

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Комп'ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

«AI-інструменти в інженерії програмного забезпечення»

Виконав: студент групи 1П-20 спеціальності

121 “Інженерія програмного забезпечення”

Іван НЕСТЕРЕНКО

Керівник: Станіслав МАРЧЕНКО

Черкаси 2024

# ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра Комп'ютерної інженерії та інформаційних технологій

(повна назва випускової кафедри)

Спеціальність 121 "Інженерія програмного забезпечення"

(шифр і назва спеціальності)

Освітня програма Інженерія програмного забезпечення

(назва освітньої програми)

ЗАТВЕРДЖУЮ

Завідувачу кафедри

Комп'ютерної інженерії та інформаційних технологій

(назва кафедри)

ХОТУНОВУ В.І.

(підпис)

(ПБ)

« \_\_\_\_\_ » \_\_\_\_\_ 20\_\_ р.

## ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Нестеренку Івану Ігоровичу

(прізвище, ім'я, по батькові студента)

1. Тема випускної роботи AI-інструменти в інженерії програмного забезпечення

Керівник роботи Марченко Станіслав Віталійович, спеціаліст I категорії  
затверджені наказом закладу вищої освіти від "13" жовтня 2023 року № 65У.

2. Строк подання студентом випускної роботи 03.06.2024

3. Вихідні дані до випускної роботи Інструменти інструменти автоматичного створення коду DeepCoder, Kite, Codota, системи автоматизації тестування Selenium, Appium, Testim, Applitools; інструменти для автоматизації CI/CD: Jenkins, Travis CI, GitLab CI/CD, CircleCI; інструменти статичного аналізу коду: SonarQube, ESLint, Pylint, Checkmarx, Fortify, Veracode; системи керування та аналізу логів: ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Loggly.

4. Зміст випускної роботи (перелік питань, які потрібно розробити) огляд предметної області (базові поняття в контексті AI інструменти в інженерії програмного забезпечення), AI-інструменти для оптимізації процесів розробки програмного забезпечення (здійснення неперервної інтеграції коду, упаковка програмного коду, розгортання програмного забезпечення, моніторинг додатка), впровадження AI інструментів в інженерію програмного забезпечення.

5. Дата видачі завдання 15.09.2023р.

## КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	20.10.2023	
2	Розділ 1. Огляд предметної області	22.12.2023	
3	Розділ 2. AI інструменти для оптимізації процесів розробки програмного забезпечення	15.03.2024	
4	Розділ 3. Впровадження AI інструментів в інженерію програмного забезпечення	15.05.2024	
5	Висновки	17.05.2024	
6	Оформлення випускної роботи (чистовий варіант)	27.05.2024	
7	Здача випускної роботи на кафедру для рецензування (за 14 днів до захисту)	31.05.2024	
8	Перевірка випускної роботи на наявність ознак плагіату (за 10 днів до захисту)	03.06.2024	
9	Подання випускної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	06.06.2024	

**Студент**

( підпис )

Нестеренко І.І

(прізвище та ініціали)

**Керівник роботи**

—

( підпис )

Марченко С.В.

(прізвище та ініціали)

## АНОТАЦІЯ

У кваліфікаційній роботі представлено аналіз впровадження технологій штучного інтелекту (AI) в процес розробки програмного забезпечення. Розглянуто основні концепції та класифікації AI, а також їх застосування у різних галузях. Особлива увага приділена AI-інструментам, таким як автодоповнення та генерація коду, аналізатори коду, технології генерування тестів, обробка природної мови та розумні пошукові системи.

Для оцінки ефективності впровадження AI було проаналізовано їх вплив на продуктивність розробників та наведено приклади успішної інтеграції AI в різних проєктах. У результаті дослідження встановлено, що використання технологій штучного інтелекту значно підвищує ефективність процесу розробки програмного забезпечення та покращує якість кінцевого продукту. Виявлено, що автоматизація рутинних завдань за допомогою AI сприяє підвищенню продуктивності та зниженню ймовірності помилок.

## **ANNOTATION**

This diploma provides an analysis of the implementation of artificial intelligence (AI) technologies in the software development process. The main concepts and classifications of AI are considered, as well as their application in various industries. Particular attention is paid to AI tools such as auto-complete and code generation, code analyzers, test generation technologies, natural language processing, and smart search engines.

To evaluate the effectiveness of AI implementation, the researchers analyzed their impact on developer productivity and provided examples of successful AI integration in various projects. The study found that the use of artificial intelligence technologies significantly increases the efficiency of the software development process and improves the quality of the final product. It was found that automation of routine tasks with the help of AI helps to increase productivity and reduce the likelihood of errors.

## ЗМІСТ

<b>ВСТУП</b> .....	<b>3</b>
<b>РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ</b> .....	<b>5</b>
1.1. Визначення та основні концепції штучного інтелекту (AI) .....	5
1.2 Інструменти в інженерії програмного забезпечення .....	7
<b>РОЗДІЛ 2 AI-ІНСТРУМЕНТИ ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ</b> .....	<b>16</b>
2.1. Здійснення неперервної інтеграції коду з використанням AI- інструментів.....	18
2.2 Залучення AI-засобів для упаковки програмного коду .....	24
2.3 Розгортання програмного забезпечення та моніторинг додатка з використанням штучного інтелекту .....	37
<b>РОЗДІЛ 3 АНАЛІЗ РЕЗУЛЬТАТІВ ВПРОВАДЖЕННЯ ТЕХНОЛОГІЙ НА ОСНОВІ AI В ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ</b> .....	<b>41</b>
3.1. Кейси інтеграції AI-інструментів в цикл програмної розробки. ....	41
3.2. Вимірювання покращень від задіювання ШІ-інструментів .....	42
<b>ВИСНОВКИ</b> .....	<b>45</b>
<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ</b> .....	<b>48</b>

## ВСТУП

В останні десятиліття спостерігається стрімкий розвиток технологій штучного інтелекту (ШІ), що кардинально змінює різноманітні галузі людської діяльності. Зокрема, значного прогресу досягнуто в галузі програмної розробки, де використання ШІ для автодоповнення та генерації коду стає все більш поширеним. Ці інструменти мають потенціал суттєво підвищити продуктивність розробників, зменшуючи рутинні завдання та покращуючи якість кінцевого продукту.

Автодоповнення коду являє собою технологію, яка дозволяє розробникам отримувати пропозиції щодо завершення коду під час його написання. Це може бути особливо корисно для уникнення синтаксичних помилок, збереження часу на введення повторюваних конструкцій, а також для забезпечення узгодженості стилю кодування. Проте, незважаючи на всі переваги, існують певні обмеження, такі як необхідність точного контексту для релевантних пропозицій та можливість генерації неправильного або не релевантного коду, що потребує додаткової уваги та корекції з боку розробника.

Генерація коду, передбачає повне створення фрагментів коду на основі вхідних даних або коментарів. Цей процес може значно прискорити розробку, особливо на етапі створення прототипів або під час виконання рутинних завдань. Інструменти на зразок GitHub Copilot здатні з урахуванням контексту генерувати цілі блоки коду, що дозволяє розробникам зосередитися на більш складних логічних аспектах проєкту. Водночас, генерація коду може не завжди враховувати специфічні вимоги проєкту чи найкращі практики програмування, що потребує додаткового контролю якості.

Технології обробки природної мови (NLP) також відіграють важливу роль у сучасній розробці програмного забезпечення. Вони дають змогу створювати інтерфейси для спілкування між людиною та машиною, аналізувати та відповідати на запити користувачів, що може суттєво підвищити ефективність технічної підтримки та інших служб. Використання NLP для автоматизованої

обробки текстової інформації відкриває нові можливості для оптимізації бізнес-процесів та покращення взаємодії з клієнтами.

Застосування штучного інтелекту в розробці програмного забезпечення є багатообіцяючим напрямком, що вимагає комплексного підходу та врахування багатьох факторів. Лише за умови ретельного аналізу та впровадження передових практик можна досягти максимальної ефективності та якості кінцевих продуктів.

**Мета дослідження.** Метою проекту є вдосконалення навичок побудови та впровадження конвеєру постачання програмного забезпечення, спрямованого на автоматизацію процесів розробки веб-додатків на основі мови програмування TypeScript та пов'язаних з нею технологій.

**Об'єкт дослідження.** Об'єктом дослідження виступають програмні інструменти на базі штучного інтелекту, які сприяють підтримці життєвого циклу розробки програмного забезпечення.

**Предмет дослідження.** Предметом дослідження є вдосконалення процесів інженерії програмного забезпечення за допомогою програмних засобів, які мають функціональність, що забезпечується інтелектуальними алгоритмами.

**Завдання дослідження.** Практичні завдання в контексті вказаної тематики та мети дослідження такі:

- 1) класифікувати та систематизувати сучасні активності в галузі розробки програмного забезпечення, які можуть отримати переваги від інтелектуальних інструментів;
- 2) дослідити функціональні можливості популярних AI-інструментів на різних етапах життєвого циклу програмного забезпечення;
- 3) проаналізувати та спрогнозувати вплив від впровадження AI-інструментів у виробництво програмного забезпечення.

## РОЗДІЛ 1

### ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

#### 1.1 Визначення та основні концепції штучного інтелекту (АІ)

На сьогоднішній день питання додавання АІ в цикл розробки програмного забезпечення доволі популярний. Використання АІ технологій під час розробки має багато плюсів, один з яких є ключовим, це підвищення швидкості розробки.

Штучний інтелект (ШІ або АІ) – це розробка та впровадження комп'ютерних систем або алгоритмів, які можуть виконувати завдання, що зазвичай вимагають людського інтелекту. Ці завдання можуть включати навчання на основі даних, розпізнавання закономірностей, прийняття рішень, розуміння природної мови та взаємодію з навколишнім середовищем. Системи штучного інтелекту призначені для імітації людських когнітивних здібностей, таких як міркування, вирішення проблем, сприйняття та навчання, для автоматизації завдань, підвищення продуктивності та надання інформації, яка в іншому випадку вимагала б втручання людини.

Зародження ШІ можна віднести до середини 20-го століття, коли з'явилися перші обчислювальні машини. Однак, використання ШІ в інженерії ПЗ почалося лише в 80-х роках з появою експертних систем, які допомагали автоматизувати процес прийняття рішень. Наприклад, системи для автоматизованого тестування і налагодження коду були серед перших застосувань ШІ. Протягом 90-х і 2000-х років розвиток ШІ отримав новий імпульс завдяки появі методів машинного навчання і нейронних мереж. Ці технології дозволили створювати системи, здатні навчатися на основі даних і вдосконалювати свої алгоритми без прямого втручання людини. З початку 2010-х років, з поширенням глибокого навчання і великих даних, ШІ почав впроваджуватися в усі аспекти інженерії ПЗ, від проєктування і розробки до тестування й підтримки.

Основні концепції штучного інтелекту (ШІ) є фундаментом для розуміння того, як працюють сучасні інтелектуальні системи. Ці концепції охоплюють різноманітні методи, алгоритми і підходи, що дозволяють машинам навчатися, розуміти, приймати рішення і виконувати завдання, які зазвичай вимагають

людського інтелекту. Машинне навчання (ML) є одним з найважливіших напрямів ШІ. Воно базується на ідеї, що системи можуть навчатися на основі даних, не будучи явно запрограмованими для конкретних завдань. Алгоритми машинного навчання використовують статистичні методи для аналізу даних і створення моделей, які можуть робити передбачення або приймати рішення. Існує три основні типи машинного навчання:

1) **Навчання з учителем:** у цьому підході модель навчається на мітках даних. Наприклад, у задачі класифікації зображень, система може навчатися розпізнавати об'єкти на зображеннях, використовуючи набір даних з мітками, де кожне зображення вже відмічене відповідним об'єктом;

2) **Навчання без учителя:** цей метод використовує нерозмічені дані для виявлення прихованих патернів або структури в даних. Прикладами можуть бути алгоритми кластеризації, які групують схожі об'єкти разом без попереднього знання про категорії;

3) **Навчання з підкріпленням:** модель навчається на основі взаємодії з середовищем. Вона отримує винагороди або штрафи залежно від дій, які вона виконує, і намагається максимізувати загальну винагороду. Це часто використовується в робототехніці та іграх.

Глибоке навчання (DL) є підвидом машинного навчання, яке використовує нейронні мережі з великою кількістю шарів (глибинні нейронні мережі). Ці мережі мають здатність автоматично виділяти ознаки з даних, що робить їх дуже ефективними для задач, пов'язаних з великими обсягами даних, такими як розпізнавання зображень і обробка природної мови. Глибокі нейронні мережі складаються з великої кількості пов'язаних між собою нейронів, які працюють разом, щоб розпізнавати складні патерни. Основними компонентами нейронних мереж є:

- **вхідний прошарок:** приймає початкові дані;
- **приховані прошарки:** обробляють дані, виконуючи різні перетворення і виділяючи ознаки;
- **вихідний прошарок:** видає кінцевий результат або передбачення.

Обробка природної мови (NLP) дозволяє машинам розуміти, інтерпретувати і генерувати людську мову. Це включає завдання, такі як розпізнавання мови, машинний переклад, аналіз тональності і генерація тексту. NLP використовує різноманітні методи машинного навчання і глибокого навчання для аналізу тексту і мовлення. Одним з ключових компонентів NLP є моделі мовлення, які навчаються на великих корпусах тексту. Ці моделі можуть розуміти контекст і значення слів у реченнях, що дозволяє їм виконувати складні завдання, такі як відповіді на запитання або автоматичне резюмування текстів.

Комп'ютерний зір надає машинам здатність бачити і розпізнавати об'єкти візуального світу, як це роблять люди. Це включає завдання, такі як розпізнавання облич, виявлення об'єктів, сегментація зображень і аналіз відео. Комп'ютерний зір використовує алгоритми глибокого навчання, зокрема згорткові нейронні мережі (CNN), які спеціалізуються на обробці зображень. Еволюційні алгоритми надихаються природним відбором і використовуються для вирішення оптимізаційних задач. Вони включають такі методи, як генетичні алгоритми, диференціальна еволюція і еволюційні стратегії. Ці алгоритми працюють з популяцією можливих рішень, яка еволюціонує протягом багатьох поколінь з використанням операцій, таких як відбір, схрещування і мутація.

Алгоритми посилення спрямовані на те, щоб зробити прогноз більш точним шляхом поєднання кількох слабких моделей в одну сильну модель. Одним з найбільш відомих методів є бустинг, який поступово навчає нові моделі для виправлення помилок попередніх моделей. Це дозволяє значно підвищити точність передбачень.

## **1.2 Інструменти в інженерії програмного забезпечення**

Основні напрямки ШІ інструментів в інженерії програмного забезпечення: Автодоповнення коду є однією з найпоширеніших технологій, яка допомагає розробникам писати код швидше та точніше. Ця технологія працює шляхом передбачення і пропонування можливих варіантів завершення коду, що вводиться програмістом. Автодоповнення може бути як простим, так і досить

складним, використовуючи різні моделі машинного навчання для покращення своїх рекомендацій. Переваги автодоповнення:

- автодоповнення значно скорочує час, необхідний для написання коду, особливо в великих і складних проєктах.
- Автодоповнення допомагає уникати синтаксичних помилок та помилок друку.
- Інструменти автодоповнення полегшують роботу розробника, зменшуючи необхідність запам'ятовування всіх можливих функцій та методів.
- Автодоповнення не завжди розуміє контекст коду, що може призводити до помилкових пропозицій.

Генерація коду за допомогою ШІ – це більш складний та просунутий підхід до автоматизації кодування. Ця технологія дозволяє ШІ самостійно створювати блоки коду на основі заданого контексту та вимог. Вона може використовуватись для генерації шаблонного або повторюваного коду, що значно спрощує життя розробникам і дозволяє зосередитись на більш складних та креативних аспектах програмування. Переваги генерації коду:

- Генерація коду дозволяє автоматизувати рутинні та повторювані задачі, що економить час і зусилля розробників.
- Автоматично згенерований код, за умови правильного налаштування ШІ, має менше помилок, ніж код, написаний вручну.
- Використання генерації коду дозволяє забезпечити єдність стилю та структури коду в межах одного проєкту.

Недоліки генерації коду:

- Генерація коду чудово підходить для шаблонних завдань, вона може бути не такою ефективною при вирішенні складних і унікальних проблем.

Незважаючи на всі переваги, технології автоматизації кодування мають свої виклики та обмеження. Важливо розуміти ці обмеження для максимально ефективного використання цих інструментів та уникнення потенційних проблем. Застосування ШІ у розробці програмного забезпечення порушує низку етичних питань. Наприклад, існує ризик того, що ШІ може створювати код, який порушує

права інтелектуальної власності, або код, який може бути використаний зловмисниками для створення шкідливого програмного забезпечення. Якість коду, згенерованого ШІ, може бути досить високою, але все ще не ідеальною. Розробникам необхідно ретельно перевіряти та тестувати автоматично створений код, щоб уникнути потенційних помилок та багів.

Майбутнє автоматизації кодування за допомогою ШІ виглядає дуже перспективним. Розробка нових моделей та алгоритмів, які ще більш ефективно зможуть виконувати завдання генерації та автодоповнення коду, буде продовжуватись. Ми можемо очікувати на появу нових інструментів, які зможуть ще більше полегшити життя розробників та зробити процес створення програмного забезпечення ще більш ефективним та безпечним.

Статичний аналіз коду (Static Code Analysis) включає в себе вивчення коду без його виконання. Це дозволяє розробникам виявити потенційні проблеми в коді ще на ранніх етапах розробки, що знижує витрати на їх виправлення в подальшому. Основні переваги статичного аналізу коду:

- Статичний аналіз дозволяє виявити помилки та дефекти в коді на ранніх етапах розробки, що значно знижує витрати на їх виправлення.
- Статичні аналізатори допомагають забезпечити дотримання стандартів кодування, що покращує читабельність та підтримуваність коду.
- Аналіз коду може виявити вразливості безпеки, такі як SQL-ін'єкції або буферні переповнення, до того, як код буде запущений в продуктивне середовище.
- Використання статичних аналізаторів допомагає дотримуватися одного стилю кодування у всьому проєкті, що робить код більш уніфікованим.

Динамічний аналіз коду (Dynamic Code Analysis) включає в себе перевірку коду під час його виконання. Це дозволяє розробникам виявити проблеми, які виникають лише під час виконання програми, наприклад, проблеми з продуктивністю, витоки пам'яті та інші помилки виконання. Основні переваги динамічного аналізу коду:

- Динамічний аналіз дозволяє виявити проблеми, які можуть виникнути лише під час виконання програми, наприклад, витоки пам'яті, проблеми з продуктивністю та інші дефекти.
- Аналіз коду під час його виконання дозволяє перевірити роботу програми в умовах, максимально наближених до реальних.
- Виявлення та усунення вузьких місць у виконанні програми дозволяє значно покращити її продуктивність.
- Динамічний аналіз дозволяє виявити вразливості безпеки під час виконання програми, що дозволяє їх оперативно усунути.

Основні переваги поєднання статичного та динамічного аналізу:

- Поєднання статичного та динамічного аналізу дозволяє виявити як синтаксичні та логічні помилки, так і проблеми з продуктивністю та безпекою, що виникають під час виконання коду.
- Комплексний підхід до аналізу коду забезпечує високу надійність програмного забезпечення, знижуючи ризик виникнення критичних помилок у продуктивному середовищі.
- Використання обох методів дозволяє значно покращити якість коду, забезпечуючи дотримання стандартів кодування, високу продуктивність та безпеку.
- Виявлення та усунення проблем на ранніх етапах розробки дозволяє значно знизити витрати на виправлення помилок у майбутньому.

Хоча технології аналізу коду мають безліч переваг, їх використання пов'язане з певними викликами та проблемами, які можуть ускладнити процес розробки програмного забезпечення. Основні виклики при використанні статичного аналізу коду:

- Інструменти статичного аналізу можуть генерувати хибні спрацьовування, тобто виявляти проблеми, яких насправді немає. Це може призвести до витрати часу на аналіз та виправлення неіснуючих проблем.
- Деякі інструменти статичного аналізу можуть підтримувати не всі мови програмування, що обмежує їх використання в багатомовних проєктах.

- Основні виклики при використанні динамічного аналізу коду:
- Виявлені під час динамічного аналізу помилки можуть бути складними для відтворення та виправлення, особливо якщо вони виникають лише в специфічних умовах.
- Проведення динамічного аналізу може вимагати значних ресурсів, включаючи потужні сервери та спеціалізоване програмне забезпечення для моніторингу.

Технології аналізу коду постійно розвиваються, адаптуючись до нових вимог та викликів у сфері розробки програмного забезпечення. Розглянемо деякі тенденції та перспективи розвитку цих технологій. Використання штучного інтелекту та машинного навчання:

- Використання алгоритмів машинного навчання дозволяє значно покращити точність статичного та динамічного аналізу коду, знижуючи кількість помилкових спрацьовувань та підвищуючи ефективність виявлення помилок.
- Штучний інтелект дозволяє аналізувати великі обсяги даних, виявляючи складні закономірності та проблеми, які важко виявити за допомогою традиційних методів.

Автоматичне тестування є процесом використання спеціалізованих інструментів для виконання тестів програмного забезпечення без втручання людини. Цей підхід дозволяє значно прискорити процес тестування, забезпечуючи швидкий зворотний зв'язок розробникам. Основні цілі автоматичного тестування включають:

- Автоматичне виконання рутинних завдань дозволяє розробникам і тестувальникам зосередитися на більш складних і творчих аспектах тестування.
- Автоматичні тести зменшують ризик людських помилок, які можуть виникати під час ручного тестування.

- Автоматичні тести легко інтегруються в процеси CI/CD (Continuous Integration/Continuous Delivery), забезпечуючи швидке виявлення та виправлення помилок.

Автоматичне тестування включає різні види тестів, кожен з яких має свої особливості та призначення:

- Тести на рівні окремих модулів або функцій. Юніт-тести забезпечують перевірку коректності роботи найменших компонентів системи.
- Тести, спрямовані на перевірку взаємодії між окремими модулями або компонентами системи. Інтеграційне тестування допомагає виявити проблеми, які можуть виникнути під час з'єднання різних частин системи.
- Тести, які перевіряють повну інтегровану систему, щоб переконатися, що вона відповідає заданим вимогам.
- Тести, які перевіряють, чи не з'явилися нові помилки після внесення змін до коду.

Генерація тестів – це процес автоматичного створення тестових сценаріїв для перевірки програмного забезпечення. Цей підхід дозволяє значно розширити покриття тестами та зменшити ручну працю, пов'язану з написанням тестових випадків. Основні переваги генерації тестів включають:

- Генерація тестів дозволяє автоматично створювати велику кількість тестових сценаріїв, що забезпечує високе покриття програмного коду.
- Автоматична генерація тестів економить час і ресурси, які зазвичай витрачаються на ручне написання тестів.
- Автоматично згенеровані тести можуть виявляти помилки, які можуть бути пропущені під час ручного тестування.

Майбутнє автоматичного тестування та генерації тестів пов'язане з подальшим розвитком ШІ та машинного навчання. Очікується, що нові технології будуть ще більш інтегровані в процес розробки програмного забезпечення, забезпечуючи ще більшу автоматизацію та ефективність.

Технології управління проєктами відіграють важливу роль у сучасній інженерії програмного забезпечення. Вони забезпечують структурування

інформації та інтеграцію даних з різних джерел, що дозволяє ефективно керувати розробкою, тестуванням та впровадженням програмних рішень. З розвитком штучного інтелекту (ШІ) та машинного навчання (МН), ці технології стали ще більш потужними, надаючи можливість прогнозувати потенційні проблеми та оптимізувати використання ресурсів.

Однією з головних задач управління проектами є збір та структурування інформації з різних джерел. Це включає дані про прогрес розробки, результати тестування, відгуки користувачів та багато іншого. Структурована інформація дозволяє керівникам проектів швидко оцінювати стан проекту та приймати обґрунтовані рішення.

Сучасні системи управління проектами здатні інтегруватися з різними модулями та інструментами, що використовуються в процесі розробки програмного забезпечення. Це включає системи контролю версій, платформи для спільної роботи, інструменти для автоматизованого тестування та багато іншого. Така інтеграція дозволяє отримувати актуальні дані в реальному часі та уникати дублювання зусиль. Штучний інтелект дозволяє автоматизувати процес аналізу великих обсягів даних, що значно спрощує структурування інформації. Наприклад, алгоритми машинного навчання можуть класифікувати помилки, виявлені під час тестування, визначати їх пріоритетність та рекомендувати шляхи вирішення. Це дозволяє зменшити час, витрачений на рутинні завдання, та підвищити ефективність роботи команди.

Якість коду є одним з найважливіших аспектів розробки програмного забезпечення. Високоякісний код забезпечує стабільність, безпеку та продуктивність програмного продукту. Технології управління проектами відіграють важливу роль у забезпеченні якості коду та його тестування.

Автоматизація тестування дозволяє значно підвищити якість програмного забезпечення, знижуючи кількість людських помилок та забезпечуючи високу швидкість перевірки коду. Сучасні інструменти для автоматизованого тестування використовують ШІ для створення ефективних тестових сценаріїв, аналізу результатів тестування та виявлення потенційних проблем.

Інструменти для статичного та динамічного аналізу коду також активно використовують ШІ для виявлення помилок та потенційних вразливостей.

Статичний аналіз коду дозволяє знаходити проблеми ще на етапі написання коду, тоді як динамічний аналіз виявляє помилки під час виконання програми. Використання таких інструментів дозволяє забезпечити високу якість коду та знизити ризик виникнення критичних помилок. Технічний борг є ще одним важливим аспектом якості коду. Він виникає, коли розробники використовують тимчасові рішення або обходи, що в майбутньому можуть стати причиною проблем. Технології управління проектами дозволяють відстежувати технічний борг, оцінювати його вплив на проект та планувати його виправлення.

Використання ШІ дозволяє автоматично виявляти такі проблеми та пропонувати оптимальні шляхи їх вирішення. Прогнозування проблем у коді є одним з ключових напрямків розвитку технологій управління проектами. Використання ШІ та машинного навчання дозволяє передбачати потенційні проблеми ще до їх виникнення, що значно знижує ризик виникнення критичних помилок та підвищує стабільність програмного забезпечення. Моделі машинного навчання використовуються для аналізу історичних даних про помилки та виявлення шаблонів, що можуть вказувати на потенційні проблеми. Наприклад, аналізуючи частоту та типи помилок у коді, можна передбачити, в яких модулях найбільш імовірно виникнуть проблеми. Це дозволяє розробникам зосередити свої зусилля на найбільш критичних частинах проекту.

Крім прогнозування помилок, технології управління проектами дозволяють передбачати продуктивність та масштабованість програмного забезпечення.

Використання ШІ для моделювання навантаження та аналізу продуктивності дозволяє виявити потенційні вузькі місця та оптимізувати роботу системи ще до її впровадження. Це особливо важливо для великих проектів, де виявлення проблем на пізніх стадіях може призвести до значних витрат. Прогнозування проблем у коді також включає оцінку ризиків та планування дій на випадок їх виникнення. Технології управління проектами дозволяють автоматично оцінювати ризики, пов'язані з певними змінами у коді,

та пропонувати план дій для їх мінімізації. Використання таких інструментів дозволяє знизити ризик виникнення непередбачуваних проблем та забезпечити стабільну роботу програмного продукту. Прогнозування ресурсів є ще одним важливим аспектом управління проектами. Воно включає оцінку потреб у людських ресурсах, матеріалах, фінансах та інших ресурсах, необхідних для успішного виконання проекту. Використання ШІ та машинного навчання дозволяє значно підвищити точність таких прогнозів та оптимізувати використання ресурсів. Оцінка потреб у людських ресурсах є одним з найбільш складних аспектів управління проектами. Використання ШІ дозволяє автоматично аналізувати дані про попередні проекти та визначати оптимальну кількість розробників, тестувальників та інших фахівців, необхідних для виконання завдань. Це дозволяє уникнути як нестачі, так і надлишку персоналу, що сприяє ефективному використанню ресурсів.

Прогнозування вартості проекту є важливим елементом планування. Технології управління проектами дозволяють автоматично оцінювати вартість виконання різних завдань на основі даних про попередні проекти та поточні ринкові умови. Використання ШІ для аналізу таких даних дозволяє підвищити точність прогнозів та уникнути перевищення бюджету. Прогнозування потреб у матеріалах та обладнанні також є важливою частиною управління проектами. Використання ШІ для аналізу даних про споживання ресурсів дозволяє оптимізувати їх використання та знизити витрати. Наприклад, алгоритми машинного навчання можуть аналізувати дані про використання обладнання та прогнозувати його знос, що дозволяє своєчасно планувати заміну або ремонт. Розвиток ШІ та машинного навчання відкриває нові можливості для технологій управління проектами. В майбутньому можна очікувати появи ще більш потужних інструментів, що дозволять автоматизувати більшу частину процесів управління та забезпечити ще вищу ефективність.

## РОЗДІЛ 2

### AI-ІНСТРУМЕНТИ ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Питання інтеграції ШІ технологій в інженерію програмного забезпечення стоїть доволі гостро. І їх вже настільки багато, що вони заповнили кожен з сфер інженерії програмного забезпечення рис. 2.1



Рисунок 2.1 – Межі впливу технологій ШІ в інженерії програмного забезпечення

Найпопулярнішим з яких є автоматизація тестування та інтелектуальні асистенти рис. 2.2.



Рисунок 2.2 – Частка використання технологій ШІ в популярних галузях розробки

ШІ в галузі тестування дозволяє оптимізувати процес написання тест кейсів, а саме надає можливість їх генерувати в простих ситуаціях, що дозволяє суттєво економити час розробника, що і робить цю сферу такою популярною для ШІ. Бо в роботі тестувальника більше всього часу іде на процес покриття малих і простих модулів коду, з чим на відмінно справляється ШІ-технології.

Інтелектуальні асистенти є найбільш популярною галуззю застосування ШІ, оскільки надає можливість розробникам ефективніше розробляти продукт, шляхом генерування простих блоків коду під час процесу розробки. Важливо розуміти одне, що весь процес розробки складається здебільшого, маленьких але частих повторюваних шаблонів коду які на даний момент вже навчилися генерувати ШІ правильно. Самі по собі технології генерування як тестів так і самого коду, сильно були зав'язані на масиві даних на якому вони вчилися, і наразі вони вже дійшли до того мінімального рівня, який дозволяє їх ефективно інтегрувати процес розробки. Але ефективна інтеграція технологій тестів напряду зав'язана на ефективності генерації коду від ШІ, оскільки генерація проводить на основі правильного розуміння коду, що і прокачують технології генерації коду. Тому сміливо можна заявити, що відсоток інтелектуальних асистентів, а саме технологій генерації коду будуть приблизно однакові з відсотком використання технологій генерації тестів. Сфера управління проектами, теж отримала велике покращення ефективності завдяки інтеграції ШІ технологій в цей процес. ШІ технології дозволили робити ефективні прогнози можливих помилок і навантажень. Завдяки отриманню статистики про тестування і навантаження на продукт 24/7. Завдяки цьому ШІ аналізує дані і проводить модуляцію різних ситуацій, що і дозволяє робити точні прогнози, щодо продукту. Завдяки ШІ технологіям з'явилася нова сфера їх впливу, це оптимізація коду. Раніше коли технології на основі ШІ навчалися ефективно працювати, цю роль виконували розробники самі, вони назначали людину яка перевіряє якість їх коду на фінальному етапі його додавання в маси. Але з появою ШІ технологій цей процес вдалося значно оптимізувати, завдяки отриманню рекомендацій щодо базової структури коду відразу в середовищі розробки, це

дозволило зменшити кількість коду який потрібно оптимізувати. Також ці технології достатньо розвилися, щоб мати можливість робити прогнози, щодо потенційних проблем коду, що дозволило підняти сферу оптимізації коду на новий рівень.

## 2.1 Здійснення неперервної інтеграції коду з використанням AI-інструментів

Технології які дозволяють покращити неперервну інтеграції коду: CodeClimate, Kite, TabNine, DeepSource, IntelliCode. Проведений дослід показав, що завдяки використанню ІІІ технологій стало можливим збільшити шанс на виявлення вразливостей проєкту на 25%, а також дозволили покращити супроводжуваність коду на 20%, та зменшити кількість помилок після релізу на 15% і пришвидшити процес кодування на 15%.

Приклад роботи технології неперервної інтеграції коду: рис.2.3 – 2.12.

```

1  const todo = (state, action) => {
2    switch (action.type) {
3      case 'ADD_TODO':
4        return { id: action.id, text: action.text, completed: false }
5      case 'TOGGLE_TODO':
6        return { ...state, completed: !state.completed }
7    }
8  }

```



Рисунок. 2.3 – Приклад роботи пропозицій автозаповнення Kite

```

1  const todo = (state, action) => {
2    switch (action.type) {
3      case 'ADD_TODO':
4        return { id: action.id, text: action.text, completed: false }
5      case 'TOGGLE_TODO':
6        return { ...state, completed: !state.completed }
7    }
8  }
9  }

```

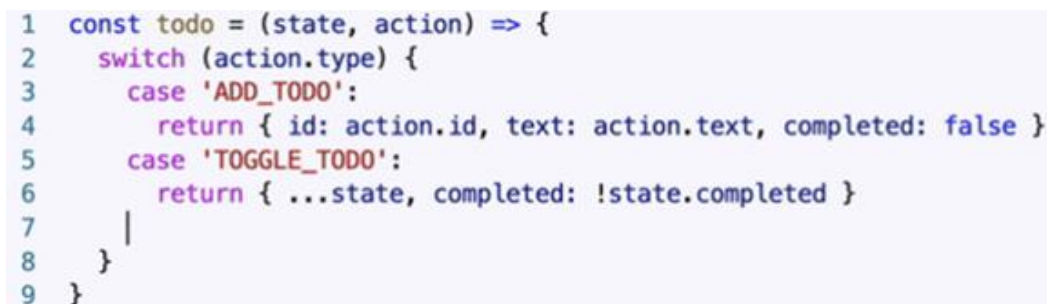


Рисунок 2.4 – Приклад авто доповненого коду від Kite

```

sdwan_sdk.py — tabnine_demo
sdwan_sdk.py 4 | device.yaml
sdwan_sdk.py > ...
1 from vmanage.api.authentication import Authentication
2 from vmanage.api.device import Device
3 import pprint
4 import os
5
6 vmanage_host = os.environ.get('VMANAGE_HOST')
7 vmanage_usernames = os.environ.get('VMANAGE_USERNAME')
8 vmanage_password = os.environ.get('VMANAGE_PASSWORD')
9 pp = pprint.PrettyPrinter(indent=2)
10
11
12 auth = Authentication(host=vmanage_host, username=vmanage_username, password=vmanage_password.login())
13 vmanage_device = Device(auth, vmanage_host)
14
15 device_config_list =
16
  os.environ.get('
  os

```

Рисунок 2.5 – Приклад автодоповнення коду від TabNine

Переваги та недоліки використання технологій автодоповнення для неперервної інтеграції коду. Переваги:

- Автодоповнення пришвидшує кодування на 15%, надаючи пропозиції щодо завершення коду, зменшуючи час, який розробник витрачає на введення повторюваного або шаблонного коду.
- Автодоповнення допомагає запобігти помилкам і синтаксичним помилкам, пропонуючи правильні фрагменти коду на основі контексту, що призводить до більш точного коду.
- Автодоповнення сприяє узгодженості стилю кодування та угод про імена, пропонуючи стандартизовані
- Автодоповнення може слугувати навчальним інструментом для початківців, знайомлячи їх із загальноживаним синтаксисом.
- Автодоповнення дозволяє розробнику ефективніше вивчати API та бібліотеки, надаючи миттєвий доступ до сигнатур методів, списків параметрів та документації.

Проте основний недолік – автодоповнення може іноді надавати неправильні або нерелевантні пропозиції, особливо у складних або неоднозначних контекстах.

Ефективність автозаповнення залежить від якості контекстного аналізу та повноти навчальних даних. У деяких випадках йому може бути складно надавати точні пропозиції для спеціалізованих або специфічних для домену кодових баз.

Технологія статичного аналізу коду дозволяє дивитися як змінювалась якість коду від перших версії до останньої. Завдяки цьому вони можуть бачити різницю між проблемами вже до цього написаним кодом і поточним, що дозволяє краще відслідковувати проблеми пов'язані з ним. Також технологія статичного аналізу коду надає детальну інформацію про кожну з версій коду, що дозволяє розробникам краще розуміти більш глобальні проблеми їх коду, шляхом отримання рапортів з загальною оцінкою поточної версії.

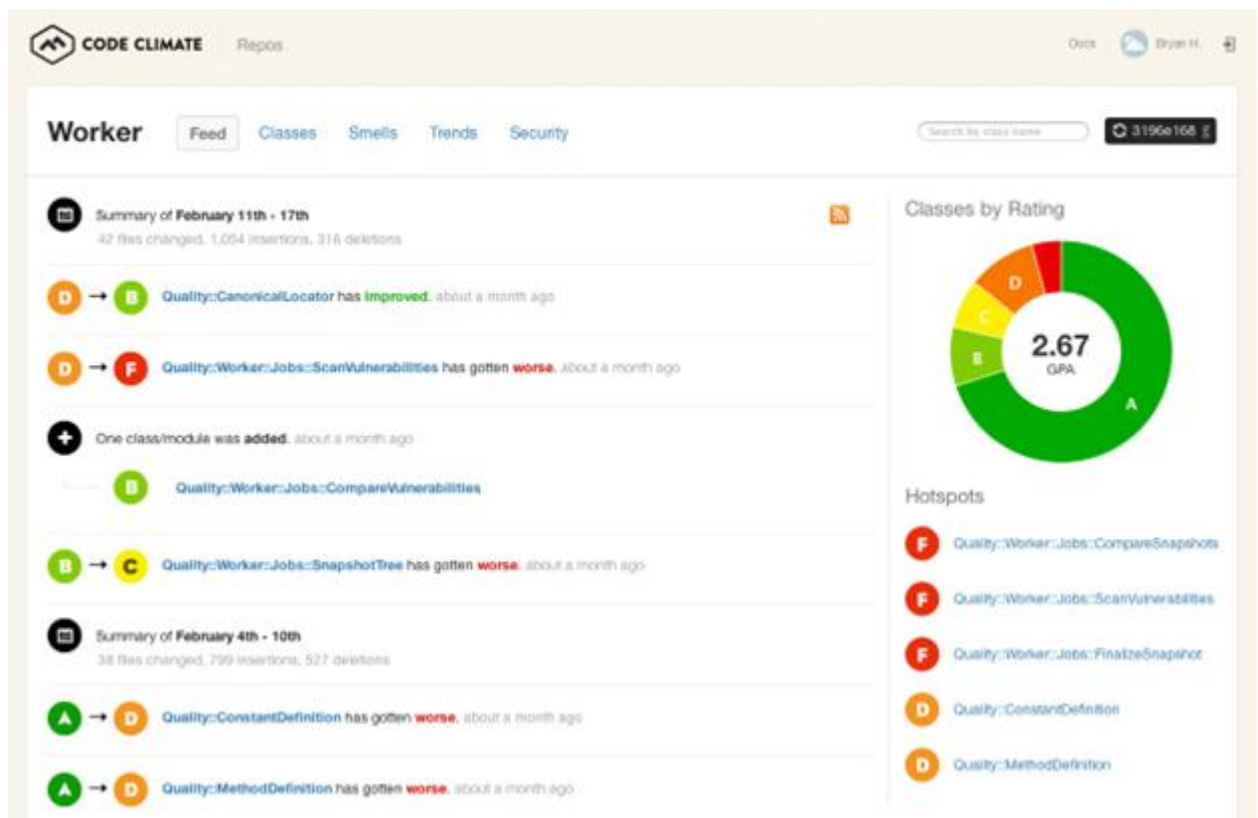


Рисунок 2.6 – Приклад аналізу коду в Code Climate

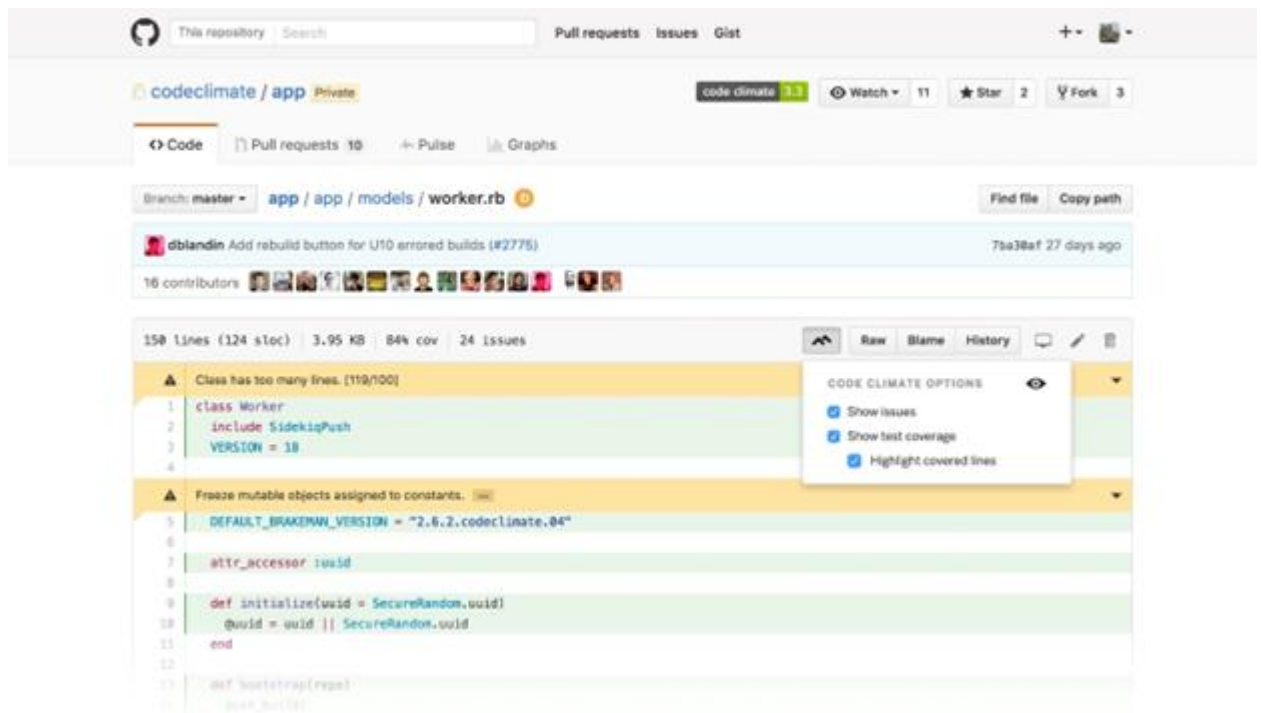


Рисунок 2.7 – Приклад аналізу коду від Code Climate в середовищі GitHub

Переваги технології неперервного статичного аналізу коду:

- Ви отримуєте неперервну оцінку вашого поточного продукту і можете дивитися звіти попередніх версій, що дозволяє краще відслідковувати проблеми проєкту.

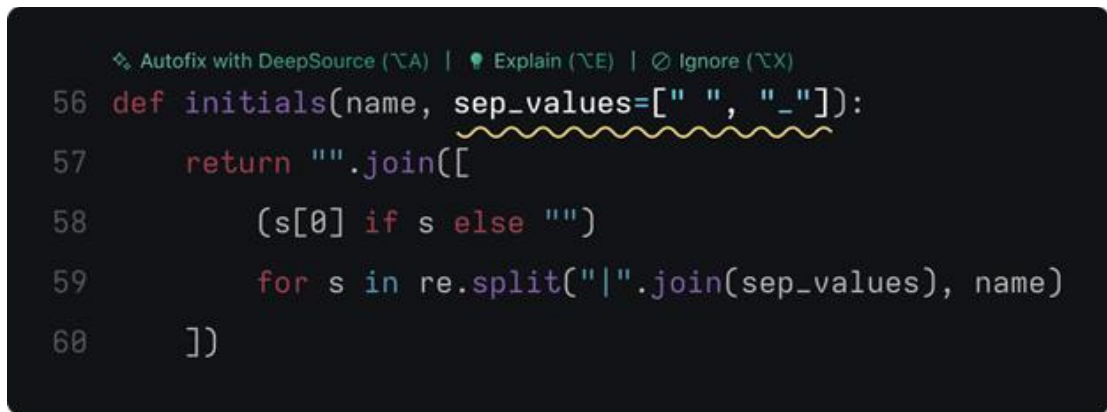
- Також ви отримуєте ґрунтовні звіти по кожній з версій, які містять в собі загальну оцінку версії і як вона вплинула на якість всього проєкту загалом.

- Мінуси:

- Для отримання звітів код потрібно залити в середовище контролю версій, тому отримання звітів під час самого процесу розробки не можливе, також використання технологій аналізу коду не гарантує 100% впевненість в безпеці проєкту, оскільки інформація про проєкт міститься на хмарі самого сервісу де і зберігаються звіти про проєкт.

Технології динамічного аналізу коду дозволяють отримувати звіти про якість коду відразу під час процесу написання коду, шляхом підсвічування проблемних місць в коді, технології динамічного аналізу пропонують і відразу

рішення для цих проблем завдяки інтеграції з автодоповненням, що дозволяє швидко і ефективно писати якісний код.

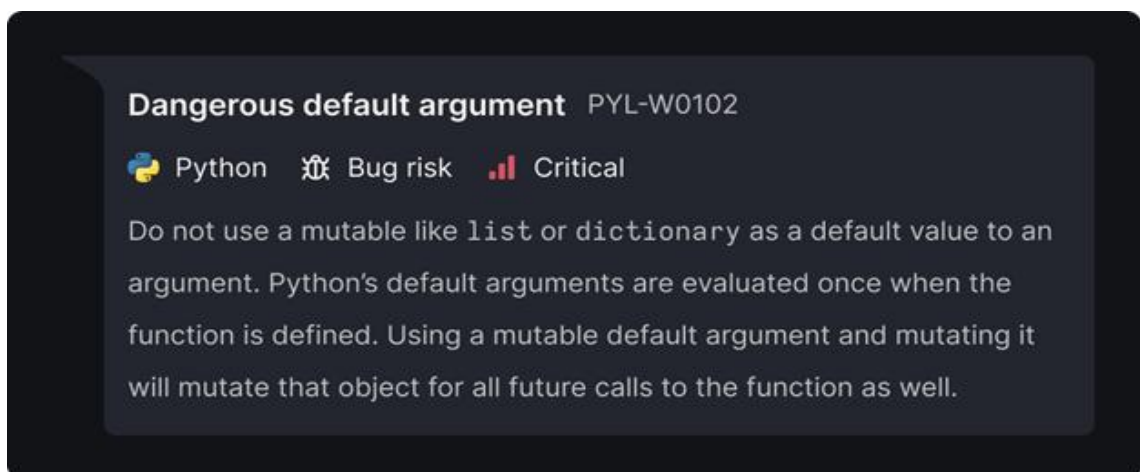


```

Autofix with DeepSource (⌘A) | Explain (⌘E) | Ignore (⌘X)
56 def initials(name, sep_values=[" ", "_"]):
57     return "".join([
58         (s[0] if s else "")
59         for s in re.split("|".join(sep_values), name)
60     ])

```

Рисунок 2.8 – Приклад виявлення проблемного коду який може спричинити помилку. Динамічний аналізатор DeepSource

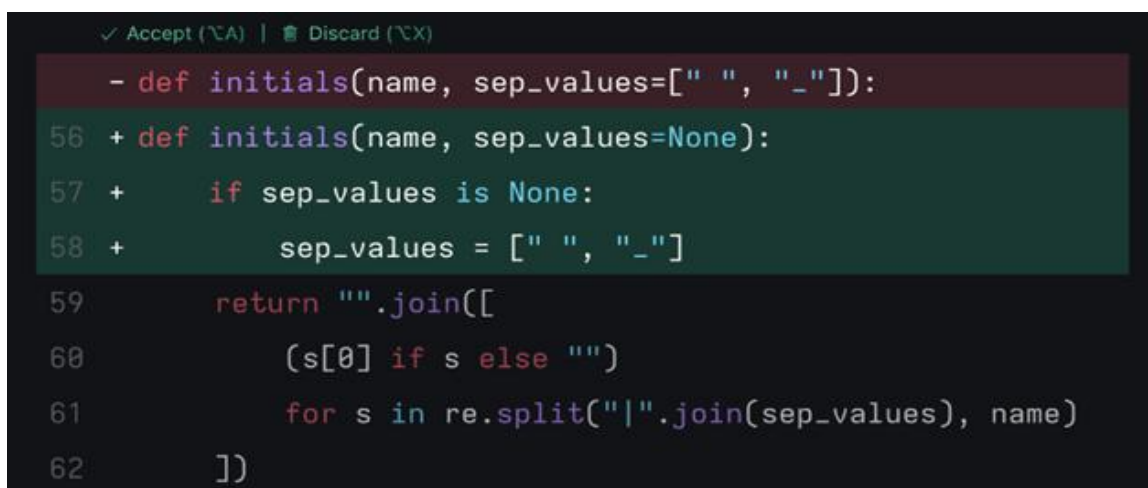


**Dangerous default argument** PYL-W0102

Python Bug risk Critical

Do not use a mutable like list or dictionary as a default value to an argument. Python's default arguments are evaluated once when the function is defined. Using a mutable default argument and mutating it will mutate that object for all future calls to the function as well.

Рисунок 2.9 – Приклад рекомендації від DeepSource під час розробки



```

Accept (⌘A) | Discard (⌘X)
- def initials(name, sep_values=[" ", "_"]):
56 + def initials(name, sep_values=None):
57 +     if sep_values is None:
58 +         sep_values = [" ", "_"]
59     return "".join([
60         (s[0] if s else "")
61         for s in re.split("|".join(sep_values), name)
62     ])

```

Рисунок 2.10 – Приклад рекомендації покращення коду від DeepSource.



Рисунок 2.11 – Приклад рефакторингу коду за допомогою IntelliCode



Рисунок 2.12 – Приклад роботи розумних підказок від IntelliCode

Плюси використання динамічних аналізаторів коду для здійснення непервної інтеграції коду:

– Завдяки використанню динамічного аналізатору коду вдалося прибрати проблему людського фактору.

– Також динамічні аналізатори коду дозволили підняти загальну якість коду, а також покращили його підтримуваність завдяки використанню найкращих практик та шаблонів написання коду, що дозволяє легше його підтримувати.

Головним недоліком цих технологій є проблема охоплення великого контексту, через що його рекомендації можуть не працювати в складних ситуаціях, коли модулі коду мають складні залежності між собою.

Використання технологій неперервної інтеграції коду дозволили збільшити шанс на виявлення вразливостей проєкту на 25%, також дозволило зменшити кількість помилок після релізу на 15%, та пришвидшує процес кодування на 15%. Що доволі сильно покращило загальну якість проєкту і полегшило його підтримку в подальшому іншими розробниками.

## **2.2 Залучення AI-засобів для упаковки програмного коду**

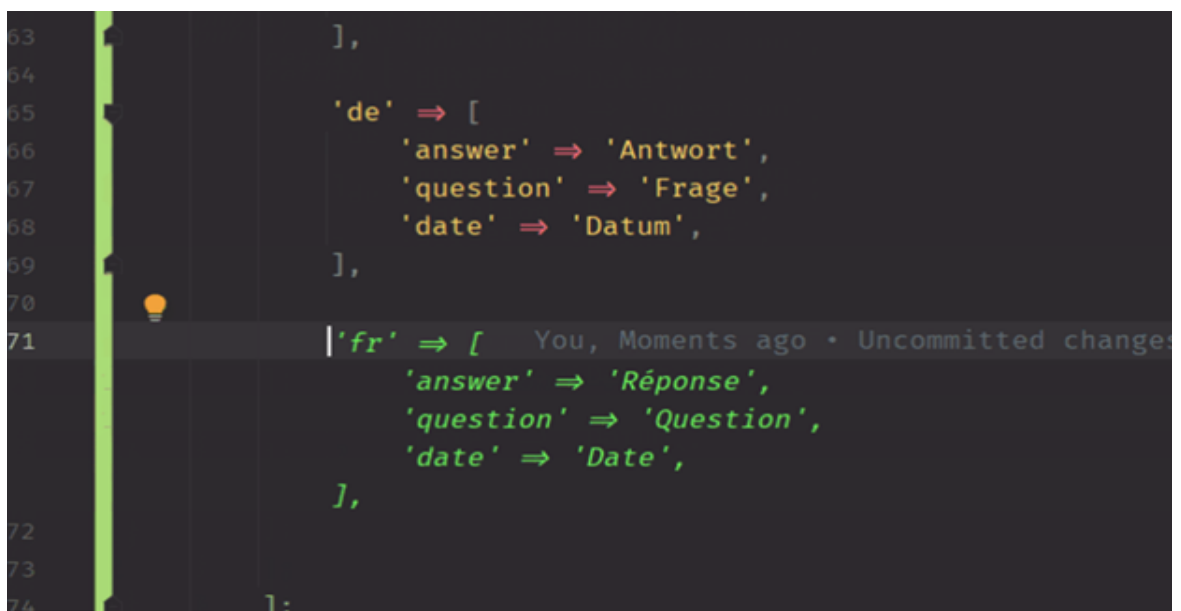
Технології упаковки програмного коду включають в себе процеси тестування, генерації коду та генерації документації проєкту. Завдяки використанню цих технологій вдалося покращити час написання всього проєкту на всіх етапах і полегшило процес підтримки проєкту в подальшому. Завдяки згенерованій документації процес підтримки сильно полегшився, тому що інші розробники яким віддали цей проєкт, можуть легко зрозуміти який модуль за що відповідає і як з ним потрібно працювати. Це дозволило зменшити час потрібний розробнику на аналіз поточного проєкту з метою зрозуміння принципів його роботи. Також це дозволило швидше працювати і іншій стороні, а саме поточним розробникам яким потрібно було писати цю документацію але завдяки технології генерації документації вдалося пришвидшити цей процес з двох сторін.

Технології автоматичного тестування дозволили зменшити час який потрібен для покриття всіх модулів коду, що дозволило зосередитися

тестувальнику на більш складних місця для тестування де ШІ не зміг генерувати тести. Це дозволило пришвидшити час потрібний на покриття всіх модулів тестами на 40–50% завдяки тому, що ШІ генерує прості тест кейси на які і йде більшість часу тестувальника, також завдяки ШІ вдалося зменшити шанс виникнення людського фактору під час розробки і дозволило збільшити шанс виявлення критичних помилок в коді на 30%. Одним з головних аспектів засобів упаковки програмного коду на одному рівні з тестуванням, є генерації коду. Вона дозволяє суттєво пришвидшити якість та швидкість написання коду, що в комбінації з генеруванням документації та тестів дає сильне покращення загальної швидкості розробки проєкту.

Генерації коду дозволяє розробнику покращити час який потрібен розробнику на йому на основну його частину роботи, це рутинне і однотипне написання коду яке швидше генерує ШІ ніж людина це пише, там немає проблем які часто зустрічають всі ШІ технології, а саме проблема складного контексту, оскільки такі задачі цікаво робити розробнику, то він може зосередитися саме на них, що підвищити загальну продуктивність на 30%.

Приклад роботи ШІ технологій для упаковки програмного коду рис. 2.13 – 2.22.



```
63 ],
64
65   'de' => [
66     'answer' => 'Antwort',
67     'question' => 'Frage',
68     'date' => 'Datum',
69   ],
70
71   |'fr' => [ You, Moments ago · Uncommitted changes
72     'answer' => 'Réponse',
73     'question' => 'Question',
74     'date' => 'Date',
75   ],
76 ]
```

Рисунок 2.13 – Технологія генерації коду на основі контексту від GitHub Copilot

```

1
2 def common_prefix(a, b):
3     """Return the common prefix of two lists."""
4     if len(a) < len(b):
5         return common_prefix(b,a)
6     for i in range(len(a)):
7         if a[i] != b[i]:
8             return a[:i]
9     return a
10
11 def test_common_prefix():
12     assert common_prefix(['a','b','c'], ['a','b','c','d']) == ['a','b','c']
13     assert common_prefix(['a','b','c'], ['b','c','d']) == ['b','c']
14     assert common_prefix(['a','b','c'], ['b','c']) == ['b','c']
15     assert common_prefix(['a','b','c'], ['a','b']) == ['a','b']
16     assert common_prefix([], []) == []

```

Рисунок 2.14 – Приклад генерації тест кейсів від технології генерації коду.  
GitHub Copilot

```

sendTweet.py 0 x  config.py 0
sendTweet.py
1 # importar librerias necesarias para el programa
2 import tweepy

```

Рисунок 2.15 – Приклад генерації коду на основі коментарів. GitHub Copilot

Приклад генерації сторінки вебсайту за допомогою коментарів наведено на рис. 2.16–2.17.

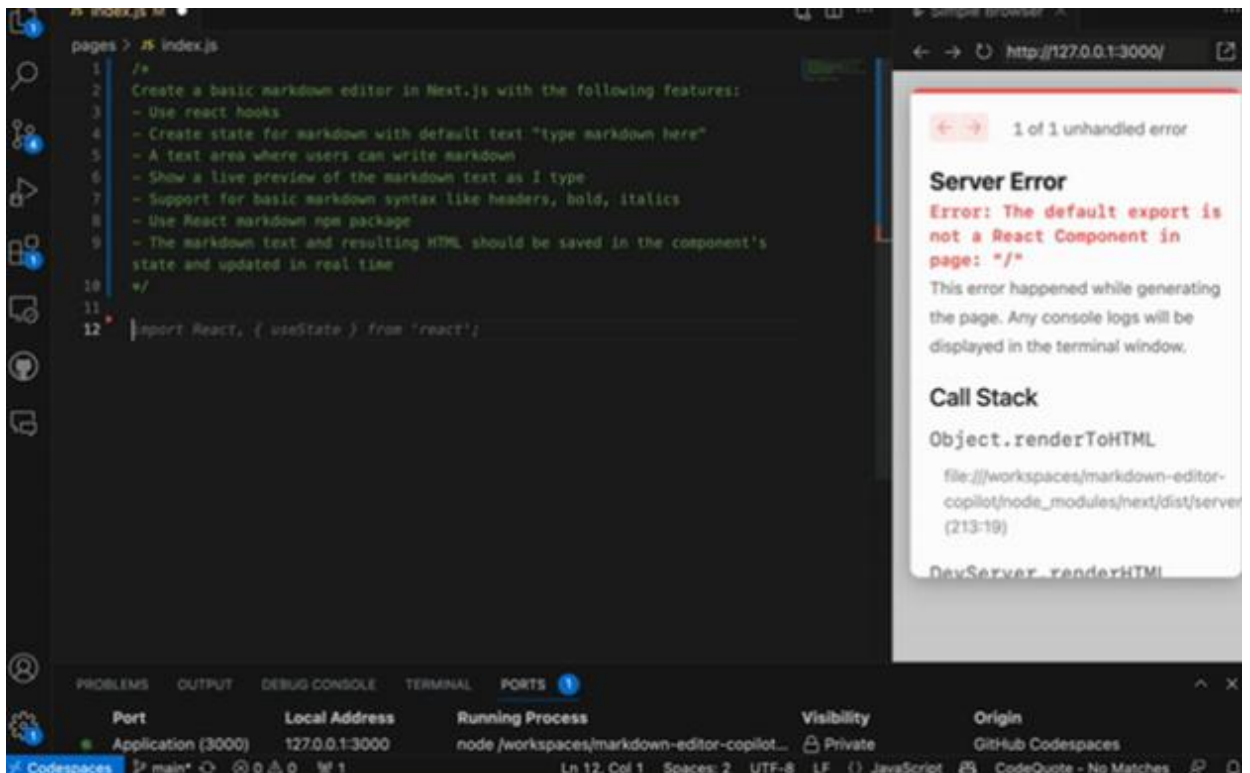


Рисунок 2.16 – Задання умови генерації за допомогою коментаря для GitHub Copilot

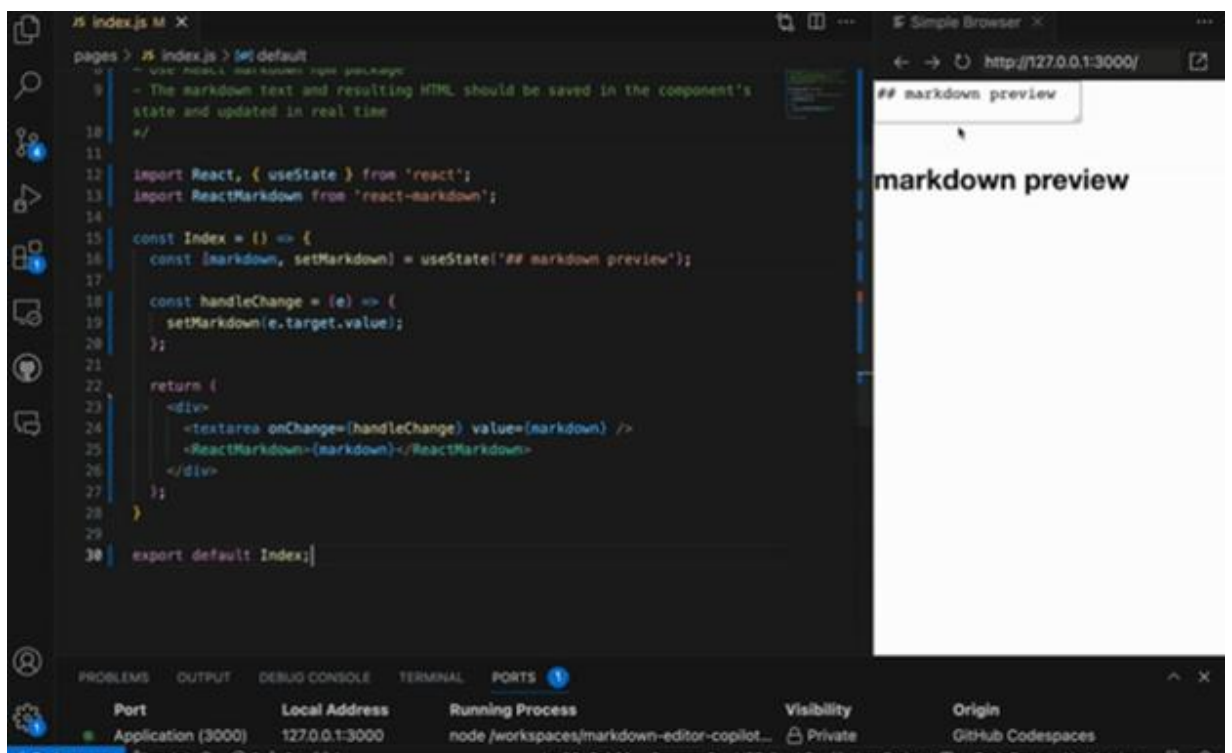


Рисунок 2.17 – Приклад згенерованою сторінки на основі коментаря заданого на рис. 2.16. GitHub Copilot

```

J Calculation.java U X
1 public class Calculation {
2
3     Run | Debug | Tabnine - generate test
4     public static void main(String[] args) {
5     }
6
7     Tabnine - generate test
8     public int findMax(int arr[]) {
9         int max = 0;
10        for (int i = 0; i < arr.length; i++) {
11            if (max < arr[i])
12                max = arr[i];
13        }
14        return max;
15    }
16
17    // find minimal element in the array
18    Tabnine - generate test
19    public int findMin(int arr[]) {
20        int min = arr[0];
21        for (int i = 1; i < arr.length; i++) {
22            if (min > arr[i])
23                min = arr[i];
24        }
25        return min;
26    }
27 }

```

Рисунок 2.18 – Приклад генерації тест-кейсів за допомогою GitHub Copilot

```

J Calculation.java U X
1 public class Calculation {
2
3     Run | Debug | Tabnine - generate test
4     public static void main(String[] args) {
5     }
6
7     Tabnine - generate test
8     public int findMax(int arr[]) {
9         int max = 0;
10        for (int i = 0; i < arr.length; i++) {
11            if (max < arr[i])
12                max = arr[i];
13        }
14        return max;
15    }
16
17    // find minimal element in the array
18    Tabnine - generate test
19    public int findMin(int arr[]) {
20        int min = arr[0];
21        for (int i = 1; i < arr.length; i++) {
22            if (min > arr[i])
23                min = arr[i];
24        }
25        return min;
26    }
27 }

J // @SuppressWarnings("all") Calculation.java
1 // Tests generated by Tabnine
2 // @SuppressWarnings("all")
3
4
5 public class CalculationTest {
6
7     @Test
8     public void findMaxTest(){
9         Calculation calc = new Calculation();
10        int [] arr = new int[]{2, 3, 5, 11, 10};
11        int max = calc.findMax(arr);
12        assertEquals(11, max);
13    }
14
15 }
16
17
18 public class CalculationTest {
19     @Test
20     void testFindMax() {
21         Calculation calculation = new Calculation();
22         int[] arr = new int[] {10, 25, 20, 15};
23         int max = calculation.findMax(arr);
24         assertEquals(25, max);
25     }
26 }
27
28
29 public class CalculationTest {
30
31     Calculation calculation = new Calculation();
32
33     @Test
34     public void testFindMax() {
35         int[] arr = {1,2,3,4,5,6};

```

Рисунок 2.19 – Приклад згенерованих тестів від GitHub Copilot.

Технології генерування коду є ключовими засобами для максимального покращення ефективності розробника. Їх ключова особливість це генерації блоків коду, а не автодоповнення, це дозволяє збільшити швидкість та ефективність розробки шлях делегування написання однотипних шаблонів коду до ШІ. Він здатний аналізувати контекст в межах файлу, тому головним правилом для його коректної роботи, це правильно названі модулі зовнішніх модулів коду, оскільки він не має можливості аналізувати їх вміст і робить висновки по їх функціоналу, тільки на основі їх назви і контексту іншого коду в якому він викликається, тому якщо буде неправильно називатися зовнішній модуль коду, то він не буде правильно працювати, але частіше всього його використання буде зав'язане не на цьому рис. 2.13. Також технологія генерації коду, окрім самої генерації коду на основі контексту, він може вільно генерувати тести для функції, щоб перевірити їх коректність роботи рис. 2.14. Одною з передових можливостей технологій генерації коду, є генерація на основі коментарів рис. 2.16 – 2.17. Вона дозволяє розробникам генерувати код за принципом опису його функціоналу, важливих моментів які там повинні використовуватися і кінцевого результату, така генерація є не дуже ефективною але її перспективи доволі серйозні, оскільки це дозволить розробникам перейти на новий спосіб написання коду. Така Генерація наразі є дуже не ефективною, оскільки якість такого коду дуже погана, технології наразі складно генерувати нормальний якісний код з правильною архітектурою, тому що всі технології генерації наразі не мають можливості створювати файли та папки в проєктах для свої потреб, що як мінімум не дозволяє технології зробити гарну архітектуру папок, тому цей напрямок поки що не широко використовується.

Одним з головних плюсів використання технологій генерації, є покращення загальної ефективності розробника на 30%. Проте наразі в технологій генерації коду є один головний мінус – це неможливість охопити весь контекст додатку. Але як тільки усунуть цю проблему, без цієї технології не буде ні однієї розробки. Проблема контексту полягає в тому, що технологія не має можливості зрозуміти всієї проблеми і складності коду яку вона повинна

згенерувати, тому наразі в таких ситуаціях розробники просто не використовують цю технологію для вирішення таких задач.

Порівняння двох технологій генерації коду на основі коментарів. Порівняння двох різних представників генерації коду на основі коментарів, покаже на якому саме зараз рівні звичайна генерація, оскільки умови для їх генерації будуть однакові, то це дозволить зрозуміти різницю в якості їх згенерованого коду. Головними метриками порівняння будуть:

- якість коду – наявність кращих практик написання коду загальноприйнятих в програмуванні;
- швидкість генерації;
- використання різних бібліотек;
- обробка помилок.

Якщо порівнювати між собою різні технології які генерують код. Ми будемо порівнювати дві технології генерації, а саме GitHub Copilot і ChatGPT.

Порівняємо код цих двох додатків, поставивши для них однаковий запит. Цей запит повинен бути англійською для кращого отримання результатів, що зв'язано з тим, що всі ці технології, в першу чергу, навчалися на текстових наборах англійською, а потім вже адаптувалися під інші мови. “Write code that will make a request to an arbitrary api, taking into account all the best practices of writing code + you are not limited to using libraries, all this must be in ReactJS, but you can choose any of the js or ts libraries to work with it” – Це Prompt, який ми задали в дві різні додатка для порівняння. Він включає в себе умову задачі, її аспекти які повинні бути реалізовані і умову самої генерації, чи обмеження генерації.

Результат:

Лістинг 2.1 – Код, згенеровий OpenAI Codex

```
import React, { useState, useEffect } from 'react';
const API_URL = 'https://api.example.com/data'; // Replace this with your API URL
const App = () => {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
```

```

const fetchData = async () => {
  try {
    setIsLoading(true);
    const response = await fetch(API_URL);
    if (!response.ok) {
      throw new Error('Failed to fetch data');
    }
    const jsonData = await response.json();
    setData(jsonData);
    setError(null);
  } catch (error) {
    setError(error.message);
  } finally {
    setIsLoading(false);
  }
};

fetchData();
}, []); // Empty dependency array to run effect only once
return (
  <div>
    <h1>API Data</h1>
    {isLoading ? (
      <p>Loading...</p>
    ) : error ? (
      <p>Error: {error}</p>
    ) : (
      <div>
        {/* Render your data here */}
        <pre>{JSON.stringify(data, null, 2)}</pre>
      </div>
    )}
  </div>
);
};
export default App;

```

## ЛІСТИНГ 2.2 – Код, згенерований GitHubCopilot

```

import React, { useEffect, useState } from 'react';
import axios from 'axios';
interface ApiResponse {
  // Define the structure of your API response here.
  // This is just a placeholder. Replace it with your actual data structure.
  data: any;
}
const ApiComponent: React.FC = () => {
  const [data, setData] = useState<ApiResponse | null>(null);
  const [error, setError] = useState<string | null>(null);
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get<ApiResponse>('https://api.example.com');

```

```

    setData(response.data);
  } catch (err) {
    setError(err.message);
  }
};
fetchData();
}, []);
if (error) {
  return <div>Error: {error}</div>;
}
if (!data) {
  return <div>Loading...</div>;
}
return (
  <div>
    { /* Render your API data here. */ }
    { /* This is just a placeholder. Replace it with your actual rendering logic. */ }
    <pre>{JSON.stringify(data, null, 2)}</pre>
  </div>
);
};
export default ApiComponent;

```

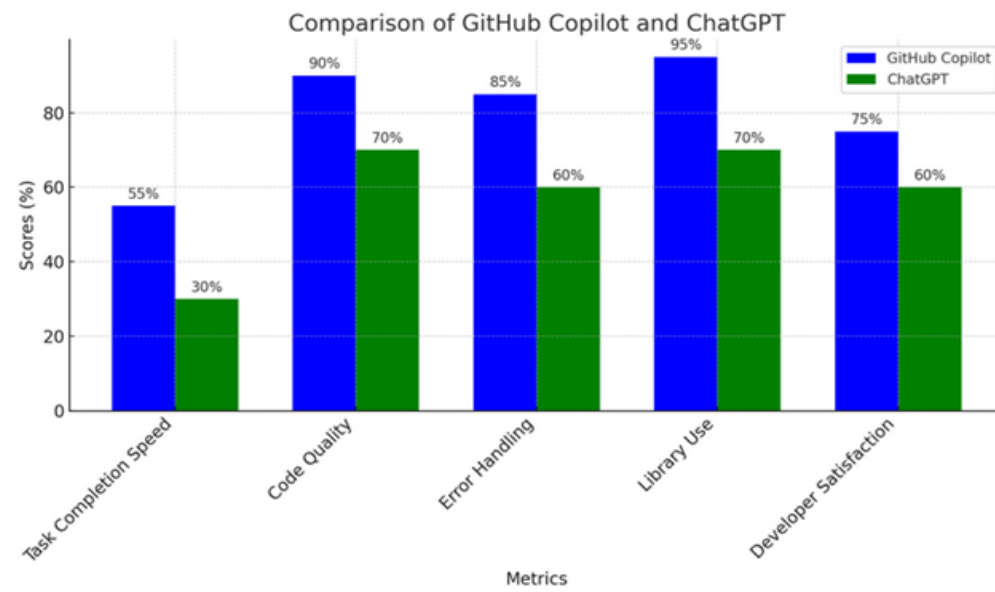


Рисунок 2.20 – Різниця між двома інструментами генерації коду

Якщо проводити аналіз коду який згенерували два додатки рис. 2.20, то тут можна зробити перший явний висновок, якщо не обмежувати кожну з AI технологій, то GitHub Copilot показав кращий результат, він генерував код повільніше в 2 рази, а саме 5 секунд, в порівнянні з ChatGPT який генерував 3 секунди але його якість на порядок вище, ключова їх різниця заключається в

виборі технологій, GitHub Copilot використав бібліотеку для запитів axios яку краще вибрати в ситуації коли вам потрібно зробити запит і контролювати його стан + GitHub Copilot використав бібліотеку на JS TypeScript яка дуже популярна і її використання вважається гарною практикою написання коду на JS. До того ж GitHub Copilot створив interface на ts в якому він описав data яка йому прийде в запиті що теж вважається гарною практикою написання коду. Обидві додатки додали обробку помилок до свого коду але в них теж є різниця. В GitHub Copilot використовується кращий патерн написання обробки помилок і коду загалом коли в свою чергу ChatGPT запропонував гірший варіант з точки зору ясності коду.

Технології генерування тестів відіграють дуже важливу роль у загальному покращенні ефективності розробки всією командою. Оскільки на швидкості роботи тестувальника зав'язаний весь проєкт, тому що проєкт не можна запустити в користування звичайними користувачами, допоки всі модулі проєкта не будуть протестовані. Покриття всіх модулів проєкту тестами доволі довгий процес, тому для покращення ефективності роботи тестувальника, можна використовувати ШІ, який буде займатися покриттям тестів простих модулів проєкту, а таких більшість. Це дозволяє суттєво пришвидшити час який потрібен на тестування і зменшити час який потрібен на написання тестових кейсів на 40–50%, також на 30% краще дозволяє виявляти критичні помилки. Завдяки цьому тестувальник може покривати тестами складні модулі які не може обробити ШІ, а поки він це робить ШІ буде покривати тестами більш прості модулі, що сильно економить час.

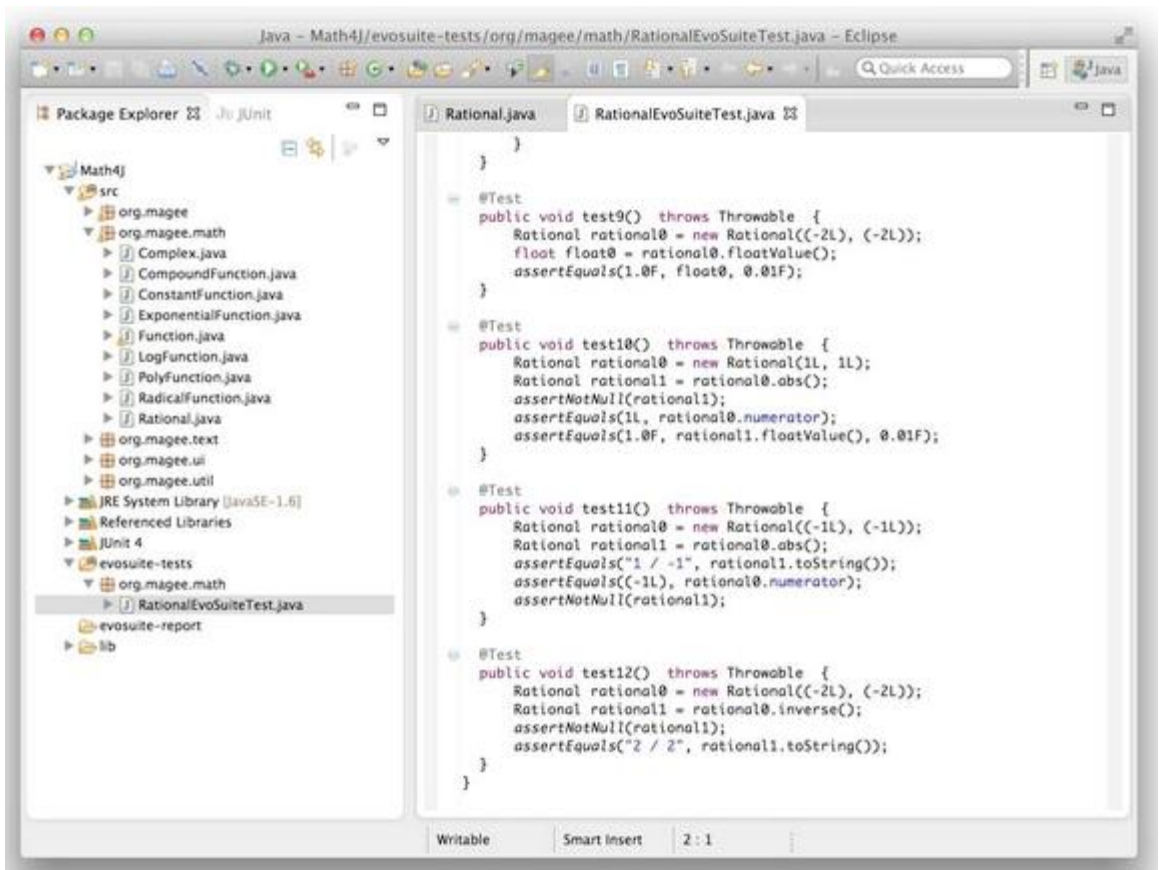


Рисунок 2.21 – Приклад автоматичної генерації тестів для коду від EvoSuite

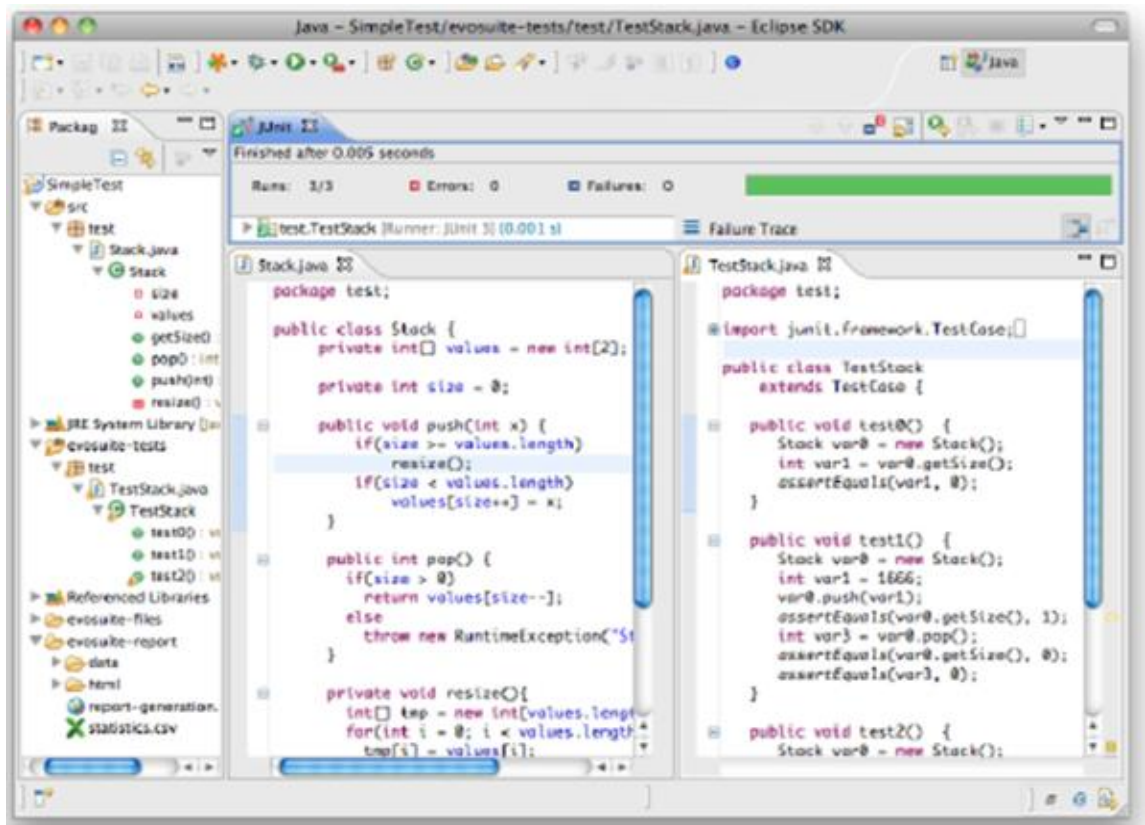


Рисунок 2.22 – Приклад автоматичної генерації тестів для коду від EvoSuite

Головним плюсом, є покращення ефективності розробки загалом. Завдяки швидкому покриттю всіх готових модулів коду тестами це дозволило зменшити час потрібний на тестування на 40–50%, що в межах проєкту і однієї команди доволі велика цифра. Головним мінусом використання цих технологій, така сама як і проблема в технологіях генерації коду або автодоповнення, це не можливість зрозуміти весь контекст і складність модуля. Через це все складне і комплексне, що повинен генерувати ШІ, він генерує погано. Питання генерування документації коду, стоїть доволі гостро в великих проєктах в великою командою, оскільки зміна кадрів на великих проєктах, не є чимось новим, то новим розробникам треба швидко влитися в контекст проєкту, проте інколи проєкти можуть бути настільки складними і комплексними, що для розуміння що там відбувається, можуть піти дні але це питання, може вирішити технології документації коду, вони генерують документацію на основі вибраних модулів, і описують її роботу, та спосіб їх взаємодії. Це дозволяє розробникам швидко зрозуміти модулі з якими треба працювати розробнику і як найшвидше початий йому працювати.

```

/!*
 * |
 * @param FileNo
 * @param Directory
 * @param Filename
 * @param CUID
 * @return
 */
unsigned EmitDwarfFileDirective(unsigned FileNo, StringRef Directory,
                               StringRef Filename,
                               unsigned CUID = 0) override;
void EmitDwarfLocDirective(unsigned FileNo, unsigned Line,
                           unsigned Column, unsigned Flags,
                           unsigned Isa, unsigned Discriminator,
                           StringRef FileName) override;
MCSymbol *getDwarfLineTableSymbol(unsigned CUID) override;

```

Рисунок 2.23 – Автоматичне генерування документації для коду на основі написаного модуля коду. Doxygen

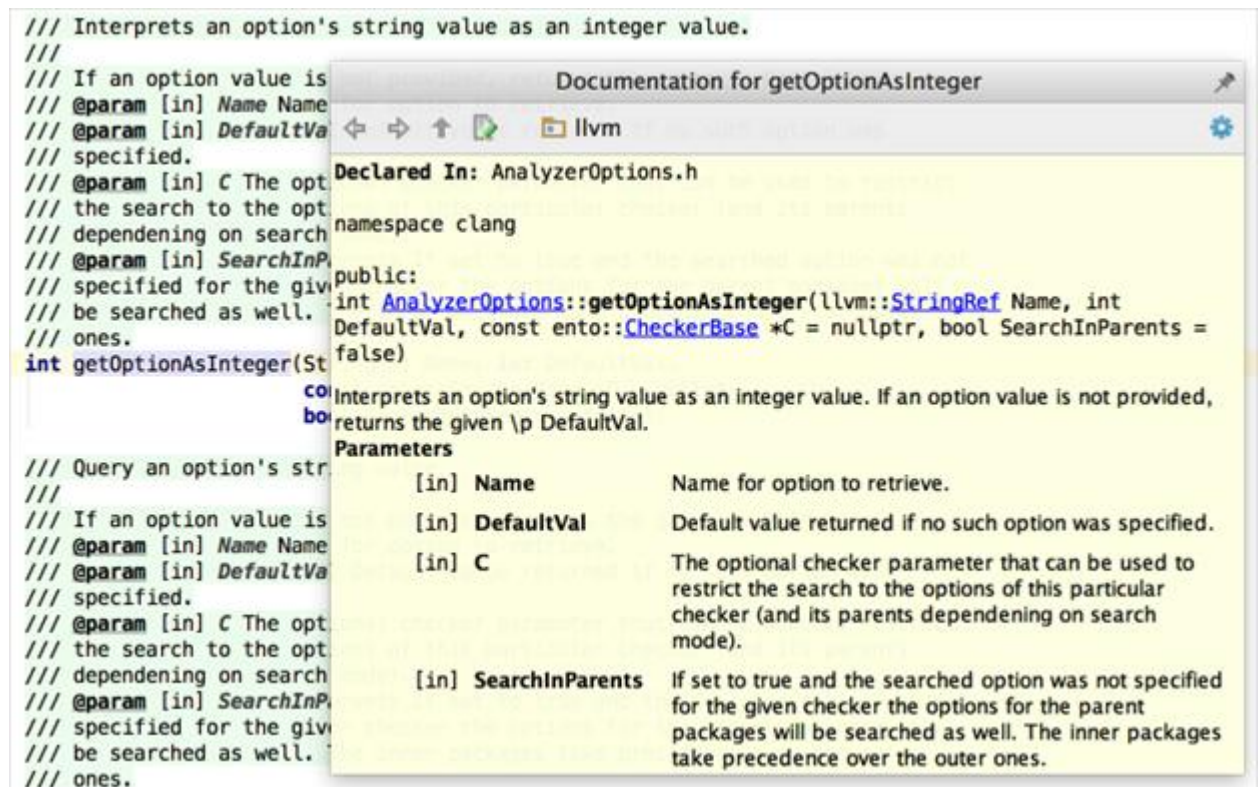


Рисунок 2.24 – Приклад вигляду автоматично згенерованої документації за допомогою Doxygen

Питання документації проєкту стоїть доволі гостро, оскільки на її написання потрібен час, який можна було потратити на написання нових модулів або вдосконалення старих, але завдяки ШІ-технологіям для генерації документації цей час можна зменшити на 80%, що дозволяє покращити загальну підтримку проєкту для подальших команд. Головним плюсом використання цих технологій, є покращення подальшої підтримки поточного проєкту іншими розробниками. Використання технологій упаковки програмного коду дозволили зменшити час потрібний на генерування тест кейсів на 40–50%, а також дозволило краще виявляти критичні помилки на 30%, також це дозволило підвищити ефективність розробників на 30% завдяки генерації коду, що надало можливість зменшити шанс людських помилок під час розробки, а генератори документації, надали кращу підтримку проєкту в подальшому, завдяки краще задокументованому коду.

### **2.3 Розгортання програмного забезпечення та моніторинг додатка з використанням штучного інтелекту**

ШІ технології моніторингу додатку дозволяють в процесі реального часу отримувати інформацію про поточний стан проєкту завдяки збору і аналізу даних які вони отримують під час процесу CI/CD, воно включає процес аналізу і моніторингу проєкту, що дозволяє отримувати логи про виявлені помилки або про незвичайні патерни поведження системи. Також технології моніторингу додатку дозволяють отримувати прогнози щодо навантаження та витрат, що дозволяє забезпечити оптимальне використання ресурсів. Також вони мають можливість виявлення потенційних проблем з безпекою, і при виникненні таких проблем, вони намагаються вжити відповідні заходи для запобігання їх або спроби їх виправлення. Одна з популярних технологій для забезпечення моніторингу додатка це DataDog APM. Ця технологія надає можливість моніторингу додатку. Завдяки цьому розробники можуть отримати інформацію про результати тестів для кожної з версій проєкту, також завдяки ШІ стало можливе одне з потужніших фіч технологій для моніторингу – це прогнозування. Кожна з технологій надає можливість отримати звіт з прогнозуванням можливої проблеми в коді або дізнатися можливі навантаження чи потенційні витрати. Це дозволяє тверезо розуміти ситуацію для розробників, що в свою чергу покращує саму підтримку проєкту. Також технології моніторингу надають доступ до інформації завдяки якій вони і роблять прогнози. Це результати тестів, або кількість запитів на сайт в кожен відрізок часу, час завантаження сторінок, кількість одночасних користувачів на сайті, навантаження серверної частини проєкту і її стан загалом, можливість отримати данні про найбільш ресурсомісткі модулі коду. Або робити аналіз користувацьких операції і їх вплив на систему. Це технології формують звіти які містять всю цю впорядковану інформацію рис. 2.25 – 2.26. В свою чергу є інша технологія Kubiya яка надає можливість інтегрувати її в текстове середовище, де вся команда зможе робити зміни в один запит. Вона надає загальну інформацію про проєкт з прогнозами на помилки і

витрати, а також має можливість аналізувати ресурси проєкту і змінювати їх через текст рис. 2.27.

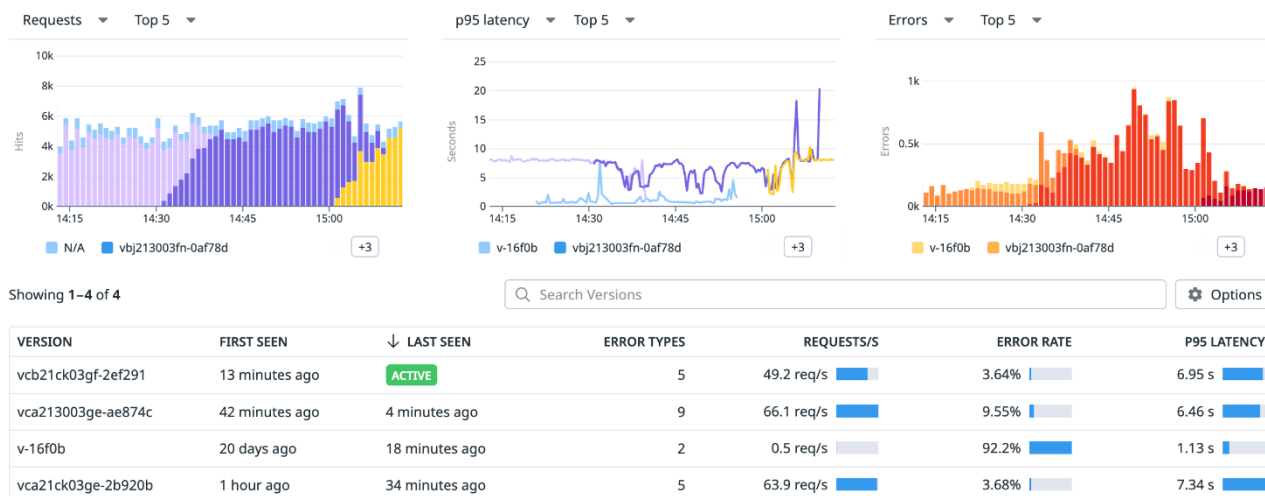


Рисунок 2.25 – Приклад звіту за 4 версії проєкту. DataDog APM

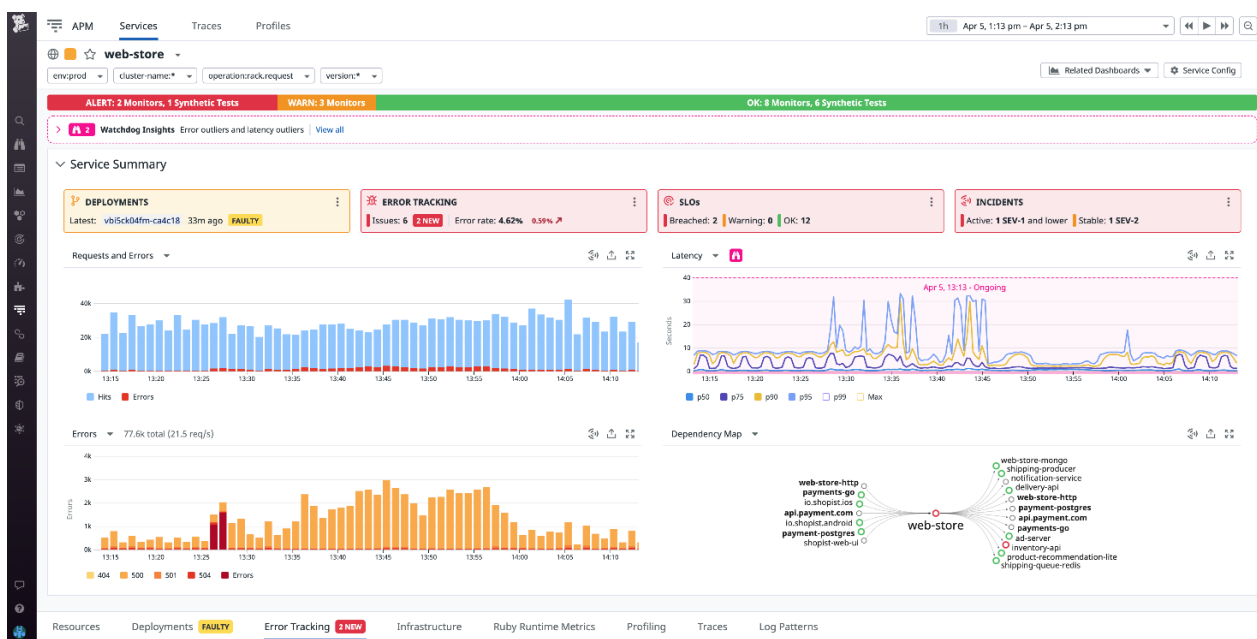


Рисунок 2.26 – Приклад звіту з інформацією про залежності між модулями, та інформацією про помилки та кількість запитів протягом дня. DataDog APM



Рисунок 2.27 – Приклад роботи Kubiya

Використання технологій моніторингу дозволяє сильно підняти ефективність підтримки вже розгорнутих додатків, шляхом отримання звітів через вибрану технологію, це дозволило покращити ефективність роботи додатку на 10–15% завдяки автоматичному тестуванню, на 15% зменшити час який потрібен на відновлення роботи додатку і покращення його доступності, а також це дозволило оптимізувати ресурси всього проєкту на 10%. Що має під собою суттєві цифри які дозволяють ґрунтовно покращити роботу розгорнутого проєкту. Головним плюсом їх використання, це можливість отримати всю уструктуровану інформацію з прогнозами помилок і витрат в одному місці, що дозволяє зменшити час потрібний для аналізу проєкту, також технології надають функціонал який сам намагається усунити проблеми які могли виникнути під час роботи додатку. Це дозволило підняти загальну підтримки ефективність розгорнутого додатку на 10–15%.

Головний мінус цих технологій – це безпека, оскільки вся інформація яка збирається про додаток, а оскільки для аналізу йому треба мати доступ до всіх сервісів, що використовуються, то може піднятися питання безпеки, оскільки все це буде зберігатися на хмарі, і за безпеку цих даних буде відповідати інша компанія.

## РОЗДІЛ 3

### АНАЛІЗ РЕЗУЛЬТАТІВ ВПРОВАДЖЕННЯ ТЕХНОЛОГІЙ НА ОСНОВІ АІ В ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

#### 3.1 Кейси інтеграції АІ–інструментів в цикл програмної розробки.

Google використовує штучний інтелект у своїх продуктах та послугах, включаючи машинне навчання для Google пошуку, розпізнавання мови у Google Translate, управління даними у Google Photos та інші. Завдяки інтеграції ШІ до Google пошуку, вдалося отримати результат розуміння бажань користувача завдяки повному аналізу вподобань і контексту запитів користувача. ШІ аналізує ситуацію в якій користувач робить запит, що він дивився і шукав до цього, і загальна ситуація в його країні. Це надало реалізувати потужну пошукову систему від Google. Google Translate за допомогою інтеграції ШІ навчився розпізнавати краще структура та контекст речення, а також надав функціонал для голосових перекладів. Інтеграція ШІ в Google Photos надала можливість Google краще реалізувати персоналізацію і управління даними за рахунок аналізу на основі ШІ.

Facebook використовує АІ для покращення своїх алгоритмів рекомендацій, фільтрації контенту, розпізнавання облич та багато іншого. ШІ в Facebook став краще класифікувати фото до реальних людей, що створило потужну систему за допомогою якої знайти потрібну людину стало в рази легше. Також ШІ в Facebook пропонує автоматично тегувати людей які знаходяться на фото, що значно покращило враження користувача від програми.

Netflix використовує АІ для рекомендацій контенту своїм користувачам. Алгоритми аналізують попередні перегляди та уподобання користувачів, щоб запропонувати їм персоналізований контент, що збільшує задоволення від перегляду та збільшує час, проведений на платформі.

Amazon використовує АІ для автоматизації свого процесу логістики, управління запасами та аналізу великих обсягів даних. Це допомагає покращити ефективність поставок, зменшити час доставки та покращити обслуговування клієнтів. А також Amazon використовує ШІ для розумного пошуку, який

дозволяє краще підлаштуватися під кожного користувача, що значно підвищило загальну задоволеність користування сервісом. Також Amazon практикують ШІ чат-боти які видить підтримку з клієнтами в розрізі простих питань. Amazon також використовує ШІ для аналізу відгуків, щоб як найкраще покращити користування своїми сервісами.

Tesla використовує AI для автономного водіння у своїх електромобілях. Це дозволяє автомобілям Tesla самостійно реагувати на дорожні умови та інші транспортні засоби, забезпечуючи безпеку та зручність водіння шляхом прийняття рішень в реальному часі.

### **3.2 Вимірювання покращень від задіявання ШІ-інструментів**

В питанні впливу ШІ технологій на ефективність розробки, нема чіткої відповіді, оскільки з однієї сторони, вплив ШІ технологій можна оцінити по кількості виконаних задач, а з іншої повного ефекту від використання виміряти не можна. Це зв'язано з тим, що доволі велику роль в цьому всьому відіграє ментальний відпочинок, тому усіх благ які надають ШІ технології в повній мірі не можна оцінити. Також коли повстає питання про інтеграцію, для досягнення максимального ефекту, потрібно інтегрувати ШІ технології на кожен етап розробки для подальшого їх з'єднання в одну еко систему, яка в рази підвищує їх загальну ефективність на проєкті. Тому для отримання повного ефекту, необхідна повна інтеграція всіх технологій на усіх можливих стадіях розробки.

Для вимірювання впливу ми беремо дослідження від GitHub в якому проводилися дослідження впливу інтеграції всіх ШІ технологій в процес розробки, для цього дослідження відібрали групу людей в яких заміряли їх ефективність роботи в перший тиждень без ШІ технологій, та їх відгуки про цей тиждень і провели ще один тиждень досліджень але тепер вже з інтегрованими ШІ технологіями в всі процеси розробки. Середні результати за дослідженням були такі:

– використання AI технологій дозволило розробникам скоротити час розробки на 50%, витрачений на рутинні завдання з кодування, і на 30%

підвищили загальну продуктивність. Шляхом делегування повторюваних мікро задач до AI, що дозволяє зменшити людський фактор під час розробки. Найбільш впливовою AI технологією, є автогенерація коду на основі контексту, за проведеними дослідженнями, його коректність на дуже високому рівні, що дозволяє його вільно імплементувати в процес розробки;

- AI-технології для генерування показали, що автоматизації тестування зі штучним інтелектом дозволяє скоротити час, необхідний для створення тестових кейсів, на 40–50% і дозволяє краще виявляти критичні помилки на 30% порівняно з традиційними методами тестування;

- технології AI-конвеєрів CI/CD показало, що середній час збірки скоротився на 20%, а кількість успішних розгортань збільшилася на 15%. Також частота розгортання збільшилася на 25%, а кількість помилок при розгортанні зменшилася на 20%;

- технології статичного аналізу коду зі штучним інтелектом показало, що виявлення вразливостей безпеки збільшилося на 25%, а кількість хибних спрацювань зменшилася на 30%, що призвело до створення більш безпечних і надійних кодових баз.

- інструменти управління логів на основі AI дозволили пришвидшити час виявлення аномалій на 35%, а час, необхідний для вирішення операційних проблем, скоротився на 40%;

- інструменти якості коду на основі AI продемонстрували покращення супроводжуваності коду на 20% та зменшення кількості помилок після релізу на 15%.

Всі люди, які були в задіяні експерименті, давали відгуки до і після використання ШІ-технологій, і деякі з них відмітили, що надійність коду, створеного штучним інтелектом, і точність автоматизованих тестів були гірші ніж зазвичай у складноконтекстних ситуаціях. Вони відмітили, що наразі їх повна інтеграція в ролі окремого розробника допоки не можлива, оскільки їх контекст сильно обмежений можливостями ШІ і ресурсами які їх потрібні для такої роботи.

Майбутнє штучного інтелекту в інженерії програмного забезпечення виглядає багатообіцяючим, а дослідження, що тривають, спрямовані на розширення можливостей штучного інтелекту та його більш плавну інтеграцію в робочі процеси розробки. Основні напрямки включають

Наразі тривають дослідження, спрямовані на покращення здатності генерувати тестові кейси з описів природною мовою, підтримуючи такі практики, як Test-Driven Development (TDD). Розробка адаптивних систем ШІ, які безперервно навчаються на основі результатів тестування для створення більш ефективних тестових кейсів, є ключовим напрямком досліджень. Ці системи можуть динамічно коригувати свої стратегії на основі зворотного зв'язку, вдосконалюючись з часом. Розширення застосування ШІ за межі традиційного кодування і тестування до таких сфер, як управління проектами, оцінка ризиків і дизайн користувацького досвіду, є захоплюючою перспективою. Інструменти ШІ можуть допомогти в прогнозуванні проектних ризиків, оптимізації розподілу ресурсів і поліпшенні користувацьких інтерфейсів на основі аналізу поведінкових даних.

Отже, інтеграція інструментів штучного інтелекту в інженерію програмного забезпечення має величезний потенціал для підвищення продуктивності, покращення якості програмного забезпечення та оптимізації процесу розробки. Поточні дослідження та реальні застосування продовжують демонструвати цінність ШІ, роблячи його життєво важливим компонентом сучасної практики розробки програмного забезпечення.

## ВИСНОВКИ

У рамках цієї роботи було проведено детальний аналіз та порівняння різних інструментів і технологій, що використовуються в інженерії програмного забезпечення. Оцінювалася їх ефективність, швидкість, якість згенерованого коду, а також зручність використання в різних контекстах розробки. Основна увага була приділена наступним аспектам:

- Здатність інструментів враховувати контекст та специфічні вимоги проєкту.
- Швидкість та ефективність генерації коду.
- Якість та узгодженість згенерованого коду з точки зору стилю та дотримання стандартів.
- Зручність інтеграції інструментів в робочий процес розробника.

Одним із важливих аспектів є здатність ШІ-інструментів враховувати специфічні вимоги кожного окремого проєкту. Це вимагає високої точності та гнучкості алгоритмів, які використовуються для автодоповнення та генерації коду. Інструменти, що можуть адаптуватися до різних контекстів та вимог, мають значні переваги перед менш гнучкими аналогами. Швидкість та ефективність генерації коду є ще одним критично важливим показником. Інструменти, які здатні швидко та точно генерувати код, можуть значно зменшити час розробки та підвищити продуктивність команди розробників. Проте, швидкість не повинна бути досягнута за рахунок якості, тому необхідний баланс між цими двома параметрами. Якість згенерованого коду є важливим критерієм для оцінки ефективності ШІ-інструментів. Код повинен бути не лише функціональним, але й відповідати стандартам стилю та найкращим практикам програмування. Узгодженість коду допомагає забезпечити його легкість в обслуговуванні та розширенні, що є ключовим фактором для довготривалих проєктів. Зручність інтеграції інструментів у робочий процес розробника також відіграє важливу роль. Інструменти повинні бути легко інтегровані у існуючі середовища розробки та підтримувати робочі процеси команди. Це включає

підтримку різних мов програмування, сумісність з існуючими інструментами та платформами, а також надання зручного інтерфейсу для користувачів.

Результати проведеного аналізу показали, що технології автодоповнення та генерації коду мають значний потенціал для підвищення продуктивності розробників. Проте, їх ефективне використання вимагає глибокого розуміння можливостей та обмежень кожного конкретного інструмента, а також вміння адаптувати їх до потреб конкретного проєкту. Впровадження ШІ в процес розробки програмного забезпечення відкриває нові горизонти для оптимізації та автоматизації рутинних завдань, дозволяючи розробникам зосередитися на більш креативних та стратегічних аспектах роботи.

На основі проведеного аналізу основних сфер застосування AI-технологій можна зробити наступні висновки. Використання AI технологій вигідне для збільшення ефективності роботи розробника. В імплементації цих технологій теж нема проблем. Головні аспекти зазначених технологій:

- Автодоповнення коду: Головним недоліком технологій автодоповнення є неможливість повного охоплення контексту, що змушує розробників часто переписувати згенерований код. Попри це, автодоповнення прискорює процес кодування, зменшуючи час на введення повторюваного коду, допомагає запобігти помилкам, сприяє узгодженості стилю кодування, та виступає навчальним інструментом для початківців. Однак, іноді автодоповнення може пропонувати нерелевантні варіанти, що залежить від якості контекстного аналізу та навчальних даних

- Генерація коду: Використання технологій генерації коду, таких як GitHub Copilot, значно пришвидшує розробку, дозволяючи створювати повні блоки коду з урахуванням контексту. Важливо зазначити, що якість згенерованого коду залежить від навчальних даних та запиту розробника. GitHub Copilot, показав кращі результати в порівнянні з іншими інструментами завдяки використанню бібліотек, що є стандартом у розробці.

- Обробка природної мови: Технології обробки природної мови (NLP) мають великий потенціал для використання в різних сферах, включаючи

технічну підтримку та генерацію тексту. Це дозволяє створювати моделі поведінки для AI, що можуть аналізувати та відповідати на запити користувачів, що підвищує ефективність та якість взаємодії.

Загалом, імплементація AI-технологій має як переваги, так і недоліки, що вимагає чіткого розуміння, що ви хочете від цих технологій і які проблеми вони повинні вам закрити. Автодоповнення та генерація коду здатні значно прискорити процес розробки, але потребують уважного контролю якості згенерованого коду. Використання NLP-технологій відкриває нові можливості для автоматизації та покращення взаємодії з користувачами.

В майбутньому ці технології будуть ставати все краще, і будуть мати все менше і менше проблем пов'язаних з їх використанням і проблема обмеженості контексту в майбутньому зникне і їх використання стане обов'язковим наступним кроком циклу розробки якісного ПО але потім стане нове питання, питання непотрібності деяких сфер в програмуванні оскільки їх поля відповідальності віддадуть AI технологіям, яким треба буде менше плати.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is AI. URL: <https://www.ibm.com/topics/artificial-intelligence> (дата звернення: 20.04.2024).
2. Kite vs Visual Studio IntelliCode. URL: [https://www.reddit.com/r/vscode/comments/gyrsck/kite\\_vs\\_visual\\_studio\\_intellicode\\_extensions\\_for/](https://www.reddit.com/r/vscode/comments/gyrsck/kite_vs_visual_studio_intellicode_extensions_for/) (дата звернення: 20.04.2024).
3. Best AI Extensions for Visual Studio Code. URL: <https://medium.com/@jeremydickey/exploring-the-best-ai-extensions-for-visual-studio-code-3adb65861408> (дата звернення: 20.04.2024).
4. Kite official site. URL: <https://www.kite.com/blog/product/kite-is-saying-farewell/> (дата звернення: 20.04.2024).
5. Best Coding Assistants. URL: <https://levelup.gitconnected.com/ai-coding-assistants-battle-tabnine-vs-intellicode-vs-kite-6aca84bfe224> (дата звернення: 20.04.2024).
6. Kite vs Visual Studio IntelliCode. URL: [https://www.reddit.com/r/vscode/comments/gyrsck/kite\\_vs\\_visual\\_studio\\_intellicode\\_extensions\\_for/](https://www.reddit.com/r/vscode/comments/gyrsck/kite_vs_visual_studio_intellicode_extensions_for/) (дата звернення: 20.04.2024).
7. Deep code official site. URL: <https://snyk.io/platform/deepcode-ai/>
8. GitHub Copilot official site. URL: <https://docs.github.com/en/copilot/using-github-copilot/getting-started-with-github-copilot> (дата звернення: 10.05.2024).
9. Best Alternatives to GitHub Copilot. URL: <https://medium.com/@gargg/top-10-ai-coding-assistants-for-developers-in-2023-64a6676a651> (дата звернення: 10.05.2024).
10. AI Coding Assistants for Developers URL: <https://medium.com/@jeremydickey/exploring-the-best-ai-extensions-for-visual-studio-code-3adb65861408> (дата звернення: 10.05.2024)
11. Code climate official site URL: <https://codeclimate.com/> (дата звернення: 15.05.2024)

12. CodeSonar official site URL: <https://codesecond.com/our-products/codesonar/> (дата звернення: 15.05.2024)
13. AI tech recommendation URL: <https://www.pragmaticcoders.com/blog/make-sure-your-developers-use-these-ai-tools> (дата звернення: 15.05.2024)
14. DeepSource official site [Електронний ресурс] – Режим доступу до ресурсу: <https://deepsources.com/>
15. EvoSuite official site URL: <https://www.evosuite.org/> (дата звернення: 15.05.2024)
16. ChatGPT official site URL: <https://chatgpt.com/> (дата звернення: 26.05.2024)
17. Watson Assistant official site URL: <https://www.ibm.com/products/watsonx-assistant> (дата звернення: 26.05.2024)
18. ChatGPT vs Watson Assistant URL: <https://www.linkedin.com/pulse/chatgpt-vs-watson-assistant-alex-vianna/> (дата звернення: 27.05.2024)
19. CodeX vs GitHub Copilot URL: <https://www.spaceo.ai/blog/codex-vs-github-copilot/> (дата звернення: 27.05.2024)
20. ChatGPT vs Watson Assistant URL: <https://openai.com/index/openai-codex/> (дата звернення: 27.05.2024)
21. What is the difference between ChatGPT and IBM Watson? URL: <https://www.quora.com/What-is-the-difference-between-ChatGPT-and-IBM-Watson> (дата звернення: 27.05.2024)
22. ChatGPT vs Watson Assistant URL: <https://www.linkedin.com/pulse/chatgpt-vs-watson-assistant-alex-vianna/> (дата звернення: 30.05.2024)
23. OpenAI Codex official site URL: <https://openai.com/index/openai-codex/> (дата звернення: 30.05.2024)

24. OpenAI Codex vs. GitHub Copilot URL: <https://www.geeksforgeeks.org/openai-codex-vs-github-copilot/> (дата звернення: 30.05.2024)
25. Doxygen official site URL: <https://www.doxygen.nl/> (дата звернення: 30.05.2024)
26. Best tool for documenting code URL: [https://www.reddit.com/r/cpp/comments/15yw0pv/best\\_tool\\_for\\_documenting\\_c\\_code/](https://www.reddit.com/r/cpp/comments/15yw0pv/best_tool_for_documenting_c_code/) (дата звернення: 30.05.2024)
27. Elasticsearch official site URL: <https://www.elastic.co/elasticsearch> (дата звернення: 30.05.2024)
28. Elasticsearch vs. Traditional Databases URL: <https://medium.com/@rajeevprasanna/elasticsearch-vs-traditional-databases-diving-into-elastic-searchs-strengths-c6f55b9b449f> (дата звернення: 30.05.2024)
29. Pros and cons of Elasticsearch and SQL databases URL: <https://www.quora.com/What-are-the-pros-and-cons-of-Elasticsearch-and-SQL-databases> (дата звернення: 31.05.2024)
30. Elasticsearch vs. Traditional Databases URL: <https://medium.com/@rajeevprasanna/elasticsearch-vs-traditional-databases-diving-into-elastic-searchs-strengths-c6f55b9b449f> (дата звернення: 31.05.2024)
31. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot URL: <https://ar5iv.labs.arxiv.org/html/2302.06590> (дата звернення: 03.06.2024)
32. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT URL: <https://ar5iv.labs.arxiv.org/html/2304.10778> (дата звернення: 03.06.2024)

33. Understanding the Application and Impact of AI in Software Development Life Cycle URL: <https://dzone.com/articles/application-and-impact-of-ai-in-the-sdlc> (дата звернення: 03.06.2024)
34. Measuring GitHub Copilot's Impact on Productivity URL: <https://cacm.acm.org/research/measuring-github-copilots-impact-on-productivity/> (дата звернення: 03.06.2024)
35. Measuring GitHub Copilot's Impact on Productivity URL: <https://www.tabnine.com/blog/measuring-productivity-in-an-ai-world/> (дата звернення: 03.06.2024)
36. Measuring productivity in an AI world URL: <https://cacm.acm.org/research/measuring-github-copilots-impact-on-productivity/> (дата звернення: 03.06.2024)
37. Tabnine Unveils Tabnine Chat, Boosting Developer Productivity Through the Power of Generative AI URL: <https://www.globenewswire.com/en/news-release/2023/06/29/2697038/0/en/Tabnine-Unveils-Tabnine-Chat-Boosting-Developer-Productivity-Through-the-Power-of-Generative-AI.html> (дата звернення: 03.06.2024)
38. Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach. URL: <https://thuvienso.hoasen.edu.vn/v/web/viewer.html?file=/bitstream/handle/123456789/8967/Contents.pdf?sequence=3&isAllowed=y> (дата звернення: 16.06.2024)
39. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. URL: <https://www.deeplearningbook.org/> (дата звернення: 16.06.2024)
40. Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective URL: [https://books.google.com.ua/books?hl=en&lr=&id=RC43AgAAQBAJ&oi=fnd&pg=PR7&dq=Murphy,+K.+P.+\(2012\).+Machine+Learning:+A+Probabilistic+Perspective+-+MIT+Press.&ots=ungCeARw17&sig=w4aB6qXW\\_hUbkWjkFQ7u8b3tWhU&redir\\_esc=y#v=onepage&q=Murphy%2C%20K.%20P.%20\(2012\).%20Mach](https://books.google.com.ua/books?hl=en&lr=&id=RC43AgAAQBAJ&oi=fnd&pg=PR7&dq=Murphy,+K.+P.+(2012).+Machine+Learning:+A+Probabilistic+Perspective+-+MIT+Press.&ots=ungCeARw17&sig=w4aB6qXW_hUbkWjkFQ7u8b3tWhU&redir_esc=y#v=onepage&q=Murphy%2C%20K.%20P.%20(2012).%20Mach)

[ine%20Learning%3A%20A%20Probabilistic%20Perspective%20-%20MIT%20Press.&f=false](#) (дата звернення: 16.06.2024)