МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інженерії програмного забезпечення
______________ Євген ДАВИДЕНКО
«___»_________ 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

ПЛАТФОРМА АВТОМАТИЗОВАНОЇ ПЕРЕВІРКИ КОДУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ

Спеціальність 121 Інженерія програмного забезпечення

Освітня програма «Інженерія програмного забезпечення»

Здобувач __________________ **___Євгеній БАРДОВСЬКИЙ_**

«___»__________________ 2024 р.

Керівник роботи

канд. техн. наук,

доцент _________________ **_______Гліб ГОРБАНЬ_______**

«___»_________________ 2024 р.

Миколаїв – 2024

Чорноморський національний університет імені Петра Могили

(повне найменування закладу вищої освіти)

Факультет	Комп'ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Другий (магістерський)
Освітній ступень	Магістр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри інженерії

програмного забезпечення

____________Євген ДАВИДЕНКО

«___»_______2024 р.

ЗАВДАННЯ

на кваліфікаційну магістерську роботу здобувача

Бардовського Євгенія

1. Тема кваліфікаційної роботи «Платформа автоматизованої перевірки коду з використанням технологій штучного інтелекту» затверджена наказом ЧНУ ім. Петра Могили від «04»_ 09 _2024 р. № 220

2. Строк представлення кваліфікаційної роботи «___»____________2024 р.

3. Очікуваний результат роботи: платформа для автоматизованої перевірки коду з використанням штучного інтелекту.

4. Перелік питань, що підлягають розробці

– аналітичний огляд та постановка задачі;

– моделювання процесу та методів перевірки коду. Функціональні та інформаційні моделі;

– архітектура, моделювання та проєктування програмного забезпечення;
– кодування та тестування ПЗ платформи автоматизованої перевірки коду.

5. Перелік графічних матеріалів: презентація.
6. Консультанти:

Консультант	**Кафедра (організація)**	**Частина роботи**

Керівник роботи __________________ ___Гліб ГОРБАНЬ__
Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

Здобувач _____________________ Євгеній_БАРДОВСЬКИЙ
Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

Дата видачі завдання «____» _______________2024 р

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: **Платформа автоматизованої перевірки коду з використанням технологій штучного інтелекту**

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КМР	10.09.2024 р.	11.09.2024 р.	виконано
2.	Огляд літератури за темою роботи	13.09.2024 р.	19.09.2024 р.	виконано
3.	Складання календарного плану КМР	20.03.2024 р.	21.03.2024 р.	виконано
4.	Аналітичний огляд та постановка задачі	24.09.2024 р.	26.09.2024 р.	виконано
5.	Моделювання процесу та методів перевірки коду.	28.09.2024 р.	30.09.2024 р.	виконано
6.	Моделювання та конструювання ПЗ	01.10.2024 р.	06.10.2024 р.	виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів	11.10.2024 р.	07.11.2024 р.	виконано
8.	Оформлення КМР та презентації	10.11.2024 р.	28.12.2024 р.	виконано
9.	Відгук керівника КМР	01.12.2024 р.	10.12.2024 р.	виконано
10.	Попередній захист	02.12.2024 р.	02.12.2024 р.	виконано
11.	Рецензування	10.12.2024 р.	12.12.2024 р.	виконано
12.	Завершення оформлення КМР та презентації	02.12.2024 р.	12.12.2024 р.	виконано
13.	Захист кваліфікаційної роботи	19.12.2024 р.	19.12.2024 р.	виконано

Здобувач ___________________ **Євгеній БАРДОВСЬКИИ**

«__» ____________ 2024 р.

Керівник роботи
канд. техн. наук,
доцент ___________________ **___Гліб ГОРБАНЬ__**

«__»_____________ 2024 р.

АНОТАЦІЯ

до кваліфікаційної магістерської роботи

«Платформа автоматизованої перевірки коду з використанням технологій штучного інтелекту»

Здобувач 608м гр.: Бардовський Євгеній

Керівник: канд. техн. наук, доцент Горбань Гліб

Актуальність та науково-практичне значення обраної теми полягає у необхідності підвищення ефективності та якості процесу розробки програмного забезпечення шляхом впровадження автоматизованих засобів перевірки коду. Використання штучного інтелекту дозволяє не лише автоматизувати рутинні завдання, але й забезпечити більш глибокий аналіз коду з точки зору кращих практик та потенційних вразливостей.

Об'єктом кваліфікаційної роботи є процес перевірки коду в розробці програмного забезпечення.

Предметом кваліфікаційної роботи є методи та засоби автоматизованої перевірки коду з використанням технологій штучного інтелекту.

Метою роботи є підвищення ефективності процесу перевірки коду в розробці програмного забезпечення шляхом розробки платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту.

Відповідно до мети визначено такі **завдання**:

– проаналізувати існуючі рішення для автоматизованої перевірки коду та виявити їх обмеження;

– дослідити можливості використання технологій штучного інтелекту для аналізу та перевірки коду;

– розробити методи та засоби інтеграції технологій штучного інтелекту у процес автоматизованої перевірки коду;

– реалізувати прототип платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту;

– оцінити ефективність розробленої платформи у покращенні якості коду

Обґрунтування необхідності нової розробки базується на аналізі сучасного стану проблеми. Існуючі інструменти автоматизованої перевірки коду часто обмежені в можливостях та не враховують контекстуальні особливості коду. Вони можуть виявляти синтаксичні помилки або прості патерни, але не здатні забезпечити глибокий семантичний аналіз.

Обґрунтування основних проєктних рішень полягає у виборі технологій штучного інтелекту як базового інструменту для аналізу коду, що дозволяє перевіряти код приблизно як людина, враховуючи контекст.

Сфера застосування результатів включає компанії та команди розробників програмного забезпечення, які прагнуть підвищити якість свого коду та оптимізувати процеси розробки.

Пояснювальна записка складається зі вступу, чотирьох розділів, висновків та додатків. У першому розділі здійснено аналіз існуючих аналогів систем платформ для автоматизованої перевірки коду. У другому розділі розглядаються моделювання та методи автоматизованої перевірки коду. У третьому розділі описано архітектуру, моделювання та проектування платформи. У четвертому розділі розглянуто та описано програмну реалізацію платформи для автоматизованої перевірки коду та її тестування.

Кваліфікаційна магістерська робота містить 95 сторінок (без додатків), 33 рисунків, 32 джерела та 3 додатка.

Ключові слова: перевірка коду, штучний інтелект, pull request, GitHub, .NET, Angular.

ABSTRACT

to the qualification master's thesis

"Platform for automated code verification using artificial intelligence technologies"

Student of group 608m: Yevhenii Bardovskyi

Supervisor: PhD, Associate Professor Hlib Horban

The relevance and scientific and practical significance of the chosen topic lies in the need to improve the efficiency and quality of the software development process by implementing automated code verification tools. The use of artificial intelligence allows not only to automate routine tasks, but also to provide a deeper analysis of the code from the point of view of best practices and potential vulnerabilities.

The object of the qualification work is the code verification process in software development.

The subject of the qualification work is the methods and tools of automated code verification using artificial intelligence technologies.

The purpose of the work is to improve the efficiency of the code verification process in software development by developing a platform for automated code verification using artificial intelligence technologies.

In accordance with the purpose, the following **tasks** have been defined:

– analyze existing solutions for automated code verification and identify their limitations;

– investigate the possibilities of using artificial intelligence technologies for code analysis and verification;

– develop methods and tools for integrating artificial intelligence technologies into the automated code verification process;

– implement a prototype of a platform for automated code verification using artificial intelligence technologies;

– assess the effectiveness of the developed platform in improving code quality.

The justification for the need for a new development is based on an analysis of the current state of the problem. Existing automated code verification tools are often limited in capabilities and do not take into account the contextual features of the code.

They can detect syntax errors or simple patterns, but are not able to provide deep semantic analysis.

The justification for the main design decisions is to choose artificial intelligence technologies as the basic tool for code analysis, which allows checking the code approximately like a person, taking into account the context.

The scope of application of the results includes companies and software development teams that seek to improve the quality of their code and optimize development processes.

The explanatory note consists of an introduction, four sections, conclusions and appendices. The first section analyzes existing analogues of automated code verification platform systems. The second section considers modeling and methods of automated code verification. The third section describes the architecture, modeling and design of the platform. The fourth section considers and describes the software implementation of the automated code verification platform and its testing.

The qualifying master's thesis contains 95 pages (without appendices), 33 figures, 32 sources and 3 appendices.

Keywords: code review, artificial intelligence, pull request, GitHub, .NET, Angular.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4
ВСТУП 5
1 АНАЛІТИЧНИЙ ОГЛЯД ТА ПОСТАНОВКА ЗАДАЧІ 7
1.1 Проблема та важливість перевірки програмного коду 7
1.2 Структурні особливості процесу перевірки програмного коду 9
1.3 Функціональні особливості процесу перевірки програмного коду 10
1.4 Огляд і аналіз минулого та сучасного стану інформаційних технологій у даній предметній області 11
1.5 Огляд і аналіз існуючих методів і засобів вирішення завдань КМР 17
1.6 Обґрунтування та вибір підходів щодо виконання завдань КМР 20
1.7 Специфікації вимог до програмного забезпечення 24
Висновки до розділу 1 28
2 МОДЕЛЮВАННЯ ПРОЦЕСУ ТА МЕТОДІВ ПЕРЕВІРКИ КОДУ. ФУНКЦІОНАЛЬНІ ТА ІНФОРМАЦІЙНІ МОДЕЛІ 30
2.1 Вибір та обґрунтування підходів до моделювання 30
2.2 Функціональна модель платформи для автоматизованої перевірки коду ... 32
2.3 Інформаційна модель платформи для автоматизованої перевірки коду 38
2.4 Алгоритм роботи платформи для автоматизованої перевірки коду 40
Висновки до розділу 2 42
3 АРХІТЕКТУРА, МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 44
3.1 Розробка архітектури платформи для автоматичної перевірки коду 44
3.2 Структура бази даних платформи 49
3.3 Вибір технологій та мов програмування для розробки платформи 51
3.4 Програмна архітектура платформи 59
Висновки до розділу 3 61
4 КОДУВАННЯ ТА ТЕСТУВАННЯ ПЗ ПЛАТФОРМИ ДЛЯ АВТОМАТИЗОВАНОЇ ПЕРЕВІРКИ КОДУ 62

4.1 Налаштування GitHub OAuth 62
4.2 Опис реалізації мікросервісу Pull Request API 63
4.3 Опис реалізації мікросервісу Reviewer API 72
4.4 Опис реалізації мікросервісу Event Handler..74
4.5 Опис реалізації користувацького інтерфейсу 76
Висновки до розділу 4 90
ВИСНОВКИ 91
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 93
ДОДАТОК А Реалізація Back-End частини 96
ДОДАТОК Б Реалізація Front-End частини 113
ДОДАТОК В Апробація кваліфікаційної магістерської роботи...........................122

ПЕРЕЛІК СКОРОЧЕНЬ

КМР – кваліфікаційна магістерська робота

БД – база даних

ШІ – штучний інтелект

ПЗ – програмне забезпечення

ОЗ – операційна система

ООП – об’єктно-орієнтоване програмування

СУБД – система управління базою даних

URL – Uniform Resource Locator

API – Application Programming Interface

UI – User Interface

JSON – JavaScript object notation

SQL – Structed Query Language

NLP – Natural Language Processing

IDEF – Integration Definition for Function Modeling

DFD – Data Flow Diagram

UML – Unified Modeling Language

ER – Entity-Relationship

HTTP – HyperText Transfer Protocol

XML – EXtensible Markup Language

REST – Representational State Transfer

ORM – Object-relational mapping

CI/CD – Continuous Integration/Continuous Delivery

DI – Dependency Injection

ВСТУП

У сучасній індустрії розробки програмного забезпечення якість коду є критично важливою для успішної реалізації проєктів. Традиційні методи перевірки коду вимагають значних ресурсів та часу, що може призводити до затримок у розробці та появи помилок у фінальному продукті. З появою технологій штучного інтелекту, зокрема моделей на зразок ChatGPT, відкриваються нові можливості для автоматизації процесу перевірки коду.

Актуальність та науково-практичне значення обраної теми полягає у необхідності підвищення ефективності та якості процесу розробки програмного забезпечення шляхом впровадження автоматизованих засобів перевірки коду. Використання штучного інтелекту дозволяє не лише автоматизувати рутинні завдання, але й забезпечити більш глибокий аналіз коду з точки зору кращих практик та потенційних вразливостей.

Об'єктом кваліфікаційної роботи є процес перевірки коду в розробці програмного забезпечення.

Предметом кваліфікаційної роботи є методи та засоби автоматизованої перевірки коду з використанням технологій штучного інтелекту.

Метою роботи є підвищення ефективності процесу перевірки коду в розробці програмного забезпечення шляхом розробки платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту.

Відповідно до мети визначено такі **завдання**:

– проаналізувати існуючі рішення для автоматизованої перевірки коду та виявити їх обмеження;

– дослідити можливості використання технологій штучного інтелекту для аналізу та перевірки коду;

– розробити методи та засоби інтеграції технологій штучного інтелекту у процес автоматизованої перевірки коду;

– реалізувати прототип платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту;

– оцінити ефективність розробленої платформи у покращенні якості коду.

Обґрунтування необхідності нової розробки базується на аналізі сучасного стану проблеми. Існуючі інструменти автоматизованої перевірки коду часто обмежені в можливостях та не враховують контекстуальні особливості коду. Вони можуть виявляти синтаксичні помилки або прості патерни, але не здатні забезпечити глибокий семантичний аналіз. Провідні компанії, такі як Microsoft та OpenAI, активно досліджують можливості застосування штучного інтелекту для покращення процесів розробки. Однак, на даний момент існують прогалини у застосуванні моделей на зразок ChatGPT саме для автоматизованої перевірки коду.

Обґрунтування основних проєктних рішень полягає у виборі технологій штучного інтелекту як базового інструменту для аналізу коду, що дозволяє перевіряти код приблизно як людина, враховуючи контекст. Інтеграція цих технологій у платформу автоматизує процес перевірки, забезпечуючи більш глибокий аналіз та рекомендації для розробників.

Сфера застосування результатів включає компанії та команди розробників програмного забезпечення, які прагнуть підвищити якість свого коду та оптимізувати процеси розробки. Розроблена платформа може використовуватись як інструмент підтримки розробників у повсякденній роботі.

Апробація результатів КМР відбулась під час XXVII Всеукраїнської науково-практичної конференції «Могилянські читання – 2024», Миколаїв, 06-10 листопада, 2024 р. (Додаток В) [1].

1 АНАЛІТИЧНИЙ ОГЛЯД ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Проблема та важливість перевірки програмного коду

У сучасній розробці програмного забезпечення процес перевірки коду відіграє ключову роль у забезпеченні його якості та надійності. Об'єктом даного дослідження є саме цей процес перевірки коду, який включає в себе різноманітні методи та практики, такі як код-рев'ю, статичний та динамічний аналіз. Основна мета перевірки коду полягає у виявленні помилок, недоліків, порушень стандартів кодування, а також проблем, які можуть вплинути на безпеку та продуктивність програмного забезпечення.

З ростом IT-індустрії швидкість розробки проєктів значно підвищилась, терміни виконання проєктів зменшується. Підтримувати якість та чистоту програмного коду стає значно важко. Тому виникають нові виклики: потреба у швидкій та ефективній перевірці програмного коду, мінімізація людського фактору або взагалі його відсутність та зменшення кількості помилок в програмному коді.

Час і ресурси, витрачені на виявлення та виправлення помилок на пізніх етапах розробки, можуть бути значно скорочені завдяки ефективній автоматизації. За даними різних досліджень, близько 20-30% часу розробників витрачається на тестування та налагодження коду. Впровадження автоматизованих інструментів може скоротити цей показник до 10-15%, дозволяючи розробникам більше зосередитися на безпосередньому написанні коду, а не на його перевірці [2].

Збільшення вимог до якості програмного забезпечення спричинило активний розвиток інструментів та методологій для автоматизації перевірки коду [3]. Від класичних статичних та динамічних аналізаторів до сучасних систем, які використовують технології штучного інтелекту, цей процес постійно вдосконалюється. Зокрема, автоматизовані інструменти для перевірки коду дозволяють виявляти помилки на ранніх етапах розробки, що суттєво скорочує витрати на виправлення недоліків у подальших етапах. Однак, попри наявні

досягнення, існує чимало недоліків, що знижують ефективність таких інструментів, особливо при роботі з великими або складними проектами.

Традиційні методи автоматизованої перевірки, такі як статичний аналіз коду, хоча і забезпечують певний рівень автоматизації, все ще мають обмежену можливість адаптації до нових патернів помилок [4]. Водночас, штучний інтелект, зокрема машинне навчання, дозволяє розробляти системи, які можуть самостійно навчатися на основі попередніх помилок і даних про тестування, що значно підвищує їх ефективність.

Процес перевірки коду охоплює кілька етапів, включаючи статичний і динамічний аналіз, тестування на відповідність вимогам, перевірку на безпеку та ефективність. Статичний аналіз зазвичай виконується без фактичного виконання коду, що дозволяє виявляти синтаксичні помилки, порушення стандартів та деякі логічні помилки. Динамічний аналіз, у свою чергу, включає перевірку коду під час його виконання, що дозволяє знаходити помилки, які можуть виникати лише в процесі роботи програми.

В рамках цієї роботи важливим аспектом є дослідження та впровадження алгоритмів штучного інтелекту для вдосконалення цих процесів. Штучний інтелект, завдяки своїй здатності до самонавчання, дозволяє підвищити точність виявлення помилок та передбачити потенційні проблеми в коді ще до його запуску. Машинне навчання дозволяє системам, що перевіряють код, аналізувати великі обсяги даних і будувати моделі на основі попередніх тестувань, що робить процес перевірки більш точним і масштабованим.

Ключовим аспектом дослідження є те, як можна інтегрувати технології штучного інтелекту в існуючі методології перевірки коду. Використання моделей глибокого навчання для аналізу коду може значно прискорити процес перевірки та виявлення вразливостей. Крім того, штучний інтелект здатен автоматично оновлювати свої моделі на основі нових даних, що робить його більш ефективним у порівнянні з традиційними системами перевірки.

1.2 Структурні особливості процесу перевірки програмного коду

Процес перевірки коду має чітко структуровані етапи, кожен із яких грає свою роль у забезпеченні якості та ефективності коду [5]. Основні структурні елементи процесу перевірки коду включають:

1) підготовка до перевірки коду – цей етап починається після того, як розробник завершив роботу над певною частиною коду і вважає її готовою до перевірки. Перед тим як передати код на перевірку, розробник має переконатися, що він відповідає всім необхідним вимогам та стандартам. Це включає перевірку відповідності коду загальним правилам форматування, наявності достатньої кількості коментарів та наочності.

2) процес рецензування – на цьому етапі інший розробник або група розробників отримують доступ до коду для його аналізу. У перевірці можуть брати участь декілька людей, зокрема старші програмісти або спеціалісти, які мають великий досвід у певній області. Важливо, щоб у процесі перевірки дотримувались стандарти, які були узгоджені в команді чи організації, щоб уникнути суб'єктивності. Рецензент звертає увагу на декілька аспектів:

– *читабельність коду*. Чи зрозумілий код для інших членів команди? Чи достатньо коментарів для пояснення складних моментів?

– *логіка коду*. Чи правильно побудовані умови та цикли? Чи може код викликати помилки або некоректну поведінку?

– *оптимізація*. Чи можна покращити код для підвищення продуктивності? Чи немає зайвих елементів, які уповільнюють виконання коду?

– *відповідність стандартам*. Чи відповідає код прийнятим у команді стандартам кодування?

3) зворотній зв'язок – рецензент залишає коментарі до коду, виділяючи місця, які потребують покращення. Основна мета зворотного зв'язку — не тільки

виправити помилки, але й допомогти розробнику навчитися та уникати таких помилок у майбутньому;

4) виправлення коду – після отримання зворотного зв'язку розробник повинен внести зміни у свій код відповідно до рекомендацій рецензента. У деяких випадках потрібні незначні виправлення, тоді як в інших – значні зміни у логіці або структурі коду. Після внесення всіх змін розробник повторно передає код на перевірку;

5) остаточне затвердження – після того як усі зауваження враховані, рецензент або група рецензентів остаточно затверджують код. Це означає, що код готовий до інтеграції у загальну систему або продукт і може бути переданий на наступні етапи розробки, такі як тестування або випуск (реліз).

1.3 Функціональні особливості процесу перевірки програмного коду

Основними функціональними особливостями перевірки коду є:

– виявлення помилок та багів – однією з ключових функцій перевірки коду є виявлення помилок ще на етапі розробки. Це включає як синтаксичні, так і логічні помилки, які можуть вплинути на правильність виконання програмного забезпечення. Оскільки помилки в коді можуть бути важкими для виявлення на пізніших етапах (після інтеграції або в тестуванні), перевірка коду дозволяє їх вчасно ідентифікувати;

– покращення якості коду – перевірка коду допомагає підвищити загальну якість коду за рахунок забезпечення відповідності найкращим практикам розробки. Це стосується не лише технічних аспектів, але й читабельності, структурованості та логічної послідовності коду. Хороший код є не лише функціональним, але й зрозумілим для інших членів команди, що полегшує його підтримку та подальше вдосконалення [6];

– забезпечення відповідності стандартам – у багатьох командах і організаціях існують певні стандарти кодування, яких мають дотримуватися всі

розробники. Це можуть бути внутрішні правила форматування коду, вимоги до надання коментарів або специфічні архітектурні підходи. Процес перевірки коду гарантує, що код відповідає цим стандартам, що допомагає підтримувати єдиний стиль написання та спрощує співпрацю між розробниками;

– навчання та обмін знаннями – перевірка коду є також важливим навчальним інструментом, особливо для молодих розробників. Вони отримують зворотний зв'язок від досвідчених колег, що допомагає їм вчитися на власних помилках і вдосконалювати свої навички. Більш досвідчені фахівці можуть також отримувати нові ідеї від своїх колег, що сприяє обміну знаннями всередині команди;

– підвищення командної ефективності – завдяки регулярним перевіркам коду, команда забезпечує швидке виявлення та усунення помилок, що сприяє загальному підвищенню продуктивності та ефективності процесу розробки. Це знижує ризики, пов'язані з виявленням критичних помилок на пізніших етапах проєкту.

1.4 Огляд і аналіз минулого та сучасного стану інформаційних технологій у даній предметній області

Сфера перевірки коду (Code Review) постійно розвивається завдяки сучасним інформаційним технологіям, які роблять цей процес більш автоматизованим, ефективним та інтегрованим у загальний цикл розробки програмного забезпечення. З розвитком практик гнучкої розробки (Agile) та безперервної інтеграції та розгортання (CI/CD), вимоги до швидкості та якості перевірки коду постійно зростають, що стимулює появу нових інструментів та підходів у даній галузі [3].

Початкові підходи до перевірки коду були дуже ручними та неструктурованими. Процес полягав у тому, що один або декілька розробників перевіряли код, написаний їх колегами, шукаючи помилки або недоліки. Це дозволяло забезпечувати певний рівень якості коду, але було дуже трудомістким та залежало від досвіду і навичок рецензентів. Виявлення та виправлення помилок

могли займати значний час, і не завжди було можливим виявити всі потенційні проблеми [7].

Перші формальні підходи до перевірки коду з'явилися в 1970-х роках у вигляді **оглядів коду** (code walkthroughs) та **оглядів з перевіркою** (code inspections) [8]. Ці процеси включали ретельний аналіз кожного рядка коду за участю всієї команди розробки, але вимагали багато часу та ресурсів.

З появою інструментів для контролю версій, таких як **CVS (Concurrent Versions System)** і пізніше **Subversion (SVN)**, стала можливою більш організована співпраця над кодом [9]. Контроль версій дозволив розробникам відстежувати зміни, бачити історію коду та робити його перевірку більш ефективною, адже було простіше порівнювати версії коду, коментувати та пропонувати покращення. Однак ці системи все ще не забезпечували повноцінного процесу перевірки коду.

Справжнім проривом у перевірці коду стала поява **Git** – розподіленої системи контролю версій, створеної Лінусом Торвальдсом у 2005 році [10]. Git дозволив створювати розгалуження (branches), інтегрувати зміни з різних джерел і, що найважливіше, забезпечив можливість легкого рецензування коду перед його злиттям у основну гілку проєкту. Git також сприяв появі платформ таких як GitHub, GitLab та Bitbucket для спільної розробки коду та перевірки змін у зручному інтерфейсі. Ці платформи надали командам ефективні інструменти для організації процесу рев'ю коду, що значно підвищило якість та продуктивність розробки.

GitHub, GitLab та Bitbucket – це найбільш популярні платформи, які зробили процес перевірки коду централізованим та інтуїтивно зрозумілим. Вони пропонують такі інструменти та функції:

– pull/merge requests – платформи Git надають механізм для створення pull requests (на GitHub) або merge requests (на GitLab і Bitbucket). Це дозволяє розробнику запропонувати зміни до коду, які інші члени команди можуть переглянути, обговорити та затвердити.

– коментування та обговорення – усі ці платформи надають можливість додавати коментарі до конкретних рядків коду або до всього pull request, що дозволяє вести обговорення та давати зворотний зв'язок у контексті запропонованих змін. Приклад pull request з коментарями наведено на рис. 1.

– автоматичні перевірки якості коду – завдяки інтеграції з іншими інструментами, Git-платформи дозволяють автоматично запускати тести, аналіз якості коду та інші перевірки при створенні pull/merge requests. Це забезпечує першу автоматизовану перевірку коду перед його рецензуванням вручну.

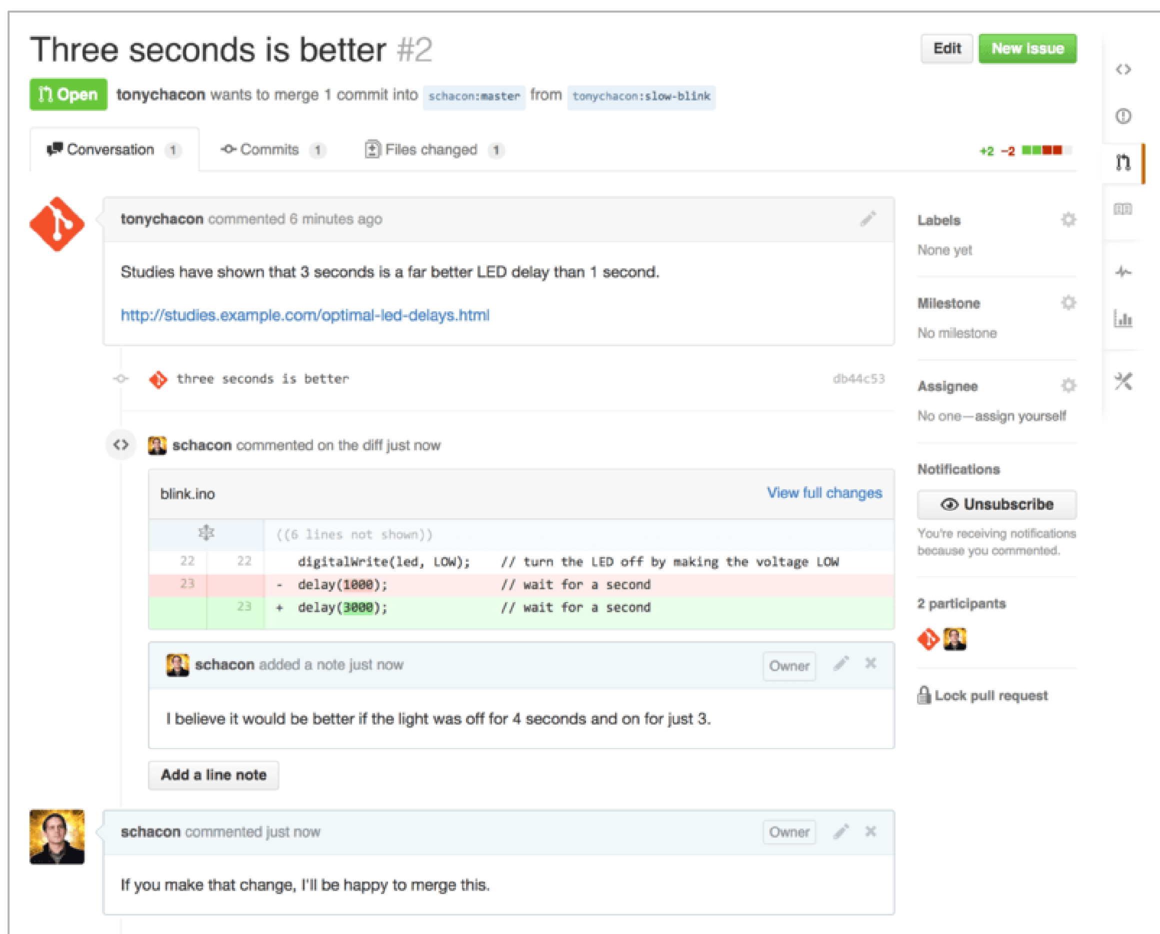


Рисунок 1.1 – Pull request з коментарями до коду

З розвитком вимог до якості програмного забезпечення з'явилися спеціалізовані інструменти для автоматизації перевірки коду, які дозволяють

проводити статичний та динамічний аналіз коду, виявляти помилки, порушення стилю та потенційні вразливості.

SonarQube – потужний інструмент для статичного аналізу коду. Він підтримує більше 25 мов програмування та дозволяє проводити комплексну перевірку коду на відповідність стандартам, безпеці та продуктивності [11]. SonarQube інтегрується з системами контролю версій та CI/CD, що дозволяє здійснювати автоматичні перевірки коду після кожного commit або pull request. Основний спосіб використання SonarQube – веб-застосунок. Інструмент має потужний веб-застосунок, за допомогою якого можна відстежувати якість коду кожної гілки репозиторію. Частину інтерфейсу веб-застосунку наведено на рис. 2. На цьому рисунку наведено статус перевірки коду, кількість проблем з ним, відсоток покриття тестами проєкту та відсоток нового дублюючого коду.

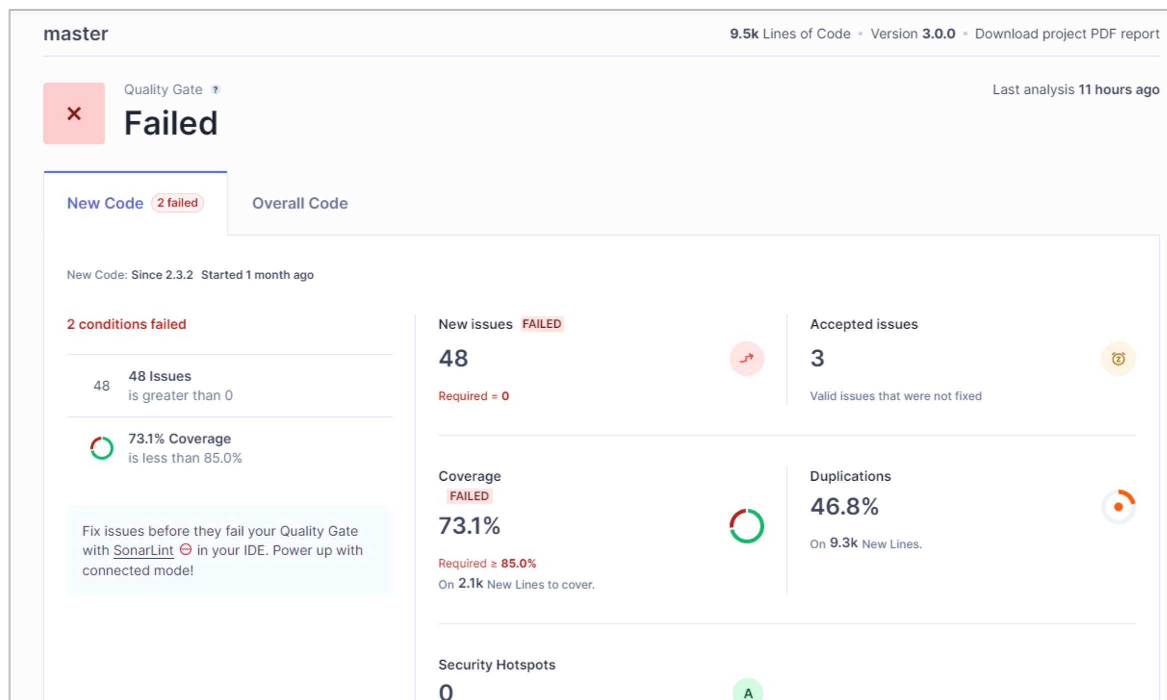


Рисунок 1.2 – Інтерфейс веб-застосунку SonarQube для репозиторію самого проєкту

Попри багатий функціонал, цей функціонал не є доступним повністю безкоштовною для звичайного користування. Обрізана версія є безкоштовною та

підійде для одного проєкту для тестового використання SonarQube. Для відкриття повного функціоналу, існує 3 види підписки на сервіс: Developer – для маленьких команд розробників, Enterprise – для великого бізнесу, Data Center – для дата центрів. Сьогодні підписка Developer коштуватиме 160$ доларів США. Для всіх інших за ціну треба домовлятись з продавцем. Також можна спробувати будь-яку підписку безкоштовно на термін пробного періоду.

Ще одним плюсом у платформи SonarQube є те, що сам проєкт є open source, його репозиторій знаходиться у відкритому доступі на Github, тобто кожний охочий може проєкт розробляти, доповнювати своїми ідеями та покращувати його.

Також є розроблений бот для Github, який результат аналізу коду SonarQube коментує саме в pull request, підкреслюючи код де є помилка або покращення коду. Це лише доступно для мови програмування Scala, так як це окремо розроблений спільнотою розробників open source плагін для SonarQube. Приклад використання SonarQube наведено на рис. 3.

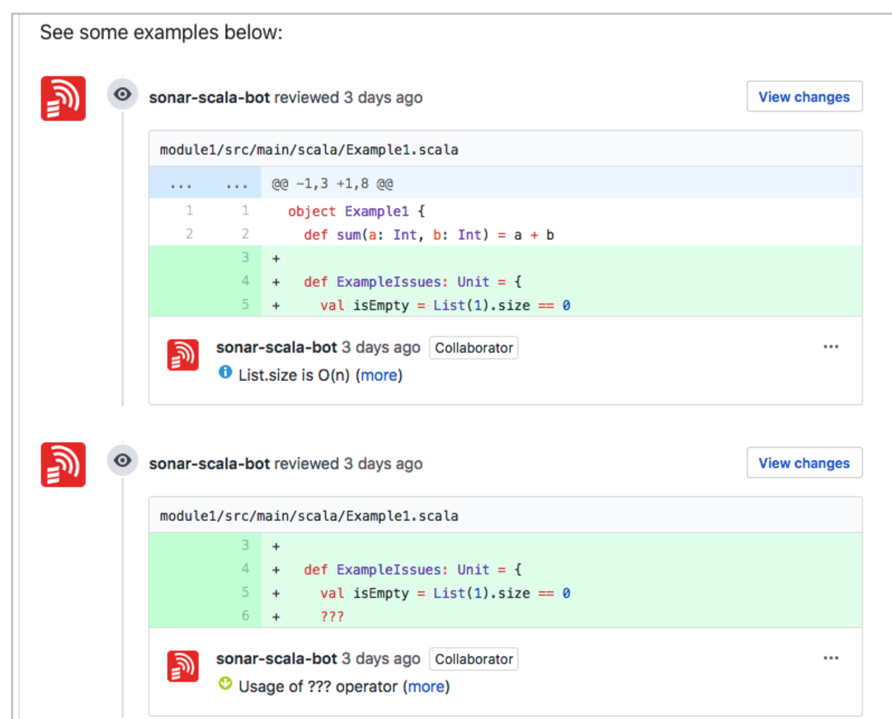


Рисунок 1.3 – Використання боту від SonarQube

Дуже схожим на SonarQube є **Codacy**. Майже теж саме, що й SonarQube, але не є open source проєктом. Також дуже корисною особливістю є те, що відразу можна легко налаштувати та використовувати бота для Github, GitLab і т.д. Взагалі Codacy дуже швидко налаштовується та легше використовувати ніж SonarQube. Codacy є хорошим вибором для команд, які віддають перевагу хмарним інструментам та шукають швидку інтеграцію з Git-системами.

Цікавим інструментом є **Danger** – гнучкий інструмент для автоматизації перевірки pull requests під час CI. Він дозволяє створювати правила для перевірки коду, забезпечує автоматичне виявлення потенційних проблем, та допомагає підтримувати стандарти якості коду в команді [12]. Але щоб цей інструмент працював як потрібно, треба вручну налаштовувати його, додаючи власні правила. Розглянемо на прикладі додавання правила, при якому будь-який pull повинен мати назву, починаючи з «TASK-...»:

```
unless github.pr_title.start_with?("TASK-")
  fail("Назва Pull Request повинна починатися з 'TASK-'.")
end
```

Це може відштовхнути розробників користуватись ним та обрати замість Danger саме SonarQube, так як SonarQube не вимагає детальних налаштувань.

Ще немало важливим плюсом Danger є те, що це програмне забезпечення open source, тобто воно безкоштовне та, маючи ліцензію The MIT License (MIT), можна використовувати його де завгодно.

Також є статичні аналізатори, які аналізують код на проблемні шаблони в коді. Одним із них є **ESLint** – статичний аналізатор для популярних мов програмування JavaScript та TypeScript. Він допомагає виявляти синтаксичні помилки, порушення стилю та інші проблеми, які можуть вплинути на якість та продуктивність коду [13]. Він дуже легко встановлюється для JavaScript проєкту. За допомогою конфігураційних файлів ESLint можна налаштовувати правила перевірки коду відповідно до вимог проєкту або команди. Але на відмінну від

SonarQube, ESLint тільки підтримує JavaScript та TypeScript, в той час, як SonarQube підтримує більше 25 мов програмування.

Взагалі майже для кожної мови програмування є статичні аналізатори, які в основному допомогають виявляти синтаксичні помилки та порушення стилю. Наприклад, для Python є Pylint, для Java – CheckStyle. Але аналізатори як правило, фокусуються на конкретних аспектах коду – це може бути певна мова програмування, стандарти кодування або безпека. Такі платформи, як SonarQube чи Codacy, зазвичай забезпечують комплексний підхід, підтримуючи аналіз багатьох мов програмування, включаючи виявлення вразливостей безпеки, технічного боргу, відповідність стандартам стилю, а також моніторинг якості коду в реальному часі.

1.5 Огляд і аналіз існуючих методів і засобів вирішення завдань КМР

Створення платформи для автоматизованої перевірки коду є складним процесом, що включає використання різних методів аналізу, технологій розробки та підходів до інтеграції з існуючими інструментами розробки. У цьому розділі буде здійснено огляд основних підходів до побудови таких платформ, проаналізовано існуючі інструменти та їх можливості, а також висвітлено обмеження та перспективи розвитку технологій автоматизованої перевірки коду.

Основним завданням платформи є забезпечення автоматичної перевірки якості коду на різних етапах його розробки. При створенні таких платформ використовуються різні підходи, які базуються на сучасних практиках розробки та автоматизації.

Більшість платформ для автоматизованої перевірки коду інтегруються зі статичними аналізаторами (ESLint, Checkstyle, Pylint) для виявлення помилок у синтаксисі, стилі та порушенні стандартів кодування без виконання коду. Динамічні аналізатори, такі як Jest, JUnit, PyTest дозволяють оцінювати поведінку коду під час його виконання. Такі платформи виконують перевірки автоматично під час створення pull request, commit або в CI/CD пайплайнах. Цей підхід ефективно

виявляє певні помилки, але обмежений в аналізі складної логіки, багаторівневої взаємодії компонентів та структурних вразливостей коду. Статичні аналізатори, які покладаються на визначені правила, можуть генерувати багато хибно позитивних або хибно негативних результатів.

Використання технологій штучного інтелекту та машинного навчання дозволяє покращити процес автоматизованої перевірки коду, забезпечуючи аналіз структури та контексту коду, а також розпізнавання патернів можливих помилок на основі історичних даних проєкту. Алгоритми машинного навчання можуть аналізувати минулі помилки та автоматично вчитися на них, пропонуючи точніші та контекстуальні перевірки. Цей підхід забезпечує глибший аналіз логіки та структури коду. Моделі ШІ можуть передбачати можливі проблеми на ранніх етапах та пропонувати оптимізації, зменшуючи технічний борг. Для навчання моделей потрібні великі обсяги даних та потужні обчислювальні ресурси. Крім того, результати таких перевірок можуть бути залежними від специфіки проєкту та необхідно враховувати можливість помилкових висновків або неточних рекомендацій.

Одною такою технологією ШІ є саме обробка природної мови (NLP) – це галузь штучного інтелекту, яка фокусується на взаємодії між комп’ютерами та людською мовою, включаючи її розуміння, інтерпретацію та генерування [14]. Хоча традиційно NLP застосовується до тексту, написаного природною мовою, його принципи також можна використовувати для аналізу програмного коду. Код має певні схожості з природною мовою: він написаний за суворими синтаксичними правилами, використовує ключові слова та структури, що несуть сенс, і може бути розділений на блоки для полегшення розуміння. Завдяки цьому методи NLP можуть бути використані для покращення процесу автоматизованої перевірки коду та виявлення складних патернів у коді, які важко розпізнати звичайними інструментами статичного аналізу. Однією з проблем при аналізі коду є відповідність між кодом і коментарями до нього, а також перевірка узгодженості

змін у коді з оновленнями документації. NLP дозволяє аналізувати зміст коментарів та документації для перевірки їх узгодженості з фактичним кодом. Методи обробки природної мови використовуються для аналізу текстових описів у коді та виявлення невідповідностей, двозначностей або пропущених деталей. Це дозволяє підвищити якість документації, зробити код більш зрозумілим для розробників, які працюють над проєктом, та забезпечити актуальність коментарів. Моделі NLP можуть використовуватися для вивчення структури коду, виявлення повторюваних патернів та ідентифікації аномалій, які відрізняються від звичних способів написання. Наприклад, вони можуть знаходити невідповідності в умовних структурах, некоректне використання циклів або неправильне використання змінних. Завдяки цьому автоматизована перевірка коду стає більш ефективною та комплексною. Використовуючи методи NLP, платформи можуть автоматично створювати коментарі та документацію на основі коду. Моделі можуть аналізувати код і генерувати текстові описи функцій, класів та методів, що дозволяє розробникам швидко отримати уявлення про структуру та логіку коду. Це також допомагає автоматизувати процес оновлення документації при внесенні змін до коду.

Але також є недоліки використання методів NLP:

1. Складність інтерпретації та контексту. Моделі можуть неправильно розуміти контекст або структуру коду, особливо якщо він складний чи містить специфічні патерни. Це може призвести до хибно позитивних результатів або недооцінки потенційних проблем у коді.

2. Велика потреба в обсязі даних для навчання. Недостатній обсяг навчальних даних може призвести до зниження ефективності та точності моделі, а також до збільшення кількості хибно позитивних або хибно негативних результатів.

3. Складність роботи з багатомовними проєктами. Використання моделі NLP для аналізу багатомовних проєктів може призвести до зниження ефективності

аналізу, адже модель може не враховувати особливості кожної з мов або неправильно інтерпретувати структуру коду.

4. Труднощі з генерацією та поясненням результатів. У випадку помилкових або некоректних рекомендацій розробники можуть витрачати додатковий час на перевірку результатів аналізу, знижуючи загальну ефективність процесу автоматизованої перевірки коду.

5. Неможливість повного розуміння бізнес-логіки. Модель може не розпізнати складні логічні помилки, які залежать від взаємодії компонентів програми або від бізнес-вимог. Відтак, критичні баги або потенційні проблеми з дизайном коду можуть залишитися непоміченими.

6. Недостатня адаптивність до змін у коді. Модель може видавати застарілі рекомендації або не враховувати нові патерни, які з'явилися в коді. Це знижує точність і корисність аналізу, а також може призвести до хибно позитивних результатів.

1.6 Обґрунтування та вибір підходів щодо виконання завдань КМР

Розраховуючи існуючи підходи до розробки платформи для автоматизованої перевірки коду, а саме модуль перевірки коду в п. 1.5, кращим рішенням є саме розробка платформи з використанням методів ШІ, а саме методів NLP. Але не можна нехтувати недоліками цих методів, а саме головне, що розробка платформи для автоматизованої перевірки коду на основі методів NLP є дуже складною та обсяговою роботою. Для створення такої платформи буде потрібно великий обсяг часу, даних, комп'ютерних потужностей. Це не можливо собі дозволити для виконання завдань КРМ. Але якщо є вже така технологія, яка може розуміти мову людини, розуміти контекст тексту, розуміти код та навчатися на помилках та у вільному доступі для використання? Відносно недавно стався інформаційний «бум» в сфері ШІ – поява **ChatGPT**.

ChatGPT, розроблений компанією OpenAI, побудований на архітектурі трансформерів, які здатні обробляти великі обсяги тексту, розуміти складні запити та генерувати релевантні відповіді [15]. Модель може працювати з текстом природної мови та кодом, що дозволяє використовувати її для виявлення проблем у структурі програмного забезпечення, генерації рекомендацій для покращення, а також надання пояснень щодо знайдених помилок чи патернів. Основні переваги ChatGPT – можливість працювати з різними мовами програмування, обробляти контекст коду та виявляти складні логічні помилки. Це робить модель потужним інструментом для автоматизованої перевірки коду.

Сучасні інструменти автоматизованої перевірки коду забезпечують базовий статичний аналіз коду, виявляючи синтаксичні помилки, порушення стилю кодування та потенційні баги. Проте їхні можливості обмежені відсутністю розуміння контексту коду та його бізнес-логіки, що призводить до великої кількості хибно позитивних або хибно негативних результатів. Використання ChatGPT надає можливість вирішити ці проблеми за рахунок його здатності аналізувати не лише синтаксис, але й семантику та контекст коду, що сприяє точнішому виявленню проблем. Це дозволяє надати розробнику рекомендації щодо виправлення з урахуванням специфіки проєкту та особливостей мови програмування. Модель здатна працювати з великими фрагментами коду, розуміти їх структуру, семантику та логіку. Це дозволяє моделі не лише знаходити помилки, але й пояснювати їх значення та потенційний вплив на програму. Враховуючи здатність ChatGPT генерувати осмислені тексти, розробники можуть отримувати більш якісний зворотний зв'язок, що допомагає у виправленні помилок та підвищенні продуктивності роботи. Також модель може навчатися на основі отриманих результатів аналізу, що дозволяє покращувати точність перевірки коду з часом.

Інтеграція платформи з ChatGPT є доволі простою. ChatGPT має власний API, до якого можна надсилати запити з текстовими фрагментами коду та коментарями розробників. Платформа для перевірки коду буде направляти запити до ChatGPT

API, включаючи фрагменти коду, які потрібно проаналізувати, контекст (як-от ціль коду або причину змін), а також питання, які можуть допомогти ChatGPT зрозуміти, що саме потрібно перевірити. Багато мов програмувань мають власні бібліотеки, які реалізують взаємодію з ChatGPT API у більш комфортному вигляді для розробника, у вигляді наборів класів та методів.

Хоча ChatGPT є потужним інструментом для аналізу тексту та програмного коду, його використання в автоматизованій перевірці коду має певні обмеження та недоліки. Нижче наведено основні з них:

– відсутність глибокого розуміння бізнес-логіки та контексту проєкту. ChatGPT здатний обробляти текст і навіть виявляти деякі логічні помилки в коді, але йому важко зрозуміти глибокий контекст та бізнес-логіку проєкту. Це обмежує можливість аналізу складних взаємозв'язків між компонентами коду та розуміння того, як вони узгоджуються із загальними цілями проєкту. Як наслідок, модель може видавати рекомендації, які не відповідають вимогам конкретного проєкту.

– висока ймовірність хибних результатів. Оскільки ChatGPT базується на моделях обробки природної мови та коду, іноді він може неправильно інтерпретувати фрагменти коду та контекст, у якому вони використовуються. Це призводить до хибно позитивів (коли модель ідентифікує проблему, якої насправді немає) та хибно негативів (коли модель не бачить реальної проблеми). Це може ускладнювати процес code review та знижувати довіру до результатів аналізу.

– неможливість перевірки продуктивності та виконання коду. ChatGPT виконує статичний аналіз коду без його реального виконання. Модель не може протестувати продуктивність коду, перевірити, як він поводиться в різних середовищах виконання або виявити потенційні проблеми з ресурсами (наприклад, витоки пам'яті). Це обмежує можливості виявлення багів, які проявляються лише під час роботи програми.

– залежність від правильного формулювання запитів. Ефективність використання ChatGPT значною мірою залежить від того, як розробники

формулюють запити. Якщо питання нечітке або не містить достатньо контексту, модель може надати некоректну або неповну відповідь. Це вимагає від користувача певних навичок у взаємодії з моделлю та вміння правильно формулювати проблеми або запити.

– ресурсомісткість та вартість. Використання ChatGPT для аналізу коду може бути ресурсомістким з точки зору обчислювальних потужностей, особливо для великих проєктів із великою кодовою базою. Це може призвести до додаткових витрат на використання інфраструктури, а також на підписку або доступ до API ChatGPT, які можуть бути високими при інтенсивному використанні.

Також у ChatGPT є різні моделі. Найновіша актуальна модель ChatGPT є GPT-4. GPT-4 є найновішою та найпотужнішою моделлю, яка значно перевершує попередні версії за якістю та точністю. GPT-4 краще розуміє контекст, здатна виявляти складні вразливості та надавати більш детальні та точні рекомендації щодо покращення коду. GPT-4 рекомендується для використання в платформі автоматизованої перевірки коду, якщо пріоритетом є висока якість аналізу та точність результатів. GPT-4 здатна глибоко аналізувати код, виявляти складні помилки та надавати розгорнуті рекомендації. Однак варто врахувати, що вартість використання GPT-4 є вищою, і можуть існувати обмеження щодо доступу до цієї моделі. GPT-3.5 може бути вибором у випадку обмеженого бюджету або якщо необхідна швидша обробка запитів. GPT-3.5 забезпечує хороший рівень аналізу та може впоратися з більшістю стандартних завдань перевірки коду. Але платформою для КМР не буде навантажена тисячами запитами до ChatGPT, тому модель GPT-4 була використана для розробки платформи для автоматизованої перевірки коду КМР. Також є модель o1, але вона зараз доступна лише на спробу і поки що не є основною моделлю, але у майбутньому це буде найпотужніша модель ChatGPT, так як основне покращення цієї моделі є саме можливості міркування як у людини.

1.7 Специфікації вимог до програмного забезпечення

1 ПРИЗНАЧЕННЯ ТА МЕЖІ ПРОЄКТУ

1.1 Призначення системи (застосунку), для якої розробляється програмне забезпечення

Система автоматизованої перевірки коду призначена для інтерактивної допомоги розробникам у виявленні та виправленні помилок, покращенні якості коду та підвищенні його безпеки. Вона інтегрується з популярною системами контролю версій GitHub та забезпечує автоматичний аналіз pull requests або commits. В основі системи лежить модель ChatGPT, яка надає рекомендації в реальному часі щодо стилю, продуктивності, структури та безпеки коду.

1.2 Погодження, що ухвалені в програмній документації

Система повинна відповідати вимогам сумісності з платформою GitHub та підтримувати інтеграцію із сучасними мовами програмування.

1.3 Межі проєкту ПЗ

Система охоплює процес аналізу та перевірки коду з моменту створення commit або pull request до отримання звіту про аналіз із рекомендаціями.

2 ЗАГАЛЬНИЙ ОПИС

2.1 Сфера застосування

Система застосовується у процесі розробки програмного забезпечення для автоматизованої перевірки коду, покращення його якості, відповідності стандартам, продуктивності та безпеки. Основна мета – інтеграція в процес контролю версій та забезпечення швидкого та релевантного зворотного зв’язку для розробників.

2.2 Характеристики користувачів

– розробники: використовують систему для перевірки власного коду або коду команди перед його злиттям у головну гілку проєкту;

– технічні лідери та архітектори: контролюють якість коду в команді, відслідковують технічний борг та відповідають за стандарти стилю програмування;

– DevOps інженери: інтегрують систему в CI/CD пайплайни для автоматичного аналізу коду на кожному етапі розгортання.

2.3 Загальна структура і склад системи:

– модуль інтеграції з системами контролю версій GitHub, який відстежує зміни у репозиторії та викликає аналіз коду;

– модуль аналізу коду з використанням ChatGPT, що приймає фрагменти коду, аналізує їх та надає рекомендації;

– модуль генерації звітів, який формує звіти з результатами аналізу та пропонує виправлення або оптимізації;

– модуль взаємодії з користувачем, який надає інтерфейс для перегляду результатів аналізу та спілкування з ChatGPT.

2.4 Загальні обмеження

– система не виконує динамічного аналізу коду (аналіз під час виконання);

– не охоплює повний спектр тестування коду (наприклад, юніт-тестування), лише перевірку на відповідність стандартам та виявлення потенційних проблем.

3 ФУНКЦІЇ СИСТЕМИ

3.1 Функція інтеграції з Git-платформами

3.1.1 Опис функції: відстеження commits, pull requests та виклик аналізу коду при кожному оновленні pull request.

3.1.2 Вхідна і вихідна інформація: вхідна – створенні pull requests, фрагменти коду; вихідна – запит на аналіз коду, звіти.

3.1.3 Функціональні вимоги: підтримка основних платформи GitHub, наявність API для інтеграції.

3.2 Функція аналізу коду

3.2.1 Опис функції: виконання статичного аналізу коду з використанням ChatGPT, виявлення синтаксичних помилок, відхилень від стандартів стилю, потенційних вразливостей.

3.2.2 Вхідна і вихідна інформація: вхідна – фрагменти коду; вихідна – рекомендації, звіти про помилки, можливі оптимізації та покращення коду.

3.2.3 Функціональні вимоги: здатність працювати з основними мовами програмування, контекстний аналіз коду, інтерактивне спілкування для пояснення рекомендацій.

3.3 Функція генерації звітів

3.3.1 Опис функції: формування звіту про аналіз коду з переліком виявлених помилок, рекомендаціями та пропозиціями щодо покращення.

3.3.2 Вхідна і вихідна інформація: вхідна – результати аналізу; вихідна – структурований звіт із описом знайдених проблем.

3.3.3 Функціональні вимоги: можливість отримання звітів на веб-застосунку платформи, відправлення звітів у вигляді коментарів до pull request платформи GitHub.

4 ВИМОГИ ДО ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Джерела і зміст вхідної інформації (даних)

Фрагменти коду, коментарі до комітів, метадані pull requests.

4.2 Нормативно-довідкова інформація

Набори правил стилю для різних мов програмування (ESLint для JavaScript, Pylint для Python тощо), стандарти безпеки (OWASP).

4.3 Вимоги до способів організації, збереження та ведення інформації

Збереження результатів аналізу та звітів у захищеній базі даних, доступність історії аналізу для відстеження змін.

5 ВИМОГИ ДО ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

– працездатність серверної інфраструктури компанії OpenAI для роботи моделі ChatGPT-4;

– працездатність серверної інфраструктури компанії Microsoft для роботи Webhook платформи GitHub.

6 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

6.1 Архітектура програмної системи

Мікросервісна архітектура, де кожен модуль виконує окрему задачу (інтеграція, аналіз, формування звітів).

6.2 Системне програмне забезпечення

ОС Windows або Windows Server, Linux дістрібутиви.

6.3 Мережне програмне забезпечення

HTTP/HTTPS для зв’язку з GitHub, WebSocket для зворотного зв’язку реального часу.

6.4 Програмне забезпечення ведення інформаційної бази

Реляційна база даних PostgreSQL для зберігання звітів.

6.5 Мова і технологія розробки ПЗ

Мікросервіси – ASP.NET застосунки на платформі .NET 8 з використанням мови програмування C#, Angular та TypeScript для розробки веб-застосунку для платформи автоматизованої перевірки коду.

7 ВИМОГИ ДО ЗОВНІШНІХ ІНТЕРФЕЙСІВ

7.1 Інтерфейс користувача

Веб-додаток з інтерфейсом для перегляду звітів, результатів аналізу та спілкування з ChatGPT.

7.2 Апаратний інтерфейс

Не потребує спеціального апаратного забезпечення на стороні клієнта.

7.3 Програмний інтерфейс

RESTful API для виклику аналізу та отримання результатів, API для інтеграції з платформою GitHub.

7.4 Комунікаційний протокол

HTTP/HTTPS для передачі даних між мікросервісами, WebSocket для інтерактивного взаємодії між користувачем та модулем аналізу коду.

8 ВИМОГИ ДО ЗОВНІШНІХ ІНТЕРФЕЙСІВ

8.1 Доступність

Система повинна бути доступною тільки тоді, колі її локально встановили на машину, повина працювати без помилок та із мінімальними затримками у відповіді.

8.2 Супроводжуваність

Можливість легкої модифікації та оновлення компонентів системи за рахунок мікросервісної архітектури.

8.3 Переносимість

Застосунки на .NET 8 та Angular є кросплатформними, отже, платформа для автоматичної перевірки коду може працювати на таких ОС, як Windows, Linux-дістрибутивах та MacOS.

8.4 Продуктивність

Швидка обробка запитів на аналіз коду та надання рекомендацій у реальному часі.

8.5 Надійність

Стійкість до помилок, забезпечення збереження результатів аналізу.

8.6 Безпека

Згенеровані рекомендації забезпечують належний рівень безпеки.

Висновки до розділу 1

У розділі 1 проведено огляд та аналіз сучасного стану технологій у сфері автоматизованої перевірки коду, методів та інструментів, що застосовуються для поліпшення якості коду та підвищення ефективності розробників. Встановлено, що на ринку існує велика кількість інструментів для статичного аналізу коду, які забезпечують автоматичну перевірку на наявність помилок, потенційних вразливостей, відхилень від стандартів стилю та інших проблем. Однак, більшість з них має обмеження, оскільки вони працюють лише на основі заздалегідь визначених правил і часто не враховують контекст розробки.

Огляд інструментів, таких як SonarQube, Codacy та Danger, показав, що вони активно застосовуються для перевірки коду, але мають певні недоліки, зокрема,

обмежену можливість розуміння контексту змін у коді та проблем із логікою, що вимагає більш інтелектуального підходу. Використання штучного інтелекту, зокрема моделей обробки природної мови (NLP), відкриває нові можливості для аналізу коду, дозволяючи виявляти більш складні помилки та надавати рекомендації, що враховують контекст коду.

Розглянуто можливості інтеграції платформи автоматизованої перевірки коду з моделлю ChatGPT, яка здатна аналізувати код на основі його семантики та надавати рекомендації, зважаючи на контекст і логіку виконання програм. Показано, що використання ChatGPT дозволяє значно розширити можливості статичного аналізу коду, забезпечуючи кращу взаємодію з розробниками та можливість надавати більш релевантні рекомендації.

Висвітлено недоліки існуючих методів аналізу коду на основі статичних аналізаторів, порівняно з використанням платформ на основі штучного інтелекту. Сформульовано завдання, які мають бути вирішені в межах кваліфікаційної роботи, а також обґрунтовано вибір технологій для створення системи автоматизованої перевірки коду, яка буде інтегрована з моделлю ChatGPT.

2 МОДЕЛЮВАННЯ ПРОЦЕСУ ТА МЕТОДІВ ПЕРЕВІРКИ КОДУ. ФУНКЦІОНАЛЬНІ ТА ІНФОРМАЦІЙНІ МОДЕЛІ

2.1 Вибір та обґрунтування підходів до моделювання

Моделювання є важливим етапом розробки системи автоматизованої перевірки коду, оскільки воно дозволяє візуалізувати, зрозуміти та оптимізувати функціональні процеси, інформаційні потоки та взаємодію між компонентами системи. Основна мета моделювання — створити концептуальні та практичні моделі, які відображають реальний процес перевірки коду з використанням ChatGPT, і визначити оптимальний спосіб впровадження системи в робочий процес розробників. Ці моделі сприяють розумінню функціональних та інформаційних залежностей системи, дозволяють ідентифікувати потенційні проблеми та забезпечують структуроване планування розробки.

Для ефективного проєктування та розробки системи автоматизованої перевірки коду необхідно використовувати інструменти моделювання, які дозволяють відобразити всі функціональні та інформаційні процеси системи, розробити детальні моделі інформаційних потоків та забезпечити коректну взаємодію компонентів системи. Важливим завданням є вибір оптимальних інструментів моделювання, які відповідають вимогам до системи та можуть ефективно представити всі її аспекти. Нижче розглянуто різні типи інструментів та підходів, що можуть бути використані для створення моделей:

– **IDEF0 (Integration Definition for Function Modeling)** − це методологія моделювання, що використовується для аналізу та опису функцій системи або процесу [16]. IDEF0 використовує графічну нотацію для представлення функцій, вхідних та вихідних даних, контрольних механізмів та інших елементів системи. Це допомагає зрозуміти та вдосконалювати функціональні аспекти, виявляти проблеми та покращувати ефективність. IDEF0 є потужним інструментом для аналізу бізнес-процесів, розробки системних архітектур та вдосконалення організаційних процесів. Перевагами є легко зрозумілий графічний формат, що дозволяє

моделювати високорівневі функції системи та їх взаємодію. Забезпечує ієрархічну декомпозицію функцій для кращого розуміння роботи системи. Недоліками такої методології є обмежена деталізація для моделювання процесів низького рівня та не підходить для моделювання складних алгоритмів або процесів із великою кількістю подій;

– **IDEF3 (Process Description Capture Method)** – це методологія для опису процесів у системі, яка фокусується на візуалізації послідовності виконання функцій та подій, що їх викликають. Перевагами є те, що ця методологія дозволяє детально описати процеси, їхні зв'язки та умови виконання. IDEF3 підходить для відображення складних подій та процесів. Недоліками є високий рівень деталізації, що в свою чергу може ускладнювати розуміння процесу при великій кількості елементів або умов;

– **DFD-діаграми (Data Flow Diagrams)** – діаграми потоків даних, які використовуються для моделювання інформаційних потоків між компонентами системи, відображення процесів обробки та передачі даних. DFD дозволяє відобразити, як інформація пересувається між модулями системи та яким чином вона змінюється під час обробки. Ці діаграми чітко відображають інформаційні потоки, їх джерела та процеси обробки, що сприяє розумінню структури даних у системі. Легко інтегрується з функціональними моделями IDEF0. Недоліком є те, що DFD-діаграми не відображають деталей алгоритмів або логічних розгалужень. Основний фокус – передача та трансформація даних;

– **блок-схеми** використовуються для побудови алгоритмів виконання функцій та опису логіки роботи кожного модуля системи. Блок-схеми зрозумілі для опису покрокового алгоритму виконання конкретної функції. Недоліками блок-схем є те, що жля великих алгоритмів блок-схеми можуть стати складними та важкими для розуміння;

– **UML діаграми (Unified Modeling Language)** включає різні типи діаграм, такі як діаграми активності (Activity Diagrams), діаграми класів (Class Diagrams), які

дозволяють моделювати поведінку та структуру системи на різних рівнях. UML діаграми надають широкий спектр нотацій для моделювання різних аспектів системи, включаючи класову структуру, взаємодію об'єктів та послідовність виконання дій. Недоліками UML діаграм є те, що вони потребують спеціальних навичок для використання та розуміння нотації;

– **ER-діаграми (Entity-Relationship)** використовуються для відображення структури даних, які зберігаються та обробляються у системі. Вони показують, як різні об’єкти даних пов'язані між собою, та дозволяють оптимізувати структуру бази даних. Перевагами є чітке представлення структури даних, їх властивостей та зв’язків та допомагає зрозуміти логічну організацію бази даних. Недоліками є лише те, що ER-діаграми фокусуються лише на даних, не враховуючи поведінку системи або алгоритми обробки.

Отже, маючи широкий вибір інструментів для моделювання функціональних та інформаційних аспектів системи, конкретний інструмент буде використаний залежно від мети та особливостей моделювання певної частини системи.

2.2 Функціональна модель платформи для автоматизованої перевірки коду

Система автоматизованої перевірки коду, інтегрована з ChatGPT, виконує ряд ключових функцій, спрямованих на забезпечення якісного аналізу коду, надання рекомендацій розробникам, підвищення продуктивності та безпеки розроблюваного програмного забезпечення. Ці функції охоплюють весь цикл перевірки коду – від моменту створення pull request або commit до них до надання зворотного зв'язку розробникам у зручному форматі. Далі розглянуто основні функції системи та їх значення у контексті роботи системи.

Інтеграція з GitHub. Ця функція забезпечує зв’язок між системою автоматизованої перевірки коду та платформами контролю версій GitHub. Вона

відслідковує зміни у репозиторії, включаючи pull request та commit до них, та автоматично запускає процес аналізу коду.

Збір та підготовка даних для аналізу. Ця функція відповідає за отримання та підготовку коду для подальшого аналізу. Це включає збір фрагментів коду, метаданих про коміти, гілки та Pull/Merge Requests. Також враховуються контекстні дані, такі як зміни в порівнянні з попередньою версією коду, коментарі розробників та ціль змін (наприклад, виправлення багу або додавання нової функціональності).

Аналіз коду з використанням ChatGPT. Основна функція системи, яка забезпечує перевірку коду за допомогою технологій штучного інтелекту. Після отримання фрагментів коду, система виконує статичний аналіз за допомогою моделі ChatGPT, виявляючи синтаксичні помилки, недоліки в стилі кодування, можливі вразливості та проблеми продуктивності. Аналіз включає перевірку відповідності коду стандартам, пошук дублювань, виявлення некоректних або потенційно небезпечних практик програмування.

Формування рекомендацій та пропозицій щодо виправлень. На основі результатів аналізу коду система генерує рекомендації для розробників. Це можуть бути поради щодо виправлення синтаксичних помилок, пропозиції щодо оптимізації коду або рекомендації з покращення безпеки. Рекомендації формуються у зручному для розробників форматі, із зазначенням конкретних рядків коду та детальним поясненням проблеми.

Генерація звітів про результати аналізу. Система автоматично формує звіти про результати перевірки коду, які можуть включати загальний аналіз якості коду, статистику виявлених помилок та їх серйозність, рекомендації з виправлення та покращення. Звіти можуть бути представлені у вигляді коментарів до pull request або через інтерфейс платформи.

Взаємодія з користувачами та інтерактивна підтримка. Система надає можливість розробникам вести діалог з ChatGPT для уточнення рекомендацій або обговорення можливих виправлень. Це дозволяє отримувати більше деталей щодо

знайдених проблем, а також розглядати альтернативні рішення або оптимізації для коду.

Збереження та відстеження результатів аналізу. Ця функція включає збереження історії результатів перевірки коду, виявлених помилок та їх виправлень. Всі результати аналізу зберігаються у базі даних, що дозволяє відстежувати прогрес команди, контролювати виконання рекомендацій та підтримувати історію покращень коду.

На рис. 2.1 наведено контекстну діаграму IDEF0 0-го рівня.

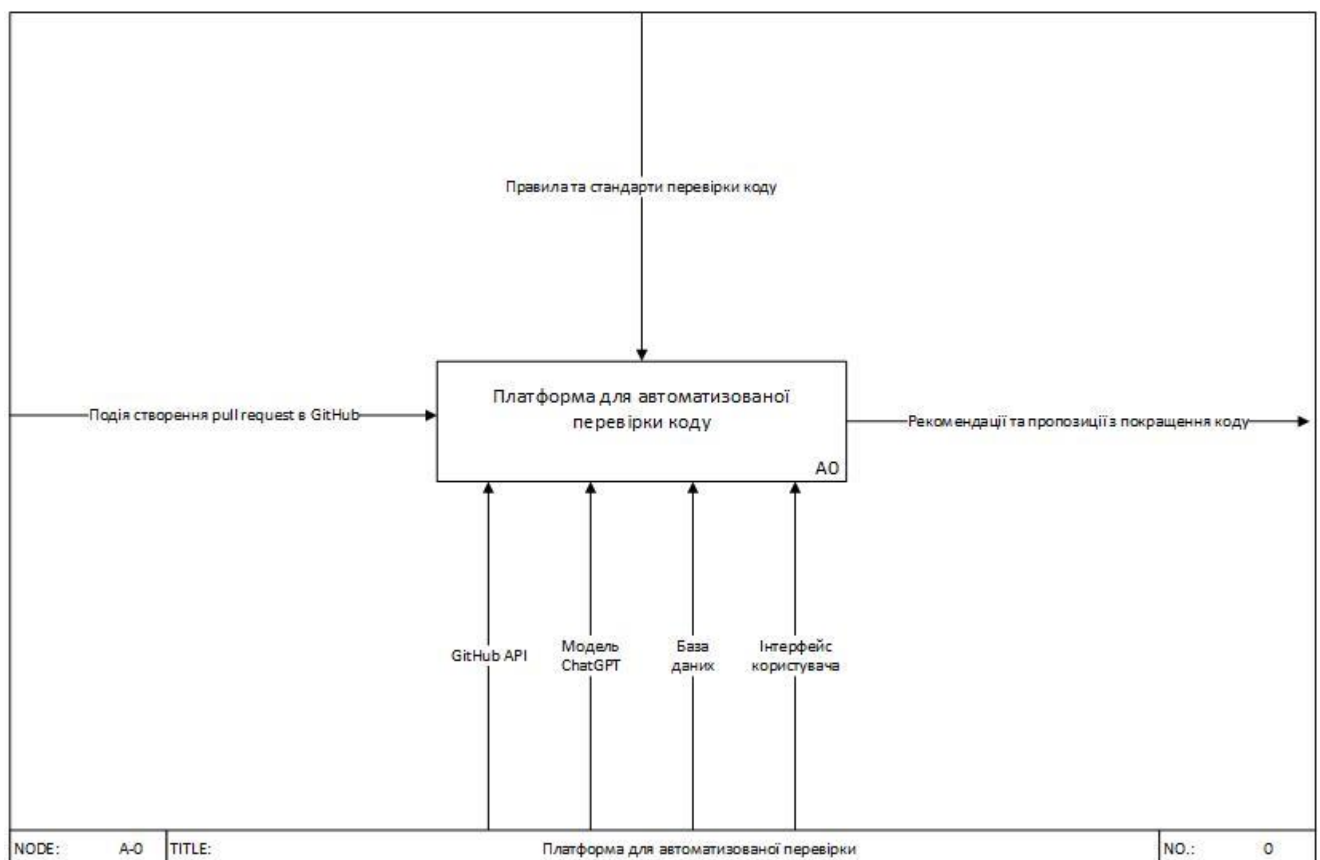


Рисунок 2.1 – Контекстна діаграма IDEF0 A-0

Як можна побачити на рисунку, система має 1 вхід, 1 вихід, 4 механізма та 1 контроль. На вході система приймає подію створення pull request, після чого система починаю свою роботу. На виході система саме генерує рекомендації та пропозиції щодо покращення коду. Механізмами є GitHub API, що включає

webhook для отримання інформації про створення pull request, модель ChatGPT, яка буде аналізувати код, база даних платформи, яка буде зберігати результати перевірки коду та інтерфейс користувача, за допомогою якого можна взаємодіяти з платформою. Контролем є правила та стандарти перевірки коду, яких слід дотримуватись для написання якісного коду.

На рис. 2.2 наведено діаграму IDEF0 1-го рівня для платформи для автоматизованої перевірки коду.

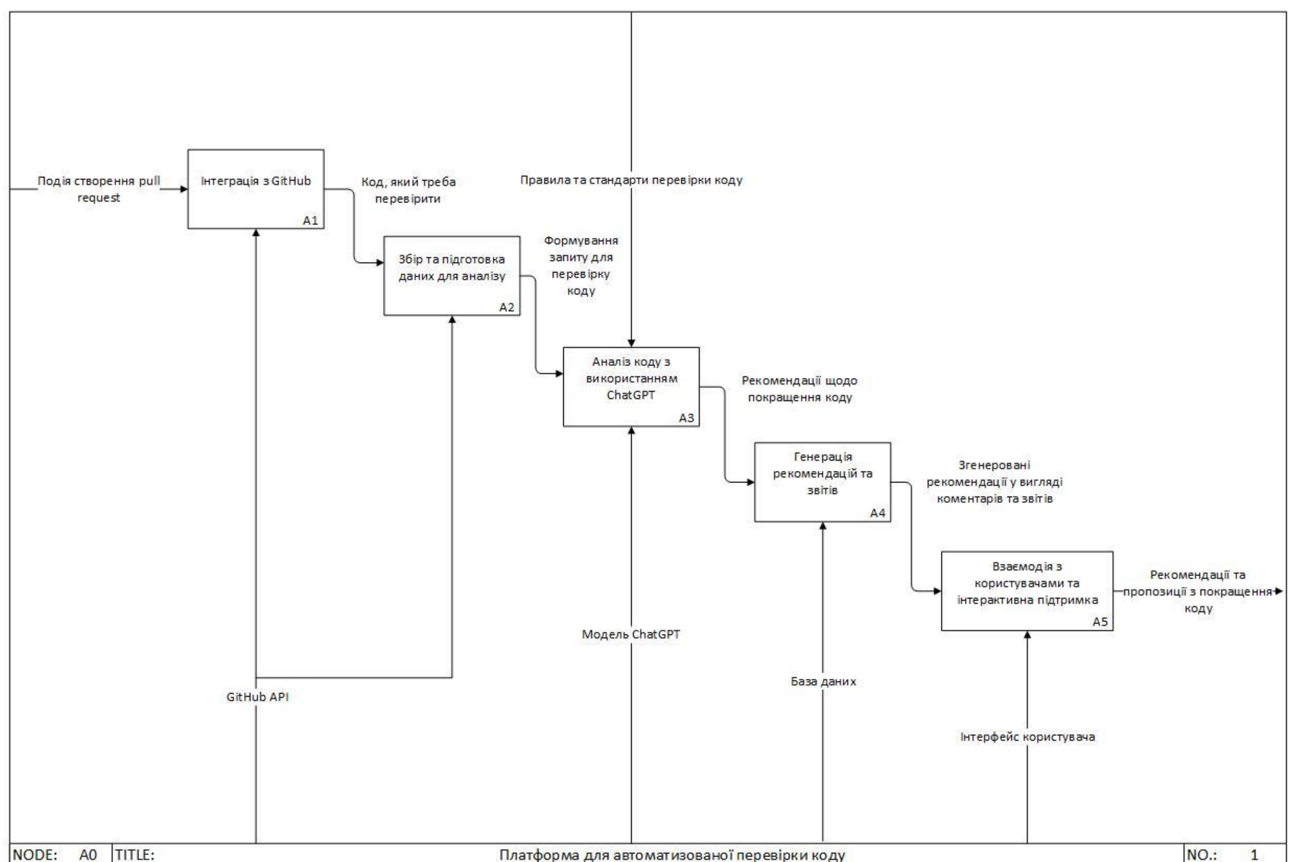


Рисунок 2.2 – Діаграма IDEF0 A0

На рис. 2.3 наведено діаграму IDEF0 2-го рівня для функції «Інтеграція з GitHub».

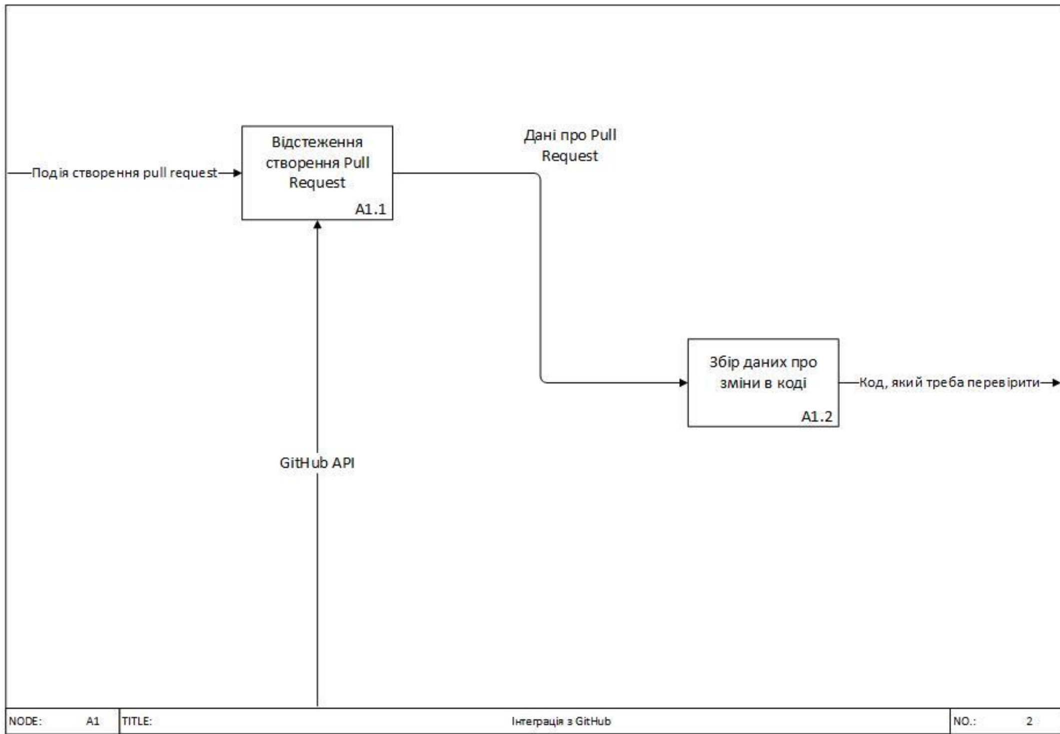


Рисунок 2.3 – Діаграма IDEF0 A1 «Інтеграція з GitHub»

На рис. 2.4 наведено діаграму IDEF0 2-го рівня для функції «Збір та подготовка даних для аналізу».

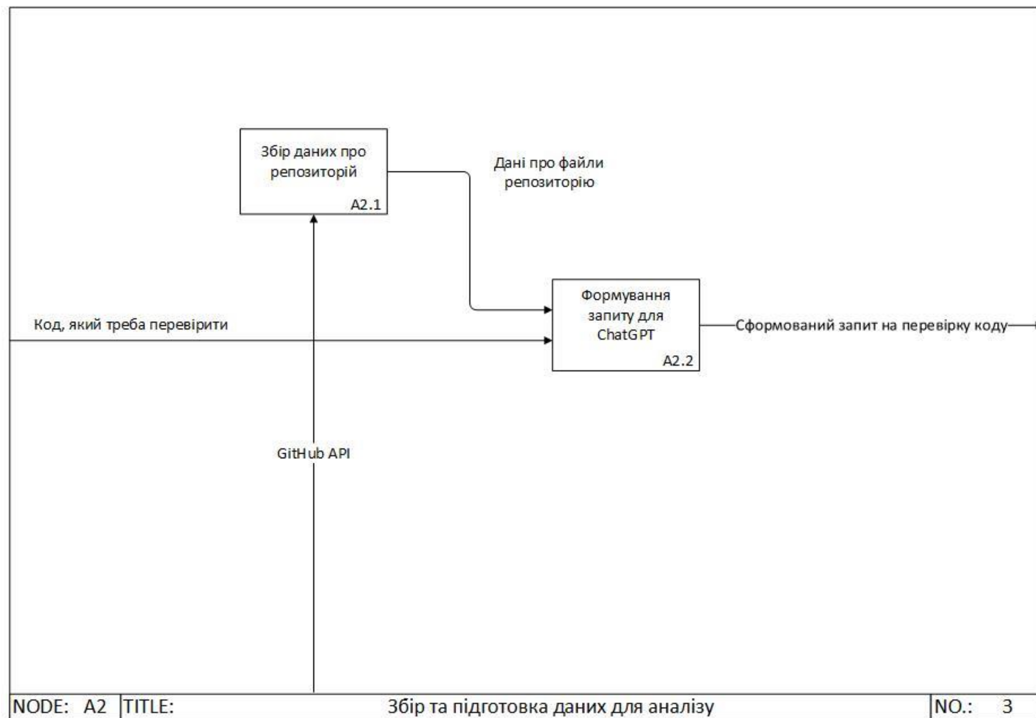


Рисунок 2.4 – Діаграма IDEF0 A2 «Збір та подготовка даних для аналізу»

На рис. 2.5 наведено діаграму IDEF0 2-го рівня для функції «Аналіз коду з використанням ChatGPT».

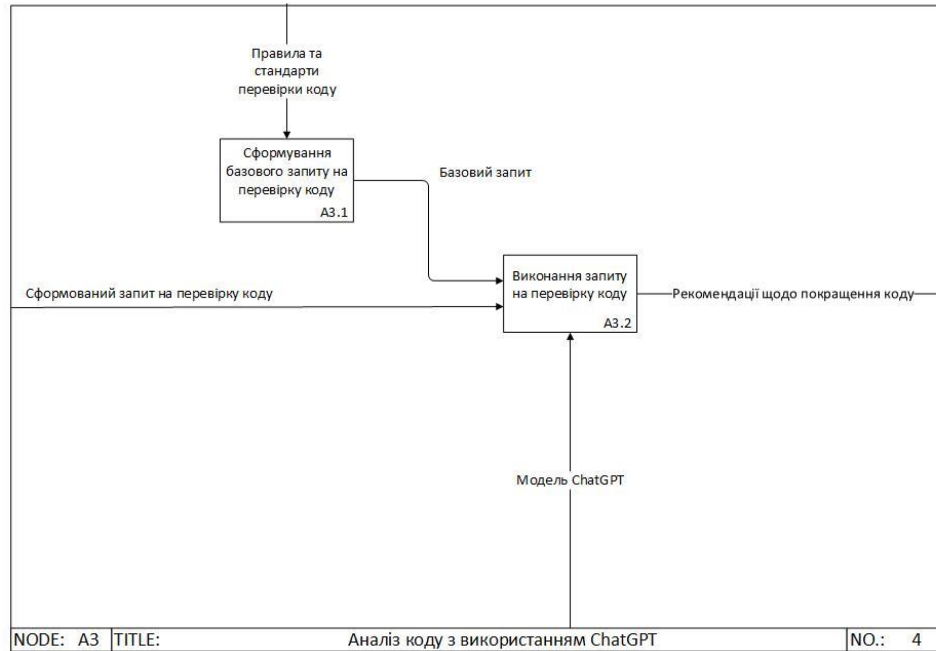


Рисунок 2.5 – Діаграма IDEF0 А3 «Аналіз коду з використанням ChatGPT»

На рис. 2.6 наведено діаграму IDEF0 2-го рівня для функції «Генерація рекомендацій та звітів».

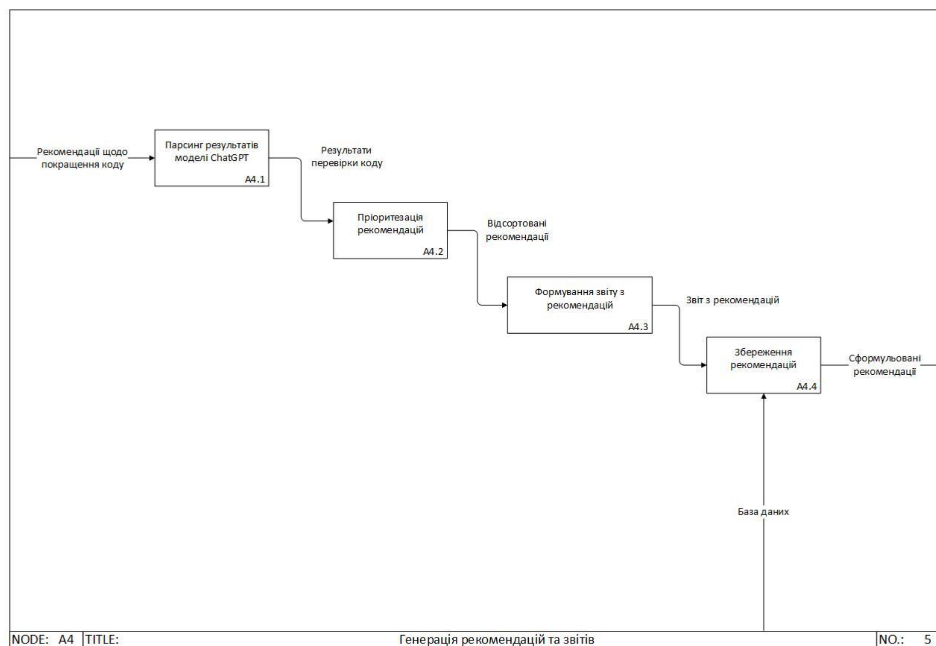


Рисунок 2.6 – Діаграма IDEF0 А4 «Генерація рекомендацій та звітів»

На рис. 2.7 наведено діаграму IDEF0 2-го рівня для функції «Взаємодія з користувачами та інтерактивна підтримка».

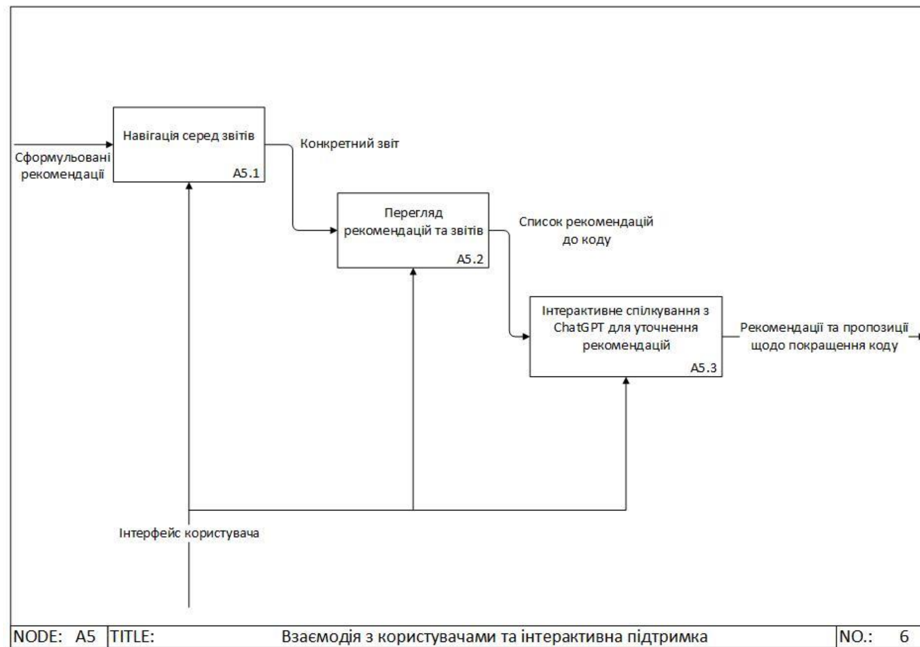


Рисунок 2.7 – Діаграма IDEF0 A5 «Взаємодія з користувачами та інтерактивна підтримка»

Створені діаграми IDEF визначають функціональні аспекти платформи автоматизованої перевірки коду, детально описуюсь процеси, їхні зв'язки та умови виконання.

2.3 Інформаційна модель платформи для автоматизованої перевірки коду

Для відображення потоку даних між процесами побудовано DFD діаграми. Спочатку на рис. 2.8 побудовано контекстну DFD діаграму всієї платформи для автоматизованої перевірки коду.

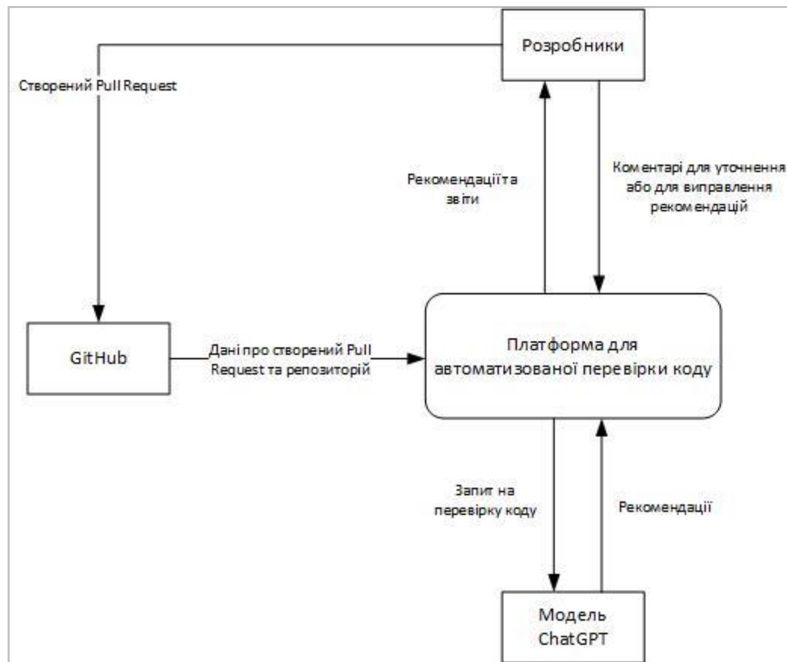


Рисунок 2.8 – Контекстна діаграма DFD

На рис. 2.9 було зроблено декомпозицію контекстної діаграми для того, щоб отримати діаграму DFD 1-го рівня. Діаграма виявилась доволі ємкою з багатьма процесами. Для більш детального аналізу потоків даних зроблено діаграми рівнем нижче для найважливіших процесів платформи для автоматизованої перевірки коду. Вони наведені на рисунках 2.10 та 2.11 відповідно.

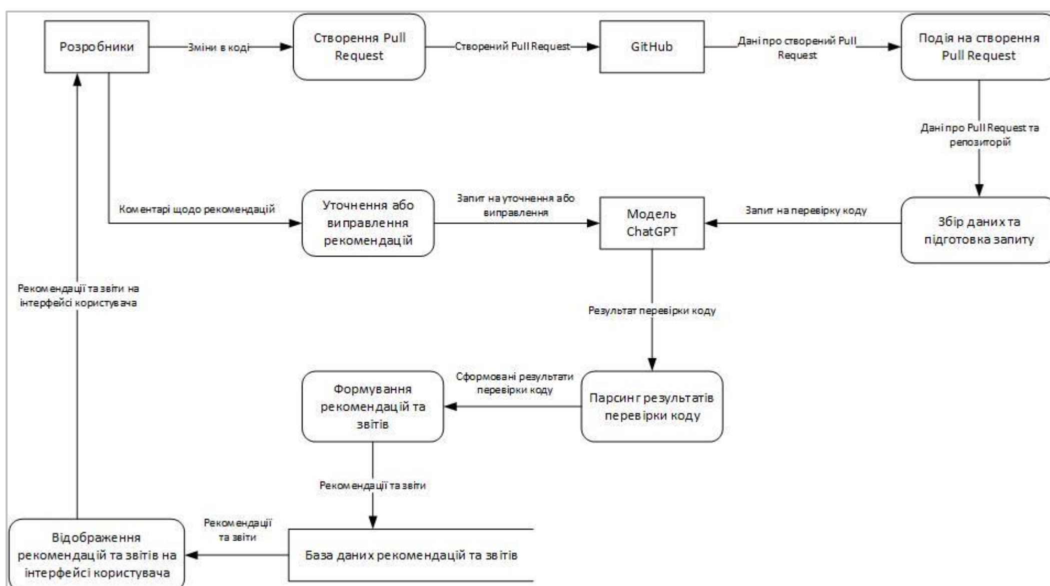


Рисунок 2.9 – Діаграма DFD 1-го рівня

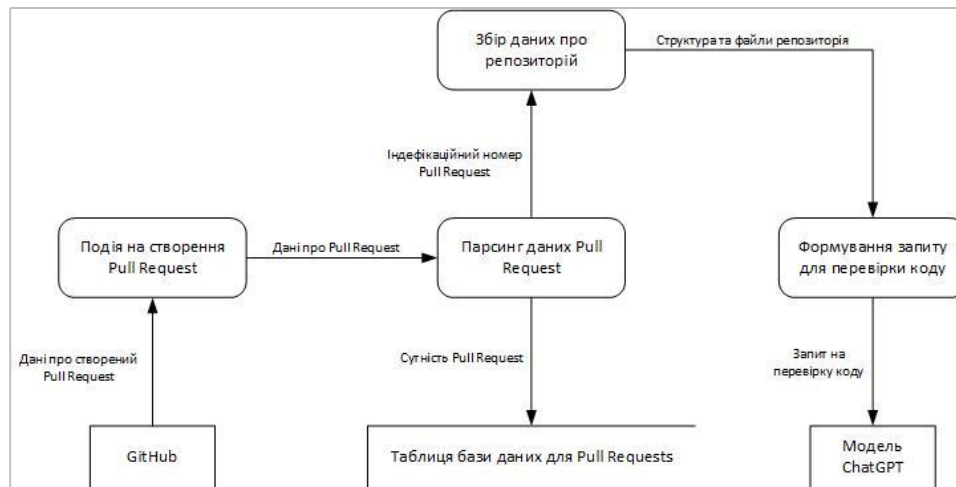


Рисунок 2.10 – Діаграма DFD процесу «Збір даних та підготовка запиту»

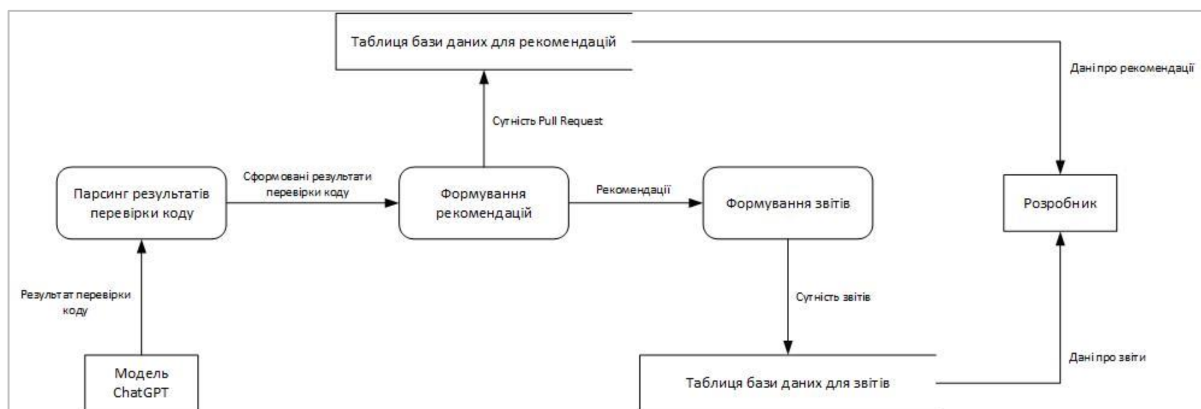


Рисунок 2.11 – Діаграма DFD процесу «Формування рекомендацій та звітів»

Створені діаграми DFD відображають процесів обробки та передачі даних, які дозволяють відобразити, як інформація пересувається між модулями системи та яким чином вона змінюється під час обробки.

2.4 Алгоритм роботи платформи для автоматизованої перевірки коду

Для розуміння роботи всієї платформи для автоматизованої перевірки коду побудовано алгоритм роботи всієї платформи у вигляді блок-схем (рис. 2.12). Цей алгоритм забезпечує працездатність платформи автоматизованої перевірки коду.

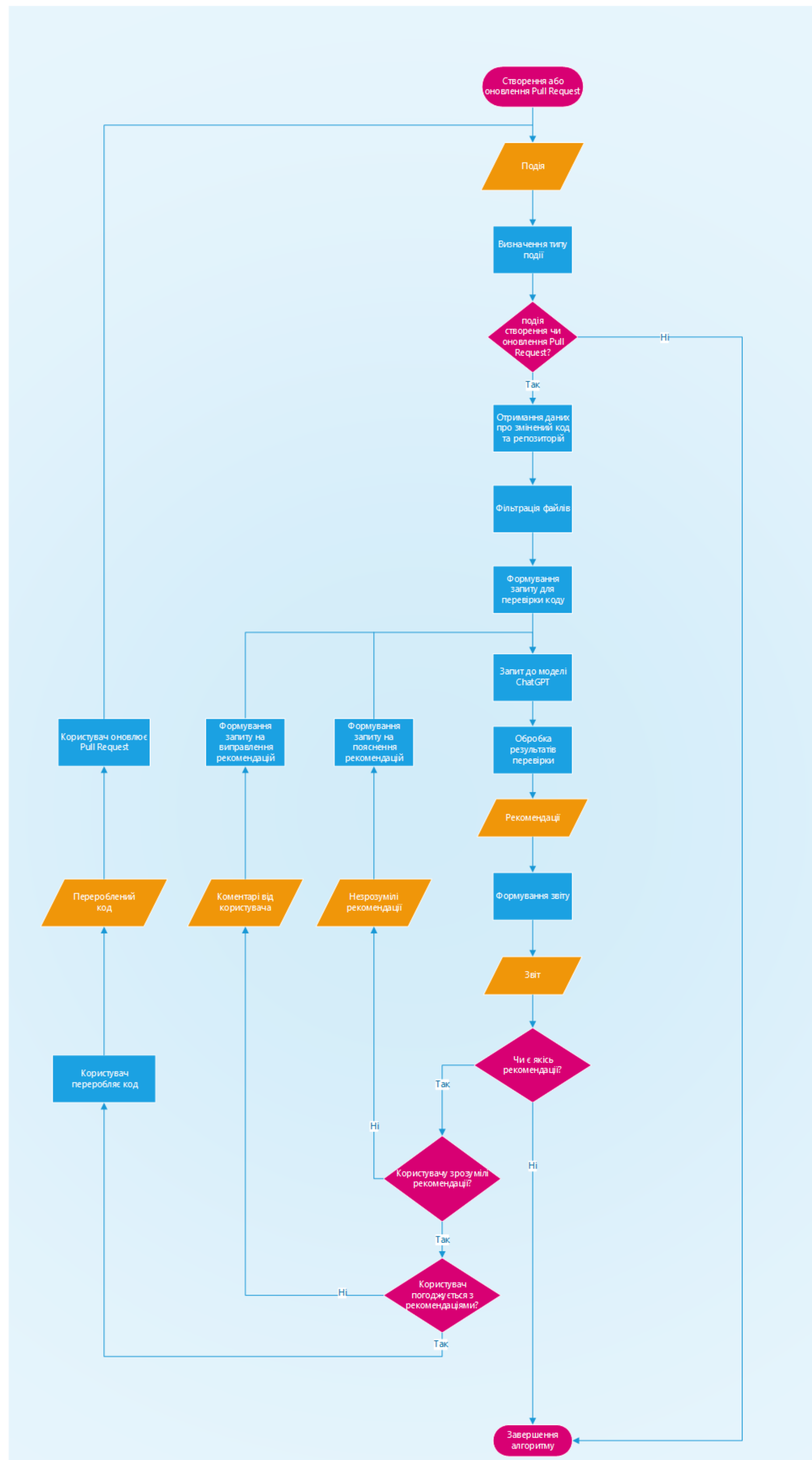


Рисунок 2.12 – Алгоритм роботи платформи для автоматизованої перевірки коду

Спочатку відстежується подія з платформи GitHub. Якщо подія – це створення чи оновлення pull request, тоді алгоритм починає свою роботу зі збору даних, інакше

алгоритм завершує свою роботу. При збору даних алгоритм визначає які відбулись зміни в коді, які файли були змінені, створенні або видаленні, яка структура проєкту репозиторію і т.д.

Далі алгоритм починає формувати запит до ChatGTP, який буде аналізувати код. Після результату аналізу коду починається обробка саме результату аналізу: визначення рекомендацій та їхній рівень (покращення, помилка, уразливість).

Після формування рекомендацій формується звіт, який буде показувати скільки помилок в коді, скільки місць, де можна покращити код та скільки уразливостей в коді. Якщо нема жодних рекомендацій, тоді алгоритм завершується. Якщо користувачу не зрозумілі рекомендації, він може написати коментарі до рекомендації. В такому випадку буде знову сформований запит до ChatGPT, але з метою пояснення власних рекомендацій. Після чого алгоритм знову робить запит та обробляє результат перевірки. Якщо користувачу зрозумілі рекомендації, але він з ними не згоден, то він за аналогією з незрозумілими рекомендаціями пише коментарі до рекомендацій. Таким чином знову формується запит на виправлення рекомендації та запит обробляється моделлю ChatGPT. Якщо користувачу зрозумілі рекомендації та він з ними згоден, тоді він виправляє код та оновлює pull request. В такому випадку алгоритм починає роботу з початку.

Висновки до розділу 2

У розділі 2 проведено детальне моделювання об’єкта та предмету роботи, а також розроблено функціональні та інформаційні моделі для платформи для автоматизованої перевірки коду. Розглянуто основні підходи та методи побудови моделей, які описують всі ключові процеси системи, починаючи від інтеграції з платформою GitHub, збору та підготовки даних, проведення аналізу коду за допомогою технологій штучного інтелекту, та закінчуючи формуванням звітів і взаємодією з користувачами.

Проаналізовано мету та завдання моделювання, а також визначено вимоги до моделей, які дозволили побудувати комплексне представлення роботи системи. Вибрано оптимальні інструменти моделювання, які забезпечують ефективну декомпозицію процесів, візуалізацію інформаційних потоків та розуміння алгоритмів роботи системи.

Розроблені моделі IDEF0 дали змогу відобразити функціональну структуру системи та взаємодію її компонентів, що забезпечує гнучкість і масштабованість системи в контексті її використання та подальшого розвитку. Побудовано DFD моделі для представлення інформаційних потоків та структурованої передачі даних між різними компонентами системи, що відображає ефективну обробку даних та інтеграцію з зовнішніми сутностями. Для більш детального розуміння логіки роботи системи розроблено блок-схеми алгоритму, який покроково демонструє виконання роботи платформи для автоматизованої перевірки коду.

Таким чином, створені моделі дозволяють зрозуміти логіку та функціональність системи автоматизованої перевірки коду, забезпечують структуроване представлення її компонентів, оптимізують процеси обробки та перевірки коду, а також формування звітів і рекомендацій. Це дозволяє перейти до наступних етапів проєктування та розробки системи, спираючись на детально опрацьовані та перевірені моделі.

3 АРХІТЕКТУРА, МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Розробка архітектури платформи для автоматичної перевірки коду

Загальна архітектура платформи поділена на три головні компонента платформи: серверна частина (Back-End), клієнтська частина (Front-End) та база даних. Розділення платформи на серверну та клієнтську частини робить архітектуру платформи більш масштабованою: з'являється можливість вибирати будь-які технології для розробки серверної та клієнтської частини, так як вони не пов'язані між собою. Але є значний мінус у цього підходу при розробці – довша розробка.

Серверна частина спроектована за допомогою мікросервісної архітектури. Мікросервісна архітектура є лідером серед інших архітектурних підходів сервісної частини у великих проєктах по типу Enterprice. Якщо коротко, то мікросервіс – незалежний сервіс, який має функцію або набір функцій, доступних через певний API (Application Interface) [17]. Майже завжди за API відповідає саме веб API – вебсервер на базі протоколу HTTP. Важливим аспектом Web API є так звані ендпоінти (endpoints). Ендпоінти відповідають за функції мікросервіса, які будуть використовуватись зовнішніми ресурсами. Саме ендпоінти складають програмний інтерфейс мікросервісу. За звичаєм ендпоінти викликаються через HTTP запит по якомусь посиланню URL. Також через протокол HTTP повертає відповідь (Response) ендпоінту у будь-якому структурованому форматі, який описує дані. Частіше за всього це JSON, рідше – XML. Також при розробці мікросервісної архітектури суворо дотримувався архітектурний стиль REST (Representational State Transfer). Цей архітектурний стиль був створений для рекомендації розробки вебзастосунків. Цей архітектурний стиль містить багато нюансів, але головними з них для мікросервісної архітектури є конвенції, пов'язані з ресурсами та uniform interface. Uniform interface – це одна з ключових характеристик архітектурного стилю REST що забезпечує стандартизовану взаємодію між клієнтом і сервером.

Uniform interface визначає набір обов'язкових принципів для створення API, які допомагають зробити REST-інтерфейси узгодженими, передбачуваними та незалежними від специфіки сервера чи клієнта. Важливими складовими uniform interface для мікросервісної архітектури включають:

– ідентифікація ресурсів: кожен ресурс (наприклад, користувач, стаття, коментар) має унікальний ідентифікатор у вигляді URI (Uniform Resource Identifier), який дозволяє отримати доступ до ресурсу через стандартні HTTP-операції (GET, POST, PUT, DELETE);

– маніпуляція ресурсами через представлення: клієнт працює не з самим ресурсом, а з його представленням (зазвичай JSON або XML), яке повертає сервер. Зміни, зроблені клієнтом у представленні, можуть бути надіслані серверу для оновлення самого ресурсу.

Правильна побудова ідентифікаторів ресурсів є також важливою частиною для RESTful вебсервісів:

– назви ресурсів (користувач, стаття, коментар) повинні бути іменниками в множині: users, articles, comments.

– дія над ресурсом – це HTTP-метод, який вказує клієнт API: GET – отримати ресурс, POST – створити ресурс, PUT – змінити ресурс повністю, DELETE – видалити ресурс, PATCH – частково змінити ресурс.

Таким чином взаємодія з ресурсами є доволі прозорою, не дивлячись на реалізацію ендпоінту.

Серверна частина платформи складається з трьох RESTful мікросервісів:

– Event Handler API – мікросервіс, який обробляє події створення pull requests;

– Pull Request API – мікросервіс для взаємодії з pull request. За допомогою цього мікросервісу користувач платформи може, наприклад, отримати інформації про всі pull requests свого репозиторію. Також за допомогою цього API користувачі можуть авторизуватись;

– Reviewer API – мікросервіс для перевірки коду pull request. За допомогою цього мікросервісу Event Handler мікросервіс зможе ініціювати перевірку коду pull request.

Діаграма послідовності мікросервісів наведена на рисунку 3.1.

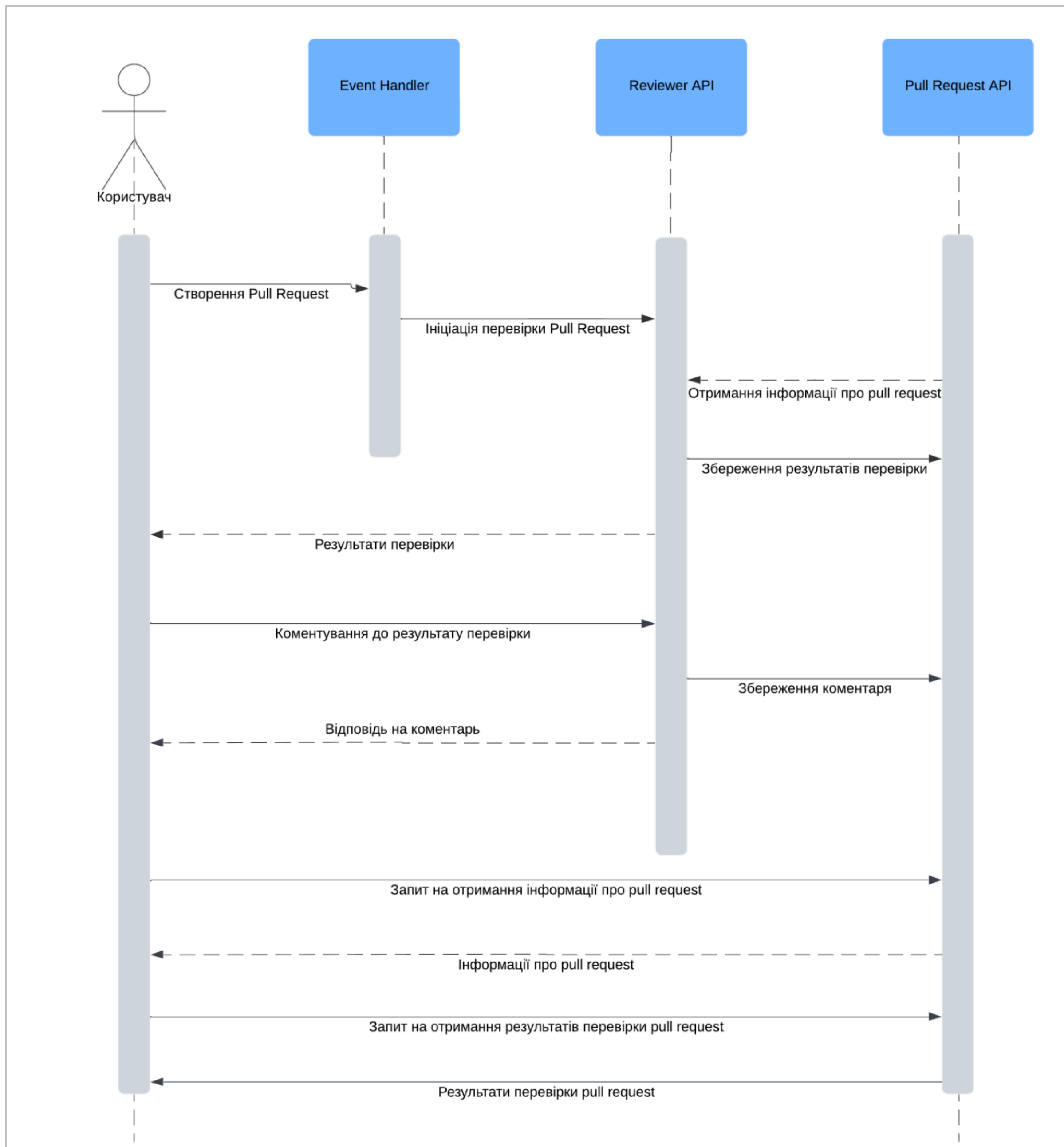


Рисунок 3.1 – Діаграма послідовності мікросервісів

Ця діаграма послідовності показує взаємодію між мікросервісами платформи для автоматичної перевірки коду.

Загалом 12 ендпоінтів реалізовують усю логіку платформи для автоматизованої перевірки коду.

Загальна архітектура платформи для автоматизованої перевірки коду наведена на рисунку 3.2.

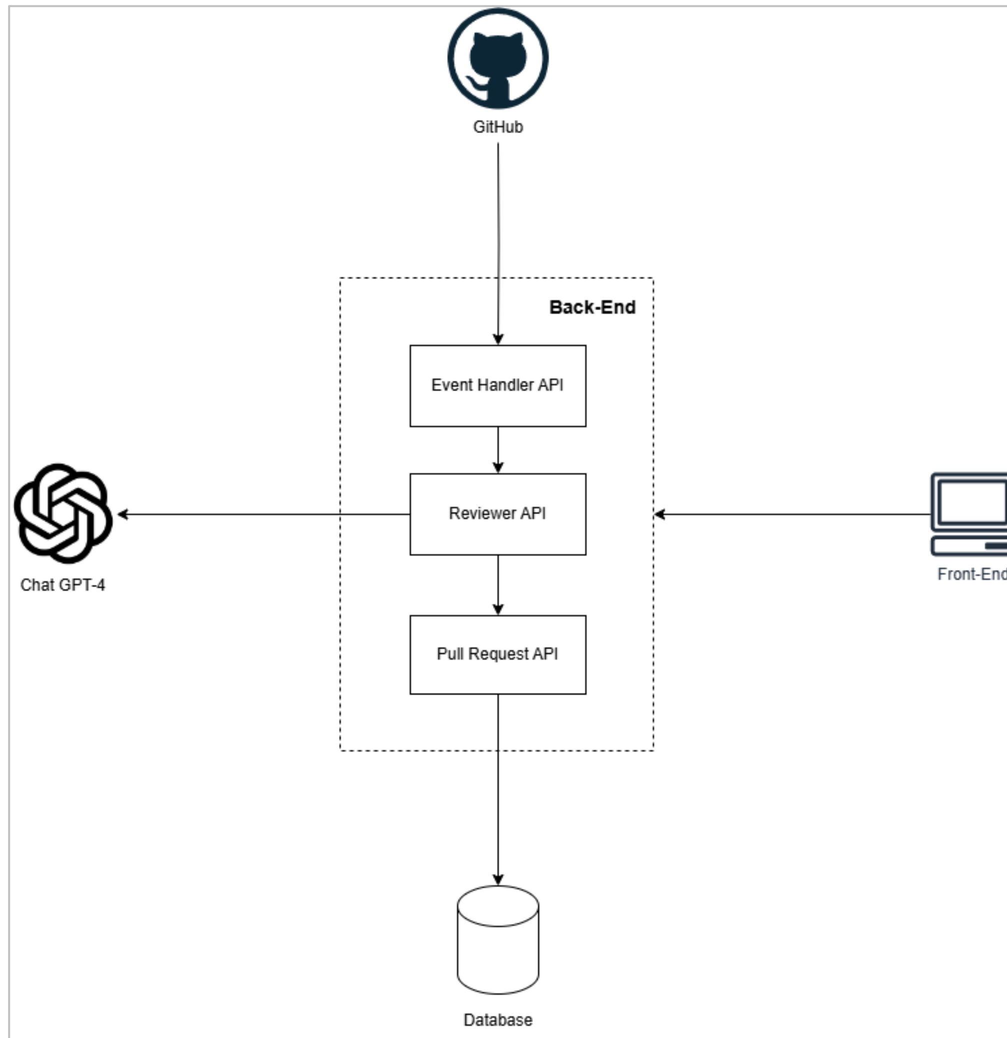


Рисунок 3.2 – Загальна архітектура платформи

Для більш детальної демонстрації загальної структури, ієрархії та взаємовідношення класів системи платформи автоматизованої перевірки коду, побудовано 2 діаграми класів. Перша діаграма класів призначена для мікросервісу Pull Request API. Вона зображена на рисунку 3.3.

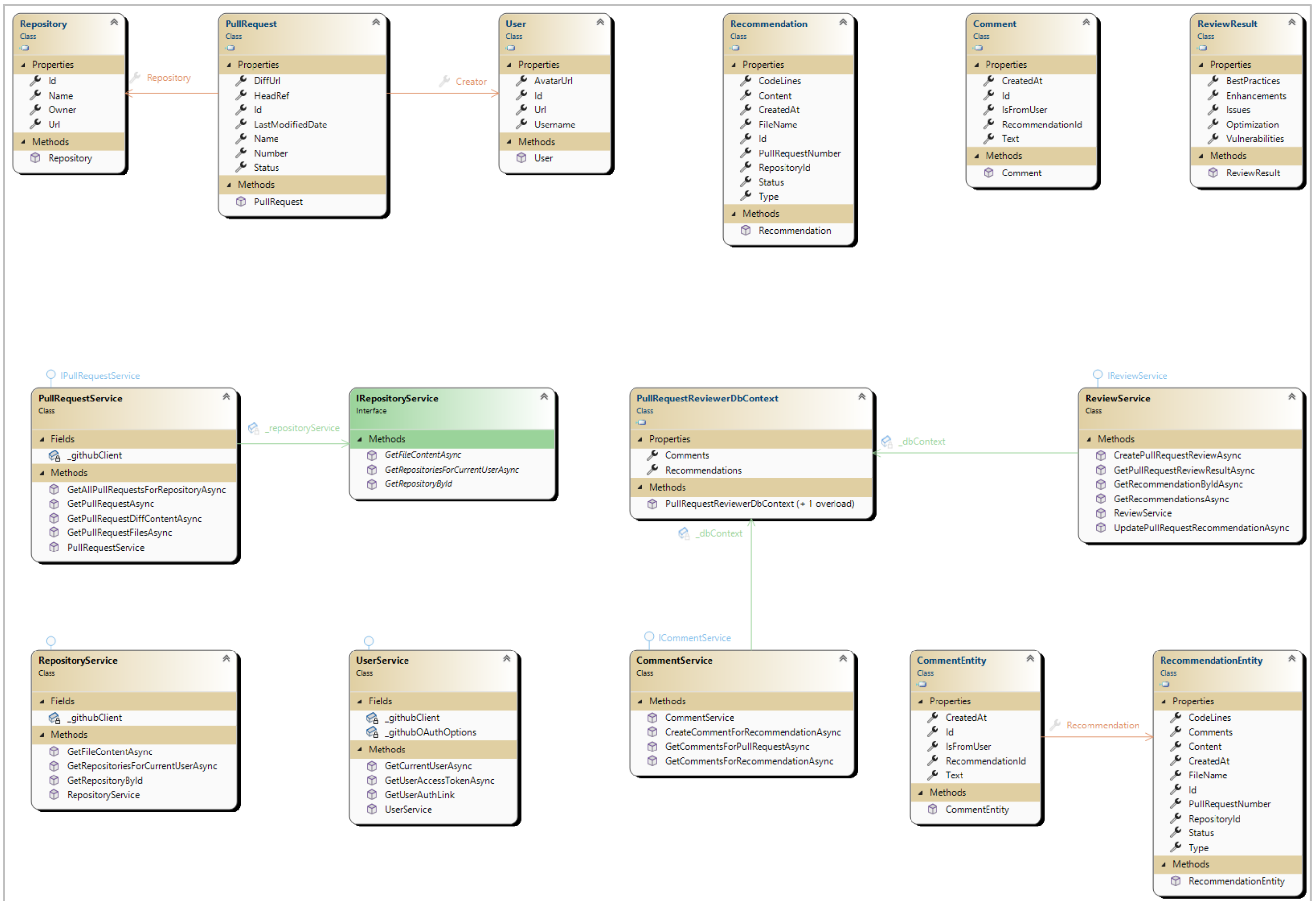


Рисунок 3.3 – Діаграма класів для мікросервісу Pull Request API

Другу діаграму класів призначено разом для мікросервісів Event Handler API та Reviewer API, яку зображено на рисунку 3.4.

Загалом на цих двох діаграмах показаний тільки доменний та інфраструктурний шар архітектури.

Рисунок 3.4 - Діаграма класів для мікросервісів Event Handler API та Reviewer API

Створені діаграми класів дозволяють детальної показати загальні структури, ієрархії та взаємовідношення класів системи платформи автоматизованої перевірки коду.

3.2 Структура бази даних платформи

Для збереження результатів перевірок коду pull requests платформа для автоматизованої перевірки коду використовує базу даних. Тип бази даних вибрано реляційну, так як:

– реляційні бази даних підходять до вирішення багатьох задач зі збереження даних;

– реляційні бази даних засновані на реляційній моделі даних, яка була заснована в 1970 році [18];

– реляційні бази даних використовують мову структурованих запитів SQL, для якого ще з 1983 року розроблялися стандарти [18].

База даних платформи складається з двох таблиць. Таблиці бази даних платформи та їхні колонки:

а) Recommendations – рекомендації щодо коду певного pull request:

1) Id – первинний ключ та ідентифікатор рекомендації;

2) Content – зміст рекомендації;

3) FileName – ім’я файлу, до якого належить рекомендація;

4) CodeLines – границі номерів ліній коду, до яких належить рекомендація;

5) RecommendationType – тип рекомендації;

6) RecommendationStatus – статус рекомендації;

7) CreatedAt – дата створення рекомендації.

б) Comments – коментарі до рекомендацій:

1) Id – первинний ключ та ідентифікатор коментаря;

2) Text – текст коментаря;

3) RecommendationId – зовнішній ключ, який вказує на рекомендацію, ідентифікатор рекомендації;

4) IsFromUser – індикатор, який вказує чи коментар від користувача;

5) ParentCommentId – зовнішній ключ, який вказує на батьківський коментар;

6) CreatedAt – дата створення коментаря.

На рисунку 3.5 наведено загальну ER-діаграму бази даних.

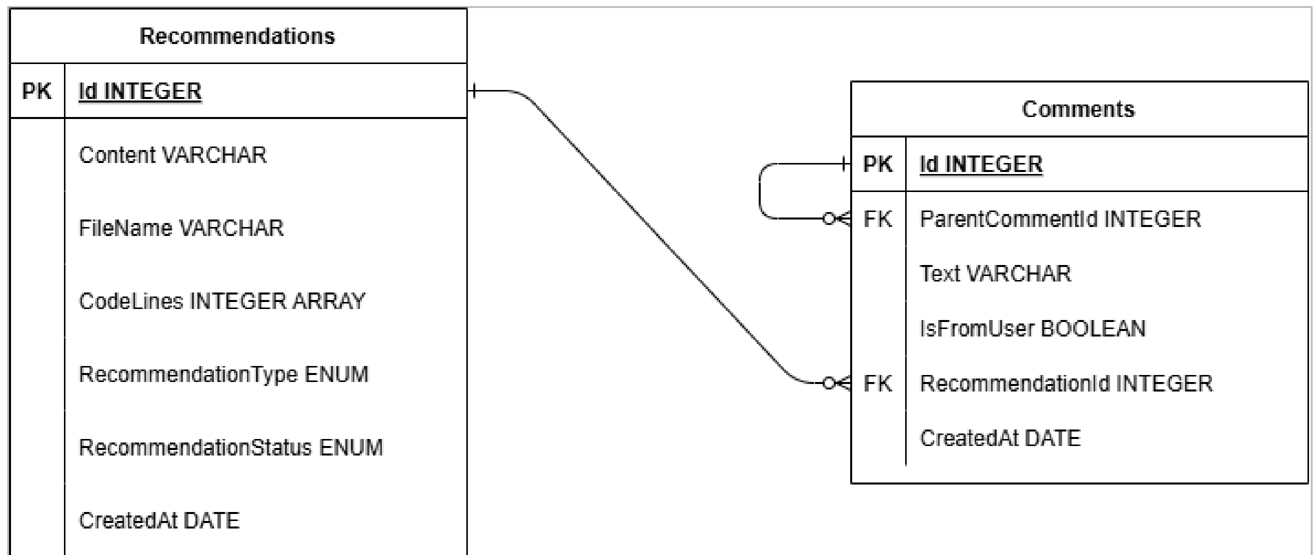


Рисунок 3.5 – ER-діаграма бази даних платформи

Ця ER-діаграма показує для структури даних, які зберігаються та обробляються у платформі автоматизованої перевірки коду.

3.3 Вибір технологій та мов програмування для розробки платформи

До вибору технологій та мов програмування для розробки платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту слід поставитись ретельно. Хоча й можна вибрати будь-яку мову програмування, фреймворк і т.д, але це рішення не є доцільним, та в майбутньому можуть виникнути величезні проблеми, пов’язані з вибором технологій для розробки проєкту. На етапі вибору технологій, вибір кращих та ефективних технологій значно підвищує ймовірність успішності розробки проєкту та подальшу його підтримку.

Спочатку вирішено обрати технології та мову програмування для серверної частини. Сьогодні існують велика кількість мов програмувань, на яких можна ефективно розробляти серверну частину на мікросервісной архітектурі. Найпопулярніші мови програмування та їхні фреймворки для розробки мікросервісів:

– C# – статично строго типізована ООП мова програмування, розроблений компанією Microsoft для платформи .NET [19]. Основним фреймворком для написання серверних вебзастосунків є ASP NET;

– Java – статично строго типізований ООП мова програмування, який сьогодні оновлюється компанією Oracle [20]. Основним фреймворком для написання серверних вебзастосунків є Spring Boot;

– JavaScript – мова програмування з динамічною типізацією, розроблена для написання інтерактивних сценаріїв для вебсайтів, згодом став найпопулярнішою мовою у 2014 році [21] та досі тримає лідерство серед найпопулярніших мов програмування у різних рейтингах. Із розвитком мови з'явилась платформа для виконання коду JavaScript за межами веббраузера – NodeJs. Фреймворки для написання мікросервісів: мінімалістичний Express.js та для написання крупних серверних застосунків NestJS;

– TypeScript – мова програмування, яка розширює можливості JavaScript, додаючи можливість явного статичного вказування типів та багато іншого [22]. Розроблений компанією Microsoft;

– Go – строго статична типізована мова програмування, розроблена компанією Google для розробки високошвидкісних мінімалістичних застосунків з використанням багатопоточністі [23]. Go вже має вбудований функціонал для розробки мікросервісів, який знаходиться в стандартній бібліотеці (net/http). Але також є високопродуктивний веб-фреймворк з мінімалістичним підходом – Gin;

– Python – динамічна строго типізована мова програмування, цінується за свою простоту читання та велику кількість бібліотек [24]. Є декілька фреймворків для створення мікросервісів, а саме мінімалістичний Flask та потужний з багатим функціоналом Django;

– PHP – динамічна мова програмування, яка широко використовується для розробки серверних вебзастосунків [25]. Існує багато фреймворків для створення мікросервісів на PHP, але основними серед них є популярний фреймворк з чистим

та елегантним синтаксисом Laravel та потужний фреймворк з модульною структурою Symfony.

Як можна побачити, існує великий вибір серед технологій та мов програмування для написання мікросервісів. Але для великих проєктів слід обирати такі мови програмування, які впевнено використовуються для написання реальних великих проєктів (Enterprise): C# або Java. Ці дві мови програмування дуже сильно схожі один на іншого. При розробці власної мови програмування C# для власної платформи .NET, розробники Microsoft явно надихалися мовою програмування Java, враховуючи як плюси, так і мінуси. Не дивлячись на якісний продукт у вигляді мови програмування C#, у Java все одно була значна перевага – кросплатформність. Ця перевага була значима, так як найпопулярнішою ОС для серверів була та є Linux-подібні ОС. Але тоді платформа .NET підтримувалась тільки для Windows під назвою .NET Framework. Згодом Microsoft випустила на світ .NET Core, який був open source та отримав кросплатформність. Після цього .NET Core почав отримувати популярність серед розробників серверних застосунків, включаючи розробників мікросервісів. Після появи .NET 5 назва .NET Core перестала існувати, з'явилась загальна назва .NET. Сьогодні остання стабільна версія для .NET є 8-ма. Тому, враховуючи всі плюси та мінуси кожного варіанту, вирішено розробляти серверну частину платформи для автоматизованої перевірки коду на платформі .NET 8 з використанням мови програмування C# версії 12. Також вибір мови програмування був на користь власного досвіду програмування.

Основним фреймворком платформи .NET для написання мікросервісів є ASP.NET. Цей фреймворк підтримує створення масштабованих, надійних та продуктивних мікросервісів. Зокрема ASP.NET Core є кросплатформенним та більш оптимізованою версією ASP.NET. ASP.NET надає зручні інструменти для створення RESTful API, що є важливим для побудови мікросервісів, орієнтованих на обробку HTTP-запитів. Завдяки простій інтеграції з атрибутами контролерів та можливістю створення добре організованих маршрутів, ASP.NET Core спрощує

реалізацію REST API та забезпечує ефективну передачу даних між сервісами. ASP.NET Core розроблено з модульною структурою, що дозволяє використовувати лише необхідні компоненти та розширення, зберігаючи при цьому продуктивність і ефективність. Це є важливим аспектом при побудові мікросервісів, де кожен сервіс повинен бути легким, самодостатнім та відповідати за одну бізнес-функцію. ASP. ASP.NET Core постійно демонструє високу продуктивність у різних тестах, часто перевершуючи інші веб-фреймворки. Наприклад, на онлайн ресурсі TechEmpower Web Framework Benchmarks, де оцінюють продуктивність багатьох веб-фреймворків у різних сценаріях, ASP.NET Core займає позиції серед лідерів [26]. Враховуючи все вище перераховане, ASP.NET Core є чудовим вибором для побудови мікросервісів.

Для інтеграції з GitHub з метою отримання даних про створення pull request репозиторію використано GitHub Webhook – це механізм, який дозволяє автоматично відправляти повідомлення про події, що відбуваються в GitHub-репозиторії, на вказаний зовнішній сервер [27]. Webhooks можуть реагувати на різні події в репозиторії, такі як новий commit, створення pull request, новий реліз, зміна статусу збирання коду та інші. Це дозволяє інтегрувати GitHub із зовнішніми сервісами, що автоматично отримують інформацію про події та реагують на них. Кожен запит від GitHub Webhook надсилається як HTTP POST запит у форматі JSON, який містить дані про подію. Наприклад, у разі коміту цей запит може містити інформацію про автора коміту, змінені файли, коментарі та інші деталі. Це забезпечує універсальний і стандартизований формат обміну даними. GitHub дозволяє обирати, які саме події будуть надсилатися на Webhook. Це може бути одна конкретна подія, декілька подій або всі події у репозиторії. Такий підхід дозволяє налаштувати Webhook під конкретні потреби проєкту, що оптимізує роботу з отриманими даними.

Також платформа для автоматизованої перевірки коду повинна використовувати дані з GitHub, які не надходять автоматично через Webhook,

наприклад, дані про репозиторії користувача. Тому платформа потребує можливості отримувати дані напряму з GitHub API, що дозволяє доступ до ширшого спектра інформації та забезпечує більшу гнучкість. GitHub API – це API, яке дозволяє розробникам автоматизувати роботу з GitHub, отримувати доступ до даних репозиторіїв, взаємодіяти з користувачами, управляти з pull requests, робити релізи та багатьма іншими аспектами платформи. Для платформи важливо, що GitHub API може отримувати дані про pull requests та репозиторіїв. Для взаємодії з GitHub API зручним способом у .NET-застосунках існує офіційна бібліотека Octokit. Octokit для .NET – це бібліотека-клієнт, розроблена GitHub для роботи з GitHub API у .NET-додатках, яка надає об'єктно-орієнтований інтерфейс для спрощеної роботи з API та автоматизації завдань. Octokit реалізує всі основні функції GitHub API, забезпечуючи доступ до репозиторіїв, pull requests, commits та інших аспектів платформи. Octokit спрощує роботу з GitHub API, надаючи готові об’єктно-орієнтовані класи та методи, які дозволяють виконувати запити без необхідності вручну обробляти HTTP-запити. Це значно спрощує процес розробки та зменшує обсяг коду. Octokit побудований з урахуванням сучасних стандартів .NET та підтримує асинхронне виконання запитів, що дозволяє ефективно обробляти великий обсяг даних без блокування основного потоку виконання.

Для інтеграції платформи для автоматизованої перевірки коду з ChatGPT існує необхідність використовувати OpenAI API – це API, що дозволяє розробникам взаємодіяти з потужними моделями штучного інтелекту, такими як GPT-3, GPT-4, DALL-E, Whisper та Codex, створеними компанією OpenAI. Цей API забезпечує доступ до технологій обробки природної мови, генерації зображень, розпізнавання мови та програмування, що дозволяє створювати інтелектуальні та творчі рішення в різних галузях. Але користуватись лише OpenAI API через API в .NET-застосунках не так зручно, як користуватись OpenAI SDK (Software Development Kit) для .NET. OpenAI SDK надає зручний інтерфейс для взаємодії з OpenAI API в додатках, побудованих на платформі .NET. Це дозволяє інтегрувати можливості

OpenAI, зокрема такі моделі як ChatGPT, GPT-4, DALL-E та інші, безпосередньо у .NET-застосунки, спрощуючи розробку продуктів на базі штучного інтелекту

Для збереження даних платформи для автоматизованої перевірки коду використано реляційну СУБД (система управління базами даних) PostgreSQL. PostgreSQL – потужна open source об'єктно-реляційна система управління базами даних, яка надає широкий набір функцій для зберігання, організації та обробки даних. Вона була розроблена з акцентом на надійність, масштабованість і сумісність зі стандартом SQL. PostgreSQL відома своєю здатністю підтримувати великі обсяги даних, складні запити та численні розширення, що робить її популярною як для невеликих проєктів, так і для корпоративних рішень. PostgreSQL є абсолютно безкоштовною СУБД, що працює під ліцензією з відкритим кодом PostgreSQL License. Це дозволяє використовувати її без обмежень і ліцензійних платежів, що робить її привабливою як для комерційних, так і для некомерційних проєктів. PostgreSQL має велику спільноту розробників та користувачів, що забезпечує постійну підтримку та регулярні оновлення. Існує багато документів, форумів і навчальних матеріалів, які полегшують вивчення PostgreSQL і допомагають вирішувати технічні питання.

Щоб використовувати функції СУБД для .NET-мікросервісів використано бібліотеку Entity Framework Core – система об'єктно-реляційного зіставлення (ORM) з відкритим кодом для .NET, розроблений компанією Microsoft, який дозволяє розробникам працювати з базами даних за допомогою об’єктів .NET, не пишучи при цьому SQL-запити вручну [28]. Завдяки Entity Framework можна легко взаємодіяти з базами даних, використовуючи об'єктно-орієнтований підхід, що спрощує доступ до даних, автоматизує зіставлення між базою даних і об’єктами, та підвищує продуктивність розробки. Entity Framework дозволяє автоматично перетворювати об’єкти .NET у записи бази даних і навпаки, використовуючи моделі. Це означає, що розробник може працювати з об’єктами та їх властивостями, не думаючи про SQL-запити для збереження або отримання даних. Entity Framework

підтримує систему міграцій, яка дозволяє керувати змінами у структурі бази даних на основі змін у моделях. Це забезпечує легкий спосіб відстеження змін та оновлення структури бази даних у процесі розробки. Для того, щоб повноцінно використовувати Entity Framework для СУБД PostgreSQL використано бібліотеку Npgsql.EntityFrameworkCore.PostgreSQL – провайдер СУБД PostgreSQL для Entity Framework.

Для завантаження всіх бібліотек, написаних вище, використано NuGet – це офіційний менеджер пакетів для платформи .NET, який дозволяє легко знаходити, встановлювати, оновлювати і управляти бібліотеками та інструментами, які можна використовувати в додатках .NET. Завдяки NuGet розробники можуть інтегрувати сторонні бібліотеки та фреймворки, спрощуючи процес розробки і підвищуючи продуктивність, а також створювати і публікувати власні пакети для використання в інших проєктах.

Вибір технологій та мови програмування, які дозволяють створювати інтуїтивно зрозумілий інтерфейс, ефективно обробляти дані та інтегруватися з Back-end, для розробки Front-end частини також немаловажний, як і для Back-end. Серед мов програмувань для Front-end головними, безконкурентними лідерами є JavaScript та його розширення TypeScript. Є ще мова програмування C# з фреймворком Blazor від Microsoft, на якому можна розробляти UI на платформі .NET. Також Blazor підтримує WebAssembly – технологія для виконання коду застосунку в браузері, що дозволяє запустити C#-код безпосередньо в браузері, що в свою чергу прискорює виконання коду. Але WebAssembly підтримується не всіма версіями браузерів, що може обмежити доступність для певних користувачів. Але Blazor відносно молодий фрейморк, який був створений 6 років тому, через це він не є зрілим та перевіреним відносно інших мастодонтів, які вже понад 10 років в індустрії, маючи широку екосистему та багатий набір інструментів. Тому цей фреймворк не є популярним вибором серед front-end розробників.

Серед фреймворків для JavaScript/TypeScript, на відмінну від мов програмування для Front-end, є значна конкуренція. Сьогодні існує величезна кількість фреймворків для розробки Front-end, але найпопулярнішими фреймворками є:

– React – бібліотека JavaScript, розроблена Facebook, яка спеціалізується на побудові інтерфейсів користувача [29]. Вона є компоненто-орієнтованою, що дозволяє створювати динамічні інтерфейси, розділяючи їх на незалежні, багаторазові компоненти. React підтримує віртуальний DOM, що дозволяє швидко оновлювати інтерфейс без повного перерендера сторінки;

– Angular – це фреймворк, розроблений Google, який надає повний набір інструментів для побудови масштабованих веб-додатків [30]. Angular базується на TypeScript і підтримує архітектуру MVC (Model-View-Controller), що допомагає структурувати додаток та забезпечувати його гнучкість;

– Vue – це прогресивний фреймворк JavaScript, створений для того, щоб бути простим у навчанні і гнучким у використанні [31]. Vue підтримує поступове впровадження, що дозволяє розробникам використовувати його для частин проєкту або поступово розширювати існуючі додатки.

Маючи такий вибір технологій, складно зробити вибір. Взагалі вибір конкретного фреймворку серед цих залежить від бажань розробників користуватися ним, так як функціонують ці технології майже однаково. Вони відрізняються лише структурно та набором інструментів «з коробки». Тому вибрано фреймворк Angular для розробки Front-end частини платформи для автоматизованої перевірки коду. На відмінну від React, Angular має повний набір інструментів «з коробки» для розробки комплексних застосунків. Angular використовує TypeScript, що підвищує стабільність і знижує ймовірність помилок за рахунок статичної типізації TypeScript. Також, на відмінну від Vue, за яким на сьогодні не стоїть якась велика компанія, Angular розробляється величезною компанією Google. Також Angular вибрано через власний досвід програмування. Також додатково до Angular

використано бібліотеку готових компонентів Material Design від компанії Google. Це в свою чергу зекономило час розробки UI частини.

3.4 Програмна архітектура платформи

Програмна архітектура платформи Back-End частини є дуже схожою на «цибульну» архітектуру (Onion Architecture). Представлення цього типу архітектури можна подивитись на рисунку 3.6. Onion Architecture – це архітектурний підхід до розробки програмного забезпечення, який фокусується на ізоляції бізнес-логіки (ядра додатка) від інфраструктурних залежностей. Ця архітектура була запропонована Джеффрі Пальмером і отримала свою назву через схожість з шаруватою структурою цибулі, де кожен шар має свої функції і залежності тільки від внутрішніх шарів [32].

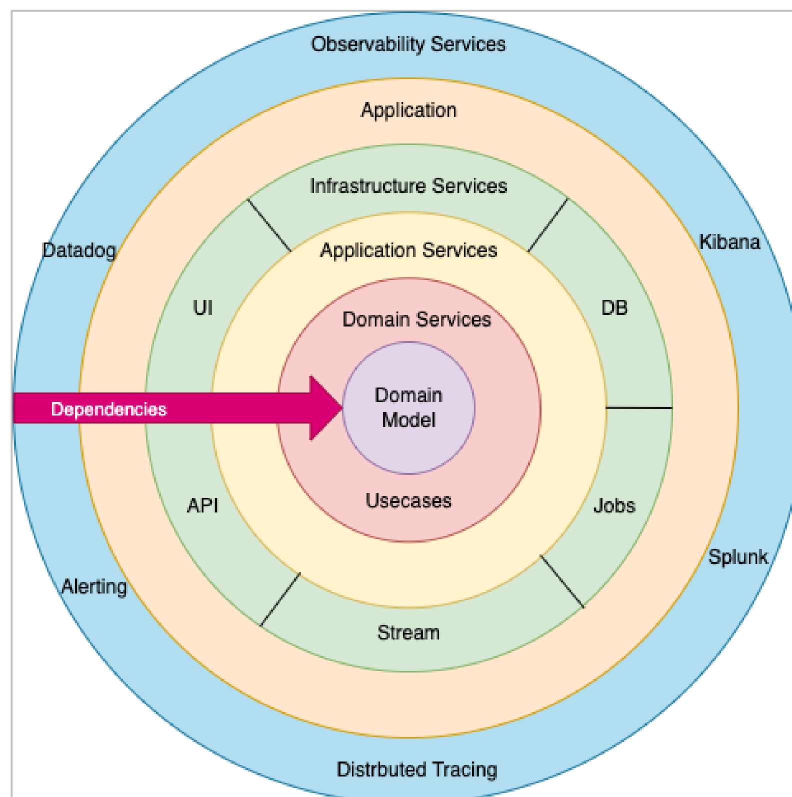


Рисунок 3.6 – Представлення «цибульної» архітектури

Головна концепція Onion Architecture полягає в тому, що всі зовнішні залежності (бази даних, веб-сервіси, інтерфейси користувача тощо) повинні бути ізольовані від бізнес-логіки. Це досягається за допомогою організації додатка у вигляді шарів, які мають чіткі залежності "до центру". Центральний шар (ядро) містить основну логіку і не залежить від жодних зовнішніх компонентів.

Переваги «цибульної» архітектури:

– ізоляція бізнес-логіки: основна логіка додатка не залежить від конкретної бази даних, веб-фреймворку чи інших технологій, що робить додаток менш схильним до змін у зовнішніх компонентах;

– гнучкість і розширюваність: завдяки шаруватій структурі легко додавати нові функціональні можливості або змінювати існуючі. Наприклад, змінити тип бази даних або додати новий спосіб взаємодії з користувачами;

– тестованість: ізольованість бізнес-логіки дозволяє легко писати юніт-тести для ядра системи без необхідності підключення до зовнішніх систем;

– масштабованість: завдяки модульності «цибульна» архітектура легко масштабується, дозволяючи розділяти роботу над різними шарами між командами.

Структуру програмної архітектури Back-End частини платформи зображено на рисунку 3.7.

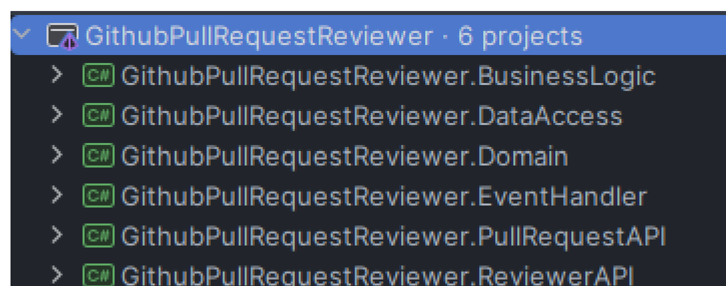


Рисунок 3.7 – Програмна архітектура Back-End частини платформи

Цю архітектуру можна назвати «цибульною» через чітку організацію додатка навколо його бізнес-логіки, що відповідає ключовим принципам цієї архітектури. Опис C# проєктів платформи:

– *GithubPullRequestReviewer.Domain* – містить основні елементи, які представляють бізнес-логіку додатка. Це ядро додатка, яке не має залежностей від інших частин проєкту;

– *GithubPullRequestReviewer.BusinessLogic* – містить реалізації бізнес-логіки. Тут відбувається обробка даних, реалізація правил і виконання основних операцій додатка;

– *GithubPullRequestReviewer.DataAccess* – відповідає за доступ до зовнішніх джерел даних (база даних, зовнішні API). Він знаходиться у зовнішньому шарі, оскільки залежить від конкретних інфраструктурних рішень;

– *GithubPullRequestReviewer.EventHandler* – частина інфраструктурного шару, яка відповідає за API для інтеграції з GitHub Webhooks для конфігурації з системою платформи та отримання подій на створення pull requests;

– *GithubPullRequestReviewer.PullRequestAPI* – частина інфраструктурного шару, яка відповідає за API для отримання даних з GitHub та запис результатів аналізу коду до власної бази даних;

– *GithubPullRequestReviewer.ReviewerAPI* – частина інфраструктурного шару, яка відповідає за API для інтеграції з ChatGPT для реалізації механізму перевірки коду та можливості коментування.

Висновки до розділу 3

У третьому розділі були розглянуті ключові аспекти архітектури, моделювання та проєктування програмного забезпечення платформи для автоматизованої перевірки коду.

Побудовано ґрунтовну архітектурну основу для платформи, визначено технології, методи та інструменти для реалізації та підтримки програмного забезпечення. Запропоновані рішення забезпечують гнучкість, продуктивність і надійність платформи, що дозволяє виконати всі поставлені завдання з автоматизованої перевірки коду.

4 КОДУВАННЯ ТА ТЕСТУВАННЯ ПЗ ПЛАТФОРМИ ДЛЯ АВТОМАТИЗОВАНОЇ ПЕРЕВІРКИ КОДУ

Для реалізації Back-End частини платформи створено рішення (Solution) для .NET проєктів – GithubPullRequestReviewer.sln. Рішення – це контейнер для інших проєктів.

4.1 Налаштування GitHub OAuth

Для того, щоб користувач мав змогу користуватись платформою для автоматизованої перевірки коду, він повинен бути авторизованим через GitHub. Можливість авторизації через GitHub надається завдяки GitHub OAuth.

Для конфігурації GitHub OAuth треба створити OAuth застосунок у GitHub. Для того, щоб створити застосунок GitHub OAuth, треба авторизуватись у GitHub, зайти у налаштування (Settings), зайти у Developer Settings, вибрати OAuth Apps та натиснути на OAuth Apps, як показано на рисунку 4.1.

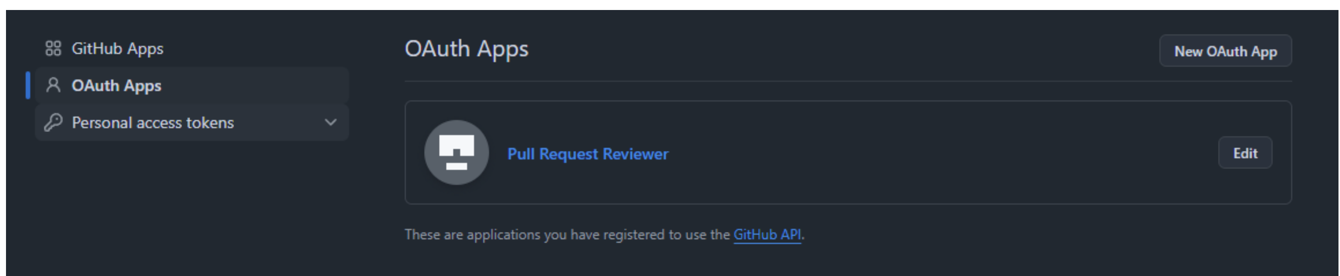


Рисунок 4.1 – Сторінка для налаштування GitHub OAuth Apps

Створено власний застосунок GitHub OAuth під назвою Pull Request Reviewer з такими налаштуванням:

– Homepage URL: адреса вебзастосунку. В цьому випадку локальна адреса Front-End застосунку – http://localhost:4200;

– Authorization callback URL: адреса вебзастосунку для передачі коду через параметр запиту для отримання токену. В цьому випадку ця адреса буде http://localhost:4200/auth.

Після натиснення на кнопки Create OAuth App створиться застосунок GitHub OAuth. Щоб ним можна було користуватись, то треба записати та використовувати такі секретні дані, як Client ID та Client secrets.

4.2 Опис реалізації мікросервісу Pull Request API

Для реалізації мікросервісу Pull Request API створено проєкт GithubPullRequestReviewer.PullRequestAPI.

Спочатку реалізовано механізм аутентифікації майже для всіх ендпоінтів системи, так як майже всі операції вимагатимуть щоб користувач був аутентифікований через GitHub. Також це є захистом від можливості доступу сторонніх користувачів.

Аутентифікацію ендпоінтів реалізовано за допомогою готового механізму аутентифікації в .NET – «схеми» аутентифікації. Для цього у папці Authorization створено клас GithubUserAuthenticationHandler, який спадкується від класу AuthenticationHandler<AuthenticationSchemeOptions> та перевизначає метод HandleAuthenticateAsync, який визначає як визначати, чи користувач аутентифікований чи ні. Лістинг коду класу GithubUserAuthenticationHandler наведено в додатку А.

Як можна побачити в лістингу, цей клас використовує інший клас, який реалізовує інтерфейс ITokenService – інтерфейс для роботи з токеном. Так, як авторизація буде відбуватись через GitHub OAuth, то результатом авторизації буде саме токен доступу (Access Token), за допомогою якого можна використовувати Octokit для отримання даних з GitHub.

```
public interface ITokenService
{
    Task<bool> ValidateAndSetTokenAsync(string token);
    string GetToken();
}
```

Цей простий інтерфейс має лише 2 методи: для валідації та присвоєння токена, якщо він валідний. Реалізує цей інтерфейс клас GithubTokenService. Лістинг коду класу GithubTokenService наведено у додатку А.

Токен зберігається у властивості Credentials класу GitHubClient, який реалізован в бібліотеці Octokit.NET. Перевіряється токен доступу через можливість отримання даних про поточного користувача. Якщо дані поточного користувача можна отримати, тоді токен валідний.

Для того, щоб ця «схема» аутентифікації змогла працювати, треба було налаштувати її у Dependency Injection (DI) контейнері в головному класі програми Program.

```
builder.Services
    .AddAuthentication("GithubUserAuthenticationScheme")
    .AddScheme<AuthenticationSchemeOptions,
GithubUserAuthenticationHandler>("GithubUserAuthenticationScheme",
options => { });
```

Після цього можна було використовувати атрибут Authenticate з «схемою» авторизації GithubUserAuthenticationScheme для методів контролерів, які потребують аутентифікацію від користувача.

```
[HttpGet]
[Route("current")]
[Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
public async Task<User> GetUserInfo()
{
    return await _userService.GetCurrentUserAsync();
}
```

Після розробки способу аутентифікації для ендпоінтів, далі розроблено контролери (Controllers). Контролери – це класи, які реалізують логіку ендпоінтів. Спочатку розроблено контролер UserController для розробки ендпоінтів, які

повертають дані про користувача: його користувацькі дані та репозиторії. Також є ендпоінти для отримання адреси авторизації через GitHub та для отримання токену доступу після авторизаціїї через GitHub. Лістинг коду контролеру UserController наведено у додатку А. Можа побачити, що цей контролер використовує класи, які реалізовують інтерфейси IUserService та IRepositoryService. Інтерфейс IUserService має 3 методи: 1 для повернення інформації про користувача, 2 для аутентифікації користувача.

```
public interface IUserService
{
    Task<User> GetCurrentUserAsync();
    string GetUserAuthLink();
    Task<string> GetUserAccessTokenAsync(string code);
}
```

Також створено домена клас-модель користувача.

```
public class User
{
    public long Id { get; set; }
    public string Username { get; set; }
    public string Url { get; set; }
    public string AvatarUrl { get; set; }
}
```

Ця модель має такі властивості:

- Id – ідентифікатор користувача у GitHub;
- Username – ім’я користувача у GitHub;
- Url – посилання на профіль користувача у GitHub;
- AvatarUrl – посилання на аватар користувача у GitHub.

Інтерфейс IRepositoryService має 2 методи для повернення репозиторіїв користувача.

```
public interface IRepositoryService
{
```

```
Task<IReadOnlyList<Repository>> GetRepositoriesForCurrentUserAsync();
Task<Repository> GetRepositoryById(long repositoryId);
}
```

Також створено домена клас-модель репозиторію.

```
public class Repository
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public User Owner { get; set; }
}
```

Ця модель має такі властивості:

– Id – ідентифікатор репозиторію у GitHub;
– Name – назва репозиторію у GitHub;
– Url – посилання на репозиторій у GitHub;
– Owner – об'єкт класу User, який представляє власника репозиторію.

Класи, які реалізовують інтерфейси IUserService та IRepositoryService: UserService та RepositoryService відповідно. Їхню реалізацію наведено у додатку А. Ці класи також використовують клас GitHubClient для отримання даних з GitHub.

Реалізовано контролер PullRequestController для реалізації ендпоінтів для отримання інформації про pull requests. Лістинг контролеру PullRequestController наведено у додатку А. Цей контролер має лише єдину залежність – клас який реалізовує інтерфейс IPullRequestService.

```
public interface IPullRequestService
{
    Task<PullRequest> GetPullRequestAsync(long repositoryId, int
pullRequestNumber);
    Task<IList<PullRequest>>
GetAllPullRequestsForRepositoryAsync(long repositoryId);
    Task<string> GetPullRequestDiffContentAsync(long repositoryId,
int pullRequestNumber);
```

```
    Task<IEnumerable<PullRequestFile>> GetPullRequestFilesAsync(long repositoryId, int pullRequestNumber);
}
```

Цей інтерфейс має 4 методи:

– GetPullRequestAsync: повертає інформацію про певний pull request за ідентифікатором репозиторію та номером pull request;

– GetAllPullRequestsForRepositoryAsync: повертає всі pull request репозиторію за його ідентифікатором;

– GetPullRequestDiffContentAsync: повертає зміст Git Diff певного pull request за ідентифікатором репозиторію та номером pull request;

– GetPullRequestFilesAsync: повертає файли певного pull request за ідентифікатором репозиторію та номером pull request.

Створено домену клас-модель, яка представляє pull request.

```
public class PullRequest
{
    public long Id { get; set; }
    public int Number { get; set; }
    public string Name { get; set; }
    public string HeadRef { get; set; }
    public string DiffUrl { get; set; }
    public PullRequestStatus Status { get; set; }
    public DateTime LastModifiedDate { get; set; }
    public User Creator { get; set; }
    public Repository Repository { get; set; }
}
```

Ця модель має такі властивості:

– Id – ідентифікатор pull request у GitHub;

– Number – номер pull request у GitHub;

– Name – назва pull request у GitHub;

– HeadRef – посилання на branch у GitHub;

– DiffUrl – посилання на Git Diff;
– Status – enum, який вказує на статус pull request: відкритий чи закритий;
– LastModifiedDate – дата останнього оновлення pull request;
– Creator – об’єкт класу User, який представляє створювача pull request;
– Repository – об’єкт класу Repository, який представляє репозиторій цього pull request.

Клас, який реалізовує інтерфейс IPullRequestService – PullRequestService. Його реалізацію наведено у додатку А. У нього також є залежність від класу GitHubClient для отримання даних з GitHub, та ще є залежність від класу, який реалізовує інтерфейс IRepositoryService.

Реалізовано контролер ReviewController для реалізації ендпоінтів для отримання та збереження інформації про перевірок коду. Лістинг контролеру ReviewController наведено у додатку А. Цей контролер має лише єдину залежність – клас який реалізовує інтерфейс IReviewService

```
public interface IReviewService
{
    Task CreatePullRequestReviewAsync(CreateReviewDto
createReviewDto);
    Task UpdatePullRequestRecommendationAsync(Recommendation
recommendation);
    Task<Recommendation> GetRecommendationByIdAsync(int
recommendationId);
    Task<IEnumerable<Recommendation>> GetRecommendationsAsync(long
repositoryId, int pullRequestNumber);
}
```

Цей інтерфейс має 1 метод для збереження результатів перевірки коду, 2 метода для отримання рекомендацій до коду та 1 метод для оновлення рекомендації до коду.

Створено домену клас-модель, яка представляє рекомендацію – Recommendation. Цей клас наведено у додатку А. Ця модель має такі властивості:

– Id – ідентифікатор рекомендації;
– Content – текст рекомендації;
– FileName – назва файлу, для якого є рекомендація;
– CodeLines – границі ліній коду, для яких є рекомендація;
– RepositoryId – ідентифікатор репозиторію;
– PullRequestNumber – номер pull request;
– Type – enum, який вказує на тип рекомендації: баг (Issue), уразливість (Vulnerability), оптимізація (Optimization), покращення (Enhancement), найкраща практика (BestPractice);
– Status – enum, який вказує на статус рекомендації: відкрита (Open) чи вирішена (Resolved);
– CreatedAt – дата утворення рекомендації.

Створено Data Transfer Object (DTO) клас, який передає дані для запису результатів перевірок коду – CreateReviewDto. Цей клас наведено у додатку А. Цей DTO має такі властивості:

– RepositoryId – ідентифікатор репозиторію;
– PullRequestNumber – номер pull request;
– Issues – список з об’єктів класу CreateReviewDtoItem, який представляє DTO для запису рекомендації щодо багу;
– Vulnerabilities – список з об’єктів класу CreateReviewDtoItem, який представляє DTO для запису рекомендації щодо вразливості;
– Optimizations – список з об’єктів класу CreateReviewDtoItem, який представляє DTO для запису рекомендації щодо оптимізації;
– Enhancements – список з об’єктів класу CreateReviewDtoItem, який представляє DTO для запису рекомендації щодо покращення;
– BestPractices – список з об’єктів класу CreateReviewDtoItem, який представляє DTO для запису рекомендації щодо кращої практики.

Створено DTO-клас, який передає дані для запису рекомендації щодо коду – CreateReviewDtoItem. Цей клас наведено у додатку А. Цей DTO має такі властивості:

– Description – текст рекомендації;
– File – назва файлу, для якого є рекомендація;
– BeginsAtCodeLine – початкова лінія коду, для якої є рекомендація;
– EndsAtCodeLine – кінцева лінія коду, для якої є рекомендація.

Клас, який реалізовує інтерфейс IReviewService – ReviewService. Його реалізацію наведено у додатку А. У класу є лише одна залежність – клас PullRequestReviewerDbContext. Це клас, який надає контекст для роботи з базою даних. Його реалізація наведена у додатку А. Цей клас успадковується від класу бібліотеці Entity Framework – DbContext. Клас PullRequestReviewerDbContext має такі його властивості та методи:

– Recommendations – об’єкт класу DbSet<RecommendationEntity>, який представляє сутності рекомендацій в базі даних;

– Comments – об’єкт класу DbSet<CommentEntity>, який представляє сутності коментарів в базі даних;

– OnConfiguring – метод, який перевизначає метод базового класу DbContext. Цей метод використовується для підключення до БД при виконанні міграцій.

Для того щоб клас PullRequestReviewerDbContext функціонував у залежних від нього класах, треба додати його в DI контейнер в класі Program:

```
builder.Services.AddDbContext<PullRequestReviewerDbContext>(options
=>
options.UseNpgsql(builder.Configuration.GetConnectionString("Database
")));
```

Connection String БД вказано у файлі для конфігурації застосунку appsettings.Development.json.

Для того, щоб БД правильно працювала, треба спочатку створити саму базу даних, потім інші її елементи, такі як таблиці, індекси, обмеження тощо. Для цього використано підхід Code First – коли спочатку пишеш код для ORM, потім через міграції оновлюється БД.

Спочатку була створена база даних GithubPullRequestReviewer за допомогою застосунку для СУБД PostgreSQL – PgAdmin. Потім були створені класи сутностей в проєкті GithubPullRequestReviewer.DataAccess для рекомендацій та коментарів: класи RecommendationEntity та CommentEntity. Вони майже нічим не відрізняються від доменних моделей Recommendation та Comment. Після створення класів сутностей, створено первинну міграцію для БД за допомогою контекстного меню в Rider – Entity Framework Core → Add Migration. Після виконання команди на створення міграції, в папці Migrations створено клас первинної міграції InitialCreate. Код класу InitialCreate наведено у додатку А. Для того, щоб запустити міграцію використано команду з контекстного меню Rider – Entity Framework Core → Update Database. Після виконання цієї команди можна було використовувати клас PullRequestReviewerDbContext для взаємодії з БД.

Реалізовано контролер CommentController для реалізації ендпоінтів для отримання та збереження інформації про коментарі до рекомендацій. Лістинг контролеру CommentController наведено у додатку А. Цей контролер має лише єдину залежність – клас який реалізовує інтерфейс ICommentService.

```
public interface ICommentService
{
    Task CreateCommentForRecommendationAsync(string text, bool
isFromUser, int recommendationId);
    Task<IList<Comment>> GetCommentsForPullRequestAsync(long
repositoryId, int pullRequestNumber);
    Task<IList<Comment>> GetCommentsForRecommendationAsync(int
recommendationId);
}
```

Цей інтерфейс має 1 метод для збереження коментаря до рекомендації, 2 метода для отримання коментарів: до певної рекомендації та до всіх рекомендацій pull request.

Створено домену клас-модель, яка представляє коментар – Comment. Код цього класу наведено у додатку А. Ця модель має такі властивості:

– Id – ідентифікатор коментаря;
– RecommendationId – ідентифікатор рекомендації, до якої належить коментар;
– Text – текст коментаря;
– IsFromUser – Boolean значення, яке вказує чи коментар від користувача, чи від системи перевірки коду;
– CreatedAt – дата створення коментарю.

Клас, який реалізовує інтерфейс ICommentService – CommentService. Його реалізацію наведено у додатку А. У класу є лише одна залежність – клас PullRequestReviewerDbContext для взаємодії з даними у БД.

4.3 Опис реалізації мікросервісу Reviewer API

Мікросервіс Review API призначений для перевірки коду за допомогою ChatGPT. Тому проєкт містить всього лише 1 контролер – ReviewerController. Реалізація контролера ReviewerController наведено у додатку А. Цей контролер має 4 залежності:

– клас, який реалізовує інтерфейс IGenerativeModelProvider;
– клас, який реалізовує інтерфейс IPullRequestApiClient;
– клас, який реалізовує інтерфейс IReviewApiClient;
– клас, який реалізовує інтерфейс ICommentApiClient.

```
public interface IGenerativeModelProvider
{
   Task<string> SendMessageAsync(string text);
   Task<string> SendMessagesAsync(IEnumerable<ChatMessage> messages);
```

```
}
```

Цей інтерфейс має 2 метода для відправки повідомлень до генеративної моделі:

– метод SendMessageAsync відправляє 1 повідомлення та отримує відповідь на нього;

– метод SendMessagesAsync відправляє ланцюжок повідомлень від користувача та системи з метою збереження історій повідомлень.

Клас ChatMessage – клас бібліотеки Open.AI, який представляє повідомлення. Він є базовим класом для класів UserChatMessage та SystemChatMessage, які представляють повідомлення від користувача та системи відповідно.

Клас, який реалізовує інтерфейс IGenerativeModelProvider – ChatGptModelProvider. Код цього класу наведено у додатку А.

Інтерфейси IPullRequestApiClient, IReviewApiClient, ICommentApiClient містять методи для виклику HTTP запитів до відповідних контролерів мікросервісів: PullRequestController, ReviewController, CommentController. Класи PullRequestApiClient, ReviewApiClient, CommentApiClient реалізовують відповідно інтерфейси IPullRequestApiClient, IReviewApiClient, ICommentApiClient. Також ці класи успадковуються від базового класу BaseApiClient, який має набір методів для викликів HTTP запитів. Цей клас залежить від значення адреси мікросервісу та від класу, який реалізовує ITokenService – GithubTokenService для отримання токена доступу та подальшої його передачі у заголовку запитів. Зі значення адреси мікросервісу створюється об’єкт класу HttpClient – клас для роботи з HTTP запитами. Реалізацію класу BaseApiClient наведено у додатку А.

Клас PullRequestReviewer використовує статичний клас Prompts для побудови промптів для ChatGPT з метою перевірки коду. Взагалі створення промпту в даному випадку було важливою задачею, так як від якісного промпту можна отримати гарні відповіді від генеративної моделі. Промпт побудовано так, щоб ChatGPT відповідав тільки у JSON форматі, який відповідає класу ReviewResultResponse. Цей клас

майже такий же самий, як CreateReviewDto, але без властивостей RepositoryId та PullRequestNumber, а також замість класу CreateReviewDtoItem використовується клас ReviewResultResponseItem, у якого властивості такі ж самі, як у CreateReviewDtoItem. У самому промпті на перевірку коду є так звані плейсхолдери, які при побудові кінцевого промпту замінюються такі дані: назва pull request та зміст Git Diff.

Клас Prompts містить не тільки промпт для перевірки pull request, а ще й промпт на відповідь на коментар від користувача, який містить плейсхолдери для таких даних: назва файлу, текст рекомендації, коментар від користувача.

Реалізація класу Prompts наведено в додатку А.

Після того, як проведено перевірку pull request, клас PullRequestReviewer зберігає результати перевірки, викликавши ендпоінт мікросервісу Pull Request API для збереження результату перевірки. Також само й для коментарів: після того, як ChatGPT дав відповідь на коментар користувача, клас PullRequestReviewer зберігає коментар самого користувача та коментар від ChatGPT, викликавши ендпоінт мікросервісу Pull Request API для збереження коментарів.

4.4 Опис реалізації мікросервісу Event Handler

Мікросервіс Event Handler призначений для налаштування інтеграції до GitHub за допомогою GtiHub Webhooks. Проєкт містить всього лише 1 контролер – GithubWebhookController. Реалізація контролера GithubWebhookController наведена у додатку А. Цей контролер має залежність від класу, який реалізовує інтерфейс IGithubWebhookService.

```
public interface IGithubWebhookService
{
    Task<IReadOnlyList<RepositoryHook>>
GetAllRepositoryWebhooksAsync(long repositoryId);
    Task CreateWebhookAsync(long repositoryId);
}
```

Цей Інтерфейс має такі методи:

– GetAllRepositoryWebhooksAsync – повертає всі webhooks певного репозиторію;

– CreateWebhookAsync – метод для створення webhook для певного репозиторію. Webhook буде реагувати на події створення та оновлення pull request та відправляти дані про події на задану вебадресу.

Клас, який реалізовує інтерфейс IGithubWebhookService – GithubWebhookService. Цей клас залежить від конфігурації застосунку та від класу GitHubClinet бібліотеки Octokit.NET. Реалізація класу GithubWebhookService наведена у додатку А.

Клас, який відповідає за обробку подій створення або оновлення pull request є клас PullRequestWebhookEventProcessor, який успадковується від базового класу WebhookEventProcessor – клас бібліотеки Octokit.Webhooks, який створює ендпоінт для отримання даних через GitHub Webhook. У цього класу можна перевизначити логіку обробки деяких подій. В даному випадку перевизначено метод ProcessPullRequestWebhookAsync, який відповідає за обробку подій, пов’язаних з pull request.

Клас PullRequestWebhookEventProcessor має залежність від класу, який реалізовує інтерфейс IReviewerApiClient, тобто від класу ReviewerApiClient.

Клас PullRequestWebhookEventProcessor працює таким чином: якщо був створений pull request, тоді виконуємо запит на ендпоінт мікросервісу Reviewer API для перевірки pull request (метод ReviewPulRequestAsync). Якщо pull request був оновлений, тоді виконуємо запит на ендпоінт мікросервісу Reviewer API для перевірки commit.

Реалізація класу PullRequestWebhookEventProcessor наведена у додатку А.

4.5 Опис реалізації користувацького інтерфейсу

Як було описано, користувацький інтерфейс створено за допомогою фреймворку Angular. Тому створено проєкт за допомогою утиліти angular-cli, яка надає команди для швидкого створення різних компонентів фреймворку. За допомогою команди ng new GithubPullRequestReviewer.UI створено сам проєкт.

Для взаємодії з розробленими API згенеровано готові Angular сервіси за дорогою Node Package Manager (NPM) пакету ng-openapi-gen. Спершу створено JSON Swagger файли, зміст котрих було скопійовано з таких адрес API:

[адреса API]/swagger/v1/swagger.json.

В результаті було створено 3 файли:

– pull-request-open-api.json – swagger файл для генерації сервісу для Pull Request API;

– reviewer-open-api.json – swagger файл для генерації сервісу для Reviewer API;

– webhook-open-api.json – swagger файл для генерації сервісу для Webhook API.

Також створено NPM скрипти в файлі package.json, які саме використовують утиліту пакету ng-openapi-gen для генерації готових сервісів:

```
"generate-pull-request-api": "ng-openapi-gen --input open-api/pull-request-open-api.json --output src\\app\\api\\pull-request",
    "generate-webhook-api": "ng-openapi-gen --input open-api/webhook-open-api.json --output src\\app\\api\\event-handler",
    "generate-reviewer-api": "ng-openapi-gen --input open-api/reviewer-open-api.json --output src\\app\\api\\reviewer"
```

Після виконання таких скриптів генеруються готові сервіси, які використовують ендпоінти розроблених API. Приклад згенерованого сервісу для Event Handler мікросервісу наведено у додатку А.

Першою сторінкою для розробки стала сторінка авторизації. Для цього створено Angular-компонент через команду утиліти angular-cli: ng g c auth. Таким чином створилась папка з назвою auth з такими файлами компоненти:

– auth.component.ts: в таких файлах знаходиться логіка UI-елемента, написана на мові програмування Type Script;

– auth.component.html: в таких файлах знаходиться шаблон UI-елемента на мові побудови шаблонів HTML;

– auth.component.scss: в таких файлах знаходиться стилі UI-елемента, написаних на мовах CSS, SCSS, LESS. В цьому випадку на мові SCSS.

Компонент авторизації має лише такі UI-елементи:

– назва платформи;

– гасло платформи;

– кнопку авторизації через GitHub.

Компонент авторизації має залежності в конструкторі, які здійснюються через DI:

– клас UserService – згенерований сервіс для отримання даних про користувача. Цей сервіс використовується в компоненті для отримання токена доступу по коду, який буде в параметрах адреси вебзастосунку;

– клас AuthService – створений сервіс, який має методи для взаємодії з авторизацією користувача, в тому числі для збереження токену доступу.

– клас Router – клас для роботи з переміщенням між маршрутами вебзастосунку. В цьому випадку використовується для переміщення користувача на головну сторінку застосунку після успішної авторизації;

– клас ActivatedRoute – клас для отримання даних про поточний маршрут застосунку. В цьому випадку за допомогою методу subscribe бібліотеки для реактивного програмування RxJS відстежується параметр адреси застосунку code. Якщо був переданий код параметром, тоді отримаємо токен доступу по цьому коду, якщо код валідний.

Взагалі логіка авторизації так працює:

1) користувач натискає на кнопку «Sign via GitHub»;

2) користувач потрапляє на сторінку GitHub для авторизації;

3) користувач після авторизації попадає на адресу, яку було вказано як Authorization callback URL з параметром code зі значенням коду. В цьому випадку користувач попадає на цю ж сторінку авторизації, але з параметром адреси code;

4) компонент авторизації реагує на те, що є параметр code, використовуючи сервіс UserService, який в свою чергу викликає ендпоінт Pull Request API, який повертає токен за кодом, значення якого знаходиться в параметрі адресі code.

5) компонент авторизації отримує токен доступу від сервісу UserService та записує його у Local Storage за допомогою сервісу AuthService.

6) компонент перенаправляє користувача на головну сторінку платформи.

Код класу AuthComponent та код шаблону сторінки авторизації наведено у додатку Б. Результат розробки сторінки авторизації наведено на рисунку 4.2.

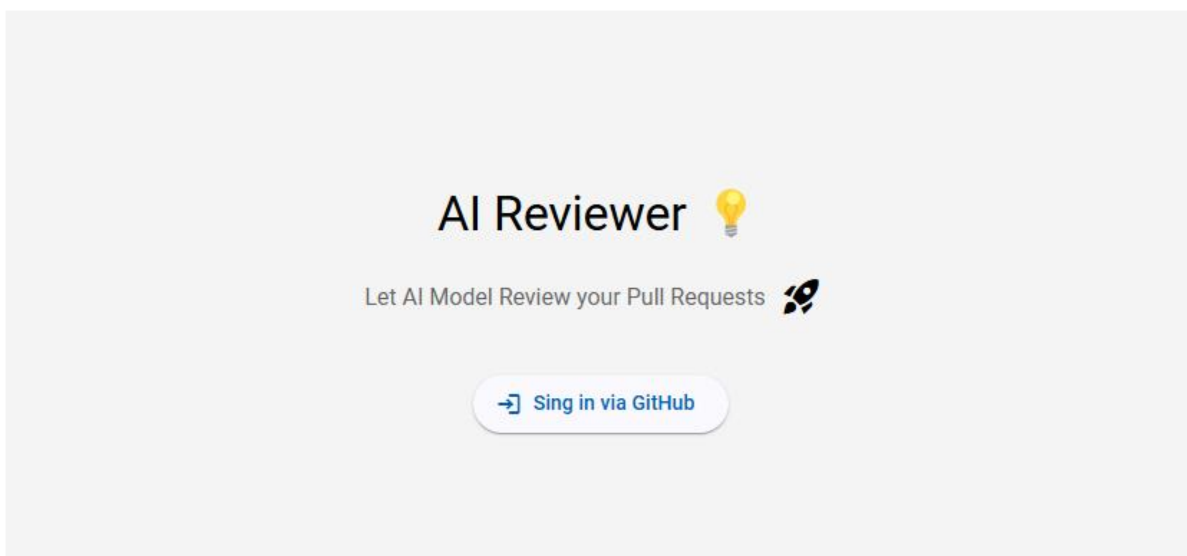


Рисунок 4.2 – Сторінка авторизації

Перед тим, як розроблялись інші сторінки, розроблено механізм недопуску користувача до головних сторінок платформи, якщо користувач не авторизований. Для цього було розроблено власний Angular Guard – AuthGuard. Angular Guards –

механізми, які перевіряють чи може користувач переходити по вказаним маршрутам за вказаною умовою. Власний AuthGuard працює таким чином: якщо користувач намагається потрапити на сторінку за її маршрутом, але він не авторизований, тоді користувач автоматично відправляється на сторінку атворизаціїї.

AuthGuard вказується для маршрів, які будуть недоступними для неавторизованого користувача. Це вказується в об'єкті маршрутів Routes, який знаходиться у файлі app.routes.ts.

Після розробки AuthGuard, розроблено основні сторінки платформи:

– сторінка для конфігурації репозиторіїв;
– сторінка зі списком pull requests;
– сторінка аналізу коду.

Всі ці сторінки складаються з двох загальних Angular-компонентів: компонент заголовку та компонент навігації.

Компонент заголовку платформи є статичним, знаходиться зверху та скадається з таких елементів:

– назва платформи;
– аватар користувача;
– кнопка виходу з системи;

При натисканні на кнопку виходу, користувач вилогінюється з системи та повертається на сторінку авторизації. UI-дизайн компонента можна побачити на рисунку 4.3.

Логіку компонента заголовку реалізовано в Angular-комопненті HeaderComponent. Код класу HeaderComponent та код шаблону компоненту заголовку наведено у додатку Б. Клас HeaderComponent має залежності від сервісів ApiService для отримання даних про авторизованого користувача за допомогою метода getAuthenticatedUser та AuthService для вилогування користувача за

допомогою методу logOut. Також шаблон компоненту заголовку містить готовий компонент MatToolbar бібліотеки Material.

Компонент навігації є статичним та знаходиться зліва та виконує роль навігаційного меню. UI-дизайн компонента можна побачити на рисунку 4.3. Навігаційне меню платформи має кнопки для навігації між сторінками. Компонент навігації містить в шаблоні компонент MatSidenav з бібліотеки Material. Також він містить в шаблоні елемент router-outlet, який діє, як місце для інших компонентів в залежності від поточного маршруту. В цьому випадку це компоненти для основних сторінок платформи. Маршрути для компонентів вказуються в об'єкті Routes.

Логіку компонента навігації реалізовано в класі NavigationComponent, який має одне поле pageTitle для зображення заголовку сторінки та один метод navigateToPage для навігації між сторінками. Цей клас має такі залежності:

– клас Router для навігації між сторінками;

– клас ActivatedRoute для зчитування поточного стану маршруту для визначення поля pageTitle.

Код класу NavigationComponent та код шаблону компоненту навігації наведено у додатку Б.

Першою основною сторінкою платформи була розроблена сторінка для конфігурації репозиторіїв. Мета цієї сторінки – надати можливість користувачу вибирати, які його репозиторії будуть аналізуватися системою при створені pull requests.

Сторінка для конфігурації репозиторіїв містить список репозиторіїв. Кожний елемент списку містить перемикач зі стану «не налаштований» (Not Configured) на стан «налаштований» (Configured). При перемиканні стану, у репозиторію створюється GitHub webhook, який буде відправляти події до Event Handler API.

Сторінка для конфігурації репозиторіїв була розроблена у вигляді окремого Angular-компонента. Його шаблон містить такі компоненти з бібліотеки Material:

– MatCard для стилізації елемента списку;

– MatSlideToggle для використання готової реалізації перемикача.

Логіка компонента для сторінки конфігурації репозиторіїв реалізована у класі RepositoriesComponent, який містить такі залежності:

– клас ApiService для отримання репозиторіїв користувача;

– клас GithubWebhookService для створення GitHub Webhook для репозиторію, який вибрав користувач;

– клас AuthService для отримання токена доступу поточного користувача.

Компонент має метод onConfigureToggleChanged, який передається вхідним аргументом у компонент MatSlideToggle для реагування на зміни перемикача.

Сторінка для конфігурації репозиторіїв зображена на рисунку 4.3. Код класу RepositoriesComponent та код шаблону сторінки налаштування репозиторіїв наведено у додатку Б.

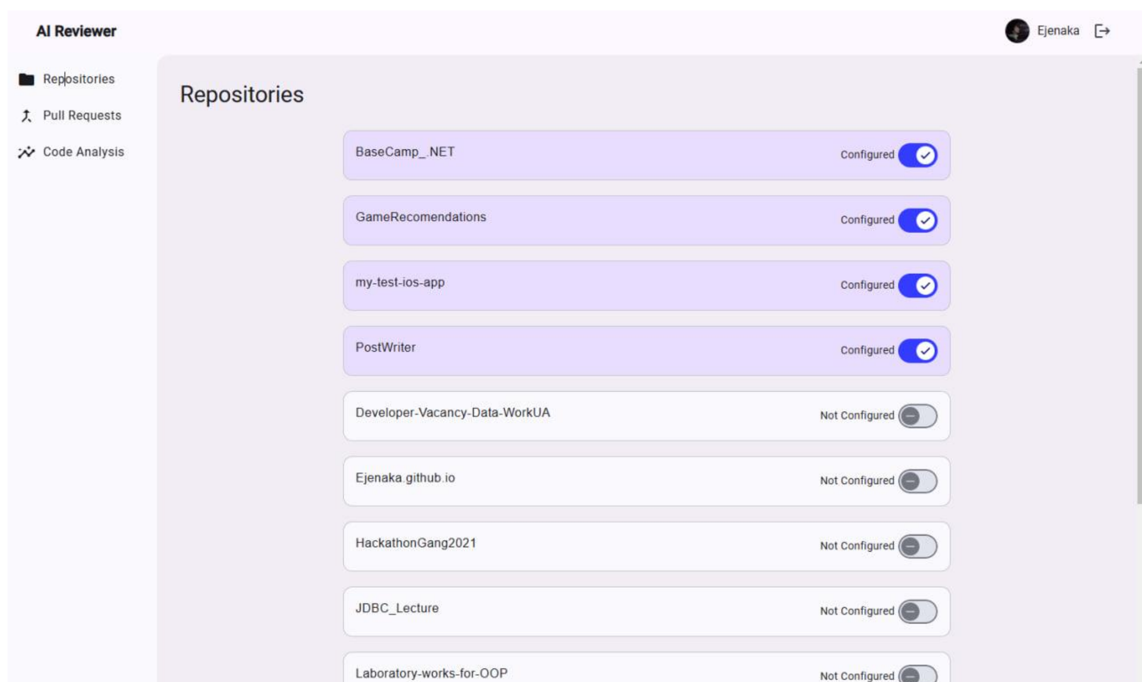


Рисунок 4.3 – Сторінка для конфігурації репозиторіїв

Сторінка зі списком pull request містить список pull requests всіх налаштованих репозиторіїв користувача та статистику аналізу коду для цих pull

requests. Основна мета цієї сторінки – показати користувачу коротку статистику аналізу по всіх pull request репозиторіїв, налаштованих користувачем.

Шаблон сторінки зі списком pull request містить компонент MatCard з бібліотеки Material для стилізації елемента списку, як на сторінці для конфігурації репозиторіїв. Також шаблон містить розроблений компонент статусу аналізу коду. Цей елемент вирішено відокремити в окремий компонент задля повторного використання його у сторінці аналізу коду.

Логіку сторінки зі списком pull request реалізовано у класі PullRequestsComponent, який має лише залежність від класу ApiService для отримання даних про pull requests репозиторіїв користувача та запитів на отримання рекомендацій по аналізу до pull request. Компонент має лише 1 метод getRecommendationsForPullRequest, який повертає запити на отримання рекомендацій, які потім передаються в параметр recommendations$ компонента статусу аналізу коду.

Сторінка зі списком pull request зображена на рисунку 4.4. Код класу PullRequestsComponent та код шаблону сторінки зі списком pull request наведено у додатку Б.

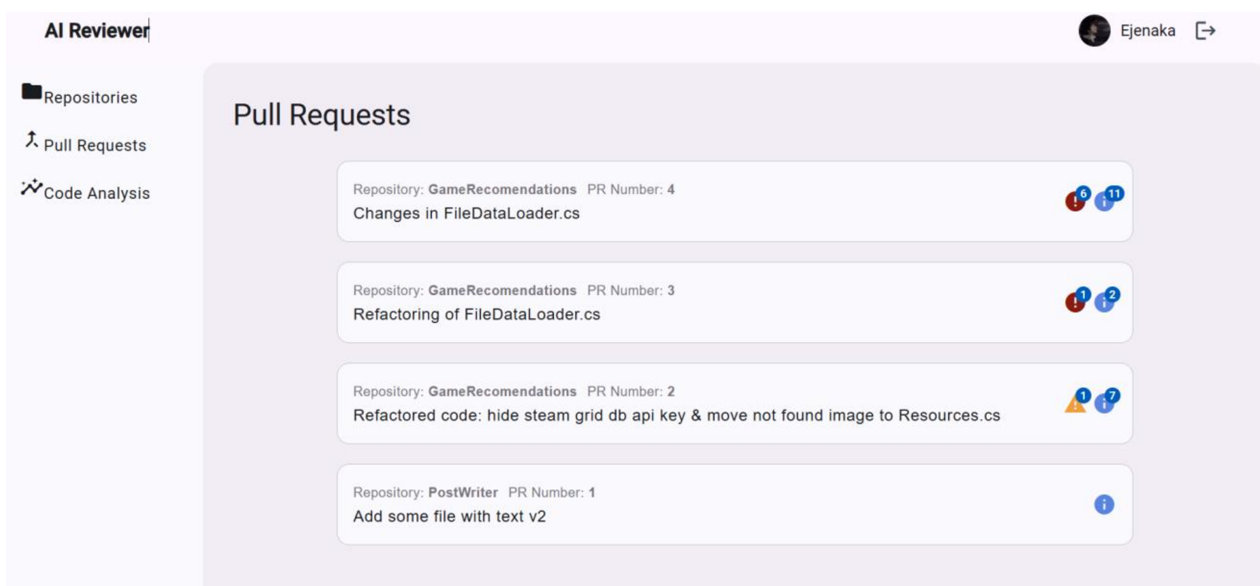


Рисунок 4.4 – Сторінка зі списком pull request

Компонент статусу аналізу коду містить блок для зображення значків, які візуально помічають чи є у аналізі коду певного pull request рекомендації такого типу. Також на цих значках для сторінки зі списком pull request зверху надписано кількість таких рекомендацій. Всього є 3 значків:

– блакитний для рекомендацій типу Optimization, Enhancement та Best Practice;

– помаранчевий для рекомендацій типу Vulnerability;

– червоний для рекомендацій типу Issue.

Шаблон для компонента статусу аналізу коду містить компонент MatIcon для зображення значків.

Логіку для компонента статусу аналізу коду реалізовано у класі ReviewStatusComponent. Компонент містить такі вхідні параметри:

– *recommendations$*: поле типу Observable<Recommendation[]>, який вказує на запити на отримання рекомендацій;

– *singleRecommendation*: поле типу Recommendation, який вказує на рекомендацію.

Якщо запити передаються в параметрі recommendations$, тоді вони виконуються безпосередньо у компоненті статусу аналізу коду. В цьому випадку кількість рекомендацій за її типом відображаються на значках. Якщо лише одна рекомендація передається в параметрі singleRecommendation, тоді лише 1 значок зображено, без лічильника рекомендацій. Параметр recommendations$ використовується для компонента статусу аналізу коду на сторінці зі списком pull request, а параметр singleRecommendation використовується для компонента статусу аналізу коду на сторінці аналізу коду.

Компонент статусу аналізу коду зображено на рисунку 4.4. Можна побачити, що для першого pull request є 1 рекомендація типу Vulnerability та 4 рекомендацій типу Optimization або Enhancement, або Best Practice. Код класу

ReviewStatusComponent та код шаблону компоненту статусу аналізу коду наведено у додатку Б.

Сторінка аналізу коду містить такі UI-елементи:

– заголовок сторінки «Code Analysis»;

– поле вибору репозиторію;

– поле вибору pull request для вибраного репозиторію;

– заголовок pull request, який складається зі тексту «Pull Request», його номер та назви pull request;

– список рекомендацій змінених файлів.

Основною метою сторінки аналізу коду є надати користувачу інтерфейс для перегляду результатів аналізу pull request та надати користувачу можливість комунікації з системою перевірки коду за допомогою коментарів до рекомендацій.

Список рекомендацій змінених файлів є панеллю, яка розгортається при натисканні на неї. Ця панель складається з шляху файлу та з компонента статусу коду для зображення значка, який показує якого типу рекомендація. Панель може бути заблокована, якщо статус рекомендації є «вирішеним» (Resolved). Сама розгорнута панель містить такі UI-елементи:

– текст рекомендації;

– блок з кодом, для якого рекомендація належить;

– кнопки з текстом «Resolve» для зміни статусу рекомендації на «вирішений»;

– блок з коментарями системи перевірки коду та користувача;

– полем вводу коментаря від користувача;

– кнопки з текстом «Comment» для відправки коментаря від користувача, який написано у полі вводу коментаря.

Шаблон для сторінки аналізу коду містить такі компоненти з бібліотеки Material:

– MatSelect для вибору репозиторію та pull request;

– MatExpansion для використання готової реалізації для панелі, яка розгортається при натисканні;

– MatButton для стилізації кнопок;

– MatDivider для використання полоси розділу.

Також шаблон містить компонент Code з бібліотеки ngx-highlightjs для зображення коду. Сам цей компонент за основу використовую популярну бібліотеку для зображення коду highlight.js.

Логіку сторінки аналізу коду реалізовано в класі CodeAnalysisComponent. Цей клас має такі методи:

– onRepositorySelected, який викликається при зміні репозиторію з метою отримання pull requests вибраного репозиторію;

– onPullRequestSelected, який викликається, коли користувач вибирає pull request з метою отримання рекомендацій та зміст файлів, для яких є рекомендація;

– onRecommendationOpened, який передається у компонент панелі MatExpansion з метою отримання коментарів для рекомендації, коли відкривається панель рекомендації;

– onRecommendationResolved, який передається у подію click для кнопки з метою оновлення статусу рекомендації на «вирішений» при натисканні кнопки;

– onCommentInputChanged, який передається у подію input для поля написання коментарю з метою запису коментаря у змінну;

– onCommentSubmitButtonClick, який передається у подію submit для форми заповнення коментарю з метою запису коментарю користувача у систему перевірки коду. Також цей метод після запису коментарю користувача отримує відповідь на нього, та записує його у масив коментарів для відображення відповіді на коментар користувача;

– getFileContentFromFileContents, який викликається для компоненту Code з метою отримання зміст файлу для рекомендації з об’єкту змісту файлів рекомендацій;

– getFileContent, який є приватним та викликається у методі onPullRequestSelected для запису змісту файлів у об’єкт змісту файлів рекомендацій.

Код класу CodeAnalysisComponent та код шаблону сторінки аналізу коду наведено у додатку Б.

Для тестування сторінки аналізу коду було створено pull request для власного репозиторію GameRecomendations, який був налаштований для платформи автоматизованої перевірки коду на сторінці конфігурації репозиторіїв (див. рис. 4.3). Pull request має номер 4, називається «Changes in FileDataLoader.cs» та містить зміну одного файлу, код якого спеціально було змінено для погіршення його якості та впровадження помилок.

Після створення pull request, на сторінці зі списком він з’явився (див. рис. 4.4). За результатом перевірки платформи, цей pull request містить 6 помилок та 11 рекомендацій щодо покращення коду.

На сторінці аналізу коду також зображено частину цих рекомендації на рисунку.

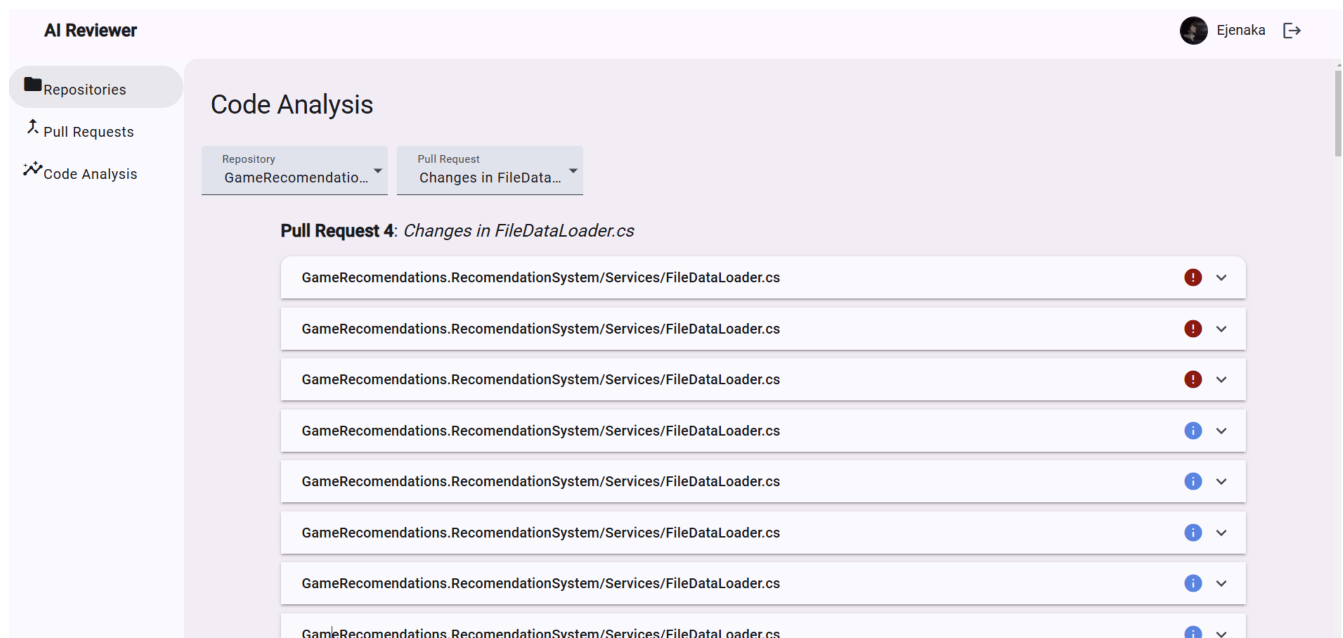


Рисунок 4.5 – Сторінка аналізу коду для створеного pull request

Переглянуто декілька з них та виділено такі рекомендації:

– помилка неправильного використання типів у C#, яка зображена на рисунку 4.6;

– рекомендація щодо правильного використання пробілу в коді, яка зображена на рисунку 4.7;

– помилка при додаванні числа та рядка, яка зображена на рисунку 4.8;

– рекомендація щодо видалення методу, який не використовується, яка зображена на рисунку 4.9.

Рисунок 4.6 – Помилка неправильного використання типів

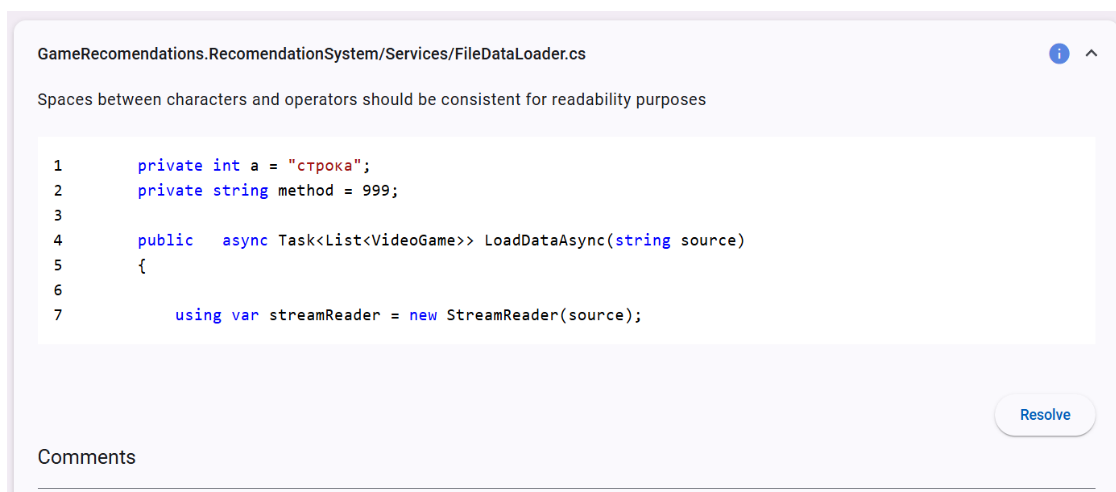


Рисунок 4.7 – Рекомендація щодо правильного використання пробілу в коді

Ці рисунки показують, що система перевірки коду платформи дійсно працює та аналізує код на помилки та рекомендує покращення, які можна зробити в коді.

Для тестування системи коментарів написано коментар щодо пояснення до рекомендації щодо правильного використання пробілу в коді (див. рис. 4.7). Цей коментар та відповідь на нього зображено на рисунку 4.10.

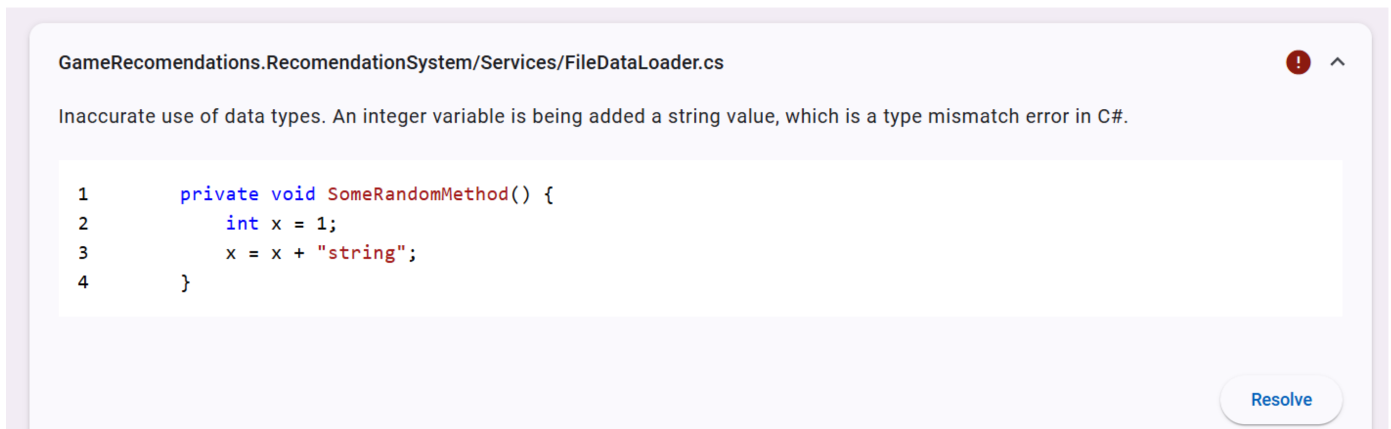


Рисунок 4.8 – Помилка при додаванні числа та рядка

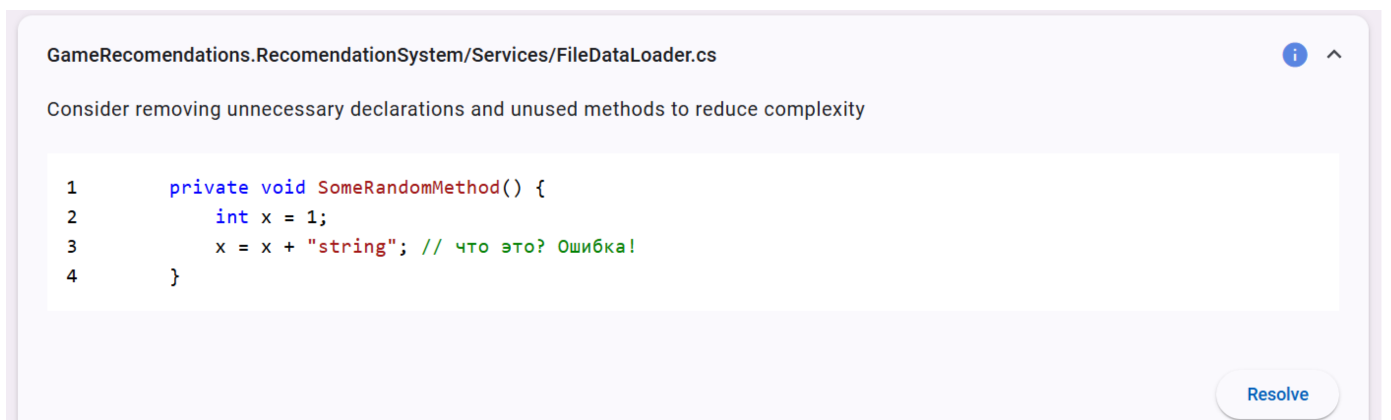


Рисунок 4.9 – Рекомендація щодо видалення методу, який не використовується

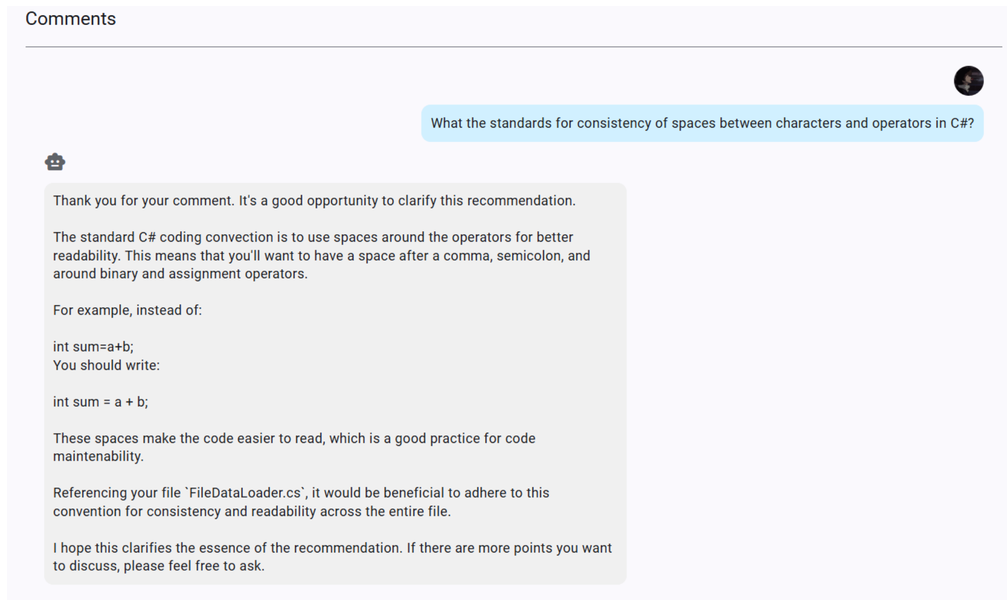


Рисунок 4.10 – Коментар та відповідь на нього

Система детальна відповіла на коментар користувача та пояснила, які стандарти використання пробілів між операторами для мови програмування C#.

Для перевірки механізму зміни статусу рекомендації на «вирішений», натиснуто на кнопку «Resolve» для декількох рекомендацій. Результат цих операцій зображено на рисунку 4.11.

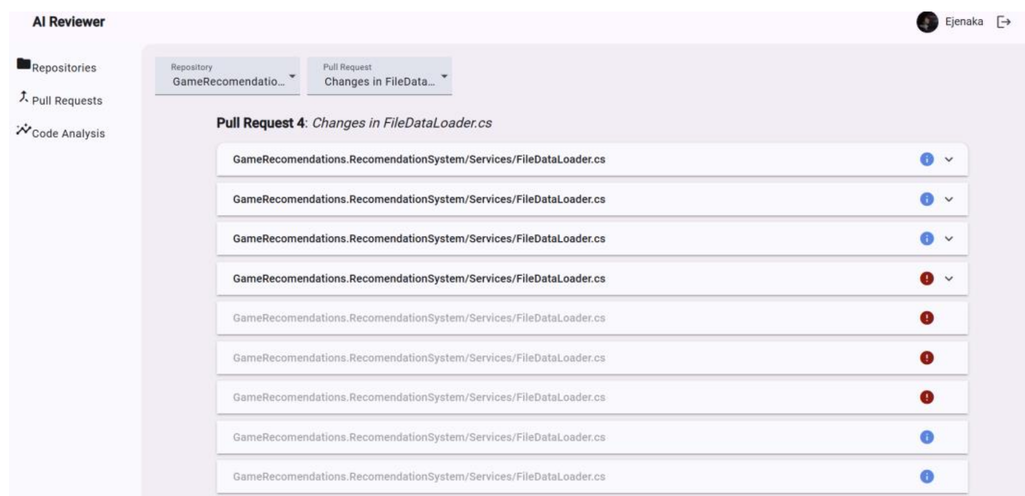


Рисунок 4.11 – «Вирішені» рекомендації

Як можна побачити, на цьому рисунку 5 рекомендацій закриті для перегляду, тому що їхній статус змінено на «вирішений».

Висновки до розділу 4

Цей розділ присвячений безпосередній реалізації та тестуванню програмного забезпечення платформи для автоматизованої перевірки коду, включаючи Back-End, Front-End складові, а також інтеграцію з GitHub. Спочатку було створено .NET рішення з мікросервісною архітектурою, у якому реалізовано механізми аутентифікації та авторизації користувачів через GitHub OAuth. Це дозволило користувачам безпечно отримувати доступ до можливостей платформи.

Мікросервіс Pull Request API відповідає за отримання інформації про користувача, репозиторії та pull requests. Завдяки інтеграції з бібліотекою Octokit.NET здійснюється отримання даних безпосередньо з GitHub. Впроваджено окремі сервіси для роботи з користувачами, репозиторіями та pull requests, а також для управління рекомендаціями та коментарями до коду. Це забезпечує гнучкість, модульність та розширюваність системи.

Мікросервіс Reviewer API забезпечує взаємодію з ChatGPT для аналізу коду у pull requests та надання рекомендацій. Механізм побудови промптів та налаштування відповідей у форматі JSON дозволяє системі структурувати результати та зберігати їх у базі даних. Для взаємодії з БД використано підхід Code First і Entity Framework Core, що спрощує керування міграціями та еволюцію схеми даних.

Мікросервіс Event Handler інтегрується з GitHub Webhooks для автоматичного запуску перевірки при створенні чи оновленні pull requests. Завдяки цьому процес виявляється максимально автоматизованим.

На Front-End боці використано Front-End фреймворк Angular із застосуванням компонентів з бібліотеки Material. Створено сторінку авторизації, сторінку для конфігурації репозиторіїв, сторінку зі списком pull request, сторінку аналізу коду. В результаті реалізовано повноцінну платформу, яка дозволяє користувачам отримувати детальні рекомендації щодо покращення якості коду.

ВИСНОВКИ

З ростом IT-індустрії швидкість розробки проєктів значно підвищилась, терміни виконання проєктів зменшується. Підтримувати якість та чистоту програмного коду стає значно важко. Тому виникають нові виклики: потреба у швидкій та ефективній перевірці програмного коду, мінімізація людського фактору або взагалі його відсутність та зменшення кількості помилок в програмному коді. Тому розробка платформи автоматизованої перевірки коду стає все більш актуальною.

У ході виконання кваліфікаційної роботи було реалізовано поставлену мету – підвищити ефективність процесу перевірки коду за допомогою платформи, інтегрованої з технологіями штучного інтелекту, зокрема впровадження генеративною моделлю штучного інтелекту ChatGPT.

Всі поставлені завдання КМР були виконані, а саме:

– проаналізовано існуючі рішення для автоматизованої перевірки коду та виявити їх обмеження. Встановлено, що на ринку існує велика кількість інструментів для статичного аналізу коду, які забезпечують автоматичну перевірку на наявність помилок, потенційних вразливостей, відхилень від стандартів стилю та інших проблем. Однак, більшість з них має обмеження, оскільки вони працюють лише на основі заздалегідь визначених правил і часто не враховують контекст розробки;

– досліджено можливість використання технологій штучного інтелекту для аналізу та перевірки коду. Огляд існуючих інструментів перевірки коду показав, що вони мають певні недоліки, зокрема, обмежену можливість розуміння контексту змін у коді та проблем із логікою, що вимагає більш інтелектуального підходу. Використання штучного інтелекту відкриває нові можливості для аналізу коду, дозволяючи виявляти більш складні помилки та надавати рекомендації, що враховують контекст коду;

– розроблено методи та засоби інтеграції технологій штучного інтелекту у процес автоматизованої перевірки коду. Розглянуто можливості інтеграції платформи автоматизованої перевірки коду з моделлю ChatGPT. Показано, що використання ChatGPT дозволяє значно розширити можливості статичного аналізу коду, забезпечуючи кращу взаємодію з розробниками та можливість надавати більш релевантні рекомендації. Інтеграція з моделлю ChatGPT-4 розроблена через інтеграцію з Open.AI SDK, який надає програмний інтерфейс для взаємодії з моделлю;

– реалізовано прототип платформи для автоматизованої перевірки коду з використанням технологій штучного інтелекту. Розроблено працездатний прототип платформи, який складається з двох основних частин: Back-End з мікросервісною архітектурою на платформі .NET та Front-End з використанням вебфреймворку Angular з використанням готових компонентів з бібліотеки Material. Прототип містить інтерактивний, мінімалістичний користувацький інтерфейс, який є дуже зрозумілим для користувача в контексті платформи автоматизованої аналізу коду;

– оцінено ефективність розробленої платформи у покращенні якості коду. У процесі тестування платформи було продемонстровано виявлення помилок різного типу у тестовому коді. Але існує можливість покращення системи перевірки коду шляхом впровадження інтелектуального методу надання зміст файлів для моделі ChatGPT.

Запропоноване рішення для виконання КМР не лише допомагає прискорити виявлення помилок, а й надає рекомендації з урахуванням стилю та логіки проєкту, покращуючи ефективність роботи команди розробників та загальну якість програмного забезпечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Бардовський Є. В., Горбань Г. В. Платформа для автоматизованої перевірки коду з використанням технологій штучного інтелекту. *Моделі, методи та засоби програмної інженерії* : тези доп. Всеукр, наук.-пр. конф. «Могилянські читання – 2024» : Технічні науки. Миколаїв, 06-10 лист. 2024 р. : Чорноморський національний університет ім. Петра Могили, 2024. С. 27-31.

2. Devon H. O'Dell. The Debugging Mindset : Understanding the psychology of learning strategies leads to effective problem-solving skills. 2009. Vol. 15, P. 71-90. DOI: 10.1145/3055301.3068754.

3. Comprehensive Guide to Code Quality: Best Practices and Tools. *Coding Sans*: вебсайт. URL: https://codingsans.com/blog/code-quality (дата звернення: 09.09.2024).

4. Walker, A., Coffey, M., Tisnovsky, P., Cerny, T. On Limitations of Modern Static Analysis Tools. *Information Science and Applications. Lecture Notes in Electrical Engineering*. 2020. Vol 621. Springer, Singapore. DOI: 10.1007/978-981-15-1465-4_57.

5. Understanding the Code Review Process. *SmartBear*: вебсайт. URL: https://smartbear.com/learn/code-review/guide-to-code-review-process (дата звернення 10.09.2024).

6. A 6-Step Code Review Process to Improve Code and Collaboration. *Swimm*: вебсайт. URL: https://swimm.io/learn/code-reviews/a-6-step-code-review-process-to-improve-code-and-collaboration (дата звернення 10.09.2024).

7. The ancient origins of code review. *Graphit*: вебсайт. URL: https://graphite.dev/blog/the-ancient-origins-of-code-review (дата звернення 12.09.2024).

8. Code Review vs. Code Walkthrough vs. Code Inspection. *Anar*: вебсайт. URL: https://anarsolutions.com/code-review-vs-code-walkthrough (дата звернення 13.09.2024).

9. Concurrent Versions System. *Wikipedia*: вебсайт. URL: https://en.wikipedia.org/wiki/Concurrent_Versions_System (дата звернення 13.09.2024).

10. A Short History of Git. *Git*: вебсайт. URL: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git (дата звернення 15.09.2024).

11. SonarQube. *SonarSource*: вебсайт. URL: https://www.sonarsource.com/products/sonarqube (дата звернення 18.09.2024).

12. What is Danger?. *GitHub*: вебсайт. URL: https://github.com/danger/danger (дата звернення 18.09.2024).

13. ESLint. *ESLint*: вебсайт. URL: https://eslint.org (дата звернення 20.09.2024).

14. Diksha K., Aditya K., Kiran K. Natural language processing: state of the art, current trends and challenges. *Multimed Tools Appl*. 2023. Vol. 82, P. 3713-3744. DOI: 10.1007/s11042-022-13428-4

15. Bansal, G., Chamola, V., Hussain, A. Transforming Conversations with AI—A Comprehensive Study of ChatGPT. *Cogn Comput*. 2024. Vol. 16. P. 2487–2510. DOI: 10.1007/s12559-023-10236-2.

16. IDEF0. *Wikipedia*: вебсайт. URL: https://uk.wikipedia.org/wiki/IDEF0 (дата звернення 23.09.2024).

17. Що таке мікросервісна архітектура: значення, складові, переваги. *Wezom*: вебсайт. URL: https://wezom.com.ua/ua/blog/scho-take-mikroservisna-arhitektura-znachennya-skladovi-perevagi (дата звернення 26.09.2024).

18. Relational database. *Wikipedia*: вебсайт. URL: https://en.wikipedia.org/wiki/Relational_database (дата звернення 02.10.2024).

19. C#. The modern, innovative, open-source programming language for building all your apps. *Microsoft*: вебсайт. URL: https://dotnet.microsoft.com/en-us/languages/csharp (дата звернення 05.10.2024).

20. What is Java technology and why do I need it? *Java*: вебсайт. URL: https://www.java.com/en/download/help/whatis_java.html (дата звернення 05.10.2024).

21. Programming Language Hall of Fame. *TIOBE*: вебсайт. URL: https://www.tiobe.com/tiobe-index (дата звернення 05.10.2024).

22. TypeScript. *TypeScript Lang*: вебсайт. URL: https://www.typescriptlang.org (дата звернення 05.10.2024).

23. Як і для чого вивчати Golang. Переваги і недоліки мови. *DOU*: вебсайт. URL: https://dou.ua/forums/topic/41933 (дата звернення 05.10.2024).

24. Python: About. *Python*: вебсайт. https://www.python.org/about (дата звернення 05.10.2024).

25. PHP. *Wikipedia*: вебсайт. https://uk.wikipedia.org/wiki/PHP (дата звернення 05.10.2024).

26. Web Framework Benchmarks. *Tech Empower*: вебсайт. URL: https://www.techempower.com/benchmarks/#hw=ph&test=fortune§ion=data-r22 (дата звернення 08.10.2024).

27. About webhooks. *GitHub Docs*: вебсайт. URL: https://docs.github.com/en/webhooks/about-webhooks (дата звернення 08.10.2024).

28. Entity Framework documentation hub. *Microsoft Learn*: вебсайт. URL: https://learn.microsoft.com/en-us/ef (дата звернення 10.10.2024).

29. React. The library for web and native user interfaces. *React*: вебсайт. URL: https://react.dev (дата звернення 11.10.2024).

30. What is Angular? *React*: вебсайт. URL: https://angular.dev/overview (дата звернення 11.10.2024).

31. Vue. The Progressive JavaScript Framework. *Vue*: вебсайт. URL: https://angular.dev/overview (дата звернення 11.10.2024).

32. Onion Architecture. *Medium*: вебсайт. URL: https://medium.com/expedia-group-tech/onion-architecture-deed8a554423 (дата звернення 15.10.2024).

ДОДАТОК А

Реалізація Back-End частини

Код класу GithubUserAuthenticationHandler

```
namespace GithubPullRequestReviewer.PullRequestAPI.Authorization
{
    public class GithubUserAuthenticationHandler : AuthenticationHandler<AuthenticationSchemeOptions>
    {
        private readonly ITokenService _tokenService;

        public GithubUserAuthenticationHandler(
            IOptionsMonitor<AuthenticationSchemeOptions> options,
            ILoggerFactory logger,
            UrlEncoder encoder,
            ITokenService tokenService)
            : base(options, logger, encoder)
        {
            _tokenService = tokenService;
        }

        protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
        {
            var token = Request.Headers["access_token"].FirstOrDefault();

            if (await _tokenService.ValidateAndSetTokenAsync(token))
            {
                var claims = new[] { new Claim(ClaimTypes.Name, "CustomUser") };
                var identity = new ClaimsIdentity(claims, Scheme.Name);
                var principal = new ClaimsPrincipal(identity);
                var ticket = new AuthenticationTicket(principal, Scheme.Name);

                return AuthenticateResult.Success(ticket);
            }
            else
            {
                return AuthenticateResult.Fail("Invalid token.");
            }
        }
    }
}
```

Код класу GithubTokenService

```
public class GithubTokenService : ITokenService
    {
        private readonly GitHubClient _githubClient;
        private readonly IOptions<GithubOAuthAppOptions> _options;

        public GithubTokenService(GitHubClient githubClient, IOptions<GithubOAuthAppOptions> options)
        {
            _githubClient = githubClient;
            _options = options;
        }

        public string GetToken()
        {
            return _githubClient.Credentials.GetToken();
        }
```

```
        public async Task<bool> ValidateAndSetTokenAsync(string token)
        {
            if (string.IsNullOrWhiteSpace(token))
            {
                return false;
            }

            _githubClient.Credentials = new Credentials(token);

            var currentUser = await _githubClient.User.Current();

            return currentUser != null;
        }
    }
```

Код класу UserController

```
namespace GithubPullRequestReviewer.PullRequestAPI.Controllers
{
    [ApiController]
    [Route("api/users")]
    public class UserController : Controller
    {
        private readonly IUserService _userService;
        private readonly IRepositoryService _repositoriesService;

        public UserController(IUserService userService, IRepositoryService repositoriesService)
        {
            _userService = userService;
            _repositoriesService = repositoriesService;
        }

        [HttpGet]
        [Route("current")]
        [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
        public async Task<User> GetUserInfo()
        {
            return await _userService.GetCurrentUserAsync();
        }

        [HttpGet]
        [Route("current/repositories")]
        [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
        public async Task<IReadOnlyList<Repository>> GetUserRepositoriesAsync()
        {
            return await _repositoriesService.GetRepositoriesForCurrentUserAsync();
        }

        [HttpGet]
        [Route("auth/url")]
        public string GetLoginUrl()
        {
            return _userService.GetUserAuthLink();
        }

        [HttpGet]
        [Route("auth/token")]
        public async Task<string> GetAccessTokenAsync([FromQuery] string code)
        {
            return await _userService.GetUserAccessTokenAsync(code);
        }
    }
```

Код класу UserService

```
public class UserService : IUserService
{
    private readonly Octokit.GitHubClient _githubClient;
    private readonly IOptions<GithubOAuthAppOptions> _githubOAuthOptions;

    public UserService(Octokit.GitHubClient githubClient, IOptions<GithubOAuthAppOptions> githubOAuthOptions)
    {
        _githubClient = githubClient;
        _githubOAuthOptions = githubOAuthOptions;
    }

    public async Task<User> GetCurrentUserAsync()
    {
        var currentUser = await _githubClient.User.Current();
        return currentUser.ToDomain();
    }

    public string GetUserAuthLink()
    {
        var authRequest = new OauthLoginRequest(_githubOAuthOptions.Value.ClientId)
        {
            Scopes = { "user", "repo" }
        };
        return _githubClient.Oauth.GetGitHubLoginUrl(authRequest).ToString();
    }

    public async Task<string> GetUserAccessTokenAsync(string code)
    {
        var token = await _githubClient.Oauth.CreateAccessToken(new OauthTokenRequest(
            _githubOAuthOptions.Value.ClientId,
            _githubOAuthOptions.Value.Secret,
            code));
        return token.AccessToken;
    }
}
```

Код класу UserService

```
public class RepositoryService : IRepositoryService
{
    private readonly Octokit.GitHubClient _githubClient;

    public RepositoryService(Octokit.GitHubClient githubClient)
    {
        _githubClient = githubClient;
    }

    public async Task<IReadOnlyList<Repository>> GetRepositoriesForCurrentUserAsync()
    {
        var currentUser = await _githubClient.User.Current();
        var repositories = await _githubClient.Repository.GetAllForUser(currentUser.Login);

        return repositories.Select(x => x.ToDomain()).ToList();
    }

    public async Task<Repository> GetRepositoryById(long repositoryId)
    {
        var repository = await _githubClient.Repository.Get(repositoryId);

        return repository.ToDomain();
    }
}
```

Код класу PullRequestController

```
[ApiController]
[Route("api")]
public class PullRequestController : Controller
{
    private readonly IPullRequestService _pullRequestService;

    public PullRequestController(IPullRequestService pullRequestService)
    {
        _pullRequestService = pullRequestService;
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<PullRequest> GetPullRequestAsync(long repositoryId, int pullRequestNumber)
    {
        return await _pullRequestService.GetPullRequestAsync(repositoryId, pullRequestNumber);
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<IList<PullRequest>> GetAllPullRequestForRepositoriesAsync(long repositoryId)
    {
        return await _pullRequestService.GetAllPullRequestsForRepositoryAsync(repositoryId);
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}/diff")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<string> GetPullRequestDiffContentAsync(long repositoryId, int pullRequestNumber)
    {
        return await _pullRequestService.GetPullRequestDiffContentAsync(repositoryId, pullRequestNumber);
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}/files")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<IEnumerable<PullRequestFile>> GetPullRequestFileContentAsync(long repositoryId, int
pullRequestNumber)
    {
        return await _pullRequestService.GetPullRequestFilesAsync(repositoryId, pullRequestNumber);
    }
}
```

Код класу PullRequestService

```
public class PullRequestService : IPullRequestService
{
    private Octokit.GitHubClient _githubClient;
    private IRepositoryService _repositoryService;

    public PullRequestService(Octokit.GitHubClient githubClient, IRepositoryService repositoryService)
    {
        _githubClient = githubClient;
```

```
        _repositoryService = repositoryService;
    }

    public async Task<PullRequest> GetPullRequestAsync(long repositoryId, int pullRequestNumber)
    {
        var pullRequest = await _githubClient.PullRequest.Get(repositoryId, pullRequestNumber);
        var repository = await _repositoryService.GetRepositoryById(repositoryId);

        return pullRequest.ToDomain(repository);
    }

    public async Task<IList<PullRequest>> GetAllPullRequestsForRepositoryAsync(long repositoryId)
    {
        var pullRequests = await _githubClient.PullRequest.GetAllForRepository(repositoryId);
        var repository = await _repositoryService.GetRepositoryById(repositoryId);

        return pullRequests.Select(p => p.ToDomain(repository)).ToList();
    }

    public async Task<string> GetPullRequestDiffContentAsync(long repositoryId, int pullRequestNumber)
    {
        var pullRequest = await GetPullRequestAsync(repositoryId, pullRequestNumber);

        using var httpClient = new HttpClient();
        var response = await httpClient.GetAsync(pullRequest.DiffUrl);
        var content = await response.Content.ReadAsStringAsync();

        return content;
    }

    public async Task<IEnumerable<PullRequestFile>> GetPullRequestFilesAsync(long repositoryId, int
pullRequestNumber)
    {
        return await _githubClient.Repository.PullRequest.Files(repositoryId, pullRequestNumber);
    }
}
```

Код класу ReviewController

```
[ApiController]
[Route("api")]
public class ReviewController : Controller
{
    private readonly IReviewService _reviewService;

    public ReviewController(IReviewService reviewService)
    {
        _reviewService = reviewService;
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}/recommendations")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<IEnumerable<Recommendation>> GetRecommendationsAsync(long repositoryId, int
pullRequestNumber)
    {
        return await _reviewService.GetRecommendationsAsync(repositoryId, pullRequestNumber);
    }
```

```
    [HttpGet]
    [Route("recommendations/{recommendationId}")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<Recommendation> GetRecommendationByIdAsync(int recommendationId)
    {
        return await _reviewService.GetRecommendationByIdAsync(recommendationId);
    }

    [HttpPatch]
    [Route("recommendations")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task UpdatePullRequestRecommendationAsync([FromBody] Recommendation recommendation)
    {
        await _reviewService.UpdatePullRequestRecommendationAsync(recommendation);
    }

    [HttpPost]
    [Route("reviews")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task CreatePullRequestReviewAsync([FromBody] CreateReviewDto createReviewDto)
    {
        await _reviewService.CreatePullRequestReviewAsync(createReviewDto);
    }
}
```

Код класу Recommendation

```
public class Recommendation
{
    public int Id { get; set; }
    public string Content { get; set; }
    public string FileName { get; set; }
    public int[] CodeLines { get; set; }
    public long RepositoryId { get; set; }
    public int PullRequestNumber { get; set; }
    public RecommendationType Type { get; set; }
    public RecommendationStatus Status { get; set; }
    public DateTime CreatedAt { get; set; }
}
```

Код класу CreateReviewDto

```
public record CreateReviewDto
{
    public long RepositoryId { get; set; }
    public int PullRequestNumber { get; set; }
    public List<CreateReviewDtoItem> Issues { get; set; }
    public List<CreateReviewDtoItem> Vulnerabilities { get; set; }
    public List<CreateReviewDtoItem> Optimization { get; set; }
    public List<CreateReviewDtoItem> Enhancements { get; set; }
    public List<CreateReviewDtoItem> BestPractices { get; set; }
}
```

Код класу ReviewService

```
public class ReviewService : IReviewService
{
    private readonly PullRequestReviewerDbContext _dbContext;

    public ReviewService(PullRequestReviewerDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task CreatePullRequestReviewAsync(CreateReviewDto createReviewDto)
    {
        createReviewDto.Issues ??= [];
        createReviewDto.Vulnerabilities ??= [];
        createReviewDto.BestPractices ??= [];
        createReviewDto.Enhancements ??= [];
        var recommendations = createReviewDto.Issues
            .Select(x => new { Recomendation = x, Type = RecommendationType.Issue })
            .Concat(createReviewDto.Vulnerabilities.Select(x => new { Recomendation = x, Type = RecommendationType.Vulnerability }))
            .Concat(createReviewDto.Enhancements.Select(x => new { Recomendation = x, Type = RecommendationType.Enhancement }))
            .Concat(createReviewDto.Optimization.Select(x => new { Recomendation = x, Type = RecommendationType.Optimization }))
            .Concat(createReviewDto.BestPractices.Select(x => new { Recomendation = x, Type = RecommendationType.BestPractice }));
        var recommendationsDb = recommendations.Select(x => new RecommendationEntity
        {
            RepositoryId = createReviewDto.RepositoryId,
            PullRequestNumber = createReviewDto.PullRequestNumber,
            CodeLines = [x.Recomendation.BeginsAtCodeLine, x.Recomendation.EndsAtCodeLine],
            Content = x.Recomendation.Description,
            FileName = x.Recomendation.File,
            Type = x.Type,
            Status = RecommendationStatus.Open,
            CreatedAt = DateTime.Now,
        });
        await _dbContext.AddRangeAsync(recommendationsDb);
        await _dbContext.SaveChangesAsync();
    }

    public async Task UpdatePullRequestRecommendationAsync(Recommendation recommendation)
    {
        _dbContext.Update(recommendation.ToDb());
        await _dbContext.SaveChangesAsync();
    }

    public async Task<Recommendation> GetRecommendationByIdAsync(int recommendationId)
    {
        var recommendation = await _dbContext.Recommendations.FirstOrDefaultAsync(r => r.Id == recommendationId);

        return recommendation.ToDomain();
    }

    public async Task<IEnumerable<Recommendation>> GetRecommendationsAsync(long repositoryId, int pullRequestNumber)
    {
        var recommendations = await _dbContext.Recommendations
            .Where(r => r.RepositoryId == repositoryId && r.PullRequestNumber == pullRequestNumber)
            .ToListAsync();

        return recommendations.Select(r => r.ToDomain());
    }
}
```

Код класу PullRequestReviewerDbContext

```
public class PullRequestReviewerDbContext : DbContext
{
    public DbSet<RecommendationEntity> Recommendations { get; set; }
```

```
    public DbSet<CommentEntity> Comments { get; set; }

    public PullRequestReviewerDbContext() { }

    public PullRequestReviewerDbContext(DbContextOptions<PullRequestReviewerDbContext> options) : base(options) {
}

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
optionsBuilder.UseNpgsql("Host=localhost:5432;Username=postgres;Password=qwerty;Database=GithubPullRequestRevie
wer");
    }
}
```

Код класу InitialCreate

```
public partial class InitialCreate : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Recommendations",
            columns: table => new
            {
                Id = table.Column<int>(type: "integer", nullable: false)
                    .Annotation("Npgsql:ValueGenerationStrategy", NpgsqlValueGenerationStrategy.IdentityByDefaultColumn),
                Content = table.Column<string>(type: "text", nullable: true),
                FileName = table.Column<string>(type: "text", nullable: true),
                CodeLines = table.Column<int[]>(type: "integer[]", nullable: true),
                RepositoryId = table.Column<long>(type: "bigint", nullable: false),
                PullRequestNumber = table.Column<int>(type: "integer", nullable: false),
                Type = table.Column<int>(type: "integer", nullable: false),
                Status = table.Column<int>(type: "integer", nullable: false),
                CreatedAt = table.Column<DateTime>(type: "timestamp without time zone", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Recommendations", x => x.Id);
            });

        migrationBuilder.CreateTable(
            name: "Comments",
            columns: table => new
            {
                Id = table.Column<int>(type: "integer", nullable: false)
                    .Annotation("Npgsql:ValueGenerationStrategy", NpgsqlValueGenerationStrategy.IdentityByDefaultColumn),
                RecommendationId = table.Column<int>(type: "integer", nullable: false),
                Text = table.Column<string>(type: "text", nullable: true),
                IsFromUser = table.Column<bool>(type: "boolean", nullable: false),
                CreatedAt = table.Column<DateTime>(type: "timestamp without time zone", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Comments", x => x.Id);
                table.ForeignKey(
                    name: "FK_Comments_Recommendations_RecommendationId",
                    column: x => x.RecommendationId,
```

```
                principalTable: "Recommendations",
                principalColumn: "Id",
                onDelete: ReferentialAction.Cascade);
        });

    migrationBuilder.CreateIndex(
        name: "IX_Comments_RecommendationId",
        table: "Comments",
        column: "RecommendationId");
  }

  /// <inheritdoc />
  protected override void Down(MigrationBuilder migrationBuilder)
  {
    migrationBuilder.DropTable(
        name: "Comments");

    migrationBuilder.DropTable(
        name: "Recommendations");
  }
}
```

Код класу CommentController

```
[ApiController]
[Route("api")]
public class CommentController : Controller
{
  private readonly ICommentService _commentService;

  public CommentController(ICommentService commentService)
  {
    _commentService = commentService;
  }

  [HttpGet]
  [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}/review/comments")]
  [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
  public async Task<IList<Comment>> GetCommentsForPullRequestAsync(long repositoryId, int pullRequestNumber)
  {
    return await _commentService.GetCommentsForPullRequestAsync(repositoryId, pullRequestNumber);
  }

  [HttpGet]
  [Route("recommendations/{recommendationId:int}/comments")]
  [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
  public async Task<IList<Comment>> GetCommentsForRecommendationAsync(int recommendationId)
  {
    return await _commentService.GetCommentsForRecommendationAsync(recommendationId);
  }

  [HttpPost]
  [Route("comments")]
  [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
  public async Task CreateCommentForRecommendationAsync([FromBody] CreateCommentRequest
createCommentRequest)
  {
    await _commentService.CreateCommentForRecommendationAsync(
```

```
            createCommentRequest.Text,
            createCommentRequest.IsFromUser,
            createCommentRequest.RecommendationId);
    }
}
```

Код класу Comment

```
public class Comment
{
    public int Id { get; set; }
    public int RecommendationId { get; set; }
    public string Text { get; set; }
    public bool IsFromUser { get; set; }
    public DateTime CreatedAt { get; set; }
}
```

Код класу CommentService

```
public class CommentService : ICommentService
{
    private readonly PullRequestReviewerDbContext _dbContext;

    public CommentService(PullRequestReviewerDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<IList<Comment>> GetCommentsForPullRequestAsync(long repositoryId, int pullRequestNumber)
    {
        var commentsDb = await _dbContext.Comments
            .Include(x => x.Recommendation)
            .Where(c => c.Recommendation.RepositoryId == repositoryId && c.Recommendation.PullRequestNumber ==
pullRequestNumber)
            .ToListAsync();

        var comments = commentsDb.Select(c => c.ToDomain()).ToList();

        return comments;
    }

    public async Task<IList<Comment>> GetCommentsForRecommendationAsync(int recommendationId)
    {
        var commentsDb = await _dbContext.Comments
            .Where(c => c.RecommendationId == recommendationId)
            .ToListAsync();

        return commentsDb.Select(c => c.ToDomain()).ToList();
    }

    public async Task CreateCommentForRecommendationAsync(string text, bool isFromUser, int recommendationId)
    {
        var commentDb = new CommentEntity
        {
            Text = text,
            IsFromUser = isFromUser,
            RecommendationId = recommendationId,
```

```
            CreatedAt = DateTime.Now
        };

        await _dbContext.Comments.AddAsync(commentDb);
        await _dbContext.SaveChangesAsync();
    }
}
```

Код класу ReviewerController

```
public class ReviewerController : Controller
{
    private readonly IPullRequestReviewer _pullRequestReviewer;

    public ReviewerController(IPullRequestReviewer pullRequestReviewer)
    {
        _pullRequestReviewer = pullRequestReviewer;
    }

    [HttpGet]
    [Route("repositories/{repositoryId}/pull-requests/{pullRequestNumber}/review")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<ReviewResultResponse> ReviewPulRequestAsync(long repositoryId, int pullRequestNumber)
    {
        var reviewResult = await _pullRequestReviewer.ReviewPullRequestAsync(repositoryId, pullRequestNumber);
        return reviewResult;
    }

    [HttpPost]
    [Route("comments")]
    [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
    public async Task<Comment> AddCommentForRecommendationAndGetResponseAsync([FromBody]
CreateCommentRequest createCommentRequest)
    {
        return await
_pullRequestReviewer.AddCommentForRecommendationAndGetResponseAsync(createCommentRequest);
    }
}
```

Код класу ChatGptModelProvider

```
public class ChatGptModelProvider : IGenerativeModelProvider
{
    private readonly ChatClient _chatClient;

    public ChatGptModelProvider(string apiKey)
    {
        _chatClient = new ChatClient("gpt-4", apiKey);
    }

    public async Task<string> SendMessageAsync(string text)
    {
        var chatResult = await _chatClient.CompleteChatAsync(text);
        return chatResult.Value.Content[0].Text;
```

```
    }

    public async Task<string> SendMessagesAsync(IEnumerable<ChatMessage> messages)
    {
        var chatResult = await _chatClient.CompleteChatAsync(messages);
        return chatResult.Value.Content.Last().Text;

}
```

Код класу PullRequestReviewer

```
public class PullRequestReviewer : IPullRequestReviewer
{
    private readonly IGenerativeModelProvider _generativeModelProvider;
    private readonly IPullRequestApiClient _pullRequestApiClient;
    private readonly IReviewApiClient _reviewApiClient;
    private readonly ICommentApiClient _commentApiClient;

    public PullRequestReviewer(
        IGenerativeModelProvider generativeModelProvider,
        IPullRequestApiClient pullRequestApiClient,
        IReviewApiClient reviewApiClient,
        ICommentApiClient commentApiClient)
    {
        _generativeModelProvider = generativeModelProvider;
        _pullRequestApiClient = pullRequestApiClient;
        _reviewApiClient = reviewApiClient;
        _commentApiClient = commentApiClient;
    }

    public async Task<ReviewResultResponse> ReviewPullRequestAsync(long repositoryId, int pullRequestNumber)
    {
        var pullRequestDetails = await _pullRequestApiClient.GetPullRequestAsync(repositoryId, pullRequestNumber);
        var pullRequestDiffContent = await _pullRequestApiClient.GetPullRequestDiffContentAsync(repositoryId,
pullRequestNumber);

        var reviewRequestPrompt = Prompts.BuildPromptForReview(pullRequestDetails.Name, pullRequestDiffContent);
        var modelResponse = await _generativeModelProvider.SendMessageAsync(reviewRequestPrompt);

        if (string.IsNullOrEmpty(modelResponse))
        {
            return null;
        }

        var reviewResult = JsonSerializer.Deserialize<ReviewResultResponse>(modelResponse, new JsonSerializerOptions {
PropertyNameCaseInsensitive = true });

        var createReviewRequest = new CreateReviewDto()
        {
            PullRequestNumber = pullRequestNumber,
            RepositoryId = repositoryId,
            Issues = reviewResult.Issues.Select(i => i.ToDto()).ToList(),
```

```
            Vulnerabilities = reviewResult.Vulnerabilities.Select(i => i.ToDto()).ToList(),
            Optimizations = reviewResult.Optimization.Select(i => i.ToDto()).ToList(),
            Enhancements = reviewResult.Enhancements.Select(i => i.ToDto()).ToList(),
            BestPractices = reviewResult.BestPractices.Select(i => i.ToDto()).ToList(),
        };

        await _reviewApiClient.CreatePullRequestReviewAsync(createReviewRequest);

        return reviewResult;
    }

    public async Task<Comment> AddCommentForRecommendationAndGetResponseAsync(CreateCommentRequest
createCommentRequest)
    {
        var recommendationId = createCommentRequest.RecommendationId;
        var recommendation = await _reviewApiClient.GetRecommendationByIdAsync(recommendationId);
        var recommendationComments = await
_commentApiClient.GetCommentsForRecommendationAsync(recommendationId);

        var messagesForGenerativeModel = new List<ChatMessage>()
        {
            new UserChatMessage(Prompts.CommentPrompt),
            new SystemChatMessage(Prompts.CommentPromptModelResponse),
        };

        messagesForGenerativeModel.AddRange(recommendationComments
            .OrderBy(c => c.CreatedAt)
            .Select<Comment, ChatMessage>(c => c.IsFromUser ? new UserChatMessage(c.Text) : new
SystemChatMessage(c.Text))
        );

        var userCommentRequest = Prompts.BuildCommentRequest(
            createCommentRequest.Text,
            recommendation.FileName,
            recommendation.Content);

        messagesForGenerativeModel.Add(new UserChatMessage(userCommentRequest));

        var modelResponse = await _generativeModelProvider.SendMessagesAsync(messagesForGenerativeModel);
        var modelCreateCommentRequest = new CreateCommentRequest(modelResponse, false, recommendationId);

        await _commentApiClient.CreateCommentForRecommendationAsync(createCommentRequest);
        await _commentApiClient.CreateCommentForRecommendationAsync(modelCreateCommentRequest);

        return new Comment
        {
            RecommendationId = modelCreateCommentRequest.RecommendationId,
            Text = modelCreateCommentRequest.Text,
            IsFromUser = modelCreateCommentRequest.IsFromUser,
            CreatedAt = DateTime.Now,
        };
```

```
    }
}
```

Код класу BaseApiClient

```
public abstract class BaseApiClient
{
    private readonly HttpClient _httpClient;

    protected BaseApiClient(string clientUrl, ITokenService tokenService)
    {
        _httpClient = new HttpClient();
        _httpClient.BaseAddress = new Uri(clientUrl);
        _httpClient.DefaultRequestHeaders.Add("access_token", tokenService.GetToken());
    }

    protected async Task<T> RunGetRequestAsync<T>(string relativeUrl)
    {
        return await _httpClient.GetFromJsonAsync<T>(relativeUrl);
    }

    protected async Task<string> RunGetRequestAsync(string relativeUrl)
    {
        var response = await _httpClient.GetAsync(relativeUrl);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }

    protected async Task RunDeleteRequestAsync(string relativeUrl)
    {
        var response = await _httpClient.DeleteAsync(relativeUrl);
        response.EnsureSuccessStatusCode();
    }

    protected async Task RunPostRequestAsync<T>(string relativeUrl, T body)
    {
        var response = await _httpClient.PostAsJsonAsync(relativeUrl, body);
        response.EnsureSuccessStatusCode();
    }

    protected async Task<TR> RunPostRequestAsync<T, TR>(string relativeUrl, T body)
    {
        var response = await _httpClient.PostAsJsonAsync(relativeUrl, body);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadFromJsonAsync<TR>();
    }

    protected async Task RunPutRequestAsync<T>(string relativeUrl, T body)
    {
        var response = await _httpClient.PutAsJsonAsync(relativeUrl, body);
        response.EnsureSuccessStatusCode();
    }

    protected async Task RunPatchRequestAsync<T>(string relativeUrl, T body)
    {
        var response = await _httpClient.PatchAsJsonAsync(relativeUrl, body);
        response.EnsureSuccessStatusCode();}
}
```

Код класу Prompts

```
public static class Prompts
{
    private static readonly string ReviewPromptTemplate = """
        I would like you to succinctly analyze the git diff of the pull request. Please strictly follow these rules:
        1. You must determine whether a piece of code needs to be changed. If a code section does not contain identifiable issues, vulnerabilities, optimizations, or enhancements, you should skip analysis for that section.
        2. If you find any places in code, which should be reworked, then you must determine a category of your analysis.
        There are categories, sorted by the priority:
        A) Issues (contains bugs and potential issues)
        B) Vulnerabilities (contains any security vulnerability)
        C) Optimization (contains code optimization opportunities)
        D) Enhancements (contains suggested code enhancements)
        F) Best Practice (contains any general coding standard adherence issues that may not fit the prioritized categories)
        3. Descriptions should be clear, concise, and actionable, focusing on why a change is needed and suggesting specific improvements when relevant.
        4. You must analyze the code with adherence to best practices.
        5. You must structure your answer in provided JSON format below. Output only valid JSON in a compact, single-line format in the format provided below, without any extra commentary or text:
        {"issues":[{"description":"% TEXT OF ISSUE DESCRIPTION %","file":"% FULL PATH OF THE FILE CONTAINING THE ISSUE %","beginsAtCodeLine":% START LINE NUMBER FOR ISSUE %,"endsAtCodeLine":% END LINE NUMBER FOR ISSUE %,}],"vulnerabilities":[{"description":"% TEXT OF VULNERABILITY DESCRIPTION %","file":"% FULL PATH OF THE FILE CONTAINING THE VULNERABILITY %","beginsAtCodeLine":% START LINE NUMBER OF VULNERABILITY %,"endsAtCodeLine":% END LINE NUMBER OF VULNERABILITY %,}],"optimization":[{"description":"% TEXT OF OPTIMIZATION DESCRIPTION %","file":"% FULL PATH OF THE FILE CONTAINING POTENTIAL OPTIMIZATION %","beginsAtCodeLine":% START LINE NUMBER OF OPTIMIZATION %,"endsAtCodeLine":% END LINE NUMBER OF OPTIMIZATION %,}],"enhancements":[{"description":"% TEXT OF ENHANCEMENT DESCRIPTION %","file":"% FULL PATH OF THE FILE CONTAINING CODE TO BE ENHANCED %","beginsAtCodeLine":% START LINE NUMBER OF ENHANCEMENT %,"endsAtCodeLine":% END LINE NUMBER OF ENHANCEMENT %,}],"bestPractices":[{"description":"% DESCRIPTION OF BEST PRACTICE ISSUE OR SUGGESTION %","file":"% FULL PATH OF FILE WHERE BEST PRACTICE ISSUE OCCURS %","beginsAtCodeLine":% START LINE NUMBER OF BEST PRACTICE ISSUE %,"endsAtCodeLine":% END LINE NUMBER OF BEST PRACTICE ISSUE %}]}

        TEXT in uppercase between % describes what you should input there.
        Each array may contain multiple elements or remain empty.
        Ensure that the JSON output is valid to facilitate structured parsing and that it matches the provided format exactly.

        To calculate 'beginsAtCodeLine' and 'endsAtCodeLine' you must analyze diff chunk headers
        Each chunk is prepended by a header enclosed within @@ symbols in git diff. The content of the header is a summary of changes made to the file.
        Example diff chunk header: @@ -34,6 +34,8 @@
        In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.
        -------------------

        Pull Request Title: %PR_TITLE%
        Pull Request diff:

        %PR_DIFF%
        """;

    public static readonly string CommentPrompt = """
       You reviewed GitHub Pull Request and provided your analysis.
       You have received a comment from the user associated to your recommendation you gave on certain file.
       I would like you to respond to a user's comment on a pull request.
       Please follow these rules to ensure a clear, professional, and constructive response:
       1. Begin by acknowledging the user's comment to show that their input is valued.
       2. If the comment raises an issue or question, provide a concise explanation or clarification.
       3. If the comment suggests a change or improvement:
         A) Agree if the suggestion is valid and explain how it will be addressed.
         B) Politely disagree if the suggestion is not feasible, providing reasons and possible alternatives.
      4. Mention specific lines of code, files, or parts of the pull request related to the comment. Use line numbers or code snippets for clarity.
```

```
    5. If further action is required, clearly state what you will do next.
    6. Suggest alternatives or seek clarification if needed.
    7. Keep the response polite and professional, even if you disagree with the comment.
    """;

  public static readonly string CommentPromptModelResponse =
    "Of course! Provide me your commend and I will response you using my best analytical possibilities.";

  public static readonly string CommentRequestTemplate =
    """
    Your review information for particular file:
    File name: %FILE NAME%
    Recommendation content:
    %RECOMMENDATION%
    User comment:
    %COMMENT%
    """;

  public static readonly string PullRequestTitlePlaceholder = "%PR_TITLE%";
  public static readonly string PullRequestDiffPlaceholder = "%PR_DIFF%";
  public static readonly string FileNamePlaceholder = "%FILE NAME%";
  public static readonly string RecommendationContentPlaceholder = "%RECOMMENDATION%";
  public static readonly string CommentPlaceholder = "%COMMENT%";

  public static string BuildPromptForReview(string pullRequestTitle, string pullRequestDiff)
  {
    return ReviewPromptTemplate
      .Replace(PullRequestTitlePlaceholder, pullRequestTitle)
      .Replace(PullRequestDiffPlaceholder, pullRequestDiff);
  }

  public static string BuildCommentRequest(string comment, string fileName, string recommendationContent)
  {
    return CommentRequestTemplate
      .Replace(FileNamePlaceholder, fileName)
      .Replace(CommentPlaceholder, comment)
      .Replace(RecommendationContentPlaceholder, recommendationContent);
  }
}
```

Код класу GithubWebhookController

```
public class GithubWebhookController : Controller
{
  private readonly IGithubWebhookService _githubWebhookService;

  public GithubWebhookController(IGithubWebhookService githubWebhookService)
  {
    _githubWebhookService = githubWebhookService;
  }

  [HttpGet]
  [Route("repositories/{repositoryId}/github-webhooks")]
  [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
  public async Task<IReadOnlyList<RepositoryHook>> GetAllRepositoryWebhooksAsync(long repositoryId)
  {
    return await _githubWebhookService.GetAllRepositoryWebhooksAsync(repositoryId);
  }

  [HttpPost]
  [Route("github-webhooks")]
  [Authorize(AuthenticationSchemes = "GithubUserAuthenticationScheme")]
```

```
    public async Task CreateWebhookAsync(long repositoryId)
    {
        await _githubWebhookService.CreateWebhookAsync(repositoryId);
    }
}
```

Код класу GithubWebhookService

```
public class GithubWebhookService : IGithubWebhookService
{
    private readonly Octokit.GitHubClient _client;
    private readonly IConfiguration _configuration;

    public GithubWebhookService(GitHubClient client, IConfiguration configuration)
    {
        _client = client;
        _configuration = configuration;
    }

    public async Task<IReadOnlyList<RepositoryHook>> GetAllRepositoryWebhooksAsync(long repositoryId)
    {
        return await _client.Repository.Hooks.GetAll(repositoryId);
    }

    public async Task CreateWebhookAsync(long repositoryId)
    {
        Dictionary<string, string> newWebhookConfig = new()
        {
            { "content_type", "json" },
            { "url", $"{_configuration.GetSection("ApiBaseUrls")["EventHandlerApi"]}/api/github/webhooks" },
            { "insecure_ssl", "0" },
        };
        string[] newWebhookEvents = ["pull_request"];
        var newWebhook = new NewRepositoryHook("web", newWebhookConfig)
        {
            Events = newWebhookEvents
        };
        await _client.Repository.Hooks.Create(repositoryId, newWebhook);
    }
}
```

Код класу PullRequestWebhookEventProcessor

```
public class PullRequestWebhookEventProcessor : WebhookEventProcessor
{
    private readonly IReviewerApiClient _reviewerApiClient;

    public PullRequestWebhookEventProcessor(IReviewerApiClient reviewerApiClient)
    {
        _reviewerApiClient = reviewerApiClient;
    }

    protected override async Task ProcessPullRequestWebhookAsync(WebhookHeaders headers, PullRequestEvent pullRequestEvent,
PullRequestAction action)
    {
        if (pullRequestEvent.Action == PullRequestAction.Opened)
        {
            await _reviewerApiClient.ReviewPulRequestAsync(pullRequestEvent.Repository.Id, pullRequestEvent.PullRequest.Number);
        }
```

ДОДАТОК Б

Реалізація Front-End частини

Код класу згенерованого Angular-сервісу для Event Handler API

```
@Injectable({ providedIn: 'root' })
export class GithubWebhookService extends BaseService {
  constructor(config: ApiConfiguration, http: HttpClient) {
    super(config, http);
  }

  /** Path part for operation `apiRepositoriesRepositoryIdGithubWebhooksGet()` */
  static readonly ApiRepositoriesRepositoryIdGithubWebhooksGetPath = '/api/repositories/{repositoryId}/github-
webhooks';

  /**
   * This method provides access to the full `HttpResponse`, allowing access to response headers.
   * To access only the response body, use `apiRepositoriesRepositoryIdGithubWebhooksGet$Plain()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiRepositoriesRepositoryIdGithubWebhooksGet$Plain$Response(params:
ApiRepositoriesRepositoryIdGithubWebhooksGet$Plain$Params, context?: HttpContext):
Observable<StrictHttpResponse<Array<RepositoryHook>>> {
    return apiRepositoriesRepositoryIdGithubWebhooksGet$Plain(this.http, this.rootUrl, params, context);
  }

  /**
   * This method provides access only to the response body.
   * To access the full response (for headers, for example),
`apiRepositoriesRepositoryIdGithubWebhooksGet$Plain$Response()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiRepositoriesRepositoryIdGithubWebhooksGet$Plain(params:
ApiRepositoriesRepositoryIdGithubWebhooksGet$Plain$Params, context?: HttpContext):
Observable<Array<RepositoryHook>> {
    return this.apiRepositoriesRepositoryIdGithubWebhooksGet$Plain$Response(params, context).pipe(
      map((r: StrictHttpResponse<Array<RepositoryHook>>): Array<RepositoryHook> => r.body)
    );
  }

  /**
   * This method provides access to the full `HttpResponse`, allowing access to response headers.
   * To access only the response body, use `apiRepositoriesRepositoryIdGithubWebhooksGet$Json()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiRepositoriesRepositoryIdGithubWebhooksGet$Json$Response(params:
ApiRepositoriesRepositoryIdGithubWebhooksGet$Json$Params, context?: HttpContext):
Observable<StrictHttpResponse<Array<RepositoryHook>>> {
    return apiRepositoriesRepositoryIdGithubWebhooksGet$Json(this.http, this.rootUrl, params, context);
  }

  /**
   * This method provides access only to the response body.
```

```
   * To access the full response (for headers, for example),
`apiRepositoriesRepositoryIdGithubWebhooksGet$Json$Response()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiRepositoriesRepositoryIdGithubWebhooksGet$Json(params:
ApiRepositoriesRepositoryIdGithubWebhooksGet$Json$Params, context?: HttpContext):
Observable<Array<RepositoryHook>> {
    return this.apiRepositoriesRepositoryIdGithubWebhooksGet$Json$Response(params, context).pipe(
      map((r: StrictHttpResponse<Array<RepositoryHook>>): Array<RepositoryHook> => r.body)
    );
  }

  /** Path part for operation `apiGithubWebhooksPost()` */
  static readonly ApiGithubWebhooksPostPath = '/api/github-webhooks';

  /**
   * This method provides access to the full `HttpResponse`, allowing access to response headers.
   * To access only the response body, use `apiGithubWebhooksPost()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiGithubWebhooksPost$Response(params: ApiGithubWebhooksPost$Params, context?: HttpContext):
Observable<StrictHttpResponse<void>> {
    return apiGithubWebhooksPost(this.http, this.rootUrl, params, context);
  }

  /**
   * This method provides access only to the response body.
   * To access the full response (for headers, for example), `apiGithubWebhooksPost$Response()` instead.
   *
   * This method doesn't expect any request body.
   */
  apiGithubWebhooksPost(params: ApiGithubWebhooksPost$Params, context?: HttpContext): Observable<void> {
    return this.apiGithubWebhooksPost$Response(params, context).pipe(
      map((r: StrictHttpResponse<void>): void => r.body)
    );
  }
}
```

Код класу AuthComponent

```
export class AuthComponent implements OnInit, OnDestroy {
  private paramMapSub$: Subscription;
  isLoading: boolean;

  constructor(
    private readonly userApiService: UserService,
    private readonly localStorageService: LocalStorageService,
    private readonly router: Router,
    private readonly activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    this.paramMapSub$ = this.activatedRoute.queryParamMap.subscribe(param => {
      const code = param.get('code');
      if (code) {
        this.userApiService.apiUsersAuthTokenGet$Plain({ code: code, access_token: '' }).subscribe(token => {
          this.localStorageService.saveItem({ key: TOKEN_STORAGE_KEY, value: token });
```

```
        this.router.navigate([''], { replaceUrl: true })
      });
    }
  });
 }

 ngOnDestroy() {
  this.paramMapSub$.unsubscribe();
 }

 loginWithGitHub() {
  this.isLoading = true;
  this.userApiService.apiUsersAuthUrlGet$Plain({ access_token: '' }).subscribe({
   next: (url) => {
    window.location.href = url;
   },
   complete: () => this.isLoading = false,
   error: () => this.isLoading = false
  });
 }
}
```

Шаблон компоненту авторизації

```
<div class="auth-container">
 <div class="project-info">
  <h1 class="project-title">AI Reviewer 💡 </h1>
  <div class="decription-container">
   <h2 class="project-description">Let AI Model Review your Pull Requests</h2>
   <mat-icon class="project-icon">rocket_launch</mat-icon>
  </div>
 </div>
 <div class="sign-in-container">
  <button [disabled]="isLoading" mat-raised-button color="primary" (click)="loginWithGitHub()">
   <mat-icon class="project-icon">login</mat-icon>
   Sing in via GitHub
  </button>
  @if (isLoading) {
   <mat-spinner [diameter]="30"></mat-spinner> } </div></div>
```

Код класу HeaderComponent

```
export class HeaderComponent implements OnInit {
 user: Observable<User>;

 constructor(
  private readonly apiSerivce: ApiService,
  private readonly authService: AuthService) { }

 ngOnInit() {
  this.user = this.apiSerivce.getAuthenticatedUser();
 }

 onlogOutClick() {
  this.authService.logOut();
```

```
  }
}
```

Шаблон компоненту заголовку

```
<mat-toolbar class="header">
  <span class="project-title">AI Reviewer</span>
  <span class="spacer"></span>
  <div *ngIf="user | async as userData" class="user-info">
    <img class="user-avatar" [src]="userData.avatarUrl">
    <span class="user-name">{{ userData.username }}</span>
    <button (click)="onlogOutClick()" mat-icon-button aria-label="Log Out">
      <mat-icon>logout</mat-icon>
    </button>
  </div>
</mat-toolbar>
```

Код класу NavigationComponent

```
export class NavigationComponent implements OnInit {
  pageTitle: string;

  constructor(
    private readonly router: Router,
    private readonly activatedRouter: ActivatedRoute) { }

  ngOnInit(): void {
    this.activatedRouter.data.subscribe(data => {
      this.pageTitle = data['pageTitle'];
    })
  }

  navigateToPage(pagePath: string) {
    if (pagePath === 'repositories') {
      this.pageTitle = 'Repositories';
      this.router.navigate(['repositories']);
    } else if (pagePath === 'pull-requests') {
      this.pageTitle = 'Pull Requests';
      this.router.navigate(['pull-requests']);
    } else if (pagePath === 'code-analysis') {
      this.pageTitle = 'Code Analysis';
      this.router.navigate(['code-analysis']);
    }
  }
}
```

Шаблон компоненту навігації

```
<app-header></app-header>
  <mat-sidenav-container class="sidenav-container">
    <mat-sidenav mode="side" opened class="sidenav">
```

```
      <mat-nav-list class="sidenav-list">
        <a (click)="navigateToPage('repositories')" mat-list-item>
          <mat-icon>folder</mat-icon>
          <span>Repositories</span>
        </a>
        <a (click)="navigateToPage('pull-requests')" mat-list-item>
          <mat-icon>merge_type</mat-icon>
          <span>Pull Requests</span>
        </a>
        <a (click)="navigateToPage('code-analysis')" mat-list-item>
          <mat-icon>insights</mat-icon>
          <span>Code Analysis</span>
        </a>
      </mat-nav-list>
    </mat-sidenav>
    <mat-sidenav-content class="sidenav-content">
      <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Код класу RepositoriesComponent

```
export class RepositoriesComponent implements OnInit {
  repositories: Repository[];
  repositoriesWithHook: RepositoryModel[];

  constructor(
    private readonly apiService: ApiService,
    private readonly webhookApiService: GithubWebhookService,
    private readonly authService: AuthService) { }

  ngOnInit(): void {
    this.apiService.getUserRepositories()
      .subscribe(repos => {
        this.repositoriesWithHook = repos.sort((a, b) => +!!b.webhook - +!!a.webhook);
      });
  }

  onConfigureToggleChanged(data: MatSlideToggleChange) {
    if (data.checked) {
      const repositoryId = +data.source.id;
      const repositoryWithHook = this.repositoriesWithHook.find(x => x.repository.id === repositoryId);
      this.webhookApiService.apiGithubWebhooksPost({ repositoryId: repositoryId, access_token:
this.authService.getAccessToken() })
        .subscribe(() => {
          repositoryWithHook.configured = true;
        });
    }
  }
}
```

Шаблон сторінки конфігурації репозиторіїв

```
<h1 class="content-header">Repositories</h1>
<div class="repositories-container">
    @for (repositoryWithHook of repositoriesWithHook; track repositoryWithHook.repository.id) {
```

```
        <mat-card class="repository-card" [ngClass]="{ 'repository-configured' : repositoryWithHook.configured }"
appearance="outlined">
            <mat-card-content>
                <div class="card-content">
                    {{ repositoryWithHook.repository.name }}
                    <mat-slide-toggle
                        id="{{repositoryWithHook.repository.id}}"
                        labelPosition="before"
                        [checked]="!!repositoryWithHook.webhook"
                        (change)="onConfigureToggleChanged($event)">
                        {{ repositoryWithHook.configured ? 'Configured' : 'Not Configured' }}
                    </mat-slide-toggle>
                </div>
            </mat-card-content>
        </mat-card>
    }
</div>
```

Код класу PullRequestsComponent

```
export class PullRequestsComponent implements OnInit {
  pullRequests: PullRequest[] = [];

  constructor(private readonly apiService: ApiService) { }

  ngOnInit(): void {
    this.apiService.getUserRepositories()
      .pipe(
        mergeMap(repos => {
          const reposId = repos.filter(r => r.configured).map(r => r.repository.id) as number[];
          return this.apiService.getPullRequestsByRepositories(reposId);
        }),
      ).subscribe(pullRequests => this.pullRequests.push(...pullRequests));
  }

  getRecommendationsForPullRequest(repoId: number, prNumber: number): Observable<Recommendation[]> {
    return this.apiService.getRecommendations(repoId, prNumber);
  }
}
```

Шаблон сторінки зі списком pull request

```
<h1 class="content-header">Pull Requests</h1>
<div class="repositories-container">
    @for (pr of pullRequests; track pr.id) {
        <mat-card class="repository-card" [ngClass]="{ 'repository-configured' : false }" appearance="outlined">
            <mat-card-content class="card-container">
                <div class="card-content">
                    <div class="card-header">
                        <span>Repository: <b>{{ pr.repository?.name }}</b></span>
                        <span>PR Number: <b>{{ pr.number }}</b></span>
                    </div>
                    {{ pr.name }}
                </div>
```

```
        <app-review-status class="review-status"
[recommendations$]="getRecommendationsForPullRequest(pr.repository.id, pr.number)"></app-review-status>
      </mat-card-content>
    </mat-card>
  }

</div>
```

Код класу PullRequestsComponent

```
export class CodeAnalysisComponent implements OnInit {
  accordion = viewChild.required(MatAccordion);
  selectedRepository = signal<Repository | null>(null);
  selectedPullRequest = signal<PullRequest | null>(null);
  currentUser: User;
  repositories: Repository[] = [];
  pullRequests: PullRequest[] = [];
  recommendations: Recommendation[] = [];
  files: PullRequestFile[] = [];
  fileContents: { [key: string]: string } = {};
  commentsForRecommendation: { [key: number]: Comment[] } = {};
  userComments: { [key: number]: string } = {};

  constructor(private readonly apiService: ApiService) {
    effect(() => {
      const repo = this.selectedRepository();
      if (repo) {
        this.onRepositorySelected(repo.id as number);
      }
    });

    effect(() => {
      const repo = this.selectedRepository();
      const pr = this.selectedPullRequest();
      if (repo && pr) {
        this.onPullRequestSelected(repo.id as number, pr.number as number);
      }
    });
  }

  ngOnInit(): void {
    this.apiService.getUserRepositories()
      .subscribe(repos => this.repositories = repos.filter(r => r.configured).map(r => r.repository));

    this.apiService.getAuthenticatedUser()
      .subscribe(user => this.currentUser = user);
  }

  onRepositorySelected(repoId: number) {
    this.apiService.getPullRequestsByRepository(repoId).subscribe(pullRequests => this.pullRequests = pullRequests);
  }

  onPullRequestSelected(repoId: number, pullRequestNumber: number) {
    forkJoin([this.apiService.getRecommendations(repoId, pullRequestNumber), this.apiService.getPullRequestFiles(repoId,
pullRequestNumber)])
      .subscribe(res => {
        this.recommendations = res[0];
        this.files = res[1];
        this.recommendations.forEach(async recommendation => {
          const fileContent = await this.getFileContent(recommendation.fileName, this.selectedRepository().name)
          const splited = fileContent.split('\n');
          const section = splited.slice(recommendation.codeLines[0] - 1, recommendation.codeLines[1]);
          const joined = section.join('\n');
```

```
        this.fileContents[`${recommendation.id}`] = joined;
      })
    });
  }

  onRecommendationOpened(recommendationId: number) {
    this.apiService.getCommentsForRecommendation(recommendationId).subscribe(comments => {
      this.commentsForRecommendation[recommendationId] = comments;
    });
  }

  onRecommendationResolved(recommendation: Recommendation) {
    recommendation.status = 1;
    this.apiService.updateRecommendationStatus(recommendation).subscribe(() => this.accordion().closeAll());
  }

  onCommentInputChanged(event: Event, recommendationId: number) {
    const value = (event.target as HTMLTextAreaElement).value;
    this.userComments[recommendationId] = value;
  }

  onCommentSubmitButtonClick(recommendationId: number) {
    const userComment = this.userComments[recommendationId];
    this.commentsForRecommendation[recommendationId].push({ recommendationId: recommendationId, text: userComment,
isFromUser: true });

    this.apiService.createCommentForRecommendation({
      recommendationId: recommendationId,
      isFromUser: true,
      text: userComment
    }).subscribe(responseComment => {
      this.commentsForRecommendation[recommendationId].push(responseComment);
    });
  }

  getFileContentFromFileContents(recommendationId: number): string {
    return this.fileContents[`${recommendationId}`];
  }

  private async getFileContent(fileName: string, repositoryName: string): Promise<string> {
    const getFileRequest$ = this.apiService.getFileContent({
      repositoryName: repositoryName,
      filePath: fileName,
      headRef: this.selectedPullRequest().headRef
    }).pipe(
      map(encodedContent => Buffer.from(encodedContent, 'base64').toString())
    );

    return await lastValueFrom(getFileRequest$);
  }
}
```

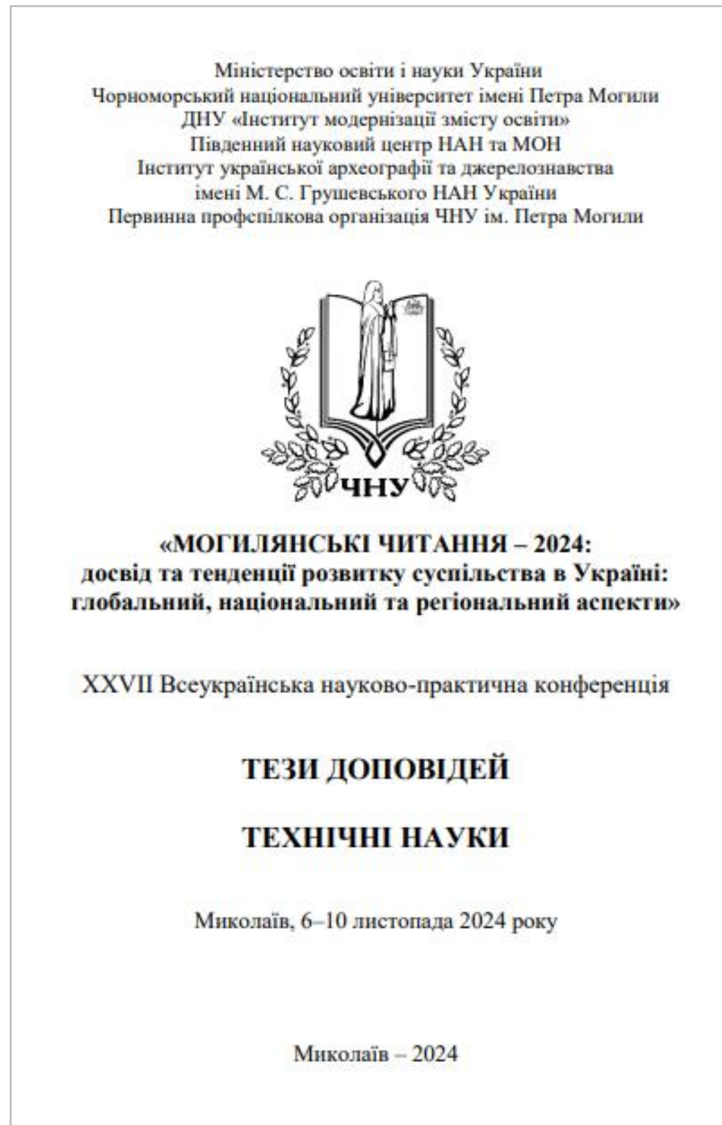
Шаблон сторінки аналізу коду

```
<h1 class="content-header">Code Analysis</h1>
<div class="code-analysis-container">
    <form class="selections">
        <mat-form-field>
            <mat-label>Repository</mat-label>
            <mat-select (selectionChange)="selectedRepository.set($event.value)">
                @for (repository of repositories; track repository.id) {
                  <mat-option [value]="repository">{{repository.name}}</mat-option>
                }
              </mat-select>
```

```
      </mat-form-field>
      <mat-form-field>
        <mat-label>Pull Request</mat-label>
        <mat-select (selectionChange)="selectedPullRequest.set($event.value)">
          @for (pr of pullRequests; track pr.number) {
            <mat-option [value]="pr">{{pr.name}}</mat-option>
          }
        </mat-select>
      </mat-form-field>
    </form>
    <div class="code-analysis" *ngIf="selectedRepository() && selectedPullRequest()">
      <h4><b>Pull Request {{ selectedPullRequest().number }}</b>: <i>{{ selectedPullRequest().name }}</i></h4>
      <mat-accordion class="code-analysis-content" multi>
        @for (recommendation of recommendations; track recommendation.id) {
          <mat-expansion-panel [disabled]="recommendation.status === 1"
(opened)="onRecommendationOpened(recommendation.id)">
            <mat-expansion-panel-header>
              <mat-panel-title>{{ recommendation.fileName }}</mat-panel-title>
              <mat-panel-description>
                <app-review-status class="review-status" [singleRecommendation]="recommendation" ></app-review-status>
              </mat-panel-description>
            </mat-expansion-panel-header>
            <div class="card">
              <div class="card-title">{{ recommendation.content }}</div>
              <pre class="card-code">
                <code
                  [highlightAuto]="getFileContentFromFileContents(recommendation.id)"
                  lineNumbers >
                </code>
              </pre>
              <button mat-raised-button (click)="onRecommendationResolved(recommendation)">Resolve</button>
            </div>
            <h4 class="comments-title">Comments</h4>
            <mat-divider></mat-divider>
            @if (commentsForRecommendation[recommendation.id]?.length === 0) {
              <h3 class="no-comments">There is no comments yet</h3>
            } @else {
              <div class="comments">
                @for (comment of commentsForRecommendation[recommendation.id]; track comment.id) {
                  @if (comment.isFromUser) {
                    <img class="user-avatar" [src]="currentUser.avatarUrl" >
                  } @else {
                    <mat-icon class="system-avatar">smart_toy</mat-icon>
                  }
                  <div class="comment"
                    [ngClass]="{ 'user-comment': comment.isFromUser, 'system-comment': !comment.isFromUser }">
                    {{ comment.text }}
                  </div>
                }
              </div>
            }
            <form (submit)="$event.preventDefault(); onCommentSubmitButtonClick(recommendation.id)" class="comments-form">
              <mat-form-field>
                <mat-label>Leave a comment</mat-label>
                <textarea (input)="onCommentInputChanged($event, recommendation.id)" matInput></textarea>
              </mat-form-field>
              <button [disabled]="!userComments[recommendation.id]" mat-raised-button>Submit</button>
            </form>
          </mat-expansion-panel>
        }
      </mat-accordion>
    </div>
</div>
```

ДОДАТОК В

Апробація кваліфікаційної магістерської роботи



Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили
ДНУ «Інститут модернізації змісту освіти»
Південний науковий центр НАН та МОН
Інститут української археографії та джерелознавства
імені М. С. Грушевського НАН України
Первинна профспілкова організація ЧНУ ім. Петра Могили

**«МОГИЛЯНСЬКІ ЧИТАННЯ – 2024:
досвід та тенденції розвитку суспільства в Україні:
глобальний, національний та регіональний аспекти»**

XXVII Всеукраїнська науково-практична конференція

ТЕЗИ ДОПОВІДЕЙ

ТЕХНІЧНІ НАУКИ

Миколаїв, 6–10 листопада 2024 року

Миколаїв – 2024

Рисунок В.1 – Обкладинка збірника тез конференції Могилянські читання – 2024