження та необхідність автоматизації рутинних справ. Перша проблема полягає в тому, що користувачі змушені переключатися між різними додатками, зокрема поштою, календарем, нотатками, месенджерами, для виконання повсякденних завдань, що знижує ефективність. За рахунок інформаційного перенавантаження виникають складнощі з відстеженням важливої інформації та завдань у великому потоці даних. Також ще одним наслідком такого перевантаження є велика кількість дій, пов'язаних з управлінням інформацією та плануванням, які хотілось би мати змогу автоматизувати. У відповідності до окресленої проблеми та необхідності її вирішення, можна сформулювати основні елементи постановки задачі дипломної роботи.

Об’єктом дослідження є процес взаємодії користувача з цифровими інформаційними потоками та інструментами персональної продуктивності в умовах використання сучасних технологій штучного інтелекту, зокрема великих мовних моделей, через інтерфейс месенджера Telegram.

Предметом дослідження є методи, моделі та програмні засоби для розробки персонального Telegram-бот помічника, здатного інтегруватися з

різними сервісами, та обробляти запити користувача природною мовою за допомогою великої мовної моделі.

Метою дипломної роботи є розробка та програмна реалізація персонального Telegram-бот помічника, що використовує велику мовну модель для інтелектуальної обробки запитів користувача та ефективної взаємодії з інтегрованими зовнішніми сервісами керування поштою, календарем та нотатками, з метою підвищення особистої продуктивності користувача та спрощення його щоденних завдань.

У даній дипломній роботі пропонується розробка саме такого персонального Telegram-бот помічника. Основна ідея полягає у створенні програмного застосунку, який би виступав як єдина точка взаємодії для користувача з його цифровим простором. Використання месенджерів, зокрема Telegram, як єдиної точки доступу до персонального помічника є логічним кроком, оскільки месенджери вже стали центральним хабом для комунікації та споживання інформації для багатьох користувачів. Бот буде розроблено на мові програмування Python з використанням сучасних бібліотек та фреймворків, таких як LangChain та LangGraph. Ці інструменти дозволяють ефективно будувати ланцюжки обробки запитів, керувати взаємодією з великими мовними моделями, підключати зовнішні інструменти через прикладний програмний інтерфейс (API) та створювати програмних агентів, здатних до складних міркувань та послідовного виконання дій для досягнення поставленої мети. Розроблюваний бот зможе:

- взаємодіяти з поштовим сервісом Gmail;
- керувати подіями в Google Calendar;
- працювати з нотатками в базі даних (БД) у Notion;
- надавати стислі відповіді на запитання загального характеру.

Ключовою особливістю запропонованого рішення є поєднання модульної структури з інтелектуальним підходом до обробки запитів – тобто здатністю розуміти природню мову, аналізувати контекст, зберігати стан діалогу та виконувати логічні операції на основі намірів користувача.

Актуальність даної роботи зумовлена кількома факторами. По-перше, зростаюча складність цифрового життя та постійне збільшення обсягів інформації вимагають нових, більш ефективних інструментів для управління нею. Персональні інтелектуальні помічники є одним з найбільш перспективних напрямків для вирішення цієї проблеми. По-друге, стрімкий розвиток великих мовних моделей відкриває безпрецедентні можливості для створення чат-ботів, здатних до глибокого розуміння запитів користувача та надання релевантної допомоги. По-третє, платформа Telegram є надзвичайно популярною та зручною для користувачів, що робить розробку бота для неї доцільною з точки зору доступності та поширення. Нарешті, потреба в інтегрованих та персоналізованих рішеннях, які б адаптувалися до конкретних інструментів та робочих процесів користувача, залишається високою.

Для досягнення поставленої мети необхідно вирішити наступні завдання дослідження:

- провести аналіз предметної області;
- обґрунтувати вибір програмних засобів та технологій;
- розробити архітектуру персонального Telegram-бот помічника;
- реалізувати програмні модулі бота для взаємодії з інтегрованими сервісами;
- інтегрувати велику мовну модель для обробки запитів;
- розробити функціонал для надання відповідей на запитання загального характеру;
- провести тестування розробленого бота;
- проаналізувати результати роботи та сформулювати висновки.

Послідовне виконання цих завдань дозволить створити комплексне програмне рішення, що відповідатиме меті дипломної роботи, та надати оцінку його ефективності.

Наукова новизна роботи проявляється в застосуванні комбінації фреймворків LangChain та LangGraph для побудови архітектури персонального Telegram-бот помічника, що дозволяє створювати гнучкі та

керовані ланцюжки взаємодії з LLM та зовнішніми сервісами. Таке рішення забезпечує можливість реалізації складних сценаріїв обробки запитів через систему агентів із потенціалом масштабування такої системи при її подальшому розвитку.

Сенс роботи полягає у створенні програмного прототипу, який продемонструє переваги використання великих мовних моделей та сучасних інструментів розробки для створення потужного та гнучкого персонального помічника. Успішна реалізація проєкту дозволить не лише отримати практично корисний інструмент, але й зробити внесок у розуміння можливостей та викликів створення інтелектуальних персональних асистентів. Подальша робота включатиме детальний огляд технологій, опис процесу проєктування та реалізації, а також аналіз отриманих результатів.

1 АНАЛІЗ ТЕНДЕНІЙ РОЗРОБКИ ПЕРСНАЛЬНИХ АСИСТЕНТІВ

1.1 Еволюція та сучасний стан помічників

Ідея створення автоматизованих помічників, здатних допомагати людині у виконанні різноманітних завдань, має довгу історію, що сягає корінням у ранні концепції штучного інтелекту. Перші спроби були обмежені технологічними можливостями того часу і часто реалізовувалися у вигляді експертних систем або простих програм, що діяли за жорстко заданими алгоритмами, як це описано в роботах [1][2]. Проте, саме вони заклали фундамент для майбутніх розробок у цій галузі.

З розвитком комп'ютерних технологій, зокрема збільшенням обчислювальних потужностей, появою Інтернету та прогресом у галузі ШІ, концепція персонального асистента почала набувати більш реальних рис. Справжній прорив у розвитку персональних інтелектуальних асистентів стався з появою та вдосконаленням технологій машинного навчання (МН), обробки природної мови та, особливо в останні роки, великих мовних моделей, про що висловлюються автори [3][4][5]. Ці технології дозволили перейти від простих командно-орієнтованих інтерфейсів до систем, здатних розуміти контекст діалогу, обробляти складні запити, виражені природною мовою, та генерувати змістовні, людиноподібні відповіді.

Спираючись на дослідження [6], класифікацію сучасних персональних інтелектуальних асистентів можна виконати за кількома ознаками:

а) за типом взаємодії:

- голосові асистенти, що активуються та керуються переважно за допомогою голосових команд;
- текстові асистенти, взаємодія з якими відбувається через текстові повідомлення, які часто інтегруються в месенджери, веб-сайти та мобільні додатки;
- мультимодальні асистенти, що підтримують кілька способів взаємодії.

б) за призначенням та функціональністю:

- універсальні асистенти, що прагнуть охопити широкий спектр завдань;
- спеціалізовані асистенти, які орієнтовані на вирішення конкретних завдань у певній галузі.

в) за платформою та способом розгортання:

- вбудовані в операційні системи;
- автономні додатки;
- боти для месенджерів.

Ключовими функціями, які очікуються від сучасного персонального асистента, особливо в контексті підвищення особистої продуктивності згідно з [7], є:

- управління інформацією – пошук, агрегація, структурування та надання інформації з різних джерел;
- управління завданнями та часом – планування подій у календарі, ведення списку справ;
- автоматизація рутинних справ – перевірка пошти, допомога в написанні та відправленні електронних листів, управління контактами.

Серед найвідоміших універсальних персональних асистентів можна виділити Apple Siri та Google Assistant. Siri, вперше представлена у 2011 році, стала одним із перших масових голосових помічників, інтегрованих в мобільну операційну систему [8]. Вона забезпечує взаємодію з функціями пристрою та вбудованими додатками. Google Assistant, запущений у 2016 році, вирізняється потужними можливостями пошуку та інтеграцією з екосистемою сервісів Google та здатністю підтримувати більш контекстуальні діалоги, як це зазначено в роботі [9].

Незважаючи на значні досягнення, ці популярні асистенти мають і певні обмеження. Часто вони тісно прив'язані до екосистем своїх розробників, що ускладнює глибоку інтеграцію з сервісами конкурентів або менш поширеними

спеціалізованими інструментами. Рівень персоналізації може бути недостатнім для користувачів зі специфічними потребами, а можливості кастомізації логіки роботи асистента зазвичай обмежені.

Водночас, спостерігається тенденція до розробки більш відкритих та гнучких рішень, які прагнуть надати користувачам та розробникам більше контролю над своїми даними та функціональністю асистентів, про що вказано в працях [3][5]. Саме в цьому контексті розробка ботів-асистентів, які використовують потужності сучасних LLM та інтегруються з різним набором інструментів, набуває особливої актуальності.

1.2 Технології та інструменти для розробки додатків на основі LLM

1.2.1 Великі мовні моделі як основа для обробки запитів

Великі мовні моделі – це класи моделей ШІ, які навчаються на величезних масивах текстових даних із метою розуміння, генерування, узагальнення та перекладу природної мови. Їх функціонування базується на архітектурі трансформерів, запропонованої дослідниками компанії Google у 2017 році [10]. Трансформери дозволяють ефективно обробляти контекст і залежності між словами у тексті, що дає змогу моделям з високою точністю відтворювати мовні закономірності та логіку людського мислення.

Можливості сучасних LLM охоплюють широкий спектр задач:

- генерація зв’язного тексту;
- створення підсумків;
- машинний переклад;
- автоматизована підтримка клієнтів;
- аналіз настроїв;
- моделювання поведінки користувача;
- побудова багатоструктурованих діалогових систем.

У контексті цифрових помічників LLM виступають ядром, що здатне інтерпретувати запити користувача в природній мові, адаптувати відповіді до контексту, запам’ятовувати інформацію, а також взаємодіяти з іншими сервісами чи базами знань через програмні інтерфейси.

Серед провідних LLM-платформ варто відзначити GPT-серію від OpenAI, зокрема моделі GPT-3.5 та GPT-4 [11]. Ці моделі відзначаються високою якістю генерації тексту, підтримкою мультимодальності, а також широкою інтеграцією з різними інструментами. Google Gemini – ще одна передова серія моделей, орієнтованих на точність і контекстуальність відповідей, а також тісну інтеграцію з екосистемою Google [12]. Компанія Meta розвиває відкриту лінійку моделей LLaMA, серед яких LLaMA 3 і LLaMA 4 [13]. Це потужні моделі з відкритим вихідним кодом, що активно застосовуються в академічних дослідженнях і корпоративних рішеннях.

Більшість популярних моделей, таких як GPT-4, Gemini 2.0 Flash або LLaMA 4, можуть бути використані через API, що надає розробникам можливість інтегрувати інтелектуальну обробку природної мови у власні додатки. Але за часту використання цих програмних інтерфейсів супроводжується певними обмеженнями – як-от кількість безкоштовних запитів, швидкість обробки чи доступ до конкретних можливостей, згідно до [11][12][13].

Крім стандартного використання LLM через API, останні дослідження у сфері когнітивного агентування привели до появи нових підходів до інтеграції мовних моделей у складні робочі процеси. Одним із таких підходів є ReAct (Reasoning and Acting) – архітектура, яка поєднує логічне міркування та взаємодію з середовищем. Згідно з дослідженням [14], ReAct дозволяє моделям одночасно будувати ланцюжки думок та виконувати зовнішні дії, що значно підвищує їхню ефективність у виконанні складних задач. Саме ці ідеї стали підґрунтям для появи фреймворків, таких як LangChain та LangGraph, які реалізують агентно-орієнтовану архітектуру на базі LLM.

1.2.2 Побудова LLM-додатків з LangChain та LangGraph

Однією з ключових технологічних основ розробки інтелектуального бота помічника є мова програмування Python. Вона має домінуюче становище у галузях штучного інтелекту, машинного навчання та обробки природної мови, за дослідженням [15]. Python має розвинену екосистему бібліотек, велике різноманіття документації, а також простий синтаксис, що прискорює розробку прототипів. Особливо важливо, що більшість сучасних фреймворків для роботи з великими мовними моделями – зокрема, LangChain та LangGraph – написані саме на Python, що забезпечує їх безшовну інтеграцію з LLM API та іншими сервісами.

LangChain – це фреймворк з відкритим вихідним кодом, який надає зручні інструменти для побудови застосунків на базі великих мовних моделей, відповідно до [16]. Його ключовою ідеєю є впорядкування взаємодії між LLM, зовнішніми інструментами та користувачем. За допомогою LangChain можна будувати ланцюжки операцій, що складаються з кількох кроків обробки запиту – наприклад, генерація відповіді, виклик зовнішнього API, збереження даних. Основні концепції LangChain включають:

- *promts* – шаблони інструкцій, що адаптуються до контексту;
- *chains* – послідовності кроків, які комбінуються у ланцюжок дій;
- *agents* – модулі, які приймають рішення про подальші дії на основі отриманих даних;
- *tools* – інструменти, що виконують певну дію, які агент може викликати;
- *memory* – збереження історії розмов для підтримки контексту для довгих ланцюжків дій.

Таким чином, LangChain не просто дозволяє використовувати LLM як чорну скриньку, а забезпечує керовану, контекстно-орієнтовану логіку роботи.

LangGraph є надбудовою над LangChain, яка базується на ідеї моделювання логіки LLM-додатка у вигляді орієнтованого графа станів, згідно

з [17]. Це дозволяє розробникам створювати гнучких та керованих агентів, які можуть переходити між різними станами залежно від результатів попередніх дій. Кожна вершина графа представляє певну логіку або крок агента, а ребра – правила переходу між цими станами. На відміну від простих ланцюгів у LangChain, LangGraph дає змогу:

- реалізувати умовні гілки логіки на основі відповіді моделі або стану контексту;
- будувати циклічні або рекурсивні сценарії;
- масштабувати агентів з великою кількістю інструментів і складною поведінкою без втрати читабельності або керованості коду.

Завдяки цим можливостям, LangGraph є надзвичайно актуальним для побудови багатоінструментальних агентів, які не лише відповідають на запити, а й здатні інтегруватися з різними зовнішніми сервісами, і підтримувати складні діалогові сценарії.

Але попри те, що LangGraph підтримує агентоподібну поведінку через переходи між станами, важливо розуміти, що його основна ціль – оркестрація робочих процесів, а не реалізація повної агентної автономії. Згідно з аналітичним оглядом в роботі [18], LangGraph дозволяє будувати чітко визначені, контрольовані потоки з умовною логікою, але не підтримує динамічне змінення цілей чи самостійну ініціативу з боку агента, як це роблять фреймворки на кшталт AutoGPT, BabyAGI чи AutoGen. Це означає, що LangGraph найкраще підходить для створення керованих, передбачуваних сценаріїв із високим ступенем надійності, а не для розробки агентів, які живуть своїм життям у складному середовищі. Такий підхід, з огляду на цілі дипломної роботи, є більш доцільним, адже дозволяє забезпечити контроль над логікою асистента, уникати неочікуваної поведінки, а також забезпечити інтеграцію з персональними сервісами користувача.

Саме така архітектура була обрана для реалізації бота помічника в межах даної роботи, оскільки вона дозволяє досягнути достатньої гнучкості, розширюваності та адаптивності програмної системи.

1.3 Розробка чат-ботів на платформі Telegram

Telegram є однією з найпопулярніших платформ для створення чат-ботів завдяки відкритому API, високій стабільності, зручності інтеграції та широкому охопленню аудиторії. Платформа активно використовується не лише для спілкування, а й для автоматизації бізнес-процесів, реалізації сервісів підтримки, інформування користувачів, а також для створення персональних асистентів, згідно з [19]. Telegram має низку переваг як платформа для створення ботів:

- відкритий та добре задокументований API для створення ботів (Telegram Bot API);
- висока швидкість доставки повідомлень і стабільність платформи;
- підтримка багатьох мов програмування;
- вбудовані інструменти безпеки, шифрування та обмеження доступу.

Telegram Bot API – простий у використанні інтерфейс, який дозволяє створювати ботів, що працюють у середовищі Telegram [20]. Інтерфейс базується на REST-архітектурі. Його основні можливості включають:

- прийом і надсилання текстових повідомлень, зображень, відео, документів;
- створення меню, кнопок швидкого доступу;
- підтримку команд та їх обробку;
- інтеграцію з іншими API через запити до зовнішніх серверів.

Telegram вважається однією з найзручніших платформ для створення персональних асистентів завдяки поєднанню низки технічних і користувацьких переваг, відповідно до [19]. Насамперед, Telegram є кросплатформеним сервісом, що дозволяє одному й тому самому боту працювати однаково стабільно на різних операційних системах. Це забезпечує постійну доступність бота незалежно від типу пристрою, яким користується людина. Додатково, асистенти на базі Telegram можуть взаємодіяти з контактами користувача, групами або каналами, що відкриває шлях до

створення як індивідуальних, так і колективних помічників. Мобільність платформи гарантує, що користувач завжди має під рукою інструмент для швидкого збереження інформації, отримання нагадувань чи комунікації зі сторонніми сервісами. Одним із важливих чинників також є підтримка мультимедійних повідомлень і розширених форматів взаємодії, що дозволяє створювати інтерфейс бота максимально зручним і гнучким.

Існує чимало прикладів Telegram-ботів, які виконують функції персональних асистентів чи забезпечують інтеграцію із зовнішніми сервісами:

– SkeddyBot – бот-нагадувач, який використовує повідомлення написані природною мовою для створення нагадувань;
– GmailBot – бот, що дозволяє читати вхідні листи, відповідати на них, створювати поштові фільтри;
– ChatGPT_Bot — приклад інтеграції різних моделей МН з Telegram через програмні інтерфейси, що дозволяє відповідати на питання, генерувати зображення чи музику.

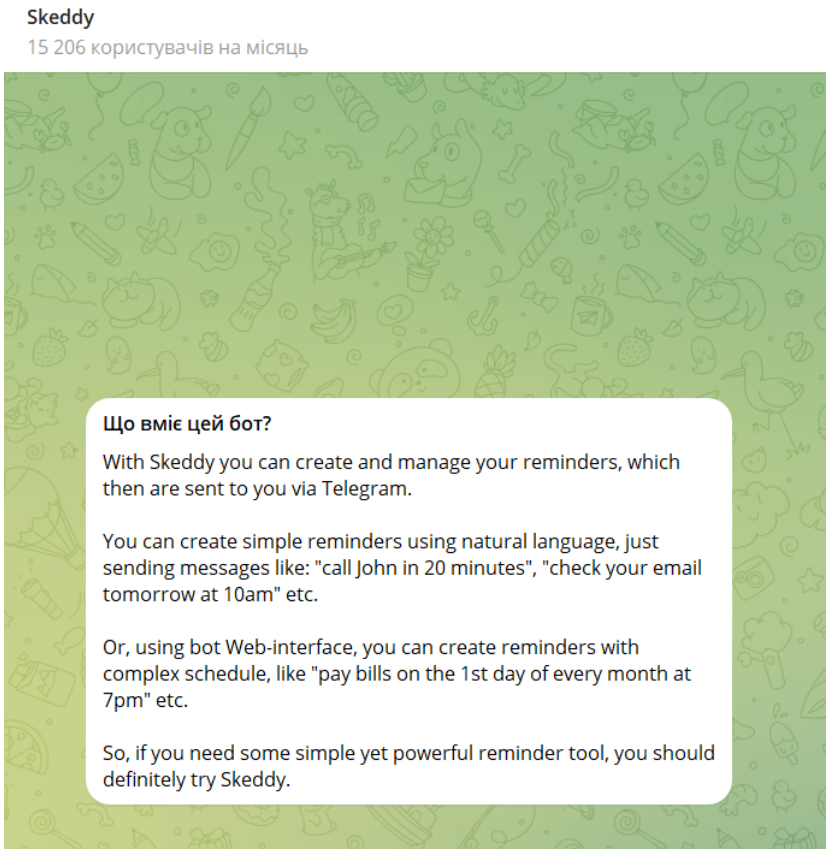


Рисунок 1.1 – SkeddyBot

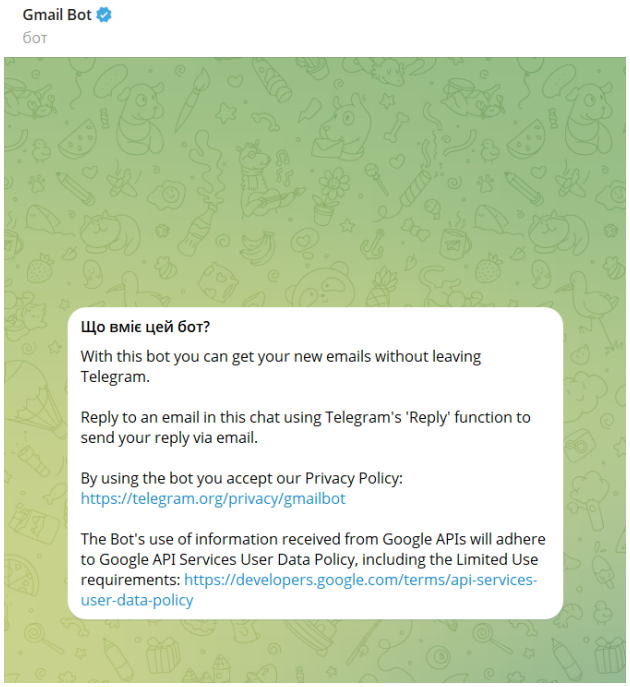


Рисунок 1.2 – GmailBot

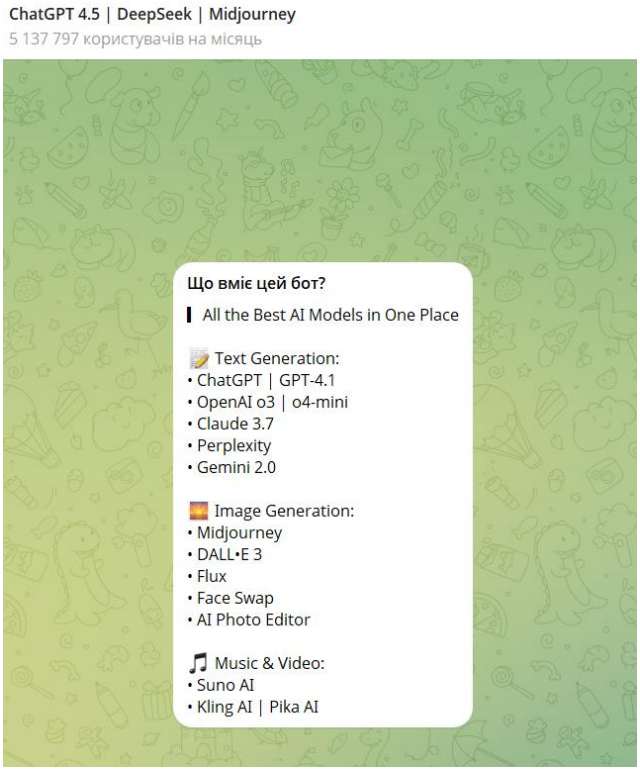


Рисунок 1.3 – ChatGPT_Bot

Завдяки широким можливостям Telegram Bot API та доступності середовища Telegram, розробка персонального бота-помічника стає не лише доцільною, а й ефективною стратегією для створення індивідуалізованого сервісу з підтримкою обробки повідомлень за допомогою LLM.

1.4 Специфіка інтеграції із зовнішніми сервісами

Створення справді функціонального та корисного персонального асистента неможливе без його глибокої інтеграції з тими цифровими інструментами та сервісами, якими користувач послуговується щоденно. У контексті даної дипломної роботи, такими ключовими сервісами є Gmail для управління електронною поштою, Google Calendar для організації запланованих подій, і Notion для ведення нотаток та списку власних завдань. Основою такої інтеграції є використання програмних інтерфейсів, які надаються цими сервісами. Однак, процес інтеграції через API супроводжується низкою загальних викликів та вимагає застосування специфічних підходів для кожного окремого сервісу.

Використання API стало де-факто стандартом для взаємодії між різними програмними системами в сучасній цифровій екосистемі. Вони дозволяють стороннім додаткам, таким як розроблюваний персональний асистент, отримувати доступ до даних та функціональності інших сервісів від імені користувача або від свого власного імені. Однак, ефективна та безпечна інтеграція через API вимагає вирішення кількох фундаментальних завдань.

Одним із найважливіших аспектів є автентифікація та авторизація. Автентифікація – це процес перевірки ідентичності клієнта який намагається отримати доступ до API. В нашому випадку клієнтом виступає бот. Авторизація – це процес надання цьому клієнту дозволу на виконання певних дій або доступ до певних ресурсів після успішної автентифікації. Існує кілька методів автентифікації, таких як використання API-ключів, Basic Auth, та найпоширеніший сьогодні стандарт для делегованого доступу – OAuth 2.0, згідно із ресурсом [21]. Саме OAuth 2.0 є ключовим для доступу до даних користувача в сервісах Google, за даними [22]. Цей протокол дозволяє додатку отримувати доступ до ресурсів користувача без необхідності знати його логін та пароль. Відповідно до [21][22], якщо не вдаватись у глибокі подробиці, то процес OAuth 2.0 включає направлення користувача на сервер авторизації, де він входить у свій акаунт та надає згоду на доступ додатку до певних даних,

визначених через області видимості. Після успішної згоди сервер авторизації повертає додатку авторизаційний код, який той обмінює на токен доступу та, опціонально, токен оновлення. Токен доступу є короткоживучим і використовується для авторизації кожного запиту до API, тоді як токен оновлення дозволяє отримувати новий токен доступу після закінчення терміну дії попереднього без повторної участі користувача. Ознайомившись з певними особливостями доступу до API, перейдемо до огляду можливостей програмних інтерфейсів конкретних сервісів, якими щоденно послуговуються користувачі.

Gmail API надає розробникам потужні можливості для програмного доступу до поштових скриньок користувачів Gmail. Для персонального асистента, здатність читати, аналізувати, шукати та надсилати електронні листи є однією з ключових функцій. Доступу до даних користувача в Gmail повністю покладається на протокол OAuth 2.0, детально описаний до цього. Серед можливостей даного API, згідно з [23], найбільш всього ми зацікавлені в читанні листів та їх надсиланні. Програмний інтерфейс дозволяє отримувати списки повідомлень з можливістю фільтрації через параметри запиту. Це значно розширює пошукові можливості які можна інтегрувати в персонального помічника. Отримавши список повідомлень можна завантажити повний вміст кожного листа, включаючи його різні елементи. Сам вміст можна завантажити в бажаному форматі, але текстовий формат є найбільш універсальним. Функціонал пов’язаний з надсиланням листа реалізує лише його безпосередню відправку, тобто задача формування та кодування листа, для його передачі в тілі запиту, покладається на сам додаток. Окрім цих основних функцій, Gmail API також дозволяє програмно керувати мітками, працювати з чернетками, керувати налаштуваннями фільтрів, а також організовувати листи в ланцюжки.

Google Calendar API надає програмний доступ до календарів користувачів, дозволяючи створювати, переглядати, оновлювати та видаляти події, а також керувати самими календарями та їхніми налаштуваннями, як це

зазначається в [24]. Для персонального асистента ця інтеграція є ключовою для реалізації функцій планування. Аналогічно до Gmail API, доступ до даних Google Calendar здійснюється через протокол OAuth 2.0. Серед можливостей які важливо інтегрувати в бота, слід виділити перегляд і пошук запланованих подій, та створення нових подій. Для виконання пошуку надається великий спектр фільтрів, але основними є ті, що відповідають за час проведення цих подій. Із списку відфільтрованих подій можна отримати необхідну інформацію про них у звичайному текстовому форматі. Для створення нової події через програмний інтерфейс необхідно заповнити такі атрибути як назва події, її опис, час початку та завершення. Надзвичайно важливим є правильний формат дат. Даний API також дозволяє додавати учасників, вказувати місцезнаходження та налаштовувати нагадування, але це не є обов'язковим. Окрім створення та перегляду, за допомогою Google Calendar API можна також оновлювати існуючі події, водночас дозволяється змінювати будь-які їхні атрибути, та видаляти події. Серед більш складних функцій є можливість працювати з кількома календарями користувача, отримувати списки календарів, створювати нові та керувати їхніми налаштуваннями спільного доступу.

Notion API відкриває можливості для програмної взаємодії з гнучким та багатофункціональним робочим простором Notion, що дозволяє користувачам створювати нотатки, бази даних, керувати проєктами тощо. Для персонального асистента інтеграція з даним сервісом є важливим для автоматизації роботи із списком завдань та структурованою інформацією. Автентифікація в Notion API відрізняється від підходу Google і базується на токенах інтеграції. Розробник створює інтеграцію у налаштуваннях свого Notion-акаунту. Кожна інтеграція отримує унікальний секретний токен, який потім використовується для авторизації всіх запитів до API шляхом передачі його у заголовку. Важливо, що для того, щоб інтеграція мала доступ до певних сторінок або баз даних у Notion, користувач повинен явно надати їй цей доступ через меню відповідного ресурсу. Але слід зазначити що такий варіант

автентифікації підходить лише для внутрішніх інтеграцій що спеціалізуються на власному використані. Для публічних інтеграцій передбачено використання протоколу OAuth 2.0, відповідно до [25]. Серед можливостей інтерфейсу нас найбільше цікавить робота з базами даних та сторінками, оскільки кожен запис у БД в Notion є окремою сторінкою. Запити до БД є основним методом для отримання інформації з неї. В запиті дозволяється вказувати різні фільтри для більш гнучкого пошуку необхідних даних. Фільтри можуть бути досить складними, поєднуючи кілька умов. Результати запиту можна також сортувати за значеннями певних властивостей. При роботі із сторінками, тобто записами в БД, надається можливість додавати нові дані та оновлювати існуючі. За таких умов слід розуміти, що для правильної маніпуляції із записами потрібно знати точну структуру бази даних, тобто назви та типи всіх стовпчиків та значень які там зберігаються. Одним із викликів при роботі з Notion API є розуміння його ієрархічної структури та специфічного формату даних для різних типів властивостей та блоків. Конструювання об'єктів фільтрів для запитів до БД також може бути нетривіальним завданням, що вимагає точного дотримання документації програмного інтерфейсу, згідно з [25].

Успішна інтеграція з зовнішніми сервісами, такими як Gmail, Google Calendar та Notion, є наріжним каменем у створенні по-справжньому ефективного та багатофункціонального персонального асистента. Така взаємодія дозволяє персональному асистенту виступати в ролі єдиного центру управління для користувача, автоматизувати рутинні операції, надавати консолідовану інформацію та значно підвищувати загальну продуктивність. Хоча сучасні фреймворки та бібліотеки можуть абстрагувати частину цієї складності, глибоке розуміння базових принципів роботи з API залишається важливим фактором при розробці сучасних додатків.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА TELEGRAM-БОТ ПОМІЧНИКА

2.1 Загальний підхід до розробки та вибір інструментарію

Розробка персонального Telegram-бот помічника, здатного інтелектуально обробляти запити користувача та взаємодіяти з різними зовнішніми сервісами, є комплексним завданням, що вимагає ретельного планування та вибору відповідних технологій. Основна мета, як було визначено раніше, полягає у створенні інструменту, що підвищує особисту продуктивність шляхом централізації управління інформацією та завданнями через зручний інтерфейс Telegram.

В основі архітектурного рішення лежить концепція модульної системи, керованої центральним агентом-менеджером, який взаємодіє з набором спеціалізованих агентів-виконавців. Кожен агент-виконавець відповідає за взаємодію з конкретним зовнішнім сервісом: Gmail, Google Calendar, Notion – тоді як агент-менеджер аналізує запити користувача, визначає необхідного виконавця або самостійно надає відповідь на загальні питання чи запити, що стосуються функціональності бота.

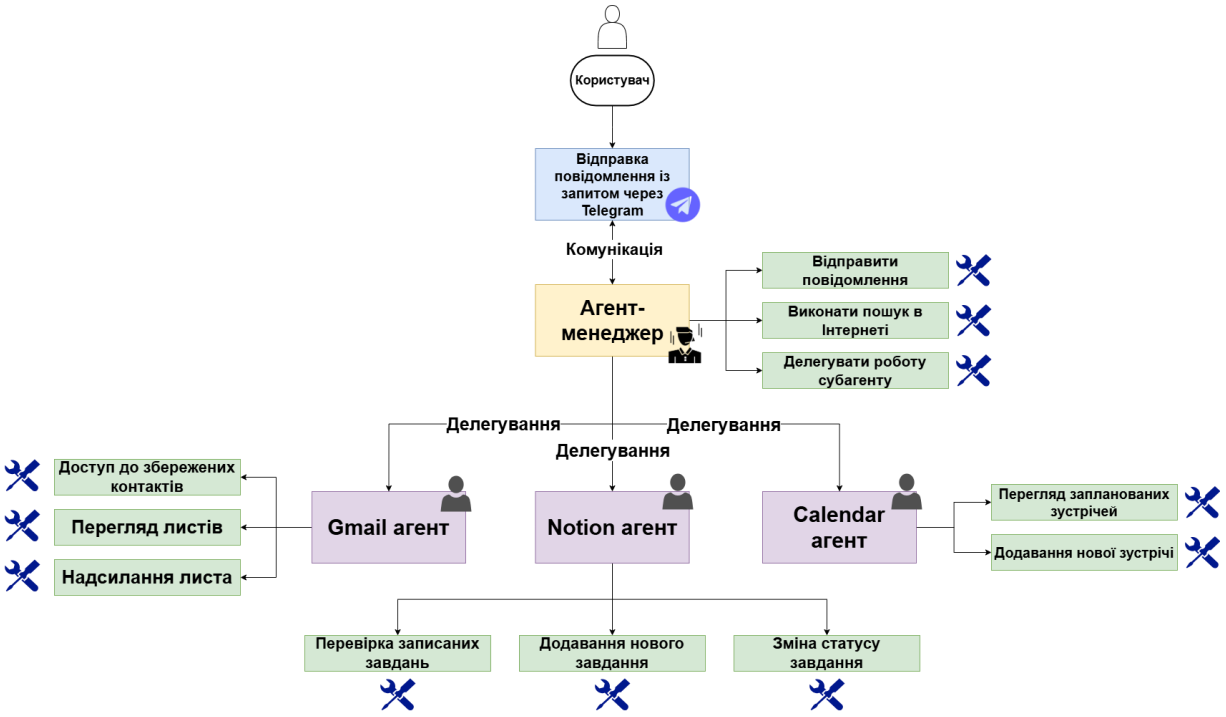
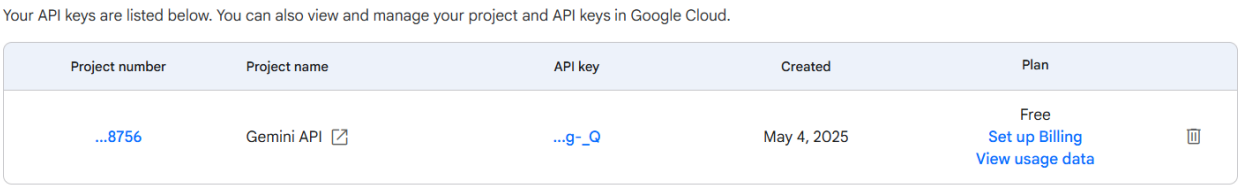


Рисунок 2.1 – Концептуальна схема процесу роботи помічника

Для реалізації зазначеної системи було обрано мову програмування Python, що займає провідні місця у сфері машинного навчання та штучного інтелекту завдяки своїй простоті, великій кількості спеціалізованих бібліотек і фреймворків:

– LangChain – для інтеграції з великою мовною моделлю, створення агентів, налаштування ланцюгів обробки запитів;
– LangGraph – для побудови складних сценаріїв виконання з підтримкою циклів, умовних переходів та агентного управління, що забезпечує гнучке оркестрування логіки.

В якості базової великої мовної моделі було обрано Gemini 2.0 Flash, яка надається компанією Google через програмний інтерфейс. Модель забезпечує достатню якість генерації відповідей при високій швидкодії, що робить її придатною для обробки інтерактивних запитів в режимі реального часу. Взаємодія з моделлю реалізована за допомогою HTTP-запитів до API, а якість відповідей значною мірою залежить від продуманої інженерії інструкцій.


Your API keys are listed below. You can also view and manage your project and API keys in Google Cloud.

Project number	Project name	API key	Created	Plan
...8756	Gemini API	...g-_Q	May 4, 2025	Free Set up Billing View usage data

Рисунок 2.2 – Токен для використання великої мовної моделі

Особливу увагу слід звернути на вибір зовнішніх сервісів, інтеграція з якими реалізована у проєкті. Gmail, Google Calendar і Notion були обрані через їхню популярність, широке поширення серед користувачів та наявність повнофункціональних офіційних API, що були детально описані та розглянуті в розділі 1. Gmail дозволяє обробляти поштову кореспонденцію, що є критично важливим для інформаційної продуктивності. Google Calendar забезпечує керування подіями та нагадуваннями, а Notion – є універсальним сховищем нотаток та списків справ, що дає змогу централізувати особисту організаційну інформацію.

Передбачається, що розроблювана система буде мати певні обмеження. По-перше, її функціональність буде залежати від доступності та стабільності API зовнішніх сервісів. По-друге, ефективність розуміння запитів значною мірою буде залежати від якості заданих інструкцій та можливостей обраної мовної моделі. Також слід зазначити, що на даному етапі розробки не ставилося завдання реалізації багатокористувацького режиму зі складною ізоляцією даних, фокус був на персональному використанні.

Загальну задачу розробки бота було декомпозовано на такі основні підзадачі:

- налаштування середовища та реалізація базової взаємодії з Telegram Bot API;
- розробка інструментів та спеціалізованих агентів-виконавців для взаємодії з Gmail, Google Calendar та Notion, включаючи налаштування API доступів;
- розробка центрального агента-менеджера, відповідального за оркестрацію запитів, делегування завдань та надання відповідей на загальні питання;
- інтеграція всіх компонентів у єдину систему та її тестування.

Загалом, спроєктована архітектура з урахуванням усіх зазначених факторів має дозволити створити гнучку, модульну систему, придатну до масштабування та подальшого розширення як за рахунок підключення нових агентів, так і шляхом оптимізації взаємодії з існуючими. За допомогою такого підходу можна поетапно розробляти та тестувати окремі модулі системи, забезпечуючи кращу керованість процесом розробки.

2.2 Налаштування середовища та взаємодія з Telegram Bot API

Одним із ключових етапів реалізації персонального помічника стало налаштування інтеграції з Telegram, який був обраний як основна платформа взаємодії користувача із системою. Першим кроком у реалізації проєкту було

налаштування робочого середовища та встановлення механізму обміну повідомленнями з платформою.

Процес створення Telegram-бота розпочинається з використання офіційного бота Telegram під назвою BotFather. За допомогою команд користувача було створено нового бота, якому було надано ім’я та тег. У відповідь було надіслано токен доступу – унікальний рядок, який дозволяє програмно здійснювати авторизовану взаємодію з Telegram Bot API. Отриманий токен є конфіденційною інформацією, що використовувати для всіх подальших HTTP-запитів до програмного інтерфейсу. У зв’язку з цим цей токен був записаний у файлі середовища з метою безпечного зберігання та збереження можливості його динамічного зчитування під час виконання програми.

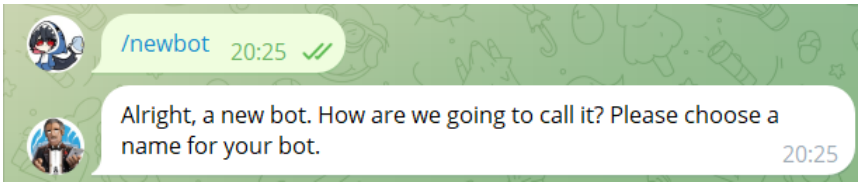


Рисунок 2.3 – Створення бота в Telegram

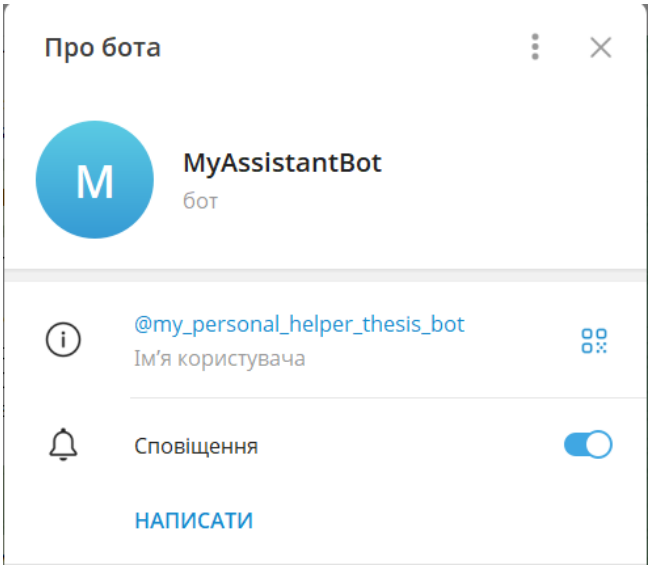


Рисунок 2.4 – Інформація про створеного бота

У даному проєкті було прийнято рішення використовувати бібліотеку *requests* для надсилання HTTP-запитів безпосередньо до Bot API, замість застосування спеціалізованих обгорток. Такий підхід дозволив отримати

глибше розуміння структури запитів та протоколу взаємодії, а також залишив повний контроль над логікою обробки повідомлень. Взаємодія з Telegram Bot API виконується через два основних методи: *getUpdates* та *sendMessage*. Перший використовується для отримання нових повідомлень від користувача, а другий – для надсилання текстових повідомлень у відповідь користувачу. Алгоритм роботи взаємодії з Telegram можна описати наступним чином:

1. програма періодично надсилає запит *getUpdates* до Bot API для отримання повідомлень;
2. при надходженні нового повідомлення витягується його вміст;
3. повідомлення передається далі до модуля обробки запитів, який виконує необхідні дії, та формує відповідь;
4. відповідь надсилається користувачу через запит до методу *sendMessage*.

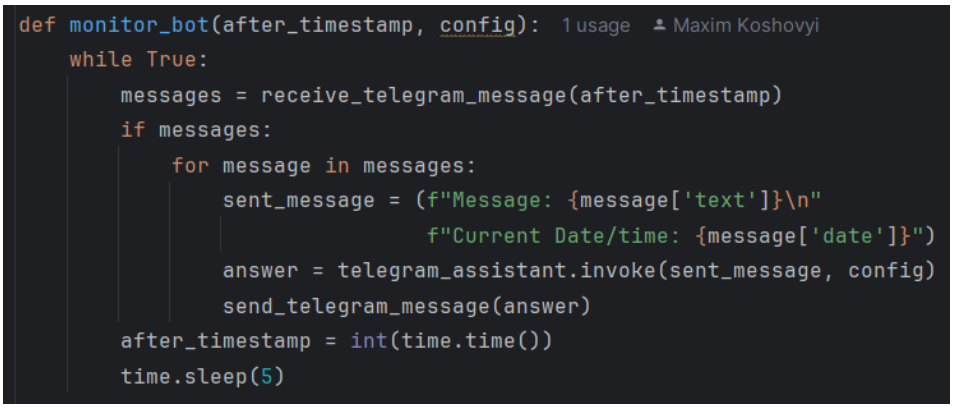


Рисунок 2.5 – Основний алгоритм взаємодії

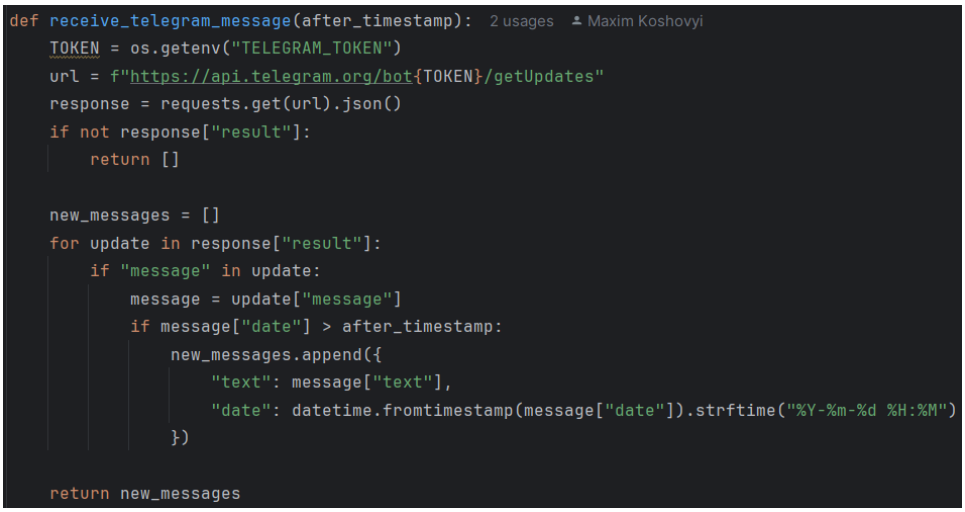


Рисунок 2.6 – Отримання повідомлення від користувача

```python
def send_telegram_message(text):  2 usages  Maxim Koshovyi
    TOKEN = os.getenv("TELEGRAM_TOKEN")
    CHAT_ID = os.getenv("CHAT_ID")

    url = f"https://api.telegram.org/bot{TOKEN}/sendMessage?chat_id={CHAT_ID}&text={text}&amp;parse_mode=MarkdownV2"
    response = requests.get(url).json()
    if not response["ok"]:
        return "Failed to send message"

    return "Message sent successfully on Telegram"
```

Рисунок 2.7 – Відправка повідомлення ботом

Незважаючи на труднощі що виникали через ручне формування запитів та обробку відповіді від API, обраний підхід дозволив глибше зануритися в принципи роботи Telegram Bot API та забезпечив необхідну гнучкість під час розробки й налагодження системи. Також це було корисним досвідом для подальшої інтеграції з більш складними логіками агентів. Повні коди розроблених елементів наведено на лістингах А.1-А.2 (додаток А).

2.3 Розробка модулів інтеграції із зовнішніми сервісами

2.3.1 Налаштування API доступів

Після налагодження базової взаємодії з Telegram, наступним ключовим етапом стала розробка механізмів для взаємодії бота із зовнішніми сервісами: Gmail, Google Calendar та Notion. Даний етап реалізовано через створення спеціалізованих агентів, кожен з яких оснащений набором інструментів, що інкапсулюють логіку запитів до API відповідного сервісу. Як вже було описано в розділі 1, в екосистемі LangChain інструмент – це інтерфейс, який дозволяє LLM-агенту взаємодіяти із зовнішнім світом. Для того, щоб мовна модель могла ефективно використовувати інструмент, він повинен мати чіткий опис, який пояснює, що саме робить інструмент та які вхідні дані він очікує. Цей опис використовується для прийняття рішення про те, який інструмент, або їх послідовність, має бути викликана для виконання запиту користувача. Ці інструменти надаються агенту під час його ініціалізації, щоб він міг делегувати частину своїх функцій зовнішньому коду.

Але перед тим як перейти до розробки агентів та їх інструментів, спочатку необхідно було виконати налаштування API доступів до сервісів, що потрібні для роботи. Для сервісів Google довелося створити проєкт в Google Cloud Console, увімкнути API для бажаних функцій та заповнити облікові дані для доступу бота до цих сервісів. Після чого було завантажено JSON-файл який містить необхідні дані для взаємодії.

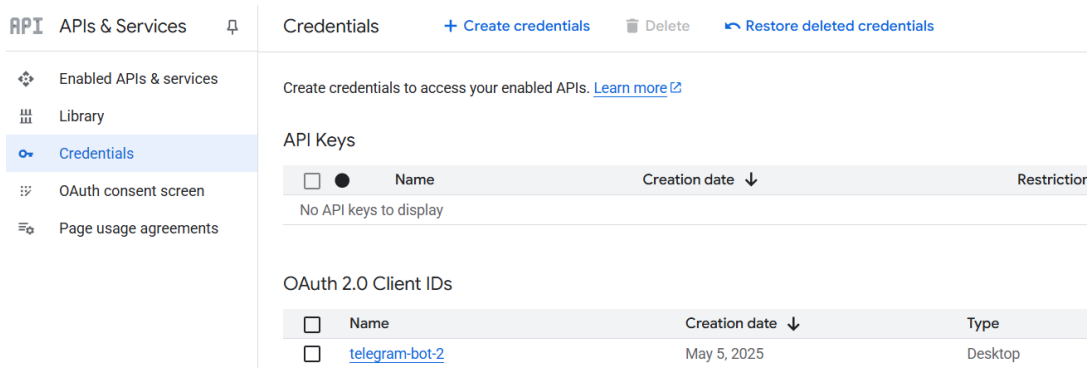


Рисунок 2.8 – Налаштування доступів до сервісів Google

У випадку з Notion потрібно було створити інтеграцію з необхідними правами доступу. Після чого було згенеровано токен, що застосовується для виконання запитів до Notion API. Але щоб створена інтеграція мала доступ до певної сторінки, її необхідно власноруч додати туди.

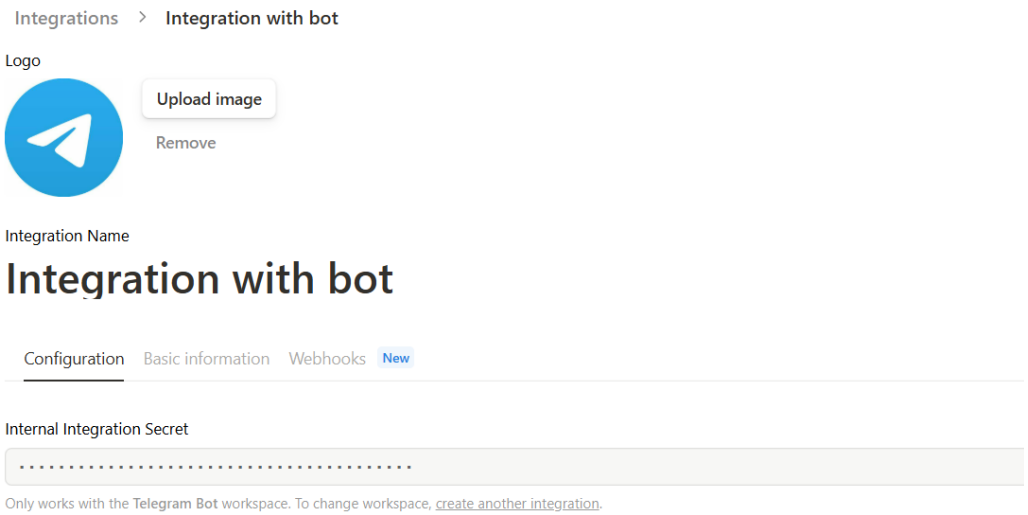


Рисунок 2.9 – Створення інтеграції Notion для взаємодії з додатком

Таким чином, після виконання необхідних налаштувань, можемо переходити до безпосередньої розробки самих агентів та їх інструментів.

2.3.2 Розробка агента для взаємодії з Gmail

Агент Gmail призначений для виконання операцій з електронною поштою користувача. Його функціональність реалізуються через інструменти:

– перегляду пошти за певний період;

– надсилання листа;

– отримання адреси відправки з контактів користувача.

З допомогою першого інструменту агент може отримати список вхідних листів за вказаний часовий інтервал. Використовується Gmail API щоб сформувати запит для отримання списку повідомлень користувача. Виконується пошук в заданому часовому проміжку, та витягується інформація про тему листа, його короткий зміст, адресу відправника та дату відправки. Щоб усе це реалізувати застосовується бібліотека *google-api-python-client*. За допомогою звичайних функцій Python парситься рядок із початковою та кінцевою датою пошуку і вони перетворюються у необхідний формат. Через запит *users.messages.get*, з відповідними фільтрами, отримуємо список листів та витягаємо необхідний вміст.

```
def fetch_emails(self, from_date, to_date, email=""):  1 usage  Maxim Koshovyi
    """
    Fetches emails from Gmail inbox
    """
    try:
        creds = self.get_credentials()
        service = build( serviceName: 'gmail',  version: 'v1', credentials=creds)

        from_date = datetime.fromisoformat(from_date.replace('/', '-')).replace(tzinfo=timezone.utc)
        to_date = datetime.fromisoformat(to_date.replace('/', '-')).replace(tzinfo=timezone.utc)

        query = f'after:{from_date.strftime("%Y/%m/%d")} before:{to_date.strftime("%Y/%m/%d")}'
        if email:
            query += f' from:{email}'

        results = service.users().messages().list(userId='me', q=query).execute()
        messages = results.get('messages', [])

        if not messages:
            return "No emails found in the specified time range."

        email_list = []
        for message in messages:
            msg = service.users().messages().get(userId='me', id=message['id']).execute()

            subject = next((header['value'] for header in msg['payload']['headers'] if header['name']
            from_email = next((header['value'] for header in msg['payload']['headers'] if header['nam
            date = next((header['value'] for header in msg['payload']['headers'] if header['name'] ==
```

Рисунок 2.10 – Інструмент перегляду пошти

За умови що користувач надасть текст та тему листа, агент зможе використати інструмент для відправки листів для виконання запиту користувача. При наявності в запиті адреси відправки, інструмент формує необхідний об’єкт листа і заповнює його. Після цього виконується відправка за допомогою Gmail API, який повертає підтвердження про успішне надсилання або повертає повідомлення про помилку. Реалізація даного інструменту також полягає у використанні бібліотеки *google-api-python-client* та звичайних вбудованих функцій Python. Щоб утворити лист, який буде відправлено через Gmail, підключаємо модуль *email* та створюємо об’єкт *MIMEMultipart*. Після заповнення необхідної інформації – відправляємо лист через *smtplib*.

```python
class SendEmail(BaseTool):  4 usages  Maxim Koshovyi
    name: str = "SendEmail"
    description: str = "Use this to send emails to my contacts on my behalf"
    args_schema: Type[BaseModel] = SendEmailInput

    def send_email_with_gmail(self, email_recipient, email_subject, email_body):  1 usage  Maxim Koshovyi
        """
        Sends an email using Gmail SMTP
        """
        try:
            sender_email = os.getenv("GMAIL_MAIL")
            app_password = os.getenv("GMAIL_APP_PASSWORD")

            msg = MIMEMultipart()
            msg['From'] = sender_email
            msg['To'] = email_recipient
            msg['Subject'] = email_subject
            msg.attach(MIMEText(email_body, _subtype: 'plain'))
            print(msg)

            server = smtplib.SMTP_SSL( host: 'smtp.gmail.com',  port: 465)
            server.login(sender_email, app_password)
            text = msg.as_string()
            server.sendmail(sender_email, email_recipient, text)
            server.quit()
            return "Email sent successfully."
        except Exception as e:
            return f"Email was not sent successfully, error: {e}"
```

Рисунок 2.11 – Інструмент відправки листа

Останній інструмент агент може використати для визначення адреси отримувача за ім’ям збереженого контакту. Для його реалізації

використовується People API, що дозволяє переглядати збережені контакти і витягати необхідну інформацію, таку як ім’я, телефонний номер, або адресу електронної скриньки. Використовується та сама бібліотека що і для попередніх інструментів даного агента. Вхідне ім’я застосовується для пошуку необхідного контакту з метою витягання потрібної інформації. За цих обставин під час пошуку всі записані контакти приводимо до одного формату. По суті даний інструмент є суто допоміжним, який агент має використовувати у парі з попереднім інструментом для відправки листа.

```
def fetch_contact(self, contact_name):  1 usage  Maxim Koshovyi
    """
    Fetches contact information from Google Contacts
    """
    try:
        creds = self.get_credentials()
        service = build( serviceName: 'people',  version: 'v1', credentials=creds)

        # Search for the contact
        results = service.people().searchContacts(
            query=contact_name,
            readMask='names,phoneNumbers,emailAddresses'
        ).execute()

        connections = results.get('results', [])

        if not connections:
            return f"No contact found with the name: {contact_name}"

        matching_contacts = []

        for connection in connections:
            contact = connection['person']
            names = contact.get('names', [])
            print(names)
            if names:
```

Рисунок 2.12 – Інструмент пошуку збереженого контакту

Самого агента створюємо за допомогою методу *create_react_agent*, що реалізує ReAct підхід, за допомогою якого LLM чергує кроки міркування та дій. Як було зазначено під час вибору мовної моделі в 2.1, для цього та для всіх інших агентів було використано мовну модель від Google – Gemini 2.0 Flash. Також слід зазначити важливість надання інструкцій агенту, щодо його ролі, доступних інструментів та очікуваного формату відповіді.

```python
# Initialize LLM model
model = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0.1)

# Initialize the tools
tools = [SendEmail(), FindContactEmail(), ReadEmails()]

# Create the ReAct agent with the specified LLM, tools, system prompt
email_agent = create_react_agent(model, tools, state_modifier=EMAIL_AGENT_PROMPT)


@traceable(run_type="llm", name="Email Agent")  4 usages  Maxim Koshovyi
def invoke_email_agent(task: str) -> str:
    inputs = {"messages": [("user", task)]}
    output = email_agent.invoke(inputs)
    print_agent_output(output)
    return output["messages"][-1].content
```

Рисунок 2.13 – Створення агента з визначеними інструментами

```
EMAIL_AGENT_PROMPT = """
**# Role**

You are my expert email manager agent, responsible for managing my full email inbox, you
my behalf. You are a subagent of my personal assistant agent.

**# Task**

- You will be triggered when the assistant manager agent delegate a task to you, and you
my behalf such as reading emails, sending emails, and writing emails from my inbox.
- Depending on the task given to you by your manager, you will identify the right tools
- After accomplishing the task you will always report back to the manager agent.
- Some examples of tasks that could be delegated to you: "please send en email to Emily
Or: "check if I have received any important emails today".

**# SOP**

Depending on the task that is delegated to you, you must analyze its content and think s
what order, and to ensure that you achieve the desire outcome.

**# Tools **

You have the following tools to assist you in managing my email inbox:

* **FindContactEmail:** Use this tool to get the email of one of my contact when you onl
```

Рисунок 2.14 – Інструкція для агента

На рисунку 2.14 можна побачити частину інструкції що надається даному агенту. В ній зазначається його призначення, завдання на які на нього покладаються та доступні інструменти для їх вирішення.

Детально ознайомитись з повним кодом даного агента можна за допомогою лістингу А.3 (додаток А).

2.3.3 Розробка агента для взаємодії з Google Calendar

Агент Calendar працює з подіями календаря та виконує роботу за допомогою інструментів:

– перегляду запланованих подій;

– створення нової події.

Агент може переглянути всі заплановані події користувача в певному проміжку. Для цього використовується Google Calendar API. При наявності подій в проміжку, можемо дізнатись їх короткий опис, але також враховується можливість відсутності подій у заданому інтервалі. Цей агент також відповідальний за взаємодію із середовищем Google, тому знову для реалізації інструментів використовується бібліотека *google-api-python-client*. Можна сказати що інструмент працює схожим чином до інструменту перегляду електронної пошти. Ми так само парсимо інформацію про початкову та кінцеву дату пошуку для формування запиту через *events.list* із часовими фільтрами. Але ціллю нашого пошуку тепер є не листи користувача, а його заплановані події в календарі, які зберігають інформацію про назву події, її опис та час проведення.

```
def get_calendar_events(self, start_time, end_time):  1 usage  Maxim Koshovyi
    """
    Fetches calendar events from Google Calendar
    """
    try:
        creds = self.get_credentials()
        service = build( serviceName: "calendar",  version: "v3", credentials=creds)

        # Convert string times to datetime objects and ensure they're in UTC
        start_datetime = datetime.fromisoformat(start_time).replace(tzinfo=timezone.utc)
        end_datetime = datetime.fromisoformat(end_time).replace(tzinfo=timezone.utc)

        # Format date-times
        start_format = start_datetime.isoformat().replace( _old: '+00:00',  _new: 'Z')
        end_format = end_datetime.isoformat().replace( _old: '+00:00',  _new: 'Z')

        events = service.events().list(
            calendarId='primary',
            timeMin=start_format,
            timeMax=end_format,
            singleEvents=True,
            orderBy='startTime'
        ).execute()
```

Рисунок 2.15 –Інструмент перегляду запланованих подій

Інший інструмент використовується агентом для створення нової події в календарі. Для цього агенту необхідно отримати інформацію про назву події, її короткий опис, та дату проведення. В реалізації використовується вже згадана до цього бібліотека. Щоб створити та заповнити інформацію про заплановану подію застосовується словник, який в Python позначається через фігурні дужки, що можна побачити на рисунку 2.16. Форматування запланованого проміжку проведення зустрічі доволі складана задача, через необхідність врахування годин, хвилин та секунд які потрібно виділити із повідомлення користувача. Тому час завершення зустрічі статично задається в одну годину після її початку. Для додавання події в календар використовується запит *events.insert*.

```
def create_event(self, title, desc, start):  1 usage  Maxim Koshovyi
    """
    Creates an event on Google Calendar
    """
    try:
        creds = self.get_credentials()
        service = build( serviceName: "calendar",  version: "v3", credentials=creds)

        # Convert the string to a datetime object
        event_datetime = datetime.fromisoformat(start)

        event = {
            'summary': title,
            'description': desc,
            'start': {
                'dateTime': event_datetime.isoformat(),
                'timeZone': 'UTC',
            },
            'end': {
                'dateTime': (event_datetime + timedelta(hours=1)).isoformat(),
                'timeZone': 'UTC',
            },
        }
```

Рисунок 2.16 – Інструмент створення події

Даний агент створюється схожим чином до попереднього, але має свій набір інструментів для використання, та свою інструкцію за якою він має працювати. В ній ми обов’язково зазначаємо що агент повинен допомагати

користувачу із керуванням його подій в календарі. Наводимо приклади можливих запитів що стосуються роботи із календарем і прописуємо які інструменти він має під своїм контролем та для чого кожен з них може бути використаний. Через лістинг А.4 можна ознайомитись з повним кодом даного агента.

```python
# Initialize LLM model
model = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0.1)

# Initialize the tools
tools = [CreateEvent(), GetCalendarEvents()]

# Create the ReAct agent with the specified LLM, tools, system prompt
calendar_agent = create_react_agent(model, tools, state_modifier=CALENDAR_AGENT_PROMPT)

@traceable(run_type="llm", name="Calendar Agent")  4 usages  Maxim Koshovyi
def invoke_calendar_agent(task: str) -> str:
    inputs = {"messages": [("user", task)]}
    output = calendar_agent.invoke(inputs)
    print_agent_output(output)
    return output["messages"][-1].content
```

Рисунок 2.17 – Створення агента з визначеними інструментами

```
CALENDAR_AGENT_PROMPT = """
**# Role**

You are my expert calender manager agent, responsible for managing my full calender,
on my behalf. You are a subagent of my personal assistant agent.

**# Task**

- You will be triggered when the assistant manager agent delegate a task to you, and
my behalf such as checking my availability, creating events, and getting my calendar
- Dependign on the task given to you by your manager, you will identify the right to
- After accomplishing the task you will always report back to the manager agent.
- Some examples of tasks that could be delegated to you: "check if I have any meetin

**# SOP**

Depending on the task that is delegated to you, you must analyze its content and thi
what order, and to ensure that you achieve the desire outcome.

**# Tools **

You have the following tools to assist you in managing my calendar events:

* **FindContactEmail:** Use this tool to get the email of one of my contact when you
```

Рисунок 2.18 – Інструкція для агента

2.3.4 Розробка агента для взаємодії з Notion

Останній агент націлений на взаємодію з Notion. Він призначений для роботи з нотатками та списком справ користувача, що зберігаються в БД. Для цього розроблені інструменти:

– перегляду завдань за конкретну дату;

– створення нових завдань;

– зміни статусу завдання.

Перший інструмент використовується агентом для виконання запиту до Notion API з метою пошуку записаних на конкретну дату завдань. Витягається необхідна інформацію про саме завдання та його статус. Враховується варіант відсутності завдань за вказаною датою. Для реалізації даного та подальших інструментів, що взаємодіють з нотатками, використовується бібліотека *notion-client*. За допомогою словника Python формується фільтр для запиту до бази даних у Notion в якій зберігається список справ користувача. Щоб отримати список на вказану дату виконується запит *databases.query*. Після цього, з отриманого списку витягається конкретна інформація про самі завдання.

```
class GetMyTodoListInput(BaseModel):  1 usage  Maxim Koshovyi
    date: str = Field(description="Date for which to retrieve tasks (YYYY-MM-DD)")

class GetMyTodoList(BaseTool):  4 usages  Maxim Koshovyi
    name: str = "GetMyTodoList"
    description: str = "Use this to retrieve tasks from your Notion to-do list for a
    args_schema: Type[BaseModel] = GetMyTodoListInput

    def get_tasks_for_date(self, target_date):  1 usage  Maxim Koshovyi
        try:
            # Parse the target date string into a datetime object
            try:
                target_datetime = datetime.strptime(target_date, format: "%Y-%m-%d")
            except ValueError:
                print(f"Error: Invalid date format. Please use YYYY-MM-DD format.")
                return []

            # Set up the filter to get tasks due on the target date
            filter_params = {
                "filter": {
                    "property": "Date",
                    "date": {
                        "equals": target_date
                    }
                }
            }
```

Рисунок 2.19 – Інструмент перегляду завдань

Для створення нового завдання, агент має дізнатись назву завдання та дату на яку воно планується. Маючи необхідну інформацію, агент за допомогою даного інструменту формує запит до Notion API. В результаті сервіс повідомляє про успішне створення або про помилку. Саме створення завдання виконується через словник. Він заповнюється необхідною інформацією що була отримана від користувача. Водночас при створенні завжди задаємо статичний статус нового завдання – *Not started*. Новий запис до БД додається через запит *pages.create*.

```python
class AddTaskInTodoListInput(BaseModel):  1 usage  Maxim Koshovyi
    task: str = Field(description="Task to be added")
    date: str = Field(description="Date and time for the task (YYYY-MM-DD) (HH:MM)")

class AddTaskInTodoList(BaseTool):  4 usages  Maxim Koshovyi
    name: str = "AddTaskInTodoList"
    description: str = "Use this to add a new task to your Notion to-do list"
    args_schema: Type[BaseModel] = AddTaskInTodoListInput

    def add_task(self, task, due_date=None):  1 usage  Maxim Koshovyi
        try:
            new_task = {
                "Title": {"title": [{"text": {"content": task}}]},
                "Status": {"status": {"name": TaskStatus.NOT_STARTED.value}},
            }
            if due_date:
                new_task["Date"] = {"date": {"start": due_date}}

            notion.pages.create(parent={"database_id": database_id}, properties=new_task)

            return f"Task '{task}' added successfully to Todo list for {due_date}."
        except Exception as e:
            return f"An error occurred: {str(e)}"
```

Рисунок 2.20 – Інструмент створення завдання

Останній інструмент застосовується агентом при потребі в зміні статусу вже наявного завдання. Для формування необхідного запиту до сервісу, потрібно визначити саме завдання та статус який для нього потрібно встановити. Завдання, статус якого необхідно змінити, визначається за назвою та датою. За аналогією до попередньо описаних реалізацій, використовуємо словник Python щоб задати фільтри для пошуку завдання. Після виконання запиту *databases.query* з необхідними параметрами при наявності результатів

витягаємо перший з них. Для витягнутого завдання необхідно змінити статус. Для цього в запиті *pages.update* вказуємо *id* отриманого завдання та в параметрі *properties*, що представлено словником, задаємо для ключа *Status* бажаний статус завдання.

```
class SetMyTaskStatus(BaseTool):  4 usages  Maxim Koshovyi
    name: str = "SetMyTaskStatus"
    description: str = "Use this to change specific task status from your Notion to-
    args_schema: Type[BaseModel] = SetMyTaskStatusInput

    def set_task_status_for_date(self, target_title, target_date, new_status):  1 usag
        try:
            # Parse the target date string into a datetime object
            try:
                target_datetime = datetime.strptime(target_date, format: "%Y-%m-%d")
            except ValueError:
                print(f"Error: Invalid date format. Please use YYYY-MM-DD format.")
                return []

            # Set up the filter to get tasks due on the target date
            filter_params = {
                "filter": {
                    "and": [
                        {"property": "Title", "title": {"equals": target_title}},
                        {"property": "Date", "date": {"equals": target_date}},
                    ]
                }
            }
```

Рисунок 2.21 – Інструмент зміни статусу завдання

Як і у випадках до цього, даний агент має схожу логіку роботи, але власний набір інструментів. Детально ознайомитись з повним кодом даного агента можна за допомогою лістингу А.5.

```
# Initialize LLM model
model = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0.1)

# Initialize the tools
tools = [GetMyTodoList(), AddTaskInTodoList(), SetMyTaskStatus()]

# Create the ReAct agent with the specified LLM, tools, system prompt
notion_agent = create_react_agent(model, tools, state_modifier=NOTION_AGENT_PROMPT)


@traceable(run_type="llm", name="Notion Agent")  4 usages  Maxim Koshovyi
def invoke_notion_agent(task: str) -> str:
    inputs = {"messages": [("user", task)]}
    output = notion_agent.invoke(inputs)
    print_agent_output(output)
    return output["messages"][-1].content
```

Рисунок 2.22 – Створення агента з визначеними інструментами

В інструкції до описаного агента обов’язково зазначаємо його роль як помічника при роботі з нотатками в Notion. Надаємо йому перелік можливих запитів що стосуються таких дій а також доступних інструментів, що можуть бути їм використані для задовільнення потреб користувача.

```
NOTION_AGENT_PROMPT = """
**# Role**

You are my expert notion manager agent, responsible for managing my full notion todo
my behalf. You are a subagent of my personal assistant agent.

**# Task**

- You will be triggered when the assistant manager agent delegate a task to you, and
my behalf such as getting tasks from my todo list, adding new tasks on my behalf.
- Depending on the task given to you by your manager, you will identify the right to
- After accomplishing the task you will always report back to the manager agent.
- Some examples of tasks that could be delegated to you: "check my todo list for tom

**# SOP**

Depending on the task that is delegated to you, you must analyze its content and thi
what order, and to ensure that you achieve the desire outcome.

**# Tools **

You have the following tools to assist you in managing my todo list:

* **GetMyTodoList:** Use this tool to get all tasks from my todo list.
```

Рисунок 2.23 – Інструкція для агента

Використання підходу ReAct для всіх агентів, реалізація яких була описана протягом даного розділу, дозволяє їм не просто сліпо викликати інструменти, а міркувати про те, який інструмент найкраще підходить для поточного фрагмента запиту, які параметри йому передати, та як інтерпретувати результат перед тим, як сформувати проміжну або фінальну відповідь. Це робить їх більш гнучкими та адаптивними. Створені інструкції дозволяють чітко окреслити роль агента і контролювати стиль його взаємодії з інструментами, що критично важливо для забезпечення стабільності та релевантності відповіді.

Таким чином, реалізація агентів з окремими інструментами дозволила ефективно розділити відповідальність між компонентами системи, зберігши логічну гнучкість завдяки ReAct-підходу та підтримуючи природну мовну взаємодію завдяки сучасній LLM.

2.4 Створення агента-менеджера та оркестрація взаємодії

Після створення спеціалізованих агентів для роботи із зовнішніми сервісами, наступним кроком є розробка центрального агента-менеджера. Його основне завдання – приймати запити від користувача, аналізувати їх та вирішувати, чи потрібно делегувати виконання одному зі спеціалізованих агентів, чи можна відповісти самостійно, наприклад, на питання загального характеру або запити щодо функціональності самого бота.

Агент-менеджер виступає як головний координатор у системі. Він є першою точкою контакту для всіх повідомлень користувача. Його архітектура побудована з використанням фреймворку LangGraph. Він дозволяє будувати граф станів, де вузлами виступають функціональні блоки, а ребра – є правилами переходу між ними. В контексті розробленого асистента граф взаємодії включає наступні вузли:

- *telegram_agent* – отримання запиту від користувача та вирішення подальшого напрямку обробки;
- *email_agent* – виклик агента що взаємодіє з Gmail;
- *calendar_agent* – виклик агента що взаємодіє з Google Calendar;
- *notion_agent* – виклик агента що взаємодіє з Notion;

Ребра між вузлами визначаються умовною логікою. Тобто, в залежності від вмісту повідомлення користувача, менеджер керується даними умовами для вибору кращого шляху досягнення необхідної мети. Якщо запит містить слова, пов’язані з поштою, здійснюється перехід до *gmail_agent*, інакше – до іншого відповідного вузла. Важливо, що після виконання запиту ми завжди повертаємось до вузла менеджера. LangGraph дозволяє керувати станом діалогу протягом усього описаного процесу.

```
class ManagerAgent:  2 usages  Maxim Koshovyi *

    def create_workflow(self):  1 usage  Maxim Koshovyi *
        workflow = StateGraph(State)

        workflow.add_node("telegram_agent", self.call_model)
        workflow.add_node("email_agent", self.call_email_agent)
        workflow.add_node("calendar_agent", self.call_calendar_agent)
        workflow.add_node("notion_agent", self.call_notion_agent)

        workflow.add_edge(START, end_key: "telegram_agent")

        workflow.add_conditional_edges(
            source: "telegram_agent",
            self.route_primary_assistant,
            path_map: {
                "email_agent": "email_agent",
                "calendar_agent": "calendar_agent",
                "notion_agent": "notion_agent",
                END: END,
            },
        )

        workflow.add_edge( start_key: "email_agent",  end_key: 'telegram_agent')
        workflow.add_edge( start_key: "calendar_agent",  end_key: 'telegram_agent')
        workflow.add_edge( start_key: "notion_agent",  end_key: 'telegram_agent')
```

Рисунок 2.24 – Код організованої взаємодії вузлів

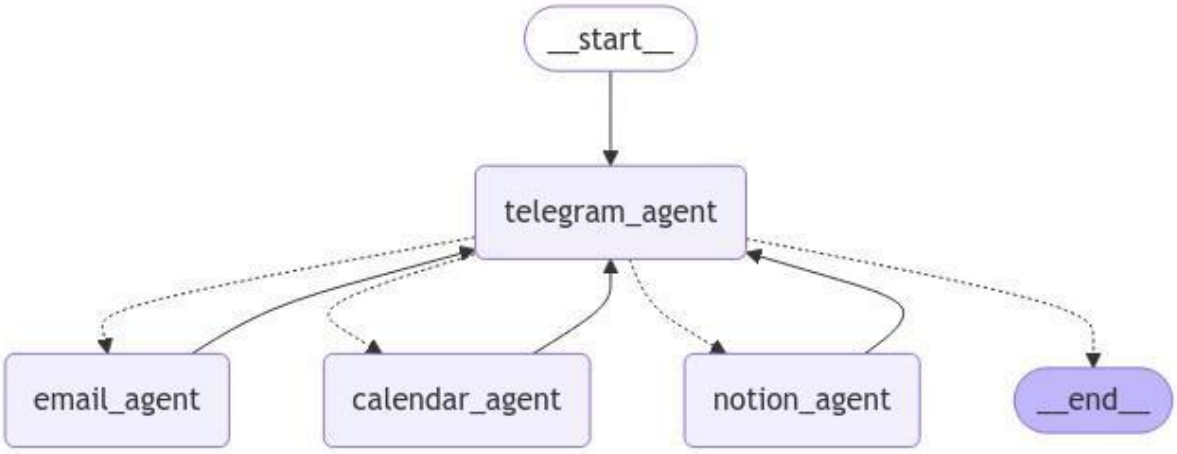


Рисунок 2.25 – Утворений граф взаємодії вузлів

Як вже було зазначено, після обробки запиту агент-менеджер визначає, якому саме спеціалізованому агенту потрібно делегувати запит, передає йому відповідні параметри та чекає на результат. Делегування реалізоване таким чином, що субагенти працюють як окремі агенти з власними інструментами та логікою, а менеджер слугує координаційною ланкою.

```python
class DelegateInput(BaseModel):  1 usage  Maxim Koshovyi
    agent_name: str = Field(description="Name of the subagent to delegate to")
    task: str = Field(description="Task to delegate to the subagent")

class Delegate(BaseTool):  3 usages  Maxim Koshovyi
    name: str = "Delegate"
    description: str = "Use this to delegate a task to one of your subagents"
    args_schema: Type[BaseModel] = DelegateInput

    def delegate(self, agent_name: str, task: str) -> str:  1 usage  Maxim Koshovyi
        """
        Simulates delegating a task to a subagent
        """
        print(f"Task '{task}' has been delegated to the {agent_name} agent.")
        if agent_name == "Email Agent":
            return invoke_email_agent(task)
        elif agent_name == "Calendar Agent":
            return invoke_calendar_agent(task)
        elif agent_name == "Notion Agent":
            return invoke_notion_agent(task)
        else:
            return "Invalid agent name"
```

Рисунок 2.26 – Інструмент делегування

Такий підхід дозволяє легко масштабувати систему. Подальше додавання нового агента не потребує зміни архітектури в цілому, достатньо лише додати новий вузол до графу та відповідне правило переходу. Теж саме стосується інструментів які використовуються агентами. При додаванні нового функціоналу, додаткового налаштування потребує лише агент який буде працювати з цим інструментом. Він повинен мати доступ до цього інструменту та бути проінформованим про його можливості, що досягається шляхом додавання відповідних даних до інструкції.

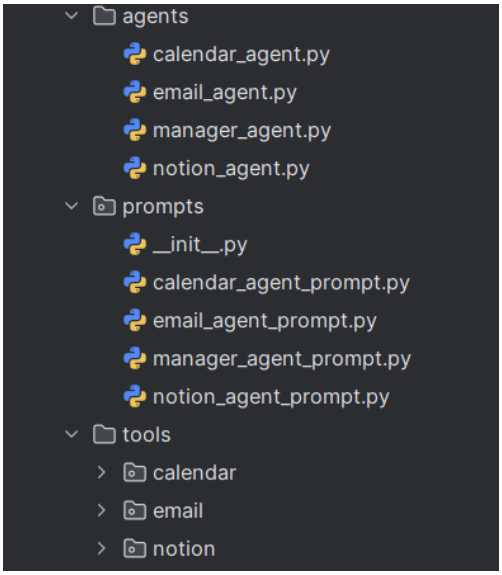


Рисунок 2.27 – Модульність проєкту

Для запитів, що не стосуються жодного із зовнішніх сервісів, реалізовано обробку через мовну модель з підтримкою Tavily Search API. Цей інструмент допомагає асистенту відповідати на запитання про поточні події або маловідомі факти, з якими модель може не бути ознайомлена, використовуючи актуальні зовнішні відомості з Інтернету, згідно з [26]. Для цього, при розпізнаванні в повідомленні запитання загального характеру, використовується метод *invoke* для об'єкта *TavinlySearchResults* що доступний через бібліотеку *langchain-community*. Згаданий метод виконує пошук інформації в Інтернеті за повідомленням користувача. Отриманий результат оброблюється LLM, що задіяна в агенті, після чого формується відповідь.

```
# Get the raw search results
search_results = self.tavily_tool.invoke({'query': last_tool_calls[0]['args']['query']})

# Format the search results
formatted_results = "\n\n".join(
    f"Title: {result['title']}\nURL: {result['url']}\nContent: {result['content']}"
    for result in search_results
)

return {"messages": [ToolMessage(
    content=f"I found these results:\n\n{formatted_results}",
    name="tavily_search_results_json",
    tool_call_id=tool_call_id
)]}
```

Рисунок 2.28 – Використання Tavily Search API

Одним із ключових елементів роботи агента-менеджера є системна інструкція. Вона має чітко описувати роль даного агента як головного координатора, описувати доступні інструменти та напрямки делегування роботи та містити інформацію щодо того, як розпізнавати різні типи запитів і якого спеціалізованого агента викликати для цього завдання. В інструкції обов'язково вказуємо які субагенти доступні та з якими саме завданнями вони можуть допомогти. Також наводяться приклади можливих запитів від користувача, та вказується який саме допоміжний агент зміг би вирішити проблему.

```
TELEGRAM_ASSISTANT_MANAGER_PROMPT = """
**# Role**

Your are my personal assistant, your are in charge of managing tasks related
Additionally, you can answer general knowledge questions by searching the we

**# Tasks**

- You will be triggered when I send you a telegram messages, you must analyz
course of actions to take to complete the tasks given.
- Some examples of messages that you might receive: "Please tell me all the
"Please add finishing client project, as high priority to my to do list in n
- You are communicating with me through Telegram, so ensure your messages ar

**# Tools & Subagents**

To delegate a task to one of your subagents, use the **Delegate* tool. Provi

* **Email Agent:** The email agent can do any tasks related to managing my i

* **Calendar Agent:** The calendar agent can do any tasks related to managin
my calendar events for a specific time frame.
```

Рисунок 2.29 – Системна інструкція для агента-менеджера

Але навіть з добре продуманими інструкціям, велика мовна модель іноді може неправильно зрозуміти намір користувача або вибрати не той інструмент. Це вимагало ітеративного процесу вдосконалення інструкцій, додавання прикладів, уточнення описів інструментів. Іноді доводилося надавати користувачеві можливість уточнити свій запит. Подолання цих труднощів було невід'ємною частиною процесу дослідження та розробки, що дозволило створити більш надійний та функціональний кінцевий прототип. Повний код агента-менеджера представлено на лістингу А.6.

Таким чином, агент-менеджер виступає мозковим центром системи, який координує її роботу, забезпечує розумне делегування функцій та підтримує ефективну взаємодію з користувачем. Завдяки LangGraph вдається реалізувати гнучку, масштабовану й контрольовану архітектуру програми.

3 РЕЗУЛЬТАТИ РОЗРОБКИ ТА ЇХ АНАЛІЗ

3.1 Загальна характеристика отриманих результатів

Основним результатом даної роботи є розробка та програмна реалізація функціонального прототипу персонального Telegram-бот помічника. Цей помічник призначений для підвищення ефективності управління особистою інформацією та завданнями шляхом інтеграції з ключовими сервісами та використання великої мовної моделі для обробки запитів користувача. Тобто, можна зробити висновок, що поставлена на початку роботи мета була досягнута. Для її досягнення були успішно реалізовані всі ключові завдання:

- проведено детальний аналіз предметної області;
- розроблено архітектуру системи на основі модульного підходу;
- налаштовано роботу з Telegram Bot API;
- інтегровано велику мовну модель для обробки запитів;
- реалізовані модулі роботи із зовнішніми сервісами: Gmail, Google Calendar, Notion;
- здійснено оркестрацію роботи за допомогою LangGraph;
- виконано тестування розробленого програмного рішення.

Інтеграція великої мовної моделі забезпечила здатність системи розуміти запити, сформульовані природною мовою, виокремлювати наміри користувача та генерувати адекватні відповіді. Водночас, використання фреймворків LangChain та LangGraph дозволило побудувати гнучку архітектуру, що ефективно координує взаємодію з API Gmail, Google Calendar та Notion, а також із пошуковим сервісом Tavily Search, що в сукупності сприяє досягненню головної мети – підвищенню продуктивності користувача.

Досягненню поставленої мети сприяло послідовне та ретельне виконання ключових завдань. Фундаментальним етапом стала аналітична робота, що включала глибоке дослідження предметної області. Цей аналіз дозволив сформувати чітке уявлення про сучасний стан технологій створення

інтелектуальних персональних асистентів, оцінити можливості та обмеження великих мовних моделей, а також вивчити специфіку програмних інтерфейсів цільових сервісів. На основі отриманих знань було здійснено обґрунтований вибір програмних засобів та технологій, зокрема мови програмування Python, фреймворків LangChain та LangGraph, моделі Gemini 2.0 Flash та відповідних бібліотек для взаємодії з API, що забезпечило сучасний технологічний стек для реалізації проєкту.

В основу архітектурного рішення було покладено концепцію модульної системи, керованої центральним агентом-менеджером, який координує роботу спеціалізованих допоміжних агентів. Такий підхід був обраний з метою забезпечення гнучкості, масштабованості та спрощення подальшої підтримки й розвитку системи. Наступним логічним кроком стала безпосередня програмна реалізація спроєктованих модулів бота. Цей етап охоплював розробку механізмів взаємодії з Telegram Bot API, обробки запитів користувача, а також створення інструментів для взаємодії з Gmail, Google Calendar та Notion, що дозволило боту виконувати конкретні операції з даними користувача в цих сервісах.

Критично важливим етапом стала інтеграція великої мовної моделі в систему. Ця інтеграція наділила бота здатністю до інтелектуального розуміння запитів природною мовою, аналізу намірів користувача, генерації осмислених відповідей та координації складних, багатоетапних дій через підключені інструменти. Для розширення інформаційних можливостей помічника та надання йому здатності відповідати на питання загального характеру, що не стосуються безпосередньо зовнішніх сервісів, було успішно інтегровано пошуковий сервіс Tavily Search. Це перетворило бота на більш універсальний та корисний інструмент.

Завершальними етапами роботи стали ручне тестування розробленого бота та аналіз отриманих результатів. Під час тестування перевірялась функціональність всіх реалізованих модулів, надійність їхньої взаємодії, коректність обробки різноманітних запитів користувача, а також зручність та

ефективність взаємодії з ботом в цілому. Зібрані дані та спостереження лягли в основу оцінки досягнутих результатів, представленої в цьому розділі, та формування загальних висновків дипломної роботи.

Таким чином, послідовне та планомірне виконання всіх поставлених завдань забезпечило не лише досягнення основної мети дослідження, але й створення завершеного програмного рішення, що демонструє високий потенціал для практичного застосування. Розроблений Telegram-бот помічник є наочним підтвердженням того, як сучасні досягнення в галузі штучного інтелекту можуть бути ефективно використані для створення інноваційних інструментів, спрямованих на вирішення актуальних проблем користувачів.

3.2 Демонстрація роботи розробленого помічника

В даному підрозділі наводяться приклади взаємодії з розробленим Telegram-бот помічником, що ілюструють його ключові можливості: розуміння природної мови, виконання завдань через інтеграцію з Gmail, Google Calendar, Notion, а також надання відповідей на загальні питання.

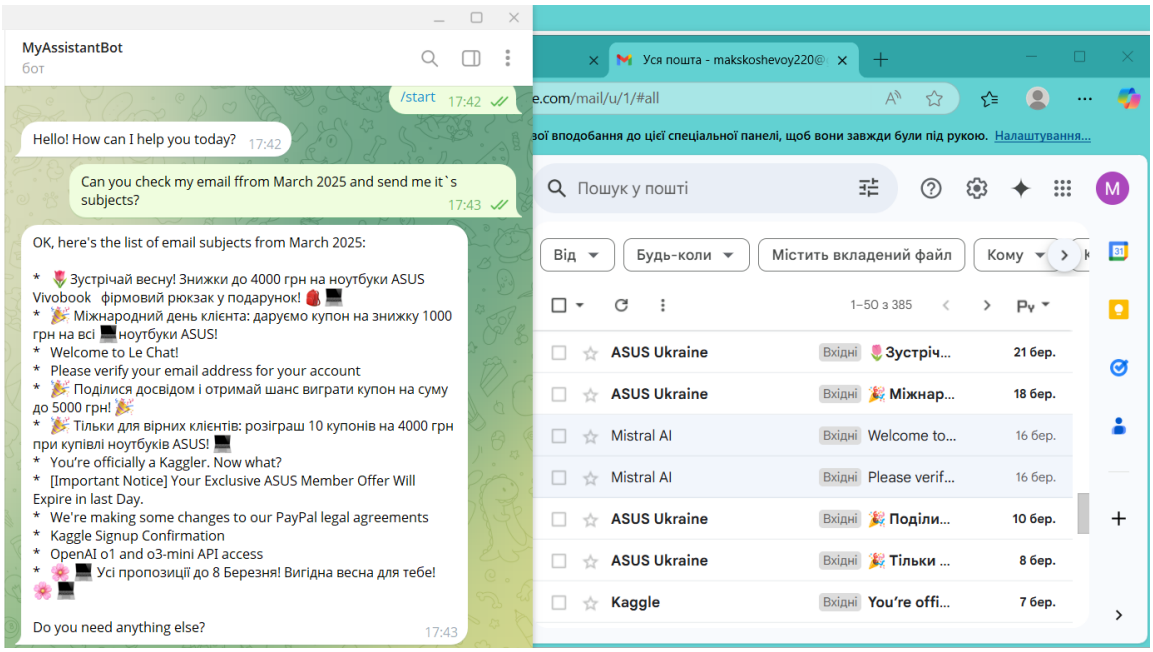


Рисунок 3.1 – Перевірка пошти за певний проміжок

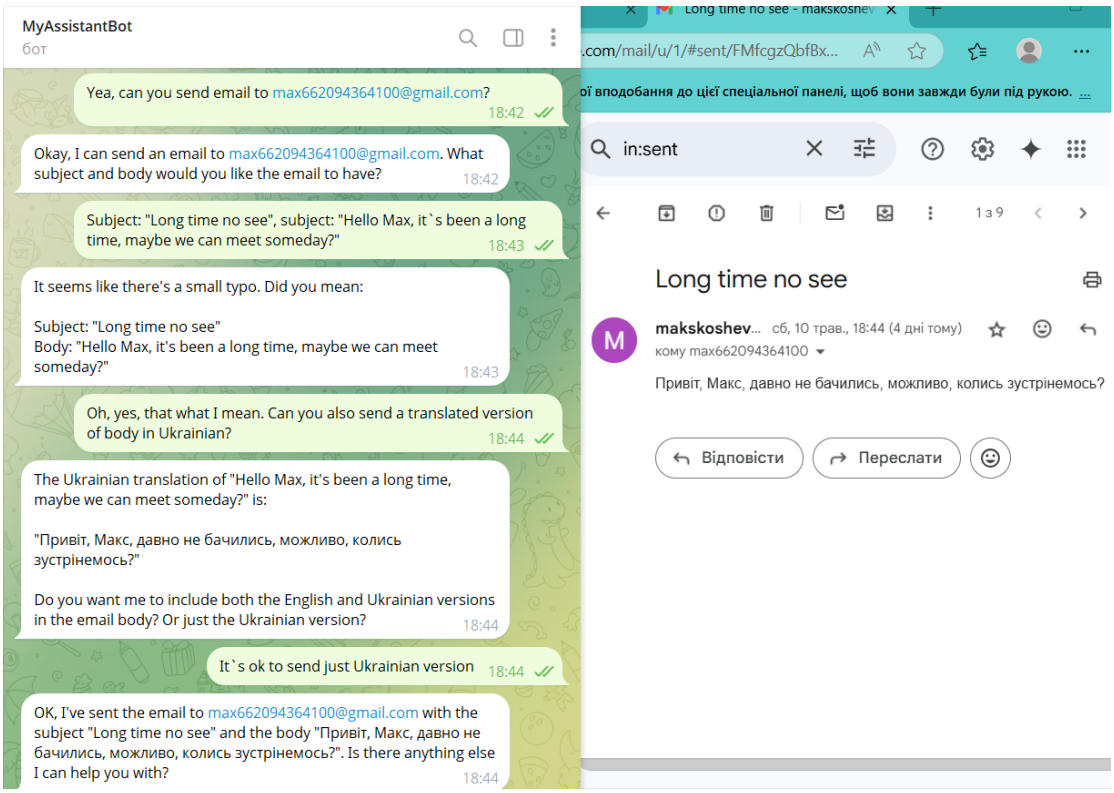


Рисунок 3.2 – Надсилання листа із збереженням контексту розмови

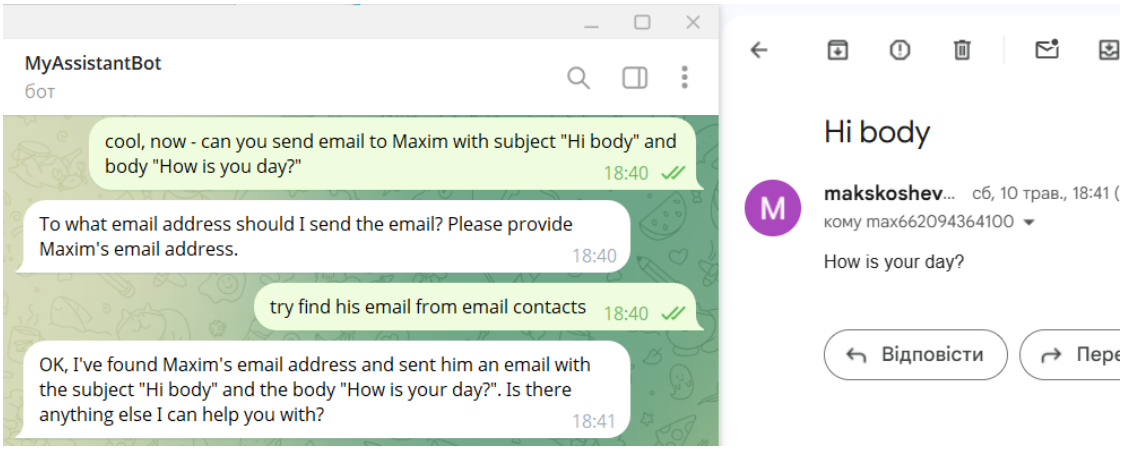


Рисунок 3.3 – Надсилання листа збереженому контакту

Наведені рисунку демонструють можливість бота розпізнавати наміри користувача що пов’язані з роботою із сервісом Gmail. Слід відзначити що усі повідомлення були надіслані природною мовою у вільному форматі, і за рахунок використання LLM це не стало перешкодою для виконання поставленого боту завдання. Також за рахунок цього бот зміг легко зрозуміти прохання користувача перекласти повідомлення на іншу мову, незважаючи на відсутність прямої реалізації такого функціоналу в інструментах даного агента.

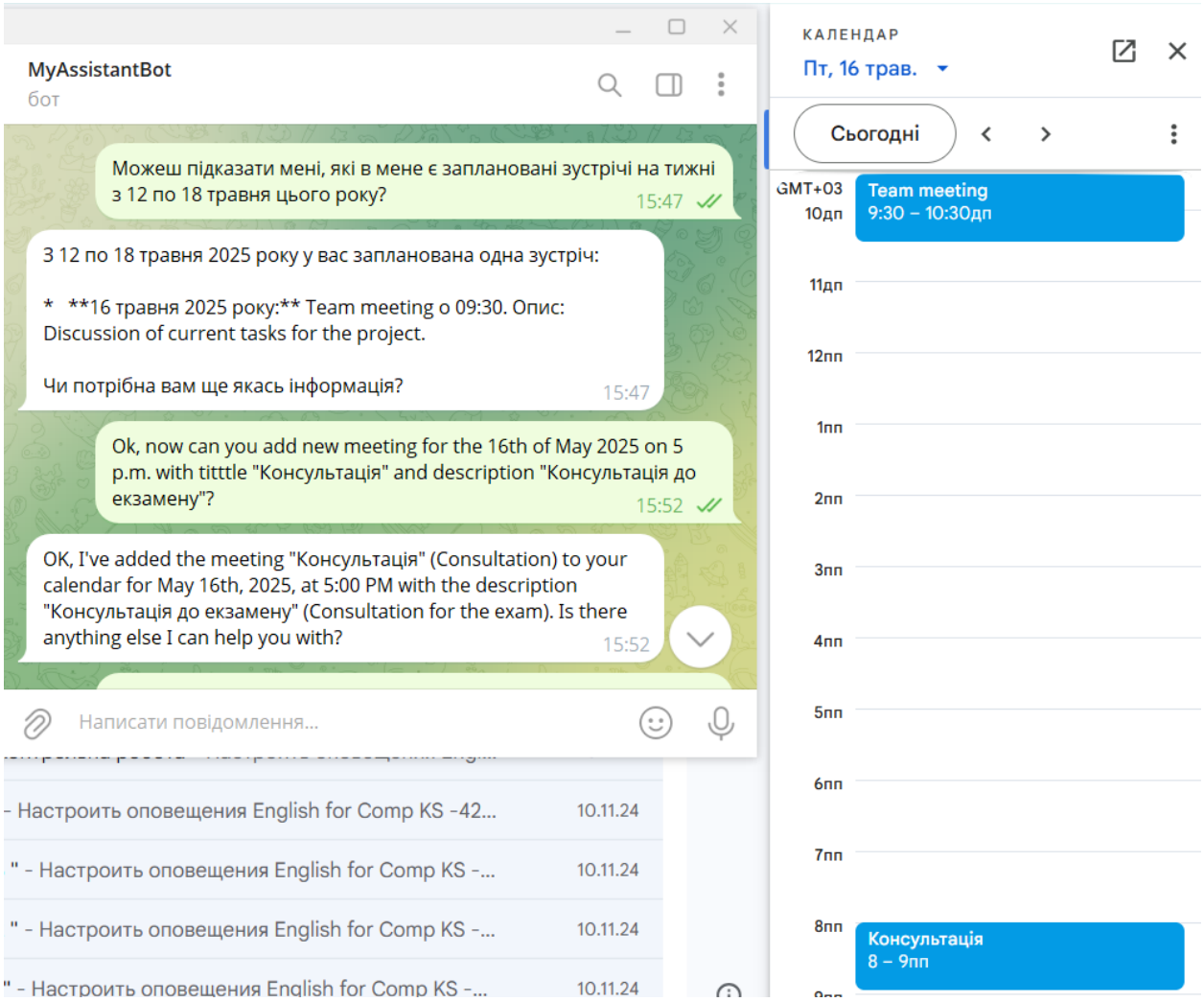


Рисунок 3.4 – Взаємодія із календарем

Наступний приклад демонструє взаємодію бота із календарем користувача. Спочатку користувач дізнався про заплановані події протягом тижня і потім додав нову зустріч на конкретну дату.

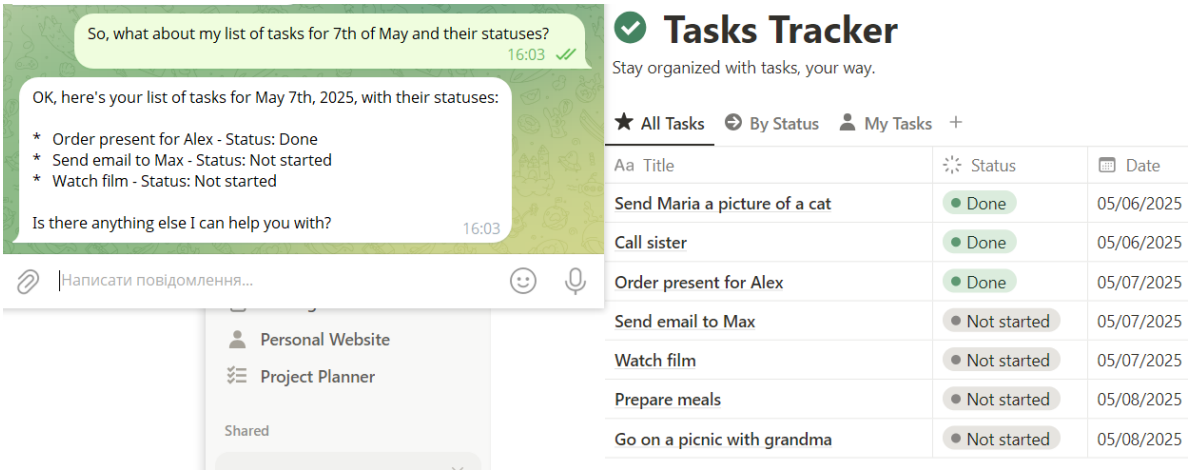


Рисунок 3.5 – Перегляд нотаток зі списком справ

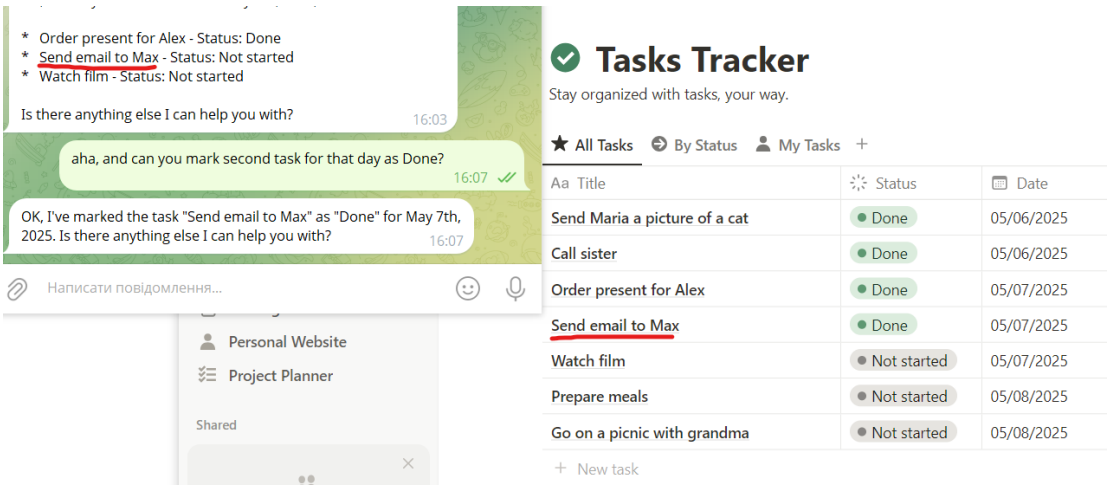


Рисунок 3.6 – Зміна статусу завдання

Усі заплановані можливості взаємодії користувача із нотатками в Notion також відпрацьовують коректно. Бот надає інформацію про список справ на конкретну дату, дозволяє додавати нові завдання та змінювати статус існуючих. Усе керування відбувається через звичайні текстові повідомлення від користувача.

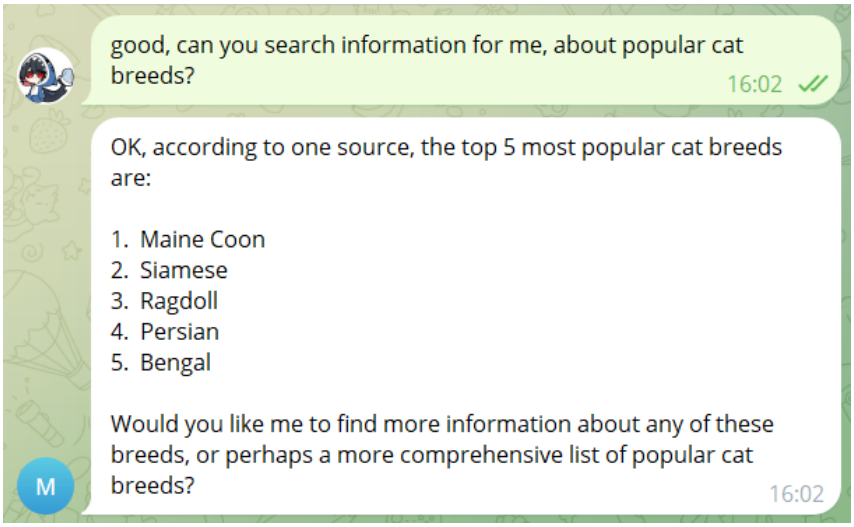


Рисунок 3.7 – Пошук відповіді на загальне питання

Останній приклад демонструє можливості бота знаходити необхідну інформацію використовуючи відомості з Інтернету.

Наведені приклади роботи на рисунках 3.1 – 3.7 показують гнучкість бота у розумінні намірів користувача та його здатність взаємодіяти з різними сервісами. Окрім вже наданих описів, особливої уваги заслуговує демонстрація роботи на рисунках 3.3 та 3.4. В першому випадку можемо

помітити що бот здатен продовжувати діалог із користувачем, зберігаючи контекст розмови, для виконання його запиту. В другому – навіть при надсиланні повідомлення іншою мовою, бот спроможний зрозуміти поставлене йому завдання.

За допомогою консолі можна відслідковувати хід взаємодії бота із користувачем. Також роботу бота можна простежити через систему трасування LangSmith, яка дозволяє переглядати хід виконання запитів, процес переходу між вузлами, вибір агентів та використання інструментів.

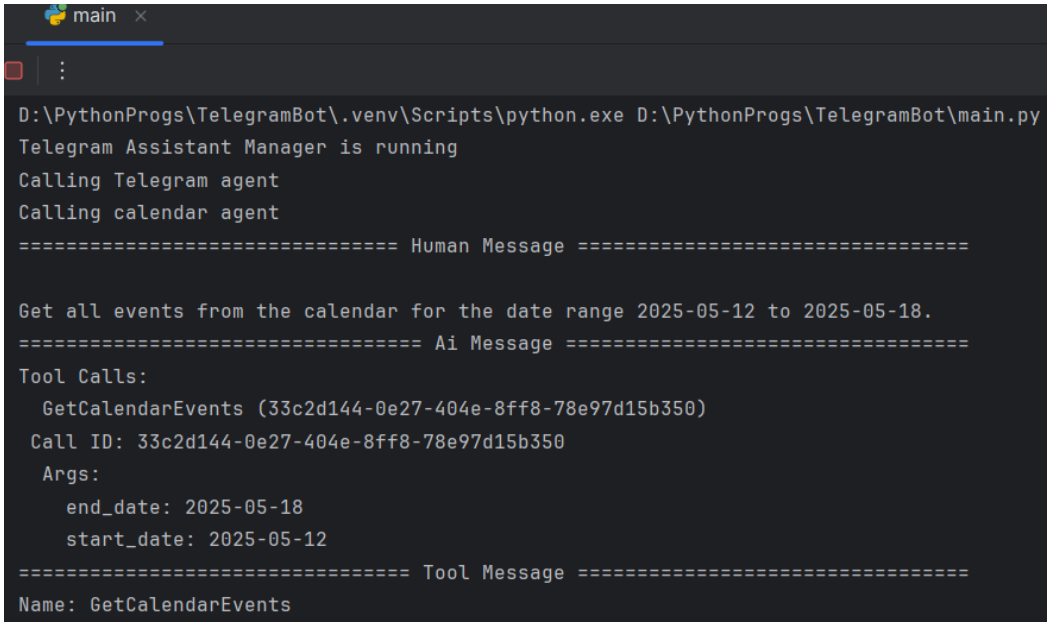


Рисунок 3.8 – Логування роботи асистента в консолі

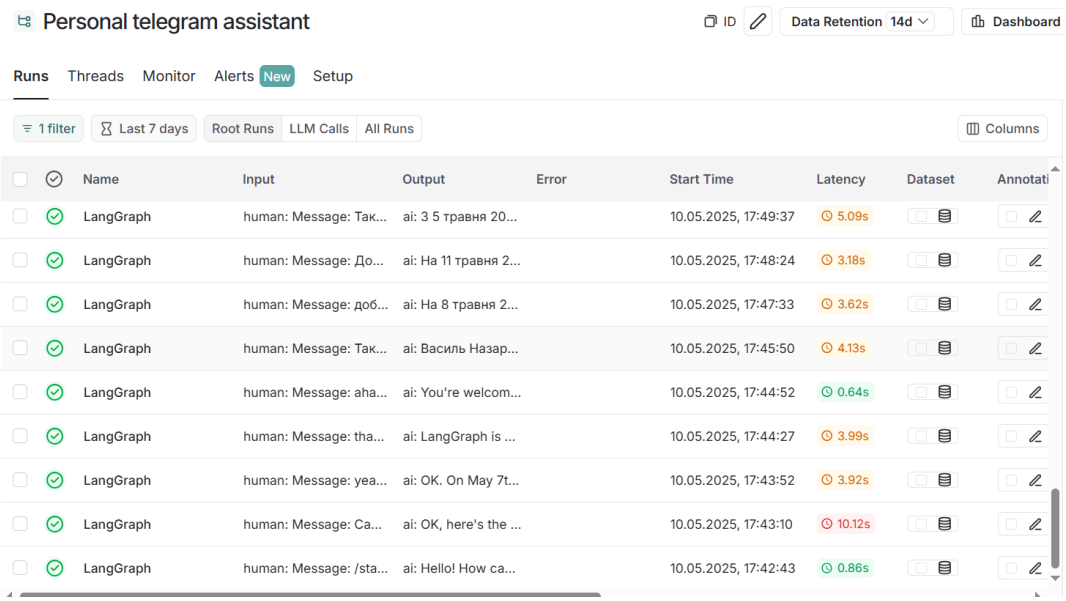


Рисунок 3.9 – Логування роботи асистента через LangSmith

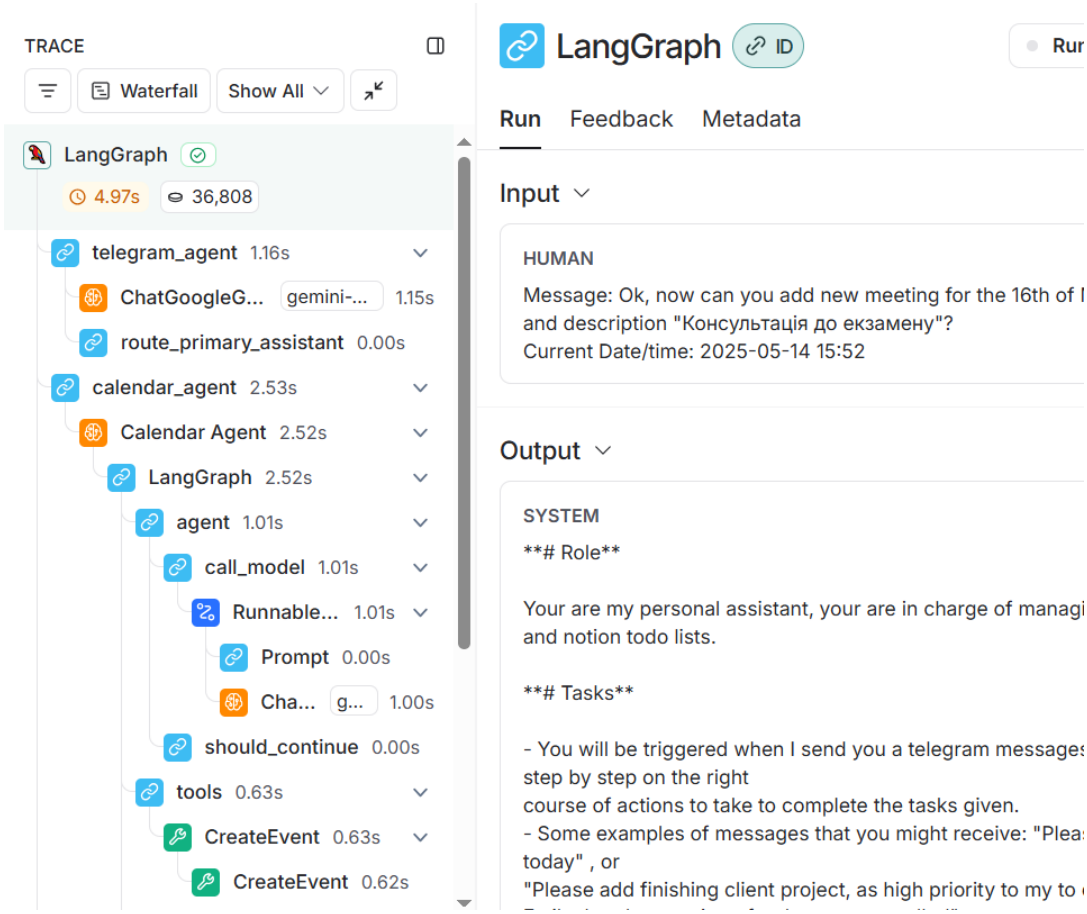


Рисунок 3.10 – Трасування обробки окремого запиту

Використання LangSmith суттєво допомогло у налагодженні логіки агентів, оптимізації інструкцій та забезпеченні коректної взаємодії між компонентами системи.

3.3 Оцінка результатів, напрями застосування та значущість

Отримані результати оцінюються як позитивні – реалізовано повноцінний прототип, який демонструє високу ефективність взаємодії, гнучкість у роботі з різними інструментами та здатність адаптуватися до мовного запиту користувача. Система коректно обробляє запити різними мовами, що є важливим елементом сучасного програмного забезпечення.

Водночас під час реалізації були проявлені певні обмеження, частина з яких була передбачена при проєктуванні:

– залежність від стабільності API зовнішніх сервісів;

- вразливість до неточних або надто складних запитів, які можуть потребувати уточнення;
- затримка у відповідях при роботів мовної моделі разом із запитами до API зовнішніх сервісів;
- орієнтація лише на персональне використання.

Деякі з перелічених проблеми є типовими для систем на базі LLM і частково вирішуються через налаштування інструкцій та покращення внутрішньої логіки агента-менеджера.

З огляду на світові тенденції, розроблений проєкт відповідає актуальним світовим тенденціям у галузі ШІ та розробки програмного забезпечення. Спостерігається стрімке зростання популярності великих мовних моделей та їх застосування для створення інтелектуальних агентів, згідно з дослідженнями [3]. Концепція автоматизації рутинних справ та створення персоналізованих ШІ-помічників, що інтегрують різноманітні сервіси для підвищення продуктивності, є одним із ключових напрямків розвитку як для індивідуальних користувачів, так і для корпоративного сектору. Дана робота демонструє практичну реалізацію цих тенденцій.

Результати даної дипломної роботи та розроблений прототип бота можуть знайти застосування у сферах особистого планування та продуктивності, а також слугувати основною для подальших розробок.

З наукової точки зору, дана робота демонструє практичне застосування агентно-орієнтованого підходу на базі LLM у реальному середовищі з використанням сучасних фреймворків. Вона ілюструє, як складні процеси можуть бути формалізовані у вигляді керованих робочих процесів. Господарська значущість полягає в потенціалі автоматизації рутинних задач користувача, що може зекономити значну кількість часу. Щодо соціальної значущості, то вона полягає у поширенні практики інтеграції ШІ у повсякденне життя.

ВИСНОВКИ

У результаті виконання даної дипломної роботи було успішно розроблено та програмно реалізовано персонального Telegram-бот помічника, що використовує велику мовну модель для інтелектуальної обробки запитів користувача та ефективної взаємодії з інтегрованими зовнішніми сервісами. Створений прототип продемонстрував здатність підвищувати особисту продуктивність користувача шляхом автоматизації рутинних завдань та централізації управління інформацією.

Створене рішення є прикладом ефективного використання архітектури багатоагентної системи. Особливо цінними елементами проєкту є:

- створення функціонального цифрового асистента;
- ефективне застосування великої мовної моделі, що дозволило забезпечити гнучке розуміння запитів від користувача;
- практична інтеграція з популярними сервісами, що реалізовано через механізми взаємодії з API.

У ході розробки довелося долати певні труднощі, зокрема пов'язані з роботою із API інтегрованих сервісів, забезпеченням стабільної роботи мовної моделі при різних запитах, а також налагодженням взаємодії між різними компонентами агентської системи. Ці випробування дозволили отримати цінний практичний досвід та глибше зрозуміти специфіку створення LLM-орієнтованих додатків.

Аналізуючи процес розробки, можна відзначити критичну важливість ітеративного підходу розробки кожного компонента. Якість та деталізація інструкцій для великої мовної моделі є визначальним фактором для коректної роботи агентів. Фреймворки LangChain та LangGraph значно спростили реалізацію взаємодії LLM із зовнішніми сервісами, дозволивши зосередитися на логіці самого додатка.

Розглядаючи перспективи подальшого розвитку та вдосконалення розробленого рішення, можна виділити наступні напрямки:

- розширення списку інтегрованих сервісів – наприклад підтримка систем управління проєктами чи взаємодія із хмарними сховищами;
- покращення продуктивності шляхом оптимізації запитів до зовнішніх сервісів;
- дослідження та інтеграція нових LLM, з метою використання найбільш підходящих моделей для конкретних груп завдань.

Окремо слід виділити можливість потенційної інтеграції протоколу взаємодії між агентами та зовнішніми ресурсами (MCP). Його концепція спрямована на уніфікацію способів передачі контекстної інформації великим мовним моделям та отримання від них структурованих даних, що розкрито в роботі [27]. Залучення цієї технології могло б дозволити оптимізувати взаємодію моделі із інструментами та удосконалити управління стану діалогу при взаємодії з LLM. Наприклад, стандартизована передача історії діалогу, вподобань користувача, результатів роботи інструментів та інших елементів контексту дозволила б LLM більш точно й надійно інтерпретувати запити, зменшуючи ймовірність нерелевантних дій. MCP міг би запропонувати більш ефективні механізми для опису можливостей інструментів, якими оперує агент, та для отримання від великої мовної моделі структурованих команд на їх виклик, що підвищило б надійність та керованість усієї системи. Інтеграція з MCP відкрила б шлях до створення більш інтероперабельних, гнучких та потужних цифрових асистентів.

Результати, досягнуті в ході виконання даної дипломної роботи, переконливо демонструють актуальність обраної теми. В епоху цифрової трансформації та інформаційного перевантаження потреба в інтелектуальних інструментах, здатних оптимізувати особисту продуктивність, лише зростає. Розроблений Telegram-бот помічник є практичним втіленням відповіді на цей запит, демонструючи, як синергія великих мовних моделей, сучасних

програмних фреймворків та популярних онлайн-сервісів може призвести до створення дійсно корисних та інноваційних рішень.

Загалом, виконана дипломна робота демонструє успішне застосування сучасних технологій штучного інтелекту для вирішення практично значущого завдання – створення персонального цифрового помічника. Розроблений прототип має потенціал для подальшого розвитку та може слугувати основою для створення більш досконалих інструментів підвищення власної продуктивності. Досвід, отриманий під час проєктування та реалізації, є цінним внеском у практичні знання в галузі комп'ютерних наук та штучного інтелекту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. David L. Poole, Alan K. Mackworth. Artificial Intelligence: Foundations of Computational Agents, 3rd Edition. Cambridge University Press, 2023. 800с.
2. Глибовець М. М., Олецький О.В. Штучний інтелект. Київ: ВД «Академія», 2002. 366с.
3. Haenlein M., Kaplan A. A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. ResearchGate, 2019. URL:https://www.researchgate.net/publication/334539401_A_Brief_History_of_Artificial_Intelligence_On_the_Past_Present_and_Future_of_Artificial_Intelligence. Дата звернення: 15.04.2025.
4. Deepak N. Y. Advancements in Natural Language Processing (NLP) and Its Applications in Voice Assistants and Chatbots. ResearchGate, 2023. URL:https://www.researchgate.net/publication/381790528_Advancements_in_Natural_Language_Processing_NLP_and_Its_Applications_in_Voice_Assistants_and_Chatbots. Дата звернення: 17.04.2025.
5. Asif R. RagaAI, Transforming Conversational AI with LLM. URL: https://raga.ai/blogs/conversational-ai-llm-transformation. Дата звернення: 23.04.2025.
6. Lung L. Qualimero, What is a Digital Assistant. URL: https://www.qualimero.com/en/blog/digital-assistants-ai-revolution. Дата звернення: 30.04.2025.
7. Personal, How an AI personal assistant can save time and boost efficiency. URL:https://www.personal.ai/pi-ai/how-an-ai-personal-assistant-can-save-time-and-boost-efficiency. Дата звернення: 30.04.2025.
8. Harris D. Thinglabs, The History of Siri: Apple’s Pioneering Digital Assistant. URL: https://thinglabs.io/the-history-of-siri-apples-pioneering-digital-assistant. Дата звернення: 03.05.2025.

9. Fedewa J. HowToGeek, What is Google Assistant, and What Can It Do? URL:https://www.howtogeek.com/692895/what-is-google-assistant-and-what-can-it-do/. Дата звернення: 03.05.2025.
10. Vaswani A., Shazer N., Parmer N., Uszkoreit J., Jones L., Kaiser L., Polosukhin I. Attention Is All You Need. ArXiv, 2017. URL: https://arxiv.org/abs/1706.03762. Дата звернення: 05.05.2025.
11. OpenAI, ChatGPT series. URL: https://openai.com/chatgpt/overview/. Дата звернення: 06.05.2025.
12. Google Research, Gemini AI. URL: https://ai.google/get-started/gemini-ecosystem/. Дата звернення: 06.05.2025.
13. Meta, LLaMA: Leading intelligence. URL: https://www.llama.com/. Дата звернення: 06.05.2025.
14. Yao S., Zhao J., Yu D., Du N., Shafran I., Narasimhan K., Cao Y. ReAct: Synergizing Reasoning and Acting in Language Models. ArXiv, 2022. URL: https://arxiv.org/abs/2210.03629. Дата звернення: 07.05.2025.
15. Bluecoding, Why Python Is Still the Best Option for Machine Learning and AI. URL: https://www.bluecoding.com/post/why-python-remains-the-top-choice-for-ai-and-machine-learning. Дата звернення: 08.05.2025.
16. LangChain, AI-first toolkit for developers when building with GenAI. URL: https://www.langchain.com/langchain. Дата звернення: 08.05.2025.
17. LangChain, LangGraph – Balance agent control with agency. URL: https://www.langchain.com/langgraph. Дата звернення: 08.05.2025.
18. Hajebi S. Medium, LangGraph is Not a True Agentic Framework. URL:https://medium.com/@saeedhajebi/langgraph-is-not-a-true-agentic-framework-3f010c780857. Дата звернення: 08.05.2025.

19. BotPenguin, Telegram Chatbot. URL: https://botpenguin.com/glossary/telegram-chatbot. Дата звернення: 09.05.2025.
20. Telegram APIs, Bot API. URL: https://core.telegram.org/. Дата звернення: 09.05.2025.
21. Auth0, What is OAuth 2.0. URL: https://auth0.com/intro-to-iam/what-is-oauth-2. Дата звернення: 10.05.2025.
22. Google Identity, Using OAuth 2.0 to Access Google APIs. URL:https://developers.google.com/identity/protocols/oauth2. Дата звернення: 10.05.2025.
23. Google Workspace, Gmail API Overview. URL: https://developers.google.com/workspace/gmail/api/guides. Дата звернення: 10.05.2025.
24. Google Workspace, Google Calendar API Overview. URL:https://developers.google.com/workspace/calendar/api/guides/overview. Дата звернення: 10.05.2025.
25. Notion, Notion API Overview. URL: https://developers.notion.com/docs/getting-started. Дата звернення: 10.05.2025.
26. Tavily, Connect Your LLM to the Web. URL: https://tavily.com/. Дата звернення: 11.05.2025.
27. Sergio. Zencoder, Model Context Protocol (MCP) – Everything You Need to Know. URL: https://zencoder.ai/blog/model-context-protocol. Дата звернення: 11.05.2025.

ДОДАТОК А
ОСНОВНИЙ ВИХІДНИЙ КОД ПРОГРАМИ

Лістинг А.1 – Основний цикл програми

```
import time
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI
from src.agents.manager_agent import ManagerAgent
from src.utils import send_telegram_message,
receive_telegram_message

# Load .env variables
load_dotenv()

# Use gemini for telegram manager agent
gemini = ChatGoogleGenerativeAI(model="gemini-2.0-flash",
temperature=0.1)

telegram_assistant = ManagerAgent(gemini)

def monitor_bot(after_timestamp, config):
    while True:
        messages = receive_telegram_message(after_timestamp)
        if messages:
            for message in messages:
                sent_message = (f"Message:
{message['text']}\n"
                                f"Current Date/time:
{message['date']}")
                answer =
telegram_assistant.invoke(sent_message, config)
                send_telegram_message(answer)
        after_timestamp = int(time.time())
        time.sleep(5)


if __name__ == "__main__":
    print("Telegram Assistant Manager is running")
    config = {"configurable": {"thread_id": 42}}
    initial_timestamp = int(time.time())
    monitor_bot(initial_timestamp, config)
```

Лістинг А.2 – Взаємодія з Telegram Bot API

```
import os, requests
from datetime import datetime

SCOPES = [
    "https://www.googleapis.com/auth/calendar.events",
```

```
        'https://www.googleapis.com/auth/contacts',
        "https://www.googleapis.com/auth/contacts.readonly",
        'https://www.googleapis.com/auth/gmail.readonly'
    ]

    def print_agent_output(output):
        for message in output["messages"]:
            message.pretty_print()

    def send_telegram_message(text):
        TOKEN = os.getenv("TELEGRAM_TOKEN")
        CHAT_ID = os.getenv("CHAT_ID")

        url                                                    =
f"https://api.telegram.org/bot{TOKEN}/sendMessage?chat_id={CHAT_
ID}&text={text}&amp;parse_mode=MarkdownV2"
        response = requests.get(url).json()
        if not response["ok"]:
            return "Failed to send message"

        return "Message sent successfully on Telegram"

    def receive_telegram_message(after_timestamp):
        TOKEN = os.getenv("TELEGRAM_TOKEN")
        url = f"https://api.telegram.org/bot{TOKEN}/getUpdates"
        response = requests.get(url).json()
        if not response["result"]:
            return []

        new_messages = []
        for update in response["result"]:
            if "message" in update:
                message = update["message"]
                if message["date"] > after_timestamp:
                    new_messages.append({
                        "text": message["text"],
                        "date":
datetime.fromtimestamp(message["date"]).strftime("%Y-%m-%d
%H:%M")
                    })

        return new_messages
```

Лістинг А.3 – Код агента що взаємодіє з Gmail

```
    from langgraph.prebuilt import create_react_agent
    from langchain_google_genai import ChatGoogleGenerativeAI
    from langsmith import traceable
    from  src.tools.email  import  SendEmail,  FindContactEmail,
ReadEmails
    from src.prompts import EMAIL_AGENT_PROMPT
    from src.utils import print_agent_output
```

```
    from dotenv import load_dotenv

    # Load .env variables
    load_dotenv()

    # Initialize LLM model
    model = ChatGoogleGenerativeAI(model="gemini-2.0-flash",
temperature=0.1)

    # Initialize the tools
    tools = [SendEmail(), FindContactEmail(), ReadEmails()]

    # Create the ReAct agent with the specified LLM, tools, system
prompt
    email_agent = create_react_agent(model, tools,
state_modifier=EMAIL_AGENT_PROMPT)


    @traceable(run_type="llm", name="Email Agent")
    def invoke_email_agent(task: str) -> str:
        inputs = {"messages": [("user", task)]}
        output = email_agent.invoke(inputs)
        print_agent_output(output)
        return output["messages"][-1].content
```

Лістинг А.4 – Код агента що взаємодіє з Google Calendar

```
    from langgraph.prebuilt import create_react_agent
    from langchain_google_genai import ChatGoogleGenerativeAI
    from langsmith import traceable
    from src.tools.calendar import CreateEvent, GetCalendarEvents
    from src.tools.email import FindContactEmail
    from src.prompts import CALENDAR_AGENT_PROMPT
    from src.utils import print_agent_output
    from dotenv import load_dotenv

    # Load .env variables
    load_dotenv()

    # Initialize LLM model
    model = ChatGoogleGenerativeAI(model="gemini-2.0-flash",
temperature=0.1)

    # Initialize the tools
    tools = [CreateEvent(), GetCalendarEvents()]

    # Create the ReAct agent with the specified LLM, tools, system
prompt
    calendar_agent = create_react_agent(model, tools,
state_modifier=CALENDAR_AGENT_PROMPT)

    @traceable(run_type="llm", name="Calendar Agent")
```

```
    def invoke_calendar_agent(task: str) -> str:
        inputs = {"messages": [("user", task)]}
        output = calendar_agent.invoke(inputs)
        print_agent_output(output)
        return output["messages"][-1].content
```

Лістинг А.5 – Код агента що взаємодіє з Notion

```
    from langgraph.prebuilt import create_react_agent
    from langchain_google_genai import ChatGoogleGenerativeAI
    from langsmith import traceable
    from src.tools.notion import GetMyTodoList,
AddTaskInTodoList, SetMyTaskStatus
    from src.prompts import NOTION_AGENT_PROMPT
    from src.utils import print_agent_output
    from dotenv import load_dotenv

    # Load .env variables
    load_dotenv()

    # Initialize LLM model
    model = ChatGoogleGenerativeAI(model="gemini-2.0-flash",
temperature=0.1)

    # Initialize the tools
    tools = [GetMyTodoList(), AddTaskInTodoList(),
SetMyTaskStatus()]

    # Create the ReAct agent with the specified LLM, tools, system
prompt
    notion_agent = create_react_agent(model, tools,
state_modifier=NOTION_AGENT_PROMPT)


    @traceable(run_type="llm", name="Notion Agent")
    def invoke_notion_agent(task: str) -> str:
        inputs = {"messages": [("user", task)]}
        output = notion_agent.invoke(inputs)
        print_agent_output(output)
        return output["messages"][-1].content
```

Лістинг А.6 – Код агента-менеджера

```
    import sqlite3
    from typing import Annotated, Literal, TypedDict
    from langgraph.checkpoint.sqlite import SqliteSaver
    from langgraph.graph import END, START, StateGraph
    from langgraph.prebuilt import tools_condition
    from langgraph.graph.message import AnyMessage, add_messages
    from langchain_core.messages import HumanMessage,
SystemMessage, ToolMessage
```

```
    from src.prompts.manager_agent_prompt import
TELEGRAM_ASSISTANT_MANAGER_PROMPT
    from src.agents.email_agent import invoke_email_agent
    from src.agents.calendar_agent import invoke_calendar_agent
    from src.agents.notion_agent import invoke_notion_agent
    from src.tools.delegate_tool import Delegate
    from langchain_community.tools.tavily_search import
TavilySearchResults

    # Initialize sqlite3 DB
    conn = sqlite3.connect("db/checkpoints.sqlite",
check_same_thread=False)

    class State(TypedDict):
        messages: Annotated[list[AnyMessage], add_messages]

    class ManagerAgent:
        def __init__(self, llm):
            self.tavily_tool =
TavilySearchResults(max_results=1)
            self.tools = [Delegate(), self.tavily_tool]
            self.model = llm.bind_tools(self.tools)
            self.checkpointer = SqliteSaver(conn)
            self.workflow = self.create_workflow()

        def create_workflow(self):
            workflow = StateGraph(State)

            workflow.add_node("telegram_agent", self.call_model)
            workflow.add_node("email_agent",
self.call_email_agent)
            workflow.add_node("calendar_agent",
self.call_calendar_agent)
            workflow.add_node("notion_agent",
self.call_notion_agent)

            workflow.add_edge(START, "telegram_agent")

            workflow.add_conditional_edges(
                "telegram_agent",
                self.route_primary_assistant,
                {
                    "email_agent": "email_agent",
                    "calendar_agent": "calendar_agent",
                    "notion_agent": "notion_agent",
                    END: END,
                },
            )

            workflow.add_edge("email_agent", 'telegram_agent')
            workflow.add_edge("calendar_agent",
'telegram_agent')
            workflow.add_edge("notion_agent", 'telegram_agent')
```

```
            self.app = workflow.compile(self.checkpointer)

            return self.app

        def call_model(self, state: State):
            print("Calling Telegram agent")
            messages = state['messages']
            response = self.model.invoke(messages)
            return {"messages": [response]}

        def route_primary_assistant(
                self,
                state: State,
        ) -> Literal[
            "email_agent",
            "calendar_agent",
            "notion_agent",
            "search_agent",
            "__end__",
        ]:
            route = tools_condition(state)
            if route == END:
                return END
            tool_calls = state["messages"][-1].tool_calls
            if tool_calls:
                if tool_calls[0]["name"] == Delegate.__name__:
                    if  tool_calls[0]['args']["agent_name"]  ==
"Email Agent":
                        return "email_agent"
                    elif tool_calls[0]['args']["agent_name"] ==
"Calendar Agent":
                        return "calendar_agent"
                    elif tool_calls[0]['args']["agent_name"] ==
"Notion Agent":
                        return "notion_agent"
                # Handle search tool call if available
                elif          tool_calls[0]["name"]          ==
"tavily_search_results_json":
                    return "search_agent"
            raise ValueError("Invalid route")

        def call_email_agent(self, state: State):
            print("Calling email agent")
            last_manager_tool_calls    =    state["messages"][-
1].tool_calls
            task = last_manager_tool_calls[0]['args']["task"]
            tool_call_id = last_manager_tool_calls[0]['id']
            response = invoke_email_agent(task)

            return  {"messages":  [ToolMessage(content=response,
name="Delegate", tool_call_id=tool_call_id)]}
```

```
        def call_calendar_agent(self, state: State):
            print("Calling calendar agent")
            last_manager_tool_calls    =    state["messages"][-
1].tool_calls
            task = last_manager_tool_calls[0]['args']["task"]
            tool_call_id = last_manager_tool_calls[0]['id']
            response = invoke_calendar_agent(task)

            return  {"messages":  [ToolMessage(content=response,
name="Delegate", tool_call_id=tool_call_id)]}

        def call_notion_agent(self, state: State):
            print("Calling notion agent")
            last_manager_tool_calls    =    state["messages"][-
1].tool_calls
            task = last_manager_tool_calls[0]['args']["task"]
            tool_call_id = last_manager_tool_calls[0]['id']
            response = invoke_notion_agent(task)

            return  {"messages":  [ToolMessage(content=response,
name="Delegate", tool_call_id=tool_call_id)]}

        def call_search_agent(self, state: State):
            print("Calling search agent")
            last_tool_calls = state["messages"][-1].tool_calls
            tool_call_id = last_tool_calls[0]['id']

            # Get the raw search results
            search_results  =  self.tavily_tool.invoke({'query':
last_tool_calls[0]['args']['query']})

            # Format the search results
            formatted_results = "\n\n".join(
                f"Title:                {result['title']}\nURL:
{result['url']}\nContent: {result['content']}"
                for result in search_results
            )

            return {"messages": [ToolMessage(
                content=f"I             found             these
results:\n\n{formatted_results}",
                name="tavily_search_results_json",
                tool_call_id=tool_call_id
            )]}

        def invoke(self, message, config):
            if
len(self.app.get_state(config=config).values.get("messages", []))
== 0:
                self.app.update_state(config,       {"messages":
[SystemMessage(content=TELEGRAM_ASSISTANT_MANAGER_PROMPT)]})
            sent_message = HumanMessage(content=message)
```

```
            final_state      =      self.app.invoke({"messages":
[sent_message]}, config=config)
            return final_state["messages"][-1].content
```