Вінницький національний технічний університет
Факультут інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук

Бакалаврська кваліфікаційна робота на тему:

РОЗРОБКА ЧАТ-БОТУ «АСИСТЕНТ ДЛЯ РОЗРОБНИКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

Виконав: студент 4 курсу, групи 1КН-21б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напрямку підготовки, спеціальності)
Крикливий О. В.
(прізвище та ініціали)

Керівник: к.т.н., ст. вик.
Петришин С. І.
(прізвище та ініціали)

«04» червня 2025 р.

Рецензент: к.т.н., доц. кафедри АПТ
Варчук І. В.
(прізвище та ініціали)

«05» червня 2025 р.

Допущено до захисту
Зав. кафедри д.т.н., проф. Яровий А.А.
(прізвище та ініціали)
«06» червня 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Яровий А.А.
«05» травня 2025 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Крикливому Олексію Володимировичу

1. Тема роботи: Розробка чат-боту "Асистент для розробника програмного забезпечення"

2. Керівник роботи: Петришин Сергій Іванович
Затверджені наказом вищого навчального закладу «20» 03 2025 року №97

2. Термін подання студентом роботи 30 травня 2025 р.

3. Вихідні дані до роботи :
Мова програмування – об'єктно-орієнтована;
Підтримка загального інтерфейсу мессенджера;
Кількість одночасного використання – не більше 100 чол..

4. Зміст текстової частини (перелік питань, які потрібно розробити):
проаналізувати існуючі технології, методи і моделі по обробці тексту у контексті LLM;
розробити алгоритм використання роботи модуля для генерації тексту;
спроектувати та реалізувати інтерфейс мессенджера.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень):
Модель роботи програмного модуля
Схема алгоритму роботи модуля для генерації тексту
Діаграми класів та компонентів проекту
Загальний вигляд інтерфейсу користувача
Приклад роботи програми

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	Петришин С.І., к.т.н., ст. вик.		

7. Дата видачі завдання 05 травня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз існуючих чат-ботів та технологій	05.05.25	07.05.25	
2	Проектування та інтеграція чат-бота для розробника	08.05.25	11.05.25	
3	Реалізація, тестування та оцінка ефективності чат-бота	12.05.25	15.05.25	
4	Висновки та аналіз отриманих результатів	16.05.25	26.05.25	
5	Оформлення додатків	27.05.25	29.05.25	
6	Розробка інструкції користувача	30.05.25	01.06.25	
7	Оформлення матеріалів до захисту БКР	02.06.25	04.06.25	

Студент _____ (підпис) **Крикливий О. В.** (прізвище та ініціали)

Керівник проекту (роботи) _____ (підпис) **Петришин С.І.** (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 104 сторінок формату А4, на яких є 28 рисуноків та 3 таблиці, список використаних джерел містить 20 найменувань.

Бакалаврська кваліфікаційна робота є присвяченою створенню чат-боту «Асистент для розробника програмного забезпечення». В даній бакалаврській роботі розроблено функціонал для обробки, аналізу, збереження даних та засобів для підтримки функціонування чат-боту. Розроблено інтерфейс функціонуючого месенджера на платформі Qt для ведення діалогу зі штучним інтелектом, охарактеризовано базові та другорядні вимоги для створення подібних чатів.

Ключові слова: Чат-бот, написання програмного коду, штучний інтелект, API.

ABSTRACT

The bachelor's qualification work consists of 104 pages of A4 format, on which there are 28 figures and 3 tables, the list of used sources contains 20 names.

The bachelor's qualification work is dedicated to the creation of a chatbot "Assistant for a software developer". In this bachelor's thesis, the functionality for processing, analyzing, storing data and tools to support the functioning of the chatbot has been developed. The interface of a functioning messenger on the Qt platform for conducting a dialogue with artificial intelligence has been developed, the basic and secondary requirements for creating such chats have been characterized.

Keywords: Chatbot, writing software code, artificial intelligence, API.

ЗМІСТ

ВСТУП

Актуальність теми. Розробка чат-бота «Асистент для розробника програмного забезпечення» є актуальною темою, оскільки чат-боти стають невід'ємною частиною сучасних бізнес-процесів, автоматизуючи взаємодію з користувачами та підвищуючи ефективність обслуговування. Використання таких ботів у сфері розробки програмного забезпечення може значно покращити продуктивність та зручність роботи розробників [1].

Чат-боти можуть виконувати різноманітні завдання, такі як [2]:

- Збір вимог: допомога у взаємодії із зацікавленими сторонами для виявлення потреб та бажаних функцій.
- Дизайн взаємодії з користувачем: надання відгуків щодо концепцій UX, каркасів або макетів, сприяючи покращенню зручності використання та залучення користувачів.
- Обробка природної мови (NLP): створення чат-ботів, віртуальних помічників або програм з голосовою підтримкою для обробки запитів користувачів та надання відповідей.
- Тестування та гарантія якості: імітація взаємодії користувача, генерація тестових прикладів для автоматизації процесів тестування та виявлення потенційних проблем.
- Обробка помилок і усунення їх: надання можливих причин або рішень для вирішення проблем, рекомендації щодо методів налагодження або змін коду.
- Генерація документації: автоматизація процесу створення документації до програми, що економить час і зусилля при створенні вичерпної документації.
- Безперервна інтеграція та розгортання (CI/CD): надання рекомендацій щодо практик та стратегій CI/CD для забезпечення ефективного та надійного розгортання програми.

- Служби підтримки та служби підтримки: відповідь на поширені запитання, надання вказівок щодо усунення несправностей або загальної інформації про функції програми.

Розробка чат-бота на основі моделей штучного інтелекту, дозволяє створювати інструменти, здатні розуміти контекст та генерувати відповіді, схожі на людські. Це сприяє більш природній та ефективній взаємодії між розробниками та системою.

В Україні розробка чат-ботів є перспективним напрямком, що викликає все більший інтерес серед розробників, бізнесменів та користувачів. Чат-боти стають необхідною складовою для автоматизації бізнес-процесів та покращення обслуговування клієнтів [3].

Таким чином, розробка чат-бота «Асистент для розробника програмного забезпечення» є актуальною та перспективною темою, яка може значно покращити ефективність роботи розробників та сприяти автоматизації процесів у сфері розробки програмного забезпечення.

Мета дослідження – підвищення якості роботи розробника програмного забезпечення з використанням чат-боту.

Об'єкт дослідження – процес написання програмного коду з використанням чат-боту.

Предмет дослідження – програмні засоби для написання програмного коду з використанням чат-боту.

Для досягнення поставленої мети потрібно виконати наступні **задачі дослідження**:

- Аналіз існуючих чат-ботів;
- Обґрунтування роботи чат-боту;
- Дослідження процесу написання коду;
- Програмна реалізація чат-боту;
- Тестування та оцінка ефективності чат-бота;
- Розробка інструкції користувача.

Результати дослідження впроваджено в роботу ТОВ «ІТІ».

1 АНАЛІЗ ІСНУЮЧИХ ЧАТ-БОТІВ ТА ТЕХНОЛОГІЙ

1.1 Огляд сучасних чат-ботів та їх особливості

Чат-бот – це спеціалізована комп'ютерна програма, що виконує роль співрозмовника або віртуального помічника, і здатна імітувати природне людське спілкування. Така програма зазвичай працює за наперед заданими сценаріями, алгоритмами та логічними структурами, які дозволяють їй аналізувати вхідні запити користувачів та видавати відповідні відповіді. Завдяки цьому чат-бот може оперативно реагувати на стандартні питання та завдання, забезпечуючи швидку підтримку користувачів навіть без участі живого оператора [4].

За допомогою інтегрованих технологій штучного інтелекту та обробки природної мови, сучасні чат-боти здатні не лише відтворювати заздалегідь прописані діалоги, але й адаптуватися до змін у запитах користувачів, розпізнавати контекст і навіть навчатися на основі попереднього досвіду взаємодії. Така функціональність дозволяє їм брати на себе значну частину рутинних завдань, що традиційно виконуються операторами підтримки, що, в свою чергу, оптимізує робочі процеси та підвищує ефективність надання сервісу [4].

За даними статистики, майже 40% користувачів по всьому світу віддають перевагу спілкуванню з чат-ботами, оскільки це дозволяє отримати миттєві відповіді без очікування на живого співрозмовника. Цей тренд посилюється завдяки зростанню оцифровування в різних індустріях, зокрема в охороні здоров'я та роздрібній торгівлі, де інтеграція чат-ботів сприяє підвищенню якості обслуговування та зменшенню часу на вирішення запитів клієнтів [4].

Крім стандартної функції надання інформації, чат-боти можуть виконувати низку додаткових завдань, таких як підтримка клієнтів при

оформленні покупок, рекомендації щодо вибору товару або послуг, а також активація спеціальних акцій та програм лояльності. Такі функції дозволяють не лише оптимізувати діалог з користувачем, але й стимулювати продажі та підвищувати загальний рівень задоволеності клієнтів. Завдяки цьому підприємства отримують можливість ефективніше управляти своїми бізнес-процесами в умовах високої конкуренції на ринку [4].

Серед найпопулярніших чат-ботів зі штучним інтелектом можна навести: ChatGPT, Claude, DeepSeek, Gemini, Mistral та Llama. Порівняльна характеристика сучасних чат-ботів зі штучним інтелектом наведена в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика чат-ботів

Чат-бот	Основні переваги	Найкраще підходить для	Обмеження	Особливості
ChatGPT (OpenAI)	Універсальність, мультимодальні можливості, найкраща структурована подача інформації, реальна обізнаність	Повсякденних завдань, генерації зображень, голосової взаємодії, всебічного використання ШІ	Може давати надмірно детальні відповіді	Маркетплейс користувацьких GPT, голосова взаємодія, пошук в інтернеті
Claude (Anthropic)	Природний стиль письма, потужна робота з кодом, аналітичний підхід	Розробників, письменників, аналітиків, творчої роботи	Обмеженіший функціонал порівняно з ChatGPT	Візуалізація коду в реальному часі через Artifacts, більш глибокий аналіз
DeepSeek	Найкращі здібності до глибокого міркування, високі математичні здібності	Складних математичних задач, критичного аналізу, наукових досліджень	Менш універсальний, повільніший	Перевершує інші моделі в глибоких логічних завданнях
Gemini (Google)	Лаконічність, технічна точність, зручність використання	Новачків, технічних запитів, аналізу даних	Менша креативність порівняно з ChatGPT	Інтеграція з екосистемою Google
Mistral	Економічна ефективність, відкритий код, ефективне використання ресурсів	Розробки з обмеженим бюджетом, кодування, налаштування під специфічні завдання	Менш потужний у складних завданнях	Висока продуктивність при менших обчислювальних ресурсах
Llama (Meta)	Гнучкість, контроль, можливість тонкого налаштування	Дослідників, розробників, спеціалізованих завдань	Менш доступний для пересічних користувачів	Відкритий код, можливість розгортання локально

ChatGPT від OpenAI залишається найуніверсальнішим інструментом на ринку ШІ. Завдяки інтеграції технологій генерації зображень, голосової взаємодії та пошуку в інтернеті, він пропонує користувачам повноцінну екосистему можливостей. Його перевага полягає в збалансованому поєднанні потужності, доступності та різноманіття функцій, що робить його оптимальним вибором для загального використання [5].

Claude від Anthropic виділяється своїм природнішим стилем комунікації та виняткової здатності до творчої роботи. Його відповіді часто виглядають більш людськими, а здатність працювати з кодом через функцію Artifacts робить його особливо цінним для розробників. Claude краще підходить для користувачів, які потребують глибини, а не широти можливостей [5].

DeepSeek займає особливу нішу як чемпіон з глибокого міркування та математичних здібностей. Хоч він може поступатися іншим ботам у швидкості та універсальності, його здатність до критичного аналізу та розв'язання складних логічних задач є неперевершеною, що робить його незамінним для наукових та аналітичних завдань [6].

Gemini від Google пропонує збалансований підхід з акцентом на технічну точність та лаконічність. Його інтерфейс розроблений з думкою про доступність для новачків, при цьому він зберігає високу ефективність у технічних завданнях. Інтеграція з екосистемою Google додає додаткові переваги для користувачів, які вже залучені до цієї екосистеми [7].

Французький проект Mistral AI вирізняється своєю економічною ефективністю та підходом відкритого коду. Він демонструє вражаючу продуктивність при менших обчислювальних ресурсах, що робить його привабливим для розробників з обмеженим бюджетом. Його сильні сторони у генерації коду роблять його популярним серед технічної аудиторії [8].

Llama від Meta пропонує найбільшу гнучкість серед усіх моделей завдяки відкритому коду та можливості локального розгортання. Це робить його ідеальним вибором для дослідників та розробників, які потребують

повного контролю над моделлю та можливості її тонкого налаштування під специфічні завдання [9].

1.2 Архітектура штучного інтелекту чат-ботів

Архітектура штучного інтелекту чат-ботів складається з декількох ключових компонентів, які забезпечують ефективну обробку запитів користувачів та генерацію відповідей. Одним із основних елементів є модель трансформера, представлена дослідниками Google у 2017 році. Ця модель використовує механізм багатоголової уваги для аналізу відносин між словами в реченні, що дозволяє чат-ботам краще розуміти контекст запитів. У процесі обробки запиту, текст перетворюється на числові представлення, відомі як токени. Кожен токен кодується у вектор за допомогою таблиці векторних представлень слів (word embeddings). На кожному шарі трансформера токени аналізуються в межах контекстного вікна за допомогою механізму уваги, що дозволяє підсилювати значущі токени та зменшувати вагу менш важливих [10].

Ілюстрація трансформера від Google зображено на рисунку 1.1.

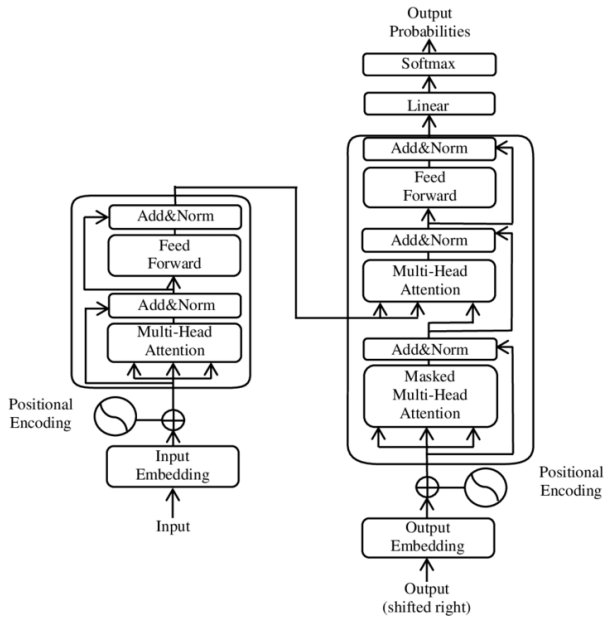

Рисунок 1.1 – Ілюстрація трансформера від Google

Серед основних компонентів архітектури чат-ботів виділяють адаптивний шар взаємодії, генеративний рушій ШІ, управління діалогом, інтеграцію з бекендом та систему спостереження та навчання. Адаптивний шар взаємодії відповідає за динамічне відображення елементів інтерфейсу залежно від контексту розмови, наприклад, відображення кнопок або форм при обговоренні певних тем [11].

Генеративний рушій ШІ використовує LLM (великі мовні моделі) для аналізу та генерації відповідей, враховуючи коротко- та довгостроковий контекст, що зберігається у зовнішніх базах даних. Для запобігання галюцинаціям модель може використовувати прив'язку до знань та механізми самоперевірки.

Управління діалогом забезпечує контроль за ходом розмови, використовуючи гібридний підхід, що поєднує вільний режим, де модель керує відкритими обговореннями, та керований режим, де використовуються автоматизовані процеси для регульованих робочих процесів.

Інтеграція з бекендом дозволяє чат-боту взаємодіяти з реальними даними та виконувати дії, такі як отримання інформації з зовнішніх API або баз даних. Це забезпечує актуальність та точність відповідей.

Система спостереження та навчання відповідає за постійне вдосконалення чат-бота шляхом відстеження показників, таких як релевантність контексту та частота використання резервних відповідей. Це дозволяє виявляти та виправляти аномалії в поведінці моделі.

Використання модульної архітектури в розробці чат-ботів дозволяє кожному компоненту виконувати свою специфічну функцію, що сприяє ефективності та надійності системи [12].

1.3 Методи обробки природної мови у чат-ботах

У чат-ботах зі штучним інтелектом використовуються різноманітні методи обробки природної мови (NLP), що дозволяють їм розуміти та генерувати людську мову. Одним із ключових методів є токенізація, яка розбиває текст на менші одиниці, звані токенами, такі як слова чи фрази. Це забезпечує основу для подальшого аналізу, включаючи визначення частин мови та розпізнавання іменованих сутностей [13].

Визначення частин мови (POS-тегування) передбачає присвоєння кожному слову в реченні відповідної граматичної категорії, наприклад, іменник, дієслово чи прикметник. Це допомагає зрозуміти синтаксичну структуру речення та є важливим для завдань, таких як семантичний аналіз і вилучення інформації.

Розпізнавання іменованих сутностей (NER) спрямоване на ідентифікацію та класифікацію ключових елементів у тексті, таких як імена людей, місця чи організації. Це відіграє важливу роль у системах запитань та відповідей, а також у рекомендаційних системах.

Аналіз настроїв дозволяє визначити емоційний тон тексту, тобто чи є він позитивним, негативним або нейтральним. Це корисно для розуміння ставлення користувача та відповідного налаштування відповіді чат-бота.

Семантичний парсинг полягає в перетворенні природної мови у формальне представлення, зрозуміле для машини, що дозволяє чат-боту точно інтерпретувати значення висловлювань користувача та генерувати відповідні відповіді.

Використання векторних представлень слів дозволяє перетворювати слова в числові вектори, де слова з подібним значенням мають близькі векторні представлення. Це сприяє кращому розумінню контексту та семантики слів у тексті [14].

Контекстуальне управління є важливим для підтримки зв'язності діалогу, оскільки дозволяє чат-боту запам'ятовувати попередні повідомлення та використовувати цю інформацію для розуміння поточних запитів.

Застосування цих методів NLP дозволяє чат-ботам ефективно взаємодіяти з користувачами, розуміти їхні запити та надавати релевантні відповіді, що покращує загальний досвід спілкування.

1.4 Висновок до розділу 1

Сучасний ринок чат-ботів зі штучним інтелектом представлений різноманітними рішеннями, кожне з яких має свої унікальні переваги та недоліки. Вибір оптимального чат-бота залежить від конкретних потреб користувача:

- Для повсякденного використання та різноманітних завдань: ChatGPT
- Для творчої роботи та глибокого аналізу: Claude
- Для складних математичних та логічних задач: DeepSeek
- Для новачків та технічної роботи: Gemini
- Для економічно ефективних рішень: Mistral
- Для дослідників, які потребують контролю та налаштування: Llama

Архітектура сучасних чат-ботів зі штучним інтелектом базується на моделі трансформера, яка використовує механізм багатоголової уваги для аналізу відносин між словами. Ефективність чат-ботів забезпечується взаємодією кількох ключових компонентів:

- Адаптивний шар взаємодії, що динамічно налаштовує інтерфейс відповідно до контексту розмови.
- Генеративний рушій ШІ, заснований на великих мовних моделях (LLM), що аналізує та генерує відповіді з урахуванням контексту.
- Система управління діалогом, що контролює хід розмови через гібридний підхід вільного та керованого режимів.

- Інтеграція з бекендом для взаємодії з реальними даними та виконання дій.
- Система спостереження та навчання, що забезпечує постійне вдосконалення чат-бота.

Модульна архітектура дозволяє кожному компоненту виконувати свою специфічну функцію, що підвищує загальну ефективність та надійність системи.

Чат-боти зі штучним інтелектом використовують різноманітні методи обробки природної мови для розуміння та генерації людської мови:

- Токенізація розбиває текст на менші одиниці (токени), що створює основу для подальшого аналізу.
- POS-тегування присвоює кожному слову відповідну граматичну категорію, допомагаючи зрозуміти синтаксичну структуру.
- Розпізнавання іменованих сутностей (NER) ідентифікує ключові елементи в тексті, такі як імена, місця або організації.
- Аналіз настроїв визначає емоційний тон тексту, що дозволяє адаптувати відповіді до емоційного стану користувача.
- Семантичний парсинг перетворює природну мову у формальне представлення, зрозуміле для машини.
- Векторні представлення слів сприяють кращому розумінню контексту та семантики.
- Контекстуальне управління забезпечує зв'язність діалогу через запам'ятовування попередніх повідомлень.

Комплексне застосування цих методів NLP дозволяє чат-ботам ефективно взаємодіяти з користувачами, точно розуміти їхні запити та надавати релевантні відповіді, що суттєво покращує загальний досвід спілкування.

2 ПРОЕКТУВАННЯ ТА ІНТЕГРАЦІЯ ЧАТ-БОТА ДЛЯ РОЗРОБНИКА

2.1 Обґрунтування архітектури штучного інтелекту чат-бота

При розробці чат-бота для розробників необхідно створити архітектуру, яка забезпечить ефективне розуміння технічних запитів, генерацію релевантного коду та підтримку контексту протягом усієї розмови. Фундаментом такої системи є сучасні великі мовні моделі (LLM), що базуються на архітектурі трансформерів. Розглянемо ключові компоненти цієї архітектури та обґрунтуємо їх важливість для чат-бота, орієнтованого на розробників програмного забезпечення. Ось основні елементи цієї архітектури:

- Механізм уваги (Attention mechanism)
- Самоувага (Self-attention)
- Багатошарова структура
- Тренування
- Токенізація
- Векторні представлення (Embeddings)
- Масштаб

Механізм уваги є революційною інновацією в обробці природної мови, який забезпечує ефективну обробку контексту в текстових даних. На відміну від попередніх підходів, що обробляли текст послідовно, механізм уваги дозволяє моделі динамічно "фокусуватися" на різних частинах вхідної послідовності при генерації кожного елемента вихідної послідовності.

$$AttentionScore(Q,K) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right), \quad (2.1)$$

де Q (запит), K (ключ) – матриці, що представляють різні проекції вхідних даних, а d_k – розмірність ключа.

Для чат-бота розробника ця функція має критичне значення, оскільки дозволяє моделі розуміти складні технічні запити, пов'язуючи терміни та концепції між собою. Наприклад, коли розробник запитує про оптимізацію певного алгоритму, механізм уваги дозволяє моделі пов'язати згадані структури даних, методи оптимізації та контекст використання, навіть якщо вони знаходяться в різних частинах запиту.

Самоувага є специфічною формою механізму уваги, де всі запити (Q), ключі (K) та значення (V) походять з одного і того ж джерела. Це дозволяє кожному токену "звертати увагу" на всі інші токени в послідовності, включаючи сам себе.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2.2)$$

Цей механізм обчислює вагові коефіцієнти для кожного токена відносно всіх інших токенів, створюючи зважену суму представлень, що відображає відносну важливість кожного токена для поточного контексту.

У контексті чат-бота для розробників самоувага дозволяє ефективно інтерпретувати кодові фрагменти та технічну документацію, де розуміння залежностей між різними частинами коду або компонентами системи є ключовим. Наприклад, при аналізі коду самоувага допомагає моделі пов'язати виклики функцій з їх визначеннями, змінні з їх типами та значеннями, навіть якщо вони розташовані на відстані один від одного.

Розширенням механізму самоуваги є багатоголова увага, яка дозволяє моделі одночасно враховувати різні типи взаємозв'язків між словами:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O \quad (2.3)$$

де $head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right)$

Кожна "голова" уваги фокусується на різних аспектах взаємозв'язків між токенами, що збагачує представлення даних. У контексті чат-бота для розробників це особливо важливо для розуміння складних програмних

конструкцій, де один фрагмент коду може мати різні типи зв'язків з іншими частинами (синтаксичні, семантичні, функціональні).

Архітектура трансформера складається з декількох однакових шарів, кожен з яких містить два основні підшари: багатоголову механізм уваги та повнозв'язну нейронну мережу. Кожен підшар оточений залишковим з'єднанням (residual connection) і супроводжується нормалізацією шару:

$$LayerNorm\big(x + Sublayer(x)\big) \tag{2.4}$$

де Sublayer(x) – функція, що реалізується підшаром.

Для чат-бота розробника глибока багатошарова структура дозволяє моделі обробляти інформацію на різних рівнях абстракції:

- Нижні шари захоплюють базові синтаксичні особливості та локальні взаємозв'язки
- Середні шари формують семантичне розуміння технічних концепцій
- Верхні шари інтегрують високорівневе розуміння алгоритмів, архітектурних патернів та програмних парадигм

Така ієрархічна обробка є ключовою для здатності чат-бота розуміти складні запити щодо архітектури програмного забезпечення, алгоритмічних підходів та системного дизайну.

Перед тим як текст може бути оброблений моделлю, він проходить процес токенізації – розбиття на менші одиниці (токени). Для чат-бота розробника особливо важливо використовувати токенізатор, що ефективно обробляє кодові конструкції різних мов програмування.

Популярні підходи до токенізації включають:

- BPE (Byte-Pair Encoding) – ітеративно об'єднує найчастіші пари символів чи підслів
- WordPiece – подібний до BPE, але використовує різні критерії об'єднання
- SentencePiece – працює на рівні необроблених послідовностей символів без попередньої токенізації слів

Після токенізації кожен токен перетворюється у векторне представлення (embedding) фіксованої розмірності d_{model}:

$$Token \rightarrow Embedding(d_{model}) \quad (2.5)$$

Крім того, додаються позиційні кодування, які надають моделі інформацію про порядок токенів у послідовності:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (2.6)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (2.7)$$

де pos – позиція токена, а i – вимір у векторному представленні.

Для чат-бота розробника якісна токенізація та векторні представлення забезпечують точне розуміння синтаксису різних мов програмування, правильну обробку спеціальних символів, ключових слів та програмних конструкцій.

Сучасні LLM відрізняються своїм масштабом, що вимірюється кількістю параметрів (вагів) моделі. Для ефективної роботи чат-бота розробника важливо обрати модель з оптимальним балансом між потужністю та ефективністю, порівняння розмірів моделей зображено на таблиці 2.1.

Таблиця 2.1 – Порівняння розмірів моделей

Розмір моделі	Кількість параметрів	Переваги	Недоліки
Маленькі	1-3 мільярди	Швидка відповідь, низькі вимоги до ресурсів	Обмежене розуміння складних концепцій
Середні	7-13 мільярдів	Збалансоване співвідношення якості та швидкості	Обмежені можливості для найскладніших задач
Великі	20-70 мільярдів	Глибоке розуміння складних тематик, висока якість коду	Високі обчислювальні вимоги, повільніша відповідь
Дуже великі	100+ мільярдів	Найкраще розуміння контексту та генерація коду	Надзвичайно високі вимоги до обчислювальних ресурсів

Для чат-бота розробника, залежно від конкретних вимог, оптимальним вибором часто є моделі середнього розміру (7-13B параметрів), які

забезпечують хороший баланс між якістю відповідей та швидкістю роботи. Для складних задач архітектурного проектування та оптимізації алгоритмів можуть бути доцільними більші моделі (20-70B).

Ефективність чат-бота для розробників значною мірою залежить від процесу тренування та якості навчальних даних. Тренування LLM зазвичай відбувається у кілька етапів:

1. Попереднє тренування (Pre-training) – навчання на великих масивах загальнодоступного тексту методом прогнозування наступного токена:

$$L_{PT}(\theta) = -\sum_i log\ P(x_i\ |x_{<i}, \theta) \quad (2.8)$$

де θ – параметри моделі, x_i – і-й токен, а $x_{<i}$ – всі попередні токени.

2. Тонке налаштування (Fine-tuning) – додаткове навчання на спеціалізованих даних, що включають:
   - Документацію різних мов програмування та фреймворків
   - Кодові бази з відкритим кодом
   - Питання та відповіді з технічних форумів (Stack Overflow, GitHub Discussions)
   - Технічні блоги та статті про розробку програмного забезпечення
3. Навчання з людським зворотним зв'язком (RLHF – Reinforcement Learning from Human Feedback) – подальше покращення моделі через зворотний зв'язок від людей-експертів:

$$L_{RLHF}(\theta) = E_x[log(P_\theta(y_w \mid x))] \quad (2.9)$$

де y_w – відповідь, яку людина-експерт оцінила як корисну.

Для чат-бота розробника особливо важливою є якість даних для тонкого налаштування, які повинні охоплювати різноманітні мови програмування, парадигми, фреймворки та інструменти розробки. Це забезпечує здатність моделі розуміти та генерувати релевантний код і технічні пояснення.

Важливим аспектом архітектури чат-бота для розробників є розмір контекстного вікна – максимальна кількість токенів, які модель може

обробляти одночасно. Це особливо критично для роботи з кодом, де контекст часто охоплює великі фрагменти коду, документацію та пояснення.

Сучасні підходи до розширення контекстного вікна включають:

1. Розріджена увага (Sparse Attention) – замість обчислення повної матриці уваги $O(n^2)$ використовуються різні патерни розрідженості, що зменшують складність до $O(n \log(n))$ або навіть $O(n)$:

$$BlockSparse(Q,K,V) = softmax(\frac{Mask(QK^T)}{\sqrt{d_k}})V \quad (2.10)$$

де Mask() – функція, що обмежує взаємодію між токенами за певним патерном.

2. Рекурентна обробка (Recurrent Processing) – послідовна обробка тексту з збереженням стану:

$$State_t = f(State_{\{t-1\}}, Input_t) \quad (2.11)$$

3. KV-кешування (KV-Caching) – збереження ключів та значень з попередніх обчислень для ефективнішої роботи з довгим контекстом.

Для чат-бота розробника розширене контекстне вікно (8K-32K токенів або більше) є критичним, оскільки дозволяє:

1. Аналізувати великі фрагменти коду цілісно
2. Зберігати контекст попередніх питань та відповідей
3. Підтримувати складні багатокрокові діалоги щодо розробки та налагодження

Архітектура чат-бота для розробників повинна передбачати можливості інтеграції з різноманітними інструментами розробки:

1. Інтерфейси API – для взаємодії з документацією, пакетними менеджерами та іншими зовнішніми ресурсами
2. Плагіни для IDE – для безпосередньої інтеграції в робоче середовище розробника

3. Системи контролю версій – для контекстно-залежної допомоги на основі код-бази проекту
4. Функції виконання коду – для перевірки та демонстрації згенерованих рішень

Ця інтеграція розширює можливості базової LLM та забезпечує більш практичну цінність для розробників.

2.2 Визначення основних функцій чат-бота

Чат-боти - це програмні продукти, які можуть виконувати різноманітні завдання за допомогою комунікації з людьми за допомогою текстових повідомлень. Вони забезпечують ефективне та автоматизоване виконання рутинних завдань, поліпшують комунікацію між користувачем та сервісом, а також дозволяють економити час та зусилля. [15]

Однією з основних функцій чат-ботів є покращення обслуговування користувачів. Вони можуть відповідати на їх запитання, допомагати у вирішенні проблем та надавати додаткову інформацію про програмування та послуги. Чат-боти дозволяють зменшити час очікування та поліпшити якість обслуговування. Чат-боти можуть бути використані для просування продуктів та послуг. Вони можуть відповідати на запитання, допомагати у виборі продуктів та послуг та рекомендувати інші товари. Крім того, чат-боти можуть створювати персоналізовані пропозиції для покращення того чи іншого аспекту програмування.

Чат-боти можуть допомагати у вирішенні технічних проблем клієнтів. Вони можуть надавати інформацію про можливі проблеми та шляхи їх вирішення, а також надавати детальні інструкції щодо використання продуктів та послуг компанії. Чат-боти можуть бути використані для автоматизації рутинних програмувань. Наприклад, вони можуть допомагати збирати інформацію про клієнтів, реєструвати замовлення та виконувати інші

завдання, які можуть бути автоматизовані. Чат-боти можуть бути використані для навчання користувачів. Вони можуть надавати інформацію про продукти та послуги, виконувати роль тренера, надавати рекомендації щодо вирішення проблем та допомагати в зборі даних.

Діаграма взаємодії користувача з сервером через чат-бот зображено на рисунку 2.1.

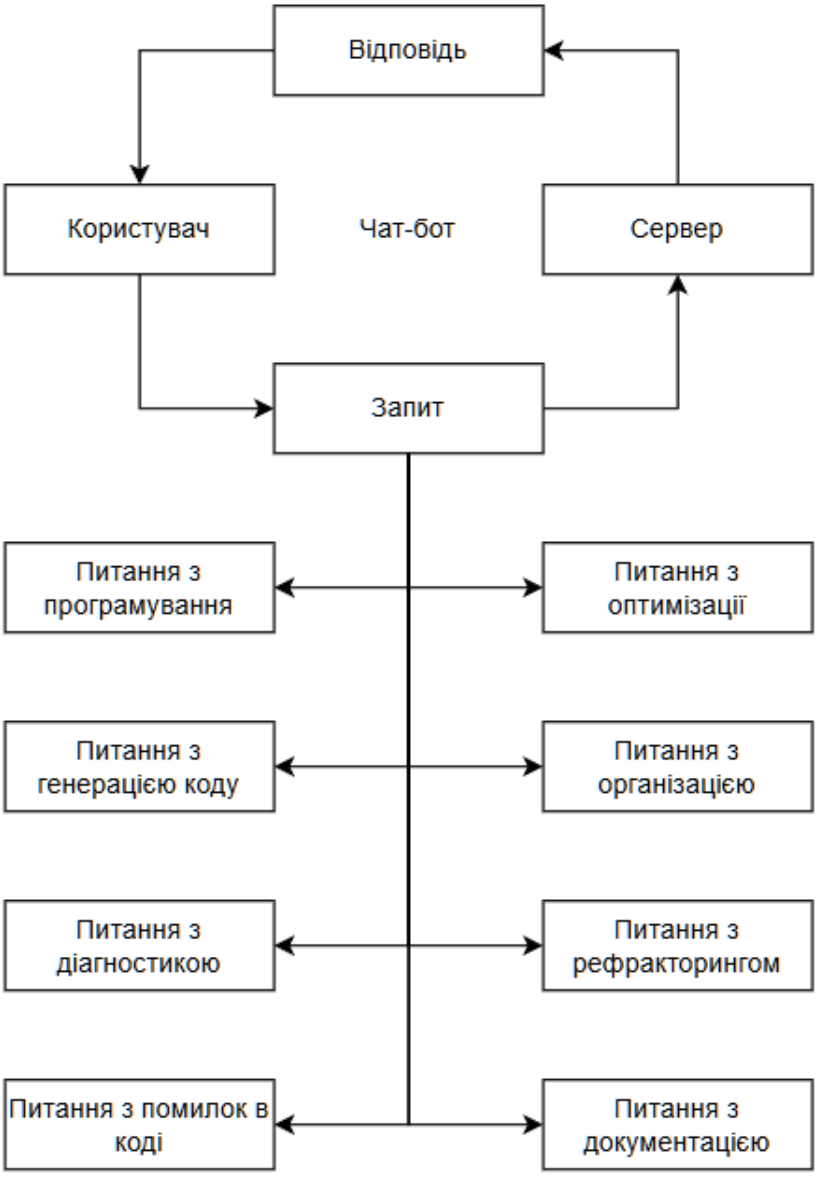

Рисунок 2.1 – Взаємодія користувача з сервером

Чат-боти можуть бути використані для організації роботи команди. Вони можуть допомагати у плануванні завдань, контролювати виконання завдань та надавати звіти про результати. Чат-бот для розробників програмного забезпечення повинен виконувати ряд спеціалізованих функцій, що відповідають потребам цільової аудиторії. На основі аналізу робочих процесів розробників, типових завдань та проблем, з якими вони стикаються, визначимо основні функціональні можливості чат-бота. Ключовою функцією чат-бота для розробників є підтримка в різних аспектах роботи з кодом:

1. Генерація коду за описом:
    - Створення функцій, класів і модулів на основі текстового опису функціональності
    - Реалізація алгоритмів за їх словесним описом
    - Створення шаблонів для типових шаблонів проектування
2. Аналіз та рефакторинг коду:
    - Виявлення потенційних помилок та "антипатернів"
    - Пропозиції щодо оптимізації структури та продуктивності
    - Допомога у покращенні читабельності та підтримуваності коду
3. Документування коду:
    - Генерація документаційних коментарів (JavaDoc, DocStrings, JSDoc тощо)
    - Створення README файлів та технічної документації
    - Формування пояснень до складних алгоритмів або логіки

Також важливим функціоналом для даного чат-бота буде допомога з налагодженням та виправленням помилок. Цей функціональний блок фокусується на допомозі розробникам у виявленні та виправленні проблем у коді:

1. Аналіз повідомлень про помилки:
    - Інтерпретація повідомлень про помилки і трасувань стеку
    - Переклад технічних повідомлень про помилки на зрозумілу мову

- Підказки про поширені причини конкретних помилок

2. Діагностика логічних помилок:
    - Аналіз коду для виявлення логічних помилок і граничних випадків
    - Пропозиції тестових випадків для виявлення проблем
    - Візуалізація виконання коду для розуміння проблемних областей
3. Пропозиції з виправлення:
    - Генерація патчів для виправлення виявлених помилок
    - Альтернативні реалізації для уникнення проблем
    - Рекомендації щодо кращих практик для запобігання подібним помилкам

Визначення основних функцій чат-бота є ключовим етапом у його розробці. Чат-боти, як програмні агенти, здатні взаємодіяти з користувачами за допомогою текстових повідомлень, відкриваючи широкі можливості для автоматизації різноманітних завдань. Їхнє застосування вже давно вийшло за межі простого обслуговування клієнтів, охоплюючи галузі освіти, бізнесу та навіть розваг. Ефективність чат-ботів полягає в їх здатності забезпечувати швидку, цілодобову підтримку, зменшувати навантаження на людські ресурси та персоналізувати взаємодію з користувачами. У контексті розробки програмного забезпечення, де швидкість отримання інформації та ефективність виконання рутинних завдань є критично важливими, чат-бот може стати незамінним помічником, оптимізуючи робочий процес та підвищуючи продуктивність.

Однією з фундаментальних функцій такого чат-бота є підтримка в різних аспектах роботи з кодом. Це охоплює не лише генерацію початкових фрагментів коду на основі текстового опису, що значно прискорює процес розробки типових рішень, але й більш складні завдання, такі як реалізація цілих алгоритмів за їх словесним представленням. Уявіть собі можливість просто описати потрібну функціональність, і чат-бот запропонує вам готовий до використання код. Крім того, він може створювати шаблони для часто

використовуваних архітектурних патернів, допомагаючи підтримувати єдиний стиль та структуру в проекті.

Не менш важливою є функція аналізу та рефакторингу коду. Чат-бот може стати вашим віртуальним код-рев'юером, виявляючи потенційні помилки ще на етапі написання, а також вказуючи на "антипатерни", які можуть призвести до проблем у майбутньому. Пропозиції щодо оптимізації структури коду та підвищення його продуктивності допоможуть зробити ваш код більш ефективним та легким у підтримці. Додатково, чат-бот може суттєво покращити читабельність коду, пропонуючи зміни для кращого розуміння його логіки, що особливо цінно при роботі в команді або при поверненні до коду через тривалий час.

Діаграма аналізу контексту повідомлень-запитів зображено на рисунку 2.2.

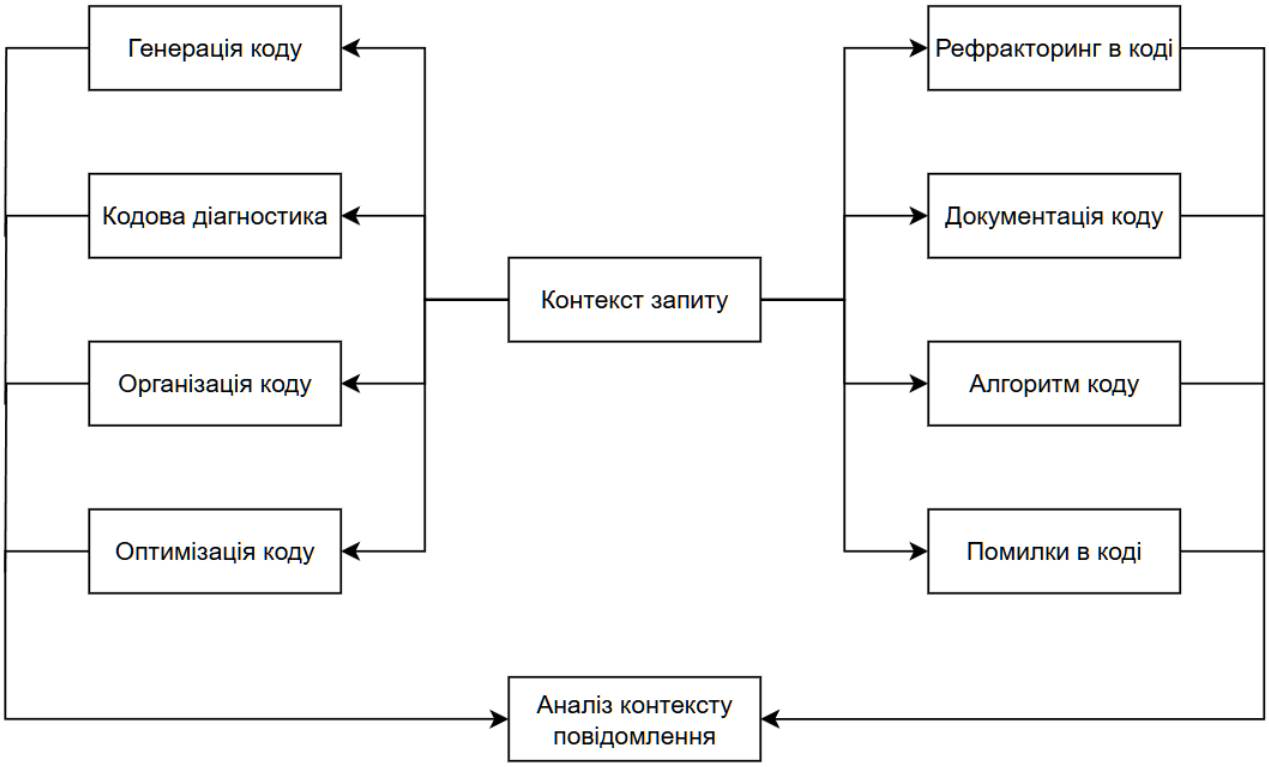

Рисунок 2.2 – Аналіз контексту повідомлень-запитів

Процес документування коду, який часто сприймається розробниками як рутинне та нецікаве завдання, також може бути значно полегшений за допомогою чат-бота. Він може автоматично генерувати документаційні коментарі у різних форматах, таких як JavaDoc, DocStrings або JSDoc, на основі аналізу коду. Крім того, він може допомогти у створенні README файлів, які є важливою частиною будь-якого проекту, а також у підготовці більш детальної технічної документації. Здатність чат-бота формувати пояснення до складних алгоритмів або неочевидної логіки може значно спростити розуміння кодової бази для нових членів команди або при передачі проекту. Функціональні можливості чат-бота для розробників програмного забезпечення зображено на таблиці 2.2.

Таблиця 2.2 – Функціональні можливості чат-бота

Категорія	Функції	Опис функцій
Робота з кодом	Генерація коду	• Створення функцій, класів і модулів на основі текстового опису • Реалізація алгоритмів за їх словесним описом • Створення шаблонів для типових шаблонів проектування
	Аналіз та рефакторинг	• Виявлення потенційних помилок та "антипатернів" • Пропозиції щодо оптимізації структури та продуктивності • Допомога у покращенні читабельності та підтримуваності коду
	Документування коду	• Генерація документаційних коментарів (JavaDoc, DocStrings, JSDoc) • Створення README файлів та технічної документації • Формування пояснень до складних алгоритмів або логіки

Таблиця 2.2 – Продовження функціональних можливостей чат-бота

Категорія	Функції	Опис функцій
Налагодження та виправлення помилок	Аналіз помилок	• Інтерпретація повідомлень про помилки і трасувань стеку • Переклад технічних повідомлень про помилки на зрозумілу мову • Підказки про поширені причини конкретних помилок
	Діагностика логічних помилок	• Аналіз коду для виявлення логічних помилок і граничних випадків • Пропозиції тестових випадків для виявлення проблем • Візуалізація виконання коду для розуміння проблемних областей
	Пропозиції з виправлення	• Генерація патчів для виправлення виявлених помилок • Альтернативні реалізації для уникнення проблем • Рекомендації щодо кращих практик для запобігання помилкам
Загальні функції чат-ботів	Обслуговування користувачів	• Відповіді на запитання • Допомога у вирішенні проблем • Зменшення часу очікування
	Бізнес-функції	• Просування продуктів та послуг • Створення персоналізованих пропозицій • Збір інформації про клієнтів
	Автоматизація	• Реєстрація замовлень • Автоматизація рутинних завдань • Організація роботи команди
Допоміжні функції	Навчання та підтримка	• Надання інформації про продукти та технології • Виконання ролі тренера • Допомога у плануванні завдань
	Інтеграція з інструментами	• Взаємодія з системами контролю версій • Інтеграція з IDE • Робота з пакетними менеджерами та CI/CD

Ще одним критично важливим функціональним блоком є допомога з налагодженням та виправленням помилок. Кожен розробник стикається з помилками, і чат-бот може стати цінним помічником у їхньому вирішенні. Він може аналізувати повідомлення про помилки та трасування стеку, надаючи їхню інтерпретацію зрозумілою мовою, що особливо корисно для початківців. Чат-бот може також підказувати поширені причини виникнення конкретних помилок, заощаджуючи час на пошук рішень.

Крім того, чат-бот може допомагати в діагностиці логічних помилок, які часто є найбільш складними для виявлення. Аналізуючи код, він може вказувати на потенційні логічні невідповідності та пропонувати тестові випадки, які допоможуть виявити приховані проблеми. Візуалізація виконання коду, яку може запропонувати чат-бот, допоможе краще зрозуміти потік виконання програми та локалізувати проблемні області. Нарешті, чат-бот може надавати конкретні пропозиції з виправлення знайдених помилок, генеруючи патчі або пропонуючи альтернативні реалізації, а також рекомендуючи кращі практики для запобігання подібним проблемам у майбутньому.

Для всебічної підтримки розробників, чат-бот також може надавати допомогу у вивченні та розумінні технологій. Він може пояснювати складні технічні концепції та термінологію простою мовою, порівнювати різні технології та підходи, а також роз'яснювати принципи роботи конкретних інструментів та бібліотек. Надання прикладів використання є ще однією важливою функцією, що дозволяє розробникам швидко зрозуміти, як застосовувати певні технології на практиці. Чат-бот може також здійснювати швидкий пошук та узагальнення інформації з офіційної документації, заощаджуючи час на самостійне вивчення великих обсягів тексту.

Функціонал управління знаннями та найкращими практиками також є надзвичайно цінним. Чат-бот може надавати рекомендації щодо стилю кодування, допомагати у виборі інструментів та технологій, порівнюючи їхні переваги та недоліки. Можливість збереження та обміну корисними

фрагментами коду та знаннями сприятиме підвищенню ефективності роботи як окремих розробників, так і цілих команд.

Нарешті, інтеграція з інструментами розробки є ключем до зручного та ефективного використання чат-бота. Взаємодія з системами контролю версій, інтеграція з IDE, робота з пакетними менеджерами та інструментами CI/CD дозволять розробникам використовувати функціонал чат-бота безпосередньо в їхньому звичному робочому середовищі, роблячи його невід'ємною частиною їхнього інструментарію.

2.3 Алгоритми взаємодії чат-бота з розробником

Ефективна взаємодія є ключовим аспектом для будь-якого чат-бота, особливо такого, що орієнтований на професійну аудиторію, як розробники програмного забезпечення. Алгоритми, що лежать в основі цієї взаємодії, повинні забезпечувати не лише розуміння запитів користувача, але й контекстуальне реагування, здатність до навчання та адаптації, а також зручність використання. [16]

Одним з фундаментальних алгоритмів є обробка природної мови (NLP). Цей комплекс алгоритмів відповідає за розуміння текстових запитів розробника і має в собі алгоритм багатоголової уваги та алгоритм масштабованого скалярного добутку уваги. На першому етапі відбувається лексичний аналіз, де вхідний текст розбивається на окремі токени (слова, символи, розділові знаки). Потім застосовуються методи морфологічного аналізу для визначення граматичних характеристик кожного токена (частина мови, число, рід тощо). Наступним кроком є синтаксичний аналіз, який встановлює граматичну структуру речення та зв'язки між токенами. Для розуміння семантики запиту використовуються алгоритми семантичного аналізу, які намагаються визначити значення слів та фраз у контексті.

Для чат-бота розробника особливо важливим є розпізнавання намірів (Intent Recognition). Цей алгоритм класифікує запит користувача відповідно до його мети (наприклад, "згенерувати код", "пояснити помилку", "знайти документацію"). Для цього можуть використовуватися методи машинного навчання, такі як класифікація текстів на основі векторних представлень (embeddings) слів та речень, а також моделі на основі трансформерів, які враховують контекст усього запиту.

Алгоритм обробки природньої мови зображено на рисунку 2.3.

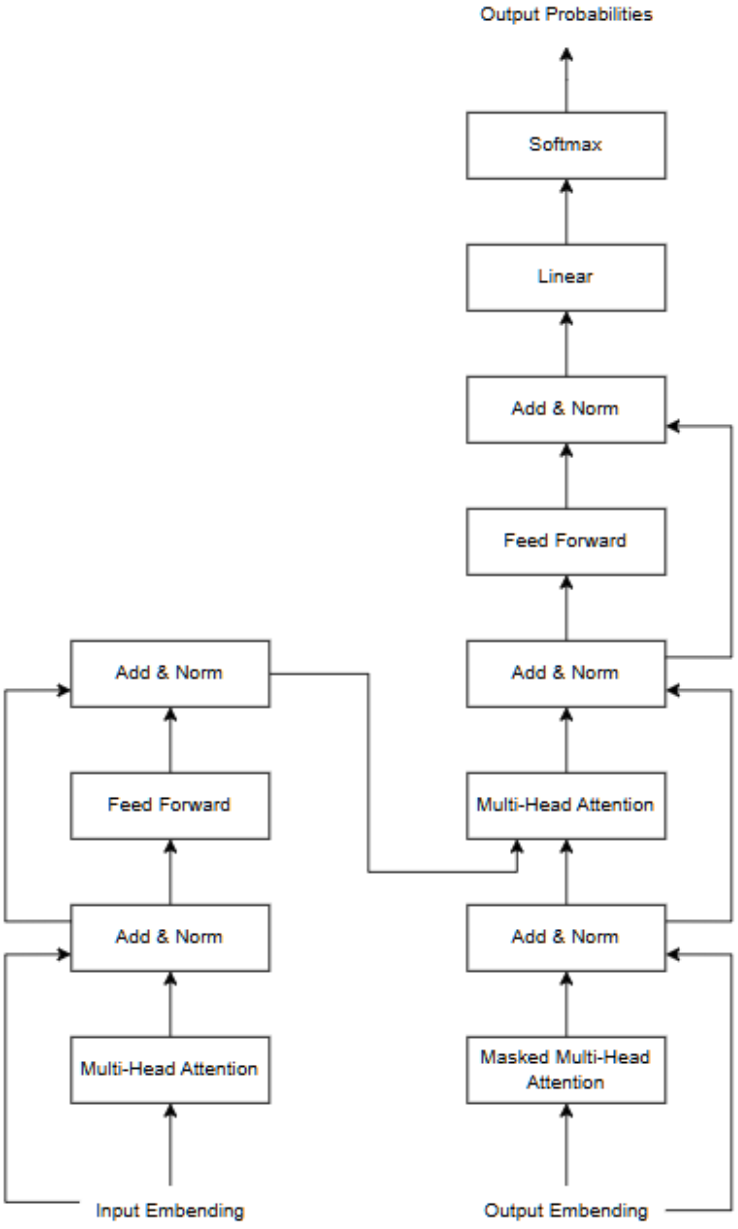

Рисунок 2.3 – Алгоритм NLP

Після розпізнавання наміру необхідно вилучити сутності (Entity Extraction) – ключові інформаційні елементи, що містяться в запиті. Для розробника це можуть бути назви мов програмування, фреймворків, бібліотек, назви функцій, класи, конкретні технології або навіть фрагменти коду. Алгоритми вилучення сутностей використовують як лінгвістичні правила, так і статистичні моделі, навчені на великих обсягах текстових даних, включаючи технічну документацію та код.

Для підтримки контексту розмови чат-бот повинен використовувати алгоритми управління діалогом (Dialogue Management). Ці алгоритми відстежують історію взаємодії з користувачем, запам'ятовують попередні запити та відповіді, а також витягнуті сутності. Це дозволяє чат-боту розуміти посилання на попередньо обговорені теми та підтримувати більш природний та змістовний діалог. Для цього можуть застосовуватися різні підходи, від простих скінченних автоматів до складних моделей управління станом на основі машинного навчання.

Алгоритм багатоголової уваги зображено на рисунку 2.4.

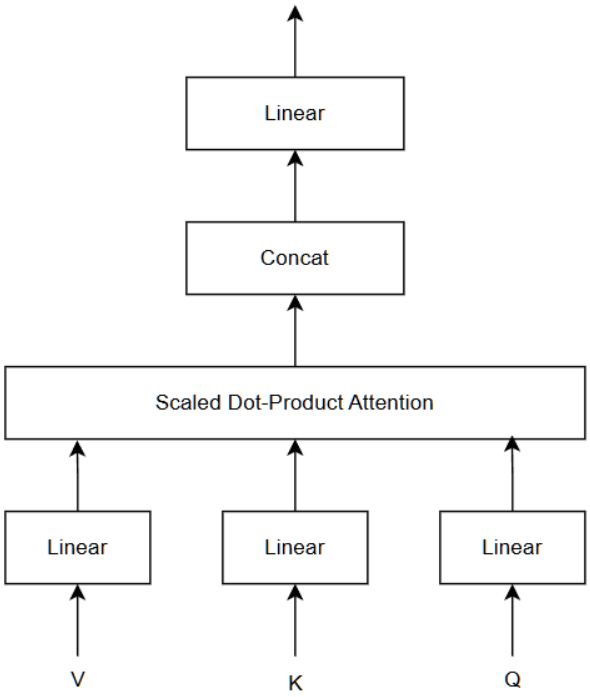

Рисунок 2.4 – Алгоритм багатоголової уваги

Генерація відповіді є ще одним важливим аспектом взаємодії. Залежно від наміру користувача, чат-бот може використовувати різні алгоритми для формування відповіді. Якщо запит стосується генерації коду, можуть застосовуватися алгоритми програмованої генерації (Programmed Generation), які на основі шаблонів та вилучених сутностей створюють відповідний код. Для більш складних запитів, що вимагають розгорнутих пояснень або відповідей на питання, можуть використовуватися генеративні мовні моделі (Generative Language Models), які на основі контексту та попередньої історії діалогу генерують текстову відповідь.

Алгоритм масштабованого скалярного добутку уваги зображено на рисунку 2.5.

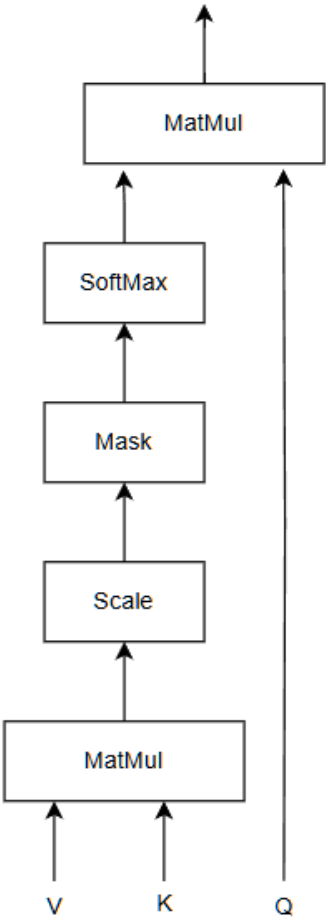

Рисунок 2.5 – Алгоритм масштабованого скалярного добутку уваги

Особливу увагу слід приділити алгоритмам, що забезпечують взаємодію з зовнішніми інструментами та ресурсами. Чат-бот може використовувати API для доступу до документації, пошукових систем, баз даних коду, систем контролю версій та інших інструментів, які можуть бути корисними для розробника. Алгоритми повинні забезпечувати ефективне формування запитів до цих інструментів, обробку отриманих даних та інтеграцію їх у відповідь користувачеві.

Для покращення якості взаємодії важливими є алгоритми оцінки та навчання. Система може збирати дані про взаємодію з користувачами, включаючи оцінки відповідей та відгуки. Ці дані можуть використовуватися для навчання та тонкого налаштування моделей NLP, управління діалогом та генерації відповідей, щоб підвищити їхню точність та релевантність. Можуть застосовуватися як методи навчання з учителем (на основі розмічених даних), так і навчання з підкріпленням (на основі зворотного зв'язку від користувачів).

Зручність використання чат-бота значною мірою залежить від алгоритмів представлення інформації. Згенерований код повинен бути чітко відформатований та легко копіюватися. Технічні пояснення повинні бути структурованими, логічними та містити приклади. При взаємодії з великими обсягами інформації (наприклад, документацією) важливими є алгоритми для виділення найрелевантніших фрагментів та їх стислого представлення.

Для забезпечення надійної та ефективної роботи чат-бота необхідні алгоритми обробки помилок та виняткових ситуацій. Система повинна вміти коректно обробляти незрозумілі запити, відсутності необхідної інформації у зовнішніх джерелах або інші непередбачені ситуації, надаючи користувачеві інформативні повідомлення та пропонуючи можливі шляхи вирішення проблеми або переформулювання запиту.

2.4 Процес оброблення запитів

Коли розробник звертається до чат-бота з певним запитом, запускається складна послідовність дій, що включає кілька етапів обробки для забезпечення точної та релевантної відповіді.

Першим кроком у цьому процесі є отримання та попереднє оброблення вхідного запиту. Коли користувач вводить текст свого запитання або команди, цей текст надходить до системи чат-бота. На цьому етапі відбувається ряд підготовчих дій, таких як видалення зайвих пробілів, приведення тексту до єдиного регістру (за потреби) та, що особливо важливо для обробки природної мови, токенізація. Як ми вже згадували в попередньому розділі, токенізація полягає у розбитті вхідного тексту на менші смислові одиниці – токени. Вибір ефективного методу токенізації, який враховує особливості мов програмування та технічної термінології, є критично важливим для подальшого аналізу.

Наступним етапом є аналіз запиту, який включає кілька підпроцесів. Розпізнавання наміру є одним з ключових елементів цього етапу. Модель машинного навчання, навчена на великому обсязі даних, аналізує послідовність токенів і визначає основну мету запиту користувача. Наприклад, чи хоче розробник згенерувати код, отримати пояснення помилки, знайти інформацію про певну технологію, чи можливо, потребує допомоги з рефакторингом існуючого коду. Паралельно з розпізнаванням наміру відбувається вилучення сутностей. Це процес ідентифікації та виокремлення ключових інформаційних елементів у запиті, які є важливими для виконання наміру користувача. У контексті розробки це можуть бути назви мов програмування (наприклад, Python, Java), назви бібліотек (наприклад, React, Spring), конкретні терміни (наприклад, "рекурсія", "багатопотоковість"), або навіть фрагменти коду, які користувач вставляє у свій запит.

Після того як намір розпізнано, а ключові сутності вилучено, відбувається контекстуалізація запиту. На цьому етапі система враховує

попередню історію спілкування з користувачем. Алгоритми управління діалогом аналізують збережений контекст, щоб правильно інтерпретувати поточний запит, особливо якщо він містить посилання на попередні питання або відповіді. Підтримка контексту дозволяє чат-боту вести більш осмислені та послідовні діалоги з розробником, уникаючи необхідності повторювати інформацію в кожному новому запиті.

На основі розпізнаного наміру, вилучених сутностей та контексту розмови відбувається вибір алгоритму обробки та формування відповіді. Залежно від типу запиту, може бути активовано один або кілька спеціалізованих алгоритмів. Наприклад, якщо запит стосується генерації коду, буде задіяно алгоритм програмованої генерації, який на основі отриманих параметрів (наприклад, мови програмування, потрібної функціональності) створить відповідний код. Якщо користувач запитує про пояснення помилки, може бути використаний алгоритм аналізу повідомлень про помилки, який спробує інтерпретувати наданий текст помилки та запропонувати можливі причини та шляхи вирішення. Для запитів, що потребують доступу до зовнішніх ресурсів, таких як документація або бази знань, буде активовано алгоритм взаємодії з API відповідних сервісів.

Після того як алгоритм обробки завершив свою роботу, формується відповідь користувачеві. Цей етап включає не лише генерацію текстового повідомлення, але й форматування коду (якщо він був згенерований), структурування інформації для кращого сприйняття, а також, за потреби, підготовку додаткових матеріалів або посилань на зовнішні ресурси. Важливо, щоб відповідь була не лише точною та релевантною, але й зрозумілою та зручною для використання розробником.

Останнім етапом є надання відповіді користувачеві через відповідний інтерфейс (текстове повідомлення, інтеграція в IDE тощо). Після цього система може перейти в режим очікування наступного запиту, продовжуючи підтримувати контекст поточної розмови. Важливо зазначити, що протягом усього процесу оброблення запиту можуть використовуватися алгоритми

оцінки якості та логування. Це дозволяє відстежувати ефективність різних етапів обробки, виявляти потенційні проблеми та збирати дані для подальшого навчання та вдосконалення чат-бота. Аналіз типових запитів та реакції користувачів на надані відповіді є цінним джерелом інформації для оптимізації алгоритмів та покращення загальної якості взаємодії.

2.5 Висновок до розділу 2

Оптимальна архітектура чат-бота для розробників базується на сучасних LLM з трансформерною архітектурою, доповнених спеціалізованими компонентами для роботи з кодом та технічною документацією. Ключовими компонентами є:

1. Механізми уваги (самоувага, багатоголова увага) – для ефективного розуміння взаємозв'язків у коді та технічних текстах
2. Глибока багатошарова структура – для обробки інформації на різних рівнях абстракції
3. Спеціалізовані системи токенізації – для ефективної обробки кодових конструкцій
4. Розширене контекстне вікно – для роботи з великими фрагментами коду та підтримки тривалих діалогів
5. Тонке налаштування на технічних даних – для глибокого розуміння програмних концепцій та патернів
6. Інтеграційні інтерфейси – для з'єднання з інструментами розробки

Така архітектура забезпечує розробників інтелектуальним помічником, здатним розуміти складні технічні запити, генерувати якісний код, пояснювати алгоритми та допомагати у вирішенні різноманітних задач програмування та системного дизайну.

Визначення основних функцій чат-бота "Асистент для розробника програмного забезпечення" охоплює широкий спектр потреб розробників, від

допомоги у написанні та налагодженні коду до вивчення нових технологій та інтеграції з існуючими інструментами. Ретельне опрацювання кожної з цих функцій забезпечить створення потужного та корисного інструменту, здатного значно підвищити продуктивність та полегшити щоденну роботу розробників.

Впровадження та ефективна робота алгоритмів обробки природної мови, розпізнавання намірів, вилучення сутностей, управління діалогом, програмованої генерації, взаємодії з зовнішніми інструментами та ресурсами, оцінки та навчання, представлення інформації та обробки помилок та виняткових ситуацій є запорукою того, що чат-бот "Асистент для розробника програмного забезпечення" стане дійсно корисним та цінним інструментом у повсякденній роботі розробників, допомагаючи їм вирішувати складні завдання, швидко отримувати необхідну інформацію та підвищувати свою продуктивність.

Процес обробки запитів розробників чат-ботами представляє собою комплексну багатоетапну систему, спрямовану на забезпечення точних та ефективних відповідей. Цей процес включає такі критичні етапи: попередня обробка запиту з токенізацією тексту; аналіз запиту через розпізнавання наміру та вилучення важливих сутностей; контекстуалізація запиту з урахуванням історії спілкування; застосування відповідних алгоритмів обробки залежно від типу запиту; формування та надання структурованої відповіді користувачеві.

Особливо важливими аспектами цієї системи є здатність підтримувати контекст розмови, що дозволяє вести послідовний діалог без повторення інформації, та використання спеціалізованих алгоритмів для різних типів запитів (генерація коду, аналіз помилок, доступ до зовнішніх ресурсів).

Така складна архітектура обробки забезпечує не лише технічну точність відповідей, але й зручність їх використання розробниками, що є ключовим фактором ефективності взаємодії між людиною та чат-ботом у контексті розробки програмного забезпечення.

3 РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ЧАТ-БОТА

3.1 Реалізація інтерфейсу чат-бота

Для реалізації інтерфейсу чат-бота буде використана бібліотека PyQt5. Qt – це набір кросплатформних бібліотек C++, які реалізують високорівневі API для доступу до багатьох аспектів сучасних настільних та мобільних систем. До них належать служби визначення місцезнаходження та позиціонування, мультимедіа, підключення NFC та Bluetooth, веб-браузер на базі Chromium, а також традиційна розробка інтерфейсу користувача. [17]

PyQt5 – це комплексний набір прив'язок Python для Qt v5. Він реалізований у вигляді понад 35 модулів розширення та дозволяє використовувати Python як альтернативну мову розробки додатків для C++ на всіх підтримуваних платформах, включаючи iOS та Android. [17]

Отже імпортуємо всі необхідні інструменти (рисунок 3.1).

```
1  import sys
2  import markdown2
3  import json
4
5  from time import sleep, time
6
7  from PyQt5 import QtWidgets, QtWebEngineWidgets
8  from PyQt5.QtCore import pyqtSlot, QObject, pyqtSignal, QThreadPool, QTimer
9  from PyQt5.QtWebChannel import QWebChannel
```

Рисунок 3.1 – Імпорт необхідних інструментів

Далі створемо загальний клас чат-боту який буде нащадком від QtWidgents.QMainWindow. QtWidgents.QMainWindow це головне вікно, що забезпечує основу для побудови інтерфейсу користувача програми. Qt має QMainWindow та пов'язані з ним класи для керування головним вікном.

QMainWindow має власне макетування, до якого можна додавати QToolBar, QDockWidget, QMenuBar та QStatusBar. [18]

Абстракція класу ChatBot зображена на рисунку 3.2

```
167   class ChatBot(QtWidgets.QMainWindow):
168 >     def __init__(self, page_html, config_parameters):···
198
199 >     def resizeEvent(self, a0):···
202
203 >     def send_message(self, content, role):···
208
209 >     def check_if_fully_loaded(self):···
211
212 >     def handle_ready_state(self, state):···
217
218 >     def on_page_loaded(self):···
227
228 >     def removeMessage(self):···
239
240 >     def retryMessage(self):···
283
284 >     def processMessage(self, text:str):···
```

Рисунок 3.2 – Абстракція класу ChatBot

Даний клас має методи свого батька що дозволить при створені його в основній функції одразу мати всі його методи, тож залишається тільки зробити доналаштування. Клас приймає сторінку HTML та параметри конфігурації для доналаштування.

Далі для опрацювання відповідей чат-боту створемо клас який буде представляти саму модель чат-бота. Абстракція класу modelAI зображена на рисунку 3.3.

```
72   class modelAI:
73 >     def __init__(self, config_parameters):···
85
86 >     def check_model_status(self) -> bool:···
95
96 >     def create_stream_response(self):···
115
116 >     def process_stream_part_response(self, part):···
```

Рисунок 3.3 – Абстракція класу modelAI

Даний клас буде запускатись в середині класу ChatBot і там же виконуватись для опрацювання самого чату та генерації відповідей.

Далі для того щоб реалізувати інтерфейс у вигляді HTML документу треба створити клас ChatBridge який буде служити мостом для HTML документу та Python коду. Реалізація класу ChatBridge зображена на рисунку 3.4.

```
142  class ChatBridge(QObject):
143      receivedMessage = pyqtSignal(str)
144      stop_flag = False
145      regenerateMessage = pyqtSignal()
146      deleteMessage = pyqtSignal()
147
148      @pyqtSlot(str)
149      def sendMessage(self, message):
150          self.receivedMessage.emit(message)
151
152      @pyqtSlot()
153      def endStream(self):
154          self.stop_flag = True
155
156      @pyqtSlot()
157      def retryMessage(self):
158          self.regenerateMessage.emit()
159
160      @pyqtSlot()
161      def removeMessage(self):
162          self.deleteMessage.emit()
```

Рисунок 3.4 – Реалізація класу ChatBridge

Даний клас є необхідним для зв’язку Python з HTML кодом сторінки, бо має відбуватись взаємодія інтерфейсу з тим що Python буде робити, тобто сигнали з елементів інтерфейсу зчитуються завдяки JavaScript в сторінці HTML та передаються у Python для опрацювання чат-ботом та генерації відповідей.

Далі реалізуємо функцію init класу ChatBot для прив’язки інших класів та самої ініціалізації програми. Реалізація функції init класу ChatBot зображена на рисунку 3.5.

```
167  class ChatBot(QtWidgets.QMainWindow):
168      def __init__(self, page_html, config_parameters):
169          super().__init__()
170
171          self.setWindowTitle("ChatBot - Assistant of Software Developer")
172          self.setMinimumSize(800, 600)
173
174          self.the_model = modelAI(config_parameters)
175
176          central_widget = QtWidgets.QWidget()
177          self.setCentralWidget(central_widget)
178          main_layout = QtWidgets.QVBoxLayout(central_widget)
179
180          self.html_view = QtWebEngineWidgets.QWebEngineView()
181          main_layout.addWidget(self.html_view)
182          self.html_view.setHtml(page_html)
183          self.check_if_fully_loaded()
184
185          settings = self.html_view.settings()
186          settings.setAttribute(QtWebEngineWidgets.QWebEngineSettings.WebAttribute.ScrollAnimatorEnabled, True)
187
188          self.bridge = ChatBridge()
189          self.bridge.receivedMessage.connect(self.processMessage)
190          self.bridge.regenerateMessage.connect(self.retryMessage)
191          self.bridge.deleteMessage.connect(self.removeMessage)
192          self.channel = QWebChannel()
193          self.channel.registerObject("bridge", self.bridge)
194          self.html_view.page().setWebChannel(self.channel)
195
196          self.needs_update = False
```

Рисунок 3.5 – Реалізація функції init класу ChatBot

За даною реалізацією видно, що параметри передаються у клас modelAI та створюється основний віджет в якому відображається сама сторінка HTML яка також передається з основної функції. PyQtWebEngine — це набір прив'язок Python для фреймворку Qt WebEngine від компанії The Qt Company. Фреймворк надає можливість вбудовувати веб-контент у програми та базується на браузері Chrome. Прив'язки розташовані поверх PyQt5 та реалізовані як три окремі модулі, що відповідають різним бібліотекам, що складають фреймворк. [19]

Далі створемо саму сторінку HTML документу для перевірки працездатності двигуна. На рисунку 3.6 зображено головну сторінку HTML.

```html
<html>
    <head>
        <script src="qrc:///qtwebchannel/qwebchannel.js"></script>
        <style>···
        </style>
    </head>

    <body id="main-body">
        <div id="chat-container">
            <p class="message system">AI is loading...</p>
        </div>
        <div id="input-container">
            <textarea id="message-input" placeholder="Type a message..." rows="1"></textarea>
            <button id="send-button" class="input-button">△</button>
            <button id="stop-button" class="input-button">□</button>
        </div>

        <script>···
        </script>
    </body>
</html>
```

Рисунок 3.6 – Головна сторінка HTML

Щоб сторінка краще виглядала створемо стиль сторінки, який буде одразу вписаний в файл сторінки, бо так буде простіше завантажувати все разом. Також особливістю даної сторінки в тому що вона використовує скрипт який береться з <script src="qrc:///qtwebchannel/qwebchannel.js"></script>, що є строчками JavaScript коду який можна виконувати через Python.

Особливість реалізації стилю сторінки ще полягає в тому, що буде використовуватись стиль від markdown2. Цей інструмент є корисним для конвертації тексту від чат-боту в HTML розмітку для додавання стилю як курсивний текст, жирний, заголовки, код і тому подібне. Реалізація використання markdown2 зображено на рисунку 3.7.

```python
def message_refiner(input):
    text = markdown2.markdown(input, extras=["extra",
                                             "tables",
                                             "fenced-code-blocks",
                                             "task_list",
                                             "strike",
                                             "cuddled-lists"])
    result = json.dumps(text)
    return result
```

Рисунок 3.7 – Реалізація використання markdown2

Тепер можна зробити тестування сторінки. На рисунку 3.8 зображено тестовий запуск програми.

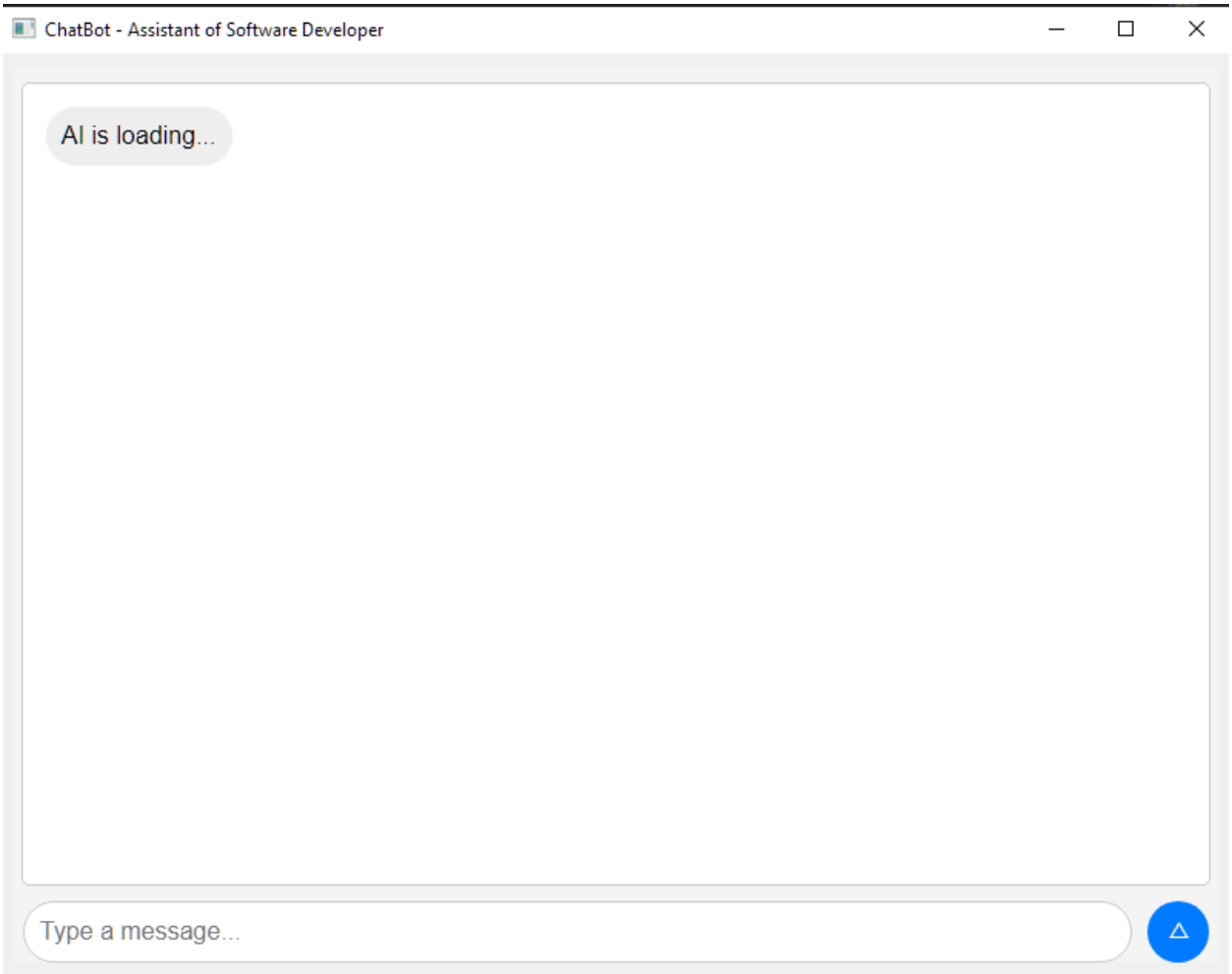

Рисунок 3.8 – Тестовий запуск програми

Як видно, інтерфейс виглядає приємно та зручно для користувача. Синя кнопка відповідає за відправлення повідомлення, вона же буде замінена на червону якщо відбувається генерація відповіді. Поле для введення тексту розташоване поруч з кнопкою тому буде зручно для користувача. Основне поле повідомлень буде мати в собі всі повідомлення які відправляє користувач та чат-бот.

Далі створюємо реалізацію динаміки сторінки, що дозволить їй якраз взаємодіяти з Python частиною програми. На рисунку 3.9 зображено початкову реалізацію динаміки сторінки.

```
new QWebChannel(qt.webChannelTransport, function(channel) {
    window.bridge = channel.objects.bridge;
});

let chatContainer = document.getElementById('chat-container');
let inputField = document.getElementById('message-input');
let sendButton = document.getElementById('send-button');
let stopButton = document.getElementById('stop-button');

const tmpl_message = document.createElement('div');
tmpl_message.className = 'message';
tmpl_message.innerHTML = '...';

const tmpl_thinking = document.createElement('div');
tmpl_thinking.className = 'message thinking';
tmpl_thinking.innerHTML = `
    <button class="dropdown-toggle">
        Reasoning
        <svg class="arrow" viewBox="0 0 24 24" fill="none" stroke="currentColor" stroke-width="2">
            <path stroke-linecap="round" stroke-linejoin="round" d="m8.25 4.5 7.5 7.5-7.5 7.5"></path>
        </svg>
    </button>
    <div class="dropdown-content">...</div>
`;

const tmpl_retry_button = document.createElement('button');
tmpl_retry_button.id = 'retry-delete-message-button';
tmpl_retry_button.innerHTML = '↻';
tmpl_retry_button.style = 'font-size: 20;'

const tmpl_delete_button = document.createElement('button');
tmpl_delete_button.id = 'retry-delete-message-button';
tmpl_delete_button.innerHTML = '✕';

const tmpl_buttons_container = document.createElement('div');
tmpl_buttons_container.style = 'display: flex;';
tmpl_buttons_container.appendChild(tmpl_retry_button);
tmpl_buttons_container.appendChild(tmpl_delete_button);
```

Рисунок 3.9 – Початкова реалізація динаміки сторінки

Як видно, тут вже є реалізація інших елементів взаємодії інтерфейсу, як кнопка геренерації відповіді та видалення, як самі повідомлення.

Далі пишемо продовження реалізації динаміки сторінки, в якій вже будуть основні функції інтерфейсу. На рисунку 3.10 зображено кінцеву реалізацію динаміки сторінки.

```
    sendButton.addEventListener('click', sendMessageIn);
    inputField.addEventListener('keypress', function(event) {···

    stopButton.addEventListener('click', function(event) {···
    tmpl_retry_button.addEventListener('click', function(event) {···
    tmpl_delete_button.addEventListener('click', function(event) {···

    inputField.addEventListener('input', function () {···

    function sendMessageIn() {···

    document.addEventListener('click', function(event) {
        const button = event.target.closest('.dropdown-toggle');

        if(button) {
            const content = button.nextElementSibling;

            button.classList.toggle('active');
            content.classList.toggle('expanded');

            if (!content.classList.contains('expanded')) {
                content.style.maxHeight = null;
            }
        }
    });

    function addMessage(content, role) {···

    function updateLastMessage(newContent) {···

    function updateLastThinkingMessage(newContent) {···

    function updateLastRespondingMessage(newContent) {···
</script>
```

Рисунок 3.10 – Кінцева реалізацію динаміки сторінки

Тут вже є інтеграція функціоналу інтерфейсу з тим що буде робити Python і навпаки, якраз завдяки QWebChannel сигнали з інтерфейсу будуть передаватись у Python частину програми.

Тепер маючи інтерфейс можна продовжити реалізацію Python частини програми. Реалізація init функції класу modelAI є важливою частиною чат-бота, де буде створений клієнт OpenAI який якраз буде займатись генерацією текстових відповіді. На рисунку 3.11 зображено реалізацію фунції init класу modelAI.

```python
class modelAI:
    def __init__(self, config_parameters):
        self.__dict__.update(config_parameters)

        self.chat_history = [{"role": "system", "content": "AI is loading..."}]

        self.client = OpenAI(
            base_url=self.BASE_URL,
            api_key=self.API_KEY,
        )

        self.current_thinking = ""
        self.current_response = ""
```

Рисунок 3.11 – Реалізація функції init класу modelAI

Як видно, налаштування клієнта беруться з параметрів конфігу який завантажується з класу ChatBot.

В загальному, при запуску програми відбувається перевірка доступності чат-боту, що дозволяє виявити помилки його налаштування. На рисунку 3.12 зображено реалізацію перевірки чат-бота.

```python
    def check_model_status(self) -> bool:
        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=[{"role": "user", "content": "Ping"}],
            max_tokens=1
        )
        if response:
            return True
        return False
```

Рисунок 3.12 – Реаліація перевірки чат-бота

Всі вони завжди відповідають «Pong». Далі реалізовуємо функцію генератора відповіді від чат-боту, яка є потоковою функцією, щоб можна було бачити процес генерації відповіді по кожному токену. На рисунку 3.13 зображено реалізацію генератора чат-бота.

```python
    def create_stream_response(self):
        """Outputs: (reasoning|None, content|None)"""

        chat_messages = [x for x in self.chat_history if x["role"] in ("assistant", "user")]
        chat_messages.append({"role": "system", "content": self.BIAS_PROMPT})

        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=chat_messages,
            stream=True
        )

        for chunk in response:
            reasoning = getattr(chunk.choices[0].delta, "reasoning", None)
            content = getattr(chunk.choices[0].delta, "content", None)

            reason = '' if reasoning is None else reasoning

            yield (reason, content)
```

Рисунок 3.13 – Реалізація генератора чат-бота

Особливість даного генератора від стандартного у тому, що він розділяє генерацію на reasoning та content, що є процесами роздумів та відповіді чат-боту відповідно. Також там добавляється BIAS_PROMPT для доналаштування генерації тексту, щоб був якийсь початковий контекст того, чим займається чат-бот і чого йому очікувати від користувачів.

Тепер розпишемо реалізацію опрацювання дельти генератора. Вона потрібна для вписування результату генерації у загальну змінну повідомлень, які використовуються для генерації відповідей. На рисунку 3.14 зображено реалізацію опрецювання дельти генератора.

```python
    def process_stream_part_response(self, part):
        thinking, content = part

        if thinking:
            self.current_thinking += thinking
        if content:
            self.current_response += content

        if len(self.chat_history) >= 2 and self.chat_history[-2]["role"] == "thinking" and self.chat_history[-1]["role"] == "assistant":
            self.chat_history[-2]["content"] = self.current_thinking
            self.chat_history[-1]["content"] = self.current_response
        else:
            thinking_entry = {"role": "thinking", "content": self.current_thinking}
            response_entry = {"role": "assistant", "content": self.current_response}
            self.chat_history.append(thinking_entry)
            self.chat_history.append(response_entry)

```

Рисунок 3.14 – Реалізація опрацювання дельти генератора

Тепер вже повернемось до реалізація самого класу ChatBot, а конкретніше головна функція яка взаємодіє з інтерфейсом. На рисунку 3.15 зображено ініціалізацію функції processMessage().

```
289     def processMessage(self, text:str):
290         if text:
291             self.the_model.chat_history.append({"role": "user", "content": "\n".join(line+"\n" for line in text.split("\n"))})
292 
293             self.the_model.current_thinking = ""
294             self.the_model.current_response = ""
295             self.generator = self.the_model.create_stream_response()
296 
297             def generate_response():
298                 js_code = f"""
299                 sendButton.style.display = 'none';
300                 stopButton.style.display = 'block';
301                 tmpl_retry_button.style.display = 'none';
302                 tmpl_delete_button.style.display = 'none';
303                 addMessage({message_refiner(self.the_model.chat_history[-1]["content"])}, 'user');
304                 addMessage("...", 'thinking');
305                 addMessage("...", 'assistant');
306                 """
307                 self.html_view.page().runJavaScript(js_code)
```

Рисунок 3.15 – Ініціалізація функції processMessage()

Тепер створення генератора з modelAI на рисунку 3.16.

```
309                 try:
310                     for part in self.generator:
311                         if self.bridge.stop_flag:
312                             break
313 
314                         countT += 1
315 
316                         self.the_model.process_stream_part_response(part)
317                         js_code_part = f"""
318                         updateLastThinkingMessage({message_refiner(self.the_model.chat_history[-2]["content"])});
319                         updateLastRespondingMessage({message_refiner(self.the_model.chat_history[-1]["content"])});
320                         """
321                         self.html_view.page().runJavaScript(js_code_part)
322 
323                 except Exception as e:
324                     self.send_message(f"Error: {e}", "error")
```

Рисунок 3.16 – Створення генератора з modelAI

І далі вже завершення генерації повідомлення на рисунку 3.17.

```
334                     self.generator = None
335                     self.bridge.stop_flag = False
336 
337                     js_code = f"""
338                     sendButton.style.display = 'block';
339                     stopButton.style.display = 'none';
340                     tmpl_retry_button.style.display = 'flex';
341                     tmpl_delete_button.style.display = 'flex';
342                     """
343                     self.html_view.page().runJavaScript(js_code)
344 
345                 QThreadPool.globalInstance().start(generate_response)
```

Рисунок 3.17 – Завершення генерації повідомлення

Ця функція є основою генерації всіх повідомлень і пов'язання всіх технологій які використовуються в чат-боті. Вона виконується асинхронно, щоб не зупиняти роботу всієї програми під час генерації відповіді.

На рисунку 3.18 зображено UML діаграму класів чат-боту.

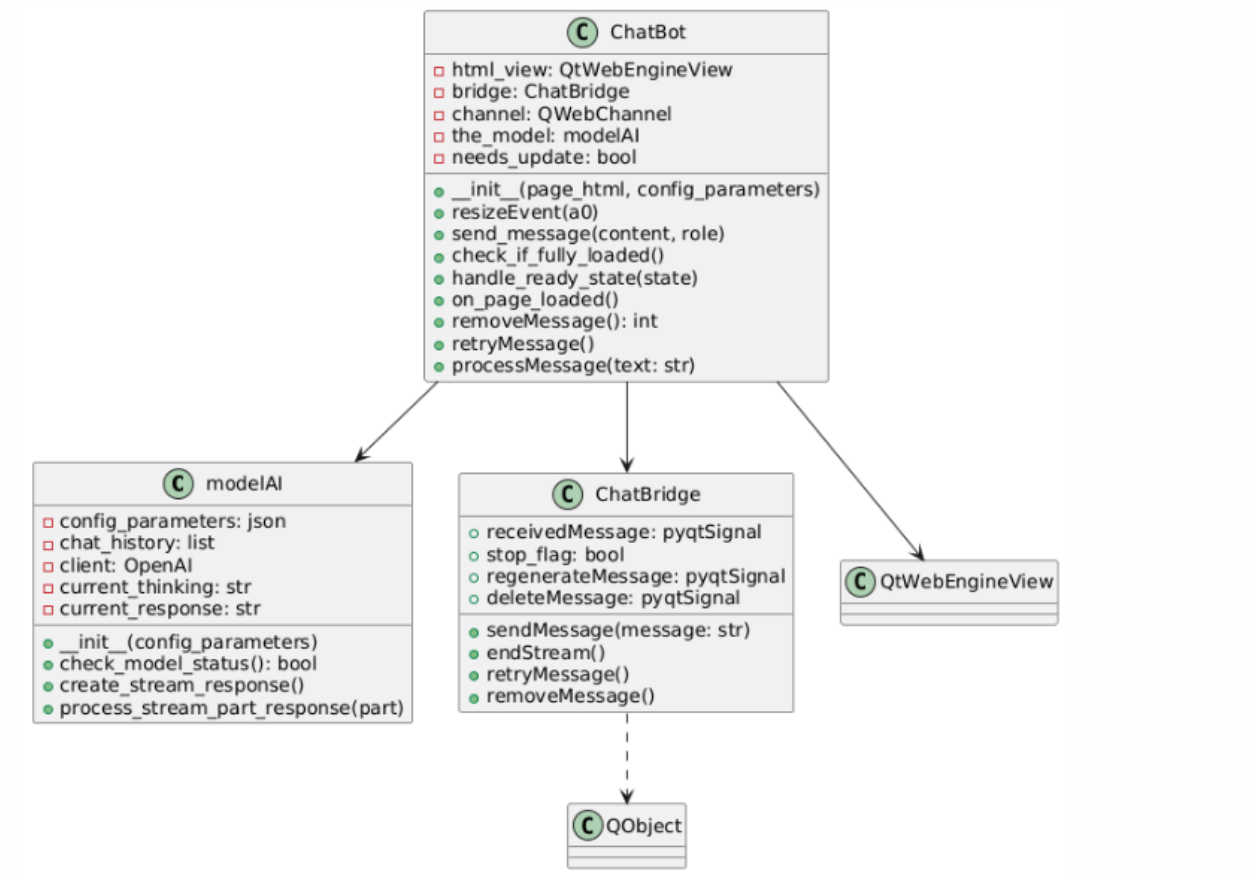

Рисунок 3.18 – UML діаграма класів чат-боту

Дана структура програми має дуже високий потенціал розширення завдяки використанню мови гіпертекстової розмітки в якості інтерфейсу та пайтон для додавання логіки програми. Тепер, маючи готову структуру програми, можна перейти до її тестування та оцінки ефективності її роботи та перевірки якості відповідей.

3.2 Тестування та перевірка якості відповідей

Для тестування якості відповідей чат-боту AoSD (Assistant of Software Developer) будемо оцінювати такі фактори як час на генерацію відповіді, кількість токенів та в якості запиту буде генерація програмного коду для оцінки самого коду та його роботи. Тестування проводиться у вигляді написання запиту до AoSD та ChatGPT та буде порівняння їх.

Запит: «write simple snake game on python»

ChatGPT: 42 секунд, 929 токенів. На рисунку 3.19 зображено результат тестування ChatGPT. [20]

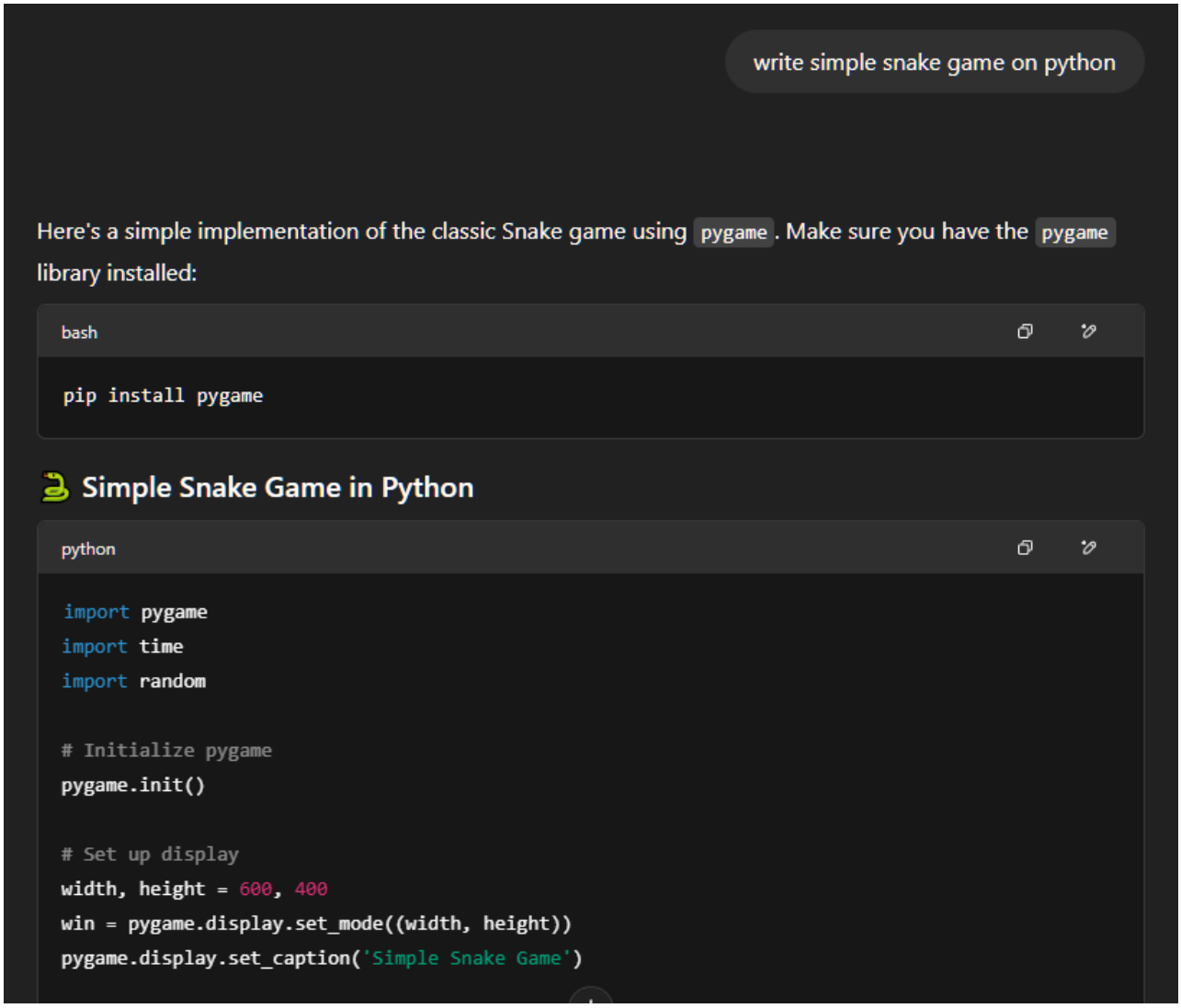

Рисунок 3.19 – Тестування ChatGPT

AoSD: 164 секунд, 10314 токенів. На рисунку 3.20 зображено результат тестування AoSD.

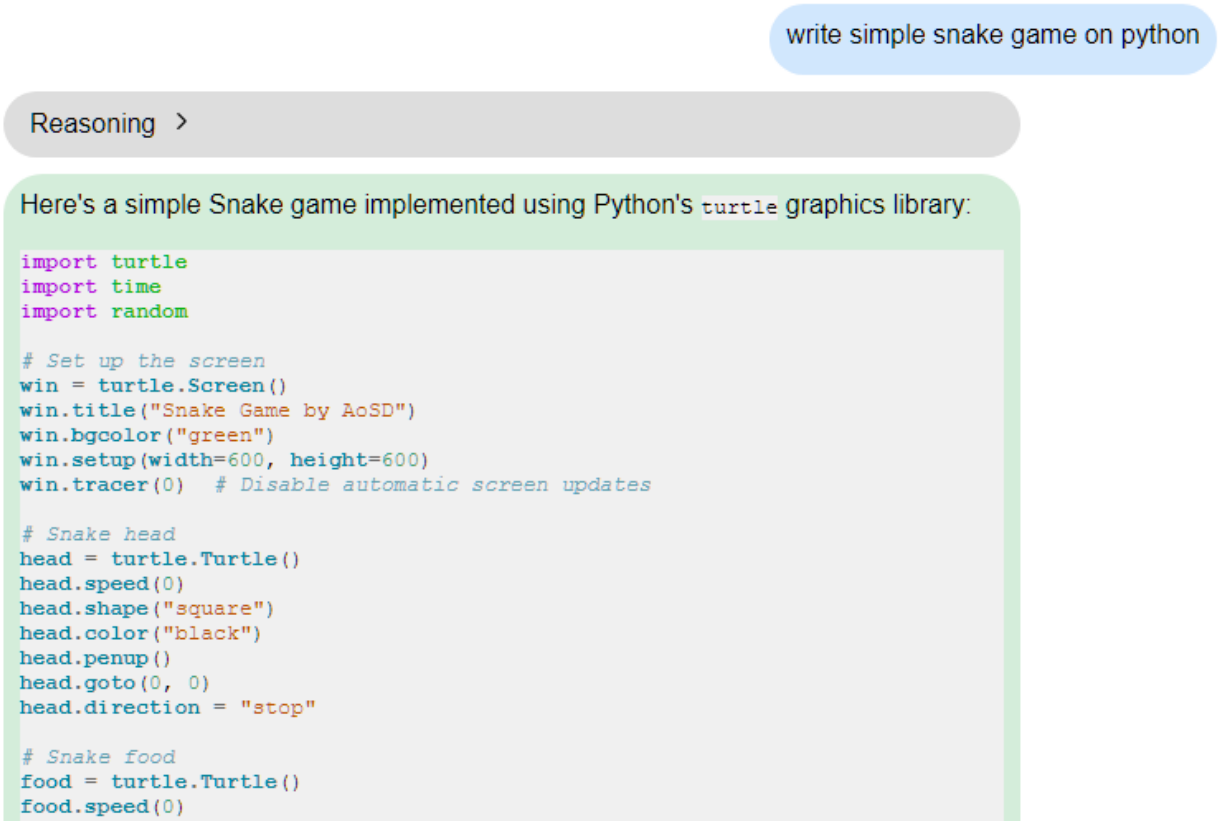

Рисунок 3.20 – Тестування AoSD

З даних результатів видно що ChatGPT справився швидше зі своєю роботою ніж AoSD за меншу кількість згенерованих токенів, але AoSD також використовував більшу частину токенів для обдумування відповіді, чого ChatGPT не робив, тому він значно довше генерував відповідь. Також, якщо обрахувати швидкість генерації, тобто кількість токенів поділити на час то вийде, що ChatGPT – 22 токенів/секунду, а AoSD – 63 токенів/секунду, що майже в 3 рази швидше за ChatGPT.

3.3 Оцінка продуктивності та ефективності чат-бота

За результатами попереднього тестування чат-бота, а саме програмний код, який він згенерував, то ось порівняння AoSD з ChatGPT:

- Технологічні відмінності
  - AoSD (Turtle Graphics):
    - Використовує вбудовану бібліотеку turtle - простіша для початківців
    - Розмір вікна: 600×600 пікселів
    - Об'єктно-орієнтований підхід з окремими turtle-об'єктами для змії, їжі та тексту
  - ChatGPT (Pygame):
    - Використовує pygame - більш потужну ігровуу бібліотеку
    - Розмір вікна: 600×400 пікселів
    - Процедурний підхід з функціями для малювання
- Функціональність
  - AoSD переваги:
    - Високий рахунок: зберігає найкращий результат між іграми
    - Запобігання зворотному руху: змія не може рухатись у протилежному напрямку
    - Плавніша анімація: менша затримка (0.1с) та більші блоки (20×20)
    - Автоматичний перезапуск: гра продовжується після програшу
  - ChatGPT переваги:
    - Меню програшу: можна вибрати перезапуск (C) або вихід (Q)

        - Контроль FPS: використовує clock.tick() для стабільної частоти кадрів
        - Кращі кольори: різноманітна палітра кольорів
- Якість коду
    - AoSD:
        - Більш читабельний та структурований код
        - Логічне розділення функцій руху
        - Ефективне управління сегментами змії
    - ChatGPT:
        - Компактніший, але складніший для розуміння
        - Рекурсивний виклик game_loop() може призвести до переповнення стеку
        - Менш оптимальне управління подіями

Отже з даних кодів можна сказати, що AoSD код кращий для навчання та повсякденного використання завдяки:

- Простішій технології (turtle)
- Кращій архітектурі коду
- Більш повній функціональності (високий рахунок, запобігання зворотному руху)
- Стабільнішій роботі

ChatGPT код може бути кращим для розробників, які хочуть розширювати гру, оскільки pygame надає більше можливостей для складніших ігор.

На рисунку 3.21 зображено результат роботи програми від ChatGPT.

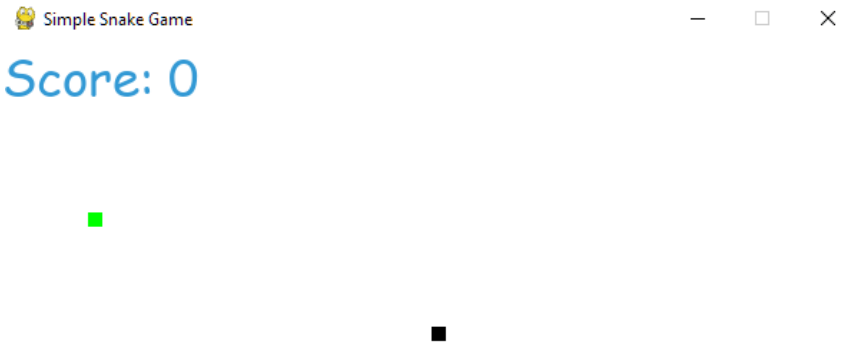

Рисунок 3.21 – Результат роботи програми від ChatGPT

На рисунку 3.22 зображено результат роботи програми від AoSD.

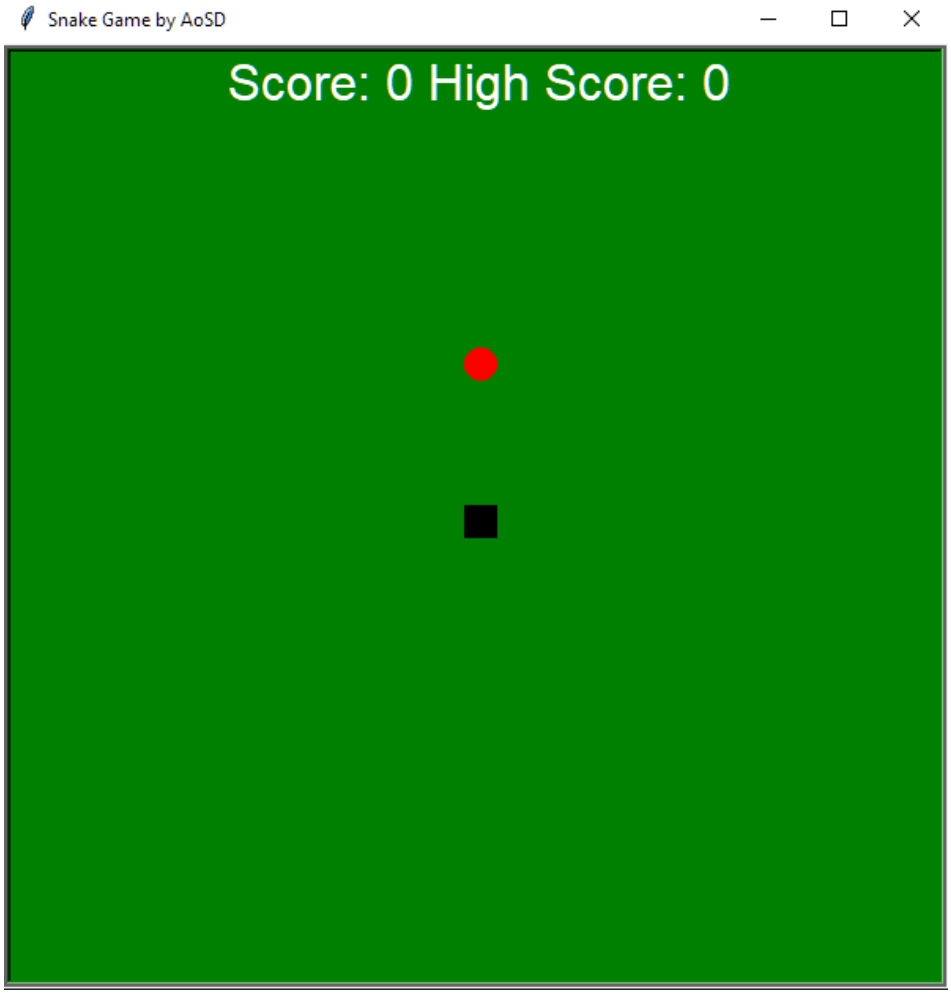

Рисунок 3.22 – Результат роботи програми від AoSD

Також було проведено тестування серед користувачів в галузі ІТ. Для тестування на достовірність правдивості відповідей було обрано вісім спеціалістів, що працюють в сфері інформаційних технологій. Оцінка роботи виставлялась від 1 до 10, в залежності від того була дана програма корисною чи ні. Результати тестування наведені в табл. 3.1.

Таблиця 3.1 – Результати тестування чат-боту

Спеціаліст	1	2	3	4	5	7	7	8
Оцінка	10	7	10	9	9	8	7	9

Проаналізувавши дані, маємо загальну оцінку 69 із 80. Отже, можна дійти висновку що якість функціонування чат-боту AoSD складає 88%. Це свідчить про досить високий рівень підвищення якості роботи асистента для розробників програмного коду.

3.4 Висновок до розділу 3

У даному розділі було успішно реалізовано повнофункціональний чат-бот AoSD (Assistant of Software Developer) з використанням сучасних технологій та підходів. Розроблено модульну архітектуру системи, що складається з трьох основних компонентів: класу ChatBot для основного інтерфейсу користувача на базі PyQt5, класу modelAI для обробки та генерації відповідей через OpenAI API, та класу ChatBridge для забезпечення взаємодії між HTML-інтерфейсом та Python-кодом.

Технологічна реалізація успішно інтегрувала кросплатформне рішення з використанням PyQt5 та PyQtWebEngine, що дозволило створити сучасний веб-орієнтований інтерфейс з підтримкою HTML, CSS та JavaScript. Було реалізовано потокову генерацію відповідей з підтримкою markdown-розмітки,

що забезпечує комфортну взаємодію користувача з системою в режимі реального часу.

Порівняльне тестування з ChatGPT показало, що AoSD демонструє вищу швидкість генерації токенів, досягаючи 63 токени за секунду проти 22 у ChatGPT. Проте система потребує більше часу на загальну генерацію через додаткові процеси обдумування відповіді, що компенсується якістю результату. Тестування нової розробки показало що якість функціонування чат-боту за таблицею 3.1 складає 88% що доводить підвищення якості роботи розробника програмного забезпечення.

ВИСНОВКИ

У ході виконання бакалаврської кваліфакаційної роботи було розроблено чат-бота «Ассистент для розробника програмного забезпечення» AoSD та досягенно задач дослідження у кожному розділі.

У першому розділі було проведено аналіз існуючих чат-ботів, а саме огляд сучасних чат-ботів та їх особливості з порівняльною характеристикою таких як ChatGPT, LLaMa, Claude, Gemini, DeepSeek, Mistral, разом з аналізом архітектури штучного інтелекту чат-ботів та методами обробки природньої мови.

У другому розділі було проведено проектування та інтеграцію чат-бота для розробника, а саме обґрунтування архітектури штучного інтелекту чат-ботів у більш детальному змісті, як і визначення основних функцій чат-бота де дослідили процес написання коду. В алгоритмах взаємодії чат-бота з розробником було обґрунтовано роботу чат-боту і процес оброблення запитів.

У третьому розділі було проведено реалізацію, тестування та оцінку ефективності чат-бота, разом з реалізацією інтерфейсу, тестуванням та перевіркою якості та оцінки ефективності і продуктивності чат-бота.

Мета розробки – підвищення якості роботи розробника програмного забезпечення з використанням чат-боту, досягнута за рахунок реалізації чат-боту AoSD, що успішно виконує поставлені завдання, та демонструє конкурентоспроможність з існуючими рішеннями. Система характеризується високою швидкістю обробки, якісною генерацією коду та зручним користувацьким інтерфейсом. Модульна архітектура забезпечує можливості для подальшого розвитку та розширення функціоналу системи. Отримані результати підтверджують ефективність обраних технологічних рішень та підходів до розробки, що робить систему придатною для практичного використання в сфері розробки програмного забезпечення. Відповідна іструкція користувача була розроблена. [21]

СПИСОК ВИКОРИСТАННОЇ ЛІТЕРАТУРИ

1. Розробка власного чат-бота на основі ChatGPT [Електронний ресурс] – Режим доступу: https://tto-studio.com.ua/rozrobka-vlasnoho-chat-bota-na-osnovi-chatgpt

2. Використання ChatGPT для сценаріїв розробки програмного забезпечення [Електронний ресурс] – Режим доступу: https://visuresolutions.com/uk/blog/leveraging-ai-for-use-cases-in-software-development

3. Розробка чат-ботів: технології та інструменти [Електронний ресурс] – Режим доступу: https://a4.com.ua/rozrobka-chat-botiv-tehnologii-ta-instrumenti

4. Що таке чат-бот: секрети використання та основні переваги для бізнесу [Електронний ресурс] – Режим доступу: https://helpcrunch.com/blog/uk/shcho-take-chat-bot

5. DeepSeek vs. ChatGPT: How Do They Compare? [Електронний ресурс] – Режим доступу: https://www.datacamp.com/blog/deepseek-vs-chatgpt

6. Claude vs. ChatGPT: What's the difference? [Електронний ресурс] – Режим доступу: https://zapier.com/blog/claude-vs-chatgpt

7. ChatGPT VS Gemini AI [Електронний ресурс] – Режим доступу: https://www.godofprompt.ai/blog/chatgpt-vs-gemini

8. Mistral vs GPT: A Comprehensive Comparison of Leading AI Models [Електронний ресурс] – Режим доступу: https://dev.to/abhinowww/mistral-vs-gpt-a-comprehensive-comparison-of-leading-ai-models-2lk2

9. Llama 3.2 vs ChatGPT-4: A Look At The AI Differences [Електронний ресурс] – Режим доступу: https://geekheads.au/blog/llama-3-2-vs-chatgpt-4-a-look-at-the-ai-differences

10. Transformer (deep learning architecture) [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)

11. Chatbot Architecture 101: A Comprehensive Guide to Building Intelligent Conversational AI [Електронний ресурс] – Режим доступу: https://sitebot.co/blog/chatbot-architecture-101-comprehensive-guide

12. Why architecture matters in AI chatbot development [Електронний ресурс] – Режим доступу: https://telnyx.com/resources/architecture-chatbot-development

13. Understanding the Principles of Natural Language Processing in Chatbots [Електронний ресурс] – Режим доступу: https://blog.lbenicio.dev/articles/2022-11-20-understanding-the-principles-of-natural-language-processing-in-chatbots/

14. Word embedding [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Word_embedding

15. Які завдання вирішує чат-бот, можливості та впровадження бота [Електронний ресурс] – Режим доступу: https://gerabot.com/article/yaki_zavdannya_virishu_chatbot_mozhlivosti_ta_vprovadzhennya_bota

16. 7 NLP Techniques for Extracting Information from Unstructured Text using Algorithms [Електронний ресурс] – Режим доступу: https://www.width.ai/post/extracting-information-from-unstructured-text-using-algorithms

17. PyQt5 - Comprehensive Python Bindings for Qt v5 [Електронний ресурс] – Режим доступу: https://pypi.org/project/PyQt5/

18. QMainWindow [Електронний ресурс] – Режим доступу: https://doc.qt.io/archives/qtforpython-5/PySide2/QtWidgets/QMainWindow.html

19. PyQtWebEngine - Python Bindings for the Qt WebEngine Framework [Електронний ресурс] – Режим доступу: https://pypi.org/project/PyQtWebEngine/

20. ChatGPT: Simple Snake Game [Електронний ресурс] – Режим доступу: https://chatgpt.com/share/6837282e-75b8-8012-b013-85002b57e340

21. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 122 «Комп'ютерні науки» [Електронний ресурс] – Режим доступу: https://iq.vntu.edu.ua/repository/getfile.php/6095.pdf

ДОДАТОК А (ОБОВ'ЯЗКОВИЙ)
ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка чат-боту "Асистент для розробника програмного забезпечення"

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра комп'ютерних наук
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 4,09 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Яровий А.А., зав. каф. КН _____
(прізвище, ініціали, посада) (підпис)

Колесницький О.К., проф. каф КН _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Озеранський В.С.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Петришин С.І.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Крикливий О.В.
(підпис) (прізвище, ініціали)

ДОДАТОК Б (ДОВІДНИКОВИЙ)
ІНСТРУКЦІЯ КОРИСТУВАЧА

1. Після запуску програми відкриється інтерфейс, в якому треба дочекатись повідомлення «done».
2. Далі коли з'являється повідомлення «done» можна вже починати писати повідомлення-запит у текстовому полі внизу.
3. Щоб відправити повідомлення достатно натиснути на клавішу Enter без натиснутої Shift, або на синю кнопку.
4. Якщо все пройде добре, то почнеться генерація відповіді, під час цього можна натиснути червону кнопку, тоді процес генерації зупинеться.
5. Коли генерація не відбувається, біля повідомлень чат-боту є кнопки рестарту та хрестик, рестарт перегенерує дане повідомлення, хрестик видаляє останні повідомлення чат-боту і користувача.

ДОДАТОК В (ОБОВ'ЯЗКОВИЙ)
ЛІСТИНГ ПРОГРАМИ

main.py

```
import sys
import markdown2
import json

from time import sleep, time

from PyQt5 import QtWidgets, QtWebEngineWidgets
from PyQt5.QtCore import pyqtSlot, QObject, pyqtSignal, QThreadPool, QTimer
from PyQt5.QtWebChannel import QWebChannel

from openai import OpenAI

class FakeModelAI:
    def __init__(self):
        self.chat_history = [{"role": "system", "content": "AI is loading..."}]

        self.current_thinking  = ""
        self.current_response = ""

    def check_model_status(self) -> bool:
        sleep(1)
        return True

    def create_stream_response(self):
```

```
        """Outputs: (reasoning|None, content|None)"""

        chat_messages = [x for x in self.chat_history if x["role"] in ("assistant",
"user")]

        # full_response = {
            #     "reasoning": "this is a reasoning message after user's:" +
chat_messages[-1]["content"],
            #     "content": "shown message after user's:" + chat_messages[-
1]["content"]
        # }

        full_response = {
            "reasoning": chat_messages[-1]["content"],
            "content": chat_messages[-1]["content"]
        }

        response: list[dict[str|None, str|None]] = [
            {"reasoning": x, "content": ""} for x in full_response["reasoning"]
        ] + [
            {"reasoning": "", "content": x} for x in full_response["content"]
        ]

        for chunk in response:
            reasoning = chunk["reasoning"]
            content = chunk["content"]

            reason = '' if reasoning is None else reasoning

            sleep(0.01)
```

```
                yield (reason, content)

            def process_stream_part_response(self, part):
                thinking, content = part

                if thinking:
                    self.current_thinking += thinking
                if content:
                    self.current_response += content

                    if len(self.chat_history) >= 2 and self.chat_history[-2]["role"] ==
"thinking" and self.chat_history[-1]["role"] == "assistant":
                        self.chat_history[-2]["content"] = self.current_thinking
                        self.chat_history[-1]["content"] = self.current_response
                    else:
                        thinking_entry = {"role": "thinking", "content": self.current_thinking}
                        response_entry = {"role": "assistant", "content": self.current_response}
                        self.chat_history.append(thinking_entry)
                        self.chat_history.append(response_entry)

        class modelAI:
            def __init__(self, config_parameters):
                self.__dict__.update(config_parameters)

                self.chat_history = [{"role": "system", "content": "AI is loading..."}]

                self.client = OpenAI(
                    base_url=self.BASE_URL,
                    api_key=self.API_KEY,
```

```
        )

        self.current_thinking  = ""
        self.current_response = ""

    def check_model_status(self) -> bool:
        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=[{"role": "user", "content": "Ping"}],
            max_tokens=1
        )
        if response:
            return True
        return False

    def create_stream_response(self):
        """Outputs: (reasoning|None,  content|None)"""

        chat_messages = [x for x in self.chat_history if x["role"] in ("assistant",
"user")]
                        chat_messages.append({"role":     "system",     "content":
self.BIAS_PROMPT})

        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=chat_messages,
            stream=True
        )

        for chunk in response:
```

```
            reasoning = getattr(chunk.choices[0].delta, "reasoning", None)
            content = getattr(chunk.choices[0].delta, "content", None)

            reason = '' if reasoning is None else reasoning

            yield (reason, content)

        def process_stream_part_response(self, part):
            thinking, content = part

            if thinking:
                self.current_thinking += thinking
            if content:
                self.current_response += content

            if len(self.chat_history) >= 2 and self.chat_history[-2]["role"] ==
"thinking" and self.chat_history[-1]["role"] == "assistant":
                self.chat_history[-2]["content"] = self.current_thinking
                self.chat_history[-1]["content"] = self.current_response
            else:
                thinking_entry = {"role": "thinking", "content": self.current_thinking}
                response_entry = {"role": "assistant", "content": self.current_response}
                self.chat_history.append(thinking_entry)
                self.chat_history.append(response_entry)

    def message_refiner(input):
        text = markdown2.markdown(input, extras=["extra",
                                                 "tables",
                                                 "fenced-code-blocks",
                                                 "task_list",
```

```
                                    "strike",
                                    "cuddled-lists"])
    result = json.dumps(text)
    return result

class ChatBridge(QObject):
    receivedMessage = pyqtSignal(str)
    stop_flag = False
    regenerateMessage = pyqtSignal()
    deleteMessage = pyqtSignal()

    @pyqtSlot(str)
    def sendMessage(self, message):
        self.receivedMessage.emit(message)

    @pyqtSlot()
    def endStream(self):
        self.stop_flag = True

    @pyqtSlot()
    def retryMessage(self):
        self.regenerateMessage.emit()

    @pyqtSlot()
    def removeMessage(self):
        self.deleteMessage.emit()



class ChatBot(QtWidgets.QMainWindow):
```

```
        def __init__(self, page_html, config_parameters):
            super().__init__()

            self.setWindowTitle("ChatBot - Assistant of Software Developer")
            self.setMinimumSize(800, 600)

            # self.the_model = FakeModelAI()
            self.the_model = modelAI(config_parameters)

            central_widget = QtWidgets.QWidget()
            self.setCentralWidget(central_widget)
            main_layout = QtWidgets.QVBoxLayout(central_widget)

            self.html_view = QtWebEngineWidgets.QWebEngineView()
            main_layout.addWidget(self.html_view)
            self.html_view.setHtml(page_html)
            self.check_if_fully_loaded()

            settings = self.html_view.settings()
             settings.setAttribute(QtWebEngineWidgets.QWebEngineSettings.WebAttribute.ScrollAnimatorEnabled, True)

            self.bridge = ChatBridge()
            self.bridge.receivedMessage.connect(self.processMessage)
            self.bridge.regenerateMessage.connect(self.retryMessage)
            self.bridge.deleteMessage.connect(self.removeMessage)
            self.channel = QWebChannel()
            self.channel.registerObject("bridge", self.bridge)
            self.html_view.page().setWebChannel(self.channel)
```

```
        self.needs_update = False

    def resizeEvent(self, a0):
        super().resizeEvent(a0)
        # self.request_update()

    def send_message(self, content, role):
        js_code_part = f"""
        addMessage({message_refiner(content)}, '{role}');
        """
        self.html_view.page().runJavaScript(js_code_part)

    def check_if_fully_loaded(self):
            self.html_view.page().runJavaScript('typeof chatContainer !==
"undefined"', self.handle_ready_state)

    def handle_ready_state(self, state):
        if state == True:
            self.on_page_loaded()
        else:
            QTimer.singleShot(1000, self.check_if_fully_loaded)

    def on_page_loaded(self):
        try:
            check = self.the_model.check_model_status()
        except Exception as e:
            self.send_message(f"Error: {e}", "error")
            return

        if check:
```

```
            self.send_message("done", "system")

        def removeMessage(self):
            messages = self.the_model.chat_history

            try:
                index = next(i for i, msg in enumerate(reversed(messages)) if msg["role"] == "user")
                trimmed = messages[:len(messages)-index-1]
            except StopIteration:
                trimmed = []

            self.the_model.chat_history = trimmed
            return index

        def retryMessage(self):
            self.the_model.current_thinking = ""
            self.the_model.current_response = ""
            self.generator = self.the_model.create_stream_response()

            def generate_response():
                js_code = f"""
                sendButton.style.display = 'none';
                stopButton.style.display = 'block';
                tmpl_retry_button.style.display = 'none';
                tmpl_delete_button.style.display = 'none';
                """
                self.html_view.page().runJavaScript(js_code)

                try:
```

```
                for part in self.generator:
                    if self.bridge.stop_flag:
                        break
                    # print(part[0], end="")
                    # print(part[1], end="")

                    self.the_model.process_stream_part_response(part)
                    js_code_part = f"""
                    updateLastThinkingMessage({message_refiner(self.the_model.chat_history[-2]["content"])});
                     updateLastRespondingMessage({message_refiner(self.the_model.chat_history[-1]["content"])});
                    """
                    self.html_view.page().runJavaScript(js_code_part)

            except Exception as e:
                self.send_message(f"Error: {e}", "error")

            self.generator = None
            self.bridge.stop_flag = False

            js_code = f"""
            sendButton.style.display = 'block';
            stopButton.style.display = 'none';
            tmpl_retry_button.style.display = 'flex';
            tmpl_delete_button.style.display = 'flex';
            """
            self.html_view.page().runJavaScript(js_code)

        QThreadPool.globalInstance().start(generate_response)
```

```
        def processMessage(self, text:str):
            if text:
                    self.the_model.chat_history.append({"role": "user", "content":
"\n".join(line+"\n" for line in text.split("\n"))})

                self.the_model.current_thinking = ""
                self.the_model.current_response = ""
                self.generator = self.the_model.create_stream_response()

                def generate_response():
                    js_code = f"""
                    sendButton.style.display = 'none';
                    stopButton.style.display = 'block';
                    tmpl_retry_button.style.display = 'none';
                    tmpl_delete_button.style.display = 'none';
                            addMessage({message_refiner(self.the_model.chat_history[-
1]["content"])}, 'user');
                    addMessage("...", 'thinking');
                    addMessage("...", 'assistant');
                    """
                    self.html_view.page().runJavaScript(js_code)

                    startTimer = time()
                    countT = 0

                    try:
                        for part in self.generator:
                            if self.bridge.stop_flag:
                                break
```

```
                    # print(part[0], end="")
                    # print(part[1], end="")

                    countT += 1

                    self.the_model.process_stream_part_response(part)
                    js_code_part = f"""
                      updateLastThinkingMessage({message_refiner(self.the_mod
el.chat_history[-2]["content"])});
                      updateLastRespondingMessage({message_refiner(self.the_m
odel.chat_history[-1]["content"])});
                    """
                    self.html_view.page().runJavaScript(js_code_part)

                except Exception as e:
                    self.send_message(f"Error: {e}", "error")

                print(f'Compute time: {time() - startTimer}')
                print(f'Tokens count: {countT}')

                self.generator = None
                self.bridge.stop_flag = False

                js_code = f"""
                sendButton.style.display = 'block';
                stopButton.style.display = 'none';
                tmpl_retry_button.style.display = 'flex';
                tmpl_delete_button.style.display = 'flex';
                """
                self.html_view.page().runJavaScript(js_code)
```

```
            QThreadPool.globalInstance().start(generate_response)

    if __name__ == "__main__":
        with open("2025_diplomna\page.html", "r", encoding="utf-8") as file:
            page_html = file.read()
        with open("2025_diplomna\config.json", "r", encoding="utf-8") as file:
            config_parameters = json.load(file)

        app = QtWidgets.QApplication(sys.argv)
        window = ChatBot(page_html, config_parameters)
        window.show()
        sys.exit(app.exec_())
```

page.html

```
<html>
    <head>
        <script src="qrc:///qtwebchannel/qwebchannel.js"></script>
        <style>
            .codehilite .hll { background-color: #ffffcc }
            .codehilite  { background: #f0f0f0; }
            .codehilite .n { color: #008197; font-weight: bold } /* Keyword */
            .codehilite .c { color: #60a0b0; font-style: italic  } /* Comment */
            .codehilite .err { border: 1px solid #FF0000 } /* Error */
            .codehilite .k { color: #007020; font-weight: bold } /* Keyword */
            .codehilite .o { color: #666666 } /* Operator */
                .codehilite .ch { color: #60a0b0; font-style: italic } /*
Comment.Hashbang */
```

```
.codehilite .cm { color: #60a0b0; font-style: italic } /* Comment.Multiline */
.codehilite .cp { color: #007020 } /* Comment.Preproc */
.codehilite .cpf { color: #60a0b0; font-style: italic } /* Comment.PreprocFile */
.codehilite .c1 { color: #60a0b0; font-style: italic } /* Comment.Single */
.codehilite .cs { color: #60a0b0; background-color: #fff0f0 } /* Comment.Special */
.codehilite .gd { color: #A00000 } /* Generic.Deleted */
.codehilite .ge { font-style: italic } /* Generic.Emph */
.codehilite .gr { color: #FF0000 } /* Generic.Error */
.codehilite .gh { color: #000080; font-weight: bold } /* Generic.Heading */
.codehilite .gi { color: #00A000 } /* Generic.Inserted */
.codehilite .go { color: #888888 } /* Generic.Output */
.codehilite .gp { color: #c65d09; font-weight: bold } /* Generic.Prompt */
.codehilite .gs { font-weight: bold } /* Generic.Strong */
.codehilite .gu { color: #800080; font-weight: bold } /* Generic.Subheading */
.codehilite .gt { color: #0044DD } /* Generic.Traceback */
.codehilite .kc { color: #007020; font-weight: bold } /* Keyword.Constant */
.codehilite .kd { color: #007020; font-weight: bold } /* Keyword.Declaration */
.codehilite .kn { color: #b300dd; font-weight: bold } /* Keyword.Namespace */
.codehilite .kp { color: #007020 } /* Keyword.Pseudo */
```

```
.codehilite .kr { color: #007020; font-weight: bold } /* Keyword.Reserved */
.codehilite .kt { color: #902000 } /* Keyword.Type */
.codehilite .m { color: #40a070 } /* Literal.Number */
.codehilite .s { color: #ad7217 } /* Literal.String */
.codehilite .na { color: #4070a0 } /* Name.Attribute */
.codehilite .nb { color: #007020 } /* Name.Builtin */
.codehilite .nc { color: #0e84b5; font-weight: bold } /* Name.Class */
.codehilite .no { color: #60add5 } /* Name.Constant */
.codehilite .nd { color: #555555; font-weight: bold } /* Name.Decorator */
.codehilite .ni { color: #d55537; font-weight: bold } /* Name.Entity */
.codehilite .ne { color: #007020 } /* Name.Exception */
.codehilite .nf { color: #06287e } /* Name.Function */
.codehilite .nl { color: #002070; font-weight: bold } /* Name.Label */
.codehilite .nn { color: #0eb51c; font-weight: bold } /* Name.Namespace */
.codehilite .nt { color: #062873; font-weight: bold } /* Name.Tag */
.codehilite .nv { color: #bb60d5 } /* Name.Variable */
.codehilite .ow { color: #007020; font-weight: bold } /* Operator.Word */
.codehilite .w { color: #bbbbbb } /* Text.Whitespace */
.codehilite .mb { color: #40a070 } /* Literal.Number.Bin */
.codehilite .mf { color: #40a070 } /* Literal.Number.Float */
.codehilite .mh { color: #40a070 } /* Literal.Number.Hex */
.codehilite .mi { color: #40a070 } /* Literal.Number.Integer */
.codehilite .mo { color: #40a070 } /* Literal.Number.Oct */
.codehilite .sa { color: #4070a0 } /* Literal.String.Affix */
.codehilite .sb { color: #4070a0 } /* Literal.String.Backtick */
.codehilite .sc { color: #4070a0 } /* Literal.String.Char */
```

```
.codehilite .dl { color: #4070a0 } /* Literal.String.Delimiter */
.codehilite .sd { color: #4070a0; font-style: italic } /* Literal.String.Doc */
.codehilite .s2 { color: #bb5001 } /* Literal.String.Double */
.codehilite .se { color: #4070a0; font-weight: bold } /* Literal.String.Escape */
.codehilite .sh { color: #4070a0 } /* Literal.String.Heredoc */
.codehilite .si { color: #70a0d0; font-style: italic } /* Literal.String.Interpol */
.codehilite .sx { color: #c65d09 } /* Literal.String.Other */
.codehilite .sr { color: #235388 } /* Literal.String.Regex */
.codehilite .s1 { color: #4070a0 } /* Literal.String.Single */
.codehilite .ss { color: #517918 } /* Literal.String.Symbol */
.codehilite .bp { color: #007020 } /* Name.Builtin.Pseudo */
.codehilite .fm { color: #06287e } /* Name.Function.Magic */
.codehilite .vc { color: #bb60d5 } /* Name.Variable.Class */
.codehilite .vg { color: #bb60d5 } /* Name.Variable.Global */
.codehilite .vi { color: #bb60d5 } /* Name.Variable.Instance */
.codehilite .vm { color: #bb60d5 } /* Name.Variable.Magic */
.codehilite .il { color: #40a070 } /* Literal.Number.Integer.Long */

code {
    background: #f0f0f0;
}
pre {
    background: #f0f0f0;
}
p {
    margin: 0;
    padding: 0;
```

```
    padding-bottom: 5px;
}
table {
    width: 100%;
    border-collapse: collapse;
    margin:  20px 0;
    font-size: 16px;
    text-align:  left;
}
th, td {
    padding: 12px;
    border: 1px solid #aaa;
}
th {
    background-color: #e0e0e0;
    text-align:  center;
}
tr:nth-child(even)  {
    background-color: #f9f9f9;
}
tr:nth-child(odd)  {
    background-color: #f0f0f0;
}
td {
    vertical-align:  top;
    word-break: break-word;
}



#main-body {
```

```
    font-family: Arial, sans-serif;
    padding: 10px;
    background-color: #f4f4f4;
    display: flex;
    flex-direction: column;
    align-items: center;
    width: 750px;
    margin: auto;
    height: 100vh;
    overflow: hidden;
}


#chat-container {
    scroll-behavior: smooth;
    display: flex;
    flex-direction: column;
    width: 100%;
    flex-grow: 1;
    overflow-y: auto;
    padding: 10px;
    border: 1px solid #ccc;
    background-color: white;
    border-radius: 5px;
    max-height: 100vh;
}
#chat-container::-webkit-scrollbar {
    width: 5px;
    height: 8px;
    background-color: #fff;
```

```
}
#chat-container::-webkit-scrollbar-thumb {
    background: #aaa;
}
#retry-delete-message-button {
    display: flex;
    justify-content: center;
    align-items: center;
    width: 30px;
    height: 30px;
    padding: 0px;
    border: none;
    background-color: #fff;
    color: #444;
    cursor: pointer;
    border-radius: 1.5rem;
    margin-left: 10px;
    outline: none;
    font-size: 16px;
}
#retry-delete-message-button:hover {
    background-color: #ccc;
}
.message {
    max-width: 80%;
    padding: 10px;
    border-radius: 1.5rem;
    word-wrap: break-word;
    margin: 5px;
}
```

```
.system {
    align-self:  flex-start;
    background-color: #eee;
}
.error {
    align-self:  flex-start;
    background-color: #ffbbbb;
}
.user {
    align-self:  flex-end;
    background-color: #d1e7fd;
}
.assistant {
    align-self:  flex-start;
    background-color: #d4edda;
}
.thinking  {
    align-self:  flex-start;
    background-color: #ddd;
}



.dropdown-content {
    max-height:  0;
    overflow: hidden;
    transition:  max-height  0.3s ease-in-out;
}
.expanded {
    max-height:  300px;
    overflow-y:  auto;
```

```
}
.expanded::-webkit-scrollbar  {
    width: 5px;
    height: 8px;
    background-color: #fff;
}
.expanded::-webkit-scrollbar-thumb  {
    background: #aaa;
}
.dropdown-toggle {
    display: flex;
    width: 100%;
    background: none;
    border: none;
    cursor: pointer;
    font-size: 16px;
}
.dropdown-toggle:focus {
    border: none;
    outline:  none;
}
.arrow {
    margin-left:  8px;
    width: 14px;
    height:  14px;
    transition:  transform 0.3s ease-in-out;
}
.dropdown-toggle.active .arrow {
    transform: rotate(90deg);
}
```

```
#input-container {
    position: sticky;
    display: flex;
    width: 770px;
    margin-top: 10px;
    margin-bottom: 10px;
    flex-shrink: 0;
}
#message-input {
    font-family: Arial, sans-serif;
    flex-grow: 1;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 1.5rem;
    font-size: 16px;
    resize: none;
    overflow-y: hidden;
    min-height: 40px;
    max-height: 120px;
}
#message-input:focus {
    outline: none;
    background-color: #eee;
}
.input-button {
    display: block;
    width: 40px;
    height: 40px;
```

```
            padding: 0px;
            border: none;
            background-color: #888;
            color: white;
            cursor: pointer;
            border-radius: 1.5rem;
            margin-left:  10px;
            outline:  none;
            font-size: 16px;
        }
        #send-button {
            display: block;
            background-color: #007bff;
        }
        #send-button:hover {
            background-color: #0056b3;
        }
        #stop-button {
            display: none;
            background-color: #ff0000;
        }
        #stop-button:hover {
            background-color: #b30000;
        }
    </style>
</head>

<body id="main-body">
    <div id="chat-container">
        <p class="message system">AI is loading...</p>
```

```
        </div>
        <div id="input-container">
            <textarea id="message-input" placeholder="Type a message..." rows="1"></textarea>
            <button id="send-button" class="input-button">△</button>
            <button id="stop-button" class="input-button">□</button>
        </div>

        <script>
            new QWebChannel(qt.webChannelTransport, function(channel) {
                window.bridge = channel.objects.bridge;
            });

            let chatContainer = document.getElementById('chat-container');
            let inputField = document.getElementById('message-input');
            let sendButton = document.getElementById('send-button');
            let stopButton = document.getElementById('stop-button');

            const tmpl_message = document.createElement('div');
            tmpl_message.className = 'message';
            tmpl_message.innerHTML = '...';

            const tmpl_thinking = document.createElement('div');
            tmpl_thinking.className = 'message thinking';
            tmpl_thinking.innerHTML = `
                <button class="dropdown-toggle">
                    Reasoning
                        <svg class="arrow" viewBox="0 0 24 24" fill="none" stroke="currentColor" stroke-width="2">
```

```
                        <path stroke-linecap="round" stroke-linejoin="round"
d="m8.25 4.5 7.5 7.5-7.5 7.5"></path>
                    </svg>
                </button>
                <div class="dropdown-content">...</div>
            `;

            const tmpl_retry_button = document.createElement('button');
            tmpl_retry_button.id = 'retry-delete-message-button';
            tmpl_retry_button.innerHTML = '↻';
            tmpl_retry_button.style = 'font-size: 20;'

            const tmpl_delete_button = document.createElement('button');
            tmpl_delete_button.id = 'retry-delete-message-button';
            tmpl_delete_button.innerHTML = '✕';

            const tmpl_buttons_container = document.createElement('div');
            tmpl_buttons_container.style = 'display: flex;';
            tmpl_buttons_container.appendChild(tmpl_retry_button);
            tmpl_buttons_container.appendChild(tmpl_delete_button);

            sendButton.addEventListener('click', sendMessageIn);
            inputField.addEventListener('keypress', function(event) {
                if(event.key === 'Enter' && !event.shiftKey) {
                    event.preventDefault();
                    sendMessageIn();
                }
            });
```

```
stopButton.addEventListener('click', function(event) {
    bridge.endStream();
});
tmpl_retry_button.addEventListener('click', function(event) {
    bridge.retryMessage();
});
tmpl_delete_button.addEventListener('click', function(event) {
    bridge.removeMessage();
    const messages = Array.from(chatContainer.children);

    let lastIndex = -1;
    for(let i = messages.length - 1; i >= 0; i--) {
        if(messages[i].classList.contains('user')) {
            lastIndex = i;
            break;
        }
    }

    if(lastIndex !== -1) {
        for(let i = messages.length - 1; i >= lastIndex; i--) {
            if(messages[i].classList.contains('message')) {
                messages[i].remove();
            }
        }
    }
});

inputField.addEventListener('input', function () {
    this.style.height = "auto";
    this.style.height = (this.scrollHeight) + "px";
```

```
});

function sendMessageIn() {
    let message = inputField.value.trim();
    if(message !== "") {
        bridge.sendMessage(message);
        inputField.value  = "";
        inputField.style.height  = "auto";
        inputField.focus();
    }
}

document.addEventListener('click', function(event) {
    const button = event.target.closest('.dropdown-toggle');

    if(button) {
        const content = button.nextElementSibling;

        button.classList.toggle('active');
        content.classList.toggle('expanded');

        if (!content.classList.contains('expanded')) {
            content.style.maxHeight  = null;
        }
    }
});

function addMessage(content, role) {
    if(role === 'thinking')  {
        const clone_think = tmpl_thinking.cloneNode(true);
```

```
                clone_think.querySelector('.dropdown-content').innerHTML =
content;

                chatContainer.appendChild(clone_think);
                return
            }

            const clone = tmpl_message.cloneNode(true);
            clone.classList.add(role);
            clone.innerHTML = content;

            chatContainer.appendChild(clone);
            if(role === "assistant") {
                chatContainer.appendChild(tmpl_buttons_container);
                //chatContainer.appendChild(tmpl_delete_button);
            }
        }

        function updateLastMessage(newContent) {
            const lastMessage = chatContainer.querySelector('.message:last-
child');

            if(!lastMessage) return;

            if(lastMessage.classList.contains('assistant')) {
                lastMessage.innerHTML = newContent;
            }

            if(lastMessage.classList.contains('thinking')) {
```

```
                    lastMessage.querySelector('.dropdown-content').innerHTML =
newContent;
                }
            }

            function updateLastThinkingMessage(newContent) {
                                                const messages =
chatContainer.querySelectorAll('.message.thinking');
                const lastMessage = messages[messages.length - 1];

                if(!lastMessage) return;

                const dropdown = lastMessage.querySelector('.dropdown-content');
                if(!dropdown) return;

                dropdown.innerHTML = newContent;
            }

            function updateLastRespondingMessage(newContent) {
                                                const messages =
chatContainer.querySelectorAll('.message.assistant');
                const lastMessage = messages[messages.length - 1];

                if(!lastMessage) return;

                lastMessage.innerHTML = newContent;
            }
        </script>
    </body>
</html>
```

ДОДАТОК Г (ОБОВ'ЯЗКОВИЙ)
ГРАФІЧНА ЧАСТИНА

РОЗРОБКА ЧАТ-БОТУ
«АСИСТЕНТ ДЛЯ РОЗРОБНИКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

Виконав: студент 4-го курсу,
групи 1КН-21б
спеціальності 122 «Комп'ютерні науки»
(шифр і назва напряму підготовки, спеціальності)

Крикливий О. В.
(прізвище та ініціали)

Керівник:
к.т.н., ст. вик. Петришин С. І.
(прізвище та ініціали)

«03» червня 2025 р.

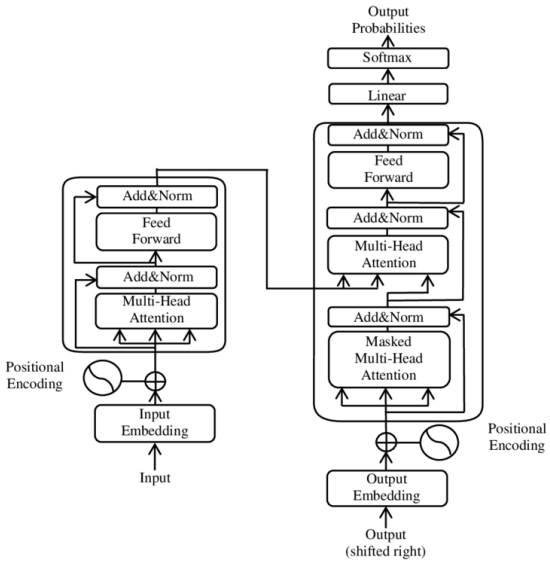

Рисунок Г.1 – Ілюстрація трансформера від Google.

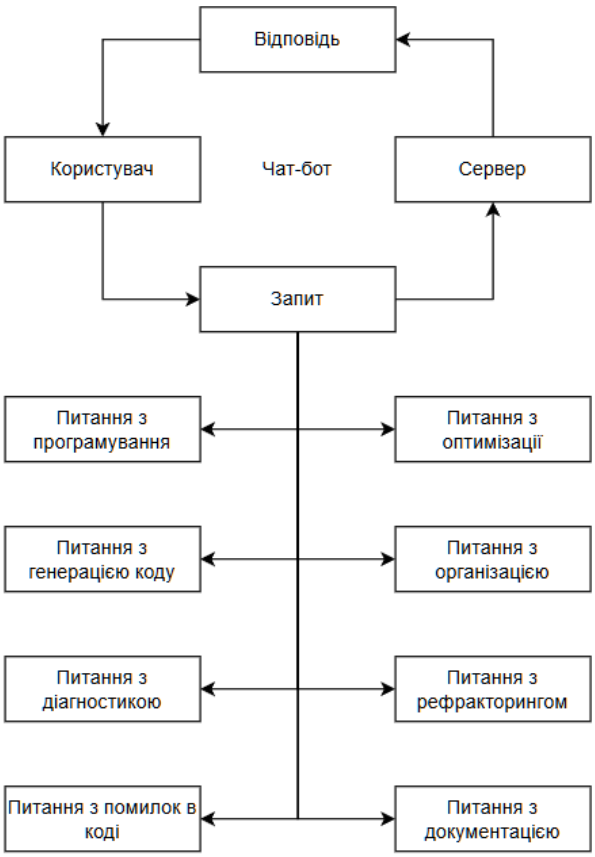

Рисунок Г.2 – Взаємодія користувача з сервером

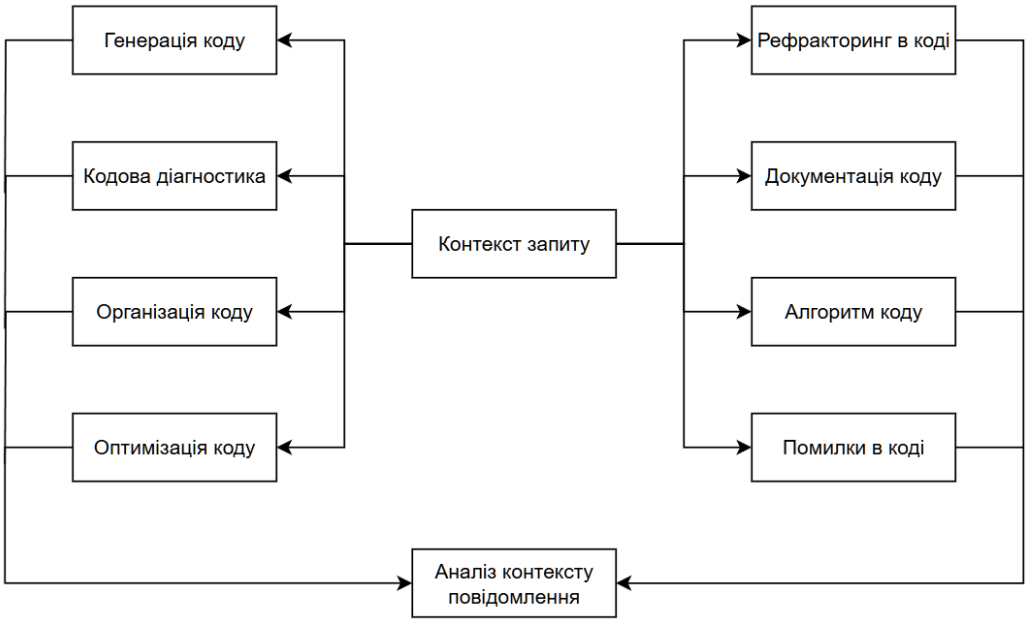

Рисунок Г.3 – Аналіз контексту повідомлень-запитів

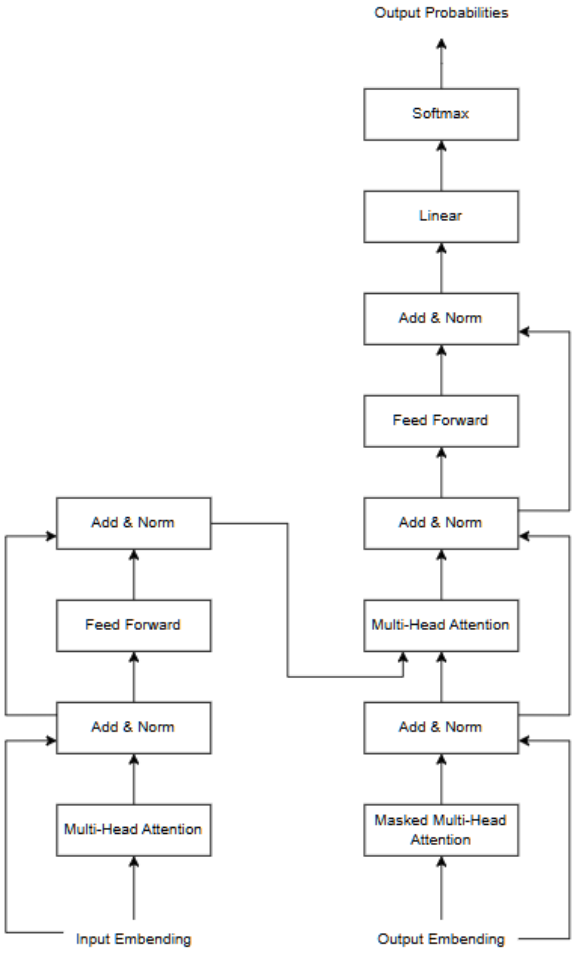

Рисунок Г.4 – Алгоритм NLP

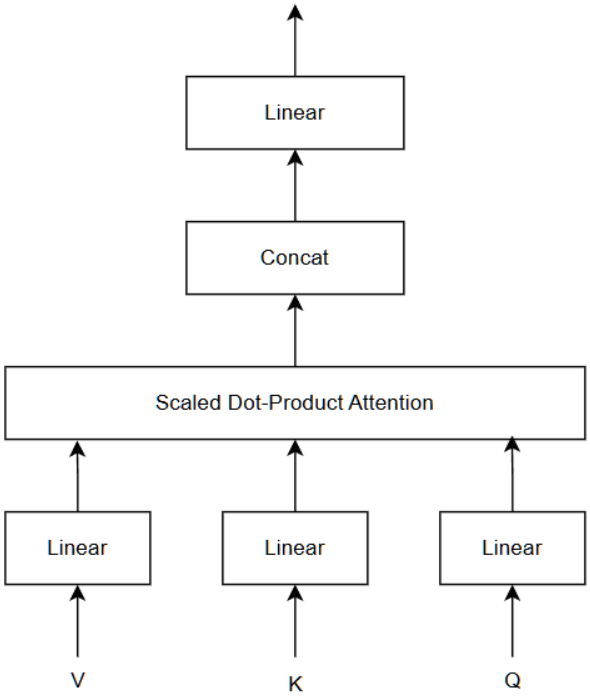

Рисунок Г.5 – Алгоритм багатоголової уваги

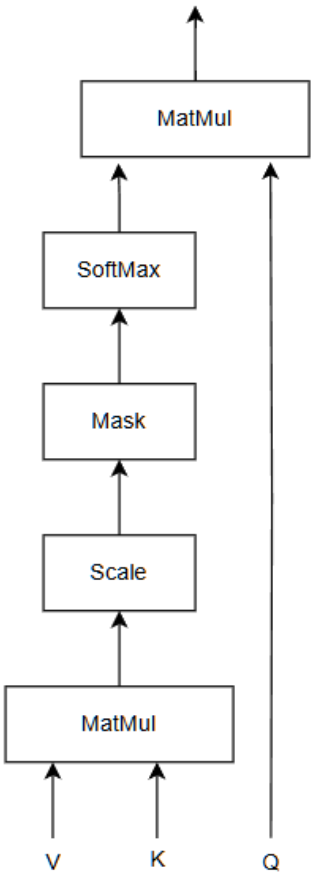

Рисунок Г.6 – Алгоритм масштабованого скалярного добутку уваги

```python
import sys
import markdown2
import json

from time import sleep, time

from PyQt5 import QtWidgets, QtWebEngineWidgets
from PyQt5.QtCore import pyqtSlot, QObject, pyqtSignal, QThreadPool, QTimer
from PyQt5.QtWebChannel import QWebChannel
```

Рисунок Г.7 – Імпорт необхідних інструментів

```
167  class ChatBot(QtWidgets.QMainWindow):
168      def __init__(self, page_html, config_parameters):···
198
199      def resizeEvent(self, a0):···
202
203      def send_message(self, content, role):···
208
209      def check_if_fully_loaded(self):···
211
212      def handle_ready_state(self, state):···
217
218      def on_page_loaded(self):···
227
228      def removeMessage(self):···
239
240      def retryMessage(self):···
283
284      def processMessage(self, text:str):···
```

Рисунок Г.8 – Абстракція класу ChatBot

```
72   class modelAI:
73       def __init__(self, config_parameters):···
85
86       def check_model_status(self) -> bool:···
95
96       def create_stream_response(self):···
115
116      def process_stream_part_response(self, part):···
```

Рисунок Г.9 – Абстракція класу modelAI

```python
class ChatBridge(QObject):
    receivedMessage = pyqtSignal(str)
    stop_flag = False
    regenerateMessage = pyqtSignal()
    deleteMessage = pyqtSignal()

    @pyqtSlot(str)
    def sendMessage(self, message):
        self.receivedMessage.emit(message)

    @pyqtSlot()
    def endStream(self):
        self.stop_flag = True

    @pyqtSlot()
    def retryMessage(self):
        self.regenerateMessage.emit()

    @pyqtSlot()
    def removeMessage(self):
        self.deleteMessage.emit()
```

Рисунок Г.10 – Реалізація класу ChatBridge

```python
class ChatBot(QtWidgets.QMainWindow):
    def __init__(self, page_html, config_parameters):
        super().__init__()

        self.setWindowTitle("ChatBot - Assistant of Software Developer")
        self.setMinimumSize(800, 600)

        self.the_model = modelAI(config_parameters)

        central_widget = QtWidgets.QWidget()
        self.setCentralWidget(central_widget)
        main_layout = QtWidgets.QVBoxLayout(central_widget)

        self.html_view = QtWebEngineWidgets.QWebEngineView()
        main_layout.addWidget(self.html_view)
        self.html_view.setHtml(page_html)
        self.check_if_fully_loaded()

        settings = self.html_view.settings()
        settings.setAttribute(QtWebEngineWidgets.QWebEngineSettings.WebAttribute.ScrollAnimatorEnabled, True)

        self.bridge = ChatBridge()
        self.bridge.receivedMessage.connect(self.processMessage)
        self.bridge.regenerateMessage.connect(self.retryMessage)
        self.bridge.deleteMessage.connect(self.removeMessage)
        self.channel = QWebChannel()
        self.channel.registerObject("bridge", self.bridge)
        self.html_view.page().setWebChannel(self.channel)

        self.needs_update = False
```

Рисунок Г.11 – Реалізація функції init класу ChatBot

```
<html>
    <head>
        <script src="qrc:///qtwebchannel/qwebchannel.js"></script>
        <style>···
        </style>
    </head>

    <body id="main-body">
        <div id="chat-container">
            <p class="message system">AI is loading...</p>
        </div>
        <div id="input-container">
            <textarea id="message-input" placeholder="Type a message..." rows="1"></textarea>
            <button id="send-button" class="input-button">△</button>
            <button id="stop-button" class="input-button">□</button>
        </div>

        <script>···
        </script>
    </body>
</html>
```

Рисунок Г.12 – Головна сторінка HTML

```
def message_refiner(input):
    text = markdown2.markdown(input, extras=["extra",
                                             "tables",
                                             "fenced-code-blocks",
                                             "task_list",
                                             "strike",
                                             "cuddled-lists"])
    result = json.dumps(text)
    return result
```

Рисунок Г.13 – Реалізація використання markdown2

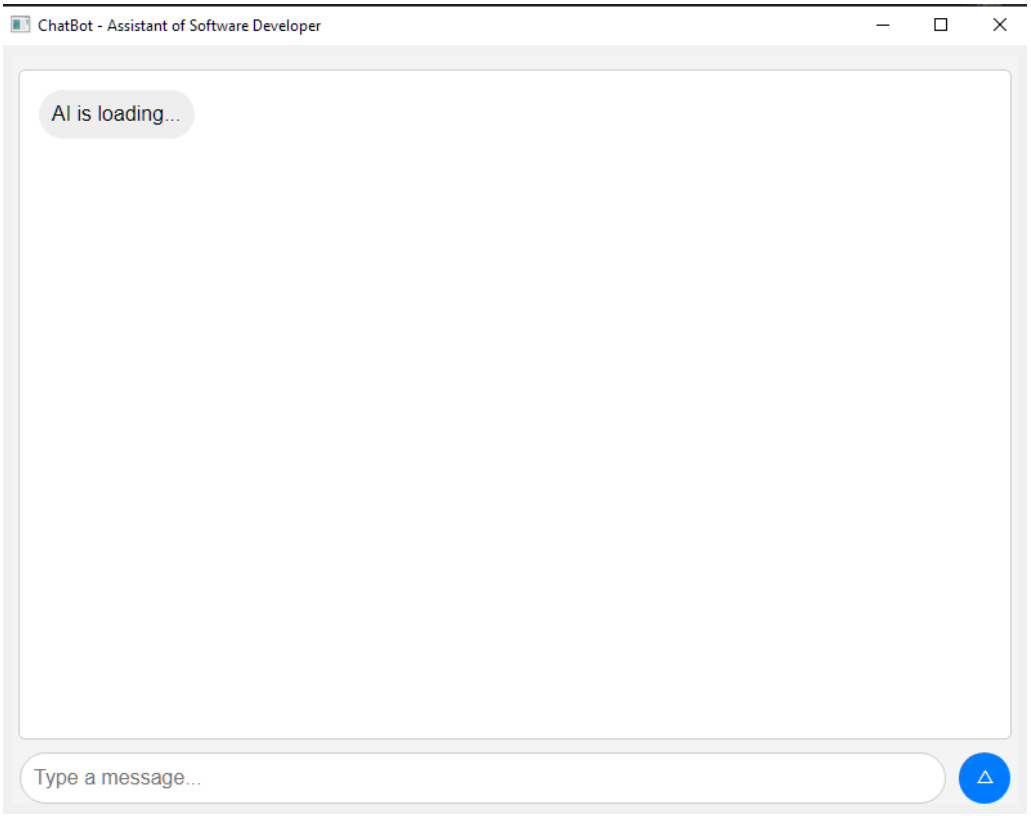

Рисунок Г.14 – Тестовий запуск програми

```
new QWebChannel(qt.webChannelTransport, function(channel) {
    window.bridge = channel.objects.bridge;
});

let chatContainer = document.getElementById('chat-container');
let inputField = document.getElementById('message-input');
let sendButton = document.getElementById('send-button');
let stopButton = document.getElementById('stop-button');

const tmpl_message = document.createElement('div');
tmpl_message.className = 'message';
tmpl_message.innerHTML = '...';

const tmpl_thinking = document.createElement('div');
tmpl_thinking.className = 'message thinking';
tmpl_thinking.innerHTML = `
    <button class="dropdown-toggle">
        Reasoning
        <svg class="arrow" viewBox="0 0 24 24" fill="none" stroke="currentColor" stroke-width="2">
            <path stroke-linecap="round" stroke-linejoin="round" d="m8.25 4.5 7.5 7.5-7.5 7.5"></path>
        </svg>
    </button>
    <div class="dropdown-content">...</div>
`;

const tmpl_retry_button = document.createElement('button');
tmpl_retry_button.id = 'retry-delete-message-button';
tmpl_retry_button.innerHTML = '↻';
tmpl_retry_button.style = 'font-size: 20;'

const tmpl_delete_button = document.createElement('button');
tmpl_delete_button.id = 'retry-delete-message-button';
tmpl_delete_button.innerHTML = '✕';

const tmpl_buttons_container = document.createElement('div');
tmpl_buttons_container.style = 'display: flex;';
tmpl_buttons_container.appendChild(tmpl_retry_button);
tmpl_buttons_container.appendChild(tmpl_delete_button);
```

Рисунок Г.15 – Початкова реалізація динаміки сторінки

```
        sendButton.addEventListener('click', sendMessageIn);
        inputField.addEventListener('keypress', function(event) {···

        stopButton.addEventListener('click', function(event) {···
        tmpl_retry_button.addEventListener('click', function(event) {···
        tmpl_delete_button.addEventListener('click', function(event) {···

        inputField.addEventListener('input', function () {···

        function sendMessageIn() {···

        document.addEventListener('click', function(event) {
            const button = event.target.closest('.dropdown-toggle');

            if(button) {
                const content = button.nextElementSibling;

                button.classList.toggle('active');
                content.classList.toggle('expanded');

                if (!content.classList.contains('expanded')) {
                    content.style.maxHeight = null;
                }
            }
        });

        function addMessage(content, role) {···

        function updateLastMessage(newContent) {···

        function updateLastThinkingMessage(newContent) {···

        function updateLastRespondingMessage(newContent) {···
    </script>
```

Рисунок Г.16 – Кінцева реалізацію динаміки сторінки

```
72  class modelAI:
73      def __init__(self, config_parameters):
74          self.__dict__.update(config_parameters)
75
76          self.chat_history = [{"role": "system", "content": "AI is loading..."}]
77
78          self.client = OpenAI(
79              base_url=self.BASE_URL,
80              api_key=self.API_KEY,
81          )
82
83          self.current_thinking = ""
84          self.current_response = ""
```

Рисунок Г.17 – Реалізація фунції init класу modelAI

```python
    def check_model_status(self) -> bool:
        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=[{"role": "user", "content": "Ping"}],
            max_tokens=1
        )
        if response:
            return True
        return False
```

Рисунок Г.18 – Реаліація перевірки чат-бота

```python
    def create_stream_response(self):
        """Outputs: (reasoning|None, content|None)"""

        chat_messages = [x for x in self.chat_history if x["role"] in ("assistant", "user")]
        chat_messages.append({"role": "system", "content": self.BIAS_PROMPT})

        response = self.client.chat.completions.create(
            model=self.AI_MODEL,
            messages=chat_messages,
            stream=True
        )

        for chunk in response:
            reasoning = getattr(chunk.choices[0].delta, "reasoning", None)
            content = getattr(chunk.choices[0].delta, "content", None)

            reason = '' if reasoning is None else reasoning

            yield (reason, content)
```

Рисунок Г.19 – Реалізація генератора чат-бота

```python
def process_stream_part_response(self, part):
    thinking, content = part

    if thinking:
        self.current_thinking += thinking
    if content:
        self.current_response += content

    if len(self.chat_history) >= 2 and self.chat_history[-2]["role"] == "thinking" and self.chat_history[-1]["role"] == "assistant":
        self.chat_history[-2]["content"] = self.current_thinking
        self.chat_history[-1]["content"] = self.current_response
    else:
        thinking_entry = {"role": "thinking", "content": self.current_thinking}
        response_entry = {"role": "assistant", "content": self.current_response}
        self.chat_history.append(thinking_entry)
        self.chat_history.append(response_entry)
```

Рисунок Г.20 – Реалізація опрацювання дельти генератора

```
289     def processMessage(self, text:str):
290         if text:
291             self.the_model.chat_history.append({"role": "user", "content": "\n".join(line+"\n" for line in text.split("\n"))})
292 
293             self.the_model.current_thinking = ""
294             self.the_model.current_response = ""
295             self.generator = self.the_model.create_stream_response()
296 
297             def generate_response():
298                 js_code = f"""
299                 sendButton.style.display = 'none';
300                 stopButton.style.display = 'block';
301                 tmpl_retry_button.style.display = 'none';
302                 tmpl_delete_button.style.display = 'none';
303                 addMessage({message_refiner(self.the_model.chat_history[-1]["content"])}, 'user');
304                 addMessage("...", 'thinking');
305                 addMessage("...", 'assistant');
306                 """
307                 self.html_view.page().runJavaScript(js_code)
```

Рисунок Г.21 – Ініціалізація функції processMessage()

```
309                 try:
310                     for part in self.generator:
311                         if self.bridge.stop_flag:
312                             break
313 
314                         countT += 1
315 
316                         self.the_model.process_stream_part_response(part)
317                         js_code_part = f"""
318                         updateLastThinkingMessage({message_refiner(self.the_model.chat_history[-2]["content"])});
319                         updateLastRespondingMessage({message_refiner(self.the_model.chat_history[-1]["content"])});
320                         """
321                         self.html_view.page().runJavaScript(js_code_part)
322 
323                 except Exception as e:
324                     self.send_message(f"Error: {e}", "error")
```

Рисунок Г.22 – Створення генератора з modelAI

```
334                     self.generator = None
335                     self.bridge.stop_flag = False
336 
337                     js_code = f"""
338                     sendButton.style.display = 'block';
339                     stopButton.style.display = 'none';
340                     tmpl_retry_button.style.display = 'flex';
341                     tmpl_delete_button.style.display = 'flex';
342                     """
343                     self.html_view.page().runJavaScript(js_code)
344 
345                 QThreadPool.globalInstance().start(generate_response)
```

Рисунок Г.23 – Завершення генерації повідомлення

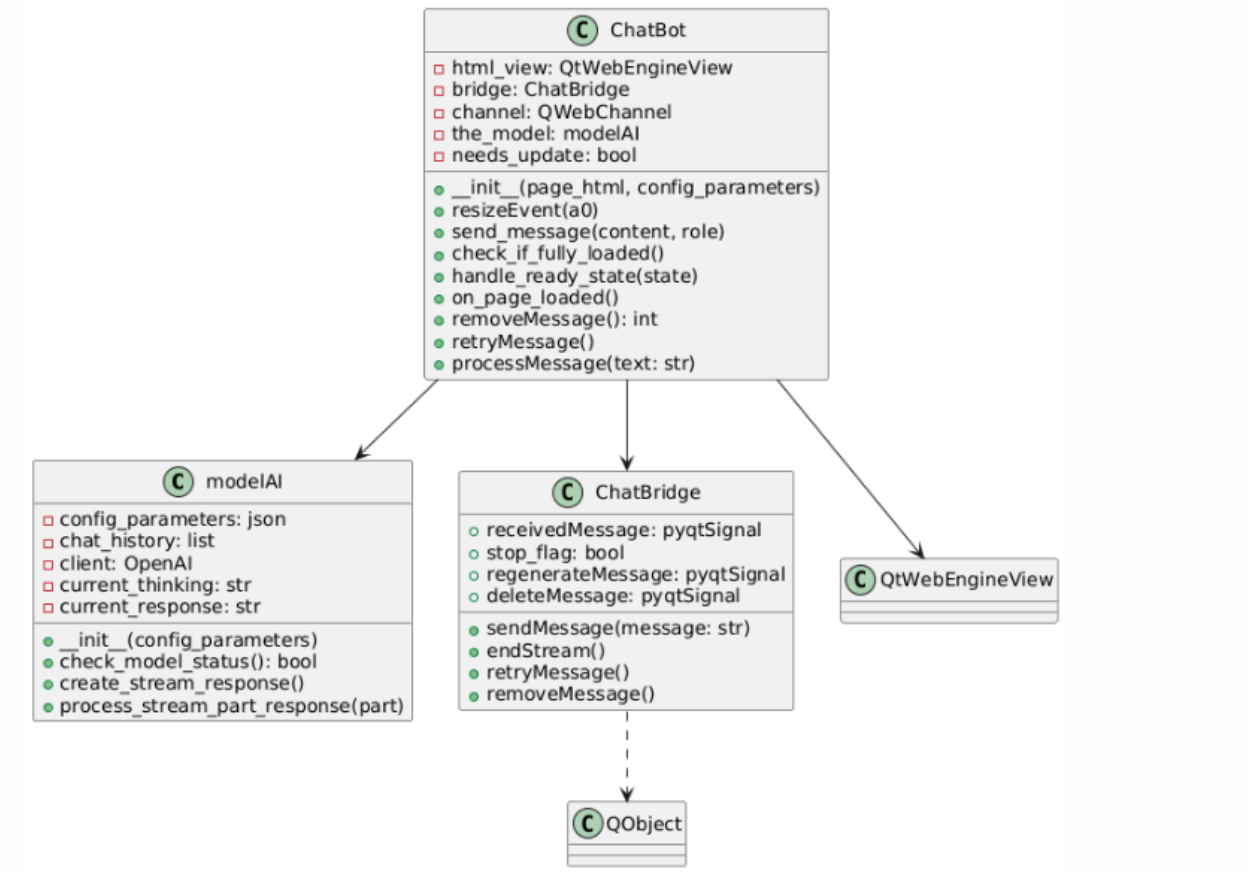

Рисунок Г.24 – UML діаграма класів чат-боту

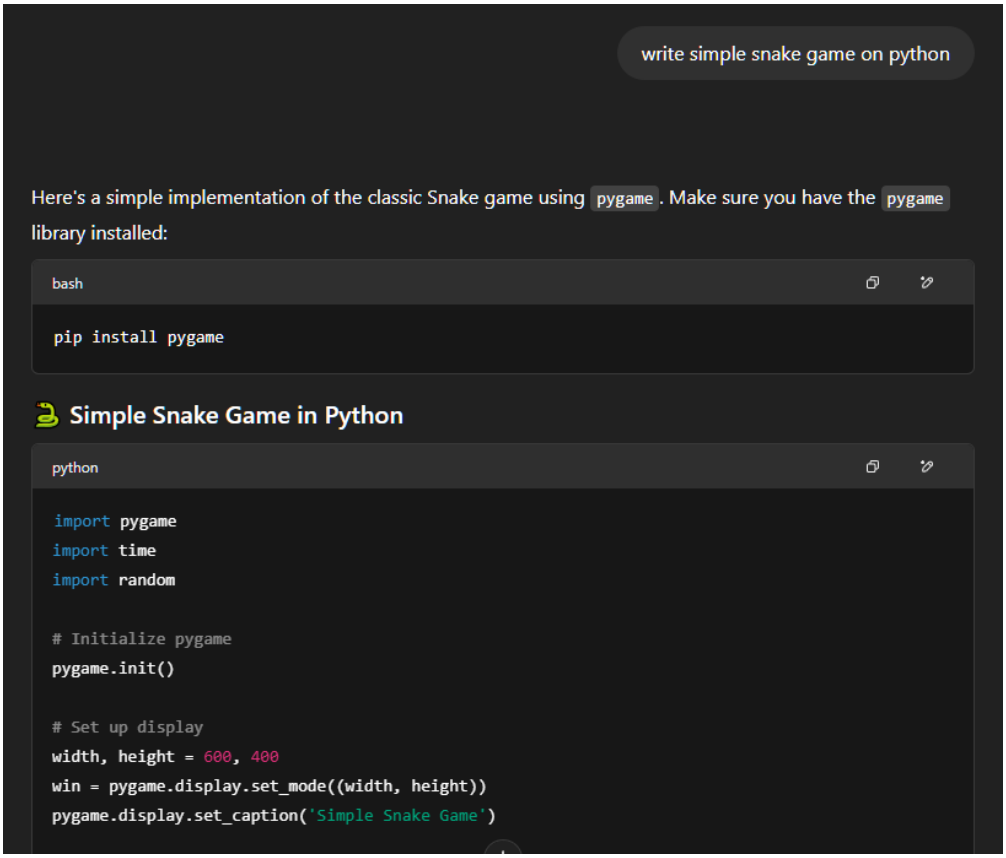

Рисунок Г.25 – Тестування ChatGPT

write simple snake game on python

Reasoning >

Here's a simple Snake game implemented using Python's `turtle` graphics library:

```
import turtle
import time
import random

# Set up the screen
win = turtle.Screen()
win.title("Snake Game by AoSD")
win.bgcolor("green")
win.setup(width=600, height=600)
win.tracer(0)  # Disable automatic screen updates

# Snake head
head = turtle.Turtle()
head.speed(0)
head.shape("square")
head.color("black")
head.penup()
head.goto(0, 0)
head.direction = "stop"

# Snake food
food = turtle.Turtle()
food.speed(0)
```

Рисунок Г.26 – Тестування AoSD

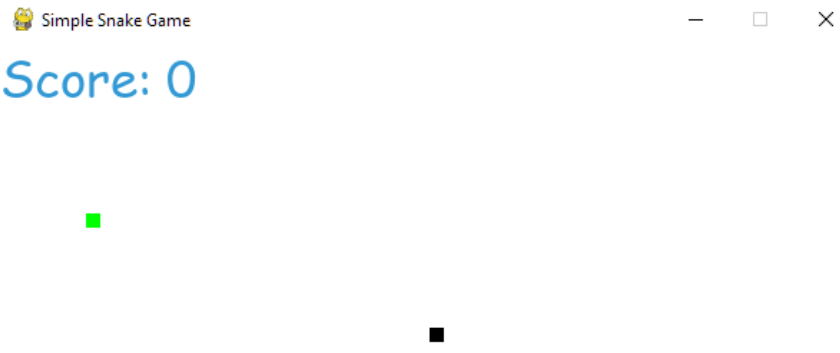

Рисунок Г.27 – Результат роботи програми від ChatGPT

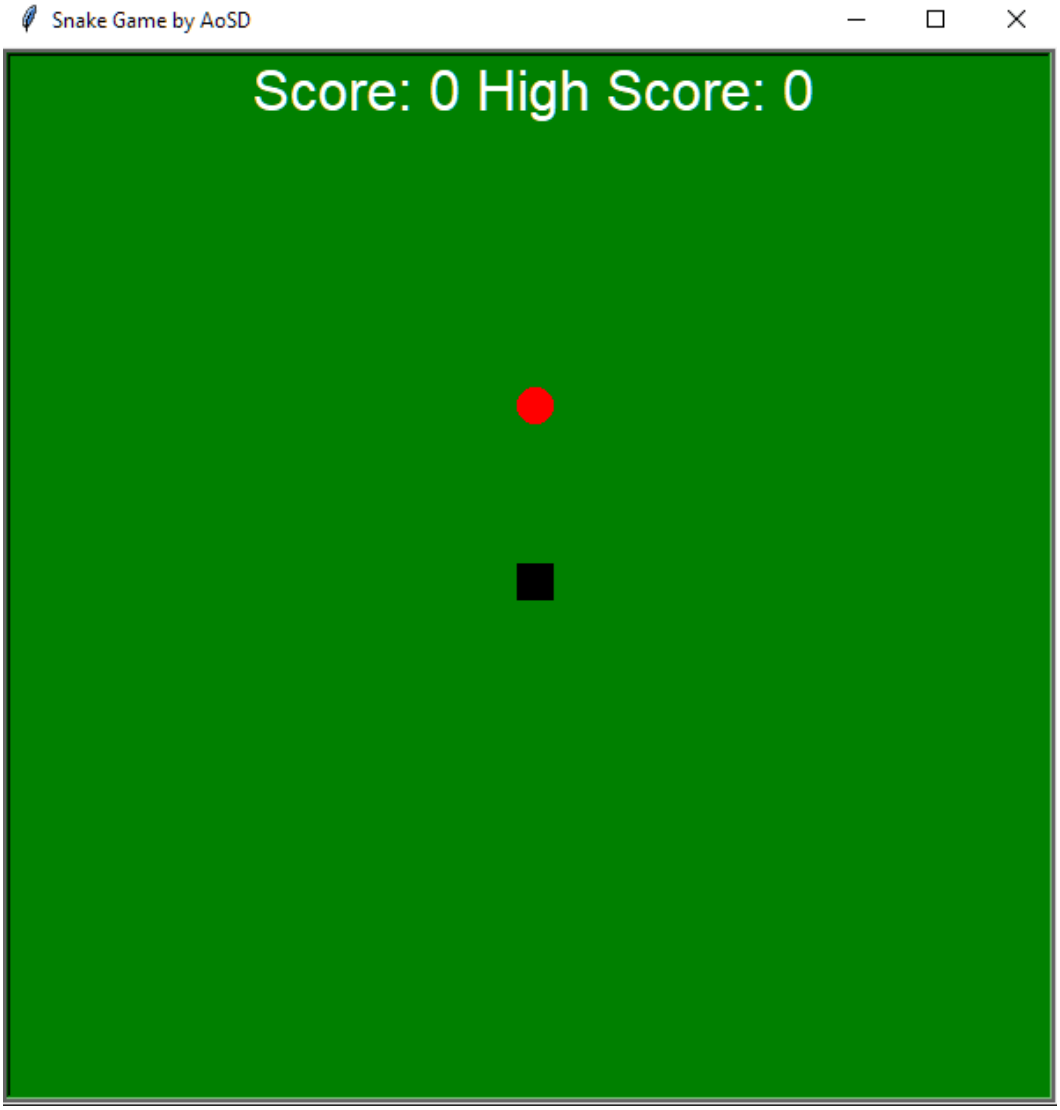

Рисунок Г.28 – Результат роботи програми від AoSD

ДОДАТОК Д (ДОВІДНИКОВИЙ)
ДОВІДКА ПРО ВПРОВАДЖЕННЯ

МАЛЕ НАУКОВО-ВИРОБНИЧЕ ПІДПРИЄМСТВО "ТОВ "ІТІ"
Україна, 21021, м.Вінниця, вул. Келецька, 56

№ 47 від "6" 06 2025 р.

ДОВІДКА

Дана Крикливому Олексію Володимировичу в тому, що результати, одержані ним в процесі виконання бакалаврської кваліфікаційної роботи, а саме чат-бот «Асистент для розробника програмного забезпечення», планується використати в розробках ТОВ «ІТІ».

Зам. директора ТОВ «ІТІ» Бодяк В.М.