Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження впливу застосування інструментів штучного інтелекту на ефективність розробки програмного забезпечення

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-4

Бунтовський В.С.

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник проф. каф. ПІ Смеляков К.С.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ____________ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» ________ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові Бунтовському Владиславу Сергійовичу

1. Тема роботи «Дослідження впливу застосування інструментів штучного інтелекту на ефективність розробки програмного забезпечення»

Затверджена наказом по університету від 20.03.2024 р. № 250Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20.06.2024

3. Вихідні дані до роботи Провести дослідження впливу інструментів генерації коду, заснованих на ШІ, на швидкість та якість розробки програмного забезпечення. Дослідити типи та види таких інструментів, провести порівняння декількох найбільш популярних рішень, та обравши один, провести практичне дослідження на групі розробників, ціллю якого є збір метрик які у подальшому дозволять провести розробку програмного рішення для прогнозування раціональності використання інструментів ШІ у роботі.

4. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз предметної галузі і постановка задачі, огляд та аналіз літературних джерел з дослідження, дослідження теоретичне, дослідження практичне.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	01.05.2024 – 03.05.2024	*Виконано*
2	Розробка постановки задачі	04.05.2024 – 06.05.2024	*Виконано*
3	Розробка ПЗ	07.05.2024 – 10.05.2024	*Виконано*
4	Проведення експерименту	13.05.2024 – 14.05.2024	*Виконано*
5	Оформлення пояснювальної записки	15.05.2024 – 27.05.2024	*Виконано*
6	Підготовка презентації та доповіді	01.06.2024 – 02.06.2024	*Виконано*
7	Попередній захист	05.06.2024	*Виконано*
8	Нормо-контроль, рецензування	14.06.2024	*Виконано*
9	Здача роботи у електронний архів	16.06.2024	*Виконано*
10	Допуск до захисту у зав. кафедри	17.06.2024	*Виконано*

Дата видачі завдання 27.04.2024 р.

Студент ____________ Бунтовський В.С.
(підпис)

Керівник роботи ____________ проф. каф. ПІ, Смеляков К.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Робота містить: 63 сторінки, 8 рисунків, 19 таблиць, 5 додатків, 24 джерел.

НЕЙРОННІ МЕРЕЖІ, РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, AI-ASSISTANT, ATLASSIAN SDK, CODE-WHISPERER, COPILOT, GPT, GITLAB, JIRA PLUGIN, PMD, STAR-CODER,

Дане дослідження присвячене визначенню впливу інструментів генерації програмного коду на продуктивність та якість розробки програмного забезпечення. Всеохоплююча тенденція сучасного світу інтегрувати інструменти штучного інтелекту в усі сфери людської діяльності ставить перед IT-бізнесом питання використання таких інструментів в процесі розробки на рівні методології а не тільки за особистим бажанням кожного окремого розробника. Головне питання – чи сприятливо дані технології впливають на розроблюваний продукт та процеси компанії.

Об’єкт дослідження – використання штучного інтелекту (AI) асистентів, зокрема моделей на кшталт GPT (Generative Pre-trained Transformer[1]) та GitHub Copilot[2], у процесі розробки програмного забезпечення. В цьому контексті, особлива увага приділяється аналізу впливу цих технологій на швидкість та якість процесів розробки.

Мета роботи – систематичне дослідження та оцінка впливу AI асистентів на процеси розробки програмного забезпечення. Це включає в себе вимірювання змін у продуктивності та якості коду, а також аналіз ефективності цих інструментів у різних аспектах програмування, від автоматизації рутинних завдань до підтримки складних процесів розробки.

Результат роботи – розроблене програмне рішення для збору статистики, сформований перелік завдань для реалізації в контексті дослідження, проведений

аналіз конкуруючих рішень в сфері генерації коду, проведено практичний експеримент а також зібрано та проаналізовано його результати з формулюванням висновків.

AI-ASSISTANT, ARTIFICIAL INTELLIGENCE, ATLASTIAN SDK, CODEWHISPERER, COPILOT, GPT, GITLAB, JIRA PLUGIN, NEURAL NETWORKS, PMD, SOFTWARE DEVELOPMENT, STARCODER

This research is dedicated to determining the impact of code generation tools on the productivity and quality of software development. The all-encompassing trend of the modern world to integrate artificial intelligence tools into all spheres of human activity poses a question for the IT industry about using such tools in the development process at a methodological level, not just based on the personal preference of each individual developer. The main question is whether these technologies positively influence the developed product and the company's processes.

The subject of the study is the use of artificial intelligence (AI) assistants, particularly models like GPT (Generative Pre-Trained Transformer) and GitHub Copilot, in the process of software development. In this context, special attention is given to analyzing the impact of these technologies on the speed and quality of development processes.

The goal of the work is a systematic study and evaluation of the impact of AI assistants on software development processes. This includes measuring changes in productivity and code quality, as well as analyzing the effectiveness of these tools in various aspects of programming, from automating routine tasks to supporting complex development processes.

The outcome of the work is the developed software solution for collecting statistics, a compiled list of tasks for implementation in the context of the study, an analysis of competing solutions in the field of code generation, a practical experiment conducted, and its results collected and analyzed with conclusions formulated.

Я, Бунтовський Владислав Сергійович, студент групи ІПЗм-22-4, здобувач освіти на другому (магістерському) рівні кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему Дослідження впливу застосування інструментів штучного інтелекту на ефективність розробки програмного забезпечення, що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ПЕРЕЛІК СКОРОЧЕНЬ

API	– програмний інтерфейс застосунку
AST	– абстрактне синтаксичне дерево
DevSecOps	– розробка, безпека та операції
JSON	– JavaScript Object Notation
LLM	– велика мовна модель
MR	– запит на злиття
XML	– мова розмітки, що розширюється
ШІ	– штучний інтелект

ЗМІСТ

Вступ 10
1 Постановка задачі 11
1.1 Проблематика що розглядається 11
1.2. Формування завдання дослідження 12
1.3 Бачення концепції проведення дослідження 13
1.4 Технічні засоби що використано у дослідженні 14
2 Аналіз предметної галузі та конкурентів 16
2.1 Аналіз сфери інструментів на базі ШІ 16
2.2 Аналіз альтернативних рішень та конкурентів 17
3 Підготовка до проведення дослідження 24
3.1 Концепція експерименту 24
3.2 Система контролю версій Git та платформа GitLab 25
3.3 Система управління agile-проектами Jira 27
3.4 Статичний аналізатор коду PMD 29
3.5 Вимоги до розробника що приймають участь в експерименті 31
3.6 Вимоги до рецензента що приймає участь в експерименті 31
4 Програмна реалізація 32
4.1 Схема зберігання даних 32
4.2 Отримання звіту PMD 33
4.3 Отримання статистики коментарів з MR GitLab 35
4.4 Розрахунок оцінки продуктивності за даними з Jira 37
5 Проведення експерименту та аналіз результатів 39
Висновки 46
Перелік джерел посилання 47
Додаток А. Перелік джерел посилання на працівників кафедри 50
Додаток А. Звіт про перевірку на плагіат 51

Додаток Б. Слайди презентації 52
Додаток В. Тези доповіді для конференції 58
Додаток Г. Результати перевірки роботи на відповідність вимогам 63

ВСТУП

Найбільшим технологічним досягненням останніх 3 років є поширення інструментів заснованих на нейронних мережах, основною задачею яких є виконання рутинних задач у багатьох сферах людської діяльності, починаючи від прикладних наук та закінчуючи творчими сферами на кшталт художнього або музичного мистецтва[3]. Для пересічного користувача інтернету дані інструменти також надають корисні можливості, прискорюючи пошук та обробку інформації або надаючи функції генерації зображень та тексту[4].

Проте не менш очевидним способом використання нейронних мереж є створення програмного забезпечення, адже воно нерідко представляє собою вирішення тривіальних задач але з невеликими відмінностями в залежності від сфери застосування, типу та кількості даних з якими доводиться працювати та цільовою аудиторією під яку дане програмне забезпечення створюється. Саме тому нейронні мережі які генерують рішення на основі інформації яка вже була в інтернеті та використовувалася для навчання моделі, чудово підходять для даної сфери діяльності.

Станом на зараз вже створено декілька інструментів – асистентів розробки програмного забезпечення які базуються на нейронних мережах, частина яких інтегрована у середовища розробки, інші представляють інтерфейс у вигляді чату, проте всі вони виконують приблизно однакові функції – проектування архітектури застосунків, генерація коду, пошук помилок у наявному коді, написання тестів, генерація тестових наборів даних та інше.

Метою даного магістерського дослідження є дослідження та оцінка впливу асистентів заснованих на ШІ на процеси розробки програмного забезпечення. Під оцінкою мається на увазі вимірювання змін у продуктивності та якості коду, а також аналіз ефективності цих інструментів у різних напрямках програмування.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Проблематика що розглядається

Станом на зараз вже існує декілька доволі потужних проектів спрямованих на генерацію коду на основі нейронних мереж. Вони відрізняються моделлю, інтерфейсом взаємодії із розробником, способом розгортання та іншим. Не менш важливим показником є покриття нейронною мережею певних мов програмування, адже універсальність є не менш важливим показником особливо у комерційній діяльності.

Таким чином, актуальною проблемою є вибір певного код-асистента для конкретної задачі або напряму розробки а також питання чи використання асистента впливає на швидкість та якість розробки. І що не менш важливо, чи буде змінюватися ефективність розробки із використанням код-асистента з плином часу, адже людина пристосувавшись до роботи із інструментом може почати втрачати знання та навички що у перспективі може призвести до деградації.

Отже нам потрібно знайти підтвердження або спростування для даних припущень:

- використання ШІ – асистента пришвидшує роботу програміста;
- використання ШІ – асистента знижує продуктивність програміста;
- використання ШІ – асистента знижує якість коду програміста;
- час на перевірку коду від ШІ – асистента збільшує час роботи програміста та зменшує продуктивність [5];
- зростання ефективності різне в залежності від напрямку відділу та науковості проекту;
- вартість ліцензії на AI-асистента не варта сумарного підвищення ефективності в компанії [5];
- вартість ліцензії на AI-асистента збільшує прибуток компанії, деяких відділів або проектів.

Саме отримання відповідей на ці запитання і є головною проблематикою нашого дослідження. Для їх отримання буде використано як математичні розрахунки засновані на метриках конкретних асистентів, так і проводитиметься практичний експеримент на реальних робочих задачах.

1.2 Формування завдання дослідження

Метою магістерського роботи є проведення дослідження, яке включатиме у себе теоретичний блок, практичне дослідження, розробку програмного застосунку, проведення експерименту, оцінку його результатів та формування висновків. На першому етапі роботи буде виконано наступний перелік складових дослідження:

- створення технічного завдання магістерського дослідження;
- теоретичний блок, спрямований на розуміння концепцій та моделей які будуть використовуватися для порівняння та дослідження;
- обрання інструментів та засобів які використовуватимуться у дослідженні;
- детальний розгляд предметної області;
- аналіз аналогів та конкурентних систем;
- вибір інструментів для проведення експериментів;
- розробка програмної реалізації для збору статистичних даних отриманих в ході експерименту;
- проведення безпосередньо експерименту;
- аналіз результатів та формування висновків.

Звичайно цей список не є вичерпним переліком завдань а лише намічає траєкторію магістерського дослідження, якої потрібно притримуватись. В подальших розділах даного документу будуть детально описані кожен етап дослідження та надані усі необхідні роз’яснення.

1.3 Бачення концепції проведення дослідження

Метою впровадження системи допомоги програмістам є покращення їх продуктивності, зниження часу розробки та поліпшення якості коду. Ідея полягає в тому, щоб надати програмістам інструменти, які автоматично надають рекомендації, генерують шаблони коду, аналізують проблеми та надають можливості для автоматизації рутинних завдань. Це може бути досягнуто за допомогою підписки на існуючі системи, такі як GitHub Copilot, Amazon Code Whisperer або розгортання системи Star Coder від Hugging Face на власному сервері.

Для проведення експерименту та перевірки гіпотез треба порівняти роботу програмістів з AI-асистентом та без AI-асистента, працюючих в однаковому середовищі. Треба прийняти до уваги що:

– на вивчення роботи ШІ - асистента потрібен час тож людина не може давати адекватний результат відразу після початку використання;
– знаючі що людина приймає участь в експерименті може статися BIAS в даних які ми збираємо[6];
– не можна порівнювати різні відділи або проекти. В ідеальних умовах треба порівнювати результати ефективності тої самої людини до AI-асистента та через певний час після початку використання;
– перевіряти швидкість роботи можна там де використовується певна система контролю розробки (наприклад Jira);
– перевіряти якість можна там де використовуються практика перегляду коду та де результати такого «перегляду» можна відстежувати.

Пропонується знайти групу добровольців на проекти що вже довго в роботі та де виконуються усі вищезазначені умови. Для порівняння результатів до та після впровадження ШІ - асистента можна використати:

– A/B тести
– Trial тести

Також необхідно підрахувати метрики часу / якості роботи групи, впровадити для групи ШІ – асистент (можуть бути різні асистенти для різних груп для порівняння систем між собою), дати час на освоєння системи та поставити задачу почати використовувати асистента у робочих задачах. Через певний час заміряти метрики по групі знов (час / якість) та визначити наявність приросту швидкості [7] та/або якості та чи покриває він вартість ліцензії.

1.4 Технічні засоби що використано у дослідженні

Як вже було сказано раніше, у дослідженні буде використано декілька найбільш популярних AI-асистентів, таких як GPT, GitHub Copilot, Amazon Code Whisperer, Star Coder від Hugging Face. Також буде проводитися розробка плагіна для Jira який відслідковуватиме швидкість та якість виконання завдань та формуватиме статистичний звіт про доцільність використання AI-асистентів. Розробка плагіна буде проводитися із використанням мови програмування Java а також за допомогою Atlassian SDK.

Пропоную трохи детальніше зупинитися на кожному компоненті.

GPT-4, розроблена OpenAI, є однією з передових моделей у великій родині мовних алгоритмів, відомих як Generative Pre-trained Transformers. Ця технологія застосовує розширені методи машинного навчання [8][9] для різноманітних текстових процесів, таких як створення текстів, відповідей на запитання, узагальнення інформації, переклад між мовами та інші завдання, які потребують обробки мови. Модель "Code-DaVinci" є частиною серії моделей GPT-4 від OpenAI, спеціалізованою на генерації та розумінні програмного коду. Ця модель поєднує можливості генеративного мовного моделювання зі здатністю розуміння та написання програмного коду в широкому діапазоні мов програмування.

GitHub Copilot – це інструмент штучного інтелекту, розроблений GitHub у співпраці з OpenAI, який служить як розширення для редакторів коду, таких як

Visual Studio Code. Цей інструмент використовує машинне навчання, зокрема модель GPT-3, для автоматизації процесу написання програмного коду.

Code Whisperer є інструментом розробленим Amazon Web Services (AWS), який використовує штучний інтелект для допомоги розробникам писати більш ефективний та оптимізований код. Цей інструмент працює як плагін для популярних середовищ розробки (IDE), таких як Visual Studio Code, що дозволяє йому інтегруватися безпосередньо у процес написання коду.

Star Coder є передовою моделлю мови для програмування, яка розроблена як частина великих мовних моделей (LLM [10]) для коду. Ось деякі ключові характеристики та аспекти Star Coder: тренування на різноманітних даних, фокус на програмуванні (Star Coder розроблений спеціально для мов програмування з метою допомоги програмістам писати якісний та ефективний код у скорочені строки), широкий спектр використання мов, покращення доступності та автоматизації.

Atlassian SDK (Software Development Kit) – це набір інструментів, розроблений компанією Atlassian, призначений для розробки, тестування та випуску розширень (плагінів) для продуктів Atlassian, таких як Jira, Confluence, Bitbucket, і Bamboo.

2 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА КОНКУРЕНТІВ

2.1 Аналіз сфери інструментів на базі ШІ

Використання штучного інтелекту (ШІ), зокрема нейронних мереж, у розробці програмного забезпечення, особливо для генерації коду, є перспективним напрямком, який здатний революціонізувати процеси програмування.

Моделі штучного інтелекту, зокрема нейронні мережі, формуються шляхом створення архітектури, яка імітує нейронні з'єднання мозку. Це включає визначення шарів, нейронів у кожному шарі, та способів [11], якими ці нейрони взаємодіють.

Навчання моделей ШІ відбувається на основі великих наборів даних. Дані можуть включати текст, зображення, звук або інші типи інформації. Навчання полягає у коригуванні ваг нейронних зв'язків для мінімізації помилок у відповідях моделі.

ШІ може бути ефективно використаний для автоматизації тестування, аналізу коду, прогнозування помилок, та генерації коду. Менш ефективним ШІ може бути у випадках, які вимагають глибокого творчого мислення або унікального контекстуального розуміння.

Далі наведено список з кількох аспектів на які потрібно першочергово звертати увагу при виборі AI асистента для розробки програмного забезпечення.

- автоматизація: ШІ може автоматизувати певні аспекти написання коду, зменшуючи рутинну роботу для розробників;
- підвищення Продуктивності: ШІ може допомагати у швидшому написанні коду та забезпечувати рекомендації для його оптимізації;
- покращення Якості Коду: ШІ може аналізувати код на наявність помилок або вразливостей;
- навчання та Адаптація: нейронні мережі можуть навчатися на великих наборах даних і адаптуватися до специфічних вимог та стилів кодування;
- виклики з Точністю та Надійністю: генерація коду ШІ може мати виклики з точністю та відповідністю конкретним вимогам проекту.

Використання штучного інтелекту в програмуванні відкриває нові можливості для автоматизації, аналізу та персоналізації [12], значно підвищуючи ефективність розробки програмного забезпечення. Проте, необхідно зважати на виклики, пов'язані з точністю, надійністю та етичними аспектами використання ШІ. Штучний інтелект у розробці програмного забезпечення представляє собою баланс між інноваційним потенціалом та відповідальним впровадженням технологій.

2.2 Аналіз альтернативних рішень та конкурентів

В цьому документі вже неодноразово згадувався перелік альтернативних рішень інструментів генерації коду на основі нейронних мереж. Проте для проведення дослідження маючи невелику команду дослідників, потрібно сфокусуватися на одному AI-асистенті але покрити максимальну кількість напрямів розробки / тестування із використанням максимальної кількості мов програмування. Таким чином, нам потрібно проаналізувати недоліки та переваги кожної альтернативи та обрати найкращій у співвідношенні ширина покриття технологій / якість коду що генерується.

Розглянемо множину альтернатив більш детально.

GitHub Copilot є інтегрованою системою, що використовує машинне навчання для автоматичного генерування коду на основі контексту та існуючих відкритих джерел коду. Система надає функції автоматичного доповнення, генерації функцій та шаблонів коду. З регулярними оновленнями, включаючи Copilot X, що базується на GPT-4, підписка на GitHub Copilot надає доступ до нових функцій для розробників.

Amazon Code Whisperer є інтелектуальною системою, що використовує машинне навчання для аналізу коду, оптимізації та виявлення потенційних проблем. Система базується на найкращих практиках, оцінює продуктивність та поліпшує якість коду. Підписка на Amazon Code Whisperer надає розробникам доступ до інструментів аналізу та оптимізації коду.

Star Coder від Hugging Face – це система генерації коду, яка може бути розгорнута на власному сервері. Це дозволяє контролювати конфіденційність та безпеку даних, використовуючи навчені моделі для генерації коду та автоматизації завдань. З оцінкою ресурсів для розгортання на команду 1-5 користувачів, Star Coder вимагає сервера з RAM 20-60Gb та GPU 24-62Gb. Існує також Visual Studio Code плагін для інтеграції.

GPT – це потужна система question answering, що базується на машинному навчанні. Застосовується як інтелектуальна система допомоги програмісту, не вимагає власного серверу та пройшла fine-tune від команд OpenAI та Microsoft. Однак, потребує самостійної інтеграції з IDE, плати за ліцензію та має обмежені гарантії конфіденційності переданих даних.

Ознайомившись із множиною альтернатив, пропоную сформувати множину критеріїв (див.табл.2.1):

Таблиця 2.1 – Множина критеріїв

Група критерію	Критерій
Якість коду	Читабельність: код повинен бути легко зрозумілим іншими розробниками, включаючи коментарі, належні імена змінних та структурованість. Ефективність: згенерований код повинен бути оптимізованим та ефективним з точки зору виконання завдань. Відповідність стандартам: код повинен дотримуватися встановлених стандартів та кращих практик для відповідної мови програмування.
Підтримка мов програмування	Ширина спектру мов: система повинна підтримувати різноманіття мов програмування для задоволення різних потреб розробників. Актуальність підтримки: підтримка нових версій мов та їх оновлення.
Інтеграція та	Інтеграція в IDE: можливість використовувати систему

Кінець таблиці 2.1

Група критерію	Критерій
сумісність	генерації коду безпосередньо у вибраному середовищі розробки (наприклад, Visual Studio, VS Code, або інші).
Інтеграція та сумісність	Підтримка сторонніх інструментів: здатність взаємодіяти з іншими інструментами та сервісами для повного циклу розробки.
Безпека та приватність	Гарантія конфіденційності: заходи для захисту конфіденційності коду та даних розробки. Безпечність генерації: уникнення генерації коду, який може бути вразливим до атак.
Оновлення та підтримка	Регулярність оновлення: система повинна регулярно оновлюватися з врахуванням нових технологій та покращень. Підтримка та відгуки: активна підтримка користувачів, відповіді на запитання, виправлення помилок та прийняття відгуків

Наступним кроком є створення шкал оцінювання для кожного критерію (див.табл.2.2):

Таблиця 2.2 – Шкали оцінювання

Критерій	Шкала оцінювання
Якість коду	Низька якість: генерований код часто некоректний та не оптимізований. Середня якість: код переважно функціональний, але може характеризуватися як низько ефективний та/або погано читабельний. Висока якість: згенерований код ефективний, оптимізований та відповідає кращим практикам.
Підтримка мов програмування	Обмежена підтримка: система підтримує лише найбільш поширені мови програмування. Середня підтримка: підтримка великої кількості мов проте в роботі все одно можуть виникати ситуації коли потрібної

Кінець таблиці 2.2

	мови чи інструменту немає в списку підтримуваних.
	Розширена підтримка: система підтримує широкий спектр мов програмування та інструментів, можуть бути
	недоступними лише найрідші чи вже не використовувані мови програмування.
Інтеграція та сумісність	Обмежена інтеграція: система не інтегрується або важко взаємодіє з популярними середовищами розробки. Середня інтеграція: є можливість інтеграції, але не в усіх середовищах розробки. Висока інтеграція: легка інтеграція в основні середовища розробки та підтримка для сторонніх інструментів.
Безпека та приватність	Недостатня безпека: відсутність заходів для забезпечення безпеки коду та даних. Середня безпека: присутні певні заходи для захисту, але існують слабкі місця. Висока безпека: запроваджені ефективні заходи для захисту конфіденційності та безпеки даних.
Оновлення та підтримка	Низька підтримка: рідкі оновлення та відсутність відповіді на відгуки користувачів. Середня підтримка: регулярні оновлення, але обмежена підтримка користувачів. Висока підтримка: регулярні оновлення та активна підтримка користувачів.

Усі шкали, порядкові оцінюються за балами від 1 до 3. Створимо векторний опис (див.табл.2.3):

Таблиця 2.3 – Векторний опис

Критерій	Альтернатива			
	Copilot	Code Whisperer	Star Coder	GPT-4
Якість коду	2	1	3	1
Підтримка мов	3	3	2	3
Інтеграція та сумісність	2	3	1	2

Кінець таблиці 2.3

Критерій	Альтернатива			
	Copilot	Code Whisperer	Star Coder	GPT-4
Безпека та приватність	2	2	3	2
Оновлення та підтримка	1	2	3	2

Значення були виставлені згідно моєї оцінки кожного критерію. Нормалізуємо за вагою [13]: значення були виставлені за експертним рішенням (див.табл.2.4).

Таблиця 2.4 – Нормалізовані ваги

Вага	**Критерій**	**Альтернатива**			
		Copilot	Code Whisperer	Star Coder	GPT-4
0,3	Якість коду	2	1	3	1
0,1	Підтримка мов	3	3	2	3
0,2	Інтеграція та сумісність	2	3	1	2
0,1	Безпека та приватність	2	2	3	2
0,3	Оновлення та підтримка	1	2	3	2

Лінійна адитивна згортка з ваговими коефіцієнтами – кожен критерій множиться на свій ваговий коефіцієнт, а потім усі зважені критерії підсумовуються і утворюють зважену цільову функцію, значення якої інтерпретується як «коефіцієнт якості» [13] отриманого рішення (формула 2.1, таблиця 2.5)).

$$F = \sum_{j=1}^{n} \alpha_j \beta_j a_{ij} \tag{2.1}$$

де $a_j = \frac{1}{\sum_{i=1}^{m} a_{ij}}$ – нормуючі множники,

β_j – вагові коефіцієнти, сума яких дорівнює 1,

a_{ij} – значення критеріїв.

Таблиця 2.5 – Коефіцієнт якості

Альтернатива / Критерій	**Якість коду**	**Підтримка мов**	**Інтеграція сумісність**	**Безпека та приватність**	**Оновлення та підтримка**
Copilot	0,6	0,3	0,4	0,2	0,3
Code Whisperer	0,3	0,3	0,6	0,2	0,6
Star Coder	0,9	0,2	0,2	0,3	0,9
GPT-4	0,3	0,3	0,4	0,2	0,6
Вага	0,3	0,1	0,2	0,1	0,3

За правилом Парето, ніяка альтернатива не має переваги над іншою [14]. Виконаємо розрахунок (таблиця 2.6, формула 2.2):

Таблиця 2.6 – Правило Парето

Альтернатива / Критерій	**Якість коду**	**Підтримка мов**	**Інтеграція сумісність**	**Безпека та приватність**	**Оновлення та підтримка**	**Z**
Star Coder	0,6	0,3	0,4	0,2	0,3	0,22
Code Whisperer	0,3	0,3	0,6	0,2	0,6	0,24
Copilot	0,9	0,2	0,2	0,3	0,9	0,30
GPT-4	0,3	0,3	0,4	0,2	0,6	0,22

$$Z = \frac{\sum_{1}^{5} xij}{\sum_{1}^{25} xij} \qquad (2.2)$$

Таким чином, за нашою згортковою моделлю, найкраще рішення: Copilot від Microsoft, з балом: 0.3 що є найбільшим серед визначених.

Виходячи з попереднього аналізу, найкращим рішенням є система Copilot, отже саме її ми будемо використовувати в якості основи для нашого дослідження. Від цього моменту будь-яке вживання терміну «ШІ - інструмент» буде стосуватися саме системи Copilot з метою полегшення тексту.

GitHub Copilot — це інструмент розробки, створений компанією GitHub у співпраці з OpenAI, який був офіційно запущений у червні 2021 року. Copilot використовує машинне навчання, зокрема модель Codex, для надання рекомендацій з написання коду в реальному часі, підтримуючи десятки мов програмування. Цей інструмент позиціонується як "ваш персональний помічник з програмування", що допомагає розробникам швидше писати код та знаходити оптимальні рішення.

3 ПІДГОТОВКА ДО ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

3.1 Концепція експерименту

Для моніторингу часу який був витрачений на виконання певного завдання у комерційних проектах зазвичай використовується система управління agile - проектами Jira від Confluence [15]. Даний інструмент дозволяє оцінити співвідношення оціночного часу до часу який був реально відзвітований розробником в ході роботи. Що стосується якості коду, то цей параметр ми будемо визначати по кількості коментарів які залишають рецензенти merge-запита у системі контролю версій такій як Git – на прикладі продукту GitLab (адже нам потрібно оцінити об'єктивну якість згенерованого коду і з цим можуть найкраще впоратися тільки висококваліфіковані спеціалісти) – чим більше коментарів тим нижче оцінюється якість коду, а також за кількістю знайдених невідповідностей кращим практикам програмування для певної мови чи технології що буде реалізовано за допомогою інструменту PMD [16] що є сучасним та потужним рішенням – чим більше знайдених проблем тим нижче оцінка.

Таким чином, нам потрібно розробити програмну систему яка буде оперувати статистичними даними з систем Jira та GitLab для оцінки швидкості та якості розробки програмного забезпечення. Найбільш простим та зручним рішенням для розв’язання такої задачі є плагін для браузеру, тому приймемо рішення взяти за основу цей підхід. Також у наступних розділах розглянемо більш детально систему контролю версій Git, DevSecOps платформу GitLab, систему управління agile - проектами Jira а також способи отримання та аналізу статистичних даних за допомогою їх публічних API.

В експерименті прийматиме участь 9 розробників з 3 основних напрямів розробки (backend, frontend та мобільні застосунки) а також по 1 технічному експерту з кожного напрямку. В процесі перегляду коду прийматиме участь

технічний експерт напрямку та 2 інших розробника. Без підтвердження кожного з них завдання не може бути зарахованим.

Для досягнення максимальної чистоти експерименту, кількість завдань з використанням інструментів ШІ має становити мінімум 10, натомість для завдань без використання ШІ будуть враховуватися лише 10 останніх щоб виключити похибку з професійного росту спеціаліста. Також у експерименті не зараховуватимуться завдання з невеликими виправленнями (у такому випадку за часту кількість коментарів буде зведена до мінімуму як і час виконання, до того ж такі завдання часто не оцінюються по часу завчасно).

3.2 Система контролю версій Git та платформа GitLab

Як було сказано вище, одним з головних показником у експерименті є якість генерованого коду. Найпростішим способом реалізувати моніторинг цього показника є перевірка коду у кожному коміті (від commit) у системі контролю версій Git. Що стосується перевірки з боку людини, то було прийнято рішення використовувати DevSecOps [17] систему GitLab яка дозволяє переглядати код у зручному web - інтерфейсі, залишати коментарі та вести повноцінні обговорення, а також використовувати різноманітні функції автоматичної розгортки що значно пришвидшує процес експерименту та виключає ризики помилок через людський фактор.

Розглянемо більш детально схему реалізації системи комітів у Git. Кожен коміт у системі Git ідентифікується за допомогою 40(сорока)-значного шістнадцяткового хешу, створеного з використанням алгоритму SHA-1 [18]. Подібний хеш є унікальним і генерується на основі вмісту самого коміту а також його метаданих. Хеш призваний забезпечувати цілісність даних, оскільки навіть мінімальна зміна у вмісті коміту або його метаданих невідворотно призведе до зміни його хешу.

Кожен коміт у Git містить кілька ключових атрибутів. Основні з них наведено у таблиці 3.1:

Таблиця 3.1 – Ключові атрибути Git коміту

Атрибут	Опис
Author	Ім'я та електронна адреса особи, яка створила коміт
Committer	Ім'я та електронна адреса особи, яка здійснила коміт (це може бути різним, наприклад, якщо коміт було змінено в ході перегляду коду)
Date	Час та дата створення коміту
Commit message	Повідомлення коміту, що зазвичай містить пояснення того, що було змінено та чому. Примітка: в ході даного експерименту повідомлення обов’язково має містити будь-який ідентифікатор jira - тікета (ключ/ім’я/посилання) для того щоб система могла пов’язати їх.

Кожен коміт також містить посилання на дерево (tree), яке представляє структуру каталогів та файлів у коміті. Дерево може містити посилання на інші дерева (підкаталоги) і об'єкти blob (зміст файлів). Як і коміти, кожне дерево має унікальний хеш, який змінюється, якщо змінюється структура каталогу чи вміст файлів.

На рисунку 3.1 можна ознайомитися із структурою коміту та атрибутами які до нього входять.

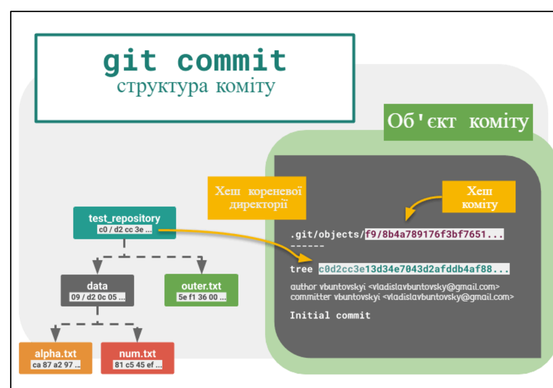

Рисунок 3.1 – Структура коміту

Крім вищезазначеного, кожен коміт містить посилання на один чи кілька "батьківських" комітів. Це вказує на коміти, з яких поточний коміт був похідним. Коміти без батьків (зазвичай початковий коміт у репозиторії) називаються "root commits". Git зберігає коміти як об'єкти у своїй базі даних об'єктів. Кожен об'єкт містить свій тип (наприклад, commit, tree, blob), розмір, вміст та заголовок. Ці аспекти формують основу системи контролю версій Git, забезпечуючи гнучкість, ефективність і безпеку при роботі з кодом.

3.3 Система управління agile-проектами Jira

Наступним не менш важливим аспектом є швидкість розробки, адже цей критерій у багатьох випадках навіть в більшому ступені впливає на бізнес аніж якість коду. Існує велика кількість систем керування проектів, натомість канонічним варіантом є Jira і саме її ми будемо використовувати в контексті нашого дослідження. По-перше вона має відкритий програмний інтерфейс який ми можемо використовувати у власному програмному рішенні враховуючи лише кількість запитів на добу (проте 100 запитів що надаються більш ніж достатньо для приватного використання), по друге вона надає великий перелік функцій відстеження статистичних даних і на останок я, як менеджер дослідження, гарно знайомий з її функціоналом та можливостями що допоможе більш вдало підготувати та провести дослідження.

Для наших цілей є достатнім отримання двох параметрів – очікуваний час на реалізацію функціоналу та реально відзвітований розробником час в процесі виконання поставленої задачі.

Розберемося із атрибутами які ми можемо стягнути з інтерфейсу Jira та визначимо які з них можуть бути використані при підрахунку ефективності використання інструментів ШІ. Переглянути список атрибутів можна у таблиці 3.2:

Таблиця 3.2 – Ключові атрибути Jira Ticket

Атрибут	Опис
ID та ключ	Унікальні ідентифікатори для кожного тікету
Тип	Наприклад, помилка (bug), завдання (task), поліпшення (enhancement)
Статус	Поточний стан завдання, наприклад, "Відкритий" (Open), "У процесі" (In Progress), "Закритий" (Closed)
Пріоритет	Важливість завдання в контексті проекту
Опис	Деталізація завдання
Оціночний час (Original Estimate)	Первісна оцінка часу, необхідного для завершення завдання
Витрачений час (Time Spent)	Фактичний час, відзвітований учасниками проекту
Залишковий час (Remaining Estimate)	Очікуваний час, який ще потрібно витратити

В контексті нашого дослідження найбільш цікавими будуть атрибути «оціночний час» та «витрачений час» адже саме з порівняння цих параметрів буде визначатися оцінка продуктивності виконання завдання. Також для роботи програмного застосунку будуть потрібні ідентифікатор тікету та статус виконання завдання.

Для взаємодії з Jira використовується їх REST API, який дозволяє нам отримувати, створювати, модифікувати та видаляти тікети через HTTP запити. Для виконання автентифікації можливе використання OAuth або API токенів. Для отримання деталей тікету можна використовувати GET запит до URL-адреси типу https://your-domain.atlassian.net/rest/api/3/issue/{issueIdOrKey}.

API повертає дані у форматі JSON, який буде інтерпретуватися в об’єкт та використовуватися у нашій програмі. Щоб зменшити навантаження на сервер та покращити продуктивність, ми скористаємось параметрами, які дозволяють вказувати, які саме поля вам потрібно отримати в відповіді.

3.4 Статичний аналізатор коду PMD

Задля досягнення максимальної об’єктивності в оцінці якості коду, було прийнято рішення ввести ще одну інстанцію контролю – статичний аналізатор якості коду. Даний інструмент не може повідомляти про помилки компіляції натомість дозволяє виявити неефективний код та погані звички розробників (bad practices [19]), які можуть знизити продуктивність і зручність обслуговування програми, якщо вони накопичуються. Далі наводжу неповний перелік основних можливостей інструмента:

- помилки – порожні блоки try/catch/finally/switch;
- мертвий код – невикористані локальні змінні, параметри та приватні методи які не можуть бути викликані;
- порожні оператори if/while;
- надскладні вирази – непотрібні оператори if для циклів, які можуть бути циклами while;
- неоптимальний код – марнотратне використання String/String Buffer;
- класи з високими вимірюваннями циклометричної складності [20];
- дубльований код – скопійований/вставлений код може означати скопійовані/вставлені помилки та погіршити ремонтопридатність.

PMD використовує різні аналізатори для різних мов програмування. Кожен аналізатор забезпечує інтерпретацію вихідного коду та його перетворення в абстрактне синтаксичне дерево (AST). Аналіз ведеться на основі наборів правил, що визначають що є помилкою або практикою, якої слід уникати. Правила конфігуруються для кожної мови і можуть бути налаштовані згідно з потребами проекту. Код аналізується та представляється у вигляді AST, де кожен вузол дерева представляє елементи мови програмування. Правила аналізу виконуються на цьому дереві для ідентифікації потенційних проблем. Після завершення аналізу результати представляються у формі звіту, який може бути виведений в різних форматах, таких

як HTML, XML, текст тощо. Атрибути які можна знайти у звіті аналізатора наводжу у таблиці 3.3:

Таблиця 3.3 – Ключові атрибути статичного аналізатора PMD

Атрибут		**Опис**
File		Шлях до файлу, де було знайдено порушення
Violation	Line	Номер рядка, де відбулось порушення
	Rule	Назва правила, яке було порушено
	Ruleset	Набір правил, до якого належить дане правило
	Priority	Серйозність порушення, де 1 означає найвищу серйозність, а 5 – найнижчу

Найзручнішим форматом для подальшого використання у програмній системі є XML. На рисунку 3.2 приведено приклад такого звіту для помилки.

```
<pmd version="6.34.0" timestamp="2022-08-20T12:34:56.789">
    <file name="/path/to/your/project/Example.java">
        <violation beginline="23"
                   endline="23"
                   begincolumn="10"
                   endcolumn="32"
                   rule="UnusedLocalVariable"
                   ruleset="Best Practices"
                   priority="3">
            Avoid declaring and not using local variables.
        </violation>
    </file>
</pmd>
```

Рисунок 3.2 – Звіт аналізатора PMD

Таким чином, розшифровуючи звіт ми розуміємо що File: …/Example.java – шлях до файлу, де було виявлено порушення, "Avoid declaring and not using local variables" – порушення, що було виявлене, 23 – номер рядка, де відбулось порушення, "UnusedLocalVariable" – назва правила, яке було порушено, "Best

Practices" – набір правил, до якого належить дане правило, Priority: 3 – серйозність порушення, де 1 означає найвищу серйозність, а 5 – найнижчу. Проте найголовнішим параметром для нас буде кількість помилок та їх серйозність, усі інші – допоміжні і також можуть бути використані під час розробки програмної системи.

3.5 Вимоги до розробника що приймають участь в експерименті

Як вже було сказано у попередньому розділі, в експерименті прийматиме участь 9 розробників у 3 напрямах (back-end (Java, Scala), front-end (JavaScript, Angular) та mobile (Java)). Основною вимогою для проведення експерименту є рівнозначність завдань у парах «без використання ШІ» - «з використанням ШІ». Також під час використання інструменту ШІ заборонено виправляти явні помилки та недоліки коду окрім випадків коли він є не функціонуючим. Заборонено використовувати будь-які інші інструменти ШІ окрім того що бере участь у експерименті. Необхідно використовувати прилад для вимірювання часу який був витрачений на виконання завдання. Також не буде зайвим вести нотатки з особистим враженням від використання інструмента, проблемами що виникають а також позитивними моментами які б хотілося відмітити.

3.6 Вимоги до рецензента що приймає участь в експерименті

Головною вимогою при перевірці коду є пошук будь-яких важливих недоліків, включно з недотриманням кращих практик, архітектурних рішень, актуальності бібліотек що використовуються і т.д. Усі проблеми необхідно занотувати у системі GitLab із дотриманням прив'язки до номеру рядка – адже помилки знайдені рецензентом будуть аналізуватися та порівнюватися із тими що знаходяться у звіті аналізатора коду. Для зручності порівняння, рецензенти отримають доступ до класифікації помилок системи PMD а також будуть виставляти оцінки суттєвості помилки виходячи з власного розсуду.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ

Для зручності проведення дослідження, збору та аналізу статистичних даних було прийнято рішення розробити плагін який працюватиме у браузері. Для розробки такого плагіна, який інтегрується з GitLab, PMD, та Jira, ідеальним вибором буде JavaScript, а точніше Typescript, який додає типізацію та поліпшене управління залежностями. Це також включає використання Web API для взаємодії з DOM браузера та зовнішніми API. Для реалізації визначеного функціоналу будуть використовуватися наступні бібліотеки: axios для HTTP запитів, xml2js для інтерпретації XML з PMD, webextension-polyfill для підтримки API розширень браузерів.

4.1 Схема зберігання даних

Першим кроком створення програмної реалізації є проектування схеми зберігання даних. Для наших цілей (дані оновлюються не дуже часто, необхідність збору статистики) відмінно підходить реляційна структура зберігання. Реляційні бази даних мають кілька важливих переваг, які роблять їх популярним вибором у багатьох сферах застосування - використовують строго визначену схему, яка забезпечує консистентність і точність даних, завдяки чому дані легко класифікувати, зберігати, змінювати та витягувати.

Реляційні системи управління базами даних пропонують розширені можливості контролю доступу [21] та ведення транзакцій, що забезпечує високий рівень безпеки і можливість відновлення даних після збоїв або помилок. Завдяки мові SQL, що є стандартом для роботи з реляційними базами даних, користувачі можуть легко складати складні запити, адаптувати дані під змінювані бізнес-вимоги та ефективно масштабувати систему відповідно до зростаючих потреб. Для зручності зберігання даних та інтеграції з нашими сервісами було обрано СУБД – PostgreSQL. Схему даних наведено на рисунку 4.1:

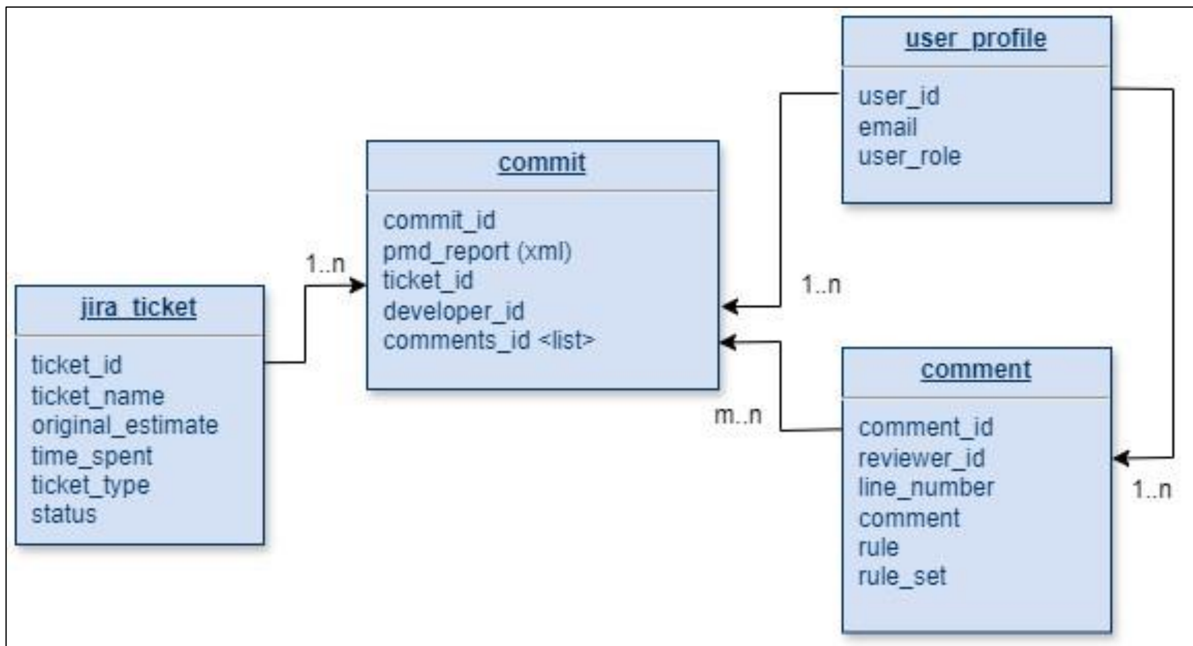

Рисунок 4.1 – ER-діаграма реляційного сховища

Зупинимося більш детально на важливих деталях схеми. Сутність user_profile має таки атрибути як user_id, email, user_role (під роллю користувача мається на увазі розробник чи рецензент). Сутність comment має такі атрибути як comment_id, reviewer_id, line_number (номер рядку коду до якого відноситься коментар), comment, rule (правило яке було порушено), rule_set (група правил). Наступна сутність – jira_ticket має такі атрибути як ticket_id, ticket_name, original_estimate (попередня оцінка), time_spent (використаний час), ticket_type, status. Головною сутністю є commit – має такі поля як commit_id, pmd_report (містить результат звіту статичного аналізатора PMD у форматі XML), ticket_id, developer_id, comments_id (містить масив сутностей коментар).

4.2 Отримання звіту PMD

Для реалізації системи, яка аналізує коміт з використанням PMD, треба виконати наступні кроки: встановлення веб-хука [22] Git для відстеження нових комітів, проаналізувати коміт для визначення, до якого тікета він відноситься (якщо така інформація включена в коміт-повідомлення), запуск PMD для аналізу змінених

файлів у коміті, після чого згенерувати та обробити звіт PMD. На рисунку 4.2 наведено код сервісу для генерації звіту PMD:

```
import express, { json } from 'express';
import { exec } from 'child_process';
import { Parser } from 'xml2js';

const app = express();
app.use(json());

const PORT = 3000;
const PMD_PATH = 'pmd-analizer/pmd-bin/bin/run.sh pmd';
const PMD_RULES_PATH = 'src/ruleset.xml';

app.post('/webhook', async (req, res) => {
    const commitInfo = req.body;
    const repoUrl = commitInfo.repository.git_http_url;
    const commitId = commitInfo.after;

    exec(`git clone ${repoUrl} repo && cd repo && git checkout ${commitId}`, (error, stdout, stderr) => {
        if (error) {
            console.error(`error: ${error.message}`);
            return;
        }

        exec(`${PMD_PATH} -d ./repo -f xml -R ${PMD_RULES_PATH} -r report.xml`, async (error, stdout, stderr) => {
            if (error) {
                console.error(`error: ${error.message}`);
                return;
            }

            const parser = new Parser();
            const xmlData = fs.readFileSync('report.xml');
            const result = await parser.parseStringPromise(xmlData);
            const violations = result.pmd.file.flatMap(file => file.violation);
            console.log(`Total PMD Errors: ${violations.length}`);
        });
    });
    res.status(200).send('Webhook received and processed');
});

app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

Рисунок 4.2 – Сервіс генерації звітів PMD для нового коміта

Для налаштування інтеграції з веб-хуками GitLab, належного встановлення PMD на сервері, а також забезпечення безпеки цього процесу, вам потрібно виконати наступні кроки:

– створення веб-хука: входимо у проект на GitLab та у розділі "Settings" вибираємо "Webhooks". Тут ми можемо створити новий веб-хук, вказавши URL нашого сервера, де будуть оброблятися запити веб-хука;
– безпечне з'єднання: використовуємо HTTPS для вашого веб-хука, щоб забезпечити захищене з'єднання;
– встановлення PMD: завантажуємо та розпаковуємо PMD з офіційного сайту. Важливо щоб Java була встановлена на вашому сервері, оскільки PMD потребує її для роботи;
– конфігурація PMD: налаштуємо шлях до файлу run.sh в сценарії, який ми використовуємо для запуску PMD. Також налаштуємо файл правил, який визначає, як PMD перевірятиме код;
– перевірка секрету веб-хука: налаштуємо веб-хук так, щоб він використовував секретний токен. Сервер, який приймає запити веб-хука, повинен перевіряти цей токен перед обробкою запитів.

Вищезазначених параметрів достатньо для налаштування перехоплення коду з кожного коміту розробника для подальшого його аналізу.

4.3 Отримання статистики коментарів з MR GitLab

Для отримання даних про кількість коментарів до коміта ми виконаємо запит до API GitLab. Для виконання даної операції нам потрібно авторизуватися за допомогою GitLab токену, отримати значення ідентифікатору останнього MR, після чого виконати запит на отримання усіх коментарів для даного MR. Відповідь отримаємо у форматі JSON, інтерпретувавши який ми зможемо отримати статистичні дані про кількість коментарів та класифікацію помилок за типом та значенням. На рисунку 4.3 можна ознайомитися із кодом сервісу для доступу до даних про коментарі до MR через API системи GitLab:

```
import { get, post } from 'axios';

const gitlabToken = getAuthToken(process.env.EMAIL, process.env.PASSWORD);
const projectId = process.env.PROJECT_ID;
const mergeRequestId = getLastMergeRequestId(projectId, gitlabToken);

async function fetchMergeRequestComments() {
  const url = `https://gitlab.com/api/v4/projects/${encodeURIComponent(projectId)}/merge_requests/${mergeRequestId}/notes`;
  try {
    const response = await get(url, {
      headers: { 'PRIVATE-TOKEN': gitlabToken }
    });
    const comments = response.data;
    console.log('Comments:', comments);
  } catch (error) {
    console.error('Error fetching comments:', error.message);
  }
}

async function getAuthToken(email, password) {
    const authUrl = 'https://gitlab.com/api/v4/oauth/token';
    try {
      const response = await post(authUrl, {
        grant_type: 'password',
        username: email,
        password: password
      });
      const { access_token } = response.data;
      console.log('Access Token:', access_token);
      return access_token;
    } catch (error) {
      console.error('Error obtaining token:', error.response ? error.response.data : error.message);
      return null;
    }
  }

  async function getLastMergeRequestId(projectId, gitlabToken) {
    const url = `https://gitlab.com/api/v4/projects/${projectId}/merge_requests?order_by=created_at&sort=desc`;
    try {
      const response = await axios.get(url, {
        headers: { 'PRIVATE-TOKEN': gitlabToken }
      });
      if (response.data.length > 0) {
        const lastMergeRequestId = response.data[0].iid;
        console.log(`Last Merge Request IID: ${lastMergeRequestId}`);
        return lastMergeRequestId;
      } else {
        console.log("No merge requests found for this project.");
        return null;
      }
    } catch (error) {
      console.error(`Error fetching merge requests: ${error.message}`);
      return null;
    }
  }

fetchMergeRequestComments();
```

Рисунок 4.3 – Сервіс доступу до коментарів рецензентів

Як можна побачити на зображенні, наведено лише фрагмент коду з доступом до коментарів. Інтерпретація JSON файлів є тривіальною задачею тому він не був включений до лістингу з метою скорочення обсягу.

4.4 Розрахунок оцінки продуктивності за даними з Jira

Останнім розрахунковим параметром для статистики є продуктивність роботи. Вона вираховується з порівняння оціночного часу та часу який було відзвітовано в процесі виконання завдання. Це включає авторизацію за допомогою API токена, виконання запиту до API для отримання деталей тікета та аналіз відповіді, щоб знайти потрібні поля. Приклад даних отриманих з Jira API наведено на рисунку 4.4:


```
{
    "id": "714",
    "key": "ACH-24",
    "self": "https://jira.nixs.com/rest/api/3/issue/714",
    "fields": {
      "summary": "Registration",
      "issuetype": {
        "id": "714",
        "description": "AI-research - Frontend - Registration",
        "name": "Task",
        "subtask": false
      },
      "creator": {
        "displayName": "vladyslav buntovskyi",
        "emailAddress": "vladislavbuntovsky@gmail.com"
      },
      "status": {
        "name": "In Progress"
      },
      "description": {
        "type": "doc",
        "version": 1,
        "content": [
          {
            "type": "paragraph",
            "content": [
              {
                "text": "Create registration page & service",
                "type": "text"
              }
            ]
          }
        ]
      },
      "timetracking": {
        "originalEstimate": "16h",
        "timeSpent": "9h"
      }
    }
  }
```


Рисунок 4.4 – Jira response

На рисунку 4.5 наведено код сервісу для доступу до даних Jira Ticket:

```
import { get } from 'axios';

const jiraBaseUrl = process.env.JIRA_BASE_URL;

async function fetchIssueTimeTracking(issueId, jiraBaseUrl, email, apiToken) {
    const url = `${jiraBaseUrl}/rest/api/3/issue/${issueId}`;
    const auth = Buffer.from(`${email}:${apiToken}`).toString('base64');
    try {
        const response = await get(url, {
            headers: {
                'Authorization': `Basic ${auth}`,
                'Accept': 'application/json'
            },
            params: {
                fields: 'timetracking'
            }
        });
        const timeTracking = response.data.fields.timetracking;
        console.log(`Original Estimate: ${timeTracking.originalEstimate}`);
        console.log(`Time Spent: ${timeTracking.timeSpent}`);
        return timeTracking;
    } catch (error) {
    console.error('Failed to fetch issue time tracking:', error.response ?
     error.response.data : error.message);
    }
}

async function getAuthToken(email, password) {
    const authUrl = jiraBaseUrl + '/oauth/token';
    try {
      const response = await post(authUrl, {
        grant_type: 'password',
        username: email,
        password: password
      });
      const { access_token } = response.data;
      console.log('Access Token:', access_token);
      return access_token;
    } catch (error) {
      console.error('Error obtaining token:', error.response ? error.response.data : error.message);
      return null;
    }
  }

fetchIssueTimeTracking(issueId, jiraBaseUrl, process.env.EMAIL,
    getAuthToken(process.env.EMAIL, process.env.PASSWORD));
```

Рисунок 4.5 – Сервіс доступу до даних Jira Ticket

Варто відмітити що Jira може мати обмеження на кількість запитів, які можна виконати за певний час і в нашому випадку даний ліміт складає 100 запитів на добу для одного користувача.

5 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

Останнім етапом нашого дослідження є проведення експерименту на реальному комерційному проекті. Було розроблено по 10 завдань для кожного розробника та виконано попередню оцінку (на рисунку 5.1 можна побачити фрагмент дошки – через авторські права реальні імена розробників були перекриті). Кожен розробник отримав інструкції з інсталяції плагіна та налаштування змінних середовища [23] для отримання доступу плагіном до ресурсів проекту. Після закінчення підготовчого етапу почався етап розробки.

Рисунок 5.1 – Фрагмент Agile-дошки проекту

У таблиці 5.1 описаний скоуп робіт для Java розробників. Описані завдання є однаковими для усіх трьох розробників цього напряму розробки. Таким чином, у колонках Without AI та With AI можна побачити ідентифікатор Jira тікету. Саме цей

ідентифікатор повинен бути вказаний в коментарі до кожного коміта для того щоб система могла відслідкувати зв'язок.

Таблиця 5.1 – Перелік завдань для Java розробників

Direction	Task description	Without AI	With AI
Java	Subscription service	ACH-01	ACH-11
	Defining subscription service entities	ACH-02	ACH-12
	Create REST API endpoint to create new subscription	ACH-03	ACH-13
	Create REST API endpoint to get user`s subscription details	ACH-04	ACH-14
	Create REST API endpoint to update subscription	ACH-05	ACH-15
	Create REST API endpoint to cancel subscription	ACH-06	ACH-16
	Create integration tests for subscription service	ACH-07	ACH-17
	Setup MySQL database schema	ACH-08	ACH-18
	Setup Spring security with JWT	ACH-09	ACH-19
	Create CRUD operations for products	ACH-10	ACH-20

У таблиці 5.2 наведено дані про виконані завдання першим розробником Java частини без використання інструментів штучного інтелекту. Також у таблиці присутні такі поля як оціночний час, витрачений час та кількість помилок знайдених рецензентами та статичним аналізатором PMD.

Таблиця 5.2 – Результати виконання завдань без інструментів ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
JavaDev_1	Without AI	ACH-01	2	3	1	3
		ACH-02	4	3	3	5
		ACH-03	8	8	5	8
		ACH-04	3	4	1	2
		ACH-05	4	4	2	2
		ACH-06	2	2	0	1

Кінець таблиці 5.2

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
JavaDev_1	Without AI	ACH-07	8	7	4	3
		ACH-08	2	2	0	2
		ACH-09	4	4	2	1
		ACH-10	8	7	3	3

У таблиці 5.3 зазначені дані про виконання завдання першим розробником Java частини з використанням інструмента штучного інтелекту. Варто зазначити що структура таблиці не відрізняється від попередньої – наявні такі поля як оціночний час, витрачений час та кількість помилок знайдених рецензентами та статичним аналізатором PMD.

Таблиця 5.3 – Результати виконання завдань з інструментом ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
JavaDev_1	With AI	ACH-11	2	1	2	4
		ACH-12	4	2	1	5
		ACH-13	8	5	3	7
		ACH-14	3	3	1	3
		ACH-15	4	2	1	3
		ACH-16	2	2	1	2
		ACH-17	8	5	2	3
		ACH-18	2	2	0	1
		ACH-19	4	2	0	1
		ACH-20	8	5	2	2

На наступній таблиці (таблиця 5.4) описаний скоуп робіт для Frontend частини дослідження. За аналогічним до попередньої частини принципом, завдання є спільними до всіх 3 розробників та для виконання з інструментом ШІ та без такого.

Таблиця 5.4 - Перелік завдань для Frontend розробників

Direction	Task description	Without AI	With AI
Frontend	Login	ACH-21	ACH-31
	Profile	ACH-22	ACH-32
	Add post	ACH-23	ACH-33
	Registration	ACH-24	ACH-34
	Working with followers	ACH-25	ACH-35
	User single page	ACH-26	ACH-36
	Specific group list	ACH-27	ACH-37
	Settings page	ACH-28	ACH-38
	Feedback	ACH-29	ACH-39
	Profile deleting	ACH-30	ACH-40

На таблиці 5.5 можна ознайомитися із деталями реалізації Frontend частини без використання ШІ інструментів.

Таблиця 5.5 – Результати виконання завдань без інструменту ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
FrontDev_1	Without AI	ACH-21	3	3	1	2
		ACH-22	3	4	1	1
		ACH-23	5	4	3	2
		ACH-24	3	3	2	2
		ACH-25	8	7	4	2
		ACH-26	8	8	5	3
		ACH-27	6	7	4	5
		ACH-28	4	4	2	2
		ACH-29	4	5	1	1
		ACH-30	2	2	1	0

На таблиці 5.6 представлені дані про де деталі реалізації Frontend частини з використанням інструменту ШІ.

Таблиця 5.6 – Результати виконання завдань з інструментом ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
FrontDev_1	With AI	ACH-31	3	2	2	4
		ACH-32	3	2	1	2
		ACH-33	5	3	2	2
		ACH-34	3	2	1	3
		ACH-35	8	5	1	1
		ACH-36	8	5	3	1
		ACH-37	6	3	2	3
		ACH-38	4	2	1	1
		ACH-39	4	2	0	1
		ACH-40	2	1	1	2

На наступній таблиці (таблиця 5.7) описаний скоуп робіт для мобільної частини дослідження. За аналогічним до попередньої частини принципом, завдання є спільними до всіх 3 розробників та для виконання з інструментом ШІ та без такого.

Таблиця 5.7 – Перелік завдань для Mobile розробників

Direction	Task description	Without AI	With AI
Mobile	Main tabs setup	ACH-41	ACH-51
	Home screen	ACH-42	ACH-52
	Network data fetching	ACH-43	ACH-53
	UI for movie list	ACH-44	ACH-54
	Search bar	ACH-45	ACH-55
	Movie screen	ACH-46	ACH-56
	UI for movie screen	ACH-47	ACH-57
	Add to favorites	ACH-48	ACH-58
	Video player	ACH-49	ACH-59
	Guest mode	ACH-50	ACH-60

В таблиці 5.8 представлені деталі реалізації мобільної частини без використання інструментів ШІ.

Таблиця 5.8 – Результати виконання завдань без інструменту ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
MobDev_1	Without AI	ACH-41	4	5	1	2
		ACH-42	16	15	6	5
		ACH-43	8	8	4	3
		ACH-44	8	7	2	2
		ACH-45	4	4	2	2
		ACH-46	6	5	1	1
		ACH-47	8	7	4	5
		ACH-48	6	6	2	3
		ACH-49	6	7	3	2
		ACH-50	2	2	0	1

В таблиці 5.9 представлені деталі реалізації Java частини з використанням інструмента ШІ.

Таблиця 5.5 - Результати виконання завдань з інструментом ШІ

Developer	AI	Ticket	Original estimate	Time spent	Comments count	PMD errors count
MobDev_1	With AI	ACH-51	4	3	2	3
		ACH-52	16	10	4	4
		ACH-53	8	5	3	4
		ACH-54	8	4	3	3
		ACH-55	4	3	3	4
		ACH-56	6	5	2	1
		ACH-57	8	3	4	4
		ACH-58	6	4	2	5
		ACH-59	6	4	4	1
		ACH-60	2	1	2	1

Отже, я навів приклад отриманих даних для одного розробника з кожного напряму для скорочення кількості даних. Натомість на наступній таблиці 5.10 буде розміщена загальна статистика з якої можна буде зробити висновки про те як інструменти ШІ впливають на швидкість та якість розробки програмного забезпечення.

Таблиця 5.10 – Результати експерименту для трьох напрямів

Direction		**Productivity**	**Quality (comment per MR)**
Java	Without AI	1.18*estimated	3.24
	With AI	0.79*estimated	4.26
Frontend	Without AI	1.07*estimated	2.17
	With AI	0.71*estimated	3.85
Mobile	Without AI	0.96*estimated	2.9
	With AI	0.68*estimated	3.51

Таким чином, можна зробити висновок що використання інструментів ШІ суттєво прискорює швидкість розробки програмного забезпечення – приріст складає 28%, натомість у кожному напрямку розробки фіксується зниження якості коду (від 11 до 29% в залежності від напряму розробки), основна категорія якого складає недотримання кращих практик написання коду для певної мови програмування. Таким чином, впровадження помічника має сенс за вимоги підвищення контролю якості коду [24]. Більш детальні висновки буде зроблено у наступному розділі цього документу.

ВИСНОВКИ

Тенденції сучасного світу – інтеграція інструментів штучного інтелекту в усі сфери людської діяльності – не є безпідставною. Дані інструменти не просто знижують навантаження на робітників, але й суттєво підвищують продуктивність виконання завдань що дуже позитивно впливає на успішність бізнесу. Натомість треба бути надзвичайно уважним у перевірці відповідей які генерують помічники, адже вони нерідко побудовані на моделях які можуть бути застарілими або не навченими на певні аспекти діяльності що в певних умовах може призвести до зниження якості виконуваної роботи.

В ході проведення магістерського дослідження, я провів експеримент з використання помічників у написанні програмного коду побудованих на нейронних мережах, та прийшов до висновку що на відносно великій вибірці досвідчених розробників з немалим досвідом комерційної розробки, помічники в усіх випадках проявили себе однаково – суттєво пришвидшили розробку але призвели до зниження якості коду. Звісно виграний час можна використати на виправлення недоліків генерації, проте чи даний підхід виявиться швидшим ніж за відсутністю помічників – залишається питанням для відповіді на яке потрібне окреме дослідження.

Також варто відмітити особисту думку розробників які використовували інструменти генерації – усі вони відмітили що використання інструменту потребує попередньої підготовки та вивчення можливостей та особливостей роботи, а також той факт що більша частина часу який вони використали на розробку з використанням ШІ – це підготовка правильного запиту який би поволік за собою отримання правильно генерованого коду. Таким чином, постійне використання інструментів ШІ в роботі призведе до навчання спеціаліста до правильності написання запитів та ще додатково збільшить продуктивність.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Голубенко О .І., Підмогильний О.О. Generative Pre-Trained Transformer 3. IT SYNERGY. URL: https://its.istu.edu.ua/ITS/article/view/16 (дата звернення 20.05.2024)

2. GitHub Copilot. Мій досвід використання. DOU.UA. URL: https://dou.ua/forums/topic/44884/ (дата звернення 20.05.2024)

3. Каук В. І. Генеративний штучний інтелект – креативний помічник дизайнера / В. І. Каук // Поліграфічні, мультимедійні та web-технології. Сучасний стан: монографія. – Харків: ТОВ «Друкарня Мадрид», 2023. – С. 283-294.

4. Турута О. П. Використання моделі dall-e для створення у програмній системі на основі штучного інтелекту для поштового клієнту GMAIL / О. П. Турута, Ж. В. Дейнеко, І. Є. Мічурін // Поліграфічні, мультимедійні та web-технології : тези доп. IX Міжнар. наук.-техн. конф., 14-18 травня 2024 р. – Т. 1. – Харків: ТОВ «Друкарня Мадрид», 2024. – С. 95-96.

5. A Clear Explanation of Transformer Neural Networks - URL: https://www.linkedin.com/pulse/clear-explanation-transformer-neural-networks-ebin-babu-thomas (дата звернення 25.05.2024)

6. 5 Types of statistical BIAS to avoid in your analyses. Harvard Business School. URL: https://online.hbs.edu/blog/post/types-of-statistical-bias (дата звернення 25.05.2024)

7. Zhao H. Global asymptotic stability of Hopfield neural network involving distributed delays / H. Zhao // Neural Networks. - 2004. - Vol. 17, N 1. - P. 48 - 53.

8. Mastering Advanced Machine Learning Techniques: A Comprehensive Guide. Dig8italX. URL: https://dig8italx.com/adv-machine-learning-tech/ (дата звернення 25.05.2024)

9. Alammar J. The illustrated GPT-2 (visualizing transformer language models). Jay Alammar Blog. URL: https://jalammar.github.io/illustrated-gpt2/ (date of access: 21.05.2024)

10. What are Large Language Models (LLM). AWS documentation. URL: https://aws.amazon.com/what-is/large-language-model/?nc1=h_ls (дата звернення 26.05.2024)

11. A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. California Management Review. URL: https://www.researchgate.net/publication/334539401_A_Brief_History_of_Artificial_Intelligence_On_the_Past_Present_and_Future_of_Artificial_Intelligence (дата звернення: 29.05.2024)

12. Haykin S. Neural Networks. A comprehensive Foundation / S. Haykin // Prentice Hall, Inc. N.J. - 2ed. - 1999. -P. 690.

13. Нечітка логіка. Основні поняття теорії прийняття рішень. URL: https://elearning.sumdu.edu.ua/free_content/lectured:075b2e8a0bfe48bcef0ab3106c6d51679abc41f9/20171117122502//117285/index.html (дата звернення: 29.05.2024)

14. Learn the Pareto Principle (The 80/20Rule). Work Section. URL: https://worksection.com/en/blog/pareto-principle-80-20-rule.html (дата звернення: 30.05.2024)

15. Getting started with Jira, Confluence, and the scrum framework. Atlassian documentation. URL: https://www.atlassian.com/agile/scrum/jira-confluence-scrum (дата звернення: 01.06.2024)

16. Introduction to PMD. Baeldung. URL: https://www.baeldung.com/pmd (дата звернення: 01.06.2024)

17. What is DevSecOps? AWS Documentation. URL: https://aws.amazon.com/what-is/devsecops/?nc1=h_ls (дата звернення: 02.06.2024)

18. Secure Hash Algorithm 1 (SHA-1): A Comprehensive Overview | 2023. Medium. URL: https://cyberw1ng.medium.com/secure-hash-algorithm-1-sha-1-a-comprehensive-overview-2023-7be2ca9a06eb (дата звернення: 02.06.2024)

19. 15+ Poor Practices Every Java Developer Must Avoid. LinkedIn. URL: https://www.linkedin.com/pulse/15-poor-practices-every-java-developer-must-avoid-olawale-agboola (дата звернення: 02.06.2024)

20. Cyclomatic Complexity, how to Calculate Cyclomatic Complexity. GeeksForGeeks. URL: https://www.geeksforgeeks.org/cyclomatic-complexity/ (дата звернення: 02.06.2024)

21. Relational database. PostgreSQL. PostgreSQL documentation. URL: https://www.postgresql.org/docs/current/tutorial.html (дата звернення: 02.06.2024)

22. Web hook administration Rake tasks. GitLab documentation. URL: https://docs.gitlab.com/ee/raketasks/web_hooks.html (дата звернення: 02.06.2024)

23. How to set or change the PATH system variable. Oracle documentation. URL: https://www.java.com/en/download/help/path.html (дата звернення: 02.06.2024)

24. Quality Assurance, Quality Control, and Testing – the Basics of Software Quality Management. Altexsoft. URL: https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/ (дата звернення: 02.06.2024)