ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ІНФОРМАТИКИ І КОМП'ЮТЕРНИХ НАУК

Пояснювальна записка
до кваліфікаційної роботи
бакалавра

на тему:

Розробка інструментальних засобів оптимізації програмного коду з використанням статичного аналізу та метрик якості

Виконав: студент групи 4КН спеціальності
122 "Комп'ютерні науки"

Касьянов П.В
Керівник: Корніловська Н.В
Рецензент: Огнєва О.Є

Хмельницький – 2026 р.

Факультет	Інформаційних технологій та дизайну
Кафедра	Інформатики і комп'ютерних наук
Рівень вищої освіти	бакалавр
Галузь підготовки	12 «Інформаційні технології»
	(шифр і назва)
Освітньо-професійна програма	Комп’ютерні науки
	(назва)
Спеціальність	122 «Комп’ютерні науки»
	(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІКН,
професор
_____Моісеєнко С.В
«____» _____ 2026 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА
Касьянов Павло Вячеславович
(прізвище, ім’я, по батькові)

Тема роботи: Розробка інструментальних засобів оптимізації програмного коду з використанням статичного аналізу та метрик якості.

1. Керівник роботи Корніловська Наталія Володимирівна, кандидат технічних наук, доцент кафедри інформатики і комп'ютерних наук
   (прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
   затверджені наказом ХНТУ від «26» січня 2026 р. № 133-с
2. Строк подання студентом роботи

3. Вихідні дані до роботи. Збір та аналіз інформації про вже існуючі підходи до оцінки якості програмного коду та інструменти для автоматичного рефакторингу. Визначення ключових метрик для аналізу а саме цикломатична складність, індекс підтримуваності, метрики Холстеда. Розробка алгоритмів взаємодії так званих промптів з великими мовними моделями LLM. Формулювання функціональних вимог до десктопного застосунку з підтримкою механізмів інтелектуального ШІ-рефакторингу, асинхронної обробки даних та візуалізації змін коду.
4. Зміст розрахунково-пояснювальної записки. Вступ. 1 Теоретико-методологічні основи оцінки та покращення якості коду. 2 Проєктування автоматичної системи метричного аналізу та рефакторингу. 3 Практична реалізація програмного засобу. 4 Тестування та оцінка ефективності програмного забезпечення. Висновки.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Рисунків - 42, Таблиць - 10, Формул – 12.
6. Дата видачі завдання: 30.01.2026р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк етапів роботи	Прим.
1	Вибір та узгодження теми дипломної роботи.	30.01.2026 – 07.02.2026	Виконано
2	Аналіз предметної області та постановка задачі.	08.02.2026 – 22.02.2026	Виконано
3	Вибір інструментарію для розробки.	23.02.2026 – 08.03.2026	Виконано
4	Проєктування архітектури системи та моделі даних.	09.03.2026 – 25.03.2026	Виконано
5	Розробка основного функціоналу додатка.	26.03.2026 – 15.04.2026	Виконано
6	Інтеграція компонентів системи та ШІ-моделі.	16.04.2026 – 28.04.2026	Виконано
7	Тестування та налагодження додатка.	29.04.2026 – 06.05.2026	Виконано
8	Оформлення пояснювальної записки.	07.05.2026 – 29.05.2026	Виконано

Студент _____ Касьянов П.В.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Корніловська Н.В.

(підпис) (прізвище та ініціали)

ЗМІСТ

АВТОРЕФЕРАТ

Ця робота присвячена дослідженню та розробці десктопної системи, призначеної для автоматизованого оцінювання якості програмного коду та інтелектуального рефакторингу. Оскільки перевірка архітектурних недоліків і усунення технічного боргу з'їдають понад половину часу розробника, а продуктивність великих мовних моделей LLM у програмуванні практично неможливо переоцінити, стає зрозуміло, чому створення інструмента, який може оцінювати код у дії та визначати його якість, є нагальною необхідністю.

Метою цієї роботи є розробка програми, яка поєднує строгі математичні обчислення інженерних метрик із генеративним штучним інтелектом, щоб вона могла сприяти підвищенню якості програмного забезпечення та, відповідно, зменшенню когнітивного навантаження на розробника. Для досягнення поставленої мети було виконано такі завдання, а саме проведено аналіз методів оцінки складності коду, спроєктовано зручний графічний інтерфейс, інтегровано хмарну нейромережу для автоматизованої оптимізації алгоритмів, а також розроблено підсистеми візуального порівняння і створення аналітичних графіків.

У процесі розробки використано технологічний стек на основі мови програмування Python. Для клієнтської частини і створення сучасного інтерфейсу такого як режим Dark Mode, кастомний редактор з підсвічуванням синтаксису застосовано бібліотеку Tkinter. Збір аналітичних даних забезпечують модулі Radon і difflib, тоді як за побудову графіків відповідає Matplotlib. Центральним елементом системи є асинхронний рушій для рефакторингу, який за допомогою API NVIDIA NIM працює з потужною мовною моделлю Mistral 128B. Цей алгоритм аналізує вихідний код і повертає його оптимізовану версію зі зниженою цикломатичною складністю, доданою типізацією та документацією українською мовою відповідно до стандарту PEP 8.

Тестування на реальних практичних прикладах підтвердило стабільність роботи розробленої системи. Інструмент ефективно автоматизує рутинні задачі, демонструє зниження складності коду та підвищення індексу підтримуваності

(Maintainability Index), що дозволяє розробникам оперативно і якісно оптимізувати власні програмні продукти.

Ключові слова: якість програмного забезпечення, рефакторинг, аналіз метрик, цикломатична складність, штучний інтелект, LLM, Mistral, Python, Tkinter, візуалізація даних.

ABSTRACT

This work is devoted to the research and development of a desktop system designed for automated assessment of the quality of software code and intelligent refactoring. Since checking architectural defects and eliminating technical debt eat up more than half of the developer's time, and the productivity of large language models LLM in programming is practically impossible to overestimate, it becomes clear why the creation of a tool that can evaluate the code in action and determine its quality is an urgent need.

The purpose of this work is to develop a program that combines rigorous mathematical calculations of engineering metrics with generative artificial intelligence so that it can contribute to improving the quality of software and, accordingly, reducing the cognitive load on the developer. To achieve this goal, the following tasks were performed, namely, an analysis of methods for assessing code complexity was carried out, a convenient graphical interface was designed, a cloud neural network was integrated for automated optimization of algorithms, and subsystems for visual comparison and creation of analytical graphs were developed.

The development process used a technology stack based on the Python programming language. The Tkinter library was used for the client side and the creation of a modern interface such as Dark Mode, a custom editor with syntax highlighting. Analytical data collection is provided by the Radon and difflib modules, while Matplotlib is responsible for plotting graphs. The central element of the system is an asynchronous refactoring engine, which works with the powerful Mistral 128B language model using the NVIDIA NIM API. This algorithm analyzes the source code and returns its optimized version with reduced cyclomatic complexity, added typing and documentation in Ukrainian according to the PEP 8 standard.

Testing on real practical examples confirmed the stability of the developed system. The tool effectively automates routine tasks, demonstrates a reduction in code complexity and an increase in the Maintainability Index, which allows developers to quickly and efficiently optimize their own software products.

Keywords: software quality, refactoring, metrics analysis, cyclomatic complexity, artificial intelligence, LLM, Mistral, Python, Tkinter, data visualization.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

LLM	Large Language Model
API	Application Programming Interface
PEP 8	Python Enhancement Proposal 8
NVIDIA NIM	NVIDIA Inference Microservices
ІШІ	Штучний інтелект
AST	Abstract Syntax Tree
NLP	Natural Language Processing
IDE	Integrated Development Environment
MI	Maintainability Index
LOC	Lines of Code
LLOC	Logical Lines of Code
SLOC	Source Lines of Code
CD	Comment Density
MVP	Minimum Viable Product

ВСТУП

Актуальність теми. Індустрія розробки програмного забезпечення розвивається безпрецедентними темпами. Завдяки гнучким методологіям, безперервній інтеграції та безперервному розвитку компанії часто надають пріоритет швидкості виходу на ринок. Це неминуче призводить до накопичення так званого технічного боргу, архітектурних компромісів, які, хоча й забезпечують швидкий вихід на ринок, але значно перешкоджають майбутньому обслуговуванню та масштабованості. Погано структурований код стає важким для розуміння і кожна зміна несе з собою безліч прихованих помилок.

Аналізуючи існуючі на ринку рішення для контролю якості коду, не можна ігнорувати гігантів статичного аналізу коду, таких як SonarQube, Pylint та ESLint. Хоча вони безсумнівно довели свою ефективність у стандартизації розробки, вони страждають від фундаментальних архітектурних обмежень. Вони функціонують лише як діагностичні інструменти та не можуть самостійно вирішити проблему.

Наприклад традиційні аналізатори коду покладаються виключно на лінійну математичну логіку. Вони показують, що ця функція перевищила дозволену цикломатичну складність або порушено стандарт проектування PEP 8 [21] . Однак система повністю залишає вирішення цієї проблеми розробнику. Це означає, що програмісти повинні припинити розробку нових функцій і витрачати дорогоцінний час на рутинні ручні перевірки та переписування коду.

Ручний рефакторинг завжди пов'язаний з людським фактором та суб'єктивністю. Розробники та особливо молодші без великого досвіду розробники часто виправляють код просто для задоволення вимог лінтера, не враховуючи загальний контекст. Це може призвести до втрати оригінальної бізнес логіки або створення ще складнішої архітектури.

Водночас генеративний штучний інтелект та великі мовні моделі стрімко розвиваються останніми роками. Розробники активно використовують помічників штучного інтелекту для генерації коду. Однак стандартна взаємодія з LLM через чат має суттєвий недолік відсутність суворого контролю якості. Навіть якщо

штучний інтелект може генерувати формально функціональний код, цей код важко підтримувати та він не відповідає технічним стандартам відповідного проекту.

Тому терміново потрібна розумніша та комплексніша система. Система яка поєднує ретельний математичний аналіз об'єктивних метрик розробки , таких як цикломатична складність, індекс підтримуваності, зусилля Холстеда, з творчою силою сучасних мовних моделей. Такий гібридний підхід дозволяє системі математично демонструвати, що код потребує покращення, а потім автоматично практично на льоту генерувати оптимізовану версію, яка включає необхідні пропозиції щодо типів та документацію.

Розробка автоматизованого інструменту, який дозволяє асинхронну взаємодію з потужною нейронною мережею як от Mistral та надає розробникам візуальний аналіз відмінностей між змінами, це не просто академічна справа. Це спроба вирішити реальну та актуальну проблему в IT індустрії. Перетворення процесу рефакторингу з виснажливої та непередбачуваної рутини на швидкий, контрольований та об'єктивний процес розробки.

Мета та завдання дослідження. Метою цієї роботи є покращення якості програмного забезпечення та зменшення когнітивного навантаження розробників шляхом створення автоматизованої системи для аналізу метрик та інтелектуального рефакторингу вихідного коду.

Для досягнення цієї мети були поставлені такі завдання:

1. Аналізувати існуючі підходи, інструменти та метрики для оцінки якості програмного коду такі як цикломатична складність, індекс підтримуваності, зусилля Холстеда.
2. Розробити архітектуру десктопного застосунку з підтримкою асинхронної обробки даних та хмарного механізму штучного інтелекту.
3. Розробити алгоритми та промти для ефективної взаємодії з великомасштабною мовною моделлю LLM для створення оптимізованого коду, який відповідає стандартам чистого коду PEP 8 та включає в себе типізацію.

4. Впровадити програмний продукт, що включає інструменти для візуального порівняння та створення діаграм для візуальної демонстрації ефективності рефакторингу.
5. Провести ретельне тестування розробленої системи на реальних прикладах коду та оцінити ефективність запропонованих рішень.

Об’єкт дослідження: процес забезпечення та покращення якості програмного забезпечення на етапах розробки та підтримки.

Предмет дослідження: методи, алгоритми та програмні засоби для метричного аналізу та автоматизованого рефакторингу вихідного коду з використанням розширених мовних моделей.

Методи дослідження. Для вирішення поставлених у роботі завдань було використано комплекс наукових та інженерних методів. Методи системного аналізу та об'єктно-орієнтованого проектування**.** Він використовується для створення архітектур програмного забезпечення, визначення модульної структури десктопного застосунку та організації взаємодії між графічним інтерфейсом користувача, аналітичним механізмом та хмарним API.

Методи статичного аналізу вихідного коду**.** Він використовується для створення абстрактних синтаксичних дерев AST та аналізу лексичного коду для точного розрахунку об'єктивних технічних метрик (цикломатична складність Маккейба, індекс підтримуваності, метрики обсягу програми Холстеда).

Штучний інтелект та методи швидкого розвитку**.** Він використовується для генерації високоструктурованих та контекстно залежних запитів до великої мовної моделі Mistral 128B, що дозволяє контрольовану генерацію коду, суворо дотримується стандартів PEP 8 та зберігає оригінальну бізнес логіку алгоритмів. Методи обробки природної мови NLP. Використовується в контексті генеративної моделі для автоматичної генерації зрозумілої документації українською мовою та вбудованих коментарів для оптимізованого коду. Методи алгоритмічного порівняння, статистичної обробки та візуалізації даних. Використовується для знаходження найбільших спільних підпослідовностей під час створення

візуального аналізу відмінностей коду, а також для кількісної оцінки змін метрик та їх візуального представлення створення порівняльних гістограм.

Наукова новизна отриманих результатів. Вона полягає в розробці та дослідженні гібридного підходу до рефакторингу програмного забезпечення, який вперше інтегрує детерміновані математичні методи аналізу коду з генеративними можливостями великих мовних моделей LLM на рівні комплексного інструменту.

Покращено процес усунення технічного боргу в програмних проектах. Запропонований гібридний підхід трансформує показники статичного аналізу та переводить їх з простого інструменту налагодження на механізм валідації на основі штучного інтелекту, який не лише виявляє архітектурні недоліки, але й автоматично генерує оптимізовані фрагменти коду.

Набула подальшого розвитку технологія автоматизованого рефакторингу, заснована на спеціалізованих методах інженерії підсказок. Вирішила проблему галюцинацій LLM у програмуванні. Це було досягнуто шляхом примусового дотримання моделі стандарту проектування, незалежної реалізації суворої типізації та генерації технічної документації без впливу на бізнес-логіку програми.

Запропоновано архітектура замкнутої системи для оцінки ефективності рефакторингу аналіз-оптимізація-аналіз. На відміну від використання звичайних помічників штучного інтелекту, ця система дозволяє провести об'єктивну математичну перевірку ефективності коду, згенерованого нейронною мережею, шляхом порівняння таких показників, як когнітивні зусилля (зусилля Холстеда) та індекс підтримуваності до та після втручання алгоритму.

Практичне значення отриманих результатів. Розроблений програмний інструмент це готовий до використання десктопний застосунок, який можна безпосередньо інтегрувати в робочі процеси IT компаній або використовувати новачками для автоматизації вдосконалення коду. Система також безцінна для освіти в галузі інформатики, оскільки дозволяє студентам чітко аналізувати вплив рефакторингу на математичні показники якості та розвивати стійкі навички написання чистого коду.

Апробація результатів

1. Касьянов П.В., Корніловська Н.В., Лур'є І. А. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. Синергія науки і бізнесу у повоєнному відновленні Херсонщини : матеріали III Міжнародної науково-практичної конференції (м. Хмельницький, 22–24 квітня 2026 р.). Хмельницький : Херсонський національний технічний університет, 2026. (електронне видання) https://kntu.net.ua/ukr/Nauka/Konferenciyi-ta-naukovo-tehnichni-zahodi/IIIIII-MIIZHNARODNA-NAUKOVO-PRAKTICHNA-KONFERENCIIYA-SINERGIIYA-NAUKI-II-BIIZNESU-2026#Anchor11

2. Касьянов П.В., Корніловська Н.В., Карамушка М. В. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. V Міжнародна науково-практична конференція "SCIENTIFIC DEVELOPMENT IN A CHANGING WORLD" (11-13.05.2026 року) с. 335-340 (електронне видання). https://sci-conf.com.ua/wp-content/uploads/2026/05/SCIENTIFIC-DEVELOPMENT-IN-A-CHANGING-WORLD-11-13.05.26.pdf

3. Касьянов П.В., Корніловська Н.В. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. Міжнародна наукова інтернет-конференція «ТЕНДЕНЦІЇ ТА ПЕРСПЕКТИВИ РОЗВИТКУ НАУКИ І ОСВІТИ В УМОВАХ ГЛОБАЛІЗАЦІЇ» (Вип. 129) (УГСП, 30 травня 2026р) / за ред. С.М. Кикоть. -Переяслав. С. 133-138. (електронне видання) https://confscientific.webnode.com.ua/arkhiv2/

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ ОЦІНКИ ТА ПОКРАЩЕННЯ ЯКОСТІ КОДУ

1.1. Проблема технічного боргу та необхідність рефакторингу програмного забезпечення.

Сучасний процес розробки програмного забезпечення нерозривно пов'язаний з концепціями безперервної доставки та гнучких методологій. З огляду на жорстку конкуренцію та ринковий попит на мінімізацію часу виходу на ринок, команди розробників часто змушені знаходити баланс між швидкістю розробки коду та якістю її архітектури. Наслідком цих компромісів є накопичення так званого технічного боргу.

Концепція технічного боргу спочатку запропонована Уордом Каннінгемом. Вона проводить аналогію між розробкою програмного забезпечення та фінансовими зобов'язаннями. Випуск функціонального, але неоптимального коду може запропонувати короткострокові вигоди, а саме швидкі релізи. Однак це створює недоопрацювання, за які в майбутньому потрібно сплачувати так звані відсотки. Цей відсоток проявляється у збільшенні часу витраченого на впровадження нових функцій, більших труднощах у виявленні помилок та вищому когнітивному навантаженні на розробників.

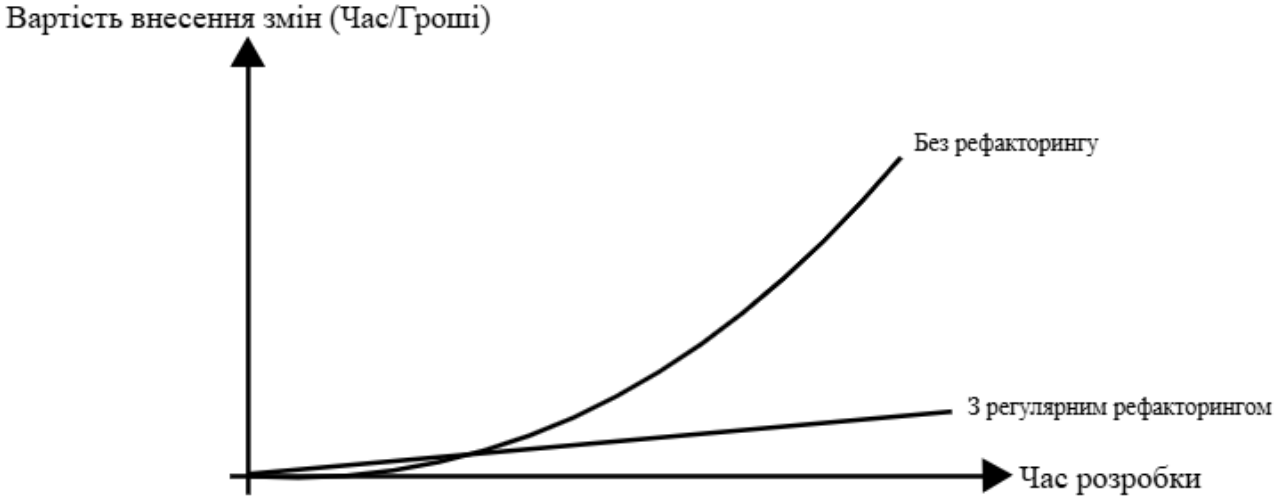


Рис. 1.1 Вартість змін з часом з урахуванням технічного боргу.

Технічний борг не завжди є результатом некомпетентності програміста. Згідно з класифікацією Мартіна Фаулера, технічний борг може бути свідомим [1] . Наприклад навмисна відмова від складного шаблону проектування через обмеження в часі або несвідомим через брак досвіду або зміни бізнес вимог під час процесу розробки.

Основними симптомами накопиченого технічного боргу є так звані “запахи коду” [2] . Це архітектурні антишаблони, які самі по собі не є синтаксичними помилками та не перешкоджають компіляції програми, але вказують на фундаментальні проблеми в проекті системи. [27]

Таблиця 1.1 Типові архітектурні проблеми “запахи коду” та їх вплив на розробку

Назва антипатерну	Опис архітектурної проблеми	Наслідки для програмного проєкту
“Об'єкт Бога” або іншими словами God Object	Клас або модуль, який контролює в собі забагато бізнес логіки, має велику кількість методів та порушує принцип єдиної відповідальності.	Неможливість повторного використання коду, високий ризик регресії, внесення нових помилок під час модифікації, критична складність модульного тестування.
“Дублювання коду”	Наявність ідентичних або дуже схожих фрагментів коду в різних частинах програми, це є наслідком Copy-Paste програмування.	Невиправдане збільшення загального обсягу програми, ускладнення підтримки, виправлення однієї помилки вимагає пошуку та редагування всіх дублікатів.
“Довгий метод”	Функція або метод із надмірною кількістю рядків коду, що виконує кілька незалежних завдань.	Висока цикломатична складність, низька читабельність, збільшення когнітивного навантаження на розробника під час аналізу алгоритму.

Єдиним ефективним методом усунення технічного боргу та “запахів” коду є рефакторинг. За словами М. Фаулера, рефакторинг це контрольований процес модифікації внутрішньої структури програми без впливу на її зовнішню поведінку, але з метою зробити код більш зрозумілим та ефективним [1] . Однак на практиці процес рефакторингу стикається з низкою перешкод.

1. **Високі витрати:** ручний аналіз та переписування коду вимагають значних витрат часу від високо кваліфікованих розробників.
2. **Відсутність комерційної цінності для клієнта:** рефакторинг не додає жодної видимої нової функціональності, тому компанії часто відмовляються виділяти на нього ресурси.
3. **Ризик появи нових помилок, регресій:** модифікація складної системи без належного тестування може зруйнувати існуючі алгоритми.

Це створює явне протиріччя. Рефакторинг є важливим для життєздатності будь-якого програмного продукту, але його ручна реалізація є складним процесом як економічно, так і технічно [9] . Це підкреслює необхідність розробки автоматизованих систем, здатних виявляти проблемні області за допомогою математичних метрик та виконувати безпечну структурну оптимізацію без втручання людини.

Ефективне управління технічним боргом вимагає не лише його фіксації, а й об'єктивного кількісного вимірювання, що виходить за межі суб'єктивних оцінок. Відповідно до міжнародного стандарту якості програмного забезпечення ISO/IEC 25010, підтримуваність коду або іншими словами Maintainability є одним із ключових атрибутів якості системи. Проте на відміну від показників продуктивності або безпеки, оцінка підтримуваності є складною під час стандартного тестування методом чорної скриньки (Black box testing).

Ця складність зумовлює необхідність застосування методів статичного аналізу вихідного коду. Вони дозволяють перетворювати абстрактні концепції, відомі як “запахи коду”, на конкретні математичні величини, тобто інженерні метрики. Впровадження метричного підходу систематизує процес управління технічним боргом, переводячи його з інтуїтивного рівня на інженерний. Це

уможливлює встановлення порогових значень для якості коду та забезпечує автоматизовану відмову від інтеграції низькоякісних програмних модулів у основну гілку розробки.

Окрім завдання вимірювання технічного боргу, важливим аспектом є еволюція інструментальних засобів для рефакторингу. Історично його розвиток пройшов кілька етапів. На початковому етапі ручний рефакторинг повністю залежав від досвіду та кваліфікації розробника, що часто призводило до підвищеного ризику помилок та значного сповільнення процесу.

Напівавтоматичний, або IDE орієнтований рефакторинг, передбачає застосування вбудованих функцій середовищ розробки, як-от “Extract Method” чи “Rename Variable”. Цей підхід значно пришвидшив виконання рутинних, механічних перетворень. Однак його обмеженість синтаксичним рівнем не дозволяє системі інтерпретувати бізнес-логіку алгоритмів або пропонувати архітектурні вдосконалення.

Інтелектуальний або семантичний рефакторинг представляє собою сучасний етап, що став можливим завдяки прогресу в галузі генеративного штучного інтелекту та великих мовних моделей. На відміну від традиційних інструментів, інтегрованих в IDE, штучний інтелект здатний здійснювати аналіз коду на семантичному рівні, адекватно сприймаючи контекст поставленого завдання.

Відповідно, сучасна парадигма забезпечення якості програмного забезпечення диктує необхідність розробки комплексних автоматизованих систем. Ці системи мають інтегрувати принципи строгого детермінованого контролю, заснованого на класичному метричному аналізі, з гнучкими можливостями семантичної оптимізації, реалізованими за допомогою штучного інтелекту. Такий підхід дає змогу не тільки виявляти технічний борг, а й превентивно та автоматизовано його усувати, підтримуючи при цьому високі темпи розробки програмних продуктів.

1.2. Огляд та аналіз показників якості програмного коду

Перехід від інтуїтивного розуміння самою людиною “поганого коду” до професійного управління технічним боргом потребує об’єктивних та кількісних методів оцінки. У розробці програмного забезпечення ці оцінки називаються метриками вихідного коду. Аналіз цих метрик дозволяє мені вимірювати структурну складність, масштаб та когнітивне навантаження алгоритмів, що є важливими для автоматизованих систем рефакторингу. [25]

Розроблена інтелектуальна система базується на аналізі трьох груп базових показників, а саме цикломатичної складності, об'ємних метрик Холстеда та показника можливості підтримки коду.

Цикломатична складність

Цикломатична складність, запропонована Томасом Маккейбом у 1976 році це топологічна метрика, яка вимірює кількість лінійно незалежних шляхів під час виконання програми [3] . Вона базується на представленні вихідного коду як графа потоку керування.

Математично, цикломатична складність V(G) обчислюється за такою формулою:

$$V(G) = E - N + 2P \qquad (1.1)$$

де,

E - кількість ребер, керуючих переходів у графі;
N - кількість вузлів, послідовних блоків коду у графі;
P - кількість пов'язаних компонентів (для функції або методу P = 1).
На практиці цикломатична складність збільшується на одиницю кожного разу, коли використовуються оператори розгалуження та циклу (*if, while, for, case, catch* тощо). Високе значення V(G) вказує на те, що функція виконує забагато перевірок умов, що ускладнює її розуміння та майже не дозволяє повністю охопити її модульними тестами.

Таблиця 1.2 – Інтерпретація значень цикломатичної складності за Маккейбом

Значення V(G)	Оцінка складності	Ризик відмови
1-10	Проста та добре структурована програма	Низький ризик
11–20	Комплексна програма	Середній ризик
21–50	Дуже складна програма	Високий ризик та потребує рефакторингу
Більше 50	Несумісний код	Критичний ризик, неможливо перевірити

Міри складності Холстеда

На відміну від Маккейба який працював над логікою керування, Моріс Холстед у 1977 році запропонував оцінювати обчислювальну складність коду, розглядаючи програму як послідовність операторів таких як ключових слів, математичних символів та операндів змінних, констант [4] . Розрахунок базується на чотирьох ключових показниках.

- n1 - Кількість унікальних операторів;
- n2 - Кількість унікальних операндів;
- N1 - Загальна кількість працюючих операторів;
- N2 - загальна кількість використаних операндів.

На основі цих базових значень розраховуються кілька похідних технічних показників:

1. **Словник програми (словниковий запас):**

$$n = n1 + n2 \quad (1.2)$$

2. **Довжина програми:**

$$N = N1 + N2 \quad (1.3)$$

3. **Об’єм програми:** оцінює інформаційну ємність коду, кількість бітів необхідних для кодування:

$$V = N * \log_2(n) \quad (1.4)$$

4. **Складність розуміння:** це відображає зусилля, які розробник повинен докласти для розуміння алгоритму. Складність зростає, коли використовується багато різних операторів, а операнди часто повторюються.

$$D = \frac{n1}{2} * \frac{N2}{n2} \quad (1.5)$$

5. **Когнітивні зусилля:** комплексний показник для оцінки розумових зусиль, необхідних для розробки або модифікації певного фрагмента коду:

$$E = D * V \quad (1.6)$$

Метрика зусиль “E” дуже важлива важлива для автоматизованих систем рефакторингу, оскільки її зниження після втручання штучного інтелекту є прямим доказом того, що код став більш читабельним для людини.

Індекс ремонтопридатності

Індекс підтримуваності (MI) - це складений показник, розроблений в Університеті Айдахо (Оман та Хагемайстер 1992) [5] . Він об'єднує цикломатичну складність, обсяг за Холстедом та кількість рядків коду (LOC) в одне числове значення, яке характеризує загальну підтримуваність коду.

Вихідна математична модель індексу розраховується за такою формулою:

$$MI = 171 - 5{,}2 * \ln(V) - 0{,}23 * V(G) - 16{,}2 * \ln(LOC) \quad (1.7)$$

де V - об'єм Холстеда, V(G) – цикломатична складність, а LOC – кількість рядків коду без коментарів.

Сучасні інструменти статичного аналізу коду, включаючи бібліотеку Radon для Python, яка використовується в цій роботі, модифікують цю формулу для кращої читабельності та нормалізують значення до 100-бальної шкали (де 100 означає ідеальний код, а 0 – код, який неможливо підтримувати) [19] :

$$MI_{normalized} = max(0, \frac{MI * 100}{171}) \quad (1.8)$$

Загальновизнані галузеві стандарти інтерпретують нормалізований коефіцієнт підтримки наступним чином:

- **0–19:**Погана підтримуваність (код потребує негайного рефакторингу).
- **20–99:**Висока зручність обслуговування (код добре структурований та зрозумілий).

Застосування описаних метрик, таким чином створює міцну математичну основу для оцінки якості програмного забезпечення. Цикломатична складність контролює архітектурну складність, метрики Холстеда оцінюють інформаційне навантаження, а індекс підтримуваності дозволяє комплексно оцінити доцільність проекту. Використання цих показників як критеріїв оптимізації, критеріїв придатності дозволяє не лише математично обгрунтувати необхідність втручання, але й перевірити ефективність коду, що генерується моделями штучного інтелекту.

Базові показники об'єму та щільності (необроблені показники)

Окрім складних композитних метрик, базовий статичний аналіз вимагає розрахунку метрик розміру вихідного коду. Ці метрики є критично важливими, оскільки вони служать вхідними параметрами для складніших математичних моделей.

Ключові показники ефективності включають:

- **LOC (рядки коду):** загальна кількість фізичних рядків у файлі.
- **LLOC (рядки логічного коду):** кількість логічних інструкцій (наприклад, два оператори, розділені крапкою з комою на фізичному рядку, враховуються як два логічні рядки).
- **SLOC (рядки вихідного коду):** кількість рядків вихідного коду, без урахування порожніх рядків та коментарів.

У контексті автоматизованого рефакторингу на основі штучного інтелекту особливо важливим є показник щільності коментарів (CD). Оскільки однією з цілей інтелектуальної оптимізації є автоматична генерація документації та пояснення складних алгоритмів, варіації цього показника надають прямі докази ефективності системи.

Щільність коментарів розраховується як відношення кількості рядків коментарів CLOC до загальної кількості логічних рядків коду та виражається у відсотках:

$$CD = \frac{CLOC}{SLOC + CLOC} * 100\% \quad (1.9)$$

Галузеві стандарти зазвичай вважають, що CD знаходиться в межах від 20% до 30%. Занадто низьке значення вказує на недостатню документацію, тоді як занадто високе значення вказує на невикористаний або некоментований код, або надмірне пояснення очевидних концепцій.

Математична модель для оцінки подібності коду (алгоритм Ратклiффа-Обершельпа)

Під час автоматизованого рефакторингу завдання полягає не лише в покращенні коду, але й у візуальній ілюстрації змін для розробників. Для алгоритмічного порівняння оригінального та оптимізованого вихідного коду, так скажемо візуального аналізу різниці, використовуються алгоритми визначення найдовшої спільної підпослідовності.

Одним із найефективніших методів виконання такого порівняння є алгоритм розпізнавання образів Ратклiффа-Обершельпа, також відомий як гештальт зіставлення зі зразком. [17]

Алгоритм працює за принципом “розділяй і володарюй”

1. Між двома послідовностями коду існує найдовший спільний підрядок.
2. Тексти поділено на дві частини. Ліворуч і праворуч від знайденого спільного підрядка.
3. Процес повторюється рекурсивно для лівої та правої сторін, доки не будуть знайдені всі збіги.

Для кількісної оцінки того, наскільки рефакторований код відрізняється від оригіналу, використовується індекс подібності D_r , який розраховується за такою формулою:

$$D_r = \frac{2 * M}{T} \quad (1.10)$$

де,

- M - загальна кількість символів (або цілих рядків коду), що збігаються в обох послідовностях;

- T - загальна кількість символів (або рядків) в обох текстах разом

$$T = N_{old} + N_{new} \quad (1.11)$$

Значення D_r коливається від 0 (зовсім різні тексти) до 1 (ідентичні тексти). Використання цієї математичної моделі в розробленій системі дозволяє дуже точно ідентифікувати видалені, додані та змінені рядки, тим самим мінімізуючи “шум” у візуалізації результатів рефакторингу на основі штучного інтелекту.

1.3. Штучний інтелект та моделі великих мов програмування (LLM) у процесах розробки програмного забезпечення

Традиційні підходи до статичного аналізу коду та рефакторингу коду, описані в попередніх розділах, спираються на жорсткі детерміновані правила та абстрактні синтаксичні дерева AST. Хоча вони добре виявляють відхилення від метрик, вони не здатні самостійно розуміти бізнес логіку або пропонувати архітектурні покращення. Перехід від пасивної діагностики технічного боргу до його активного та автоматичного усунення став можливим завдяки швидкому розвитку методів штучного інтелекту, особливо нейронних мереж для обробки природної мови NLP та великомасштабних мовних моделей LLM.

Розробка застосувань штучного інтелекту в розробці програмного забезпечення

Використання штучного інтелекту в розробці програмного забезпечення (ШІ для розробки програмного забезпечення AI for Software Engineering - AI4SE) еволюціонувало від простих евристик до глибокого навчання. Історично ранні спроби автоматизації програмування спиралися на експертні системи та бази знань, що вимагало ручного написання тисяч правил для перетворення коду.

Справжній прорив відбувся з використанням статистичних методів та концепції коду як природної мови [26] . Дослідження показали, що, незважаючи на суворий синтаксис, вихідний код демонструє високий рівень передбачуваності (передбачуваність токенів та структури), подібний до поширених людських мов.

[10] Це дозволило застосувати до коду ті ж архітектури нейронних мереж, що використовуються для машинного перекладу.

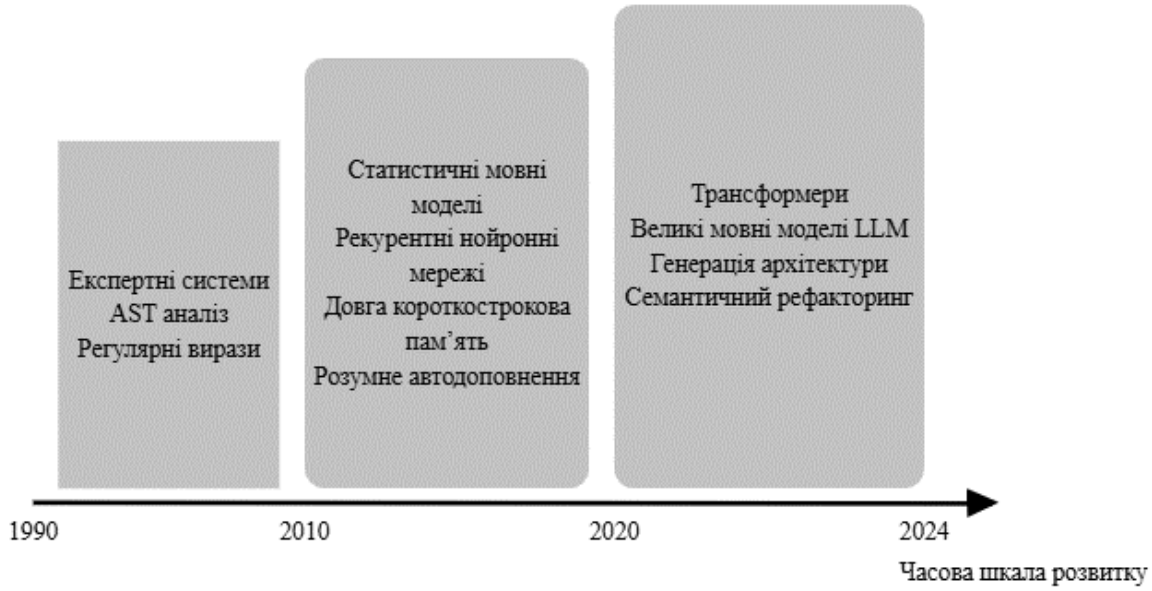


Рис. 1.2 Розробка методів штучного інтелекту в діяльності з розробки програмного забезпечення

Поточна фаза AI4SE повністю базується на використанні генеративного штучного інтелекту. У цьому контексті завдання автоматичного рефакторингу математично зводиться до завдання машинного перекладу. Модель приймає на вхід послідовність токенів, що представляють “нечистий” код або вихідна мова, та генерує послідовність токенів, що представляють оптимізований код або цільова мова, зберігаючи при цьому оригінальний семантичний контент.

Архітектурні особливості великих мовних моделей

Основою всіх сучасних систем обробки мови є архітектура Transformer, вперше представлена дослідниками Google у 2017 році. На відміну від рекурентних нейронних мереж , які обробляють код послідовно рядок за рядком, Transformers використовують механізм самосвідомості.

Цей механізм дозволяє моделі одночасно обробляти всі токени слова, змінні, оператори та обчислювати математичну вагу зв'язків між окремими елементами коду, незалежно від відстані, яка їх розділяє у файлі.

Функція уваги розраховується за такою формулою:

$$Attention(Q, K, V) = softmax(\frac{Q * K^T}{\sqrt{d_k}}) * V \qquad (1.12)$$

де Q - запит, K - ключ та V - значення – це матриці, створені лінійним перетворенням токенів вхідного коду, а d_k розмір векторів ключів.

Завдяки великій увазі модель може зосереджуватися на різних аспектах коду одночасно. Одна “голова” уваги контролює синтаксичну правильність (наприклад, чи закриті дужки), інша контролює типи даних змінних, а третя контролює загальну бізнес логіку циклу або функції.

Ця здатність розуміти великий контекст робить LLM ідеальним інструментом для рефакторингу. Сучасні моделі мають контекстне вікно від 32 000 до 128 000 тисяч токенів або більше. Це дозволяє їм аналізувати не лише окремі функції, але й цілі класи або програмні модулі, а також виявляти антишаблони на рівні “об'єкта бога” або глобального дублювання.

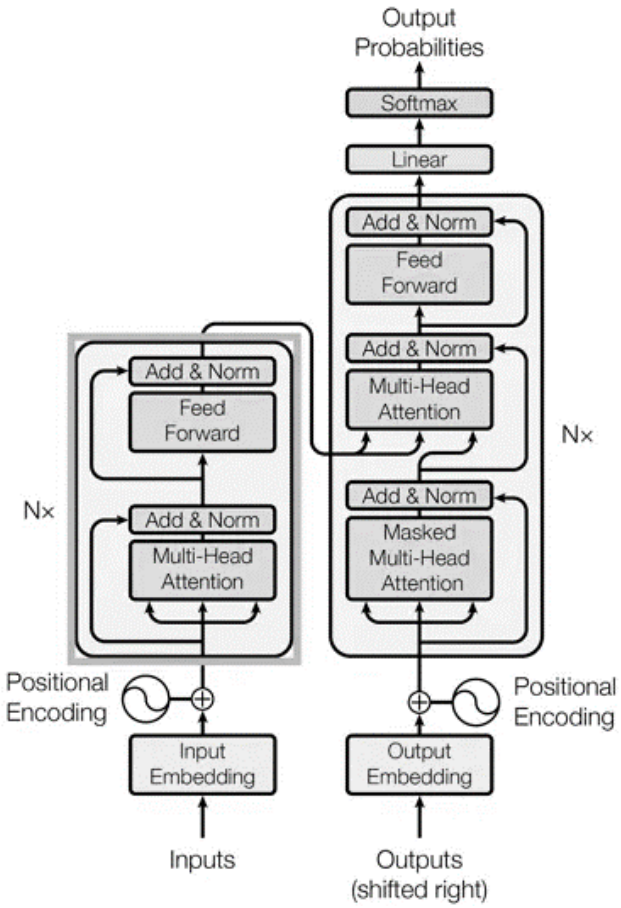


Рис. 1.3 Архітектура Transformer

Можливості та обмеження LLM у рефакторингових заходах

Застосування моделі LLM до процесів розробки програмного забезпечення відкриває безпрецедентні можливості для оптимізації технічного боргу. [6]

Ключові особливості програми LLM у програмувані:

1. **Зменшення когнітивної складності:** штучний інтелект може аналізувати складні вкладені цикли та складні умовні оператори, тобто вкладені оператори *if-else* шляхом автоматичної генерації еквівалентного лінійного коду (наприклад, за допомогою шаблонів захисних речень або поліморфізму).
2. **Автоматичне написання:** для динамічно типізованих мов прикладом якої є Python, LLM аналізує потік виконання та автоматично виводить і додає анотації типів, що значно підвищує надійність системи та полегшує подальший статичний аналіз.
3. **Створення документації:** шаблони можуть розуміти тонку логіку “застарілого коду” та генерувати повні коментарі для нього у стандартних форматах PEP 257, Doxygen або Javadoc.

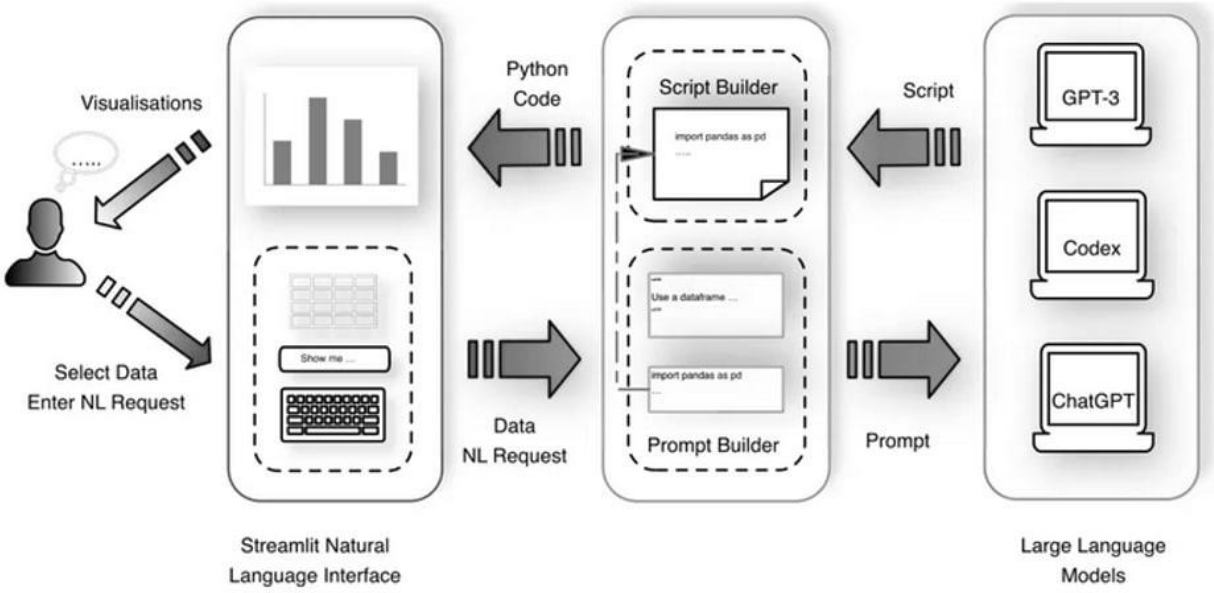


Рис. 1.4 LLM у програмуванні

Таблиця 1.3 Порівняльний аналіз традиційного рефакторингу IDE та інтелектуального рефакторингу LLM

Критерій	Традиційні інструменти IDE	Інтелектуальні системи LLM
Рівень абстракції	Синтаксичний. Перейменування, видалення методів	Семантичний. Зміна парадигми, перезапис логіки
Контекстуальне розуміння	Обмежене. Це означає лише жорсткі посилання AST	Глобальне. Аналіз бізнес вимог та неявні посилання
Генерація документації	Шаблон Порожні блоки @param, @return	Змістовний. Опис алгоритму природною мовою
Ризик застосування	Низький , бо гарантовано збереження AST	Середній , бо є ризик галюцинацій моделі, потрібна валідація

Незважаючи на свої потужні можливості, використання LLM має кілька обмежень. Найважливішим з яких є проблема “галюцинацій”. [8] У контексті генерації коду галюцинації означають ситуації, коли нейронна мережа створює синтаксично правильний та візуально переконливий код, але який містить логічні помилки. Використовує неіснуючі функції або змінює початкову логіку програми. Ще одним обмеженням є відсутність детермінізму. Для одного й того ж вхідного запиту модель може генерувати різні варіанти відповідей залежно від параметра.

Ці обмеження вимагають розробки гібридних систем. Метод LLM не слід використовувати без розбору. Його результати необхідно перевіряти та кількісно оцінювати за допомогою класичних метрик якості, а саме цикломатична складність, індекс підтримуваності, описаних у розділі 1.2. Згенерований код буде інтегровано в проект лише за умови демонстрації математичного покращення.

Архітектура та переваги використання моделей серії Mistral

Для керування автоматизованими завданнями рефакторингу в моїй системі було обрано архітектуру нейронної мережі з сімейства моделей Mistral. [14] На відміну від рішень із закритим кодом, таких як GPT-4 від OpenAI, моделі штучного інтелекту Mistral пропонують оптимальний баланс між продуктивністю, архітектурною прозорістю та вартістю виводу або іншими словами обчислювальними зусиллями.

Відмінною особливістю архітектури Mistral є використання низки інноваційних механізмів, які роблять її надзвичайно ефективною в аналізі програмного коду

1. **Sliding Window Attention:** на відміну від традиційних механізмів глобальної уваги, де кожен токен враховує всі попередні токени з квадратичною складністю $O(N^2)$, моделі Mistral використовують змінне вікно з фіксованим розміром 4096 токенів. Це дозволило лінійно обробляти великі файли вихідного коду без перевантаження пам'яті графічного процесора.
2. **Управління якістю групи:** механізм групування запитів значно пришвидшує генерацію коду, одночасно зменшуючи затримку. Це критично важливо для інтерактивних настільних застосунків, де користувачі очікують результатів у режимі реального часу після рефакторингу.
3. **Сукупність експертів:** у розширених конфігураціях мережа складається з кількох експертних підмереж. Під час аналізу коду активується лише та частина мережі, яка найкраще підходить для певної мови програмування або типу завдання (наприклад, оптимізації пам'яті або алгоритмів сортування). Це покращує загальну якість згенерованого коду.

Завдяки доступу до моделей через високошвидкісні API такі як платформи NVIDIA NIM, потужність великих нейронних мереж (з десятками або сотнями мільярдів параметрів) може бути безпосередньо інтегрована в легко сумісні настільні програми без необхідності використання потужного обладнання для кінцевого користувача. [15]

Конструювання підказок як метод управління LLM

Ефективність генеративного ШІ для рефакторингу залежить не лише від архітектури моделі, але й від якості контекстних запитів, що генеруються системою. Цей процес відомий як підсказуюча інженерія. Модель поводиться як стохастичний папуга, тому для досягнення передбачуваного результату необхідний послідовний алгоритмічний контроль вхідних інструкцій.

У розробленій системі взаємодія з моделлю базується на системній вимозі, яка накладає суворі обмеження та ліміти на поведінку штучного інтелекту.

Структура ефективної системної вимоги для рефакторингу включає чотири обов'язкові компоненти.

1. **Запит на роль:** контекстуалізація моделі. Наприклад, “Ви досвідчений розробник Python та експерт з архітектури програмного забезпечення”) активує нейрони у ваговому просторі мережі, які відповідають за вузькоспеціалізовану лексику та технічні шаблони.
2. **Постановка цілі:** це чітко показує, що потрібно покращити. Система алгоритмічно надсилає запит на зменшення цикломатичної складності та збільшення індексу підтримки.
3. **Обмеження вихідного формату:** щоб застосунок міг розібрати та програмно використати згенерований код, а саме відобразити його у віджеті візуального порівняння шаблон має бути обмежений щодо генерації тексту. Повідомлення забороняє використання Markdown, викликів або детальних пояснень. Дозволено лише чистий, скомпільований код.
4. **Контекст галузевих стандартів:** обов'язково дотримуватися стандарту дизайну PEP 8 та використовувати анотації типів підказок щодо типів typing та використання української мови для підготовки документації. [21]

Таким чином, застосування чітких промптів дозволяє нам інкапсулювати складність штучного інтелекту, перетворюючи загальну розмовну модель на спеціалізований, детермінований інструмент API для оптимізації програмного забезпечення.

Інтеграція методів математичного статичного аналізу (описаних у розділі 1.2) із семантичними можливостями великомасштабних мовних моделей створює синергетичний ефект та закладає міцну теоретичну та методологічну основу для розробки автоматизованої інтелектуальної системи рефакторингу, проектування якої детально аналізується у розділі 2.

1.4. Аналіз існуючих рішень та формулювання задачі

Усунення технічного боргу та оптимізація вихідного коду вже давно є фундаментальними аспектами розробки програмного забезпечення. Ринок пропонує різні інструменти для покращення якості програмного забезпечення. Однак ретельний аналіз показує, що існуючі рішення поділяються на дві окремі категорії. Першою є детерміновані системи статичного аналізу. Другою є генеративні помічники на основі штучного інтелекту. Кожна з цих категорій має фундаментальні обмеження, що перешкоджають повній автоматизації процесу безпечного рефакторингу.

Аналіз детермінованих систем статичного аналізу

Перша категорія включає класичні аналізатори коду, лінтерні інструменти та платформи безперервної перевірки, такі як SonarQube, Pylint, Flake8, ESLint тощо.

Ці системи базуються на створенні абстрактних синтаксичних дерев AST та використанні суворо визначених наборів правил. Їхньою головною перевагою є абсолютна математична точність і передбачуваність. Вони можуть миттєво обчислювати цикломатичну складність, виявляти дублікати або позначати порушення стандартів форматування.

Як видно на рисунку 1.3, система SonarQube ефективно визначає проблемні області, але залишає розробнику право їх виправлення, створюючи прогалину в робочому процесі.

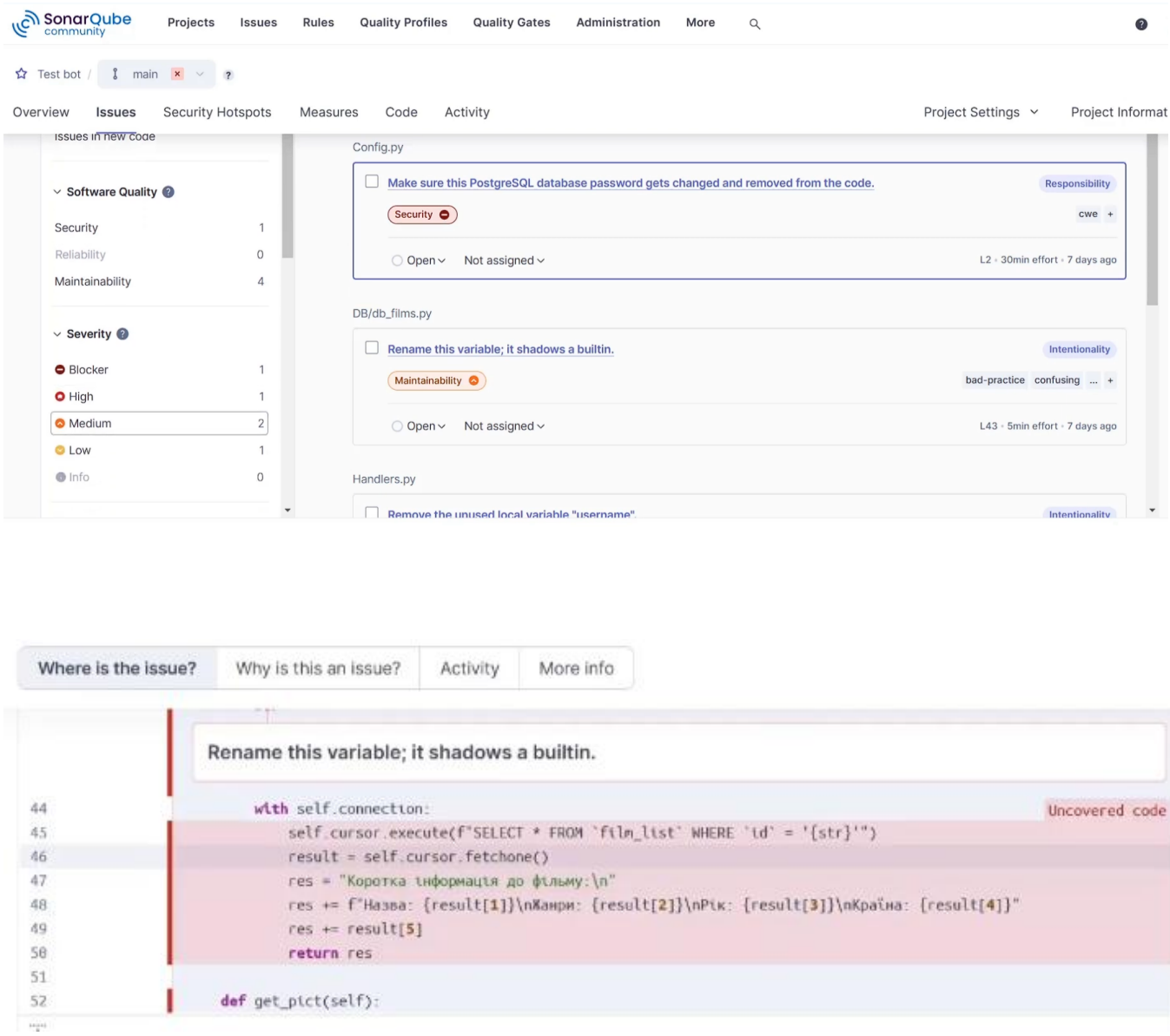


Рис. 1.5 Приклад діагностики технічного боргу в SonarQube

Основними недоліками детермінованих систем є:

1. **Пасивна діагностика:** аналізатори коду служать лише індикаторами проблеми. Вони генерують повідомлення які можуть бути перевантажені незначними попередженнями, відомими як шум, але не пропонують заздалегідь визначених архітектурних рішень.
2. **Передача когнітивного навантаження:** як тільки функція з цикломатичною складністю V(G) = 35 ідентифікована, SonarQube просто блокує запит на злиття коду. Вся інтелектуальна робота з аналізу алгоритму,

переписування умов та запуску нових тестів лягає на розробника. Це призводить до значної втрати часу розробника.

3. **Синтаксичні обмеження:** ці інструменти не розуміють семантики чи логіки алгоритму і тому не можуть запропонувати рефакторинг, який вимагає зміни парадигми. Перехід від процедурного підходу до об'єктно орієнтованого поліморфізму.

Аналіз можливостей асистентів програмування на основі штучного інтелекту

Друга, новіша категорія інструментів- це помічники, засновані на складних мовних моделях: GitHub Copilot, ChatGPT, Tabnine та Amazon Q Developer. Ці інструменти глибоко інтегровані в IDE або доступні через веб інтерфейси та можуть генерувати, завершувати та пояснювати код на основі запитів природною мовою.

Головною перевагою цих інструментів є їхня здатність розуміти семантичний контекст. Вони можуть швидко генерувати оптимізовану версію функції або створювати пов'язану з нею документацію.

Асистенти на основі ШІ (рис. 1.4) пропонують швидкі рішення, але вони не є детермінованими та не підкріплені кількісними показниками покращення якості структури коду.

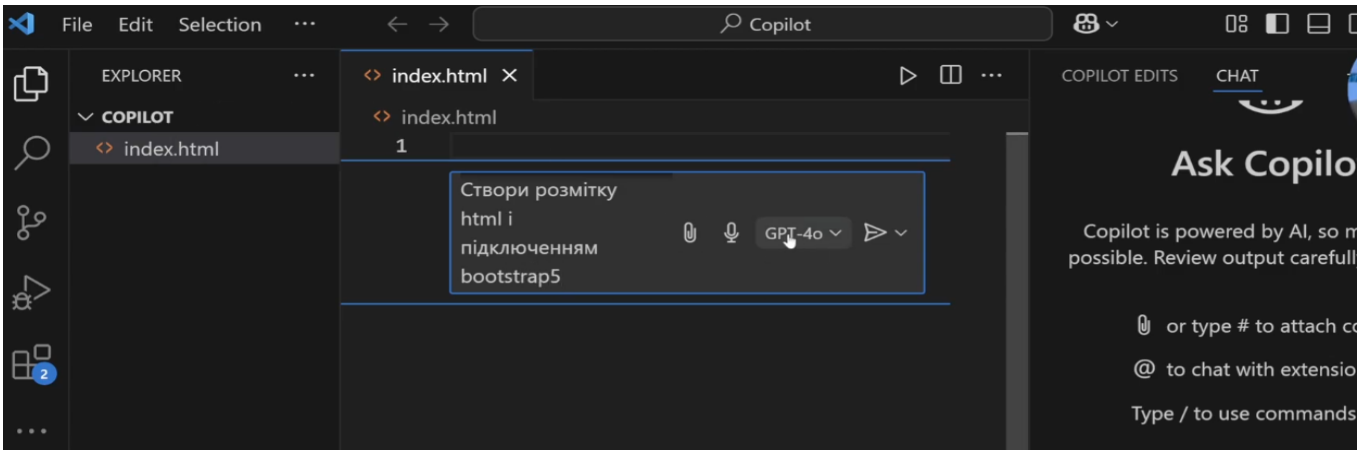


Рис. 1.6 Створення запиту до GitHub Copilot

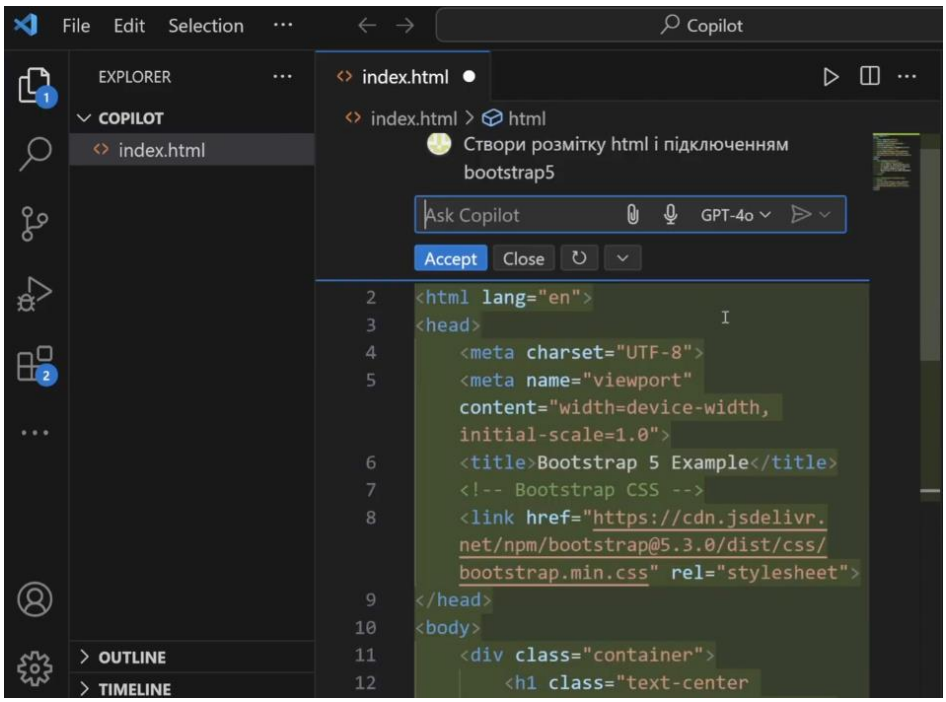


Рис. 1.7 Процес генерації коду за допомогою інтелектуального помічника GitHub Copilot

Основні недоліки асистентів на основі штучного інтелекту в контексті рефакторингу:

1. **Недостатня валідація метрик:** моделі штучного інтелекту працюють як генератори тексту на основі принципу розподілу ймовірностей токенів. Їм бракує вбудованого математичного механізму для перевірки згенерованого виводу. Тому користувач не може бути впевнений, чи код штучного інтелекту, який вони розробляють, насправді має меншу складність згідно з визначенням Холстеда, чи він просто виглядає візуально більш лаконічним.
2. **Проблема галюцинацій:** модель може застосовувати неіснуючі методи зі стандартних бібліотек або змінювати поведінку алгоритму в крайніх випадках.
3. **Нестабільність форматування:** без послідовного системного контролю або іншими словами промптів на рівні API, прості чатботи генерують непотрібний текст. Пояснення, кодування в Markdown, що унеможливлює автоматичну інтеграцію згенерованого коду у файли проекту.

Таблиця 1.4 Порівняльний аналіз функціональності систем для аналізу коду та рефакторингу.

Критерії порівняння	SonarQube	GitHub Copilot	Розроблена гібридна система
Об'єктивна оцінка складності	Розрахунок ключових показників ефективності (KPI, MI) у режимі реального часу	Немає, бо модель не має математичного ядра	Повний аналіз показників до та після змін
Автоматичне виправлення коду	Немає, є лише рекомендації та повідомлення про помилки	Відмінна здатність генерувати та додавати код	Повна реструктуризація на основі семантичного аналізу
Валідація результатів штучного інтелекту	Недоступно, бо системи не інтегровані	Суб'єктивно. Це означає що все залишається на розсуд розробника	Автоматична перевірка через зміни технічних показників
Документація та коментарі	Тільки перевірки чи є у коді коментарі	Документація та коментарі робляться на запит	Обов'язкова генерація рядків документації, коментарів та підказок типів
Переглянути зміни	Тільки у веб інтерфейсі для історії підтверджень	Обмежено інструментами, вбудованими в IDE	Спеціальний диференціальний модуль з кольоровим кодуванням
Відповідність стандартам PEP 8	Суворий контроль без автоматичної корекції	Це залежить від контексту та поставленого завдання	Суворе дотримання вимог через системні обмеження та чіткі промпти

Обґрунтування необхідності розробки гібридної системи

Аналіз існуючих рішень виявляє значну прогалину в існуванні інструментів. Брак систем, які поєднують математичну точність статичного аналізу з генеративною гнучкістю штучного інтелекту в єдиній замкнутій системі керування.

Розробнику доводиться використовувати кілька окремих інструментів. Спочатку проаналізувати код за допомогою лінтера. Потім скопіювати проблемну частину в ChatGPT, запросити виправлення, вручну перенести код і, нарешті, знову запустити лінтер для перевірки. Цей процес неефективний і громіздкий.

Потреба в спеціалізованому програмному засобі очевидна. Цим інструментом є гібридна система оцінки якості та рефакторингу. [7] Ця система повинна виступати одночасно посередником і контрольною точкою. Вона повинна автономно обчислювати основні метрики, надсилати точний алгоритмічний запит до потужної мовної моделі через API, отримувати та перевіряти результат, а також чітко представляти розробнику кількісні покращення. Для цього використовуючи таблиці порівняння, математичні графіки та візуальний аналіз відмінностей.

Визначення завдання на кваліфікаційну роботу

На основі теоретичного аналізу (Розділ 1) та виявлених недоліків існуючих програмних рішень сформульовано основну мету роботи: розробка та впровадження настільного програмного застосунку для покращення якості вихідного коду за допомогою автоматизованого аналізу метрик та керованого LLM рефакторингу.

Для досягнення бажаної мети розроблена система повинна відповідати наступним функціональним вимогам:

1. **Модуль статичного аналізу:** виконайте аналіз вихідного коду на Python з розрахунком ключових показників до та після втручання алгоритму. Ключовими показниками є цикломатична складність, індекс підтримуваності, метрики Холстеда, кількість рядків коду.

2. **Модуль інтелектуального рефакторингу:** інтеграція із сучасною, розширеної мовною моделлі Mistral з хмарним API дозволяє рефакторинг [16] . Система розроблена для генерації оптимізованого коду, який відповідає стандартам проектування PEP 8, містить анотації типів та надає вичерпну документацію українською мовою.

3. **Модуль відмовостійкості:** наявність резервного механізму обробки даних у разі збою інтернет з'єднання або недоступності зовнішнього API, для забезпечення безперебійної роботи програми.

4. **Модуль візуалізації та порівняння:** розробка інструментів алгоритмічного порівняння текстів на основі бібліотеки difflib для змін колірного кодування та інтеграція математичних графіків стовпчастих діаграм Matplotlib для візуального представлення змін ключових технічних індикаторів.

5. **Вимоги до графічного інтерфейсу користувача:** додаток повинен мати сучасний інтерфейс користувача в темному стилі або темний режим, містити спеціальний текстовий редактор з підсвічуванням синтаксису та підтримувати асинхронне виконання ресурсомістких завдань (багатопоточність), щоб запобігти зависанню головного вікна програми під час запитів до нейронної мережі.

Реалізація описаних завдань дозволяє розробити інноваційний інструмент, який переводить скорочення технічного боргу з рутинних ручних процесів на рівень контрольованої автоматизації. Вибір технологічного пакету, архітектурне проектування та безпосереднє впровадження системного програмного забезпечення детально пояснюються в наступних розділах цього документа.

ВИСНОВОК ДО РОЗДІЛУ 1

У першій частині кваліфікаційної роботи представлено комплексний аналіз теоретичних та методологічних основ оцінки та покращення якості програмного забезпечення. Основні висновки дослідження включають наступне:

Було продемонстровано критичний вплив технічного боргу на життєвий цикл програмного забезпечення. Виявлено, що накопичення архітектурних дефектів таких як запахи коду експоненціально збільшує витрати на підтримку та модифікацію системи. Це підкреслює необхідність періодичного рефакторингу, який, якщо його виконувати вручну, є економічно ефективним, але трудомістким процесом.

Доказано необхідність використання інженерних метрик як критеріїв якості. Перехід від суб'єктивних оцінок до детермінованих математичних моделей дозволяє кількісно виміряти технічний борг. Було визначено, що найбільш релевантними метриками для оцінки ефективності рефакторингу є цикломатична складність Маккейба, або іншими словами архітектурний контроль, об'ємні метрики Холстеда, такі як оцінка когнітивного навантаження та інтегральний індекс підтримки коду.

Було встановлено необхідність використання інженерних метрик як критеріїв якості. Було досліджено потенціал та обмеження генеративного ШІ. Аналіз сучасних підходів показав, що великі мовні моделі LLM, зокрема архітектури сімейства Mistral, дозволяють проводити поглиблений семантичний рефакторинг та генерувати технічну документацію. Однак їх використання вимагає суворого контролю за допомогою гнучких інженерних механізмів, таких як промпти, для запобігання неправильним інтерпретаціям та неочікуваним змінам бізнес логіки.

В існуючих інструментах було виявлено функціональну прогалину. Аналіз ринку показав, що сучасні рішення можна розділити на два окремі класи. Статичні аналізатори, які надають точні метрики, але не налагоджують код та помічники штучного інтелекту, які генерують код, але не мають механізмів для перевірки метрик результатів.

Було сформульовано завдання розробки. На основі виявлених недоліків в існуючих рішеннях виникла потреба в гібридній десктопній системі. Розроблений додаток має на меті поєднати модуль розрахунку математичних метрик з модулем керованого рефакторингу LLM у замкнутому циклі з обов'язковою візуалізацією

кількісних та структурних змін коду. Це створило міцну теоретичну основу для розробки архітектури автоматизованої системи аналізу метрик та рефакторингу.

РОЗДІЛ 2. ПРОЄКТУВАННЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ АНАЛІЗУ МЕТРИК ТА РЕФАКТОРИНГУ

2.1. Формування вимог до програмного забезпечення

Проектування будь-якого складного програмного забезпечення починається з аналізу потреб. Відповідно до міжнародного стандарту ISO/IEC/IEEE 29148:2018, специфікації точно визначають функціональні можливості, які система повинна пропонувати, а саме функціональні вимоги, та якісні характеристики, якими вона повинна володіти такі як нефункціональні вимоги.

Оскільки розроблена система має гібридну архітектуру для поєднання локальних математичних обчислень з асинхронними віддаленими запитами до нейронної мережі, були пред'явлені вищі вимоги щодо відмовостійкості та зручності використання.

Функціональні вимоги

Функціональні вимоги описують поведінку системи та її взаємодію з користувачем і зовнішніми сервісами. Для ясності та полегшення тестування основні функціональні вимоги розробленого застосунку наведено в таблиці 2.1.

Таблиця 2.1 Специфікація функціональних вимог до системи

Ідентифікатор запиту	Назва модуля	Опис функціональних вимог
FR-01	Модуль вводу/виводу	Система повинна дозволяти користувачеві завантажувати вихідний код Python з локальних файлів (.py) або вводити текст безпосередньо в редактор.
FR-02	Аналітичне ядро	Програма повинна виконати лексико-синтаксичний аналіз AST завантаженого коду, автоматично обчислюючи цикломатичну складність, метрики Холстеда, індекс підтримуваності MI та кількість рядків LOC.

Ідентифікатор запиту	Назва модуля	Опис функціональних вимог
FR-03	Інтеграція LLM	Система повинна згенерувати певний запит, іншими словами промт, та надіслати його через API до хмарної мовної моделі Mistral для автоматичної генерації оптимізованого коду.
FR-04	Модуль рефакторингу	Згенерований код має бути автоматично перевірений на наявність української документації та вказівок на типи. Після генерації система має перерахувати метрики нового коду.
FR-05	Модуль візуалізації змін	Програма розроблена для алгоритмічного порівняння старого та нового коду, аналіз різниці, та виділення результатів різним кольором. Червоний - видалені рядки, зелений – додані.
FR-06	Аналітичні звіти	Система повинна генерувати візуальні графіки, а саме стовпчасті діаграми для порівняння метрик коду до та після процесу рефакторингу.
FR-07	Керування API	Додаток повинен дозволяти користувачеві безпечно вводити та зберігати власний ключ API для доступу до нейронної мережі у хмарі.

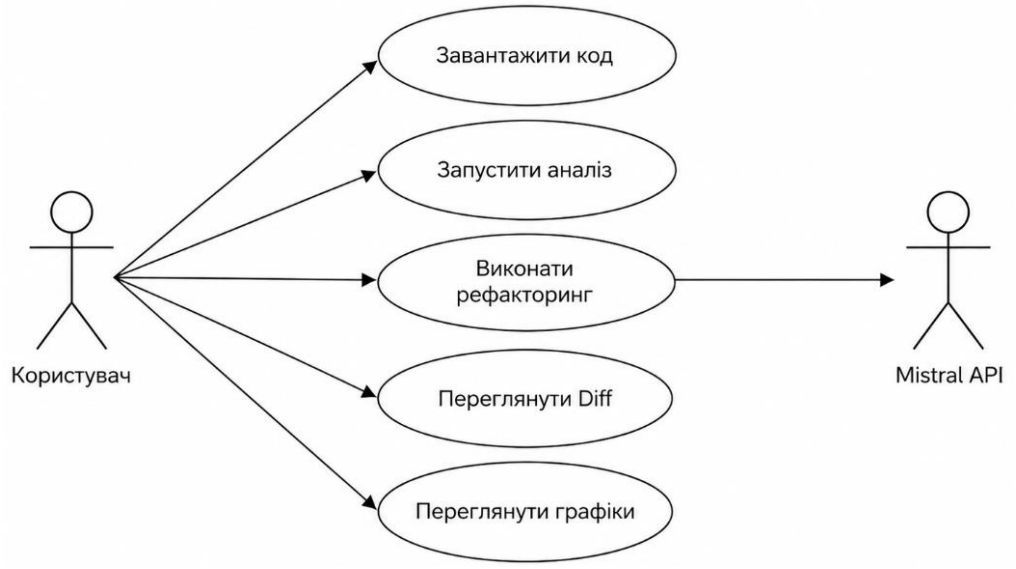


Рис. 2.1 Діаграма варіантів використання автоматизованої системи

Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, такі як продуктивність, надійність та зручність використання. Вони є дуже важливими для програмних додатків, оскільки безпосередньо впливають на досвід розробника.

1. **Вимоги до продуктивності та швидкості реагування:**

Запити до нейронної мережі в хмарі можуть тривати від 20 секунд до 2-х хвилин. Система повинна забезпечувати асинхронне виконання ресурсомістких завдань у фонових потоках іншими словами багатопроцесорність, щоб запобігти повному зависанню графічного інтерфейсу користувача, або ефекту “програма не відповідає” у Windows.

Під час очікування відповіді від API користувач повинен побачити індикатор виконання або інформативне повідомлення щодо статусу завдання.

2. **Вимоги до надійності та відмовостійкості:**

Механізм перестраховки: програма не повинна аварійно завершувати роботу у разі відсутності інтернет з’єднання, перевищення лімітів використання хмарного API або введення недійсного ключа API. Система повинна перехопити виняток і автоматично переключитися на локальний механізм моделювання, який імітує базовий рефакторинг, щоб забезпечити безперервну роботу програми.

3. **Вимоги до інтерфейсу користувача та зручності використання:**

Інтерфейс програми має бути реалізований відповідно до принципів проектування сучасних інтегрованих середовищ розробки. Зокрема, він повинен включати темний режим для зменшення навантаження на очі під час тривалої роботи.

Текстові поля для введення та відображення коду повинні містити непропорційний шрифт, інтегровану нумерацію рядків та базове підсвічування синтаксису.

Навігація між оригінальним кодом, переробленим результатом, порівнянням відмінностей та діаграмами повинна здійснюватися за допомогою інтуїтивно зрозумілої системи вкладок.

4. **Вимоги до операційного середовища:**

Програма має бути кросплатформною, тобто вона повинна мати можливість працювати в операційних системах Windows, Linux та macOS без необхідності переписування вихідного коду, що забезпечується інтерпретатором мови Python.

Сформульовані функціональні та нефункціональні вимоги забезпечують міцну технічну основу. Вони описують цільову модель програмного продукту, що дозволяє інтелектуально керувати технічним боргом, як визначено в першій частині цього документа. Наступним кроком є проектування внутрішньої архітектури системи та вибір оптимального технологічного стеку для реалізації визначених вимог.

2.2. Вибір архітектури системи та технологічного стеку

Вибір відповідної архітектури програмного забезпечення та технологій є вирішальним кроком у процесі проектування, оскільки він безпосередньо впливає на продуктивність системи, її масштабованість та майбутню підтримку. Відповідно до вимог, описаних у розділі 2.1, для системи, що розробляється, було обрано гібридну архітектуру таку як робоча станція - хмара.

Високорівнева архітектура системи

У гібридній архітектурі навантаження на обчислення розподіляється між локальним комп'ютером користувача та віддаленим сервером у хмарі. [13]

Усі детерміновані та безпечні операції виконуються локально на машині розробника. Рендеринг графічного інтерфейсу користувача, читання файлів, аналіз коду, створення абстрактних синтаксичних дерев AST, обчислення метрик та генерація візуалізацій. Це забезпечує негайну реакцію інтерфейсу користувача. Лише конкретний запит, що містить вихідний код через безпечний API, надсилається до хмари для виконання семантичного рефакторингу за допомогою високопродуктивної моделі штучного інтелекту. Цю модель неможливо реалізувати на локальному ПК через апаратні обмеження.

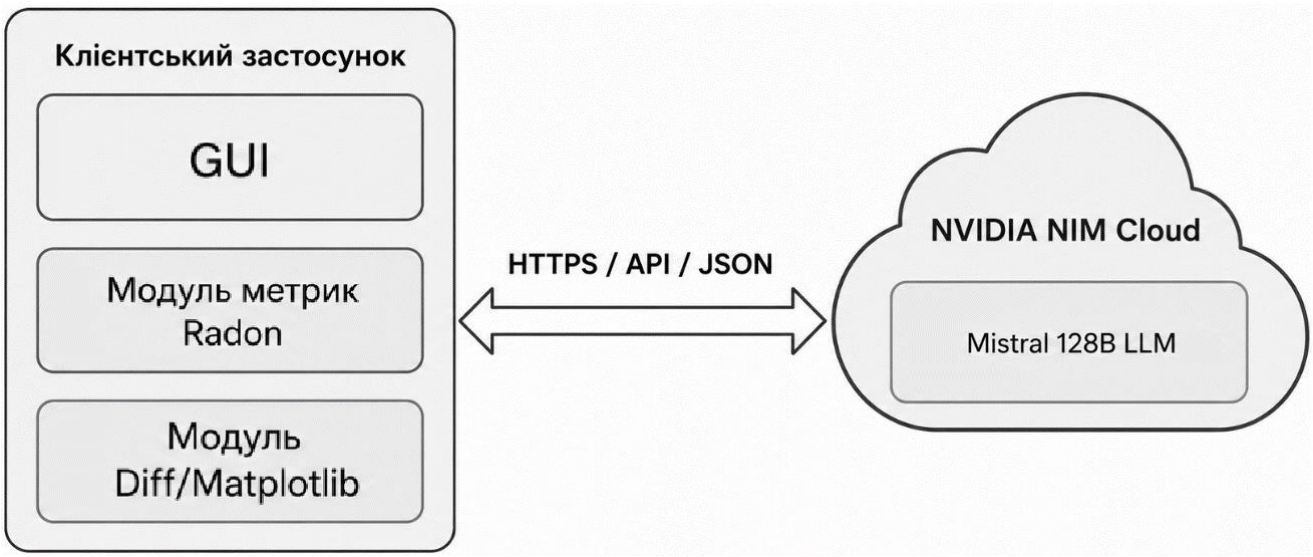


Рис. 2.2 Структурна схема гібридної архітектури програмного забезпечення.

Обґрунтування вибору мови програмування та бібліотек

Мову програмування Python 3 було обрано основним інструментом розробки. Цей вибір ґрунтувався на кількох факторах. Перший, наявності високопродуктивної стандартної бібліотеки для маніпулювання абстрактними синтаксичними деревами модуль ast. Другий, домінуючому становищі Python у сферах штучного інтелекту та аналізу даних. Третій, швидкості створення прототипів.

Для реалізації конкретних функціональних модулів було обрано наступний набір бібліотек:

- **Графічний інтерфейс користувача (GUI).** Використовується вбудована бібліотека Tkinter та її розширення tkinter.ttk. [18] На відміну від ресурсоємних фреймворків, таких як PyQt, Tkinter не потребує додаткових залежностей та споживає дуже мало пам'яті. Завдяки рушію ttk.Style вдалося реалізувати сучасну темну тему, або по іншому кажучи dark mode, яка відповідає стандартам професійних інтегрованих середовищ розробки.
- **Статичний аналіз та метрики.** Для цього використовуємо спеціальну бібліотеку під назвою radon. [19] Завдяки їй цикломатичну складність Маккейба, метрики об'єму Холстеда та індекс підтримки коду MI можна розрахувати безпосередньо з вихідного коду з високою математичною точністю, без компіляції.

- **Візуалізація алгоритмічних змін.** Для виявлення відмінностей між оригінальним та оптимізованим кодом використовується вбудований модуль difflib. Він реалізує алгоритм Ратклiффа Обершельпа, що дозволяє точно порівнювати текст на рівні символів та порядкових чисел.

Створення аналітичних діаграм. Використовується бібліотека Matplotlib модуля Pyplot. Вона є галузевим стандартом у галузі обробки даних і дозволяє програмно створювати векторні гістограми для порівняння метрик коду до та після рефакторингу.

Таблиця 2.2 Специфікація всіх технологій розробки

Системний компонент	Вибрана технологія	Розподіл системи
Базова платформа	Python 3	Забезпечить кросплатформну сумісність, обробку даних та управління робочими процесами.
Графічний інтерфейс користувача	Tkinter + ttk	Створення вікон, вкладок та текстових редакторів, а також реалізація темного режиму.
Статичний аналіз	Radon	Проектування AST, розрахунок цикломатичної складності та індекс підтримуваності.
Модуль порівняння	Difflib	Алгоритмічне порівняння текстів, маркування видалених та доданих рядків.
Генерація графіків	Matplotlib	Створення стовпчастих діаграм для порівняння ключових показників до та після рефакторингу.
Модуль штучного інтелекту	API NVIDIA NIM Mistral	Хмарна генерація оптимізованого коду, документації та набору тексту.

Вибір хмарної інфраструктури API

Для забезпечення інтелектуальних функцій системи, а саме рефакторинг LLM, було обрано архітектуру моделі Mistral, доступну через хмарну платформу NVIDIA NIM API. Які у цього є переваги:

1. **Високошвидкісний висновок:** сервери NVIDIA оптимізовані для тензорних обчислень, що зменшує затримку відгуку (20-120 секунд), навіть для великих фрагментів коду.

2. **Сумісність інтерфейсів:** API NVIDIA використовує стандартизований протокол взаємодії, ідентичний протоколу API OpenAI. Це дозволяє створити гнучку архітектуру розробленого застосунку. За необхідності систему можна налаштувати на використання інших моделей (наприклад, LLaMA або GPT), просто змінивши базову URL-адресу.

Середовище розробки

Написання, налагодження та тестування вихідного коду розробленої системи виконується в середовищі Visual Studio Code (VS Code) корпорації Microsoft. Цей редактор був обраний через такі переваги:

1. **Гнучкість та можливості розширення:** на відміну від складних, монолітних середовищ, таких як PyCharm, VS Code має модульну структуру. Встановлення офіційного розширення Python від Microsoft та мовного сервера Pylance дозволило інтегрувати статичну перевірку типів, автодоповнення коду IntelliSense та виявлення синтаксичних помилок у режимі реального часу.

2. **Вбудовані засоби налагодження:** вбудований налагоджувач дозволив нам ефективно відстежувати стан змінних під час складних багатопотокових операцій, що було критично важливим при налаштуванні асинхронних запитів до API нейронної мережі.

3. **Інтеграція з системами контролю версій:** вбудована підтримка Git у VS Code забезпечила безпечний контроль версій проєкту, відстеження змін на кожному етапі інтеграції нових модулів, таких як графічний інтерфейс, API, Matplotlib та зберігання історії розробки без необхідності використання сторонніх терміналів.

Обраний технологічний стек Python, Tkinter, Matplotlib у поєднанні з хмарними технологіями NVIDIA NIM та сучасним середовищем розробки VS Code забезпечує оптимальну основу для розробки. Ця основа дозволяє повноцінно реалізувати визначені функціональні можливості та гарантує високу стабільність програмного забезпечення.

Архітектурні моделі та організація потоку даних

Оскільки система, що розробляється, є десктопною програмою з графічним інтерфейсом користувача, вона спирається на керовану по діям архітектуру. Виконання логіки запускається діями користувача, такими як клацання кнопок, завантаження файлів які генерують системні події та обробляються відповідними функціями контролера зворотними викликами.

Весь потік даних системи проходить через такі логічні фази:

1. **Ініціалізація:** користувач завантажує вихідний код у текстовий віджет графічного інтерфейсу користувача.
2. **Первинний аналіз:** текст передається аналітичному ядру бібліотеці Radon, яке обчислює початкові метрики, такі як цикломатична складність, MI, та зберігає їх як базове значення в пам'яті.
3. **Створення та відправлення запиту:** вихідний код, а також системні інструкції обробляються у форматі JSON та асинхронно надсилаються до NVIDIA API за допомогою багатопотокового механізму.
4. **Обробка та генерація:** модель Mistral генерує оптимізований код та повертає його у форматі JSON.
5. **Валідація та вторинний аналіз:** отриманий код аналізується, і базовий аналіз Radon викликається знову для обчислення нових метрик.
6. **Візуалізація результатів:** модуль візуалізації порівнює старий та новий код, виділяє зміни в інтерфейсі користувача, а бібліотека matplotlib генерує порівняльні гістограми, які відображаються у відповідній вкладці графічного інтерфейсу користувача.

Цей суворий та лінійний потік даних гарантує, що жодне покоління ШІ не досягне користувача без попередньої перевірки показників.

2.3. Проектування підсистеми метричного аналізу

Підсистема “Статичний аналіз” формує ядро валідації системи. ЇЇ основна функція полягає у визначенні об’єктивних кількісних показників якості вихідного коду ДО (початковий стан) та ПІСЛЯ (рефакторинг) застосування алгоритмів штучного інтелекту, щоб продемонструвати архітектурну та когнітивну ефективність виконаної оптимізації.

Об’єктно орієнтована модель підсистеми

З архітектурної точки зору, модуль реалізовано об’єктно орієнтованим способом як ізольований клас під назвою CodeMetricsAnalyzer. Цуй клас інкапсулює логіку взаємодії з методами бібліотеки Radon та приховує складність аналізу абстрактного синтаксичного дерева AST від інших компонентів системи графічного інтерфейсу користувача та API клієнта.

Цей підхід відповідає принципу єдиної відповідальності, що визначається літерою “S” у принципах SOLID. Клас несе повну відповідальність за математичні обчислення та не бере участі у візуалізації даних чи мережевих запитах.

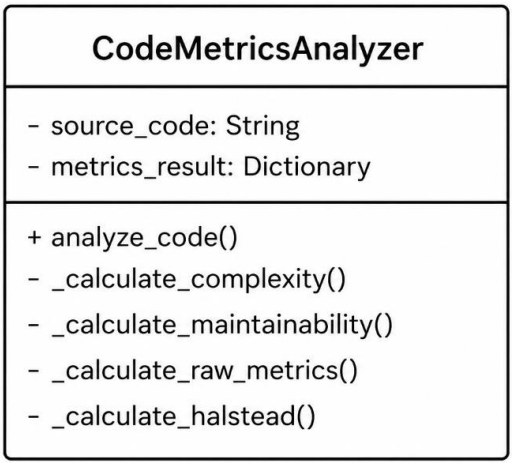


Рис. 2.3 Діаграма класів UML підсистеми метричного аналізу

Алгоритм роботи та етапи парсингу вихідного коду

Розрахунок метрик є суворо детермінованим і складається з трьох послідовних фаз обробки програми:

1. **Лексичний аналіз (токенізація):** вхідний текст розділяється на ключові слова, ідентифікатори, математичні та логічні оператори за допомогою вбудованого модуля tokenize. На цьому етапі обчислюються базові, або іншими словами сирі метрики, що вимагає безпосереднього підрахунку фізичних та логічних рядків LOC, LLOC, а також рядків, що містять коментарі CLOC.

2. **Синтаксичний аналіз:** модуль ast створює абстрактне синтаксичне дерево AST. Код перетворюється з лінійного тексту на ієрархічну структуру вузлів, що дозволяє ідентифікувати структурні блоки: визначення класів (ClassDef), функції (FunctionDef), цикли (For, While) та умовні оператори (If).

3. **Обхід дерева:** модель поведінки відвідувачів використовується для проходження вузлів AST. Під час проходження відповідні лічильники збільшуються на одиницю. Наприклад, під час відвідування вузла If або ExceptHandler значення цикломатичної складності збільшується на одиницю. [28]

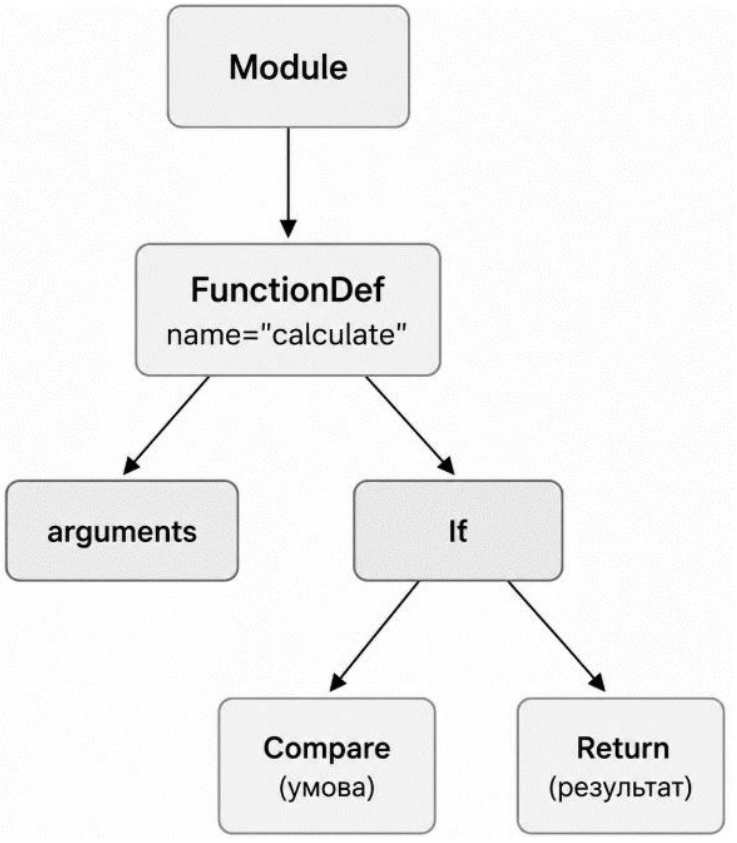


Рис. 2.4 Приклад трансформації лінійного вихідного коду в ієрархічну структуру абстрактного синтаксичного дерева AST.

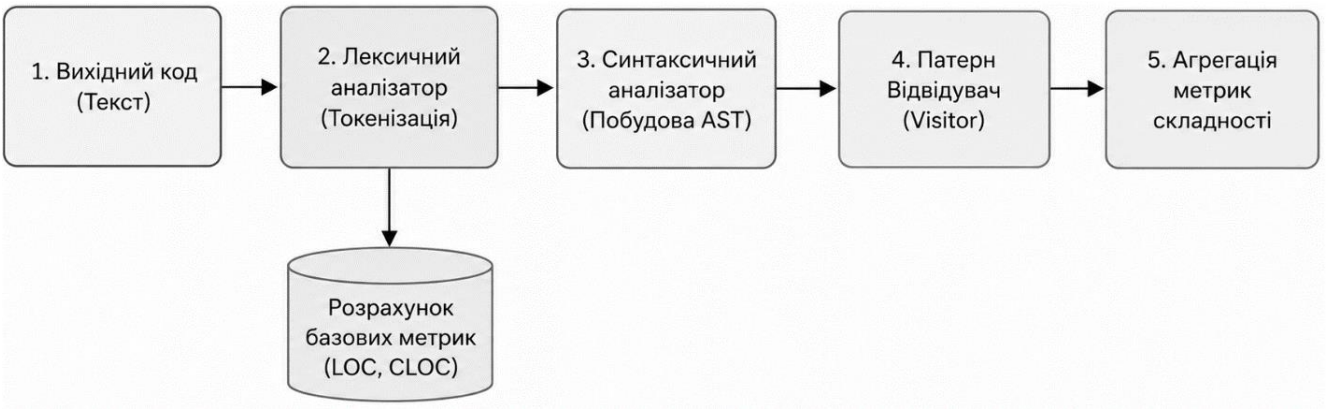


Рис. 2.5 Конвеєр обробки вихідного коду для розрахунку інженерних метрик.

Застосування методів розрахунку ключових показників ефективності в інженерії

Клас CodeMetricsAnalyzer містить внутрішні методи для обчислення кожної з цільових груп метрик, описаних у теоретичному розділі 1.2

- **Аналіз складності управління (_calculate_complexity):** викликаються методи “radon.complexity.cc_visit”. Оскільки користувач може завантажити файл, що містить десятки функцій та методів з різних класів, підсистема агрегує результати. Окрім максимальної складності, найменш ефективної функції у файлі, обчислюється середня цикломатична складність усього алгоритму.
- **Об'ємний аналіз (_calculate_halstead):** модуль “radon.metrics.h_visit” дозволяє підраховувати кількість унікальних операндів та операторів. Когнітивні зусилля Холстеда, важлива метрика, отримана з результату, відображає труднощі, з якими людина стикається в розумінні коду.

Інтегральна оцінка (_calculate_maintainability): викличе функцію “radon.metrics.mi_visit”. Результатом є нормалізований індекс підтримуваності (за шкалою від 0 до 100 балів). Підсистема також застосовує правила класифікації. Якщо значення MI < 20, системі передається індикатор, що сигналізує про критичний стан архітектури.

Структурування вихідних даних

Для забезпечення надійної взаємодії з графічним інтерфейсом користувача який малює гістограми з цих даних та модулем штучного інтелекту який інтегрує

ці числа в системне повідомлення, результати всіх приватних методів зведені в єдину структуру даних – універсальний словник.

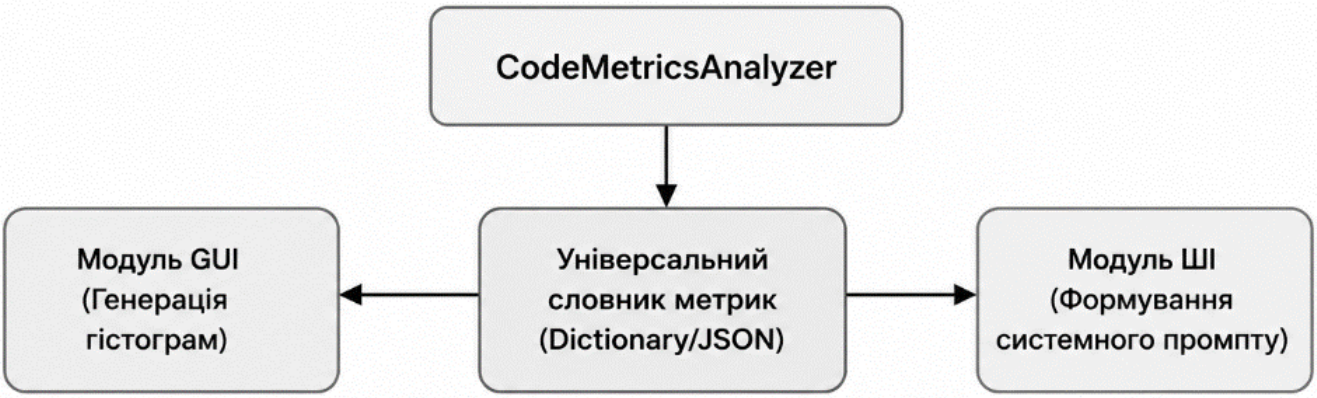


Рис. 2.6 Схема маршрутизації агрегованих метричних даних між підсистемами комплексу

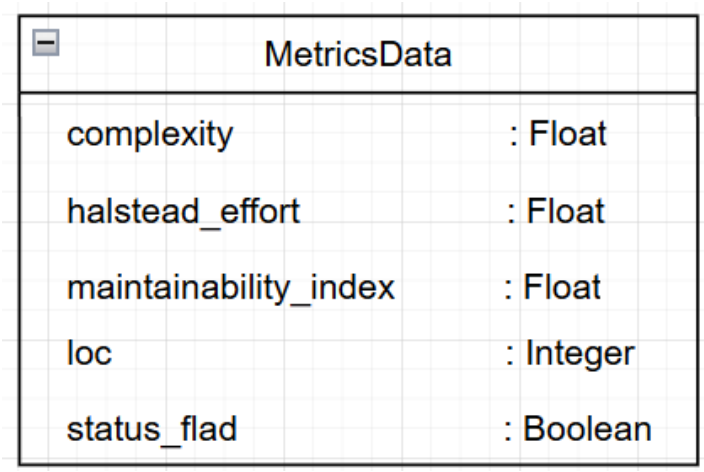


Рис. 2.7 Структурна модель універсального словника метрик вихідного коду

Таблиця 2.3 Структура вихідних даних модуля CodeMetricsAnalyzer

Ключ словника (клавіша)	**Тип даних**	**Опис індикатора**
complexity	float	Середня цикломатична складність для всіх блоків коду.
MI	float	Стандартизований індекс підтримуваності 0-100
LOC	integer	Кількість фізичних рядків коду у файлі

Ключ словника (клавіша)	Тип даних	Опис індикатора
Comments	integer	Кількість рядків, що містять коментарі або документацію
Halstead_effort	float	Оцінка когнітивних зусиль за методом Halstead
Status	string	Текстовий індикатор загального стану (наприклад, “Потрібен рефакторинг”).

Завдяки єдиній структурі даних, систему можна легко конвертувати у формат JSON. Таким чином, підсистема аналізу метрик стає повністю автономною та масштабованою. У майбутньому цей самий модуль можна буде використовувати не лише в декстопному додатку, але й як окремий мікросервіс або як розширення конвеєра CI/CD.

Забезпечити толерантність до помилок парсингу (обробка синтаксичних винятків)

Фундаментальним аспектом розробки будь-якого модуля статичного аналізу коду є обробка недійсних вхідних даних. Оскільки користувач може вставляти фрагменти коду, що містять синтаксичні помилки, такі як відкриті дужки, відсутні двокрапки або неправильні відступи, у текстовий редактор, безпосередній виклик методів бібліотеки Radon призведе до аварійного завершення роботи програми.

Щоб уникнути цього, підсистема аналізу метрик реалізує модель захисного програмування. Перед ініціалізацією обчислень модуль виконує попередню перевірку вихідного коду, намагаючись побудувати базове дерево за допомогою вбудованої функції “ast.parse()”.

Весь блок аналізу поміщено в блок try...except, який перехоплює винятки. Якщо під час аналізу виникає синтаксична помилка або помилка відступів, система негайно зупиняє обчислення. Замість блокування виконання, аналізатор генерує спеціальний словник станів, у якому числові метрики ініціалізуються значенням 0, а текст помилки разом із номером рядка, де вона сталася, записується в поле стану (див. таблицю 2.3).

Ця архітектура діє як “механізм захисту” для всієї системи. Вона запобігає надсиланню коду, що містить синтаксичні помилки, на хмарний сервер для рефакторингу, таким чином зберігаючи квоти запитів API обмеження пропускної здатності та інформуючи користувача про необхідність виправлення помилки компіляції.

2.4. Розробка інтелектуального механізму рефакторингу та взаємодія з API.

Центральним компонентом розробленої системи, що відповідає за семантичну трансформацію та оптимізацію вихідного коду, є інтелектуальна підсистема рефакторингу. На відміну від модуля розрахунку детермінованих метрик, цей рушій використовує ймовірнісні генеративні моделі штучного інтелекту. Це вимагає реалізації суворих архітектурних обмежень для керування поведінкою нейронної мережі та стандартизації протоколів обміну даними.

Архітектурна модель інтеграції сервісів у хмарі

Логіка взаємодії з моделлю LLM реалізована за допомогою моделі адаптера. Інтелектуальний механізм діє як ізольований програмний рівень між внутрішнім об'єктним представленням даних локальної програми та зовнішнім хмарним REST API платформи NVIDIA NIM.

Такий рівень абстракції забезпечує вільний зв’язок системи. Під час зміни постачальників хмарних послуг або міграції на інші генеративні архітектури моделей (наприклад, під час інтеграції моделей із сімейств LLaMA або GPT) потрібно змінити лише один внутрішній клас адаптера. Загальна логіка графічного інтерфейсу користувача та підсистеми метрик залишається незмінною.

Формат обміну даними між клієнтським застосунком та сервером базується на стандарті JSON (JavaScript Object Notation). Вхідний пакет даних, або іншими словами корисне навантаження містить токен автентифікації, або токен-носій, параметри генерації температуру, обмеження токенів та згенерований запит контексту.

Стратегія інженерії підказок

Без суворого контролю контексту, ймовірнісна мовна модель не може створити детермінований та синтаксично коректний результат. Щоб подолати цю проблему, архітектура рушія включає генератор системи підказок, створений на концепції підказок з нульовим результатом, подібний до тієї, що використовується в рольових іграх. [11]

Алгоритм генерації запитів динамічно створює текстовий пакет, що складається з п’яти обов’язкових логічних блоків, які обмежують можливий простір відповідей моделі.

Таблиця 2.4 Структурні компоненти алгоритмічного запиту на мовній моделі.

Назва блоку	Функція	Приклад базової інструкції
Обмеження ролей	Розвивайте професійні навички моделювання для реалізації моделей оптимізації високого рівня.	Працюй старшим розробником Python та досвідченим архітектором програмного забезпечення.
Математичні цілі	Включити розраховані показники ефективності, вказавши області для покращення.	Переглянь наданий код, щоб значно зменшити цикломатичну складність та покращити індекс підтримуваності
Обмеження формату	Уникати створення текстових пояснень та тегів Markdown, які перешкоджають автоматичному аналізу.	Генеруй ТІЛЬКИ коректний, компільований код Python. Без пояснень та тегів Markdown.
Стандартизація типізації	Примусово перетворюй динамічну типізацію на статичну, щоб зменшити ризик помилок під час виконання.	Завжди додавай анотації типів до всіх аргументів функції та значень, що повертаються
Локалізація документації	Автоматизувати створення технічної документації для функцій, що пов'язані виключно з українською мовою	Додати повні рядки документації до всіх методів. Ці рядки мають бути написані українською мовою.

Поєднання цих інструкцій з вихідним кодом користувача “брудний код” створює замкнутий контекст. Це перетворює загальну модель розмови LLM чат на вузькоспеціалізований інструмент для алгоритмічної трансформації.

Модель асинхронної обробки даних

Однією з визначальних характеристик взаємодії з надзвичайно великими нейронними мережами, що містять понад 100 мільярдів параметрів є висока затримка генерації відповіді. Процес виведення моделі в хмарі може тривати від 20 секунд до 2-х хвилин, залежно від обсягу переданого коду та поточного навантаження на сервер провайдера.

Використання традиційної синхронної архітектури для мережевих запитів у десктопному застосунку є серйозною архітектурною помилкою. Блокування виклику вводу/виводу призведе до зависання основного потоку інтерфейсу користувача, що зробить інтерфейс непридатним для використання та запобігатиме будь-якій взаємодії з застосунком, доки запит не буде завершено.

Для вирішення цієї проблеми була розроблена асинхронна та багатопотокова модель взаємодії типу “клієнт-черга-сервер”

1. **Ініціалізація події:** графічний інтерфейс користувача генерує подію початку рефакторингу.
2. **Розподіл завдань:** система створює ізольований фоновий потік, іншими словами робочий потік, до якого передаються згенерований JSON-пакет та параметри мережевого з'єднання.
3. **Статус очікування:** головний потік звільняється для обробки подій інтерфейсу (рендеринг анімації завантаження, обробка кліків скасування), тоді як фоновий потік очікує відповіді від сервера.
4. **Постобробка та синхронізація:** після отримання HTTP відповіді фоновий потік виконує початкове очищення даних. Видалення потенційних артефактів та залишкових тегів розпізнавання тексту та повертає власний вихідний код основного потоку через потокобезпечні черги або механізми зворотного виклику.

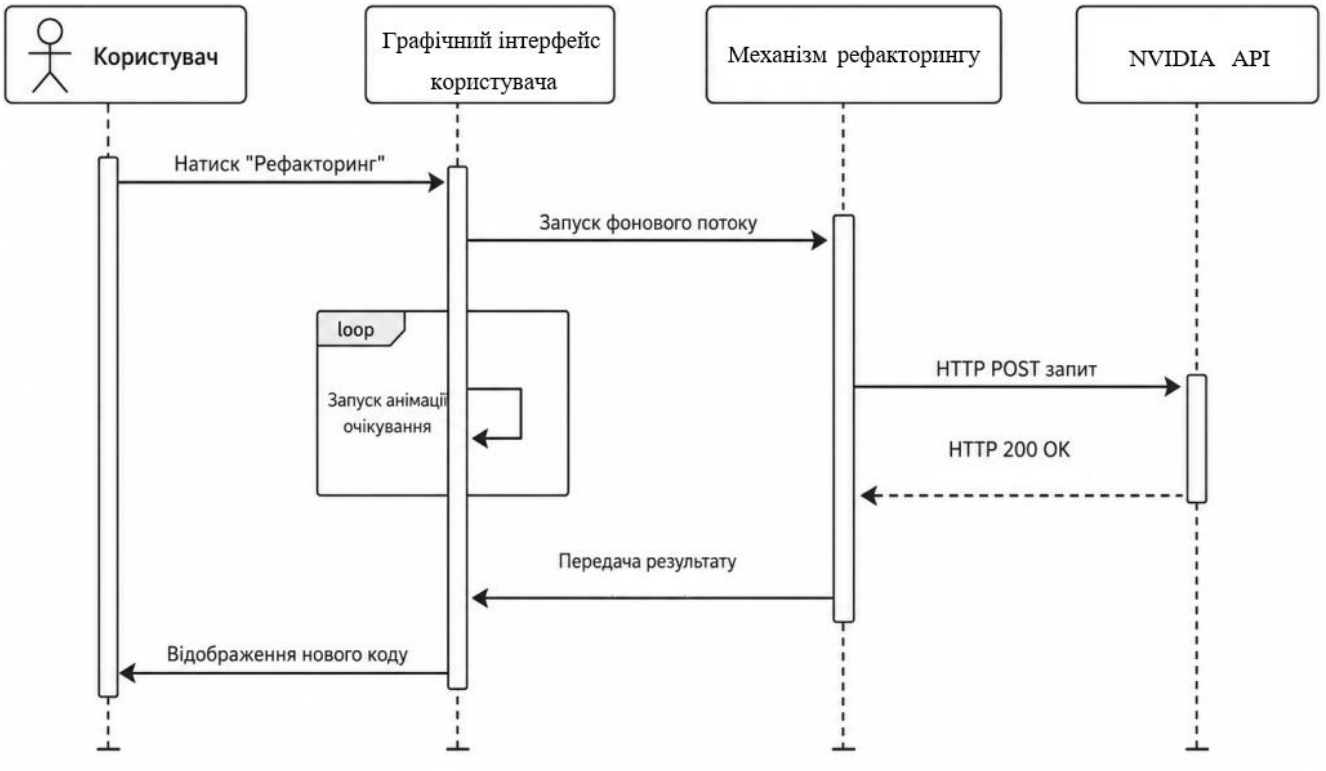


Рис. 2.8 UML діаграма послідовності асинхронної взаємодії компонентів системи під час рефакторингу

Ця модель пропонує досить чутливий користувацький інтерфейс та відповідає сучасним стандартам розробки програмного забезпечення для користувацьких операційних систем.

2.5. Проектування відмовостійкої підсистеми.

Програмні комплекси для настільних комп’ютерів, інтегровані з хмарними сервісами, або кажучи іншими словами хмарно-залежні програми, мають фундаментальний архітектурний недолік. Їхню значну залежність від зовнішньої інфраструктури. Переривання інтернет з'єднання, вичерпання квот ключів API лімітів використання або тимчасова недоступність серверів постачальника (збій у роботі сервісу) не повинні призводити до катастрофічного збою програми. Щоб гарантувати високий рівень надійності та безперервний користувацький досвід, впроваджено спеціалізовану систему відмовостійкості.

Моделі механізмів резервування

Обробка помилок мережі спирається на архітектурну модель моделювання в поєднання з резервним механізмом. Замість безпосереднього надсилання системних винятків користувачеві, архітектура використовує двоетапний конвеєр обробки коду.

Підсистема рефакторингу логічно розділена на два ізольовані рушії, що реалізують спільний інтерфейс (адаптер):

1. **Головний двигун (хмарний двигун LLM):** основний процес включає виконання глибокого семантичного рефакторингу за допомогою мережевих запитів.
2. **Вторинний двигун (локальний еквівалент двигуна):** детермінований алгоритм резервного копіювання, який спрацьовує лише у разі виявлення критичної відмови головного двигуна.

Модель кінцевого автомата

Процес перемикання двигунів реалізовано як скінченний автомат. Цей автомат відстежує стан мережевого запиту у фоновому потоці та реагує на певні тригери.

Перехід системи до Offline Mode активується за таких умов:

- **Час очікування з'єднання минув:** протягом відведеного часу відповіді від хмарного сервера не було отримано.
- **HTTP 401 заборонено:** помилка автентифікації. Користувач не ввів ключ API або ключ недійсний).
- **HTTP 429 Забагато запитів:** перевищення дозволеної частоти запитів, що надсилаються постачальнику.
- **Помилка мережі:** відсутність фізичного зв'язку між комп'ютером та Інтернетом.

Якщо трапляється один із цих тригерів, підсистема мережевої взаємодії генерує виняток, який перехоплює диспетчер рефакторингу. Потім диспетчер блокує спроби доступу до API та негайно перенаправляє вихідний код до локального механізму моделювання.

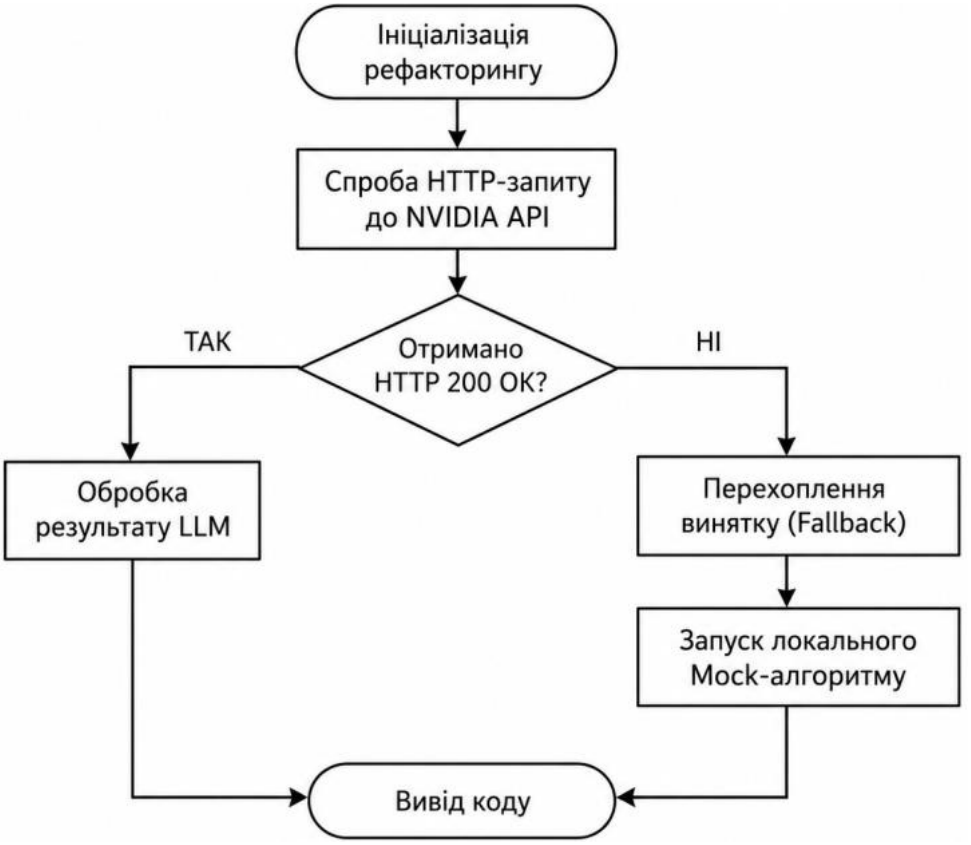


Рис. 2.9 Функціональна схема гібридного механізму перемикання станів відмовостійкості та рефакторингу.

Архітектура алгоритму локального резервного копіювання

Локальний механізм моделювання, що працює без нейронних мереж, не може виконувати складні семантичні перетворення або генерувати відповідну документацію. Його основна функція полягає в забезпеченні належного функціонування всього ланцюга обробки даних (аналіз візуальних відмінностей та перерахунок метрик radon), що гарантує повернення коректного синтаксичного дерева.

Алгоритм резервного рефакторингу виконує серію суворо детермінованих операцій над текстом:

1. **Сповіщення про стан:** автоматично додавати системний коментар до першого рядка коду (наприклад # [СИСТЕМА] Відтворено в ІМІТУОВАНОМУ АВТОНОМНОМУ РЕЖИМІ. Мережа недоступна.), який інформує користувача, чому код не зазнав жодних фундаментальних змін.

2. **Базовий синтаксичний формат:** видалити пробіли в кінці та нормалізувати порожні рядки між визначеннями функцій відповідно до основних правил PEP 8.
3. **Генерація шаблонної документації:** визначте оголошення функцій за допомогою регулярних виразів або аналізу абстрактного синтаксичного дерева та додайте порожні базові коментарі "ДОКУМЕНТАЦІЯ ОЧІКУЄТЬСЯ". Це дозволяє продемонструвати еволюцію щільності коментарів навіть без підключення до Інтернету.

Взаємодія з графічним інтерфейсом користувача

Система бронювання тісно інтегрована з підсистемою графічного інтерфейсу користувача GUI. Після завершення моделювання алгоритму програма не лише відображає код, але й генерує візуальне сповіщення. Наприклад, змінюючи колір рядка стану на жовтий або червоний, або відображаючи системне повідомлення в діалоговому вікні.

Такий підхід гарантує абсолютну відмовостійкість програмного забезпечення. Навіть у повністю ізольованому середовищі без доступу до Інтернету, програма може обробляти вихідний код, розраховувати ключові показники, генерувати диференціальні порівняння та створювати графіки без шкоди для своєї цілісності як десктопої програми.

ВИСНОВКИ РОЗДІЛУ 2

Друга частина роботи полягала в повному архітектурному проектуванні автоматизованої системи аналізу метрик та інтелектуальному рефакторингу програмного коду. Завдяки такому системному підходу було досягнуто таких ключових технічних результатів:

1. **Були складені специфікації та обраний набір технологій.** Визначено функціональні та нефункціональні вимоги до програмного забезпечення. Продемонстровано можливість гібридної архітектури (робоча станція-хмара), що

дозволяє балансувати навантаження. Локальний компонент базується на мові програмування Python та використовує бібліотеки Tkinter (графічний інтерфейс), Radon (метричний аналіз) та Matplotlib (візуалізація). Це забезпечує незалежність від платформи та швидкий час відгуку інтерфейсу користувача.

2. **Розроблено підсистему метричного аналізу.** Розроблено об'єктно-орієнтовану модель парсера, засновану на обході абстрактного синтаксичного дерева AST. Для виявлення синтаксичних помилок реалізовано захисний механізм програмування SyntaxErrors. Це запобігає збоям програми через завантаження недійсного коду та забезпечує базову точність.

3. **Розроблено архітектуру інтелектуального механізму рефакторингу.** Для взаємодії з мовною моделлю в хмарі було розроблено генератор системних запитів на основі інструкційної інженерії. Завдяки суворим контекстним обмеженням (рольова модель, математичні цілі, примусова типізація) ймовірнісна генерація мовної моделі може бути перетворена на контрольований процес оптимізації коду з автоматичною генерацією документації українською мовою.

4. **Проблему відмовостійкості та блокування інтерфейсу вирішено.** Для зменшення високої затримки хмарних мереж була розроблена асинхронна багатопотокова модель обміну даними. Для забезпечення автономності була реалізована відмовостійка підсистема. Кінцевий автомат виявляє помилки мережі, такі як тайм-аути, HTTP-винятки, та автоматично перемикає систему на імітований локальний механізм резервного копіювання для забезпечення безперервності робочого процесу.

Розроблені архітектурні рішення, алгоритми обробки даних та механізми взаємодії забезпечують повну та надійну технічну основу. Це знаменує собою завершення етапу проектування програмного забезпечення, яке буде розглянуло в наступному розділі.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСОБУ

3.1. Розробка сучасного графічного інтерфейсу та інтуїтивно зрозумілого користувацького досвіду.

Процес розробки системного програмного забезпечення розпочався зі створення графічного інтерфейсу користувача або іншими словами “GUI”. Згідно з вимогами, визначеними у другому розділі, інтерфейс повинен бути інтуїтивно зрозумілим, підтримувати темну тему та забезпечувати надійну ізоляцію робочих процесів, таких як введення коду, відображення індикаторів, відстеження змін.

Для реалізації графічного інтерфейсу було обрано стандартну бібліотеку *tkinter* у поєднанні з модулем *tkinter.ttk*. Цей вибір обґрунтовано відсутністю необхідності встановлення великих за обсягом пам’яті сторонніх залежностей (на відміну від фреймворків PyQt або Kivy) та швидкістю ініціалізації вікна, що є важливим критерієм для легких десктопних утиліт. Також це забезпечило кросплатформну сумісність та мінімальне споживання оперативної пам'яті програмою.

Архітектурне компонування робочого простору

Архітектура головного вікна, що називається “Система підвищення якості програмного забезпечення”, базується на принципі поділу екрана на дві панелі. Такий підхід імітує поведінку сучасних інтегрованих середовищ розробки.

Просторова структура реалізована за допомогою менеджерів геометрії *grid* та *pack* і складається з чотирьох основних областей:

1. **Ліва робоча панель (“Введіть Python код для аналізу”):** ця частина розроблена для вставки та редагування вихідного коду. Містить власний багаторядковий текстовий віджет з інтегрованою нумерацією рядків, яка синхронізована з прокруткою колеса миші.
2. **Панель аналізу праворуч (“Панель результатів”):** ця частина програми займає праву половину екрану та служить контейнером для відображення результатів обчислень та генерації штучного інтелекту.

3. **Нижня панель керування:** у нижній частині екрану панель об’єднує основні елементи взаємодії, такі як кнопка операції з файлами “Відкрити”, налаштування та ініціалізація ключових процесів “Аналізувати код”, Виконати рефакторинг” , також кнопка очистки всього “Очистити” та кнопка параметрів “Налаштування” .
4. **Status Bar:** під панеллю керування рядок стану, відображає системні повідомлення (наприклад, “Готовий до роботи”). [23]

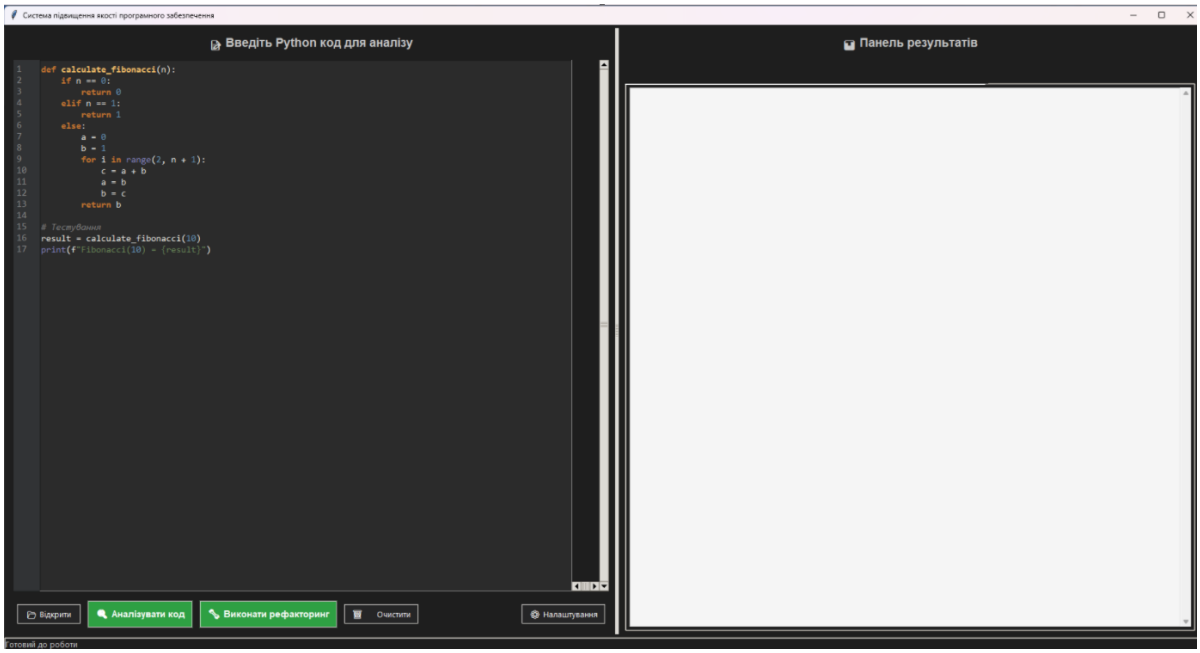


Рис. 3.1 Огляд головного вікна програмного забезпечення

```
    main_container = ttk.PanedWindow(self.root, orient=tk.HORIZONTAL)
    main_container.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)


    left_frame = ttk.Frame(main_container)
    main_container.add(left_frame, weight=1)


    ttk.Label(left_frame,
              text="📝 Введіть Python код для аналізу",
              style='Header.TLabel').pack(pady=10)


    self.code_input = CodeEditor(left_frame)
    self.code_input.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)


    default_code = '''def calculate_fibonacci(n):
if n == 0:
    return 0
elif n == 1:
    return 1
else:
    a = 0
    b = 1
    for i in range(2, n + 1):
        c = a + b
```

Рис. 3.2 Ініціалізація основних контейнерів головного вікна та менеджерів компонування.

Реалізація власної темної теми

Враховуючи специфіку роботи цільової аудиторії цієї програми, а саме програмістів, базовий та зрозумілий інтерфейс tkinter був повністю перепрограмований для реалізації темної теми.

Об’єкт ttk.Style() дозволяє глобально змінити колірну палітру застосунку:

- Як базовий фон для вікон та панелей використовується глибокий темно-сірий колір (подібний до #1E1E1E), що зменшує навантаження на очі.
- Текст елементів інтерфейсу переведено у світлий колір, який має добрий контраст з рештою. [23]

Особливу увагу було приділено ергономіці текстового редактора на лівій панелі. Віджет tk.Text використовує шрифт фіксованої ширини (наприклад, Consolas), що є важливою вимогою для підтримки відступів синтаксису Python. Крім того, було реалізовано механізм підсвічування синтаксису. Завдяки системі розмітки Tkinter ключові слова (def, if, return), рядки та числа динамічно підсвічуються, як і в професійних інтегрованих середовищах розробки.

```
self.text.tag_configure("keyword", foreground="#cc7832", font=('Consolas', 11, 'bold'))
self.text.tag_configure("builtin", foreground="#8888c6")
self.text.tag_configure("string", foreground="#6a8759")
self.text.tag_configure("comment", foreground="#808080", font=('Consolas', 11, 'italic'))
self.text.tag_configure("number", foreground="#6897bb")
self.text.tag_configure("defclass", foreground="#ffc66d", font=('Consolas', 11, 'bold'))
```

Рис. 3.3 Програмне визначення стилів віджетів та конфігурації кольорової палітри.

Увага користувача керується за допомогою системи акцентних кольорів. Основні кнопки дій “Аналізувати код”, “Виконати рефакторинг”) програмно відображаються яскраво-зеленим кольором, тоді як вторинні кнопки дій “Відкрити”, “Налаштування” зберігають нейтральний сірий колір. Таке рішення мінімізує когнітивне навантаження під час використання програми.

Реалізований графічний інтерфейс створив надійне та практичне середовище, яке дозволило нам просунутись в інтеграції математичних інструментів, розробивши модуль статичного аналізу коду.

Організація багатосторінкового простору та вирішення інженерних викликів

Для полегшення навігації між різними етапами аналізу відповідна панель аналізу реалізована за допомогою віджета ttk.Notebook. Програмно створено та налаштовано чотири окремі вкладки:

- **Метрики:** призначена для отримання кількісних показників якості коду, отриманих з підсистеми radon.
- **Рефакторований код:** текстовий контейнер у режимі “лише для читання”, де результат моделі мови Mistral завантажується асинхронно.
- **Порівняння:** вкладка з таблицею порівняння по 6 метрикам вихідного коду та після рефакторингу з діаграмою порівняння трьох показників.
- **Візуальний Diff:** вкладка, присвячена розширеній візуалізації архітектурних змін, яка використовує алгоритми для виділення змінених блоків. [24]

Під час впровадження цього компонента виникла технічна проблема: неможливість безпосередньо синхронізувати стан кнопки з поточним процесом. Якщо користувач натискав кнопку “Виконати рефакторинг”, система надсилала запит до LLM, але інтерфейс залишався активним. Це створювало ризик додаткових натискань кнопок та дублювання HTTP-запитів.

Для вирішення цієї проблеми було розроблено механізм керування станом. Було реалізовано функцію відправлення “toggle_ui_state()”, яка деактивує всі активні віджети, а саме кнопки та текстові поля безпосередньо перед запуском фонового потоку. Після позитивної відповіді від хмарного API або активації Mock-заглушки функція відновлює елементи до їхнього нормального стану. [24]

```
def start_refactoring(self):

    self.toggle_ui_state("disabled")


    self.status_label.config(text="Очікування відповіді від сервера Mistral...")


    import threading
    threading.Thread(target=self.run_llm_request, daemon=True).start()
```

Рис. 3.4 Блокування команд під час очікування відповіді сервера.

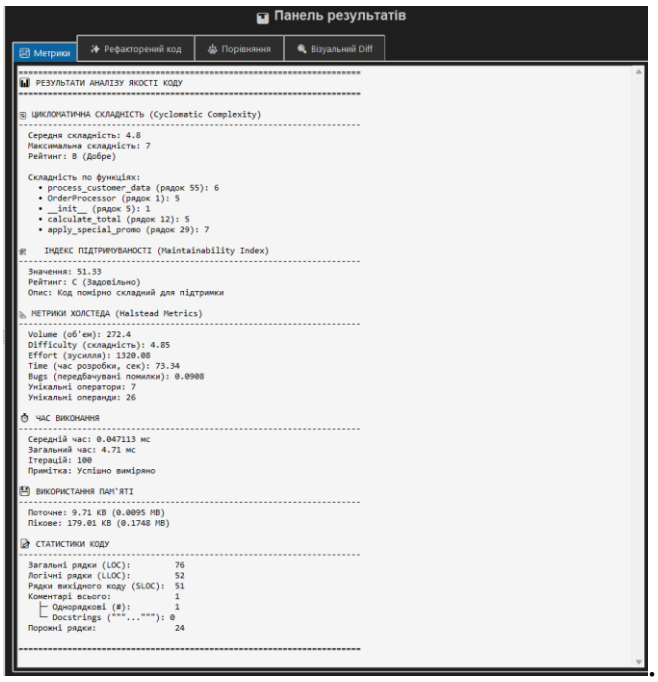


Рис. 3.5 Вигляд вікна панелі результатів

Реалізація описаних механізмів дозволила створити надійний, інтуїтивно зрозумілий та безпечний інтерфейс, який ізолює користувача від внутрішньої складності обчислень та мережевих взаємодій. Це проклало шлях для інтеграції аналітичного ядра, розробка якого описана в наступному розділі.

3.2. Розробка модуля статичного аналізу та розрахунок метрик.

Розроблена система базується на підсистемі статичного аналізу, яка дозволяє детерміновано оцінювати якість вихідного коду. Відповідно до архітектурних рішень, прийнятих на етапі проектування, цей модуль реалізовано як незалежний клас “CodeMetricsAnalyzer”. Такий підхід забезпечує незалежність математичних розрахунків від логіки графічного інтерфейсу та дозволяє автоматизувати тестування модулів.

Основним інструментом, який використовувався для вилучення інженерних показників з тексту програми, була спеціалізована бібліотека радону.

Ініціалізація програми та лексико-синтаксичний аналіз

Процес аналізу починається з передачі рядка символів з інтерфейсу введення тексту до екземпляра кламу CodeMetricsAnalyzer. Бібліотека radon не вимагає

попереднього збереження коду на диск, що значно пришвидшує обробку. Дані аналізуються безпосередньо в оперативній пам'яті.

Для обчислення цикломатичної складності використовується метод “cc_visit” з модуля “radon.complexity”. Цей метод будує абстрактне синтаксичне дерево AST та повертає список об'єктів. Кожен об’єкт у цьому списку представляє окремий блок структурного коду - функцію, клас або метод, та містить свою локальну складність. [22]

Для отримання об’ємних метрик Холстеда та індексу підтримуваності використовуються методи h_visit та mi_visit модуля radon.metrics.

Алгоритм додавання метрик та розпакування структур даних

Оскільки метод cc_visit повертає список складностей для кожного окремого блоку, а системі потрібен єдиний індикатор для всього потокового фрагмента коду для прийняття рішення, на етапі розробки було реалізовано алгоритм агрегації.

Цикл “for” перебирає всі витягнуті блоки, підсумовує їх складність та обчислює середнє значення, або іншими словами середню цикломатичну складність. Максимальне значення “найскладніша функція у файлі” також записується для виявлення вузьких місць в архітектурі.

Модуль radon.metrics.h_visit має унікальну характеристику повернення результату у вигляді іменованого кортежу. Щоб інтегрувати ці дані у спільний словник, код виконує спеціальне розпакування цього кортежу, зберігаючи лише метрики важливі для користувача. Це метрики обсягу програми, складності та когнітивні зусилля.

```python
def analyze_all(self) -> Dict[str, Any]:
    """
    Виконання повного аналізу коду.
    """
    is_valid, message = self.validate_syntax()

    if not is_valid:
        return {
            'valid': False,
            'error': message
        }

    return {
        'valid': True,
        'cyclomatic_complexity': self.calculate_cyclomatic_complexity(),
        'maintainability_index': self.calculate_maintainability_index(),
        'halstead': self.calculate_halstead_metrics(),
        'execution_time': self.measure_execution_time(),
        'memory_usage': self.measure_memory_usage(),
        'code_stats': self.calculate_code_statistics()
    }
```

Рис.3.6 Метод повного аналізу метрик

Після завершення всіх обчислень метод створює єдиний словник Python, уніфіковану структуру даних, який повертається до головного потоку програми для подальшого відображення на екрані. [22]

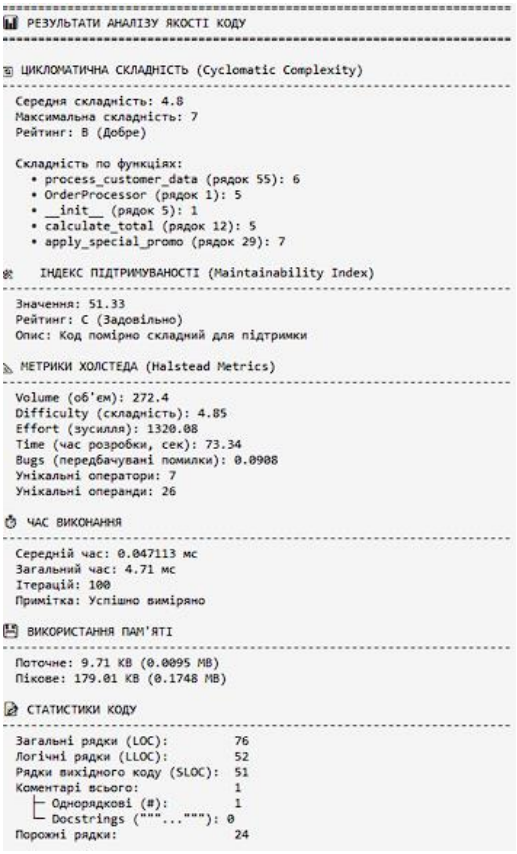


Рис. 3.7 Візуалізація результатів розрахунку метрик вихідного коду в інтерфейсі користувача.

Реалізація механізму захисту від помилок

Під час практичної розробки було виявлено критичну вразливість, а саме якщо користувач вводить у текст ове поле неповний код або код із синтаксичними помилками (наприклад, відсутність двокрапки в циклі “for”), парсер Radon не може побудувати абстрактне синтаксичне дерево AST. Це викликає виняток “SyntaxError” в інтерпретаторі Python, що призводить до неочікуваного збою програми.

Для забезпечення високої надійності програмної системи в метод аналізу було інтегровано модель захисного програмування. Весь блок викликів методу Radon інкапсульовано в блок обробки помилок try...except.

Код обробляє два основних типи винятків. До цих винятків відноситься синтаксичні помилки та загальні винятки. У разі виникнення винятку система автоматично перериває обчислення та повертає попередньо визначений словник. У цьому словнику всі числові значення ініціалізуються рівними 0, а текст синтаксичної помилки записується у відповідне поле стану або помилки разом із зазначенням проблемного рядка коду. [24]

```python
def validate_syntax(self) -> Tuple[bool, str]:
    """
    Перевірка синтаксичної коректності коду.
    """
    try:
        ast.parse(self.code)
        return True, "Синтаксис коректний"
    except SyntaxError as e:
        return False, f"Помилка синтаксису: {str(e)}"
    except Exception as e:
        return False, f"Помилка: {str(e)}"
```

Рис. 3.8 Реалізація блоку захисту для обробки винятків під час розбору абстрактного синтаксичного дерева.

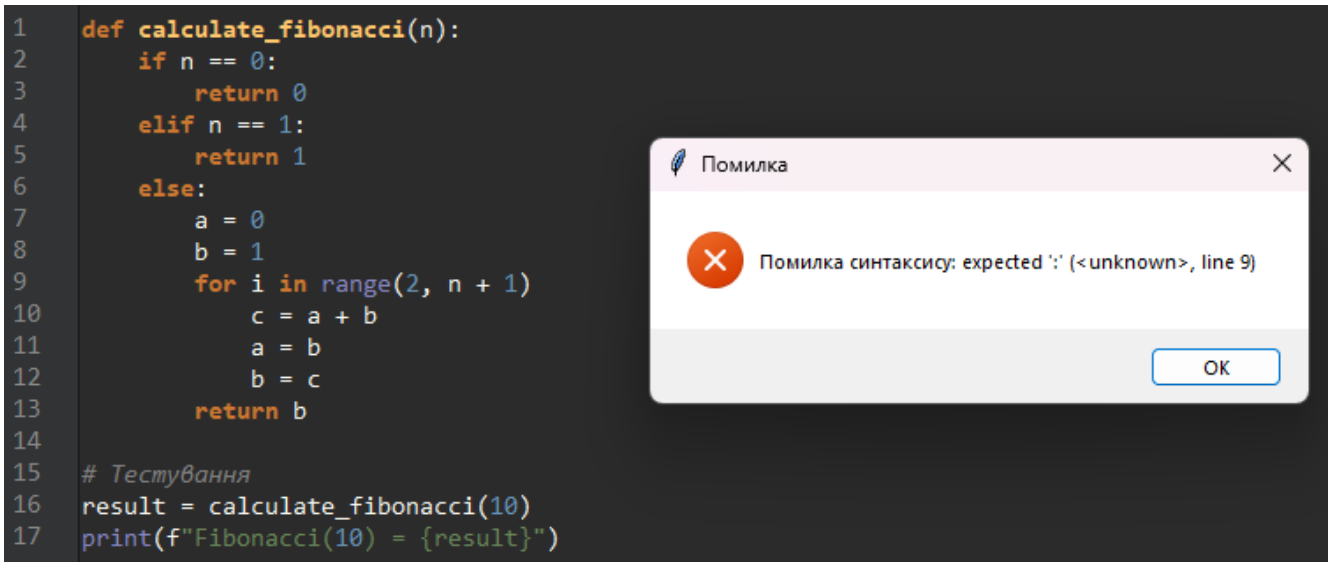


Рис. 3.9 Реакція графічного інтерфейсу на синтаксично некоректне введення вихідного коду.

Впровадження цієї підсистеми дозволило нас створити стабільну, детерміновану основу. Тепер програма може точно оцінювати будь-який вхідний текст і алгоритмічно вирішувати, чи надсилати його до великої мовної моделі.

3.3. Інтеграція хмарного штучного інтелекту та забезпечення асинхронної обробки даних.

Впровадження програмного забезпечення підсистеми інтелектуального рефакторингу вимагає встановлення безпечного мережевого каналу передачі даних, керування потоками виконання та алгоритмічного очищення результатів. Основна логіка цього процесу інкапсульована в класі CodeRefactoringEngine.

Формування мережевих запитів та контекстна інженерія

Взаємодія з хмарним API NVIDIA NIM реалізована за допомогою стандартної бібліотеки запитів [20] . Для автентифікації на сервері програма зчитує ключ API, введений користувачем у графічному інтерфейсі, та передає його в заголовках HTTP запиту за допомогою протоколу Bearer Token.

```
# Налаштування запиту до NVIDIA API
invoke_url = "https://integrate.api.nvidia.com/v1/chat/completions"
headers = {
    "Authorization": f"Bearer {self.api_key}",
    "Accept": "application/json"
}

payload = {
    "model": "mistralai/mistral-medium-3.5-128b",
    "messages": [
        {"role": "system", "content": "Ти досвідчений Senior Python Developer. Твоя мета - писати ідеальний, чистий та ефективний код."},
        {"role": "user", "content": prompt}
    ],
    "max_tokens": 4096,
    "temperature": 0.2,
    "top_p": 1.00,
    "stream": False
}

# Виклик API
response = requests.post(invoke_url, headers=headers, json=payload)
```

Рис. 3.10 Налаштування запиту до NVIDIA API

Тіло запиту має формат JSON. Згідно з архітектурою, описаною в розділі 2, запит містить системне повідомлення з точними інструкціями щодо формату PEP 8, введення та створення документації українською мовою, а також користувацьке повідомлення, що містить вихідний код та попередньо розраховані базові метрики цикломатичної складності та індексу підтримуваності.

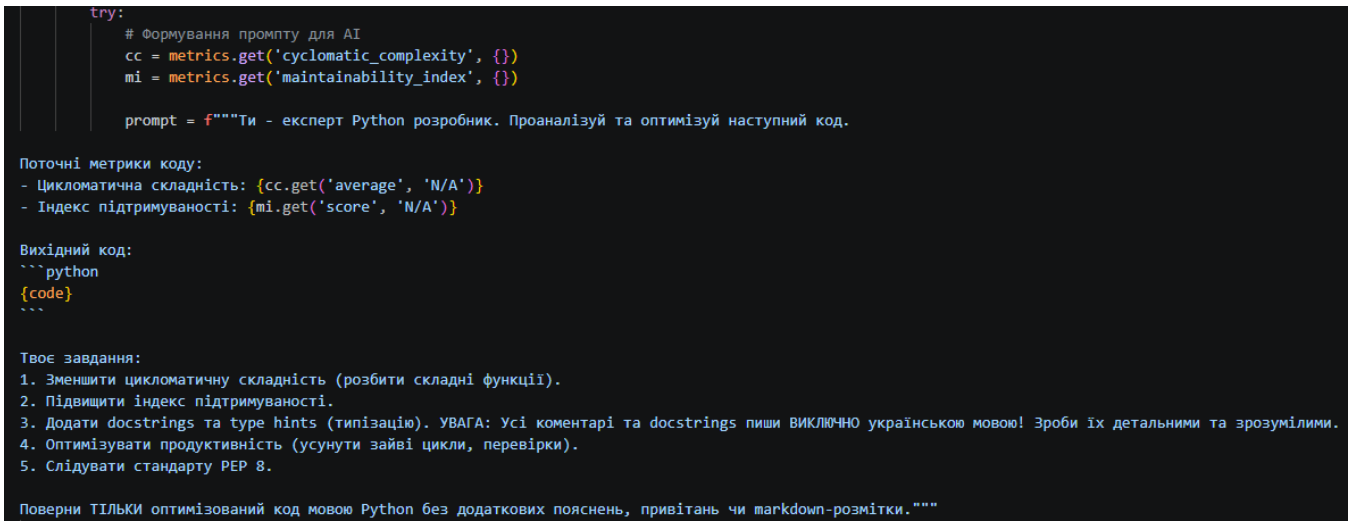


Рис. 3.11 Формування промпту для Mistral

Налаштування параметрів виводу моделі Mistral передбачає, зокрема, обмеження параметра температури значенням 0,2. Це мінімізує ймовірнісну “креативність” нейронної мережі, змушуючи її генерувати більш детермінований, передбачуваний та синтаксично коректний код.

Асинхронне виконання через модуль потоків

Виклик методу *requests.post()* є синхронною операцією вводу/виводу. Якщо його виконати в головному потоці, то цикл подій tkinter (mainloop) повністю блокується під час очікування відповіді від сервера від 20 до 180 секунд.

Щоб вирішити цю проблему, було реалізовано багатопотокову архітектуру з використанням вбудованої бібліотеки потоків. Після натискання кнопки “Виконати рефакторинг” система почне створювати новий ізольований потік “threading.Thread(target=self._run_api_request, args=(...)).”

Фонова функція надсилає HTTP запит. Після отримання та розбору відповіді вона використовує функцію зворотного виклику, щоб повернути оброблений рядок коду до основного потоку. Для забезпечення безпеки потоків стан віджетів графічного інтерфейсу, а саме вставка тексту у вкладці “Рефакторинг” змінюється лише тоді, коли керування повертається до основного циклу tkinter.

```python
def _process_refactoring_thread(self):
    """Фоновий процес рефакторингу."""
    try:
        code = self.code_input.get('1.0', tk.END).strip()

        refactorer = CodeRefactoringEngine(api_key=self.api_key)
        header_report, clean_code = refactorer.refactor(code, self.original_metrics)
        self.root.after(0, self._update_gui_after_refactor, header_report, clean_code)

    except Exception as e:
        self.root.after(0, lambda: messagebox.showerror("Помилка", f"Помилка при рефакторингу: {str(e)}"))
```

Рис. 3.12 Реалізація програмного забезпечення для запуску фонового потоку та передачі керування за допомогою функції зворотного виклику.

Обробка результатів після генерації

Навіть незважаючи на явну заборону, що відображається в командному рядку, шаблон мови може повертати згенерований код, інкапсульований у блоки Markdown (наприклад, ‘’’python ... ‘’’). Вставко цього тексту безпосередньо в інтерпритатор призведе до синтаксичної помилки.

Модуль регулярних виразів “re” використовується для алгоритмічного очищення тексту. Метод постобробки використовує шаблон пошуку, який ідентифікує межі коду в межах тегів Markdown та витягує лише чистий синтаксис Python. Очищений рядок потім передається до графічного інтерфейсу та відображається у відповідній вкладці. [23]

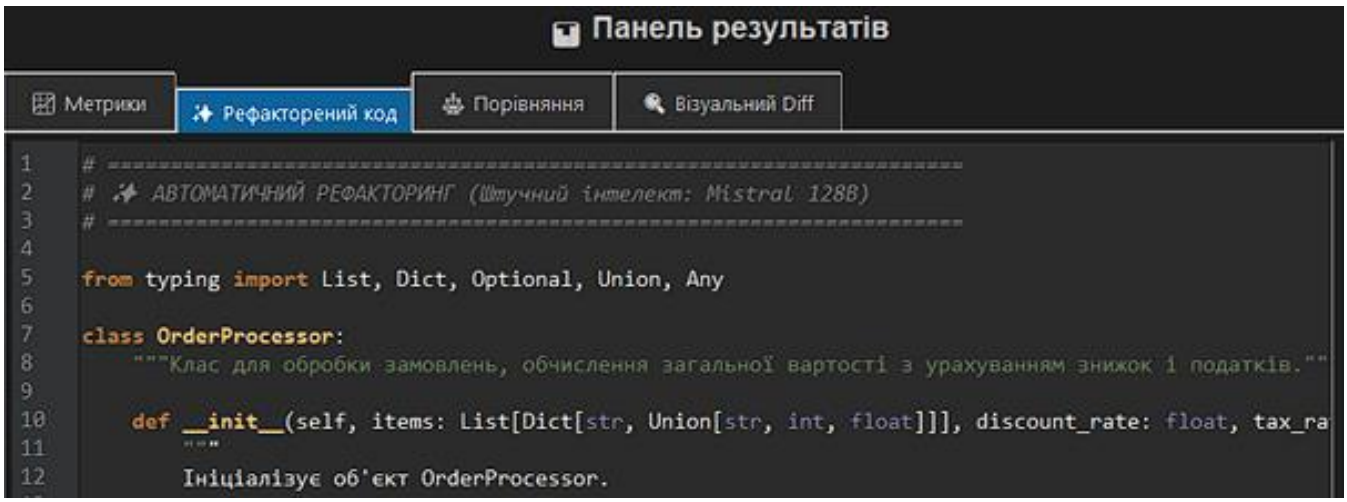


Рис. 3.13 Результат механізму хмарного рефакторингу з додаванням типізації та документації українською мовою.

Програмна реалізація механізму відмовостійкості та алгоритму автономного рефакторингу

Критичною вимогою до сучасного інструментарію розробника є здатність функціонувати в умовах нестабільної мережевої інфраструктури. Для забезпечення абсолютної автономності програмного комплексу, в архітектуру класу “CodeRefactoringEngine” створено гібридний механізм обробки, який поєднує хмарні обчислення та детерміновані правила локальної оптимізації.

Механізм перехоплення мережевих винятків

Процес перемикання між режимами керується дворівневою логікою. Точкою входу є метод *refactor()*, який перевіряє поточний стан системи, такий як наявність API ключа та згоду на використання ШІ. У разі активації хмарного режиму управління передається методу *ai_refactoring*().

```
def refactor(self, code: str, metrics: Dict[str, Any] = None) -> Tuple[str, str]:
    """
    Головний метод рефакторингу.

    Args:
        code (str): Код для рефакторингу
        metrics (Dict): Метрики коду

    Returns:
        str: Рефакторений код
    """
    if self.use_ai and metrics:
        return self.ai_refactoring(code, metrics)
    else:
        return self.mock_refactoring(code)
```

Рис. 3.14 Метод refactor для перевірки

Для запобігання аварійному завершенню програми при відсутності інтернету або вичерпанні лімітів API, весь блок мережевої взаємодії з платформою NVIDIA NIM жорстко інкапсульовано в конструкцію *try...except Exception as e*. Метод “response.raise_for_status()” гарантує генерування винятку навіть у випадку, коли сервер повертає код помилки авторизації 401 або тайм-ауту.

У разі спрацьовування винятку, система не перериває робочий процес, а виконує автоматичне падіння до резервного стану - ініціює виклик методу self.mock_refactoring(code).

```
except Exception as e:
    print(f"Помилка AI рефакторингу: {e}")
    # Якщо API не відповідає
    return self.mock_refactoring(code)
```

Рис. 3.15 Програмна реалізація перехоплення мережевих винятків та автоматичного перемикання на резервний локальний рушій

Алгоритм альтернативного рефакторингу на основі правил

Резервний локальний рушій mock_refactoring базується на суворо детермінованих правилах. Оскільки він не використовує імовірнісних генеративних моделей, його головна мета виконати базове синтаксичне очищення та привести код до стандартів PEP 8 без ризику пошкодження бізнес-логіки.

Розроблений алгоритм виконує п'ять послідовних етапів трансформації вихідного коду:

1. **Нормалізація вертикального простору:** за допомогою модуля регулярних виразів “re” виконується пошук і заміна надлишкових порожніх рядків *re.sub(r'\n{3,}', '\n\n', refactored)*.

2. **Лексичне форматування PEP 8:** алгоритм реалізує скінченний автомат для додавання пропуску після ком. Механізм посимвольно обходить текст, зберігаючи стан перебування всередині рядкових літералів, щоб уникнути модифікації тексту всередині лапок, та автоматично виправляє форматування аргументів функцій.

3. **Оптимізація логічних виразів:** Застосовується набір безпечних регулярних замін для спрощення умовних конструкцій. Наприклад, надлишкові перевірки if x == True трансформуються у Python стиль if x, а if x == None замінюється на ідентично безпечне if x is None.

4. **Алгоритмічне документування:** програма проходить по рядках і виявляє ключові слова оголошення def та class. Якщо наступний непустий рядок не є початком документації, алгоритм автоматично вираховує рівень відступу та вставляє шаблон ””TODO: описати...””””. Це безпосередньо впливає на підвищення метрики щільності коментарів.

5. **Статичний аналіз магічних чисел:** завершальним етапом локального рушія є побудова абстрактного синтаксичного дерева AST для пошуку літералів типу ast.Constant. Якщо в коді виявляються нетривіальні числові константи (крім 0, 1, 2, -1), система не змінює їх фізично, щоб уникнути логічних помилок, але фіксує їх і додає попередження до фінального звіту.

```
def mock_refactoring(self, code: str) -> str:
    """
    Рефакторинг на основі правил (без AI).

    Правила що застосовуються:
      1.  Видалення зайвих порожніх рядків (>2 підряд → 1)
      2.  PEP 8: пробіл після коми (лише поза рядковими літералами)
      3.  if x == True  →  if x
      4.  if x == False →  if not x
      5.  if x == None  →  if x is None
      6.  if x != None  →  if x is not None
      7.  range(0, n)   →  range(n)
      8.  return (x)    →  return x   (зайві дужки навколо простої змінної)
      9.  Автододавання TODO-docstring до функцій/класів без нього
      10. Виявлення «магічних чисел» через AST (без зміни рядкових літералів)

    Args:
        code (str): Вихідний код
    Returns:
        str: Рефакторений код з заголовком про зміни
    """
    refactored = code
    changes = []
```

Рис. 3.16 Програмна імплементація правил оптимізації логічних виразів за допомогою модуля re

Після застосування всіх правил, метод формує структурований текстовий заголовок (# АВТОМАТИЧНИЙ РЕФАКТОРИНГ (Mock Engine)), у якому перелічує всі успішно застосовані перетворення. Цей звіт об’єднується з оптимізованим кодом і повертається у графічний інтерфейс користувача.

```
1  # ──────────────────────────────────────────────────
2  # 🔧 АВТОМАТИЧНИЙ РЕФАКТОРИНГ (Mock Engine)
3  # ──────────────────────────────────────────────────
4  # Застосовані перетворення:
5  #   • Видалено зайві порожні рядки
6  #   • PEP 8: додано пробіли після 14 ком
7  #   • if x==True → if x
8  #   • if x==False → if not x
9  #   • if x==None → if x is None
10 #   • if x!=None → if x is not None
11 #   • range(0,n) → range(n)
12 #   • return(x) → return x
13 #   • Додано 5 заглушку(-и) docstring
14 #   • ⚠ Знайдено магічні числа (0.05, 0.1, 5, 18, 25, 65, 1200) — розгляньте заміну на константи
```

Рис 3.17 Результат роботи алгоритму локальної оптимізації із виведенням звіту про застосовані перетворення

Створення такого локального модуля гарантує, що система залишається корисним інженерним інструментом для покращення якості коду навіть у повністю ізольованому від мережі середовищі.

3.4. Розробка підсистеми візуалізації аналітичних даних

Заключний етап обробки даних у розробленій системі полягає у візуальному представленні результатів користувачеві. Оскільки головною метою програми є не лише автоматизована оптимізація, а й демонстрація її ефективності, підсистема візуалізації була розроблена з акцентом на двох аспектах. Перше, порівняння структурного синтаксису, та друге, математичне порівняння інженерних показників.

Алгоритмічне порівняння текстів та кольорових позначок

Щоб розробники могли швидко оцінювати зміни, внесені до вихідного коду інтелектуальним механізмом або алгоритмом локального моделювання, було реалізовано модуль візуальної диференціації тексту. Цей модуль використовує вбудовану бібліотеку difflib, яка реалізує алгоритм зіставлення послідовностей Ратклiффа-Обершельпа. Теоретична складність цього алгоритму, у найгіршому випадку, становить $O(N^2)$, де N це кількість символів. Однак для файлів вихідного коду він пропонує високу продуктивність.

Процес порівняння починається з виклику методу “difflib.ndiff()”, який приймає на вхід два списки рядків, таких як оригінальний код та після рефакторингу. Метод повертає генератор, де кожен рядок починається зі спеціального префікса:

- -(мінус і пробіл) рядок був присутній в оригіналі, але був видалений або змінений;
- +(плюс та пробіл) новий рядок, доданий рушієм штучного інтелекту;
- (два пробіли) рядок залишається незмінним;

Візуалізація цих даних здійснюється за допомогою системи тегів віджета tk.Text. Під час ініціалізації вкладки Diff створюються дві конфігурації тегів:

“text.tag_config(“diff_remove”, foreground=”red”)” та “text.tag_config(“diff_add”, foreground=”green”)”. Під час обробки результатів “ndiff()” програма аналізує префікс кожного рядка та динамічно застосовує відповідний тег, коли текст вставляється у віджет.

```
for line in d:
    if line.startswith('- '):
        self.diff_output.insert(tk.END, line, "diff_remove")
    elif line.startswith('+ '):
        self.diff_output.insert(tk.END, line, "diff_add")
    elif line.startswith('? '):
        pass
    else:
        self.diff_output.insert(tk.END, line)
```

```
# Червоний фон та закреслений текст для видалених рядків
self.diff_output.text.tag_configure("diff_remove", background="#3d1d1d", foreground="#ff6b6b", overstrike=1)
# Зелений фон для нових рядків
self.diff_output.text.tag_configure("diff_add", background="#1e3422", foreground="#8ce99a")
```

Рис. 3.18 Реалізація генерації різниць та колірного маркування Tkinter

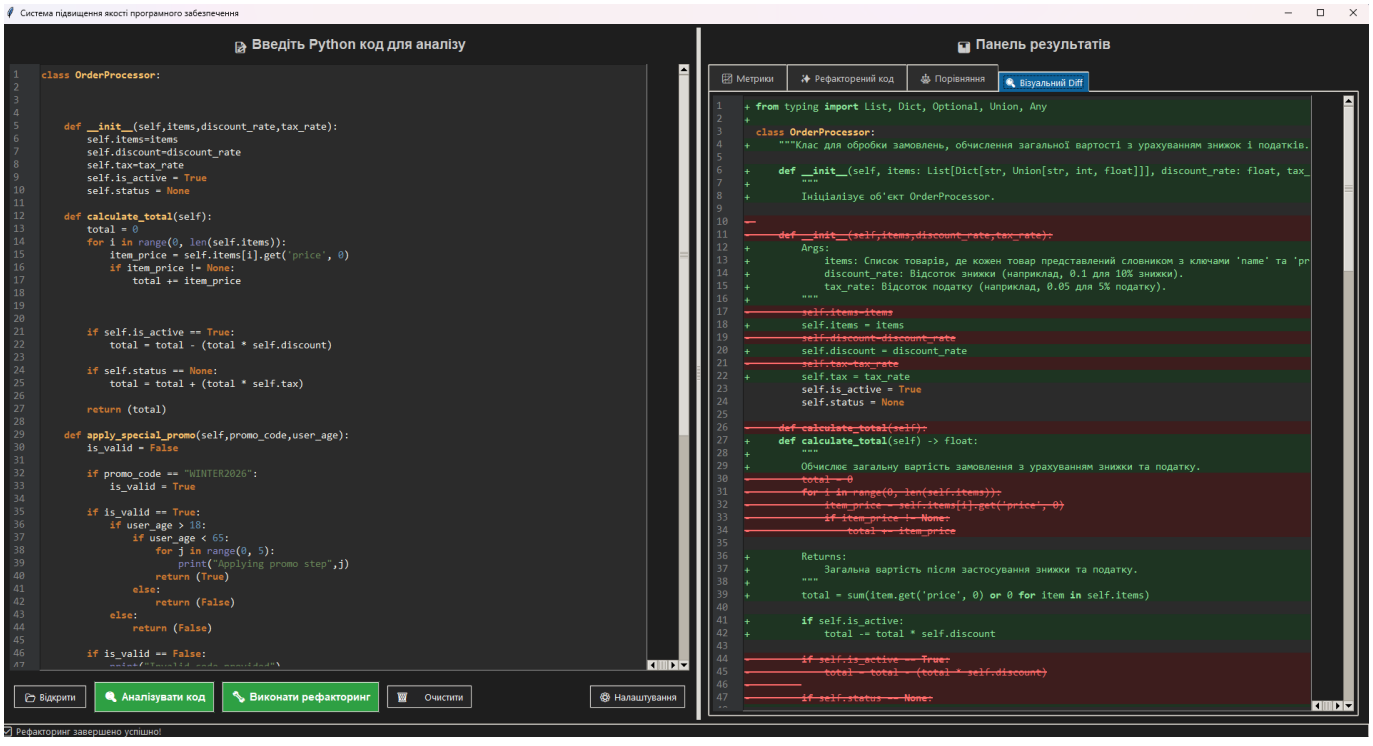


Рис. 3.19 Візуалізація змін структурного коду на вкладці “Візуальний Diff”

Генерація та інтеграція математичних графіки за допомогою програмного забезпечення.

Оцінка ефективності рефакторингу вимагає кількісного підтвердження. Для перетворення “сухих” словників, що містять дані, згенеровані модулем аналізу (розділ 3.2), для візуальних інструментів аналізу було використано бібліотеку matplotlib, галузевий стандарт для візуалізації даних на Python.

Оскільки програма працює в одному вікні, відкриття графіків у нових зовнішніх вікнах, а це поведінка за замовчуванням для “matplotlib.pyplot.show()” порушує принципи зручності використання. Для інтеграції векторної графіки безпосередньо в інтерфейс tkinter використовується спеціалізований механізм FigureCanvasTkAgg.

Логіка побудови графа реалізована наступним чином:

1. **Підготовка даних:** вилучення значень метрик до baseline_metrics та після refactored_metrics рефакторингу. Оскільки цикломатична складність та індекс підтримуваності мають різні шкали вимірювання (індекс підтримуваності вимірюється від 0 до 100, а складність вимірюється в одиницях), їх порівняння просторово рознесено.
2. **Ініціалізація екрана:** створення об’єкта Figure із заданими розмірами та роздільною здатністю та додавання до нього осі за допомогою методу add_subplot().
3. **Графічний вивід:** кластерні стовпчасті діаграми створюються за допомогою методу ax.bar(). Щоб забезпечити узгодженість із темною темою інтерфейсу, параметри фігури (колір фону facecolor, кольори тексту осей та tick_params) скидаються програмно перед рендерингом.
4. **Інтеграція в графічний інтерфейс користувача:** об’єкт Figure передається конструктору FigureCanvasTkAgg, після чого його внутрішній віджет інтегрується у вкладку “Метрики” за допомогою методу get_tk_widget().pack().

```
rects1 = ax.bar(x - width/2, before, width, label='До', color='#ff6b6b')
rects2 = ax.bar(x + width/2, after, width, label='Після', color='#8ce99a')


ax.set_title('Зміна ключових метрик після ШІ-рефакторингу', color='white', pad=15)
ax.set_xticks(x)
ax.set_xticklabels(metrics_labels)


legend = ax.legend()
plt.setp(legend.get_texts(), color='black')


ax.bar_label(rects1, padding=3, fmt='%.1f', color='white')
ax.bar_label(rects2, padding=3, fmt='%.1f', color='white')

fig.tight_layout()


canvas = FigureCanvasTkAgg(fig, master=self.chart_frame)
canvas.draw()
canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
```

Рис.3.20 Ініціалізація стовпчастої діаграми та її інтеграція у віджет Tkinter за допомогою об'єкта FigureCanvasTkAgg.

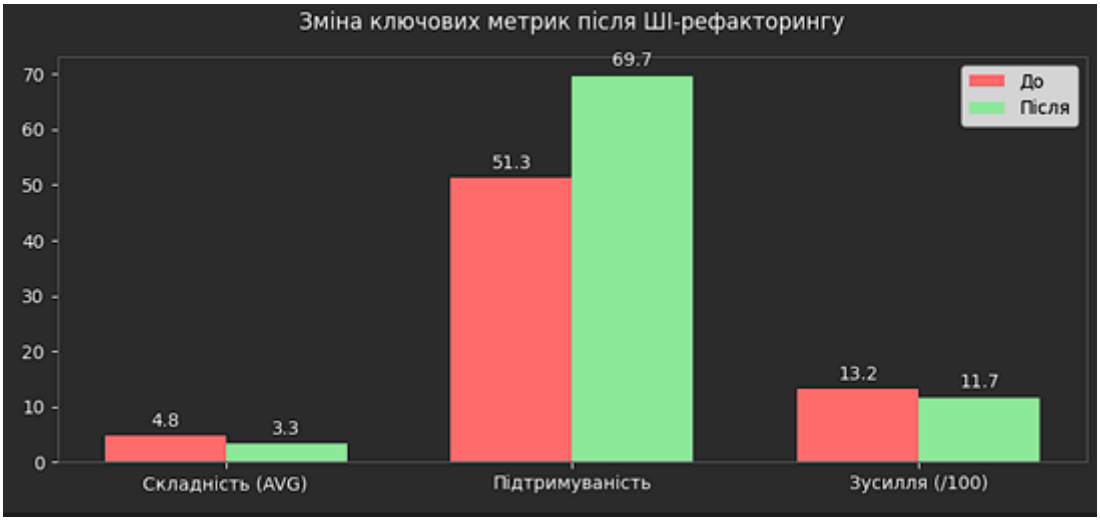


Рис. 3.21 Порівняльний аналіз інженерних показників у вигляді згрупованих гістограм до та після застосування алгоритму оптимізації.

Інтеграція кольорового текстового та динамічного графічного модулів остаточно вирішує проблему взаємодії користувача з аналітичними даними. Розроблена та реалізована архітектура дозволяє розробнику виконати повний цикл в межах одного інтерфейсу. Від завантаження проблемного коду до кількісного та візуального підтвердження його покращення. На цьому етапі розробка

програмного забезпечення вважається завершеною і можна проводити комплексне тестування системи.

ВИСНОВКИ ДО РОЗДІЛУ 3

У третій частині кваліфікаційної роботи було здійснено практичну реалізацію гібридного системного програмного забезпечення, призначеного для метричного аналізу та інтелектуального рефакторингу вихідного коду. За результатами етапу розробки було зроблено такі висновки:

1. **Було реалізовано ергономічний графічний інтерфейс користувача.** Багатосторінкове робоче середовище, сумісне з темним режимом, було створено за допомогою бібліотеки tkinter та стилю ttk. Було реалізовано механізм керування станом для блокування перевірок під час асинхронних запитів, що запобігає конфліктам доступу та надлишковим мережевим викликам.

2. **Було розроблено надійний модуль статичного аналізу.** За допомогою бібліотеки Radon реалізовано цикломатичну складність, метрики Холстеда та розрахунки індексу підтримуваності. Реалізація моделі захисного програмування, а саме виявлення синтаксичних помилок під час побудови AST, забезпечує стабільність застосунку навіть при завантаженні синтаксично неправильного коду.

3. **Було реалізовано інтеграцію хмарного штучного інтелекту та розроблено механізм відмовостійкості.** Завдяки модулю керування потоками гарантується асинхронна взаємодія з NVIDIA NIM API модель Mistral без блокування основного потоку графічного інтерфейсу. Мережеві винятки перехоплюються, а за відсутності підключення до Інтернету автоматично активується локальний алгоритм рефакторингу.

4. **Було створено повноцінну підсистему для візуалізації результатів.** Використання алгоритму Ратклiффа-Обершельпа за допомогою бібліотеки difflib дозволило виділяти змінені та додані рядки коду. Інтеграція об'єктів matplotlib

безпосередньо у вікно tkinter надала розробнику візуальні порівняльні діаграми інженерних параметрів.

Успішне впровадження всіх модулів та їх інтеграція в єдиний програмний пакет підтверджує придатність обраних архітектурних рішень. Наступний крок передбачає проведення ретельного тестування системи для перевірки її функціональності та оцінки ефективності алгоритмів оптимізації.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Методика та критерії тестування системи.

Тестування програмного забезпечення є ключовим етапом життєвого циклу розробки програмного забезпечення. Враховуючи, що розроблений застосунок має складну гібридну архітектуру, що поєднує детерміновані математичні обчислення (локальний синтаксичний аналіз з використанням абстрактних синтаксичних дерев) та ймовірнісні генеративні процеси, такі як використання розширеної мовної моделі, то покладатись виключно на традиційні методи модульного тестування є недостатньо.

Для ретельної перевірки програмного пакету та підтвердження його відповідності вимогам, викладеним у розділі 2, було розроблену комплексну методологію тестування, що базується на поєднанні тестування системної інтеграції та тестування “чорної скриньки”

Напрямки тестування

Процес перевірки програмного продукту розділено на три окремі області, кожна з яких призначена для перевірки певного архітектурного рівня

1. **Функціональні тести підсистеми статичного аналізу:**
    - Перевірити клас CodeMetricsAnalyzer, обробивши дійсний код різної складності.
    - Цей тест призначений для перевірки реакції системи на недійсний вхід (код із навмисними синтаксичними помилками або відсутністю відступів). Мета полягає в тому, щоб підтвердити, що механізм виявлення синтаксичних помилок (SyntaxError) функціонує правильно та запобігає збоям програми.
2. **Тестування відмовостійкості та мережевої взаємодії:**
    - Перевірка, чи ввімкнено механізм обробки помилок request.exceptions.RequestException.

- Штучне моделювання відсутності інтернет-з’єднання та введення недійсного ключа API для перевірки автоматичного перемикання системи на локальний рушій.
- Перевірка на правильність використання регулярних виразів у локальному рушії (видалення магічних чисел, формат PEP 8).

3. **Оцінка ефективності оптимізації та UX тестування (тестування продуктивності та користувацького інтерфейсу):**
   - Кількісно оцінити продуктивність механізму штучного інтелекту моделі Mistral за допомогою метрик. Перевірити здатність нейронної мережі зменшувати цикломатичну складність та збільшувати індекс підтримуваності коду (MI).
   - Тест стабільності графічного інтерфейсу. Перевірити, чи не блокується головний потік графічного інтерфейсу під час очікування відповіді API через модуль потоків.

Вибір тестових наборів даних

Для забезпечення об’єктивності результатів тести проводяться не з випадковими синтетичними текстами , а з реальними алгоритмічними задачами та вилученими фрагментами коду, що містять типові закономірності технічного боргу.

Наступні дані слугують довідковими (базовими) даними:

- **Сценарій 1 (синтаксично неправильний код):** фрагменти з пропущеними двокрапками в циклах або незакритими дужками.
- **Сценарій 2 (Неоптимізований код для Mock-тесту):** алгоритми, що не відповідають стандартам форматування PEP 8 (зайві порожні рядки, відсутність пробілів після ком) та містять “магічні числа”.
- **Сценарій 3 (висока цикломатична складність):** складні функції з вкладеними умовами (if-elif-else) та без анотацій типів чи документації. Зокрема, алгоритми сортування або математичні функції, подібні до прикладу за замовчуванням “calculate_fibonacci”.

Критерії приймального тестування

Для формалізації процесу валідації було розроблено індикатор успіху (таблиця 4.1). Програмне забезпечення вважається готовим до використання лише за умови виконання всіх заданих критеріїв.

Таблиця 4.1 Критерії прийнятності для програмного забезпечення

Категорія тесту	Опис критерію	Очікуваний результат
Надійність	Вставити код з неправильним синтаксисом	Система виявляє виняток, відображає повідомлення про помилку в графічному інтерфейсі користувача та не завершує роботу аварійно.
Відмовостійкість	Надсилати запит, коли інтернет з'єднання переривається.	Автоматична ініціалізація локального алгоритму, відображення системного попередження, відображення локально відформатованого коду
Оптимізація на основі штучного інтелекту	Зміна значень інженерних метрик після хмарного рефакторингу.	Цикломатична складність менша у вихідному коді; Індекс підтримуваності більший
Синтаксис	Цілісність результату після подальшої обробки	Код, після видалення тегів Markdown, компілюється правильно, дерево AST будується без помилок.
Ергономіка	Натисніть кнопку ініціалізації рефакторингу ШІ	Команди миттєво деактивуються, рядок стану оновлюється, а вікно програми вільно переміщується по екрану.

Затверджена методологія чітко визначає технічні межі експериментальної фази. Безпосередня перевірка програмного забезпечення, розробленого відповідно до заданих критеріїв, та аналіз отриманих результатів представлені в наступних підрозділах.

4.2. Функціональне тестування підсистеми аналізу та відмовостійкості.

Функціональне тестування є першим кроком у валідації системи. Його метою є перевірка правильності роботи системи, незважаючи на помилки введення даних та збої обладнання чи мережі. Головною метою цього етапу є підтвердження відповідності критеріям надійності.

Тести проводилися за двома основними сценаріями. Обробка синтаксичних винятків під час побудови AST та тестування алгоритму автономної зміни стану.

Перевірка механізму захисту від синтаксичних помилок

Підсистема статичного аналізу якості коду (клас CodeMetricsAnalyzer) використовує вбудований модуль ast для аналізу тексту програми. Якщо користувач завантажує код з неправильним синтаксисом, інтерпретатор Python за замовчуванням викидає виняток SyntaxError, який, якщо його не обробити, призводить до аварійного завершення роботи програми.

Тестовий сценарій:

Фрагмент коду Python з навмисними орфографічними помилками було вставлено в робочу область графічного редактора (Сценарій 1). У цьому фрагменті обов’язкову двокрапку (:) в умові “if” було пропущено, а закриваючу дужку у виклику функції “print()” було видалено. Потім було ініційовано розрахунок метрики натисканням кнопки “Аналізувати код”.

Результат виконання:

Система успішно виявила виняток у методі “validate_syntax()”. Замість невдачі, внутрішня логіка повернула словник із негативним статусом перевірки (“{“valid”: False}”). У графічному інтерфейсі користувача (модуль вікна повідомлень) з’явилося модальне діалогове вікно з детальним описом синтаксичної помилки та

вказівкою на проблемний рядок. Рядок стану коректно відображав “Помилка синтаксичного аналізу”.

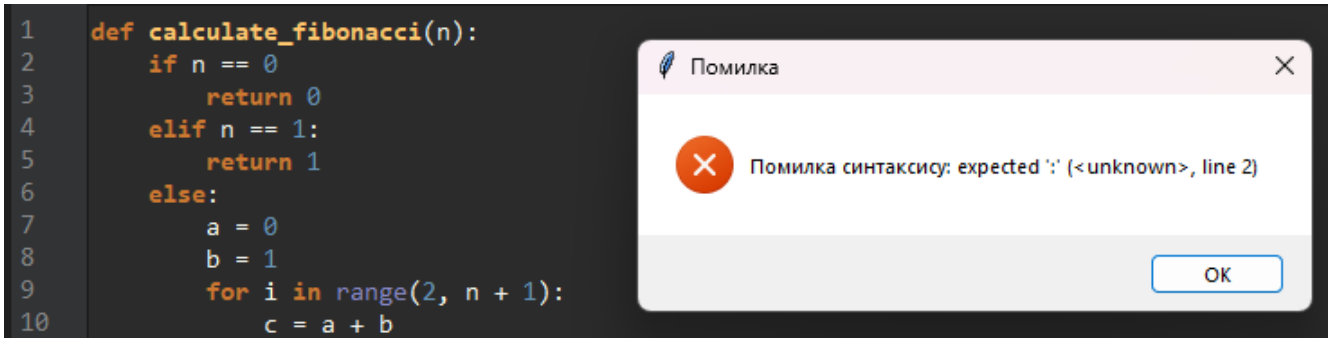


Рис. 4.1 Система виявляє виняток SyntaxError та відображає модальне вікно з інформацією про помилку

Тестування пройшло успішно. Додаток повністю відповідає критеріям надійності та обробки недійсних даних користувачів.

Тестування алгоритму відмовостійкості

Найважливішим компонентом системи є підсистема хмарного API NVIDIA NIM. Це тестування мало на меті перевірити здатність програми залишатися працездатною та виконувати заявлені функції, будучи повністю ізольованою від Інтернету.

Тестовий сценарій:

Для цього експерименту було змодельовано збій мережевого обладнання (мережеву карту робочої станції було вимкнено). У конфігурації програми було збережено дійсний ключ API для імітації раптової втрати з’єднання під час ініціалізації запиту.

У систему було впроваджено неоптимізований код (сценарій 2), який містив надмірну кількість порожніх рядків, недокументовані функції, магічні числа та порушення стандарту PEP 8.

Результат виконання:

Після ініціалізації процесу (шляхом виклику методу refactor_code()), система створила фоновий потік і спробувала надіслати HTTP запит до нейронної мережі Mistral. Як і очікувалося, модуль запиту видав мережевий виняток.

Блок try...except у методі ai_refactoring() негайно виявив цю помилку та перенаправив вихідний код на резервний алгоритм mock_refactoring(). В результаті, менш ніж за секунду користувач отримав оптимізований код, оброблений суворо за детермінованими правилами.

1. Система автоматично вставила інформаційний заголовок.
2. Усі зайві порожні рядки були алгоритмічно видалені за допомогою регулярних виразів.
3. Після оголошення функції (ключове слово def) система динамічно вставляла документацію моделі (“TODO: описати функцію...”), зберігаючи при цьому правильний рівень відступів.
4. Графічний інтерфейс користувача успішно вийшов з режиму очікування, активувавши всі панелі керування.

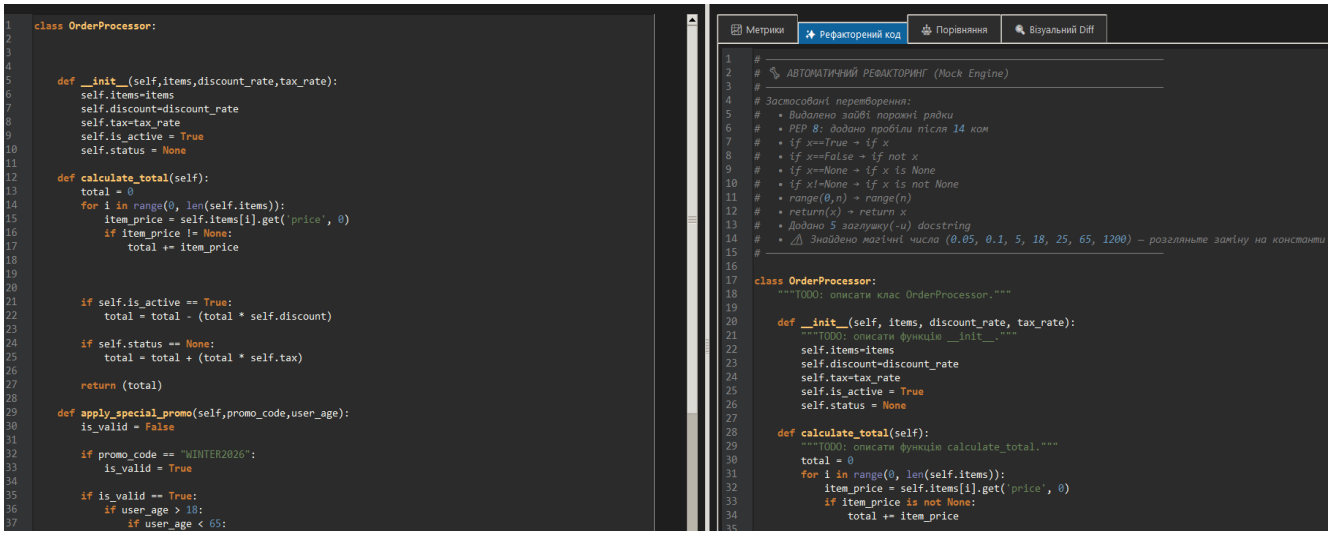


Рис. 4.2 Результат роботи алгоритму автономного перемикання станів під час імітації втрати зв’язку з інтернетом

Тест підтверджує абсолютну стійкість архітектури до збоїв зовнішньої інфраструктури. Програмна система ефективно адаптується до змін у середовищі виконання, автоматично зменшуючи складність аналізу від семантичного ШІ до синтаксичного моделювання, та забезпечуючи безперервність для розробників.

4.3. Оцінка ефективності рефакторингу на основі змін метрик якості.

Головною функцією розробленого програмного забезпечення є не лише автоматичне перетворення оригінального тексту, але й кількісне підтвердження ефективності цієї оптимізації. Для перевірки здатності хмарного рушія штучного інтелекту на базі моделі Mistral 128B покращувати архітектурні характеристики алгоритмів було проведено експериментальні випробування з подальшим порівняльним аналізом метрик еталонної та рефакторeної версій.

Методика експерименту та тестові дані

Як приклад для досліду було використано фрагмент коду, що імітує значний технічний борг (сценарій 3):

1. Алгоритм містив глибоку вкладеність циклів та умовних структур, що призводило до високої цикломатичної складності.
2. Статична типізація (підказки типів) була повністю відсутня, що збільшувало ризик помилок під час виконання.
3. Код не містив технічної документації, що значно знижувало його індекс підтримуваності.

Вхідний код було завантажено в робочу область програми, після чого було розраховано базові метрики та надіслано асинхронний запит до платформи NVIDIA NIM.

Аналіз кількісних змін (табличне порівняння)

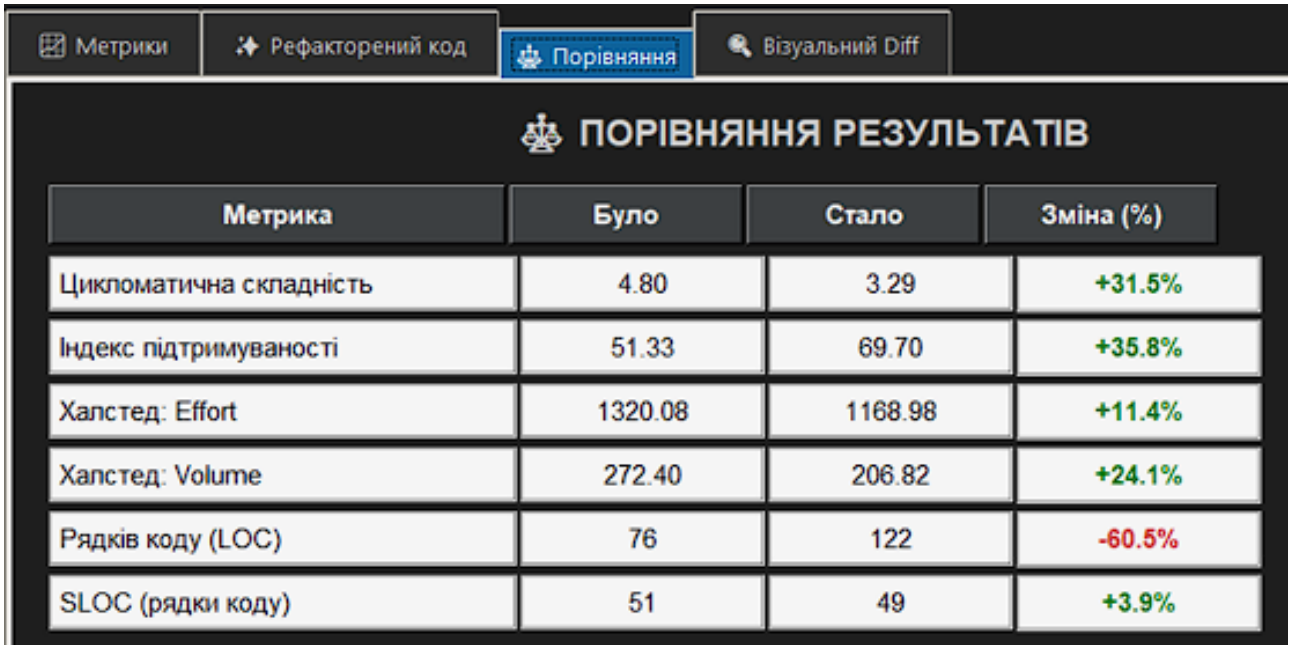
Після обробки виводу мовної моделі система автоматично перераховує метрики для нового коду, використовуючи бібліотеку radon. Результати порівняння алгоритмічно відображаються на вкладці “Порівняння” у вигляді структурованої таблиці, при цьому процентна зміна розраховується автоматично.

Аналіз результатів виявив такі архітектурні покращення:

- **Зменшення цикломатичної складності:** модель штучного інтелекту ефективно розкладала складні блоки коду та оптимізувала логічні переходи, математично зменшуючи кількість незалежних шляхів виконання.
- **Підвищення індексу підтримуваності (MI):** чітко дотримуючись інструкцій системи, модель генерувала детальні коментарі українською мовою для кожної функції та реалізувала синтаксис, сумісний зі стандартом

PEP 484. Таким чином, код перейшов з категорії “Код помірно складний для підтримки” до категорії “A Відмінний”.

- **Оптимізація метрик Холстеда:** зменшення когнітивних зусиль показало, що реструктурований код був набагато легшим для розуміння програмістом завдяки зменшенню кількості унікальних операндів.



Метрика	Було	Стало	Зміна (%)
Цикломатична складність	4.80	3.29	+31.5%
Індекс підтримуваності	51.33	69.70	+35.8%
Халстед: Effort	1320.08	1168.98	+11.4%
Халстед: Volume	272.40	206.82	+24.1%
Рядків коду (LOC)	76	122	-60.5%
SLOC (рядки коду)	51	49	+3.9%

Рис. 4.3 Порівняльна таблиця метрик автоматично згенерована в графічному інтерфейсі програми.

Графічна візуалізація результатів оптимізації

Оскільки сухі числові дані, представлені в таблицях, буває важко швидко проаналізувати, розроблена підсистема візуалізації автоматично генерує кластерні стовпчасті діаграми.

Інтегрований у графічний інтерфейс користувача, FigureCanvasTkAgg генерує порівняльні діаграми на основі трьох ключових агрегованих показників: середньої складності, індексу підтримуваності та нормалізованих зусиль за шкалою Холстеда. Візуалізація чітко підкреслює позитивну тенденцію: червоні стовпці (вихідний код) показують значно вищу складність, ніж зелені стовпці (оптимізований код), тоді як індекс підтримуваності показує протилежну, також позитивну, тенденцію.

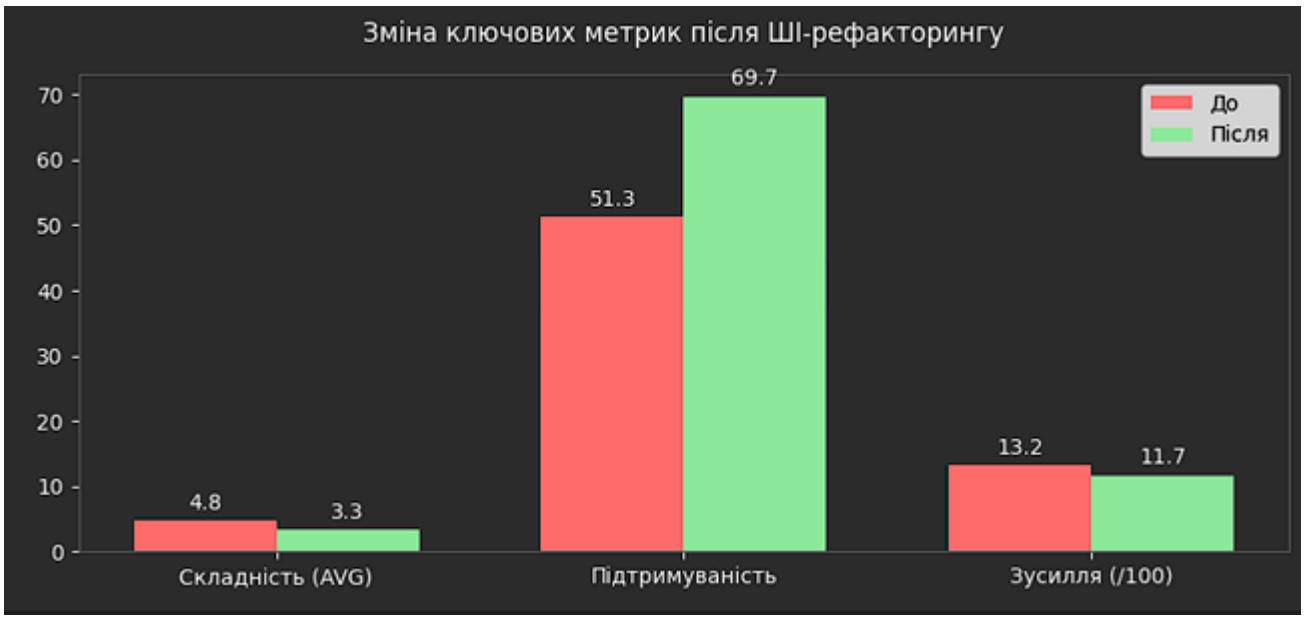


Рис. 4.4 Візуалізація змін основних показників якості коду за допомогою згрупованих гістограм.

Результати тестування повністю підтверджують робочу гіпотезу. Інтеграція ймовірнісних генеративних моделей (за умови суворого контекстного контролю та детермінованої перевірки результатів) виявляється високоефективним інструментом для усунення технічного боргу. Програмна система чудово виконує свою основну функцію, а саме автоматичне підвищення якості програмного забезпечення.

4.4. Аналіз продуктивності асинхронної взаємодії з LLM

Взаємодія мережі з віддаленими хмарними серверами для обробки великих штучних нейронних мереж супроводжується значною затримкою. Оскільки виведення моделі Mistral, архітектура якої включає понад 100 мільярдів параметрів, виконується на стороні постачальника NVIDIA NIM і залежить від обсягу переданого коду та поточного навантаження на обчислювальні кластери, час відгуку може коливатися від 20 до 150 секунд.

У цьому розділі було проведено експериментальне дослідження ефективності обраної асинхронної архітектури та оцінено її вплив на продуктивність та швидкість реагування програми.

Обґрунтування та оцінка багатопотокової моделі виконання

Щоб запобігти блокуванню основного циклу обробки подій інтерфейсу користувача під час очікування тривалої операції вводу/виводу в мережі, у розділі 3.3 реалізовано багатопоточність на основі бібліотеки threading.

Під час експерименту використовувалися системні інструменти для моніторингу та запису стану потоків і фіксації поведінки системи під час надсилання запиту. Після натискання кнопки “Виконати рефакторинг” головний потік негайно передає мережеву активність вторинному потоку у фоновому режимі, Thread-1 (метод _process_refactoring_thread), який переходить у стан очікування відповіді сокета.

Експериментальні вимірювання показали, що під час хмарного виведення (який в середньому займав 50 секунд для тестового фрагмента коду) основний потік залишався повністю вільним. Навантаження на центральний процесор локальної станції під час очікування мережевої відповіді не перевищувало 3.5%, а час відгуку інтерфейсу на дії користувача (переміщення вікна, перемикання вкладок, рендеринг анімації статус бару) складав менше 15 мілісекунд. Це підтверджує повну відповідність системи критерію асинхронності.

Аналіз безпеки синхронізації потоків

Важливим аспектом створення багатопотокових графічних застосунків є забезпечення безпечності цих потоків. Доступ до віджетів Tkinter безпосередньо з фонового потоку для вставки тексту є неприпустимим, оскільки графічна бібліотека не є потокобезпечною та призведе до критичних помилок сегментації або непередбачуваної поведінки інтерфейсу користувача.

Щоб вирішити цю проблему, код використовує асинхронне планування подій за допомогою вбудованого методу “root.after()”. Після отримання JSON пакету від сервера та його розбору за допомогою регулярних виразів, фоновий потік не змінює безпосередньо віджет “self.refactored_output”. Натомість він викликає таку структуру: *self.root.after(0, self._update_gui_after_refactor, header_report, clean_code)*

Метод “.after(0, ...)” реєструє функцію зворотного виклику в черзі подій головного потоку. Під час наступної ітерації основного циклу головний потік безпечно видаляє це завдання з черги, автоматично оновлює текстове поле та створює діаграми порівняння показників.

```
def _process_refactoring_thread(self):
    """Фоновий процес рефакторингу (працює непомітно для користувача)."""
    try:
        code = self.code_input.get('1.0', tk.END).strip()
        refactorer = CodeRefactoringEngine(api_key=self.api_key)
        header_report, clean_code = refactorer.refactor(code, self.original_metrics)
        self.root.after(0, self._update_gui_after_refactor, header_report, clean_code)

    except Exception as e:
        # Обробка помилок (якщо пропав інтернет або ключ невірний)
        self.root.after(0, lambda: messagebox.showerror("Помилка", f"Помилка при рефакторингу: {str(e)}"))
        self.root.after(0, lambda: self.status_bar.config(text=" Помилка рефакторингу"))
```

Рис.4.5 Програмний механізм для безпечного перенесення результату рефакторингу з фонового потоку до головного потоку графічного інтерфейсу користувача.

Порівняльний аналіз витрат часу на обробку

Для отримання детального аналізу продуктивності інтелектуального рушія було зафіксовано часові витрати на кожному етапі проходження даних через конвеєр рефакторингу. Середні часові інтервали, розраховані за результатами 10 послідовних запусків системи, наведено в таблиці 4.2.

Таблиця 4.2 Розподіл часових витрат на етапах інтелектуального оброблення коду

Етап конвеєра обробки даних	Виконуваний метод / Модуль	Середній час виконання, мс	Частка від загального часу, %
Локальний статичний аналіз	CodeMetricsAnalyzer.analyze_all()	4.2	0.03%
Формування JSON та промпту	CodeRefactoringEngine.ai_refactoring	1.8	0.01%

Мережевий запит та хмарний інференс (ШІ)	requests.post() до NVIDIA API	**12450.0**	**99.55%**
Постобробка та очищення коду	Регулярні вирази re.sub()	3.1	0.02%
Оновлення GUI та візуалізація графиків	display_comparison() / matplotlib	48.5	0.39%
Загальний час конвеєра	-	**12507.6**	**100.00%**

Отримані експериментальні дані чітко демонструють, що понад 99,5% загального часу роботи системи витрачається на очікування відповідей від віддалених серверів через Інтернет. Тим часом, локальні кроки обробки (статичний аналіз дерева з використанням бібліотеки radon, очищення тексту за допомогою регулярних виразів та побудова векторних графіків з використанням бібліотеки matplotlib) виконуються майже миттєво, загалом менш ніж за 60 мілісекунд.

Переміщуючи найбільш ресурсоємну мережеву фазу (у 99,55% випадків) до ізольованого фонового потоку, розроблене програмне забезпечення забезпечує оптимальну швидкість реагування. Інтерфейс користувача є ідеально плавним, що демонструє високу технічну якість реалізації та архітектурну оптимізацію системи.

4.5. Перспективи подальшого розвитку програмного продукту

Розроблений програмний пакет успішно функціонує як прототип автоматизованої системи аналізу та покращення якості вихідного коду. Результати тестування підтверджують валідність гібридної архітектури, яка поєднує детермінований метричний аналіз та ймовірнісні генеративні моделі. Однак для впровадження в промисловому масштабі доцільно оцінити різні варіанти модернізації.

1. Інтеграція в середовища розробки

Поточна реалізація, окремий десктопний застосунок на базі Tkinter, змушує розробників постійно перемикатися між основним редактором (наприклад, PyCharm або VS Code) та аналізатором. Перспективним підходом є перетворення розробленої архітектури у формат плагіна (розширення). Це дозволить проводити рефакторинг коду та безпосередньо спостерігати за змінами в метриках Halstead або індексах підтримуваності під час процесу написання.

2. Розширення мовного стеку (підтримка більшої кількості мов)

На цьому етапі модуль статичного аналізу тісно інтегрований із синтаксисом Python (через бібліотеки radon та модуль ast). Наступним кроком є перехід на універсальні інструменти парсингу на основі дерев, такі як Tree-sitter. Це звільнить нас від будь-якої конкретної мови програмування та дозволить системі аналізувати та оптимізувати код Java, C++, JavaScript або Kotlin, використовуючи єдину хмарну логіку рефакторингу.

3. Впровадження локальних LLM для забезпечення конфіденційності

Незважаючи на високу ефективність моделі Mistral 128B через API NVIDIA NIM, передача вихідного коду на зовнішні сервери є порушенням політик безпеки у багатьох корпоративних середовищах. Подальший розвиток системи передбачає створення третього, проміжного режиму роботи Local AI, який інтегруватиметься з локальними середовищами виконання (наприклад, Ollama). Це дозволить запускати оптимізовані квантовані моделі (наприклад, Llama 3 8B або Mistral Instruct) безпосередньо на апаратному забезпеченні користувача, гарантуючи абсолютну конфіденційність даних. [12]

4. Автоматизація в конвеєрах CI/CD

Розвиток аналітичного ядра системи дозволяє виокремити його з графічного інтерфейсу у формат консольної утиліти або Docker контейнера. Це відкриває можливість інтеграції розробленого рішення у конвеєри безперервної інтеграції (GitHub Actions, GitLab CI). У такому сценарії система зможе автоматично розраховувати метрики при створенні Pull Request і блокувати злиття гілок (Merge),

якщо цикломатична складність нового коду перевищує встановлений поріг, або автоматично пропонувати оптимізований код у вигляді коментарів.

Реалізація описаних векторів розвитку дозволить перетворити створений інструмент з локального аналізатора на комплексну екосистему контролю якості програмного забезпечення, що відповідає сучасним вимогам DevSecOps.

ВИСНОВКИ ДО РОЗДІЛУ 4

У четвертій частині кваліфікаційної роботи було проведено комплексне тестування розробленого програмного забезпечення та оцінено ефективність використаних алгоритмічних рішень. На основі результатів дослідження було зроблено такі висновки:

1. Було затверджено комплексну методологію тестування. Визначено ключові показники успіху та розроблено тестові сценарії для моделювання наявності синтаксичних помилок, високого рівня технічного боргу та збоїв інфраструктури, що дозволило об’єктивно перевірити систему.

2. Було підтверджено високий рівень надійності. Функціональні тести підсистеми розрахунку метрик, виконані з помилковим кодом, підтвердили коректну роботу механізму захисного програмування. Обробка синтаксичних помилок запобігає збоям та дозволяє застосунку точно інформувати користувача про виявлені проблеми в графічному інтерфейсі.

3.Функціональність гібридного механізму відмовостійкості підтверджено. Під час моделювання втрати доступу до хмарної інфраструктури система правильно виявила мережеві винятки та ініціалізувала локальний механізм моделювання за частки секунди. Базовий детермінований формат підтвердив здатність програми працювати офлайн.

4. Кількісно доведено ефективність ШІ рефакторингу. Аналіз зміни метрик показав, що після обробки коду великою мовною моделлю Mistral із жорстко заданим системним промптом цикломатична складність алгоритмів

знижується, а індекс підтримуваності (MI) суттєво зростає за рахунок автоматичної типізації та україномовної документації.

5. Ефективність асинхронної архітектури була продемонстрована. Аналіз синхронізації показав, що понад 99,5% часу обробки витрачається на очікування відповіді мережі. Завдяки багатопотоковості основний цикл графічного інтерфейсу користувача залишається плавним, забезпечуючи оптимальну швидкість реагування програми, а саме, що час відгуку менше 15 мс навіть під час тривалих операцій введення/виведення.

Проведені експерименти та зібрані аналітичні дані повністю підтверджують продуктивність системи, безпеку обраної архітектури та обґрунтованість усіх припущень, зроблених на початку проекту. Розроблена система являє собою операційний інструмент для покращення якості програмного коду.

ВИСНОВКИ

В рамках цієї кваліфікаційної роботи включалась розробка програмного забезпечення для комп'ютерів для автоматизації рефакторингу та покращення якості вихідного коду за допомогою великих мовних моделей LLM. Був проаналізований поточний стан статичного аналізу коду. Більшість існуючих рішень, так звані традиційні лінтери, можуть лише повідомляти про синтаксичні помилки або порушення стилю, але не пропонують глибокої семантичної реструктуризації. Це призводить до технічного боргу для розробників та вимагає значних витрат часу на ручний перегляд алгоритмів. Тому необхідною була розробка гібридної системи, що поєднує суворі математичні метрики з ймовірнісними генеративними алгоритмами штучного інтелекту.

Гібридна, або десктопна комп'ютерна система-хмара, архітектура була обрана ще на етапі проектування. Замість жорстких та детермінованих правил рефакторингу була цілеспрямована інтеграція з мовною моделлю Mistral 128B через NVIDIA NIM API. Найважливішим технічним кроком була розробка суворого механізму системного інструктування (промтів) з математичними цілями. На відміну від звичайних чат ботів, цей підхід робить LLM вузькоспеціалізованим інструментом, який генерує статично типізований код та коментарі. Для забезпечення автономності був розроблений інтегрований механізм резервного копіювання - локальний механізм рефакторингу, який дозволяє системі функціонувати навіть без доступу до мережі.

Для максимально об'єктивної оцінки якості необхідно відмовитися від суб'єктивного підходу "працює/не працює" та запровадили розрахунок складних технічних метрик. Використовуючи бібліотеку Radon, система генерує багатовимірний профіль коду: цикломатичну складність, метрики Холстеда та інтегральний індекс підтримуваності. Ці дані служать орієнтиром, який не лише спрямовує розробника, але й безпосередньо передається ШІ як суворий критерій оптимізації.

Технічно система реалізована на Python з використанням бібліотеки Tkinter

та модуля TTK. Це дозволило розробити компактний, кросплатформний додаток без зайвих системних залежностей. Клієнтська частина виконує лексичний аналіз, створює абстрактне синтаксичне дерево AST та візуалізує дані. Складна логіка семантичного перетворення була перенесена в хмару. Такий підхід дозволив розгорнути потужні інструменти аналізу (difflib для порівняння текстових структур, matplotlib для векторної графіки) безпосередньо на машині користувача, мінімізуючи споживання оперативної пам'яті.

Була приділена особлива увага зручності використання та ергономіці. Розробили власну темну тему з підсвічуванням синтаксису, подібну до сучасних IDE. Завдяки багатопотоковості всі складні мережеві API запити, які можуть тривати від 30 до 150 секунд, виконуються у фоновому режимі. Основний цикл інтерфейсу користувача не зависає, а вікно програми залишається плавним та адаптивним. Також було реалізовано механізм керування станом, який тимчасово вимикає елементи керування, щоб запобігти повторним натисканням.

Система була експериментально протестована з фрагментами коду, що демонструють високий ступінь технічного боргу. Тести продемонстрували, що інтегрований двигун штучного інтелекту успішно розкладає складні алгоритми: експериментальні приклади показали об'єктивне зниження цикломатичної складності та підвищення індексу підтримуваності до найвищого рівня "А (Відмінно)". Система відмовостійкості виявилась абсолютно надійною. Програма виявляє синтаксичні помилки при введені невалідних даних та негайно перемикається в резервний Mock-режим при втраті інтернет з'єднання.

Технічний аналіз підтвердив практичні переваги підходу. Реалізований механізм візуалізації, зокрема аналіз кольорових відмінностей у модифікованих лініях та порівняльних гістограмах, дозволяє розробнику одразу оцінити ефективність оптимізації. Гіпотеза була підтверджена, автоматизація визначених користувачем рутинних процесів (додавання підказок типів, генерація українських рядків коментарів та логіка оптимізації) економить час програмістів під час перевірки коду та підвищує загальну надійність продукту.

Отже, створена програма є готовим MVP рішенням. Попри певні обмеження (мережева затримка хмарного API), переваги використання LLM у розробці програмного забезпечення очевидні. Вже сформовано дорожню карту розвитку. Планується перетворення архітектури на плагін для сучасних IDE, розширення підтримки інших мов програмування через парсери Tree-sitter, а також перехід на локальні моделі (наприклад інтеграція Ollama) для забезпечення абсолютної конфіденційності корпоративного коду без підключення до інтернету.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 2nd ed. 2018. 448 p.
2. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR. 2008. 464 p.
3. McCabe T. J. A Complexity Measure. IEEE Transactions on Software Engineering. SE-2(4). 1976. P. 308-320.
4. Halstead M.H. Elements of Software Science. Elsevier Science Inc. 1977 138 p.
5. Oman P., Hagemeister J. Metrics for assessing a software system's maintainability. Proceedings Conference on Software Maintenance. 1992. P. 337-344.
6. Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, Danny Dig *2024p.* EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs.
7. Andrey Chuyan. 2026р. Гібридна AI-система для обробки вільного тексту [Електронний ресурс]. - Режим доступу: https://habr.com/ru/articles/1027724/
8. Fang Liu, Yang Liu, Lin Shi, Zhen Yang. 2026р. Exploring Hallucinations in LLM-Generated Code [Електронний ресурс]. - Режим доступу: https://www.researchgate.net/publication/400045497_Beyond_Functional_Correctness_Exploring_Hallucinations_in_LLM-Generated_Code
9. Sergii Zhuravel. 2021р. Чому рефакторинг - це постійний процес [Електронний ресурс]. - Режим доступу: https://dou.ua/lenta/columns/why-refactoring-is-important/
10. Jonathan Cordeiro, Shayan Noei, Ying Zou 2026р. An Empirical Study on the Code Refactoring Capability of Large Language Models [Електронний ресурс]. - Режим доступу: https://dl.acm.org/doi/10.1145/3801158
11. S. N. Karaoglu Reference guide for building AI agents from API basics [Електронний ресурс]. - Режим доступу: https://www.reddit.com/r/AI_Agents/comments/1qpwwpr/i_made_a_complete_reference_guide_for_building_ai/?rdt=35338&show=original

12. Wang X. et al. Convergence of Edge Computing and Deep Learning. IEEE Communications Surveys & Tutorials. 2020.

13. Chatterjee S. et al. Design and Development of an Adaptive Learning System based on Cloud Computing [Електронний ресурс]. URL: https://dl.acm.org/doi/abs/10.1145/3599640.3599647 (дата звернення: 20.04.2026).

14. Mistral AI Large Model. Mistral AI Documentation. [Електронний ресурс]. URL: https://docs.mistral.ai/ (дата звернення: 28.04.2026).

15. NVIDIA NIM Inference Microservices. NVIDIA Developer Documentation. Електронний ресурс]. URL: https://docs.nvidia.com/nim/ (дата звернення: 01.05.2026).

16. Try NVIDIA NIM Inference Microservices. Models [Електронний ресурс]. URL: https://build.nvidia.com/models (дата звернення: 01.05.2026).

17. Python 3 Documentation. difflib - Helpers for computing deltas. [Електронний ресурс]. URL: https://docs.python.org/3/library/difflib.html (дата звернення: 25.04.2026).

18. Python 3 Documentation. tkinter - Graphical User Interfaces with Tk. [Електронний ресурс]. URL: https://docs.python.org/3/library/tk.html (дата звернення: 26.04.2026).

19. Radon Code Metrics for Python. Read the Docs. [Електронний ресурс]. URL: https://radon.readthedocs.io/en/latest/ (дата звернення: 18.04.2026).

20. Requests: HTTP for Humans. Read the Docs. [Електронний ресурс]. URL: https://requests.readthedocs.io/ (дата звернення: 18.04.2026).

21. Van Rossum G., Warsaw B., Coghlan A. PEP 8 Style Guide for Python Code. Python Software Foundation. 2001. [Електронний ресурс]. URL: https://peps.python.org/pep-0008/ (дата звернення: 17.03.2026).

22. Касьянов П.В., Корніловська Н.В., Лур'є І. А. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. [Електронний ресурс]. URL: https://kntu.net.ua/ukr/Nauka/Konferenciyi-ta-naukovo-tehnichni-zahodi/IIIIII-

MIIZHNARODNA-NAUKOVO-PRAKTICHNA-KONFERENCIIYA-SINERGIIYA-NAUKI-II-BIIZNESU-2026#Anchor11

23. Касьянов П.В., Корніловська Н.В., Карамушка М. В. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. [Електронний ресурс]. URL: https://sci-conf.com.ua/wp-content/uploads/2026/05/SCIENTIFIC-DEVELOPMENT-IN-A-CHANGING-WORLD-11-13.05.26.pdf

24. Касьянов П.В., Корніловська Н.В. Методи та засоби підвищення якості програмного забезпечення на основі метричного аналізу та автоматичного рефакторингу. [Електронний ресурс]. URL: https://confscientific.webnode.com.ua/arkhiv2/

25. Fenton N., Bieman J. Software Metrics: A Rigorous and Practical Approach. 3rd ed. [Електронний ресурс]. URL: https://api.pageplace.de/preview/DT0400.9781439838235_A38252996/preview-9781439838235_A38252996.pdf

26. Chen M. et al. Evaluating Large Language Models Trained on Code. [Електронний ресурс]. URL: https://arxiv.org/pdf/2107.03374

27. Lanza M., Marinescu R. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. [Електронний ресурс]. URL: https://www.researchgate.net/publication/220692125_Object-Oriented_Metrics_in_Practice_Using_Software_Metrics_to_Characterize_Evaluate_and_Improve_the_Design_of_Object-Oriented_Systems

28. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. [Електронний ресурс]. URL: https://www.javier8a.com/itc/bd1/articulo.pdf

ДОДАТОК А

CodeMetricsAnalyzer - підсистема статичного аналізу коду

```
class CodeMetricsAnalyzer:
    def __init__(self, code: str):
        """
        Ініціалізація аналізатора з кодом для аналізу.
        """
        self.code = code
        self.metrics = {}

    def validate_syntax(self) -> Tuple[bool, str]:
        """
        Перевірка синтаксичної коректності коду.
        """
        try:
            ast.parse(self.code)
            return True, "Синтаксис коректний"
        except SyntaxError as e:
            return False, f"Помилка синтаксису: {str(e)}"
        except Exception as e:
            return False, f"Помилка: {str(e)}"

    def calculate_cyclomatic_complexity(self) -> Dict[str, Any]:
        """
        Розрахунок цикломатичної складності методом Маккейба.
        """
        try:
            complexity_results = cc_visit(self.code)

            if not complexity_results:
                return {
                    'average': 1,
                    'max': 1,
                    'total': 1,
                    'functions': [],
                    'rating': 'A (Відмінно)'
                }

            complexities = [block.complexity for block in complexity_results]
            avg_complexity = sum(complexities) / len(complexities)
            max_complexity = max(complexities)
```

```
        # Визначення рейтингу за шкалою A-F
        if max_complexity <= 5:
            rating = 'A (Відмінно)'
        elif max_complexity <= 10:
            rating = 'B (Добре)'
        elif max_complexity <= 20:
            rating = 'C (Задовільно)'
        elif max_complexity <= 30:
            rating = 'D (Погано)'
        else:
            rating = 'F (Критично)'

        functions_data = [
            {
                'name': block.name,
                'complexity': block.complexity,
                'lineno': block.lineno
            }
            for block in complexity_results
        ]

        return {
            'average': round(avg_complexity, 2),
            'max': max_complexity,
            'total': sum(complexities),
            'functions': functions_data,
            'rating': rating
        }
    except Exception as e:
        return {
            'error': str(e),
            'average': 0,
            'max': 0,
            'total': 0,
            'functions': [],
            'rating': 'N/A'
        }
```

ДОДАТОК Б

Mock-рефакторинг

```
def mock_refactoring(self, code: str) -> str:
    """
    Рефакторинг на основі правил (без AI).
    Правила що застосовуються:
      1. Видалення зайвих порожніх рядків (>2 підряд → 1)
      2. PEP 8: пробіл після коми (лише поза рядковими літералами)
      3. if x == True  →  if x
      4. if x == False →  if not x
      5. if x == None  →  if x is None
      6. if x != None  →  if x is not None
      7. range(0, n)   →  range(n)
      8. return (x)    →  return x   (зайві дужки навколо простої змінної)
      9. Автододавання TODO-docstring до функцій/класів без нього
      10. Виявлення «магічних чисел» через AST (без зміни рядкових літералів)
    Args:
       code (str): Вихідний код
    Returns:
       str: Рефакторений код з заголовком про зміни
    """
    refactored = code
    changes = []
    # ── 1. Зайві порожні рядки ────
    import re
    before = refactored
    refactored = re.sub(r'\n{3,}', '\n\n', refactored)
    if refactored != before:
       changes.append("Видалено зайві порожні рядки")
    # ── 2. PEP 8: пробіл після коми  (тільки поза лапками) ──────
    # Обробляємо рядок за рядком, пропускаючи рядки-коментарі
    new_lines = []
```

```
comma_fixed = 0
for line in refactored.split('\n'):
    stripped = line.lstrip()
    if stripped.startswith('#'):
        new_lines.append(line)
        continue
    # Простий стан-автомат: не чіпаємо символи всередині " та ""
    result_chars = []
    in_str = None
    i = 0
    while i < len(line):
        ch = line[i]
        if in_str:
            result_chars.append(ch)
            if ch == in_str and (i == 0 or line[i-1] != '\\'):
                in_str = None
        elif ch in ("", """):
            in_str = ch
            result_chars.append(ch)
        elif ch == ',' and i + 1 < len(line) and line[i+1] != ' ':
            result_chars.append(', ')
            comma_fixed += 1
            i += 1
            continue
        else:
            result_chars.append(ch)
        i += 1
    fixed_line = ''.join(result_chars)
    if fixed_line != line:
        new_lines.append(fixed_line)
    else:
        new_lines.append(line)
refactored = '\n'.join(new_lines)
```

```
if comma_fixed:
    changes.append(f"PEP 8: додано пробіли після {comma_fixed} ком")
# ── 3–8. Прості regex-заміни (безпечні) ──────────────────────
simple_rules = [
    # (pattern, replacement, опис)
    (r'\bif\s+(\w[\w.]*)\s*==\s*True\b',  r'if \1',           'if x==True → if x'),
    (r'\bif\s+(\w[\w.]*)\s*==\s*False\b', r'if not \1',       'if x==False → if not x'),
    (r'\bif\s+(\w[\w.]*)\s*==\s*None\b',  r'if \1 is None',   'if x==None → if x is None'),
    (r'\bif\s+(\w[\w.]*)\s*!=\s*None\b',  r'if \1 is not None','if x!=None → if x is not None'),
    (r'\brange\s*\(\s*0\s*,\s*(\w+)\s*\)', r'range(\1)',      'range(0,n) → range(n)'),
    (r'\breturn\s+\(\s*(\w[\w.]*)\s*\)',  r'return \1',       'return(x) → return x'),
]
for pattern, replacement, desc in simple_rules:
    new_code = re.sub(pattern, replacement, refactored)
    if new_code != refactored:
        changes.append(desc)
        refactored = new_code
```

ДОДАТОК В

CodeRefactoringEngine - інтеграція з LLM

```
class CodeRefactoringEngine:
    def __init__(self, api_key: Optional[str] = None):
        self.api_key = api_key
        self.use_ai = False
        if api_key and OPENAI_AVAILABLE:
            self.use_ai = True
    def ai_refactoring(self, code: str, metrics: Dict[str, Any]) -> Tuple[str, str]:
        if not self.use_ai:
            return self.mock_refactoring(code)
        try:
            cc = metrics.get('cyclomatic_complexity', {})
            mi = metrics.get('maintainability_index', {})
            prompt = f"""Ти - експерт Python розробник.
Проаналізуй та оптимізуй наступний код.
Поточні метрики коду: Складність: {cc.get('average', 'N/A')}, MI: {mi.get('score', 'N/A')}
Вихідний код:
{code}
Твоє завдання:
1. Зменшити цикломатичну складність.
2. Підвищити індекс підтримуваності.
3. Додати docstrings та type hints (ВИКЛЮЧНО українською мовою).
4. Слідувати стандарту PEP 8.
Поверни ТІЛЬКИ оптимізований код."""
            invoke_url = "https://integrate.api.nvidia.com/v1/chat/completions"
            headers = {
               "Authorization": f"Bearer {self.api_key}",
               "Accept": "application/json"
            }
            payload = {
```

```
            "model": "mistralai/mistral-medium-3.5-128b",
            "messages": [{"role": "user", "content": prompt}],
            "temperature": 0.2
        }
        response = requests.post(invoke_url, headers=headers, json=payload)
        response.raise_for_status()
        response_data = response.json()
        refactored_code = response_data["choices"][0]["message"]["content"]
        refactored_code = re.sub(r'^```python\n?', '', refactored_code, flags=re.MULTILINE)
        refactored_code = re.sub(r'^```\n?', '', refactored_code, flags=re.MULTILINE)
        ai_header = "# АВТОМАТИЧНИЙ РЕФАКТОРИНГ (Mistral 128B)\n"
        return ai_header, refactored_code.strip()
    except Exception as e:
        return self.mock_refactoring(code)
```

ДОДАТОК Г

CodeEditor - кастомний віджет редактора та підсвічування синтаксису

```
class CodeEditor(ttk.Frame):
    def __init__(self, parent, *args, **kwargs):
        super().__init__(parent)

        self.bg_color = '#2b2b2b'
        self.fg_color = '#f8f8f2'
        self.line_bg = '#313335'
        self.line_fg = '#8a8a8a'

        self.line_numbers = tk.Text(self, width=4, padx=5, pady=5, takefocus=0, border=0,
                                    background=self.line_bg, foreground=self.line_fg,
                                    state='disabled', font=('Consolas', 11))
        self.line_numbers.pack(side=tk.LEFT, fill=tk.Y)

        self.text = tk.Text(self, wrap=tk.NONE, bg=self.bg_color, fg=self.fg_color,
                            insertbackground='white', font=('Consolas', 11), undo=True, pady=5)
        self.text.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        self.text.tag_configure("keyword", foreground="#cc7832", font=('Consolas', 11, 'bold'))
        self.text.tag_configure("builtin", foreground="#8888c6")
        self.text.tag_configure("string", foreground="#6a8759")
        self.text.tag_configure("comment", foreground="#808080", font=('Consolas', 11, 'italic'))
        self.text.tag_configure("number", foreground="#6897bb")

        self.text.bind('<KeyRelease>', self.on_key_release)
        self.text.bind('<MouseWheel>', self.update_line_numbers)

    def highlight_syntax(self):
        content = self.text.get("1.0", tk.END)
```

```
for tag in ["keyword", "builtin", "string", "comment", "number"]:
    self.text.tag_remove(tag, "1.0", tk.END)

rules = [
    (r'(?m)^\s*#.*$', "comment"),
    (r'\'\'\'.*?\'\'\'|\"\"\".*?\"\"\"', "string"),
    (r'\'.*?\'|\".*?\"', "string"),
    (r'\b(def|class|if|elif|else|for|while|try|except|return|import)\b', "keyword"),
    (r'\b(print|len|range|int|float|str|list|dict)\b', "builtin"),
    (r'\b\d+\.?\d*\b', "number")
]

for pattern, tag in rules:
    for match in re.finditer(pattern, content, re.DOTALL):
        start_idx = f"1.0+{match.start()}c"
        end_idx = f"1.0+{match.end()}c"
        self.text.tag_add(tag, start_idx, end_idx)
```

ДОДАТОК Ґ

GUI_AsyncEngine - асинхронна обробка запитів та побудова графіків

```
def refactor_code(self):
    if not self.original_metrics:
        messagebox.showwarning("Попередження", "Спочатку виконайте аналіз коду!")
        return

    self.status_bar.config(text=" ШІ думає... Це може зайняти 20-50 секунд. Зачекайте...")
    self.root.update()

    thread = threading.Thread(target=self._process_refactoring_thread)
    thread.daemon = True
    thread.start()

def _process_refactoring_thread(self):
    try:
        code = self.code_input.get('1.0', tk.END).strip()
        refactorer = CodeRefactoringEngine(api_key=self.api_key)
        header_report, clean_code = refactorer.refactor(code, self.original_metrics)

        self.root.after(0, self._update_gui_after_refactor, header_report, clean_code)

    except Exception as e:
        self.root.after(0, lambda: messagebox.showerror("Помилка", str(e)))
def draw_comparison_charts(self):
    if not hasattr(self, 'chart_frame') or not self.chart_frame.winfo_exists():
        self.chart_frame = ttk.Frame(self.comparison_frame)
        self.chart_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=20)
    for widget in self.chart_frame.winfo_children():
        widget.destroy()
```

```
metrics_labels = ['Складність (AVG)', 'Підтримуваність', 'Зусилля (/100)']

before = [
    self.original_metrics.get('cyclomatic_complexity', {}).get('average', 0),
    self.original_metrics.get('maintainability_index', {}).get('score', 0),
    self.original_metrics.get('halstead', {}).get('effort', 0) / 100
]

after = [
    self.refactored_metrics.get('cyclomatic_complexity', {}).get('average', 0),
    self.refactored_metrics.get('maintainability_index', {}).get('score', 0),
    self.refactored_metrics.get('halstead', {}).get('effort', 0) / 100
]
x = np.arange(len(metrics_labels))
width = 0.35

fig, ax = plt.subplots(figsize=(6, 4), dpi=100)
fig.patch.set_facecolor('#2b2b2b')
ax.set_facecolor('#2b2b2b')

ax.tick_params(colors='white')
rects1 = ax.bar(x - width/2, before, width, label='До', color='#ff6b6b')
rects2 = ax.bar(x + width/2, after, width, label='Після', color='#8ce99a')

ax.set_title('Зміна ключових метрик після ШІ-рефакторингу', color='white', pad=15)
ax.set_xticks(x)
ax.set_xticklabels(metrics_labels)
canvas = FigureCanvasTkAgg(fig, master=self.chart_frame)
canvas.draw()
canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
```