МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА МІКРОСЕРВІСНОГО ВЕБЗАСТОСУНКУ З ІНТЕГРАЦІЄЮ РЕКОМЕНДАЦІЙНОЇ СИСТЕМИ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ»**

Виконала: студентка 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.О. Горпинич
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Горпинич Валерії Олегівні
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка мікросервісного вебзастосунку з інтеграцією рекомендаційної системи та великих мовних моделей

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз вимог та постановка задачі.
2. Проєктування системи вебзастосунку.
3. Реалізація та тестування вебзастосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	30.01.2026	
3.	Обробка методичних та теоретичних	23.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	16.03.2026	
5.	Розробка третього розділу.	28.04.2026	
6.	Оформлення та нормоконтроль	25.06.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	15.06.2026	

Студент _____ В.О. Горпинич
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка мікросервісного вебзастосунку з інтеграцією рекомендаційної системи та великих мовних моделей»: 59 с., 39 рис., 5 табл., 19 джерел.

ВЕКТОРНІ ПРЕДСТАВЛЕННЯ, ВЕЛИКІ МОВНІ МОДЕЛІ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ПЕРСОНАЛІЗОВАНІ РЕКОМЕНДАЦІЇ, СЕМАНТИЧНИЙ ПОШУК, AI-АСИСТЕНТ, RAG-ПІДХІД, REDIS-КЕШУВАННЯ.

Об’єкт дослідження – процес розробки мікросервісних вебзастосунків із впровадженням великих мовних моделей.

Мета роботи: розробити мікросервісний вебзастосунок, який інтегрує великі мовні моделі для впровадження персоналізованої рекомендаційної системи, семантичного пошуку та AI-взаємодії, для покращення користувацького досвіду.

Методи дослідження – методи об’єктно-орієнтованого проєктування, мікросервісної архітектури, семантичного пошуку, інтеграції великих мовних моделей у вебзастосунки.

У роботі розглянуто процес проєктування та реалізації мікросервісного вебзастосунку для публікації технічних статей із інтеграцією великих мовних моделей для реалізації персоналізованих рекомендацій, семантичного пошуку та AI-взаємодії. Для розробки програмної частини використано Spring Boot, Spring AI, PostgreSQL, Pgvector, Redis, Angular, Docker та Vertex AI. У результаті створено працездатний вебзастосунок із мікросервісною архітектурою та інтеграцією AI-функціоналу. Для перевірки роботи системи було виконано UI-тестування за допомогою Selenium і навантажувальне тестування з використанням Apache JMeter, результати яких підтвердили коректність та стабільність роботи основного функціоналу застосунку.

SUMMARY

Bachelor’s qualifying paper “Development of a Microservices-Based Web Application Integrating a Recommendation System and Large Language Models”: 59 pages, 39 figures, 5 tables, 19 references.

AI ASSISTANT, EMBEDDINGS, LARGE LANGUAGE MODELS, MICROSERVICE ARCHITECTURE, PERSONALIZED RECOMMENDATIONS, RAG APPROACH, REDIS CACHING, SEMANTIC SEARCH.

The object of the study is the process of developing microservices-based web applications with the integration of large language models.

The aim of the study is to develop a microservices-based web application integrating large language models for implementing a personalized recommendation system, semantic search, and AI-based interaction to improve user experience.

The methods of research include object-oriented design methods, microservice architecture, semantic search, and technologies for integrating large language models into web applications.

The paper considers the process of designing and implementing a microservices-based web application for publishing technical articles with the integration of large language models for personalized recommendations, semantic search, and AI-based interaction. Spring Boot, Spring AI, PostgreSQL, Pgvector, Redis, Angular, Docker, and Vertex AI were used for the implementation of the software system. As a result, a functional web application with a microservice architecture and integrated AI functionality was developed. To verify the correctness of the system operation, UI testing using Selenium and load testing using Apache JMeter were performed. The testing results confirmed the correctness and stability of the main functionality of the application.

ЗМІСТ

Завдання на кваліфікаційну роботу ……2
Реферат 4
Summary 5
Вступ..... 7
1 Аналіз існуючих рішень та вимог до системи 9
1.1 Опис програмних засобів 9
1.2 Огляд аналогічних рішень..... 11
1.3 Аналіз вимог 14
2 Проєктування системи..... 17
2.1 Концептуальне проєктування системи 17
2.2 Логічна архітектура проєкту..... 24
2.3 Проєктування фізичної моделі 35
3 Реалізація та тестування 39
3.1 Програмна реалізація..... 39
3.2 Огляд результатів реалізації 45
3.3 Тестування системи 50
Висновки 57
Перелік посилань..... 58

ВСТУП

В умовах стрімкого розвитку інформаційних технологій вебзастосунки стають одним із основних інструментів для отримання, обробки та поширення інформації. Особливої актуальності набувають системи, що поєднують роботу з контентом із сучасними технологіями штучного інтелекту, зокрема рекомендаційними системами та великими мовними моделями. Такі рішення дозволяють не лише спростити взаємодію користувача з інформацією, а й забезпечити персоналізований підхід до пошуку та споживання контенту.

Метою кваліфікаційної роботи є розробка мікросервісного вебзастосунку, який інтегрує великі мовні моделі для впровадження персоналізованої рекомендаційної системи, семантичного пошуку та AI-взаємодії, для покращення користувацького досвіду.

Досягнення поставленої мети передбачає реалізацію таких завдань:

– дослідити особливості семантичного пошуку, векторних предмтавлень та RAG-підходу;
– спроєктувати мікросервісну архітектуру системи;
– розробити серверну та клієнтську частину вебзастосунку;
– реалізувати інтеграцію великих мовних моделей у межах звичайного асистента та із включенням RAG;
– реалізувати рекомендаційну систему та семантичний пошук;
– реалізувати OAuth 2.0 автентифікацію та механізми безпеки;
– протестувати основний функціонал системи.

Об’єктом дослідження є процес розробки мікросервісних вебзастосунків із впровадженням великих мовних моделей.

Предметом дослідження є методи та засоби реалізації рекомендаційної системи, семантичного пошуку та інтеграції великих мовних моделей у межах мікросервісного вебзастосунку.

Методами дослідження є методи об’єктно-орієнтованого проєктування, мікросервісної архітектури, семантичного пошуку, інтеграції великих мовних моделей у вебзастосунки.

Для реалізації програмної частини системи було використано мову Java та фреймворк Spring Boot, що забезпечують побудову мікросервісної архітектури. Для інтеграції великих мовних моделей та роботи з embeddings використано Spring AI, Vertex AI Gemini та Vertex AI Embeddings. Як систему керування базами даних обрано PostgreSQL, а для реалізації семантичного пошуку використано розширення Pgvector. Для кешування даних і підвищення продуктивності системи використано Redis. Клієнтську частину реалізовано за допомогою Angular. Для контейнеризації та розгортання системи використано Docker та Docker Compose, а Nginx застосовано як reverse proxy та gateway-рівень системи.

Практичне значення отриманих результатів полягає у створенні вебзастосунку, що реалізує семантичний пошук і персоналізовані рекомендації на основі векторних представлень контенту, а також інтеграцію великих мовних моделей для AI-взаємодії з користувачем. Розроблені рішення можуть бути використані як основа для побудови сучасних інформаційних платформ та AI-орієнтованих вебзастосунків.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та переліку посилань. Загальний обсяг роботи становить 59 сторінок. У переліку посилань 19 використаних джерел.

У першому розділі проаналізовано предметну область, існуючі аналоги та вимоги до системи. У другому розділі наведено проєктування системи, побудовано UML-діаграми та описано архітектуру мікросервісного застосунку. У третьому розділі описано реалізацію програмної частини системи, наведено результати роботи вебзастосунку, а також виконано функціональне та навантажувальне тестування основного функціоналу системи.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ВИМОГ ДО СИСТЕМИ

1.1 Опис програмних засобів

У цьому підрозділі наведено характеристику обраних інструментів та технологій, з описом їхніх переваг та нюансів використання для розроблюваної системи.

Завдання полягає у створенні мікросервісного вебзастосунку з інтеграцією рекомендаційної системи та великих мовних моделей. Для реалізації поставленої мети було обрано та згруповано за функціональним призначенням основні програмні засоби:

- Spring Boot, Spring AI – серверна частина та бізнес логіка;
- PostgreSQL, Pgvector – зберігання та обробка даних;
- Redis – кешування даних та оптимізація продуктивності системи;
- Vertex AI Gemini, Vertex AI Embeddings – інтеграція великих мовних моделей та створення векторних представлень для даних;
- Angular – клієнтська частина;
- Nginx – reverse proxy та маршрутизація запитів між компонентами системи;
- Docker, Docker Compose – контейнеризація та розгортання;
- Selenium – автоматизація UI-тестування та перевірка основного функціоналу системи;
- Apache JMeter – навантажувальне тестування та перевірка продуктивності системи.

Наведемо узагальнену характеристику програмних засобів та їхній вплив на різні фази проєкту.

Spring Boot – фреймворк з відкритим кодом для розробки production-ready, stand-alone, та production-grade Java застосунків. Він спрощує процес розробки шляхом автоматизованої конфігурації та стандартизації архітектурних підходів, що зменшує обсяг ручного налаштування і

пришвидшує створення масштабованих та підтримуваних систем. Далі наведено ключові критерії використання для цієї роботи [1, 2].

Мікросервісна підтримка – сприяє розробці незалежних сервісів, які можна розгортати, масштабувати та підтримувати. Кожен сервіс упаковується у JAR, що спрощує розгортання у Docker-контейнері [1].

Інтеграція із Spring екосистемою забезпечує тісну взаємодію з іншими проєктами екосистеми Spring, зокрема Spring Data, Spring Security, Spring Cloud тощо. Це забезпечує цілісний та послідовний підхід до розробки програмних продуктів. Зокрема Spring AI забезпечує стандартизований підхід до інтеграції великих мовних моделей у Java застосунки [3].

PostgreSQL – забезпечує високу продуктивність та масштабованість та є повністю відкритою та сумісною із широкою вибіркою мов програмування, включно з Java, що робить її непоганим вибором для інтеграції із Spring Boot. А поєднання PostgreSQL та Pgvector дозволяє реалізувати пошук за схожістю без використання окремої векторної бази даних та підтримує масштабування завдяки механізмам індексації [4, 5].

Redis використовується для кешування даних та оптимізації продуктивності системи. Його застосування дозволяє зменшити навантаження на базу даних під час виконання повторних запитів [6].

Vertex AI Gemini застосовується для інтеграції великих мовних моделей, тоді як Vertex AI Embeddings для створення векторних представлень на основі текстових або мультимодальних даних. Отриманні вектори належатимуть до єдиного семантичного простору та можуть бути використані для подальших задач, зокрема пошуку за схожістю та механізму рекомендацій на основі content-based підходу [7, 8].

Angular – фреймворк заснований на мові TypeScript з відкритим кодом, призначений для розробки SPA вебзастосунків. Він надає компонентно-орієнтовану архітектуру, вбудовані механізми маршрутизації інтеграцію з API, що робить його придатним для створення масштабованих і підтримуваних рішень для реалізації клієнтської частини застосунку [9].

Основними перевагами Angular є чітка архітектура, підтримка

повторного використання компонентів, використання TypeScript для підвищення надійності коду. Окрім цього наявність розвинутого інструментарію, таких як AngularCLI, RxJS та механізму Signals для реалізації реактивної взаємодії та управлінням станом застосунку [9, 10].

Nginx використовується як reverse proxy та gateway-рівень системи. Він забезпечує маршрутизацію запитів між клієнтом і сервером, обслуговування SPA-застосунку та централізовану точку доступу до мікросервісів [11].

Docker призначений для контейнеризації застосунку та його залежностей, що забезпечує однакову поведінку застосунку в різних середовищах виконання. Для оркестрації та спільного запуску декількох контейнерів у межах одного застосунку використовується Docker Compose, який дозволяє описувати конфігурацію сервісів, мереж та залежностей між ними. Це дозволяє спростити розгортання та підтримку мікросервісної архітектури застосунку [12, 13].

Selenium використовується для автоматизації UI-тестування вебзастосунку. Його застосування дозволяє перевіряти коректність роботи основних користувацьких сценаріїв, взаємодію з інтерфейсом та стабільність функціоналу системи в браузерному середовищі [14].

Apache JMeter використовується для проведення навантажувального тестування вебзастосунку. Його застосування дозволяє оцінити продуктивність, стабільність та здатність системи обробляти велику кількість одночасних запитів [15].

Обраний стек технологій відповідає вимогам масштабованості, підтримуваності та інтеграції компонентів штучного інтелекту в межах розроблюваної системи.

1.2 Огляд аналогічних рішень

Для виконання поставленого завдання необхідно визначити предметну область розроблюваної системи. У сучасному світі рекомендаційні системи

використовуються в переважній більшості користувацьких застосунків, наповнених контентом для будь-яких потреб. До цього списку можна віднести соціальні мережі, музикальні платформи, навчальні форуми тощо. В межах цієї роботи було обрано розробити систему для введення навчальних блогів з програмування.

До аналогів подібних платформ можна віднести такі сайти, як Medium, GeeksForGeeks, DEV Community тощо [16-18].

Medium – популярна платформа для публікації статей та блогів. Платформа орієнтована на створення якісного текстового контенту та має зручний інтерфейс для читання і написання матеріалів. Автори можуть публікувати статті на різні теми, включаючи програмування, технології, бізнес та освіту.

Система використовує механізми рекомендацій, які формують персоналізовану стрічку статей на основі інтересів користувача, його підписок та історії переглядів. Для пошуку матеріалів застосовуються ключові слова, теги та категорії, що дозволяє знаходити публікації за заданою тематикою.

До переваг цієї системи можна віднести такі аспекти:

– простий та зручний редактор для написання статей;
– рекомендаційна стрічка для користувачів;
– велика кількість авторів і читачів;
– підтримка тегів та тематичних підбірок.

До недоліків:

– значна частина статей доступна лише за платною підпискою;
– пошук здебільшого базується на ключових словах, а не на глибокому семантичному аналізі тексту;
– відсутні інструменти, які б допомагали автору автоматично генерувати або структурувати технічні матеріали.

GeeksforGeeks – спеціалізована освітня платформа для програмістів, яка містить велику кількість навчальних матеріалів, присвячених

алгоритмам, структурам даних, мовам програмування та підготовці до технічних співбесід.

Контент на платформі має чітку структуру та орієнтований на навчання. Статті часто містять теоретичні пояснення, приклади коду та практичні задачі. Пошук інформації на платформі здійснюється за допомогою ключових слів, категорій та тематичних розділів, що дозволяє швидко знаходити матеріали з конкретних технологій.

До переваг цієї системи можна віднести такі аспекти:

– велика база навчальних матеріалів;
– структуровані статті та приклади коду;
– можливість проходження курсів і практичних завдань;
– публікація статей проходить редакційний вибір.

До недоліків:

– пошук статей здебільшого працює через збіг ключових слів;
– значна частина додаткових навчальних матеріалів доступна лише у платних курсах;
– відсутні інструменти, які б допомагали автору швидко генерувати або покращувати технічні пояснення.

DEV Community – соціальна платформа для розробників, яка дозволяє програмістам публікувати статті, обговорювати технології та ділитися досвідом. Платформа орієнтована саме на IT-спільноту.

Статті на платформі публікуються у форматі технічних блогів і часто містять практичні приклади, навчальні матеріали та особистий досвід розробників. Пошук і рекомендації контенту на платформі здебільшого базуються на популярності статей, тегах та активності користувачів.

До переваг цієї системи можна віднести такі аспекти:

– повністю безкоштовна платформа;
– можливість швидкого публікування статей;
– активна спільнота розробників;
– зручна система тегів для класифікації матеріалів.

До недоліків:

– пошук статей базується переважно на ключових словах і тегах;
– рекомендації контенту не завжди є достатньо персоналізованими;
– відсутній інтелектуальний інструмент, який би допомагав авторам створювати технічні статті;
– різний рівень якості контенту через відкритість платформи.

Аналіз розглянутих систем показує, що більшість сучасних платформ для публікації технічних статей мають спільні недоліки – обмежені можливості семантичного пошуку та інтелектуального помічника для створення контенту.

У розроблюваній системі передбачається реалізація семантичного пошуку статей на основі векторних представлень тексту, що дозволить знаходити матеріали за змістовною схожістю, а не лише за збігом ключових слів. Крім того, система може включати інтелектуального агента-помічника, який допомагатиме автору створювати структуру статті, генерувати пояснення та приклади коду. Це дозволить значно спростити процес написання навчальних матеріалів і підвищити якість контенту.

1.3 Аналіз вимог

Перед початком проєктування майбутньої системи необхідно окреслити перелік функцій, які вона буде реалізовувати. Далі визначити характеристики її функціонування, від яких залежатиме рівень якості підсумкового програмного продукту. Відповідно, на цьому етапі доцільно сформувати вимоги, розглянувши їх у двох аспектах.

Функціональні вимоги – описують, які саме дії та функції повинна виконувати система для досягнення поставленої мети. Для системи введення блогів із програмування, можна винести такі функціональні вимоги:

– реєстрація нових користувачів;
– автентифікація користувачів, що вже існують;

– введення особистого блогу (публікація, перегляд, редагування, видалення власних статей та їхніх структурних елементів);
– отримання всіх статей;
– пошук статей за семантикою та заголовком;
– введення власних підписок і вподобань;
– формування рекомендацій на основі вподобань користувача.

Нефункціональні вимоги – характеризують, наскільки якісно працює система, охоплюючи такі аспекти як продуктивність, безпека, масштабованість тощо.

З метою спрощення подальшого розширення системи доцільно її реалізувати у вигляді мікросервісної архітектури, де кожен мікросервіс виконує окрему відповідальність. Зокрема, обробка автентифікації та авторизації користувачів, керування контентом, інтеграція з великими мовними моделями та взаємодія із хмарним сховищем. Такий підхід сприяє підвищенню масштабованості, гнучкості та підтримуваності майбутньої системи.

Для безпечної та зручної автентифікації користувачів система має підтримувати протокол OAuth 2.0 з можливістю входу через зовнішні сервіси, таких як Google та GitHub. Це підвищує рівень безпеки та спрощує процеси реєстрації та входу, що позитивно впливає на користувацький досвід.

Формування рекомендаційного контенту має здійснюватися на основі порівняння векторів вподобань користувачів із загальною векторною базою даних. Такий підхід дозволяє враховувати семантичну близькість між користувацькими вподобаннями та наявним навчальним контентом, що забезпечуватиме більш точні рекомендації, порівняно з традиційними методами фільтрації.

Пошук статей у системі також повинен реалізовуватися шляхом порівняння векторного представлення пошукового запиту з векторною базою даних. Це дає змогу знаходити матеріали не лише за точним збігом ключових

слів, а й за змістовною подібністю, що значно підвищує якість результатів пошуку та зручність роботи користувачів із системою.

З метою підвищення продуктивності системи та оптимізації обробки повторних запитів доцільно використовувати Redis-кешування. Такий підхід дозволяє зменшити навантаження на базу даних і сервіси великих мовних моделей під час реалізації семантичного пошуку та рекомендаційного функціоналу.

Загалом, застосування мікросервісної архітектури, сучасних механізмів автентифікації, Redis-кешування та векторних методів пошуку й рекомендацій сприяє підвищенню продуктивності, масштабованості, гнучкості та якості взаємодії користувачів із майбутньою системою.

2 ПРОЄКТУВАННЯ СИСТЕМИ

Проєктування системи виконується поетапно з метою формалізації функціональних вимог, визначення ролей, встановлення меж відповідальності між внутрішніми модулями системи та зовнішніми сервісами. Основа побудована за мікросервісною архітектурою, де кожен сервіс має окрему роль. Такий підхід забезпечує масштабованість, ізоляцію процесів і спрощує подальший розвиток та підтримку системи.

2.1 Концептуальне проєктування системи

На концептуальному етапі проєктування визначаються основні актори та варіанти використання системи. Цей етап побудовано на поєднанні двох типів діаграм прецедентів та послідовностей. Діаграма прецедентів використана для фіксації ролей учасників і набору доступних сценаріїв, тоді як діаграми послідовностей є проміжним етапом між концепцією та логікою, що деталізують порядок виконання ключових процесів усередині системи. Це дозволяє сформувати цілісне бачення функціонування системи та взаємодії її основних компонентів.

Для поточної системи було виділені актори двох типів за природою походження «Люди» та «Сторонні системи».

Розглянемо акторів за типом «**Люди**».

Користувач системи – батьківський актор, що представляє базову взаємодію користувача із системою та має доступ до основного мінімального функціоналу, пов’язаного з переглядом та пошуком контенту. Для цього актора визначено два спеціалізовані наслідники – «Гість» та «Авторизований користувач».

Авторизований користувач – користувач, який пройшов процедуру автентифікації та отримав розширений доступ до функціоналу системи. Цей

актор, окрім базових можливостей перегляду контенту, може створювати, редагувати та зберігати статті, взаємодіяти з персональним профілем, отримувати персоналізовані рекомендації та використовувати інші функції, доступні лише після авторизації.

Гість – неавторизований користувач у системі, який успадковує базовий функціонал актора «Користувач системи» та додатково має можливість виконувати авторизацію та реєстрацію у системі.

Розглянемо акторів за типом «**Сторонні системи**».

Google Identity та *GitHub OAuth* – зовнішні сервіси автентифікації, що забезпечують можливість авторизації користувачів у системі за допомогою облікових записів Google та GitHub, а також передачу базових даних профілю користувача.

Google Cloud Storage – хмарний сервіс, який використовується для зберігання медіаресурсів системи.

Vertex AI Gemini – LLM-сервіс, що використовується для генерації текстового контенту, аналізу даних та взаємодії з інтелектуальними функціями системи.

Vertex AI Embeddings – LLM-сервіс, який використовується для генерації векторних представлень тексту, необхідних для семантичного пошуку, порівняння контенту та взаємодії з асистентом, що використовує RAG-підхід.

Прецеденти системи було згруповано відповідно до акторів та функціонального призначення.

Для актора «Користувач системи» визначено базовий набір сценаріїв взаємодії з системою:

– перегляд списку статей;
– перегляд окремої статті;
– перегляд профілю автора;
– отримання рекомендацій з популярних та нових статей;
– пошук статей;
– взаємодія з AI-advisor системи.

Базовим прецедентом пошуку є «Пошук статей», який розширюється спеціалізованими сценаріями «Пошук статей за назвою» та «Пошук статей за семантикою» через зв’язок extend, що дозволяє реалізувати декілька способів пошуку в межах єдиного функціонального процесу. Семантичний пошук використовує сервіс Vertex AI Embeddings для генерації векторних представлень запитів пошуку.

Прецедент «Переглянути профіль автора» включає отримання списку опублікованих статей та followers/following автора через зв’язок include, оскільки ці дії є невід’ємною складовою процесу перегляду профілю.

Взаємодія з AI-advisor включає надсилання повідомлень та отримання відповідей у межах поточної сесії браузера. Для реалізації інтелектуальної взаємодії використовується сервіс Vertex AI Gemini, а Vertex AI Embeddings забезпечує генерацію векторних представлень тексту для реалізації RAG-підходу та пошуку релевантного контексту.

Для актора «Гість» визначено такі сценарії:

– реєстрація у системі;
– вхід до системи.

Прецедент «Увійти в систему» розширюється сценаріями входу через логін і пароль, Google та GitHub через зв’язок extend, що забезпечує підтримку декількох способів автентифікації. Авторизація через зовнішні сервіси реалізується за допомогою Google Identity та GitHub OAuth.

Для актора «Авторизований користувач» визначено такі сценарії:

– створення або оновлення статті;
– видалення статті;
– оновлення профілю;
– перегляд власного профілю
– керування підписками на авторів;
– отримання персоналізованих рекомендацій;
– взаємодія з chat-assistant;
– керування реакціями на контент.

Прецедент «Переглянути свій профіль» є розширенням базового сценарію «Переглянути профіль автора», доступного для актора «Користувач системи». Даний сценарій додатково включає отримання чернеток непублічних статей та списку вподобань через зв’язки include.

Прецедент «Оновити аватар профілю» розширює сценарій «Оновити власний профіль» через зв’язок extend, оскільки зміна аватара є додатковою дією під час редагування профілю.

Прецедент «Отримати персоналізовані рекомендації» забезпечує формування рекомендацій на основі вподобаних користувачем статей та їх семантичної схожості з іншим контентом системи, доповнюючи результати новими та популярними публікаціями.

Прецедент «Створити або оновити статтю» включає сценарій «Керування зображеннями у статті» через зв’язок include, оскільки робота з медіаресурсами є складовою процесу створення контенту. Для зберігання зображень використовується сервіс Google Cloud Storage.

Під час створення або оновлення статті використовується сервіс Vertex AI Gemini для автоматичного формування короткого опису та визначення основного змісту статті. На основі згенерованого опису та метаданих статті сервіс Vertex AI Embeddings формує векторне представлення контенту, яке використовується для реалізації семантичного пошуку та системи рекомендацій.

На основі опису акторів, прецедентів та їх сценаріїв сформовано діаграму прецедентів (див. рис. 2.1).

Після побудови діаграми прецедентів доцільним є перехід до детальнішого опису взаємодії компонентів системи в межах окремих сценаріїв. Якщо діаграма прецедентів відображає концептуальне бачення функціональності системи та ролей її учасників, то діаграми послідовностей дозволяють деталізувати порядок виконання операцій, обмін повідомленнями між компонентами та внутрішню логіку реалізації прецедентів. Таким чином, діаграми послідовностей виступають проміжним етапом між концептуальним та логічним проєктуванням системи.

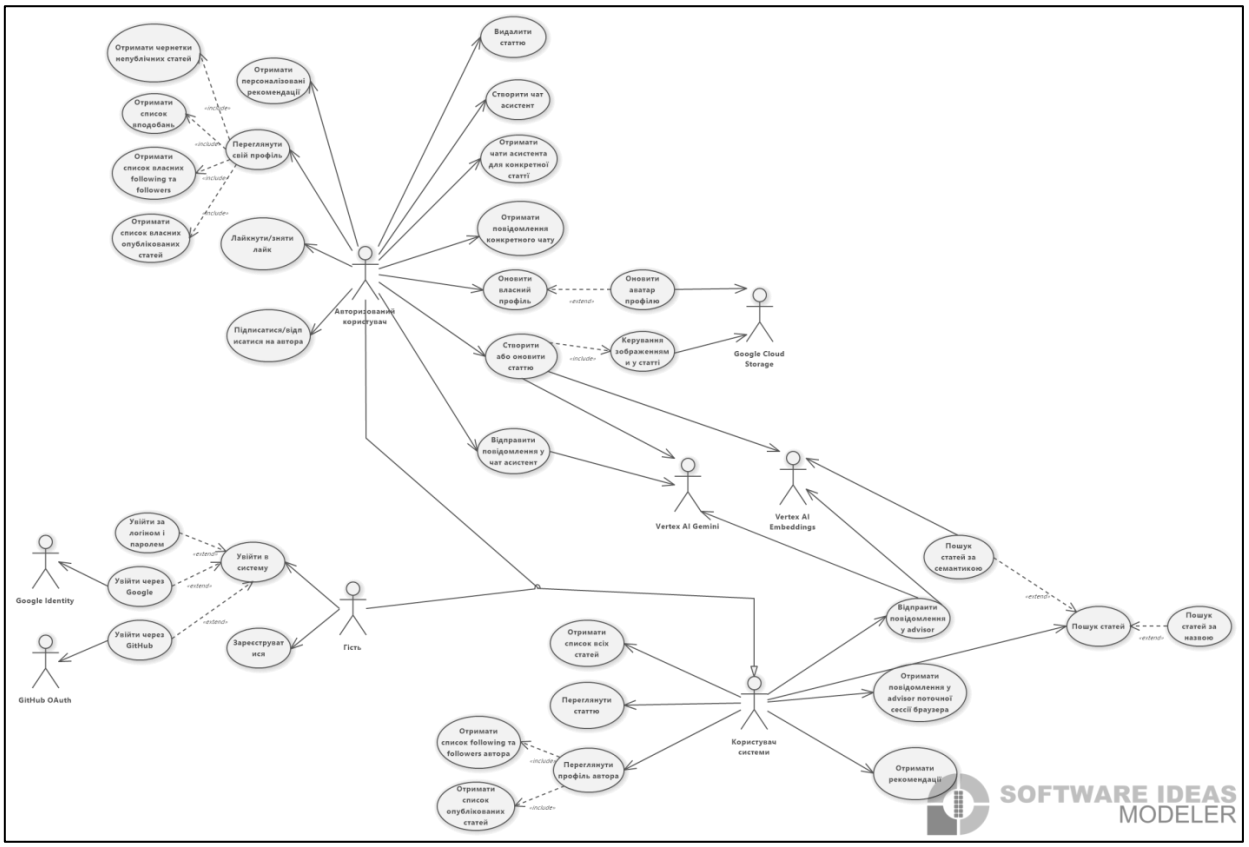


Рисунок 2.1 – Діаграма прецедентів

Для поточної системи було обрано два ключові прецеденти, що є основними:

– «Відправити повідомлення у advisor»;

– «Отримати персоналізовані рекомендації».

Далі наведено діаграму послідовностей для прецеденту «Відправити повідомлення у advisor». Даний сценарій починається з введення користувачем повідомлення у віджет advisor та надсилання запиту до сервера. Після отримання запиту система перевіряє наявність conversationId. У випадку відсутності активної сесії формується новий ідентифікатор діалогу, інакше використовується вже існуючий контекст розмови (див. рис. 2.2).

На наступному етапі AdvisorChatService завантажує попередню історію повідомлень із ChatMemory для формування контексту поточної взаємодії. Після цього текст запиту передається до сервісу Vertex AI Embeddings для генерації векторного представлення повідомлення користувача. Отриманий

embedding використовується для виконання similarity search у VectorStore та отримання набору релевантних документів, які формують контекст для RAG-взаємодії. Далі сформований контекст, релевантні документи та повідомлення користувача передаються до сервісу Vertex AI Gemini, який генерує текстову відповідь. Після отримання відповіді система зберігає повідомлення користувача у ChatMemory та AI-assistant повертає результат для відображення користувачу.

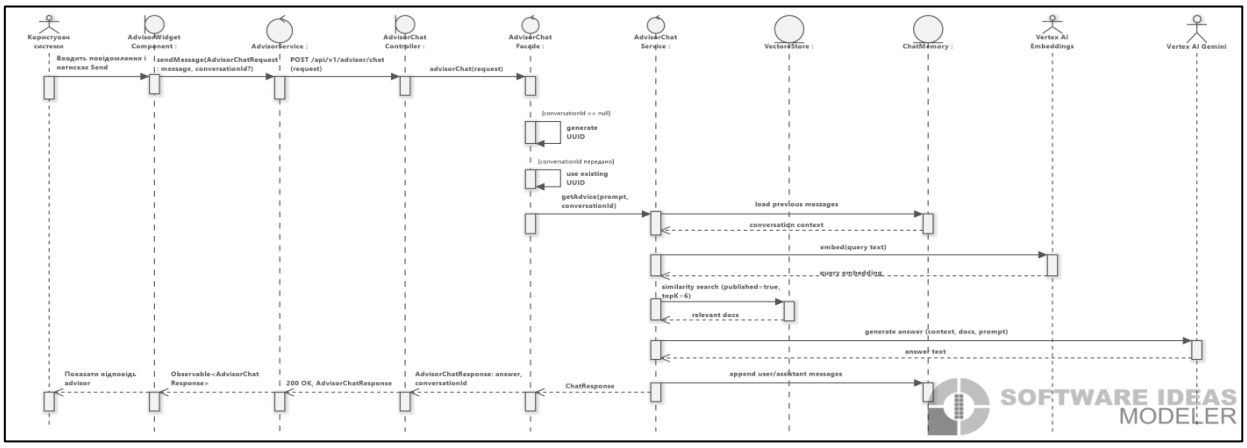


Рисунок 2.2 – Діаграма послідовностей для прецеденту «Відправити повідомлення у advisor»

Наступна діаграма відноситься до прецеденту «Отримати персоналізовані рекомендації». Даний сценарій реалізує гібридний механізм формування рекомендацій на основі вподобаних користувачем статей, семантичної схожості контенту та fallback-стрічки з популярних і нових публікацій (див. рис. 2.3). Даний сценарій реалізує гібридний механізм формування рекомендацій на основі вподобаних користувачем статей, семантичної схожості контенту та fallback-стрічки з популярних і нових публікацій. Після відкриття головної сторінки надсилається запит до ArticleReccomendationController для отримання персоналізованого набору статей. На початковому етапі система завантажує список останніх опублікованих статей. Для авторизованого користувача додатково формується множина виключень, до якої входять власні статті користувача та вже вподобані матеріали, а також отримується список останніх вподобань.

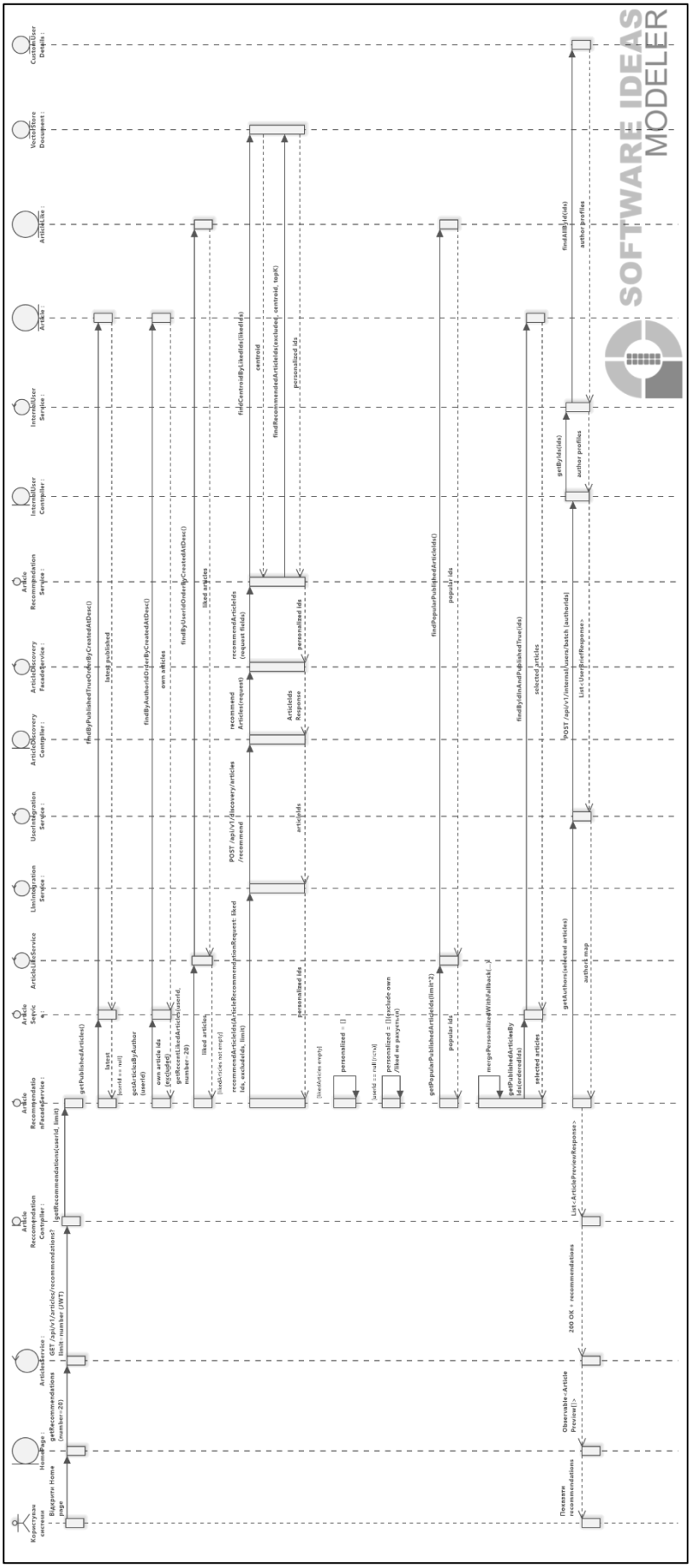


Рисунок 2.3 – Діаграма послідовностей для прецеденту «Отримати персоналізовані рекомендації»

Якщо список вподобаних статей не є порожнім, формується запит, який передається до ArticleDiscoveryController. Далі обчислюється центроїд векторних представлень вподобаних статей і виконується пошук семантично близького контенту у VectorStore, у результаті чого формується персоналізований список ідентифікаторів статей.

На наступному кроці система формує fallback-набір як змішаний список популярних і нових статей. Після цього виконується об’єднання персоналізованого та fallback-наборів відповідно до фіксованої логіки змішування: пріоритет надається персоналізованим результатам, проте fallback-документи регулярно додаються для забезпечення стабільності та різноманіття рекомендацій. У випадку відсутності персоналізованих результатів використовується виключно fallback-набір.

На завершальному етапі система завантажує деталі відібраних статей, отримує профілі їх авторів та повертає сформований список рекомендацій для подальшого відображення користувачу.

Таким чином, на етапі концептуального проєктування було визначено основних акторів системи, сформовано набір прецедентів та побудовано діаграми, що описують як загальні сценарії взаємодії користувачів із системою, так і внутрішню послідовність виконання ключових процесів. Отримані результати формують основу для подальшого логічного та архітектурного проєктування системи.

2.2 Логічна архітектура проєкту

На етапі логічного проєктування концептуальне представлення системи деталізується через визначення її внутрішньої структури, основних сутностей, зв’язків між ними та програмних компонентів. Для цього було сформовано ER-модель бази даних і діаграми класів окремих мікросервісів, що відображають логіку зберігання даних, взаємодію між модулями та розподіл відповідальності в межах мікросервісної архітектури.

Логічна структура даних системи представлена у вигляді ER-моделі. Вона відображає логічну структуру зберігання даних системи, включаючи сутності контенту, користувачів, AI-взаємодії та векторних представлень, а також взаємозв’язки між ними в межах мікросервісної архітектури (див. рис. 2.4).

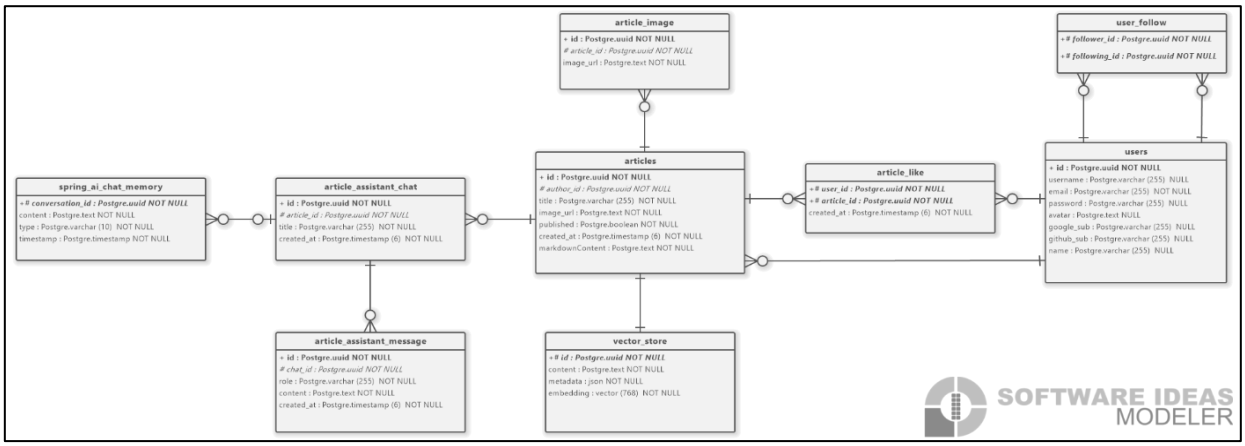


Рисунок 2.4 – ER-модель

Центральною сутністю ER-моделі є articles, яка зберігає основну інформацію про статтю: ідентифікатор, автора, назву, основне зображення, статус публікації, дату створення та markdown-вміст. Саме ця сутність використовується як базова для більшості функціональних сценаріїв системи, зокрема створення та редагування статей, перегляду контенту, формування рекомендацій і семантичного пошуку.

Сутність users містить дані користувачів системи, зокрема електронну пошту, ім’я, аватар та ідентифікатори, пов’язані з авторизацією через Google і GitHub. Зв’язок між сутностями users та articles відображає авторство статей: один користувач може створити багато статей, тоді як кожна стаття належить одному автору. Додатково взаємодія користувачів із контентом реалізується через сутність article_like, яка зберігає інформацію про вподобання статей, включаючи користувача, статтю та дату створення реакції. Дані цієї сутності використовуються як для відображення реакцій на контент, так і для формування персоналізованих рекомендацій на основі семантичної схожості вподобаних матеріалів.

Для реалізації соціальної взаємодії між користувачами передбачено сутність user_follow, яка зберігає зв’язки підписки між користувачами. Поля follower_id та following_id посилаються на сутність users і дозволяють визначати, хто на кого підписаний. Така структура забезпечує можливість формування списків followers/following у профілях користувачів.

Для роботи з медіаресурсами статей використовується сутність article_image, яка зберігає посилання на зображення, пов’язані з конкретною статтею. Такий підхід дозволяє відокремити текстовий вміст статті від медіаданих, а самі файли зберігати у зовнішньому хмарному сховищі.

Окрему роль у моделі відіграє сутність vector_store, яка містить текстовий контент, метадані та векторне представлення статей. Вона використовується для реалізації семантичного пошуку, персоналізованих рекомендацій і RAG-підходу в роботі AI-assistant. Зв’язок із сутністю articles дозволяє співвіднести векторні представлення з конкретними публікаціями системи.

Для реалізації взаємодії користувача з AI-assistant в системі передбачено сутності article_assistant_chat, article_assistant_message та spring_ai_chat_memory. Сутність article_assistant_chat зберігає інформацію про чат, пов’язаний із конкретною статтею, тоді як article_assistant_message містить окремі повідомлення в межах цього чату, зокрема роль відправника, текст повідомлення та дату створення. Сутність spring_ai_chat_memory використовується для збереження історії повідомлень і контексту діалогу в межах окремої conversation-сесії. Така структура забезпечує підтримку послідовної взаємодії користувача як зі статейним AI-assistant, так і з advisor-компонентом системи, що використовує великі мовні моделі та RAG-підхід.

Таким чином, ER-модель відображає логічну структуру даних системи та поєднує сутності, пов’язані з контентом, AI-взаємодією та векторними представленнями. Наявність окремих структур для вподобань, AI-діалогів і vector store забезпечує основу для реалізації рекомендаційної системи, семантичного пошуку та інтеграції великих мовних моделей.

Після аналізу логічної структури даних доцільним є перехід до розгляду внутрішньої структури окремих мікросервісів системи. Серверну реалізацію було поділено на чотири основні частини:

– авторизаційний сервіс – забезпечує реєстрацію та автентифікацію користувачів;
– сервіс статей – відповідає за створення, редагування, зберігання та обробку контенту;
– LLM-сервіс – реалізує інтеграцію з великими мовними моделями, семантичній пошук та AI-взаємодію;
– сервіс зберігання файлів – забезпечує роботу з медіа ресурсами через хмарне сховище.

Першим буде розглянуто мікросервіс статей. Даний мікросервіс є центральною частиною системи та відповідає за управління контентом платформи. У його межах реалізовано створення, редагування, отримання та видалення статей, роботу з реакціями користувачів, медіаресурсами, персоналізованими рекомендаціями та семантичним пошуком (див. рис. 2.5).

Основу мікросервісу формують сутності Article, ArticleLike та ArticleImage, які відповідають за зберігання статей, реакцій користувачів та медіаресурсів. Для доступу до даних використовуються інтерфейси репозиторіїв, побудовані на основі Spring Data JPA, тоді як бізнес-логіка інкапсулюється в сервіси та фасади.

Базову CRUD-логіку роботи зі статтями реалізують ArticleController, ArticleFacadeService та ArticleService. Дана група класів відповідає за створення, редагування, отримання та видалення статей, перевірку доступу до контенту.

Важливу роль у реалізації AI-функціоналу виконує LlmIntegrationService, який забезпечує інтеграцію мікросервісу статей із LLM-сервісом. Саме тут реалізується взаємодія із семантичним пошуком та персоналізованими рекомендаціями.

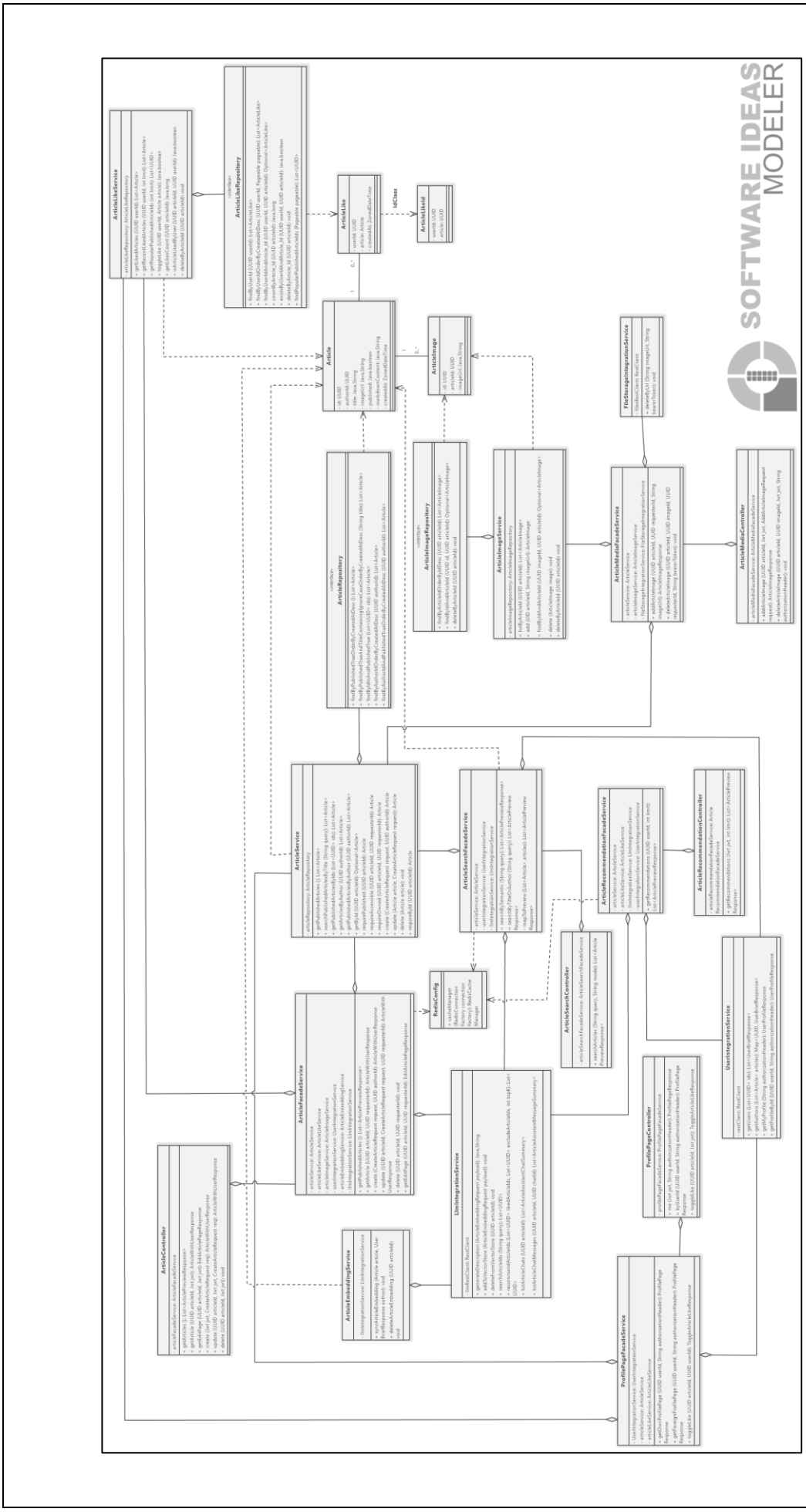


Рисунок 2.5 – Діаграма класів для мікросервісу статей

Ключовим компонентом мікросервісу є ArticleRecommendationFacadeService, який реалізує логіку формування рекомендацій. Саме через цей клас формується гібридний механізм рекомендацій, що поєднує аналіз вподобаних статей користувача, семантичну схожість контенту та fallback-підбір популярних і нових матеріалів.

Для підтримки пошуку використовується ArticleSearchFacadeService, який відповідає за підготовку пошукових запитів та формування фінальної видачі релевантних статей. У межах системи реалізовано два режими пошуку: класичний пошук за заголовками статей та семантичний пошук на основі векторної схожості контенту.

Окрему роль у мікросервісі виконує ArticleEmbeddingService, який відповідає за синхронізацію статей із vector store та оновлення embeddings після створення або зміни контенту. Це забезпечує актуальність семантичного індексу системи та коректну роботу рекомендаційного механізму та RAG-підходу.

Для роботи з медіаресурсами використовується ArticleMediaFacadeService, який відповідає за додавання та видалення зображень статей. Взаємодія із зовнішнім файловим сервісом реалізується через FileStorageIntegrationService.

UserIntegrationService відповідає за взаємодію з мікросервісом авторизації, який забезпечує зберігання та отримання даних користувачів системи.

Конфігурацію Redis-кешування реалізує компонент RedisConfig, який визначає правила кешування для списків статей, семантичного та за заголовками пошуку, рекомендацій. Це дозволяє зменшити навантаження на базу даних і LLM-сервіс під час повторних запитів.

Таким чином, діаграма класів мікросервісу статей поєднує логіку роботи з контентом, семантичний пошук, рекомендації та інтеграцію з іншими мікросервісами.

Після розгляду мікросервісу статей доцільним є перехід до LLM-сервісу, який забезпечує реалізацію AI-функціоналу системи (див. рис. 2.6).

Даний мікросервіс відповідає за інтеграцію з великими мовними моделями, роботу з embeddings, vector store, реалізацію семантичного пошуку, генерацію рекомендацій та підтримку AI-assistant функціоналу за допомогою Spring AI. На відміну від мікросервісу статей, який виконує роль оркестрації та формує фінальну видачу контенту, LLM-сервіс інкапсулює AI-логіку та операції з векторними представленнями.

Основу сервісу формують компоненти ArticleVectorService, ArticleRecommendationService, AdvisorService та AssistantService, які реалізують окремі AI-сценарії системи.

Клас ArticleVectorService відповідає за роботу з vector store, зокрема додавання статей до векторного сховища, видалення embeddings та виконання similarity search. Саме через цей компонент реалізується пошук релевантного контенту на основі векторної схожості тексту.

Для реалізації рекомендаційної системи використовується ArticleRecommendationService, який виконує пошук релевантних статей на основі embeddings вподобаного контенту. Даний клас обчислює центроїд векторних представлень вподобаних статей та виконує пошук найближчих сусідів у vector store. Результатом роботи сервісу є список ідентифікаторів релевантних статей, який передається назад до мікросервісу статей для подальшого формування фінальної рекомендаційної видачі.

Для реалізації рекомендаційної системи використовується ArticleRecommendationService, який виконує пошук релевантних статей на основі embeddings вподобаного контенту. Даний клас обчислює центроїд векторних представлень вподобаних статей та виконує пошук найближчих сусідів у vector store.

Результатом роботи сервісу є список ідентифікаторів релевантних статей, який передається назад до мікросервісу статей для подальшого формування фінальної рекомендаційної видачі.

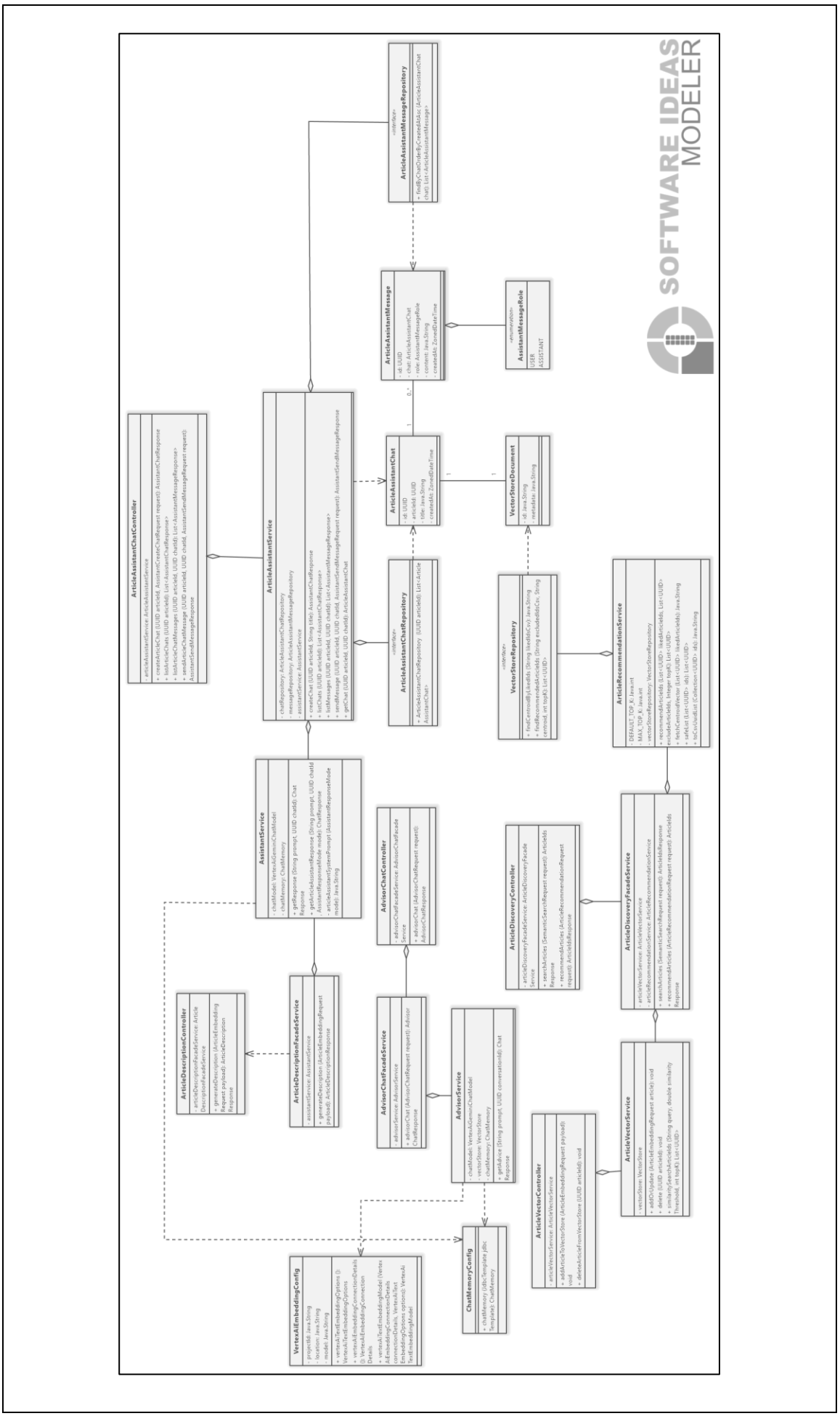


Рисунок 2.6 – Діаграма класів для LLM-мікросервісу

ArticleDiscoveryFacadeService відповідає за координацію сценаріїв семантичного пошуку та формування рекомендацій, об’єднуючи взаємодію між компонентами роботи з embeddings і vector store. Через нього реалізується обробка запитів від мікросервісу статей для пошуку та отримання рекомендацій.

Окрему роль у системі виконує AdvisorService, який забезпечує роботу AI-advisor. Даний клас взаємодіє з chatModel, vectorStore та chatMemory для реалізації RAG-підходу під час генерації відповідей.

Для підтримки AI-assistant у контексті окремих статей використовується AssistantService та пов’язані класи ArticleAssistantService, ArticleAssistantChat і ArticleAssistantMessage. Така структура дозволяє організувати окремі AI-чати для конкретних статей, зберігати історію повідомлень та підтримувати контекстну взаємодію користувача з AI-assistant.

ArticleDescriptionFacadeService використовується для автоматичної генерації короткого опису статті, який надалі застосовується під час формування embeddings та семантичного індексування контенту.

Конфігураційні класи VertexAiEmbeddingConfig та ChatMemoryConfig відповідають за налаштування інтеграції з Vertex AI Embeddings та механізмом збереження історії повідомлень.

Таким чином, діаграма класів LLM-сервісу демонструє відокремлення AI-логіки в окремий мікросервіс для підтримки LLM, рекомендацій, семантики та RAG-функціоналу.

Після розгляду LLM-сервісу доцільно перейти до авторизаційного сервісу, оскільки саме він забезпечує роботу з користувачами, профілями та механізмами автентифікації (див. рис. 2.7).

Авторизаційний сервіс відповідає за реєстрацію користувачів, вхід до системи, обробку OAuth 2.0 автентифікації через Google та GitHub, а також керування профілем користувача. Центральною сутністю сервісу є CustomUserDetails, яка містить основні дані користувача та використовується в механізмах безпеки системи.

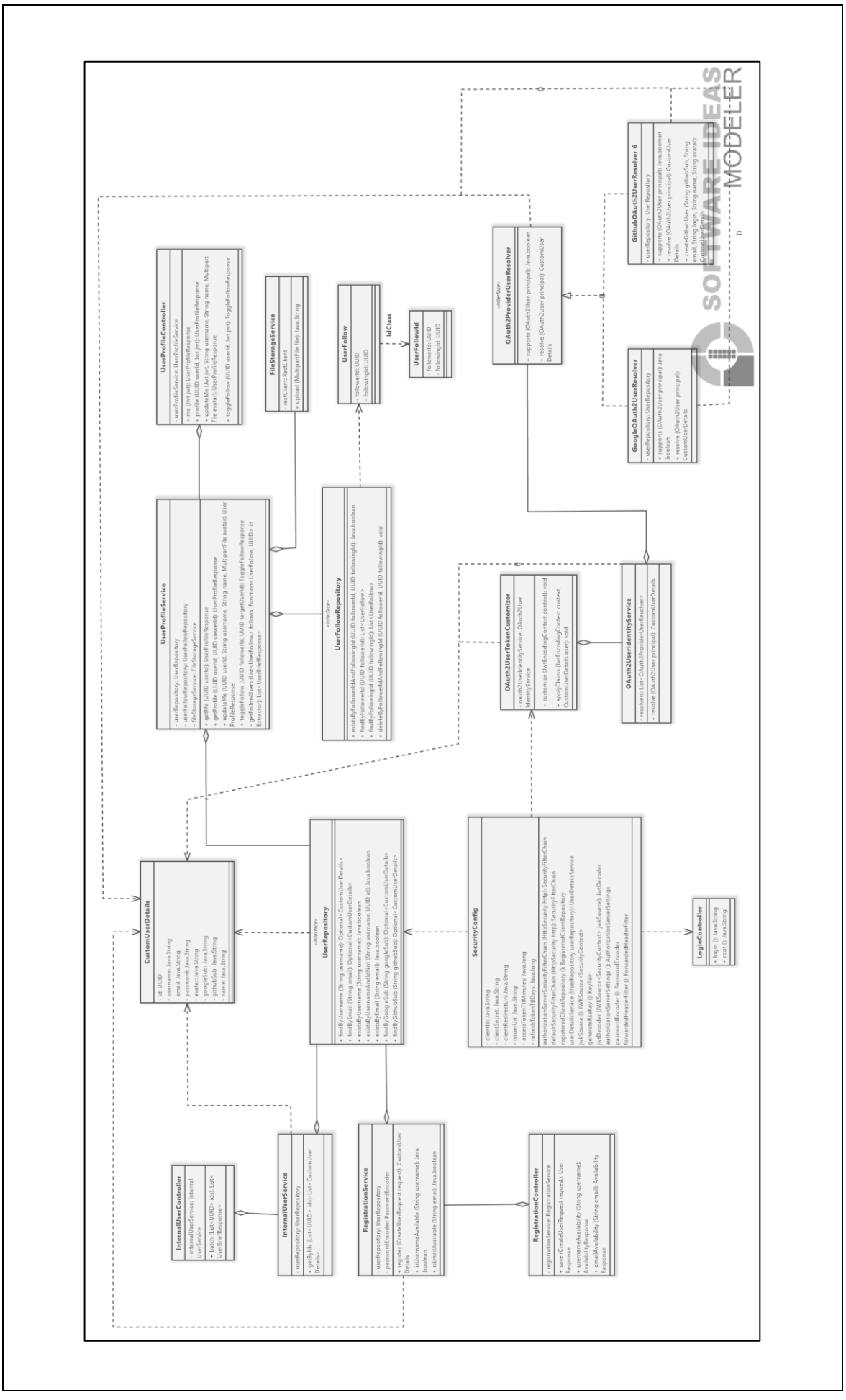


Рисунок 2.7 – Діаграма класів для мікросервісу авторизації

Базову логіку реєстрації реалізують RegistrationController та RegistrationService. Вони відповідають за створення нового користувача, перевірку доступності username та email, а також збереження даних у UserRepository.

Окрему роль виконує SecurityConfig, який налаштовує правила доступу, JWT-автентифікацію, OAuth 2.0 вхід, механізм виходу та інтеграцію з Spring Security. Для підтримки входу через зовнішні облікові записи використовуються OAuth2UserIdentityService, GoogleOAuth2UserResolver.

Функціонал профілю користувача реалізовано через UserProfileController та UserProfileService. Вони забезпечують отримання профілю, оновлення персональних даних, зміну аватара та підписку на інших користувачів.

Також сервіс містить InternalUserController та InternalUserService, які використовуються іншими мікросервісами для отримання базових даних користувачів, наприклад під час формування списку статей або рекомендацій.

Таким чином, авторизаційний сервіс поєднує механізми автентифікації, керування профілем та надання інформації про користувачів іншим мікросервісам системи.

Останнім у межах логічного проєктування буде розглянуто сервіс зберігання файлів, який відповідає за роботу з медіаресурсами системи. Файловий сервіс має компактну структуру та виконує окрему відповідальності завантаження й видалення файлів у хмарному сховищі Google Cloud Storage (див. рис. 2.8).

Основним компонентом сервісу є GoogleStorageService, який інкапсулює логіку взаємодії з Google Cloud Storage через об’єкт Storage та налаштований bucket.

Взаємодія із зовнішніми мікросервісами реалізується через GoogleStorageController, який надає шляхи для завантаження та видалення файлів. Після успішного завантаження сервіс повертає об’єкт FileResponse, що містить URL збереженого ресурсу.

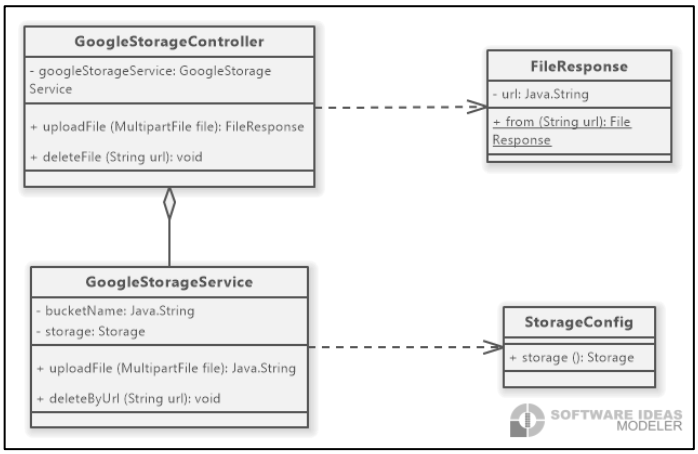


Рисунок 2.8 – Діаграма класів для мікросервісу зберігання файлів

Для конфігурації підключення до Google Cloud Storage використовується конфігурація StorageConfig, який відповідає за створення та налаштування Storage-клієнта. Такий підхід забезпечує незалежність файлового сервісу та ізоляцію логіки роботи з медіаресурсами.

Таким чином, на етапі логічного проєктування було сформовано структуру основних мікросервісів системи, визначено їх відповідальність, внутрішні компоненти та взаємодію між ними. Побудовані ER-модель і діаграми класів дозволили деталізувати логіку роботи з контентом, користувачами, AI-функціоналом і медіаресурсами, а також відобразити особливості реалізації рекомендаційної системи, семантичного пошуку та інтеграції великих мовних моделей у межах мікросервісної архітектури.

2.3 Проєктування фізичної моделі

На етапі фізичного проєктування визначається спосіб взаємодії програмних компонентів системи, особливості їх розгортання, мережевої взаємодії та використання зовнішньої інфраструктури. Для цього було побудовано діаграму компонентів та діаграму розгортання, які відображають структуру взаємодії мікросервісів, розподіл відповідальності між компонентами та середовище виконання системи.

Діаграма компонентів системи відображає загальну структуру взаємодії між клієнтською частиною, gateway рівнем, сервером, сховищами даних та зовнішніми сервісами (див. рис. 2.9).

Архітектура системи побудована за багаторівневим принципом і складається з таких основних рівнів:

– Client Layer;
– Gateway Layer;
– Application Layer;
– Data Layer;
– External Services Layer.

Рівень Client Layer представлений браузером користувача та клієнтським застосунком, реалізованим на Angular. Клієнт відповідає за взаємодію з користувачем, формування API-запитів та відображення отриманих даних.

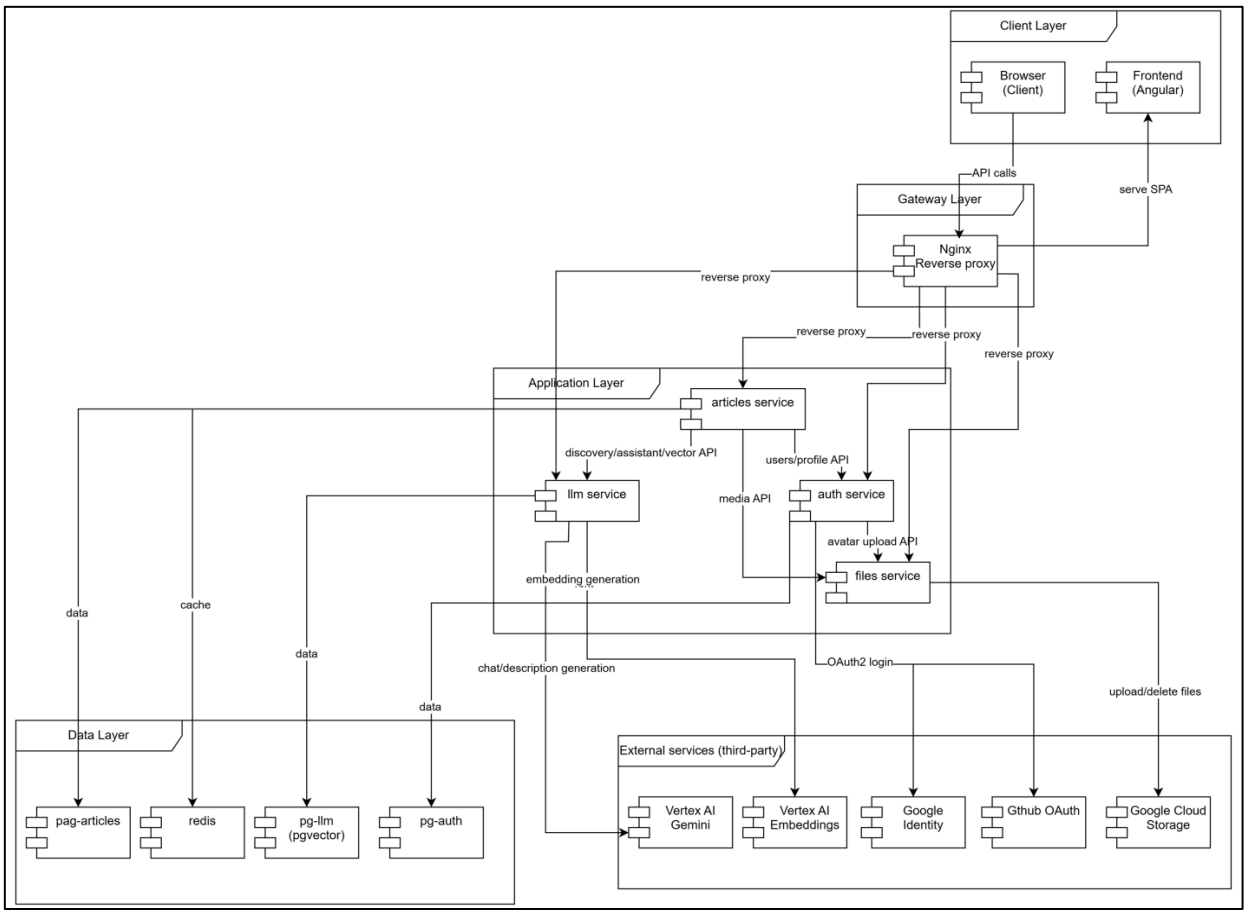


Рисунок 2.9 – Діаграма компонентів

Для маршрутизації зовнішніх запитів використовується Gateway Layer, представлений Nginx reverse proxy. Даний компонент виконує роль єдиної точки входу до системи, забезпечує обслуговування SPA-застосунку та перенаправлення API-запитів до відповідних мікросервісів.

Основна бізнес-логіка системи реалізована в Application Layer, який складається з чотирьох мікросервісів. Їх внутрішня структура, відповідальність та взаємодія були детально розглянуті на етапі логічного проєктування системи.

Рівень Data Layer містить окремі сховища даних для кожного мікросервісу. Для сервісу статей використовується база даних pg-articles, для авторизаційного сервісу – pg-auth, а для LLM-сервісу – pg-llm із підтримкою pgvector для роботи з embeddings та similarity search. Для кешування даних використовується Redis.

Окрему роль у системі відіграє рівень External Services, який містить сторонні сервіси, що забезпечують AI-функціонал, OAuth 2.0 автентифікацію та роботу з хмарним сховищем.

Таким чином, діаграма компонентів демонструє багаторівневу мікросервісну архітектуру системи, розподіл відповідальності між сервісами та інтеграцію різних компонентів у межах вебзастосунку.

Діаграма розгортання системи відображає фізичне середовище виконання компонентів системи, їх контейнеризацію, мережеву взаємодію та використання зовнішніх сервісів (див. рис. 2.10).

Розгортання системи реалізовано на основі Docker-контейнерів, об’єднаних у спільну мережу ai-articles-network. Усі серверні компоненти, клієнтська частина, бази даних та Redis-кеш функціонують як окремі ізольовані контейнери в межах одного Docker Host.

Клієнтський застосунок розгортається у контейнері frontend-app, тоді як роль reverse proxy та gateway виконує контейнер nginx-proxy, через який проходять усі зовнішні API-запити.

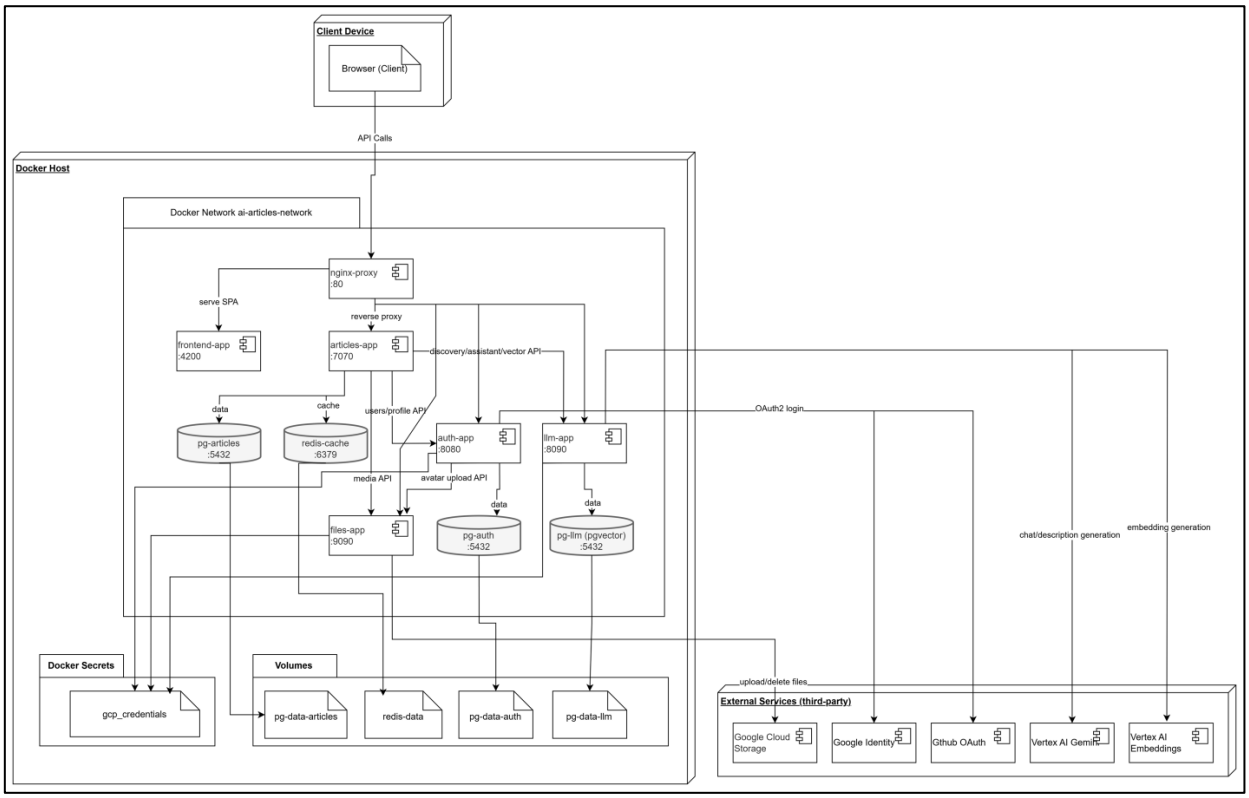


Рисунок 2.10 – Діаграма розгортання

Мікросервіси системи розгортаються як окремі Docker-контейнери та використовують ізольовані сховища даних PostgreSQL. Для LLM-сервісу додатково використовується pgvector-розширення, необхідне для роботи з embeddings і semantic search, тоді як мікросервіс статей використовує Redis-контейнер для кешування даних. Для збереження даних між перезапусками контейнерів використовуються Docker Volumes.

Окремо виділено Docker Secrets, які використовуються для зберігання конфіденційних даних, зокрема credentials для інтеграції з Google Cloud Platform. Система також взаємодіє із зовнішніми сервісами, які використовуються через API-взаємодію та не розгортаються в межах Docker Host.

Отже, діаграма розгортання демонструє фізичну структуру системи, ізоляцію компонентів через контейнеризацію.

Таким чином, етап фізичного проєктування дозволив визначити структуру розгортання системи, взаємодію її компонентів та особливості реалізації мікросервісної архітектури в контейнеризованому середовищі.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Програмна реалізація

Після завершення етапу проєктування було реалізовано програмну частину системи. У даному пункті наведено ключові фрагменти коду, що відображають основні механізми роботи системи.

Для реалізації семантичного пошуку, рекомендаційного функціоналу, RAG-підходу система використовує embeddings, що формуються за допомогою Vertex AI Embeddings. Для генерації векторних представлень було використано модель text-embedding-004. Конфігурація підключення до моделі та параметрів embeddings реалізована через окремий конфігураційний клас та функції всередині нього (див. рис. 3.1).

```
@Bean
  public VertexAiTextEmbeddingOptions vertexAiTextEmbeddingOptions() {
    return VertexAiTextEmbeddingOptions.builder().model(model).build();
  }

  @Bean
  public VertexAiEmbeddingConnectionDetails vertexAiEmbeddingConnectionDetails() {
    return VertexAiEmbeddingConnectionDetails.builder()
        .projectId(projectId)
        .location(location)
        .build();
  }
```

Рисунок 3.1 – Налаштування Vertex AI Embeddings

Після створення або оновлення статті виконується синхронізація її embeddings із vector store (див. рис. 3.2).

```
Article article = articleService.create(request, authorId);
Map<UUID, UserBriefResponse> authors =
userIntegrationService.getAuthors(List.of(article));
UserBriefResponse author = authors.get(article.getAuthorId());
articleEmbeddingService.syncArticleEmbedding(article, author);
```

Рисунок 3.2 – Синхронізація ембедінга після створення/оновлення статті

Текст статті разом із метаданими формується у документ, який надалі векторизується та зберігається у pgvector (див. рис. 3.3).

```
String content = String.format(
        "Title: %s\nAuthor: %s\nDescription: %s\nImage: %s",
        article.title(), article.author(), article.content(), article.image()
);

Document document = new Document(
        id,
        content,
        Map.of("article_id", id, "published", article.published())
);

vectorStore.add(List.of(document));
```

Рисунок 3.3 – Формування векторного документу

Для реалізації AI-advisor використовується RAG-підхід, який поєднує історію діалогу та релевантний контекст із vector store перед генерацією відповіді великою мовною моделлю.

У наведеному фрагменті AdvisorService комбінує ChatMemory та VectorStore, що дозволяє моделі генерувати відповіді з урахуванням попереднього контексту діалогу та релевантних документів (див. рис. 3.4).

```
public ChatResponse getAdvice(String prompt, UUID conversationId) {
   return ChatClient.builder(chatModel)
         .defaultAdvisors(
               MessageChatMemoryAdvisor.builder(chatMemory).build(),
               QuestionAnswerAdvisor.builder(vectorStore).build()
         )
         .build()
         .prompt()
         .advisors(a -> a.param(ChatMemory.CONVERSATION_ID, conversationId))
         .user(prompt)
         .call()
         .chatResponse();
}
```

Рисунок 3.4 – Взаємодія з advisor

Основою персоналізованої рекомендаційної системи є порівняння embeddings вподобаних статей із векторною базою даних. Для цього обчислюється центроїд embeddings користувацьких вподобань (див. рис. 3.5).

```
@Query(value = """
      SELECT AVG(embedding)::text
      FROM vector_store
      WHERE ((metadata::jsonb)->>'article_id') IS NOT NULL
        AND ((metadata::jsonb)->>'article_id')::uuid = ANY (string_to_array(:likedIdsCsv,
',')::uuid[])
      """, nativeQuery = true)
String findCentroidByLikedIds(@Param("likedIdsCsv") String likedIdsCsv);
```

Рисунок 3.5 – Центроїд та отримання ID статей
для персоналізованих рекомендацій

Після чого виконується пошук найближчих статей у vector store (див. рис. 3.6).

```
@Query(value = """
      SELECT ((metadata::jsonb)->>'article_id')::uuid AS article_id
      FROM vector_store
      WHERE ((metadata::jsonb)->>'article_id') IS NOT NULL
        AND ((metadata::jsonb)->>'article_id')::uuid <> ALL (string_to_array(:excludedIdsCsv,
',')::uuid[])
      ORDER BY embedding <=> CAST(:centroid AS vector)
      LIMIT :topK
      """, nativeQuery = true)
List<UUID> findRecommendedArticleIds(
      @Param("excludedIdsCsv") String excludedIdsCsv,
      @Param("centroid") String centroid,
      @Param("topK") int topK
);
```

Рисунок 3.6 – Пошук найближчих статей до центроїда

Такий підхід дозволяє формувати рекомендації на основі семантичної близькості контенту, а не лише точного збігу категорій або ключових слів.

Для оптимізації продуктивності системи та зменшення навантаження на базу даних і LLM-сервіс використовується Redis-кешування (див. рис. 3.7).

```
return RedisCacheManager.builder(connectionFactory)
          .cacheDefaults(defaultConfig)
          .withInitialCacheConfigurations(Map.of(
                CacheNames.SEARCH_BY_SEMANTIC,
defaultConfig.entryTtl(Duration.ofMinutes(3)),
                CacheNames.RECOMMENDATIONS_ANON,
defaultConfig.entryTtl(Duration.ofMinutes(5))
          ))
          .build();
```

Рисунок 3.7 – Конфігурація Redis cache manager

Далі наведено приклад кешування для отримання всіх опублікованих статей (див. рис. 3.8).

```
@Cacheable(cacheNames = CacheNames.ARTICLES_LIST, key = "'all'")
public List<ArticlePreviewResponse> getPublishedArticles() {
   List<Article> articles = articleService.getPublishedArticles();
   Map<UUID, UserBriefResponse> authors = userIntegrationService.getAuthors(articles);
   return articles.stream()
         .map(article -> ArticlePreviewResponse.fromEntity(article,
authors.get(article.getAuthorId())))
         .toList();
}
```

Рисунок 3.8 – Повний приклад кешування списку опублікованих статей

При створенні, оновленні або видаленні контенту виконується інвалідація відповідних кешів, що дозволяє підтримувати актуальність даних без повного вимкнення кешування (див. рис. 3.9).

```
@Caching(evict = {
      @CacheEvict(cacheNames = CacheNames.ARTICLES_LIST, key = "'all'"),
      @CacheEvict(cacheNames = CacheNames.ARTICLE_BY_ID, allEntries = true),
      @CacheEvict(cacheNames = CacheNames.SEARCH_BY_TITLE, allEntries = true),
      @CacheEvict(cacheNames = CacheNames.SEARCH_BY_SEMANTIC, allEntries =
true),
      @CacheEvict(cacheNames = CacheNames.RECOMMENDATIONS_ANON, allEntries
= true)
})
public ArticleWithUserResponse create(CreateArticleRequest request, UUID authorId) { ... }
```

Рисунок 3.9 – Інвалідація кешу після модифікації контенту

Для забезпечення безпеки системи використовується Spring Security, JWT та OAuth 2.0 автентифікація через зовнішніх провайдерів Google і GitHub. На рівні security chain визначаються правила доступу до шляхів, а також налаштовується JWT-перевірка та OAuth2 login flow.

Наведений фрагмент задає базові правила доступу до API, підключає JWT Resource Server та активує OAuth 2.0 вхід через зовнішніх провайдерів (див. рис. 3.10).

```
@Caching(evict = {
    @CacheEvict(cacheNames = CacheNames.ARTICLES_LIST, key = "'all'"),
    @CacheEvict(cacheNames = CacheNames.ARTICLE_BY_ID, allEntries = true),
    @CacheEvict(cacheNames = CacheNames.SEARCH_BY_TITLE, allEntries = true),
    @CacheEvict(cacheNames = CacheNames.SEARCH_BY_SEMANTIC, allEntries =
true),
    @CacheEvict(cacheNames = CacheNames.RECOMMENDATIONS_ANON, allEntries
= true)
})
public ArticleWithUserResponse create(CreateArticleRequest request, UUID authorId) { ... }
```

Рисунок 3.10 – Security chain

Після успішної автентифікації система повинна коректно зіставити зовнішній акаунт із внутрішнім профілем користувача (див. рис. 3.11 та 3.12).

```
return userRepository.findByGoogleSub(user.getSubject())
    .orElseGet(() -> userRepository.findByEmail(user.getEmail())
        .map(existingUser -> {
            existingUser.setGoogleSub(user.getSubject());
            return userRepository.save(existingUser);
        })
        .orElseGet(() -> {
            CustomUserDetails newUser = new CustomUserDetails();
            {Set fields}
            return userRepository.save(newUser);
        }));
```

Рисунок 3.11 – Обробка Google OAuth 2.0 користувача

Система спочатку виконує пошук користувача за googleSub. Якщо користувача не знайдено, додатково виконується пошук за email для можливого зв’язування існуючого акаунта. Лише у випадку відсутності відповідного профілю створюється новий користувач.

```
return userRepository.findByGithubSub(githubSub)
        .orElseGet(() -> {
            if (email != null) {
                return userRepository.findByEmail(email)
                        .map(existing -> {
                            existing.setGithubSub(githubSub);
                            return userRepository.save(existing);
                        })
                        .orElseGet(() -> createGithubUser(githubSub, email, login, name, avatar));
            }
            return createGithubUser(githubSub, null, login, name, avatar);
        });
```

Рисунок 3.12 – Обробка GitHub OAuth 2.0 користувача

У випадку GitHub система також виконує пошук користувача за githubSub, після чого намагається зв’язати зовнішній акаунт із уже існуючим профілем за email. Якщо відповідний користувач відсутній, створюється новий профіль на основі GitHub-атрибутів.

Таким чином, наведені фрагменти коду демонструють практичну реалізацію ключових механізмів системи, зокрема інтеграцію великих мовних моделей, формування векторних представлень, реалізацію рекомендаційного функціоналу, семантичного пошуку, кешування даних та OAuth 2.0 автентифікації. Це дозволяє забезпечити поєднання AI-функціоналу, продуктивності та безпечної взаємодії користувачів у межах мікросервісного вебзастосунку.

Повна програмна реалізація наведена у GitHub репозиторії [19].

3.2 Огляд результатів реалізації

У даному пункті наведено результати реалізації основних функціональних можливостей розробленого вебзастосунку. Для демонстрації

роботи системи використано інтерфейс клієнтського застосунку, а також приклади взаємодії користувача з системою в цілому.

Коли користувач вперше відкриває за стосунок, відбувається відображення головної сторінки системи для неавторизованого користувача. Оскільки система не має інформації про вподобання користувача, рекомендаційний механізм використовує fallback-підхід та формує стрічку на основі нових і популярних статей (див. рис. 3.13).

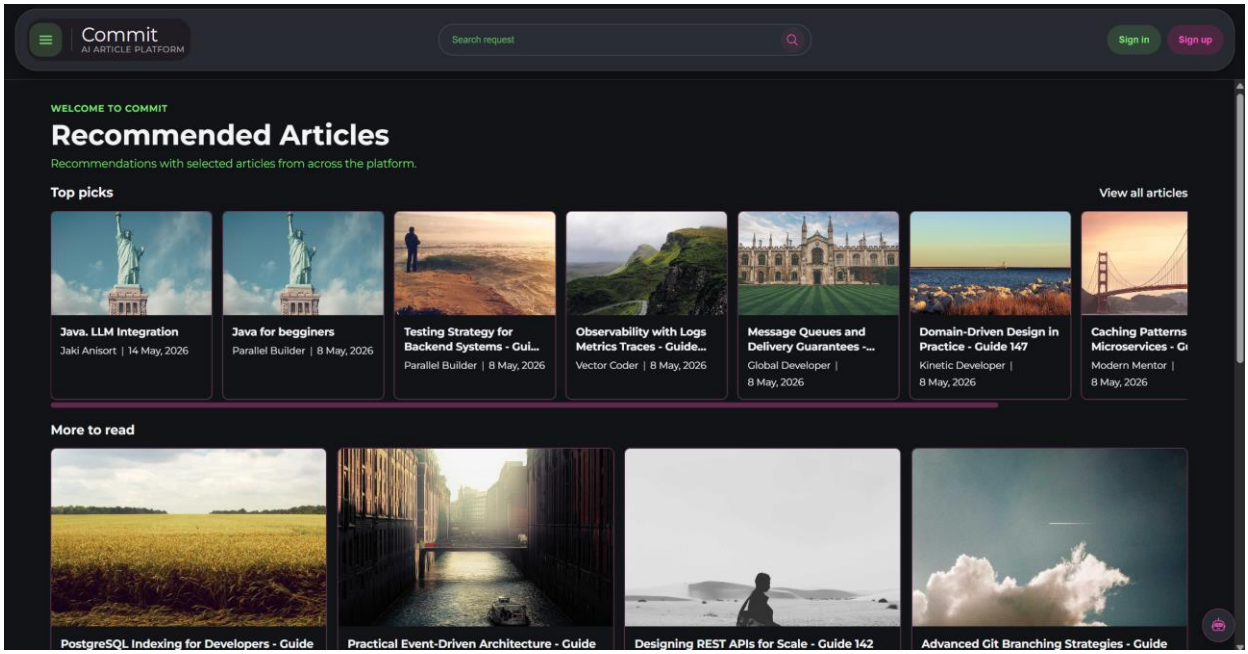


Рисунок 3.13 – Головна сторінка

Для доступу до персоналізованого функціоналу система підтримує реєстрацію та автентифікацію користувачів. Далі наведено сторінки реєстрації та авторизації, які підтримують як класичний вхід, так і OAuth2-автентифікацію через Google та GitHub (див. рис. 3.14).

Після автентифікації користувач отримує доступ до персонального профілю, який містить інформацію про опубліковані статті, вподобаний контент та соціальні взаємодії. Наступним продемонстровано профіль користувача зі списком вподобаних статей, що надалі використовуються рекомендаційною системою для формування персоналізованої стрічки (див. рис. 3.15).

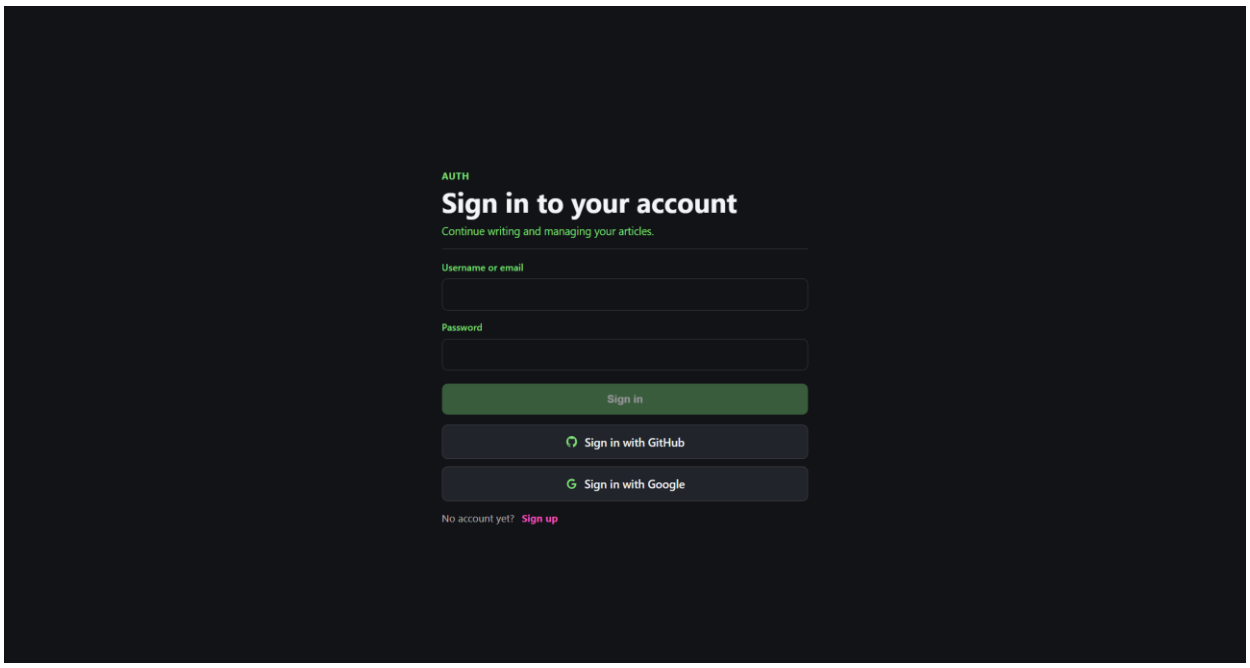


Рисунок 3.14 – Сторінка авторизації

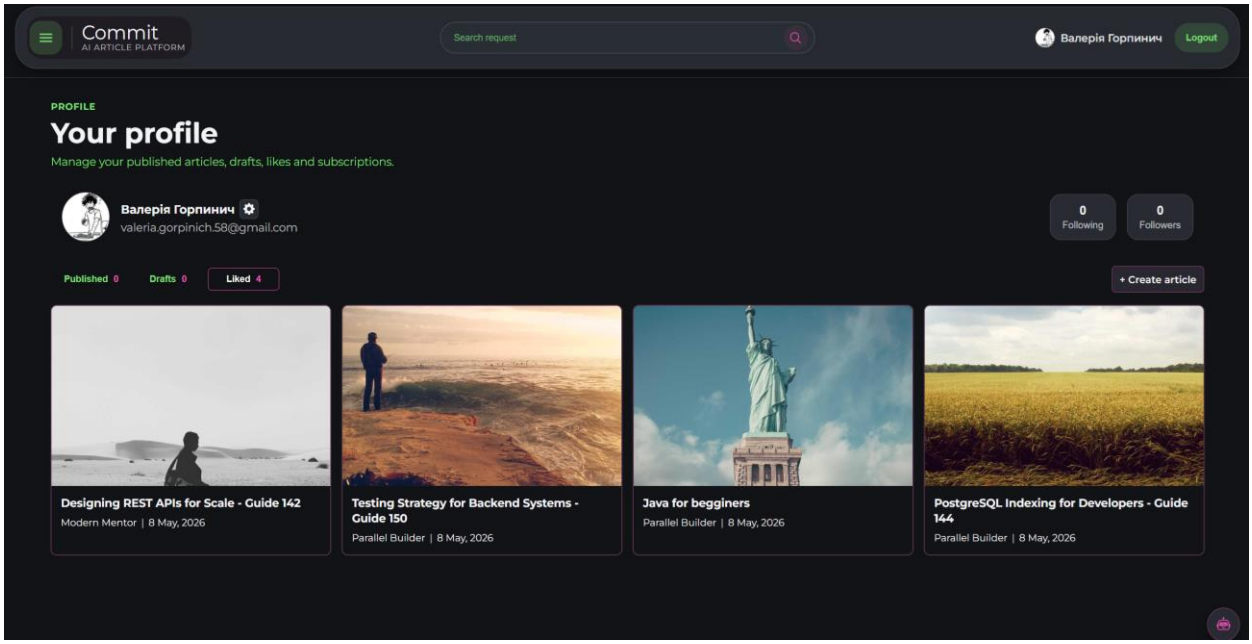


Рисунок 3.15 – Сторінка власного профілю користувача

Приклад роботи персоналізованої рекомендаційної системи після авторизації користувача. Рекомендовані статті відповідають тематиці вподобаного контенту та формуються на основі semantic similarity між embeddings вподобаних і доступних статей (див. рис. 3.16).

Далі продемонстровано роботу семантичного пошуку у системі (див. рис. 17).

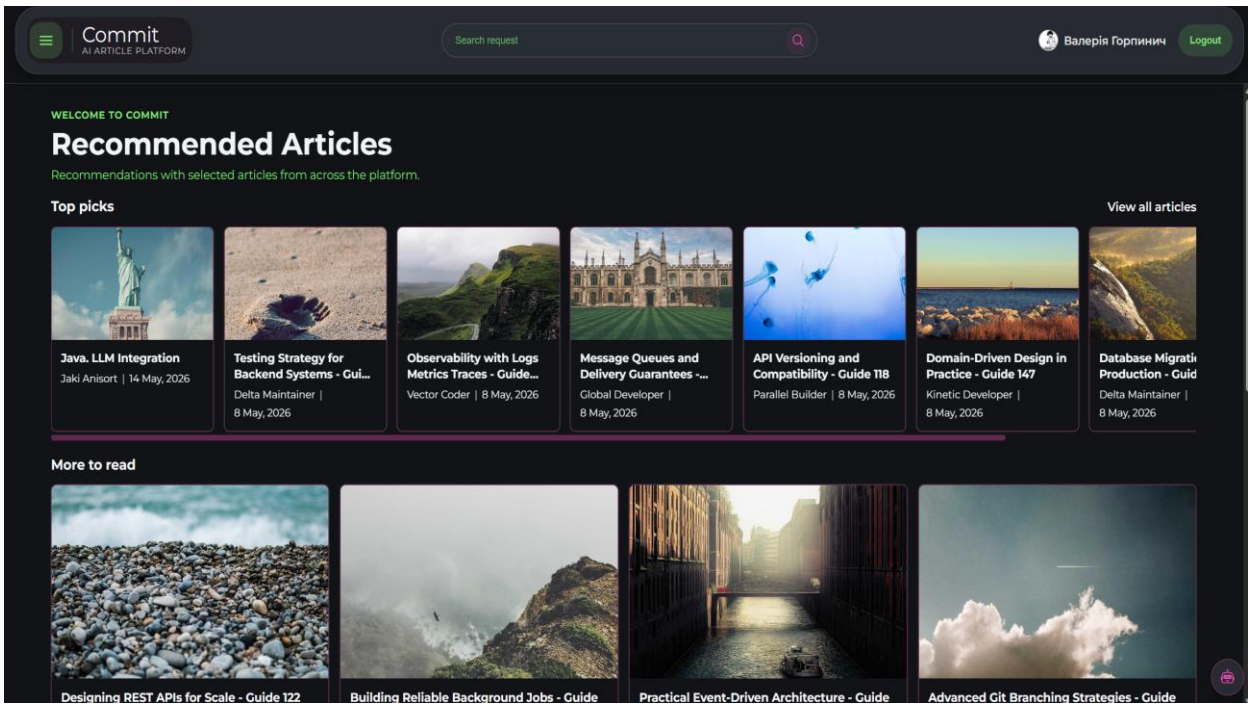


Рисунок 3.16 – Головна сторінка із персоналізованими рекомендаціями

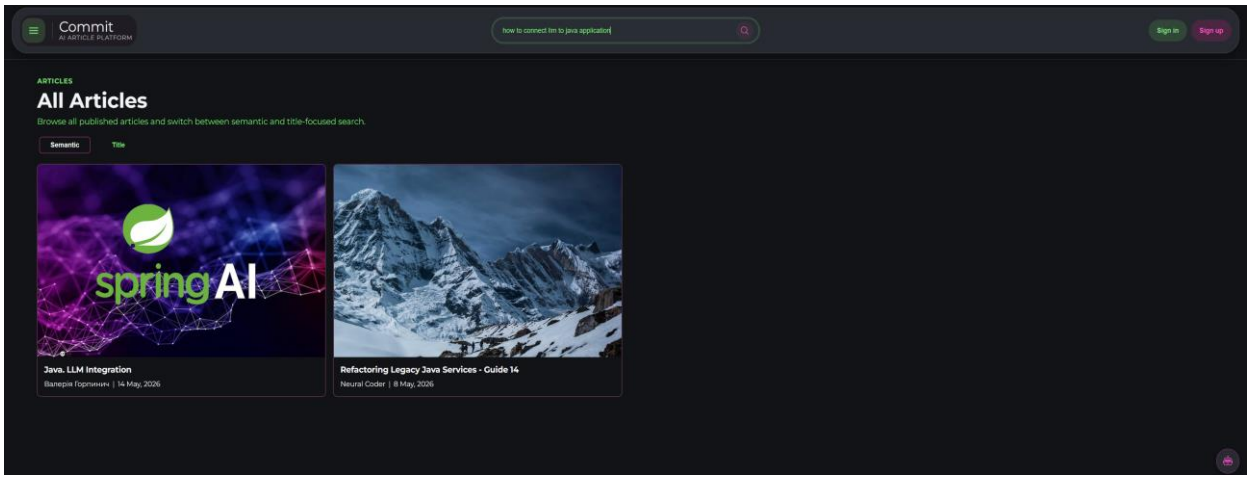


Рисунок 3.17 – Семантичний пошук

Приклад взаємодії користувача з AI-advisor. Під час формування відповіді система використовує RAG-підхід, поєднуючи історію діалогу та релевантний контекст із vector store перед генерацією відповіді великою мовною моделлю (див. рис. 3.18).

Для спрощення роботи з контентом система підтримує AI-assistant у межах редактора статей. Продемонстровано сторінку редагування статті з відкритим assistant-компонентом, який дозволяє взаємодіяти з великою мовною моделлю безпосередньо під час створення контенту (див. рис. 3.19).

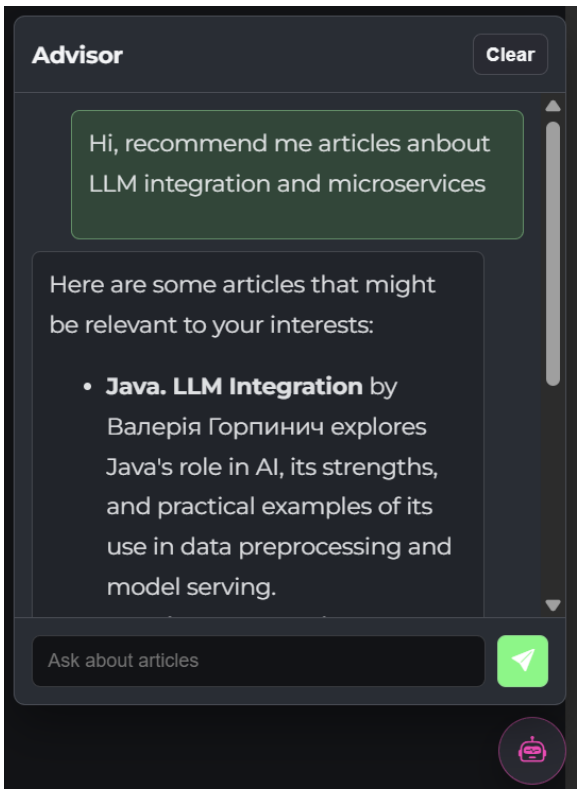


Рисунок 3.18 – Взаємодія з AI-advisor

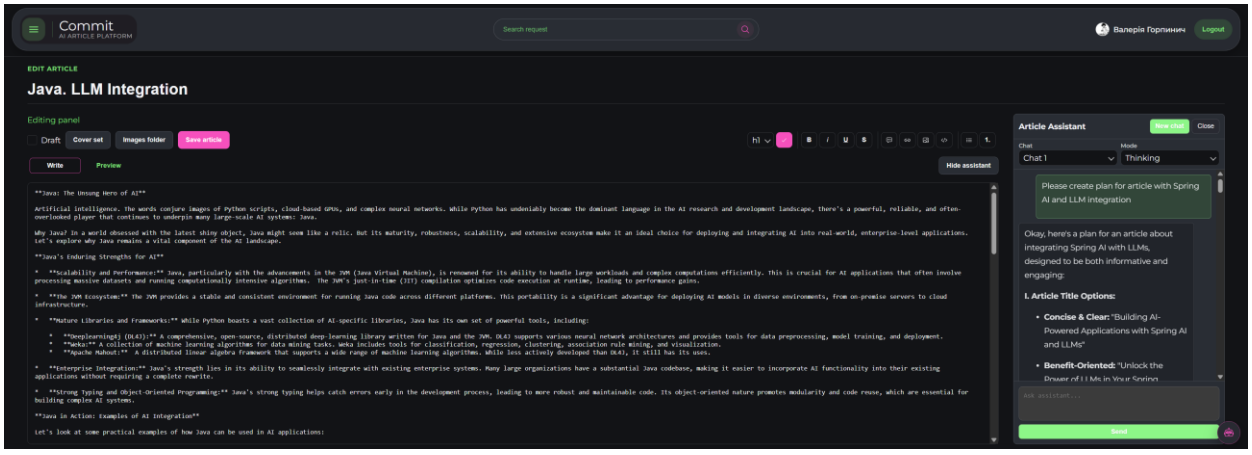


Рисунок 3.19 – Сторінка редагування статті із використанням AI-асистента

Далі наведені механізми керування медіа ресурсами статті. Система підтримує завантаження, відображення та видалення зображень через окремий файловий мікросервіс із використанням Google Cloud Storage (див. рис. 3.20).

Далі після публікації статті нас перенаправляє на сторінку перегляду статті, яка містить markdown-контент, інформацію про автора та реакції користувачів (див. рис. 3.21).

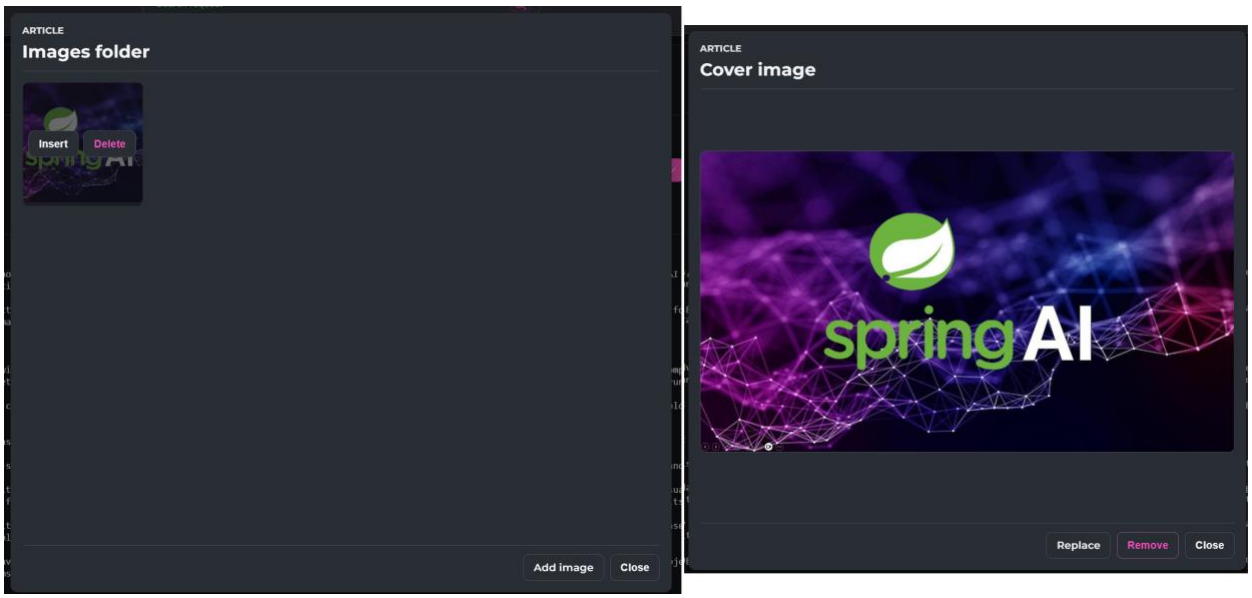


Рисунок 3.20 – Керування медіа ресурсами статті

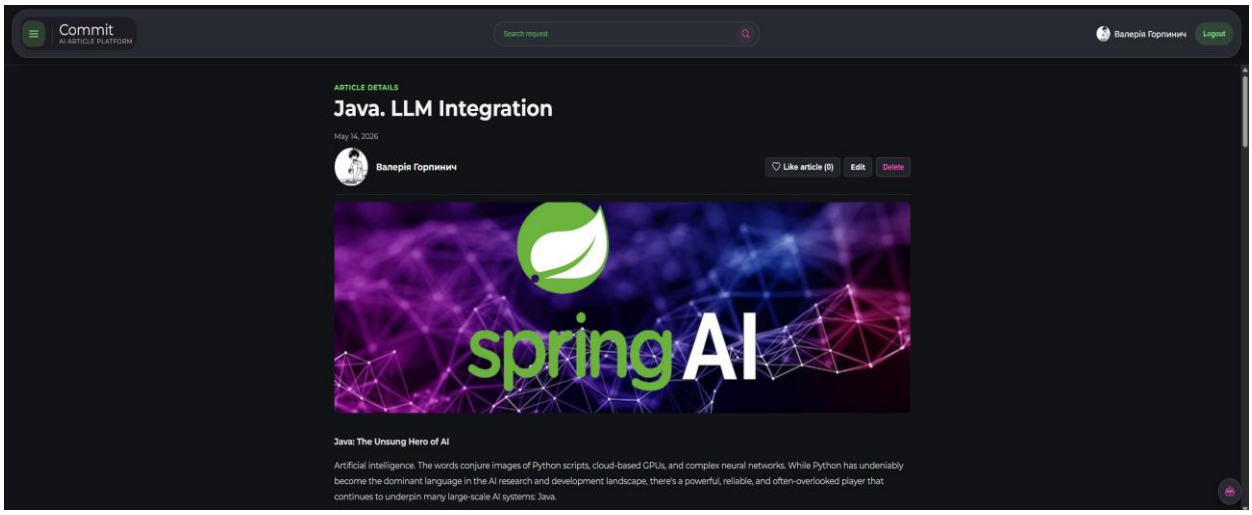


Рисунок 3.21 – Сторінка перегляду статті

3.3 Тестування системи

Для перевірки працездатності розробленої системи було сформовано набір тестових сценаріїв, що охоплюють ключові користувацькі дії у вебзастосунку. Автоматизацію перевірок користувацького інтерфейсу виконано із застосуванням Selenium.

Перший тест-кейс розписано для тестування функціоналу «Отримання сторінки рекомендацій» (див. табл. 3.1).

Таблиця 3.1 – Тест-кейс для функціоналу «Отримання сторінки рекомендацій»

Мета тестування	**Перевірити доступність сторінки рекомендацій**
Вхідні дані	Авторизований користувач
Очікуваний результат	Відкривається /home, відображається блок рекомендацій або порожній стан
Кроки виконання тесту	Перейти на http://localhost:81/home
Критерії успішності	Сторінка завантажена, є рекомендовані статті або пустий стейт

Далі наведено фрагмент коду з реалізації тестового сценарію «Отримання сторінки рекомендацій» (див. рис. 3.22).

```
open("/home");
wait.until(ExpectedConditions.or(
  visibilityOfElementLocated(By.xpath("//*[contains(.,'Recommended Articles')]")),
  visibilityOfElementLocated(By.cssSelector(".empty-state"))
));
```

Рисунок 3.22 – Фрагмент коду з реалізації тестового сценарію «Отримання сторінки рекомендацій»

Наступний тест-кейс присвячено тестуванню функціоналу «Створення статті» (див. табл. 3.2).

Таблиця 3.2 – Тест-кейс для функціоналу «Створення статті»

Мета тестування	**Перевірити створення нової статті з обкладинкою**
Вхідні дані	title, markdown, cover image
Очікуваний результат	Після публікації відкривається сторінка статті /articles/{id}
Кроки виконання тесту	Відкрити /edit-article, заповнити форму, додати cover, натиснути Publish article
Критерії успішності	Є редирект на /articles/ і отримано articleId

Фрагмент коду з реалізації тестового сценарію «Створення статті» (див. рис. 3.23).

```
open("/edit-article");
driver.findElement(By.cssSelector("input.title-input")).sendKeys(title);
driver.findElement(By.cssSelector("textarea.editor-textarea")).sendKeys(markdown);
driver.findElement(By.xpath("//button[contains(.,'Set cover')]")).click();
driver.findElement(By.cssSelector("input[type='file']")).sendKeys(coverPath);
driver.findElement(By.xpath("//button[contains(.,'Publish article')]")).click();
wait.until(d -> d.getCurrentUrl().contains("/articles/"));
```

Рисунок 3.23 – Фрагмент коду з реалізації тестового сценарію «Створення статті»

Далі наведено тестовий сценарій для функціоналу «Редагування статті» (див. табл. 3.3). Фрагмент коду з реалізації тестового сценарію «Редагування статті» (див. рис. 3.24).

Таблиця 3.3 – Тест-кейс для функціоналу «Редагування статті»

Мета тестування	**Перевірити можливість редагування існуючої статті**
Вхідні дані	articleId, новий title, новий content
Очікуваний результат	Збереження змін та повернення на детальну сторінку статті
Кроки виконання тесту	Відкрити /edit-article/{id}, змінити поля, натиснути Save article
Критерії успішності	URL містить /articles/{id}, кнопка Edit доступна

```
open("/edit-article/" + articleId);
WebElement t = driver.findElement(By.cssSelector("input.title-input"));
t.clear(); t.sendKeys(updatedTitle);
WebElement c = driver.findElement(By.cssSelector("textarea.editor-textarea"));
c.clear(); c.sendKeys(updatedContent);
driver.findElement(By.xpath("//button[contains(.,'Save article')]")).click();
wait.until(d -> d.getCurrentUrl().contains("/articles/" + articleId));
```

Рисунок 3.24 – Фрагмент коду з реалізації тестового сценарію «Редагування статті»

Тестовий сценарій для функціоналу «Like/unlike статті» (див. табл. 3.4). Далі наведено фрагмент коду для наданого тестового сценарію (див. рис. 3.25).

Таблиця 3.4 – Тест-кейс для функціоналу «Like/unlike статті»

Мета тестування	**Перевірити зміну статусу лайка для статті**
Вхідні дані	articleId статті іншого автора
Очікуваний результат	Збереження змін та повернення на детальну сторінку статті
Кроки виконання тесту	Відкрити /edit-article/{id}, змінити поля, натиснути Save article
Критерії успішності	URL містить /articles/{id}, кнопка Edit доступна

```
open("/articles/" + likeableId);
WebElement btn = wait.until(elementToBeClickable(By.cssSelector(".article-like-button")));
btn.click();
wait.until(d -> d.findElement(By.cssSelector(".article-like-
button")).getText().contains("Unlike"));
btn.click();
wait.until(d -> d.findElement(By.cssSelector(".article-like-
button")).getText().contains("Like"));
```

Рисунок 3.25 – Фрагмент коду з реалізації тестового сценарію «Редагування статті»

Останній тестовий сценарій описано для функціоналу «Видалення статті» (див. табл. 3.5).

Таблиця 3.5 – Тест-кейс для функціоналу «Видалення статті»

Мета тестування	**Перевірити видалення статті автором**
Вхідні дані	articleId власної статті
Очікуваний результат	Стаття видалена, сторінка більше недоступна для редагування/видалення
Кроки виконання тесту	Відкрити /articles/{id}, натиснути Delete, підтвердити діалог
Критерії успішності	Перехід зі сторінки статті або відсутність кнопки Delete при повторному відкритті

Фрагмент коду, що реалізовує тест-кейс «Видалення статті» (див. рис. 3.26).

```
open("/articles/" + likeableId);
WebElement btn = wait.until(elementToBeClickable(By.cssSelector(".article-like-button")));
btn.click();
wait.until(d -> d.findElement(By.cssSelector(".article-like-
button")).getText().contains("Unlike"));
btn.click();
wait.until(d -> d.findElement(By.cssSelector(".article-like-
button")).getText().contains("Like"));
```

Рисунок 3.26 – Фрагмент коду з реалізації тестового сценарію «Редагування статті»

У результаті вісім успішних тестів, окрім описаних тестів також було проведено тестування для для перевірки стабільності переходів між сторінками, коректної обробки користувацьких дій та узгодженості відображення даних після виконання операцій створення, редагування і видалення статей (див. рис. 3.27).

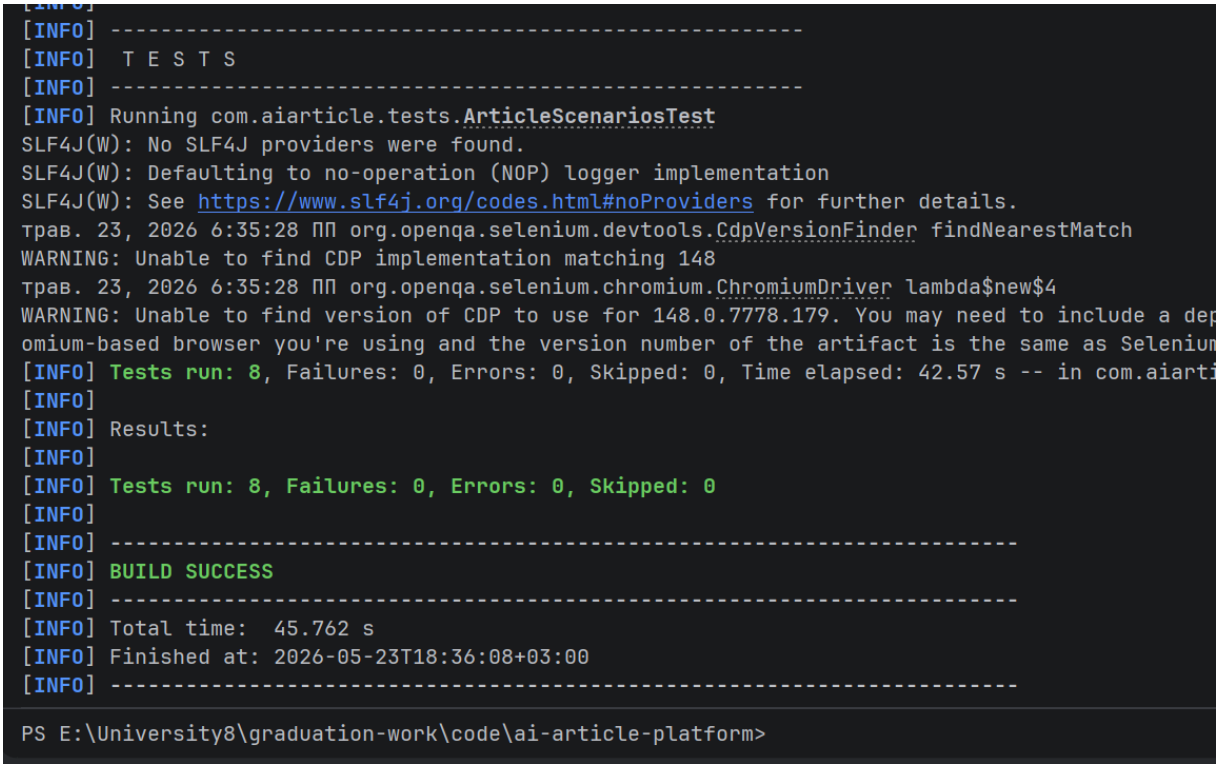


Рисунок 3.27 – Результати тестування

Для перевірки продуктивності та стабільності роботи системи було проведено навантажувальне тестування за допомогою Apache JMeter. У межах тестування перевірялась робота API отримання списку опублікованих статей. Для тесту було налаштовано 10000 віртуальних користувачів із поступовим збільшенням навантаження протягом 100 секунд, при цьому кожен користувач виконував один HTTP-запит до системи (див. рис. 3.28).

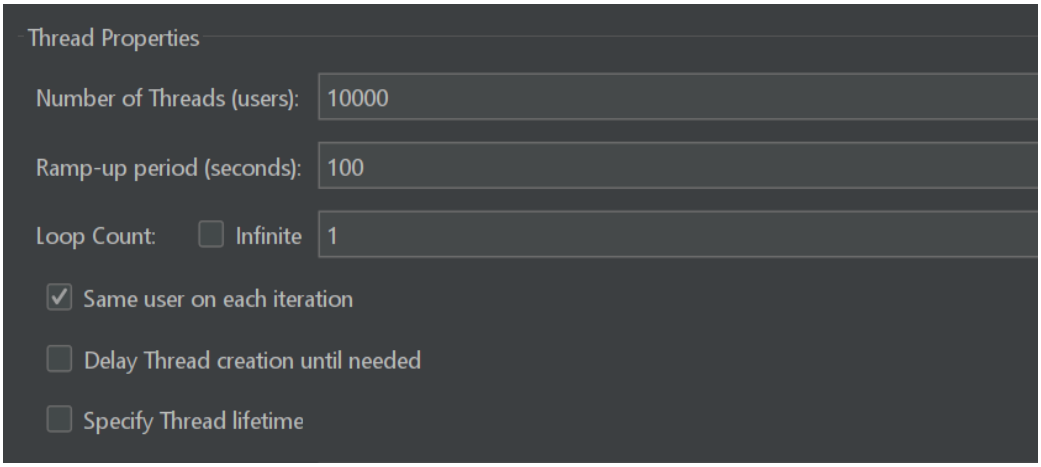


Рисунок 3.28 – Налаштування навантажувального тестування

Результати навантажувального тестування показали стабільну роботу системи в умовах підвищеного навантаження. Під час виконання 10000 HTTP-запитів відсоток помилок становив 0%, що свідчить про коректну обробку запитів серверною частиною застосунку.

Середній час відповіді склав близько 100 мс, а пропускна здатність системи досягала приблизно 100 запитів за секунду. Отримані результати підтверджують достатній рівень продуктивності та ефективність використання механізмів кешування й мікросервісної архітектури системи (див. рис. 3.29).

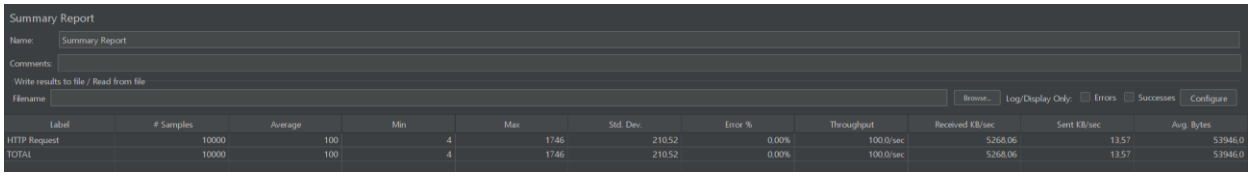


Рисунок 3.29 – Результати навантажувального тестування

Отже, виконане тестування підтвердило коректну роботу основних функціональних сценаріїв вебзастосунку, узгодженість поведінки інтерфейсу після ключових дій користувача, а також стабільність системи в умовах підвищеного навантаження. Результати функціонального та навантажувального тестування свідчать про достатній рівень продуктивності, надійності та готовності розробленого рішення до подальшого використання й розвитку.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено мікросервісний вебзастосунок для публікації технічних статей із інтеграцією рекомендаційної системи та великих мовних моделей. У межах роботи було реалізовано клієнтську та серверну частину системи, механізми OAuth 2.0 автентифікації, AI-assistant, advisor-компонент, семантичний пошук, персоналізовані рекомендації та механізми роботи з контентом.

В першому розділі проаналізовано предметну область та існуючі аналоги, визначено функціональні й нефункціональні вимоги до системи, а також сформовано підходи до реалізації семантичного пошуку, персоналізованих рекомендацій та AI-взаємодії з користувачем. У другому розділі виконано концептуальне, логічне та фізичне проєктування системи, спроєктовано мікросервісну архітектуру застосунку, ER-модель баз даних та UML-діаграми основних компонентів. У третьому розділі наведено програмну частину вебзастосунку, а також проведено UI-тестування за допомогою Selenium і навантажувальне тестування із використанням Apache JMeter, результати яких підтвердили коректність, стабільність та працездатність розробленої системи.

Практичне значення отриманих результатів полягає у створенні вебзастосунку з підтримкою семантичного пошуку, персоналізованих рекомендацій та AI-взаємодії в межах мікросервісної архітектури. Розроблені рішення можуть бути використані для побудови сучасних платформ для роботи з технічним контентом.

Подальший розвиток системи може передбачати реалізацію системи тегів, розширення соціального функціоналу, додавання механізмів модерації контенту, а також використання брокерів повідомлень, зокрема RabbitMQ або Apache Kafka, для оптимізації взаємодії між мікросервісами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Spring Boot. *Spring*. URL: https://spring.io/projects/spring-boot (дата звернення: 16.03.2026).

2. Jayeshwarke. Getting Started with Spring Boot: Advantages, Disadvantages, and Use Cases. *Medium*. URL: https://medium.com/@jayeshwarke011/getting-started-with-spring-boot-advantages-disadvantages-and-use-cases-497b0f04fb86 (дата звернення: 16.03.2026).

3. Spring AI. *Spring*. URL: https://spring.io/projects/spring-ai (дата звернення: 16.03.2026).

4. PostgreSQL: The World’s Most Advanced Open Source Relational Database. *PostgreSQL*. URL: https://www.postgresql.org/ (дата звернення: 20.03.2026).

5. Pazos P. Unlocking the Power of Vector Search with PostgreSQL and pgvector. *Medium*. URL: https://medium.com/@pablop44/unlocking-the-power-of-vector-search-with-postgresql-and-pgvector-f03b97deaeec (дата звернення: 20.03.2026).

6. Your high-performance caching solution. *Redis*. URL: https://redis.io/solutions/caching/ (дата звернення: 20.03.2026).

7. VertexAI Gemini Chat. *Spring*. URL: https://docs.spring.io/spring-ai/reference/api/chat/vertexai-gemini-chat.html (дата звернення: 02.04.2026).

8. Google VertexAI Text Embeddings. *Spring*. URL: https://docs.spring.io/spring-ai/reference/api/embeddings/vertexai-embeddings-text.html (дата звернення: 06.04.2026).

9. Quynh Pham. 8 Key Benefits of Using Angular for Development. *Orient Software*. URL: https://www.orientsoftware.com/blog/benefits-of-angular/ (дата звернення: 08.04.2026).

10. Angular Signals. *Angular*. URL: https://angular.dev/guide/signals (дата звернення: 08.04.2026).

11. NGINX Reverse Proxy. *Nginx*. URL: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy (дата звернення: 10.04.2026).

12. What is Docker and what are its advantages? *Dimensiona*. URL: https://www.dimensiona.com/en/what-is-docker-and-what-are-its-advantages/ (дата звернення: 14.04.2026).

13. Sana Afreen. What is Docker Compose: Example, Benefits and Basic Commands. *Simplilearn*. URL: https://www.simplilearn.com/tutorials/docker-tutorial/docker-compose (дата звернення: 14.04.2026).

14. The Selenium Browser Automation Project. *Selenium*. URL: https://www.selenium.dev/documentation/ (дата звернення: 20.04.2026).

15. Apache JMeter. *Apache*. URL: https://jmeter.apache.org/ (дата звернення: 20.04.2026).

16. Medium. URL: https://medium.com/ (дата звернення: 22.04.2026).

17. GeeksForGeeks. URL: https://www.geeksforgeeks.org/ (дата звернення: 22.04.2026).

18. DEV Community. URL: https://dev.to/ (дата звернення: 22.04.2026).

19. Горпинич В. О. AI Article Platform. *GitHub*. URL: https://github.com/anisort/ai-article-platform (дата звернення: 23.05.2026).