Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Інтелектуальний асистент для керування операційною системою на основі мультиагентної технології з використанням великих мовних моделей
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-5
Данило Крижній
(власне ім'я, прізвище)
Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник доц. Олександр Шевченко
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ ________________ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____» __________ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Крижнію Данилу Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Інтелектуальний асистент для керування операційною системою на основі мультиагентної технології та використання великих мовних моделей

затверджена наказом університету від 19 травня 20 25 р. № 378Ст
2. Термін подання студентом роботи до екзаменаційної комісії 18 червня 20 25 р.
3. Вихідні дані до роботи Інформація про LLM, LlamaIndex, Ollama, Electron, FastAPI, Windows 11, React-агенти, Python, OpenAI API.

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі
2) Проєктування системи-асистента
3) Програмна реалізація

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	19.05.2025	виконано
2	Дослідження предметної галузі	19.05.2025 – 23.05.2025	виконано
3	Постановка задачі	23.05.2025 – 25.05.2025	виконано
4	Проєктування системи	25.05.2025 – 27.05.2025	виконано
5	Програмна реалізація	27.05.2025 – 01.06.2025	виконано
6	Оформлення пояснювальної записки	01.06.2025 – 03.06.2025	виконано
7	Підготовка до захисту	03.06.2025 – 18.06.2025	виконано
8	Захист	18.06.2025	виконано

Дата видачі завдання _19_ _травня_ 20_25_ р.

Здобувач ______________________
(підпис)

Керівник роботи ______________________ доц. Олександр Шевченко
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 84 с., 11 рис., 3 дод., 20 джерел.

АГЕНТ, АСИСТЕНТ, МУЛЬТИАГЕНТНА СИСТЕМА, ШТУЧНИЙ ІНТЕЛЕКТ, LLM, POWERSHELL, RAG, REACT AGENT, WINDOWS.

Кваліфікаційна робота спрямована на розроблення інтелектуального асистента для керування операційною системою Windows на основі мультиагентної архітектури з використанням великих мовних моделей.

Об’єктом дослідження є процес взаємодії користувача з операційною системою за допомогою системи асистента. Предметом дослідження є програмна реалізація мультиагентної системи, що здатна інтерпретувати запити природною мовою, перетворювати їх на системні дії та безпечно виконувати ці дії в середовищі Windows.

Метою даної роботи є створення функціонального прототипу асистента, який дозволяє автоматизувати рутинні дії в ОС, підвищити доступність її інтерфейсів та забезпечити гнучке розширення можливостей системи.

У межах дослідження було проведено аналіз агентних систем, архітектури ReAct-агентів, принципів роботи LLM, а також розглянуто існуючі аналогічні продукти. Практична частина включала проєктування багаторівневої архітектури системи, реалізацію GUI, налаштування взаємодії з LLM та розробку спеціалізованих агентів для виконання системних команд.

Таким чином, дана кваліфікаційна робота вирішує актуальну проблему підвищення зручності взаємодії з операційними системами шляхом використання технологій штучного інтелекту та мультиагентного підходу. Розроблена система має потенціал для подальшого розвитку, масштабування та практичного впровадження.

ABSTRACT

Bachelor’s thesis contains: 84 pp., 11 fig., 3 ann., 20 references.

AGENT, ARTIFICIAL INTELLIGENCE, ASSISTANT, LLM, MULTI-AGENT SYSTEM, POWERSHELL, RAG, REACT AGENT, WINDOWS.

The qualification work is aimed at developing an intelligent assistant for managing the Windows operating system based on a multi-agent architecture using large language models.

The object of research is the process of user interaction with the operating system using the assistant system. The subject of the study is the software implementation of a multi-agent system capable of interpreting natural language queries, converting them into system actions, and safely performing these actions in the Windows environment.

The aim of this work is to create a functional prototype of an assistant that allows automating routine actions in the OS, increasing the accessibility of its interfaces and providing flexible expansion of the system's capabilities.

The research included an analysis of agent systems, ReAct agent architecture, LLM principles, and existing similar products. The practical part included designing a multi-level system architecture, implementing a GUI, setting up interaction with LLM, and developing specialized agents to execute system commands.

Thus, this qualification work solves the urgent problem of increasing the convenience of interaction with operating systems by using artificial intelligence technologies and a multi-agent approach. The developed system has the potential for further development, scaling and practical implementation.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 8

Вступ 9

1 Аналіз предметної галузі 10

1.1 Агентні системи 10

1.1.1 Основні поняття про програмні агенти 10

1.1.2 Типи програмних агентів 12

1.1.3 ReAct агенти 13

1.2 Великі мовні моделі (LLM) 15

1.2.1 Визначення та основні характеристики 15

1.2.2 Опис архітектури мовних моделей 17

1.2.3 Використання LLM в агентних системах 22

1.2.4 Обмеження та виклики використання LLM 24

1.3 Аналіз існуючих аналогів 26

1.3.1 Microsoft Copilot 26

1.3.2 Google Gemini 27

1.3.3 Braina 28

2 Проєктування системи-асистента 29

2.1 Формулювання вимог до системи 29

2.1.1 Функціональні вимоги 29

2.1.2 Нефункціональні вимоги 31

2.2 Особливості архітектури системи 32

2.2.1 Загальний огляд архітектури 32

2.2.2 Взаємодія між модулями 34

2.3 Технологічний стек та інструменти розробки 36

2.3.1 Взаємодія з операційною системою 36

2.3.2 Платформа локальних великих мовних моделей 37

2.3.3 Робота з LLM та агентами 38

2.3.4 Графічний інтерфейс користувача 38

3 Програмна реалізація 40
3.1 Загальна структура програмної реалізації 40
3.2 Реалізація агентної структури 41
3.2.1 Загальна архітектура агентної системи 41
3.2.2 Опис функціональних агентів системи 42
3.2.3 Взаємодія агентів у системі 45
3.3 Генерація few-shot прикладів для покращення роботи агентів 48
3.4 Приклади використання системи 51
3.4.1 Приклад 1 – запис інформації в новий файл 52
3.4.2 Приклад 2 – керування мультимедіа 52
3.4.3 Приклад 3 – виконання системної команди через PowerShell .. 53
3.4.4 Приклад 4 – запуск встановленої програми 54
3.4.5 Приклад 5 – пошук інформації в інтернеті 54
3.5 Демонстрація роботи програми 55
3.5.1 Загальний огляд інтерфейсу користувача 55
3.5.2 Демонстрація виконання запиту 58
Висновки 62
Перелік джерел посилання 64
Додаток А Програмний код 67
Додаток Б Запити до LLM 80
Додаток В Відомість кваліфікаційної роботи 84

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОС – операційна система;

ПЗ – програмне забезпечення;

ШІ – штучний інтелект;

API – Application Programming Interface – інтерфейс програмування додатків;

JSON – JavaScript Object Notation – нотація об’єктів JavaScript;

LLM – Large Language Model – велика мовна модель;

ReAct – Reasoning and Acting – міркування та дія.

ВСТУП

Сучасні операційні системи стають все більш простими для управління, але все ще взаємодія з ними часто потребує просунутих навичок, таких як знання командного рядка, спеціальних утиліт та глибокого розуміння налаштувань, що може стати перешкодою для багатьох користувачів.

Стрімкий розвиток великих мовних моделей (LLM) дозволяє проводити обробку запитів природною мовою, що спрощує кооперацію людини та комп'ютера. Інтелектуальний асистент може значно підвищити зручність використання ОС, зменшити необхідність ручного виконання рутинних завдань і розширити доступність системних функцій для широкого кола користувачів.

Метою даної роботи є розробка системи-асистента для операційної системи Windows, яка дозволяє користувачу автоматизувати та виконувати різні дії (керування файлами, запуск і встановлення програм, зміна налаштувань тощо) за допомогою запитів природною мовою. Основу системи складатимуть агенти на базі великих мовних моделей (LLM), що аналізуватимуть запити користувача, перетворюватимуть їх у відповідні системні команди та виконуватимуть їх.

Актуальність теми зумовлена зростаючою потребою в спрощенні взаємодії з комп'ютерними системами та збільшенням ролі інтелектуальних інтерфейсів у різних сферах діяльності. Розробка такого асистента має практичне значення для адміністрування систем, автоматизації робочих процесів та впровадження штучний інтелект у повсякденне користування

У роботі розглядаються принципи побудови агентних систем, застосування LLM для інтерпретації запитів та способи безпечного виконання системних команд. Результатом є функціональний прототип, здатний розуміти запити природною мовою та виконувати необхідні дії у Windows-середовищі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Агентні системи

1.1.1 Основні поняття про програмні агенти

Програмний агент – це комп’ютерна програма, яка діє від імені користувача або іншої програми в умовах певного середовища, приймаючи рішення та виконуючи дії для досягнення поставлених цілей без постійного втручання людини.

Термін «агент» походить від латинського agere («діяти») і підкреслює здатність такого програмного забезпечення самостійно обирати доречні дії в межах наданих йому повноважень.

Програмні агенти мають наступні властивості:

– автономність – агенти можуть приймати рішення та виконувати завдання незалежно від прямого керування користувачем;

– реактивність – здатність сприймати зміни в навколишньому середовищі та своєчасно реагувати на них;

– проактивність – ініціативна поведінка, спрямована на досягнення внутрішніх цілей, навіть за відсутності зовнішніх стимулів;

– соціальна здатність – можливість взаємодіяти та координувати дії з іншими агентами чи користувачами через спеціалізовані протоколи обміну повідомленнями;

– навчання та адаптивність – деякі агенти здатні покращувати свою поведінку на основі попереднього досвіду та змінювати внутрішні параметри для підвищення ефективності [1].

У контексті агентних систем програмний агент розглядається як спеціалізована обчислювальна одиниця, яка діє від імені користувача або іншої програми в певному середовищі і має змогу самостійно приймати рішення та виконувати дії для досягнення поставлених цілей без постійного

втручання людини. Походячи від латинського «agere» («діяти»), поняття «агент» підкреслює здатність програми автономно обирати доречні реакції на події середовища й ініціювати власні стратегії поведінки.

Автономність агента означає, що він управляє власними внутрішніми станами й ресурсами, самостійно обираючи, коли і які завдання виконувати, водночас реактивність забезпечує безперервний моніторинг середовища й своєчасну відповідь на його зміни. Проактивність виражається в плануванні й ініціюванні дій задля досягнення внутрішніх цілей навіть за відсутності зовнішніх стимулів, а соціальна здатність дозволяє координувати свої дії з іншими агентами чи користувачами за допомогою стандартних протоколів обміну повідомленнями. Деякі агенти додатково наділені механізмами навчання, що дають змогу адаптувати стратегії поведінки на основі попереднього досвіду й оптимізувати результати своєї діяльності з часом.

Класифікація агентів за рівнем когнітивних здібностей виділяє реактивні системи, які діють за фіксованими правилами «умова–дія» без внутрішнього моделювання світу, адаптивні – із простим відстеженням і коригуванням параметрів на основі спостережень, а також когнітивні агенти, які мають репрезентацію зовнішнього середовища і застосовують парадигму BDI (Beliefs, Desires, Intentions) для вибору стратегічних планів дій. За характером розгортання програмні агенти можуть бути розподіленими, коли їхні компоненти функціонують на різних вузлах мережі для спільного вирішення задач, або мобільними – здатними переміщувати свою логіку між вузлами, зберігаючи стан виконання.

Розрізнення за рівнем інтелекту виділяє прості «softbots», що автоматизують рутинні операції без складного міркування, та інтелектуальні агенти, які застосовують методи штучного інтелекту для висновків, навчання й адаптації до нових умов. Архітектура агента – реактивна, делібератива (BDI) або гібридна – визначає структуру його внутрішніх компонентів: набір правил «умова–дія», модель переконань-

намірів-бажань для планування чи комбіновану схему, що поєднує швидкість реакції з можливістю стратегічного планування.

Середовище, в якому функціонує агент, характеризується ступенем доступності інформації (повна або часткова сприйнятливість), динамічністю (наявність змін незалежно від дій агента), детермінованістю (передбачуваність наслідків дій) і мультиагентністю, коли інші агенти також впливають на стан середовища й результат спільної діяльності. Таким чином, програмний агент у багатофункціональній системі поєднує автономне сприйняття, планування, взаємодію та адаптивність для ефективного виконання поставлених завдань [2].

1.1.2 Типи програмних агентів

Залежно від принципу роботи та складності реалізації, програмні агенти можна розділити на кілька ключових категорій:

– прості рефлекторні агенти – агенти працюють за принципом «умова–дія». Вони аналізують поточний стан навколишнього середовища через сенсори та відповідно до заздалегідь визначених правил виконують відповідні дії. Важливо відзначити, що такі агенти не зберігають інформацію про попередні стани та не мають механізмів прогнозування;

– модельні рефлекторні агенти, відмінністю яких є використання внутрішньої моделі світу. Це дозволяє враховувати попередні стани та робити більш обґрунтовані рішення. Такі агенти більш ефективні в складних динамічних середовищах;

– цільові агенти, головна особливість яких – орієнтація на досягнення конкретної мети. Вони не просто реагують на зміну середовища, а прораховують дії, які приведуть до бажаного результату. Цільові агенти часто використовуються в системах планування та оптимізації;

– утилітарні агенти – від цільових агентів їх відрізняє здатність оцінювати всі можливі варіанти дій та вибирати найбільш ефективний,

виходячи з функції корисності. Вони можуть враховувати різноманітні параметри: швидкість виконання, ресурсозатратність, ризики тощо;

– навчальні агенти, це найскладніший тип агентів, які мають механізми навчання та вдосконалення на основі отриманого досвіду. Вони використовують методи машинного навчання та штучного інтелекту, що дозволяє їм адаптуватися до нових умов та покращувати ефективність своєї роботи [3].

1.1.3 ReAct агенти

ReAct (Reasoning + Acting) агенти – це сучасна архітектура штучного інтелекту, яка поєднує логічне міркування та виконання дій у єдиному циклі. Вона була представлена в дослідженні 2022 року «ReAct: Synergizing Reasoning and Acting in Language Models» Шунью Яо та ін [4]. Ці агенти поєднують здатність LLM до логічного мислення (наприклад, ланцюг думок, chain-of-thought) з можливістю виконувати конкретні дії, створюючи систему, яка може ефективно взаємодіяти з оточенням і вирішувати складні завдання. Наприклад, якщо агент стикається з неочікуваною інформацією, він може скоригувати свій план на основі логічного аналізу.

На відміну від традиційних систем, які розділяють процеси прийняття рішень та виконання завдань, ReAct агенти реалізують інтегрований цикл: міркування, дія, спостереження. Цей підхід дозволяє агентам адаптуватися до нових умов, враховувати попередній досвід та забезпечувати прозорість у прийнятті рішень.

Архітектура ReAct агентів зазвичай включає велику мовну модель, яка служить «мозком» агента, пам’ять для збереження історії взаємодій, набір зовнішніх інструментів (API, бази даних тощо) та підказки (prompts), що визначають поведінку агента та формат його відповідей. Завдяки такій структурі, ReAct агенти можуть ефективно розбивати складні завдання на

підзадачі, адаптуватися до нових умов та забезпечувати прозорість у прийнятті рішень [5].

Архітектуру ReAct-агентів наведена на рисунку 1.1.

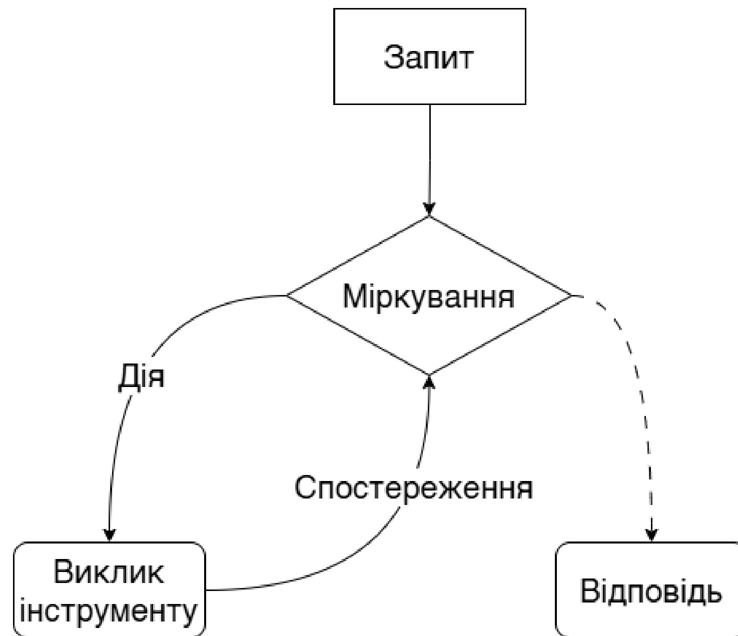

Рисунок 1.1 – Діаграма циклу роботи ReAct-агенту

Дії, зі свого боку, дозволяють агенту взаємодіяти з зовнішніми джерелами, такими як бази знань чи симульовані середовища. Це забезпечує доступ до додаткової інформації, що ґрунтує процес мислення в реальних даних. Наприклад, у завданнях перевірки фактів ReAct використовує Wikipedia для зменшення галюцинацій (помилкових відповідей, які часто виникають у традиційних методах).

Цей підхід робить ReAct гнучким і адаптивним до різних доменів, забезпечуючи прозорість у процесі прийняття рішень. ReAct дозволяє LLM координувати все, від простого задач пошукової доповненої генерації (RAG) до складних багатостепеневих робочих процесів.

Корисність агентів ReAct значною мірою випливає з основних якостей архітектури ReAct:

– універсальність – агенти ReAct можуть бути налаштовані на взаємодію з широким колом зовнішніх інструментів і API. Хоча налаштування відповідних промптів ReAct може покращити продуктивність, початкова конфігурація самої моделі не потрібна для виклику інструментів;

– адаптивність – агенти ReAct можуть використовувати свій процес міркування для пристосування до нових задач. Особливо в умовах роботи з довгим контекстом або при наявності зовнішньої пам'яті, вони здатні вчитися на минулих помилках, щоб долати непередбачувані перешкоди та ситуації;

– пояснюваність – процес міркування агента ReAct, що озвучується під час виконання, легко відслідковується, що полегшує налагодження та робить агентів відносно зручними для розробки й оптимізації;

– точність – міркування за ланцюжком думок (Chain of Thought) саме по собі має багато переваг для LLM, але також підвищує ризик галюцинацій. Поєднання цієї технології разом з доступом до зовнішніх джерел інформації значно знижує кількість галюцинацій, що робить агентів ReAct точнішими й надійнішими [6].

1.2 Великі мовні моделі (LLM)

1.2.1 Визначення та основні характеристики

Великі мовні моделі, також відомі як LLM – це дуже великі моделі глибокого навчання, які попередньо навчаються на великих обсягах даних. За останні роки саме ці моделі штучного інтелекту стали основою багатьох систем обробки та генерації природної мови. Завдяки здатності аналізувати та створювати текст, що нагадує людське мовлення, LLM знаходять широке застосування в різних сферах – від чат-ботів до автоматичного перекладу та генерації коду.

LLM – це тип алгоритмів глибинного машиного навчання, які здатні розуміти та генерувати текст, що максимально подібний до людського мовлення. Завдяки величезним обсягам тренувальної інформації, ці моделі здатні розпізнавати різні мовні конструкції та шаблони, брати до уваги контекст та семантичні зв'язки. Великі мовні моделі здатні виконувати разноманітні завдання, такі як генерація текстів, відповіді на запитання, переклад текстів та керувати агентськими системами [7].

Однією з ключових особливостей LLM є їхній великий розмір – кількість параметрів зазвичай може сягати мільярдів. Це дозволяє моделям обробляти складні мовні структури та забезпечує високу точність у виконанні завдань та здатність відтворювати знання.

Великі мовні моделі зазвичай проходять двоетапне навчання: спочатку моделі навчаються на великих загальних наборах тексту (попереднє навчання), а потім адаптуються до конкретних завдань за допомогою донавчання на спеціалізованих даних. Цей підхід дозволяє моделям ефективно переносити знання між різними завданнями.

LLM демонструють високу гнучкість у виконанні різноманітних завдань обробки природної мови. Зазвичай одна модель може виконувати різноманітні дії з текстом та не потребувати додаткового донавчання. Наприклад, сучасні великі мовні моделі можуть виконувати такі завдання, як переклад тексту, створення зведень, генерація тексту за запитом, аналіз настроїв та інші.

У використанні великих мовних моделей, можна виділити наступні переваги:

– автоматизація процесів – LLM дозволяють автоматизувати завдання, які раніше вимагали людського втручання, підвищуючи ефективність і продуктивність;

– персоналізація – моделі можуть адаптувати відповіді до індивідуальних потреб користувачів, забезпечуючи персоналізований досвід;

– мультимодальність – сучасні LLM здатні обробляти не лише текст, але й інші форми даних, такі як зображення та аудіо, що розширює їхні можливості застосування [8].

Більшість сучасних LLM базуються на архітектурі трансформерів, яка вперше була представлена в роботі «Attention Is All You Need» [9]. Ця архітектура використовує механізм самоуваги (self-attention), що дозволяє моделі ефективно враховувати контекст усього тексту при обробці кожного слова.

1.2.2 Опис архітектури мовних моделей

Архітектура великих мовних моделей визначається рядом факторів, таких як цілі конкретного дизайну моделі, наявні обчислювальні ресурси та тип завдань обробки мови, які має виконувати LLM. Загальна архітектура LLM складається з багатьох шарів, таких як повнозв’язані шари, шари ембедінгів (embeddings-layer) та шари уваги (attention-layer). Текст у векторному представлені об’єднується для створення прогнозів.

Трансформерні моделі, які здійснили революцію у завданнях обробки природної мови, зазвичай мають загальну архітектуру, що складається з наступних компонентів:

– вхідний шар у векторних представлень (далі зазначені як ембедінги чи вбудування);

– механізм самоуваги (self-attention);

– багатоголова увага (multi-head attention);

– нейромережа з повнозв’язаних шарів;

– шари декодера.

Загалом, найпростіша модель транформера складається з двох частин – енкодера (кодувальника) та декодера (декодувальника). Візуально структуру моделі трансформера наведено на рисунку 1.1.

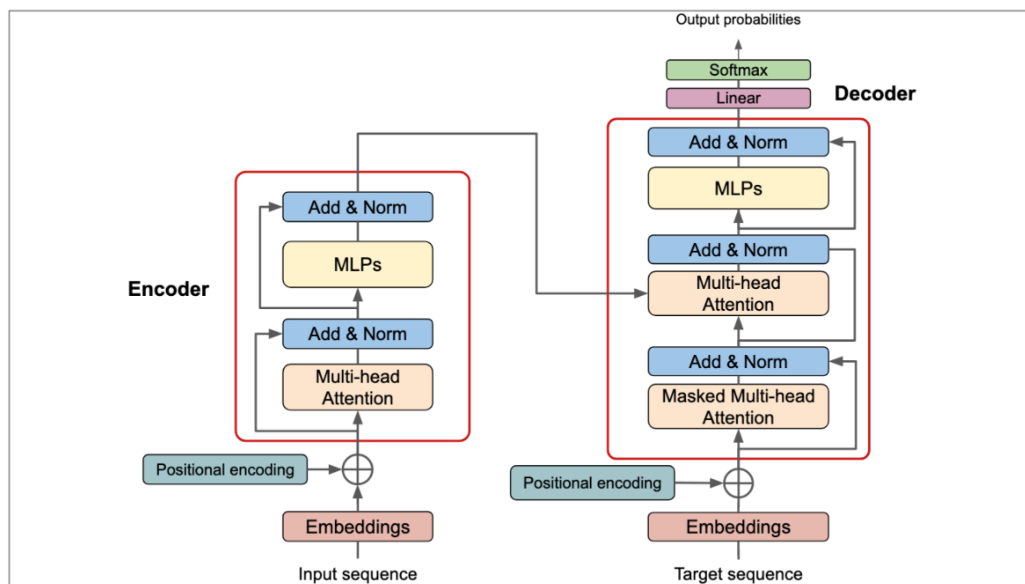

Рисунок 1.1 – Архітектура трансформера [10]

Першим елементом енкодера є вхідний шар вбудувань, який на першому етапі перетворює вхідний текст у форматі токенів (слова або частини слів) у набір векторних представлень. Отримані представлення на даному етапі не несуть ніякої інформації про їх позицію в тексті. Задля передачі цього контексту в традиційному транформері використовують позиційне кодування, що змінює ембедінги таким чином, що вони ефективно репрезентують позицію токена в тексті.

Рівень кодера служить для перетворення всіх вхідних послідовностей у безперервне, абстрактне представлення, яке укладає вивчену інформацію з усієї послідовності. Цей рівень складається з двох підмодулів: багатоголовий механізм уваги та повнозв’язана нейрона мережа.

У модулі кодера використовується спеціальний різновид уваги – механізм самоуваги (self-attention). Цей підхід дозволяє моделі співвідносити кожне слово у вхідній послідовності з усіма іншими словами, що допомагає краще зрозуміти контекст. Наприклад, модель може навчитися пов’язувати слово «твої» зі словом «справи» у запитанні «Як твої справи?».

Такий механізм дозволяє фокусуватися на різних частинах вхідної послідовності під час обробки кожного окремого токена. Для цього обчислюються оцінки уваги (attention scores) на основі трьох основних компонентів:

– запит (Query) – це вектор, який представляє певне слово або токен у механізмі уваги;

– ключ (Key) – також вектор, що відповідає кожному слову або токену у вхідній послідовності;

– значення (Value) – асоціюється з ключем і використовується для формування виходу шару уваги.

Коли запит добре збігається з ключем (тобто має високу оцінку уваги), відповідне значення більше впливає на результат. Такий підхід дозволяє моделі враховувати контекст всієї послідовності при обробці окремих елементів.

На відміну від звичайної уваги, багатоголовий механізм розбиває запити, ключі та значення на певну кількість голів, у яких самоувага виконується паралельно. Кожна голова незалежно навчається акцентувати увагу на різних аспектах вхідної інформації. Отримані результати з усіх голів об’єднуються та проходять через фінальний повнозв’язний шар.

Розглянемо архітектуру багатоголового механізму уваги. Першим етапом є множення матриць, а саме добутку запиту на ключ. Запити та ключі проходять через лінійні перетворення, після чого обчислюється добуток матриць (dot-product) між запитами та ключами. Це створює матрицю оцінок (score matrix), яка визначає, на які слова слід звертати більше уваги. Чим вищий бал – тим сильніший контекстуальний зв’язок. Також щоб уникнути великих значень добутку, кожен елемент матриці оцінок ділиться на квадратний корінь з розміру векторів запиту/ключа. Це дозволяє зробити градієнти під час навчання стабільнішими.

Далі відмасштабовані оцінки проходять через функцію softmax, яка перетворює їх у ймовірності від 0 до 1. Таким чином, модель акцентує увагу на більш важливих словах, зменшуючи вплив менш релевантних.

Отримані ваги уваги множаться на відповідні вектори значень. В результаті формується вихідний вектор, у якому переважає інформація від слів з високими оцінками softmax.

Кожна голова генерує свій вихід. Усі ці результати об’єднуються та передаються через фінальний лінійний шар, який формує підсумкове представлення. Завдяки паралельному навчанню кожна голова може вивчити різні залежності, що дозволяє моделі краще розуміти контекст вхідної інформації.

Після кожного підмодуля у складі шару кодера (зокрема після механізму багатоголової самоуваги) виконується шарова нормалізація (Layer Normalization), яка стабілізує та прискорює навчання. Крім того, в архітектурі трансформера застосовується залишкове з’єднання (Residual Connection): вихід підмодуля додається до його вхідного значення. Така операція допомагає уникнути проблеми зникнення градієнтів та дозволяє ефективно тренувати глибші моделі.

Наступним етапом обробки нормалізованого виходу є проходження через позиційно-незалежну повнозв’язну нейронну мережу, яка складається з двох лінійних шарів з функцією активації ReLU між ними. Цей підмодуль працює над покращенням представлення кожного токена незалежно від інших.

Вихід цієї мережі знову об’єднується із вхідним сигналом через залишкове з’єднання, після чого застосовується ще одна нормалізація. Це дозволяє мережі зберігати важливу інформацію з попередніх етапів і водночас збагачувати її новими ознаками.

На виході останнього шару кодера отримується набір векторів, кожен з яких репрезентує відповідний елемент вхідної послідовності, з

урахуванням глобального контексту. Ці вектори далі використовуються як вхід для декодера у трансформер-моделі.

Кодер можна уявити як вежу, що складається з N шарів, кожен з яких включає механізм уваги, повнозв’язну мережу, нормалізацію та залишкові з’єднання. Кожен новий шар аналізує вхідні дані на іншому рівні абстракції, що дозволяє поступово накопичувати все глибше розуміння вхідної інформації. Така багатошарова структура суттєво розширює здатність моделі до узагальнення та покращує її передбачувальні можливості.

Декодер у моделі трансформера виконує завдання генерації текстових послідовностей. Його архітектура багато в чому подібна до кодера: кожен шар містить два підмодулі багатоголової уваги, повнозв’язну нейронну мережу, а також застосовує залишкові з’єднання та нормалізацію після кожного підмодуля.

Декодер працює в авторегресивному режимі, тобто генерує токени послідовно, один за одним. Спочатку процес ініціюється спеціальним токеном початку послідовності (start token). Потім, при кожному кроці, декодер використовує попередньо згенеровані токени як вхід, разом із виходами кодера, що містять інформацію про вхідну послідовність.

На початку роботи декодера, аналогічно до кодера, вхідні токени проходять через шар вбудовування, де кожен токен перетворюється у вектор фіксованої розмірності.

Після вбудовування токени доповнюються позиційною інформацією, що дозволяє моделі враховувати порядок токенів у послідовності. Ці позиційні вектори додаються до ембедингів і подаються на вхід першого шару уваги.

Звичайний декодер складається з певної кількості шарів, перший з яких – маскована самоувага. Цей модуль аналогічний до механізму самоуваги у кодері, але з додатковим маскуванням, яке забороняє поточному токену «бачити» майбутні токени. Завдяки цьому модель не має доступу до інформації, яка ще не згенерована.

Наступний шар встановлює зв’язок між вхідною та вихідною послідовностями. Тут запити (queries) надходять із попереднього шару декодера, а ключі (keys) та значення (values) – з виходу кодера.

Цей крок дозволяє декодеру фокусуватися на релевантних частинах вхідної послідовності, закодованої раніше.

На наступному кроці, як і в кодері, застосовується позиційно-незалежна повнозв’язна мережа для кожного токена окремо. Вона складається з двох лінійних шарів із активацією ReLU між ними.

Останній етап – лінійна проєкція в простір словника, де кожному токену зі словника відповідає певна ймовірність. Після застосування softmax, цей вектор перетворюється у розподіл імовірностей. Найбільше значення вказує на слово, яке модель вибирає наступним.

Результат обробки передається у вигляді згенерованої послідовності токенів, де кожен новий токен залежить від раніше згенерованих. Цей процес повторюється до появи токена завершення, після чого модель зупиняє генерацію [11].

1.2.3 Використання LLM в агентних системах

Великі мовні моделі стали основою нової хвилі агентних систем, які поєднують автономність, гнучкість та здатність до навчання. Ці системи дозволяють агентам не лише розуміти природну мову, але й планувати, приймати рішення, взаємодіяти з інструментами та адаптуватися до нових умов. Інтеграція LLM у мультиагентні системи (Multi-Agent Systems, MAS) відкриває нові горизонти для автоматизації складних завдань у різних галузях – розробка ПЗ, управління бізнес-процесами, управління складними приладами тощо.

Типова LLM-агентна система складається з кількох ключових компонентів:

– планування та декомпозиція завдань – LLM-агенти здатні розбивати складні цілі на підзадачі та формувати послідовні плани дій;

– пам’ять – використання короткострокової (що знаходиться в контексті) та довгострокової пам’яті (векторні бази даних) дозволяє агентам зберігати контекст та накопичувати знання;

– використання інструментів – агенти можуть викликати зовнішні API, бази даних та інші інструменти для отримання актуальної інформації або виконання специфічних функцій;

– рефлексія та самонавчання – агенти аналізують власні дії, виявляють помилки та вдосконалюють стратегії поведінки.

Ця архітектура дозволяє створювати системи, де агенти можуть ефективно співпрацювати, обмінюватися знаннями та адаптуватися до змін у середовищі [12].

У багатокомпонентних системах кожен агент може мати спеціалізацію, наприклад, один відповідає за збір даних, інший – за аналіз, третій – за генерацію звітів. Такі системи демонструють високу ефективність у вирішенні складних завдань.

Наприклад, у сфері програмної інженерії системи, подібні до ChatDev, розподіляють ролі між агентами: один агент виступає як архітектор, інший – як розробник, третій – як тестувальник. Це дозволяє автоматизувати весь цикл розробки програмного забезпечення.

У наукових дослідженнях LLM-агенти використовуються для автоматизації експериментів, аналізу даних та генерації гіпотез. Наприклад, система ChemCrow об’єднує кілька агентів для планування та виконання хімічних синтезів.

LLM-агенти знаходять застосування в багатьох галузях:

– програмна інженерія – автоматизація розробки, тестування та документування програмного забезпечення;

– наукові дослідження, а саме планування та проведення експериментів, аналіз результатів, генерація гіпотез;

– бізнес-аналітика, що включає збір та аналіз ринкових даних, генерацію звітів та рекомендацій;

– освіта – персоналізоване навчання, оцінка знань, адаптація навчальних програм;

– симуляції, такі як моделювання соціальних процесів, поведінки користувачів, економічних систем [13].

Ці застосування демонструють гнучкість та потужність LLM-агентів у вирішенні складних та різноманітних завдань.

1.2.4 Обмеження та виклики використання LLM

Великі мовні моделі (LLM) стали потужним інструментом у сфері штучного інтелекту, демонструючи вражаючі результати в обробці природної мови. Проте їхнє впровадження супроводжується низкою обмежень та викликів, які потребують ретельного аналізу та вирішення. Основними проблемами використання великих мовних моделей є:

– обмежене оновлення знань – LLM, як правило, мають статичний характер знань, обмежений періодом їхнього навчання. Це означає, що вони не можуть самостійно оновлювати інформацію після завершення навчання, що призводить до застарілих або неточних відповідей у динамічних галузях, таких як медицина чи фінанси. Оновлення знань вимагає повторного навчання моделі, що є ресурсоємним процесом;

– проблеми з точністю та галюцинації, що пов’язано з тим, що LLM можуть генерувати інформацію, яка виглядає достовірною, але насправді є вигаданою або неточною – явище, відоме як «галюцинації». Це особливо небезпечно в контекстах, де точність є критичною, наприклад, у наукових дослідженнях або юридичних консультаціях. Дослідження показують, що такі моделі можуть генерувати вигадані джерела або факти, що підриває довіру до їхніх відповідей;

– відсутність прозорості та пояснюваності, бо LLM зазвичай функціонують як «чорні скриньки», де складно зрозуміти, як саме модель прийшла до певного висновку. Це зменшує довіру до їхніх рішень, особливо в критичних сферах, де необхідно пояснити логіку прийняття рішень, наприклад, у медицині або фінансовому аналізі;

– етичні та правові питання – використання LLM піднімає важливі етичні та правові питання, зокрема щодо конфіденційності даних, авторських прав та відповідальності за прийняті рішення. Моделі можуть ненавмисно відтворювати упередження, присутні в навчальних даних, що може призвести до дискримінаційних результатів. Крім того, існує ризик витоку конфіденційної інформації, якщо модель була навчена на чутливих даних без належного контролю;

– високі обчислювальні витрати – навчання та експлуатація LLM вимагають значних обчислювальних ресурсів, що призводить до високих фінансових витрат та екологічного впливу. Це обмежує доступ до таких технологій для малих та середніх підприємств і піднімає питання щодо сталого розвитку в сфері штучного інтелекту;

– залежність від якості навчальних даних, бо якість та репрезентативність навчальних даних безпосередньо впливають на ефективність LLM. Недостатньо якісні або упереджені дані можуть призвести до неточних або дискримінаційних результатів. Це вимагає ретельного відбору та обробки даних перед навчанням моделей;

– ризики безпеки та зловживання – LLM можуть бути використані для створення шкідливого контенту, такого як фішингові повідомлення або дезінформація. Їхня здатність генерувати переконливий текст робить їх потенційним інструментом для зловмисників. Це підкреслює необхідність впровадження механізмів контролю та фільтрації контенту, що генерується такими моделями;

– обмеження в розумінні контексту, хоча вони здатні генерувати зв’язний текст, LLM не володіють справжнім розумінням контексту або

здоровим глуздом. Їхні відповіді базуються на статистичних закономірностях, а не на глибокому розумінні, що може призвести до логічних помилок або невідповідностей у складних ситуаціях [14].

Хоча великі мовні моделі відкривають нові можливості в обробці природної мови та автоматизації завдань, їхнє використання супроводжується низкою обмежень та викликів. Для ефективного та етичного впровадження LLM необхідно враховувати ці фактори, впроваджувати відповідні механізми контролю та постійно вдосконалювати технології з урахуванням потреб суспільства та бізнесу.

1.3 Аналіз існуючих аналогів

1.3.1 Microsoft Copilot

Microsoft Copilot – це вбудований інтелектуальний помічник, який працює на основі великих мовних моделей OpenAI (GPT-4) та інтегрується безпосередньо в операційну систему. Він доступний через бокову панель Windows 11 та дозволяє користувачам взаємодіяти з системою за допомогою текстових або голосових команд.

Основні можливості Copilot включають:

– керування системними налаштуваннями, наприклад зміна теми оформлення, налаштування параметрів безпеки, пошук файлів та оптимізація продуктивності системи;

– інтеграція з Microsoft 365: у Word, Excel, PowerPoint та Outlook Copilot допомагає в написанні текстів, аналізі даних, створенні презентацій та плануванні зустрічей;

– робота з мультимедійними додатками, такими як Paint та Photos, в яких з’явилися функції на основі ШІ для редагування зображень, зокрема зміна стилю та розмиття фону;

– функція Recall, яка дозволяє шукати інформацію про попередні дії користувача на комп’ютері, зберігаючи дані у зашифрованому вигляді [15].

У порівнянні з іншими інтелектуальними помічниками, такими як Siri від Apple або Google Assistant, Microsoft Copilot вирізняється глибокою інтеграцією в операційну систему та офісні додатки. На відміну від GitHub Copilot, який орієнтований на допомогу розробникам у написанні коду, Microsoft Copilot має ширший спектр застосування, охоплюючи як повсякденні завдання, так і професійні потреби.

1.3.2 Google Gemini

Google Gemini – це нове покоління віртуального асистента, що замінює Google Assistant на більшості пристроїв Android, включаючи смартфони, смарт-годинники (Wear OS 6), автомобільні системи (Android Auto) та телевізори (Google TV). Завдяки використанню великих мовних моделей Gemini забезпечує більш природну та контекстно-залежну взаємодію з користувачем.

Основні можливості Gemini включають:

– мультимодальна взаємодія – Gemini може обробляти та генерувати текст, зображення, аудіо та відео, що дозволяє йому відповідати на запити, які включають різні типи даних;

– інтеграція з додатками Google, що демонструється інтеграцією з Gmail, Google Maps, Google Docs, Google Drive, YouTube та іншими сервісами, що дозволяє користувачам виконувати завдання, такі як пошук інформації, планування подій та керування контентом, безпосередньо через асистента;

– Gemini Live – функція, що дозволяє вести реальні розмови з асистентом, використовуючи голос, камеру або екран. Це особливо корисно для завдань, що вимагають візуального контексту або швидкої взаємодії;

– розширення на різні пристрої, бо Gemini доступний не лише на смартфонах, але й на інших пристроях, забезпечуючи єдиний досвід користувача в різних середовищах [16].

1.3.3 Braina

Braina (скорочення від «Brain Artificial») – це багатофункціональний інтелектуальний персональний асистент для Windows, розроблений компанією Brainasoft. Цей програмний продукт поєднує в собі можливості голосового управління, розпізнавання мовлення, автоматизації завдань та інтеграції з великими мовними моделями (LLM), що робить його потужним інструментом для підвищення продуктивності.

Основні можливості Braina:

– голосове керування – виконання команд за допомогою голосу, включаючи відкриття програм, пошук файлів, керування музикою та інше;

– диктування тексту (Speech-to-Text) – перетворення голосу в текст у будь-якому програмному забезпеченні, підтримка понад 100 мов;

– інтеграція з великими мовними моделями, а саме використання ChatGPT, Claude, Gemini та інших LLM для генерації тексту та взаємодії;

– автоматизація завдань – виконання складних команд, включаючи автоматичне введення тексту, відкриття вебсторінок та керування системними процесами;

– персоналізована пам’ять – збереження контексту розмови для більш природної взаємодії [17].

Braina є потужним інструментом для підвищення продуктивності, що поєднує в собі можливості голосового управління, розпізнавання мовлення, автоматизації та інтеграції з сучасними мовними моделями. Завдяки широкому спектру функцій та підтримці багатьох мов, Braina може бути корисною як для особистого використання, так і для професійної діяльності в різних галузях.

2 ПРОЄКТУВАННЯ СИСТЕМИ-АСИСТЕНТА

2.1 Формулювання вимог до системи

2.1.1 Функціональні вимоги

У цьому розділі будуть приведені функціональні вимоги розроблюваної системи асистента. Вони описують, які дії має виконувати система, які задачі вона може вирішувати, а також взаємодію користувача з інтерфейсом та іншими компонентами ПЗ.

Система являє собою інтелектуальну платформу для виконання та обробки запитів користувача з використанням мультиагентної архітектури, з агентами, що здатні виконувати складні завдання, що написані природною мовою.

Ключовою особливістю системи є інтелектуальна обробка користувацьких запитів. Використання технологій природної мови дозволить системі розуміти контекст та намір користувача, перетворюючи текстові інструкції на послідовність конкретних команд. Таке розбиття повинно реалізовуватись агентною системою, яка дотримується підходу ReAct (reasoning + acting), тобто кожен агент виконує логічний аналіз проміжних результатів і приймає рішення про наступні дії.

Основою розроблюваної системи є взаємодія програмних агентів, що мають виконувати необхідні дії для досягнення поставленої мети та кооперувати один з одним. Кожен агент спеціалізується на певному типі завдань: пошук інформації, керування файлами, взаємодія з операційною системою тощо. Агенти мають можливість обмінюватися повідомленнями, що дозволяє системі створювати складні та багатокрокові сценарії виконання завдань.

Для кожної виявленої підзадачі агент повинен сформувати відповідну команду до командного інтерфейсу Windows (PowerShell), враховуючи

синтаксис і можливі варіанти виконання в залежності від контексту. Сформована команда виконується у вбудованому модулі, який безпосередньо взаємодіє з оболонкою операційної системи.

Одною з основних функціональних вимог є інтерактивність системи. Користувач на будь-якому етапі має мати можливість зупинити виконання запиту, переглядати проміжні результати, мати змогу підтверджувати виконання критичних команд, надавати системі необхідну додаткову інформацію.

Архітектура системи розроблятиметься з урахуванням принципів модульності та гнучкості. Передбачено механізми динамічного підключення нових агентів та плагінів без необхідності внесення суттєвих змін до базової архітектури. Має бути реалізована можливість додавання нових інструментів керування ОС, налаштування нових агентів, зміна великих мовних моделей.

Графічний інтерфейс користувача повинен забезпечувати зручне введення запитів, перегляд ходу виконання задачі та результатів команд, журнал подій, а також надавати елементи керування, що дозволяють зупиняти, повторно виконувати або скасовувати певні дії.

Система веде повний облік усіх системних подій, забезпечуючи прозорість та можливість подальшого аналізу. Журнали містять детальну інформацію про виконані команди, задіяних агентів, часові мітки та статуси виконання. Вбудовані засоби аналітики дозволять генерувати звіти про системну активність, виявляти потенційні проблеми та оптимізувати роботу.

Таким чином, розроблювальна система асистента є комплексною інтелектуальною платформою, що поєднує мультиагентну архітектуру, обробку природної мови та гнучку взаємодію з операційною системою. Завдяки модульності, інтерактивності та прозорості, система забезпечує ефективне виконання складних завдань, адаптивність до нових вимог і зручність для користувача. Ці особливості дозволяють їй бути потужним

інструментом для автоматизації та оптимізації різноманітних процесів, відкриваючи широкі можливості для подальшого розвитку та масштабування.

2.1.2 Нефункціональні вимоги

Цей розділ визначає нефункціональні вимоги проєктованої мультиагентної системи. Вони не описують конкретні дії, які має виконувати система, але встановлюють обмеження, якість, продуктивність та інші характеристики, що мають вплив на зручність використання, стабільність, безпечність та масштабованість розробленого програмного забезпечення.

Одним із ключових нефункціональних вимог є надійність системи. Усі компоненти повинні функціонувати узгоджено, навіть у разі виникнення виняткових ситуацій, таких як помилкові або небезпечні команди, недоступність моделі чи мережеві проблеми.

Також особливу увага буде приділятися питанням безпеки створеної системи. Так як асистент буде виконувати усі дії на справжньому комп’ютері користувача (без використання віртуалізації), система має мати високий рівень безпеки та надійності. Усі дії, що асистент може виконувати з операційною системою користувача мають бути безпечні, а при необхідності виконання потенційно загрозливих дій мультиагентна система має отримувати дозвіл від користувача.

Система повинна бути також зручною у використанні – графічний інтерфейс має бути зрозумілим для користувача з мінімальними навичками використання персонального комп’ютера. Інтерфейс має включати поля для введення текстових запитів, зони для виводу результатів виконання, журнал подій, а також кнопки керування. Необхідно дотримуватись принципів простоти, логічної структури, достатньої контрастності та наявності підказок при введенні запитів.

Ще одним аспектом є портативність і вимоги до середовища запуску. Система має працювати в середовищі Windows 10 або вище, підтримувати запуск із використанням Python 3.12+ та сумісних бібліотек. Якщо використовується локальна LLM, її інтеграція має бути легкою, а кількість ресурсів, необхідних для її роботи – мінімізованою.

В роботі над системою-асистентом буде важливим використовувати лише програмні продукти (мови програмування, бібліотеки, API, LLM), які мають відкритий програмний код та не мають обмежень на використання, модифікацію та поширення системи в навчальних і дослідницьких цілях.

Таким чином, нефункціональні вимоги формують важливу частину проєктних обмежень і цілей. Вони визначають якість розробки, майбутню підтримку, безпечність використання та комфорт кінцевого користувача при взаємодії з інтелектуальним асистентом.

2.2 Особливості архітектури системи

2.2.1 Загальний огляд архітектури

Проєктована система-асистент для керування операційною системою призначена для взаємодії користувача з операційною системою Windows за допомогою запитів природною мовою. В основі запропонованої системи знаходиться багаторівнева та модульна архітектура, що забезпечує чітке розділення логіки програми, а саме обробку запитів користувача, координацію програмних агентів та виконання задач на рівні операційної системи.

Архітектуру системи буде реалізована у вигляді чотиришарової структури з наступними компонентами:

– рівень координації агентів;

– рівень обробки запитів великою мовною моделлю;

– рівень виконання системних дій;

– рівень взаємодії користувача.

Архітектура є повністю модульна та дозволяє легко вносити зміни в окремі компоненти без необхідності переробляти всю систему.

Перший запропонований рівень – це рівень координації агентів, який є основним модулем інтелектуальної системи. Координація агентів є невід'ємною задачею мультиагентної системи, задля забезпечення ефективної кооперації модулів. Кожен модуль має чіткі задачі, має доступ до обмеженої кількості інструментів та API та має окремий канал обміну повідомленнями для взаємодії.

Цей рівень має чітку ієрархічну структуру агентів, в голові якої знаходиться основний агент – оркестратор. Він виконує наступні задачі:

– отримання запитів користувача;

– передача запитів у модуль LLM для аналізу;

– інтеракція з користувачем – запит додаткової інформації, демонстрація проміжних результатів;

– приймання рішення щодо виконання дії – самостійно чи делегація її спеціалізованим агентам;

– збирання відповідей від агентів і формування результату для відображення користувачеві.

Наступним рівнем є рівень обробки запитів за допомогою LLM. Він організує просту та ефективну взаємодію між агентами та мовною моделлю. Він підтримує зміну моделі (використання локальних та хмарових моделей), зміну гіперпараметрів моделі, редагування підказок (prompts) для інструментів та агентів. Саме цей рівень додає інструментальні компоненти системи, такі як інтелектуальний пошук, генерація команд ОС.

Інтеграція LLM здійснюється через API (наприклад, OpenAI API) або локально через Ollama. Система обробки промптів відповідає за створення правильно структурованих запитів до LLM з урахуванням правил безпеки, обмежень і прикладів (few-shot prompting).

Рівень виконання системних дій – є найнижчим у системі, і саме він безпосередньо відповідає за реалізацію дій у Windows. Тут розташовані компоненти, що взаємодіють із середовищем ОС. Він відповідає за виконання команд через Powershell (модуль subprocess в мові Python), взаємодію з файловою системою, управління застосунками, управління налаштуваннями операційної системи та інше. Він також використовує Windows API для доступу до низькорівневих дій з ОС, наприклад зміна реєстру.

Важливою задачею цього модулю є також запобігання виконання небезпечних дій, таких як видалення системних файлів, операції з накопичувачами, виконання ризикованих інтернет-запитів тощо. На цьому рівні до всіх команд, що були надіслані агентною системою, будуть застосовані фільтри.

Останнім рівнем є рівень користувацького інтерфейсу. Цей рівень представлений графічним інтерфейсом користувача (GUI), який забезпечує зручне введення текстових запитів природною мовою та виведення відповідей системи. GUI працює як фронтенд системи й передає введені запити у вигляді структурованих повідомлень до головного рівня інтелектуальної системи для подальшої обробки.

2.2.2 Взаємодія між модулями

Система-асистент для управління Windows реалізована як багаторівнева агентна система, в якій ключові компоненти – це агент-окрекстратор, функціональні агенти, інструменти, графічний інтерфейс користувача, та модуль виконання дій у Windows. Усі вони взаємодіють між собою згідно розподілу обов’язків, а координація цієї взаємодії здійснюється центральним ReAct-агентом.

Користувацький запит надходить до системи через графічний інтерфейс, який виконує функцію точки входу в інтелектуальну систему.

Він дозволяє користувачу вводити запити природною мовою, переглядати результати та кроки вирішення задачі. Інтерфейс безпосередньо передає запит агенту, який ініціює процес міркування та аналізу.

Головний агент-оркестратор розбиває вхідний запит на послідовність кроків, де кожен крок може включати обчислення, перевірку стану системи або виклик певного інструмента. Замість виконання дій безпосередньо, агент використовує абстракцію у вигляді інструментів, так званих tools – функцій або об’єктів, через які він може безпечно і контрольовано звертатися до зовнішніх дій. Ця програмна абстракція нативно підтримується більшістю сучасних великих мовних моделей. Ці інструменти, своєю чергою, обгортають доступ до функціональних агентів, що спеціалізуються на певних категоріях задач – наприклад, файлових операціях, створення Powershell-команд, пошук в інтернеті тощо.

Коли агент вирішує, яку саме дію потрібно виконати, він генерує відповідну інструкцію у форматі виклику інструмента з параметрами. Наприклад, у випадку переміщення файлів, агент викликає інструмент, що передає команду файловому агенту на виконання переміщення із вказаним шляхом. Агент ніколи не має прямого доступу до системних ресурсів, що дозволяє зберігати безпечну ізоляцію логіки прийняття рішень від дій нижчого рівня.

Функціональні агенти виконують конкретні дії. Вони взаємодіють з операційною системою за допомогою subprocess – модуля Python, що дозволяє виконувати Powershell-команди, або використовують більш складні механізми, зокрема WinAPI чи інші стороні утиліти, якщо це необхідно. Усі дії виконуються лише за запитом від інструменту, і результат виконання передається назад через інструмент, а потім потрапляє в агент.

Агенти обробляють отримані результати, порівнюють їх із метою, і на основі цього приймають рішення про наступний крок. Після завершення всіх необхідних дій, сформована відповідь повертається чи до візуального

інтерфейсу, якщо це відповідь оркестратора, чи до агенту, який зробив виклик. Усі проміжні результати автоматично демонструються в інтерфейсі.

Варто зазначити важливість контексту в роботі агента. Він зберігає історію діалогу та внутрішні стани агента, що дозволяє реалізовувати багатокрокові або уточнюючі взаємодії. Наприклад, після завершення операції користувач може дати нову інструкцію, яка посилається на попередню і агент зможе коректно її інтерпретувати.

Така організація взаємодії дозволяє будувати модульну, розширювану архітектуру, де компоненти мають чітке розмежування обов’язків. Всі дії контролюються агентом, який приймає рішення на основі доступної інформації та наявних інструментів.

2.3 Технологічний стек та інструменти розробки

2.3.1 Взаємодія з операційною системою

Однією з ключових функцій системи-асистента є можливість взаємодії з операційною системою Windows, оскільки агент повинен не лише відповідати на запити користувача, але й виконувати реальні дії на рівні системи. Це потребує використання ряду бібліотек та модулів, які забезпечують доступ до функціоналу ОС, такого як запуск процесів, робота з вікнами, управління файлами, емуляція натискань клавіш тощо.

У розробці використовуються наступні компоненти:

– os та subprocess – ці стандартні бібліотеки Python застосовуються для виконання команд Powershell, запуску зовнішніх процесів, навігації по файловій системі та отримання інформації про середовище;

– ctypes та pywin32 – використовуються для низькорівневої взаємодії з Windows API та дозволяють викликати функції системних бібліотек Windows напряму.

2.3.2 Платформа локальних великих мовних моделей

В роботі було приділено велику увагу саме використанню відкритих та доступних великих мовних моделей. Користувач системи-асистента має мати можливість легко встановлювати та змінювати різні LLM.

З цією метою для кваліфікаційної роботи було обрано платформу Ollama – зручний інструмент для локального запуску та керування великими мовними моделями.

Ollama – це відкрита платформа, розроблена для запуску LLM локально на пристроях користувача, таких як MacOS, Windows і Linux. Вона забезпечує сумісність між платформами, що важливо для широкого застосування.

Вона автоматично виявляє доступне апаратне забезпечення, підтримуючи як GPU (NVIDIA/AMD), так і CPU. Це забезпечує гнучкість, дозволяючи запускати моделі навіть на стандартному обладнанні, що є важливим для систем, де ресурси можуть бути обмеженими.

Управління моделями в Ollama реалізовано через Modelfile – файли, які інкапсулюють усі необхідні компоненти для запуску LLM, подібно до контейнерів Docker. Це спрощує розповсюдження та виконання моделей, що корисно для інтеграції в асистента.

Крім того, Ollama підтримує квантування – техніку, яка знижує обчислювальні вимоги, дозволяючи ефективно запускати моделі на споживчому обладнанні, такі як стандартні ноутбуки чи десктопи. Це є важливим, оскільки не всі користувачі асистента матимуть доступ до високопродуктивного обладнання. Наприклад, для запуску моделей розміром 7B потрібно щонайменше 8 ГБ RAM, для 13B – 16 ГБ, а для 33B – 32 ГБ, як зазначено в офіційній документації. [18]

2.3.3 Робота з LLM та агентами

У контексті розробки інтелектуального асистента на основі великих мовних моделей, фреймворк LlamaIndex відіграє ключову роль у реалізації агентної архітектури. Цей фреймворк забезпечує ефективне поєднання LLM з зовнішніми джерелами даних, дозволяючи створювати агентів, здатних до автономного прийняття рішень та виконання складних завдань.

Фреймворк підтримує інтеграцію з різноманітними інструментами, такими як вебпошук, аналіз даних та доступ до зовнішніх API. Це дозволяє агентам виконувати широкий спектр завдань, від простого пошуку інформації до складних аналітичних операцій. Це забезпечує гнучкість у роботі з різними джерелами інформації та дозволяє масштабувати рішення відповідно до потреб користувача [19].

LlamaIndex був використаний для створення агентів, які можуть автономно обробляти запити користувача, інтегруватися з зовнішніми джерелами даних для отримання актуальної інформації та забезпечувати гнучку та масштабовану архітектуру асистента. Це дозволило реалізувати інтелектуального асистента, здатного ефективно взаємодіяти з користувачем та надавати релевантні відповіді на запити.

2.3.4 Графічний інтерфейс користувача

Однією з важливих складових системи-асистента є графічний інтерфейс користувача, що забезпечує зручну, інтуїтивно зрозумілу та ефективну взаємодію між користувачем і агентною частиною системи. З метою розробки такого інтерфейсу було обрано фреймворк Electron.js, який дозволяє створювати кросплатформені настільні додатки з використанням вебтехнологій – HTML, CSS та JavaScript, та при цьому отримати повноцінний десктопний застосунок з доступом до системних ресурсів.

Electron поєднує у собі браузерний рушій Chromium для візуалізації інтерфейсу та Node.js, що відкриває широкі можливості для взаємодії з операційною системою [20]. Завдяки цьому, розроблений інтерфейс може не лише приймати запити від користувача у зручному форматі, але й відображати процес їх обробки, результати, перегляд журналів, історії взаємодій тощо. Це особливо важливо в контексті системи, яка працює з LLM та агентами, де результат взаємодії часто залежить від складної послідовності кроків, що важливо зробити прозорими та зрозумілими для користувача.

Ключовим аспектом архітектури стало розділення інтерфейсної та логічної частини системи. Ядро агентної системи працює окремо – в іншому процесі на комп’ютері користувача. Для зв’язку між інтерфейсом (клієнтом) і ядром (сервером) використовується WebSocket. На відміну від HTTP, він забезпечує постійне двостороннє з’єднання, що дозволяє обмінюватися повідомленнями в реальному часі без повторного встановлення з’єднання.

Це дає змогу інтерактивно працювати з агентами: користувач вводить запит – інтерфейс передає його системі – агенти виконують завдання, надсилаючи результати назад – інтерфейс у реальному часі відображає хід виконання й відповіді, дозволяючи, за потреби, втручання користувача. Така архітектура забезпечує високу швидкодію, низькі затримки та масштабованість, зокрема розгортання ядра на віддалених серверах або у хмарі.

Завдяки Electron і WebSocket створюється гнучка, надійна та масштабована система візуальної взаємодії, що забезпечує зручний доступ до функцій агентної системи й сприяє їх розумінню та контролю. Це особливо важливо у системах, де рішення приймаються на основі динамічного аналізу великих обсягів даних мовними моделями.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Загальна структура програмної реалізації

Розроблена система складається з двох основних компонентів: серверної частини (бекенду) та клієнтської частини (фронтенду), які взаємодіють між собою через протокол WebSocket. Така архітектура забезпечує гнучкість, масштабованість та можливість ефективної асинхронної взаємодії між компонентами.

Бекенд реалізовано як застосунок на основі фреймворку FastAPI, що дозволяє створювати високопродуктивні асинхронні вебсервіси. Основним завданням бекенду є обробка запитів від користувача, управління агентами, взаємодія з моделями штучного інтелекту, а також виконання системних команд на локальній машині. Комунікація з клієнтською частиною здійснюється через WebSocket-з’єднання, що забезпечує двосторонню передачу даних у режимі реального часу.

Для організації логіки агентів та взаємодії з великими мовними моделями використовується фреймворк LlamaIndex. Він забезпечує зручний інтерфейс для створення агентно-орієнтованих систем, дозволяє інтегрувати LLM-агентів, координувати їхню роботу, а також зберігати та обробляти контекст попередніх запитів. Оркестрація агентів здійснюється відповідно до обраної стратегії керування, з можливістю розширення функціональності системи за допомогою додаткових модулів.

Фронтенд реалізовано за допомогою Electron JS – платформи для створення кросплатформених настільних додатків з використанням вебтехнологій. Вона забезпечує графічний інтерфейс користувача (GUI), через який відбувається введення запитів, перегляд результатів роботи агентів, а також взаємодія з елементами керування системою. Electron-застосунок підтримує обмін повідомленнями з серверною частиною через

WebSocket, що дозволяє передавати як текстові команди, так і структуровані повідомлення з результатами виконання.

Така структура дозволяє ефективно розділити обов’язки між клієнтською та серверною частинами, а також забезпечити зручність у подальшій розробці, тестуванні та масштабуванні системи. Компоненти побудовані таким чином, щоб за потреби можна було замінити окремі частини (наприклад, інтерфейс або механізм обробки команд) без порушення загальної логіки взаємодії.

3.2 Реалізація агентної структури

3.2.1 Загальна архітектура агентної системи

У межах розробленої системи реалізовано агентну архітектуру, що дозволяє організувати виконання складних завдань шляхом взаємодії між окремими спеціалізованими агентами. Такий підхід забезпечує модульність, масштабованість та спрощує додавання нових функцій до системи.

У якості основи для побудови агентної системи використовується компонент AgentWorkflow із бібліотеки LlamaIndex, який дозволяє визначати складні послідовності дій для агентів, координувати їхню взаємодію та забезпечувати обробку запитів користувача на високому рівні абстракції. Програмний код ініціалізації AgentWorkflow наведено у додатку А, лістинг А.1.

Центральним елементом агентної структури є головний агент – AgentOrchestrator, реалізований на основі архітектури ReAct. Цей агент поєднує здатність до логічного міркування та виконання дій, аналізує запит користувача, планує виконання завдання та делегує задачі відповідним функціональним агентам. ReAct-архітектура дозволяє агенту спочатку розмірковувати над кроками, які необхідно виконати, а потім послідовно їх

реалізовувати. Код для створення цього агенту демонструється у додатку А, лістинг А.2.

3.2.2 Опис функціональних агентів системи

Система включає набір функціональних агентів, кожен з яких відповідає за певний клас задач. Комунікація між агентами відбувається через узгоджений інтерфейс, що забезпечує гнучкість та масштабованість архітектури.

Особливістю системи є те, що вона дозволяє швидко додавати нових агентів завдяки великому рівню абстракції цих модулей.

3.2.2.1 Агент для роботи з файловою системою

FileAgent – це спеціалізований агент, який відповідає за виконання будь-яких файлових операцій у системі користувача. Він має набір інструментів (функцій), які дозволяють створювати, читати, редагувати, переміщувати, видаляти файли, перевіряти структуру директорій, здійснювати пошук за шаблоном та перейменовувати файли.

Функції-інструменти агента:

– `copy_file(origin, destination)` – копіює файл з однієї директорії в іншу;

– `create_file(file_path)` – створює порожній файл за вказаним шляхом;

– `delete_file(file_path)` – видаляє файл, використовуючи send2trash (переміщення в кошик);

– `write_file(content, file_path)` – записує текстовий вміст у файл (перезаписує вміст повністю);

– `read_file(file_path)` – зчитує та повертає вміст файлу;

– `dir_path(path)` – виводить структуру директорії до 3 рівнів глибини;

– `search(full_pattern, recursive=False)` – пошук файлів за шаблоном (glob), з можливістю рекурсії;

– `rename(old_path, new_path)` – перейменовує файл або переміщує його, якщо новий шлях інший.

Після завершення виконання будь-якої функції FileAgent повертає результат і передає керування назад до AgentOrchestrator, який координує подальші дії або запитує додаткову інформацію. Програмний код цього модулю наведено в додатку А, лістинг А.3

3.2.2.2 Агент для контролю медіа

MediaControlAgent – це спеціалізований агент, який відповідає за управління мультимедійним відтворенням та системною гучністю на рівні ОС Windows. Завдяки використанню бібліотек keyboard та pycaw, агент може імітувати апаратні мультимедійні кнопки та отримувати/встановлювати рівень гучності системи. Програмний код цього агенту наведено в додатку А, лістинг А.4.

Функції-інструменти агента:

– `play()` – надсилає системний сигнал «Play/Pause» для запуска відтворення;

– `pause()` – надсилає той самий сигнал «Play/Pause», що і `play()`, для зупинки медіа;

– `next_track()` – перемикає на наступний трек у програвачі;

– `previous_track()` – перемикає на попередній трек;

– `set_volume_level(volume_level: int)` – встановлює системну гучність у діапазоні 0–100;

– `get_volume_level()` – зчитує поточний рівень системної гучності та повертає його у відсотках.

Після кожної дії агент завжди завершує роботу передачею керування до AgentOrchestrator.

3.2.2.3 Агент роботи з Powershell

PowershellAgent – це універсальний агент для взаємодії з операційною системою Windows через PowerShell-команди. Він може як генерувати команди для виконання запитів користувача, так і безпосередньо виконувати їх. Агент особливо корисний у випадках, коли інші агенти не мають доступу до низькорівневих або системних функцій ОС.

Функції-інструменти агента:

– `generate_command(query: str)` – генерує за допомоги LLM список PowerShell-команд, які виконують поставлене завдання;

– `execute_command(command: str)` – виконує одну PowerShell-команду, яка вже має правильний синтаксис.

Завжди завершує виконання передачею контролю назад до AgentOrchestrator та надає чітку причину, яка залежить від результату виконання дій. Код цього модулю продемонстровано в додатку А, лістинг А.5.

3.2.2.4 Агент запуску встановлених програм

RunAppAgent – це спеціалізований агент, який використовується для запуску встановлених програм (браузерів, текстових редакторів, ігор тощо) за їхньою назвою. Він не виконує скрипти або системні команди, тому для таких задач рекомендується передати управління PowershellAgent.

Система створює локальний індекс встановлених програм, та зберігає його до .csv файлу, який надалі використовує агент для пошуку необхідної встановленої програми.

Має лише одну функцію-інструмент, а саме `run_app(app_name: str)`, який запускає застосунок за назвою (навіть частковою). Програмний код цього модулю наведено в додатку А, лістинг А.6.

3.2.2.5 Агент вебпошуку

WebSearchAgent відповідає за вебпошук у реальному часі. Він використовує API сервісу Tavily для швидкого отримання релевантної інформації з інтернету за коротким запитом користувача чи іншого агента. Програмний код цього агенту наведено в додатку А, лістинг А.7.

Має також лише одну функцію – `search_web(query: str)`, який виконує пошук за допомогою Tavily API.

3.2.3 Взаємодія агентів у системі

AgentOrchestrator взаємодіє з цими агентами через узгоджений інтерфейс, передаючи їм конкретні підзавдання та збираючи результати їх виконання. Така модель дозволяє реалізувати гнучку систему делегування повноважень, де головний агент виступає як координатор, а функціональні агенти відповідають за реалізацію конкретних дій.

Взаємодія між агентами відбувається через загальний контекст виконання, керований LlamaIndex. Фреймворк в автоматичному режимі розпізнає функціональні виклики, що допомагає інтерактивно взаємодіяти із зовнішніми інструментами та функціями, отримуючи доступ до даних, виконуючи дії та структуруючи виконання завдань у кілька кроків.

Функціональні виклики (function calling) у LlamaIndex – це механізм, що дає змогу агенту викликати заздалегідь визначені функції в процесі

обробки запиту. Ці функції можуть бути пов’язані з доступом до баз даних, виконанням пошукових запитів, надсиланням команд операційній системі, генерацією коду тощо. Кожен виклик має свої параметри виклику, наприклад, виклик копіювання файлів у системі має параметри шляху оригіналу та майбутнього шляху файлу.

Функції вбудовуються в LLM-агента через механізм FunctionTool в LlamaIndex. Агент будується на основі цих абстракцій функціональних інструментів, де кожна дія оформляється як окремий виклик функції, з параметрами, автоматично підібраними моделлю. Приклад функціонального інструменту, а саме функцію читання тексту з файлів, представлено у формі асинхронної Python-функції наведено в лістингу 3.1.

Лістинг 3.1 – Програмний код функції-інструмента читання файлів агентом FileAgent

```
async def read_file(ctx: Context, file_path:str) -> str:
    """Useful for reading content of a file using its
path. Your input must contain a full path of a file."""
    # Read file
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            return f'Contents: \n{content}'
    except FileNotFoundError:
        return "File not found."
    except PermissionError:
        return "Permission denied."
```

При ініціалізації будь-якого агента в LlamaIndex та передачі посилань на функції-інструменти в конструкторі об’єкта агенту автоматично створює об’єкт типу FunctionTool, що є зручним методом передачі необхідної інформації про доступні інструменти до запиту агента.

Основним функціональним викликом системи є виклик «handoff» – виклик для делегації дії до іншого агента. Кожен агент у системі має можливість передачі контролю іншим агентам. Головний агент, а саме AgentOrchestrator, має можливість виклику будь-якого агенту, а функціональні агенти по завершенню виконання своїх задач використовують «handoff» для повернення контролю агенту управління.

Функціональний виклик «handoff» має наступні параметри: «to_agent» – визначає агента, кому потрібно передати контроль, та «reason» – причина делегації задачі, наприклад – задача може бути виконана певним агентом. Система автоматично логує усі функціональні виклики агентів. Приклад логу функціонального виклику «handoff» можна побачити на рисунку 3.1.

```
==================================================
🤖 Agent: AgentOrchestrator
==================================================

📤 Output: Thought: The current language of the user is: English. I need to use a
 tool to help me answer the question.
Action: handoff
Action Input: {"to_agent": "PowershellAgent", "reason": "Executing command to che
ck IP address"}
🛠️  Planning to use tools: ['handoff']
🔨 Calling Tool: handoff
  With arguments: {'to_agent': 'PowershellAgent', 'reason': 'Executing command to
 check IP address'}
🔧 Tool Result (handoff):
  Arguments: {'to_agent': 'PowershellAgent', 'reason': 'Executing command to chec
k IP address'}
```

Рисунок 3.1 – Логи взаємодії агентів у процесі обробки запиту користувача (передача керування агенту PowershellAgent)

Таким чином, система має зручний та надійний інструмент взаємодії та делегації обов’язків. Для наочності можна навести наступну діаграму взаємодії агентів (рисунок 3.2).

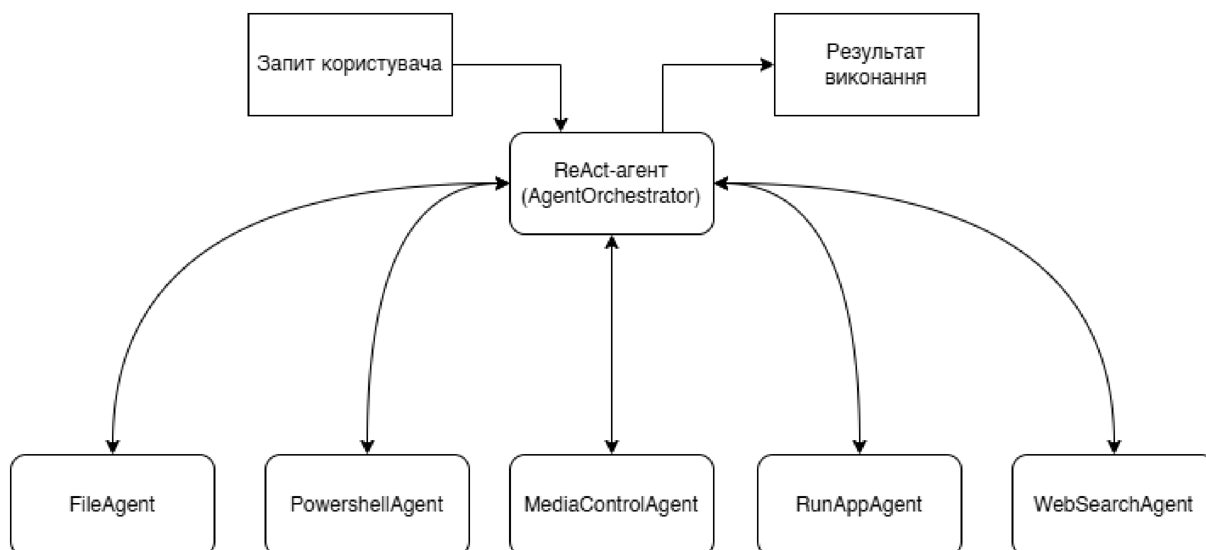

Рисунок 3.2 – Діаграма взаємодії агентів за допомогою handoff-викликів

Завдяки цьому підходу забезпечується висока адаптивність системи – її поведінка може змінюватися відповідно до контексту, а логіка розширюється без значної модифікації наявного коду. Усі агенти працюють у межах одного цілісного робочого процесу, підтримуючи як крокову взаємодію, так і контекстне міркування на основі попередніх запитів користувача.

3.3 Генерація few-shot прикладів для покращення роботи агентів

У багатьох системах на основі великих мовних моделей (LLM) однією з найефективніших стратегій є використання контекстного навчання (prompt-based learning), зокрема – few-shot prompting. Суть цього підходу полягає в тому, що моделі надаються кілька прикладів виконання завдання у вигляді пари «вхід – очікувана відповідь», після чого їй ставиться нове аналогічне запитання. Така стратегія дозволяє значно підвищити точність відповідей без необхідності донавчання самої моделі.

У даній роботі було запропоновано і реалізовано окремий модуль, що автоматично генерує релевантні few-shot приклади для використання з

локальними мовними моделями, які взаємодіють у складі мультиагентної системи асистента. Генерація прикладів відбувається з використанням потужнішої зовнішньої моделі (наприклад, GPT-4o), що виконує роль «вчителя» для локальних агентів.

Локальні LLM, які працюють на пристрої користувача (наприклад, як Qwen 2.5 14B, яка використовується в даній роботі), мають обмежену кількість параметрів, обсяг контекстного вікна та продуктивність порівняно з великими хмарними моделями. Як наслідок, у них часто спостерігається нижча точність, особливо у випадках генерації багатокрокових команд чи аналізу неформалізованих запитів.

Використання few-shot прикладів дозволяє підвищити точність локальних агентів без необхідності їх повного переобучення. Проте формування релевантних прикладів вручну – ресурсозатратна задача. Саме тому виникає ідея створити допоміжну систему, яка автоматично генерує приклади виконання подібних завдань, зберігає їх, і динамічно використовує у запитах до локальних агентів.

У якості моделі-вчителя було обрано модель GPT-4o від OpenAI, доступ до генерації тексту якої було отримано через відповідний API. У запиті до GPT-4o моделі була надана повна інформація щодо архітектури системи – перелічені усі доступні агенти, їх функції. Моделі було підказано згенерувати по 30 типових запитів для кожного агента та вивести їх у строгому форматі для полегшення подальшого використання. Повний текст запиту до моделі можна побачити у додатку Б, Б.1.

У результаті виконання генерації великою моделлю було отримано багато різноманітних пар запитів та кроків розв’язання, які було записані у JSON-форматі. Приклади таких пар у цьому форматі наведено в лістингу 3.2.

Лістинг 3.2 – Приклади few-shot пар у JSON-форматі

```
 {"query": "Delete temporary files in C:\\Temp folder.",
  "steps": [
    "FileAgent -> search('C:\\Temp\\*', recursive=False)",
    "FileAgent -> delete_file('<full path of temporary
file>')"
  ]},
{ "query": "Find and rename all .jpg files in E:\\Images to
.jpeg.",
  "steps": [
"FileAgent -> search('E:\\Images\\*.jpg', recursive=False)",
    "FileAgent -> rename('E:\\Images\\<found file>',
'E:\\Images\\<new name>.jpeg')"]},
```

Збережені таким чином приклади використовуються для автоматичного формування запитів (prompts), які передаються локальній моделі, що виконує роль агента-оркестратора. Для вибору найбільш релевантних прикладів до кожного нового запиту застосовується векторний пошук.

Кожен збережений запит із JSON-структури попередньо перетворюється на вектор за допомогою моделі генерації векторних представлень (embeddings), а саме mxbai-embed-large, яка також доступна в Ollama. Таким чином це дозволяє ефективно знаходити семантично подібні запити. При надходженні нового запиту від користувача, система також перетворює його у вектор та обчислює косинусну схожість (cosine similarity) між цим вектором та всіма запитами, збереженими у базі few-shot прикладів.

На основі цієї схожості обираються найближчі N прикладів (у даному випадку 3), які вважаються найбільш релевантними до нового запиту. Вони автоматично додаються до запиту агента-оркестратора як контекст, після чого модель намагається згенерувати відповідь, у стилі великої LLM. Таким чином, кожен новий запит отримує підказки у вигляді релевантних

розв’язків, які допомагають агенту сформулювати точні та послідовні дії для виконання завдання.

Підхід до генерації few-shot прикладів за допомогою зовнішньої великої мовної моделі має низку важливих переваг. Насамперед, він дозволяє суттєво підвищити якість відповідей локальної моделі. Надані приклади допомагають моделі краще зрозуміти контекст і структуру завдання, навіть якщо сама модель має обмежену кількість параметрів або слабку здатність до узагальнення. Крім того, такий підхід добре масштабується – чим більше прикладів генерує модель-«вчитель», тим точніше локальні агенти зможуть виконувати подібні завдання у майбутньому. Це особливо корисно для розширення функціональності асистента без необхідності змін у його внутрішній архітектурі.

Ще однією важливою перевагою є економія обчислювальних ресурсів. Замість повного донавчання локальної моделі під нові задачі, достатньо сформувати відповідні приклади – і модель вже демонструє покращену поведінку. Це робить систему придатною для використання навіть на малопотужних пристроях. Сама система генерації є гнучкою: вона легко адаптується до нових типів запитів і задач користувача, адже процес генерації прикладів є динамічним. Окрім цього, приклади, що додаються до запитів, роблять поведінку агента більш прозорою: користувач або розробник може зрозуміти, на основі яких правил чи кроків було прийнято певне рішення, що підвищує пояснюваність системи в цілому.

3.4 Приклади використання системи

Розроблена система є потужним програмним забезпеченням, здатним допомагати користувачу у виконанні різноманітних дій. В цьому розділі будуть наведені типові сценарії використання, які ілюструють можливості агентної архітектури, координацію між агентами, а також здатність до

логічного міркування і розв'язання завдань на основі природномовного запиту користувача.

3.4.1 Приклад 1 – Запис інформації в новий файл

У цьому прикладі розглянуті типові задачі системи, що пов'язані зі створенням файлів та запису їх контенту.

Типовим запитом користувача є «Створи новий файл у папці Документи і запиши туди мої справи на сьогодні: прибирання, програмування та заняття спортом»

Система отримає запит користувача, та почне виконання.

Хід виконання:

1) AgentOrchestrator розбиває запит на підзавдання: створення файлу та запис тексту;

2) оркестратор викликає FileAgent із функцією `create_file(path)`, вказуючи шлях до нового файлу;

3) після підтвердження створення файлу, запускається `write_file(content, path)` для запису списку справ;

4) після успішного виконання дій FileAgent передає контроль назад головному агенту, який підтверджує завершення сценарію;

5) головний агент генерує звіт з виконання та демонструє його користувачеві.

Таким чином, цей приклад запиту демонструє можливості системи працювати з файловою системою Windows.

3.4.2 Приклад 2 – керування мультимедіа

Цей приклад демонструє виконання задачі, що потребує взаємодії асистента з мультимедійною системою операційної системи.

Запит користувача – «Зроби гучність 30% і увімкни наступний трек».

Після введення запиту, система почне виконання наступним чином:

1) AgentOrchestrator визначає два підзавдання – змінити гучність та перемкнути трек;

2) оркестратор викликає MediaControlAgent, який виконує `set_volume_level(30)`;

3) після цього виконується `next_track()`, що імітує натискання відповідної мультимедійної кнопки;

4) AgentOrchestrator отримує підтвердження і завершує сценарій генеруванням звіту.

3.4.3 Приклад 3 – виконання системної команди через PowerShell

Наступний приклад стосується прикладу виконання найскладнішого типу завдань – таких, що потребують виконання команд системи використовуючи Powershell.

Особливістю виконання саме таких завдань є те, що вони виконуються у два етапи – генерація команди та її виконання. Система динамічно реагує на системні виводи та помилки, при необхідності змінює підхід виконання задачі.

Запит користувача – «Встанови Google Chrome та перевір чи він працює».

Запит потребує від системи виконання наступних дій:

1) AgentOrchestrator інтерпретує запит як потребу встановити певну програму та передає контроль PowershellAgent;

2) PowershellAgent виконує функцію `generate_command("Install Google Chrome")`;

3) команда генерується за допомогою LLM:

```
winget install --id=Google.Chrome -e –silent
```

4) команда передається у `execute_command(...)`, і PowershellAgent запускає процес встановлення;

5) агент переконується в коректності встановлення програми за допомогою перевірки версії програми за допомогою команди:

```
(Get-Item "C:\Program Files\Google\Chrome\Application\chrome.exe").VersionInfo.ProductVersion
```

6) PowershellAgent повертає управління оркестратору та надає інформацію про результат встановлення програми.

3.4.4 Приклад 4 – запуск встановленої програми

Цей приклад задачі є одним з найпростіших, тому що агент має лише знати назву застосунку для запуску.

Типовий запит для цього типу завдань є – «Запусти Google Chrome.».

Хід виконання:

1) AgentOrchestrator розпізнає запит на запуск програми та передає управління агенту RunAppAgent;

2) RunAppAgent викликає функцію `run_app("chrome")`;

3) агент знаходить відповідну програму в локальному індексі та запускає її;

4) після цього управління передається в AgentOrchestrator, який генерує звіт про виконання запиту.

3.4.5 Приклад 5 – пошук інформації в інтернеті

Одним з можливих запитів користувача є таким, який потребує пошуку інформації в мережі Інтернет. Саме з такою метою було імплементовано агента WebSearchAgent, що використовує сервіс Tavily API для пошуку актуальної інформації з вебджерел.

Типовим запитом до системи є запит «Яка сьогодні погода в Харкові?». Він виконується наступним чином:

1) AgentOrchestrator делегує запит WebSearchAgent;

2) агент виконує `search_web(query)`, надсилаючи запит до Tavily API;

3) отримана відповідь повертається у вигляді стисненої релевантної інформації та передається до оркестратора, що генерує звіт для користувача.

Таким чином, система демонструє здатність динамічно взаємодіяти з вебресурсами, що значно розширює її функціональні можливості й дозволяє виконувати запити, які виходять за межі локального середовища.

3.5 Демонстрація роботи програми

У цьому розділі представлено практичну демонстрацію роботи розробленої системи асистента на базі агентів із використанням фреймворків LlamaIndex для обробки природних мовних запитів та Electron для створення графічного інтерфейсу користувача.

Особлива увага приділена показу роботи ключових компонентів системи – фронтенду, що забезпечує зручний і інтуїтивний інтерфейс, та бекенду, що реалізує логіку агентів і обробляє поставлені завдання за допомогою великих мовних моделей. Демонстрація ілюструє, як система розбиває складні запити на підзадачі, виконує їх послідовно і надає користувачу зрозумілий зворотний зв’язок.

3.5.1 Загальний огляд інтерфейсу користувача

Інтерфейс користувача реалізовано з використанням фреймворку Electron, який дозволяє створювати кросплатформенні додатки із сучасним зовнішнім виглядом і зручною логікою взаємодії з використанням звичного набору технологій – HTML, CSS та Javascript.

Головне вікно програми має лаконічний дизайн, що забезпечує простоту використання навіть для користувачів без спеціальної технічної підготовки, що важливо для забезпечення доступу до системи-асистента

усіх груп користувачів. Воно основним центром взаємодії користувача з системою та дозволяє отримати доступ до усіх функцій системи.

Воно має наступні елементи інтерфейсу – поле вводу запиту, кнопку відправлення запиту, кнопка переходу до вікна історії запитів та кнопка запуску вікна налаштувань. Вигляд інтерфейсу головного вікна наведено у рисунку 3.3.

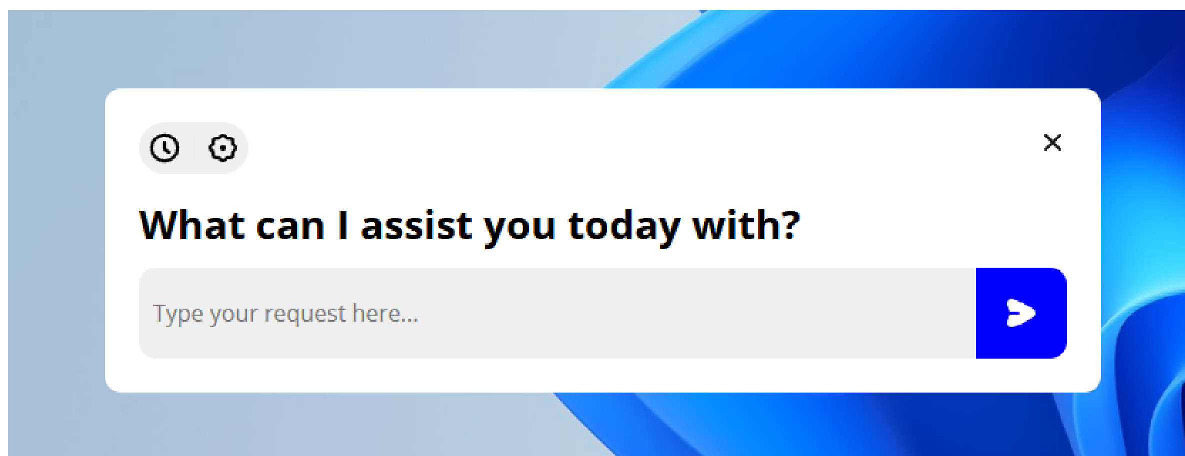

Рисунок 3.3 – Інтерфейс головного вікна програми

Головне вікно програми, представлене на рисунку 3.3, є основною точкою взаємодії користувача з асистентом. Інтерфейс розроблено з акцентом на мінімалізм та інтуїтивну зрозумілість, що дозволяє користувачеві швидко розпочати роботу без необхідності тривалого вивчення програми. Такий дизайн інтерфейсу забезпечує низький поріг входження для користувачів та зосереджує їхню увагу на ключовій функції – керуванні операційною системою за допомогою текстових команд.

При натисканні кнопки історії (кнопка зліва) відкривається вікно перегляду історії усіх запитів. У ньому відображаються усі виконанні системою запити, напроти кожного запита знаходяться дві кнопки – для видалення запису та для повторного запуску виконання запиту. Також є

можливість видалити усі записи з бази даних історії. Візуальний інтерфейс вікна можна побачити на рисунку 3.4.

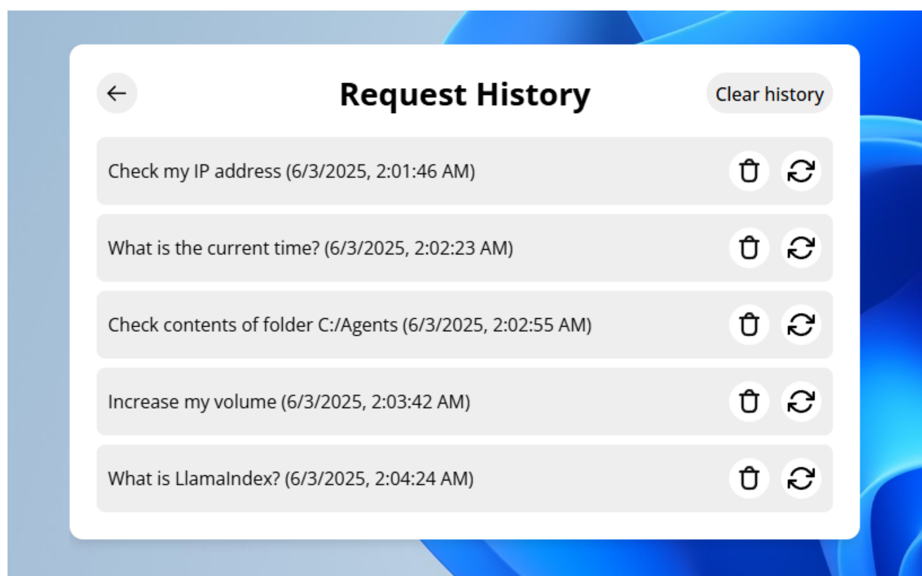

Рисунок 3.4 – Інтерфейс вікна перегляду історії

У свою чергу, натискання кнопки налаштувань (кнопка справа) відкриває вікно налаштувань програми. Воно має мінімалістичний дизайн та наступні можливості: видалення логів, видалення записів історії та зміна мови агентів. Зовнішній вигляд вікна налаштувань наведено на рисунку 3.5.

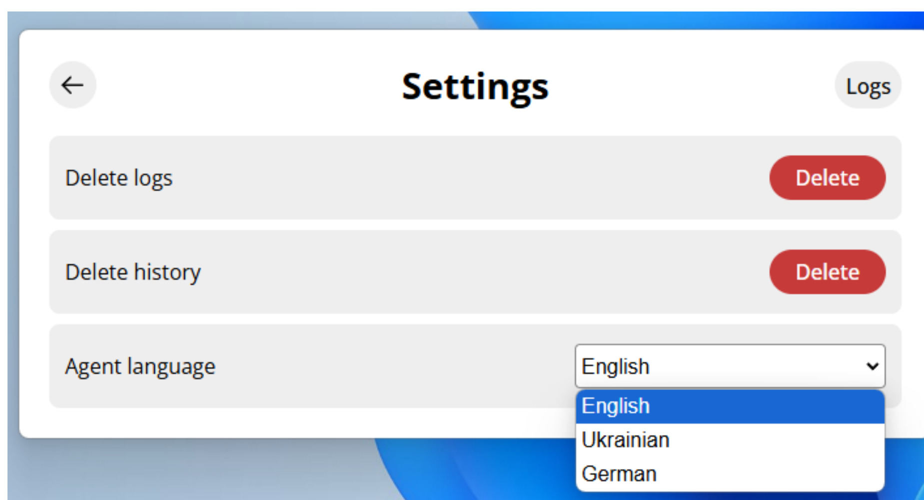

Рисунок 3.5 – Інтерфейс вікна налаштувань застосунку

Натискання кнопки «Logs» відкриває вікно перегляду логів. Логування є важливою частиною мультиагентного застосунку, тому наочності та зручності перегляду логів було приділено особливу увагу. Записи логів представлені у формі таблиці, в правій колонці якої зазначено тип логу (input_message – запит користувача, current_agent – інформація про поточного агента, tool_call – функціональний виклик, tool_result – результати функціонального виклику, agent_output – звіти агентів), а в правій частині – повідомлення логу. Візуальний інтерфейс вікна продемонстровано на рисунку 3.6.

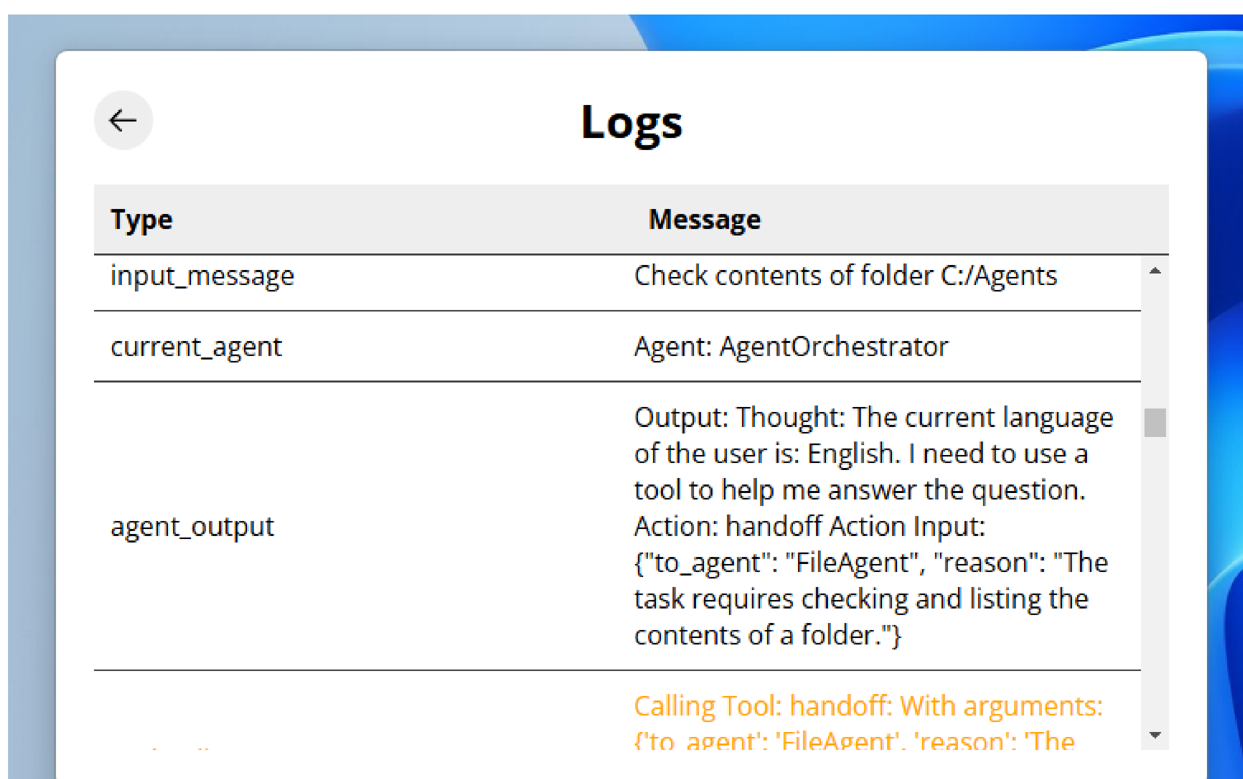

Рисунок 3.6 – Візуальний інтерфейс вікна перегляду логів

3.5.2 Демонстрація виконання запиту

Для демонстрації функціональності розробленої системи розглянемо типовий сценарій взаємодії користувача з асистентом. У якості прикладу

використовується запит природною мовою: «Зменши гучність до 30% та увімкни наступний трек». Цей запит є досить простим для демонстрації, бо має бути виконаний лише одним агентом управління медіаконтентом, але у два виклики. Незважаючи на простоту, такий запит є одним з найпоширеніших у користуванні системою-асистентом та таким чином демонструє реальну задачу використання агентської системи. Зовнішній вигляд вікна з введеним запитом продемонстровано на рисунку 3.7.

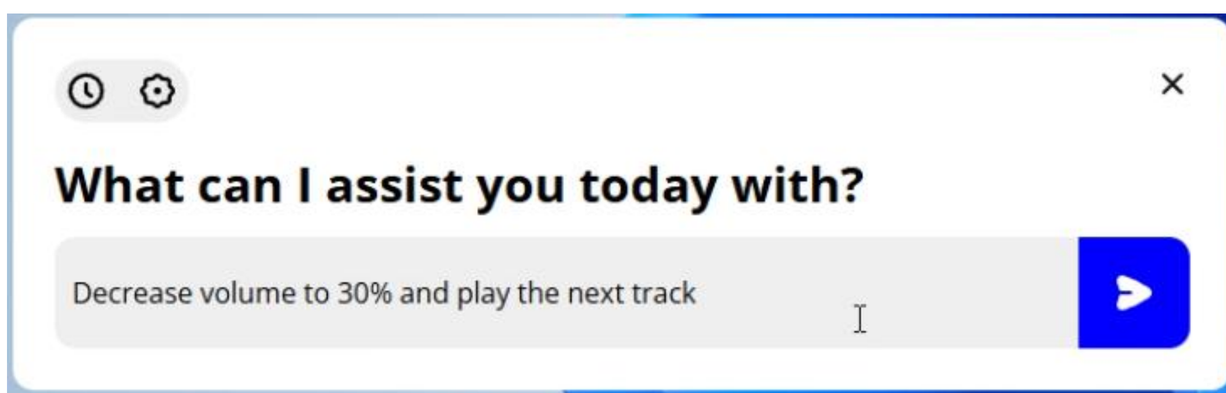

Рисунок 3.7 – Візуальний інтерфейс головного вікна програми з введеним запитом

Після натискання кнопки відправлення запиту система-асистент починає виконання дій. Система динамічно завантажує проміжні результати виконання, а саме виклики передачі управління від одного агента то іншого (handoff-виклики). Вони відображаються у вигляді невеликих вікон, які з’являються над основним вікном та демонструють ім’я агента, якому делегується дія та причина делегування. Такий спосіб демонстрації проміжних результатів надає користувачу можливість детально дізнатися хід виконання його запиту та контролювати інтелектуального асистента. У даному випадку система виконала чотири виклика агентів, що продемонстровано на рисунку 3.8.

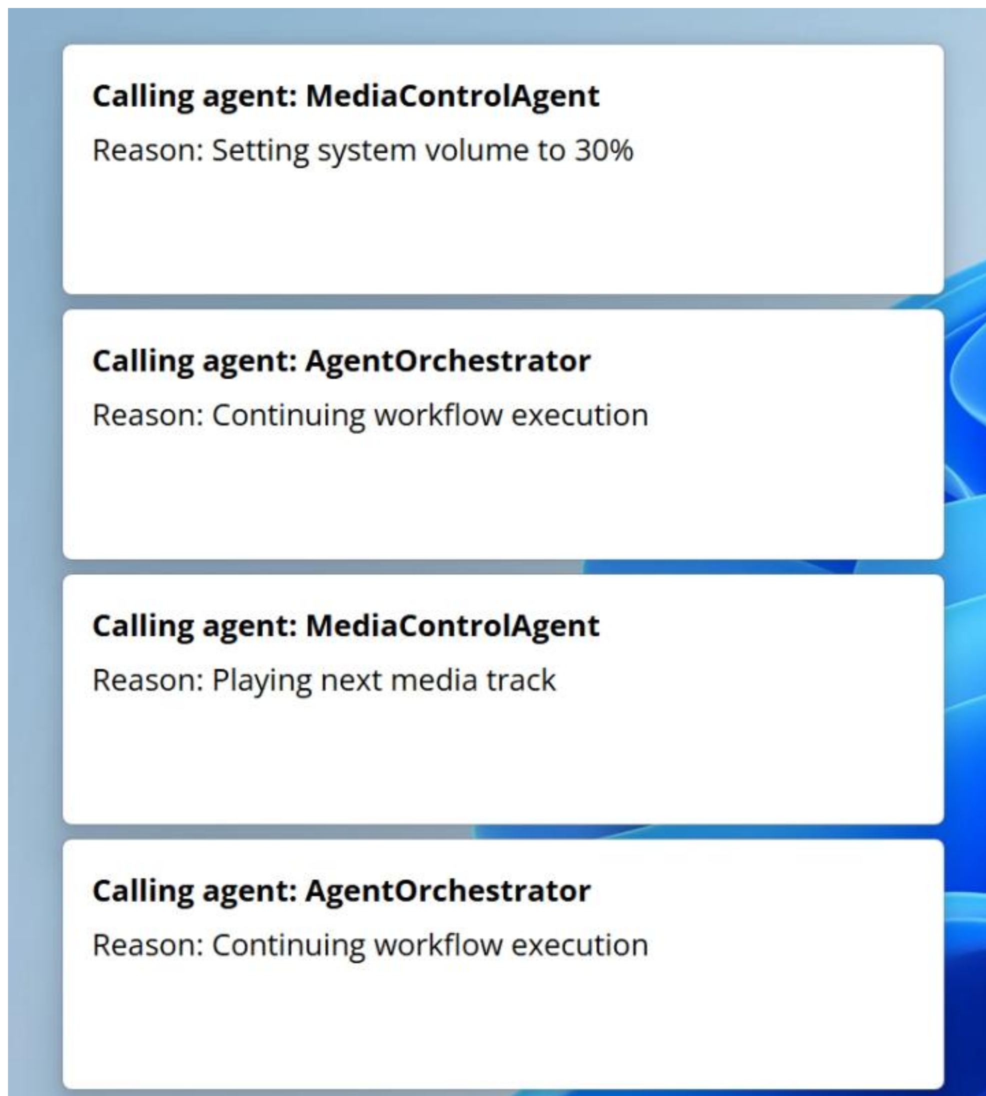

Рисунок 3.8 – Вікна, що надають інформацію про виконанні агентні виклики

При завершені виконання дій система генерує звіт з виконання. Він відображається у новому вікні та має головне поле для виводу повідомлення та елемент для демонстрації часу виконання запиту. Таке вікно відображається кожен раз при завершені виконання запиту, незважаючи на результати. Навіть у випадках помилок, система відобразить причини помилки. У даному випадку, система сповістила користувача про успішну зміну гучності програвання та увімкнення наступного треку. Звіт наведено на рисунку 3.9.

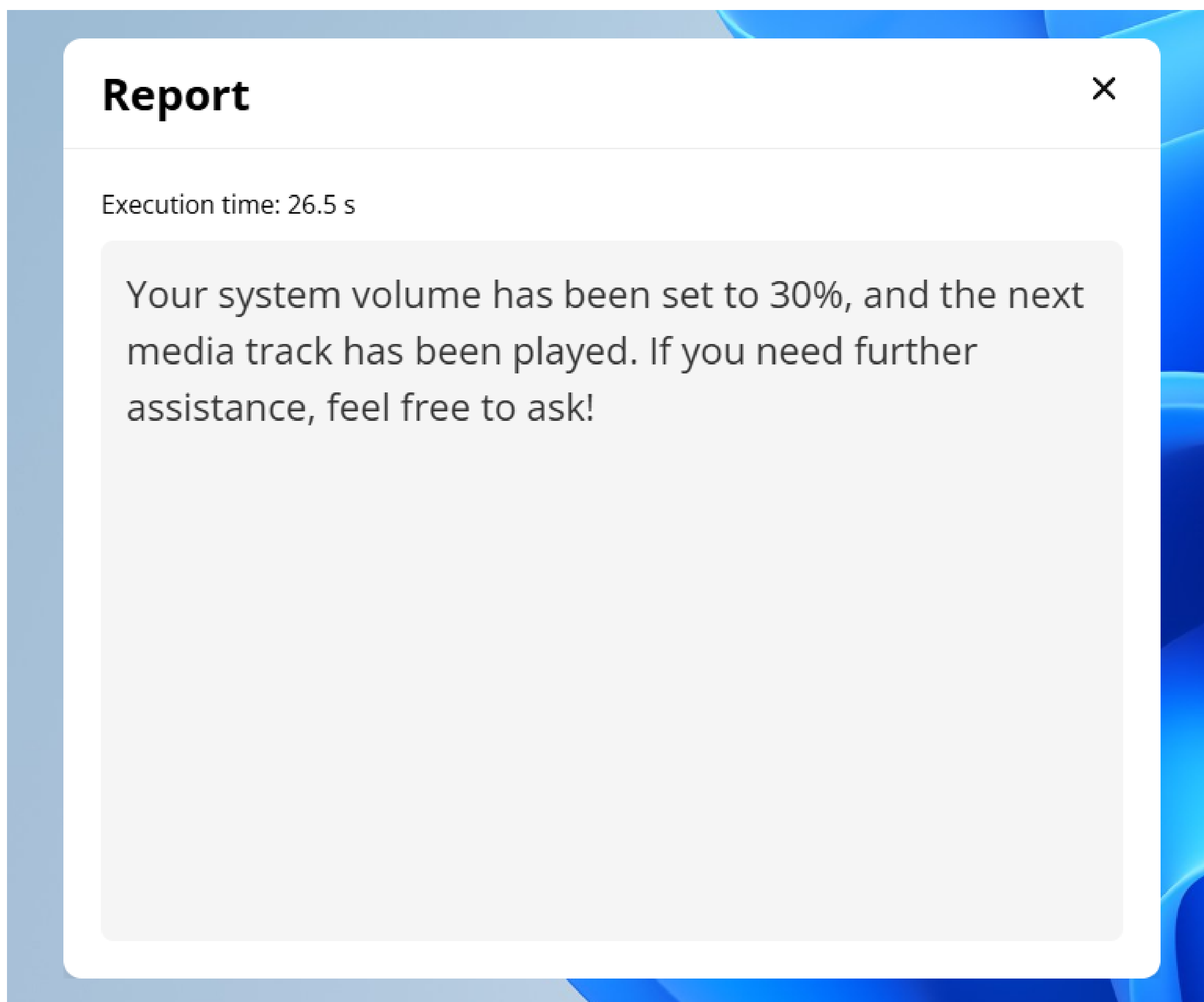

Рисунок 3.9 – Звіт з виконання запиту користувача

Таким чином, інтерфейс користувача розробленої системи-асистента забезпечує інтуїтивну взаємодію, зручний доступ до основних функцій та наочне відображення результатів виконання запитів. Це дозволяє ефективно використовувати систему як технічно підготовленим користувачам, так і людям без великого досвіду користування комп’ютера.

ВИСНОВКИ

У межах даної кваліфікаційної роботи було реалізовано систему-асистент для керування операційною системою Windows, засновану на мультиагентній архітектурі з використанням великих мовних моделей (LLM) та сучасного фреймворку для створення інтелектуального програмного забезпечення LlamaIndex. Запропонований підхід дозволив створити гнучку та розширювану платформу, здатну сприймати запити природною мовою, декомпозувати їх на підзадачі та послідовно виконувати команди у взаємодії з Powershell та іншими системними утилітами.

Однією з ключових особливостей розробленої системи стало використання локальних відкритих мовних моделей. Застосування таких моделей забезпечує високий рівень приватності, автономність і незалежність від сторонніх API-сервісів. Водночас, моделі з відкритим кодом, зокрема Qwen, Llama 3 та Gemma 3, мають суттєві переваги, що робить їх ідеальним інструментом для дослідницьких та прикладних завдань:

– доступність і відсутність обмежень на використання;

– низькі вимоги до обчислювальних ресурсів;

– гнучкість для модифікації та донавчання.

Попри те, що вони поступаються за масштабом комерційним хмарним аналогам, їх якість є цілком достатньою для вирішення широкого спектра практичних задач, особливо в контексті автономних асистентів.

Фреймворк LlamaIndex було обрано як основу для оркестрації агентів, керування контекстом, управління пам’яттю та структурованою взаємодією між компонентами. Використання цієї технології значно спростило реалізацію ReAct-агентів, що приймають рішення на основі проміжних результатів і можуть адаптувати свої дії залежно від контексту. Його інтеграція з локальними моделями дозволила зберегти модульність і

масштабованість системи, забезпечуючи водночас зручність розширення й адаптації під різні сценарії використання.

У результаті виконання кваліфікаційної роботи було:

– спроєктовано і реалізовано інтелектуальної асистент-систему з підтримкою мультиагентної архітектури на основі ReAct-агентів;

– інтегровано локальні мовні моделі для забезпечення автономної роботи системи;

– реалізовано взаємодію між агентами за допомогою LlamaIndex та ефективне керування контекстом;

– розроблено механізм генерації та виконання команд операційної системи на основі інструкцій користувача.

Таким чином, поставлена мета кваліфікаційної роботи була досягнута. Побудована система демонструє перспективність підходу, який поєднує локальні LLM-моделі та сучасні інструменти керування агентами, і може бути використана як основа для подальшого розвитку інтелектуальних інтерфейсів для автоматизації роботи з операційними системами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Understanding Software Agents and Their Autonomous Abilities | Lenovo US. *Offizielle Lenovo DE Website | Notebooks, Tablets, PCs, Rechenzentren, Phones | Lenovo Deutschland*. URL: https://www.lenovo.com/us/en/glossary/agent/ (date of access: 28.04.2025).

2. What is a Multiagent System? | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/multiagent-system (date of access: 28.04.2025).

3. Types of AI Agents | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/ai-agent-types (date of access: 30.04.2025).

4. ReAct: Synergizing Reasoning and Acting in Language Models / S. Yao et al. *arXiv preprint*. 2022.

5. The ReAct Framework in Agentic AI: Transforming Enterprise Problem-Solving. *Lowtouch.Ai*. URL: https://www.lowtouch.ai/react-framework-ai-enterprise-automation/ (date of access: 11.05.2025).

6. What is a ReAct Agent? | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/react-agent (date of access: 11.05.2025).

7. What is LLM? - Large Language Models Explained - AWS. *Amazon Web Services, Inc*. URL: https://aws.amazon.com/what-is/large-language-model/ (date of access: 01.05.2025).

8. IBM. What Are Large Language Models (LLMs)? | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/large-language-models (date of access: 01.05.2025).

9. Attention Is All You Need / A. Vaswani et al. *arXiv preprint arXiv:1706.03762*. 2017.

10. AI Research Blog - The Transformer Blueprint: A Holistic Guide to the Transformer Neural Network Architecture. *AI Research Blog* -. URL: https://deeprevision.github.io/posts/001-transformer/ (date of access: 04.05.2025).

11. How transformers work: a detailed exploration of transformer architecture. *datacamp.com*. URL: https://www.datacamp.com/tutorial/how-transformers-work (date of access: 04.05.2025).

12. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges / X. Li et al. *Vicinagearth*. 2024. Vol. 1, no. 1. URL: https://doi.org/10.1007/s44336-024-00009-2 (date of access: 04.05.2025).

13. Sypherd C., Belle V. Practical Considerations for Agentic LLM Systems. *arXiv preprint*. 2024.

14. Top 10 Cons & Disadvantages of Large Language Models (LLM). *ProjectManagers.net*. URL: https://projectmanagers.net/top-10-disadvantages-of-large-language-models-llm (date of access: 06.05.2025).

15. Пасічний О. Велике оновлення Windows 11 2024 з новим Copilot та ШІ-функціями: на що варто очікувати?. *TERAZUS*. URL: https://terazus.com/uk/2786-velike-onovlennja-windows-11-2024-z-novim-copilot-ta-shi-funktsijami-na-scho-varto-ochikuvati (дата звернення: 12.05.2025).

16. Marquardt B. The Assistant experience on mobile is upgrading to Gemini. *Google*. URL: https://blog.google/products/gemini/google-assistant-gemini-mobile/ (date of access: 12.05.2025).

17. Braina - Artificial General Intelligence (AGI) Software for PC. *Brainasoft - Home*. URL: https://www.brainasoft.com/braina/ (date of access: 12.05.2025).

18. Tahir. What is Ollama: Running Large Language Models Locally. *Medium*. URL: https://medium.com/@tahirbalarabe2/what-is-ollama-running-large-language-models-locally-e917ca40defe (date of access: 12.05.2025).

19. LlamaIndex - LlamaIndex. *LlamaIndex - LlamaIndex*. URL: https://docs.llamaindex.ai/en/latest/ (date of access: 12.05.2025).

20. What Is ElectronJS and When to Use It. *Brainhub*. URL: https://brainhub.eu/library/what-is-electron-js (date of access: 12.05.2025).