Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка автономного ШІ агента на основі блокчейну

Виконав(ла): студент(ка) 4 курсу, групи СП-41
спеціальності 121
Інженерія програмного забезпечення
(шифр і назва спеціальності)

_____ Басара А.М.
(підпис) (прізвище та ініціали)

Керівник _____ Бойко І. В.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль 2025

АНОТАЦІЯ

Розробка автономного ШІ агента на основі блокчейну // Кваліфікаційна робота освітнього рівня «Бакалавр» // Басара Андрій Миколайович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-41 // Тернопіль, 2025 // С. - 81, рис. - 25, табл. - 0, додат. - 3, бібліогр.-.10.

Ключові слова: Велика мовна модель, агентна система, вивчення іноземних мов, блокчейн, навчальний контент, валідація помилок, механізм голосування, децентралізація, модерація контенту, обробка природної мови.

На початковому етапі було проведено ретельний аналіз сучасних підходів до вивчення мов, технологій LLM та механізмів децентралізованої валідації. На основі цього аналізу було розроблено концепцію мультиагентної системи, в якій агенти, що працюють на основі LLM, забезпечують персоналізований досвід вивчення іноземних мов. Ключовим нововведенням є система голосування на основі блокчейну, де користувачі та агенти перевіряють помилки контенту, забезпечуючи якість за допомогою прозорої децентралізованої модерації.

На етапі проектування архітектура системи була розроблена з акцентом на модульність агентів, безпечні протоколи взаємодії та ефективну інтеграцію з блокчейном. Для автоматизації процесу голосування та валідації були використані смарт-контракти.

На етапі реалізації були розроблені основні компоненти, включаючи інтерфейс агента LLM, конвеєр валідації контенту та рівень взаємодії з блокчейном. Фінальне тестування включає аналіз зручність використання, коректність механіки голосування, перевірку реакції агента під час імітації користувацьких навантажень. Результати продемонстрували, що додаток успішно відповідає встановленим вимогам, забезпечуючи стабільну, високопродуктивну та інтуїтивно зрозумілу роботу користувачів.

ABSTRACT

Development of an autonomous agent based on blockchain // Qualification work for the educational level "Bachelor" / Basara Andrii Mykolaiovych // Ternopil National Technical University named after Ivan Puluj, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-41 // Ternopil, 2025 // Pages. - 81, figures. - 25, table - 0, addition - 3, literature. - 10.

Keywords: Large Language Model, agent system, foreign language learning, blockchain, educational content, mistake validation, voting mechanism, decentralization, content moderation, natural language processing.

At the initial stage, a thorough analysis of modern approaches to language learning, LLM technologies, and decentralized validation mechanisms was carried out. Based on this analysis, a concept for a multi-agent system was developed, where agents powered by LLMs provide personalized foreign language learning experiences. A key innovation is a blockchain-based voting system where users and agents validate content errors, ensuring quality through transparent, decentralized moderation.

During the design phase, the system architecture was developed with an emphasis on agent modularity, secure interaction protocols, and efficient blockchain integration. Smart contracts were used to automate the voting and validation process.

In the implementation stage, core components were developed, including the LLM agent interface, content validation pipeline, and the blockchain interaction layer. Final testing included usability analysis, correctness of voting mechanics, agent response validation, and performance benchmarks under simulated user loads. The results demonstrated that the application successfully met the specified requirements, delivering a stable, high-performing, and intuitive user experience.

ЗМІСТ

ВСТУП

За останні роки ландшафт освіти зазнав суттєвих трансформацій завдяки розвитку цифрових технологій і широкому розповсюдженню інтернету. Попит на інтелектуальні та адаптивні рішення для вивчення іноземних мов стимулював створення агентів, які здатні забезпечити персоналізований підхід до навчання. Як наслідок, розробка таких систем стала ключовим напрямком у сучасних освітніх технологіях, що задовольняють потреби як учнів, так і викладачів.

Перехід від традиційних методів навчання до цифрових платформ був зумовлений необхідністю у більш гнучких та інтерактивних рішеннях. Традиційні підходи, часто обмежені фіксованим графіком і місцем, доповнюються LLM агентами, які забезпечують адаптивне навчання з урахуванням індивідуальних особливостей користувачів. Такі системи відіграють важливу роль у трансформації навчального процесу, надаючи інтерактивні засоби для пояснення граматики, тренування словникового запасу, рольових діалогів і мотивації.

Незважаючи на прогрес у розвитку освітніх технологій, існують суттєві виклики, особливо у сфері синхронізації роботи різних агентів та ефективної адаптації контенту до потреб користувача. Часто відсутність інтегрованого підходу ускладнює процес навчання, знижуючи його ефективність і мотивацію. Цей дипломний проєкт має на меті розробити мультиагентну систему для вивчення іноземних мов, яка інтегрує одразу кілька окремих агентів. Система реалізована з використанням сучасних методів обробки знань і адаптивного навчання, забезпечуючи безперебійну та ефективну взаємодію між агентами і користувачем.

Ця робота має потенціал покращити процес вивчення іноземних мов, усунувши обмеження існуючих цифрових платформ. Використання агентного підходу та сучасних технологій обробки знань дозволить створити більш динамічну, інтерактивну та адаптивну освітню систему, що відповідає потребам сучасних користувачів.

Розробка мультиагентної системи, яка ефективно поєднує інтелектуальні агенти з різними функціями — від пояснення граматики до надання мотивації — сприятиме розвитку нових методів цифрового навчання. Запропоноване рішення може стати моделлю для майбутніх освітніх систем, демонструючи переваги гнучкого, персоналізованого та масштабованого підходу до навчання іноземним мовам.

Основною метою цієї роботи є аналіз та розробка агентної системи для вивчення іноземних мов із використанням сучасних технологій штучного інтелекту та обробки знань. Система має на меті забезпечити безпечну автентифікацію користувачів, інтуїтивний інтерфейс і комплексну підтримку навчального процесу через взаємодію формальних, розмовних, лексичних і мотиваційних агентів. Ця кваліфікаційна робота спрямована на демонстрацію потенціалу такої систем у сфері цифрової освіти.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕХНОЛОГІЙ

1.1 Огляд конкурентів

Сьогодні доволі багато застосунків для вивчення іноземних мов, які активно використовують великі мовні моделі (LLM) для персоналізації та покращення навчального процесу. Але не всі з них використовують найбільш сучасні підходи, або ж застосовують LLM лише як додаток до основного додатку. Було виділено трьох основних конкурентів, які використовують схожий підхід.

Heylama позиціонує себе як інструмент для розвитку розмовних навичок за допомогою персонального AI-тренера. Користувачі можуть вести діалоги на різноманітні теми, отримуючи детальні підказки щодо граматики, лексики та вимови. Особливістю Heylama є можливість створювати власні рольові ігри та інтерактивні вправи, що дозволяє адаптувати навчання під власні потреби. Система також використовує інтервальне повторення для закріплення нової лексики, що сприяє ефективному засвоєнню матеріалу [1].

Рисунок 1.1 – Логотип Heylama

Loora AI орієнтований насамперед на тих, хто вивчає англійську мову, і пропонує унікальний досвід завдяки поєднанню відкритих діалогів із структурованими уроками. Застосунок дозволяє практикувати англійську у реальних життєвих ситуаціях, таких як ділове спілкування, співбесіди чи повсякденні розмови. Вбудований штучний інтелект надає миттєвий фідбек щодо

вимови та граматики, а також підлаштовує навчальний контент відповідно до цілей і рівня підготовки користувача. Такий підхід створює безпечне середовище для навчання, де можна вільно експериментувати з мовою без страху зробити помилку [2].

Рисунок 1.2 – Логотип Loora AI

TalkPal вирізняється багатомовністю та широким вибором інтерактивних режимів навчання. Застосунок пропонує користувачам рольові ігри, симуляції реальних ситуацій, дебати та навіть спілкування з історичними або вигаданими персонажами. Завдяки потужному AI TalkPal забезпечує аналіз відповідей у режимі реального часу, надає рекомендації щодо покращення мовлення та дозволяє обирати теми для розмов відповідно до інтересів користувача. Така гнучкість і різноманіття навчальних сценаріїв роблять TalkPal привабливим вибором для тих, хто прагне розвивати мовні навички у різних комунікативних контекстах [3].

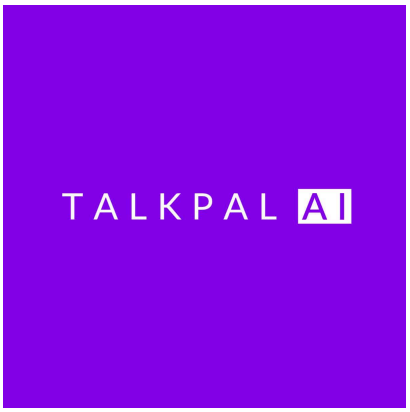


Рисунок 1.3 – Логотип TalkPal

1.2.1 Обґрунтування вибору напрямку дослідження та технологій

Вибір інструментів був зумовлений сучасними тенденціями у сфері штучного інтелекту та практичними міркуваннями щодо ефективності розробки.

Однією з ключових причин вибору мови програмування Python стало те, що абсолютна більшість сучасних фреймворків для роботи з великими мовними моделями (LLM) створені саме на цій мові. Python є галузевим стандартом для розробки у сфері штучного інтелекту завдяки простому синтаксису, високій читабельності коду, а також величезній кількості спеціалізованих бібліотек і фреймворків. Python дозволяє розробникам зосередитися на логіці та інноваціях, а не на технічних деталях реалізації, оскільки більшість складних обчислювальних процесів реалізовані на низькорівневих мовах (наприклад, C++ або Rust) і приховані за простим інтерфейсом.

Для побудови мультиагентної архітектури було обрано фреймворк LangGraph, який також написаний на Python і органічно інтегрується з іншими популярними бібліотеками для роботи з LLM. LangGraph дозволяє швидко створювати складні агентні системи, моделювати взаємодію між агентами, інтегрувати зовнішні джерела даних і масштабувати рішення під різні задачі.

Окремо слід зазначити вибір технології блокчейн для фіксації досягнень користувачів. Блокчейн забезпечує незмінність, прозорість і захищеність даних, що особливо важливо для зберігання результатів взаємодії з мультиагентною системою. Саме ці властивості блокчейну дозволяють гарантувати довіру до збережених досягнень та унеможливлюють їх фальсифікацію. Нижче наведено більше деталей стосовно використаних технологій.

1.2.2 Великі мовні моделі (LLM).

В основі сучасних великих мовних моделей лежить трансформерна архітектура, представлена в роботі "Attention Is All You Need" (Vaswani et al.,

2017). Ця архітектура замінила попередні рекурентні нейронні мережі (RNN) завдяки своїй здатності обробляти послідовності даних паралельно, що значно підвищує ефективність навчання та обробки інформації.

Ключові компоненти трансформера:

- Механізм самоуваги (Self-Attention): Дозволяє кожному елементу входу взаємодіяти з усіма іншими елементами, оцінюючи їхню важливість для поточного кроку обробки. Це забезпечує глибше розуміння контексту та взаємозв'язків у тексті.
- Багатоголова увага (Multi-Head Attention). Паралельно обробляє інформацію з різних підпросторів, що дозволяє моделі враховувати різні аспекти контексту.
- Позиційне кодування. Оскільки трансформери не мають вбудованого порядку обробки елементів, додається інформація про позицію кожного елемента в послідовності, що дозволяє моделі враховувати порядок слів у реченні.

Завдяки цим характеристикам трансформери стали основою для таких моделей, як GPT, BERT та їхніх похідних, що забезпечують високі результати в завданнях обробки природної мови. В моєму випадку була використана модель ChatGPT o4-mini [4].

1.2.3 Retrieval-Augmented Generation (RAG).

Цей підхід є інноваційним та поєднує можливості великих мовних моделей (LLM) із зовнішніми джерелами інформації, що дозволяє суттєво підвищити якість, точність і актуальність генерованих відповідей. На відміну від традиційних LLM, які працюють лише з власними навчальними даними, RAG інтегрує механізми пошуку та використання зовнішніх знань — документів, баз даних, веб-сторінок чи спеціалізованих репозиторіїв. Це особливо корисно для завдань,

які вимагають доступу до актуальних або вузькоспеціалізованих знань, відсутніх у навчальних даних моделі

Основні етапи RAG:

- Отримання інформації (Retrieval). Система отримує запит користувача, перетворює його у векторне представлення та здійснює пошук у зовнішніх джерелах даних (наприклад, документах, базах знань, веб-сторінках) для отримання релевантної інформації.
- Аугментація (Augmentation). Відібрана інформація додається до запиту користувача, формуючи розширений контекст. Це дає змогу мовній моделі враховувати не лише власні знання, а й актуальні зовнішні дані
- Генерація відповіді (Generation). Отримана інформація використовується для формування точної та контекстуально релевантної відповіді [5].

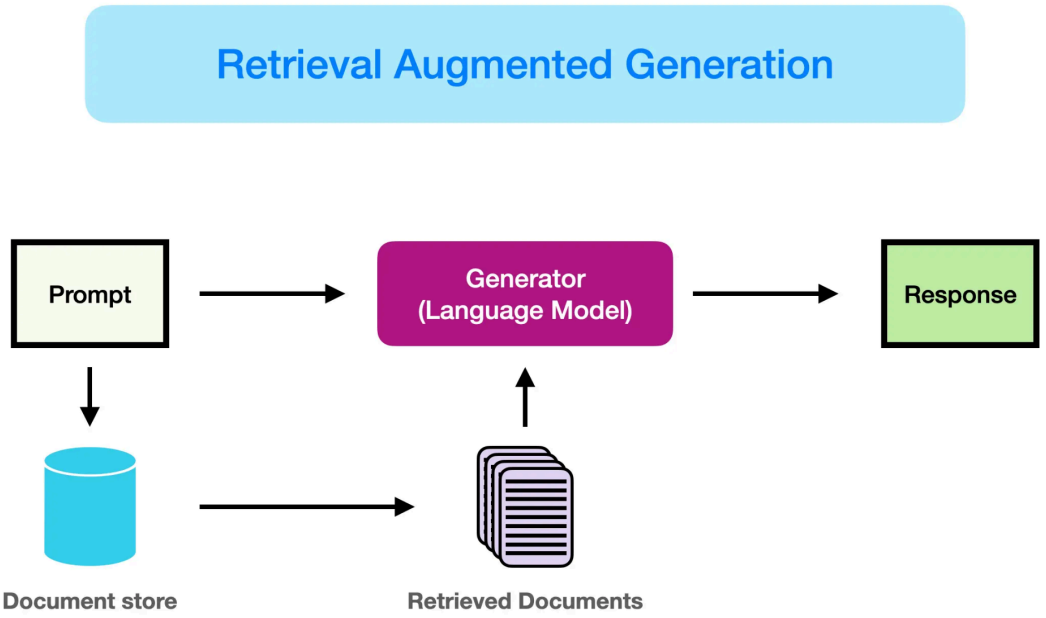


Рисунок 1.4 – Ілюстрація роботи RAG

RAG технологія дозволяє знижувати ризик виникнення "галюцинацій" (неправдивої або неточної інформації) в генерованих текстах, оскільки модель спирається на перевірені джерела даних. Крім того, RAG дозволяє інтегрувати специфічні знання без необхідності переобучення великих мовних моделей, що

робить його ефективним інструментом для створення адаптованих рішень у різних галузях.

1.2.4 LangGraph.

LangGraph це сучасний open-source фреймворк для побудови складних багатокомпонентних агентних систем на основі LLM (Large Language Models). Його ключова особливість — оркестрація агентів у вигляді графу, де кожен вузол відповідає за окрему функцію чи агента, а зв’язки (edges) визначають логіку переходів і передачу даних між ними. Така архітектура дозволяє створювати гнучкі, адаптивні та надійні системи, які легко масштабуються й ефективно управляють складними сценаріями взаємодії. Доцільність використання LangGraph у мультиагентних системах полягає у таких можливостях:

- Графова оркестрація та контроль потоків: LangGraph дозволяє будувати як лінійні, так і циклічні (ітеративні) робочі процеси, що особливо важливо для освітніх застосунків, де можуть бути потрібні повторні перевірки знань, адаптація до рівня користувача чи повернення до попередніх етапів навчання.
- Становість і пам’ять: Фреймворк забезпечує збереження та управління станом системи на всіх етапах взаємодії. Це дозволяє не лише підтримувати контекст у довготривалих сесіях, а й реалізовувати персоналізовані навчальні траєкторії для кожного користувача.
- Людина в циклі (human-in-the-loop): LangGraph спрощує інтеграцію механізмів ручної модерації, контролю якості та втручання викладача або адміністратора на критичних етапах, що підвищує надійність і довіру до системи.

- Масштабованість і продуктивність: Завдяки модульній структурі LangGraph легко розширювати систему новими агентами або функціями, інтеграції з іншими сервісами.
- Гнучкість у налаштуванні: Розробники можуть визначати власні правила переходів, створювати умовні гілки, налаштовувати поведінку агентів залежно від результатів проміжних етапів, що дає змогу реалізовувати як стандартні, так і інноваційні педагогічні сценарії [6].

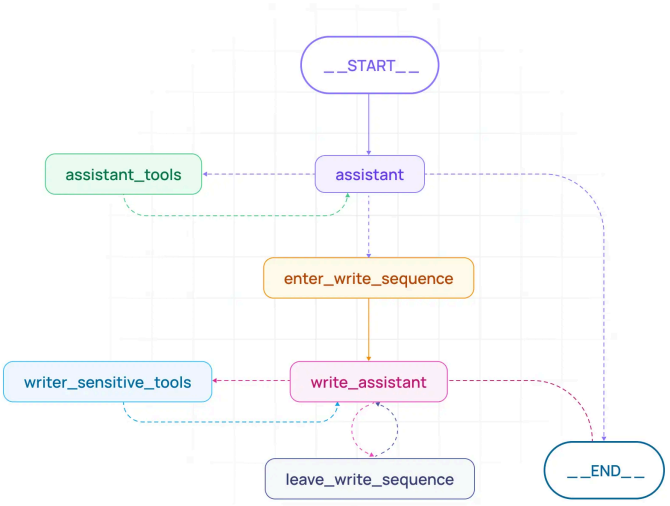


Рисунок 1.5 - Приклад графу мультиагента

Завдяки цим можливостям LangGraph підходить для створення агентних систем у сфері вивчення іноземних мов, де важливо забезпечити гнучку взаємодію між спеціалізованими агентами (наприклад, пояснювачем граматики, тренером словникового запасу, партнером для діалогу), підтримувати персоналізований контекст користувача, інтегрувати сучасні освітні практики та гарантувати високу якість і надійність навчального процесу.

1.2.5 Gradio.

Gradio — це open-source Python бібліотека, яка дозволяє швидко створювати інтерактивні веб-інтерфейси для машинного навчання, LLM-агентів і різноманітних AI-додатків. Основні переваги Gradio у контексті вашої роботи:

- Швидке створення UI: Достатньо кількох рядків коду, щоб отримати повноцінний веб-інтерфейс для взаємодії з мультиагентною системою — без необхідності писати HTML, JS чи налаштовувати сервери.
- Підтримка різних типів даних: Текст, зображення, аудіо, відео, файли — це дозволяє реалізувати багатофункціональні навчальні інтерфейси, наприклад, для тренування словникового запасу чи розмовної практики.
- Можливість спільного використання: Інтерфейс можна запускати локально, вбудовувати в Jupyter Notebook або швидко розгортати у хмарі (наприклад, через Hugging Face Spaces), що спрощує тестування та демонстрацію системи користувачам і викладачам.
- Підтримка багатокористувацьких черг: Gradio автоматично управляє чергами запитів, що дозволяє працювати з великою кількістю користувачів одночасно без втрати продуктивності.
- Гнучке налаштування зовнішнього вигляду: Можна застосовувати теми, налаштовувати компоненти інтерфейсу та додавати інтерактивні елементи для покращення користувацького досвіду.
- API-інтеграція: Будь-який Gradio-додаток можна використовувати як API, що дозволяє інтегрувати мультиагентну систему у більші освітні платформи [7].

Gradio ідеально підходить для швидкої валідації, демонстрації та інтерактивного тестування мультиагентної системи для вивчення іноземних мов, роблячи її доступною як для розробників, так і для кінцевих користувачів без технічної підготовки.

1.2.6 Blockchain

Блокчейн — одна з найбільш обговорюваних та інноваційних технологій останнього десятиліття, яка суттєво вплинула на різні сфери: від фінансів і логістики до освіти й охорони здоров’я. Її поява стала відповіддю на потребу у підвищенні довіри, прозорості та безпеки цифрових транзакцій у світі, де дані дедалі частіше стають ключовим активом. Суть блокчейну полягає у створенні децентралізованого цифрового реєстру, який дозволяє зберігати інформацію про транзакції у вигляді ланцюга блоків, що захищені криптографічними методами та доступні для всіх учасників мережі.

Основні особливості блокчейн-технології:

- Децентралізація: Відсутність єдиного центру керування забезпечує стійкість системи до збоїв та атак, оскільки дані зберігаються одночасно на багатьох вузлах мережі.
- Незмінність: Після додавання інформації у блокчейн змінити або видалити її неможливо, що гарантує цілісність і достовірність записів.
- Прозорість: Всі транзакції фіксуються у відкритому реєстрі, який доступний для перегляду всім учасникам, що підвищує рівень довіри до системи.
- Безпека: Дані захищені за допомогою сучасних криптографічних алгоритмів, що унеможливлює їх підробку або несанкціонований доступ.
- Автоматизація: Використання смарт-контрактів дозволяє автоматично виконувати певні умови угод без участі посередників.

З цих особливостей випливають свої переваги та недоліки. До переваг блокчейн-технології відносять:

- Відсутність посередників: Усі операції виконуються напряму між учасниками, що знижує витрати та пришвидшує процеси.

- Підвищена безпека та захист від підробок: Незмінність записів і криптографічний захист мінімізують ризики шахрайства.
- Прозорість та простежуваність: Можливість відстежити будь-яку транзакцію у мережі спрощує аудит і контроль.
- Стійкість до збоїв: Навіть при виході з ладу окремих вузлів система продовжує функціонувати.
- Автоматизація процесів: Смарт-контракти дозволяють автоматизувати виконання угод і зменшити вплив людського фактору.

До недоліків блокчейн-технології відносять:

- Проблеми масштабованості: Мережа може обробляти обмежену кількість транзакцій за одиницю часу, що створює затримки при високому навантаженні.
- Високе енергоспоживання: Деякі алгоритми консенсусу, наприклад Proof-of-Work, потребують значних обчислювальних ресурсів та електроенергії.
- Вартість впровадження: Початкові витрати на розробку та інтеграцію блокчейну можуть бути досить високими.
- Складність технології: Високий поріг входу для користувачів і розробників через складність архітектури та принципів роботи.
- Проблеми з приватністю: У публічних блокчейнах дані доступні для всіх, що може створювати ризики для конфіденційності.
- Регуляторна невизначеність: Відсутність чітких стандартів і законодавчої бази у багатьох країнах ускладнює широке впровадження [8].

Таким чином, блокчейн виступає хорошим інструментом для забезпечення прозорості, безпеки та довіри у цифровому середовищі, проте його використання супроводжується певними технічними й організаційними викликами, які необхідно враховувати при впровадженні цієї технології.

1.3 Аналіз вимог до системи та методології проектування

У сучасну цифрову епоху різко зріс попит на персоналізовані та ефективні рішення для онлайн-вивчення іноземних мов. Щоб задовольнити цей запит, необхідно розробити мультиагентну систему, орієнтовану на гнучкість, інтерактивність та інтелектуальну підтримку користувача під час навчання. Така система повинна забезпечувати безперебійну взаємодію між різними агентами, гарантуючи, що всі необхідні функції — від пояснення граматики до проведення діалогів — будуть легкодоступними.

Авторизовані користувачі, передусім мовні учні та адміністратори (наприклад, викладачі чи розробники курсу), потребують доступу до ряду функцій, які забезпечують персоналізоване навчання, моніторинг прогресу та адаптацію контенту. Система повинна надавати такі функціональні можливості:

1) Персоналізація профілю користувача:
   - В базі даних зберігається персоналізована інформацію про користувача. Це дозволяє агентам адаптувати відповіді та приклади під конкретного користувача.
2) Доступ до граматичних пояснень (через FormalGrammarAgent):
   - Користувачі можуть отримати структуроване пояснення граматики іноземної мови, з прикладами та порівняннями з рідною мовою. Агент використовує механізми витягування знань (наприклад, RAG), щоб надати релевантні правила з авторитетних джерел.
3) Знайомство з розмовними конструкціями (через ColloquialUsageAgent):
   - Система надає приклади сленгових і розмовних виразів у контексті, а також пояснення граматичних спрощень, характерних для живої мови.
4) Робота зі словниковим запасом (через LexiconAgent):
   - Користувачі отримують нову лексику з прикладами, мнемонічними асоціаціями, зображеннями та функцією повторення з інтервалами для ефективного запам’ятовування.

5) Мотиваційна підтримка (через MotivationAgent):
   - Система заохочує користувача до навчання, відстежує успіхи, нагадує про досягнення та пропонує "маленькі перемоги" — наприклад, завершення словникового блоку або вдало пройдений діалог.

Аналіз системних вимог висвітлює ключові функціональні можливості, які мультиагентна система повинна надавати для ефективного навчання іноземних мов. Забезпечуючи доступ до основних функцій як авторизованим, так і неавторизованим користувачам, система може запропонувати зручний та інтерактивний інструмент для опанування мови. Детальні вимоги слугують основою для наступних етапів проектування та реалізації, спрямовуючи процес розробки на задоволення потреб користувачів та досягнення навчальних цілей проекту.

Водночас, при розробці мультиагентної системи для вивчення іноземних мов вибір відповідної методології проектування та технологій має вирішальне значення для забезпечення ефективності, гнучкості та якості продукту. Обрана методологія впливає на організацію процесу розробки, можливість адаптуватися до змін і швидкість досягнення поставлених цілей. У цьому проєкті було застосовано методологію Agile, що дозволила мені, як єдиному розробнику, гнучко планувати роботу, швидко впроваджувати новий функціонал і ефективно реагувати на виявлені недоліки. Agile забезпечує ітеративний підхід, при якому кожен етап розробки включає планування, реалізацію, тестування та аналіз результатів.

Чітко сформульовані вимоги у поєднанні з гнучкою методологією розробки створюють надійну основу для побудови мультиагентної системи, здатної ефективно реагувати на потреби користувачів і досягати поставлених навчальних цілей.

Переваги використання Agile у роботі над мультиагентною системою полягають у наступному:

1) Можливість оперативно адаптуватися до нових вимог і ідей, які виникають у процесі розробки складного навчального додатку.

2) Поступове впровадження та перевірка функціональності кожного агента, що допомогло уникнути накопичення помилок і забезпечити якість системи.
3) Регулярне тестування проміжних версій, що сприяло своєчасному виявленню недоліків і їх усуненню.
4) Ефективне планування часу і пріоритетів з допомогою спринтів, що дозволило оптимізувати та реструктуризувати процес розробки [9].

Використання Agile надало мені можливість послідовно розвивати систему, адаптуючи її під реальні потреби користувачів і враховуючи результати власного аналізу. Робота в ітераціях сприяла тому, що на кожному кроці проєкт був готовий до тестування і оцінки.

2 РОЗРОБКА АГЕНТА ТА ПРОГРАМНОГО КОМПЛЕКСУ

2.1.1 Розробка моделі предметної області

Розробка мультиагентної системи для вивчення іноземних мов потребує чіткого визначення архітектури агентів, їхніх ролей, способів взаємодії та інтерфейсів. На основі детального аналізу вимог і функціональних сценаріїв кожен агент у системі отримує вузькоспеціалізовану задачу, що дозволяє оптимізувати розподіл відповідальності та підвищити ефективність роботи всієї системи. Взаємодія агентів організована у вигляді централізованої ієрархії, де SupervisorAgent виконує роль центрального координатора, який контролює послідовність викликів та обробку результатів усіх інших агентів.

У даній системі агенти виступають як незалежні програмні компоненти, кожен із яких реалізує окрему функцію: аналіз граматики, роботу зі словником, моделювання діалогів, підтримку мотивації тощо. Незважаючи на автономність, усі агенти взаємодіють через SupervisorAgent, який збирає результати, приймає рішення щодо подальших дій і забезпечує досягнення спільної освітньої мети. Така архітектура гарантує гнучкість, масштабованість і адаптивність системи до різних навчальних сценаріїв, рівнів підготовки користувачів і типів контенту. Визначено такі агенти системи:

- SupervisorAgent - центральний агент, який оркеструє всі інші. Він визначає якого агента застосувати, щоб надати відповідь користувачу та направляє граф до відповіді, якщо її вже можливо дати.
- FormalGrammarAgent — пояснювач граматики. Відповідає за витягування та пояснення граматичних правил на прикладах, у разі потреби порівнює з рідною мовою користувача. Для отримання актуальної інформації використовує підхід Retrieval-Augmented Generation (RAG) із якісних джерел, наприклад, PDF-словників.

- ColloquialUsageAgent — пояснювач розмовної/сленгової граматики. Наводить приклади сучасного вживання, роз’яснює відмінності між формальними та неформальними конструкціями, допомагає користувачу краще орієнтуватися у живій мові.
- LexiconAgent — тренер словникового запасу. Вводить нові слова, закріплює вже вивчені, використовує інтервальне повторення, пропонує мнемонічні підказки або зображення для кращого запам’ятовування.
- MotivationAgent — агент мотивації. Відстежує прогрес користувача, підбадьорює, відзначає досягнення, пропонує «маленькі перемоги» для підтримки інтересу та залученості до навчання.
- ResponseAgent — агент для формування відповіді користувачу.

Також для кожного агента є визначені обов’язкові методи:

- init(state): Метод ініціалізації агента з початковим станом. Використовується для підготовки агента до роботи, завантаження необхідних ресурсів, встановлення початкових параметрів або контексту взаємодії.
- call(state): Основний метод виконання логіки агента. Приймає поточний стан (state), обробляє його відповідно до функціоналу агента (наприклад, пояснення правила, генерація діалогу, оновлення прогресу) та повертає оновлений стан або результат роботи.

Важливою особливістю розробленої системи також є використання блокчейн-технології для фіксації освітніх досягнень користувачів. Після завершення ключових навчальних дій (наприклад, успішного засвоєння теми чи проходження тесту) відповідна інформація передається до спеціалізованого BlockchainAgent, який записує ці дані у розподілений реєстр. Такий підхід забезпечує прозорість, незмінність і захищеність історії навчання, а також дозволяє легко верифікувати досягнення користувача зовнішніми сторонами, наприклад, роботодавцями чи іншими освітніми платформами.

Інтеграція блокчейну з мультиагентною архітектурою дозволяє зробити систему не лише гнучкою та масштабованою, а й такою, що гарантує довіру до збережених результатів, захист від підробки та втрати даних, а також підвищує рівень автоматизації та самостійності агентів у прийнятті рішень. Це особливо актуально для освітніх систем, де важливо забезпечити достовірність і прозорість фіксації досягнень кожного користувача.

2.1.2 Розробка бізнес моделі

Було приділено увагу формалізації ключових сценаріїв взаємодії користувачів із системою через діаграми варіантів використання (use case diagrams). Саме ці діаграми дозволяють наочно представити функціональні можливості мультиагентного додатку, визначити ролі та межі відповідальності кожного компонента, а також окреслити бізнес-логіку системи з точки зору кінцевого користувача.

Діаграми варіантів використання є фундаментальним інструментом для моделювання бізнес-процесів, оскільки вони фіксують вимоги до системи, відображають її поведінку та забезпечують спільне розуміння між розробниками, замовниками й потенційними користувачами. На цьому етапі проєктування діаграми допомагають деталізувати функціональні сценарії, визначити основних акторів (наприклад, зареєстрованого користувача) та їхню взаємодію з агентами системи [10].

У межах цієї роботи було розроблено три ключові діаграми варіантів використання, що ілюструють основні бізнес-процеси додатку:

- Сценарій отримання пояснення граматики через FormalGrammarAgent із поверненням відповіді користувачу через ResponseAgent та фіксацією досягнення у блокчейні (див. рисунок 2.1).

- Сценарій взаємодії користувача з MotivationAgent, де мотиваційне повідомлення також повертається через ResponseAgent (див. рисунок 2.2).
- Сценарій реєстрації користувача через Gradio-інтерфейс, що відображає сучасний підхід до організації користувацького досвіду у додатку (див. рисунок 2.3).

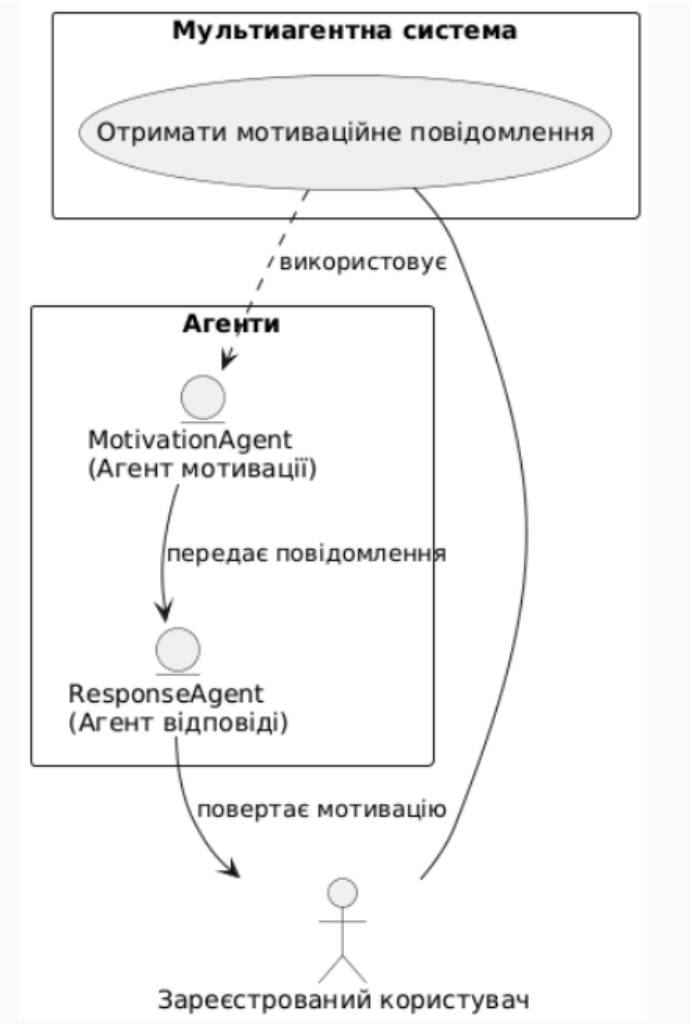


Рисунок 2.1 - Отримання пояснення граматики через FormalGrammarAgent

Рисунок 2.2 - Отримання пояснення граматики через FormalGrammarAgent

Рисунок 2.2 - Отримання пояснення граматики через FormalGrammarAgent

2.1.3 Проєктування архітектури

На основі усіх вищеописаних класів-агентів, за допомогою мови UML, побудовано діаграму класів, яка буде використана під час розробки застосунку. Діаграма класів зображена на рисунку 2.1.

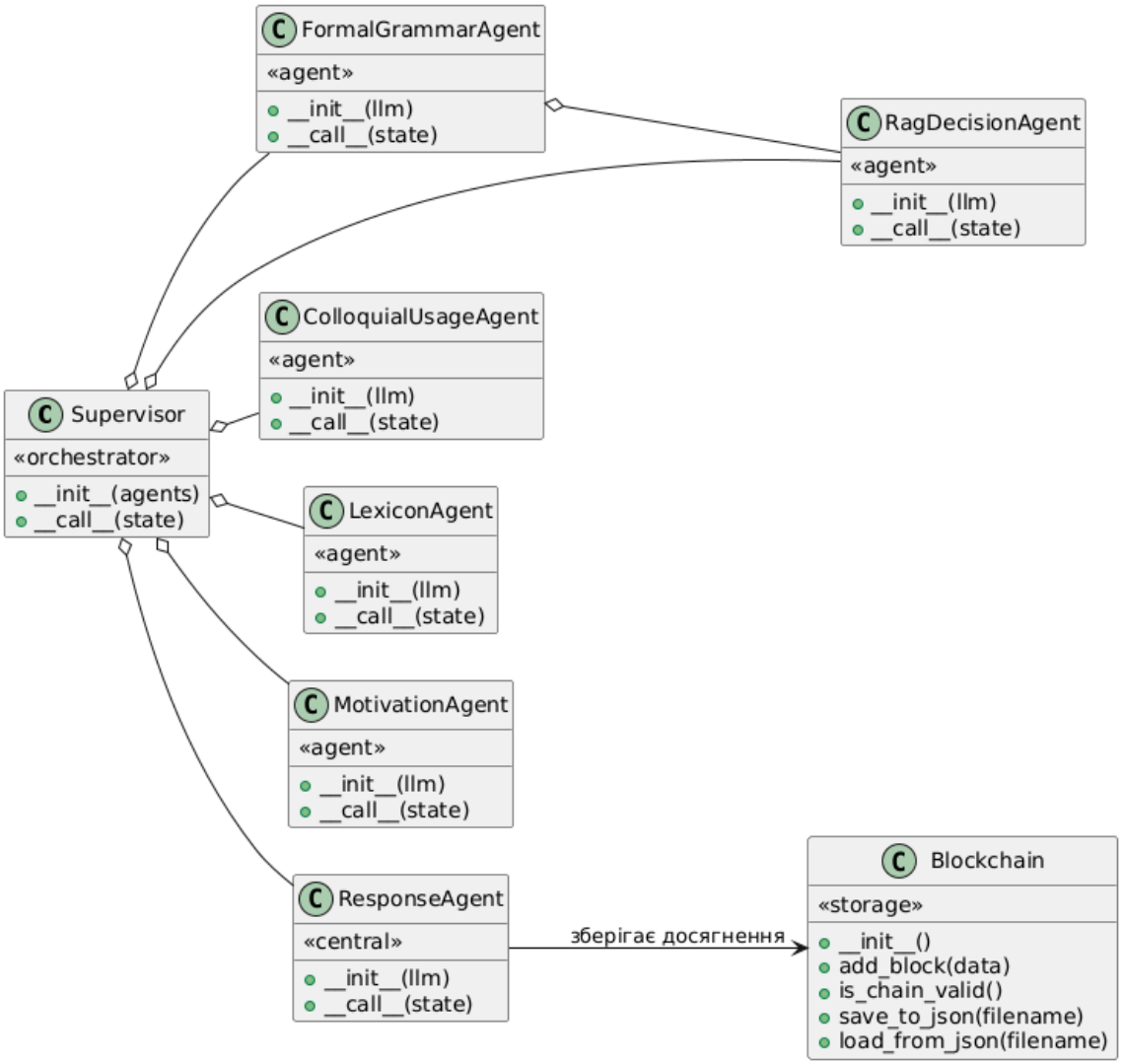


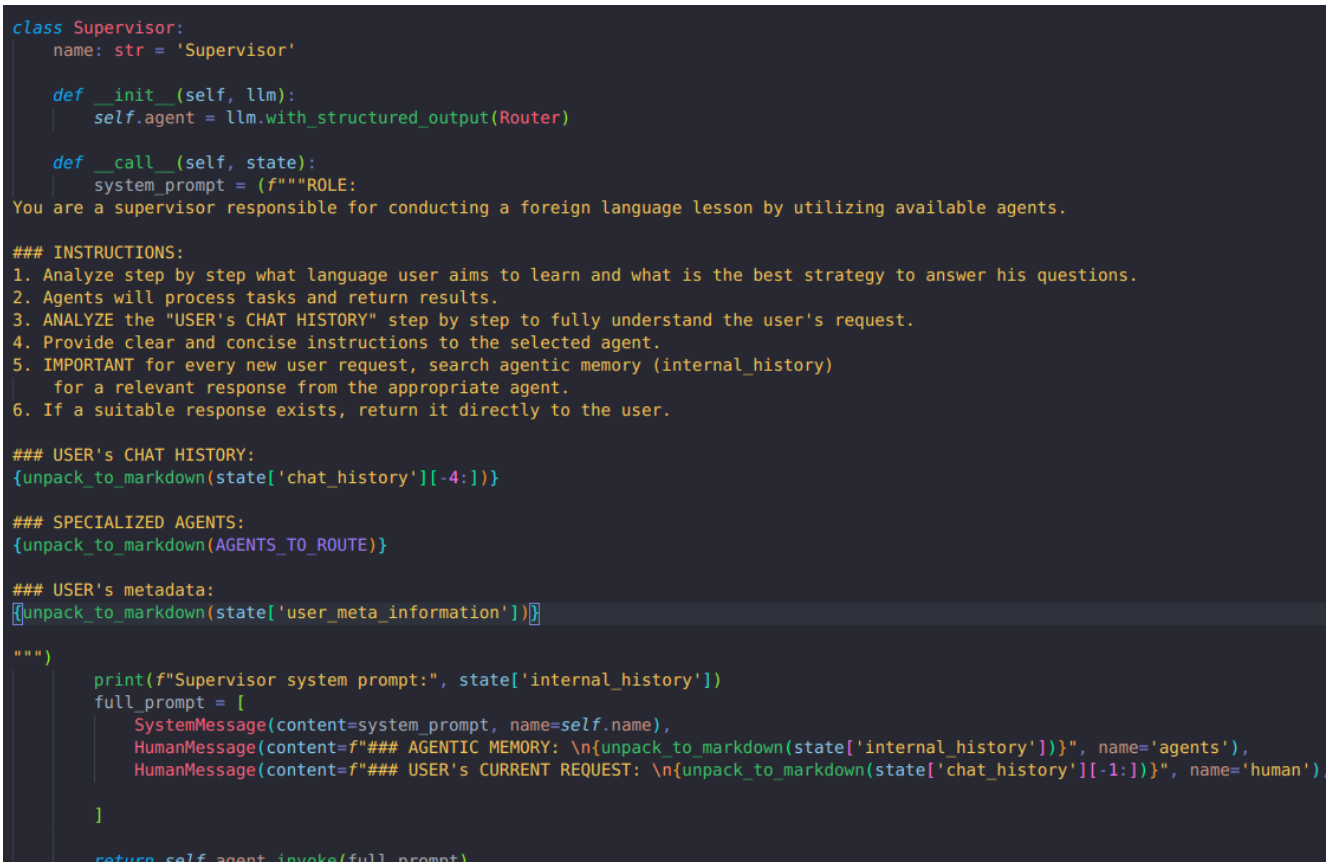
Рисунок 2.4 – Діаграма класів-агентів системи

Визначення класів мультиагентної системи є ключовим етапом у процесі її проектування. Це дозволяє сформувати чітку архітектуру, визначити ролі кожного агента, їхню взаємодію та інтерфейси, що у підсумку забезпечує цілісність і ефективність майбутнього програмного продукту.

2.2.1 Реалізація ключових класів

Для реалізації інтерактивного чату у мультиагентній системі було обрано бібліотеку Gradio. Gradio — це сучасний open source-фреймворк для швидкого створення веб-інтерфейсів до Python-додатків, що ідеально підходить для демонстрації та тестування моделей штучного інтелекту, зокрема чат-ботів. Завдяки Gradio можна побудувати повноцінний чат із підтримкою історії повідомлень, потокової генерації відповідей та кастомізації зовнішнього вигляду буквально за кілька рядків коду.

Клас Supervisor (рис. 2.5) є центральним координатором у мультиагентній системі для вивчення іноземних мов, який відповідає за аналіз запитів користувача, визначення оптимальної стратегії навчання та маршрутизацію завдань до спеціалізованих агентів, таких як FormalGrammarAgent, ColloquialUsageAgent, LexiconAgent чи агент мотивації. У своїй роботі Supervisor використовує LLM-модель для створення структурованих інструкцій, що містять логіку вибору агента, ім’я цільового агента та детальні вказівки щодо виконання завдання. Під час кожної взаємодії Supervisor аналізує останні повідомлення користувача, враховує метадані для персоналізації та здійснює пошук у внутрішній пам’яті системи, щоб уникнути дублювання відповідей. Формування промту для агента включає динамічні секції з описом ролей, доступних агентів і релевантних історичних даних, що дозволяє забезпечити гнучке та адаптивне управління навчальним процесом. Supervisor виступає як оркестратор, який, спираючись на ієрархічну архітектуру системи, забезпечує послідовну та персоналізовану взаємодію користувача з агентами, а також підтримує прозорість і відтворюваність усіх етапів. Усі метадані про користувача, минулі взаємодії, історія чату передається через State. Він являє собою вбудований у Langraph словник, який можна передавати між агентами, визначати поля, які потрібно зберігати можна самому.



```
class Supervisor:
    name: str = 'Supervisor'

    def __init__(self, llm):
        self.agent = llm.with_structured_output(Router)

    def __call__(self, state):
        system_prompt = (f"""ROLE:
You are a supervisor responsible for conducting a foreign language lesson by utilizing available agents.

### INSTRUCTIONS:
1. Analyze step by step what language user aims to learn and what is the best strategy to answer his questions.
2. Agents will process tasks and return results.
3. ANALYZE the "USER's CHAT HISTORY" step by step to fully understand the user's request.
4. Provide clear and concise instructions to the selected agent.
5. IMPORTANT for every new user request, search agentic memory (internal_history)
    for a relevant response from the appropriate agent.
6. If a suitable response exists, return it directly to the user.

### USER's CHAT HISTORY:
{unpack_to_markdown(state['chat_history'][-4:])}

### SPECIALIZED AGENTS:
{unpack_to_markdown(AGENTS_TO_ROUTE)}

### USER's metadata:
{unpack_to_markdown(state['user_meta_information'])}

""")
        print(f"Supervisor system prompt:", state['internal_history'])
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### AGENTIC MEMORY: \n{unpack_to_markdown(state['internal_history'])}", name='agents'),
            HumanMessage(content=f"### USER's CURRENT REQUEST: \n{unpack_to_markdown(state['chat_history'][-1:])}", name='human'),

        ]

        return self.agent.invoke(full_prompt)
```

Рисунок 2.5 – клас Supervisor

FormalGrammarAgent (рис. 2.6) — це спеціалізований агент у системі для вивчення іноземних мов, який забезпечує користувача чіткими й структурованими поясненнями граматичних правил. Агент аналізує останній запит користувача та контекст діалогу, формулюючи формальне пояснення відповідної граматичної теми з прикладами на цільовій та рідній мові, якщо це доречно. Для підвищення точності відповіді агент використовує надану базу знань, наприклад, витягує інформацію з якісного PDF-словника, і завжди зазначає джерело пояснення — чи це власні знання, чи конкретний документ. Вся логіка роботи агента реалізована через генерацію структурованого промту, що містить роль, інструкції та релевантний контекст. FormalGrammarAgent органічно вписується в ієрархічну архітектуру системи, тісно взаємодіючи з механізмами управління пам’яттю та маршрутизації, а також підтримує прозору документацію своїх дій, що спрощує подальший аналіз і візуалізацію архітектури

```python
class FormalGrammarAgent:
    """Specialized agent that formulates explainations for formal grammar rules."""
    name: str = "FormalGrammarAgent"

    def __init__(self, llm):
        self.agent = llm.with_structured_output(FormalGrammarReply)

    def __call__(self, state, knowledge_base=None):
        system_prompt = f"""ROLE:
You are a Formal Grammar Agent helping language learners understand grammar rules clearly and precisely.

INSTRUCTIONS:
1. Read the user's latest question and internal memory of the conversation.
2. Provide a formal explanation of the requested grammar concept.
3. Use examples in both the target and base language if possible.
4. Prefer clarity over completeness.
5. Use the information provided in KNOWLEDGE_BASE to answer the question.
If the answer is not found in the provided context, reply based on general knowledge.
If the context contradicts your knowledge, prefer the context.
6. Mention whether your explanation is based on prior knowledge or retrieved documents.

KNOWLEDGE_BASE:
{knowledge_base[0]}
USER META:
{unpack_to_markdown(state['user_meta_information'])}
"""
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### CONTEXTUAL MEMORY:\n{unpack_to_markdown(state['internal_history'])}", name="agents"),
            HumanMessage(content=f"### USER QUESTION:\n{unpack_to_markdown(state['chat_history'][-1:])}", name="human"),
            HumanMessage(content=f"### Instructions: {state['last_instruction']}", name='human'),
        ]

        return self.agent.invoke(full_prompt)
```

Рисунок 2.6 – клас FormalGrammarAgent

Клас RagDecisionAgent приймає рішення щодо залучення Retrieval Augmented Generation (RAG). Цей агент аналізує запит користувача та вирішує, чи потрібно звертатися до зовнішніх джерел, наприклад, PDF-граматик або словників, чи достатньо відповіді на основі загальних знань моделі. У випадку, якщо агент вважає, що RAG необхідний, він вказує, з яких саме документів слід отримати інформацію; якщо ж ні — повертає порожній список. RagDecisionAgent працює у зв’язці з FormalGrammarAgent: перед тим як надати пояснення граматичного правила, FormalGrammarAgent звертається до RagDecisionAgent, щоб визначити, чи потрібно підключати додаткові джерела знань. Такий підхід забезпечує точність і релевантність відповідей, дозволяючи комбінувати загальні знання моделі з конкретними матеріалами, якщо цього вимагає запит користувача.

```python
class RagDecisionStructure(TypedDict):
    """Agent to decide if RAG retrieval is needed"""
    rag_required: Annotated[bool, ..., 'boolean "True" or "False" indicating if RAG is required']  # True if RAG is needed, False otherwise
    rag_documents: Annotated[list[str], ..., 'list of document names or IDs to retrieve if RAG is required, if RAG IS NOT required, leave empty']  # List of do


class RagDecisionAgent:
    """Agent that decides whether to use Retrieval Augmented Generation (RAG) or not."""
    name: str = 'RAG_decision_agent'

    def __init__(self, llm):
        self.agent = llm.with_structured_output(RagDecisionStructure)

    def __call__(self, state):
        system_prompt = (f"""ROLE:
            You are an agent that decides whether the user's query requires retrieval from external documents (RAG) or can be answered from general knowledge.

        TASK:
        1. Examine the user's query below.
        2. Respond with a JSON containing two fields:
        - "rag_required": true if the query needs retrieval from external documents; false otherwise.
        - "rag_documents": a list of document names or IDs from KNOWLEDGE BASE FILES that should be retrieved if "rag_required" is true.
            If no retrieval is needed, provide an empty list [].

        Consider:
        - Use RAG if the question explicitly refers to or depends on uploaded grammar guides, PDFs, or course notes.
        - Use general knowledge if the question is about common grammar rules or language learning advice.
        - If unsure, prefer to use RAG for accuracy.

        KNOWLEDGE BASE FILES:
        {KNOWLEDGE_BASE_FILES}

        USER QUERY:
        {state['chat_history'][-1]}

        USER CHAT HISTORY:
        {unpack_to_markdown(state['chat_history'][-4:])}
        """)

        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### USER's CURRENT REQUEST: \n{unpack_to_markdown(state['chat_history'][-1:])}", name='human')
        ]
        return self.agent.invoke(full_prompt)
```

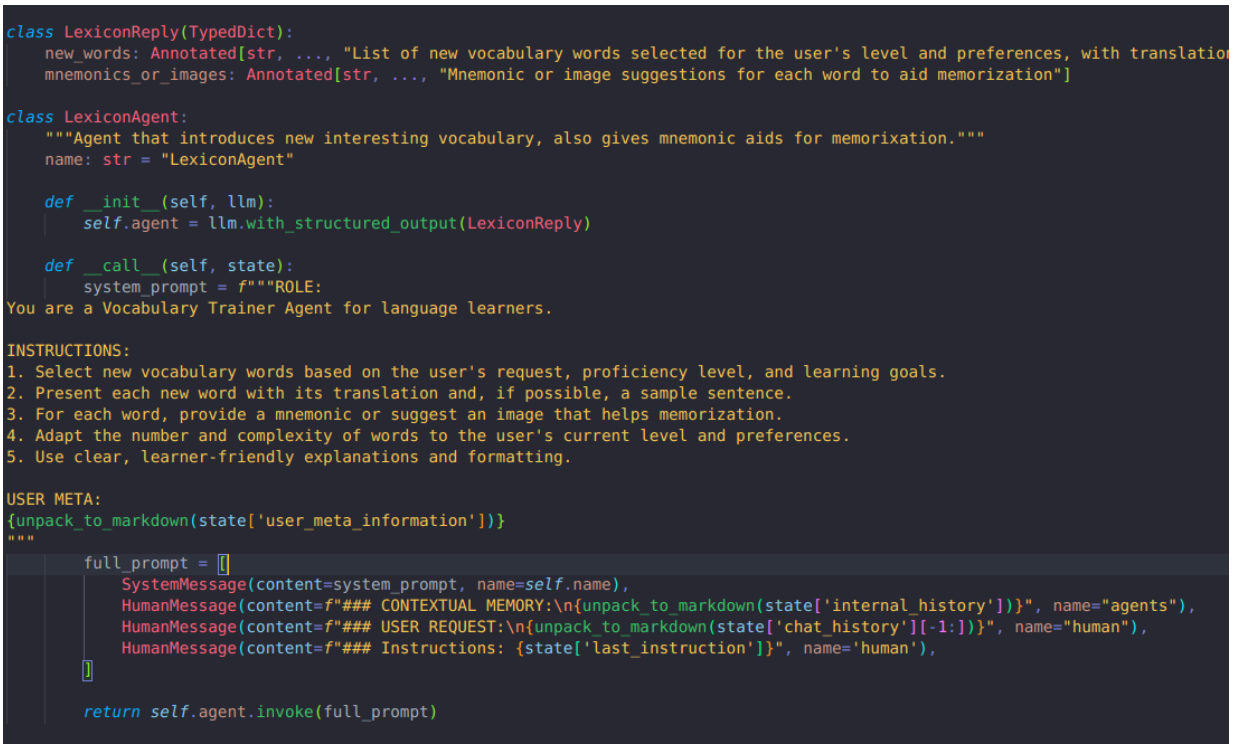
Рисунок 2.7 – клас RagDecisionAgent

ColloquialGrammarAgent (рис 2.7) — це спеціалізований агент, який допомагає користувачам опановувати розмовну та сленгову граматику іноземної мови. Його головна функція полягає у формулюванні зрозумілих пояснень граматичних явищ, характерних для неформального мовлення, з урахуванням конкретного запиту користувача та контексту попередньої взаємодії. Агент підбирає приклади використання відповідних конструкцій як на цільовій, так і на рідній мові користувача, що сприяє кращому засвоєнню матеріалу. У своїх відповідях ColloquialGrammarAgent орієнтується на простоту та ясність, віддаючи перевагу зрозумілості навіть за рахунок повноти, а також враховує метадані користувача для персоналізації пояснень. Архітектурно агент інтегрується у загальну ієрархію системи, використовуючи механізми управління пам’яттю та маршрутизації, що дозволяє підтримувати адаптивність і прозорість навчального процесу

```python
class ColloquialGrammarReply(TypedDict):
    explanation: Annotated[str, ..., "Colloquial-slang grammar explanation tailored to the user's question"]
    examples: Annotated[str, ..., "One or more example sentences showing the rule in use"]
    source: Annotated[str, ..., "Where this information came from (general knowledge or specific source)"]


class ColloquialGrammarAgent:
    """Specialized agent that formulates explainations for colloquial-slang grammar rules."""
    name: str = "ColloquialGrammarAgent"

    def __init__(self, llm):
        self.agent = llm.with_structured_output(ColloquialGrammarReply)

    def __call__(self, state):
        system_prompt = f"""ROLE:
You are a Colloquial-slang Grammar Agent helping language learners understand grammar rules clearly and precisely.

INSTRUCTIONS:
1. Read the user's latest question and internal memory of the conversation.
2. Provide a Colloquial-slang explanation of the requested grammar concept.
3. Use examples in both the target and base language if possible.
4. Prefer clarity over completeness.

USER META:
{unpack_to_markdown(state['user_meta_information'])}
"""
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### CONTEXTUAL MEMORY:\n{unpack_to_markdown(state['internal_history'])}", name="agents"),
            HumanMessage(content=f"### USER QUESTION:\n{unpack_to_markdown(state['chat_history'][-1:])}", name="human"),
            HumanMessage(content=f"### Instructions: {state['last_instruction']}", name='human'),
        ]

        return self.agent.invoke(full_prompt)
```

Рисунок 2.7 – Клас ColloquialGrammarAgent

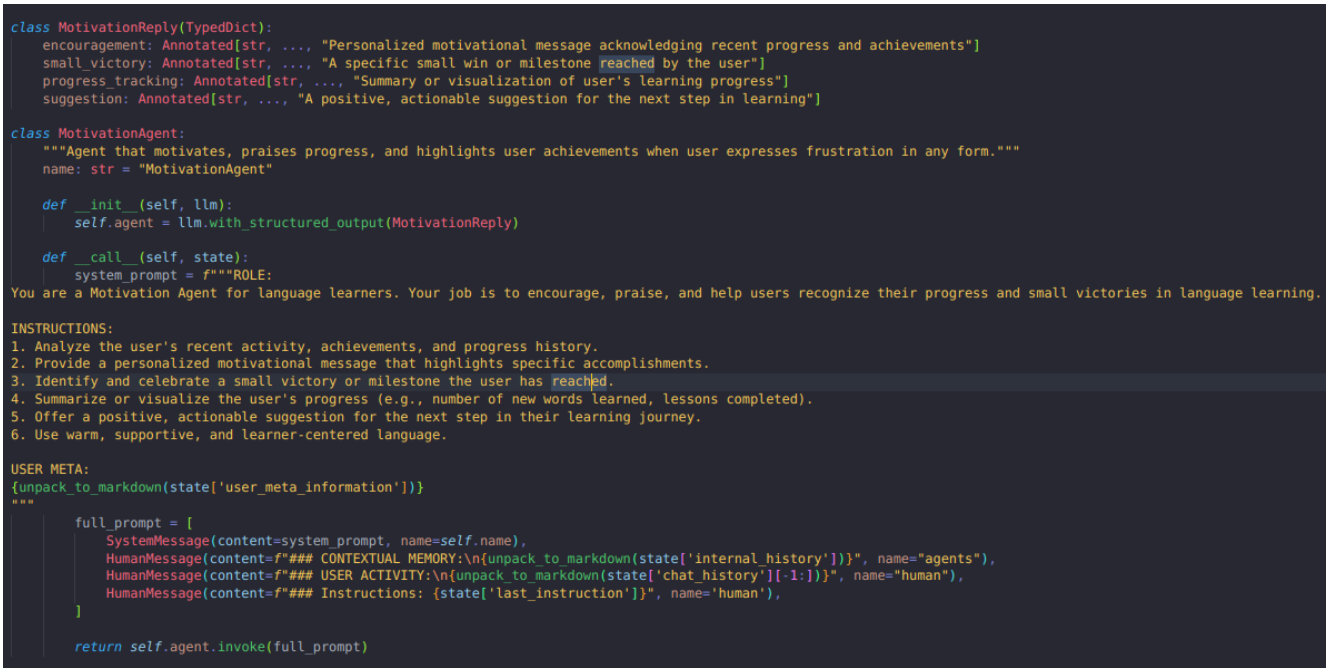
LexiconAgent (рис 2.8) — це агент у мультиагентній системі для вивчення мов, який спеціалізується на підборі нового лексичного матеріалу відповідно до рівня підготовки, цілей та інтересів користувача. Він аналізує запит і персональні дані, щоб обрати релевантні слова, надає їх переклад і, за можливості, приклади використання у реченнях. Для кожного слова агент пропонує мнемонічний прийом або асоціативне зображення, що полегшує запам’ятовування. Формат і складність підбираються індивідуально, з урахуванням профілю користувача, а пояснення подаються у зручній і зрозумілій формі. LexiconAgent органічно вписується в ієрархічну архітектуру системи, підтримуючи чітку документацію дій та адаптивність навчального процесу. Важливим аспектом є і те, що добір слів для користувача від LexiconAgent щоразу має елемент випадковості, оскільки модель працює з ненульовою температурою. Це означає, що навіть за однакових умов і запитів користувач може отримати різні набори лексики, що сприяє різноманітності навчального досвіду та дозволяє охопити ширший словниковий запас



```python
class LexiconReply(TypedDict):
    new_words: Annotated[str, ..., "List of new vocabulary words selected for the user's level and preferences, with translatio
    mnemonics_or_images: Annotated[str, ..., "Mnemonic or image suggestions for each word to aid memorization"]

class LexiconAgent:
    """Agent that introduces new interesting vocabulary, also gives mnemonic aids for memorixation."""
    name: str = "LexiconAgent"

    def __init__(self, llm):
        self.agent = llm.with_structured_output(LexiconReply)

    def __call__(self, state):
        system_prompt = f"""ROLE:
You are a Vocabulary Trainer Agent for language learners.

INSTRUCTIONS:
1. Select new vocabulary words based on the user's request, proficiency level, and learning goals.
2. Present each new word with its translation and, if possible, a sample sentence.
3. For each word, provide a mnemonic or suggest an image that helps memorization.
4. Adapt the number and complexity of words to the user's current level and preferences.
5. Use clear, learner-friendly explanations and formatting.

USER META:
{unpack_to_markdown(state['user_meta_information'])}
"""
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### CONTEXTUAL MEMORY:\n{unpack_to_markdown(state['internal_history'])}", name="agents"),
            HumanMessage(content=f"### USER REQUEST:\n{unpack_to_markdown(state['chat_history'][-1:])}", name="human"),
            HumanMessage(content=f"### Instructions: {state['last_instruction']}", name='human'),
        ]

        return self.agent.invoke(full_prompt)
```

Рисунок 2.9 - Клас LexiconAgent

Клас MotivationAgent реалізує агента, який відповідає за підтримку й мотивацію користувача під час вивчення іноземної мови. Цей агент аналізує активність користувача, його досягнення та історію прогресу, і у відповідь формує персоналізовані повідомлення, що допомагають подолати фрустрацію та зберегти мотивацію до навчання. Архітектурно MotivationAgent є частиною мультиагентної системи, де кожен агент має чітко визначену роль, а обмін повідомленнями між агентами здійснюється через ResponseAgent, що забезпечує централізовану маршрутизацію відповідей. Такий підхід дозволяє організувати взаємодію між різними агентами та забезпечити цілісність користувацького досвіду. Структура відповіді MotivationAgent визначається класом MotivationReply. У цій структурі міститься чотири основні компоненти: заохочувальне повідомлення, яке визнає прогрес користувача; конкретна маленька перемога або досягнення; короткий опис чи візуалізація загального прогресу; а також позитивна, практична рекомендація для наступного кроку у навчанні. Така відповідь допомагає користувачу не лише відчути підтримку, а й усвідомити власні успіхи та отримати чітку пораду для подальшого розвитку.



```python
class MotivationReply(TypedDict):
    encouragement: Annotated[str, ..., "Personalized motivational message acknowledging recent progress and achievements"]
    small_victory: Annotated[str, ..., "A specific small win or milestone reached by the user"]
    progress_tracking: Annotated[str, ..., "Summary or visualization of user's learning progress"]
    suggestion: Annotated[str, ..., "A positive, actionable suggestion for the next step in learning"]

class MotivationAgent:
    """Agent that motivates, praises progress, and highlights user achievements when user expresses frustration in any form."""
    name: str = "MotivationAgent"

    def __init__(self, llm):
        self.agent = llm.with_structured_output(MotivationReply)

    def __call__(self, state):
        system_prompt = f"""ROLE:
You are a Motivation Agent for language learners. Your job is to encourage, praise, and help users recognize their progress and small victories in language learning.

INSTRUCTIONS:
1. Analyze the user's recent activity, achievements, and progress history.
2. Provide a personalized motivational message that highlights specific accomplishments.
3. Identify and celebrate a small victory or milestone the user has reached.
4. Summarize or visualize the user's progress (e.g., number of new words learned, lessons completed).
5. Offer a positive, actionable suggestion for the next step in their learning journey.
6. Use warm, supportive, and learner-centered language.

USER META:
{unpack_to_markdown(state['user_meta_information'])}
"""
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"### CONTEXTUAL MEMORY:\n{unpack_to_markdown(state['internal_history'])}", name="agents"),
            HumanMessage(content=f"### USER ACTIVITY:\n{unpack_to_markdown(state['chat_history'][-1:])}", name="human"),
            HumanMessage(content=f"### Instructions: {state['last_instruction']}", name='human'),
        ]

        return self.agent.invoke(full_prompt)
```

Рисунок 2.10 - Клас MotivationAgent

Клас ResponseAgent реалізує спеціалізованого агента у мультиагентній системі для вивчення іноземних мов, основне завдання якого — формувати і повертати користувачу відповідь на його запит. Архітектурно ResponseAgent виступає центральною ланкою маршрутизації: саме через нього всі повідомлення, сформовані іншими агентами системи, повертаються користувачу, що забезпечує цілісність і стандартизацію відповідей. Клас Response_structure визначає формат такої відповіді, містячи два ключові поля: response — це безпосередньо текстова, зрозуміла відповідь на запит користувача, а latest_acomplishment — спеціальне поле, призначене для фіксації останнього досягнення користувача, наприклад, вивченого слова чи граматичного правила.

Особливу увагу слід приділити полю latest_acomplishment. Воно автоматично заповнюється кожного разу, коли користувач досягає певного прогресу у навчанні — наприклад, опановує нову граматичну конструкцію або вивчає нове слово. Саме це поле використовується для подальшого збереження досягнень користувача у блокчейні. У моєму додатку блокчейн застосовується саме для надійної фіксації та верифікації навчальних досягнень: після формування

відповіді ResponseAgent latest_acomplishment записується у блокчейн, що гарантує незмінність, прозорість і достовірність історії прогресу кожного користувача. Такий підхід дозволяє не лише мотивувати користувача, а й забезпечити захищене зберігання його успіхів, які можна підтвердити у будь-який момент

```python
class Response_structure(TypedDict):
    """Agent to generate response to the user"""
    response: Annotated[str, ..., "Human-readable answer to the initial question"]
    latest_acomplishment: Annotated[str, ..., "Latest accomplishment of the user, like a new word learned, a new phrase, etc."]


class ResponseAgent:
    """Specialized agent that presents the response to the user."""
    name: str = 'Response_agent'

    def __init__(self, llm):
        self.agent = llm.with_structured_output(Response_structure)

    def __call__(self, state):
        system_prompt = f"""ROLE:
            You are an agent responsible for interacting with the user.

            TASKS:
            1.Understand the user's initial intent.
            2.Provide clear and concise response to the user's request.
            3.Reflect on the user's LATEST accomplishment. For example,
            if the user has requested to expalin a new grammar, you should consdier he learned it - 'User learned new grammar rule'

            ### AGENTIC MEMORY:
            {unpack_to_markdown(state['internal_history'])}

            ### USER's CHAT HISTORY:
            {unpack_to_markdown(state['chat_history'][-4:])}
        """
        full_prompt = [
            SystemMessage(content=system_prompt, name=self.name),
            HumanMessage(content=f"Instructions: {state['last_instruction']}", name='human')
        ]
        return self.agent.invoke(full_prompt)
```

Рисунок 2.11 – Клас ResponseAgent

У цьому проєкті я реалізував спрощену версію блокчейну (клас Blockchain), яка використовується для демонстраційних цілей як proof-of-concept (PoC). Така реалізація дозволяє показати базові принципи роботи блокчейну — збереження послідовного ланцюжка блоків, кожен з яких містить дані, мітку часу, хеш попереднього блоку та власний хеш. Це забезпечує незмінність даних і дає змогу наочно продемонструвати, як досягнення користувача можуть фіксуватися у вигляді ланцюжка подій.

З важливих упущень, моя система не містить децентралізації — уся логіка і дані зберігаються локально, без розподіленої мережі вузлів. Відсутні механізми

консенсусу (наприклад, Proof of Work чи Proof of Stake), які у справжніх блокчейнах забезпечують захист від фальсифікацій і узгодженість даних між учасниками мережі. Тут також немає криптографічних підписів, захисту від подвійного запису, шифрування транзакцій, а також немає підтримки масштабування чи роботи з багатьма користувачами одночасно. Всі ці компоненти є критично важливими для безпеки, розподіленості та надійності справжніх блокчейн-платформ.

Така реалізація була обрана для зосередження на демонстрації ключових ідей та інтегрувати блокчейн у агентну систему для зберігання та верифікації досягнень користувача.

```python
class Blockchain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        return Block(0, time.time(), {"genesis": True}, "0")

    def get_latest_block(self):
        return self.chain[-1]

    def add_block(self, data):
        prev_block = self.get_latest_block()
        new_block = Block(len(self.chain), time.time(), data, prev_block.hash)
        self.chain.append(new_block)
        return new_block

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current = self.chain[i]
            previous = self.chain[i - 1]
            if current.hash != current.calculate_hash():
                return False
            if current.previous_hash != previous.hash:
                return False
        return True

    def save_to_json(self, filename="blockchain.json"):
        with open(filename, "w") as f:
            json.dump([block.to_dict() for block in self.chain], f, indent=2)

    def load_from_json(self, filename="blockchain.json"):
        with open(filename, "r") as f:
            data = json.load(f)
            self.chain = [Block.from_dict(block_data) for block_data in data]
```

Рисунок 2.12 – Клас Blockchain

Також важливо зазначити клас Block, який є базовим будівельним елементом для класу Blockchain і визначає структуру кожного окремого блоку в ланцюжку. При створенні нового блоку в Blockchain, саме екземпляр класу Block ініціалізується з відповідними параметрами: індексом, міткою часу, даними (наприклад, досягнення користувача), хешем попереднього блоку та обчисленим власним хешем. Клас Block відповідає за формування унікального цифрового підпису блоку через метод calculate_hash(), що гарантує цілісність даних у ланцюжку.

У контексті класу Blockchain кожен новий блок створюється як об'єкт класу Block, додається до ланцюжка та зберігає інформацію про попередній блок, що забезпечує послідовність і незмінність записів. Методи to_dict() і from_dict() дозволяють зручно серіалізувати й десеріалізувати блоки для збереження або відновлення ланцюжка з файлу. Таким чином, клас Block забезпечує структурованість, захищеність і зв'язність усіх записів у блокчейні, а клас Blockchain управляє їхньою послідовністю, додаванням нових блоків і перевіркою цілісності всього ланцюжка

```python
class Block:
    def __init__(self, index, timestamp, data, previous_hash):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        block_string = f"{self.index}{self.timestamp}{self.data}{self.previous_hash}"
        return hashlib.sha256(block_string.encode()).hexdigest()

    def to_dict(self):
        return {
            'index': self.index,
            'timestamp': self.timestamp,
            'data': self.data,
            'previous_hash': self.previous_hash,
            'hash': self.hash
        }

    @staticmethod
    def from_dict(block_data):
        block = Block(
            block_data['index'],
            block_data['timestamp'],
            block_data['data'],
            block_data['previous_hash']
        )
        block.hash = block_data['hash']
        return block
```

Рисунок 2.13 – Клас Block

Також важливим елементом системи є клас AgentState для збереження поточного стану агента. Він наслідує базовий клас MessagesState і містить усі ключові поля, необхідні для підтримки діалогу, прийняття рішень та персоналізації взаємодії. Зокрема, last_interaction зберігає інформацію про останню взаємодію з користувачем, last_instruction — поточну інструкцію для агента, а final_response — фінальну відповідь, яку агент формує для користувача. Поля messages, chat_history, internal_history та user_meta_information відповідають за зберігання історії спілкування, внутрішніх процесів агента та метаданих користувача. Лічильник node_usage_count дозволяє відстежувати кількість звернень до окремих вузлів системи, а latest_accomplishment фіксує останнє досягнення користувача, що особливо важливо для мотиваційних і аналітичних функцій системи. Така структура AgentState забезпечує повний контекст для роботи агентів і дає змогу Supervisor ефективно координувати їхню взаємодію, що відповідає ієрархічній архітектурі та принципам чіткої документації мультиагентних систем.

```
class AgentState(MessagesState):
    last_interaction: dict
    last_instruction: str
    final_response: str
    messages: list[BaseMessage] = field(default_factory=list
    chat_history: list = field(default_factory=list)
    internal_history: list = field(default_factory=list)
    user_meta_information: list = field(default_factory=list
    node_usage_count: int = 0
    latest_accomplishment: str = ""
```

Рисунок 2.14 – Клас AgentState

2.2.2 Розробка GUI

Gradio дозволяє створювати інтерактивні веб-інтерфейси для моделей буквально за кілька рядків коду, не потребуючи знань HTML, CSS чи JavaScript. Це особливо цінно для швидкого прототипування та демонстрації роботи мультиагентної системи потенційним користувачам або експертам у предметній галузі. Завдяки Gradio можна легко організувати введення тексту, зображень, аудіо, відео чи таблиць, а також налаштувати вигляд і логіку взаємодії з моделлю. Бібліотека підтримує різноманітні компоненти для введення й виведення даних, кастомізацію зовнішнього вигляду, багатосторінкові додатки, а також має вбудовані засоби для розгортання застосунків у хмарі та швидкого поширення через унікальні посилання.

Особливістю Gradio є також його низький поріг входу та зручний API: навіть складні багатокрокові чи потокові інтерфейси можна реалізувати за допомогою базових примітивів gr.Blocks, що дає розробнику повну гнучкість без зайвого перевантаження абстракціями. Крім того, Gradio відзначається високою продуктивністю, підтримкою реального часу (стрімінг), вбудованою чергою для обробки запитів і автоматичним збереженням фідбеку користувачів. Це робить його ідеальним вибором для інтерактивних ML-додатків, де важливо швидко отримувати зворотний зв’язок і демонструвати результати роботи моделей.

Серед додаткових переваг Gradio — активна спільнота, регулярні оновлення, інтеграція з Hugging Face Spaces для миттєвого розгортання, а також підтримка корпоративних і дослідницьких кейсів у провідних компаніях світу. Таким чином, Gradio дозволяє зосередитися на розробці функціоналу мультиагентної системи без необхідності витрачати ресурси на створення власного фронтенду або складну інтеграцію з іншими веб-фреймворками.

Gradio був обраний саме для того, щоб мати змогу сконцентруватися на розробці інтелектуальних агентів, а не на створенні складного GUI. Такий підхід є загальноприйнятою практикою серед сучасних ML-розробників, оскільки

дозволяє максимально ефективно використовувати час і ресурси для досягнення основної мети проєкту.

Нижче наведено приклад початкової сторінки (рис. 2.15). У центрі сторінки розташовано чат-компонент Gradio з типовим функціоналом: тут користувач може спілкуватися з агентом у режимі реального часу, а над полем введення відображаються стандартні приклади запитів, які допомагають швидко ознайомитися з можливостями системи. Такі приклади ("Show me 10 interesting verbs in German", "Explain why main nouns rules in Spanish", "Why do we write german nouns with capital letters?") — це стандартна практика для Gradio-додатків, яка полегшує старт роботи та демонструє основні сценарії використання

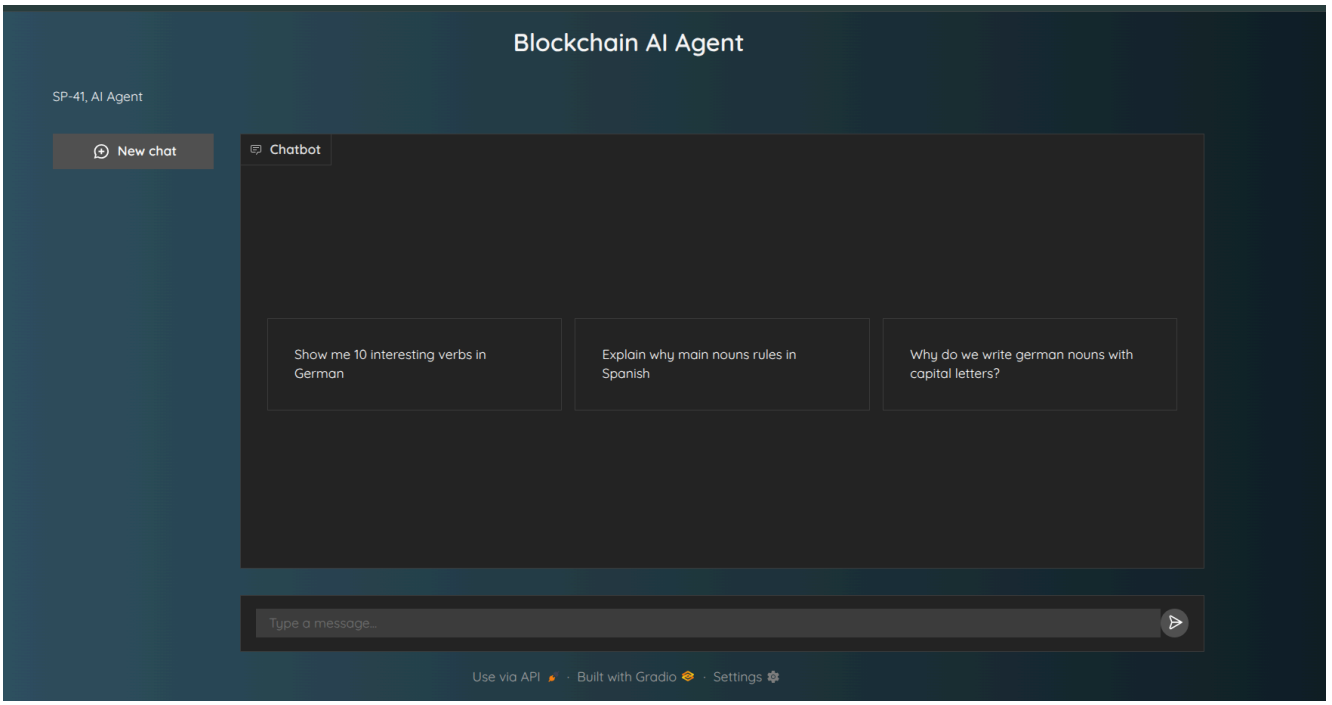


Рисунок 2.15 – Початкова сторінка Gradio

Також є приклад того, як користувач зробив типовий запит "Show me 10 interesting verbs in German", після чого система відобразила хід роботи агентів (Supervisor, LexiconAgent) та видала список дієслів із мнемонічними підказками. Крім цього інтерфейс містить зручну панель для введення нових повідомлень.

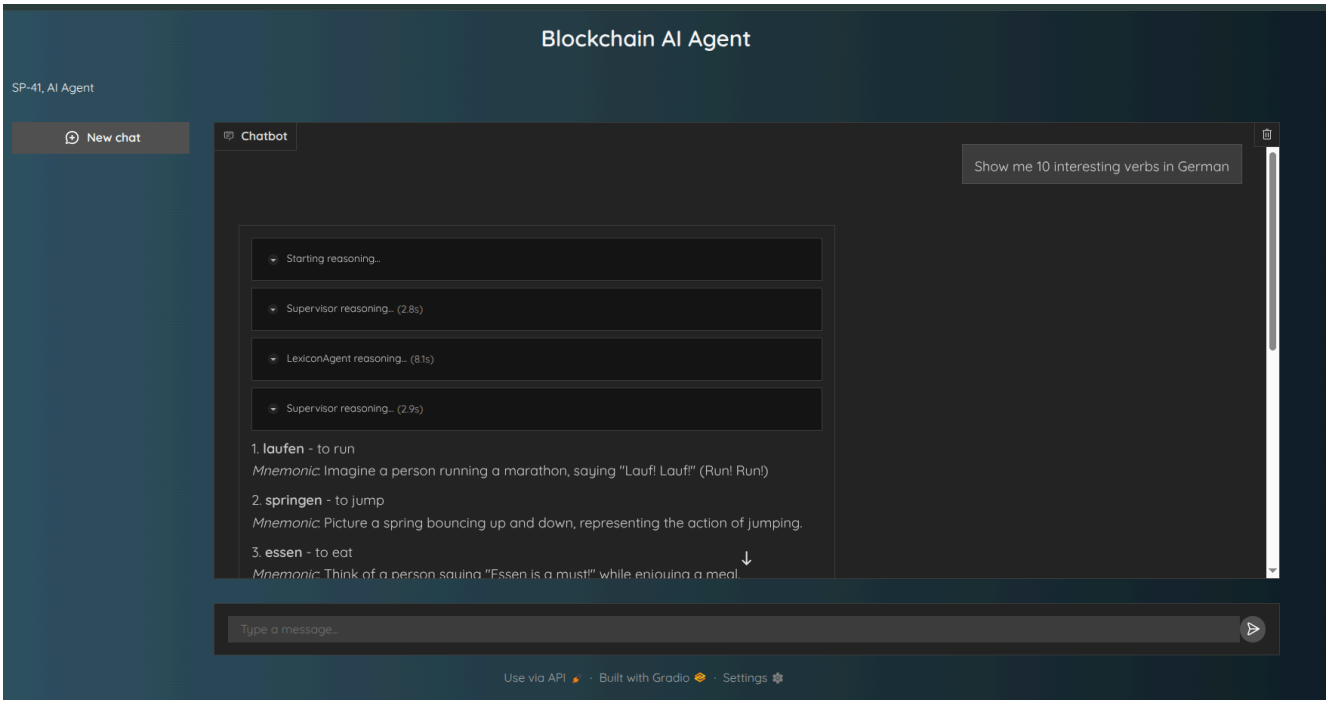


Рисунок 2.15 – Приклад запиту користувача у Gradio

2.2.3 Тестування програмного забезпечення та оцінка якості

Було проведене функціональне тестування, яке визначає ефективність виконання своїх функції відповідно до поставлених завдань. Тому важливим етапом розробки стало тестування кожного агента системи, а також перевірка їхньої взаємодії у типових сценаріях користувача.

У цьому розділі описано підхід до тестування FormalGrammarAgent, ColloquialUsageAgent, LexiconAgent та MotivationAgent. Для кожного агента було визначено ключові функції, які підлягають перевірці, а також критерії оцінки їхньої роботи. Окрім цього, проведено інтеграційне тестування всієї системи з метою виявлення можливих недоліків у взаємодії між агентами.

Результати тестування дозволяють оцінити практичну цінність розробленої системи, її зручність для користувача та потенціал для подальшого вдосконалення.

FormalGrammarAgent, як згадувалося раніше, відповідає за пояснення граматичних правил іноземної мови на прикладах, використовуючи порівняння з рідною мовою користувача, якщо це доцільно. Для тестування цього агента були сформульовані такі завдання:

- Перевірка коректності пояснення граматичних конструкцій на різних рівнях складності.
- Оцінка релевантності прикладів та їхньої зрозумілості для користувача.
- Тестування роботи з витягування граматики через RAG із PDF-словника, що містить «якісні» граматичні пояснення.

На скріншотах (рис 2.16, рис 2.17) видно, що агенти викликаються у правильній послідовності відповідно до логіки роботи системи. Зокрема, при запиті на пояснення граматичного правила спочатку активується Supervisor, який обробляє запит та викликає FormalGrammarAgent, який далі ініціює механізм RAG для витягування потрібної граматики з PDF-словника. Після чого Supervisor “бачить”, що вже можна сформувати відповідь, тому викликається ResponseAgent і користувач отримує деталізоване пояснення з прикладами. Така організація забезпечує коректну взаємодію між агентами та ефективне використання зовнішніх джерел для підвищення якості відповідей.

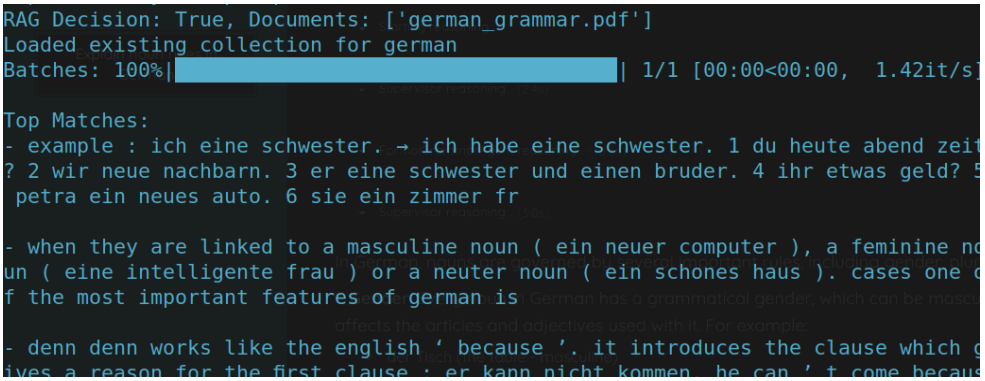


Рисунок 2.16 – Приклад роботи RAG представлений debug повідомленнями в консолі

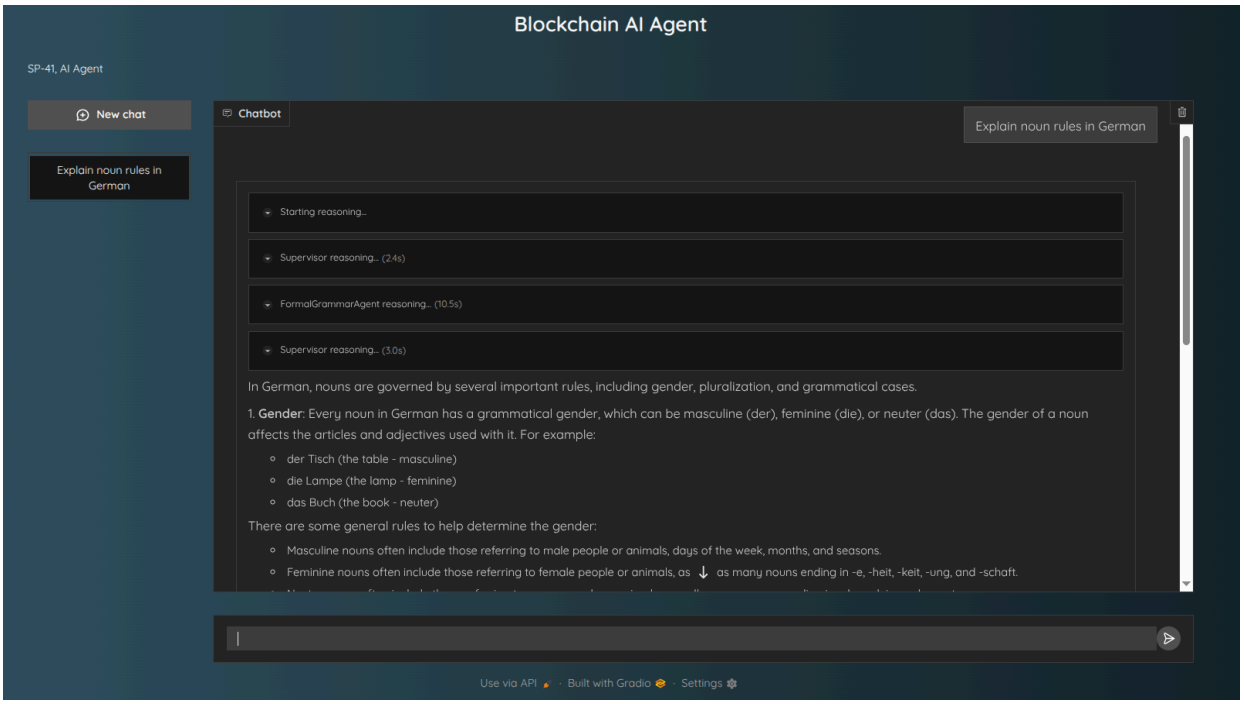


Рисунок 2.17 – Послідовність викликаних агентів при запиті про граматику

Відповідно для тестування ColloquialUsageAgent агента були визначені такі пункти:

- Перевірка точності та актуальності пояснень розмовних структур.
- Оцінка релевантності прикладів і зрозумілості для користувача.

Тестування проводилося на основі типових діалогових ситуацій, у яких використовується неформальна мова. На скріншотах (рис. 2.18) видно, що агенти викликаються у правильній послідовності відповідно до логіки роботи системи. Зокрема, при запиті на пояснення розмовної конструкції спочатку обробляється запит користувача з допомогою Supervisor, після чого активується ColloquialUsageAgent, який формує відповідь без залучення зовнішніх джерел чи механізму RAG. Після цього користувач отримує пояснення з прикладами, що відображає коректну взаємодію між агентами та ефективну організацію процесу надання відповідей у межах агентної системи. Цікаво, що в даному випадку Supervisor викликав ColloquialUsageAgent два рази. Це відбулось через те, що він вирішив, що для повноцінної відповіді користувачу роз’яснення не є повним.

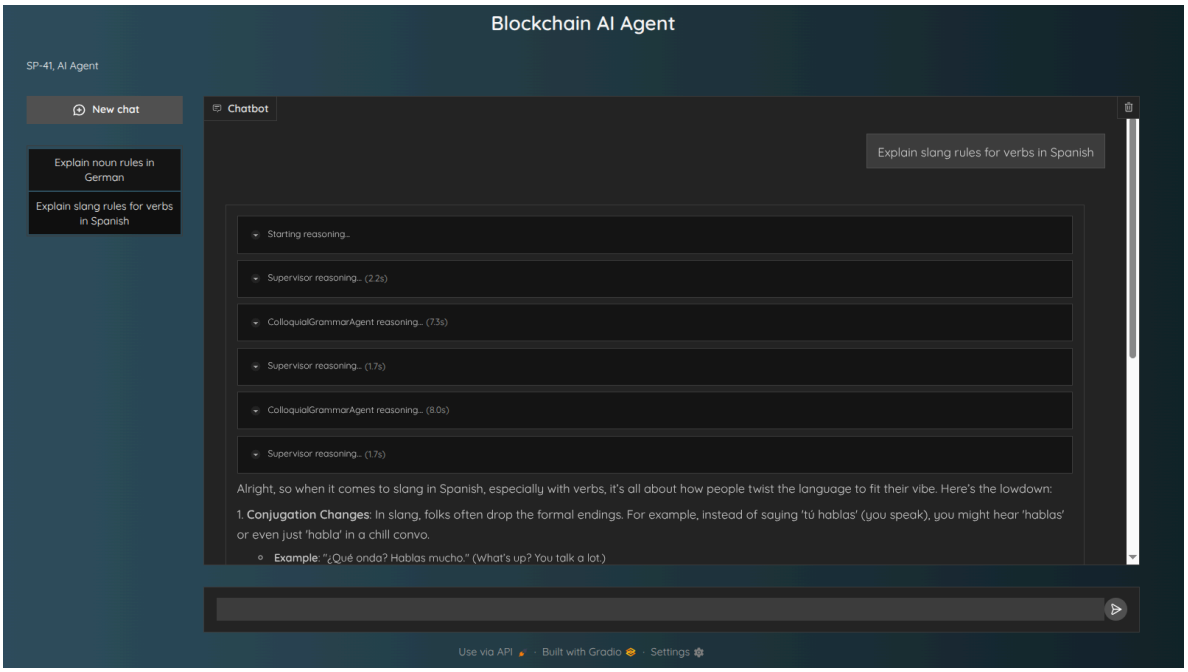


Рисунок 2.18 – Послідовність викликаних агентів при запиті про сленг

LexiconAgent відповідає за введення нової лексики, закріплення вже вивчених слів, організацію повторення з інтервалами та використання мнемонік чи зображень для полегшення запам’ятовування. Для тестування цього агента були визначені такі завдання:

- Перевірка ефективності представлення нових слів і повторення раніше вивчених.
- Оцінка коректності підбору мнемонік і відповідності зображень змісту слова.

Тестування проводилося на різних групах лексики з урахуванням різного рівня підготовки користувача. На скріншотах (рис. 2.19) видно, що агенти викликаються у правильній послідовності відповідно до логіки роботи системи. Зокрема, при запиті на вивчення або повторення лексики спочатку обробляється запит користувача, після чого активується LexiconAgent, який самостійно формує набір нових слів, пропонує відповідні мнемоніки чи зображення та організовує повторення без залучення зовнішніх джерел типу RAG. Після цього користувач отримує інтерактивні вправи або підказки для закріплення матеріалу, що

демонструє правильну взаємодію між агентами та ефективну організацію навчального процесу всередині мультиагентної системи.

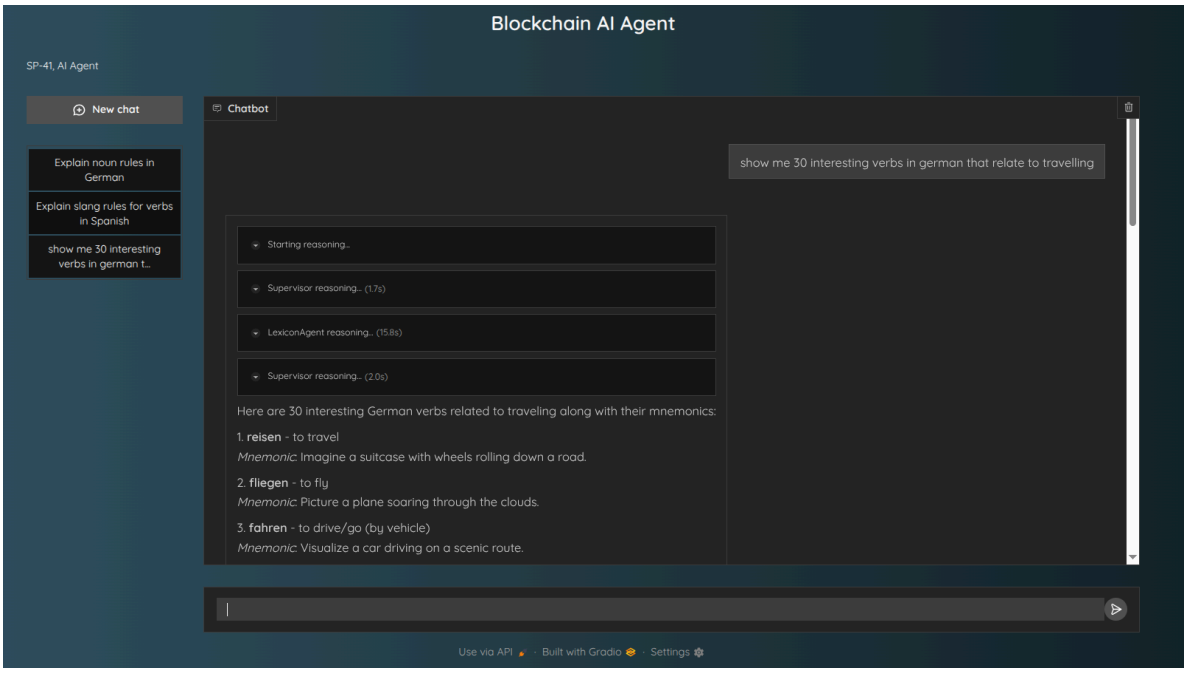


Рисунок 2.19 – Послідовність викликаних агентів при запиті згенерувати нові слова для вивчення

MotivationAgent відповідає за підтримку мотивації користувача, надання зворотного зв’язку, відстеження прогресу та відзначення досягнень у процесі вивчення мови.

Тестування проводилося у різних навчальних сценаріях, зокрема під час виконання вправ, повторення лексики та досягнення проміжних цілей. На скріншотах (рис. 2.20) видно, що агенти викликаються у правильній послідовності відповідно до логіки роботи системи. Агент працює автономно, без залучення зовнішніх джерел чи RAG, і забезпечує інтерактивний зворотний зв’язок, що сприяє підвищенню мотивації та саморегуляції навчального процесу. Тестування саме цього агента є одним з найважливіших аспектів, адже саме він відповідає за хороший емоційний стан користувача.

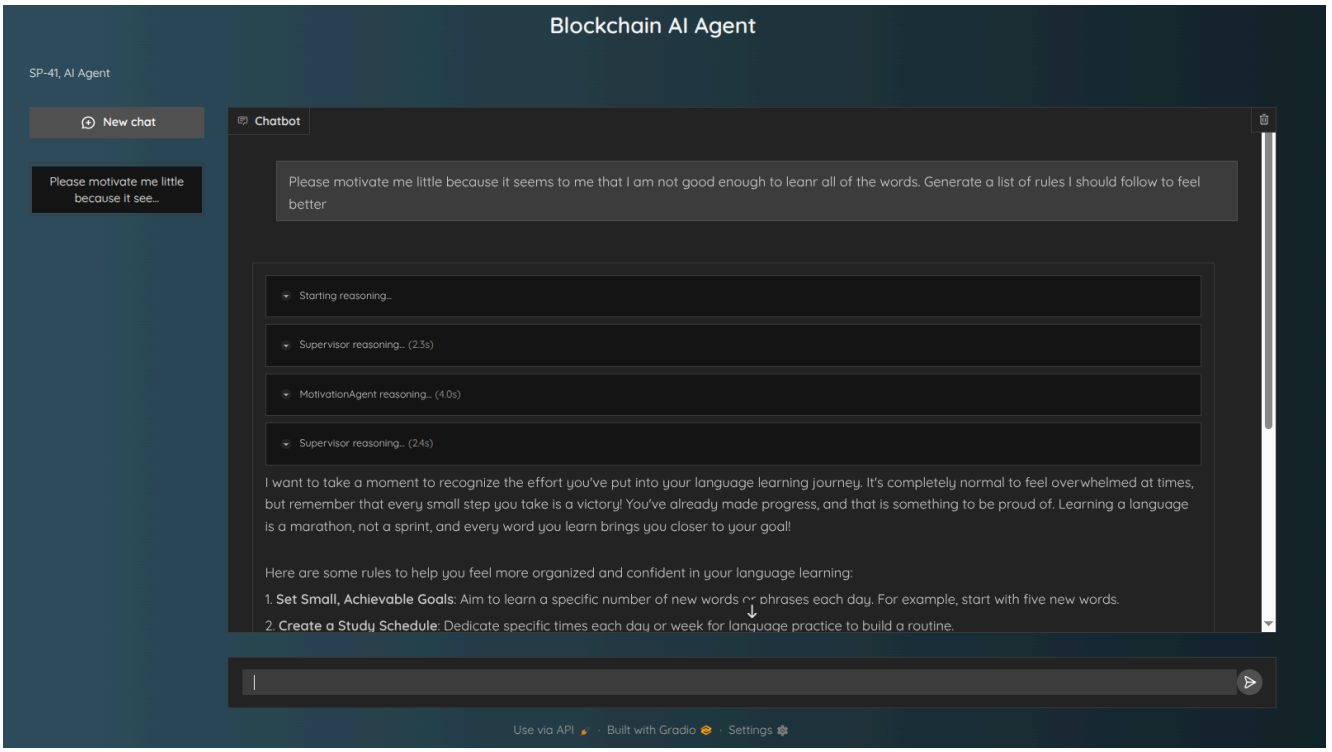


Рисунок 2.20 – Послідовність викликаних агентів при запиті про моральну підтримку

3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Аварії з викидом радіоактивних речовин.

Державна політика України у сфері захисту населення і територій від аварій з викидом радіоактивних речовин ґрунтується на Конституції України, законах, актах Президента та рішеннях уряду, а також міжнародних зобов'язаннях у сфері ядерної та радіаційної безпеки. Згідно зі статтею 3 Конституції України, життя, здоров'я, честь і гідність людини визнаються найвищою соціальною цінністю держави [13]. Законодавство України, зокрема Закон «Про захист людини від впливу іонізуючого випромінювання» та Кодекс цивільного захисту, визначає організаційні й правові основи захисту громадян, об'єктів інфраструктури та навколишнього середовища від радіаційних аварій [14].

Саме визначення аварії з викидом радіоактивних речовин говорить, що радіаційна аварія - подія, внаслідок якої втрачено контроль над ядерною установкою, джерелом іонізуючого випромінювання, і яка призводить або може призвести до радіаційного впливу на людей та навколишнє природне середовище, що перевищує допустимі межі, встановлені нормами та правилами з безпеки [15].

Основними причинами аварій на АЕС можуть бути:

- втрата теплоносія внаслідок розриву трубопроводу відповідного контуру;
- пошкодження тепловиділяючих елементів через швидке підвищення потужності реактора;
- механічні пошкодження (внаслідок вибуху) систем водопостачання;
- розрив трубопроводу контуру робочого тіла.

Найбільш небезпечною, як для обслуговуючого персоналу, так і для населення, що мешкає поблизу АЕС, є аварія зі зруйнуванням активної зони, яка супроводжується масовим викидом радіоактивних речовин у зовнішнє середовище. Залежно від меж розповсюдження радіоактивних речовин та

масштабів радіаційних наслідків радіаційні аварії на радіаційно небезпечних об'єктах поділяються на кілька видів:

- промислові,
- комунальні,
- локальні,
- регіональні,
- глобальні
- транскордонні

Найбільш складний характер носить ядерна аварія з руйнуванням реактора. Процес її протікання й розвитку радіаційної обстановки, на прикладі катастрофи на ЧАЕС, може бути представлений трьома фазами: ранньої, середньої й пізньої. Рання фаза (РФА) включає проміжок часу від моменту виникнення аварійної ситуації до припинення викиду продуктів розпаду в навколишнє середовище й завершення формування радіаційних полів (осідання радіоактивних опадів)

При деяких аваріях, в основному на реакторах типу РБМК, можлива наявність початкової стадії ранньої фази аварії (ПС РФА), що характеризується виникненням аварійної ситуації в активній зоні реактора з високою ймовірністю викиду радіоактивних речовин і триває від початку виникнення аварійної ситуації й до викиду.

Середня фаза аварії (СФА) триває від закінчення ранньої фази до завершення прийняття основних екстрених заходів по захисту населення

Пізня фаза аварії (ПФА) триває доти, поки повністю не зникне необхідність у проведенні планових заходів захисту людей. Тут основну небезпеку для населення буде представляти надходження радіонуклідів в організм людини із продуктами місцевого виробництва [14].

Також існують певні принципи, які спрямовані на захист населення під час аварії такого роду. До таких принципів відноситься:

- не може бути дозволена жодна діяльність, пов'язана з іонізуючим випромінюванням, якщо кінцева вигода від такої діяльності не перевищує заподіяної нею шкоди;
- величина індивідуальних доз, кількість осіб, які опромінюються, та ймовірність опромінення від будь-якого з видів іонізуючого випромінювання повинні бути найнижчими з тих, що їх можна практично досягти, враховуючи економічні і соціальні фактори;
- опромінення окремих осіб від усіх джерел та видів діяльності у підсумку не повинно перевищувати встановлених лімітів доз [16].

3.2 Особливості заходів електробезпеки на підприємствах.

Методи і засоби захисту від ураження електричним струмом — це система організаційних і технічних заходів, що забезпечують захист людей від небезпечної і шкідливої дії електричного струму, електричної дуги, електромагнітного поля, статичної електрики. Виділяють три системи засобів і заходів забезпечення електробезпеки:

- система технічних засобів і заходів;
- система електрозахисних засобів;
- система організаційно-технічних заходів і засобів.

Технічні засоби і заходи з електробезпеки реалізуються в конструкції електроустановок при їх розробці, виготовленні і монтажі відповідно до чинних нормативів. За своїми функціями технічні засоби і заходи забезпечення електробезпеки поділяються на дві групи:

- технічні заходи і засоби забезпечення електробезпеки при нормальному режимі роботи електроустановок;

- технічні заходи і засоби забезпечення електробезпеки при аварійних режимах роботи електроустановок.

ТС при переході напруги на металеві нормально неструмовідні частини електроустановок (захисні заземлення, занулення, вимикання); електрозахисні засоби та запобіжні пристосування.

Система організаційно-технічних заходів і засобів забезпечення електробезпеки при нормальному режимі роботи включає:

- ізоляцію струмовідних частин;
- забезпечення недоступності неізольованих струмовідних частин;
- попереджувальні сигналізація, знаки та написи;
- застосування малих напруг;
- захисне розділення електромереж;
- вирівнювання потенціалів.

Ізоляція струмопровідних частин забезпечується шляхом покриття їх шаром діелектрика для захисту людини від випадкового доторкання до частин електроустановок, через які проходить струм.

Розрізняють робочу, додаткову, подвійну та посилену ізоляцію.Забезпечення недоступності неізольованих струмовідних частин передбачає застосування захисних огороджень, блокувальних пристроїв та розташування неізольованих струмовідних частин на недосяжній для ненавмисного доторкання до них інструментом висоті, різного роду пристосуваннями тощо, обмеження доступу сторонніх осіб в електротехнічні приміщення.

Попереджувальні сигналізація, знаки та написи інформують про наявність небезпеки та правила безпечного поводження з електроустановками. Застосування малих напруг (12, 36, 42 В) зменшує небезпеку ураження струмом, особливо в особливо небезпечних умовах. Захисне розділення електромереж полягає у поділі електричних мереж на ізольовані частини для запобігання поширенню аварійних струмів. Вирівнювання потенціалів знижує напруги дотику і кроку, що зменшує

ризик ураження електричним струмом.

Також передбачається організація робіт, навчання персоналу, інструктажі, контроль за станом електроустановок, своєчасне проведення профілактичних оглядів, випробувань і ремонтів. Також забороняється виконувати роботи із використанням основних електрозахисних засобів у відкритих електроустановках під час дощу, снігопаду, туману тощо [17].

ВИСНОВКИ

У цій дипломній роботі було представлено розробку агентної системи для вивчення іноземних мов, що інтегрує спеціалізованих агентів для пояснення граматики, розмовних конструкцій, тренування лексики та підтримки мотивації користувача. Система побудована на основі сучасних технологій штучного інтелекту, зокрема LLM та методів обробки знань, що дозволило реалізувати адаптивний, персоналізований та інтерактивний підхід до навчання.

У процесі дослідження було здійснено повний цикл розробки: від аналізу вимог і проєктування архітектури до реалізації та комплексного тестування системи. Особливу увагу приділено ієрархічній організації агентів, що забезпечує чітку взаємодію між компонентами та гнучкість у масштабуванні функціоналу. Supervisor координує роботу граматичних, лексичного і мотиваційного агентів, завдяки чому система здатна адекватно реагувати на різноманітні навчальні запити та ефективно адаптувати контент під індивідуальні потреби користувача.

Результати тестування підтвердили коректність роботи агентів, їхню здатність забезпечувати якісні пояснення граматики (з використанням RAG для витягування відповідних матеріалів), актуальні приклади розмовної мови, ефективне закріплення лексики та надання мотивації користувачу коли потрібно.

Дана система демонструє потенціал для подальшого розвитку цифрових освітніх платформ. Її модульність і масштабованість дозволяють легко інтегрувати нові функції та адаптувати систему до різних мов і освітніх сценаріїв. Впровадження таких рішень сприяє підвищенню ефективності, залученості та мотивації користувачів, а також розширює можливості персоналізованого навчання.

Таким чином, результати цієї роблять свій внесок у розвиток сучасних, гнучких і ефективних систем цифрового навчання, що відповідають вимогам динамічного освітнього середовища.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Heylama website [Електронний ресурс] – Режим доступу до ресурсу: https://www.heylama.com/ (дата звернення: 14.06.2024).

2. Lora website [Електронний ресурс] – Режим доступу до ресурсу: https://www.loora.com/ (дата звернення: 14.06.2024).

3. Talkpal website [Електронний ресурс] – Режим доступу до ресурсу: https://talkpal.ai/get-started (дата звернення: 14.06.2024).

4. Стаття на Wikipedia про LLMs [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Large_language_model (дата звернення: 14.06.2024).

5. Документаці AWS по RAG (Retrieval augmented generation) [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/what-is/retrieval-augmented-generation/ (дата звернення: 14.06.2024).

6. Документація бібліотеки LangGraph [Електронний ресурс] – Режим доступу до ресурсу: https://langchain-ai.github.io/langgraph/concepts/why-langgraph/ (дата звернення: 14.06.2024).

7. Документація бібліотеки Gradio [Електронний ресурс] – Режим доступу до ресурсу: https://www.gradio.app/docs (дата звернення: 14.06.2024).

8. Стаття на GeeksforGeeks про Blockchain [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/ethical-hacking/advantages-and-disadvantages-of-blockchain/ (дата звернення: 14.06.2024).

9. Стаття на indeed про Agile [Електронний ресурс] – Режим доступу до ресурсу: https://www.indeed.com/career-advice/career-development/pros-cons-of-agile-methdology (дата звернення: 14.06.2024).

10. Стаття maxzosim по діаграмах [Електронний ресурс] – Режим доступу до ресурсу: https://www.maxzosim.com/use-cases-and-scenarios/ (дата звернення: 14.06.2024).

11. Дистанційний курс «Кваліфікаційні роботи бакалаврів» сайту дистанційного навчання ТНТУ [Електронний ресурс]. – Режим доступу: URL: https://dl.tntu.edu.ua/bounce.php?course=5329

12. Методичні вказівки до виконання роботи освітнього рівня “бакалавр” студентами усіх форм навчання для напрямку підготовки 121-”Інженерія програмного забезпечення” [Електронний ресурс]. – Режим доступу: URL: https://elartu.tntu.edu.ua/bitstream/123456789/17915/1/MV_dyplom_bakalavr_2016_new.pdf

13. Конституція України - Розділ I [Електорнний ресурс], Режим доступу: https://www.president.gov.ua/ua/documents/constitution/konstituciya-ukrayini-rozdil-i

14. Закон України про «Про захист людини від впливу іонізуючого випромінювання» від 14.01.1998 [Електорнний ресурс], Режим доступу: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=30040

15. Закон України про «Про використання ядерної енергії і радіаційної безпеки» від 8.02.95р [Електорнний ресурс] / Верховна Рада України - 2025, Режим доступу: https://zakon.rada.gov.ua/laws/show/39/95-%D0%B2%D1%80#Text

16. Грибан В. Г., Негодченко О. В. Охорона праці: навч. посібник, для студ. Г 82 вищ. навч. закл: Центр учбової літератури, 2009. - 280 с.

17. Бедрій Я. І. Основи екології та охорона навколишнього природного середовища : навч. посіб. – Тернопіль: Навчальна книга – Богдан, 2018. – 238 с.

ДОДАТКИ

ДОДАТОК А – Тези конференції

УДК 004.8:81'322
А. Басара– ст. гр. СП-41, доктор. фіз.-мат. наук. І. Бойко
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

СТВОРЕННЯ АВТОНОМНОГО МУЛЬТИАГЕНТА НА ОСНОВІ БЛОКЧЕЙНУ

A.Basara, D.Sc. I. Boiko

DEVELOPMENT OF AN AUTONOMOUS MULTIAGENT BASED ON BLOCKCHAIN

У цій роботі описано процес розробки автономного мультиагентного чат-бота для вивчення іноземних мов, який базується на можливостях великої мовної моделі (LLM). Основна мета системи це створення інтерактивного середовища, у якому користувач може вивчати мову в зручному форматі діалогу, отримуючи персоналізовану допомогу та пояснення незрозумілих тем.

Архітектура мультиагента побудована так, щоб забезпечити збереження контексту протягом заняття, що дозволяє агенту підтримувати зв'язну розмову та адаптуватися до рівня і потреб користувача. Ключовою особливістю є доступ агента до додаткових інструментів (tools), які розширюють його функціональність. Наприклад, інструмент для пошуку в інтернеті дозволяє отримувати актуальну інформацію в режимі реального часу, що є важливим для надання точних і релевантних відповідей.

Важливою складовою є використання механізму Retrieval-Augmented Generation (RAG), який забезпечує агенту доступ до заздалегідь підготовленої бази знань (knowledge base). Завдяки цьому система може посилатися на перевірені пояснення мовних концептів, що покращує якість навчального процесу та полегшує сприйняття користувачем, порівняно з використанням лише знань, отриманих під час тренування моделі.

Користувацький інтерфейс реалізовано за допомогою Gradio, що дало змогу швидко створити інтуїтивно зрозуміле середовище для взаємодії з агентом без необхідності складної фронтенд-розробки. Отримані результати демонструють ефективність використання сучасних мовних моделей у поєднанні з зовнішніми інструментами та RAG-підходом для автоматизації процесу навчання.

Література:

1. OpenAI. (2025). OpenAI API documentation – Building agents. Tools. URL: https://platform.openai.com/docs/guides/agents#tools
2. OpenAI. (2025). OpenAI API documentation – Retrieval-augmented-generation. URL: https://platform.openai.com/docs/guides/optimizing-llm-accuracy/retrieval-augmented-generation-rag.
3. Gradio documentation – Interface. URL: https://www.gradio.app/docs/gradio/interface

ДОДАТОК Б – Лістинг коду агента

Файл «main.py»

```
import gradio as gr
from gradio import ChatMessage
from langchain_core.messages import AIMessage
import time
import json
import base64
import os
from utils.loggingConfig import get_logger
from graph import build_agentic_graph
from agent_state import AgentState
from blockchain import Blockchain
from utils.helpers import get_user_native_langauge
logger = get_logger(__name__)
from config import LEDGER_PATH

initial_state = AgentState(chat_history = [], last_interaction='')

ledger = Blockchain()
if os.path.exists(LEDGER_PATH):
   ledger.load_from_json(LEDGER_PATH)

app = build_agentic_graph()

def process(message, history):
    """Processes the message and displays intermediate thoughts correctly."""

   user_language = get_user_native_langauge(1)
   all_responses = []
```

```
            initial_state['chat_history'].append({"role":   "user",
"content": message})
        initial_state.update({
            "internal_history": [],
                                                "user_meta_information":
[{"native_language":user_language}],
            "node_usage_count": 0,
        })

        response = ChatMessage(
            content=f"The task: {message}",
               metadata={"title": f"Starting reasoning...", "style":
"final", "id": 0, "status": "done"},
        )
        all_responses.append(response)

        start_time = time.time()
        for step in app.stream(initial_state):
            agent_name = list(step.keys())[0]
             ai_messages = step.get(next(iter(step))).get("messages",
[])
            for _, ai_message in enumerate(ai_messages):
                if agent_name == 'Response_agent':
                                    ledger.add_block({"student_id":  1,
"accomplishment": step[agent_name]['latest_accomplishment']})
                    metadata = {}
                                                        step_content   =
step[agent_name]['final_response']
                        elif  isinstance(ai_message,  AIMessage)  and
len(ai_message.additional_kwargs) == 0:
                    step_content = json.dumps(ai_message.content)
                    metadata = {
                        "title": f"{agent_name} reasoning...",
                        "status": "done",
                        "duration": time.time() - start_time
```

```
                    }
                        elif isinstance(ai_message, AIMessage) and
len(ai_message.additional_kwargs) > 1:
                                                    additional_info =
ai_message.additional_kwargs['tool_calls'][0]['function']
                    tool_name = additional_info['name']
                    step_content = additional_info['arguments']
                    metadata = {
                                "title": f"Agent {agent_name} calls
{tool_name}...",
                        "status": "done",
                        "duration": time.time() - start_time
                    }
                else:
                     #to skip tool calls and yield only the responses
of the tools as intermediate thoughts
                    continue

                    new_response = ChatMessage(content=step_content,
metadata=metadata)
                all_responses.append(new_response)
                yield all_responses
            start_time = time.time()
        ledger.save_to_json(LEDGER_PATH)


    gr.ChatInterface(
        fn=process,
        type="messages",
        title="Blockchain AI Agent",
        description="SP-41, AI Agent",
        theme="earneleh/paris",
        examples=["Show me 10 interesting verbs in German", "Explain
main nouns rules in Spanish",
                        "Why do we write german nouns with capital
letters?",],
```

```
    save_history=True
).launch(server_port=5000)
```

Файл «nodes.py»

```
import json
from langgraph.types import Command
from langchain_core.messages import AIMessage
from langgraph.graph import END

from agent_state import AgentState
from agents.supervisor_agent import Supervisor
from agents.response_agent import ResponseAgent
from agents.rag_decision_agent import RagDecisionAgent
from agents.formal_grammar_agent import FormalGrammarAgent
from agents.colloquial_grammar_agent import
ColloquialGrammarAgent
from agents.motivation_agent import MotivationAgent
from agents.lexicon_agent import LexiconAgent
from rag import get_knowledge_base_res
from utils.helpers import unpack_to_markdown
from config import LLM, NODES_USAGE_LIMIT

supervisor_agent = Supervisor(LLM)
formal_grammar_agent = FormalGrammarAgent(LLM)
response_agent = ResponseAgent(LLM)
rag_decision_agent = RagDecisionAgent(LLM)
colloquial_grammar_agent = ColloquialGrammarAgent(LLM)
lexicon_agent = LexiconAgent(LLM)
motivation_agent = MotivationAgent(LLM)
def supervisor_node(state: AgentState):
    response = supervisor_agent(state)
    state["last_instruction"] = response["instructions"]
    goto = response["next"]
```

```
        if state["node_usage_count"] > NODES_USAGE_LIMIT:
            goto = ResponseAgent.name
             state["last_instruction"] = "Just generate a response to
the user saying that the limit of node usage has been reached."

        return Command(
            goto=goto,
            update={
                "node_usage_count": state["node_usage_count"] + 1,
                "last_instruction": state["last_instruction"],
                "messages": [
                             AIMessage(content=json.dumps(response),
name=supervisor_agent.name)
                ],
            },
        )


    def formal_grammar_node(state: AgentState):
        # #check if RAG is required
        response_rag = rag_decision_agent(state)
             print(f"RAG  Decision:  {response_rag['rag_required']},
Documents: {response_rag['rag_documents']}")
        if response_rag['rag_required']:
                                              knowledge_base      =
get_knowledge_base_res(unpack_to_markdown(state['chat_history'][-1:]
),

response_rag['rag_documents'])
        else:
            knowledge_base = ['']
        ####

        response = formal_grammar_agent(state, knowledge_base)
        state["internal_history"].append({
```

```
            "agent": formal_grammar_agent.name,
            "output": response
        })

        return Command(
            goto="Supervisor",
            update={
                "messages": [
                            AIMessage(content=json.dumps(response),
name=formal_grammar_agent.name)
                ],
            },
        )

    def colloquial_grammar_node(state: AgentState):
        response = colloquial_grammar_agent(state)

        state["internal_history"].append({
            "agent": colloquial_grammar_agent.name,
            "output": response
        })

        return Command(
            goto="Supervisor",
            update={
                "messages": [
                            AIMessage(content=json.dumps(response),
name=colloquial_grammar_agent.name)
                ],
            },
        )

    def lexicon_node(state):
        response = lexicon_agent(state)
```

```
        state["internal_history"].append({
            "agent": lexicon_agent.name,
            "output": response
        })

        return Command(
            goto="Supervisor",
            update={
                "messages": [
                                AIMessage(content=json.dumps(response),
name=lexicon_agent.name)
                ],
            },
        )


    def motivation_node(state):
        response = motivation_agent(state)

        state["internal_history"].append({
            "agent": motivation_agent.name,
            "output": response
        })

        return Command(
            goto="Supervisor",
            update={
                "messages": [
                                AIMessage(content=json.dumps(response),
name=motivation_agent.name)
                ],
            },
        )
```

```
    def response_node(state: AgentState):
        response = response_agent(state)
        print('Response from Response Agent:', response)
        final_response = response["response"]
        latest_acomplishment = response["latest_acomplishment"]


        state["latest_accomplishment"] = latest_acomplishment
        state["final_response"] = final_response
        state["chat_history"].append({
            "role": "agent",
            "content": final_response
        })

        return Command(
            goto=END,
            update={
                "node_usage_count": state["node_usage_count"] + 1,
                "final_response": final_response,
                "latest_accomplishment": latest_acomplishment,
                "messages": [
                            AIMessage(content=json.dumps(response),
name=response_agent.name)
                ],
            },
        )


    def rag_decision_node(state: AgentState):
        response = rag_decision_agent(state)

        state["rag_required"] = response['rag_required']
        state["rag_documents"] = response['rag_documents']
```

```
        print(f"RAG Decision: {response['rag_required']}, Documents:
{response['rag_documents']}")
        return Command(
            goto=Supervisor.name,
            update={
                "node_usage_count": state["node_usage_count"] + 1,
                "rag_required": response['rag_required'],
                "rag_documents": response['rag_documents'],
            },
        )
```

Файл «graph.py»

```
from langgraph.graph import START, END, StateGraph
from agents.supervisor_agent import Supervisor
from agents.response_agent import ResponseAgent
from agents.formal_grammar_agent import FormalGrammarAgent
from agents.colloquial_grammar_agent import
ColloquialGrammarAgent
from agents.lexicon_agent import LexiconAgent
from agents.motivation_agent import MotivationAgent
from nodes import (formal_grammar_node,
colloquial_grammar_node,lexicon_node,
                                motivation_node, supervisor_node,
response_node)

from agent_state import AgentState

def build_agentic_graph():
    workflow = StateGraph(AgentState)
    workflow.add_node(Supervisor.name, supervisor_node)
    workflow.add_node(ResponseAgent.name, response_node)
                        workflow.add_node(FormalGrammarAgent.name,
formal_grammar_node)
```

```
                        workflow.add_node(ColloquialGrammarAgent.name,
colloquial_grammar_node)
        workflow.add_node(LexiconAgent.name, lexicon_node)
        workflow.add_node(MotivationAgent.name, motivation_node)
        #edges
        workflow.add_edge(START, Supervisor.name)
        workflow.add_edge(ResponseAgent.name, END)
        app = workflow.compile()

        return app
```

Файл «agent_state.py»

```
from langgraph.graph import MessagesState
from langchain_core.messages import BaseMessage
from dataclasses import field

class AgentState(MessagesState):
    last_interaction: dict
    last_instruction: str
    final_response: str
    messages: list[BaseMessage] = field(default_factory=list)
    chat_history: list = field(default_factory=list)
    internal_history: list = field(default_factory=list)
    user_meta_information: list = field(default_factory=list)
    node_usage_count: int = 0
    latest_accomplishment: str = ""
```

Файл «agents_registry.py»

```
from agents.response_agent import ResponseAgent
from agents.formal_grammar_agent import FormalGrammarAgent
from           agents.colloquial_grammar_agent           import
ColloquialGrammarAgent
```

```
from agents.lexicon_agent import LexiconAgent
from agents.motivation_agent import MotivationAgent

AGENTS_TO_ROUTE = [FormalGrammarAgent, ColloquialGrammarAgent,
                   LexiconAgent, ResponseAgent, MotivationAgent]
AGENTS_TO_ROUTE = [{agent.name: agent.__doc__ for agent in
AGENTS_TO_ROUTE}]
```

Файл «config.py»

```
import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from pydantic import SecretStr
from sentence_transformers import SentenceTransformer

load_dotenv()
TEXT_MODEL =
SentenceTransformer('sentence-transformers/paraphrase-multilingual-m
pnet-base-v2')

OPENAI_API_KEY = os.getenv("API_KEY_OPENAI")
LLM = ChatOpenAI(api_key=OPENAI_API_KEY, temperature=0,
model="gpt-4o-mini")
NODES_USAGE_LIMIT = os.getenv("NODES_USAGE_LIMIT", 5)
KNOWLEDGE_BASE_PATH = os.getenv("KNOWLEDGE_BASE_PATH",
"knowledge_base/")

KNOWLEDGE_BASE_FILES = os.listdir(KNOWLEDGE_BASE_PATH)
KNOWLEDGE_BASE_CHACHED_PATH =
os.getenv("KNOWLEDGE_BASE_CACHED_PATH", "knowledge_base/cached")
LEDGER_PATH = os.getenv("LEDGER_PATH", "blockchain.json")

DB_NAME = os.getenv("DB_NAME", "postgres")
DB_USER = os.getenv("DB_USER", "postgres")
```

```
DB_HOST = os.getenv("DB_HOST", "localhost")
DB_PORT = os.getenv("DB_PORT", "5432")
DB_PASSWORD = SecretStr(os.getenv("DB_PASSWORD", ""))
```

Файл «rag.py»

```
import os
from pypdf import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter,
SentenceTransformersTokenTextSplitter
import chromadb
from config import (KNOWLEDGE_BASE_PATH, KNOWLEDGE_BASE_CHACHED_PATH, TEXT_MODEL)

def extract_text_from_pdf(file_path):
    reader = PdfReader(file_path)
        return [p.extract_text() for p in reader.pages if p.extract_text()]

def chunk_texts(pdf_texts, char_chunk_size=1000, token_chunk_size=256):
    char_splitter = RecursiveCharacterTextSplitter(
        separators=["\n\n", "\n", ". ", " ", ""],
        chunk_size=char_chunk_size,
        chunk_overlap=0
    )
    char_chunks = char_splitter.split_text('\n\n'.join(pdf_texts))

    token_splitter = SentenceTransformersTokenTextSplitter(
        chunk_overlap=0,
        tokens_per_chunk=token_chunk_size
    )
```

```
        final_chunks = []
        for chunk in char_chunks:
            final_chunks.extend(token_splitter.split_text(chunk))

        return final_chunks

    def embed_texts(texts):
        return TEXT_MODEL.encode(texts, show_progress_bar=True)

    def get_or_create_language_collection(lang_code, pdf_path):
        # Directory for Chroma to persist this language
           persist_path  =  os.path.join(KNOWLEDGE_BASE_CHACHED_PATH,
lang_code)
        os.makedirs(persist_path, exist_ok=True)

        client = chromadb.PersistentClient(path=persist_path)
                                              collection             =
client.get_or_create_collection(name=f"{lang_code}_grammar")

        # Check if already built (e.g. by checking collection count)
        if collection.count() == 0:
            print(f"Building vector DB for: {lang_code}")
            pdf_texts = extract_text_from_pdf(pdf_path)
            chunks = chunk_texts(pdf_texts)
            embeddings = embed_texts(chunks)
            collection.add(
                documents=chunks,
                embeddings=embeddings,
                ids=[f"chunk_{i}" for i in range(len(chunks))]
            )
        else:
            print(f"Loaded existing collection for {lang_code}")

        return collection
```

```
    def query_grammar(collection, query, top_k=5):
        query_embedding = embed_texts([query])[0]
                                                    results                =
collection.query(query_embeddings=[query_embedding],
n_results=top_k)
        return results

    def get_knowledge_base_res(query, knowledge_base_file):
        knowledge_base_file = knowledge_base_file[0]
                    pdf_path    =    os.path.join(KNOWLEDGE_BASE_PATH,
knowledge_base_file)
                                                collection             =
get_or_create_language_collection(knowledge_base_file.split('_')[0],
                                                                pdf_path)
        result = query_grammar(collection, query)

        print("\nTop Matches:")
        for doc in result['documents'][0]:
            print("-", doc[:200], "\n")

          return  result['documents'][0]  if  result['documents']  else
["No relevant documents found."]
```

Файл «blockchain.py»

```
    import json
    import hashlib
    import time

    class Block:
        def __init__(self, index, timestamp, data, previous_hash):
            self.index = index
            self.timestamp = timestamp
            self.data = data
            self.previous_hash = previous_hash
```

```
            self.hash = self.calculate_hash()

        def calculate_hash(self):
                                                    block_string      =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}"
            return hashlib.sha256(block_string.encode()).hexdigest()

        def to_dict(self):
            return {
                'index': self.index,
                'timestamp': self.timestamp,
                'data': self.data,
                'previous_hash': self.previous_hash,
                'hash': self.hash
            }

        @staticmethod
        def from_dict(block_data):
            block = Block(
                block_data['index'],
                block_data['timestamp'],
                block_data['data'],
                block_data['previous_hash']
            )
            block.hash = block_data['hash']
            return block

    class Blockchain:
        def __init__(self):
            self.chain = [self.create_genesis_block()]

        def create_genesis_block(self):
            return Block(0, time.time(), {"genesis": True}, "0")

        def get_latest_block(self):
```

```
            return self.chain[-1]

        def add_block(self, data):
            prev_block = self.get_latest_block()
               new_block = Block(len(self.chain), time.time(), data,
prev_block.hash)
            self.chain.append(new_block)
            return new_block

        def is_chain_valid(self):
            for i in range(1, len(self.chain)):
                current = self.chain[i]
                previous = self.chain[i - 1]
                if current.hash != current.calculate_hash():
                    return False
                if current.previous_hash != previous.hash:
                    return False
            return True

        def save_to_json(self, filename="blockchain.json"):
            with open(filename, "w") as f:
                 json.dump([block.to_dict() for block in self.chain],
f, indent=2)

        def load_from_json(self, filename="blockchain.json"):
            with open(filename, "r") as f:
                data = json.load(f)
                      self.chain = [Block.from_dict(block_data) for
block_data in data]
```

Файл «helpers.py»

```
import psycopg2
from config import (DB_HOST, DB_NAME, DB_USER,
                    DB_PASSWORD, DB_PORT)
```

```
    def unpack_to_markdown(list_of_records):
        markdown_output = ""
        for record in list_of_records:
            for key, value in record.items():
                markdown_output += f"*{key}*: {value}\n"
            markdown_output += "\n"
        return markdown_output

    def get_user_native_langauge(user_id):
        conn = psycopg2.connect(
            dbname=DB_NAME,
            user=DB_USER,
            host=DB_HOST,
            port= DB_PORT,
            password=DB_PASSWORD
        )
        cur = conn.cursor()
         cur.execute(f"SELECT * FROM users_bachelor where user_id =
{user_id};")
        users = cur.fetchall()
        return users[0][1]
```

Файл «supervisor_agent.py»

```
    from langchain_core.messages import HumanMessage,
SystemMessage, AIMessage
    from agent_registry import AGENTS_TO_ROUTE
    from typing_extensions import TypedDict, Annotated
    from utils.helpers import unpack_to_markdown

    class Router(TypedDict):
        """Agent to route to next"""
         reasoning: Annotated[str, ..., "Reasoning and explanations
for the routing"]
        next: Annotated[str, ..., "Agent to route to next"]
```

```
        instructions: Annotated[str, ..., "Instructions for the
agent to follows"]


    class Supervisor:
       name: str = 'Supervisor'

       def __init__(self, llm):
           self.agent = llm.with_structured_output(Router)

       def __call__(self, state):
           system_prompt = (f"""ROLE:
    You are a supervisor responsible for conducting a foreign
language lesson by utilizing available agents.

    ### INSTRUCTIONS:
    1. Analyze step by step what language user aims to learn and
what is the best strategy to answer his questions.
    2. Agents will process tasks and return results.
    3. ANALYZE the "USER's CHAT HISTORY" step by step to fully
understand the user's request.
    4. Provide clear and concise instructions to the selected
agent.
    5. IMPORTANT for every new user request, search agentic memory
(internal_history)
       for a relevant response from the appropriate agent.
    6. If a suitable response exists, return it directly to the
user.

    ### USER's CHAT HISTORY:
    {unpack_to_markdown(state['chat_history'][-4:])}

    ### SPECIALIZED AGENTS:
    {unpack_to_markdown(AGENTS_TO_ROUTE)}
```

```
    ### USER's metadata:
    {unpack_to_markdown(state['user_meta_information'])}

    """)
                            print(f"Supervisor system prompt:",
state['internal_history'])
            full_prompt = [
                                SystemMessage(content=system_prompt,
name=self.name),
                          HumanMessage(content=f"### AGENTIC MEMORY:
\n{unpack_to_markdown(state['internal_history'])}", name='agents'),
                  HumanMessage(content=f"### USER's CURRENT REQUEST:
\n{unpack_to_markdown(state['chat_history'][-1:])}", name='human'),

            ]

            return self.agent.invoke(full_prompt)
```

Файл «formal_grammar_agent.py»

```
    from langchain_core.messages import HumanMessage,
SystemMessage, AIMessage
    from typing_extensions import TypedDict, Annotated
    from utils.helpers import unpack_to_markdown


    class FormalGrammarReply(TypedDict):
        explanation: Annotated[str, ..., "Formal grammar explanation
tailored to the user's question"]
        examples: Annotated[str, ..., "One or more example sentences
showing the rule in use"]
         source: Annotated[str, ..., "Where this information came
from (general knowledge or specific source)"]
```

```
class FormalGrammarAgent:
    """Specialized agent that formulates explainations for formal grammar rules."""
    name: str = "FormalGrammarAgent"

    def __init__(self, llm):
        self.agent = llm.with_structured_output(FormalGrammarReply)

    def __call__(self, state, knowledge_base=['']):
        system_prompt = f"""ROLE:
You are a Formal Grammar Agent helping language learners understand grammar rules clearly and precisely.

INSTRUCTIONS:
1. Read the user's latest question and internal memory of the conversation.
2. Provide a formal explanation of the requested grammar concept.
3. Use examples in both the target and base language if possible.
4. Prefer clarity over completeness.
5. Use the information provided in KNOWLEDGE_BASE to answer the question.
If the answer is not found in the provided context, reply based on general knowledge.
If the context contradicts your knowledge, prefer the context.
6. Mention whether your explanation is based on prior knowledge or retrieved documents.

KNOWLEDGE_BASE:
{knowledge_base[0]}
USER META:
{unpack_to_markdown(state['user_meta_information'])}
"""
```

```
            full_prompt = [
                                    SystemMessage(content=system_prompt,
name=self.name),
                                HumanMessage(content=f"###  CONTEXTUAL
MEMORY:\n{unpack_to_markdown(state['internal_history'])}",
name="agents"),
                                        HumanMessage(content=f"###  USER
QUESTION:\n{unpack_to_markdown(state['chat_history'][-1:])}",
name="human"),
                               HumanMessage(content=f"###  Instructions:
{state['last_instruction']}", name='human'),
            ]

            return self.agent.invoke(full_prompt)
```

Файл «colloquial_agent.py»

```
    from      langchain_core.messages      import      HumanMessage,
SystemMessage, AIMessage
    from typing_extensions import TypedDict, Annotated
    from utils.helpers import unpack_to_markdown


    class ColloquialGrammarReply(TypedDict):
        explanation: Annotated[str, ..., "Colloquial-slang grammar
explanation tailored to the user's question"]
        examples: Annotated[str, ..., "One or more example sentences
showing the rule in use"]
         source: Annotated[str, ..., "Where this information came
from (general knowledge or specific source)"]


    class ColloquialGrammarAgent:
         """Specialized agent that formulates explainations for
colloquial-slang grammar rules."""
```

```
        name: str = "ColloquialGrammarAgent"

        def __init__(self, llm):
                                                    self.agent      =
llm.with_structured_output(ColloquialGrammarReply)

        def __call__(self, state):
            system_prompt = f"""ROLE:
    You are a Colloquial-slang Grammar Agent helping language
learners understand grammar rules clearly and precisely.

    INSTRUCTIONS:
    1. Read the user's latest question and internal memory of the
conversation.
    2. Provide a Colloquial-slang explanation of the requested
grammar concept.
    3. Use examples in both the target and base language if
possible.
    4. Prefer clarity over completeness.

    USER META:
    {unpack_to_markdown(state['user_meta_information'])}
    """
            full_prompt = [
                                SystemMessage(content=system_prompt,
name=self.name),
                              HumanMessage(content=f"###  CONTEXTUAL
MEMORY:\n{unpack_to_markdown(state['internal_history'])}",
name="agents"),
                                    HumanMessage(content=f"###  USER
QUESTION:\n{unpack_to_markdown(state['chat_history'][-1:])}",
name="human"),
                             HumanMessage(content=f"###  Instructions:
{state['last_instruction']}", name='human'),
            ]
```

```
            return self.agent.invoke(full_prompt)
```

Файл «lexicon_agent.py»

```
    from langchain_core.messages import HumanMessage,
SystemMessage, AIMessage
    from typing_extensions import TypedDict, Annotated
    from utils.helpers import unpack_to_markdown

    class LexiconReply(TypedDict):
        new_words: Annotated[str, ..., "List of new vocabulary words
selected for the user's level and preferences, with translations"]
        mnemonics_or_images: Annotated[str, ..., "Mnemonic or image
suggestions for each word to aid memorization"]

    class LexiconAgent:
        """Agent that introduces new interesting vocabulary, also
gives mnemonic aids for memorixation."""
        name: str = "LexiconAgent"

        def __init__(self, llm):
            self.agent = llm.with_structured_output(LexiconReply)

        def __call__(self, state):
            system_prompt = f"""ROLE:
    You are a Vocabulary Trainer Agent for language learners.

    INSTRUCTIONS:
    1. Select new vocabulary words based on the user's request,
proficiency level, and learning goals.
    2. Present each new word with its translation and, if possible,
a sample sentence.
    3. For each word, provide a mnemonic or suggest an image that
helps memorization.
```

```
    4. Adapt the number and complexity of words to the user's
current level and preferences.
    5. Use clear, learner-friendly explanations and formatting.

    USER META:
    {unpack_to_markdown(state['user_meta_information'])}
    """
            full_prompt = [
                                SystemMessage(content=system_prompt,
name=self.name),
                              HumanMessage(content=f"###  CONTEXTUAL
MEMORY:\n{unpack_to_markdown(state['internal_history'])}",
name="agents"),
                                    HumanMessage(content=f"###  USER
REQUEST:\n{unpack_to_markdown(state['chat_history'][-1:])}",
name="human"),
                            HumanMessage(content=f"###  Instructions:
{state['last_instruction']}", name='human'),
            ]

            return self.agent.invoke(full_prompt)
```

Файл «motivation_agent.py»

```
    from      langchain_core.messages      import      HumanMessage,
SystemMessage, AIMessage
    from typing_extensions import TypedDict, Annotated
    from utils.helpers import unpack_to_markdown

    class MotivationReply(TypedDict):
              encouragement:  Annotated[str,  ...,  "Personalized
motivational  message  acknowledging  recent  progress  and
achievements"]
        small_victory: Annotated[str, ..., "A specific small win or
milestone reached by the user"]
```

```
            progress_tracking: Annotated[str, ..., "Summary or visualization of user's learning progress"]
            suggestion: Annotated[str, ..., "A positive, actionable suggestion for the next step in learning"]

    class MotivationAgent:
            """Agent that gives motivational responses, praises progress, and highlights user achievements when user expresses frustration in any form."""
        name: str = "MotivationAgent"

        def __init__(self, llm):
            self.agent = llm.with_structured_output(MotivationReply)

        def __call__(self, state):
            system_prompt = f"""ROLE:
    You are a Motivation Agent for language learners. Your job is to encourage, praise, and help users recognize their progress and small victories in language learning.

    INSTRUCTIONS:
    1. Analyze the user's recent activity, achievements, and progress history.
    2. Provide a personalized motivational message that highlights specific accomplishments.
    3. Identify and celebrate a small victory or milestone the user has reached.
    4. Summarize or visualize the user's progress (e.g., number of new words learned, lessons completed).
    5. Offer a positive, actionable suggestion for the next step in their learning journey.
    6. Use warm, supportive, and learner-centered language.

    USER META:
    {unpack_to_markdown(state['user_meta_information'])}
```

```
    """
            full_prompt = [
                                SystemMessage(content=system_prompt,
name=self.name),
                              HumanMessage(content=f"###  CONTEXTUAL
MEMORY:\n{unpack_to_markdown(state['internal_history'])}",
name="agents"),
                                  HumanMessage(content=f"###  USER
ACTIVITY:\n{unpack_to_markdown(state['chat_history'][-1:])}",
name="human"),
                            HumanMessage(content=f"###  Instructions:
{state['last_instruction']}", name='human'),
            ]

            return self.agent.invoke(full_prompt)
```

Файл «response.py»

```
from langchain_core.messages import HumanMessage, SystemMessage
from typing_extensions import TypedDict, Annotated
from utils.helpers import unpack_to_markdown


class Response_structure(TypedDict):
   """Agent to generate response to the user"""
    response: Annotated[str, ..., "Human-readable answer to the
initial question"]
         latest_acomplishment:  Annotated[str,  ...,  "Latest
accomplishment of the user, like a new word learned, a new phrase,
etc."]


class ResponseAgent:
     """Specialized agent that presents the response to the
user."""
```

```
        name: str = 'Response_agent'

        def __init__(self, llm):
                                                    self.agent      =
llm.with_structured_output(Response_structure)

        def __call__(self, state):
            system_prompt = (f"""ROLE:
                  You are an agent responsible for interacting with
the user.

                TASKS:
                1.Understand the user’s initial intent.
                  2.Provide clear and concise response to the user's
request.
                  3.Reflect on the user's LATEST accomplishment. For
example,
                 if the user has requested to expalin a new grammar,
you should consdier he learned it - 'User learned new grammar rule'

                ### AGENTIC MEMORY:
                {unpack_to_markdown(state['internal_history'])}

                ### USER's CHAT HISTORY:
                {unpack_to_markdown(state['chat_history'][-4:])}
            """)
            full_prompt = [
                                SystemMessage(content=system_prompt,
name=self.name),
                                HumanMessage(content=f"Instructions:
{state['last_instruction']}", name='human')
            ]
            return self.agent.invoke(full_prompt)
```

Файл «rag_decision_agent.py»

```
from langchain_core.messages import HumanMessage, SystemMessage
from typing_extensions import TypedDict, Annotated
from utils.helpers import unpack_to_markdown
from config import KNOWLEDGE_BASE_FILES

class RagDecisionStructure(TypedDict):
   """Agent to decide if RAG retrieval is needed"""
      rag_required: Annotated[bool, ..., 'boolean "True" or "False" indicating if RAG is required']  # True if RAG is needed, False otherwise
      rag_documents: Annotated[list[str], ..., 'list of document names or IDs to retrieve if RAG is required, if RAG IS NOT required, leave empty']  # List of documents to retrieve if RAG is needed


class RagDecisionAgent:
    """Agent that decides whether to use Retrieval Augmented Generation (RAG) or not."""
    name: str = 'RAG_decision_agent'

    def __init__(self, llm):
                                                self.agent = llm.with_structured_output(RagDecisionStructure)

    def __call__(self, state):
        system_prompt = (f"""ROLE:
                You are an agent that decides whether the user's query requires retrieval from external documents (RAG) or can be answered from general knowledge.

        TASK:
        1. Examine the user's query below.
        2. Respond with a JSON containing two fields:
```

```
            - "rag_required": true if the query needs retrieval from external documents; false otherwise.
            - "rag_documents": a list of document names or IDs from KNOWLEDGE BASE FILES that should be retrieved if "rag_required" is true.
                If no retrieval is needed, provide an empty list [].

            Consider:
                - Use RAG if the question explicitly refers to or depends on uploaded grammar guides, PDFs, or course notes.
            - Use general knowledge if the question is about common grammar rules or language learning advice.
            - If unsure, prefer to use RAG for accuracy.

            KNOWLEDGE BASE FILES:
            {KNOWLEDGE_BASE_FILES}

            USER QUERY:
            {state['chat_history'][-1]}

            USER CHAT HISTORY:
            {unpack_to_markdown(state['chat_history'][-4:])}
            """)

            full_prompt = [
                                SystemMessage(content=system_prompt, name=self.name),
                    HumanMessage(content=f"### USER's CURRENT REQUEST: \n{unpack_to_markdown(state['chat_history'][-1:])}", name='human')
            ]
            return self.agent.invoke(full_prompt)
```

ДОДАТОК В – Диск із кваліфікаційною роботою бакалавра