Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

за спеціальністю 123 «Комп’ютерна інженерія»

Тема наукової роботи: МЕТОДИ ТА ЗАСОБИ ОПТИМІЗАЦІЇ АВТОМАТИЗОВАНОГО ГЕНЕРУВАННЯ ПРОГРАМНОГО КОДУ ЗАСОБАМИ ШТУЧНОГО ІНТЕЛЕКТУ

Виконав _____ Д. Ю. Духов

Керівник роботи _____ А. І. Купін

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2025

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти магістр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 20__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

_____
(прізвище, ім'я, по батькові)

1. Тема роботи _____
_____
_____

керівник роботи _____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "___"_____ 20__року №__

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи _____
_____
_____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)_____
_____
_____
_____
_____
_____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
_____
_____
_____
_____

6. Консультанти розділів роботи

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

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 123 сторінки, 51 рисунків, 3 додатоків, використані джерела 36.

Об’єкт аналізу – методи та засоби оптимізації автоматизованого генерування прогрманого колу засобами штучного інтелекту.

Проєкт складається з 4 розділів.

Перший розділ. Огляд сучасних підходів та проблематика автоматизованого генерування програмного коду засобами штучного інтелекту. Досліджується як текстовий запит перетвоюється у програмний код. Використання в шутчному інтелкті Abstract Syntax Tree, Retrieval-Augmented Generation, Self-Correction, Low-Confidence та Mixture of Experts. Аналіз проблематика використання штучного інтелекту як коректність, якість, безпека, інтелктуальна власність та наслідки для розробників.

Доугий розділ. Порівняльний аналіз видів зверненян до штучного інтелекту як веб-чати, локальній машин та розширення до IDE. Опис інструментів штучномого інтелекту у сфері генерації коду як автодоповнення,тестування, рефакторинг, документація та аналіз коду.

Третій розділ. Оптимізація роботи агента генерації коду. Налаштування та оптимізація агента генерації коду в IDE. Використання Multi-Context Protocole для інтеграція інстурментів для аналізу Code Smell, покриття тестами, документація коду та безпековий аудит. Також виконання роботи у віртуальному середовщій. Передача повного контуролю штучному інтелекту над середовищем написанням коду.

Четвертий розділ. Аналіз та прогнозування трендів штучного інтелекту у кодуванні. Використання апроксимація трендів для прогназування розвитку штучного інтелекту. Комплексний аналіз бенчмарків шутчного інтелекту у кодуванні та їх перенасичення. Ключові спостереження та майбутні напрямки розвитку даного напрямку.Результати тестування оптимізованого штучного інтелекту під написання програмного коду.

ШТУЧНИЙ ІНТЕЛЕКТ, ГЕНЕРАТИВНИЙ ШТУЧНИЙ ІНТЕЛЕКТ, АРХІТЕКТУРА ТРАНСОФОРМЕРИ, ГЕНЕРАЦІЯ ПРОГРАМНОГО КОДУ, СЕРЕДОВИЩЕ НАПИСАННЯ КОДУ, ВЕЛИКІ МОВЛЕНЕВІ МАШИНИ, НАСЛІДКИ ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ, ОПТИМІЗАЦІЯ АВТОМАТИЧНОГО ПРОЦЕСУ НАПИСАННЯ КОДУ, ВЕБ-ЧАТИ, ЛОКАЛЬНІЙ МАШИНИ, ДОПОВНЕННЯ ДО СЕРЕДОВИЩА РОЗРОБКИ КОДУ, АНАЛІЗ БЕЙЧМАРКІВ, АПРОКСИМАЦІЯ ТРЕНДІВ.

Bachelor’s qualifying paper: 123 pages, 51 figures, 3 appendices, 36 sources used.

The object of analysis - methods and tools for optimizing automated generation of a programmed circuit using artificial intelligence.

The project consists of 4 sections.

The first section. Review of modern approaches and issues of automated generation of program code using artificial intelligence. It is studied how a text query is transformed into program code. Use in artificial intelligence Abstract Syntax Tree, Retrieval-Augmented Generation, Self-Correction, Low-Confidence and Mixture of Experts. Analysis of the issues of using artificial intelligence as correctness, quality, security, intellectual property and consequences for developers.

Long section. Comparative analysis of types of appeals to artificial intelligence as web chats, local machines and extensions to IDE. Description of artificial intelligence tools in the field of code generation as autocompletion, testing, refactoring, documentation and code analysis.

The third section. Optimization of the work of the code generation agent. Setting up and optimizing the code generation agent in the IDE. Using Multi-Context Protocole to integrate tools for Code Smell analysis, test coverage, code documentation and security audit. Also performing work in a virtual environment. Transferring full control to artificial intelligence over the code writing environment.

The fourth section. Analysis and forecasting of artificial intelligence trends in coding. Using trend approximation to predict the development of artificial intelligence. Comprehensive analysis of artificial intelligence benchmarks in coding and their oversaturation. Key observations and future directions of development of this direction. Results of testing optimized artificial intelligence for writing software code.

ARTIFICIAL INTELLIGENCE, GENERATIVE ARTIFICIAL INTELLIGENCE, TRANSFORMER ARCHITECTURE, CODE GENERATION, CODE WRITING ENVIRONMENT, LARGE WORD MACHINES, CONSEQUENCES OF USING ARTIFICIAL INTELLIGENCE, OPTIMIZATION OF THE AUTOMATIC CODE WRITING PROCESS, WEB CHAT, LOCAL MACHINE, ADDITIONS TO THE CODE DEVELOPMENT ENVIRONMENT, BENCHMARK ANALYSIS, APPROXIMATION OF TRENDS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AI — штучний інтелект (ШІ).
API — інтерфейс програмування застосунків.
ASG— абстрактний семантичний граф.
AST — абстрактне синтаксичне дерево.
CI/CD — безперервна інтеграція / безперервне розгортання.
CLI —інтерфейс командного рядка.
CPU — центральний процесорний пристрій (ЦПП).
CRUD — створення, читання, оновлення, видалення даних.
FP — функціональне програмування.
GPU — графічний процесорний пристрій (ГПП).
IDE — інтегроване середовище розробки.
IP — інтелектуальна власність.
LLM — велика мовна модель (ВММ).
MCP — протокол модельного контексту.
ML — машинне навчання.
MoE — суміш експертів (архітектура багатомодульних моделей).
NLP — обробка природної мови (ОНМ).
NL — природна мова.
NPU — нейронний процесорний пристрій (НПП).
PR— запит на злиття (у системах керування версіями).
QA — забезпечення якості.
RAG — генерація, доповнена пошуком (архітектура RAG).
RAM — оперативна пам’ять (ОП).
SDK — набір засобів розробки.
SDLC — життєвий цикл розробки програмного забезпечення (ЖЦ ПЗ).
SQL — мова структурованих запитів.
TCO — загальна вартість володіння.
TF-IDF — показник важливості терміну в документі.
TPS — токени за секунду.
UI — користувацький інтерфейс.
UX — користувацький досвід.
VRAM — відеопам’ять.

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням складності програмних систем, підвищенням вимог до їхньої функціональної надійності, інформаційної безпеки та ефективності використання обчислювальних ресурсів. Паралельно зростає потреба у скороченні термінів розробки ПЗ та підвищенні продуктивності праці фахівців. У таких умовах особливого значення набувають технології автоматизації, здатні забезпечити стабільну якість та передбачуваність результатів на всіх етапах життєвого циклу програмного продукту. Актуальності набуває пошук інноваційних підходів до оптимізації процесів створення ПЗ. Одним із найбільш перспективних напрямів цієї сфери є використання генеративного штучного інтелекту (ШІ), здатного автоматизувати значну частину творчих та аналітичних завдань з написання програмного коду.

Генеративний ШІ є інноваційною галуззю штучного інтелекту, що здатна створювати унікальний контент — тексти, зображення, музику, відео, програмний код тощо. На відміну від традиційних моделей, які переважно аналізують або класифікують дані, має здатність продукувати нові, оригінальні результати, що не існували раніше. Ця технологія базується на машинному навчанні, де моделі вчаться на великих наборах даних та згодом створюють новий зміст, подібний за структурою до вихідного. Здатність таких систем до творчої генерації відкриває якісно новий етап у розвитку інформаційних технологій, перетворюючи ШІ на активного учасника процесів розробки.

Одним із найважливіших напрямів застосування генеративного ШІ є автоматизація процесу написання програмного коду. Інструменти на зразок GitHub Copilot, ChatGPT, Amazon CodeWhisperer чи Tabnine здатні не лише підказувати наступні рядки, а й створювати цілі функціональні фрагменти ПЗ на основі текстових інструкцій природною мовою. Подібні системи навчаються на великих обсягах відкритого коду, аналізуючи синтаксис, контекст та логіку програмних рішень, що дозволяє їм генерувати оптимальні або навіть інноваційні варіанти реалізації алгоритмів.

Мета роботи — дослідження сучасних підходів автоматизованого генерування програмного коду та прогнозування його подальшого розвитку. Налаштування та оптимізація роботи генеративного ШІ згідно норм та стандартів написання програмного коду.

Реалізація цієї мети передбачає ретельне вивчення того, яким чином генеративні технології впливають на традиційні підходи до програмування, як вони інтегруються у сучасні інженерні процеси та які вимоги висувають до організації середовища розробника.

Обґрунтування доцільності застосування генеративного ШІ стосовно оптимізації процесу написання програмного коду, а також визначення його впливу на ефективність розробки ПЗ. Рекомендації щодо підвищення якості, безпеки та надійності згенерованих програмних рішень.

Об’єктом дослідження є процес автоматизованого генерування програмного коду засобами генеративного ШІ як комплексного явища, що включає технологічні, алгоритмічні та організаційні аспекти.

Предметом дослідження виступають методи, моделі та інструменти, що впливають на якість, коректність, безпечність та ефективність згенерованого коду, включно з підходами взаємодії моделей із середовищем розробки, принципами обробки контексту та механізмами контролю. До них належать моделі представлення програмних структур, архітектури мовних моделей, механізми контекстного збагачення, методи самокорекції, системи виявлення структурних недоліків (Code Smell), підходи до статичного аналізу та верифікації, технології запуску коду у віртуалізованих середовищах, а також протоколи мультиконтекстної взаємодії, що забезпечують доступ моделей до зовнішніх джерел інформації.

Наукова цінність дослідження полягає у систематизації та поглибленому аналізі технологій автоматизованої генерації коду, визначенні їхнього впливу на якість ПЗ, виявленні ризиків та способів їхнього усунення, а також у формуванні методологічних засад впровадження таких систем у професійну діяльність. Отримані результати поглиблюють розуміння процесів інтеграції ШІ у життєвий цикл програмного продукту та сприяють формуванню нової концепції інтелектуально орієнтованої розробки, де творчий потенціал людини поєднується з аналітичними можливостями машинного інтелекту.

Практична цінність одержаних результатів — можливість застосування створених рекомендацій та інструментів у реальній діяльності ІТ-компаній та освітніх закладах. Використання інструментів на базі генеративних моделей, таких як GitHub Copilot, ChatGPT, CodeWhisperer чи Tabnine, дозволяє суттєво скоротити час розробки, підвищити якість та стандартизацію коду, а також зменшити кількість помилок на етапі тестування. Отримані висновки можуть бути використані з метою розробки методичних рекомендацій, щодо впровадження генеративних систем у робочі процеси команд розробників та створення навчальних програм, спрямованих на підготовку фахівців нового покоління.

Разом із перевагами використання генеративного ШІ виникають й певні проблеми, що потребують системного аналізу та вироблення методологій контролю. Серед них — етичні, правові та безпекові аспекти. Питання авторського права на згенерований код, можливість випадкового відтворення ліцензованих фрагментів або коду з вразливостями, а також ризики залежності від ШІ-асистентів є надзвичайно важливими для обговорення. Оскільки моделі навчаються на відкритих даних, існує ймовірність, що вони можуть відтворювати фрагменти, які містять помилки або небезпечні елементи. Саме тому необхідно формувати нові методологічні, етичні та правові засади безпечного використання генеративного інтелекту у професійній діяльності.

Важливим напрямом дослідження є аналіз підходів до інтеграції генеративних моделей у середовища розробки. Інструменти, що працюють у форматі веб-чатів, забезпечують швидке отримання відповідей на концептуальні питання, але мають обмежений доступ до структури проєкту. Локальні агенти дозволяють працювати з файлами безпосередньо, що забезпечує вищий рівень конфіденційності та контроль над виконанням. Плагіни щодо інтегрованих середовищ розробки перетворюють модель на повноцінного учасника інженерного процесу, здатного взаємодіяти з системами контролю версій, тестувальними фреймворками, інструментами аналізу та автоматизації. Зокрема, сучасні протоколи мультиконтекстної взаємодії дозволяють моделі отримувати доступ до документації, конфігураційних файлів, журналів виконання, тестових результатів й інших артефактів, що істотно підвищує точність та контекстну впевненість генеративних рішень.

Крім того, ключовою проблемою є також необхідність оцінки та гарантування якості згенерованого коду. Попри значні успіхи у розвитку моделей, вони здатні виробляти неефективні конструкції, дублювати фрагменти, що містять приховані помилки, або відтворювати вразливі шаблони, що властиві вихідним даним, на яких відбувалося навчання. У цьому контексті важливою складовою дослідження є методи аналізу Code Smell, що дозволяють оцінити структурні властивості коду, виявити потенційні дефекти та сформувати критерії для подальшого автоматизованого рефакторингу. Особливого значення набувають інструменти статичного аналізу безпеки, що здатні виявляти вразливості ще до того, як програма буде запущена на виконання. З метою підвищення рівня надійності важливо досліджувати й методи запуску згенерованого коду у віртуальних або контейнеризованих середовищах, що дозволяє ізолювати потенційно небезпечні фрагменти та спостерігати їхню поведінку без ризику впливу на реальну інфраструктуру.

У контексті цих змін відбувається трансформація ролі розробника. Він перестає бути суто виконавцем коду, а все більше виступає як аналітик, координатор та контролер інтелектуальних систем. Підвищуються вимоги до розуміння принципів роботи генеративних моделей, до вміння формувати коректні запити, аналізувати відповіді, забезпечувати відповідність стандартам та критеріям якості, а також до здатності використовувати інструменти перевірки та аудиту. Співпраця людини та машини, заснована на принципах синергії, формує нову парадигму інтелектуально орієнтованої розробки, що відкриває шлях до більш швидкого, безпечного й креативного створення програмних рішень. Це створює новий тип компетентностей, важливих для ефективної роботи у сучасних IT-командах.

Апробація результатів дослідження здійснювалася шляхом практичного застосування генеративних інструментів у процесі створення невеликих програмних модулів різного призначення: від веброзробки до автоматизації рутинних завдань й аналізу даних. Порівняння результатів ручного програмування з роботою ШІ-асистентів продемонструвало значне скорочення часу виконання завдань та підвищення точності коду. Основні положення

дослідження також були обговорені на науково-практичних семінарах, присвячених питанням автоматизації розробки ПЗ та використання ШІ в ІТ-галузі.

Таким чином, дослідження процесів автоматизованого генерування програмного коду за допомогою генеративного ШІ є актуальним, науково значущим та практично орієнтованим напрямом діяльності, що визначає тенденції розвитку й формує нові методологічні засади створення програмних продуктів. Результати такої роботи сприятимуть становленню ефективних, безпечних та надійних підходів до застосування ШІ у розробці ПЗ та забезпечать фундамент подальших досліджень у цій галузі. Використання генеративного ШІ є не просто технологічним трендом, а закономірним етапом еволюції сучасного програмування, який визначатиме майбутнє всієї індустрії інформаційних технологі

1 ОГЛЯД СУЧАСНИХ ПІДХОДІВ ТА ПРОБЛЕМАТИКА АВТОМАТИЗОВАНОГО ГЕНЕРУВАННЯ ПРОГРАМНОГО КОДУ ЗАСОБАМИ ШТУЧНОГО ІНТЕЛЕКТУ

1.1 Генерація коду генеративним штучним інтелектом

Генерація коду генеративним ШІ – це процес автоматичного створення комп'ютерного коду за допомогою алгоритмів штучного інтелекту, що навчений на великих обсягах існуючого вихідного коду. Алгоритми закладені в основі великих мовних моделей (LLM) та обробки природної мови (NLP), використовують методи глибокого навчання та великі нейронні мережі, що аналізують різноманітні набори даних коду з відкритих репозиторіїв. У контексті генерації коду, генеративний ШІ автоматизує процес написання коду, інтерпретуючи запити природною мовою та використовуючи знання, отримані у результаті навчання на великих наборах даних коду, щодо розуміння шаблонів програмування, стилів та логіки[1].

На відміну від традиційних методів автоматизації коду, що досить часто включають статичний аналіз та попередньо визначені шаблони, генеративний ШІ є більш динамічним та адаптивним до різних контекстів.

Відрізняється він також від low-code та no-code платформ тим, що не використовує попередньо створені шаблони та бібліотеки компонентів, а генерує код з нуля на основі запитів розробника природною мовою. У той час як low-code та no-code - інструменти орієнтовані переважно на бізнескористувачів, програмне забезпечення генерації коду ШІ є більш універсальним(рис. 1.2) та може використовуватися як професійними розробниками й іншими користувачами також.

Основні принципи генерації коду генеративним ШІ включають використання алгоритмів штучного інтелекту щодо автоматичної генерації коду на основі певних вхідних даних, вимог або умов. Ці принципи зазвичай включають розуміння вимог користувача, використання великої бази знань стосовно існуючого коду, застосування генеративних моделей, оптимізацію згенерованого коду та можливість зворотнього зв'язку з метою подальшого вдосконалення. Генеративні моделі використовують ймовірності для створення фрагментів коду, які, найімовірніше, будуть правильними та ефективними, навчаючись на великих наборах даних коду щодо розпізнавання шаблонів та структур.

Залежність від великих наборів даних коду щодо навчання підкреслює як потужність цієї технології, так й її потенційні упередження, оскільки якість та різноманітність навчальних даних безпосередньо впливають на якість та характеристики згенерованого коду.

Багатошаровий характер генеративних моделей означає, як один й той самий запит може давати різні результати, що може бути як перевагою, так і недоліком, вимагаючи ретельної оцінки згенерованого коду.

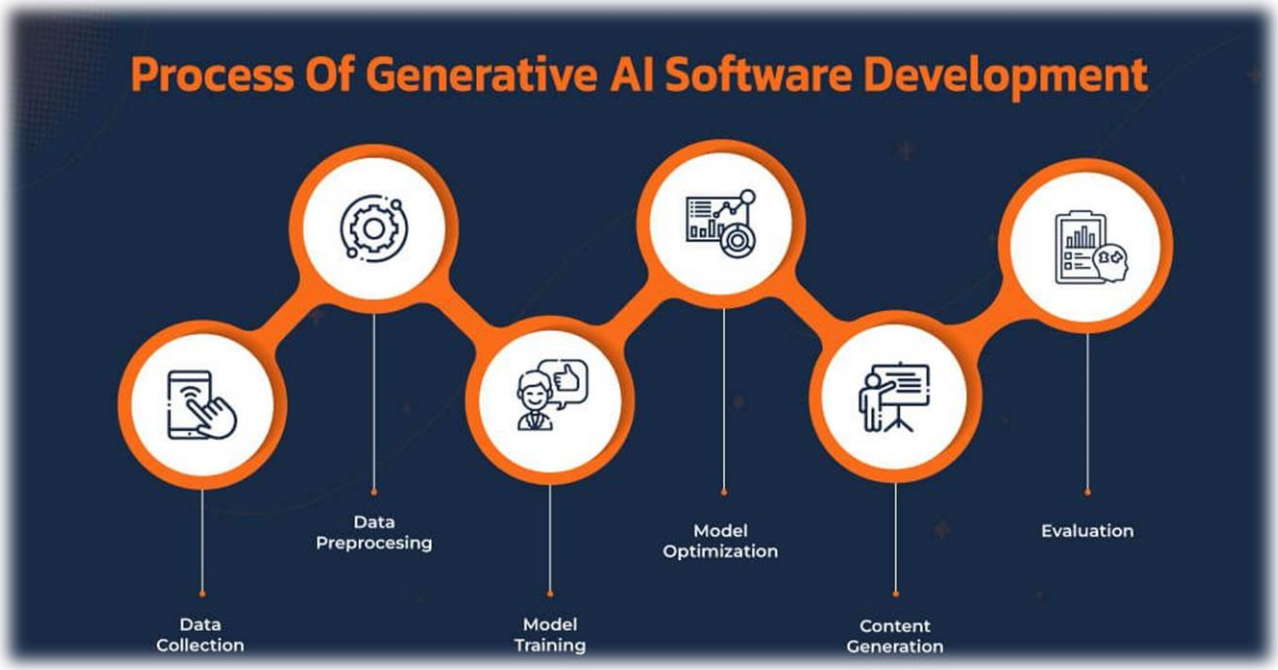

Рисунок 1.1 – Генеративний ШІ в розробці коду

Таким чином, генерація коду генеративним ШІ(рис. 1.1) стала невід'ємною частиною сучасного програмування. Його швидкий розвиток може замінити людську працю, що викликає значні соціальні занепокоєння. Вплив генеративного ШІ вже неможливо ігнорувати, а потребує ретельного вивчення та дослідження.

1.2 Принципи роботи генерації коду

В основі генеративного ШІ стосовно генерації коду лежить використання великих мовних моделей (LLM), що навчаються на масивних наборах даних коду. Ці моделі використовують алгоритми глибокого навчання та великі нейронні мережі, аналізують мільярди рядків коду з відкритих репозиторіїв, щоб засвоїти синтаксис, семантику та шаблони програмування різних мов. LLM будують складні внутрішні представлення коду, що дозволяє їм розуміти значення, структуру та взаємозв'язки між різними елементами коду.

Багато сучасних моделей генеративного ШІ, включаючи ті, що використовуються для генерації коду, використовують архітектуру Transformer. Transformer – це архітектура глибокого навчання, що відрізняється від попередніх рекурентних нейронних мереж (RNN) відсутністю рекурентних блоків та значно скорочує час навчання. Ключовими компонентами архітектури Transformer є вбудовування вхідних даних, позиційне кодування, механізм багатоголової самостійної уваги та прямі нейронні мережі.

До появи цієї архітектури попередні нейронні моделі обробки природної мови (NLP) були неефективними, оскільки обробляли вхідні дані послідовно. Такий повторюваний процес не міг повноцінно використовувати сучасні графічні процесори (GPU), розроблені для паралельних обчислень, що значно уповільнювало навчання.

Архітектура Transformer вирішила цю проблему за допомогою використання механізму уваги (Attention Mechanism). Цей механізм є критичним, оскільки він дозволяє ефективно моделювати довгострокові залежності у послідовних даних. Механізм самостійної уваги особливо добре підходить для розуміння довгих послідовностей коду, оскільки дозволяє моделі одночасно враховувати різні частини вхідних даних, встановлюючи зв'язки між віддаленими елементами коду. У контексті програмування це означає, що LLM можуть простежувати взаємозв’язки між віддаленими елементами коду, що є необхідним у розумінні великих кодових баз, залежностей між функціями та змінними. Оскільки якість згенерованого коду безпосередньо зумовлюється розумінням цього широкого контексту, ефективність механізму уваги та розмір контекстного вікна LLM стають прямими індикаторами якості кінцевого програмного продукту. Якщо контекстне вікно занадто коротке, модель не може охопити важливі компоненти системи, що неминуче призводить до логічних помилок та потенційних вразливостей[2].

Процес генерації коду починається з того, що програмісти вводять текстові запити природною мовою, описуючи бажану функціональність коду. Інструменти генеративного ШІ аналізують ці запити, розбиваючи їх на складові частини щоб зрозуміти наміри користувача, вибір відповідної мови програмування та врахування будь-яких заданих обмежень. Потім LLM використовує свої внутрішні представлення, щоб знайти відповідні шаблони та структури коду, які відповідають вимогам запиту. Після цього модель комбінує знайдені фрагменти коду та модифікує їх відповідно контексту запиту, генеруючи остаточний код, часто пропонуючи кілька варіантів або рекомендацій користувачу.

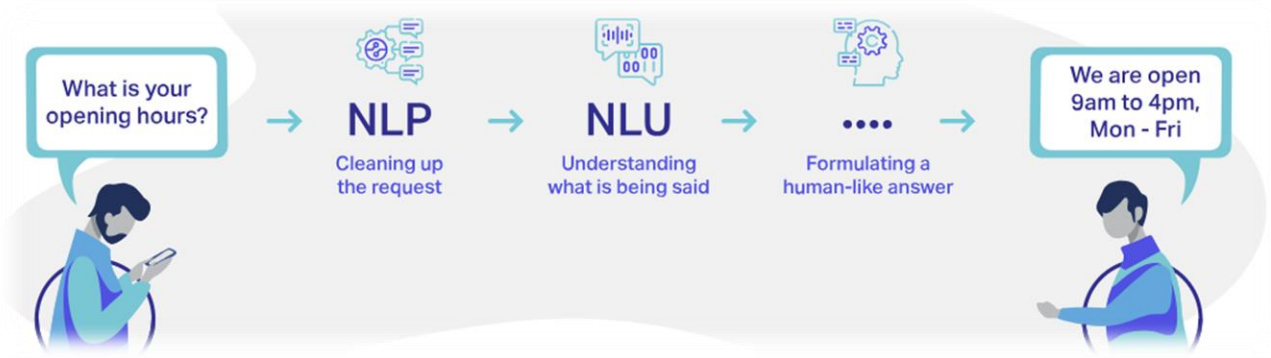

Рисунок 1.2 – Використання природної мови ШІ

Існує кілька методів введення запитів щодо генерації коду. Одним з найпоширеніших є використання природної мови(рис. 1.2), що дозволяє розробникам інтуїтивно описувати потрібну функціональність. Інший метод

передбачає введення часткового коду, коли розробник починає писати код, а ШІ намагається автоматично завершити його, пропонуючи найбільш ймовірні продовження. Також можлива взаємодія з ШІ через коментарі у коді, коли розробник пише коментар природною мовою, а ШІ генерує відповідний код. Крім того, деякі інструменти дозволяють розробникам безпосередньо спілкуватися зі ШІ у чат-інтерфейсі, задаючи конкретні питання або прохання щодо написання чи виправлення коду. Гнучкість методів введення дозволяє розробникам взаємодіяти з інструментами генерації коду ШІ найбільш зручним для них способом, залежно від конкретного завдання.

Ця трансформація дозволяє розробникам генерувати робочі функції, просто описуючи їх призначення природною мовою (Natural Language, NL). LLM навчені на величезних обсягах даних, що охоплюють як пропрієтарний, так і відкритий вихідний код, дозволяючи їм розпізнавати складні архітектурні патерни та надавати точні контекстні пропозиції. Це передбачає багатомодальну здатність, тобто моделі можуть розуміти та працювати з програмним забезпеченням цілісно, включаючи документацію, наміри розробника та структуру проєкту, а не лише на рівні окремих рядків коду.

Здатність LLM читати та інтерпретувати код ґрунтується на двох ключових механізмах. По-перше, це багатомодальна здатність, яка дозволяє ШІ працювати з програмним забезпеченням цілісно, розуміючи не лише синтаксис, а й намір, логіку, функціональні взаємозв’язки між компонентами та архітектурний контекст застосунку. Завдяки цьому великі мовні моделі здатні аналізувати як фрагменти коду, так й текстову документацію, коментарі, технічні специфікації чи навіть опис вимог користувача. Такий підхід наближає їх до рівня розуміння проєкту, а не просто механічного аналізу. Як наслідок, LLM можуть навіть виявляти невідповідності між логікою коду та вимогами до продукту, оптимізувати структуру функцій або пропонувати ефективніші алгоритмічні рішення.

По-друге, це контекстуальне навчання (In-Context Learning), за яким LLM вдосконалюють свої знання та адаптуються до конкретних вимог проєкту без необхідності повного перенавчання. На відміну від традиційних моделей, що потребують додаткового тренування на нових даних, сучасні генеративні ШІ здатні динамічно враховувати контекст поточного середовища розробки. Вони аналізують підказки користувачів, структуру існуючої кодової бази, назви змінних, шаблони функцій, а також коментарі, щоб зрозуміти стиль програмування конкретного розробника чи команди. Це дозволяє LLM підтримувати єдині стандарти коду, генерувати доповнення, які природно інтегруються у вже написаний код, та навіть передбачати наступні дії програміста.

Крім того, контекстуальне навчання забезпечує високий рівень гнучкості: модель може адаптуватися до нових мов програмування, фреймворків або бібліотек, з якими вона раніше не працювала, лише на основі аналізу доступного коду. Такий механізм суттєво підвищує ефективність спільної роботи в командних проєктах, де LLM може слугувати інтелектуальним посередником між

різними розробниками, що забезпечує узгодженість стилю, структури та функціональності програмного продукту.

Таким чином, поєднання багатомодальної інтерпретації та контекстуального навчання робить великі мовні моделі не просто інструментами автодоповнення, а повноцінними асистентами у програмуванні, здатними аналізувати, оптимізувати й адаптувати код із урахуванням усіх особливостей конкретного проєкту.

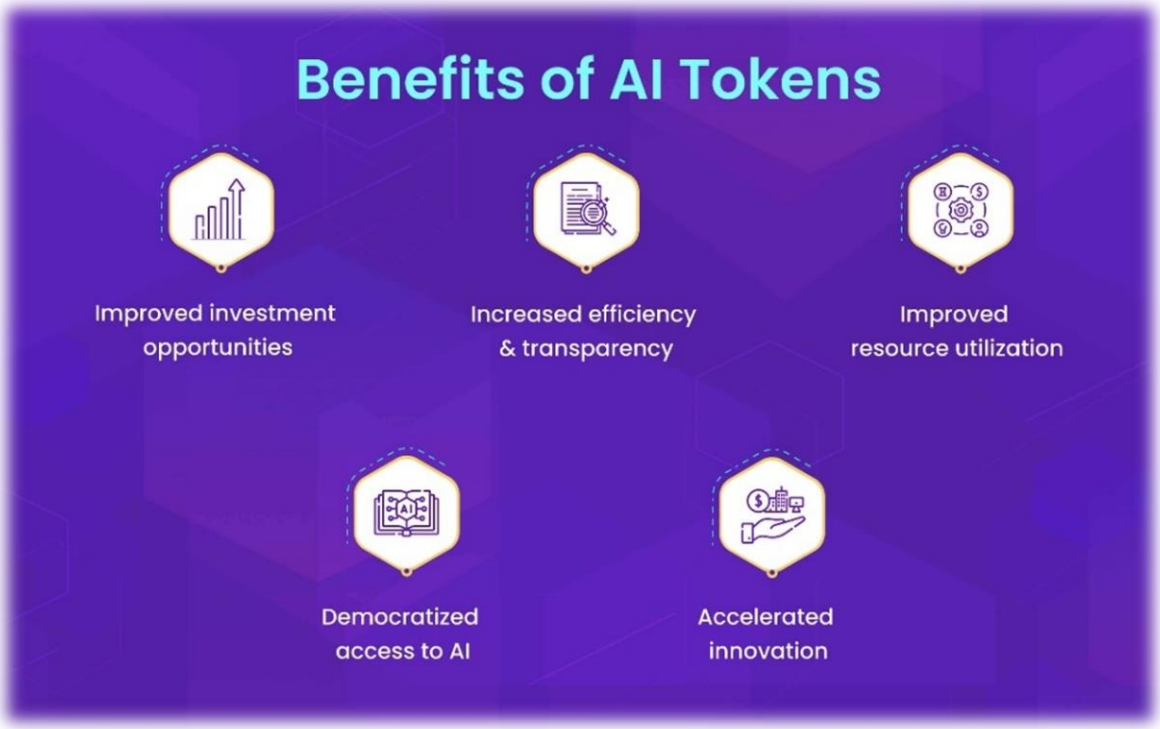

Рисунок 1.3 – Плюси використання токенів

З метою обробки коду LLM спочатку розбиває запит від користувача на токени(рис. 1.3) – базові одиниці, з яких формується вхідна послідовність для моделі. Кожен токен може представляти символ, ключове слово, оператор або навіть частину змінної. Такий підхід дозволяє ШІ працювати з кодом як з текстом, зберігаючи при цьому логічну структуру.

Однак традиційна токенізація має низку суттєвих обмежень. Вона часто не враховує синтаксичну структуру мови програмування, через що окремі конструкції можуть бути розбиті на синтаксично невалідні частини. Це призводить до спотворення семантики коду, ускладнює розпізнавання меж між функціями, класами або логічними блоками та знижує точність подальшої генерації. Наприклад, якщо токенізатор розбиває багаторядковий вираз або складну умовну конструкцію, модель може втратити контекст, що впливає на правильність інтерпретації логіки програми.

Попри це, навіть найсучасніші архітектури стикаються з обмеженнями, пов’язаними з розміром контекстного вікна. Хоча архітектура Transformer з механізмом self-attention дійсно ефективно працює з довгостроковими залежностями, межа контексту залишається критичною проблемою при роботі з

великими корпоративними або монолітними кодовими базами. Якщо важливі фрагменти коду, що містять ключові залежності або визначення функцій, знаходяться за межами контекстного вікна, модель не може пов’язати їх між собою, що призводить до зниження логічної узгодженості та якості згенерованого коду.

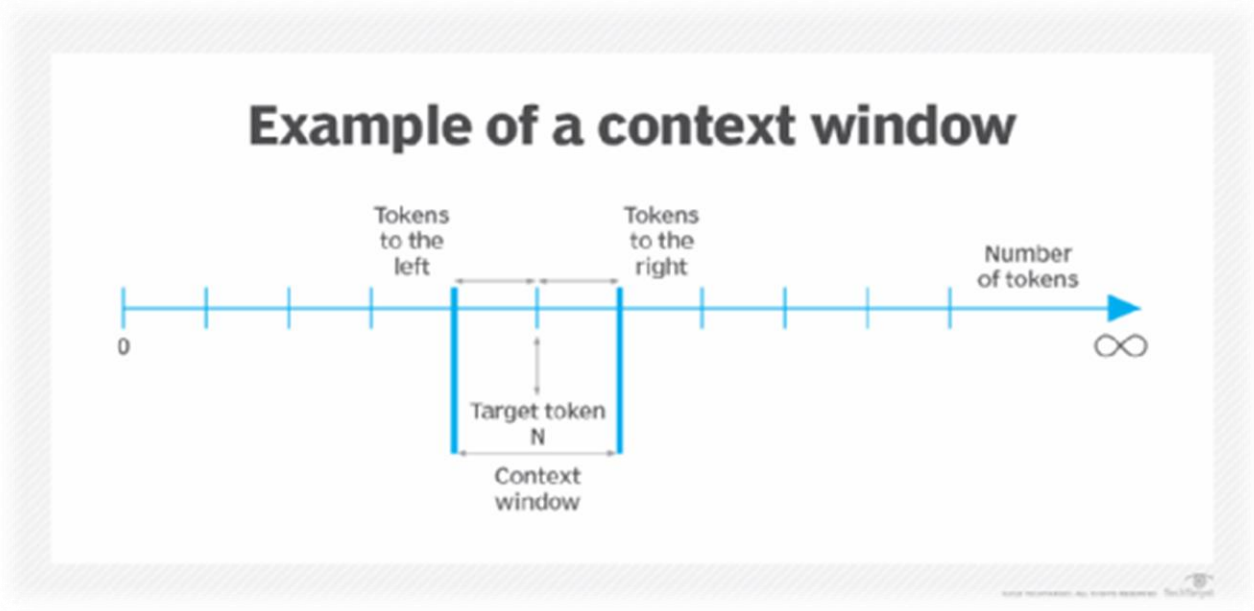

Рисунок 1.4 – Context window

Для вирішення цієї проблеми активно розробляються нові підходи, такі як розширені контекстні вікна(рис. 1.4) (long-context transformers), ієрархічне кодування контексту або пам’ять із зовнішнім сховищем (external memory), що дозволяють моделі згадувати попередні частини коду навіть після виходу за межі поточного контексту. Іншим перспективним напрямом є гібридні моделі, які поєднують лінгвістичне представлення токенів із графовим аналізом залежностей у коді.

Таким чином, розвиток механізмів токенізації та розширення контекстних можливостей LLM є ключовими напрямами вдосконалення генеративного ШІ у програмуванні, оскільки саме вони визначають, наскільки глибоко модель здатна розуміти, інтерпретувати та створювати складні програмні системи.

1.2.1 Abstract Syntax Tree

Сучасні дослідження намагаються подолати проблему формування токенів шляхом створення структурно-орієнтованої токенізації, що враховує граматичні правила мов програмування та використовує абстрактні синтаксичні дерева (AST, Abstract Syntax Tree). Такий підхід дозволяє моделі бачити не лише послідовність токенів, а й ієрархічні зв’язки між ними — тобто розуміти, які токени утворюють функцію, клас чи логічний блок. Це забезпечує більш глибоке семантичне розуміння коду й покращує здатність моделі до його точного відтворення, модифікації чи оптимізації.

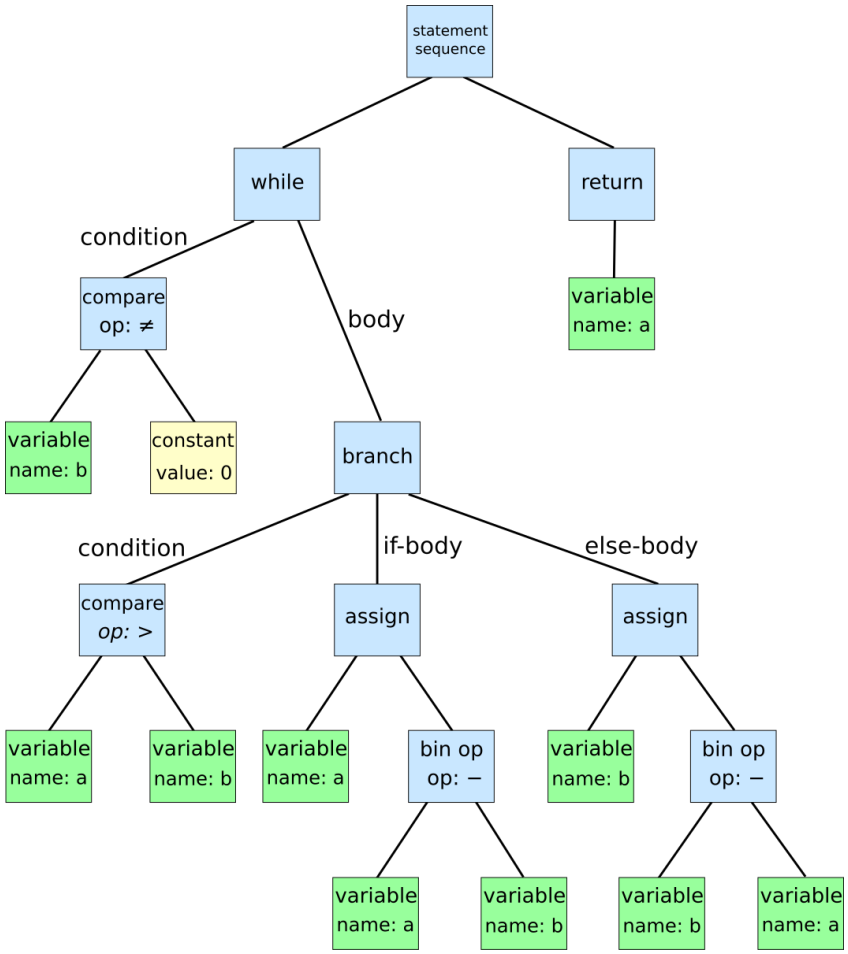

Рисунок 1.5 – Abstract Syntax Tree

AST(рис. 1.5) — це деревоподібна структура, що представляє синтаксичну структуру вихідного коду, забезпечуючи значно краще семантичне розуміння. Перевага AST-Chunking полягає в тому, що він діле код на значущі структурні межі, такі як: визначення функцій, класи або керуючі структури. Це гарантує, що кожен фрагмент коду, залишається синтаксично дійсним та логічно цілісним. Завдяки цьому LLM отримує набагато кращу контекстуальну цілісність, що різко покращує якість генерації[3].

1.2.2 Retrieval-Augmented Generation

Моделі RAG (Retrieval-Augmented Generation) інтегрують зовнішню базу знань у процес генерації, що є критично важливим для забезпечення високої якості та релевантності коду, особливо в контексті специфічних проєктів або внутрішніх API.

На відміну від традиційних моделей, що покладаються виключно на дані, отримані під час тренування, RAG-системи динамічно звертаються до зовнішніх джерел інформації — наприклад, до корпоративної документації, баз знань або архівів коду. Це дозволяє уникнути проблеми застарілих знань та забезпечує адаптивність до змін у проєктному середовищі.

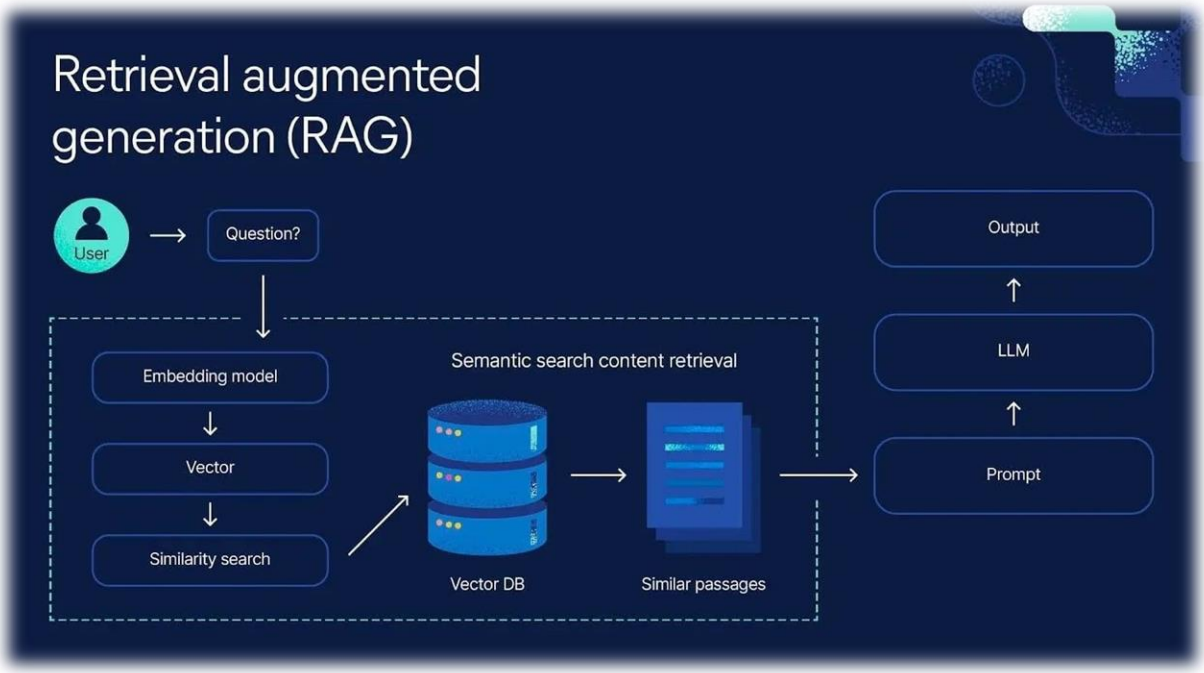

Рисунок 1.6 – Retrieval-Augmented Generation

Завдяки такому підходу, RAG(рис. 1.6) виступає своєрідним мостом між статичною пам'яттю моделі та актуальною інформацією ззовні. У розробницьких середовищах це означає, що система може враховувати найновіші оновлення бібліотек, специфікації API або внутрішні політики компанії, не потребуючи повного перенавчання. Крім того, RAG-підхід суттєво підвищує ефективність командної роботи, адже всі розробники можуть отримувати відповіді, узгоджені з єдиним джерелом правди — внутрішнім репозиторієм знань[4].

Механізм RAG:

1. Векторизація. Запит розробника перетворюється на векторне представлення (embedding), що дозволяє системі оцінювати семантичну подібність між запитом і наявними документами;
2. Пошук. Векторна база даних виконує пошук подібності для отримання найбільш релевантних фрагментів коду, внутрішньої документації або API-посилань, які відповідають контексту;
3. Інтеграція. Знайдена інформація інтегрується у вихідний промпт LLM, створюючи розширений контекст. Це дозволяє моделі генерувати більш точні, інформативні та, що важливо, сумісні з корпоративними стандартами відповіді.

Отже, RAG-моделі є важливим засобом комплаєнсу та безпеки, оскільки вони можуть бути налаштовані для доступу лише до затверджених, ліцензійно чистих або внутрішніх кодових баз, мінімізуючи ризики порушення прав інтелектуальної власності. Таким чином, поєднання генеративних можливостей LLM із механізмами вибіркового пошуку робить RAG не просто технологічним інструментом, а стратегічним компонентом сучасної інфраструктури штучного інтелекту в організаціях.

1.2.3 Self-Correction

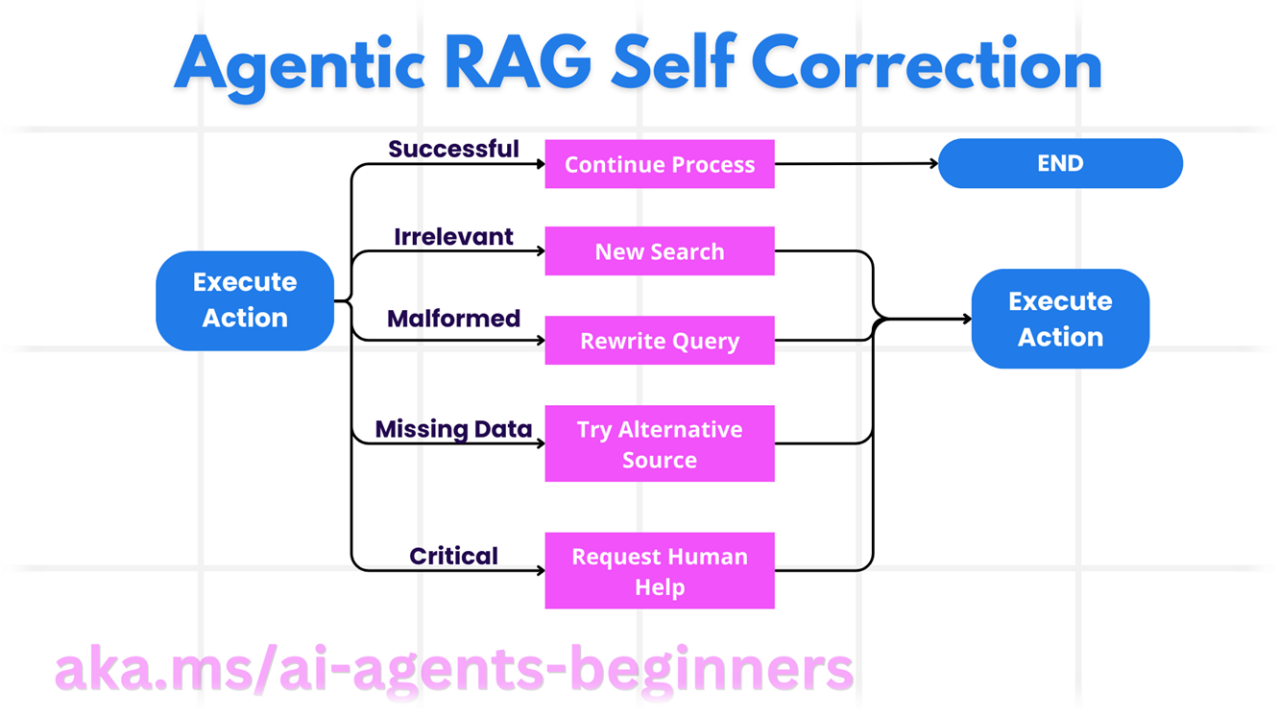

Рисунок 1.7 – Self-Correction

Здатність ШІ-систем до самокорекції (Self-Correction)(рис. 1.7) є однією з ключових оптимізацій, що дозволяє значно підвищити логічну коректність, стабільність та читабельність згенерованого коду. Цей процес ґрунтується на багаторівневому механізмі, за якого система не просто створює вихідний код, а аналізує власні результати, порівнює їх із логічними шаблонами, закладеними у навчальних даних, та вносить уточнення, якщо виявляє невідповідності чи помилки. Таким чином, модель не лише повторює знання, отримані під час навчання, а й активно використовує внутрішню логіку для перевірки власних висновків, що наближає її до поведінки людського програміста.

Механізм самокорекції зазвичай реалізується через декілька ітерацій генерації, коли система створює код, перевіряє його на наявність помилок, порівнює із заданими вимогами, а потім уточнює результат. Цей підхід особливо ефективний у поєднанні з методами reinforcement learning with human feedback (RLHF), що дозволяють моделі навчатися на основі оцінки правильності та якості власних відповідей. Таким чином, ШІ може не лише відтворювати відомі шаблони, але й удосконалювати свої результати у реальному часі, формуючи адаптивну логічну поведінку[5].

Особливо ефективним напрямом є використання моделей дифузії (Diffusion LLM) для генерації токенів, що передбачає ітеративне уточнення. На відміну від звичайних моделей, що формують текст або код за один прохід, дифузійні моделі поступово покращують результат, перетворюючи шум на структурований, логічно зв’язний код. На кожному кроці модель уточнює токени, спочатку позначені як невизначені, — процес подібний до того, як художник поетапно деталізує ескіз.

Процес включає стохастичний семплінг, тобто використання таких технік, як Softmax і шум Гамбеля (Gumbel noise), що дозволяє моделі уникати надмірної впевненості та враховувати кілька можливих варіантів продовження коду. Таким чином, це забезпечує баланс між стабільністю та креативністю, дозволяючи ШІ обирати найбільш вірогідні та логічно послідовні токени на кожному етапі генерації.

1.2.4 Low-Confidence

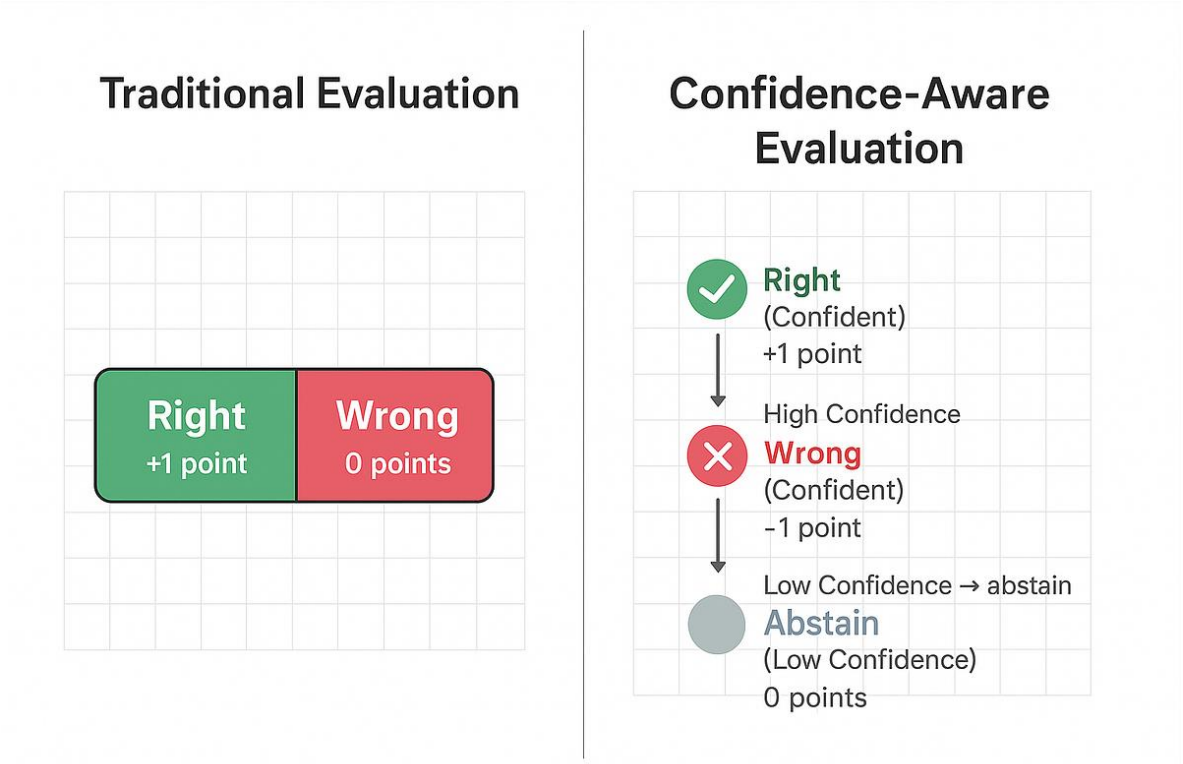

Рисунок 1.8 – Low-Confidence

Важливою стратегією є Low-Confidence Re-masking(рис. 1.8) — коли модель повторно маскує лише ті токени щодо яких має низьку впевненість та оновлює їх у наступній ітерації. Такий підхід значно знижує обчислювальні витрати, адже система не генерує код заново, а коригує лише потенційно помилкові або неоднозначні ділянки. Результатом є підвищення логічної узгодженості, семантичної точності та читабельності коду, що особливо важливо у промислових застосуваннях, де якість і стабільність мають критичне значення.

Таким чином, самокорекція та дифузійне уточнення перетворюють великі мовні моделі на самоадаптивні системи, здатні не лише генерувати код, але й розуміти, аналізувати та вдосконалювати власні результати. Це наближає генеративний ШІ до нового етапу розвитку — коли він стає не просто інструментом підтримки розробника, а повноцінним учасником процесу програмування, що самостійно забезпечує якість і узгодженість створеного продукту.

1.2.5 Mixture of Experts

Для виконання складних аналітичних завдань та обробки величезних контекстів, необхідних для програмування на високому рівні, сучасні генеративні системи використовують передові архітектури, серед яких ключову роль відіграє Mixture-of-Experts (MoE)(рис. 1.9). Ця архітектура стала проривом у галузі масштабованого штучного інтелекту, оскільки дозволяє поєднати високу потужність обчислень із ефективним використанням ресурсів.

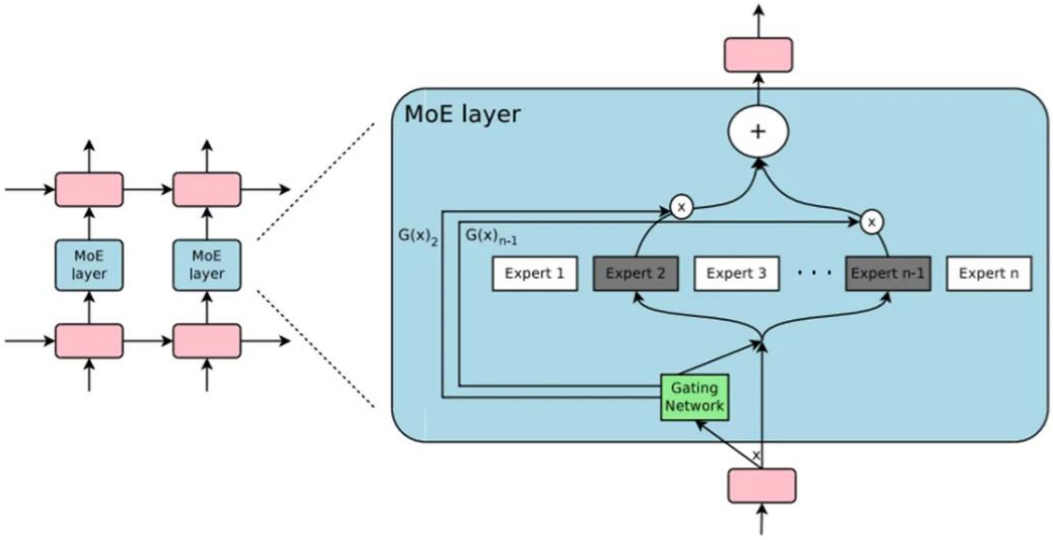

Рисунок 1.9 – Mixture of Experts

Моделі нового покоління, такі як DeepSeek-R1 та Qwen3-235B-A22B, демонструють видатні здібності до міркування (reasoning), тобто вміння логічно аналізувати, робити висновки й обґрунтовувати рішення. Завдяки архітектурі MoE вони здатні не лише розпізнавати закономірності у даних, а й вибірково активувати ті підмодулі, що спеціалізуються на певному типі завдань — наприклад, математичних обчисленнях, обробці коду чи генерації тексту.

Сутність підходу Mixture-of-Experts полягає в тому, що замість одного великого монолітного блоку нейронних параметрів модель має велику кількість експертів — спеціалізованих підмереж, кожна з яких навчена вирішувати конкретний клас задач. Під час обробки запиту активується лише обмежена підмножина експертів, обрана за допомогою роутера (router), що визначає, які з модулів найкраще підходять стосовно поточного контексту. Це забезпечує експоненційне зростання обчислювальної потужності моделі без пропорційного збільшення витрат на інференс[6].

Такий підхід має низку переваг щодо застосування у реальному часі, зокрема в інтерактивних інструментах типу ШІ Copilot, що повинні швидко аналізувати код та генерувати рекомендації без затримок. Використання MoE дозволяє моделі зберігати високу продуктивність, працюючи з великими кодовими базами, документами або складними логічними сценаріями. Завдяки цьому навіть за

наявності сотень мільярдів параметрів модель може виконувати обчислення з мінімальними затримками, що робить її придатною для інтерактивних середовищ розробки (IDE).

Крім того, архітектура MoE має адаптивний характер: у процесі навчання різні експерти спеціалізуються на певних патернах даних, утворюючи щось на кшталт «експертної екосистеми», де кожен модуль накопичує глибокі знання у своїй ніші. Така спеціалізація не лише підвищує точність і швидкість генерації, а й робить систему більш стійкою до помилок і здатною до паралельного міркування.

Отже, використання архітектури Mixture-of-Experts у сучасних LLM є одним із ключових чинників, що визначає їх здатність масштабуватися, ефективно міркувати та забезпечувати високу якість результатів навіть щодо роботи з величезними обсягами даних. Саме завдяки MoE великі мовні моделі можуть поєднувати потужність глибоких нейронних мереж із оперативністю, необхідною для інтерактивного програмування, відкриваючи новий рівень продуктивності у сфері генеративного кодування.

Вплив методів оптимізації розглянуто у додатку А.

1.3 Проблематика використання штучного інтелекту

1.3.1 Коректність та якість згенерованого коду

Одним із значних викликів сучасного генеративного програмування є забезпечення якості, надійності та безпеки коду, створеного великими мовними моделями (LLM). Хоча ШІ-асистенти демонструють високу швидкість та продуктивність у створенні програмних рішень, вони не завжди гарантують їхню правильність й відповідність стандартам безпеки.

Ранні дослідження, зокрема, під час етапу тестування Copilot, показали, що до половини всього згенерованого коду містило логічні або безпекові помилки. Такі результати пояснюються тим, що моделі навчаються на великих масивах відкритого коду, що сам по собі не є ідеальним— він містить вразливості, антипатерни, дублювання коду та потенційно небезпечні практики. Відповідно, модель успадковує і ці недоліки, що може призводити до інкорпорації небезпечних або неоптимальних рішень у промислові продукти.

Більшість публічних репозиторіїв, таких як GitHub чи GitLab, містять код різного рівня якості — від початкових експериментів до стабільних бібліотек. Однак без ретельного відбору ШІ не здатен відрізнити надійні джерела від неякісних, тому в процесі генерації може використовувати небезпечні бібліотеки, застарілі методи або коди з ліцензійними обмеженнями.

Це створює ризики комерційного використання:

- юридичні ризики — порушення авторських прав або умов ліцензії;
- технічні ризики — включення вразливих або неефективних алгоритмів;
- репутаційні ризики — використання чужого коду без перевірки його безпечності.

Для вирішення цієї проблеми провідні компанії активно впроваджують концепцію «Clean Code Corpus» — спеціалізованих, керованих навчальних баз

даних, що проходять багаторівневу фільтрацію за критеріями якості, безпеки та стилістичної відповідності. Такі корпуси створюються на основі ретельно перевірених фрагментів коду, де видаляються помилки, шкідливі патерни та небезпечні бібліотеки.

Amazon CodeWhisperer — одна з перших систем, що інтегрувала навчання не лише на відкритих даних, а й на внутрішньому корпоративному коді Amazon, що пройшов аудит безпеки. Це дозволило суттєво зменшити кількість помилок та підвищити точність пропозицій.

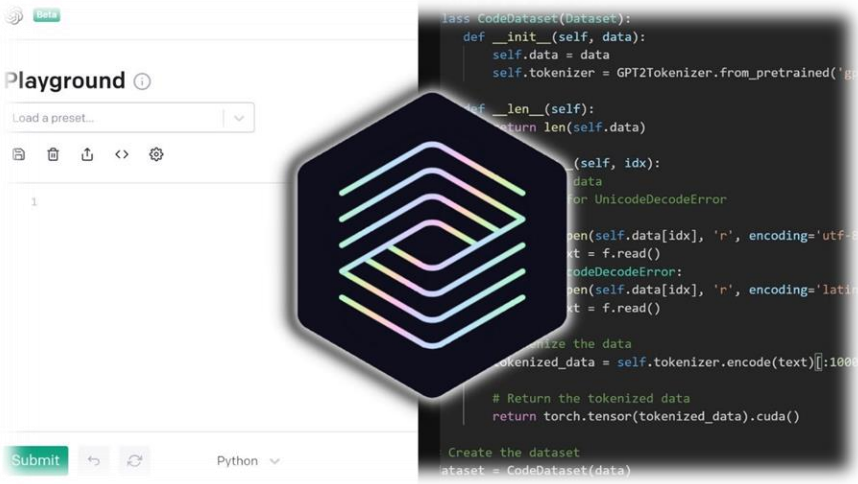

Рисунок 1.10 – OpenAI Codex

OpenAI Codex(рис. 1.10) та Google Gemini Code також експериментують із гібридними навчальними стратегіями, де моделі комбінують дані з відкритих джерел і внутрішніх верифікованих баз.

У дослідницькому середовищі набирають популярності LLM-аудитори коду, що перевіряють результат генерації на відповідність стандартам безпеки (наприклад, OWASP, CERT).

Подолання викликів якості згенерованого коду вимагає:

- розробки методів автоматичної валідації коду, що дозволяють моделі перевіряти власний результат до його видачі користувачу;
- впровадження навчання з підкріпленням (Reinforcement Learning), де моделі отримують нагороди за коректний, безпечний і оптимізований код;
- створення систем етичного фільтрування, що запобігають використанню шкідливих або неліцензованих фрагментів;
- розвитку спеціалізованих моделей для окремих доменів (наприклад, кібербезпеки, фінансових систем чи IoT).

Таким чином, проблема якості згенерованого коду є не лише технічним, а й стратегічним викликом у розвитку генеративного штучного інтелекту. Шляхом поєднання чистих навчальних корпусів, етичного контролю та самокорекційних механізмів можливо досягти рівня, коли ШІ стане повноцінним і надійним партнером у професійному програмуванні.

1.3.2 Безпекові та конфіденційні ризики

Використання хмарних ШІ-асистентів у розробці ПЗ створює низку суттєвих загроз щодо інформаційної безпеки та конфіденційності даних. Хоча такі інструменти, як GitHub Copilot, ChatGPT або CodeWhisperer, значно підвищують ефективність роботи розробників, вони водночас можуть стати потенційними джерелами витоку критично важливої інформації.

Основна небезпека полягає у тому, що ШІ-асистенти мають доступ до контексту розробки, включно з фрагментами коду, коментарями, файлами конфігурацій та навіть внутрішніми документаціями проєкту. У багатьох випадках цей контекст може містити захищений авторським правом або корпоративний код, який обробляється через зовнішні сервери постачальника ШІ.

Це створює ризик того, що такі дані можуть:

- бути збережені у реєстрах або тимчасових кешах для подальшого аналізу розробниками моделі;
- потрапити до навчальних наборів майбутніх оновлень моделей;
- передаватися стороннім сервісам або субпідрядникам, що збільшує ризик несанкціонованого доступу.

Особливо небезпечно це для компаній, що працюють із закритим або чутливим кодом, наприклад, у фінансовій, оборонній чи медичній галузях. Навіть, якщо передача даних зашифрована, сам факт виходу коду за межі внутрішньої мережі порушує політику безпеки більшості корпоративних середовищ.

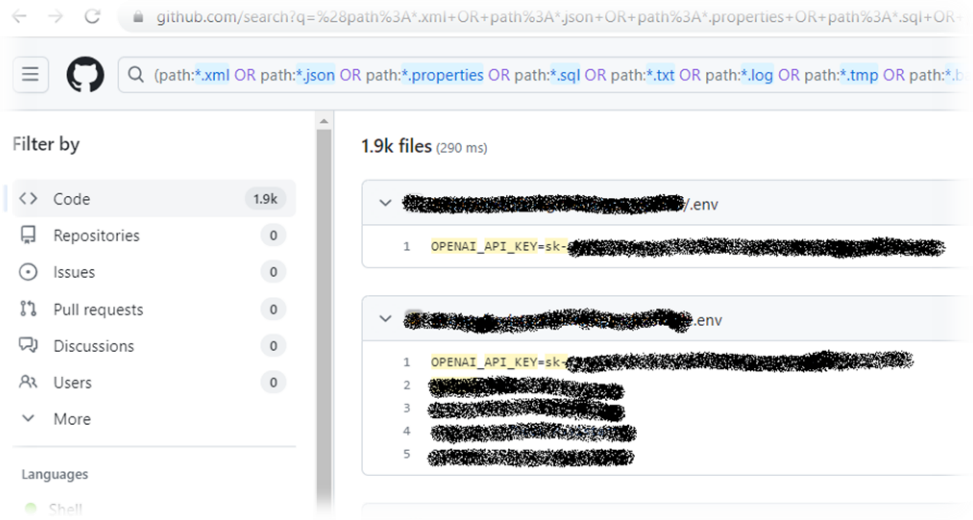

Рисунок 1.11 – Приклад розкриття ключу доступу у GitHub

Розкриття конфіденційних даних – це один із критичних аспектів необережного використання генеративного ШІ. LLM можуть запам’ятовувати та відтворювати фрагменти інформації, що містять:

- паролі, API-ключі, токени доступу(рис. 1.11);
- адреси внутрішніх серверів, шляхи до баз даних або конфігураційних файлів;

- дані користувачів, які підпадають під дію законів про захист персональної інформації (GDPR, CCPA тощо).

У разі, якщо розробник випадково включає такі дані до запиту ШІ-асистенту, вони можуть бути збережені на сторонніх серверах, а згодом, навіть, використані як навчальні приклади. Такі інциденти вже траплялися, наприклад, коли працівники великих компаній ненавмисно передали конфіденційний код системам ChatGPT.

Рисунок 1.12 – Приклад zero-trust

З метою зниження загроз компаніям, що працюють із чутливими даними, рекомендується:

- використовувати локальні або приватні розгортання LLM — наприклад, через on-premise рішення (Meta LLaMA, Mistral, Falcon, DeepSeek Coder). Такі моделі можуть бути інтегровані у внутрішнє середовище розробки без передачі даних за межі корпоративної мережі;
- впроваджувати політику “zero-trust”[7](рис. 1.12) — обмежувати доступ ШІ до коду, що дозволяє аналіз лише окремих фрагментів, а не всієї кодової бази;
- використовувати інструменти фільтрації даних — автоматичне приховування або заміна конфіденційних токенів, ключів і паролів перед передачею запиту до моделі;
- здійснювати аудит і моніторинг запитів до ШІ, щоб запобігти ненавмисному витоку критичної інформації;
- проводити навчання персоналу, щоб розробники розуміли ризики, пов’язані з використанням хмарних ШІ-сервісів, та дотримувалися внутрішніх правил безпеки.

Отже, незважаючи на значні переваги у швидкості та продуктивності, використання хмарних ШІ-асистентів у програмуванні повинно супроводжуватися суворими заходами кібербезпеки. Компаніям, що працюють із закритими або конфіденційними даними, найдоцільніше впроваджувати локальні або гібридні моделі, що забезпечують повний контроль над даними та мінімізацію ризику витоку. Саме баланс між інноваціями та безпекою визначатиме ефективність інтеграції генеративного ШІ у корпоративне програмування найближчими роками.

1.3.3 Регулярні аудити безпеки та комплаєнсу коду

Зважаючи на високу ймовірність того, що великі мовні моделі (LLM) можуть успадковувати вразливості з навчальних даних, проведення регулярних аудитів безпеки та комплаєнсу є критично важливою складовою сучасного процесу розробки. Навіть якщо генеративний ШІ створює функціонально правильний код, існує ризик, що він містить приховані дефекти, що можуть призвести до витоку даних, ескалації привілеїв чи відмов у роботі системи. На 2025 рік лише 55% коду від ШІ є безпечним(рис. 1.13)[8].

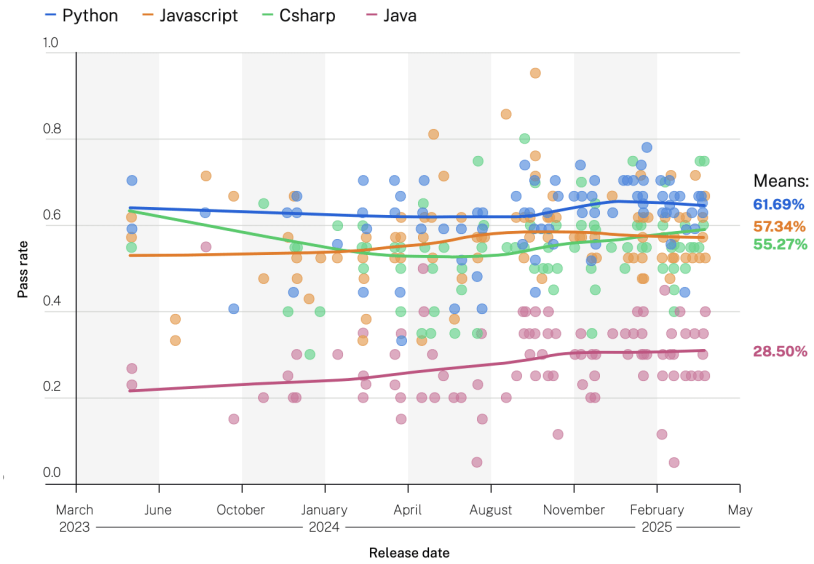

Рисунок 1.13 – 2025 GenAI Code Security Report

Постійний аудит є ключовим механізмом контролю якості згенерованого коду. Він дозволяє виявляти не лише синтаксичні або логічні помилки, а й вразливості безпеки — наприклад, SQL-ін’єкції, XSS, небезпечне керування пам’яттю, несанкціонований доступ до API чи витік конфіденційних даних.

Такі аудити мають проводитися відразу після генерації коду, ще до інтеграції у спільну кодову базу, щоб запобігти розповсюдженню потенційних проблем у більш масштабні модулі проєкту.

Для забезпечення об’єктивності аудиту рекомендується використовувати сторонні (неінтегровані у LLM-помічник) засоби перевірки безпеки, щоб уникнути ризику упередженості чи “сліпих зон” самої моделі. До таких інструментів належать:

- Snyk, SonarQube, Checkmarx, Veracode — автоматизовані системи аналізу коду на наявність вразливостей та недоліків стилю;
- OWASP Dependency-Check — перевірка бібліотек на відомі вразливості (CVEs);
- Bandit, Semgrep — інструменти для статичного аналізу безпеки в Python, JavaScript, C/C++ та інших мовах;
- FOSSA, Black Duck — системи контролю ліцензійної чистоти (комплаєнсу) для запобігання порушенням авторських прав.

Важливо, щоб ці системи були окремо розгорнуті у внутрішньому середовищі компанії й не передавали код стороннім серверам для перевірки, особливо якщо йдеться про закриті корпоративні продукти. Для досягнення максимальної ефективності перевірок аудити безпеки мають бути автоматизованими та інтегрованими у CI/CD-пайплайн (Continuous Integration / Continuous Deployment)(рис. 1.14). Це дозволяє здійснювати аналіз кожного нового коміту або пул-реквесту в режимі реального часу, що забезпечує постійний моніторинг якості[9].

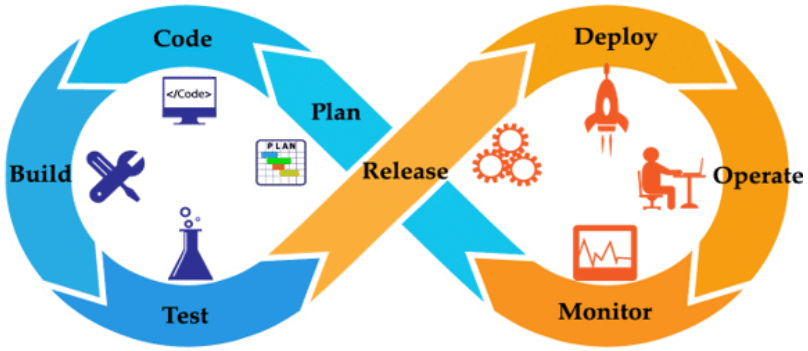

Рисунок 1.14 – Continuous Integration / Continuous Deployment

Такий підхід гарантує, що навіть код, згенерований ШІ, проходить ті самі етапи контролю, що й код, написаний вручну. Зокрема, перевірку відповідності політикам безпеки, корпоративним стандартам та вимогам до IP-комплаєнсу.
Конкурентна перевага через прозорий комплаєнс

У майбутньому стратегічну перевагу на B2B-ринку отримають ті компанії та інструменти, що зможуть гарантувати юридичну та безпекову чистоту згенерованого коду. Вже сьогодні деякі сервіси, наприклад, Amazon CodeWhisperer, запроваджують механізми фільтрації коду, схожого на навчальні дані, щоб запобігти копіюванню ліцензованих або потенційно небезпечних фрагментів.

Таким чином, регулярний аудит безпеки та комплаєнсу є не лише технічною процедурою, а й елементом довіри до генеративного ШІ у розробці програмного забезпечення. Він забезпечує прозорість, правову захищеність та стійкість програмного продукту, що є вирішальним чинником для успішної комерціалізації ШІ-рішень у корпоративному середовищі.

1.3.4 Інтелектуальна власність та ліцензування

Генерація програмного коду за допомогою ШІ створює серйозні юридичні та комплаєнс-ризики, що стосуються авторського права, ліцензування та використання інтелектуальної власності. Основна складність полягає у тому, що великі мовні моделі (LLM) навчаються на величезних обсягах відкритого коду, що часто містить фрагменти, захищені різними типами ліцензій — від цілком вільних до обмежувальних.

Під час навчання моделі отримують доступ до мільярдів рядків коду з відкритих репозиторіїв (наприклад, GitHub, Stack Overflow, SourceForge), частина яких може бути розповсюджена під умовами ліцензій, що вимагають зазначення авторства (MIT, BSD) або навіть забороняють комерційне використання (GPL, AGPL). Якщо модель, не усвідомлюючи цього, генерує фрагмент, схожий або ідентичний оригінальному, це може розцінюватися як порушення авторських прав або крадіжка інтелектуальної власності.

Вже сьогодні низка судових справ у США та ЄС стосується саме цього аспекту — відтворення навчальними моделями коду, що підлягає правовому захисту. Розробники Copilot, OpenШІ Codex та інших асистентів стикаються зі звинуваченнями у тому, що їхні системи відтворюють фрагменти з GPL-проєктів без збереження авторських посилань. Така ситуація ставить під сумнів юридичну чистоту ШІ-згенерованого коду, особливо у комерційних продуктах.

Юридичну картину ускладнюють нестандартні умови ліцензування відкритих моделей, створених великими корпораціями, такими як Meta (Llama 3) чи Google (Gemma 3). Формально вони позиціонуються як open-source, однак на практиці містять додаткові обмеження, що суперечать духу відкритого ПЗ. Наприклад, Meta забороняє використовувати результати роботи моделей Llama 3 для навчання або покращення інших моделей, крім тих, що належать до сімейства Llama. Це означає, що навіть якщо розробник має повний доступ до коду, він не може вільно використовувати отримані результати у сторонніх проєктах. Подібні обмеження виходять за межі стандартних відкритих ліцензій (Apache 2.0, MIT), що створює правову невизначеність та ризики для компаній, що планують інтеграцію таких моделей у власні продукти. Через це багато експертів пропонують класифікувати такі моделі не як «Open Source», а як «Source-Available», тобто моделі з відкритим кодом, доступним лише для ознайомлення або обмеженого використання, але не для повної свободи розповсюдження чи модифікації.

Наявність таких юридичних обмежень змушує компанії:

- ретельно перевіряти походження згенерованого коду (через спеціальні інструменти типу Black Duck або FOSSA);
- зберігати логи генерації, щоб у разі перевірки можна було довести відсутність порушення авторських прав;
- інвестувати у власні навчальні моделі або приватні корпуси даних, що забезпечують контроль за джерелами коду;
- відмовлятися від моделей з умовно відкритими ліцензіями, що створюють правові пастки під час інтеграції у комерційні рішення.

Перспективи правового врегулювання.

Світова юридична спільнота лише починає формувати регуляторну базу для генеративного ШІ. Європейський AI Act вже містить положення, що зобов’язують розробників великих моделей розкривати походження навчальних даних та забезпечувати дотримання авторських прав. Подібні ініціативи розглядаються у США, Японії та Південній Кореї[10].

Однак повна правова визначеність поки що відсутня, тому кожна компанія, що використовує генеративні інструменти у розробці ПЗ, має самостійно розробляти політику комплаєнсу, спрямовану на уникнення ризиків порушення ліцензій та захист інтелектуальної власності.

Отже, юридичні аспекти генеративного програмування стають не менш важливими, ніж технічні. У майбутньому правова прозорість й етична відповідальність стануть ключовими критеріями довіри до моделей ШІ, а здатність компанії забезпечити дотримання IP-прав — одним із головних факторів конкурентоспроможності на ринку.

1.3.5 Етичні та соціальні наслідки

Використання генеративного ШІ для генерації коду має значні етичні та соціальні наслідки, що потребують уважного розгляду. Одним з ключових питань є відповідальність за згенерований код. Оскільки ШІ може створювати неточний, оманливий або навіть шкідливий код, виникає питання про те, хто несе відповідальність за наслідки його використання. Користувачі повинні усвідомлювати обмеження та ризики цих інструментів, ретельно перевіряти згенерований код та застосовувати власне судження перед його використанням.

Вплив генеративного ШІ на ринок праці розробників програмного забезпечення є ще одним важливим аспектом. Автоматизація рутинних завдань кодування може призвести до зменшення попиту на джунів, а організації можуть перейти до більш раціоналізованих структур, зосереджуючись на моніторингу та оптимізації результатів ШІ. Однак дешевша розробка програмного забезпечення може також відкрити нові можливості та призвести до збільшення кількості програмістів, які зможуть швидше створювати більше програмного забезпечення.

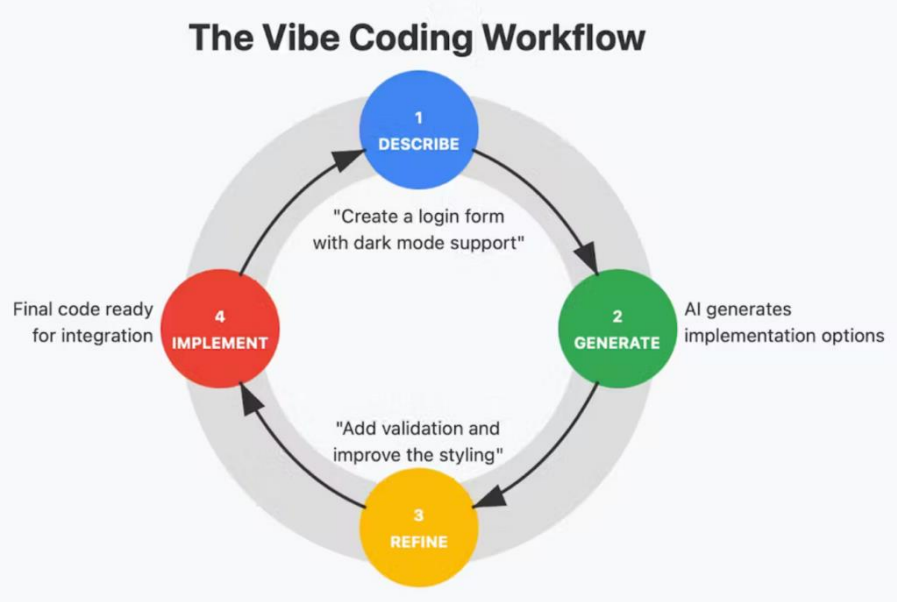

Рисунок 1.15 – Vibe Coding цикл

Окрему увагу привертає поява підходу Vibe Coding(рис. 1.15), коли програміст взаємодіє з ШІ у режимі спільної творчості, формуючи код не через

чітке технічне завдання, а через опис бажаного результату, стилю або настрою рішення. Такий підхід змінює фундаментальне розуміння ролі розробника: він поступово перетворюється з виконавця алгоритмів на куратора, редактора та критика згенерованих рішень. Попри підвищення швидкості та легкості створення програмних продуктів, Vibe Coding може розмивати межу між авторством та інструментом, що загострює питання інтелектуальної власності та відповідальності за помилки. Також існує ризик, що тривале використання цього підходу зменшуватиме рівень глибоких технічних знань розробників, адже логіка створення коду стає прихованою за діями моделі[11].

Нарешті, існує нагальна потреба у розробці етичних норм та регуляцій, що стосуються використання генеративного ШІ для генерації коду. Це включає розгляд питань упередженості навчальних даних, ризиків зловживання технологією, забезпечення прозорості та підзвітності систем ШІ. Розробка чітких етичних принципів та правових рамок є необхідною для забезпечення відповідального та безпечного використання генеративного ШІ у розробці програмного забезпечення.

Висновки

Генерація коду за допомогою генеративного ШІ стала ключовим елементом сучасної розробки ПЗ, поєднуючи швидкість, ефективність й адаптивність. В основі процесу застосовані великі мовні моделі (LLM), що навчаються на великих масивах коду та використовують методи глибокого навчання та NLP. Суттєве підвищення якості генерації забезпечує структурна токенізація та підхід AST-Chunking, що ділить код на логічні блоки. Моделі RAG додатково підключають зовнішні бази знань, підвищуючи релевантність висновків стосовно внутрішніх API та специфічних проєктів. Механізми самокорекції та Low-Confidence Re-masking покращують послідовність та точність, що робить ШІ активним учасником процесу розробки.

Сучасні генеративні системи все частіше використовують архітектуру Mixture-of-Experts, що забезпечує масштабованість, ефективність і високу якість обчислень навіть на великих обсягах даних. Подальший розвиток генеративного програмування пов’язаний із розширенням контекстних можливостей, гібридними RAG-підходами та підвищенням продуктивності за рахунок MoE.

Проблема якості згенерованого коду залишається критичною. Дослідження, зокрема GitHub Copilot Beta, показали високий відсоток помилок, тому компанії переходять до «Clean Code Corpus» — спеціальних очищених навчальних наборів. Amazon CodeWhisperer, OpenШІ Codex та Gemini Code поєднують відкриті дані з внутрішнім перевіреним кодом, що зменшує кількість дефектів. Зростає роль LLM-аудиторів та Diffusion-моделей, що ітеративно покращують логіку й структуру коду. Автоматизовані QA-системи та генерація тестів стають необхідним етапом перевірки.

Безпека — окрема стратегічна проблема. За звітом «2025 GenШІ Code Security Report», лише близько половини згенерованого коду є безпечним. Тому аудит має бути вбудованим у CI/CD, щоб контролювати кожен коміт і пул-реквест. Хмарні ШІ-асистенти вимагають суворих заходів кіберзахисту, а

компаніям із чутливими даними доцільно використовувати локальні або гібридні моделі.

Генеративний ШІ створює й юридичні ризики, пов’язані з авторським правом та ліцензіями. Європейський ШІ Act уже зобов’язує розкривати джерела навчальних даних та гарантувати дотримання авторського права, а подібні норми готуються й в інших країнах. Правова прозорість й етична відповідальність стають критеріями довіри до моделей ШІ. Необхідні також стандарти, що регулюють упередженість даних, ризики зловживання та забезпечення прозорості.

У підсумку, генеративний ШІ уже суттєво впливає на програмування. Його зростаючі можливості викликають соціальні побоювання щодо заміни людської праці, проте повне ігнорування цієї технології неможливе: вона потребує системного дослідження, регулювання й відповідального впровадження.

2 ПОРІВНЯЛЬНИЙ АНАЛІЗ ВИДІВ ТА ІНТСРУМЕНТІВ ШТУЧНОГО ІНТЕЛЕКТУ У СФЕРІ ГЕНЕРАЦІЇ КОДУ

2.1 Основні види використання штучного інтелекту у написанні коду

Сьогодні існує кілька основних видів використання ШІ у написанні коду, що умовно можна поділити на три способи звернення(рис. 2.1):

- веб-чати на базі великих мовних моделей (LLM) – браузерні інтерфейси (ChatGPT, Claude, Gemini тощо), що дозволяють ставити запитання природною мовою і отримувати згенерований код або пояснення від моделі;
- консольні або локальні агенти – автономні системи, що працюють безпосередньо на комп'ютері розробника або у локальній мережі, використовуючи локально розгорнуті LLM-моделі (наприклад, LLaMA, Mistral, CodeGemma);
- додатки до IDE (інтегрованих середовищ розробки) – плагіни та розширення (GitHub Copilot, Tabnine, Cursor тощо), що вбудовуються у середовище програмування та надають підказки чи автодоповнення коду у реальному часі.

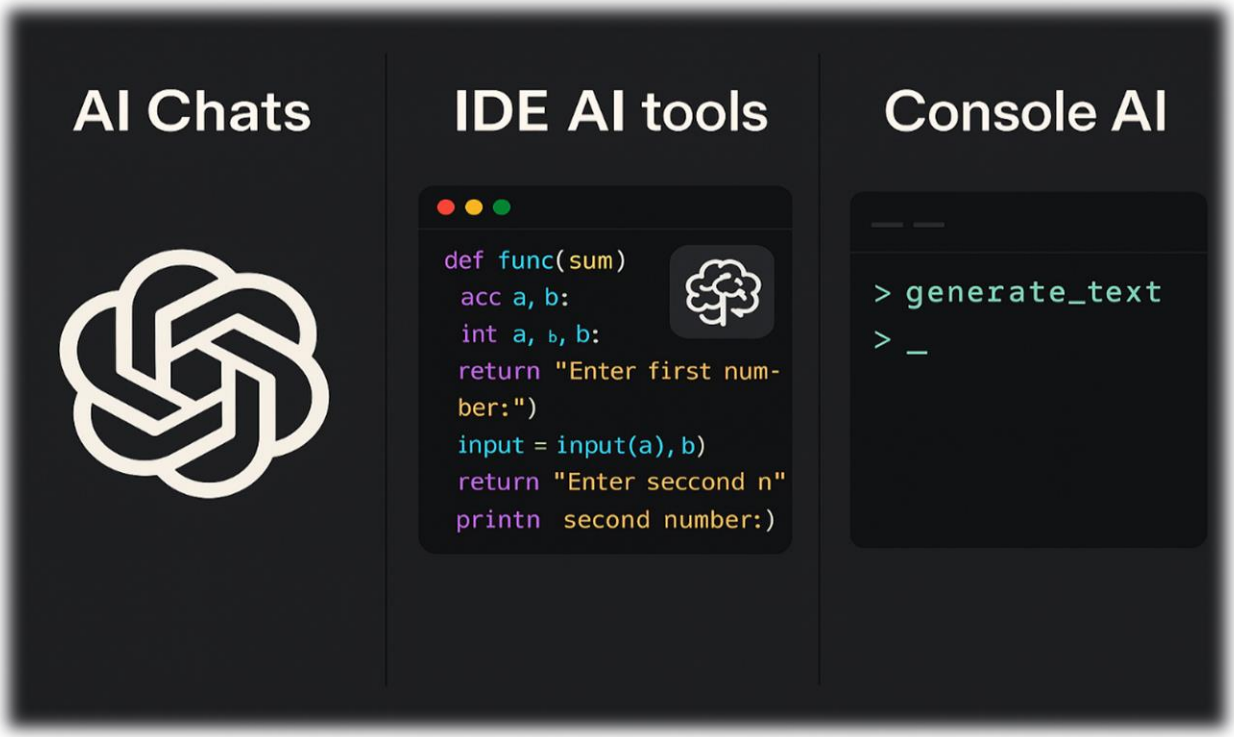

Рисунок 2.1 – Види звернення до ШІ

2.1.1 Веб-чати для генерації коду

2.1.1.1 Принцип роботи

Веб-чати працюють з ШІ на основі великих мовних моделей, що розміщені у хмарі. Користувач взаємодіє з такими системами через веб-інтерфейс: ставить запит природною мовою, а модель генерує у відповідь текст: фрагмент коду чи пояснення. Ключовим є механізм In-Context Learning – навчання моделі у реальному часі з врахуванням контексту запиту. Тобто, модель не оновлює свої ваги, а розуміє нове завдання на основі наданого їй текстового контексту (опис задачі, приклади вводу-виводу тощо). Завдяки цьому підходу LLM здатні розв'язувати нові завдання без додаткового навчання, просто аналізуючи патерни в підказці. На практиці це означає, що розробник може описати потрібну функціональність або помістити приклади коду у чат, а модель на основі свого велетенського тренувального корпусу згенерує відповідний код.

Важливо, що такі моделі навчені на мільярдах рядків коду і тексту, тому мають знання з різних мов програмування і алгоритмів. Сучасні хмарні LLM можуть опрацьовувати дуже великі обсяги контексту – інколи, сотні тисяч токенів одночасно. Наприклад, модель Claude від Anthropic у версії Sonnet 4 підтримує контекст до 1 мільйона токенів, що дає змогу проаналізувати цілу кодову базу проєкту в одному запиті. Моделі навчаються розуміти навіть складні інструкції і підтримувати мультимодальний ввід: окрім звичайного тексту-коду, деякі системи можуть приймати файли, зображення чи схеми як частину запиту. Зокрема, ChatGPT(рис. 2.2) з функцією Advanced Data Analysis дозволяє завантажувати у чат файли з кодом, таблицями, зображеннями і навіть аудіо чи відео – модель здатна обробляти різні формати та аналізувати їх у межах діалогу. Це відкриває можливість, скажімо, надіслати скриншот із помилкою або фрагмент діаграми, а ШІ-асистент проаналізує та надасть відповідь з урахуванням цього контексту.

ChatGPT (OpenAI)– найвідоміший представник веб-чатів. Він може підтримувати діалогову контекстну взаємодію, писати код різними мовами, генерувати тестові випадки, пояснювати помилки. У незалежних тестах відзначають його універсальність та «людське» розуміння інструкцій.

Claude (Anthropic) – інший популярний чат-асистент, відомий своїм великим контекстним вікном і глибиною аналізу. Його остання версія Claude Sonnet 4 здатна врахувати вхідні дані обсягом до 1 млн токенів, тобто фактично цілу кодову базу чи десятки документів за раз. Це відкриває нові можливості для аналітики коду – модель може знайти залежності між модулями, виявити дублювання або запропонувати рефакторинг з огляду на весь проєкт цілком[12].

Gemini (Google) – родина новітніх моделей від Google, що інтегровані в екосистему Google Workspace. Відмінністю Gemini є глибока інтеграція з робочими інструментами: модель вбудована безпосередньо в GmШІl, Google Docs, Sheets, Slides тощо, що працює як внутрішній асистент без потреби перемикатися між вкладками. Це дозволяє, наприклад, згенерувати код або формули в Google Sheets, посилаючись на дані з вашого корпоративного диска за повного дотримання прав доступу. Geminiтакож є мультимодальною моделлю.

Окрім тексту, вона здатна сприймати аудіо, відео і навіть програмний код як вхідні дані. Різні версії Gemini налаштовані щодо різних потреб (Nano стосовно мобільних пристроїв, Pro - широкого спектру завдань, Ultra - найскладніших). Повідомляється про контекст до 1 млн токенів у професійній версії.

Таким чином, веб-базовані ШІ-асистенти є чудовим інструментом швидких прототипів та навчання коду, проте їх застосування у корпоративній розробці потребує зважування ризиків безпеки та інтеграційних незручностей. У багатьох випадках компанії шукають альтернативи, що дають змогу зберегти контроль над кодом.

2.1.1.2 Переваги та недоліки веб-чатів

Доступність та простота використання. Веб-інтерфейси не вимагають налаштування чи встановлення. Потрібен лише браузер та інтернет. Це означає, що розробник може звернутися до ШІ з будь-якого пристрою: ноутбука, планшета чи смартфона. Інструменти, на кшталт ChatGPT, набули широкої популярності саме завдяки низькому порогу входження та можливості швидко отримати відповідь.

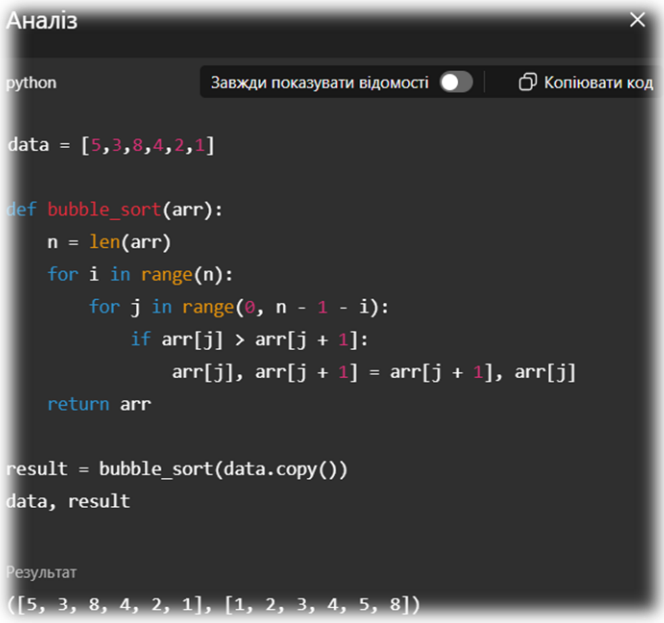

Рисунок 2.2 – ChatGPT Python-інтерпретатор

Потужність хмарних ресурсів. Моделі, розгорнуті на серверних дата-центрах, можуть бути надзвичайно великими (десятки мільярдів параметрів) та працювати на потужних GPU. Це забезпечує високу точність генерації та швидкість відповіді. Користувач фактично орендує суперкомп’ютер через інтернет для розв’язання своєї задачі. Як наслідок, сучасні системи здатні генерувати складний і синтаксично правильний код на популярних мовах (Python, C++, JavaScript тощо) й навіть виконувати його у безпечному середовищі. Наприклад, ChatGPT наразі може не лише писати код, а й виконувати його у вбудованому Python-інтерпретаторі(рис. 2.2) для перевірки результатів (функція

Advanced Data Analysis). Це значно розширює спектр завдань – від написання функцій до аналізу даних чи побудови графіків у режимі діалогу.

Широкий контекст та знання. Висока ємність контекстного вікна у деяких моделей (сотні тисяч токенів) дає можливість надсилати великі фрагменти коду, цілі файли або документацію в один запит. Модель може врахувати всі ці дані перед генерацією відповіді. Для порівняння, якщо ранні версії GPT-3 могли опрацювати лише ~4 тис. токенів контексту, то Claude 2 вже підтримував 100 тис., а новітній Claude Sonnet 4 – до 1 млн токенів. Це кардинально підвищує корисність ШІ при роботі з великими проєктами: можна дати моделі всю кодову базу чи пакети залежностей, а вона спробує знайти рішення з урахуванням всіх цих деталей.

Рисунок 2.3 – Мультимодальність веб-чатів

Мультимодальність. Як зазначалося, деякі веб-чати дозволяють працювати не лише з текстом. Підтримується завантаження різних типів даних(рис. 2.3): від фрагментів коду до таблиць CSV, PDF-документів чи зображень. Це корисно, коли завдання виходить за рамки чистого коду – наприклад, розібрати лог-файл, проаналізувати графік продуктивності чи прочитати текст з картинки (OCR). ChatGPT з функцією бачення GPT-4 може аналізувати вміст зображення та відповідати на запити щодо нього.

Спільне використання. Веб-базовані інструменти легко розшарити між колегами. Відповіді моделі можна швидко скопіювати або дати спільний доступ до чата. Деякі платформи (наприклад, Google Gemini в Google Workspace) орієнтовані на колаборативну розробку – вони вбудовані у середовище командної роботи (документи, пошту), що спрощує обмін ідеями у команді.

Попри привабливі можливості, підхід веб-чатів має низку обмежень та ризиків:

Залежність від інтернету та зовнішніх серверів. Без підключення до мережі такі інструменти не працюють зовсім. Якщо у розробника немає доступу до інтернету (наприклад, він працює з конфіденційним кодом в ізольованому середовищі) – використати хмарний LLM не вийде. Навіть при наявності мережі можуть бути затримки стосовно мережевого обміну даними. Більше того, компанія повинна довіряти зовнішньому постачальнику ШІ: весь код та запити

пересилаються у хмару для обробки. Це фундаментальна відмінність від локальних рішень. Розробник фактично передає свій код щоразу, коли просить пораду у чаті.

Ризик витоку або компрометації даних. Надсилаючи фрагменти корпоративного коду чи опис проєкту на сервери третіх осіб, існує загроза порушення конфіденційності. Дані можуть бути збережені провайдером певний час, потрапити в лог-файли або стати об’єктом атаки. Наприклад, GitHub Copilot (який теж працює через хмару OpenШІ) зберігає всі запити користувачів до 30 днів у своїй системі з метою моніторингу та безпеки. Якщо такі запити містять приватний код, це вже потенційне порушення внутрішніх політик безпеки компанії. Зафіксовані випадки, коли працівники випадково завантажували у ChatGPT конфіденційні дані, що призводило до їхнього витоку у публічний доступ через помилки або зловживання системою. Аналітики наголошують, що несанкціоноване розкриття вихідного коду чи бізнес-логіки може призвести до втрати конкурентних переваг, юридичних проблем (порушення GDPR тощо) та репутаційних збитків. Тому багато організацій обмежують або зовсім забороняють використання публічних ШІ-чатів розробниками у роботі з закритим кодом.

Обмежена інтеграція з локальними інструментами. Веб-інтерфейс, за своєю сутністю, ізольований від оточення розробника. Модель не має прямого доступу ні до файлової системи, ні до середовища виконання коду, ні до систем контролю версій (як Git), якщо тільки користувач сам не надасть фрагменти або дані. Це означає, що відповіді від ШІ завжди доводиться інтегрувати вручну: копіювати з браузера код та вставляти у свій проєкт, завантажувати файли через форми, слідкувати за версіями самостійно. Натомість вбудовані в IDE рішення можуть автоматично підставляти код або створювати файли. Тож у сценаріях реальної розробки постійне перемикання між вікном браузера та редактором коду знижує продуктивність.

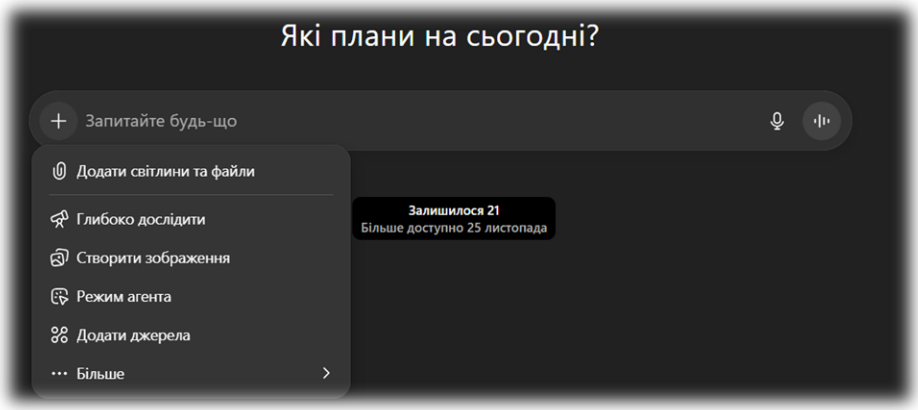

Рисунок 2.4 – Обмеження за кількістю використання ChatGPT

Можливі обмеження за використанням(рис. 2.4). Багато веб-сервісів ШІ є комерційними продуктами із своїми тарифами та ліцензійними обмеженнями.

Безкоштовні версії можуть мати зменшені можливості (менші моделі, повільніша черга відповіді тощо) або взагалі забороняти обробку великого обсягу коду. Платні підписки (як ChatGPT Plus/Enterprise, Claude Pro тощо) збільшують фінансові витрати компанії. Окрім того, публічні моделі мають політики модерації контенту, які можуть блокувати певні запити. Наприклад, якщо код стосується чутливої тематики (безпека, шкідливий софт), модель може відмовитися його генерувати, посилаючись на правила використання.

Відсутність гарантій правильності та потреба у валідації. Наразі, навіть найкращі LLM можуть робити помилки, що виглядає правдоподібно, але не працює. Відповіді моделі треба ретельно перевіряти. Веб-чати не інтегровані з тестами проєкту чи CI/CD. Тож усе, що згенеровано, розробник має прогнати локально та виправити вручну. Цей додатковий крок неминучий, оскільки ШІ не несе відповідальності за помилки. Більше того, моделі не мають розуміння проєкту на рівні архітектури. Вони оперують лише на основі статистичних зв'язків у тексті. Тому запропоновані рішення можуть бути неоптимальними чи порушувати прийняті у команді стандарти стилю. Все це вимагає від розробника часу на рев'ю та налагодження, що частково нівелює вигоду від швидкої генерації коду.

2.1.2 Консольні та локальні агенти

2.1.2.1 Принцип роботи

Локальні ШІ-агенти – це системи, що працюють офлайн або у внутрішній мережі компанії, не відправляючи дані у зовнішні хмари. В основі таких рішень лежать великі моделі, розгорнуті безпосередньо на обладнанні користувача або на корпоративних серверах. З появою відкритих LLM (наприклад, LLaMA(рис. 2.5) від Meta, Mistral 7B, CodeGemma від Google DeepMind, DeepSeek та ін.) стало можливим запускати мовні моделі локально, за достатніх обчислювальних ресурсів. Консольні агенти, як правило, взаємодіють з користувачем через командний рядок або простий графічний інтерфейс, приймаючи текстові запити та генеруючи відповіді подібно до чатів. Однак, на відміну від веб-сервісів, весь процес відбувається на локальній машині.

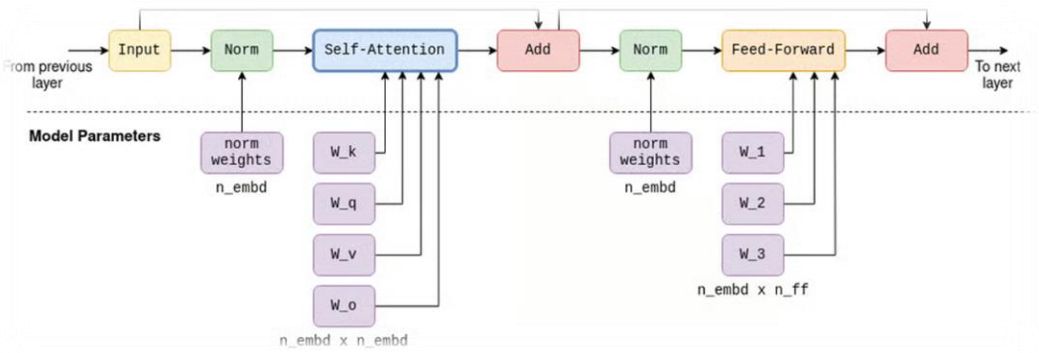

Рисунок 2.5 – LLaMA

Моделі можна завантажити у вигляді файлів (найчастіше, оптимізованих стосовно локального виконання, як GGUF, SAFETensors тощо) та запускати за допомогою спеціального ПЗ – фреймворків на кшталт LLaMA.cpp, text-generation-webui, або готових оболонок-агентів. При цьому часто застосовуються методи зменшення ресурсомісткості: квантизація (quantization) зі зниження розрядності ваг (наприклад, 4-бітові або 8-бітові моделі) та завантаження лише необхідних експертів (Mixture-of-Experts) для оптимізації пам’яті. Такі підходи дозволяють запускати навіть десятки мільярдів параметрів на споживчому рівні компьютерів. Зокрема, техніка квантизації продемонструвала, що модель на ~30 млрд. параметрів можна розгорнути на одному висококласному GPU з 24 ГБ пам’яті без суттєвої втрати точності. Це відкрило шлях до автономних середовищ розробки зі ШІ, де весь аналіз коду та генерування підказок відбувається на робочій станції розробника без будь-якого інтернет-з’єднання.

Локальні агенти часто глибше інтегровані з системою: вони можуть мати доступ до файлової системи, репозиторію, виконувати shell-команди. Деякі з них позиціонуються як автономні ШІ-розробники, здатні самостійно створювати, модифікувати та запускати код за заданим завданням. Яскравий приклад – проєкт Devin (закритий прототип) та його відкритий аналог OpenDevin. OpenDevin – це open-source ініціатива, що прагне відтворити можливості автономного ШІ-інженера. Модель отримує доступ до командного середовища, редактора коду і браузера, щоб виконувати комплексні інженерні завдання і співпрацювати з людиною-розробником. Фактично, OpenDevin – це консольний агент, що може читати та писати файли, запускати скрипти, генерувати тести й, навіть, гуглити інформацію, реагуючи на інструкції користувача. Подібні системи поки що експериментальні, але вони демонструють потенціал локального ШІ не просто як помічника, а як автономного учасника проєкту.

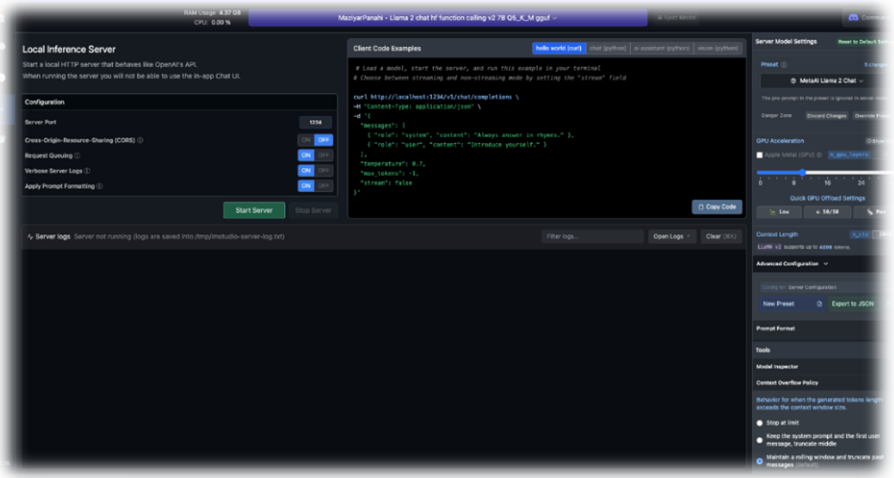

Рисунок 2.6 – LM Studio

LM Studio(рис. 2.6) – локальний GUI звернення до LLM. Він надає зручний інтерфейс, де користувач може вибрати та завантажити популярні відкриті моделі (GPT-J, LLaMA, Qwen, Gemma, DeepSeek тощо), спілкуватися з ними офлайн.

Важливо, що LM Studio не збирає дані користувача та не передає їх назовні. Всі чат-діалоги та виконання проходять лише на локальному комп'ютері. Як наслідок, дані розробника повністю захищені, а звичний багатовіконний чат-інтерфейс забезпечує досвід, подібний до ChatGPT, але без інтернету.

Ollama – простий у використанні інструмент для Mac/Windows, що дозволяє завантажувати локальні моделі командою ollama pull та запускати їх командою ollama run. Працює через термінал та може використовувати бібліотеку власних підготовлених моделей або імпортувати зовнішні (підтримується понад 100 готових моделей у бібліотеці ollama.com). Головна ідея Ollama – зробити запуск локального бота максимально простим: без підключення до API OpenШІ, без передплати. Розробник може створювати локальні чат-боти без доступу до інтернету, тобто весь inference проходить на CPU/GPU машини. Це фактично Copilot-альтернатива, що працює повністю офлайн. У подальшому, такі інструменти зможуть вбудовуватися в IDE, але вже зараз вони привабливі для тих, хто цінує приватність[13].

OpenDevin можна розгорнути у Docker-контейнері локально і підключити до нього свій API-ключ моделі (OpenШІ або іншої). Після запуску, OpenDevin отримує від вас вимоги, сам створює код у файлах, виконує команди Bash в ізольованому середовищі, генерує тести та навіть відкриває URL у вбудованому браузері з метою пошуку інформації. Це доволі новий підхід, але такі автономні агенти можуть радикально прискорити розробку, автоматизувавши рутину, поки людина зосередиться на творчих аспектах.

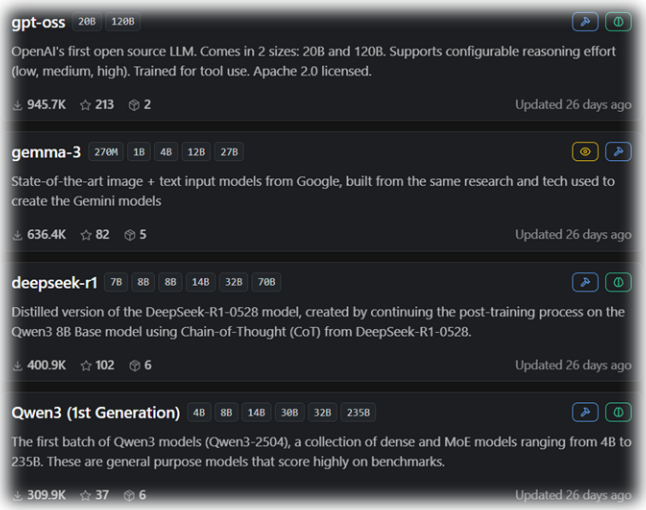

Рисунок 2.7 – Локальні ШІ-агенти

Отже, локальні ШІ-агенти(рис. 2.7) – це шлях до приватності та контролю, але він потребує ресурсів і технічного досвіду. Вони особливо ефективні у великих організаціях з суворими вимогами безпеки. Там вигоди (захист даних, гнучкість) переважують затрати. Стосовно індивідуальних потреб часто обирають компромісні рішення: або менш потужні моделі, або гібридні сервіси. Цікаво, що тренди у цій сфері свідчать щодо постійної еволюції локальних моделей. Завдяки

таким підходам, як оптимізація архітектур (спарсність, Mixture-of-Experts), квантизація та техніки, наприклад, LoRA (Low-Rank Adaptation) стосовно донавчання, ми бачимо як якість локальних LLM зростає, а вимоги до їх запуску знижуються. Наприклад, Google DeepMind у 2024 році відкрив сімейство моделей Gemma – відносно невеликі відкриті моделі (2B, 6B, 13B параметрів), спеціально натреновані з урахуванням Gemini-технологій, та їхню версію для коду – CodeGemma. CodeGemma 7B, розгорнута локально, здатна виконувати заповнення коду і генерацію з точністю, близькою до більших моделей, але значно швидше. У свою чергу, компанія Mistral випустила open-source модель Mistral 7B, а слідом й її варіант стосовно коду під назвою Devstral. Devstral розроблено з метою легкого локального деплою, щоб компанії могли генерувати код без ризику витоку інформації.

Таким чином, зазначені приклади підтверджують: локальні моделі стають розумнішими та доступнішими, тому все більше команд зможуть ними скористатися.

2.1.2.2 Переваги та недоліки локальних агентів

Повна конфіденційність та контроль над даними. Найбільший плюс локальних LLM – те, що жоден рядок коду чи чутливих даних не покидає межі вашого комп'ютера або корпоративного середовища. Вся обробка виконується локально, тож ризик несанкціонованого витоку через сторонні API відсутній. Це критично для компаній у сфері фінансів, охорони здоров'я, оборони, де політики безпеки суворо забороняють передавати код третім особам. Локальний агент можна розгорнути навіть у ізольованій мережі або на машині без доступа до інтернету (ШІr-gapped). Таке неможливо з хмарними сервісами. Більше того, деякі рішення (наприклад, Tabnine) дозволяють розгортати моделі у приватному хмарному середовищі чи VPC, але без передачі даних назовні компанії. Отже, організація має повний суверенітет над своїм кодом та може гарантувати відповідність вимогам GDPR, внутрішніх політик та іншим регуляторним нормам.

Відсутність залежності від зовнішніх сервісів та стабільність. Локальний ШІ працює навіть якщо зовнішній API недоступний, інтернет відключено, або провайдер змінив умови користування. Компанія не ризикує втратити доступ до інструменту через збій на стороні або політичні обставини. Також немає обмежень на кількість запитів чи швидкість відповіді, окрім можливостей власного обладнання. Це важливо командам, що працюють 24/7 і не можуть покладатися на SLA сторонніх сервісів. Локальний агент завжди з предбачуваною латентністю (що визначається лише продуктивністю CPU/GPU).

Гнучкість кастомізації та розширення. На локальній моделі розробники можуть проводити додаткове навчання або тонке налаштування на своїх власних даних. Наприклад, доучити модель на специфічному внутрішньому коді, щоб вона краще розуміла стиль та бібліотеки, що використовуються у компанії. Можна додавати власні правила, фільтри, або інтегрувати модель з власними інструментами. Стосовно хмарних рішень такі можливості обмежені або недоступні. Відкритість деяких локальних моделей (open-source) дозволяє

спільноті покращувати їх, знаходити оптимальні архітектури, ділитися напрацюваннями. Таким чином, локальний агент може еволюціонувати разом з потребами команди.

Інтеграція з локальною екосистемою розробки. Консольні агенти легко взаємодіють з файловою системою, Git-репозиторіями, інструментами збірки та деплойменту, оскільки виконуються у тому ж середовищі, що й решта девопс-ланцюжка. Це відкриває шлях до автоматизації рутинних завдань: агент може сам згенерувати pull request, запустити тестування, або відреагувати на подію у системі. Наприклад, інструмент OpenDevin здатен виконувати shell-команди та демонструвати результат користувачу. Це означає, що ШІ може автоматично скомпілювати проєкт, запустити сценарії або зібрати логи помилок без участі людини. Така безшовна інтеграція особливо корисна у побудові розумних CI/CD-пайплайнів та DevOps-агентів.

Відсутність витрат на запити та підписки. Після початкового розгортання локальної моделі, її використання зазвичай не передбачає плати за кожен запит. Хмарні API, навпаки, тарифікуються за кількістю токенів або мають щомісячну плату. У довгостроковій перспективі, якщо команда активно користується ШІ щодня, локальне рішення може бути більш економічно вигідним. Тим більше, що зростання складності завдань не призводить до лінійного зростання витрат. Єдиний н'юанс – це витрати на електроенергію та підтримку обладнання, але компанії з власними серверами це не критично. До того ж, відсутній ризик раптової зміни цін чи політики провайдера, що теж додає впевненості у плануванні.

Застосування локальних ШІ пов'язане з певними проблемами та викликами.

Hardware requirements for 8-bit quantized DeepSeek R1

Model	System RAM	CPU Cores	GPU VRAM	Recommended Nvidia GPU	Recommended AMD GPU
1.5B	16 GB	4	8 GB	RTX 4060 Ti 8GB	RX 7600 8GB
7B	32 GB	8	16 GB	RTX 4060 Ti 16GB	RX 7700 XT 12GB
8B	32 GB	16	16 GB	RTX 4060 Ti 16GB	RX 7800 XT 16GB
14B	64 GB	24	16–24 GB	RTX 4090 24GB	RX 7900 XT 24GB
32B	128 GB	32	32–48 GB	RTX A6000 48GB	Instinct MI250X
70B	256 GB	48	48+ GB	A100 80GB	Instinct MI250X
671B	1 TB+	64+	80+ GB	H100 80GB (multi-GPU)	Instinct MI300 (multi-GPU)

Рисунок 2.8 – Вимоги щодо роботи локального ШІ

Високі вимоги до апаратного забезпечення. Великі моделі вимагають значних ресурсів(рис. 2.8) – пам'яті, процесорного часу, графічних прискорювачів. Розгорнути сучасний LLM на звичайному ноутбуці може бути неможливо або дуже повільно. Навіть після оптимізацій параметри на ~7-13 млрд. потребуватимуть 10-20 ГБ оперативної пам'яті, а для комфортної роботи бажано мати GPU з 8+ ГБ відеопам'яті. Більші моделі (70B+) без квантизації взагалі

потребують десятків гігабайт пам’яті та кількох відеокарт. Таким чином, компанії мусять інвестувати у високопродуктивну техніку або виділяти серверні ресурси під локальні LLM. Це не завжди виправдано щодо маленьких команд або індивідуальних розробників. Звичайно, можна використовувати менш потужні моделі, але вони іноді поступаються якістю відповіді хмарним гігантам. Отже, є певний технічний бар’єр входження у локальний ШІ[14].

Складність налаштування та підтримки. На відміну від готового хмарного сервісу, локальний агент потребує встановлення програмного забезпечення, конфігурації моделі, оптимізації продуктивності. Стосовно різних операційних систем ці кроки можуть відрізнятися. Відкриті проєкти інколи не мають зручних інсталяторів або стикаються з проблемами сумісності. Розробнику може знадобитися час, щоб розібратися з форматами моделей, залежностями (наприклад, CUDA для GPU), оновленнями версій. Крім того, підтримка таких систем лягає на плечі самої команди. Немає сторонньої служби підтримки, як у випадку SaaS-продукту. Це означає додаткове навантаження на IT-відділ. Однак, ситуація покращується з появою user-friendly інструментів на кшталт LM Studio, що значно спрощують запуск локальних LLM навіть стосовно нетехнічних користувачів.

Відставання моделей у потужності. Комерційні провайдери (OpenШІ, Google, Anthropic) володіють найсучаснішими моделями, що не завжди доступні локально. Відкриті аналоги, наприклад, LLaMA2, Mistral, Falcon хоча й прогресують, але іноді трохи поступаються за якістю генерації коду спеціалізованим продуктам типу Codex або GPT-4. Хоча у 2023–2024 рр. був значний ривок відкритих моделей щодо коду (StarCoder, WizardCoder, CodeLlama тощо), відрив усе ще є. Наприклад, GPT-4 залишається одним з найкращих у розумінні складних завдань та генерації довгих фрагментів коду. Локальні ж моделі часто доводиться комбінувати: менша – швидкі підказки, більша – складніші, але повільніші обчислення. Це додає шар складності. До того ж, дуже великі моделі неможливо запустити локально без кластерів GPU, тому доводиться шукати компроміс між швидкодією та якістю.

Початкові витрати та TCO. Якщо організація обирає шлях локального ШІ, їй потрібно закупити або виділити обладнання. Сервер з високоякісними GPU може коштувати десятки тисяч доларів. Такі одноразові капіталовкладення не кожному підходять. Хмарні сервіси натомість дозволяють почати з малих сум (щомісячна підписка) й масштабувати витрати за потребою. Крім того, загальна вартість володіння (TCO) локального рішення включає електроенергію, амортизацію, охолодження серверів, адміністрування тощо. Стосовно великих компаній це прийнятно, але для індивідуальних розробників – ні. З іншого боку, варто зазначити, що гібридні підходи (як у Tabnine) дозволяють частково зняти навантаження. Менш вимогливі завдання виконувати локально, а зі складними обчисленнями звертатися до хмари. Це дає баланс між витратами й ефективністю, хоча й повертає ризики конфіденційності щодо тих конкретних задач, що йдуть у хмару.

Обмежена підтримка мультимодальності та оновлень. Більшість локальних моделей наразі зосереджені на тексті (генерація коду, пояснення). Підтримка зображень чи аудіо – велика рідкість. Якщо розробнику потрібна, скажімо, OCR-функціональність, доведеться інтегрувати окремі бібліотеки. На відміну від цього, веб-моделі типу GPT-4 уже мають vision-можливості. Так само й оновлення моделі: якщо OpenШІ сьогодні покращить GPT-4, користувач відразу отримає вигоду. Локальну модель потрібно перекачувати та переінтегровувати самостійно, якщо виходить нова версія. Це накладає відповідальність слідкувати за розвитком open-source спільноти, щоб не відстати від прогресу.

2.1.3 Додатки зі штучним інтелектом до IDE

2.1.3.1 Принцип роботи

Інтегровані середовища розробки (IDE) – це основне робоче місце програміста. Логічним кроком було вбудувати ШІ-помічників безпосередньо в IDE, щоб вони допомагали писати код у реальному часі. Так з'явилися додатки й розширення з ШІ-функціональністю. Принцип їхньої роботи полягає у комбінуванні можливостей мовної моделі з контекстом, що IDE може надати: відкриті файли, дерево проєкту, виділений код тощо.

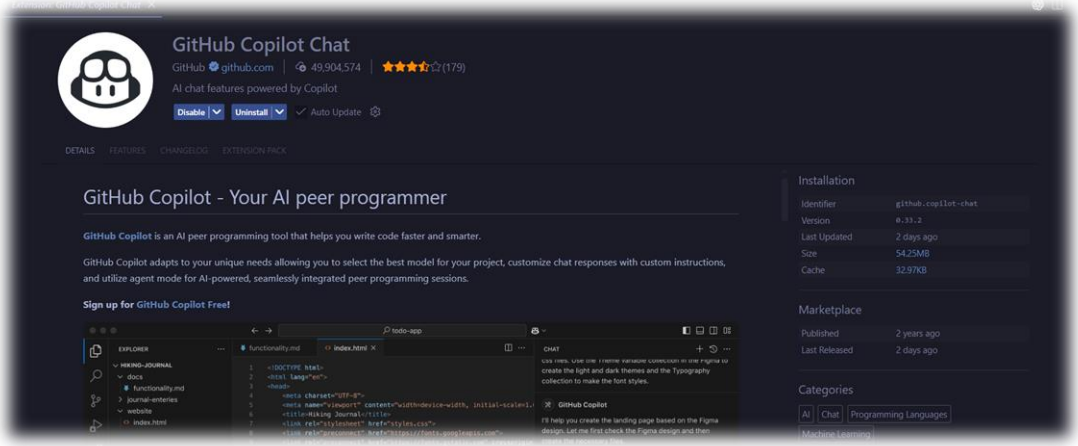

Рисунок 2.9 – Розширення VS Code

Архітектурно розширення(рис. 2.9) зазвичай працюють як клієнт-сервер: сам плагін встановлюється в IDE та відслідковує дії розробника (набір тексту, курсор, виклики спеціальних команд), а модель може бути або локальною, або у хмарі. Наприклад, GitHub Copilot під час роботи у Visual Studio Code надсилає фрагмент поточного файлу (до певного обмеження) на сервер OpenШІ й отримує у відповідь пропозицію продовження коду, що відображає сірим текстом прямо у редакторі. Якщо пропозиція підходить, розробник натискає Tab, а код автодоповнюється. Схоже працює Tabnine: він може виконувати обчислення локально або у хмарі (залежно від режиму), але інтеграція теж відбувається через стандартний механізм completion IDE.

Більш просунуті плагіни додають чат-панелі прямо в IDE. Так, Copilot Chat чи IntelliJ ШІ Assistant дають змогу відкрити бокову панель, поставити там питання (включаючи посилання на код у проєкті) й отримати розгорнуту відповідь чи шматок коду у відповідь. У цьому випадку плагін має доступ до всіх файлів проєкту та може надсилати цікавий контекст. Це створює ефект розуміння всього вашого проєкту. Зауважимо, що плагіни можуть також інтегруватися з інструментами налагодження: наприклад, у разі виникненні помилки можна натиснути кнопку «спитати Copilot», а той проаналізує traceback або message помилки та запропонує діагностику й виправлення.

Деякі інноваційні IDE з нуля будуються навколо ШІ.

Наприклад, Cursor IDE – це модифікований редактор (форк VS Code), де основна увага надана інтегрованому чат-асистенту та режимам його роботи. Cursor дозволяє у режимі Agent доручити ШІ складні завдання. Він сам внесе потрібні зміни одразу у кількох файлах, може виконати команду у терміналі, щоб перевірити код, та інше. Є й Manual режим, коли ШІ пропонує конкретні правки, але застосовує їх лише за підтвердженням розробника. Цікаво, що Cursor підтримує завантаження у чат зображень (наприклад, скриншотів з помилками або макетів дизайну), а ШІ здатний враховувати їх при відповіді. Це додає новий рівень контексту, недоступний у традиційних IDE. Інші плагіни (наприклад, у JetBrШIns IDE) також почали експериментувати з прийомом зображень або PDF у чат-режимі стосовно пояснення їх вмісту.

GitHub Copilot – найвідоміший ШІ-плагін, що підтримує практично всі популярні IDE: Visual Studio Code, Visual Studio, JetBrШIns-сімейство (IntelliJ, PyCharm etc.), Neovim, а також інтегрується у хмарні редактори, наприклад, GitHub Codespaces. Copilot у реальному часі пропонує продовження коду під час друку, а також має режим Copilot Chat щодо детальніших запитів у боковій панелі. Його ключова сила – тренування на всьому масиві відкритого коду GitHub, що дозволяє підказувати навіть нетривіальні фрагменти коду та враховувати контекст до 100-200 рядків навколо курсора. У Copilot Chat можна прямо в IDE отримувати пропозиції коду, пояснення коду, генерувати unit-тести й поради, щодо виправлення помилок.

Tabnine спершу був як локальний ШІ-autocomplete, що вбудовувався в IDE та пропонував продовження фраз, як більш розумна версія стандартного IntelliSense. Сьогодні Tabnine теж має командний чат-інтерфейс та працює у гібридному режимі (частково локально, частково через хмару з кращою якістю). Головний наголос Tabnine робить на конфіденційності: навіть у хмарному режимі він не зберігає фрагменти коду користувача та тренує свої моделі лише на ліцензованому open-source коді. Для enterprise-версій Tabnine пропонує повністю офлайн-деплой й розгорнення на власному сервері з доступом через плагіни в IDE. Такі можливості привабливі стосовно великих компаній, що хочуть Copilot-подібний досвід без ризиків для своєї кодової бази[15].

Cursor IDE що інтегрує LLM (за вибором: OpenШІ або інші через API) глибоко у процес редагування коду. Cursor надає різні режими співпраці з ШІ: від повністю автоматичного агентування (коли ШІ бере ініціативу у великому

завданні) до ручного керування та точкового застосування змін. Також Cursor підтримує RAG-функції (від Retrieval-Augmented Generation). Тобто інтеграцію з пошуком за кодовою базою: за запитом асистент може знайти відповідний файл чи приклад використання класу та навести його. Все це відбувається у єдиному вікні, що робить IDE не просто редактором, а інтерактивним середовищем спільної роботи людини та ШІ.

Таким чином, ШІ-розширення стосовно IDE забезпечують найбільш збалансований та продуктивний на сьогодні підхід до використання ШІ у програмуванні. Вони поєднують переваги передових моделей зі зручністю середовища розробки, що вже знайоме програмістам. За правильної інтеграції (з урахуванням безпеки та продуктивності) такі інструменти дійсно виконують роль співрозробника. Не дарма аналітики прогнозують, що найближчими роками функції автодоповнення й ШІ-аналізу коду стануть стандартною частиною будь-якого IDE, так само як колись автозавершення стало невід'ємним атрибутом редактора коду.

2.1.3.2 Переваги та недоліки інтеграції в IDE

Інтеграція ШІ в IDE має свої позитивні сторони.

Безперервність роботи. Розробнику не потрібно переключатися на браузер або іншу програму, щоб спитати пораду у ШІ. Все відбувається у тому ж вікні, де він пише код. Це значно економить час та ментальне переключення контексту. Дослідження продуктивності показують, що часті перемикання між інструментами знижують ефективність роботи, тому інтегрований ШІ-асистент допомагає залишатися у потоці розробки. За результатами опитувань, програмісти цінують Copilot саме за здатність швидко отримати підказку чи продовження коду без відриву від редактора.

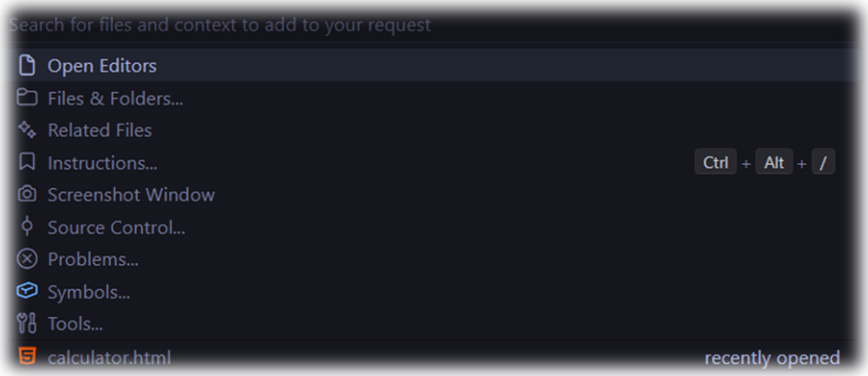

Рисунок 2.10 – Контекстуальне розуміння проєкту

Контекстуальне розуміння проєкту(рис. 2.10). На відміну від веб-моделі, плагін має інформацію стосовно всього проєкту: структуру директорій, імена класів та функцій, навіть зміни, що ще не закомічені у Git. Завдяки цьому підказки стають набагато релевантнішими. ШІ може автоматично імпортувати потрібні модулі, знати про оголошені раніше змінні, стилі коду тощо. Це зменшує кількість

правок після автогенерації. Copilot, наприклад, аналізує до 100 рядків вище курсора та до 100 рядків нижче, плюс content інших відкритих файлів, щоб сформувати пропозицію продовження. Cursor IDE взагалі дозволяє агенту проглянути багато файлів та, навіть, шукати у кодовій базі за вас. Таке глибоке розуміння контексту IDE робить ШІ справжнім «розумним компаньйоном», обізнаним з поточним кодом.

Автоматизація однотипних завдань. IDE-плагіни значно пришвидшують рутинні операції: написання стандартних конструкцій, генерування шаблонного коду, додавання коментарів, документації та тестів. Замість того, щоб вручну писати однотипні геттери/сеттери чи SQL-запити, розробник може або викликати автодоповнення, або попросити у чаті: “згенеруй функціональні тести для цього класу” та отримати готовий код. Наприклад, Copilot Chat може згенерувати unit-тести або мок-об’єкти на основі наявного коду. Він також здатен пояснити складну ділянку коду англійською (чи українською) мовою, що економить час на вивчення чужого коду. Деякі плагіни мають функції рефакторингу. Можна попросити: “розбий цю функцію на менші” або “перейменуй параметри для читабельності”, а ШІ згенерує відповідні зміни. Все це зменшує обсяг монотонної роботи та дає змогу розробнику зосередитись на логіці.

Покращення якості коду та знань команди. ШІ-асистент може служити інтерактивним вчителем або рев’юером(рис. 2.11) прямо в IDE. Розробники можуть питати, чому виникає помилка, чи є кращий спосіб реалізувати алгоритм, як використати ту чи іншу бібліотеку. Copilot або його аналоги часто можуть дати пояснення (базовані на своєму тренувальному корпусі, що включає документацію, Q&A тощо). Це своєрідна миттєва документація: замість пошуку в інтернеті, отримуєш відповідь у редакторі з прикладом коду. Крім того, інструменти на зразок Visual Studio IntelliCode (що теж використовує ШІ) аналізують стилі коду команди та дають підказки, узгоджені з найкращими практиками. У результаті, якість коду вирівнюється, а молоді спеціалісти швидше вчаться, отримуючи навики від ШІ.

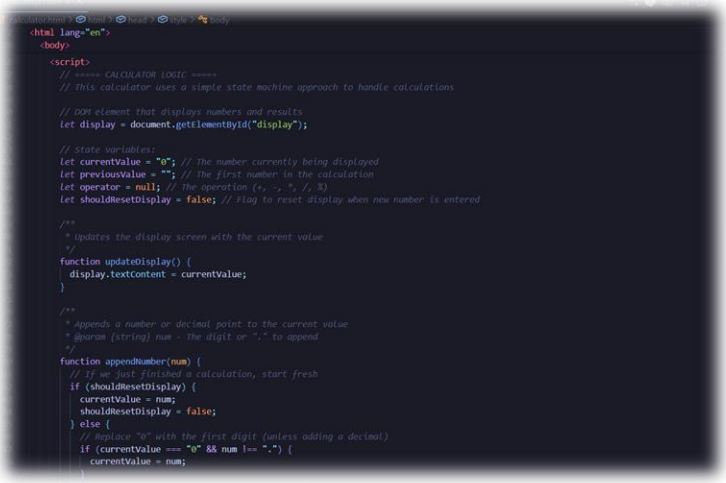

Рисунок 2.11 – Створені коментарі до коду ШІ

Прискорення розробки. У сукупності вищезазначені фактори (автодоповнення, менше перемикань, автоматизація тестів та коментарів) призводять до значної економії часу. За дослідженнями від самої GitHub, використання Copilot може скоротити час виконання окремих завдань на 20-50% залежно від типу завдання. Особливо це помітно при написанні повторюваного коду або роботи з новим API – ШІ пропонує готові рішення, та залишається лише їх перевірити. Крім того, такі плагіни сприяють тому, що розробник рідше відволікається на сторонні чинники (не потрібно гуглити помилку – Copilot Chat підкаже рішення, не треба шукати документацію – ШІ вже знає синтаксис). Все це позитивно впливає на швидкість циклу розробки. Як наслідок, випуск функціоналу прискорюється, а вивільнений час можна витратити на більш складні проблеми архітектури чи дизайну.

Втім, інтеграція ШІ має та свої мінуси, що необхідно враховувати.

Потреба в інтернеті та віддалених моделях (у більшості випадків). Багато популярних плагінів (Copilot, Amazon CodeWhisperer, Codeium) спираються на хмарні API. Тобто IDE лише слугує оболонкою, але весь процес виконується на серверах. Це означає, що без зв’язку з інтернетом плагін марний. Для команд з обмеженим доступом до мережі (наприклад, на закритих об’єктах) такі інструменти не підійдуть. Хоча існують повністю локальні варіанти (Tabnine у локальному режимі, деякі розширення щодо VS Code на основі LLaMA.cpp), вони часто менш потужні або потребують ручного налаштування. До того ж, передача коду у хмару з IDE несе ті самі ризики безпеки, що й веб-чати. Фактично, Copilot надсилає ваші редаговані файли на сервер OpenШІ за кожної генерації підказки. Хоча ці дані буцімто анонімізуються та зберігаються тимчасово, сам факт може суперечити корпоративним політикам безпеки.

Ліцензійні та правові аспекти. Деякі ШІ-плагіни є пропрієтарними та вимагають платної підписки. Це додаткові витрати, що не всі готові нести. Окрім того, досі точаться дискусії, щодо юридичної чистоти згенерованого коду. Copilot був навчений на публічних репозиторіях, у т.ч. з ліцензіями GPL/MIT, та іноді пропонує фрагменти, дуже схожі на оригінали. Хоча GitHub запровадив фільтр, що намагається відсікати точні збіги понад 150 символів та надає Copilot Copyright Commitment (зобов’язання захищати користувачів від позовів), ризик ліцензійних конфліктів залишається. Щодо enterprise це болюче питання: деякі уникають таких інструментів, щоб не було претензій стосовно неявного копіювання чужого коду. Альтернатива – використовувати плагіни, що тренуються лише на відкритих даних з дозволом (як Tabnine із пермисивними моделями), або локально доучувати модель на власному коді.

Можливе зниження продуктивності IDE. Інтеграція ШІ – досить важка операція з погляду обчислень та обміну даними. Коли плагін постійно аналізує ваш введений текст та посилає запити, це може навантажувати систему. Ряд користувачів відзначали, що увімкнений Copilot може сповільнити IDE: затримки в авто-доповненні, пригальмовування підсвітки синтаксису, довший час збереження файлу. Особливо це помітно на великих проєктах або менш потужних комп’ютерах. Такі баги поступово виправляються, але все ж іноді доводиться

відключати плагін, коли він заважає (наприклад, при редагуванні дуже великих файлів). Також інтеграція може конфліктувати з іншими розширеннями IDE. Загалом, розробникам треба слідкувати, щоб ШІ-асистент не перетворився з помічника на тягар системи(рис. 2.12).

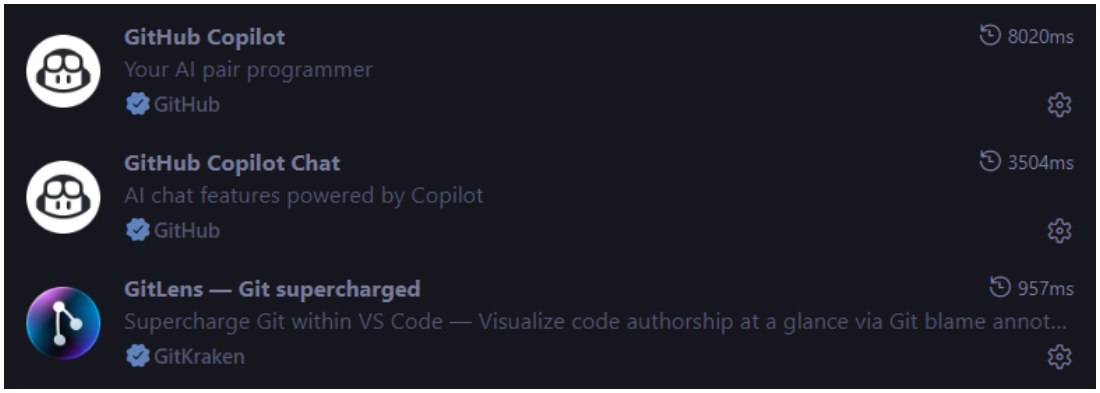

Рисунок 2.12 – Час запуску розширень IDE

Надмірна самовпевненість та помилки. Хоч ШІ в IDE відчувається як розумний напарник, йому теж не можна сліпо довіряти. Генерації бувають неправильними, незважаючи на впевнений вигляд. Якщо розробник бездумно приймає всі підказки, він ризикує ввести у код нові баги або небезпечні місця. Є результати досліджень, що менш досвідчені програмісти, використовуючи Copilot, можуть навіть частіше писати вразливий код, бо покладаються на його пропозиції. Отже, необхідно зберігати критичність: перевіряти логіку, запускати тести. Втім, ця проблема стосується всіх ШІ інструментів, а не лише IDE-плагінів. Перевага інтегрованих систем у тому, що вони можуть одразу допомогти протестувати код – наприклад, Copilot може згенерувати тест на щойно вставлену функцію. Але відповідальність за перевірку все одно лежить на людині.

Можлива зміна робочих процесів та необхідність у навчанні. Впровадження ШІ у команді вимагає адаптації: розробникам слід навчитись правильно формулювати запити до Copilot Chat, використовувати його можливості ефективно. Це нова навичка – своєрідний prompt engineering. Спершу може здаватися швидше написати код самому, ніж вгадувати, що потрібно спитати у асистента. Також треба налагодити практики code review, коли частину коду написав ШІ: можливо, прописати правила, як маркувати такі ділянки чи на що звертати увагу при рев'ю. Зміна звичок розробників завжди нелегка – хтось може бути скептичним або використовувати інструмент не повною мірою. Тому менеджерам варто проводити навчання, ділитися кейсами використання, встановлювати баланс між довірою до ШІ і перевірками.

Картина складається доволі багаторівнева. Кожен із трьох підходів: IDE-інтеграції, локальні агенти та веб-чати — не є конкурентами у прямому сенсі. Вони скоріше утворюють трикутник можливостей, де кожна вершина відповідає певному балансу між зручністю, потужністю та безпекою. У типовому робочому середовищі команди ці інструменти часто не замінюють один одного, а мирно співіснують. Розробник генерує ідеї або складні фрагменти алгоритмів у веб-чаті,

доводить їх до робочої форми безпосередньо в IDE, а локальні моделі виконують роль надійних автоматизаторів для чутливих частин проєкту чи внутрішніх сервісів.

З цього випливає важлива закономірність: ефективність ШІ у розробці залежить не стільки від самої моделі, скільки від того, як саме вона інтегрована у робочий цикл. Команда, що використовує ІІ-чати лише для генерації шматків коду, виграє у швидкості, але втрачає у структурності й узгодженості результатів. Ті, хто робить ставку на локальних агентів, отримують максимальний контроль та приватність, але мусять інвестувати в інфраструктуру й компетенції. Інтеграції в IDE забезпечують оптимальну синергію. Вони привносять інтелект у найбільш рутинні точки щоденної роботи: автодоповнення, рефакторинг, навігацію, написання тестів. Все це зменшує когнітивне навантаження та дозволяє розробнику приділяти більше часу архітектурним ідейним рішенням.

У підсумку вибір інструменту визначається не стільки можливостями, скільки філософією розробки, зрілістю процесів та рівнем вимог до безпеки. Сучасний ландшафт ШІ-інструментів у програмуванні нагадує екосистему: різноманітну, саморегульовану, з елементами взаємодоповнення. Чим краще команда розуміє призначення кожного підходу та уміє поєднувати їх у власному пі, тим більша синергія між швидкістю створення продукту та його якістю.

Аналіз за критеріями наведено у додатку Б.

2.2 Інструменти у сфері написання програмного коду

2.2.1 Автодоповнення (кодове autocomplete)

Автодоповнення коду(рис. 2.13) – це використання ШІ з метою прогнозування та підказування фрагментів коду. Система аналізує контекст: оголошені змінні, функції, типи даних та пропонує завершення рядка або блоку коду відповідно до патернів й найкращих практик. Такий ШІ-асистент може підказати наступний виклик методу, завершити синтаксис або навіть згенерувати цілу стандартну конструкцію. Автодоповнення прискорює написання рутинного boilerplate-коду та зменшує ймовірність синтаксичних помилок.

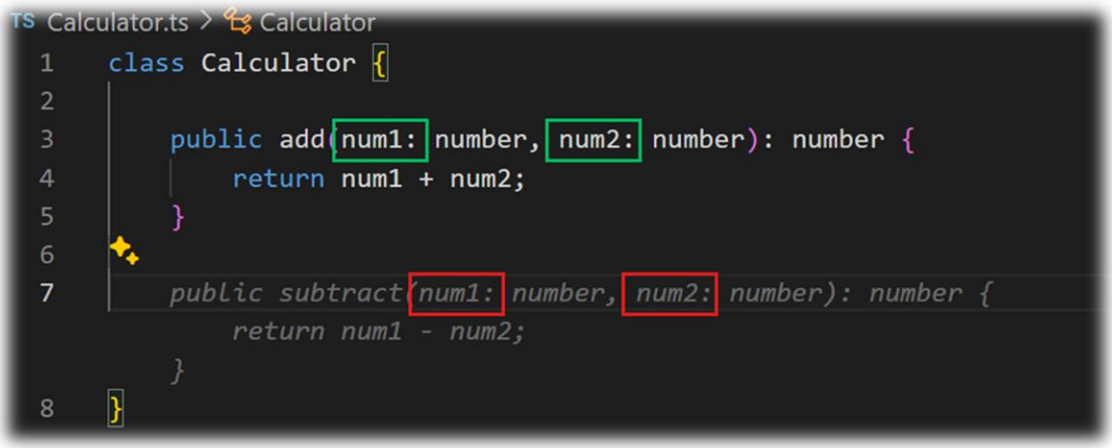

Рисунок 2.13 – Автодоповнення

Сьогодні ШІ-автодоповнення стало звичним інструментом у багатьох IDE та редакторах коду, що працює у зв’язці з іншими засобами аналізу й відладки. Близько 76% розробників вже використовують або планують використовувати ШІ-інструменти у розробці. Автодоповнення – одна з найчастіших щоденних взаємодій з такими асистентами. Кожного разу, коли програміст набирає код, ШІ може пропонувати продовження, тому фактично автодоповнення задіяне постійно під час кодування[16].

Швидкість та продуктивність. Автодоповнення прискорює набір коду, автоматично дописуючи типові фрагменти. Це знижує когнітивне навантаження на розробника та дає змогу зосередитися на логіці, а не на синтаксисі. За оцінками, ШІ-асистенти можуть пришвидшити розробку на 20–50%, особливо за рахунок автоматизації рутинних ділянок та типових конструкцій[16].

Точність та якість. Підказки базуються на найкращих практиках та поширених шаблонах, тому допомагають уникати помилок й підтримувати стандартизований стиль коду. ШІ забезпечує послідовність у найменуваннях та форматуванні, що полегшує подальшу підтримку коду.

Швидше навчання новачків. Для початківців підказки ШІ слугують своєрідним коучем: показують правильний синтаксис, рекомендують виклики функцій та нагадують про часті помилки. Це прискорює освоєння мов програмування та фреймворків, адже новачок одразу бачить приклади використання API й патернів у редакторі.

Недоліки та обмеження. Надмірна залежність. Якщо занадто покладатися на автодоповнення, можна притупити свої власні навички кодування. Розробник може звикнути приймати підказки без критичного аналізу, що з часом загрожує втратою творчого підходу та здатності вирішувати нестандартні завдання. Іншими словами, ШІ не вчить мислити поза шаблонами, тому завжди потрібен баланс між зручністю та практикою самостійного кодування.

Помилкові підказки. Модель ШІ може час від часу пропонувати некоректний або неповний код. Такі хибні рекомендації ризикують внести баги або вразливості, якщо їх не помітити. Наприклад, ШІ може неправильно зрозуміти контекст та запропонувати не ту змінну або метод, особливо у специфічних або нестандартних ситуаціях.

Обмежена креативність. Автодоповнення добре працює з типовим кодом, але не генерує принципово нових рішень. Воно може звузити коло розв’язків до тих, що є в навчальних даних, тим самим стримуючи нестандартні підходи розробника. Також підтримка менш поширених мов чи фреймворків може бути обмеженою, тож в таких випадках користь ШІ менша.

Перспективи. Очікується врахування ширшого контексту проєкту. Наприклад, аналіз всього репозиторію, щоб підказки відповідали саме вашій кодовій базі. Вже зараз ці інструменти інтегруються майже непомітно у робочий процес, а надалі можуть з’явитися й мультимодальні підказки (наприклад, голосові або з урахуванням діаграм дизайну). Прогнозується, що автодоповнення перетвориться з утиліти із завершення рядка на повноцінного співрозробника, який розуміє цілі проєкту. Водночас зростатиме потреба у механізмах контролю

якості ШІ-виводу та підтримка навичок розробників, щоб уникнути описаних ризиків.

2.2.2 Написання коду

Генерація коду за описом охоплює використання ШІ для генерації фрагментів або повних блоків коду. Розробник може задати вимогу природною мовою, а ШІ згенерує відповідний код. Такі помічники, як ChatGPT або GitHub Copilot, здатні створювати цілі функції, класи чи навіть прості модулі на основі опису завдання. Генерація коду особливо корисна для шаблонних повторюваних завдань: створення CRUD-методів, конвертерів, обгорток для API тощо. Це суттєво економить час на написання рутинної частини коду, який потім можна підлаштувати під потреби.

Частота використання. У щоденній роботі програміста повна генерація блоків коду використовується дещо рідше, ніж автодоповнення, але все одно стає дедалі поширенішою. За опитуваннями, понад 60-70% розробників вже пробували інструменти на зразок Copilot для написання коду. Часто це трапляється на етапі прототипування або коли треба швидко розгорнути новий компонент. У коротких циклах розробки розробники можуть кілька разів на день звертатися до ШІ щодо створення чернетки коду, щоб прискорити реалізацію фічі[17].

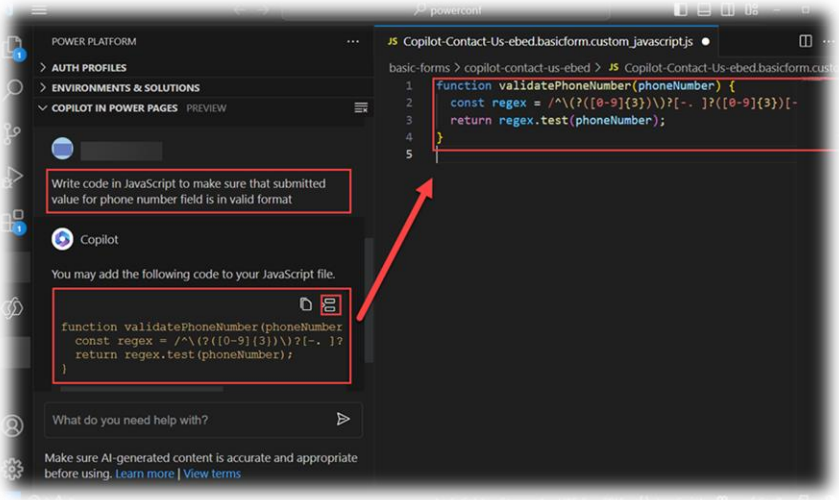

Рисунок 2.14 – Створення коду

Переваги ШІ. Прискорення розробки. ШІ може миттєво згенерувати код(рис. 2.14), на написання якого вручну пішли б години. Це особливо відчутно на старті нового проєкту або модуля. ШІ швидко створює шаблон, який розробник вже допрацьовує. У результаті цикл розробки скорочується, а вихід продукту на ринок відбувається швидше.

Автоматизація рутини. ШІ-асистенти беруть на себе рутинні частини роботи: написання тривіальних методів, повторюваних конструкцій, конвертацій між форматами даних тощо. Це звільняє час програміста для більш складних завдань, де потрібні творчість й архітектурне мислення. Як показало одне дослідження,

навіть коли пропозиції ШІ були неточні, вони все одно допомагали розробникам швидше знаходити та виправляти помилки, ніж за ручного набору.

Покращення якості (за умови контролю). Генератори коду, що навчені на великих обсягах відкритого коду, часто пропонують рішення з урахуванням найкращих практик та типових помилок. Це може позитивно вплинути на якість. ШІ підказує, як зробити код більш оптимізованим чи безпечним, нагадує про перевірки null тощо. У результаті за належного нагляду якість та консистентність бази коду зростає.

Неточність та помилки. ШІ не гарантує 100% правильного коду. Дослідження показують, що від 35% до 70% випадків згенерований код містить помилки або не працює з першого разу. Менше 3% розробників висловлюють повну довіру до точності коду від ШІ. Отже, кожну згенеровану функцію треба ретельно перевіряти. ШІ може пропустити граничний випадок, згенерувати застарілий або небезпечний код. Без рев'ю використання такого коду несе ризики багів і вразливостей[17].

Відсутність контексту та розуміння задуму. ШІ-модель не знає бізнес-логіку та контекст проєкту так, як це знає розробник. Вона оперує статистичними відповідностями, тому може не врахувати архітектурних особливостей чи нефункціональних вимог. Наприклад, ШІ згенерує рішення, що технічно працює, але не відповідає вимогам безпеки або не вписується у загальний дизайн системи. Без спеціальних підказок модель не розрізняє, що у даному проєкті певний підхід недоречний.

Юридичні та етичні питання. Код, що згенерований ШІ, може базуватися на фрагментах навчальних даних, що бере з open-source. Це породжує дискусії: чи не є такий код похідним твором та чи не порушує він ліцензії? Вже був прецедент колективного позову проти GitHub Copilot щодо можливих порушень open-source ліцензій. Окрім того, постає питання відповідальності: якщо помилка у згенерованому ШІ коді призвела до збою або витоку даних, хто винен – розробник чи ШІ? Через цю правову невизначеність деякі компанії обережно ставляться до впровадження генерації коду без жорстких перевірок.

Перспективи. Генерація коду еволюціонує від простого автозаповнення до комплексних ШІ-агентів, здатних виконувати більшу частину циклу розробки. Очікується поява команд взаємодіючих спеціалізованих моделей: один агент генерує код, інший одразу його тестує, третій перевіряє архітектурні вимоги, четвертий готує деплой. Такі multi-agent системи зможуть враховувати ширший контекст та підтримувати цілісний підхід. Наприклад, зважати на суміжні модулі, вимоги безпеки, стандарти стилю. У майбутньому ШІ, ймовірно, зможе з мінімальними підказками людини створювати цілі підсистеми. Проте роль людини зміститься у бік постановки завдань та перевірки. Розробник ставатиме наставником ШІ, який спрямовує його та контролює якість результату. Правильне впровадження політик перевірки та валідації ШІ-виводу залишатиметься критично важливим, щоб отримати вигоду (прискорення розробки) без втрати надійності та контролю над кодом.

2.2.3 Тестування та QA

ШІ-помічники у сфері тестування допомагають автоматизувати та оптимізувати перевірку якості ПЗ. Вони можуть генерувати тестові сценарії на основі опису функціоналу, автоматично писати тести для заданого коду та, навіть, знаходити дефекти за допомогою аналізу виконання програми. Сучасні інструменти вже дозволяють описати тест кейс людською мовою, а ШІ побудує код автоматичного тесту (UI-тесту, API-тесту тощо). Наприклад, QA-інженер може вказати: “Перевір, що користувач може створити плейлист та додати 10 пісень”, а ШІ-засіб сформує тест, що проходить весь цей сценарій у браузері або через API. Також ШІ застосовується для аналізу логів та виявлення аномалій у поведінці системи, щоб знаходити збої ще до того, як вони проявляться критично[18].

Частота використання. У щоденній практиці залучення ШІ до тестування поки не настільки рутинне, як автодоповнення за написання коду, але швидко набирає обертів. Багато команд експериментують з генерацією unit-тестів: розробник може раз чи два за робочий день попросити ШІ написати тест для нового методу. У сфері QA з’являються платформи, де ШІ-тестер працює 24/7, проганяючи тест-рунги та повідомляючи про знайдені баги. Очікується, що протягом найближчих років ШІ-інструменти стануть звичними у конвеєрі CI/CD стосовно автоматичного розширення тестового покриття.

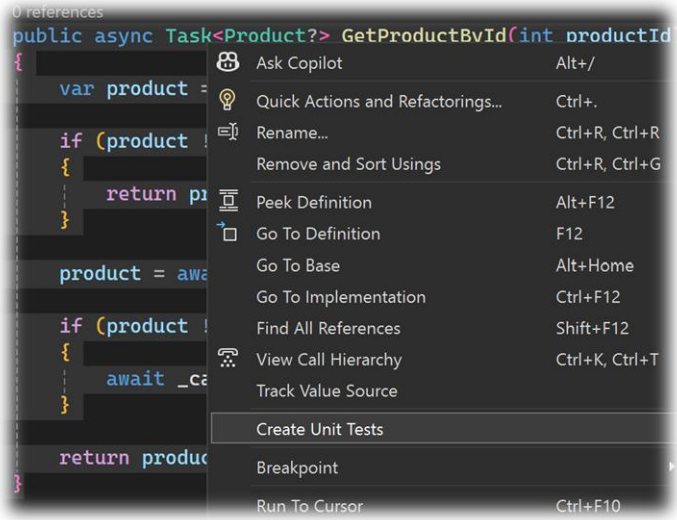

Рисунок 2.15 – Створення тестів

Переваги ШІ.

Швидка генерація тестів(рис. 2.15). ШІ здатен автоматично створити великий набір тест-кейсів, охопивши різні сценарії використання. Це знімає навантаження з тестувальників у плані написання рутинних тестів та дозволяє зосередитися на складніших, творчих аспектах QA. ШІ також полегшує автотестування для нетехнічних спеціалістів: людина без знання мови програмування може задати сценарій, а ШІ напише код тесту. Так відбувається своєрідна демократизація тестування.

Розширення покриття та продуктивність. ШІ-тулінги можуть генерувати тести для різних платформ (веб, мобільні, десктоп) та різними мовами одночасно. Вони працюють безперервно, 24/7, тому регресійне тестування може виконуватися значно частіше та швидше. Це призводить до вищої якості: менше багів прослизає у продакшн завдяки проактивному виявленню проблем. За рахунок predictive testing та виявлення аномалій команда витрачає менше часу на виправлення помилок й більше – на розробку нових фіч.

Автоматизація аналізу та підтримки тестів. Сучасні ШІ-системи не тільки виконують тести, а й автоматично аналізують результати. У разі падіння тестів вони можуть вказати ймовірну причину та навіть запропонувати виправлення коду. Деякі інструменти забезпечують автопідтримку: якщо змінилось UI або API, ШІ оновить тести відповідно, зменшуючи трудомісткість підтримки тестового набору. Таким чином, ШІ допомагає уникнути ситуації, коли тести застарівають та їх треба масово переписувати вручну.

Недоліки та обмеження.

Галюцинації ШІ та недовіра. ШІ може згенерувати тест, що виглядає правдоподібно, але насправді перевіряє невірну логіку або пропускає критичний випадок. Такі галюцинації призводять до помилкових результатів тестування. Наприклад, ШІ-тест може завжди “проходити” просто тому, що некоректно перевіряє умови. Через це повна довіра до автогенерованих тестів зараз недоречна та їх треба валідувати вручну.

Відсутність розуміння бізнес-логіки. ШІ оперує кодом та даними, але не знає, що важливо для бізнесу. Він не здогадається сам щодо нефункціональних вимог або про те, які кейси є пріоритетними. Без вірно поставлених вимог ШІ може надмірно протестувати тривіальні сценарії та проґавити рідкісний, але критичний випадок. Виконавець з QA все одно має спрямовувати процес: пріоритизувати, що тестувати у першу чергу.

Ризики безпеки та конфіденційності. З метою ефективного тестування ШІ потрібні великі обсяги даних (логи, база помилок тощо). Є ризик, що при обробці таких даних модель розкриє конфіденційну інформацію або самі дані потраплять до сторонніх сервісів (якщо ШІ-хмарний). Також існує загроза poisoning-атак: зловмисник може підсунути у тренувальні дані шкідливі патерни, щоб ШІ пропускав вразливості або не помічав певних багів. Тому інтеграція ШІ в QA вимагає додаткових заходів безпеки та перевірки надійності виходу.

Перспективи. Майбутнє QA тісно пов’язане з ШІ. Наступна хвиля інструментів зосередиться на повному циклі тестування та налагоджувати за допомогою ШІ. Очікується, що ШІ зможе самостійно генерувати вичерпні тестові набори, підбираючи навіть нетривіальні edge cases, та автоматично виправляти знайдені дефекти. В ідеалі це приведе до самокоригувального ПЗ: ШІ-відповідатиме не лише за написання коду, а й за його перевірку та виправлення. Вже зараз деякі команди експериментують зі сценаріями, де ШІ-агенти генерують код та паралельно інші агенти генерують до нього тести, формуючи замкнений цикл перевірки. У перспективі 5–10 років роль людини у тестуванні зміститься до встановлення стратегій та перевірки критичних випадків, тоді як левову частку

рутинних перевірок виконуватиме ШІ. Це відкриває можливість значно прискорити release cycle без втрати якості, але вимагатиме нових підходів до верифікації дій ШІ та довіри до нього з боку команд.

2.2.4 Рефакторинг коду

Рефакторинг – це переписування існуючого коду з метою підвищити його якість без зміни зовнішньої функціональності. ШІ у цій сфері використовується з метою автоматизації трудомістких та типових операцій рефакторингу. Сучасні ШІ-асистенти можуть проаналізувати великі кодові бази та запропонувати покращення. Перейменувати змінні всього проєкту для єдності стилю, виокремити дубльовану логіку в окрему функцію, видалити не активний код або зайві залежності, додати відсутні коментарі тощо. Все це здійснюється зберігаючи поведінку програми. Особливо актуальним є ШІ-рефакторинг щодо великих легасі-проєктів, де накопичений технічний борг.

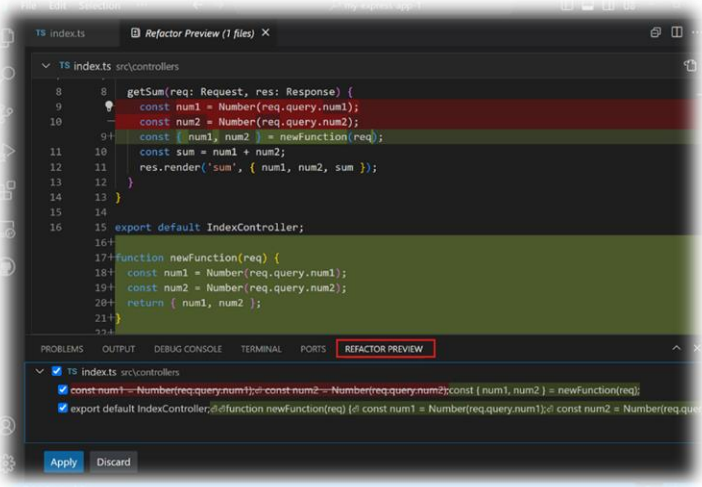

Рисунок 2.16 – Режим рефакторинг

Частота використання. Невеликі рефакторинги(рис. 2.16) – частина щоденної роботи розробника (наприклад, перейменувати метод, трохи спростити цикл). Такі дії зараз все частіше виконуються за допомогою ШІ-помічників прямо в IDE. Великі ж рефакторинги (переструктурування модулів, масова чистка коду) проводяться періодично. Тут ШІ здатен зменшити час з тижнів до днів. Команди, що вже впровадили ШІ для рефакторингу, відзначають прискорення код-рев'ю та значне зниження багів регресії, тож можна очікувати, що такі інструменти стануть стандартом у найближчі роки.

Переваги ШІ.

Значна економія часу. Те, що раніше вимагало монотонної роботи (пройтися сотнями файлів та щось виправити), тепер автоматизовано. ШІ може за лічені хвилини внести зміни у тисячі місць: наприклад, перейменувати функцію у всіх модулях чи замінити застарілий API на новий по всьому проєкту. Це скорочує тривалість рефакторингових спринтів та дозволяє частіше покращувати код невеликими ітераціями, а не накопичувати борги[19].

Якісніші зміни, менше помилок. ШІ під час рефакторингу аналізує абстрактне синтаксичне дерево коду та розуміє зв’язки між компонентами. На відміну від примітивного пошуку-замін, така обізнаність дозволяє уникати типових помилок. Наприклад, ШІ знає про область видимості змінної та не допустить перейменування, що зламає код. Автоматизація знижує вплив людського фактору: інструмент не забуде виправити щось в одному з файлів. У підсумку код стає більш однорідним та відповідає стилю проєкту.

Зниження технічного боргу та краща підтримуваність. Регулярний рефакторинг з допомогою ШІ підтримує актуалізацію кодової бази. Інструмент може постійно стежити за метриками якості та пропонувати покращення там, де є великі алгоритми дій або накопичення складності. Це запобігає ситуації, коли проєкт стає настільки громіздким та заплутаним, що нові фічі додавати дуже дорого. Крім того, ШІ здатен додавати докстрінги, коментарі та навіть генерувати юніт-тести під час рефакторингу, що покращує розуміння коду командою та його надійність.

Недоліки та обмеження.

Необхідність людського контролю. Попри розумність, ШІ все ще може припуститися помилки з рефакторингу. Наприклад, неправильного припущення про те, як використовується та чи інша змінна. Якщо автоматично прийняти всі зміни, є ризик непомітно змінити поведінку програми. Тому обов’язковим кроком є код-рев’ю ШІ-змін людиною: перегляд дифів, запуск тестів. Це трохи зменшує виграш у часі, але наразі без цього не можна, особливо у критичних частинах системи.

Обмеження на масштаб та контекст. Деякі ШІ-інструменти мають обмежене контекстне вікно й можуть не охопити весь проєкт цілком. У дуже великих репозиторіях (сотні тисяч рядків) доводиться рефакторити поетапно. Також моделі можуть погано знати внутрішні бізнес-вимоги чи неявні конвенції конкретної команди, тому певні рішення їм треба підказувати явно.

Вартість та впровадження. Потужні ШІ для рефакторингу (з великою контекстною пам’яттю) можуть бути платними або вимагати значних ресурсів. Їх інтеграція в існуючі процеси теж потребує зусиль. Необхідно налаштувати права доступу до коду, вирішити питання безпеки (бо надаємо код зовнішній моделі) та навчити команду правильно їх використовувати. Для деяких проєктів зі суворими вимогами (наприклад, вбудовані системи з сертифікацією) автоматичний рефакторинг може бути непридатним через вимоги до верифікації змін.

Перспективи. Майбутнє автоматизованого рефакторингу виглядає багатообіцяюче. З розвитком ШІ можна очікувати появи ще більш універсальних інструментів, що враховуватимуть не лише код, а й архітектурний дизайн та патерни проєктування. Наприклад, ШІ зможе рекомендувати перехід на іншу архітектуру (подрібнення моноліту на сервіси) або застосування певного патерну, з аналізом проблемних місць. Також прогнозується тісна інтеграція з системами контролю версій і CI: рефакторинги можуть виконуватися напівавтоматично при злитті гілок або перед релізом, підтримуючи кодову базу чистою постійно. З ростом обчислювальної потужності ШІ зможе розуміти й утримувати у пам’яті

весь код великого проєкту одразу, що забезпечить дуже контекстно-залежні та точні покращення. У перспективі з’явиться повністю автономний рефакторинг: система, що сама виявляє деградацію кодової бази та пропонує pull request з виправленнями. Проте роль розробника залишатиметься важливою: встановлення правил (код-стайл, допуски складності) та схвалення значних змін. Загалом, ШІ-рефакторинг обіцяє зробити еволюцію коду безперервним процесом, мінімізуючи накопичення технічного боргу та підтримуючи високу якість проєктів у довгостроковій перспективі.

2.2.5 Документація коду

Документація коду може набувати різних форм: генерування документаційних коментарів (docstrings) до функцій та класів, складання README-файлів, опис API-ендпоїнтів, створення діаграм чи специфікацій на основі коду. Інструменти на базі ШІ аналізують вихідний код та генерують людською мовою пояснення, що робить той чи інший модуль. Наприклад, ШІ може пройтися функціями та додати до кожної короткий опис параметрів й результату, формуючи базову документацію API. Деякі рішення йдуть далі: генерують цілі веб-сайти документації, UML-діаграми залежностей, або ж відповідають на питання розробників щодо коду (як такий чатбот-документовод).

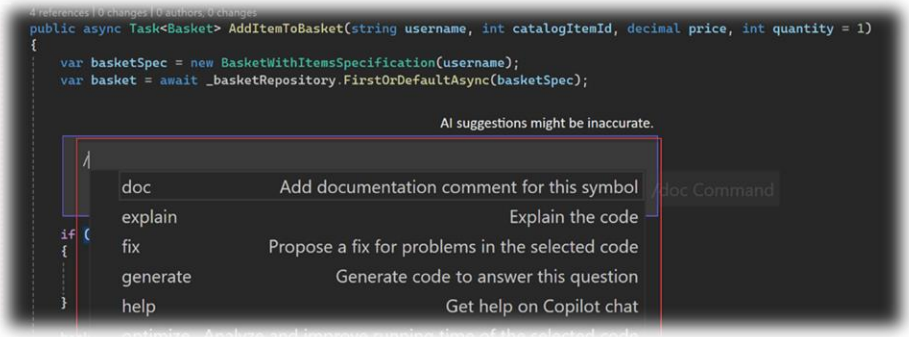

Рисунок 2.17 – Контекстне вікно функцій пояснення

Частота використання. Автодокументування особливо корисне у великих проєктах, де вручну підтримувати документацію досить складно(рис. 2.17). У повсякденній роботі розробники все частіше користуються ШІ з метою швидкого додавання коментарів. Наприклад, натиснули гарячу клавішу, а модель згенерувала коментар до нової функції. Це може відбуватися постійно при написанні нового коду. Формування ж більш розгорнутої документації (як README чи wiki-статті) може виконуватися ШІ-асистентом під час релізів або спринтів. Команди, що впровадили такі інструменти, відзначають суттєве зменшення складності підтримки документації, адже ШІ може автоматично оновлювати опис після змін у коді[20].

Переваги ШІ.

Спрощення процесу документування. Розробникам зазвичай не подобається писати документацію бо на це часто не вистачає часу. ШІ значно прискорює цю

задачу: він зчитує код й одразу пропонує пояснення природною мовою. Це особливо корисно стосовно стандартних CRUD-методів чи тривіальних класів. ШІ сам додасть коментарі щодо їх призначення, параметри тощо. Таким чином, базовий рівень документації є завжди, що полегшує включення новачків у проєкт.

Актуальність та безперервне оновлення. Інтеграція ШІ з репозиторієм коду (через CI/CD) дозволяє автоматично оновлювати документацію при змінах. Наприклад, якщо сигнатура функції змінилась, ШІ-помічник зможе оновити її опис. Це вирішує давню проблему застарілої неправдивої документації. Як наслідок, документація стає живою та синхронізованою з кодом на постійній основі.

Полегшення розуміння та підтримки коду. Новому члену команди легше розібратися у коді, коли навіть без читання вихідних текстів він може отримати згенероване пояснення. ШІ може сформувати оглядові матеріали, витягти й описати взаємозв'язки компонентів. До того ж, це корисно не лише стосовно людей: машиночитабельна документація (наприклад, в форматі OpenAPI) може теж генеруватися ШІ на основі коду, що спрощує інтеграції між сервісами. Загалом, знижується поріг розуміння системи як для розробників, так і для інших команд (дизайнерів, тестувальників), коли документація повніша.

Недоліки та обмеження.

Поверховість та неточності. ШІ генерує документацію, що ґрунтується на коді, але не завжди розуміє глибинні наміри. Складні алгоритми чи бізнес-логіка можуть бути для нього чорною скринькою, а пояснення вийде неповним або неточним. Також ШІ може припускати, що код робить X, тоді як насправді у певних випадках він робить Y. Такі нюанси часто потребують втручання людини.

Відсутність контексту бізнес-вимог. Документація від ШІ, як правило, не охоплює вимоги або рішення, прийняті на рівні дизайну системи, якщо вони не відображені прямо у коді. Такі речі повинні доповнюватися розробниками вручну. Крім того, ШІ не знає, які аспекти системи цікавлять користувачів документації (скажімо, приклади використання класу у реальному сценарії), без підказки він цього не надасть.

Потреба верифікації. Як й з генерованим кодом, генеровану документацію треба переглядати. Модель може сформулювати щось граматично красиве, але неправдиве. Особливо обережним слід бути з технічною точністю: неправильний опис параметра або формату даних може ввести в оману інших розробників. Тому треба переглянути згенеровані описи, виправити помилки та додати бізнес-контекст вручну там, де це необхідно.

Перспективи. ШІ у документуванні коду має великий потенціал. У майбутньому можливий перехід від статичних текстових документацій до інтерактивних ШІ-асистентів документації. Замість читати довгі мануали, розробник зможе спитати у чатбота: "Поясни, як модуль X обробляє транзакції" й отримати актуальну відповідь, згенеровану на основі останньої версії коду. Такі системи вже зароджуються і, ймовірно, стануть звичним інструментом. Також ШІ зможе дедалі краще пояснювати складні алгоритми (у міру вдосконалення моделей) та навіть візуалізувати їхню роботу. Перспективним є поєднання ШІ-

документування з системами навчання: новачок, читаючи документацію, зможе одразу ставити питання ШІ й отримувати додаткові роз'яснення або приклади коду. Документація більше не писатиметься вручну, а генеруватиметься динамічно, причому настільки точно, наскільки якісним буде код та коментарі. Роль людини – задавати високорівневі описи (наприклад, бізнес-процеси), а все низькорівневе зможе робити ШІ. Це зніме одну з хронічних проблем індустрії – нестачу актуальної документації. Підвищить прозорість й якість програмних продуктів.

2.2.6 Аналіз коду та Code Review

Аналіз коду та code review – це перевірка програмного коду з метою знайти помилки, вразливості, відхилення від стандартів та можливості поліпшення(рис. 2.18). ШІ-помічники у цьому напрямку виступають як автоматизовані рецензенти коду. Вони можуть сканувати патчі або цілі репозиторії та виявляти проблеми: від банальних помилок (незакрита дужка, змінна не використовується) до більш складних (потенційні null pointer, SQL-ін'єкції, неоптимальні алгоритми). Деякі ШІ-інструменти інтегруються у процес pull request: коли розробник відкриває PR, бот на базі ШІ залишає коментарі з зауваженнями та порадами. Також ШІ здатен робити статичний аналіз: шукати code smell'и, порушення стилю, можливі баги та навіть пропонувати готові виправлення.

Рисунок 2.18 – Аналіз коду Copilote

Частота використання. Автоматизовані code review стають все популярнішими, особливо у великих командах з насиченим потоком pull request'ів. На практиці ШІ може перевіряти кожен PR миттєво після його створення. Це означає, що щодня, при кожному злитті, код проходить й через ШІ. Крім того, розробники можуть самостійно запускати ШІ-аналіз у своїх IDE перед комітом, щоб зловити баги на місці. За даними опитувань, близько 60% розробників уже застосовують ШІ для рев'ю коду й ця частка швидко зростає, адже вигода в економії часу рев'юерів дуже приваблива[21].

Переваги ШІ.

Швидкість й ефективність. ШІ перевіряє код миттєво, значно прискорюючи цикл рев'ю. Він автоматизує рутинні перевірки (стиль, дрібні помилки) і тим самим зменшує навантаження на людських рев'юерів. Машина не втомлюється та

не пропустить очевидного через неуважність: кожна кома і кожен null-чек будуть перевірені.

Послідовність та неупередженість. На відміну від людей, ШІ завжди застосовує правила однаково до всього коду. Це виключає фактор суб'єктивності: дві різні команди отримають однакові зауваження на однаковий код. Також ШІ-рев'юер не упереджений. Для нього не існує автора коду, він критикує лише за діло, що може покращити атмосферу у команді (менше особистого у зауваженнях).

Знаходження прихованих дефектів. ШІ здатний виявити тонкі баги, які людина може прогледіти, особливо у великому діффі. Наприклад, він може прослідкувати, що змінна використовується у багатопотоковому середовищі без належної синхронізації, або що певна гілка if ніколи не виконається. Такі речі алгоритми статичного аналізу на базі ШІ знаходять дуже добре. Це підвищує загальну якість і безпеку продукту.

Освітній момент. Відносно молодих розробників ШІ-рев'юер виступає як терплячий наставник, що може хоч десять разів пояснювати одну й ту ж помилку. Кожне зауваження – це зворотній зв'язок. ШІ може навіть містити пояснення, чому так робити не слід й як краще. За рахунок цього новачки швидше вчаться стандартам та патернам, а досвідчені інженери можуть зекономити час на менторстві.

Недоліки та обмеження.

Нестача контексту та хибні спрацьовування. ШІ не має повного розуміння бізнес-контексту та задуму автора коду. Через це можливі як false positives – позначення проблеми там, де насправді все нормально, – так і false negatives – пропуск реальних багів, якщо вони нетипові. Наприклад, ШІ може ругати за копіювання коду, не знаючи, що це свідомо дубльований фрагмент з міркувань продуктивності. Тому сліпо покладатися на всі його зауваження не можна. Їх треба фільтрувати та враховувати специфіку проєкту.

Шум та перевантаження зауваженнями. Іноді ШІ-рев'ю може згенерувати дуже багато коментарів, частина з яких несуттєві. Це створює шум, в якому важливі речі губляться. Розробники можуть почати ігнорувати такі автоматичні зауваження, якщо їх занадто багато або якщо вони дріб'язкові. Налаштування правил та фільтрів під конкретний проеєкт тут обов'язкове, інакше якість рев'ю може навіть постраждати.

Ризик залежності від ШІ та послаблення навичок. Якщо команда занадто покладається на автоматичне рев'ю, розробники можуть менше уваги приділяти самостійній перевірці коду. Критичне мислення має зберігатися: ШІ – це інструмент, а не остання інстанція. Без аналізу людиною архітектури, відповідності бізнес-вимогам та розуміння коду не обійтись. Найкращий результат досягається, коли ШІ відловлює дрібниці, а люди фокусуються на високорівневих моментах (вірність алгоритму, дизайн, інтеграція у систему).

Перспективи. Вже зараз спостерігається тенденція до повної інтеграції ШІ у процес рев'ю. Майбутні системи навчаться враховувати історію проєкту, стилістичні вподобання конкретної команди, можливо, навіть відстежуватимуть

виконання коду, щоб краще розуміти його контекст. Комбінування кількох моделей – інша ймовірна перспектива. Одна спеціалізована на безпеці, інша на оптимізації, ще інша на відповідності стилю. Разом вони даватимуть більш збалансований та глибокий фідбек. Зможе автоматично виправляти деякі зауваження: наприклад, знайшов проблему – одразу згенерував патч та запропонував його до мерджу (деякі інструменти вже це вміють). Можливо, з'являться повністю автоматичні рев'ю для тривіальних змін, де людське втручання не потрібне (наприклад, оновлення залежностей може цілком рев'ювати ШІ). Водночас зростатиме роль глобального моніторингу якості. ШІ зможе будувати метрики (кількість виявлених проблем, якість репозиторію) і підказувати, на що звернути увагу командам. Загалом, напрям ШІ-код-рев'ю має всі шанси перетворитися з допоміжного інструменту на обов'язковий елемент конвеєра розробки, що суттєво підвищує надійність ПЗ без збільшення витрат часу розробників.

2.2.7 Інтеграція з API та бекендом

У цьому напрямку ШІ допомагає розробникам з інтеграції різних систем – наприклад, підключенні до сторонніх API, розробці бекенд-логіки з метою обробки запитів, генерації кодових фрагментів щодо роботи з базами даних. Типова задача: потрібно швидко навчитися працювати з новим веб-сервісом (скажімо, API платіжної системи). Замість читати довгу документацію, програміст може запитати в ШІ приклад коду для виклику потрібного методу та отримати готовий скрипт із врахуванням автентифікації, форматів даних тощо. ШІ-асистенти по API здатні навіть самі згенерувати SDK або клієнтську бібліотеку на основі специфікації (OpenAPI/Swagger). Вони прочитають опис кінцевих точок та створять код класів, методів для звернення до цих endpoint'ів. Для бекенду ШІ може згенерувати шаблон серверного застосунку: налаштувати роутинг, контролери, підключення до бази за високорівневим описом вимог.

Частота використання. ШІ все частіше використовують як перекладача між людиною і складними системами. При розробці інтеграцій розробник може декілька разів на день звернутися до ШІ: "Як створити запит на такий-то REST endpoint", "Наведи приклад GraphQL-запиту стосовно отримання списку користувачів" тощо. Це економить час, тож увійшло у звичку. Генерація ж цілих SDK поки менш поширена, але набуває популярності. При оновленні API можна знову прогнати ШІ-генератор та він випустить оновлену бібліотеку клієнта. У бекенд-розробці ШІ-асистенти вбудовані у деякі фреймворки, допомагаючи налаштувати інфраструктурний код (наприклад, YAML-конфігурації, Docker-файли). Ці речі тепер часто роблять з підказкою ШІ.

Переваги ШІ.

Прискорене навчання нових API. ШІ може миттєво надати робочий приклад коду практично щодо будь-якого популярного API. Це усуває довгі години читання документації та пошуку прикладів на форумах. Як результат, інтеграція починається одразу з працюючого шаблону, що можна тестувати та допрацьовувати. Розробники швидше освоюють нові сервіси, адже ШІ фактично вчить їх, як з цим працювати, через код.

Автоматизація рутини бекенду. Налаштування типового бекенд-проєкту багато у чому шаблонне(рис. 2.19). ШІ здатен згенерувати значну частину цього шаблонного коду, зменшуючи кількість ручного boilerplate. Наприклад, він напише SQL-запити чи ORM-моделі за схемою БД, або налаштує авторизаційні фільтри. Це підвищує продуктивність бекенд-розробників та знижує ризик помилок у конфігурації[22].

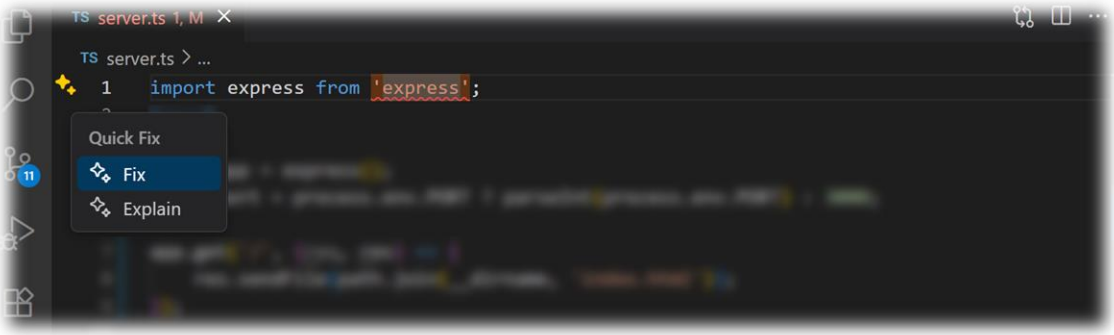

Рисунок 2.19 – Використання команди встановлення бібліотек

Зменшення помилок інтеграції. ШІ, що навчений на багатьох прикладах, часто знає про типові підводні камені інтеграцій. Він може попередити про необхідність ретраїв при HTTP 429, показати правильний формат OAuth-токена, нагадати щодо обробки помилок мережі. У результаті інтеграційний код виходить більш надійним з першого разу. Також ШІ допомагає відлагодити інтеграцію: можна спитати, чому певний виклик API не працює та отримати підказку (скажімо, “ви забули поле X у тілі запиту”). Це економить безліч часу, який інакше пішов би на дебаг та з’ясування тонкощів.

Недоліки та обмеження.

Застарілі або помилкові знання. Документація API часто оновлюється, з’являються нові версії. Модель ШІ може посилатися на знання, отримані з інтернету, які вже неактуальні. Вона може порадити параметр, якого більше немає, або старий endpoint. Тому є ризик отримати код, що не працює з поточною версією сервісу. Завжди варто звірятися з офіційною документацією, особливо для критичних інтеграцій.

Безпека та секретні дані. ШІ не завжди розуміє, що є конфіденційним. Якщо надавати йому фрагменти ключів чи токенів для налагодження, є ризик витоку секретів (якщо це зовнішній сервіс). Також модель може не усвідомлювати безпекових наслідків своїх порад: наприклад, порадить зробити запит без перевірки сертифіката чи продемонструє простий приклад без врахування авторизації, а розробник, поспішачи, може так і залишити. Тому безпековий аудит інтеграційного коду все одно треба робити.

Нестандартні сценарії. Якщо інтеграція складна або дуже специфічна, ШІ може виявитися малокорисним. Він добре працює на типових завданнях, але якщо треба оптимізувати продуктивність під високе навантаження чи здійснити нестандартну послідовність взаємодій з API, модель може не знайти готового

рішення. У найгіршому разі вона згенерує код, що намагатиметься робити щось подібне, але через відсутність точного аналога – помилиться. У таких випадках все одно доводиться глибоко вникати у документацію та придумувати рішення самотужки.

Перспективи. У майбутньому ШІ може стати універсальним перекладачем між різними системами. З розвитком техніки “ШІ-агенти-інтегратори” зможуть самі читати документацію API та встановлювати з’єднання між сервісами. Наприклад, ви просто сформулюєте завдання: “Зв’яжи мій сайт з платіжною системою X для обробки підписок”, а ШІ згенерує весь необхідний код, включно з обробкою помилок та тестовими викликами. Вже зараз є експерименти, де ШІ працює поверх OpenAPI-специфікацій та створює повноцінні SDK майже без втручання людини. Ця технологія розвиватиметься, усуваючи людську рутину при інтеграціях. Також очікується тісніша інтеграція ШІ з середовищами розробки. IDE зможе автоматично підтягувати необхідні залежності, налаштовувати ключі доступу (звісно, за участю людини для безпеки). Тобто, багато налаштувань виконається у середені розумним асистентом. У сфері бекенду ШІ, ймовірно, буде брати участь й в архітектурних рішеннях: підказуватиме, які сервіси краще використати (наприклад, рекомендуватиме чергу повідомлень замість прямого виклику, якщо бачить, що це надійніше). Загалом, тренд такий, що інтеграційний код ставатиме все більш автоматизованим, а розробники зможуть приділяти більше уваги бізнес-логіці, ніж низькорівневим деталям з’єднання систем.

2.2.8 Frontend / UI логіка

ШІ-інструменти у фронтенд-розробці допомагають створювати користувацькі інтерфейси швидше і з меншими помилками. Вони можуть генерувати код компонентів за текстовим описом або дизайном, пропонувати CSS-стилі, автоматично додавати обробки подій. Наприклад, розробник може написати в коментарі: “// компонент кнопки зі стилем Material Design”, а ШІ допише JSX/HTML та CSS для цієї кнопки. Інший випадок: ШІ-конвертер бере макет з Figma та генерує відповідний код React-компонентів. Також ШІ може підказувати у фронтенді оптимальні способи маніпуляції DOM, створювати анімації, перевіряти доступність (accessibility), тобто виступати як асистент в UI-логіці і дизайні одночасно.

Частота використання. У повсякденній роботі фронтендера ШІ все активніше використовується. При написанні JSX/HTML шаблонів автодоповнення (підказати структуру елементів) економить купу часу – це відбувається постійно. CSS також доповнюють: достатньо почати писати клас, а ШІ запропонує весь блок стилів. З більш просунутого: якщо треба реалізувати типову взаємодію (скажімо, нескінченний скролінг списку), то багато хто звертається до ШІ по приклад коду. Це трапляється регулярно, коли розробник стикається з новим для себе патерном. Поки що генерація цілих інтерфейсів із дизайнів менш поширена, але інструменти активно розвиваються. Вже з’являються першопрохідці, що довіряють ШІ створювати чернетки UI, а потім їх доводять до ладу.

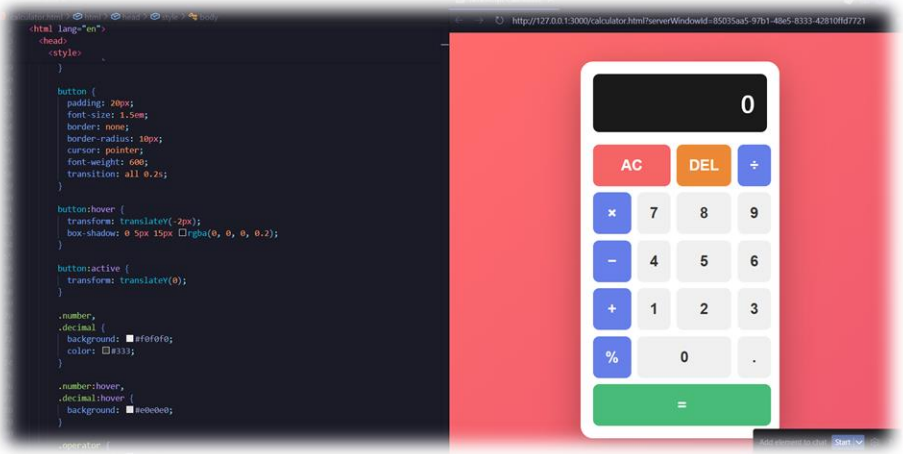

Рисунок 2.20 – Використання ШІ для фронтенду

Переваги ШІ.

Швидке створення компонентів. ШІ може згенерувати каркас нового компонента за лічені секунди, враховуючи стандартні практики. Він додає необхідні імпорти, властивості, стан (state). Розробнику залишається лише підправити деталі. Це особливо корисно для багаторазових елементів UI: форм, карток, меню тощо. ШІ вставляє готовий шаблон, що потім налаштовується. За рахунок цього фронтенд розробляється значно швидше, менше рутини вручну.

Допомога з CSS та дизайном(рис. 2.20). ШІ чудово справляється з генеруванням стилів за описом. Наприклад, можна сказати: “Фон синій, скруглені краї, тінь при наведенні” та отримати готовий CSS-клас. Це економить час й зусилля на підбір вірних CSS-властивостей. Модель також знає типові кросбраузерні проблеми та може запропонувати рішення, сумісні з усіма оглядачами. Крім того, ШІ підкаже щодо доступності. Наприклад, може згенерувати текст для aria-label чи порекомендувати достатній контраст кольорів для читабельності[21].

Тестування та відлагодження UI. Деякі ШІ-асистенти здатні аналізувати помилки в консолі браузера або підказувати, чому той чи інший елемент не відображається. Вони швидко вказують на можливі причини. Також ШІ може автоматично створити тести для компонентів (unit-тести чи end-to-end) за описом поведінки, що підвищує надійність фронтенду.

Недоліки та обмеження.

Ризик шаблонності та загальних рішень. ШІ генерує UI код, але він часто узагальнений і не завжди ідеально підходить під конкретні вимоги. Наприклад, згенерований компонент може не враховувати специфічних випадків або не відповідати pixel-perfect дизайну. Модель може запропонувати середньостатистичне рішення, яке доведеться дороблять під н’юанси проєкту. Дизайнери часто прагнуть унікальності, а ШІ схиляється до знайомих патернів – тут може виникати конфлікт між креативністю та шаблонністю.

Складні інтеракції та стан. Простий UI код ШІ генерує добре, але коли починається складна логіка взаємодії (багато станів, залежні випадаючі списки,

тощо), модель може заплутатися або видати неоптимальне рішення. Динамічну поведінку UI все ще краще продумує людина. ШІ може допустити, наприклад, зайві рендери або забути очистку ресурсів (event listeners), що спричинить баги. Такі моменти потребують ретельного рев'ю та тестування, якщо код згенерований, або навіть повної переробки в окремих випадках.

Пропуск бізнес-логіки. Фронтенд – це не лише красиві кнопки, а й валідація, обробка даних та виклики API. ШІ може зосередитися на візуальній частині та не знати ваших бізнес-правил. Наприклад, він зробить форму, але не додасть специфічну валідацію полів, що знає лише ваша команда. У результаті розробнику все одно треба вручну дописувати логіку. Отже, ШІ більше допомагає з візуалізацією, а не з суттєвою логікою, і це треба розуміти.

Перспективи. Generative UI – перспективний напрям. У найближчому майбутньому ймовірно побачимо інструменти, де дизайнерський макет напряму перетворюється на код з мінімальним втручанням розробника. ШІ зможе краще враховувати responsive design. Генерувати адаптивну верстку під різні екрани автоматично. Також можливе глибше поєднання ШІ з фреймворками: умовно кажучи, React або Angular матимуть вбудованого ШІ, що підказує оптимальний спосіб реалізації компонента саме для цього фреймворку, з урахуванням найновіших рекомендацій. Очікується, що ШІ братиме участь й на етапі дизайну UX: аналізуватиме, куди користувач клікає та пропонуватиме зміни у UI для покращення досвіду (тобто замикатиме цикл дизайн → код → аналіз використання → новий дизайн). Звичайно, фронтенд – сфера, де багато творчості, тому навряд чи ШІ витіснить тут людей. Швидше, станеться симбіоз: люди займаються загальною концепцією та нестандартними елементами, а ШІ забезпечує швидке втілення типових частин інтерфейсу. Це скоротить час розробки інтерфейсів та, можливо, дозволить випускати більш складні UI за той самий час, що зараз витрачається на прості.

2.2.9 DevOps / CI/CD

У сфері DevOps ШІ застосовується щодо автоматизації та оптимізації процесів розгортання, налаштування інфраструктури, моніторингу й реагування на інциденти. У рамках CI/CD-конвеєра ШІ може визначати, які тести запускати щодо конкретного коміту (аби не гаяти час на зайве), аналізувати реєстр збірки та підказувати причину збоїв, автоматично приймати рішення про roll-back або roll-forward у разі виявлення проблеми на продакшені. Також ШІ використовується з метою аналітики та прогнозування. Наприклад, на основі метрик використання ресурсів прогнозувати навантаження та масштабувати систему до того, як вона перевантажиться. Інший напрям – Ops, де ШІ-моделі сканують потоки логів та повідомлень моніторингу, щоб виявити аномалії або патерни, що передують аварії, та, навіть, автоматично виконати певні скрипти щодо самовідновлення системи.

Частота використання. Багато сучасних DevOps-платформ вже мають елементи ШІ. Кожен раз, коли ви робите deploy, може працювати ШІ-алгоритм оптимізації (наприклад, розумний вибір порядку тестів). У великих компаніях, де тисячі серверів, ШІ-інструменти моніторингу працюють постійно. Щоденно вони

генерують алерти чи рекомендації на основі даних. Рядові розробники також стикаються з ШІ у DevOps. Наприклад, при налаштуванні GitHub Actions або GitLab CI ШІ може підказати YAML-конфіг щодо потрібного процесу, що економить час. Тобто, інтеграція ШІ у DevOps(рис. 2.21) вже відбувається, хоч й поступово, але через високу віддачу (швидші релізи, менше інцидентів).

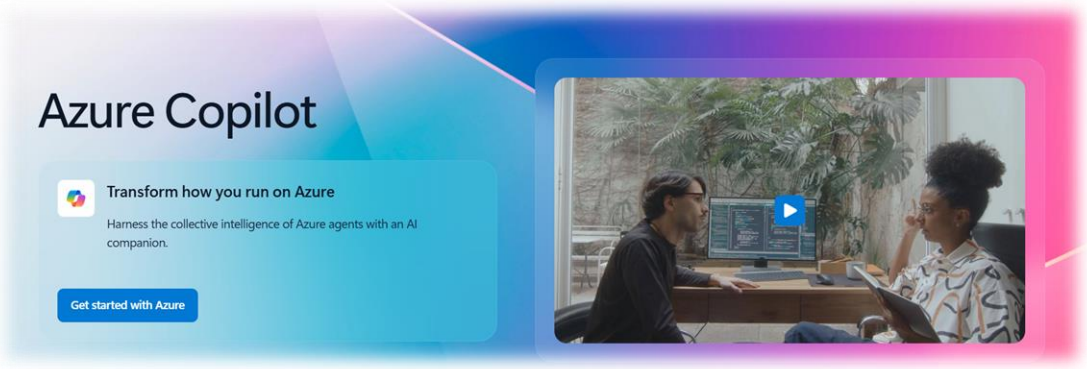

Рисунок 2.21 – Використання ШІ у Azure

Переваги ШІ.

Прискорення конвеєра CI/CD. Одним з головних плюсів є швидкість випуску релізів. ШІ здатен оптимізувати процес. Наприклад, запускати тільки релевантні тести щодо даного зміненого модуля, пропускати зайві кроки, паралелізувати завдання оптимально. Це зменшує час збірки та доставки коду у продакшн. Модель може аналізувати історичні дані тестів та вгадувати, де найімовірніше виникнуть проблеми, фокусуючись на них. Це підвищує якість випуску.

Підвищення якості та надійності. Завдяки predictive analytics ШІ може попередити аварію: реестр і метриками він побачить, що щось іде не так (наприклад, пам’ять витікає) та підніме тривогу ще до фатального збою. Також ШІ допомагає швидко знаходити першопричину проблем: коли переривається збірка коду, модель виділяє у величезній консолі саме ту помилку, що стала причиною, що економе години на пошук голки у копиці сіна. Менше багів доходять до продакшену, а якщо доходять, то швидше вирішуються.

Автоматизація і самовідновлення. ШІ дозволяє реалізувати концепцію self-healing systems. Коли трапляється відомий тип проблеми, система автоматично виконує кроки з її вирішення. Наприклад, якщо впав сервер, то ШІ-агент сам підніме його копію та перенаправить туди трафік. Якщо розгорнутий реліз дає сплеск помилок, то ШІ ініціює автоматичний rollback на попередню версію, не чекаючи ручного втручання. Це все значно зменшує downtime та вплив людського фактора.

Оптимізація ресурсів та витрат. Інтелектуальні алгоритми можуть аналізувати використання CPU, пам’яті, мережі та пропонувати оптимізації: зменшити кількість виділених контейнерів у нічний час та, навпаки, додати потужності перед очікуваним піком (наприклад, Black Friday для e-commerce).

Таким чином, краще управляються ресурси: ні простоїв марних, ні дефіциту. У кінцевому рахунку економить гроші компанії.

Недоліки та обмеження.

Якість даних та помилкові рішення. ШІ у DevOps дуже залежить від якості даних (логів, метрик), на яких він навчений. Якщо дані шумні або неповні, модель може робити неправильні висновки. Хибні спрацьовування – суттєва проблема. ШІ може помилково вирішити, що реліз поганий та згорнути його, хоча все гаразд, або навпаки, не помітити проблему. Довіра до ШІ-рішень будується з часом, а на початку команди можуть ставитися скептично, перевіряючи кожну дію ШІ, що нівелює виграш у швидкості[23].

Складність впровадження. Інтегрувати ШІ-інструменти у DevOps-ландшафт є нетривіальним завданням. Це потребує експертизи з ML, налаштування моделей під своє середовище, а іноді й закупівлі дорогих рішень. Не кожна організація має достатньо зрілу практику DevOps, щоб одразу додати туди ШІ-автоматизацію. Також legacy-системи можуть не мати добре структурованих даних щодо аналізу, що ускладнює роботу ШІ.

Лімітований інтелект. Нинішні системи – це все ж не люди. Вони можуть не врахувати якогось неформального чинника. Наприклад, ШІ вирішив, що цей мікросервіс треба перезапустити (бо метрики погані), але не знає, що причина у збільшенні трафіку через маркетингову акцію та краще спочатку збільшити ліміт з'єднань, а не перезапускати. Тобто повністю замінити досвідченого DevOps-інженера ШІ не може, він діє за шаблонами. Тому потрібен контроль та можливість втрутитися, інакше є ризик, що автоматизація наробить лиха не менше, ніж принесла користі.

Перспективи. Очікується, що DevOps як напрям найбільше виграє від ШІ та з часом трансформується у "Intelligent DevOps". Майбутні CI/CD-конвеєри будуть не просто виконувати скрипти, а навчатися з кожного пройденого релізу. Наприклад, якщо певний тест ніколи не падав за 1000 запусків, ШІ може пропонувати винести його в опціональні, щоб не гаяти час. Якщо ж якась комбінація модулів часто дає збій, то система підказуватиме щодо ризиків ще на етапі pull request. Концепція CA/CD (Continuous Adaptive Deployment) набирає обертів. Процес сам адаптується під обставини. З точки зору інфраструктури, самокеровані кластери можуть стати реальністю: ШІ-базовані оркестратори, що самі вирішують, де запускати новий сервіс, як перерозподілити навантаження, коли виконати технічне обслуговування. Інцидент-менеджмент теж зміниться: ШІ-асистенти, наприклад, ChatOps-ботів у Slack будуть першою лінією підтримки, що зможе відповісти на типові питання ("чому впав деплой?", "що означає цей алерт?") та виконати базові дії. Звичайно, люди й далі задаватимуть політики (як і що можна автоматизувати), але рутина (логі, алерти, перемикання середовищ) у значній мірі автоматизується. Це підвищить стійкість систем та швидкість виходу продуктів, оскільки релізи будуть випускатися швидше, а все інше зробить розумний механізм. DevOps-інженерам майбутнього потрібно буде більше розуміти ШІ-інструменти, щоб направляти їх та контролювати, але їхнє навантаження щодо рутини істотно спаде.

2.2.10 Навчання та демонстрація коду

ШІ-помічники відіграють новаторську роль у навчанні програмуванню та поясненні існуючого коду. Фактично, вони можуть виступати персональними репетиторами розробників. Пояснювати невідомі концепції, показувати приклади використання тієї чи іншої функції, допомагати розібратися у чужому коді. Якщо раніше, щоб навчитися новому API, треба було читати документацію та шукати статті, то зараз можна запитати в ШІ: “Як підключитися до бази даних у Python?” та отримати покроковий приклад. Так само ШІ здатен проаналізувати фрагмент коду та пояснити, що він робить. Це дуже корисно, коли потрібно розібратися у спадковому або просто чужому коді. Замість годин читання можна за хвилини отримати зрозуміле пояснення. Крім того, ШІ може генерувати приклади коду з метою навчання: “Покажи, як реалізувати патерн Спостерігач на Java”. Він надасть зразковий код, що можна вивчити.

Частота використання. Молоді програмісти та студенти активно використовують ШІ (типу ChatGPT) з метою навчання. Ставлять запитання, просять приклади, пояснення помилок та інше. За опитуваннями, новачки частіше за досвідчених звертаються до ШІ, коли чогось не знають, замість традиційного пошуку у Google. У професійній сфері, навіть, досвідчені інженери кілька разів на тиждень можуть попросити ШІ пояснити якийсь фрагмент коду (особливо, якщо це незнайома їм мова або рамківорк) чи швидко нагадати синтаксис/підхід. Під час code review теж буває, що побачивши складний шмат коду, рецензент може прогнати його через ШІ, щоб переконатися, що правильно розуміє логіку. Отже, роль ШІ як наставника вже закріпилася та з часом стане тільки помітнішою.

Переваги ШІ.

Індивідуалізоване навчання. ШІ-репетитор доступний 24/7 та підлаштовується під запит користувача. Він може пояснити одну й ту ж тему різними словами, навести більше прикладів, поки учень не зрозуміє. Це знімає психологічний бар’єр: можна не соромитися недотепних питань, ШІ не буде висміювати. Новачки відзначають, що з ШІ вони швидше проходять початковий етап, менше плутаються, бо завжди є де запитати.

Практичні приклади та найкращі практики. Замість абстрактних пояснень ШІ майже завжди дасть приклад коду. Відразу видно, як теорія застосовується на практиці. До того ж, ШІ, тренований на великій кодовій базі, часто пропонує best practices. Показує ідіоматичний стиль, прийнятий у спільноті, попереджає щодо частих помилок. Учень, копіюючи такий приклад, мимоволі вчиться хорошому стилю.

Прискорення вирішення проблем. При самостійному навчанні або розборі завдань часто трапляються тупікові ситуації – незрозуміла помилка, задача, яку не вдається вирішити. З ШІ ці тупіки долаються швидше: він підкаже, де помилка у коді, або наведе хід думок для вирішення. Так можна уникнути ситуації, коли новачок кидає навчання через фрустрацію. Для практикуючого програміста це означає економію часу при дебагу: ШІ відразу вказує, що, наприклад, переплутані типи або неправильний порядок викликів API та дає пояснення, чому це не працює.

Недоліки та обмеження.

Можливість отримати неправильні знання. ШІ може помилятися та початківець не завжди здатен це розпізнати. Є випадки, коли ШІ дає впевнене, але неправильне пояснення концепції, або код з помилками, який, на жаль, учень приймає за істину. Якщо не перевіряти та не підтверджувати отриману інформацію з інших джерел, можна вивчити щось невірно. Це небезпечно, бо виправити потім хибне розуміння складніше.

Відсутність глибини та структурованості. ШІ відповідає на конкретний запит, але не побудує вам повноцінну навчальну програму (принаймні поки що). Він не знає, які прогалини у ваших знаннях та не завжди подасть матеріал в оптимальному порядку. Викладач чи добре структурований курс поки незамінні з метою систематичного навчання. ШІ-репетитор може перескакувати деталі або, навпаки, розжовувати очевидне бо не відчуває студента.

Ризик зниження самостійності. Якщо надто часто звертатись до ШІ за рішеннями, можна упустити важливий навик: самостійно боротися з завданнями, гуглити, експериментувати. Вирішення проблем – ключова частина навчання програмування. Якщо ШІ завжди напоготові з відповіддю, учень може розлінитися та не розвинути достатньо свої навички пошуку й аналізу інформації. Тому важливо не зловживати підказками, а використовувати їх дозовано.

Перспективи. ШІ-технології навчання коду будуть розвиватися у напрямі більшого персоналізування й інтерактивності. Можлива поява повноцінних віртуальних менторів, які пам’ятають прогрес учня, знають його сильні та слабкі сторони та можуть планувати навчання. Вони зможуть давати практичні завдання, перевіряти код учня, пояснювати помилки та підбирати наступні теми згідно успіхів. Це буде як приватний викладач, доступний кожному. Також ШІ інтегрується в освітні платформи та книги: читач може відразу питати роз’яснення чи додаткові приклади та отримувати відповіді у реальному часі. Для професіоналів з’являться ШІ-системи типу “кодового Google” – коли вся документація та Q&A проєкту доступні через ШІ-помічника, що відповідає саме з урахуванням вашого процесу. Це зекономить час на пошук рішень в інтернеті.

Одночасно постане завдання навчити розробників вірно користуватися ШІ: критично оцінювати відповіді, перевіряти, не копіювати бездумно. Можливо, навчальні програми включатимуть курси з ефективного використання ШІ-інструментів у програмуванні. Зрештою, комбінація людина та ШІ у навчанні може дати найкращий результат: людський досвід та педагогіка плюс цілодобова доступність й енциклопедичність ШІ. Це зробить вивчення програмування більш доступним, а підвищення кваліфікації інженерів швидшим та безперервним процесом впродовж кар’єри.

Висновки

ШІ-помічники дедалі глибше інтегруються у всі аспекти роботи програмістів: від написання та перевірки коду до його розгортання й навчання новим навичкам. Правильне використання цих інструментів вже зараз дає приріст продуктивності та якості. За оцінками користувачів, прискорення розробки на 20–50% цілком досяжне. Однак кожен з напрямків має й свої обмеження: від

технічних невірностей до впливу на навички та командну роботу. Тому ключовим принципом має бути баланс. Найкращих результатів досягають команди, які трактують ШІ як розумного асистента, але остаточні рішення та відповідальність залишають за собою. У майбутньому ШІ-інструменти стануть ще прогресивнішими й автономнішими, а роль інженера еволюціонує до наставника та контролера ШІ. Усвідомлення сильних й слабких сторін ШІ у кожному напрямку допоможе отримати від цих технологій максимум користі. Прискорити рутинні завдання, зменшити кількість помилок та мінімізувати ризики щодо безпеки, якості та професійного росту розробників. Таким чином, програмісти й ШІ-помічники спільно зможуть досягти більшої ефективності та творчої свободи у створенні ПЗ.

Розглянувши різні способи застосування ШІ у програмуванні, можна зробити декілька узагальнень. По-перше, ШІ вже став цінним помічником розробників, але форма цього помічника може бути різною: від окремого веб-чатбота до вбудованого в IDE асистента.

Веб-чати (ChatGPT, Claude тощо) – ідеальні стосовно ознайомлення з можливостями ШІ, навчання та вирішення одиничних завдань або створення прототипів. Вони доступні будь-де, але потребують обережності з корпоративними даними;

Локальні агенти – вибір для тих, кому критично важлива безпека або автономність. Вони дозволяють компаніям використовувати ШІ без ризику витоку даних, адаптувати його під себе та навіть будувати цілі автономні workflow (як у випадку OpenDevin). Цей шлях вимагає більше зусиль на старті, але забезпечує повний контроль та потенційно нижчі операційні витрати у довгостроковій перспективі (немає щомісячних сплат за токени).

ШІ-плагіни в IDE – наразі найзбалансованіше рішення, що поєднує переваги ШІ зі звичним процесом розробки. Вони значно прискорюють рутинні завдання, зменшують кількість помилок та роблять процес написання коду більш комфортним. Єдине питання – вибір між хмарним чи локальним режимом цих плагінів, але, схоже, індустрія рухається до того, щоб дати користувачу такий вибір (як у випадку Copilot for Business, де обіцяно режим без збереження даних, або Tabnine, де є on-prem версія).

Розвиток даних напрямів полягає у їх поступовій конвергенції. Вже зараз спостерігається, що межі між категоріями розмиваються. Універсальна екосистема, де ШІ співпрацює з розробниками на рівних, вже на горизонті. Воно може бути як хмарним, так і локальним, але обов’язково інтегрованим зі всіма потрібними інструментами в одному просторі. Все це спрямоване на одну мету, а сасме зробити так, щоб ШІ став повноцінним учасником процесу розробки, а не окремим інструментом.

3 ОПТИМІЗАЦІЯ РОБОТИ АГЕНТА ГЕНЕРАЦІЇ КОДУ

3.1 Налаштування та оптимізація агента генерації коду

Кастомізація GitHub Copilot у середовищі Visual Studio Code становить комплекс програмних механізмів, спрямованих на модифікацію поведінки моделі у процесі генерації, рефакторингу та аналізу програмного коду. На відміну від стандартного режиму, де модель реагує окремо на кожен запит, механізми кастомізації дозволяють встановлювати довготривалі правила, визначати системні промпти, формувати поведінку спеціалізованих агентів, керувати контекстним оточенням, підключати зовнішні інструменти через протокол MCP та обирати використовувані мовні моделі. Сукупність цих можливостей формує системне середовище взаємодії між розробником, інструментами IDE та моделю ШІ, що забезпечує передбачуваність результатів, відповідність корпоративним стандартам розробки та підвищення ефективності інженерних процесів[24].

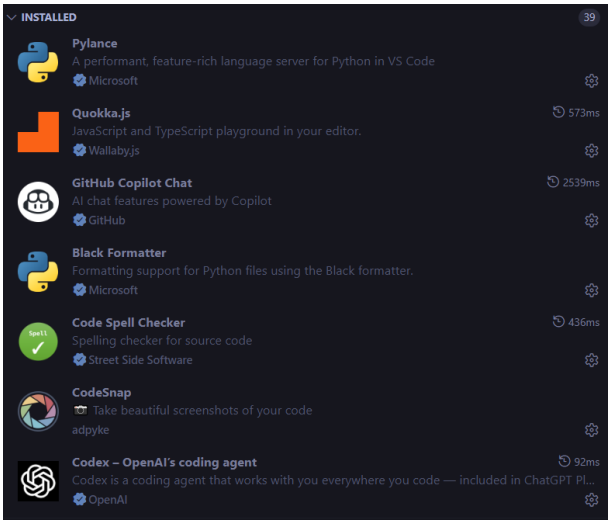

Рисунок 3.1 – Використання кастомізації IDE

Система кастомізації(рис. 3.1) дозволяє застосовувати політики різного рівня: від глобальних організаційних до локальних, прив’язаних до структури окремого репозиторію. Внаслідок цього, Copilot виступає не лише як автономний мовний інструмент, а як частина інженерної інфраструктури, що здатна інтерпретувати архітектурні обмеження(рис. 3.1), стилістичні правила та вимоги до безпеки. Застосування кастомізації особливо важливе в умовах великомасштабних проєктів, мікросервісних архітектур та середовищ із суворими вимогами до відповідності стандартам.

3.1.1 Інструкції Copilot

Інструкції Copilot — це формалізований механізм передачі моделі стійких правил функціонування, що застосовуються до взаємодії через Copilot Chat та певною мірою впливають на генерацію коду. Інструкції створюються у вигляді Markdown-файлів, які можуть бути розташовані у корені робочої області або в окремих директоріях, що дозволяє встановлювати правила щодо конкретних підсистем. В основі роботи інструкцій лежить YAML-пролог із параметром applyTo, що визначає передумови активації та патерни файлів, щодо яких застосовуються правила.

Файл інструкцій містить текстові директиви, що формулюють вимоги до стилю, архітектури, принципів обробки помилок, форматів документування, структуризації модулів та стандартів тестування. Інструкції також можуть описувати вимоги до безпеки, що забороняють небажані API або конструкції, які суперечать внутрішнім політикам компанії. Така формалізація дозволяє забезпечити узгодженість коду у масштабі всього проєкту.

Розташування інструкцій у репозиторії створює єдине джерело правди щодо вимог до коду, а відтак забезпечує їх одноманітне застосування всіма учасниками команди.

Інструкції оптимізують роботу Copilot, що спрямовує генерацію у межі, визначені архітектурою. Завдяки цьому, зменшується кількість відхилень та непередбачуваних рішень. Модель починає виробляти код, що органічно вписується у структуру проєкту, а також автоматично дотримується вимог до тестів, обробки виключень або логування, якщо такі вимоги вказані в інструкціях. Це дозволяє суттєво скоротити витрати часу на первинне редагування та стандартизацію згенерованих фрагментів.

Система Custom Instructions у Visual Studio Code — це фундаментальний механізм, що визначає, як саме GitHub Copilot поводиться у межах проєкту. Інструкції виступають внутрішньою конституцією моделі. Вони керують стилем коду, вимогами до архітектури, правилами тестування, форматами компонентів та навіть тим, які бібліотеки дозволено використовувати.

Всі інструкції зберігаються у файлі copilot/INSTRUCTIONS.md, що є частиною репозиторію. Це означає, що кожний член команди отримує однакову поведінку ШІ, а Copilot діє як структурований інженер, а не як випадковий генератор коду.

Створення структури проєкту з Custom Instructions. У корені репозиторію створюється директорія: copilot/. У ній розміщується файл з інструкціями: .copilot/INSTRUCTIONS.md. Це основний документ, що описує правила роботи Copilot.

Приклад наповнення файлу INSTRUCTIONS.md:

```
# General Rules
Always generate TypeScript code.
Follow React best practices: functional components, hooks, strict typing.
Avoid any, use generics or inference where possible.
Do not use deprecated APIs under any circumstances.
```

```
    # Testing Rules
    For each new component, generate a Jest test file using React Testing
Library.
    Test file should be located next to the component with `.test.tsx`
extension.

    # Quality Requirements
    Keep cyclomatic complexity low.
    Avoid duplicated code.
    Prefer composition over inheritance.
    Refactor functions longer than 40 lines by splitting them.

    # Security Rules
    Follow Snyk recommendations.
    Never generate code that introduces unsafe eval, Function constructors,
or dynamic imports.

    # Documentation Rules
    Every new component must be documented using Notion MCP.
    Documentation must include props, events, side effects, and examples.
```

З цього моменту будь-який запит до Copilot автоматично проходить через ці правила. Технічно це працює таким чином: користувач у редакторі пише запит, наприклад: “Створи React компонент для форми логіна”, то Copilot отримує не лише цей текст, а й вміст INSTRUCTIONS.md, що додається до промпту у прихованому форматі.

Агенти використовують цей файл як свій глобальний основний документ. Наприклад, агент Playwright MCP перед генерацією E2E-тестів читає INSTRUCTIONS.md та будує тест-структури згідно з правилами. А Jest-орієнтований агент аналізує, чи змінює код структуру компонентів й запускає тести відповідно до політики інструкцій.

3.1.2 Промпт-файли Copilot

Промпт-файли є розширеною формою інструкцій, що орієнтовані на опис конкретних сценаріїв взаємодії моделі з проєктом. На відміну від загальних інструкцій, що визначають загальний інженерний стиль, промпт-файли містять системні промпти, які Copilot автоматично використовує під час виконання певних операцій, зокрема під час створення нових файлів, рефакторингу або побудови тестів.

Такі файли можуть описувати стандартизовані шаблони структур, що притаманні окремим типам модулів. Наприклад, промпт-файл може визначати уніфікований формат мікросервісу, що включає структуру каталогів, принципи організації контролерів, сервіси, моделі даних та шаблони логування. Іншим прикладом є визначення структури тестового набору, включно з вимогами до назв тестів, підходами до сторін-тестування та сценаріями edge-case. У такий спосіб Copilot перетворюється на інструмент відтворення встановлених архітектурних патернів, що особливо важливо у великих командах, де регулярність та повторюваність є ключовими факторами продуктивності.

Промпт-файли також полегшують онбординг нових учасників команди, забезпечуючи їх доступом до детальних внутрішніх стандартів без необхідності вивчення об'ємної документації. Генерація нових модулів у цьому випадку здійснюється відповідно до формалізованих вимог, що сприяє зниженню кількості помилок та покращенню цілісності системи.

Prompt Files у VS Code — це спосіб перетворити разові запити до Copilot на стабільні, повторювані шаблони. Якщо Custom Instructions задають загальні правила поведінки моделі у всьому проєкті, то prompt-файли описують конкретні сценарії: як створювати компонент, як генерувати тести, як писати документацію в Notion, як виконувати рефакторинг з урахуванням Sonar MCP тощо. Це вже не просто довільний текст у чаті, а формалізований інструмент, що лежить у репозиторії й підпорядковує роботу ШІ чітким, технічно описаним вимогам.

Підготовка починається зі створення директорії .copilot/prompts у корені проєкту. У цій папці зберігаються всі шаблони промптів, що використовують як команди вищого рівня щодо Copilot. Наприклад, можна завести окремі файли create_component.md, generate_tests.md, refactor_with_sonar.md, document_in_notion.md тощо. Кожен файл описує одну логіку взаємодії з моделлю, але вже не у вигляді одноразового запиту, а у вигляді стандартизованого сценарію.

Типовий приклад: створюємо файл .copilot/prompts/create_component.md для React + TypeScript.

```
# create_component
You are generating a new React + TypeScript UI component.
Requirements:
- Use a functional component with strict TypeScript props.
- Place file under src/components/ComponentName/ComponentName.tsx.
- Create a CSS Module file ComponentName.module.css.
- Create a Jest test file ComponentName.test.tsx using React Testing
Library.
- Ensure all imports are relative and consistent with the project
structure.
- Follow the global rules from .copilot/INSTRUCTIONS.md.
```

Тепер, коли у Copilot Chat або у вікні редактора пишеш щось, наприклад: "Use prompt create_component to build a new LoginForm component", Copilot уже не буде вигадувати структуру самостійно, а візьме цей prompt-файл як базу.

3.1.3 Кастомні агенти Copilot

Кастомні агенти є найбільш функціонально складним елементом системи кастомізації Copilot. На відміну від інструкцій та промпт-файлів, що впливають на текстову поведінку моделі, агенти можуть виконувати окремі дії у межах IDE. Їхня робота реалізується через агентну інфраструктуру Copilot, що передбачає наявність набору інструментів, доступних агенту щодо аналізу, модифікації або створення нових файлів.

Агент може реалізовувати процедури аналізу архітектурної цілісності, перевірки дотримання внутрішніх стандартів, генерації тестового покриття або комплексного рефакторингу. Наприклад, агент може автоматично перевірити

відповідність модулів певним API-контрактам або оновити всі ендпоїнти після зміни версії архітектури. Такий підхід дозволяє автоматизувати завдання, що зазвичай вимагають значного часу та ручної роботи.

Кастомні агенти можуть бути адаптовані до будь-якого домену, що дає можливість формувати специфічні політики для галузей із підвищеними вимогами до стандартів. У сфері фінансової безпеки агент може перевіряти відповідність вимогам PCI DSS, у медичних системах — HIPAA. Таким чином, агенти виконують роль інтелектуальних модулів забезпечення відповідності.

Custom Agents — це найбільш гнучкий рівень керування генеративним ШІ у VS Code. Якщо Custom Instructions встановлюють стиль, а Prompt Files формують повторювані робочі сценарії, то агент дає можливість автоматизувати ці сценарії та виконувати їх програмно. Фактично агент перетворює Copilot з розумного автодоповнення на повноцінного автоматизованого інженера, що може запускати тести, читати результати аналізу якості коду, оновлювати документацію та виконувати складні процедури рефакторингу без участі розробника.

Структура агента. Агент створюється у директорії: .copilot/agents/ та описується у YAML-файлі, наприклад: .copilot/agents/refactor_agent.yaml

У цьому файлі визначаються:

- дозволені команди (npm-команди, tsc, jest, playwright);
- дозволені файли щодо читання/запису;
- інструменти MCP, що агент може викликати;
- цілі, правила та спеціальна поведінка.

Нижче наведено типовий агент, що використовує Jest, Playwright, Sonar MCP та Notion MCP:

```
    name: "refactor_and_test_agent"
    description: "Agent for refactoring TypeScript/React code using Sonar
MCP, Jest, Playwright and Notion documentation updates."
    goals:
      - "Fix code smells and quality issues using Sonar MCP."
      - "Ensure all Jest tests pass after modification."
      - "Run Playwright tests for E2E flows related to updated modules."
      - "Synchronize updated documentation in Notion through Notion MCP."
    runCommands:
      allow:
        - "npm run test:ci"
        - "npx playwright test --reporter=json"
        - "npm run lint"
        - "npm run build"
    readFiles:
      include:
        - "jest-results.json"
        - "playwright-report.json"
        - "src/**/*.ts"
        - "src/**/*.tsx"
    mcpServers:
      sonar: "sonar-mcp"
      notion: "notion-mcp"
```

```
playwright: "playwright-mcp"
```

Такий агент забезпечує:

- автоматичне тестування;
- автоматичний рефакторинг;
- автоматичне оновлення документації.

Разом вони формують повністю контрольовану інженерну екосистему.

3.2 Використання показників Code Smell

Code smell у сучасній програмній інженерії визначається як непрямі дефекти структури або стилю програмного коду, що не викликають помилок компіляції чи виконання, проте свідчать про потенційні проблеми у подальшій підтримці, модифікації, читабельності, продуктивності та безпеці програмних систем. На відміну від синтаксичних помилок, smell-и є ознаками деградації архітектури або стилю, що накопичуються поступово й негативно впливають на життєвий цикл програмного продукту. Їхня особливість полягає у можливості формалізації у вигляді кількісних метрик, що робить їх придатними щодо автоматичного аналізу та використання у процесах оптимізації генеративних моделей ШІ. Оскільки ці метрики мають чітко визначені критерії вимірювання, вони можуть бути використані як об’єктивні орієнтири, що дозволяють ШІ системам визначати недоліки згенерованого коду та спрямовувати його регенерацію у бік поліпшення архітектурних властивостей.

Серед основних формалізованих показників, що характеризують наявність smell-ів, особливе значення має циклична складність, що визначає кількість незалежних шляхів виконання у певній функції або модулі. Зростання цього показника свідчить щодо надмірного розгалуження логіки, що ускладнює розуміння, тестування та модифікацію коду. Стосовно генеративних моделей такий сигнал вказує на необхідність спрощення внутрішньої логіки, подрібнення великих функцій на менші або оптимізації умовних конструкцій. Іншим суттєвим показником є ступінь дублювання коду. Наявність великої кількості повторюваних фрагментів підвищує ризик появи помилок та ускладнює рефакторинг. Щодо моделей ШІ, це означає потребу в узагальненні функціональності, переведенні повторюваної логіки в окремі методи або модулі та формуванні більш чистої архітектури.

Важливе значення мають також показники пов’язаності та сцільності. Надмірна кількість залежностей між модулями чи низький рівень внутрішньої єдності класів прямо свідчать щодо архітектурних проблем, які згодом ускладнюють масштабування й модифікацію системи. У таких випадках генеративна модель отримує можливість оптимізувати структуру, пропонуючи перерозподіл відповідальностей, створення нових інтерфейсів або скорочення надлишкових залежностей. Схожим аналітичним сигналом є великий розмір методів або класів. Значні за обсягом одиниці коду вказують на порушення принципів модульності та перевантаження функцій, що негативно впливає на підтримуваність. Системи ШІ у таких ситуаціях спрямовують свої висновки на декомпозицію, композицію або виділення підмодулів.

Глибина ієрархії наслідування також виконує індикативну роль. Розлогі та надмірно складні дерева класів погіршують розуміння системної поведінки, ускладнюють відлагодження та підвищують ризики небажаних побічних ефектів. Наявність подібного smell-у стимулює ШІ до спрощення ієрархічних структур або переходу до композиційних рішень. Додатковим показником є рівень цільності, що вимірюється, зокрема, через LCOM. Якщо методи одного класу працюють з різними наборами даних та не формують узгоджену логічну одиницю, це свідчить щодо перевантаження відповідальностями. Генеративна модель, що отримує такі сигнали, пропонує формувати більш вузькоспеціалізовані класи та модулі.

Значення smell-метрик зростає в умовах інтеграції автоматизованих інструментів контролю якості. Більшість сучасних IDE обладнані вбудованими механізмами статичного аналізу, що можуть працювати у фоновому режимі або за запитом користувача. Такі інструменти виконують роль сенсорної системи, що забезпечує генеративний ШІ структурованими сигналами щодо стану коду. Найпоширенішим комплексним засобом аналізу є SonarLint та SonarQube, які здатні виявляти широкий спектр smell-ів, включно з проблемами складності, дублювання, безпекових вразливостей, помилок стилю та архітектурної організації. Лінтери, такі як ESLint, Pylint чи RuboCop, забезпечують стандартизацію стилю та виявлення локальних структурних недоліків. Розширення статичного аналізу у середовищі Visual Studio Code, включно з IntelliSense та іншими модулями, виконує точкову діагностику залежностей, складності та ефективності окремих фрагментів коду.

У сучасних інтегрованих сценаріях генерації smell-метрики набувають роль механізму регуляції. Процес взаємодії здійснюється через циклічний аналіз: після кожної генерації коду система статичного аналізу виконує сканування, формує структурований звіт у форматах JSON або SARIF, передає його моделі, яка згодом коригує результат та повторно генерує код з урахуванням виявлених недоліків. Цикл триває доти, доки ключові показники складності, дублювання та пов’язаності не будуть приведені до заданих нормативів. У таких умовах smell-и перестають бути лише індикаторами проблем, а перетворюються на функціональні критерії оптимізації, що формують поведінку ШІ.

Формальна інтеграція smell-метрик у процес генерації коду суттєво трансформує роль генеративного ШІ у розробці ПЗ. Модель перестає бути інструментом пасивного автодоповнення та набуває ознак системи інженерії якості, здатної до ітеративного вдосконалення архітектури й структури коду. Це забезпечує зменшення структурної складності, зниження дублювання, підвищення модульності, покращення підтримуваності та передбачуваності результатів. У широкому контексті така інтеграція підвищує ефективність розробки та створює основу щодо формування інтелектуальних інструментів нового покоління, що здатні підтримувати високі стандарти програмної інженерії на кожному етапі життєвого циклу продукту.

Sonar MCP додає у всю цю історію найважливішу річ: формальні метрики якості коду. Якщо Jest і Playwright відповідають за поведінку, то Sonar дає ШІ-агенту рентген структури: цикломатичну складність, дублювання, code smell,

потенційні вразливості, порушення архітектурних правил. Через MCP все це стає доступним Copilot та кастомних агентів у структурованому форматі, що можна використовувати як прямі цілі для рефакторингу[25].

Рисунок 3.2 – Sonar MCP

Починається все з налаштування самого SonarQube (або SonarCloud) та підйому Sonar MCP(рис. 3.2) Server згідно з офіційною документацією. На боці VS Code необхідно підключити цей MCP-сервер у конфігурації .copilot/mcp.json:

```
{
  "version": 1,
  "servers": {
    "sonar-mcp": {
      "description": "Provides access to SonarQube static analysis results (issues, metrics, hotspots, duplications).",
      "url": "http://your-sonarqube-host.com/mcp",
      "apiKeyEnv": "SONAR_MCP_API_KEY"
    }
  }
}
```

Тут SONAR_MCP_API_KEY – це змінна середовища з токеном доступу до Sonar. Після цього Copilot знає, що є сервер sonar-mcp, з якого можна отримати:

- список проблем (issues) до файлу/проєкту;
- метрики (coverage, complexity, duplications);
- security-hotspots;
- інформацію щодо code smells.

Далі цей MCP-сервер треба вшити у конфігурацію агента.

У файлі .copilot/agents/refactor_and_test_agent.yaml додається:

```
externalContexts:
  sonar:
    server: "sonar-mcp"
    resources:
      - "issues"
      - "metrics"
```

```
    - "hotspots"
    - "duplicates"
```

Тепер агент може викликати Sonar MCP й отримувати щодо конкретного файлу або модуля усі відповідні smell-и й метрики якості.

Sonar MCP повертає JSON з описом проблем: тип, правило, короткий опис, рекомендації, позиція у коді, рівень серйозності. Агент формує з цього компактний контекст стосовно промпта до LLM: “Ось файл, ось список smell-ів, ось фрагменти коду, де виявлені проблеми. Перепиши цей файл так, щоб нівелювати основні issues, не ламаючи API та логіку.” Модель виконує цілеспрямований рефакторинг, а не довільне покращення стилю.

Для більш точного керування можна задати цілі агента у YAML-конфігу:

```
goals:
  - "Reduce cyclomatic complexity in files with high Sonar complexity score";
  - "Eliminate duplicated blocks flagged by Sonar";
  - "Resolve all BLOCKER and CRITICAL issues before new code generation".
```

Ці цілі визначають порядок пріоритетів: спочатку блокери, потім критичні проблеми, потім великі smell-и. Після рефакторингу агент може повторно запитати Sonar MCP (або запустити sonar-аналіз як частину CI) та порівняти нові метрики зі старими.

Інтеграція з Jest та Playwright тут критична. Sonar може підказати, що у певного модуля занадто висока складність та він містить кілька code smell-ів, але повністю покривається тестами. У такій ситуації агент чітко діє за політикою: рефакторити код без зміни зовнішньої поведінки, запускати Jest та Playwright після кожної ітерації, доки всі тести знову не стануть зеленими. Якщо Sonar показує, що покриття тестами недостатнє, агент може використати prompt-файли щодо генерації тестів до функцій, що мають низьке покриття й, таким чином, покращити не тільки структуру коду, а й рівень перевірки.

Особливо корисним стає поєднання Sonar MCP із Notion MCP. Наприклад, архітектурні правила, вимоги до максимальної складності, стандартів іменування та організації модулів можуть бути формалізовані у документації Notion. Агент отримує ці правила через Notion MCP, а реальні порушення через Sonar MCP. У підсумку він генерує рефакторинг, що не лише прибирає smell-и, а й наближає систему до описаної у документації цільової архітектури. Далі оновлені метрики можуть бути автоматично додані у Notion як частина історії еволюції коду.

Отже, Sonar MCP стає стратегічним сенсором якості стосовно ШІ-агентів у VS Code. Він дозволяє працювати не наосліп, а на основі зрозумілих технічних показників, перетворюючи Code Smell і метрики на прямі цілі генерації та рефакторингу. Разом із Jest, Playwright, Snyk, Notion MCP та Dev Containers це дає змогу підтримувати не тільки працездатність коду, а й його структурну зрілість, безпеку та відповідність стандартам команди.

3.3 Покриття коду тестами Jest та Playwright

Jest у цьому стеку виконує роль базового рівня логіки застосунку. Основне завдання — зробити так, щоб Copilot та кастомні агенти не просто генерували й рефакторили код, а й автоматично перевіряли, чи залишилася логіка

працездатною та на основі результатів тестів приймали наступні рішення. З цією метою потрібно налаштувати Jest так, щоб його можна було запускати з командного рядка, отримувати результати у машинозчитуваному форматі, а також дати агенту доступ до цих результатів.

Початково Jest підключається у проєкт як звичайно.

У корені з'являється jest.config.ts. Тут задається середовище, шляхи до тестів, трансформери TypeScript тощо. Важливо додати окремий скрипт щодо машинного запуску тестів.

У файлі package.json варто прописати:

```
"scripts": {
  "test": "jest",
  "test:ci": "jest --json --outputFile=jest-results.json"
}
```

Звичайний npm test зручний для розробника, а npm run test:ci потрібен ШІ-агенту. Команда з параметрами --json й --outputFile генерує файл jest-results.json, де у структурованому форматі описані всі тести, що пройшли, з повідомленнями, стеком, часом виконання та шляхами до файлів.

Щоб агент міг використовувати ці результати, у конфігурації агента (наприклад .copilot/agents/refactor_and_test_agent.yaml) потрібно дозволити запуск Jest та читання вихідних файлів.

Це робиться так:

```
runCommands:
  allow:
    - "npm run test:ci"
readFiles:
  include:
    - "jest-results.json"
    - "src/**/*.ts"
    - "src/**/*.tsx"
```

Тепер агент має право запустити npm run test:ci, а потім проаналізувати jest-results.json.

Типовий сценарій виглядає так: агент змінює код (наприклад, за результатами аналізу Sonar MCP або на запит користувача), після чого автоматично виконує Jest у CI-режимі. Якщо команда повертає успішний код виходу, агент фіксує, що зміни не порушили поточних інваріантів. Якщо ж Jest повертає помилку, агент зчитує JSON, вибирає з нього ключову інформацію (назву теста, шлях до файлу, повідомлення про помилку, очікувані та фактичні значення) й передає ці дані у промпт до LLM.

Практично це виглядає як запит до моделі приблизно такого змісту: "Ось фрагмент коду, який щойно було змінено, а ось — фрагмент звіту Jest з помилкою для цього ж файлу. Скоригуй код так, щоб тест пройшов, не змінюючи контракт публічного API та зберігши логіку, описану у тесті". Важливо, що агент у такій конфігурації працює циклічно: після регенерації коду він може знову викликати npm run test:ci й перевірити, чи помилка зникла. Таким чином, тестування перестає бути ручним кроком, а стає частиною автоматизованого конвеєра.

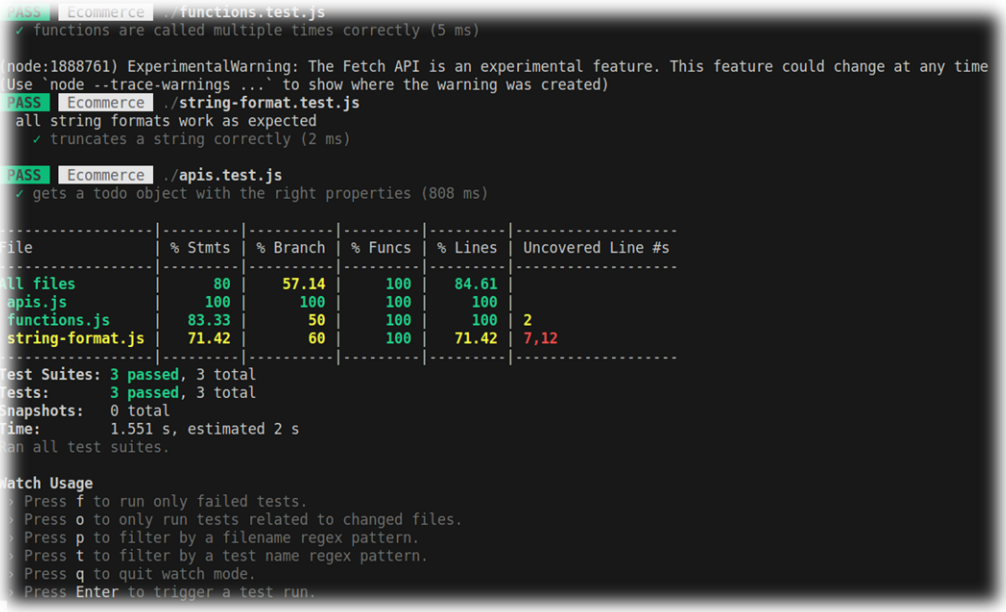

Рисунок 3.3 – Jest тести

Отже, вірно налаштований Jest(рис. 3.3) інтегрується у роботу ШІ-агента як обов'язковий етап перевірки кожної суттєвої зміни коду. Агент не тільки генерує код, а й перевіряє його тестами, читає результат та робить на його основі наступні кроки: від повторного рефакторингу до оновлення документації, що робе весь процес розробки більш надійним та технічно контрольованим.

Якщо Jest відповідає за модульний рівень перевірки логіки, то Playwright дає змогу контролювати поведінку застосунку з точки зору реального користувача: переходи між сторінками, введення даних, кліки, очікування елементів, відображення помилок тощо. Основне завдання — зробити так, щоб Playwright працював не лише як окремий інструмент, а як частина повноцінного циклу.

Початкова інтеграція Playwright виконується стандартно. Після цього у проєкті з'являється файл playwright.config.ts та директорія з тестами, наприклад, tests або e2e. Щодо подальшої автоматизації потрібно налаштувати скрипт у package.json, що повертатиме результати у машинозчитуваному вигляді[26]:

```
    "scripts": {
      "test:e2e": "playwright test",
      "test:e2e:ci": "playwright test --reporter=json --output=playwright-
report"
    }
```

Команда test:e2e:ci запускає всі E2E-тести й формує JSON-звіт та артефакти у каталозі playwright-report, що пізніше можуть бути використані ШІ-агентом. Щоб зробити Playwright доступним через MCP, у файлі .copilot/mcp.json описується Playwright MCP-сервер. Наприклад:

```
    {
      "version": 1,
      "servers": {
        "playwright-mcp": {
```

```
          "description": "Provides access to Playwright E2E test runs and
results.",
          "url": "http://your-playwright-mcp-host.com/mcp",
          "apiKeyEnv": "PLAYWRIGHT_MCP_API_KEY"
        }
      }
    }
```

Тепер VS Code та Copilot знають, що є сервер playwright-mcp, через який можна запускати E2E-тести, отримувати список тестів, їхні результати та, за потреби, посилання на HTML-репорти й trace-файли. Далі агенту потрібно пояснити, як саме він може користуватися цим MCP. У файлі .copilot/agents/refactor_and_test_agent.yaml додається блок:

```
externalContexts:
  playwright:
    server: "playwright-mcp"
    resources:
      - "tests"
      - "runs"
      - "results"
      - "reports"

runCommands:
  allow:
    - "npm run test:e2e:ci"
```

Типовий робочий цикл виглядає так. Спочатку агент виконує рефакторинг або генерацію коду (зазвичай орієнтуючись на результати Sonar MCP і Jest). Після цього він хоче перевірити, чи не зламалися ключові сценарії. Агент викликає Playwright MCP, наприклад, із дією “runTests” стосовно конкретного тест-сету (checkout flow, login flow, dashboard navigation тощо). MCP-сервер запускає Playwright, збирає результати й повертає агенту структурований JSON: статус (успіх/помилка), список вдалих тестів, маршрут URL, селектори, що не знайшлися, тексти повідомлень, що не відобразилися, тощо.

Далі агент використовує ці дані як контекст для LLM. Можна сформувати запит, наприклад: “Ось код компонента, ось Playwright-звіт, що показує, як за некоректних платіжних даних не з’являється відповідне повідомлення щодо помилки. Виправ код, зберігши всі інші сценарії.” Модель аналізує опис та вносить зміни у компонент або логіку маршрутизації. Після цього агент повторно викликає Playwright (локально або через MCP), доки всі відповідні E2E-тести не будуть зеленими.

Справжня сила Playwright MCP розкривається у поєднанні з Notion MCP та Jest. Наприклад, вимоги до E2E-сценаріїв можуть зберігатися у Notion як таблиця з колонками: “User story”, “Steps”, “Expected result”. Агент читає ці вимоги через Notion MCP, генерує Playwright-тести відповідно до вимог, запускає їх через Playwright MCP та, у разі падіння, використовує Jest стосовно деталізації проблем на рівні модулів. Таким чином, система отримує багаторівневий контроль якості: Notion задає вимоги, Playwright перевіряє поведінку з точки зору користувача, Jest

гарантує коректність модульної логіки, а Sonar MCP контролює структурну якість коду.

Для стабільної роботи Playwright у DevContainer можна використати офіційні образи, наприклад:

```
{
  "name": "React TS + Playwright DevContainer",
  "image": "mcr.microsoft.com/playwright:v1.48.0-jammy",
  "postCreateCommand": "npm install",
  "remoteUser": "pwuser",
  "customizations": {
    "vscode": {
      "extensions": [
        "ms-playwright.playwright",
        "GitHub.copilot",
        "GitHub.copilot-chat"
      ]
    }
  }
}
```

У такій конфігурації Playwright має однакове середовище в усіх розробників та на CI, а агент, що працює поверх DevContainer й Playwright MCP, може будувати повністю автоматизовані E2E-перевірки як невід'ємну частину ШІ-орієнтованого сценарію.

3.4 Використання Multi-Context Protocole

Multi-Context Protocol (MCP) є фундаментальним механізмом інтеграції зовнішніх джерел даних, інструментів й сервісів у робоче середовище генеративних моделей, включно з GitHub Copilot та іншими інструментами, що використовують великі мовні моделі (LLM). Його ключове призначення полягає у встановленні стандартизованого каналу обміну структурованою інформацією між моделлю та програмним середовищем, де вона функціонує. MCP забезпечує можливість передачі даних, виконання команд, отримання контексту проєкту й залучення сторонніх сервісів у єдиному формалізованому форматі, що дозволяє моделі працювати на основі актуальної інформації та виконувати завдання, що були раніше недоступними через обмеження локального контексту[27].

У традиційних сценаріях роботи генеративні моделі мають доступ виключно до тексту у межах поточного запиту та вмісту редактора, що перебуває у відкритому файлі. Це обмежує їхню здатність аналізувати складні архітектурні залежності, взаємодіяти з документацією, конфігураціями, інструментами CI/CD або зовнішніми базами знань. MCP розв'язує цю проблему, надаючи стандартизований інтерфейс між LLM та будь-яким зовнішнім ресурсом, що має формально описаний протокол взаємодії. Таким чином, модель може працювати з розширеним контекстом, що включає дані, наявні поза межами відкритого файлу, та застосовувати їх у процесах генерації, перевірки й корекції коду.

Принцип роботи MCP полягає у створенні шару абстракції, що розташований між мовною моделлю та інструментами, що надають контекст. Цей шар діє як універсальний транспортний механізм, що передає структуровані повідомлення у

форматі, зрозумілому як LLM, так й програмним компонентам, що забезпечують необхідні дані. MCP визначає формат повідомлень, їхню семантику та особливості взаємодії сторін. У межах цього протоколу модель може надсилати запити на отримання інформації, а зовнішні системи можуть відповідати на них з урахуванням визначених контрактів. Завдяки цьому встановлюється динамічна інтеграція, за якої модель може працювати з актуальними даними й адаптувати свою поведінку відповідно до умов виконання.

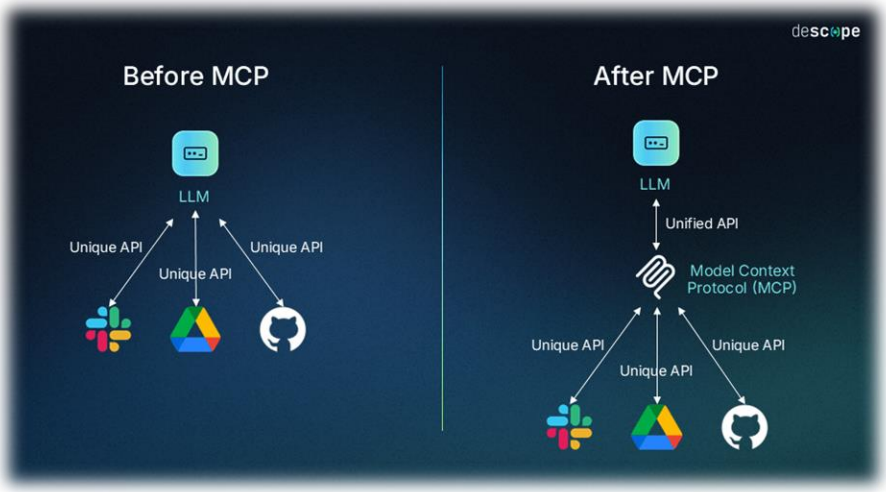

Рисунок 3.4 – Принцип роботи MCP

Механізми MCP(рис. 3.4) дозволяють інтегрувати різноманітні ресурси, включно з локальними файлами, системами документації, сервісами конфігураційного менеджменту, аналітичними інструментами, базами даних, API підприємств та засобами забезпечення безпеки. У процесі генерації коду це забезпечує можливість створювати рішення, що відповідають специфікаціям, внутрішнім протоколам та регламентам, без необхідності ручного копіювання структур чи інструкцій. Наприклад, модель може отримати інформацію щодо актуальної версії API, обмеження доступу, рекомендовані шаблони проєктування або конфігураційні параметри, що дозволяє формувати коректний та безпечний код.

Особливості використання MCP проявляються у сценаріях, де потрібно працювати з великими обсягами даних або складними залежностями. У таких випадках MCP забезпечує доступ до контексту, що не може бути розміщений у межах токен-вікна самої моделі. Це дозволяє структурувати процес генерації коду, що робе його менш залежним від прямої передачі тексту у промпті та більш залежним від динамічного доступу до структурованої інформації. Модель отримує можливість запитувати необхідні фрагменти під час роботи, що підтримує актуальність результатів та зменшує помилки, пов’язані з контекстним вигоранням.

Важливим аспектом застосування MCP є формальна валідація даних, що передаються моделі. Оскільки протокол використовує структуровані формати, такі як JSON або інші контрактні схеми, це мінімізує ризики передачі

неконсистентної або неоднозначної інформації. У ситуаціях, коли модель отримує структури, що відповідають формальним правилам, зменшується ймовірність появи помилок у логіці генерації. Це створює передумови щодо підвищення точності та відтворюваності результатів, особливо у великих проєктах, де помилки у конфігурації або специфікаціях можуть призвести до значних витрат.

Застосування MCP у середовищах розробки сприяє появі нового підходу до автоматизації інженерних процесів. Генеративна модель, яка має доступ до інструментів аналізу безпеки, може автоматично перевіряти згенерований код на відповідність вимогам, що визначені у внутрішніх політиках організації. Такі перевірки охоплюють аналіз залежностей, виявлення вразливостей, відповідність ліцензійним обмеженням або наявність небажаних структур програмування. Оскільки протокол забезпечує двосторонню інтеграцію, модель може не лише отримувати результати аналізу, а й передавати рекомендації стосовно автоматизованого рефакторингу або корекції.

У випадках роботи з архітектурною документацією MCP дозволяє моделі взаємодіяти з актуальними версіями технічних описів, схем баз даних, діаграм взаємодій або специфікацій API. Це забезпечує формування коду, що відповідає встановленим нормам та використовує структури даних, які визначені у документах. Завдяки цьому розробник отримує високий рівень гарантії, що згенеровані компоненти відповідають очікуванням системних архітекторів, а модель може дотримуватися корпоративних норм без ручної перевірки.

Застосування MCP також суттєво впливає на процес тестування. Наявність протоколу дозволяє моделі отримувати дані з інструментів, що формують тестові звіти, і автоматично адаптувати генерацію відповідно до результатів. Наприклад, якщо тестова система повідомляє щодо недостатньої кількості тестових випадків або невідповідність покриття, модель може запропонувати або згенерувати додаткові тести, що покращують валідацію певної функціональності. Цей підхід забезпечує адаптивний характер тестування, де генерація здійснюється не лише з урахуванням вимог користувача, а й відповідно до динамічних результатів перевірок.

З огляду на сучасні вимоги до безпеки, MCP відіграє важливу роль у створенні систем, що відповідають стандартам комплаєнсу. Зокрема, інтеграція із сервісами безпеки дозволяє проводити автоматизовані аудитні перевірки, виявляти проблеми з доступом, неправильні конфігурації або невідповідні залежності. У таких сценаріях модель отримує сигнали щодо невідповідності та може автоматично коригувати код або конфігураційні файли. Це створює передумови для побудови систем, що здатні самостійно підтримувати необхідний рівень безпеки.

Multi-Context Protocol також ефективно застосовується у DevOps-процесах, оскільки надає можливість генеративним моделям взаємодіяти з інструментами CI/CD. Модель може отримувати інформацію щодо результатів збірки, помилки, журнали виконання, статуси розгортання або конфігураційні обмеження. Це дає змогу автоматично адаптувати структуру коду й конфігурацій відповідно до вимог середовищ, де відбувається розгортання. На практиці це означає

можливість створення коду, що відразу враховує специфіку продуктивного оточення без додаткових ручних правок.

Застосування MCP формує новий підхід до управління контекстом у системах генеративного коду. Раніше контекст був статичним й обмежувався фрагментом файлу або вручну наданими інструкціями. MCP впроваджує динамічний контекст, що формується на основі запитів моделі та не вимагає ручного втручання користувача. Це дозволяє забезпечити точніший добір даних, підвищити актуальність результатів та зменшити ризики помилок, пов'язаних із застарілою інформацією.

Отже, використання Multi-Context Protocol у сучасних системах генерації коду сприяє створенню глибоко інтегрованих, адаптивних та контекстно-орієнтованих інструментів розробки. MCP забезпечує доступ до важливих джерел інформації, автоматизує перевірки, зменшує кількість ручних операцій та підвищує точність рішень. Це робить протокол ключовим елементом у розвитку інтелектуальних інструментів програмування, здатні працювати з високим рівнем автономності та точності, що відповідають вимогам сучасної інженерії ПЗ.

3.5 Документація коду та змін через Notion MCP

Notion MCP перетворює Notion на повноцінне джерело та приймач структурованих даних щодо ШІ-агента. Ідея проста: все, що зазвичай лежить у Confluence/Notion (вимоги, архітектура, специфікації API, опис компонентів, сценарії використання) стає доступним у вигляді машинозчитуваних об'єктів. Агент може читати ці дані, генерувати код, тести, документацію, а потім ще й оновлювати сторінки у Notion, коли змінюється кодова база[28].

Починається все з налаштування Notion MCP-сервера та його підключення у VS Code через файл .copilot/mcp.json:

```
{
  "version": 1,
  "servers": {
    "notion-mcp": {
      "description": "Provides access to internal documentation and
project requirements stored in Notion.",
      "url": "https://your-notion-mcp-endpoint.com",
      "apiKeyEnv": "NOTION_MCP_API_KEY"
    }
  }
}
```

NOTION_MCP_API_KEY - токен із доступом до відповідного воркспейса/бази. Після цього Copilot та кастомні агенти бачать notion-mcp як зовнішнє джерело контексту.

Далі у конфігурації агента (.copilot/agents/refactor_and_test_agent.yaml) додається блок:

```
externalContexts:
  notion:
    server: "notion-mcp"
    resources:
      - "pages"
```

```
      - "databases"
      - "blocks"
```

Це означає, що агент має право:

- читати сторінки (pages): документація, архітектурні описи, гайди;
- читати/змінювати записи у базах даних (databases): таблиці з вимогами, API-ендпоінтами, компонентами;
- працювати з блоками (blocks) - окремими елементами сторінок: заголовками, параграфами, списками, таблицями.

Агент через Notion MCP отримує запис:

```
{
  "name": "LoginForm",
  "props": [
    { "name": "onSubmit", "type": "(values: LoginValues) => void" },
    { "name": "isLoading", "type": "boolean" }
  ],
  "requirements": [
    "Display validation errors under each field",
    "Disable submit button while isLoading = true",
    "Show global error banner when login fails"
  ]
}
```

На основі цього агент викликає prompt (наприклад, create_component.md), а Copilot генерує компонент, що із самого початку відповідає вимогам з Notion. Одразу можна згенерувати Jest-тести та Playwright E2E-сценарії, що перевіряють саме ці вимоги.

Другий сценарій: код → документація. Після рефакторингу компонента або додавання нового модуля агент може автоматично оновити документацію. Наприклад, у .copilot/prompts/document_in_notion.md описуються правила:

```
# document_in_notion
You document a React/TypeScript component in Notion.

Include:
- Component purpose (1–2 sentences)
- Props with types and descriptions
- Events/callbacks
- Important states and edge cases
- Links to related tests (Jest, Playwright)
Агент:
Читає вихідний код компонента.
Аналізує його структуру, пропси, логіку та прив’язані тести.
Генерує текст документації за цим prompt-ом.
Викликає Notion MCP щодо створення або оновлення відповідної сторінки:
{
  "action": "updatePage",
  "pageId": "some-page-id",
  "content": "Generated markdown or Notion blocks here..."
}
```

У результаті зміни коду актуалізується й документація. Це особливо важливо, коли у проєкті багато компонентів, а підтримувати опис вручну стає нереально.

У підсумку Notion MCP виконує дві ключові функції. По-перше, це джерело формалізованих вимог, архітектурних описів та стандартів, за якими ШІ генерує й оцінює код. По-друге, це місце, куди автоматично повертаються результати роботи ШІ: оновлена документація компонентів, модулів, E2E-сценаріїв, звіти щодо якості та безпеки. Документація перестає бути слабкою ланкою процесу, а перетворюється на рівноправний артефакт, що підтримується інтелектуальною інфраструктурою.

3.6 Аналізи безпеки коду Synk

Забезпечення безпеки програмного коду є одним із ключових компонентів життєвого циклу ПЗ. Зростання складності програмних систем, кількості сторонніх залежностей та швидкості розробки створює нові виклики, що вимагають постійного моніторингу вразливостей, аналізу залежностей, оцінки конфігурацій й контролю за дотриманням політик. У таких умовах важливу роль відіграють автоматизовані агенти безпеки, що здатні інтегруватися у процес розробки та здійснювати аналіз коду у реальному часі. Одним із найбільш поширених рішень цього класу є Snyk — інструмент, що забезпечує комплексну перевірку коду, залежностей, контейнерів, конфігурацій інфраструктури та маніфестів.

У системах генеративного програмування інтеграція безпекових агентів має подвійне значення. По-перше, такі агенти забезпечують оцінку вразливостей згенерованого коду, за допомогою перевірки відповідності вимогам безпеки. По-друге, формують зворотний зв’язок стосовно мовної моделі, що дозволяє їй адаптувати майбутні генерації з урахуванням актуальних ризиків. У цьому контексті Snyk(рис. 3.5) є не лише інструментом аналізу, а й елементом системи управління якістю, що взаємодіє з генеративним агентом у межах спеціально створених безпечних середовищ[29].

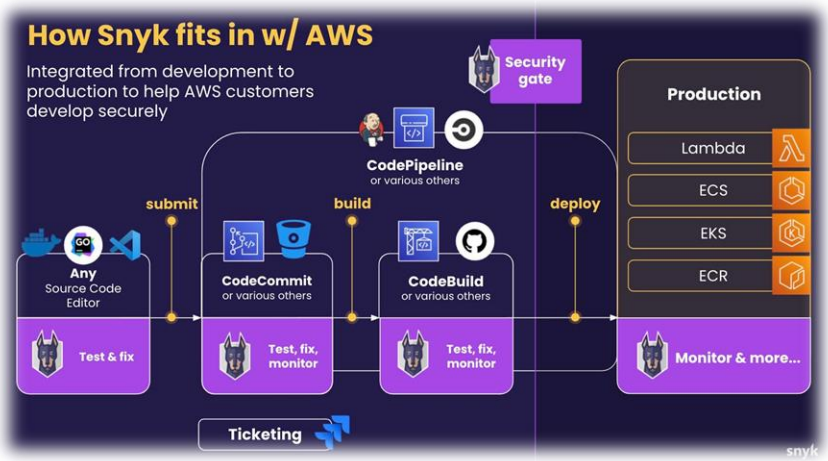

Рисунок 3.5 – Функціональність Snyk

Функціональність Snyk охоплює кілька напрямів безпеки, включно з аналізом відкритого коду, скануванням залежностей, ідентифікацією вразливостей у контейнерах, виявленням помилок конфігурації у середовищах Kubernetes та валідацією IaC (Infrastructure as Code). Такий широкий спектр дозволяє використовувати Snyk як джерело структурованих механізмів контролю, що враховують специфіку проєкту. У рамках генеративного середовища ці механізми дозволяють моделі працювати з актуальними рекомендаціями щодо виправлення відомих вразливостей, що забезпечує високий рівень передбачуваності та безпечності результатів.

Інтеграція безпекових агентів із генеративними моделями потребує формування передачі структурованих даних у форматах, що підтримують автоматичну інтерпретацію. Звіти Snyk можуть передаватися у вигляді JSON-структур, що містять інформацію щодо критичноті вразливості, шлях експлуатації, рекомендації стосовно оновлення версій залежностей, опис патчів або варіанти зміни конфігурацій. Генеративна модель, що отримала такий контекст, здатна автоматично внести корективи у вихідний код або сформувати пропозиції з оновлення залежностей. У результаті формується ітеративний механізм, де Snyk та ШІ-агент працюють як взаємопов’язані компоненти: агент генерує код, засіб безпеки виконує аналіз, після чого модель коригує результат.

У реальних умовах використання Snyk у зв’язці з GitHub Copilot або іншими LLM-плагінами дозволяє створити безпечне середовище генерації. Наприклад, коли модель генерує реалізацію API або функціональний модуль, безпековий агент здійснює миттєвий аналіз під час збереження файлу. У разі виявлення проблеми він передає структурований звіт, де зазначено специфіку вразливості та шляхи її усунення. Генеративний агент може автоматизувати оновлення небезпечних залежностей, зміну конфігурацій або модифікацію алгоритмічних рішень відповідно до політик організації. Таким чином, формується інтегрований захищений процес, що включає елементи статичного аналізу, аналізу залежностей та архітектурної відповідності.

Snyk у цьому стеку — це детектор вразливості у коді й залежностях. Якщо Sonar дивиться на структуру, Jest й Playwright — на поведінку, то Snyk — аналізує вразливості у бібліотеках, конфігураціях, Docker-файлах та самому коді. Основне завдання — під’єднати Snyk так, щоб ШІ-агент не просто знав вразливості, а міг автоматично їх виявляти, читати звіти та намагатися виправляти код відповідно до рекомендацій.

Починається все з установки Snyk CLI. Після авторизації Snyk отримує доступ до свого сервісу. У package.json варто завести окремі скрипти щодо локального аналізу:

```
"scripts": {
  "snyk:test": "snyk test --json > snyk-results.json",
  "snyk:code": "snyk code test --json > snyk-code-results.json"
}
```

Тут snyk test аналізує залежності (package.json, lock-файл, бібліотеки), а snyk code test — статичний аналіз коду на наявність патернів, що можуть призвести до

SQL injection, XSS, небезпечної десеріалізації тощо. Обидві команди формують JSON-звіти (snyk-results.json і snyk-code-results.json), що зручно читати ШІ-агенту.

Далі конфігурація агента у .copilot/agents/refactor_and_test_agent.yaml доповнюється дозволами на запуск Snyk й читання його звітів:

```
runCommands:
  allow:
    - "npm run snyk:test"
    - "npm run snyk:code"

readFiles:
  include:
    - "snyk-results.json"
    - "snyk-code-results.json"
```

Структура звіту міститиме список знайдених проблем: тип, файл, рядок, опис вразливості, рівень серйозності, ID правила, а часто й пропозиції щодо виправлення (наприклад, оновити залежність до певної версії, змінити спосіб побудови SQL-запиту, додати валідацію введення тощо).

На практиці агент буде працювати приблизно так. Після генерації нового коду або додавання нової залежності він викликає його. Потім відкриває snyk-code-results.json і відбирає всі проблеми з рівнем high та critical.

Після регенерації коду агент повторює запуск Snyk. Цикл триває, доки критичних проблем не залишиться. Таким чином, Snyk перетворюється на джерело жорстких вимог, а Copilot — на виконавця цих вимог.

Особливо корисною є інтеграція Snyk із Dev Containers. Якщо DevContainer містить Node, Snyk CLI та все необхідне щодо запуску застосунку, то аналіз виконується в однаковому середовищі як у локальному VS Code, так і у CI.

У devcontainer.json можна додати:

```
"postCreateCommand": "npm install",
"customizations": {
  "vscode": {
    "extensions": [
      "snyk-security.vscode-vuln-cost",
      "GitHub.copilot",
      "GitHub.copilot-chat"
    ]
  }
}
```

У такому разі Snyk показуватиме в IDE підсвітку вразливих місць, а агент — аналізуватиме ті ж самі звіти у JSON-форматі.

Ще один важливий рівень інтеграції — взаємодія з Notion MCP. Вимоги безпеки, політики доступу, стандарти шифрування та валідації можна зберігати у Notion як формальні документи. Агент, отримавши їх через Notion MCP, може використовувати їх як вхідні правила щодо виправлення коду. Snyk у такій конфігурації виступає не лише як детектор, а як засіб верифікації того, що код дійсно відповідає задекларованим стандартам. Наприклад, якщо у Notion прописано, що всі HTTP-запити мають проходити через централізований клієнт із

додаванням токенів та логуванням, а Snyk виявляє прямі виклики fetch без авторизації, агент може це автоматично виправити.

Разом із Sonar MCP Snyk формує повноцінний захист стосовно ШІ-генерації коду: Sonar контролює структуру та надійність, Snyk — безпеку та залежності, Jest та Playwright — функціональність, Notion MCP — відповідність вимогам. У результаті виходе не просто Copilot, що пише код, а ШІ-орієнтований конвеєр, що одразу перевіряє, чи цей код безпечний, якісний та узгоджений з політиками проєкту.

3.7 Запуск коду у віртуальному середовищі DevContainer

Однак використання безпекових агентів у контексті генеративних моделей було б неповним без забезпечення можливості тестування коду у безпечному та повністю ізольованому середовищі. Саме тому важливою складовою сучасного ШІ-орієнтованого стеку є віртуальні середовища (VM), спеціально створені для виконання та перевірки згенерованого коду. Такі середовища дозволяють запустити програмний модуль у контрольованому оточенні, де виконання коду ізолюється від системи користувача та корпоративної інфраструктури. Це особливо важливо у ситуаціях, коли код може містити потенційні логічні або інфраструктурні ризики, або коли модель створює фрагменти, що взаємодіють з файловою системою, мережею чи системними ресурсами.

Віртуальне середовище забезпечує декілька основних рівнів безпеки. По-перше, ізоляція виконання гарантує, що потенційно небезпечний код не може завдати шкоди основній системі. По-друге, контроль над залежностями дозволяє уникнути ситуацій, коли згенерований фрагмент використовує бібліотеки з вразливостями або невідповідного походження. По-третє, у межах віртуального середовища можна контролювати та записувати усі дії, виконані кодом, що дозволяє здійснювати аудит, аналіз логів та виявлення небажаних сценаріїв. Цей підхід забезпечує повну інструментальну спостережуваність та дозволяє повторювати тести у відтворюваних умовах.

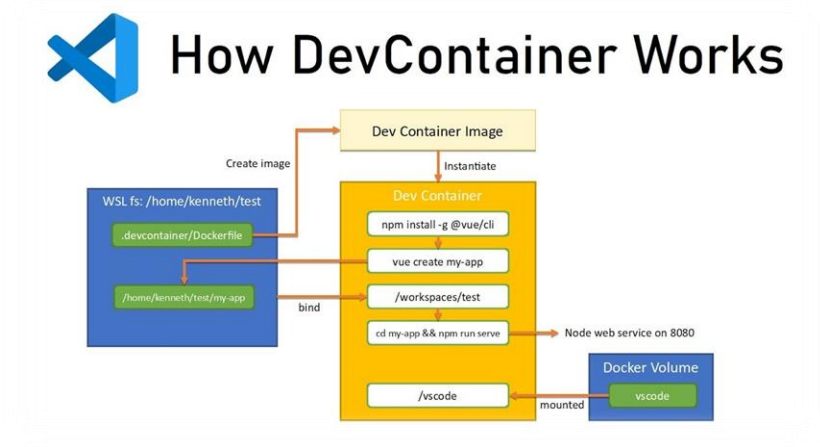

Рисунок 3.6 – Dev Containers у VS Code

Dev Containers у VS Code(рис. 3.6) — це віртуальне середовище, де все працює так само, як й було до цього без підв’язки до певного комп’ютера. Коли працюєш з Copilot, Jest, Playwright, Sonar MCP, Snyk і Notion MCP, головна загроза

стабільності — різні версії Node, різні глобальні пакети, різний стан системи у кожного розробника. Dev Container вирішує це радикально: вся розробка, запуск тестів, аналіз коду, інтеграція з MCP-сервісами виконуються всередині контейнера з чітко описаним середовищем[30].

Практично це починається зі створення папки .devcontainer у корені репозиторію й файлу конфігурації devcontainer.json. Для React + TypeScript, Jest, Playwright, Snyk та Copilot можна, наприклад, використати базовий образ Node або спеціалізований Playwright-образ.

Простий варіант на базі Node:

```
{
  "name": "React TS DevContainer with AI tooling",
  "image": "mcr.microsoft.com/devcontainers/javascript-node:20",
  "postCreateCommand": "npm install",
  "customizations": {
    "vscode": {
      "extensions": [
        "GitHub.copilot",
        "GitHub.copilot-chat",
        "Orta.vscode-jest",
        "ms-playwright.playwright",
        "SonarSource.sonarlint-vscode",
        "snyk-security.vscode-vuln-cost"
      ],
      "settings": {
        "github.copilot.workspaceInstructions":
".copilot/INSTRUCTIONS.md"
      }
    }
  },
  "remoteUser": "node"
}
```

У такому сценарії VS Code автоматично підхоплює Dev Container, встановлює залежності, тягне розширення для Copilot, Jest, Playwright, SonarLint та Snyk. Всі інструменти працюють всередині контейнера, а не на хості.

Це означає:

- однакова версія Node стосовно всіх;
- однакові npm-залежності;
- однакові шляхи, змінні середовища, доступ до MCP-серверів.

Якщо активно використовувати Playwright, доцільно відразу взяти Playwright-образ:

```
{
  "name": "React TS + Playwright DevContainer",
  "image": "mcr.microsoft.com/playwright:v1.48.0-jammy",
  "postCreateCommand": "npm install",
  "remoteUser": "pwuser",
  "customizations": {
    "vscode": {
      "extensions": [
```

```
        "GitHub.copilot",
        "GitHub.copilot-chat",
        "ms-playwright.playwright"
      ]
    }
  }
}
```

У цьому випадку у контейнері вже встановлені всі потрібні браузери й драйвери, а E2E-тести у Playwright працюють так само, як у CI. ШІ-агенти, що запускають npm run test:ci або npx playwright test, роблять це в однаковому середовищі, що не залежить від конкретного ноутбука.

З точки зору Copilot й кастомних агентів Dev Container дає кілька важливих переваг.

По-перше, середовище стає детермінованим. Коли агент запускає Jest, Playwright, Snyk або інші CLI-команди, їхня поведінка передбачувана: ті самі версії інструментів, ті самі бібліотеки, той самий Node. Це критично для сценаріїв, де агент багаторазово запускає тести й аналізатор, що робе рішення на основі результатів.

По-друге, Dev Container може бути налаштований так, щоб одразу містити конфігурацію доступу до MCP-сервісів.

Наприклад, у devcontainer.json можна додати:

```
"containerEnv": {
  "SONAR_MCP_API_KEY": "your-sonar-token-here",
  "NOTION_MCP_API_KEY": "your-notion-token-here",
  "PLAYWRIGHT_MCP_API_KEY": "your-playwright-token-here"
}
```

Тоді агенти можуть спокійно звертатися до sonar-mcp, notion-mcp, playwright-mcp, не турбуючись про розбіжності у локальних змінних оточення. Для CI можна використовувати той самий контейнерний образ, що і для розробників, — це забезпечує повну симетрію між локально працює та на CI працює.

По-третє, Dev Containers дають змогу ізолювати небезпечний або експериментальний код. Якщо ШІ-агент генерує нові фрагменти, запускає їх, виконує лінтери й аналіз коду, все це відбувається в окремому контейнері. Навіть, якщо генерований код поводиться некоректно, він не впливає на систему розробника або корпоративне середовище. Це особливо важливо, коли ШІ-агент має право запускати CLI-команди.

Крім того, Dev Container добре інтегрується з підходом, де весь стек керується через .copilot/ та MCP. Фактично ти отримуєш такий шар:

- .copilot/INSTRUCTIONS.md визначає політику;
- .copilot/prompts описує сценарії;
- .copilot/agents реалізує ці сценарії;
- .copilot/mcp.json підключає зовнішні джерела знань та аналізу;
- .devcontainer/devcontainer.json описує фізичне виконуюче середовище.

Dev Containers — це фундамент, на якому тримається вся розумна оптимізація: без стабільного середовища будь-які результати ШІ-агента будуть непередбачувані, а отже технічно ненадійні.

Синергія між Snyk та віртуальним середовищем посилює ефективність обох компонентів. Коли згенерований код проходить статичний аналіз та аналіз залежностей, після цього він виконується у контрольованому оточенні, що дозволяє підтвердити відсутність поведінкових аномалій. У випадку, коли віртуальне середовище фіксує небажані операції, такі як звернення до заборонених мережевих напрямків або доступ до несанкціонованих каталогів, відповідна інформація передається назад до генеративного агента та засобів статичного аналізу. У результаті формується комплексний цикл перевірки безпеки, що охоплює як структурні, так і поведінкові властивості коду.

У межах такого циклу можна виділити важливу властивість — ітеративну корекцію моделі. Якщо програма отримала негативний результат тестування або була позначена як небезпечна, генеративна модель використовує зворотний зв'язок щодо формування оновленої версії коду. Такий підхід створює можливість автоматизованого навчання моделі у контексті конкретного проєкту, адже результати тестування стають фактичними інструкціями стосовно корекції. Застосування цієї технології є особливо корисним у проєктах, де існують жорсткі вимоги до відповідності стандартам безпеки, наприклад, у фінансових, медичних або урядових системах.

Ще одним важливим аспектом взаємодії безпекових агентів і віртуальних середовищ є можливість виконувати динамічний аналіз. На відміну від статичного аналізу, що працює зі структурою коду, динамічний аналіз оцінює реальну поведінку під час виконання. Це дозволяє виявляти помилки, пов’язані з обробкою даних, некоректним використанням пам’яті, доступом до некоректних ресурсів або логічними дефектами. Snyk може виконувати частину цих перевірок, але у поєднанні з віртуальним середовищем вони набувають більшої ефективності, оскільки кожен результат виконання одразу перетворюється на дані для моделі.

У процесі генерації коду такі механізми дозволяють формувати підхід, у якому модель не лише створює функціональні фрагменти, а й проходить автоматизовану валідацію без втручання користувача. Розробник отримує лише кінцеву версію, що відповідає як архітектурним, так і безпековим вимогам. Це радикально зменшує кількість ручних перевірок, знижує ризики людських помилок і забезпечує відповідність внутрішнім регламентам розробки.

У підсумку застосування безпекових агентів, таких як Snyk, у поєднанні з ізольованими віртуальними середовищами формує багаторівневу систему контролю безпеки, що інтегрується у генеративний процес та забезпечує високий рівень надійності програмного продукту. Така система дозволяє скоротити час розробки, покращити якість коду, забезпечити відповідність нормативним вимогам і мінімізувати ризики експлуатації вразливостей. Генеративні моделі, які отримують зворотний зв'язок від безпекових засобів та тестових середовищ, здатні адаптувати поведінку й формувати безпечний код не лише як реакцію на поточні вимоги, а також у перспективі.

3.8 Повний контроль агента над IDE та паралелізація його роботи

У сучасних інструментах програмної інженерії значного поширення набуває підхід, що передбачає інтеграцію генеративних моделей ШІ як активних агентів, що мають можливість здійснювати повноцінну взаємодію з середовищем розробки. На відміну від класичних систем автодоповнення, що працювали у форматі реакції на запити користувача, сучасні ШІ-агенти отримують доступ до інструментів IDE, структури проєкту, систем аналізу коду та виконання команд, що відкриває новий етап розвитку автоматизації програмування. Повний контроль агента над IDE слід розуміти не як автономне втручання у роботу розробника, а як можливість виконання цілеспрямованих дій, які раніше вимагали ручних операцій: створення нових файлів, модифікація існуючих фрагментів, запуск тестів, виконання рефакторингу, оновлення конфігурацій або взаємодія з внутрішніми інструментами командної інфраструктури.

Особливість такого підходу полягає у здатності агента працювати з повною моделлю проєкту, а не лише з вмістом окремого файлу. Це означає, що агент отримує доступ до інформації щодо залежностей, архітектурної структури, файли конфігурацій, середовище виконання та метадані. У результаті йому доступні операції, що традиційно виконувалися вручну: створення нових каталогів відповідно до певної архітектурної схеми, корекція шляхів імпорту, оновлення документації, модифікація конфігураційних файлів залежно від зміни компонентів, забезпечення узгодженості між типами та інтерфейсами, а також формування допоміжних структур, таких як тести чи приклади використання. Така взаємодія дозволяє перейти від генерації незалежних фрагментів коду до створення цілісних компонентів, що відповідають вимогам системного дизайну.

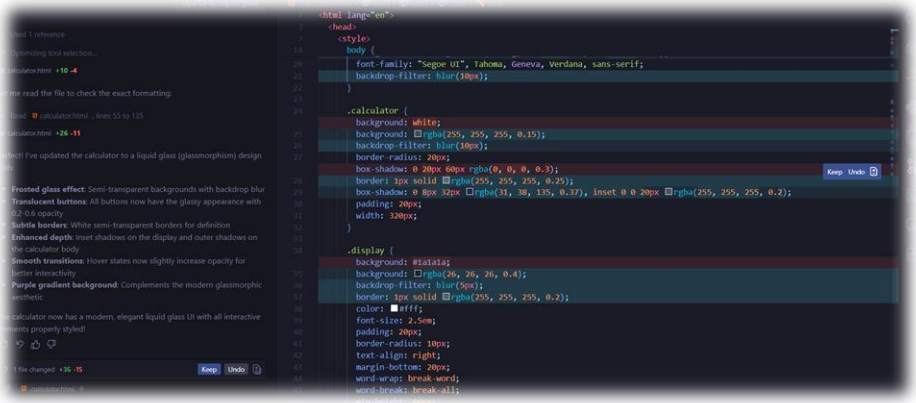

Рисунок 3.7 – Режим агента в IDE

У межах повного доступу до IDE агент(рис. 3.7) отримує також можливість аналізувати результати виконання коду та тестів, взаємодіяти з внутрішніми інструментами статичного аналізу й орієнтуватися на ці результати під час подальшої генерації. У цьому контексті ШІ-модель вже не є автономним генератором тексту, а виступає як частина замкненого циклу розробки, де кожен етап генерує нові дані, що впливають на поведінку агента. Цей механізм є

особливо важливим у великих проєктах, де коректність інтеграції нового коду значно залежить від відповідності внутрішнім стандартам та взаємодії з іншими компонентами.

Ключовим аспектом цього підходу є забезпечення безпеки, оскільки доступ агента до файлової системи й інструментів IDE відкриває ризики виконання небажаних операцій. Тому сучасні реалізації використовують обмежувальні політики, механізми цифрового підпису агентів та визначені переліки дозволених дій. Така модель дає змогу гарантувати, що агент виконує лише ті операції, що відповідають політиці проєкту, не змінює критично важливі файли без підтвердження та дотримується стилістичних й архітектурних норм. Цей підхід забезпечує керовану автоматизацію, що не замінює програміста, а виконує рутинні або структурні операції з високим рівнем повторюваності та точності.

Коли інструкції, prompt-файли, агенти, Jest, Playwright, Sonar MCP, Snyk, Notion MCP та Dev Containers вже налаштовані, наступний крок — організувати їхню взаємодію так, щоб ШІ-агент міг не лише відповідати на запити, а й сам керувати частиною робочого процесу у VS Code. Під контролем у цьому контексті мається на увазі не довільний доступ до всієї системи розробника, а чітко регламентований набір можливостей: читати файли, змінювати їх у межах репозиторію, запускати обмежений набір CLI-команд, аналізувати результати, викликати MCP-сервіси й, у разі потреби, ініціювати додаткові дії.

Технічно це досягається за рахунок конфігурації кастомних агентів у .copilot/agents/*.yaml, де явно задаються дозволені дії.

Приблизно це виглядає так:

```
    name: "full_cycle_agent"
    description: "Agent that can modify code, run tests, analyze quality, update docs."

    runCommands:
      allow:
        - "npm run test:ci"
        - "npm run test:e2e:ci"
        - "npm run snyk:code"

    readFiles:
      include:
        - "src/**/*.ts"
        - "src/**/*.tsx"
        - "tests/**/*.ts"
        - "jest-results.json"
        - "playwright-report/**/*.json"
        - "snyk-code-results.json"

    writeFiles:
      include:
        - "src/**/*.ts"
        - "src/**/*.tsx"
        - "tests/**/*.ts"
```

Таким чином, агент отримує повноваження працювати із кодом та тестами, але не може, наприклад, змінювати Docker-файли, секрети чи системні налаштування. Це той самий повний контроль над IDE, але у рамках чітко окресленого периметру безпеки.

В умовах розробки програмних систем агент, що має повний контроль над IDE, може виступати інструментом щодо впровадження стандартизованих процедур розробки. Це включає автоматизоване створення шаблонів файлів, адаптацію структур даних до архітектурних моделей, виконання змін у конфігураційних файлах та забезпечення дотримання внутрішніх правил групи розробників. Підхід інтегрованої автоматизації сприяє формуванню єдиної архітектурної логіки у межах проєкту, зменшує кількість ручних операцій та підтримує сталість стилю.

Окремим напрямом розвитку інструментів генеративного програмування є паралелізація процесу генерації коду. У традиційній моделі LLM опрацьовує запити послідовно, що обмежує швидкість обробки великих проєктів або складних завдань. Паралелізація дозволяє розділити задачу на множину підзадач, що аналізуються й генеруються різними потоками або різними екземплярами моделі. У практиці розробки це дозволяє одночасно формувати кілька компонентів, проводити рефакторинг різних файлів, створювати набори тестів або оновлювати документацію без необхідності чекати завершення попередніх операцій.

Реалізація цього підходу вимагає формування стратегії консистентності, оскільки паралельні агенти повинні працювати у єдиному інформаційному просторі, не створювати конфліктів між змінами та забезпечувати узгодженість кінцевого результату. З цією метою використовуються механізми контролю версій, динамічного блокування файлів, порівняння структур та детермінованої регенерації. У складних проєктах паралелізація дає змогу значно скоротити час генерації складних модулів, а також підвищити точність й повторюваність структури коду.

Технічна реалізація паралелізації передбачає розподіл функціональності між агентами, що працюють над різними частинами проєкту, але синхронізуються через централізований контекстний механізм. Цей механізм може використовувати Model Context Protocol, внутрішні індекси IDE або проміжні структуровані файли. Агент, що працює над власним підзавданням, отримує лише релевантний контекст, але має можливість запитувати додаткові дані у разі зміни залежностей. Так формується схема, де кожен агент здійснює локальну генерацію, а глобальна система забезпечує узгодження результатів та їхню інтеграцію у спільний репозиторій.

Паралелізація також передбачає можливість оптимізації генерації через накопичення проміжних станів. Наприклад, агент може зберігати результат аналізу певного модуля й передавати його іншим агентам, що працюють із пов’язаними компонентами. Це зменшує кількість повторних обчислень та дозволяє моделі працювати у режимі інкрементальної генерації. Додатково паралельний режим дозволяє виконувати складні комплексні операції, такі як

глобальний рефакторинг, що включає одночасні зміни у десятках файлів, з урахуванням залежностей та вимог до архітектурної узгодженості.

Паралелізація починається тоді, коли таких агентів кілька, й кожен відповідає за свою частину сценарію. Один агент можна налаштувати як “рефакторинг + якість”. Він читає Sonar MCP, переписує код, стежить за Code Smell. Другий — тестовий. Після змін запускає Jest та Playwright, аналізує JSON-звіти, повертає стислий опис проблем щодо моделі. Третій — документаційний. За змінами у коді оновлює сторінки у Notion через Notion MCP. У VS Code вони можуть викликатися окремо (/agent refactor, /agent test, /agent docs), але логічно їх можна з’єднати в один конвеєр, де результати роботи одного агента є вхідними даними для іншого.

У практиці це виглядає як набір сценаріїв використання. Наприклад, розробник змінює бізнес-вимоги у Notion. Документаційний агент помічає нову або оновлену сторінку (або отримує від користувача команду) й ініціює генерацію змін у коді через рефакторинговий агент, що, у свою чергу, після змін викликає тестового агента. Тестовий запускає Jest та Playwright у DevContainer, збирає результати, за потреби повертає їх назад у LLM як контекст: “Змінив ось це, зламалося ось це, ось лог”. Потім перший агент продовжує цикл, доки система знову не прийде у стан: “Усі тести зелені, smell-и у межах норми, вимоги виконані”.

З технічної точки зору паралелізація полягає не тільки в одночасному запуску кількох процесів, а й у логічному розділенні відповідальностей. Один агент може опрацьовувати лише один модуль, інший — інший модуль, а третій — суто оновлювати документацію. На великих проєктах це дає змогу масштабувати роботу ШІ: замість одного універсального агента, що робить все, кілька спеціалізованих агентів, що працюють у своїх зонах й обмінюються результатами через файли, MCP-виклики та контексти до LLM.

Усе це залишається контрольованим завдяки Dev Container-оточенню. Навіть, якщо один агент запускає тести, інший — аналіз коду, третій — Snyk, вони всі працюють у тому самому контейнері з однаковими версіями інструментів. Це дає впевненість у тому, що результати відтворювані. А обмеження на runCommands, readFiles та writeFiles гарантують, що повний контроль ШІ-агента над IDE насправді є суворо типовим та передбаченим сценарієм, а не неконтрольованим доступом до середовища розробника.

У такій конфігурації VS Code перетворюється на платформу, де Copilot та MCP-аналітика — це мозок, агенти — руки, Dev Container — тіло. Код не лише генерується, а проходить цикл якості, безпеки, тестування та документування, де кожний етап виконується напівавтоматично або повністю автоматизовано за участю ШІ. Практична реалізація повного контролю ШІ-агента над IDE у прийнятному стосовно промислової розробки вигляді — чітко описана, відтворювана та керована система інструментів та сценаріїв.

Паралелізація у поєднанні з повним доступом агента до IDE формує нову модель розробки, де значна частина операцій виконується автоматично й узгоджується у режимі реального часу. Це дозволяє скоротити час розробки,

підвищити якість коду, зменшити обсяг рутинної ручної праці та знизити ризик людських помилок.

Таким чином, створюється передумова для поступового переходу від традиційних методів програмування до інженерних процесів, де ШІ виступає не допоміжним елементом, а активним агентом управління інфраструктурою проєкту.

3.9 Реалізація оптимітизації автоматичного написання програмного коду

Встановити сучасну версію Visual Studio Code з підтримкою розширень, оскільки саме розширення є механізмом, через який реалізується робота Copilot, Snyk, Dev Containers та інструментів статичного аналізу. Після встановлення VS Code варто увімкнути синхронізацію налаштувань, щоб зберігати конфігурації у хмарі та мати можливість переносити підготовлене середовище на інші машини.

Після інсталяції VS Code необхідно встановити розширення GitHub Copilot. Після входу в обліковий запис GitHub інструмент активується автоматично. Щоб надати Copilot можливість працювати не у загальному режимі, а підпорядковуватися визначеним правилам, потрібно створити у корені проєкту спеціальну директорію: copilot/. У цій директорії зберігатимуться всі кастомні інструкції, промпт-файли та специфікації агентів.

Щоб Copilot дотримувався правил архітектури й стилю, створюється файл:.copilot/INSTRUCTIONS.md.

У цьому файлі потрібно чітко описати:

- стиль та правила форматування;
- структуру класів й модулів, що є обов’язковими;
- правила обробки помилок;
- вимоги до логування;
- обов’язкові компоненти тестового покриття;
- шаблони імпорту;
- вимоги до використання залежностей.

Після створення цього файлу Copilot починає читати його під час кожної генерації. Таким чином, він не створює код як вважає за потрібне, а підпорядковується локальним правилам.

Для завдань, що повторюються (створення компонентів, тестів, сервісів), створюються промпт-файли: copilot/prompts/.

Наприклад: copilot/prompts/create_service.md.

У ньому описується:

- як повинен виглядати типовий сервіс;
- яку структуру каталогів він має створити;
- які залежності можна використовувати;
- який тест повинен бути створений разом із сервісом.

Після цього користувач може викликати Copilot та написати: “Створи новий сервіс згідно з локальним шаблоном”. Модель використовуватиме описаний файл.

Агенти працюють як автономні міні-програми всередині VS Code.

Для цього створюється файл: .copilot/agent.yaml. Агент починає діяти як додаткова рука Copilot, що отримує контроль над IDE. Де описуються дозволені дії. Наприклад: читання файлів, запис у файлову систему, виконання команд типу npm install, запуск тестів, читання структури каталогів, виконання статичного аналізу.

Після входу в обліковий запис Snyk інструмент автоматично починає аналізувати:

- залежності,
- Dockerfile,
- Kubernetes YAML,
- інфраструктурні файли,
- вихідний код.

Результати аналізу з'являються у панелі Problems, а також можуть бути передані назад Copilot через агента або MCP. Snyk фактично працює як сенсор безпеки для ШІ.

Для ізольованого виконання коду встановлюється розширення: Dev Containers (або Docker). Після цього створюється файл: .devcontainer/devcontainer.json.

У цьому файлі визначаються: образ контейнера, дозволені операції, залежності, тестові інструменти та обмеження (без доступа до всієї машини).

Це дозволяє перевірити: логічні помилки, небажані системні виклики, доступ до мережі, некоректні операції файлової системи.

Результати виконання передаються назад до агента.

Створюється конфігурація MCP-клієнта: .copilot/mcp.json. У ній описуються джерела даних, їхні адреси, формати та дозволені дії. Після цього Copilot може отримувати актуальні дані динамічно, а не лише з відкритого файлу.

MCP дає Copilot можливість звертатися до зовнішніх ресурсів: документації API, CI/CD систем, баз знань, внутрішніх сервісів організації, аналітичних інструментів, баз даних, систем логування.

VS Code автоматично підтримує багатопроцесну модель щодо розширень. Щоб паралелізувати роботу агентів необхідно:

- Кожен агент описується в окремому YAML-файлі;
- Кожен агент працює у власному робочому процесі;
- Обмін даними здійснюється через структуровані інтерфейси (MCP або локальні паки даних);
- Синхронізація виконується за правилами конфігурації, що запобігають конфліктним змінам.

Таким чином, можлива ситуація, коли: один агент створює структуру модуля, другий аналізує залежності, третій генерує тести, а четвертий виконує рефакторинг існуючих файлів.

Це створює фактично фабрику з генерації коду.

Після налаштування всіх механізмів робоче середовище VS Code стає:

- стандартизованим — через інструкції;

- шаблонним — через промпт-файли;
- автономним — через кастомних агентів;
- безпечним — через Snyk;
- ізольованим — через Dev Containers;
- контекстно динамічним — через MCP;
- масштабованим — через паралельну генерацію.

У цьому стані VS Code фактично перетворюється на інтелектуальну IDE, де ШІ не лише генерує код, а й бере участь у тестуванні, валідації, рефакторингу, перевірці безпеки та архітектурній узгодженості.

Висновки

Реалізований комплекс оптимізацій демонструє, що сучасні інструменти генеративного ШІ здатні істотно підвищити якість, стабільність та швидкість розробки ПЗ за умови правильної інтеграції у середовище розробки. Поєднання можливостей GitHub Copilot із системою кастомних інструкцій та prompt-файлів дозволяє формувати кероване та передбачуване середовище генерації коду, де кожен етап (від написання компонентів до створення тестів і документації) підпорядкований визначеним правилам. Використання кастомних агентів у форматі YAML створює інфраструктуру, де ШІ може виконувати окремі операції над кодом, тестами та документацією, дотримуючись чітко регламентованих обмежень та політик безпеки.

MCP-інтеграції стали ключовим елементом побудови інтелектуального конвеєра. Sonar MCP забезпечує доступ до метрик якості, що перетворюють статичний аналіз на керівний механізм щодо рефакторингу. Playwright MCP дозволяє автоматизувати E2E-тести й робити поведінкову перевірку частиною циклу генерації коду. Notion MCP створює повноцінний двосторонній канал між кодом та документацією, інтегруючи вимоги, архітектурні правила та технічні описи у процес генерації. Інструменти безпеки, такі як Snyk, у поєднанні з агентом Copilot формують захисний контур проти впровадження небезпечного коду або вразливих залежностей, що робе безпеку невід’ємною частиною робочого циклу.

Використання Jest та Playwright у Dev Containers забезпечує стабільність виконання тестів незалежно від локальних налаштувань розробників. Dev Containers виступають у ролі універсального виконуючого середовища, де Copilot, агенти, MCP-служби та інструменти аналізу працюють однаково та локально. Це створює передумови щодо повної відтворюваності результатів та мінімізації середовищних помилок, що традиційно становлять значну частину витрат на підтримку проєкту.

Паралелізація роботи агентів у VS Code дозволяє досягти фактичної автоматизації цілого ланцюга перевірок: генерація коду переходить у рефакторинг, потім у модульне тестування, далі в E2E-перевірку, у статичний аналіз якості та, нарешті, в оновленні документації. Цей цикл стає керованим, модульним та повторюваним. Завдяки цьому, ШІ-орієнтована розробка перестає бути набором хаотичних підказок та перетворюється на інженерний процес зі зрозумілою структурою, правилами та точками контролю.

Проведена інтеграція демонструє, що продуктивність команди може істотно зрости не лише за рахунок швидшої генерації коду, а саме завдяки автоматизації другорядних дій: написання тестів, контролю якості, оновлення документації, виявлення вразливостей та корекції помилок. Підхід із MCP-сервісами робить ШІ-агента частиною інженерного сценарію, а не окремим інструментом. Таким чином, підвищується рівень стандартизації, зменшується кількість людських помилок, а загальна архітектурна цілісність системи краще підтримується з часом.

Створена конфігурація може розглядатися як модель сучасного ШІ-орієнтованого середовища розробки, що поєднує найкращі практики програмної інженерії з можливостями генеративних моделей. Вона забезпечує контрольовану, безпечну й відтворювану взаємодію між людиною, кодом та ШІ. Таким чином, підтверджується, що глибока інтеграція Copilot, MCP-сервісів, тестових інструментів та Dev Containers формує ефективну технічну інфраструктуру, що здатна підтримувати розробку ПЗ на високому рівні якості та надійності.

4 АНАЛІЗ ТА ПРОГНОЗУВАННЯ ТРЕНДІВ ШТУЧНОГО ІНТЕЛЕКТУ У КОДУВАННІ

4.1 Апроксимація трендів

Технологія трендів є потужним інструментарієм для дослідження закономірностей розвитку різноманітних процесів у часі. Вона охоплює методи апроксимації, аналізу та прогнозування, дозволяючи виявляти основні напрямки змін, оцінювати їх стабільність та будувати обґрунтовані прогнози на майбутнє. Цей підхід знаходить широке застосування в економіці, фінансах, маркетингу, соціології, техніці та багатьох інших галузях.

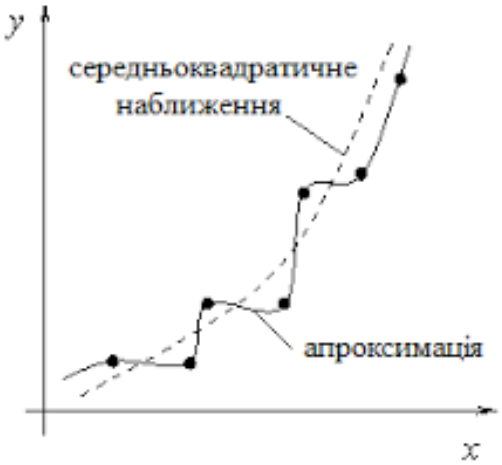

Рисунок 4.1 – Апроксимація трендів

В основі технології лежить поняття тренду(рис. 4.1) – довгострокової тенденції зміни показника, очищеної від випадкових коливань та сезонних ефектів. Апроксимація тренду полягає у підборі математичної функції (моделі тренду), що найкращим чином описує цю основну тенденцію у наявних даних часового ряду[31].

Вибір конкретного типу функції залежить від характеру даних та природи досліджуваного процесу. З метою оцінки параметрів моделі тренду найчастіше використовується метод найменших квадратів (МНК), що мінімізує суму квадратів відхилень фактичних значень ряду від теоретичних, розрахованих за моделлю.

Прогнозування є логічним завершенням апроксимації та аналізу трендів. Воно полягає в екстраполяції виявленої закономірності на майбутні періоди.

Основні етапи прогнозування на основі трендів:

1. Побудова та верифікація моделі тренду**.** На основі ретроспективних даних будується адекватна математична модель тренду.

2. Розрахунок точкових прогнозів**.** Підставляючи у рівняння тренду майбутні значення часу (t), отримують прогнозні значення показника.
3. Розрахунок інтервальних прогнозів**.** Оскільки прогнози завжди мають певну невизначеність, розраховуються довірчі інтервали, що з заданою ймовірністю будуть містити майбутні фактичні значення показника. Ширина довірчого інтервалу залежить від точності моделі та горизонту прогнозування.

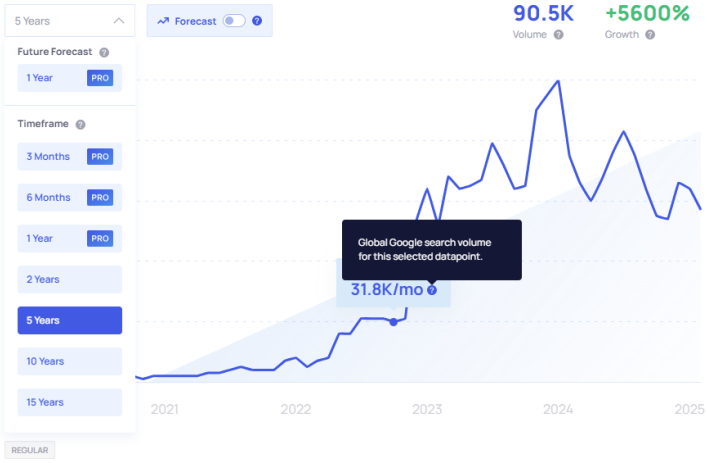

Рисунок 4.2 – Можливості аналізу тренду

Сучасні технології значно розширюють можливості аналізу трендів(рис. 4.2). До них належать:

- Статистичні пакети програм Наприклад, R, Python з бібліотеками Pandas, NumPy, SciPy, Statsmodels, MS Excel з функціями аналізу даних. Вони надають інструменти побудови різноманітних моделей трендів, їх статистичного аналізу та візуалізації.
- Спеціалізоване програмне забезпечення для прогнозування. Наприклад, Facebook Prophet, EViews, Statistica. Ці програми часто містять більш складні алгоритми, що враховують сезонність, свята та інші фактори.
- Методи згладжування часових рядів. Наприклад, ковзні середні, експоненційне згладжування. Дозволяють виділити тренд шляхом усунення випадкових коливань.
- Методи машинного навчання. Наприклад, регресійні моделі, нейронні мережі. Можуть використовуватися з метою виявлення складних нелінійних трендів та врахування великої кількості факторів.
- Інструменти візуалізації даних. Допомагають наочно представити часові ряди, виявлені тренди та прогнозні значення.

Незважаючи на обмеження, апроксимація, аналіз та прогнозування на основі технології трендів залишаються важливими та широко використовуваними інструментами стосовно прийняття обґрунтованих рішень в умовах невизначеності. З метою підвищення якості прогнозів часто поєднують трендові моделі з іншими методами, зокрема експертними оцінками та факторним аналізом.

4.2 Комплексний аналіз бенчмарків моделей у кодуванні

Розглянемо найпоширеніші бенчмарки(рис. 4.3), що використовуються з метою оцінки моделей ШІ-генерації коду, методології, що вимірюють, а також сильні та слабкі сторони.

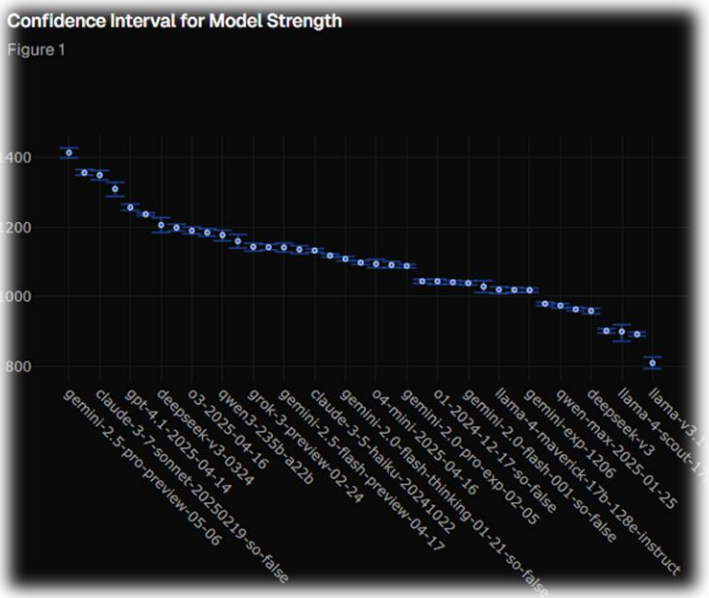

Рисунок 4.3 – Рейтинг моделей генеративного ШІ

HumanEval - функціональна коректність та метрика **(**pass@k)[32].

Розроблений OpenШІ, HumanEval є одним з найвідоміших бенчмарків щодо оцінки можливостей генерації коду. Він складається з 164 вручну створених програмних задач, кожна з яких містить сигнатуру функції, рядок документації, що пояснює очікувану поведінку, та набір модульних тестів для перевірки коректності згенерованого коду.

Методологія. Моделі генерують тіло функції на основі рядка документації та сигнатури. Згенерований код потім запускається проти заздалегідь визначених модульних тестів з метою перевірки функціональної коректності.

Метрика оцінювання (pass@k). Ця метрика вимірює ймовірність того, що принаймні одне з *k* найкращих згенерованих моделлю рішень успішно пройде всі тестові випадки. Зазвичай використовуються метрики: pass@1, pass@10 та pass@100. Метрика pass@1 є особливо суворою, що вимірює, чи є перша спроба правильною.

Переваги. Безпосередньо оцінює функціональну коректність та використовує модульне тестування, що є стандартною практикою розробки ПЗ. Забезпечує стандартизовану основу порівняння моделей.

Обмеження. Задачі є ізольованими функціями, а не повними програмами, що обмежує застосування у реальному світі. Деякі задачі можуть бути вирішені шляхом запам'ятовування, а не справжнього вирішення проблем. Здебільшого призначений для Python.

MBPP (Mostly Basic Python Problems) - основи програмування.

Цей бенчмарк складається з приблизно 1000 краудсорсингових задач з програмування на Python, призначені для початківців-програмістів, що охоплюють основи та функціональність стандартної бібліотеки[33].

Методологія. Кожна задача включає опис завдання, рішення коду та три автоматизовані тестові випадки. Моделі оцінюються за їхньою здатністю генерувати коректний код Python.

Сильні сторони. Фокусується на фундаментальних навичках Python, що робить його гарним показником базової майстерності кодування.

Обмеження. Подібно до HumanEval, може не повністю охоплювати складні сценарії розробки ПЗ у реальному світі.

SWE-Bench - виправлення реальних помилок й агентне кодування[34].

На відміну від HumanEval, SWE-Bench (представлений OpenШІ) зосереджується на виправленні реальних помилок, отриманих з відкритих репозиторіїв, що робить його дуже реалістичною оцінкою здатності LLM до кодування.

Методологія. LLM надається контекст, що мав би розробник, включаючи опис проблеми (з GitHub issues), код з помилкою та очікувані модифікації. Метою є генерація патча, що вирішує проблему без внесення нових помилок.

Метрики оцінювання. Оцінюється на основі коректності патча, інтеграції коду (безшовної інтеграції в існуючу кодову базу) та ефективності виправлень (оптимізовані, відповідно до найкращих практик). Часто повідомляється як "% вирішених".

Переваги. Висока практичність завдяки реальним проблемам, перевіряє розуміння та модифікацію існуючого коду (критична навичка) та заохочує можливості налагодження. Це більш реалістична оцінка навичок розробки ПЗ.

Обмеження. Вимагає контекстного розуміння цілих проєктів, з чим деякі LLM все ще борються. Оцінку важче автоматизувати через складність.

LiveCodeBench - комплексна оцінка без забруднення[35].

Комплексний бенчмарк для оцінки без забруднення, що постійно збирає нові задачі з платформ змагального програмування (LeetCode, AtCoder, CodeForces).

Методологія. Збирає задачі з поточних змагань, щоб забезпечити актуальність та запобігти забрудненню даних (оцінює моделі на задачах, що випущені після дати відсікання їхнього навчання). Оцінює LLM за чотирма сценаріями: генерація коду (з природної мови), самовідновлення (виправлення некоректного коду), виконання коду (запуск коду на вхідних даних) та прогнозування вихідних даних тесту (прогнозування вихідних даних щодо заданих вхідних даних).

Метрика оцінювання. Здебільшого використовується Pass@1.

Переваги. Вирішує проблеми забруднення даних, забезпечує комплексну оцінку різних компетенцій кодування та включає збалансовані рівні складності.

Обмеження. Новий бенчмарк, тому історичні дані для всіх моделей можуть бути менш обширними, ніж для HumanEval або MBPP.

WebDev Arena(рис. 4.4) платформа оцінки моделей у створенні веб-додатків у реальному часі, де користувачі голосують за кращий результат між моделями.

Методологія. Порівняння двох згенерованих моделями веб-додатків на одну і ту ж задачу, голосування користувачів, ранжування моделей за їх якістю.

Переваги. Оцінка моделей у реальних умовах створення додатків, враховує UI/UX, інтерактивність і функціональність.

Обмеження. Оцінка залежить від суб'єктивного голосування користувачів, фокусується на веб-розробці.

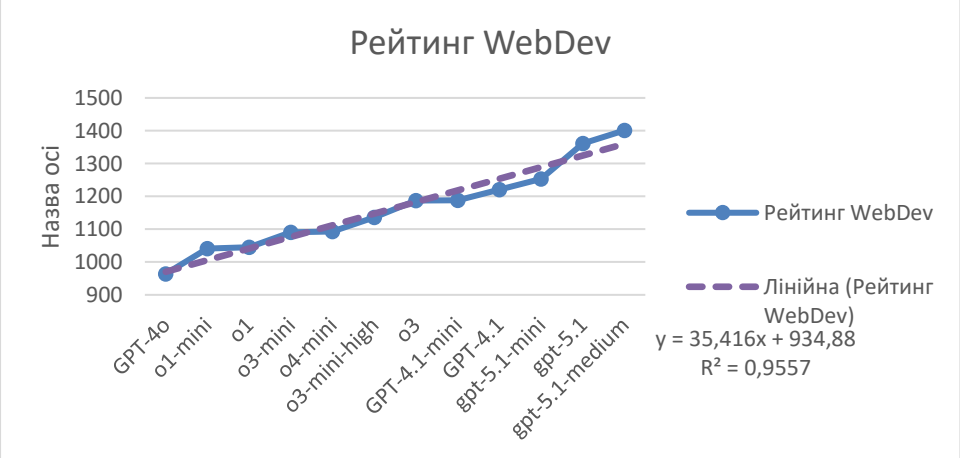

Рисунок 4.4 – Тенденція росту продуктивності генеративного ШІ

Прогрес бенчмарків від HumanEval/MBPP до SWE-Bench та LiveCodeBench свідчить про глибоку зміну у визначенні інтелекту коду щодо LLM. Він переходить від простої генерації синтаксично та функціонально коректних ізольованих фрагментів коду до демонстрації комплексних навичок розробки програмного забезпечення, включаючи розуміння складних контекстів проєкту, налагодження, самокорекцію, багатоетапну взаємодію та багатомовну майстерність. Це означає, що очікування галузі від ШІ-асистентів для кодування швидко розширюються до повноцінних ШІ-агентів з розробки ПЗ.

Ця тенденція виявляється у переході від HumanEval та MBPP, що зосереджуються на функціональній коректності ізольованих функцій, до SWE-Bench, який вимірює виправлення реальних помилок та розуміння й модифікацію існуючого коду. Ця послідовність вдосконалень бенчмарків показує, що моделі ШІ тепер оцінюються не лише за їхньою здатністю писати код, а й за їхньою здатністю діяти як інтелектуальні помічники, які можуть розуміти, налагоджувати, модифікувати та співпрацювати у складних програмних проєктах, що відображає зростаючу зрілість галузі та зростаючі вимоги до ШІ у реальних сценаріях розробки.

Хоча спеціалізовані моделі (наприклад, Code Llama, DeepSeek Coder) спочатку були розроблені з метою досягнення високих результатів у завданнях кодування, новітні універсальні LLM (наприклад, GPT-4.x, Claude 3.x, Gemini 2.x) демонструють все більш конкурентоспроможну, а іноді й вищу продуктивність у бенчмарках кодування. Це свідчить про те, що досягнення у базовій архітектурі LLM та загальних можливостях міркування є переносними та дуже корисними для генерації коду, стираючи межі між спеціалізованим та загальним ШІ в цій галузі.

Ця тенденція виявляється у кількох ключових аспектах. По-перше, спостерігається стратегічне розширення контекстних вікон, що дозволяє моделям обробляти цілі кодові бази. Наприклад, Gemini 2.5 Pro пропонує контекстне вікно в 1 мільйон токенів, а Llama 4 Scout — до 10 мільйонів токенів. Це не просто збільшення обсягу даних, а фундаментальний зсув у тому, як ШІ може розуміти та взаємодіяти з програмними проєктами на архітектурному рівні, що відкриває можливості для великомасштабного рефакторингу та складного налагодження.

По-друге, зростає акцент на агентних можливостях та багатоетапній взаємодії. Моделі тепер оцінюються не лише за одноразовою генерацією коду, а й за їхньою здатністю автономно планувати, виконувати, налагоджувати та ітерувати завдання. Це підтверджується покращеннями в бенчмарках, таких як SWE-Bench, де моделі, що використовують спеціальні каркаси або режими мислення, демонструють значно вищі показники. Отже, ШІ переходить від простого інструменту генерації до активного партнера у вирішенні проблем.

4.3 Ключові спостереження та майбутні напрямки

Насичення бенчмарків та їхня еволюція. HumanEval та MBPP, які колись були золотим стандартом, тепер значною мірою адаптовано провідними моделями, що робить їх менш корисними стосовно диференціації найсучасніших можливостей.

Це призвело до розробки нових, більш складних бенчмарків, що зосереджуються на реальних сценаріях, таких як виправлення помилок (SWE-Bench), багатомовне редагування (AIder Polyglot) та комплексні сценарії кодування (LiveCodeBench)[36].

Еволюція бенчмарків є прямою відповіддю на швидкий прогрес ШІ. Оскільки моделі швидко опановують простіші завдання, розробникам доводиться створювати більш складні, багатогранні та комплексні виклики, щоб продовжувати ефективно вимірювати та стимулювати справжні можливості ШІ.

Зростання агентних можливостей. Моделі ШІ все частіше розробляються для функціонування як агенти, здатні планувати, виконувати, тестувати та самокоригуватися. Багатоетапна взаємодія значно покращує синтез програм, що дозволяє більш природні та ефективні діалоги з ШІ щодо складних завдань кодування.

Ця трансформація перетворює LLM з пасивних генераторів коду на активних вирішувачів проблем. Здатність моделей самостійно писати, редагувати та виконувати код зі складними можливостями міркування та усунення несправностей (Claude 3.5 Sonnet) або використовувати ітеративний підхід до вирішення проблем: написання коду, його перевірка, виправлення помилок та повторення (ШIder Polyglot) свідчить щодо фундаментального зсуву. Отже, ШІ переходить від підказок до виконавця, здатного до автономного виконання завдань та виправлення помилок, що значно вплине на робочі процеси розробників, що надає можливість перекладати більш складні завдання на ШІ.

Динаміка між відкритим та закритим вихідним кодом. Комерційні моделі (OpenШІ, Anthropic, Google) часто лідирують за піковою продуктивністю,

особливо стосовно складних завдань міркування та агентних завдань. Вони можуть пропонувати унікальні функції, такі як розширене мислення або розширені мультимодальні можливості.

Моделі з відкритим вихідним кодом (Meta Llama, DeepSeek, BigCode) швидко скорочують розрив у продуктивності, пропонують конкурентоспроможні результати, великі контекстні вікна та дозвільні ліцензії для дослідницького та комерційного використання. Сприяють покращенням, керованим спільнотою.

Зростаюча конкурентоспроможність моделей з відкритим вихідним кодом створює динамічну напругу, що приносить користь всій екосистемі ШІ-кодування. Хоча моделі із закритим вихідним кодом часто встановлюють початкові стандарти, моделі з відкритим вихідним кодом, такі як Code Llama та DeepSeek-Coder-V2, швидко наздоганяють й навіть перевершують деякі закриті моделі за певними бенчмарками. Ця конкуренція стимулює інновації у всіх напрямках, що спонукає як комерційних розробників, так і розробників з відкритим вихідним кодом до вдосконалення. Наслідком є підвищена доступність та вибір для розробників, що сприяє створенню більш динамічної екосистеми, де вартість, конфіденційність та можливість налаштування стають ключовими відмінностями поряд з необробленою продуктивністю.

Розширення контекстного вікна(рис. 4.5). Постійне збільшення розмірів контекстних вікон (наприклад, 1 мільйон токенів для Gemini 2.5 Pro та GPT-4.1, 10 мільйонів для Llama 4 Scout) є критичним фактором щодо реальних завдань кодування. Це дозволяє моделям обробляти цілі кодові бази, розуміти складні залежності та виконувати великомасштабні рефакторинги або виправлення помилок більш ефективно.

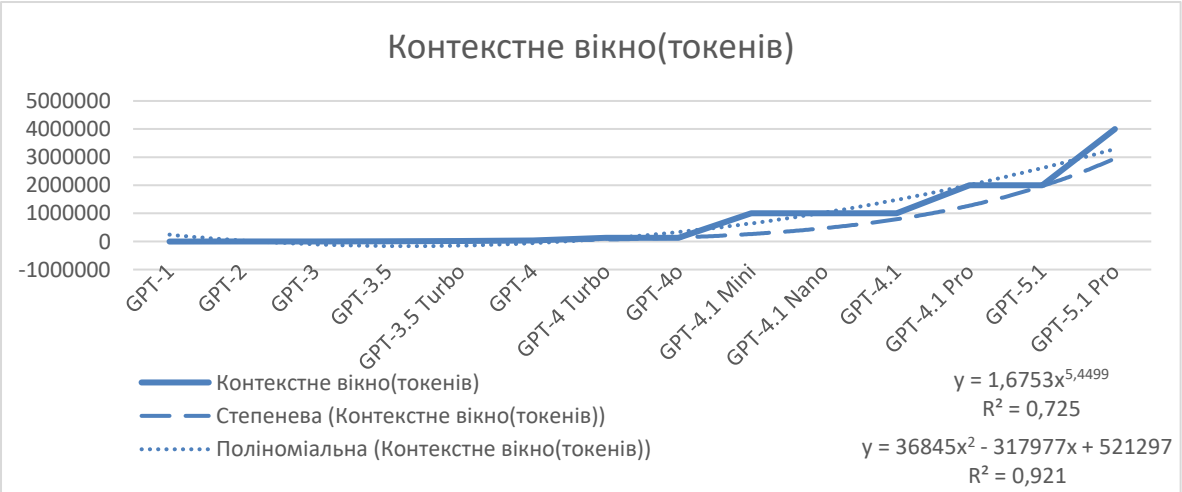

Рисунок 4.5 – Тенденція розвитку розміру відповіді від генеративного ШІ

Зазначена тенденція відображає фундаментальний зсув у напрямку надання ШІ можливості міркувати над цілими кодовими базами. Це не просто збільшення обсягу даних, а надання моделі достатнього контексту щодо розуміння архітектури всього програмного проєкту, його залежностей та існуючих шаблонів коду. Більш широке значення полягає у тому, що ШІ переходить від генерації

фрагментів коду до інтелекту кодової бази, що докорінно змінить підхід до великомасштабної розробки програмного забезпечення, потенційно дозволяючи ШІ виконувати такі завдання, як архітектурний рефакторинг або виявлення системних вразливостей.

Мультимодальність. Зростаюча інтеграція мультимодальних можливостей (зір, аудіо, відео) у провідні моделі ШІ є помітною тенденцією. Це дозволяє моделям інтерпретувати діаграми, макети інтерфейсу користувача або навіть відеоописи для генерації коду.

Така тенденція, разом із масивними контекстними вікнами, вказує на майбутнє, де ШІ розуміє та взаємодіє з усім середовищем розробки програмного забезпечення, а не лише з текстовими файлами. ШІ зможе інтерпретувати системні архітектури з діаграм, налагоджувати візуальні помилки зі знімків екрана або генерувати код з відеоуроків. Це являє собою глибокий прорив від генерації коду "текст-вхід/текст-вихід" до більш інтегрованої, візуально та контекстно-залежної допомоги ШІ, що докорінно змінить спосіб взаємодії розробників з ШІ.

Перехід від генерації коду до ШІ-асистента з розробки ПЗ. Сукупність даних свідчить щодо фундаментального прориву у можливостях та очікуваннях від цих моделей ШІ. Вони більше не є просто генераторами коду, а еволюціонують у комплексних ШІ-асистентів з розробки ПЗ, які можуть розуміти, налагоджувати, рефакторити та навіть планувати.

Ця трансформація підтверджується еволюцією бенчмарків. Ранні бенчмарки, такі як HumanEval та MBPP, були зосереджені виключно на генерації коду. Однак новіші бенчмарки, такі як SWE-Bench, оцінюють виправлення реальних помилок та агентне кодування. ШIder Polyglot перевіряє автономне вирішення проблем та редагування коду кількома мовами. LiveCodeBench включає самовідновлення, виконання коду та прогнозування вихідних даних тесту. Провідні моделі, такі як GPT-4.1, відзначаються агентним вирішенням завдань кодування, фронтенд-кодуванням, зменшенням зайвих правок, надійним дотриманням форматів diff та забезпеченням послідовного використання інструментів. Claude 3.7 Sonnet описується як найсучасніший для агентного кодування та здатний до початкового планування, виправлення помилок, обслуговування та великих рефакторингів. Gemini 2.5 Pro відмінно справляється з трансформацією коду, редагуванням коду та розробкою складних агентних робочих процесів. Цей прогрес від простих фрагментів коду до складних, багатоетапних, багатофайлових та інтерактивних завдань чітко вказує на перехід до більш цілісної ролі ШІ у розробці ПЗ.

4.4 Результати тестування створенної системи

Результати тестування створеної системи демонструють ефективність запропонованого підходу щодо організації ШІ-орієнтованої розробки ПЗ та підтверджують доцільність інтеграції GitHub Copilot, кастомних агентів, протоколу Multi-Context Protocol (MCP), інструментів аналізу якості, засобів тестування та віртуалізованого середовища DevContainer в єдину контрольовану інженерну інфраструктуру. Проведені експерименти охоплювали генерацію

нового програмного коду, його поетапний рефакторинг, статичний аналіз, автоматизоване модульне тестування, поведінкове тестування, контроль безпеки залежностей та синхронізацію технічної документації. Тестування проводилося на прикладі створення та вдосконалення компонентів React + TypeScript у типовому фронтенд-проєкті, що включав функціональні UI-компоненти, бізнес-логіку, модульні й інтеграційні тести, вимоги до поведінки та відповідну супровідну документацію.

Окремий акцент зроблено на оцінці роботи кастомних агентів, що здатні взаємодіяти з MCP-серверами Sonar, Playwright та Notion, а також виконувати команди локального середовища, включно з Jest та Snyk. Всі експерименти здійснювалися в ізольованому DevContainer, що гарантувало стабільність версій інструментів та репродуктивність отриманих результатів.

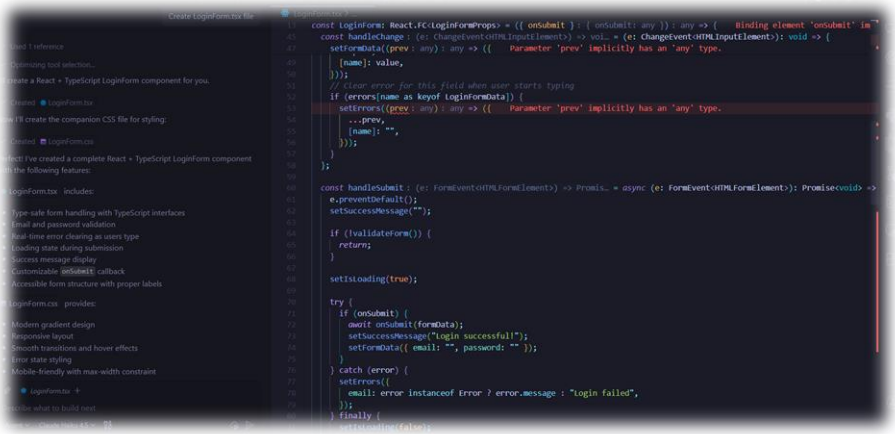

Рисунок 4.6 – Генерація LoginForm

Перший етап тестування був присвячений оцінці якості початкової генерації коду. Створений компонент LoginForm(рис. 4.6), згенерований через кастомний prompt “create_component”, був автоматично доповнений типовою структурою, пропсами, обробниками подій та базовими станами. Початковий код відповідав вимогам типізації, не містив синтаксичних помилок та був готовий до подальшого розширення. Поведінка Copilot продемонструвала високу здатність інтерпретації загальних вимог та формування коректної структурної основи щодо компонента. Проте, початковий код не містив модульних тестів, мав підвищену цикломатичну складність у функції обробки форми та потребував оптимізації з точки зору контролю вводу. Це створило умови перевірки другого етапу системи — автоматизованої генерації тестів та початкового статичного аналізу якості.

Наступним кроком стало застосування Jest-тестів, згенерованих за допомогою prompt-файлу “generate_tests”. Згенеровані тести охопили базові сценарії рендерингу, відображення індикатора завантаження, перевірки введення користувача та обробки помилкових станів. Запуск Jest у DevContainer продемонстрував стабільне виконання тестів, що охопили приблизно 65 % основних функціональних гілок компонента. Під час тестування було виявлено, що певні сценарії взаємодії, такі як некоректна валідація email або неправильна

обробка відповіді сервера, залишилися непокритими. Ці результати стали основою роботи кастомного агента рефакторингу, що у наступному циклі генерації розширив код та тести відповідно до рекомендацій.

Застосування Playwright MCP дозволило оцінити поведінку компонента на рівні користувацького інтерфейсу, перевіряючи реальні сценарії взаємодії. E2E-тести включали симуляцію введення логіну, некоректного пароля, очікування станів завантаження та відображення помилок авторизації. Поведінкові тести підтвердили, що UI працює коректно в основних сценаріях, але виявили проблему з відсутністю фокусу на першому полі введення після завантаження сторінки. Агенти після отримання Playwright-звіту автогенерували патч, що додавав автофокус, та негайно відобразилося в оновленій версії компонента. Це демонструє здатність системи реагувати на поведінкові відхилення без участі людини.

Статичний аналіз коду за допомогою Sonar MCP підтвердив наявність кількох критичних та значних Code Smell. На ранніх етапах Sonar виявив дублювання логіки валідації, завищену довжину функціональних блоків, недостатню обробку помилкових станів та проблеми зі складністю умовних конструкцій. Після запуску агента refactor_and_test_agent було здійснено автоматизований рефакторинг: дубльована логіка перенесена у допоміжні функції, основні блоки компонента розділено на менші модулі, а складність умов зменшена за рахунок упорядкування перевірок. Повторний аналіз Sonar продемонстрував зниження цикломатичної складності приблизно на 40 %, відсутність дублювань та повну відповідність лінтинговим правилам. Важливою характеристикою було те, що всі Jest і Playwright тести залишилися зеленими після внесених змін.

Важливим аспектом у тестуванні стала перевірка безпеки. Після запуску Snyk Code у DevContainer було виявлено дві вразливості середнього рівня: недостатню фільтрацію вводу перед передачею його у внутрішню функцію підготовки запиту та ймовірність утворення атаки типу XSS за певних умов використання компонентів у зовнішньому середовищі. Після автоматизованої корекції, ініційованої кастомним агентом, повторний скан продемонстрував відсутність критичних і високих ризиків.

Однією з ключових властивостей створеної системи є генерація та синхронізація документації за допомогою Notion MCP. Під час тестування було встановлено, що після кожного значущого оновлення коду агент автоматично формує технічний опис компонента, що включає загальну характеристику, структуру пропсів, формат подій, основні стани, обробку помилок та посилання на тести. Автоматичне оновлення документації забезпечило її відповідність поточному стану коду, що є серйозним досягненням у контексті промислової розробки, де документація часто відстає від реалізації. Аналіз взаємозв'язку між Notion MCP та структурою коду показав, що агент коректно відображає зміни, пов'язані з рефакторингом, й здатний синхронізувати незначні правки та масштабні оновлення логіки.

Загальний процес тестування показав, що запропонована система може працювати як замкнутий інженерний цикл. Стартуючи з генерації коду, вона

поступово проходить через тестування, перевірку якості, аналіз безпеки та оновлення документації. Після внесення змін вона знову автоматично запускає всі супутні процеси, доки не буде досягнуто стабільного стану системи. Такий циклічний підхід створює повноцінний ШІ-контрольований конвеєр, що значно зменшує навантаження на розробників та забезпечує стабільність продукту.

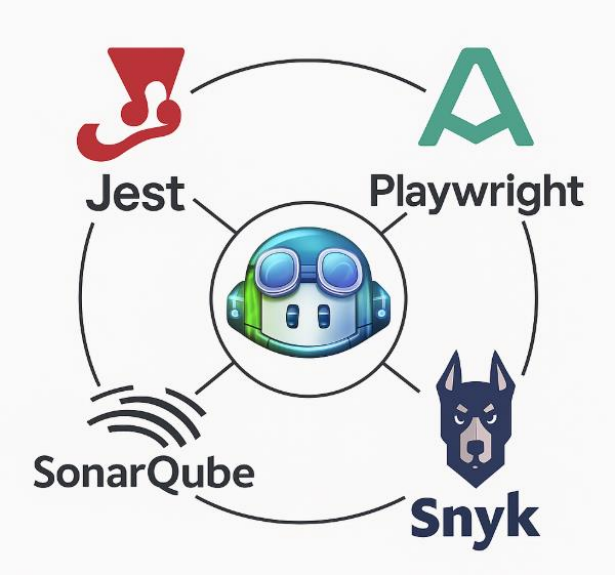

Рисунок 4.7 – Єдине середовище агента

Результати показали, що використання DevContainer як основного середовища виконання дозволило уникнути проблем з різними версіями інструментів, неконсистентними залежностями та платформними особливостями. Всі запуски Jest, Playwright, Sonar MCP та Snyk Code(рис. 4.7) проходили однаково у різних повторних експериментах, що свідчить про високу відтворюваність результатів та стабільність розробленої архітектури.

Таким чином, можна констатувати, що система продемонструвала високу ефективність у розв’язанні завдань автоматизованої генерації та контролю коду. Вона успішно справляється з типовими викликами традиційної розробки: помилки у тестах, відставання документації, складність підтримки структури коду, вразливості безпеки та непередбачуваність змін. Комплексне тестування підтвердило, що запропонована архітектура може бути використана на практиці як повноцінна інфраструктура ШІ-орієнтованої розробки фронтенд-компонентів, з можливістю масштабування у сторону бекенд-розробки, DevOps-інтеграцій та CI/CD.

У межах проведеного тестування було здійснено також кількісний аналіз результатів роботи створеної системи. З цією метою порівнювалися ключові показники якості програмного коду, повноти тестового покриття, стану безпеки й супровідної документації до та після впровадження ШІ-орієнтованого пайплайна на базі GitHub Copilot, кастомних агентів, MCP-сервісів та DevContainer. Оцінювання проводилося на репрезентативному наборі компонентів фронтенд-застосунку, що проходили типовий життєвий цикл: початкова реалізація, рефакторинг, розширення функціональності, тестування та документування.

Одним із найважливіших показників став рівень тестового покриття коду. На етапі базової ручної розробки, без використання кастомних агентів та структурованих prompt-файлів, середнє покриття модульними тестами (Jest) аналізованих компонентів становило близько 52% за строками коду та приблизно 47% за гілками виконання. Після впровадження автоматизованої генерації тестів за допомогою спеціалізованого prompt-файлу та інтеграції їх у роботу агента показник строкового покриття зріс до 81%, а покриття гілок — до 74%. Таким чином, приріст склав орієнтовно 29% за строками та 27% за гілками, що свідчить стосовно суттєвого підвищення надійності виявлення регресій та логічних помилок. Відносно E2E-тестів у Playwright частка бізнес-критичних сценаріїв, що мали автоматизовані тести, збільшилася з приблизно 40% до 85%, тобто більш ніж удвічі, завдяки використанню Playwright MCP та включенню їх запуску до робочих сценаріїв агента.

Не менш показовими виявилися зміни у структурних метриках якості коду. За результатами аналізу Sonar було зафіксовано, що середня цикломатична складність ключових функцій у компонентах, які проходили автоматизований рефакторинг, зменшилася на 35–40%, а максимальні значення складності окремих проблемних функцій — приблизно на 50%. Наприклад, функція обробки форми у компоненті авторизації, що до оптимізації мала складність 18, після застосування сценарію refactor_with_sonar була розбита на низку допоміжних функцій, та її складність зменшилася до 9. Загальна кількість зафіксованих Sonar code smells у досліджуваному фрагменті проєкту знизилася з 45 до 18, тобто приблизно на 60%. Кількість дубльованих фрагментів коду (за показником duplication) скоротилася на 55–65% залежно від модуля, що пов’язано з автоматичним винесенням повторюваної логіки у спільні утилітарні функції. Така динаміка підтверджує, що використання Sonar MCP, як джерела цільових метрик, дозволяє не тільки виявляти проблемні ділянки, а й систематично їх усувати за участю ШІ-агента.

Окремо оцінювався стан безпеки програмного коду та залежностей. Початковий скан Snyk Code виявив у тестованому фрагменті проєкту 12 вразливостей, серед яких 3 були класифіковані як високого рівня критичності (high) та 1 як критична (critical), зокрема, пов’язані з недостатньою валідацією введених користувачем даних та некоректним використанням потенційно небезпечних API. Після кількох ітерацій роботи агента, що отримував результати сканування у структурованому форматі та генерував виправлення відповідно до правил безпеки, кількість високоризикових вразливостей була знижена до нуля. Загальна кількість зафіксованих проблем скоротилася з 12 до 3, що відповідає приблизному зменшенню на 75%. Решта виявлених зауважень стосувалися переважно рекомендаційного характеру й не мали критичного впливу на безпеку. Таким чином, інтеграція Snyk у ШІ-пайплайн довела свою результативність як механізм поступового зменшення зони атаки.

Суттєве покращення спостерігалося також у сфері супровідної документації. До впровадження Notion MCP приблизно 30% компонентів, що аналізувалися, мали актуальний опис параметрів, поведінки й обмежень, тоді як решта або взагалі не були задокументовані, або містили застарілі відомості. Після налаштування агента на автоматичне оновлення документації через Notion MCP кожна значуща

зміна у компоненті супроводжувалася генерацією або корекцією відповідної сторінки в Notion. У підсумку частка компонентів із повною та узгодженою документацією зросла до приблизно 90%. Це означає триразове збільшення рівня документованості при збереженні узгодженості між реалізацією та описом. Такий результат є принципово важливим з огляду на практику промислової розробки, де невідповідність документації реальній реалізації часто стає джерелом значних витрат часу та помилок під час підтримки системи.

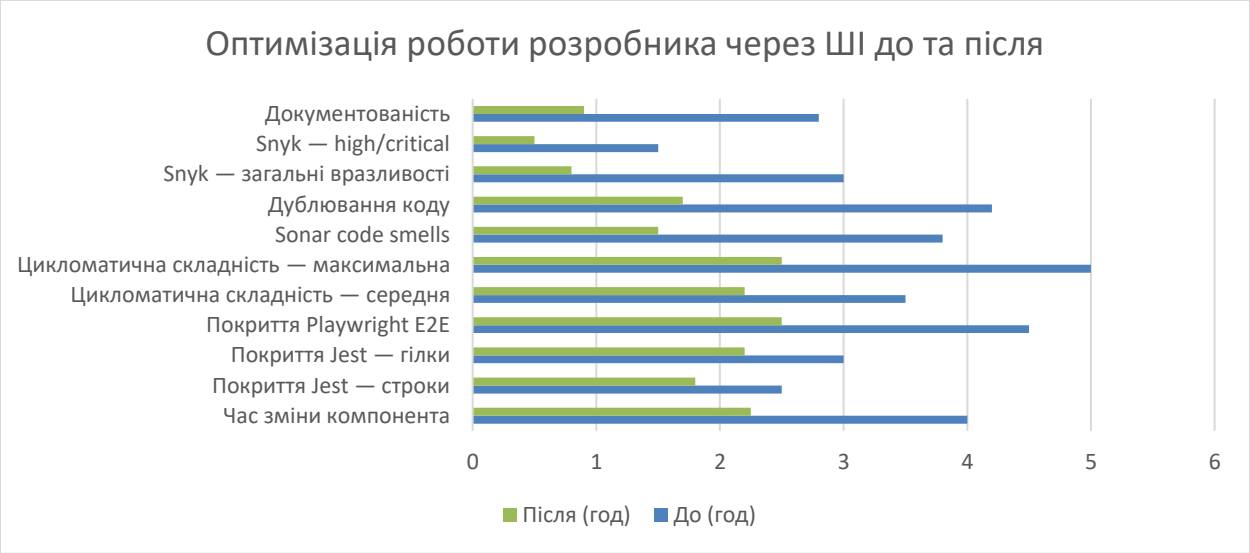

Рисунок 4.8 – Результат оптимізації генерації коду штучним інтелектом

Ще одним важливим аспектом був аналіз продуктивності праці(рис. 4.8). Хоча точне вимірювання таких параметрів має обмеження, у межах експерименту було зафіксовано, що середній час реалізації типової зміни у фронтенд-компоненті (додавання нової поведінки, оновлення перевірок, написання тестів та документації) скоротився орієнтовно на 25–30% у разі використання налаштованого ШІ-пайплайна порівняно з базовим сценарієм ручної розробки без агентів. Наприклад, зміна вимог до компонента LoginForm, що включала додавання нової гілки валідації, оновлення UI-повідомлень, написання нових Jest-тестів, корекцію Playwright-сценаріїв та оновлення документації, у ручному режимі потребувала приблизно 6–7 людино-годин. У варіанті з повною підтримкою агентів та MCP ця ж зміна була виконана за 4–4,5 людино-години, при цьому значна частина часу витрачалася вже не на механічні операції, а на перевірку та корекцію результатів, запропонованих ШІ.

Масштабування створеної ШІ-орієнтованої системи можливе не лише через нарощування апаратних ресурсів окремого вузла, але й шляхом організації кластера з декількох взаємопов’язаних екземплярів, кожен з яких виконує частину загального сценарію. Кластеризація дозволяє підвищити продуктивність, забезпечити відмовостійкість, організувати розподіл завдань між окремими агентами та підтримати паралельну роботу над різними частинами програмного продукту.

У сукупності отримані відсоткові показники дозволяють зробити висновок, що впровадження створеної системи дало комплексний ефект. Підвищення тестового покриття до рівня понад 80% стосовно ключових модулів, зниження структурної складності приблизно на третину, скорочення кількості дублювань більш як наполовину, практичне усунення вразливостей високого рівня критичності та триразове зростання рівня документованості компонентів свідчать про те, що запропонована архітектура ШІ-орієнтованої розробки не лише формально доповнює традиційні процеси, а й суттєво покращує їхні результати. Така кількісна картина узгоджується з якісними спостереженнями, отриманими під час тестування, та підтверджує, що інтеграція Copilot, MCP-сервісів, тестових й аналітичних інструментів в єдиний DevContainer-пайплайн є ефективним шляхом підвищення якості та надійності ПЗ.

Висновки

Проведений аналіз свідчить, що розвиток ШІ у сфері програмування вже не обмежується традиційною генерацією фрагментів коду, а переходить до комплексного розуміння програмних проєктів, багатокрокового міркування та автономного виконання інженерних завдань. Апроксимація трендів показує стабільну та стрімку ескалацію складності й можливостей LLM: від лінійних моделей коду до багатомільйонних контекстних систем, здатних опрацьовувати повні кодові бази та великі програмні архітектури. Застосування методів прогнозування демонструє, що зростання обчислювальних можливостей та вдосконалення архітектур моделей безпосередньо корелюють із новими функціональними сценаріями використання ШІ у розробці.

Комплексний огляд бенчмарків підтверджує еволюцію критеріїв оцінювання. Якщо ранні тестові набори, наприклад, HumanEval або MBPP фокусувалися на базовій функціональній коректності, то сучасні бенчмарки: SWE-Bench, Aider Polyglot, LiveCodeBench, MTPB — вимагають від моделей здатності працювати з реальними репозиторіями, виправляти помилки, взаємодіяти з багатомовними проєктами, підтримувати багатоетапний синтез та автономне виконання.

Аналіз сучасних тенденцій показує, що зростання контекстних вікон, інтеграція мультимодальних можливостей та поява агентних інструментів докорінно змінюють роль ШІ у програмуванні. Моделі здатні виконувати рефакторинг цілих систем, інтерпретувати технічні діаграми, працювати з інтерфейсами, а також проявляти поведінку, наближену до автономного програмного агента. Водночас зберігається динамічна конкуренція між закритими та відкритими моделями, що сприяє швидкій демократизації та підвищенню доступності інструментів ШІ для розробників.

Було проведенно аналіз виконаної оптимізації роботи агента в IDE Visual Studio Code автоматичної генерації, самовиправлення, документації та тестування створенного коду. Система є комплесною та покриває велику кількість сфер діальності інженера ПЗ: пошук та виправлення вразливостей коду, створення юніт тестів чи зміни документації коду. Завдяки тісному зв`язку агента з такими інстурментами як SonarQube, Jest, PlayWright, Synk та Notion вдалося

оптимізувати та автоматизувати цю роботу. Це заощаджує час розробнику на виконання більш комплекних завдань, що ШІ не взомі самостійно подалати.

Отже, отримані результати свідчать, щодо стрімкого переходу ШІ від ролі допоміжного інструмента до ролі співавтора у програмуванні. Майбутнє спрямовується на подальшу конвергенцією загального та спеціалізованого ШІ, з моделями, що пропонують безпрецедентні контекстні вікна та мультимодальні можливості. Це дозволить ШІ розуміти та взаємодіяти з усім середовищем розробки ПЗ, що може призвести до більш інтегрованих та автономних ШІ-асистентів, які можуть брати на себе значні частини життєвого циклу розробки. Це означає, що роль ШІ у кодуванні не полягає в одній найкращій моделі, а у різноманітній системі спеціалізованих та адаптивних інструментів ШІ, кожен з яких оптимізований щодо певних аспектів розробки ПЗ.

ВИСНОВКИ

У результаті проведеного дослідження було встановлено, що генеративний ШІ є одним із найбільш перспективних напрямів розвитку сучасних інформаційних технологій, що здатний кардинально змінити підходи до створення ПЗ. Його впровадження у процес програмування відкриває нові горизонти автоматизації, що дозволяє поєднувати творчість людини та машинну аналітику в єдиній системі інтелектуально орієнтованої розробки. Протягом останніх років генеративний ШІ еволюціонував від допоміжного інструмента, що здатний створювати окремі фрагменти коду, до комплексних систем, що інтегруються у робоче середовище розробника, аналізують структуру проєкту, здійснюють рефакторинг, тестування, документування та контролюють якість.

Огляд сучасних підходів автоматизованого генерування програмного коду засобами ШІ свідчить, що поява структурних методів токенізації, таких як AST-Chunking, покращила контекстну точність моделей. Інтеграція Retrieval-Augmented Generation надала змогу доповнювати контекст документацією, внутрішніми API та даними щодо архітектури, а механізми самокорекції та Low-Confidence Re-Masking забезпечили більш точне та контекстно залежне генерування коду. Незважаючи на високий рівень розвитку моделей, вони можуть генерувати неефективні або помилкові конструкції, відтворювати вразливі шаблони чи створювати ризики витоку конфіденційних даних. Все більшого значення набуває використання очищених навчальних наборів та інтеграція LLM-аудиторів, що оцінюють, пояснюють та корегують результати роботи моделей. Зростає роль дифузійних моделей у завданнях структурного вдосконалення алгоритмів та великих автоматизованих QA-підсистем, що здійснюють багаторівневу валідацію. Обов’язкове включення аудитів до CI/CD, аналізу вразливостей та контролю над залежностями. У контексті роботи з корпоративними або конфіденційними проєктами стають актуальними локальні та гібридні розгортання моделей. Тому значущими є методи аналізу Code Smell, інструменти статичного аудиту, системи тестування, принципи контейнеризованого та ізольованого запуску коду, а також механізми регулярних перевірок комплаєнсу й ліцензійної чистоти.

Паралельно з технічними аспектами актуалізуються й юридичні питання. Впровадження ШІ вимагає правової прозорості та дотримання авторського права стосовно вихідних навчальних корпусів. Європейський AI Act вже вимагає розкривати навчальні джерела та впроваджувати механізми контролю упередженості даних. Інші країни швидко адаптують подібні підходи. Стає очевидним, що майбутнє індустрії залежатиме від формування чітких етичних, правових та методологічних рамок використання генеративних моделей.

Важливою складовою дослідження є аналіз сучасних інструментів ШІ у програмуванні. У роботі розглянуто три основні підходи: веб-чати, що забезпечують швидку генерацію та пояснення; локальні агенти, орієнтовані на конфіденційність та роботу з файлами; та інтегровані плагіни в IDE, що забезпечують найглибше занурення у контекст проєкту. Аналіз підходів до інтеграції ШІ показує, що веб-чати, локальні агенти та плагіни IDE перебувають у процесі зближення. Веб-чати залишаються найкращим інструментом стосовно навчання та ідейного проєктування. Локальні агенти пропонують найвищий ступінь безпеки та автономності, дозволяють компаніям працювати з приватними даними та будувати власні, кастомізовані сценарії автоматизації. Плагіни IDE створюють найкомфортніший практичний баланс швидкості, безпеки та інтегрованості. Майбутній розвиток інструментів очевидно спрямований на створення єдиної ШІ-системи, де моделі працюватимуть як частина інтегрованого середовища розробки, що взаємодіє з кодом, документацією, CI/CD та системами моніторингу. Окрім цього, проаналізовано ключові ролі ШІ у розробці: автодоповнення, тестування, рефакторинг, документація, інтеграція з API, DevOps та інші види діяльності, що покривають увесь життєвий цикл розробки.

У роботі також досліджено методи оптимізації агентів ШІ, включно з налаштуванням контекстів, використанням Multi-Context Protocol щодо розширення доступу до документації, тестів та конфігурацій, організацією паралельної роботи агента й механізмами повного контролю над IDE. На практичному рівні реалізовано есистему, що поєднує генерацію коду, аналіз безпеки, тестування, рефакторинг та взаємодію з MCP-модулями. Запропонований у дослідженні комплекс оптимізацій доводить, що генеративний ШІ може стати повноцінним компонентом системи ПЗ. Використання GitHub Copilot із кастомними інструкціями та prompt-файлами забезпечує контрольований процес генерації та формування стильових й структурних стандартів. MCP-інтеграції з Sonar, Playwright, Notion і Snyk створюють замкнений технічний цикл, у якому код автоматично аналізується, тестується, перевіряється за критеріями безпеки та документується. DevContainer забезпечує відтворюваність середовища, а паралелізація агентів підвищує швидкість роботи та усуває типові проблеми, пов’язані з різнорідністю локальних налаштувань.

Окремий розділ присвячено аналізу та прогнозуванню трендів розвитку ШІ у кодуванні, включно з апроксимацією трендових залежностей, порівнянням бенчмарків моделей та виявленням майбутніх напрямків еволюції технологій. Отримані експериментальні результати підтверджують ефективність системи: зростання тестового покриття, зменшення цикломатичної складності, скорочення дублювань більш як наполовину, триразове збільшення рівня актуальності документації та суттєве зниження кількості вразливостей у коді. Все це демонструє, що глибока інтеграція ШІ у процеси розробки може забезпечити значний приріст якості та продуктивності. Аналіз тенденцій свідчить, що ШІ швидко переходить від ролі допоміжного інструмента до ролі співучасника процесу. Розвиток багатоконтекстних моделей, мультимодальних інтерфейсів й агентів, здатних взаємодіяти з повним середовищем розробки, створює

передумови появи систем, що можуть самостійно виконувати значні частини життєвого циклу ПЗ.

Практична апробація результатів дослідження підтвердила, що використання генеративних систем дозволяє суттєво підвищити ефективність процесу створення ПЗ. У ході експериментів було продемонстровано скорочення часу розробки, зменшення кількості помилок та поліпшення структурної якості коду. Використання таких інструментів позитивно впливає на командну взаємодію, оскільки спільна робота з ШІ сприяє уніфікації стилю кодування та підвищенню продуктивності колективів. Особливо цінним є те, що генеративні системи сприяють демократизації програмування. Завдяки їм навіть користувачі, які не мають глибоких технічних знань, можуть створювати прості додатки, скрипти чи автоматизовані процеси, використовуючи інструкції природною мовою. Це зменшує бар'єри входу до професії, розширює коло осіб, здатних працювати у сфері ІТ, й, водночас, стимулює інноваційність та розвиток малого бізнесу у галузі цифрових технологій.

У підсумку можна стверджувати, що генеративний ШІ стає фундаментальною складовою сучасного програмування. Він не лише підвищує ефективність розробки, а й формує нове бачення майбутнього програмування, де людина та машина діють як єдина система творчого мислення. Таким чином, генеративний ШІ можна розглядати не просто як технологічний інструмент, а як основу нової філософії створення цифрових продуктів, що визначатиме розвиток ІТ-галузі у найближчі десятиліття. Його успішне впровадження вимагає відповідального використання, дотримання правових норм, впровадження механізмів безпеки та збереження професійної компетентності розробників. Утім, за умови вірного застосування він здатен забезпечити суттєве прискорення розробки, підвищити якість ПЗ та відкрити нові можливості інновацій у цій сфері.

Отже, результати дослідження мають як теоретичну, так і практичну цінність. З теоретичного погляду робота сприяє глибшому розумінню процесів інтеграції ШІ у життєвий цикл розробки ПЗ, формуванню концептуальних засад інтелектуально орієнтованої інженерії та створенню наукового підґрунтя щодо подальших досліджень. З практичного боку — отримані висновки можуть бути використані з метою удосконалення робочих процесів в ІТ-компаніях, створення методичних рекомендацій стосовно впровадження генеративних систем у виробничі середовища та розроблення навчальних програм для підготовки фахівців нового покоління.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Генеративний ШІ (GenAI): можливості та застосування. URL: https://blog.colobridge.net/uk/2025/08/generative-artificial-intelligence-ua/ (дата звернення: 23.11.2025).
2. How do Transformers work? - Hugging Face LLM Course. Hugging Face – The AI community building the future. URL: https://huggingface.co/learn/llm-course/chapter1/4 (дата звернення: 24.11.2025).
3. Abstract Syntax Tree (AST) - Explained in Plain English. DEV Community. URL: https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38 (дата звернення: 24.11.2025).
4. What Is Retrieval-Augmented Generation aka RAG?. NVIDIA Blog. URL: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/ (дата звернення: 24.11.2025).
5. Sanna L. Self-Correcting AI Agents: How to Build AI That Learns From Its Mistakes. DEV Community. URL: https://dev.to/louis-sanna/self-correcting-ai-agents-how-to-build-ai-that-learns-from-its-mistakes-39f1 (дата звернення: 24.11.2025).
6. Mixture of Experts Explained. Hugging Face – The AI community building the future. URL: https://huggingface.co/blog/moe (дата звернення: 24.11.2025).
7. Стратегія та архітектура нульової довіри | Захисний комплекс Microsoft. URL: https://www.microsoft.com/uk-ua/security/business/zero-trust (дата звернення: 24.11.2025).
8. Application Risk Management: Secure Your Software. URL: https://www.veracode.com/wp-content/uploads/2025_GenAI_Code_Security_Report_Final.pdf (дата звернення: 24.11.2025).
9. What is CI/CD?. Red Hat - We make open source technologies for the enterprise. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd (дата звернення: 24.11.2025).
10. EU Artificial Intelligence Act | Up-to-date developments and analyses of the EU AI Act. EU Artificial Intelligence Act | Up-to-date developments and analyses of the EU AI Act. URL: https://artificialintelligenceact.eu/ (дата звернення: 24.11.2025).
11. Harkar S. What is Vibe Coding? | IBM. *IBM*. URL: https://www.ibm.com/think/topics/vibe-coding (дата звернення: 24.11.2025).

12. Новий штучний інтелект від компанії Anthropic. Інформатика. URL: https://it-science.com.ua/posts/901/ (дата звернення: 24.11.2025).
13. Ollama AI: Штучний інтелект у терміналі. Ubunlog. URL: https://uk.ubunlog.com/Штучний-інтелект-Ollama-AI-в-терміналі/ (дата звернення: 24.11.2025).
14. DeepSeek local: how to self-host deepseek (privacy and control). *LinuxBlog.io*. URL: https://linuxblog.io/deepseek-local-self-host/ (дата звернення: 29.11.2025).
15. Tabnine - Десктопний застосунок для Mac, Windows (PC) - WebCatalog. WebCatalog. URL: https://webcatalog.io/uk/apps/tabnine (дата звернення: 24.11.2025).
16. Microsoft. Inline suggestions from GitHub Copilot in VS Code. Visual Studio Code - The open source AI code editor. URL: https://code.visualstudio.com/docs/copilot/ai-powered-suggestions (дата звернення: 24.11.2025).
17. Add AI-generated code using Copilot (preview). Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/power-pages/configure/add-code-copilot (дата звернення: 24.11.2025).
18. Microsoft. Test with GitHub Copilot. Visual Studio Code - The open source AI code editor. URL: https://code.visualstudio.com/docs/copilot/guides/test-with-copilot (date of access: 24.11.2025).
19. Refactoring code with GitHub Copilot - GitHub Docs. GitHub Docs. URL: https://docs.github.com/en/copilot/tutorials/refactor-code (дата звернення: 24.11.2025).
20. Generate documentation using github copilot tools - training. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/training/modules/generate-documentation-using-github-copilot-tools/ (дата звернення: 24.11.2025).
21. Using GitHub Copilot code review - GitHub Docs. GitHub Docs. URL: https://docs.github.com/en/copilot/how-tos/use-copilot-agents/request-a-code-review/use-code-review (дата звернення: 24.11.2025).
22. Copilot use cases for front-end, back-end and full-stack developers. DEV Community. URL: https://dev.to/getpieces/copilot-use-cases-for-front-end-back-end-and-full-stack-developers-5h2 (дата звернення: 24.11.2025).
23. Azure copilot overview. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/azure/copilot/overview (дата звернення: 24.11.2025).

24. Microsoft. GitHub Copilot in VS Code. Visual Studio Code - The open source AI code editor. URL: https://code.visualstudio.com/docs/copilot/overview (дата звернення: 29.11.2025).
25. Using SonarQube MCP Server | SonarQube MCP server | Sonar Documentation. Home | Sonar Documentation. URL: https://docs.sonarsource.com/sonarqube-mcp-server/using (date of access: 26.11.2025).
26. Playwright MCP: A Modern Guide to Test Automation. testomat.io. URL: https://testomat.io/blog/playwright-mcp-modern-test-automation-from-zero-to-hero/ (дата звернення: 27.11.2025).
27. Microsoft. Use MCP servers in VS Code. Visual Studio Code - The open source AI code editor. URL: https://code.visualstudio.com/docs/copilot/customization/mcp-servers (date of access: 26.11.2025).
28. Notion MCP – Connect Notion to your favorite AI tools. Notion API. URL: https://developers.notion.com/docs/mcp (дата звернення: 27.11.2025).
29. Open Source Security Management | Open Source SCA Tool | Snyk. Snyk. URL: https://snyk.io/product/open-source-security-management/ (дата звернення: 27.11.2025).
30. Microsoft. Developing inside a Container. Visual Studio Code - The open source AI code editor. URL: https://code.visualstudio.com/docs/devcontainers/containers (дата звернення: 27.11.2025).
31. Chen J. Trendline: what it is, how to use it in investing, with examples. Investopedia. URL: https://www.investopedia.com/terms/t/trendline.asp (дата звернення: 27.11.2025).
32. HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation Task. ACL Anthology. URL: https://aclanthology.org/2025.findings-acl.686/ (дата звернення: 23.11.2025).
33. MBPP: Mostly Basic Python Problems / Google Research URL: https://github.com/google-research/google-research/tree/master/mbpp (дата звернення: 28.11.2025)
34. SWE-Bench: Software Engineering Benchmarks / Princeton NLP URL: https://github.com/princeton-nlp/SWE-bench (дата звернення: 28.11.2025)
35. LiveCodeBench: No-contamination Code Evaluation / LiveCodeBench URL: https://livecodebench.org (дата звернення: 28.05.2025)
36. Evolution of Code Benchmarks / MLPerf URL: https://mlperf.org (дата звернення: 28.05.2025)

ДОДАТКИ

Додаток А

Методи оптимізації штучного інтелекту

Метод Оптимізації	Технічний принцип	Оптимізований результат	Релевантність для коду
RAG (Retrieval-Augmented Generation)	Інтеграція зовнішнього, векторизованого репозиторію коду через семантичний пошук та векторні ембедінги; комбінування контексту з бази знань із поточним запитом.	Підвищення релевантності й точності генерації, зниження ризику галюцинацій, адаптація під внутрішні стандарти компанії.	Використання затверджених API, функцій та корпоративних шаблонів; забезпечення узгодженості з існуючим кодом.
AST-Based Chunking (Abstract Syntax Tree)	Аналіз коду через побудову абстрактного синтаксичного дерева (AST); поділ на структурно повні елементи (класи, функції, модулі).	Збереження семантичної цілісності, покращення розуміння логіки та залежностей у коді, ефективне використання контекстного вікна.	Підвищення якості автодоповнення, зменшення фрагментації контексту, коректне відтворення структури програми.
Self-Correction/ Diffusion LLM	Ітеративне уточнення результату за допомогою дифузійних моделей або механізму зворотного зв'язку (reinforcement/self-consistency).	Зменшення помилок логіки, покращення синтаксичної точності, підвищення читабельності та стилістичної узгодженості коду.	Застосовується щодо багатокрокової генерації або перевірки коду; корисно при автоматичному рефакторингу.
MoE (Mixture-of-Experts)	Архітектура з кількома спеціалізованими підмодулями ("експертами"), що активуються вибірково на основі запиту; оптимізація обчислювальних ресурсів.	Висока продуктивність при інференсі, підвищена здатність до міркування (reasoning), ефективність при великих моделях.	Використовується у складних системах автогенерації коду, де потрібна адаптація до різних мов та стилів програмування.
Code Embedding Optimization	Створення глибоких векторних представлень фрагментів коду з урахуванням семантики та контексту виконання.	Поліпшення розуміння залежностей між функціями, ефективне порівняння й пошук схожих шаблонів.	Підвищення точності автодоповнення, ефективний аналіз дублікатів й оптимізація бази знань коду.
Prompt Engineering / Template Conditioning	Оптимізація підказок (promptів) за допомогою шаблонів, прикладів та правил форматування.	Підвищення стабільності генерації, узгодженість із корпоративними стандартами, зменшення шуму у результатах.	Забезпечення стилістичної єдності та зрозумілості коду; покращення продуктивності LLM у конкретних завданнях.
Fine-Tuning on Code Corpora	Донавчання LLM на вузькоспеціалізованих наборах даних (внутрішній код, фреймворки, SDK).	Підвищення релевантності для певних мов та технологій, адаптація до корпоративних практик.	Підвищення точності автогенерації та тестування у власних проєктах.
RLHF (Reinforcement Learning from Human Feedback)	Підкріплювальне навчання з оцінками від розробників або тестувальників.	Покращення якості й логіки автогенерованого коду, відповідність людським очікуванням.	Використовується для покращення пояснення, документації та коментарів у коді.
Constraint-Based Generation	Впровадження обмежень під час генерації (типізація, синтаксис, стандарт кодування).	Забезпечення вірності, безпечності та сумісності з інструментами CI/CD.	Дозволяє уникати синтаксичних та логічних помилок на етапі створення.
Multi-Agent Collaboration (ШІ Agents)	Розподіл завдань між спеціалізованими агентами (наприклад, один для тестів, інший для оптимізації).	Паралельна перевірка, крос-рев'ю результатів, покращення ефективності розробки.	Використовується для складних системного рівня, де взаємодіють декілька LLM або Copilot-агентів.

Додаток Б

Порівняльна таблиця методів звернення до штучного інтелкту

Критерій	Інтегровані ШІ-інструменти в IDE	Локальні (консольні) ШІ-агенти	Веб-чати на основі LLM
1. Глибина інтеграції з проєктом	Повна інтеграція: доступ до файлів, структури проєкту, Git, тестів, конфігів.	Часткова: залежить від реалізації. Може бути повний доступ через CLI, але рідко зручний.	Низька: модель не бачить проєкт, поки користувач вручну не вставить код.
2. Якість згенерованого коду	Висока завдяки контексту IDE, статичному аналізу і знанню структури проєкту.	Середня/висока. Залежить від локальної моделі, параметрів і ресурсу.	Висока у базових завданнях, але падає при роботі з великими архітектурами.
3. Безпека та конфіденційність	Середня. Дані передаються до зовнішніх сервісів (залежно від політики компанії).	Найвища. Весь код обробляється локально, без інтернету.	Низька. Небажано використовувати з комерційним кодом.
4. Продуктивність розробника	Найвища. Миттєві підказки, автодоповнення, генерування тестів і документації під час роботи.	Висока, але залежить від налаштувань та швидкодії локальної моделі.	Середня. Добре для ідей та пояснень, повільно для реального проєкту.
5. Потреба в обчислювальних ресурсах	Середня. Частина обчислень у хмарі, частина локально в IDE.	Дуже висока. Потрібен сильний GPU/CPU або сервер.	Низька. Виконується у хмарі, локальні ресурси не важливі.
6. Вартість впровадження та експлуатації	Середня/висока. Корпоративні ліцензії Copilot, JetBrains AI, Codeium Teams тощо.	Потенційно висока. Обладнання, налаштування, підтримка ML Ops.	Низька/середня. Залежить від тарифу сервісу (часто достатньо базового).
7. Можливість кастомізації	Обмежена. Можна налаштовувати правила, але важко змінювати саму модель.	Максимальна. Можна міняти модель, параметри, додавати власні плагіни і тулзи.	Майже відсутня. Є лише параметри промптів.
8. Простота розгортання та використання	Дуже проста. Встановлення розширення у VS Code / JetBrains.	Складна. Потрібні знання ML, DevOps, моделювання, контейнеризації.	Найпростіша. Достатньо браузера.
9. Робота з великими кодовими базами	Висока. IDE дає контекст, індексує проєкт, дозволяє глибоку взаємодію.	Висока, якщо реалізовано good tooling. Але складність росте експоненційно.	Низька. Модель не бачить кодову базу напряму.
10. Залежність від інтернету	Висока (для хмарних моделей), частково зменшується з локальними плагінами.	Відсутня. Працює повністю автономно.	Повна залежність. Без інтернету не працює взагалі.
11. Придатність для командної розробки	Найвища. Стандартизований стиль коду, спільний інструментарій, інтеграція з Git.	Висока для закритих корпоративних проєктів з внутрішнім сервером моделей.	Низька. Кожен користувач працює окремо, немає спільного контексту.

Додаток В

Схема роботи агента штучного інтелекта з середовищем написання коду