

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА РІШЕННЯ ДЛЯ ОТРИМАННЯ КОНСОЛІДОВАНОЇ
ВІДПОВІДІ ВІД ДЕКІЛЬКОХ ШІ-МОДЕЛЕЙ НА PYTHON**

**DEVELOPING A SOLUTION TO OBTAIN A CONSOLIDATED RESPONSE
FROM MULTIPLE AI MODELS PYTHON**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-41
Свиридюк Андрій Анатолійович

Керівник:
Бойко Лев Степанович

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Свиридюку Андрію Анатолійовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка рішення для отримання консолідованої відповіді від декількох ШІ - моделей на Python»

Керівник роботи: доц., Бойко Л. С.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: предметна область взаємодії з великими мовними моделями та агрегації їх відповідей, технології розробки програмного забезпечення мовою Python, засоби роботи з API мовних моделей (OpenAI, Mistral, OpenRouter), бібліотеки для обробки HTTP-запитів та асинхронного програмування, вимоги до інформаційних систем та забезпечення надійності обробки даних.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): аналіз сучасного стану великих мовних моделей та підходів до агрегації відповідей, постановка завдання кваліфікаційної роботи, формування вимог до програмного рішення, обґрунтування вибору технологій і засобів розробки, практична реалізація програмного забезпечення для взаємодії з AI-моделями, розробка механізму агрегації відповідей, тестування та налагодження програмного забезпечення, аналіз результатів роботи програмного рішення.

5. Перелік графічного матеріалу: 5 рисунків, 5 лістингів.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Бойко Л. С.</i>		
Специфікація вимог до розробленої системи	<i>Бойко Л. С.</i>		
Розробка об'єкта проектування	<i>Бойко Л. С..</i>		
Нормоконтроль	<i>Суринович О. М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	<i>%</i>		
Академічна доброчесність	<i>Бойко Л. С..</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти _____

Андрій СВИРИДЮК

Керівник кваліфікаційної роботи _____

Лев БОЙКО

АНОТАЦІЯ

Свиридюк А. С. Розробка рішення для отримання консолідованої відповіді від декількох ШІ-моделей на Python. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У роботі досліджено сучасні підходи до використання великих мовних моделей та отримання відповідей через API. Проведено аналіз існуючих рішень у сфері взаємодії з AI-моделями, розглянуто проблему неузгодженості відповідей та підходи до їх агрегації.

У першому розділі виконано аналіз предметної області та сучасного стану розвитку мовних моделей. У другому розділі сформувано вимоги до програмного рішення, обґрунтовано вибір технологій та виконано проектування структури програмного забезпечення. У третьому розділі описано процес реалізації програмного рішення мовою Python, механізми взаємодії з API мовних моделей, агрегації відповідей та результати тестування.

У результаті виконання роботи розроблено програмне рішення для отримання відповідей від декількох мовних моделей з подальшим формуванням консолідованого результату.

Ключові слова: великі мовні моделі, AI, API, агрегація відповідей, Python, OpenAI, Mistral, OpenRouter.

ABSTRACT

Svyrydiuk A. S. Developing a solution to obtain a Consolidated response from Multiple AI Models Python. Qualification work for bachelor's degree in Software Engineering, specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification work consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The work investigates modern approaches to the use of large language models and obtaining responses through APIs. Existing solutions for interaction with AI models are analyzed, as well as the problem of response inconsistency and approaches to response aggregation.

The first chapter analyzes the problem domain and the current state of language model development. The second chapter defines the requirements for the software solution, substantiates the choice of technologies, and describes the software structure design. The third chapter presents the implementation of the software solution in Python, mechanisms for interaction with language model APIs, response aggregation, and testing results.

As a result of the work, a software solution for obtaining responses from multiple language models with further generation of a consolidated result was developed.

Keywords: large language models, AI, API, response aggregation, Python, OpenAI, Mistral, OpenRouter.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ СИСТЕМ ОТРИМАННЯ ТА АГРЕГАЦІЇ ВІДПОВІДЕЙ ШІ-МОДЕЛЕЙ	9
1.1 Аналіз сучасного стану проблеми	9
1.2 Постановка завдання на кваліфіковану роботу бакалавра	15
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО СИСТЕМИ АГРЕГАЦІЇ ВІДПОВІДЕЙ ДЕКІЛЬКОХ ШІ-МОДЕЛЕЙ	17
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення	17
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання	22
РОЗДІЛ 3 РОЗРОБКА СИСТЕМИ АГРЕГАЦІЇ ВІДПОВІДЕЙ ДЕКІЛЬКОХ ШІ-МОДЕЛЕЙ	27
3.1 Практична реалізація об'єкта проектування	27
3.2 Тестування та налагодження інформаційно-комп'ютерної системи	28
3.3 Загальна архітектура системи	30
3.4 Реалізація взаємодії з API моделей	34
3.5 Реалізація механізму агрегації відповідей	37
3.6 Реалізація користувацького інтерфейсу	39
3.7 Організація збереження даних	40
ВИСНОВКИ	44
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	46

ВСТУП

Актуальність теми. Упродовж останніх років великі мовні моделі стали одним із найбільш перспективних напрямів у сфері штучного інтелекту. Основою сучасних мовних моделей стали трансформерні архітектури, що забезпечили суттєве підвищення якості обробки та генерації тексту [1]. Подальший розвиток великих мовних моделей дозволив використовувати їх для генерації тексту, аналізу інформації, програмування та автоматизації різних процесів [2].

Разом із поширенням мовних моделей виникає проблема неузгодженості відповідей. Навіть при використанні однакових запитів різні моделі можуть формувати результати, що відрізняються за змістом, структурою або точністю. У деяких випадках відповіді можуть доповнювати одна одну, однак інколи між ними виникають суперечності, що ускладнює використання окремих моделей у задачах, де важливими є стабільність та узгодженість результатів [3].

Одним із напрямів вирішення цієї проблеми є використання підходів, пов'язаних із агрегацією та узагальненням відповідей кількох моделей. У сучасних дослідженнях розглядаються методи self-consistency, багатоагентної взаємодії та покрокового узгодження результатів. Подібні підходи активно досліджуються, однак більшість існуючих рішень орієнтовані переважно на дослідницькі або вузькоспеціалізовані задачі.

Окремою особливістю сучасних AI-сервісів є активний розвиток API платформ, які забезпечують доступ до мовних моделей через HTTP-запити. Сьогодні доступ до моделей надають OpenAI, Mistral та інші сервіси, що створює можливість одночасного використання декількох моделей у межах одного програмного рішення [4]. Передача даних під час взаємодії з сервісами мовних моделей переважно здійснюється у форматі JSON, що забезпечує уніфікований підхід до обробки відповідей.

Актуальність роботи полягає у необхідності створення програмного рішення, яке дозволяє отримувати відповіді від декількох мовних моделей,

виконувати їх агрегацію та формувати консолідований результат. Такий підхід дозволяє підвищити інформативність і узгодженість відповідей у порівнянні з використанням окремих моделей.

Метою кваліфікаційної роботи є розробка програмного рішення для отримання консолідованої відповіді від декількох ШІ-моделей із використанням мови програмування Python.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати сучасний стан розвитку великих мовних моделей та існуючих підходів до агрегації відповідей;
- визначити вимоги до розроблюваного програмного рішення;
- обґрунтувати вибір технологій та засобів розробки;
- спроектувати структуру програмного забезпечення;
- реалізувати взаємодію з API мовних моделей;
- реалізувати механізм агрегації відповідей;
- забезпечити збереження історії взаємодій;
- провести тестування та налагодження програмного рішення.

Об'єктом дослідження є процес отримання та обробки відповідей великих мовних моделей.

Предметом дослідження є методи та засоби розробки програмного рішення для агрегації відповідей від декількох AI-моделей.

Практичне значення роботи полягає у створенні програмного рішення, яке може використовуватись для отримання більш узгоджених відповідей шляхом одночасного використання декількох мовних моделей та їх подальшого узагальнення.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ СИСТЕМ ОТРИМАННЯ ТА АГРЕГАЦІЇ ВІДПОВІДЕЙ ШІ-МОДЕЛЕЙ

1.1 Аналіз сучасного стану проблеми

Великі мовні моделі (Large Language Models, LLM) є одним із ключових напрямів розвитку сучасних систем штучного інтелекту, орієнтованих на обробку природної мови [1]. Основною функцією таких моделей є аналіз текстових даних та генерація відповідей на основі вхідного запиту користувача, що дозволяє застосовувати їх у широкому спектрі задач, зокрема автоматичному перекладі, узагальненні текстів та інформаційному пошуку [2].

Сучасні мовні моделі, як правило, базуються на архітектурі трансформера, яка забезпечує ефективну обробку послідовностей даних за рахунок використання механізму самоуваги (self-attention) [1]. Даний підхід дозволяє враховувати контекст взаємозв'язків між словами незалежно від їх розташування у тексті, що є критично важливим для формування змістовних відповідей.

Процес навчання великих мовних моделей зазвичай включає два основні етапи: попереднє навчання (pre-training) на великих масивах текстових даних та подальше донавчання (fine-tuning), орієнтоване на конкретні прикладні задачі [2]. Така структура навчання забезпечує універсальність моделей і можливість їх адаптації до різних сценаріїв використання.

Разом із цим, незважаючи на високий рівень якості генерації тексту, мовні моделі мають обмеження, пов'язані з відсутністю фактичного розуміння змісту інформації, що може призводити до появи неточностей або суперечливих відповідей [3]. Це обумовлює необхідність додаткових підходів до підвищення надійності результатів.

Таким чином, великі мовні моделі є потужним інструментом обробки природної мови, проте їх використання вимагає врахування особливостей

функціонування та потенційних обмежень, що визначає актуальність подальших досліджень у цій області.

Використання великих мовних моделей, як правило, здійснюється через програмні інтерфейси доступу Application Programming Interface (API), що дозволяє інтегрувати можливості штучного інтелекту у зовнішні програмні системи без необхідності локального розгортання моделей [4]. Такий підхід обумовлений значними обчислювальними ресурсами, необхідними для функціонування сучасних моделей, що робить їх використання у вигляді віддалених сервісів більш доцільним [2].

З технічної точки зору взаємодія з API мовних моделей реалізується шляхом формування HTTP-запитів, які передаються на сервіс, що надає доступ до мовної моделі, після чого система отримує відповідь у структурованому вигляді [5]. Найчастіше обмін даними здійснюється у форматі JSON, що забезпечує універсальність та сумісність із більшістю мов програмування [6]. Така організація взаємодії дозволяє розробникам використовувати мовні моделі як окремі сервіси без необхідності втручання у їх внутрішню реалізацію.

Серед підходів до організації доступу до мовних моделей можна виділити прямий та агрегований доступ. У першому випадку взаємодія відбувається безпосередньо з API конкретного постачальника, що забезпечує повний контроль над параметрами виклику та поведінкою моделі [7]. У другому випадку використовується проміжний сервіс, який надає єдиний інтерфейс доступу до декількох моделей одночасно, що значно спрощує інтеграцію та дозволяє швидко змінювати використовувані моделі без суттєвих змін у програмному коді [8].

Перевагою використання API є гнучкість та масштабованість, оскільки розробник може змінювати параметри генерації, тип моделі або навіть постачальника сервісу без зміни архітектури системи. Водночас такий підхід має і певні обмеження, серед яких варто виділити залежність від зовнішніх сервісів, затримки, пов'язані з мережею, а також можливі обмеження на кількість запитів або їх вартість [4].

Таким чином, використання API є базовим механізмом інтеграції мовних моделей у сучасні інформаційні системи, який забезпечує баланс між доступністю технології та складністю її реалізації.

У процесі використання великих мовних моделей виявляється одна з ключових особливостей їх роботи – відсутність повної узгодженості відповідей. Йдеться не лише про різницю між окремими моделями, але й про ситуації, коли одна й та сама модель може формувати різні результати для однакових або близьких за змістом запитів [9]. Така поведінка не є випадковою помилкою, а впливає з принципу функціонування моделей, які генерують текст на основі ймовірнісних оцінок, а не жорстко визначених правил.

У процесі використання мовних моделей спостерігається варіативність результатів генерації. При зверненні до декількох моделей з однаковим запитом можуть формуватися відповіді, що відрізняються не лише стилістично, але й за рівнем деталізації або окремими фактами. У ряді випадків такі відмінності є незначними, однак інколи вони ускладнюють інтерпретацію результату, особливо при використанні системи для отримання узагальненої або довідкової інформації.

Окремо слід враховувати вплив параметрів генерації. Зміна таких значень, як *temperature* або *top-p*, безпосередньо впливає на варіативність відповіді, оскільки визначає ступінь «випадковості» при виборі наступних елементів тексту [10]. У результаті навіть при використанні однієї і тієї ж моделі та ідентичного запиту того ж запиту можна отримати різні варіанти відповіді, що додатково знижує передбачуваність системи.

Ще один аспект пов'язаний з відмінностями між самими моделями. Вони можуть бути навчені на різних наборах даних, мати різну архітектуру або проходити різні етапи донавчання. Через це відповіді формуються з урахуванням різного «досвіду» моделі, що призводить до розбіжностей навіть у базових поясненнях або описах [9]. Фактично кожна модель інтерпретує запит по-своєму, що не завжди є очевидним для користувача.

З одного боку, така варіативність може розглядатися як перевага, оскільки дозволяє отримати декілька альтернативних відповідей. З іншого боку, у прикладних системах це створює проблему відсутності єдиного узгодженого результату. Особливо це критично для задач, де необхідна стабільність, наприклад при автоматизації обробки запитів або інтеграції у користувацькі сервіси.

Таким чином, неузгодженість відповідей є системною властивістю великих мовних моделей, яка прямо впливає на можливість їх використання у прикладних рішеннях. Саме ця проблема визначає необхідність пошуку підходів до об'єднання або узагальнення результатів, що і є одним із напрямів даного дослідження.

Проблема варіативності та неузгодженості відповідей мовних моделей зумовила появу низки підходів, спрямованих на підвищення стабільності результатів. На сьогодні не існує єдиного універсального рішення, однак у наукових та прикладних роботах простежується декілька характерних напрямів, які застосовуються для зменшення розбіжностей між відповідями моделей.

Одним із найбільш поширених підходів є використання ансамблевих методів. Їх суть полягає у поєднанні результатів декількох моделей або декількох запусків однієї моделі з метою отримання більш узгодженого підсумкового результату [11]. У класичному вигляді ансамблі передбачають голосування, усереднення або вибір найбільш ймовірного варіанту. У контексті мовних моделей цей підхід набуває дещо іншої форми, оскільки результати представлені у вигляді тексту, а не числових значень, що ускладнює їх безпосереднє порівняння.

Різновидом ансамблевого підходу є методи, засновані на багаторазовій генерації відповіді з подальшим вибором найбільш узгодженого варіанту. Зокрема, підхід self-consistency передбачає генерацію декількох відповідей для одного запиту та вибір результату, який найчастіше повторюється або має найбільшу логічну узгодженість [12]. Такий метод дозволяє частково зменшити

вплив випадковості, однак потребує додаткових обчислювальних ресурсів і не гарантує повної відсутності суперечностей.

Ще одним підходом до зменшення неузгодженості відповідей мовних моделей є використання багатоагентних систем, у яких декілька моделей або екземплярів моделей взаємодіють між собою, обмінюючись результатами та уточнюючи відповіді [13]. У таких системах кожен агент може виконувати окрему функцію, зокрема генерацію відповіді, її перевірку або редагування. Взаємодія між агентами дозволяє поступово підвищувати якість результату, однак структура таких рішень є значно складнішою у порівнянні з прямим використанням окремих API викликів.

Також на практиці застосовуються підходи, засновані на ранжуванні результатів (reranking). У цьому випадку декілька згенерованих відповідей оцінюються за певними критеріями, після чого обирається найбільш релевантний варіант. Критерії можуть включати як формальні метрики, так і оцінку за допомогою додаткової моделі. Незважаючи на відносну простоту ідеї, ефективність такого підходу значною мірою залежить від якості функції оцінювання.

Слід зазначити, що наведені підходи не є взаємовиключними і часто комбінуються між собою. Наприклад, ансамблеві методи можуть доповнюватися ранжуванням або використовуватися в межах багатоагентних систем. Водночас кожен із них має власні обмеження, пов'язані зі складністю реалізації, витратами ресурсів або залежністю від додаткових компонентів.

Таким чином, сучасні підходи до вирішення проблеми неузгодженості відповідей мовних моделей демонструють різні способи комбінування результатів, проте жоден із них не забезпечує універсального та одночасно простого у реалізації рішення. Це створює передумови для пошуку більш гнучких підходів, орієнтованих на практичне використання у прикладних системах.

Незважаючи на різноманітність підходів до зменшення неузгодженості відповідей мовних моделей, жоден із них не забезпечує одночасно простого

впровадження, стабільності результатів та універсальності застосування. У процесі аналізу стає очевидним, що більшість рішень орієнтовані або на підвищення якості відповіді, або на гнучкість системи, проте рідко поєднують ці характеристики у повному обсязі.

Окремо варто звернути увагу на відсутність універсального інструменту, який дозволяв би працювати з різними мовними моделями в єдиному середовищі. Більшість існуючих рішень орієнтовані на конкретні платформи або моделі, що обмежує їх застосування у більш широкому контексті. У результаті інтеграція декількох моделей часто потребує додаткових зусиль на рівні розробки, що ускладнює створення узагальнених рішень [14].

Ще одним важливим аспектом є зручність взаємодії з системою. Багато існуючих реалізацій орієнтовані переважно на дослідницьке використання або внутрішні інструменти, що призводить до недостатньої уваги до користувацького інтерфейсу. Це ускладнює практичне застосування таких рішень, особливо у випадках, коли система використовується не технічними спеціалістами.

Таким чином, аналіз існуючих підходів показує, що проблема узгодження відповідей мовних моделей вирішується частково, а наявні рішення мають обмеження, пов'язані зі складністю, ресурсними витратами та відсутністю універсальності. Це створює передумови для розробки більш простих і водночас ефективних методів, орієнтованих на практичне використання.

Таким чином, необхідність створення системи для отримання консолідованої відповіді від декількох мовних моделей обумовлена як технічними особливостями сучасних LLM, так і обмеженнями існуючих підходів до їх використання. Реалізація такого рішення дозволяє поєднати переваги різних моделей та підвищити якість кінцевого результату без значного ускладнення архітектури системи.

1.2 Постановка завдання на кваліфіковану роботу бакалавра

На основі проведеного аналізу предметної області встановлено, що використання великих мовних моделей супроводжується проблемою варіативності та неузгодженості відповідей, що ускладнює їх застосування у прикладних інформаційних системах. Незважаючи на наявність підходів до часткового вирішення цієї проблеми, більшість із них або є складними у реалізації, або потребують значних обчислювальних ресурсів, що обмежує їх практичне використання.

З урахуванням зазначеного, метою кваліфікаційної роботи є розробка програмної системи для отримання консолідованої відповіді на основі результатів декількох мовних моделей із використанням механізму їх узагальнення.

Для досягнення поставленої мети необхідно виконати наступні завдання:

1) провести аналіз сучасних підходів до використання великих мовних моделей та способів доступу до них через API з метою визначення можливостей їх інтеграції у програмні системи;

2) розробити структуру програмної системи, яка забезпечує можливість одночасного отримання відповідей від декількох мовних моделей на один запит користувача;

3) реалізувати механізм формування запитів до різних моделей із урахуванням уніфікованого підходу до їх виклику та обробки результатів;

4) розробити метод агрегації відповідей, який дозволяє формувати узагальнений результат на основі декількох варіантів, отриманих від різних моделей;

5) забезпечити обробку помилок та нестабільності роботи зовнішніх API з метою підвищення надійності функціонування системи;

6) реалізувати користувацький інтерфейс, який забезпечує зручну взаємодію із системою та наочне представлення отриманих результатів;

7) передбачити можливість розширення системи шляхом додавання нових мовних моделей без суттєвих змін у її архітектурі;

8) провести тестування розробленої системи та оцінити її роботу у різних сценаріях використання.

У рамках поставлених задач особливу увагу доцільно приділити механізму агрегації відповідей, оскільки саме він визначає якість кінцевого результату. Вибір підходу до узагальнення повинен враховувати як точність сформованої відповіді, так і складність реалізації, що є важливим для практичного застосування системи.

Крім того, при розробці необхідно враховувати обмеження, пов'язані з використанням зовнішніх API, зокрема затримки при обробці запитів, обмеження на кількість викликів та залежність від стабільності сторонніх сервісів [4]. Це вимагає реалізації додаткових механізмів контролю та обробки виключних ситуацій.

У результаті виконання роботи повинна бути створена програмна система, яка дозволяє підвищити узгодженість відповідей за рахунок використання декількох мовних моделей та механізму їх узагальнення, зберігаючи при цьому відносну простоту реалізації та зручність використання.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО СИСТЕМИ АГРЕГАЦІЇ ВІДПОВІДЕЙ ДЕКІЛЬКОХ ШІ-МОДЕЛЕЙ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Основною метою розробки програмного забезпечення є створення системи, яка забезпечує отримання та узагальнення відповідей від декількох мовних моделей з використанням єдиного інтерфейсу взаємодії.

На відміну від окремого використання мовних моделей, розроблювана система орієнтована на поєднання результатів декількох джерел. Це дозволяє підвищити якість кінцевої відповіді та зменшити вплив обмежень окремих моделей.

Для досягнення поставленої мети визначено такі основні завдання:

- організувати взаємодію з кількома API мовних моделей [4];
- забезпечити можливість передачі одного запиту до різних моделей;
- реалізувати отримання та збереження відповідей у структурованому вигляді;
- розробити механізм агрегації відповідей із формуванням єдиного результату;
- створити зручний консольний інтерфейс для взаємодії з користувачем;
- забезпечити можливість розширення системи шляхом додавання нових моделей.

Окремо варто зазначити, що система не орієнтована на вирішення вузькоспеціалізованої задачі. Вона розглядається як універсальний інструмент для роботи з мовними моделями, який може бути використаний у різних сценаріях – від навчальних до прикладних.

Таким чином, у межах даного проєкту основний акцент зроблено не лише на отриманні відповідей, а й на їх подальшому узгодженні та зручному представленні користувачу.

Розроблюване програмне рішення позиціонується як локальний інтерактивний застосунок із текстовим інтерфейсом користувача (TUI), призначений для роботи з декількома мовними моделями через API з подальшою агрегацією отриманих відповідей. Такий підхід дозволяє поєднати простоту використання з достатньо гнучкою внутрішньою структурою програмного забезпечення. Взаємодія з програмою здійснюється через консольний інтерфейс, що забезпечує швидке введення запитів та отримання результатів. Програмне рішення орієнтоване переважно на індивідуальне використання, тому не передбачає складного розподілу прав доступу. Водночас його функціональність дозволяє використовувати програму у різних сценаріях, зокрема для навчальних задач або дослідження роботи мовних моделей.

Для користувача процес взаємодії з програмою зводиться до введення текстового запиту та отримання результату. При цьому внутрішня обробка включає декілька послідовних етапів, які виконуються безпосередньо під час роботи програми. Типовий сценарій використання можна описати таким чином: користувач вводить запит у консоль, далі система одночасно або послідовно передає цей запит до кількох мовних моделей через відповідні API та після отримання відповідей вони зберігаються та передаються на етап агрегації. На основі цього формується фінальна відповідь, яка відображається користувачу разом із проміжними результатами (за потреби).

Важливою частиною роботи системи є збереження історії взаємодій. Кожен запит і отримані відповіді записуються у файл, що дозволяє аналізувати попередні результати або використовувати їх як контекст. Це особливо корисно при повторному тестуванні або порівнянні поведінки моделей.

Також у процесі аналізу вимог було враховано необхідність стабільної роботи при можливих збоях. Оскільки система взаємодіє із зовнішніми API, існує ймовірність помилок або затримок. Тому передбачено механізм обробки виключень (помилки), який дозволяє продовжити роботу навіть у разі відмови окремих моделей.

Загалом можна виділити кілька основних вимог, що впливають із аналізу сценаріїв використання:

- система повинна приймати текстовий запит від користувача;
- забезпечувати взаємодію з кількома мовними моделями;
- обробляти отримані відповіді та формувати фінальний результат;
- зберігати історію взаємодій;
- коректно обробляти помилки при роботі з API;

забезпечувати зручну роботу через консольний інтерфейс.

Таким чином, аналіз вимог показує, що система має поєднувати кілька аспектів: роботу з зовнішніми сервісами, обробку текстових даних та організацію взаємодії з користувачем. Саме ці вимоги враховуються при подальшому проєктуванні.

На основі проведеного аналізу визначено основні функціональні можливості системи, які забезпечують її роботу та взаємодію з користувачем. Функціональні вимоги описують, що саме повинна виконувати система у процесі роботи.

Насамперед програмне рішення повинно забезпечувати введення текстового запиту користувачем через консольний інтерфейс. Після введення запит має передаватися до мовних моделей у незмінному вигляді, що дозволяє зберегти початковий зміст і контекст повідомлення.

Однією з основних функціональних можливостей є взаємодія з декількома мовними моделями через API. Для кожної моделі запит формується окремо з урахуванням особливостей відповідного сервісу. При цьому повинна забезпечуватися можливість одночасної роботи з декількома AI-моделями у межах одного запиту користувача.

Після отримання відповідей від мовних моделей виконується їх збереження, приведення структури даних до єдиного формату та підготовка до подальшого узагальнення. Важливою вимогою є коректне збереження результатів навіть у випадках, коли окремі API-виклики завершуються помилкою.

Одним із ключових етапів роботи програмного рішення є агрегація відповідей, отриманих від декількох мовних моделей. На основі сформованих результатів окрема модель-агрегатор виконує побудову узагальненої відповіді. Від коректності цього процесу значною мірою залежить якість підсумкового результату, оскільки саме на цьому етапі виконується об'єднання та узгодження інформації з різних джерел.

Окрім цього, програмне рішення повинно забезпечувати відображення результатів у зручному для користувача вигляді. Передбачається виведення як фінальної відповіді, так і, за необхідності, проміжних результатів окремих моделей.

Важливою функціональною можливістю також є збереження історії взаємодій. Передбачається фіксація запитів користувача, відповідей моделей та підсумкового результату у файл. Це дозволяє виконувати подальший аналіз роботи програмного рішення та повторно використовувати отримані дані.

Окремо необхідно передбачити обробку помилок під час взаємодії з API. У разі відмови однієї з моделей виконання не повинно припинятися – обробка продовжується з використанням доступних результатів із відповідним повідомленням користувача. Узагальнюючи, функціональні вимоги можна подати у вигляді наступного переліку:

- введення текстового запиту користувача;
- виклик кількох мовних моделей через API;
- отримання та збереження відповідей;
- агрегація результатів за допомогою окремої моделі;
- виведення фінальної відповіді;
- збереження історії взаємодій;
- обробка помилок під час роботи з API.

Таким чином, функціональні вимоги охоплюють повний цикл роботи системи – від введення запиту до отримання підсумкового результату.

Окрім функціональних можливостей, при розробці системи важливу роль відіграють нефункціональні вимоги. Вони визначають не те, що система робить,

а те, як саме вона це робить – тобто характеристики якості, продуктивності та зручності використання.

Однією з ключових вимог є швидкодія системи. Оскільки обробка запитів передбачає звернення до кількох зовнішніх API, час отримання відповіді може збільшуватися. Тому важливо організувати виконання запитів таким чином, щоб мінімізувати затримки, зокрема за рахунок асинхронної обробки.

Важливою вимогою також є забезпечення стабільної роботи програмного рішення у випадках помилок під час взаємодії з API. Навіть якщо окремі сервіси не повертають відповідь або тимчасово недоступні, обробка інших запитів повинна продовжуватись. Для цього передбачено механізми перехоплення помилок та використання резервного сценарію обробки, що дозволяє уникнути аварійного припинення виконання програми.

Ще одна вимога – масштабованість. Архітектура системи повинна дозволяти додавання нових моделей або заміну існуючих без значних змін у коді. Це особливо актуально з урахуванням швидкого розвитку технологій у сфері штучного інтелекту.

Окрему увагу слід приділити зручності використання. Незважаючи на те, що система реалізована у вигляді консольного застосунку, вона повинна забезпечувати зрозумілу взаємодію з користувачем. Це включає логічну структуру виводу, читабельність результатів та передбачувану поведінку системи.

Важливою нефункціональною вимогою є надійність збереження даних. Історія взаємодій повинна записуватися у файл без втрати інформації, що дозволяє аналізувати результати роботи та використовувати їх у подальшому.

Узагальнюючи, основні нефункціональні вимоги можна подати у вигляді:

- забезпечення прийнятної швидкості обробки запитів;
- стабільна робота при помилках API;
- можливість масштабування системи;
- зручний та зрозумілий консольний інтерфейс;
- надійне збереження історії взаємодій.

Таким чином, нефункціональні вимоги доповнюють функціональні характеристики системи та забезпечують її практичну придатність до використання.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Для реалізації програмного рішення, що передбачає взаємодію з мовними моделями через API та подальшу агрегацію отриманих відповідей, можуть використовуватися різні технологічні підходи. Вибір засобів розробки визначає складність реалізації, швидкодію та можливості подальшого розширення програмного продукту.

Одним із підходів є використання мов програмування загального призначення, які підтримують роботу з HTTP-запитами та обробку структурованих даних. До таких мов належать Python, JavaScript та Java. Їх застосування обумовлене необхідністю інтеграції із зовнішніми API сервісів мовних моделей, що передбачає надсилання запитів та обробку відповідей у форматі JSON [6].

Мова JavaScript широко використовується у веб-розробці та має розвинену екосистему для роботи з API, зокрема у середовищі Node.js. Java забезпечує стабільність та використовується у складних програмних системах, однак її застосування може бути менш гнучким у контексті швидкого прототипування. Python, у свою чергу, поєднує простоту синтаксису з наявністю бібліотек для роботи з HTTP-запитами та асинхронною обробкою, що робить її зручною для реалізації задач інтеграції з API мовних моделей.

Разом із цим для задач, пов'язаних із обробкою тексту та інтеграцією з AI – моделями, найбільш поширеним є використання Python. Це пояснюється наявністю великої кількості бібліотек, простотою синтаксису та активною підтримкою у сфері штучного інтелекту.

Також існують спеціалізовані фреймворки для роботи з LLM, які дозволяють будувати складні системи з декількох моделей. Однак такі інструменти часто мають надлишкову функціональність і можуть ускладнювати реалізацію відносно простих задач.

У контексті даної роботи доцільно використовувати підхід із мінімально необхідним набором інструментів. Це дозволяє краще контролювати логіку системи та уникнути зайвої складності.

Таким чином, при виборі засобів розробки враховувалися такі критерії як простота реалізації, можливість роботи з API, підтримка асинхронних викликів та гнучкість у розширенні системи. На основі цього було сформовано подальший вибір технологій.

Для реалізації програмного рішення було обрано мову програмування Python. Такий вибір зумовлений як особливостями поставленого завдання, так і практичними аспектами розробки.

Насамперед Python добре підходить для роботи з текстовими даними та інтеграції з AI-сервісами. Більшість сучасних інструментів у сфері штучного інтелекту або безпосередньо реалізовані на Python, або мають офіційну підтримку цієї мови. Це спрощує роботу з мовними моделями та дозволяє використовувати готові рішення без додаткових адаптацій [15].

Додатковою перевагою Python є простота синтаксису. У порівнянні з іншими мовами програмування, Python дозволяє швидше реалізувати базову логіку системи та зосередитись саме на вирішенні задачі, а не на технічних деталях. Це особливо важливо у випадку, коли система включає взаємодію з кількома зовнішніми сервісами.

Окремою перевагою Python є наявність вбудованих засобів для асинхронного програмування [15]. Під час обробки запитів до мовних моделей виникає необхідність одночасного виконання декількох API-викликів, тому можливість їх паралельної обробки дозволяє суттєво зменшити загальний час отримання відповіді. Для реалізації такого підходу під час розробки

використовувалися механізми асинхронного виконання, зокрема бібліотека `asyncio` [18].

Також важливим є наявність бібліотек для роботи з HTTP-запитами та обробки даних у форматі JSON. Це дозволяє безпосередньо взаємодіяти з API мовних моделей без необхідності створення складної інфраструктури.

Не менш важливим фактором є зручність створення консольних застосунків. Python дозволяє швидко реалізувати CLI-інтерфейс, що відповідає вимогам даної роботи та забезпечує достатній рівень зручності для користувача.

З урахуванням зазначених факторів, Python є оптимальним вибором для реалізації системи, яка поєднує роботу з API, обробку тексту та організацію інтерактивної взаємодії з користувачем.

У процесі реалізації програмного рішення використано бібліотеки стандартного та прикладного рівня, які забезпечують взаємодію з API, обробку запитів та організацію виконання програми. Вибір інструментів визначався простотою інтеграції та відповідністю поставленим задачам.

Для реалізації взаємодії з API мовних моделей застосовано бібліотеки для роботи з HTTP-запитами, зокрема `requests` та `httpx`. Вони забезпечують формування запитів, передачу параметрів та отримання відповідей у структурованому вигляді, що є необхідним для подальшої обробки даних [5, 16].

Важливу функцію виконує бібліотека `asyncio`, яка використовується для організації асинхронного виконання. У межах даної системи це дозволяє виконувати кілька запитів до різних моделей без жорсткого очікування завершення кожного з них. У результаті зменшується загальний час обробки запиту, що позитивно впливає на швидкодію.

Для обробки отриманих даних використовуються JSON-структури, оскільки більшість API мовних моделей повертають відповіді саме у цьому форматі [6]. У процесі обробки необхідно коректно отримувати текст відповіді та службові параметри, після чого передавати їх на наступні етапи виконання.

Окремо варто передбачити збереження історії запитів користувача та отриманих відповідей у файл, що забезпечить можливість подальшого аналізу результатів і використання попереднього контексту під час роботи.

Таким чином, використані бібліотеки та інструменти забезпечують основні функціональні можливості системи: взаємодію з API, асинхронну обробку запитів, роботу з даними та стабільність виконання. При цьому їх використання не ускладнює архітектуру та дозволяє зберігати контроль над логікою роботи системи.

Для забезпечення взаємодії з мовними моделями може використовуватись декілька підходів доступу до API. Одним із варіантів є пряме використання API окремих сервісів, зокрема Mistral, який надає доступ до власних моделей [7]. Іншим підходом є використання універсальних платформ, таких як OpenRouter, що забезпечують єдиний інтерфейс для роботи з різними AI-моделями [8]. Використання комбінованого підходу дозволяє спростити інтеграцію моделей та забезпечити гнучкість під час подальшого розширення програмного рішення.

З практичної точки зору обидва підходи мають схожий принцип роботи. Система формує HTTP-запит, який містить текст користувача та параметри генерації. Після цього запит передається на сервер, де виконується обробка, і повертається відповідь у форматі JSON.

Використання OpenRouter дозволяє працювати з кількома моделями в єдиному форматі. У даній системі це дає можливість легко комбінувати моделі, змінювати їх або додавати нові без суттєвих змін у коді. Це особливо зручно при реалізації багатомодельного підходу.

У свою чергу, використання API Mistral забезпечує прямий доступ до моделі без додаткових рівнів абстракції. Це може позитивно впливати на стабільність і передбачуваність роботи, а також дозволяє точніше контролювати параметри запитів.

У системі ці два підходи поєднуються. Частина моделей викликається

безпосередньо, інша – через агрегований сервіс. При цьому результати обробляються однаково, що дозволяє використовувати їх у межах єдиного алгоритму агрегації.

РОЗДІЛ 3

РОЗРОБКА СИСТЕМИ АГРЕГАЦІЇ ВІДПОВІДЕЙ ДЕКІЛЬКОХ ШІ-МОДЕЛЕЙ

3.1 Практична реалізація об'єкта проєктування

Практична реалізація програмної системи здійснювалась з використанням мови програмування Python у середовищі розробки PyCharm. Вибір даного середовища обумовлений зручністю роботи з проєктами середньої складності, наявністю інструментів для навігації по коду, вбудованими засобами налагодження та підтримкою роботи з віртуальними середовищами. У процесі розробки використовувався модульний підхід, що дозволило розділити функціональність системи на окремі логічні компоненти та спростити подальшу підтримку коду.

Розробка системи відбувалась поетапно. На початковому етапі було реалізовано базову структуру проєкту та механізм обробки запитів користувача. Після цього було додано модулі для взаємодії з API мовних моделей, що дозволило отримувати відповіді від зовнішніх сервісів. Наступним кроком стала реалізація механізму агрегації, який забезпечує об'єднання результатів декількох моделей у єдину відповідь. Завершальним етапом було додавання користувацького інтерфейсу та системи збереження історії запитів.

Для взаємодії з API мовних моделей використовувались HTTP-запити та обробка JSON-даних, що є стандартним форматом обміну інформацією між клієнтською програмою та сервером [6, 17]. Окрему увагу приділено використанню асинхронних викликів, які дозволяють паралельно виконувати запити до декількох моделей і зменшити загальний час отримання відповіді [15, 18].

Під час реалізації логіки програмного рішення використовувались структури даних Python для збереження відповідей моделей та передачі результатів між окремими модулями. Отримані відповіді групувались

відповідно до моделі, від якої вони були отримані, що спрощувало їх подальшу обробку та використання у модулі агрегації.

Окремо слід відзначити реалізацію механізму формування запиту до моделі-агрегатора. У цьому випадку використовується динамічне формування текстового запиту (prompt), який містить інструкції для моделі, поточне питання користувача та відповіді інших моделей. Такий підхід дозволяє використовувати можливості мовних моделей для узагальнення інформації без необхідності реалізації складних алгоритмів на рівні коду. У процесі розробки також було враховано питання обробки помилок. Для цього використовувались конструкції try/except, що дозволяють перехоплювати виключення при роботі з API та забезпечувати стабільність роботи системи навіть у випадку часткових збоїв. Це є важливим аспектом, оскільки зовнішні сервіси не завжди гарантують стабільну відповідь.

Загалом реалізація системи показала, що мова Python є ефективним інструментом для створення подібних програмних рішень. Її гнучкість, велика кількість бібліотек та простота синтаксису дозволяють швидко реалізовувати складну логіку та інтегрувати різні сервіси в межах одного застосунку. Отриманий результат підтверджує можливість використання Python для розробки систем, що працюють із сучасними технологіями штучного інтелекту.

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

У процесі розробки програмної системи було проведено тестування та налагодження її основних компонентів з метою перевірки коректності роботи, стабільності взаємодії з API та узгодженості отриманих результатів. Тестування здійснювалось поетапно, починаючи з перевірки окремих функцій та модулів і завершуючи комплексною перевіркою всієї системи в умовах, максимально наближених до реального використання. Основна увага приділялась перевірці роботи модулів взаємодії з API, механізму агрегації відповідей, а також коректності обробки введених користувачем запитів.

Для тестування використовувались функціональні сценарії, що включали введення різних типів запитів (інформаційних, уточнюючих, складних багаточастинних), аналіз отриманих відповідей від окремих моделей та перевірку якості фінального результату після агрегації. Окремо перевірялась поведінка системи у випадках часткової відмови сервісів, зокрема при відсутності відповіді від однієї з моделей або при помилках API. У таких ситуаціях система повинна була продовжувати роботу та формувати відповідь на основі доступних даних, що було досягнуто за рахунок реалізації механізмів обробки виключень.

У процесі налагодження було виявлено ряд типових проблем, пов'язаних із роботою зовнішніх API, зокрема нестабільність відповіді, перевищення часу очікування та помилки авторизації. Для їх усунення було впроваджено перевірку наявності API-ключів, додано контроль статусу HTTP-відповідей, а також обмеження часу виконання запитів. Крім того, було оптимізовано формування запитів до моделей, що дозволило зменшити кількість некоректних або неповних відповідей.

Окрему увагу приділено тестуванню механізму агрегації, оскільки саме він визначає якість фінального результату. Було встановлено, що при використанні декількох моделей система забезпечує більш стабільну та змістовну відповідь у порівнянні з використанням однієї моделі. Також перевірялась коректність обробки контексту попередніх повідомлень, що дозволяє підтримувати логіку діалогу.

Мінімальні системні вимоги для роботи розробленої системи включають наявність персонального комп'ютера з доступом до мережі Інтернет, встановленого інтерпретатора Python версії 3.10 або новішої, а також необхідних бібліотек, зазначених у файлі залежностей. Рекомендованими параметрами є наявність стабільного інтернет-з'єднання, достатнього обсягу оперативної пам'яті (від 8 ГБ) та сучасного процесора, що забезпечує комфортну роботу з декількома запитами одночасно.

У результаті проведеного тестування та налагодження було забезпечено стабільну роботу системи, коректну взаємодію з API мовних моделей та належну якість сформованих відповідей. Виявлені недоліки були усунені в процесі розробки, що дозволило отримати працездатне програмне рішення, придатне для подальшого використання та розвитку.

3.3 Загальна архітектура системи

У межах даної роботи було розроблено програмну систему, призначену для отримання відповідей від декількох мовних моделей та їх подальшого узагальнення. При проєктуванні основна увага приділялась не лише реалізації базового функціоналу, але й побудові такої структури, яка дозволяє легко змінювати окремі компоненти, додавати нові моделі та адаптувати систему під інші сценарії використання. З цієї причини архітектура була організована за модульним принципом, де кожна частина системи виконує чітко визначену роль, а взаємодія між ними відбувається через зрозумілі інтерфейси. Такий підхід дозволяє уникнути надмірної зв'язаності коду і спрощує подальший розвиток проєкту. Для кращого розуміння взаємодії основних компонентів програмного рішення було побудовано діаграму компонентів, яка відображає загальну структуру системи та взаємозв'язки між її модулями (рис. 3.1).

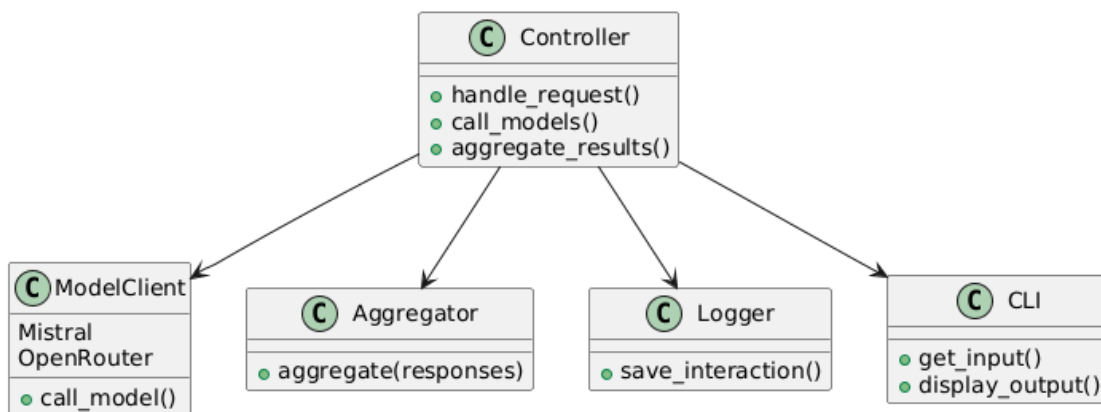


Рисунок 3.1 – Діаграма класів

Структура проекту побудована у вигляді ієрархії директорій, що відображає логіку роботи системи (рис 3.2). Центральною точкою входу є файл `main.py`, який відповідає за запуск застосунку, ініціалізацію основних компонентів і передачу керування у цикл обробки запитів. Важливо, що в цьому файлі відсутня складна бізнес-логіка – він виконує роль координатора, що відповідає за загальний сценарій роботи. Такий підхід дозволяє ізолювати основну логіку в інших модулях і не перевантажувати точку входу зайвими деталями.

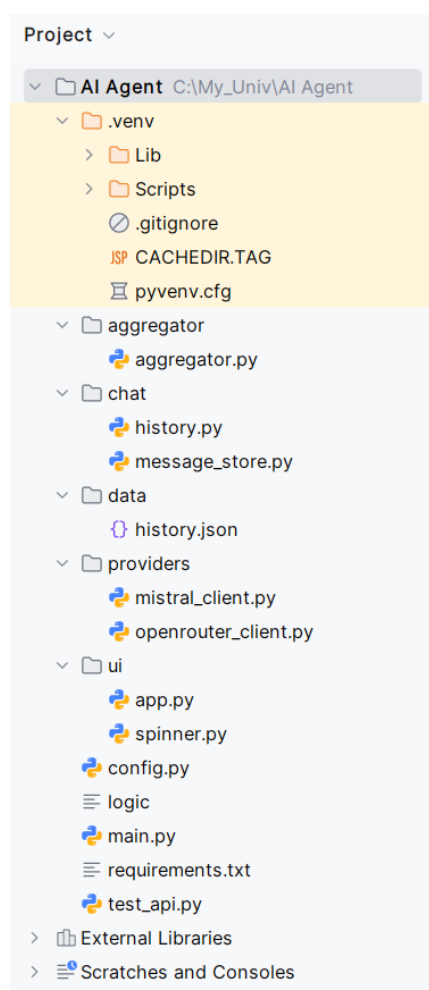


Рисунок 3.2 – Структура програмного проєкту у середовищі PyCharm

Окремо у програмному рішенні використовується файл конфігурації `config.py`, у якому зберігається інформація про доступні мовні моделі та параметри їх виклику. Такий підхід дозволяє зберігати налаштування моделей

ізолюваний від основної логіки програми та спрощує додавання нових API без внесення значних змін у код. Завдяки цьому структура програмного рішення залишається більш гнучкою та зручною для подальшого розширення.

Взаємодія з користувачем реалізована через модуль `chat`, який включає в себе цикл обробки запитів та механізм збереження історії повідомлень. Файл `chat_loop.py` відповідає за послідовну обробку введених користувачем запитів, виклик моделей та передачу результатів на наступні етапи. У свою чергу, `message_store.py` використовується для фіксації історії взаємодій, що дозволяє не лише зберігати попередні відповіді, але й використовувати їх у подальшій роботі. Такий поділ виглядає незначним, але на практиці він дозволяє уникнути змішування логіки обробки запитів і роботи з даними.

Ключовою частиною системи є модуль `providers`, який відповідає за безпосередню взаємодію з API мовних моделей. У його межах реалізовано окремі клієнти для різних сервісів, зокрема `mistral_client.py` та `openrouter_client.py`. Кожен із них інкапсулює специфіку конкретного API, включаючи формування запитів, обробку відповідей та базову обробку помилок. Це дозволяє уніфікувати процес виклику моделей: незалежно від того, який сервіс використовується, система отримує результат у спільному форматі. Такий підхід суттєво спрощує подальшу агрегацію та зменшує залежність від конкретних постачальників API.

Окремо слід виділити модуль `aggregator`, який фактично є центральним елементом усієї системи. Саме тут реалізовано логіку об'єднання відповідей, отриманих від різних моделей. На відміну від простого вибору одного варіанту, агрегатор формує нову відповідь, використовуючи результати кількох моделей як вхідні дані. Такий підхід дозволяє зменшити вплив помилок окремих моделей і отримати більш узгоджений результат. Реалізація цього механізму винесена в окремий файл `aggregator.py`, що дозволяє змінювати алгоритм агрегації незалежно від інших частин системи.

Для взаємодії з користувачем також передбачено окремий модуль `ui`, у якому реалізовано текстовий інтерфейс із використанням бібліотеки `Textual`. У

порівнянні зі звичайним консольним введенням, такий підхід забезпечує більш зручну навігацію, краще відображення інформації та загалом покращує досвід використання системи. При цьому інтерфейс залишається достатньо легким і не потребує складної графічної реалізації.

Збереження результатів роботи організовано через використання файлу `./data/history.json`, який використовується для фіксації всіх взаємодій. Це дозволяє не лише відновлювати попередні сесії, але й аналізувати результати роботи системи, що є важливим на етапі тестування. Фактично цей механізм виконує роль простого сховища даних, яке не потребує окремої бази даних, але при цьому забезпечує достатній рівень функціональності. Структура взаємодії між основними програмними модулями також була формалізована у вигляді UML-діаграми класів (рис. 3.3). Вона демонструє розподіл відповідальності між окремими компонентами та логіку обміну даними між ними.

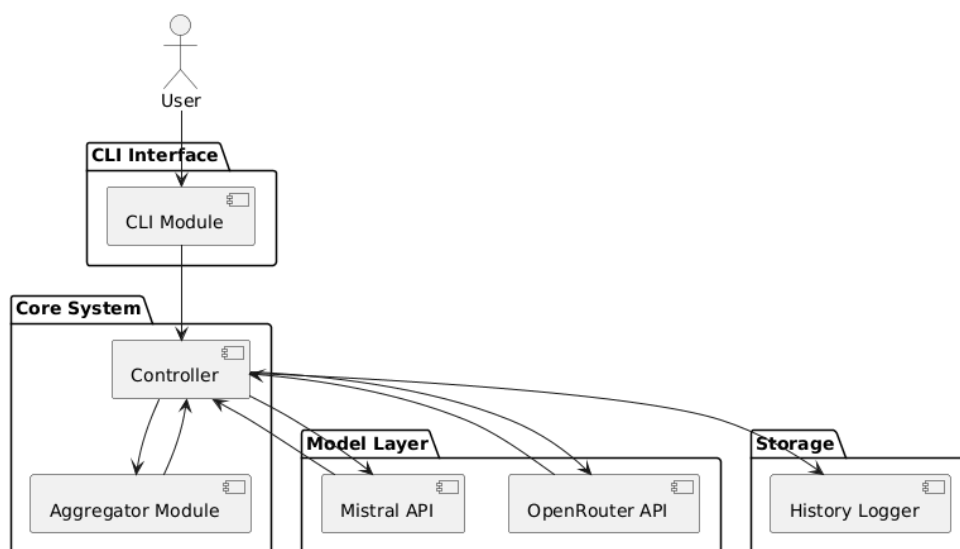


Рисунок 3.3 – Діаграма компонентів

Узагальнюючи, можна сказати, що архітектура системи побудована за принципом розділення відповідальності, де кожен модуль виконує окрему функцію і взаємодіє з іншими через чітко визначені точки. Це дозволяє забезпечити гнучкість, спростити підтримку та створити основу для подальшого розвитку системи. Такий підхід добре відповідає поставленій задачі, оскільки

дозволяє експериментувати з різними моделями та алгоритмами агрегації без суттєвих змін у загальній структурі.

3.4 Реалізація взаємодії з API моделей

Одним із ключових етапів реалізації системи є організація взаємодії з мовними моделями через API. Оскільки у межах даної роботи використовується декілька моделей від різних постачальників, було прийнято рішення розділити логіку взаємодії на окремі клієнтські модулі. Це дозволило ізолювати специфіку кожного сервісу та забезпечити уніфікований підхід до отримання відповідей незалежно від джерела.

У системі реалізовано два основних способи взаємодії з моделями: прямий виклик API сервісу Mistral та використання агрегованого API через платформу OpenRouter. Такий підхід дозволяє поєднати переваги різних моделей та забезпечити гнучкість у їх виборі. Важливо, що незалежно від способу виклику, кожна модель повертає результат у текстовому вигляді, що спрощує подальшу обробку та агрегацію.

Для роботи з API Mistral (ліст. 3.1) було реалізовано окрему функцію, яка інкапсулює процес формування запиту та обробки відповіді. Перед виконанням запиту система отримує API-ключ із змінних середовища, що дозволяє уникнути його жорсткого кодування у програмі та підвищує рівень безпеки. У випадку відсутності ключа генерується помилка, що запобігає некоректній роботі системи.

Лістинг 3.1 – Реалізація виклику Mistral API

```
def ask_mistral(prompt: str) -> str:
    api_key = os.getenv("MISTRAL_API_KEY")
    if not api_key:
        raise RuntimeError("MISTRAL_API_KEY не встановлено")

    client = Mistral(api_key=api_key)

    response = client.chat.complete(
        model="mistral-small-latest",
```

```

        messages=[{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content

```

кінець лістингу 3.1

Як видно з лістингу, функція приймає текстовий запит, формує повідомлення у відповідному форматі та передає його до API. Отримана відповідь обробляється шляхом доступу до структури результату, після чого повертається лише текстова частина. Такий підхід дозволяє мінімізувати обсяг обробки та отримувати результат у зручному для подальшого використання вигляді.

Для роботи з іншими моделями використовується платформа OpenRouter (ліст. 3.2), яка надає єдиний інтерфейс доступу до різних постачальників. У цьому випадку взаємодія реалізована через HTTP-запити з використанням бібліотеки requests. Такий підхід є більш універсальним, але потребує ручного формування заголовків та тіла запиту.

Лістинг 3.2 – Реалізація виклику OpenRoute API

```

def ask_openrouter(prompt: str, model: str) -> str:
    api_key = os.getenv("OPENROUTER_API_KEY")
    if not api_key:
        raise RuntimeError("OPENROUTER_API_KEY не встановлено")

    headers = {
        "Authorization": f"Bearer {api_key}",
        "Content-Type": "application/json",
        "HTTP-Referer": "http://localhost",
        "X-Title": "Bachelor AI Agent"
    }

    payload = {
        "model": model,
        "messages": [{"role": "user", "content": prompt}],
        "temperature": 0.7
    }

    response = requests.post(
        OPENROUTER_URL,
        headers=headers,

```

```

        json=payload,
        timeout=60
    )

    if response.status_code != 200:
        raise RuntimeError(response.text)

    return response.json()["choices"][0]["message"]["content"]

```

кінець лістингу 3.2

У даному випадку функція також приймає текст запиту, але додатково отримує назву моделі як параметр, що дозволяє використовувати різні моделі в межах одного інтерфейсу. Формування запиту відбувається шляхом створення JSON-структури, яка містить параметри генерації та сам запит користувача. Відповідь обробляється аналогічно до попереднього випадку, але з урахуванням формату, який використовується у OpenRouter.

Важливою частиною реалізації є використання конфігураційного файлу, у якому визначено перелік моделей та їх параметри. Це дозволяє динамічно змінювати набір моделей без внесення змін у код функцій виклику. Такий підхід спрощує підтримку системи та робить її більш гнучкою у використанні. Для забезпечення гнучкості системи використовується конфігураційний підхід, при якому перелік моделей та їх параметри визначаються окремо від основної логіки програми (ліст. 3.3).

Лістинг 3.3 – Конфігурація мовних моделей

```

INFO_MODELS = {

    "Mistral": {
        "type": "mistral",
        "model": "mistral-small-latest"
    },
    "LLaMA": {
        "type": "openrouter",
        "model": "meta-llama/llama-3.1-8b-instruct"
    },
    "Gemma": {
        "type": "openrouter",
        "model": "deepseek/deepseek-chat"
    }
}

```

```

    }
}

AGGREGATOR_MODELS = {
    "Mistral-7B": "mistralai/mistral-7b-instruct",
    "LLaMA-8B": "meta-llama/llama-3.1-8b-instruct",
    "Gemma-7B": "deepseek/deepseek-chat"
}

```

кінець лістингу 3.3

Узагальнюючи, можна зазначити, що реалізація взаємодії з API побудована за принципом уніфікації: незалежно від джерела, система отримує відповідь у однаковому форматі. Це створює основу для подальшої агрегації та дозволяє працювати з різними моделями як із єдиним набором джерел інформації.

3.5 Реалізація механізму агрегації відповідей

Однією з ключових особливостей розробленої системи є реалізація механізму агрегації відповідей, який дозволяє об'єднувати результати, отримані від декількох мовних моделей, у єдину узгоджену відповідь. На відміну від простого вибору одного варіанту, у даній системі використовується підхід, що базується на формуванні нового результату шляхом аналізу декількох незалежних відповідей.

Реалізація цього механізму винесена в окремий модуль, що забезпечує незалежність логіки агрегації від інших частин системи (ліст. 3.4). Основна функція, яка виконує об'єднання результатів, приймає як вхідні параметри запит користувача, набір відповідей від моделей, назву моделі-агрегатора, а також додатковий контекст попередніх повідомлень. Такий набір параметрів дозволяє враховувати не лише поточний запит, але й попередню історію взаємодії.

Лістинг 3.4 – Реалізація механізму агрегації відповідей

```

def aggregate_answers(

```

```

    user_question: str,
    answers: dict,
    aggregator_model: str,
    context: str = ""
) -> str:
    prompt = (
        "You are an AI assistant that maintains conversational
context.\n"
        ...
    )

    if context:
        prompt += context + "\n"

    prompt += f"Current question:\n{user_question}\n\n"
    prompt += "Answers from different models:\n\n"

    for name, answer in answers.items():
        prompt += f"[{name}]: {answer}\n\n"

    prompt += (
        "Combine the answers into one clear, concise, and accurate
response."
    )

    return ask_openrouter(prompt, aggregator_model)

```

кінець лістингу 3.4

Як видно з наведеного фрагменту коду, основою механізму агрегації є формування спеціального текстового запиту (prompt), який передається до окремої мовної моделі. У цьому запиті явно задається роль моделі, правила формування відповіді, а також обмеження щодо формату виводу. Особливу увагу приділено забезпеченню узгодженості мови відповіді: модель отримує інструкцію використовувати ту ж мову, що і користувач, що дозволяє уникнути небажаного перемикання мов.

Важливою складовою є включення контексту попередніх повідомлень. Якщо такий контекст наявний, він додається до запиту, що дозволяє моделі враховувати попередню інформацію і формувати більш цілісну відповідь. Це особливо корисно у випадках, коли запити користувача пов'язані між собою.

Далі до запиту додається поточне питання користувача, після чого формується блок відповідей від різних моделей. Кожна відповідь маркується

відповідною назвою, що дозволяє агрегатору розрізняти джерела інформації. Такий підхід дає змогу моделі оцінити різні варіанти відповіді та сформуванати узагальнений результат.

Завершальним етапом є передача сформованого запиту до моделі-агрегатора через API. У даному випадку використовується той самий механізм виклику, що і для інших моделей, що забезпечує уніфікованість системи. Отриманий результат повертається як фінальна відповідь користувачу.

Таким чином, запропонований механізм агрегації базується не на формальних правилах або алгоритмах ранжування, а на використанні можливостей мовних моделей для узагальнення інформації. Це дозволяє отримувати більш гнучкі та змістовні результати, особливо у випадках, коли відповіді окремих моделей відрізняються між собою.

3.6 Реалізація користувацького інтерфейсу

У процесі розробки системи окрему увагу було приділено організації взаємодії з користувачем, оскільки саме інтерфейс визначає зручність практичного використання програмного рішення. На відміну від стандартних консольних застосунків, де взаємодія обмежується введенням тексту та виводом відповіді, у даній системі було реалізовано розширений текстовий інтерфейс, який дозволяє зробити роботу більш наочною та керованою (рис. 3.4).

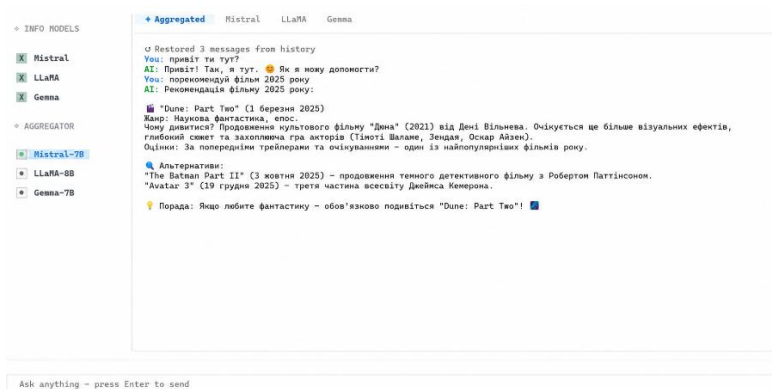


Рисунок 3.4 – Робота текстового інтерфейсу користувача

Основою інтерфейсу є поєднання класичного CLI та бібліотеки Textual, що дозволяє створювати інтерактивні елементи без використання графічного середовища. Такий підхід був обраний свідомо, оскільки він не потребує значних ресурсів, легко запускається у стандартному середовищі розробки та водночас забезпечує більш структуроване відображення інформації. Реалізація інтерфейсу винесена в окремий модуль, що дозволяє ізолювати логіку взаємодії з користувачем від основної частини системи.

У процесі роботи користувач вводить запит, після чого система відображає стан обробки та поступово формує відповідь. Для покращення сприйняття було реалізовано індикатор виконання (spinner), який сигналізує про те, що запит обробляється. Це особливо важливо з урахуванням того, що взаємодія з API може займати певний час і без зворотного зв'язку користувач не розумітиме, чи система працює коректно. Відображення результатів також організовано таким чином, щоб користувач міг бачити як окремі відповіді моделей, так і фінальний узагальнений результат. Це дозволяє оцінити різницю між моделями та зрозуміти, як саме формується кінцева відповідь. Такий підхід є корисним не лише з точки зору користування, але й під час тестування системи.

Крім цього, інтерфейс враховує можливість повторних запитів та використання історії повідомлень. Завдяки цьому користувач може працювати із системою у форматі діалогу, а не як з окремими незалежними запитами. Це наближує роботу програми до поведінки сучасних чат-асистентів.

У цілому реалізований інтерфейс не перевантажений зайвими елементами, але при цьому забезпечує достатній рівень зручності для повсякденного використання. Обраний підхід дозволяє поєднати простоту реалізації з практичною ефективністю, що є важливим для систем такого типу.

3.7 Організація збереження даних

У процесі розробки системи було реалізовано механізм збереження даних, який забезпечує фіксацію історії взаємодії користувача з програмою. Основною

метою цього компонента є збереження запитів, відповідей окремих моделей та фінального результату агрегації, що дозволяє аналізувати роботу системи, відстежувати її поведінку та використовувати попередній контекст у подальшій роботі (ліст. 3.5). На відміну від складних рішень із використанням баз даних, у даному випадку було обрано простий підхід із записом інформації у файл формату JSON, що є достатнім для задач, які вирішуються у межах даної роботи (рис. 3.5).

Лістинг 3.5 – Збереження історії взаємодій

```
def save_interaction(
    question: str,
    answers: dict,
    final_answer: str,
    aggregator: str,
) -> None:

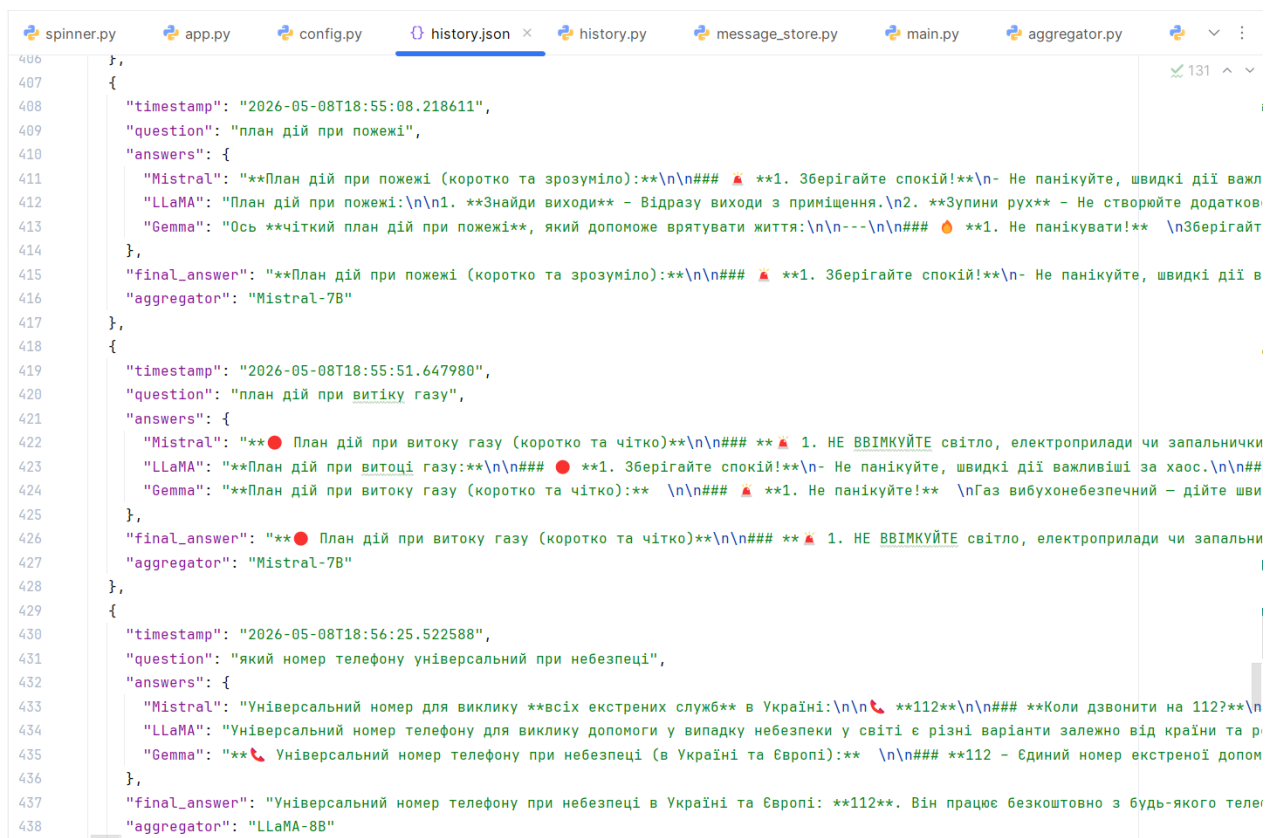
    record = {
        "timestamp": datetime.now().isoformat(),
        "question": str(question),
        "answers": {k: str(v) for k, v in answers.items()},
        "final_answer": str(final_answer),
        "aggregator": str(aggregator),
    }

    if HISTORY_FILE.exists():
        history = json.loads(HISTORY_FILE.read_text(encoding="utf-8"))
    else:
        history = []

    history.append(record)

    HISTORY_FILE.write_text(
        json.dumps(history, ensure_ascii=False, indent=2),
        encoding="utf-8",
    )
```

кінець лістингу 3.5



```

406 },
407 {
408     "timestamp": "2026-05-08T18:55:08.218611",
409     "question": "план дій при пожежі",
410     "answers": {
411         "Mistral": "**План дій при пожежі (коротко та зрозуміло):**\n\n## 🚒 **1. Зберігайте спокій!**\n- Не панікуйте, швидкі дії важл
412         "LLaMA": "План дій при пожежі:\n\n1. **Знайди виходи** - Відразу виходи з приміщення.\n2. **Зупини рух** - Не створюйте додатков
413         "Gemma": "Ось **чіткий план дій при пожежі**, який допоможе врятувати життя:\n\n---\n\n## 🧯 **1. Не панікувати!**\n \nЗберігайт
414     },
415     "final_answer": "**План дій при пожежі (коротко та зрозуміло):**\n\n## 🚒 **1. Зберігайте спокій!**\n- Не панікуйте, швидкі дії в
416     "aggregator": "Mistral-7B"
417 },
418 {
419     "timestamp": "2026-05-08T18:55:51.647980",
420     "question": "план дій при витіку газу",
421     "answers": {
422         "Mistral": "**🔴 План дій при витокі газу (коротко та чітко):**\n\n## 🚒 1. НЕ ВВИМКУЙТЕ світло, електроприлади чи запальнички
423         "LLaMA": "**План дій при витокі газу:**\n\n## 🔴 **1. Зберігайте спокій!**\n- Не панікуйте, швидкі дії важливіші за хаос.\n\n##
424         "Gemma": "**План дій при витокі газу (коротко та чітко):**\n\n \n\n## 🚒 **1. Не панікуйте!**\n \nГаз вибухонебезпечний – дійте шви
425     },
426     "final_answer": "**🔴 План дій при витокі газу (коротко та чітко):**\n\n## 🚒 1. НЕ ВВИМКУЙТЕ світло, електроприлади чи запальни
427     "aggregator": "Mistral-7B"
428 },
429 {
430     "timestamp": "2026-05-08T18:56:25.522588",
431     "question": "який номер телефону універсальний при небезпеці",
432     "answers": {
433         "Mistral": "Універсальний номер для виклику **всіх екстрених служб** в Україні:\n\n📞 **112**\n\n## **Коли дзвонити на 112?**\n
434         "LLaMA": "Універсальний номер телефону для виклику допомоги у випадку небезпеки у світі є різні варіанти залежно від країни та р
435         "Gemma": "**📞 Універсальний номер телефону при небезпеці (в Україні та Європі):**\n\n \n\n## 📞 **112 - Єдиний номер екстреної допом
436     },
437     "final_answer": "Універсальний номер телефону при небезпеці в Україні та Європі: **112**. Він працює безкоштовно з будь-якого теле
438     "aggregator": "LLaMA-8B"

```

Рисунок 3.5 – Приклад збереження історії взаємодій у форматі JSON

Збереження даних реалізовано у директорії data, де файл history.json виступає у ролі сховища всієї історії взаємодій. Кожен запис містить текст запиту користувача, відповіді від різних моделей, а також підсумкову відповідь після агрегації. Така структура дозволяє зручно працювати з даними, оскільки JSON є універсальним форматом, який легко обробляється засобами мови Python. Окрім цього, використання файлового підходу спрощує реалізацію та не потребує налаштування додаткових сервісів, що є доцільним для локального застосунку.

Для роботи з історією використовується окремий модуль, який відповідає за запис нових даних та їх зчитування. Це дозволяє ізолювати логіку збереження від інших частин системи та уникнути дублювання коду. У процесі реалізації було враховано можливість поступового накопичення даних, що дає змогу зберігати повну історію роботи без втрати попередніх результатів.

З практичної точки зору реалізований механізм збереження даних виконує одразу декілька функцій: забезпечує базове логування роботи системи, дозволяє

аналізувати відповіді моделей та створює передумови для подальшого розширення, наприклад використання контексту в межах діалогу. Таким чином, навіть при відносно простій реалізації цей компонент відіграє важливу роль у загальній архітектурі системи та підвищує її практичну цінність.

ВИСНОВКИ

У кваліфікаційній роботі бакалавра вирішено актуальне завдання розробки програмного рішення для отримання консолідованої відповіді від декількох ШІ-моделей із використанням мови програмування Python. У ході виконання роботи було досягнуто поставлену мету та виконано всі визначені завдання.

У першому розділі проведено аналіз сучасного стану розвитку великих мовних моделей, особливостей їх функціонування та способів взаємодії через API. Досліджено проблему неузгодженості відповідей різних мовних моделей, проаналізовано існуючі підходи до агрегації результатів, зокрема ансамблеві методи, підхід self-consistency, багатоагентні системи та методи ранжування відповідей. На основі проведеного аналізу обґрунтовано доцільність створення програмного рішення, яке забезпечує отримання узагальненої відповіді на основі результатів декількох моделей.

У другому розділі визначено функціональні та нефункціональні вимоги до програмного забезпечення. Сформовано вимоги щодо введення користувацьких запитів, взаємодії з кількома мовними моделями, обробки та агрегації відповідей, збереження історії взаємодій і забезпечення надійної роботи системи. Проведено аналіз можливих технологічних рішень та обґрунтовано вибір мови програмування Python, бібліотек asyncio, requests та httpx, а також сервісів OpenRouter і Mistral для реалізації доступу до мовних моделей. Спроектовано архітектуру програмного забезпечення з урахуванням вимог масштабованості, модульності та можливості подальшого розширення функціональності.

У третьому розділі виконано практичну реалізацію програмного рішення. Розроблено систему, яка забезпечує автоматизовану взаємодію з декількома мовними моделями через API, отримання відповідей, їх обробку та подальшу агрегацію. Реалізовано механізм формування узагальненого запиту до моделі-агрегатора, який використовує відповіді декількох моделей для створення єдиного підсумкового результату. Також реалізовано механізм

збереження історії взаємодій користувача, що дозволяє накопичувати та аналізувати результати роботи системи.

Під час розробки особливу увагу приділено забезпеченню надійності функціонування системи. Реалізовано механізми обробки виключних ситуацій, контролю коректності API-викликів, перевірки статусів HTTP-відповідей та контролю часу очікування. Це дозволило забезпечити стабільну роботу програмного рішення навіть у випадках часткової недоступності зовнішніх сервісів або виникнення помилок під час обміну даними.

Для перевірки працездатності програмного забезпечення проведено функціональне тестування основних компонентів системи. Було протестовано роботу модулів взаємодії з API мовних моделей, механізму агрегації відповідей, підсистеми збереження історії взаємодій та консольного інтерфейсу користувача. Перевірка виконувалась на різних типах запитів, включаючи інформаційні, уточнюючі та складні багатокomпонентні запити. Також проведено тестування роботи системи в умовах часткової відмови окремих сервісів, що підтвердило коректність реалізованих механізмів обробки помилок та можливість формування підсумкової відповіді навіть за відсутності результатів від окремих моделей.

Результати тестування показали коректність функціонування всіх основних компонентів програмного рішення, стабільність взаємодії із зовнішніми сервісами та ефективність запропонованого механізму агрегації відповідей. Розроблена система забезпечує отримання більш повної та узгодженої інформації порівняно з використанням окремих мовних моделей, а її архітектура дозволяє у майбутньому розширювати перелік підтримуваних моделей та вдосконалювати алгоритми формування консолідованих відповідей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Vaswani A. et al. Attention Is All You Need. URL: <https://arxiv.org/abs/1706.03762> (дата звернення: 10.04.2026).
2. Brown T. et al. Language Models are Few-Shot Learners. URL: <https://arxiv.org/abs/2005.14165> (дата звернення: 15.04.2026).
3. Bender E. et al. On the Dangers of Stochastic Parrots. URL: <https://arxiv.org/abs/2103.00020> (дата звернення: 18.04.2026).
4. OpenAI API Documentation. URL: <https://platform.openai.com/docs/> (дата звернення: 15.11.2023).
5. Python Requests Documentation. URL: <https://requests.readthedocs.io/en/latest/> (дата звернення: 18.04.2026).
6. RFC 8259 – JSON Format. URL: <https://www.rfc-editor.org/rfc/rfc8259> (дата звернення: 19.04.2026).
7. Mistral AI Documentation. URL: <https://docs.mistral.ai/> (дата звернення: 20.04.2026).
8. OpenRouter Documentation. URL: <https://openrouter.ai/docs/quickstart> (дата звернення: 22.04.2026).
9. Zhao W. et al. A Survey of Large Language Models. URL: <https://arxiv.org/abs/2303.18223> (дата звернення: 23.04.2026).
10. Liu X. et al. Pre-train, Prompt, and Predict. URL: <https://arxiv.org/abs/2107.13586> (дата звернення: 25.04.2026).
11. Tree of Thoughts: Deliberate Problem Solving with LLMs. URL: <https://arxiv.org/abs/2305.10601> (дата звернення: 26.04.2026).
12. Wang X. et al. Self-Consistency Improves Chain of Thought Reasoning. URL: <https://arxiv.org/abs/2203.11171> (дата звернення: 26.04.2026).
13. Park J. et al. Generative Agents. URL: <https://arxiv.org/abs/2304.03442> (дата звернення: 26.04.2026).
14. Karpas E. et al. MRKL Systems. URL: <https://arxiv.org/abs/2205.00445> (дата звернення: 27.04.2026).

15. Python Documentation (asyncio). URL: <https://docs.python.org/3/library/asyncio.html> (дата звернення: 29.04.2026).
16. HTTPX Documentation. URL: <https://www.python-httpx.org/> (дата звернення: 29.04.2026).
17. Mozilla HTTP Overview. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP> (дата звернення: 30.04.2026).
18. Async IO in Python. URL: <https://realpython.com/async-io-python/> (дата звернення: 01.05.2026).