

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ Комп'ютерних наук
(повна назва)
Кафедра _____ Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____ перший (бакалаврський)

_____ Інтелектуальна система автоматизованого контролю якості та стилістики
_____ програмного коду на основі методів машинного навчання
(тема)

Виконав:
здобувач _____ четвертого _____ року навчання,
групи _____ ІТШ-22-3

_____ Аліна БОБК
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки

(код і повна назва спеціальності)

Тип програми _____ освітньо-професійна

Освітня програма _____ Штучний інтелект

(повна назва освітньої програми)

Керівник _____ доц. Олена ВОЛОЩУК
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ

(підпис)

_____ Лариса ЧАЛА
(власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет _____ Комп'ютерних наук _____

Кафедра _____ Штучного інтелекту _____

Рівень вищої освіти _____ перший (бакалаврський) _____

Спеціальність _____ 122 Комп'ютерні науки _____

(код і повна назва)

Тип програми _____ освітньо-професійна _____

Освітня програма _____ Штучний інтелект _____

(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____

(підпис)

« _____ » _____ 20 ____ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві _____ Вовк Аліні Віталіївні _____

(прізвище, ім'я, по батькові)

1. Тема роботи Інтелектуальна система автоматизованого контролю якості та стилістики програмного коду на основі методів машинного навчання

затверджена наказом університету від 5 червня 2026 р. № 488Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 14 липня 2026 р.

3. Вихідні дані до роботи Наукові публікації та технічна документація за напрямом штучного інтелекту; вихідний код програм мовою Python та відкриті репозиторії GitHub; двомодальна трансформерна архітектура CodeBERT; модульні бібліотеки синтаксичного розбору абстрактного дерева (AST); асинхронний серверний фреймворк FastAPI; локальна інфраструктура розгортання типу On-Premise.

4. Перелік питань, що потрібно опрацювати в роботі _____

1) Аналіз предметної галузі та постановка задачі

2) Розробка методики та архітектури інтелектуальної системи аналізу коду

3) Програмна реалізація та експериментальні дослідження

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.06.2026	виконано
2	Аналіз літературних джерел, існуючих лінтерів та SAST-систем	13.06.2026	виконано
3	Розробка математичної моделі та архітектури системи	17.06.2026	виконано
4	Програмна реалізація модулів AST-аналізу та серверної частини API	23.06.2026	виконано
5	Донавчання (Fine-tuning) нейромережевої моделі CodeBERT	29.06.2026	виконано
6	Проведення експериментальних досліджень та розрахунок метрик	02.07.2026	виконано
7	Написання та редагування пояснювальної записки	04.07.2026	виконано
8	Перевірка на академічний плагіат та нормоконтроль	05.07.2026	виконано
9	Підготовка презентаційного матеріалу та доповіді	06.07.2026	виконано
10	Попередній захист роботи на кафедрі	07.07.2026	виконано
11	Рецензування кваліфікаційної роботи	08.07.2026	виконано
12	Захист перед ЕК	14.07.2026	

Дата видачі завдання 8 червня 20 26 р.

Здобувач 
(підпис)

Керівник роботи _____
(підпис)

доц. Олена ВОЛОЩУК
(посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 71 с., 4 рис., 5 табл., 2 дод., 20 джерел.

АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, ГІБРИДНИЙ АНАЛІЗ КОДУ, КОНТРОЛЬ ЯКОСТІ КОДУ, МАШИННЕ НАВЧАННЯ, НЕЙРОМЕРЕЖЕВІ ЕНКОДЕРИ, СТАТИЧНИЙ АНАЛІЗ, СТИЛІСТИКА КОДУ, ТРАНСФОРМЕРНА АРХІТЕКТУРА, CODEBERT, FASTAPI.

Об'єкт дослідження – процедури комп'ютерного моніторингу та оцінювання вихідних текстів програмного забезпечення з метою виявлення відхилень від стилістичних і структурних регламентів.

Предмет дослідження – алгоритми глибинного навчання, нейромережеві енкодери та AST-трансформації, призначені для розпізнавання аномалій і структурних дефектів у програмних модулях.

Мета роботи – проєктування та реалізація інтелектуальної мікросервісної платформи для комплексного оцінювання надійності програмного забезпечення, автоматизованого проведення Code Review та динамічного розрахунку технічного боргу.

Методи дослідження – гібридний підхід до аналізу, що поєднує детермінований розбір синтаксичного дерева (за допомогою модульних бібліотек AST мови Python) для вилучення інженерних метрик (цикломатичної складності, довжини методів) та ймовірніське семантичне кодування з використанням донавченої двомодальної трансформерної моделі CodeBERT.

У кваліфікаційній роботі проаналізовано виклики підтримання якості кодової бази та обмеження існуючих лінтерів. Запропоновано архітектуру гібридної системи, що поєднує синтаксичний розбір із нейромережевими кодувальниками, що дозволяє знизити рівень хибних спрацьовувань та автоматизувати аудит ПЗ.

ABSTRACT

Bachelor's thesis contains: 71 pp., 4 fig., 5 tabl., 2 ann., 20 references.

ABSTRACT SYNTAX TREE, CODEBERT, HYBRID CODE ANALYSIS, CODE QUALITY CONTROL, MACHINE LEARNING, NEURAL ENCODERS, STATIC ANALYSIS, CODE STYLE, TRANSFORMER ARCHITECTURE, FASTAPI.

Object of research – computer monitoring and evaluation procedures for software source code aimed at identifying deviations from stylistic and structural regulations.

Subject of research – deep learning algorithms, neural encoders, and AST transformations designed to recognize anomalies and structural defects in software modules.

Purpose of the work – to design and implement an intelligent microservice platform for comprehensive software reliability assessment, automated Code Review, and dynamic technical debt calculation.

Research methods – a hybrid analysis approach combining deterministic syntax tree parsing (using Python AST modules) to extract engineering metrics (cyclomatic complexity, method length) and probabilistic semantic encoding using a fine-tuned dual-modal transformer model CodeBERT.

The thesis analyzes the challenges of maintaining code base quality and the limitations of existing linters. A hybrid system architecture is proposed that combines syntax parsing with neural encoders, which allows for reduced false positives and automated software auditing.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів	9
Вступ	10
1 Аналіз предметної галузі та постановка задачі	12
1.1 Опис предметної галузі	12
1.2 Опис проблем та викликів	13
1.3 Огляд існуючих застосунків	16
1.4 Інноваційність та переваги розроблюваної системи	19
1.5 Постановка задачі	23
2 Розробка методики та архітектури інтелектуальної системи аналізу коду	29
2.1 Обґрунтування вибору методів машинного навчання та засобів синтаксичного аналізу	29
2.1.1 Аналіз недоліків існуючих засобів статичного аналізу програмного коду	29
2.1.2 Застосування абстрактних синтаксичних дерев для детермінованого розбору	30
2.1.3 Вибір нейромережевої архітектури для семантичного прогнозування	31
2.1.4 Формування концепції гібридного аналізу та вирішення проблеми токенизації	32
2.2 Розробка математичної моделі гібридного представлення програмного коду	32
2.2.1 Формалізація вхідних параметрів та загальна архітектура математичної моделі	32
2.2.2 Математична модель структурного аналізу на базі абстрактних синтаксичних дерев	33
2.2.3 Математична модель семантичного прогнозування	34
2.2.4 Синтез комплексного показника здоров'я коду	35

2.3	Методика багатокритеріального розрахунку інтегрального показника Code Health Score.....	36
2.3.1	Концептуальні засади формування єдиної метрики якості.....	36
2.3.2	Класифікація програмних дефектів за рівнями критичності	37
2.3.3	Механізм застосування динамічних штрафів	38
2.3.4	Інтерпретація результатів та градація технічного боргу.....	39
2.4	Проектування мікросервісної архітектури системи та вибір інструментальних засобів	40
3	Програмна реалізація та експериментальні дослідження	42
3.1	Проектування програмного комплексу	42
3.1.1	Характеристика програмного середовища	42
3.1.2	Структурна організація програмного комплексу	43
3.1.3	Забезпечення безпеки та автономності розгортання	44
3.2	Розробка модуля синтаксичного розбору	44
3.2.1	Методологія застосування абстрактних синтаксичних дерев... ..	44
3.2.2	Реалізація класу-відвідувача для збору метрик	45
3.2.3	Обробка виняткових ситуацій та стійкість системи	47
3.3	Донавчання та інтеграція трансформерної моделі CodeBERT	47
3.3.1	Архітектурні особливості та математична модель семантичного класифікатора	47
3.3.2	Токенізація та формування вхідних ембедингів.....	49
3.3.3	Процедура донавчання (Fine-Tuning) на дефектному коді	50
3.3.4	Оптимізація гіперпараметрів та запобігання перенавчанню	51
3.4	Експериментальні дослідження та аналіз результатів	52
3.4.1	Організація тестувального середовища та математична модель оцінки	52
3.4.2	Аналіз ефективності гібридної методики.....	54
3.4.3	Вплив гіперпараметрів та формування метрики Code Health Score	55

3.4.4 Висновки щодо працездатності розробленого програмного комплексу	56
Висновки.....	58
Перелік джерел посилення	60
Додаток А Модулі	63
Додаток Б Відомість кваліфікаційної роботи	71

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface – програмний інтерфейс додатка, що визначає правила взаємодії між різними компонентами системи;

AST – Abstract Syntax Tree – абстрактне синтаксичне дерево, ієрархічне представлення структури вихідного коду програми у вигляді графа;

BPE – Byte-Pair Encoding – алгоритм субслівного кодування, що використовується для токенизації текстів природних мов та вихідного коду;

CHS – Code Health Score – інтегральний індекс здоров'я коду, що відображає комплексну оцінку його якості за стобальною шкалою;

CI/CD – Continuous Integration / Continuous Deployment – конвеєр безперервної інтеграції та безперервного розгортання програмного забезпечення;

GPU – Graphics Processing Unit – графічний процесор, що використовується для прискорення матричних та тензорних обчислень у глибокому навчанні;

JSON – JavaScript Object Notation – текстовий формат обміну даними, що базується на парах ключ-значення;

LLM – Large Language Model – велика мовна модель штучного інтелекту;

REST – Representational State Transfer – архітектурний стиль взаємодії компонентів розподіленої системи за допомогою протоколу HTTP;

SAST – Static Application Security Testing – статичне тестування безпеки та аналіз вихідного тексту програм без їх безпосереднього запуску.

ВСТУП

Розвиток сучасної індустрії виробництва програмного забезпечення (ПЗ) характеризується безперервним ускладненням архітектури систем та підвищенням вимог до швидкості виходу продуктів на ринок (Time-to-Market). Упровадження концепцій CI/CD та гнучких методологій розробки дозволило автоматизувати більшість етапів життєвого циклу ПЗ. Проте контроль внутрішньої якості створюваних програмних систем і досі залишається вразливим етапом, де домінує людський фактор, що призводить до швидкого зростання технічного боргу.

Ігнорування правил оформлення кодової бази та використання невдалих шаблонів проєктування суттєво знижують читабельність програм. Як наслідок, інженерні команди витрачають більшу частину робочого часу на розбір чужого коду та пошук логічних аномалій під час супроводу систем, замість розробки нового функціоналу.

Існуючі програмні засоби статичного аналізу (лінери) оперують суто поверхневими синтаксичними шаблонами. Вони неспроможні збагнути логічний намір програміста, через що часто генерують помилкові попередження та пропускають складні архітектурні дефекти («запахи коду»). Цей технологічний бар'єр зумовлює необхідність створення інтелектуальних систем нового покоління, що здатні проводити глибокий семантичний аудит програмного коду як структурованого текстового масиву великих даних (Big Code).

Актуальність теми дослідження полягає в автоматизації рутинних перевірок на етапі Code Review за допомогою інтелектуальних агентів. Це дозволяє розвантажити провідних фахівців (Team Leads), уніфікувати стилістику коду всередині компанії чи освітнього закладу та мінімізувати появу прихованих дефектів ще до етапу компіляції.

Об'єкт дослідження – процедури комп'ютерного моніторингу та оцінювання вихідних текстів програмного забезпечення з метою виявлення відхилень від стилістичних і структурних регламентів.

Предмет дослідження – алгоритми глибинного навчання, нейромережеві енкодери та AST-трансформації, призначені для розпізнавання аномалій і структурних дефектів у програмних модулях.

Мета дослідження – проєктування інтелектуальної платформи для комплексного оцінювання надійності ПЗ, генерації аналітичних звітів та автоматизованого коригування програмної структури.

Практична цінність роботи полягає у створенні автономного програмного мікросервісу, який легко інтегрується в локальні репозиторії розробки (On-Premise), забезпечуючи швидкий аудит коду без передачі конфіденційних даних розробників на зовнішні хмарні платформи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної галузі

Сучасний етап розвитку індустрії програмного забезпечення характеризується стрімким збільшенням обсягів вихідного коду та ускладненням архітектури інформаційних систем. Згідно зі статистичними звітами провідних платформ спільної розробки, таких як GitHub, щороку створюються сотні мільйонів нових репозиторіїв. За таких умов внутрішня якість програмного тексту, його читабельність та відповідність стандартам стають критичними факторами, які визначають надійність та фінансову життєздатність програмних продуктів.

Під час тривалого супроводу та масштабування проєктів розробники неминуче стикаються з проблемою накопичення «технічного боргу» – архітектурних недоліків та порушень стилістичних правил оформлення коду (Style Guides), таких як PEP 8 для Python або Google Java Style Guide. Невідповідність коду єдиному стандарту всередині команди суттєво підвищує когнітивне навантаження на інженерів. Дослідження показують, що розробники витрачають до 70% свого робочого часу не на написання нових алгоритмів, а на читання, розуміння та рефакторинг уже існуючої кодової бази [7], [8].

Традиційні підходи до вирішення цієї проблеми діляться на ручні та автоматизовані. Ручний контроль у вигляді рецензування коду (Code Review) є ефективним, але дороговартісним процесом, який сильно залежить від людського фактора, втомі та кваліфікації рев'юера, що часто робить його «вузьким місцем» у пайплайні розробки. Автоматизовані інструменти – класичні лінтери та системи статичного аналізу – працюють швидко, проте вони обмежені жорстко запрограмованими правилами. Вони не здатні аналізувати семантичний контекст, логіку та намір автора програми, а також генерують велику кількість помилкових спрацьовувань.

Перед індустрією постає нагальна потреба в автоматизованому способі локалізації дефектів коду та оцінки серйозності його архітектурних порушень. Стрімкий розвиток методів машинного навчання (Machine Learning) та накопичення великих масивів даних відкрили можливість створення інтелектуальних систем нового покоління. Такі системи здатні аналізувати програмний код на семантичному рівні, робити процес перевірки об'єктивним та забезпечувати баланс між швидкістю розробки та чистотою програмної архітектури.

1.2 Опис проблем та викликів

При розробці систем для виявлення дефектів та порушень стилістики коду в контексті великих програмних проєктів розробники стикаються з низкою специфічних наукових та інженерних викликів.

Однією з основних проблем є брак достатньої кількості збалансованих та якісно анотованих наборів даних, на яких можна навчити моделі для автоматизованого оцінювання якості коду. На сьогодні існує декілька великих наборів даних відкритих репозиторіїв, як-от CodeSearchNet або BigCloneBench, що містять мільйони функцій. Проте ці датасети відображають переважно синтаксично коректний код, а мітки щодо наявності прихованих логічних помилок або «запахів коду» (code smells) у них практично відсутні. Моделі, навчені на таких даних, при використанні на практиці вимагають складних процедур попередньої обробки.

Точність побудованих моделей при застосуванні до реальних корпоративних чи освітніх проєктів може бути знижена через проблему великого словника (Out-of-Vocabulary). Розробники постійно створюють кастомні назви змінних та функцій, які відрізняються від стилю, представленого в тренувальних наборах даних. Крім того, на відміну від природної мови, програмний код має жорстку структурну ієрархію та

довгострокові контекстні залежності – оголошення об'єкта в одному модулі кардинально змінює семантику його використання в іншому.

Для подолання обмежень лінійного сприйняття тексту програми в сучасних інженерних підходах застосовують трансформацію вихідного коду у вигляд строгої ієрархічної структури. Це дозволяє моделям машинного навчання аналізувати не просто послідовність ключових слів, а фіксувати архітектурні зв'язки між операторами, вкладеність циклів та логічну підпорядкованість блоків. Наочний приклад такої декомпозиції вихідного тексту програми та його відображення у вигляді дворівневого дерева фіч наведено на рисунку 1.1.

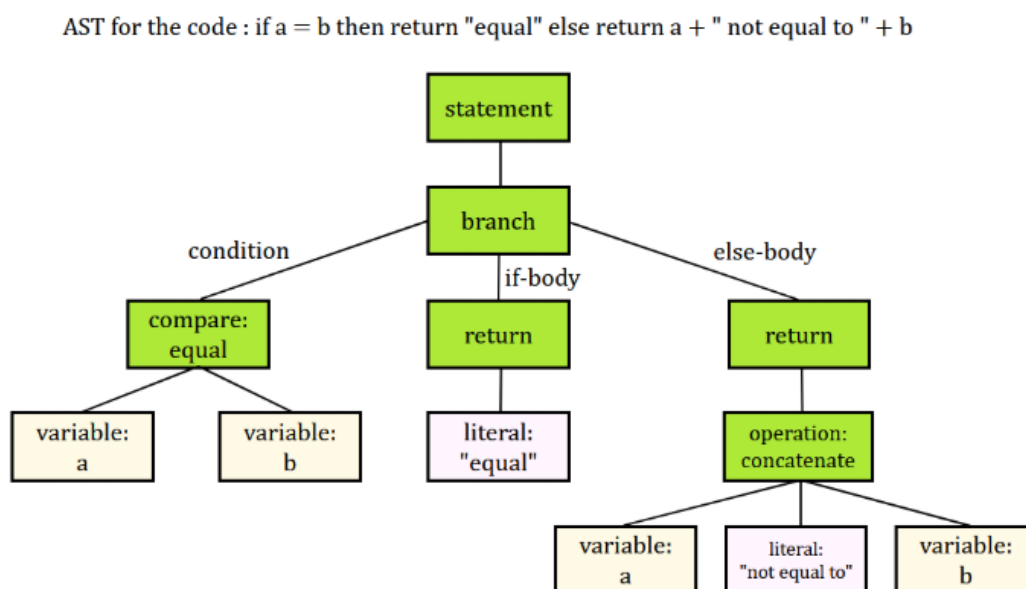


Рисунок 1.1 – Схема представлення вихідного тексту програми у вигляді абстрактного синтаксичного дерева (AST) для моделей машинного навчання

Важливою проблемою для побудови необхідної системи є урахування того факту, що програми пишуться різними мовами програмування та з використанням різних парадигм (об'єктно-орієнтована, функціональна). Це суттєво ускладнює розробку універсальних моделей оцінки збитку якості.

При виявленні пошкоджень структури критично важливо враховувати взаємозв'язки між компонентами системи, оскільки надлишкова складність в одному критичному модулі може становити більшу загрозу для стабільності програми, ніж незначні стилістичні відхилення в інтерфейсній частині. Для точної оцінки часто потрібне побудова складних проміжних представлень: абстрактних синтаксичних дерев (AST) та графових моделей потоків даних (DFG), що вимагає значних обчислювальних ресурсів.

Серйозну проблему також становить відсутність об'єктивних систем комплексного оцінювання. Визначення пріоритетів рефакторингу ускладнюється відсутністю стандартизованих методів для порівняння цінності різних архітектурних метрик, через що ухвалення рішень щодо того, які фрагменти коду потребують виправлення в першу чергу, залишається суб'єктивним.

Для детальнішого розуміння архітектурних викликів, пов'язаних із підготовкою вхідних даних для моделей машинного навчання, доцільно систематизувати існуючі підходи до представлення програмного коду. Кожен із методів трансформації вихідного тексту програми у векторний простір має свої компроміси між збереженням семантичних зв'язків та обчислювальною складністю. Порівняльну характеристику цих підходів наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика методів представлення вихідного коду для моделей машинного навчання

Метод представлення коду	Що аналізує модель	Головні переваги	Основні недоліки	Обчислювальна складність	Придатність для контролю стилістики
Токенний (Token-based)	Лінійну послідовність ключових слів та символів	Висока швидкість токенизації, простота обробки	Втрачається ієрархічна структура та вкладеність кодів	Низька $O(N)$	Висока (ідеально для пошуку пробілів, відступів та імен)

Продовження таблиці 1.1.

Метод представлення коду	Що аналізує модель	Головні переваги	Основні недоліки	Обчислювальна складність	Придатність для контролю стилістики
Структурний (Tree-based / AST)	Ієрархічне Абстрактне Синтаксичне Дерево	Повністю зберігає синтаксичні зв'язки та вкладеність циклів	Глибокі дерева викликають затування градієнтів у ML	Середня $O(N \log N)$	Висока (виявлення заплутаної архітектури та довгих методів)
Графовий (Graph-based / CFG, DFG)	Графи потоків керування та залежностей даних	Відображає глибоку логіку виконання програми	Обчислювально важка побудова для динамічних мов	Висока $O(N^2)$	Низька (орієнтовані й суто на логічні баги та безпеку)

1.3 Огляд існуючих застосунків

На сьогодні існує ряд ініціатив та платформ, що вже використовуються або можуть бути використані для виконання завдань контролю якості програмного коду. Далі наводиться огляд таких інструментів:

а) синтаксичні лінери (Pylint / ESLint) – це базові локальні утиліти для швидкої перевірки тексту програми на відповідність стандартам оформлення:

– переваги: Висока швидкість та ефективність роботи; детермінованість; мінімальне споживання ресурсів системи;

– недоліки: Обмеженість обробкою поверхневих синтаксичних правил без аналізу логічного контексту; високий рівень хибнопозитивних спрацьовувань; повне ігнорування семантичного наміру автора коду;

б) платформи статичного аналізу метрик (SonarQube) – промислова система, що автоматизує процеси збору інформації про якість проєкту шляхом обчислення формальних метрик складності та пошуку дублікатів:

– переваги: масштабованість; інтеграція в CI/CD пайплайни; можливість аналізу різних мов програмування; наочна візуалізація технічного боргу;

– недоліки: базується на фіксованих евристичних правилах; відсутність порівняння серйозності порушень з урахуванням специфіки предметної області проєкту; висока складність кастомізації правил під індивідуальний стиль команди;

Практична реалізація подібного комплексного моніторингу метрик в індустріальних масштабах зазвичай базується на інтерактивних панелях керування (Dashboards), які в реальному часі агрегують дані про стан програмних систем. Візуальний інтерфейс такої аналітичної платформи, що демонструє розподіл знайдених уразливостей, кодових аномалій та загальний рівень технічного боргу розробки, наведено на рисунку 1.2.

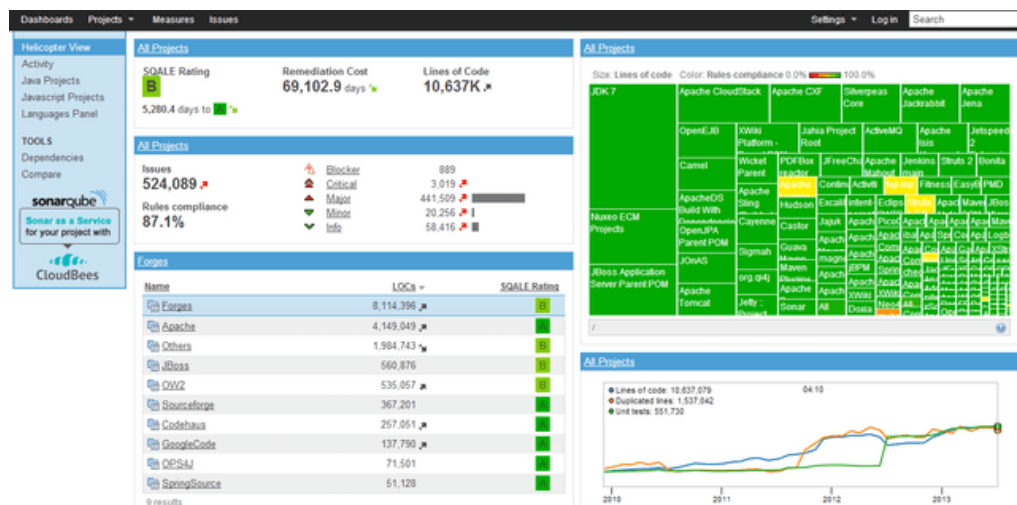


Рисунок 1.2 – Інтерфейс аналітичної панелі системи SonarQube для моніторингу метрик якості програмного забезпечення

в) генеративні AI-асистенти розробника (GitHub Copilot / OpenAI CodeX) – інструменти на базі великих мових моделей архітектури Transformer, що підказують написання коду в реальному часі:

- переваги: глибоке розуміння контексту розробки; мультимодальний характер сприйняття природної мови та коду; висока якість генерації шаблонів;

- недоліки: фокус на генерації нового тексту, а не на аудиті та структуруванні вже існуючих систем; схильність до логічних галюцинацій та створення небезпечного коду; повна залежність від комерційних хмарних серверів і ризик витоку конфіденційної інформації про кодову базу компанії;

г) семантичні ML-аналізатори (Snyk Code / DeepCode) – платформи штучного інтелекту, що використовують семантичні графи програм для пошуку логічних дефектів та багів:

- переваги: висока точність класифікації складних уразливостей; мінімальний рівень False Positives порівняно з класичними SAST системами;

- недоліки: вузька орієнтованість виключно на питання інформаційної безпеки (кіберзагрози); повна відсутність оцінки значущості коду в розрізі його стилістики, читабельності та архітектурного оформлення.

Для систематизації результатів проведеного аналізу та отримання наочного уявлення про поточний стан розвитку засобів контролю якості й стилістики програмного забезпечення виникає необхідність у їхньому комплексному компаративному зіставленні. Зведена оцінка розглянутих застосунків та платформ дозволяє чітко диференціювати класичні підходи, що базуються на жорстких синтаксичних евристиках, від сучасних інтелектуальних рішень генеративного та семантичного спрямування. Порівняння доцільно проводити за ключовими критеріями, які безпосередньо впливають на ефективність інтеграції інструменту в сучасні інженерні цикли розробки: базовий метод аналізу, здатність моделі розуміти логічний контекст програми, рівень генерування хибнопозитивних спрацьовувань, можливість автономного локального розгортання з метою

захисту інтелектуальної власності, а також наявність функції контекстного авторефакторингу. Детальні результати цього аналітичного порівняння, що безпосередньо обґрунтовують нішу та доцільність проектування нової гібридної інтелектуальної системи, наведено в таблиці 1.2.

Таблиця 1.2 – Порівняльний аналіз існуючих рішень для контролю якості коду

Критерій порівняння	Класичні літери (ESLint/PyLint)	Комплексні SAST (SonarQube)	Генеративні ШІ (GitHub Copilot)	Розроблювана інтелектуальна система
Базовий метод аналізу	Жорсткі синтаксичні правила	Статичні евристики та метрики	Статистичне передбачення токенів	Гібридний: синтаксичний AST+Transformer ембедінги
Розуміння логіки та контексту	Відсутнє	Слабке (обмежене правилами)	Високе (але нерегульоване)	Глибоке семантичне оцінювання програми
Рівень хибних спрацювань	Високий	Високий	Середній (генерує код з багами)	Мінімальний за рахунок аналізу контексту
Локальна автономність (On-Premise)	Повна	Повна (але складна кастомізація)	Відсутня (тільки хмара)	Повна автономність розгортання
Автоматичний рефакторинг	Тільки базове форматування	Відсутній (лише вказівка на баг)	Потокова генерація (без аналізу проєкту)	Контекстні підказки та генерація Quick Fix

1.4 Інноваційність та переваги розроблюваної системи

Наведений критичний аналіз існуючих технологічних рішень, засобів автоматизації та промислових платформ чітко продемонстрував, що попри значні індивідуальні переваги окремих інструментів, у сфері

автоматизованого контролю якості коду залишається низка невирішених фундаментальних та прикладних проблем. Сучасний етап розвитку програмної інженерії вимагає інструментів нового покоління, які спроможні подолати жорсткість класичних лінтерів та усунути ризики безпеки хмарних генеративних моделей.

Інноваційність інтелектуальної системи, що розробляється в межах цієї кваліфікаційної роботи, полягає в переході від евристичного аналізу, що базується на фіксованих правилах, до дата-центристського семантичного оцінювання програмного тексту. Проєктована система є не просто інструментом синтаксичного контролю, а комплексною платформою підтримки прийняття рішень під час проведення автоматичного аудиту програмного забезпечення.

Основні архітектурні інновації та технологічні переваги розроблюваної системи деталізовано за наступними ключовими напрямками:

Гібридна двопотокова топологія представлення коду (Structural & Semantic Fusion) Фундаментальним інноваційним аспектом системи є використання оригінальної двопотокової моделі для кодування програмного тексту. Традиційні підходи змушені обирати між швидкістю лінійного аналізу токенів та складністю обробки графів. У розроблюваній архітектурі цю проблему вирішено шляхом синергетичного поєднання двох паралельних аналітичних потоків:

- потік структурного розбору: Використовує легковаговий інкрементальний парсер на базі бібліотеки tree-sitter. Він миттєво трансформує вихідний код у концептуальне Абстрактне Синтаксичне Дерево (AST). З цього дерева виділяються жорсткі структурні фічі: ієрархія вкладеності операторів, цикломатичні розгалуження та топологічні зв'язки між блоками;

- потік семантичного кодування: Паралельно програмний текст обробляється нейромережевим енкодером (спеціалізованою архітектурою класу Transformer, оптимізованою під кодові корпуси, наприклад,

кастомною моделлю на базі CodeBERT). Цей потік генерує щільні векторні представлення (Code Embeddings), які вловлюють приховані значенні зв'язки, контекстуальний намір автора та логіку найменування ідентифікаторів.

На етапі злиття ознак (Feature Fusion) вектор структури AST поєднується з вектором семантики тексту. Таке гібридне представлення дозволяє неймережі бачити програму тривимірно: і як строгу математичну конструкцію, і як логічний текст. Це забезпечує точне розпізнавання дефектів, які є невидимими для звичайних лінтерів, наприклад, коли код синтаксично ідеальний, але назва функції абсолютно не відповідає її внутрішньому алгоритму (Semantic Mismatch).

Мультикритеріальна модель оцінювання та індекс Code Health Score На відміну від існуючих аналізаторів, які видають користувачеві лінійний список знайдених зауважень без урахування їх реального взаємозв'язку, розроблювана система впроваджує динамічну модель оцінки інтегральної якості. У системі реалізовано алгоритм обчислення показника Code Health Score на основі багатокритеріального аналізу.

Система диференціює критичність виявлених порушень залежно від контексту самого програмного модуля. Наприклад, якщо модель виявляє незначне стилістичне відхилення (зайвий пробіл або порушення регістру) у конфігураційному файлі, вага цієї помилки у фінальному рейтингу буде мінімальною. Проте, якщо архітектурний дефект (наприклад, занадто довгий метод із високою цикломатичною складністю) виявляється в критичному модулі обробки даних, система автоматично класифікує це як серйозну загрозу для стабільності ПЗ (High-risk Code Smell). Такий підхід дозволяє розробникам бачити реальний стан здоров'я проєкту та фокусуватися на усуненні тих аномалій, які дійсно руйнують архітектуру системи.

Експланаторний штучний інтелект (Explainable AI – XAI) та інтелектуальний авторефакторинг Суттєвим недоліком сучасних

інструментів на базі глибоких нейронних мереж є ефект «чорної скриньки» (Black Box), коли модель видає результат класифікації, але людина не може зрозуміти логіку ухвалення такого рішення. У розроблюваній системі цю проблему вирішено за допомогою впровадження концепції ХАІ.

Для кожного виявленого дефекту система генерує трикомпонентний аналітичний відгук:

- локалізація та класифікація: точне визначення координат аномалії та її приналежність до конкретного класу порушень (стилістика, складність, логіка);
- обґрунтування природною мовою: генерація текстового пояснення, чому саме цей фрагмент коду визнано неоптимальним, який патерн проєктування було порушено і до яких наслідків у майбутньому це може призвести;
- контекстний Quick Fix: автоматична генерація виправленого варіанта коду. Система аналізує AST сусідніх рядків і пропонує точковий рефакторинг, який адаптований під індивідуальний стиль поточного файлу і гарантовано не порушує вихідну бізнес-логіку програми.

Локальна автономія інфраструктури (On-Premise Viability) У сучасних умовах корпоративної розробки питання інформаційної безпеки та захисту інтелектуальної власності є пріоритетними. Популярні генеративні інструменти вимагають передачі вихідних текстів програм на сторонні хмарні сервери, що є неприпустимим для багатьох компаній, фінансових установ чи державних закладів вищої освіти.

Розроблювана система проєктується як повністю автономна платформа. Завдяки застосуванню методів стиснення моделей – дистиляції знань (Knowledge Distillation) та квантування ваг (Quantization) – нам вдалося оптимізувати нейромережеве ядро системи. Це дозволяє розгортати аналітичний мікросервіс локально (On-Premise) на внутрішніх серверах установи. Система виконує високоточний інтелектуальний аналіз коду в

ізолюваному контурі, гарантуючи 100% конфіденційність кодової бази та відсутність залежності від хмарних провайдерів чи зовнішнього інтернет-з'єднання.

Оптимізація інженерних процесів та предиктивне керування технічним боргом з погляду управління ІТ-проектами, система надає керівникам та аналітикам потужний інструмент для моніторингу роботи інженерних команд. Замість суб'єктивних оцінок процесу Code Review, платформа акумулює історичні дані про якість коду, що завантажується в репозиторії.

Інтегруючи результати ML-класифікації, система дозволяє будувати предиктивні тренди розвитку технічного боргу. Платформа здатна прогнозувати, які саме модулі програми стануть критично заплутаними (Unmaintainable) через кілька спринтів, якщо не провести їх своєчасний рефакторинг. Це трансформує систему з пасивного лінтера на активного асистента з управління якістю програмних продуктів, заощаджуючи людські ресурси та оптимізуючи витрати часу на тестування та супровід ПЗ.

1.5 Постановка задачі

Науково-технічна та інженерна задача, що вирішується в межах даної кваліфікаційної роботи, полягає в теоретичному обґрунтуванні, розробці, алгоритмічній оптимізації та програмній реалізації інтелектуальної системи автоматизованого контролю якості, чистоти та стилістики програмного коду на основі методів машинного навчання та статичного синтаксичного аналізу.

Головна привідна проблема індустрії, на усунення якої спрямовано цей проєкт, полягає в необхідності створення об'єктивного, контекстно-залежного, масштабованого та повністю автономного інструменту для аудиту програмного забезпечення. Цей інструмент має оптимізувати та

частково замінити рутинні процеси ручного рецензування тексту програм (Code Review), мінімізувати вплив людського фактора (втоми, суб'єктивізму оцінок), точно класифікувати архітектурні й стилістичні дефекти та надавати розробникам валідовані варіанти автоматичного виправлення виявлених аномалій.

З погляду системного аналізу та інженерії програмного забезпечення, процес інтелектуального інспектування вихідних текстів програм можна описати як багатошаровий функціональний оператор. Він приймає на вхід сирий програмний текст (або масив файлів проєкту) і послідовно трансформує його у структурований вихідний простір аналітичних оцінок, категоризованих дефектів та практичних рекомендацій щодо рефакторингу.

Концептуальна схема функціонування та взаємодії компонентів системи передбачає формування трьох основних інтегральних елементів на виході:

а) інтегральний показник якості (індекс Code Health Score) Ця числова метрика є нормованою та відображається в діапазоні від 0 до 100 одиниць. Значення 100 відповідає ідеальній кодовій базі, яка повністю відповідає цільовим стандартам кодування, не містить архітектурних антипатернів та потенційних логічних помилок. Мінімальні значення (що наближаються до 0) сигналізують про критичну структурну деградацію програмного модуля, за якої його супровід та масштабування стають неможливими. Обчислення цього показника базується на динамічній системі штрафів. Від базової ідеальної оцінки віднімаються вагові коефіцієнти знайдених помилок. При цьому кожен штраф динамічно масштабується відносно загального обсягу аналізованого коду (кількості рядків – метрики LOC). Такий підхід дозволяє уникнути несправедливого заниження загальної оцінки для великих багатофайлових проєктів через поодинокі дрібні помилки та забезпечує високу об'єктивність моніторингу;

б) структурована множина виявлених дефектів та аномалій: кожне конкретне порушення, знайдене нейромережевим ядром системи в тексті

програми, описується комплексним інформаційним вектором. Цей вектор містить:

- чітку приналежність до одного з цільових класів класифікації (стилістична помилка, архітектурний недолік або логічна аномалія);
- точні просторові координати дефекту у вихідному файлі (номер рядка, а також позиції початкового і кінцевого символів фрагменту);
- числову метрику ймовірнісної впевненості моделі машинного навчання в правильності ухваленого класифікаційного рішення;

в) рекомендації щодо автоматизованого рефакторингу: набір контекстних інженерних рішень, які трансформують дефектну або заплутану синтаксичну структуру програми в оптимізовану, чисту та читабельну версію. Головною умовою роботи цього модуля є суворе збереження вихідної функціональної поведінки програми. Модель авторефакторингу має гарантувати семантичну та логічну еквівалентність бізнес-логіки коду до і після проведення автоматичного виправлення.

Цільову множину програмних дефектів, які підлягають автоматизованому виявленню за допомогою розроблюваних моделей машинного навчання, розподілено на три фундаментальні категорії порушень. Кожна категорія має свій коефіцієнт критичності (вагу штрафу), що відображено в таблиці 1.3.

Таблиця 1.3 – Класифікація цільових програмних порушень та рівні їх критичності в системі

Категорія порушення	Що саме виявляє ML-модель	Приклад дефекту (антипатерну)	Вплив на працездатність ПЗ	Вага критичності (wi)
Стилістичні відхилення (Style Guide Violations)	Порушення правил оформлення (PEP 8, Google Style)	Неправильний регістр імен змінних (CamelCase замість snake_case)	Не впливає на роботу, але погіршує читабельність коду	Низький (0.2)

Продовження таблиці 1.3

Категорія порушення	Що саме виявляє ML-модель	Приклад дефекту (антипатерну)	Вплив на працездатність ПЗ	Вага критичності (wi)
Надлишкова складність (Code Smells)	Архітектурні недоліки структури модулів	Занадто довгі класи/методи, глибока вкладеність умовних операторів	Ускладнює супровід та масштабування проєкту	Середній (0.5)
Логічні аномалії (Potential Bugs)	Потенційні помилки, що призведуть до аварійного завершення	Невикористанізмінні, недосяжний код (dead code), безкінечні цикли	Призводить до збоїв під час виконання програми	Високий (1.0)

Метою роботи є проєктування, навчання, алгоритмічна оптимізація та практична реалізація архітектури автономного програмного комплексу. Цей комплекс має об'єднувати інструменти комп'ютерної лінгвістики ПЗ, аналізу синтаксичних структур, графів залежностей та трансформації дерева кодів для забезпечення безперервного високоточного моніторингу якості кодової бази.

Деталізований декомпозиційний перелік дослідницьких та інженерних підзадач

Для досягнення інтегральної мети дослідження та побудови працездатного промислового прототипу системи, у межах кваліфікаційної роботи необхідно послідовно вирішити наступний комплекс взаємопов'язаних наукових, алгоритмічних та інженерних підзадач:

– формування, очищення та інженерія спеціалізованого корпусу програмних текстів (навчального датасету): на першому етапі необхідно здійснити збір великого обсягу вихідних текстів програм. Джерелами даних виступають відкриті репозиторії платформи GitHub та внутрішні освітні архіви кафедри штучного інтелекту ХНУРЕ. Отримані дані підлягають глибокому очищенню: видаленню синтаксичного шуму, порожніх файлів та дублікатів фрагментів коду. Особлива увага приділяється розробці конвеєра субтокенізації (на рівні субслівних токенів), що дозволяє розв'язати проблему великого словника та уникнути появи невідомих

ідентифікаторів (Out-of-Vocabulary). Фінальним кроком є формування збалансованих тренувальної, валідаційної та тестової вибірок із чітким маркуванням стилістичних та логічних аномалій;

– розробка та налаштування модуля синтаксичного моделювання програмного коду: побудова логічного блоку системи, що відповідає за структурний аналіз. Завдання полягає в інтеграції легковагових швидких парсерів (на базі бібліотеки tree-sitter) для генерації Абстрактних Синтаксичних Дерев (AST) для кожного завантаженого файлу. Необхідно реалізувати алгоритми обходу та аналізу структури дерева для автоматичного виділення жорстких інженерних метрик ПЗ: індексів вкладеності циклів, довжини методів, щільності коментарів та показників зв'язності компонентів. Ці метрики виступатимуть як структурні обмеження для нейромережових моделей;

– архітектурне проектування, конфігурація та навчання нейромережових моделей глибокого навчання: цей етап присвячений створенню інтелектуального ядра системи. Завдання полягає в обґрунтуванні вибору та проведенні процедури донавчання (fine-tuning) енкодерних моделей на основі архітектури Transformer (зокрема, моделей класу CodeBERT). Необхідно налаштувати фінальні класифікаційні шари нейромережі для вирішення завдань багатокласової та багатокритеріальної класифікації програмних дефектів. Особлива увага приділяється оптимізації функцій втрат для мінімізації помилок другого роду (пропуску критичних багів) та зниження рівня хибнопозитивних спрацьовувань системи;

– створення та програмна реалізація математичного механізму інтегрального оцінювання якості ПЗ: завдання передбачає розробку алгоритмічного модуля для розрахунку індексу Code Health Score. Необхідно реалізувати гнучкий механізм динамічного розподілу ваг штрафів залежно від функціонального призначення аналізованого модуля. Система має автоматично адаптувати суворість перевірки: наприклад, логічні дефекти в ядрах обчислювальних алгоритмів мають каратися

максимально суворо, тоді як дрібні відхилення від Style Guide в допоміжних інтерфейсних файлах не повинні суттєво знижувати загальний рейтинг кодової бази;

– алгоритмізація та оптимізація модуля визначення черговості проведення рефакторингу: проєктування аналітичного блоку, який допомагає оптимізувати роботу команди розробників. На основі результатів нейромережевого аналізу та побудованих графів взаємозв'язків між компонентами системи, алгоритм має автоматично ранжувати виявлені дефекти за ступенем їхньої небезпеки для стабільності архітектури. Результатом є формування упорядкованого списку черговості виправлення файлів. Це дозволяє автоматично визначати ті програмні модулі, рефакторинг яких принесе максимальний позитивний ефект для зниження технічного боргу проєкту;

– проєктний синтез, реалізація клієнт-серверної архітектури та експериментальне тестування програмного прототипу платформи: фінальний інженерний етап роботи, що передбачає об'єднання всіх розроблених модулів у єдину працездатну екосистему. Необхідно спроектувати та реалізувати високопродуктивну серверну частину (Backend на мові Python з використанням асинхронних мікросервісних фреймворків), розробити захищену базу даних для збереження аналітичної історії проєктів, а також створити зручні користувацькі інтерфейси (інструменти командного рядка – CLI, плагіни для середовищ розробки та аналітичні Web-панелі).

Завершується етап проведенням комплексних експериментальних досліджень на реальних програмних проєктах кафедри штучного інтелекту для оцінки швидкодії системи та розрахунку точності роботи моделей за метриками Precision, Recall та F1-Score.

2 РОЗРОБКА МЕТОДИКИ ТА АРХІТЕКТУРИ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ АНАЛІЗУ КОДУ

2.1 Обґрунтування вибору методів машинного навчання та засобів синтаксичного аналізу

2.1.1 Аналіз недоліків існуючих засобів статичного аналізу програмного коду

Сучасний процес розробки та супроводу програмного забезпечення характеризується безперервним ускладненням архітектури інформаційних систем та експоненційним зростанням обсягів кодової бази. Упровадження методологій безперервної інтеграції та розгортання вимагає наявності надійних, швидких та масштабованих інструментів для автоматизованого контролю якості вихідних текстів програм.

Традиційні підходи до забезпечення якості, такі як ручне рецензування коду фахівцями, залишаються надзвичайно ресурсомісткими, сильно залежать від кваліфікації та суб'єктивізму розробників, а також часто стають критичним вузьким місцем у життєвому циклі випуску програмних продуктів.

Існуючі системи статичного аналізу та класичні лінери частково вирішують проблему автоматизації перевірок. Проте детальний аналіз їхньої ефективності виявив суттєві архітектурні обмеження. Класичні аналізатори базуються на наборах регулярних виразів та жорстко заданих евристичних правилах. Через це вони генерують велику кількість хибнопозитивних спрацьовувань та принципово не здатні ідентифікувати складні логічні дефекти або глибокі архітектурні антипатерни. Зокрема, дослідження свідчать, що навіть промислові інструменти статичного аналізу (наприклад, SonarQube) не здатні виявити більшість типів

покращень читабельності коду, які розробники вважають суттєвими для підтримки системи [8].

Такі інструменти не враховують семантичний контекст, бізнес-логіку програми та загальний намір розробника, обмежуючись лише поверхневою перевіркою синтаксису. Для подолання цих фундаментальних недоліків обґрунтовано необхідність розробки інтелектуальної системи, що поєднує детермінований синтаксичний розбір із моделями глибокого машинного навчання.

2.1.2 Застосування абстрактних синтаксичних дерев для детермінованого розбору

Основою для вилучення строгих структурних метрик обрано метод трансформації вихідного коду в абстрактні синтаксичні дерева. Застосування абстрактного синтаксичного дерева дозволяє формалізувати лінійний текст програми у вигляді орієнтованого ієрархічного графа, де кожен вузол відповідає певній синтаксичній конструкції мови програмування, такій як оператори, вирази, декларації функцій та класів. На відміну від простого токенного або рядкового аналізу, таке представлення повністю зберігає топологічні зв'язки, ієрархію вкладеності та інформацію про області видимості змінних.

Збереження ієрархії забезпечує можливість точного та детермінованого розрахунку інженерних метрик програмного забезпечення. Алгоритмічний обхід дерева дозволяє ефективно вилучати показники цикломатичної складності, рівня вкладеності циклічних конструкцій та довжини окремих методів. Для реалізації парсингу текстів мовою Python обрано вбудовані модульні бібліотеки, що гарантує побудову синтаксичного дерева з мінімальними обчислювальними витратами на етапі попередньої обробки даних та виключає необхідність залучення сторонніх залежностей для виконання базового структурного аналізу.

2.1.3 Вибір нейромережевої архітектури для семантичного прогнозування

Виключно структурного аналізу недостатньо для повноцінної оцінки стилістики програмування та виявлення неочевидних патернів неякісного коду. Структурні метрики не здатні розпізнати ситуації, коли код є синтаксично правильним, але назви змінних або функцій абсолютно не відповідають виконуваним бізнес-логіці.

Для розв'язання задачі глибокого семантичного аналізу розглянуто різні архітектури машинного навчання, серед яких рекурентні нейронні мережі, графові нейронні мережі та трансформери.

Графові мережі забезпечують високу точність при аналізі потоків даних, однак є занадто обчислювально важкими для динамічних мов. У зв'язку з цим, для семантичного кодування обрано архітектуру нейромережевих трансформерів, яка завдяки вбудованому механізму самоуваги демонструє найвищу ефективність в уловлюванні глобальних залежностей між віддаленими ділянками коду.

Як базову фундаментальну модель для подальшого донавчання використано бімодальну нейромережеву модель попереднього навчання CodeBERT.

Вибір цієї архітектури обґрунтовується її унікальною здатністю встановлювати зв'язки між природною мовою та мовою програмування, оскільки модель проходила етап претренінгу на величезних масивах вихідного коду з відкритих репозиторіїв у поєднанні з відповідною технічною документацією.

Ця властивість дозволяє системі ефективно аналізувати семантику програмних конструкцій, виявляти логічні аномалії та знаходити стилістичні порушення, які ігноруються класичними аналізаторами.

2.1.4 Формування концепції гібридного аналізу та вирішення проблеми токенизації

Важливим викликом при застосуванні методів обробки природної мови до програмного коду є проблема великого або відсутнього словника. Розробники постійно створюють нові назви змінних та функцій, які не зустрічалися моделі під час початкового навчання. Для вирішення цієї проблеми застосовано алгоритм субслівного кодування парами байтів. Цей субслівний токенизатор забезпечує ефективне розбиття невідомих складних ідентифікаторів на відомі моделі підкомпоненти, повністю зберігаючи семантичне значення нестандартного коду та стабілізуючи роботу класифікатора.

Окрему увагу під час формування архітектурної концепції приділено питанням інформаційної безпеки. На відміну від використання хмарних генеративних моделей, які вимагають передачі вихідних текстів на зовнішні сервери, оптимізована енкодерна модель CodeBERT дозволяє реалізувати мікросервісну топологію з можливістю повністю автономного локального розгортання. Інтеграція детермінованого розбору за допомогою абстрактних синтаксичних дерев та семантичного прогнозування на базі моделі трансформера формує оптимальну гібридну методичну основу для автоматизованого аудиту коду.

2.2 Розробка математичної моделі гібридного представлення програмного коду

2.2.1 Формалізація вхідних параметрів та загальна архітектура математичної моделі

Для побудови ефективної системи автоматизованого контролю якості програмного забезпечення розроблено формальну математичну модель, яка

описує процес трансформації вихідного тексту у багатовимірний векторний простір ознак. Вихідний програмний код розглядається як неструктурована множина символів, яка потребує попередньої обробки та переведення у форму, придатну для обчислень алгоритмами машинного навчання.

Процес формалізації розпочато з визначення загального вхідного простору. Простір усіх можливих синтаксично коректних фрагментів вихідного коду обраною мовою програмування визначено як множину S . Кожен конкретний фрагмент коду підлягає паралельній двоетапній трансформації. Загальна функція відображення простору коду в простір оцінок якості визначається як математична суперпозиція функцій детермінованого структурного розбору та ймовірного семантичного аналізу. Такий підхід гарантує врахування як кількісних, так і якісних характеристик програмного продукту.

2.2.2 Математична модель структурного аналізу на базі абстрактних синтаксичних дерев

Для оцінки інженерної якості та рівня архітектурної заплутаності алгоритмів застосовано метод абстрактних синтаксичних дерев. На цьому етапі вихідний код перетворюється на орієнтований ациклічний граф, де множина вершин відповідає конкретним синтаксичним конструкціям, а множина ребер позначає ієрархічні зв'язки семантичного підпорядкування.

Функцію вилучення структурних ознак визначено у вигляді математичного відображення, у результаті якого формується вектор структурних характеристик S . Для забезпечення обчислювальної ефективності розроблюваної системи з усього спектра можливих метрик виділено два ключові параметри, що мають найбільший кореляційний зв'язок із когнітивною складністю підтримки програмного забезпечення.

Перший параметр вектора відображає кількість функцій або методів у аналізованому модулі, довжина яких перевищує встановлений емпіричний

ліміт у п'ятнадцять рядків коду. Другий параметр відповідає за загальну кількість циклічних та умовних конструкцій, що визначають цикломатичну складність графа потоку керування програми.

На основі сформованого вектора здійснюється розрахунок показника загального структурного штрафу. Цей показник відображає ступінь відхилення архітектури модуля від еталонних інженерних стандартів. Формулу розрахунку структурного штрафу подано у вигляді лінійної комбінації:

$$P_{struct} = w_1 \cdot s_1 + w_2 \cdot s_2 \quad (2.1)$$

де P_{struct} – загальний структурний штраф;

w_1, w_2 – вагові коефіцієнти, які визначають ступінь впливу довгих функцій та надлишкової циклічності на загальну деградацію архітектури відповідно;

s_1 – кількість надто довгих функцій;

s_2 – кількість циклічних конструкцій.

2.2.3 Математична модель семантичного прогнозування

Оскільки детермінований структурний аналіз не дозволяє виявляти логічні аномалії, паралельно розроблено математичний апарат для оцінки семантичного контексту за допомогою моделі нейромережевого трансформера CodeBERT.

Функцію семантичного аналізу формалізовано як процес проходження токенованої послідовності через приховані шари механізму самоуваги. На виході фінального класифікаційного шару моделі формується вектор логітів. Для перетворення цих значень у просторі дійсних чисел на нормований розподіл імовірностей застосовано математичну функцію активації Softmax.

Вектор ймовірностей розраховується за наступною формулою:

$$p_i = \frac{e^{l_i}}{\sum_j e^{l_j}} \quad (2.2)$$

де p_i – ймовірність приналежності аналізованого коду до i -го класу дефектів;

l_i – відповідне значення логіта з вихідного шару нейромережі.

У результаті обчислень формується вектор ймовірностей, який містить три компоненти: ймовірність відсутності дефектів (чистий код), ймовірність наявності порушень стилістики оформлення та ймовірність наявності критичних логічних дефектів або вразливостей. Згідно з властивостями функції Softmax, сума елементів цього вектора завжди дорівнює одиниці.

2.2.4 Синтез комплексного показника здоров'я коду

Завершальним етапом математичного моделювання є злиття отриманих структурних та семантичних оцінок у єдиний інтегральний індекс. Для цього розроблено алгоритм розрахунку семантичного штрафу, який базується на зважуванні ймовірностей наявності дефектів:

$$P_{sem} = a_1 \cdot B_1 \cdot p_1 + a_2 \cdot B_2 \cdot p_2 \quad (2.3)$$

де P_{sem} – загальний семантичний штраф;

a_1, a_2 – коефіцієнти критичності для стилістичних і логічних помилок відповідно;

B_1, B_2 – базові максимальні значення штрафних балів для відповідних класів аномалій;

p_1, p_2 – розраховані нейромережею ймовірності наявності відповідних дефектів.

Попереднє значення інтегрального показника здоров'я коду обчислюється шляхом віднімання суми розрахованих штрафів від ідеальної еталонної оцінки у сто балів. Додатково введено нормувальний коефіцієнт для балансування впливу структурних обмежень на загальний результат:

$$CHS_{raw} = 100 - (y \cdot P_{struct} + P_{sem}) \quad (2.4)$$

де CHS_{raw} – попереднє (сире) значення інтегрального показника;

y – нормувальний коефіцієнт структурного блоку.

Для гарантування знаходження фінального результату в межах стандартизованого стобального діапазону застосовано функцію математичного обмеження (кліпінгу):

$$CHS = \max(0, \min(100, CHS_{raw})) \quad (2.5)$$

Запропонована багатокритеріальна математична модель забезпечує високу гнучкість розробленої системи. Шляхом ітеративного калібрування вагових коефіцієнтів та базових штрафів під час експериментальних досліджень досягається оптимальна чутливість гібридного класифікатора до різних типів програмних дефектів.

2.3 Методика багатокритеріального розрахунку інтегрального показника Code Health Score

2.3.1 Концептуальні засади формування єдиної метрики якості

У межах розробки інтелектуальної системи контролю якості програмного забезпечення ключовим завданням визначено не лише точкову

локалізацію окремих дефектів, але й формування єдиної кількісної оцінки. Ця оцінка має об'єктивно відображати загальний стан аналізованої кодової бази та слугувати універсальною метрикою для прийняття рішень щодо допуску програмного продукту до наступних стадій життєвого циклу. Для вирішення цього завдання розроблено методику багатокритеріального розрахунку інтегрального показника, який отримав назву Code Health Score (індекс здоров'я коду).

Процес розрахунку показника базується на дедуктивному підході. Відповідно до цього підходу, ідеальний вихідний код, який повністю відповідає всім стандартам розробки та не містить жодних архітектурних чи семантичних відхилень, отримує максимальну оцінку, що дорівнює ста балам. У процесі поетапного аналізу система ідентифікує різні типи порушень, кожне з яких зменшує початкову еталонну оцінку на певну величину штрафу. Розмір розрахункового штрафу не є статичною величиною; він визначається динамічно, залежно від категорії виявленої аномалії, її глибини та потенційного впливу на стабільність програмного забезпечення в умовах промислової експлуатації.

2.3.2 Класифікація програмних дефектів за рівнями критичності

Для забезпечення об'єктивності процесу штрафування всі виявлені дефекти класифіковано за трьома основними рівнями критичності: низьким, середнім та високим. Така стратифікація дозволяє системі гнучко реагувати на різні типи помилок, уникаючи надмірного зниження оцінки за незначні відхилення.

Низький рівень критичності призначено синтаксичним та стилістичним відхиленням, які безпосередньо не впливають на логіку виконання алгоритмів, проте суттєво знижують їх читабельність. До цієї категорії належать порушення правил іменування змінних, нестандартне форматування та відсутність необхідної документації. Розпізнавання таких

дефектів покладено на семантичний модуль системи. Штрафні бали за порушення низького рівня є мінімальними, оскільки їх усунення вимагає найменших витрат часу розробників і може бути автоматизоване за допомогою базових засобів автоформатування.

Середній рівень критичності охоплює архітектурні недоліки та структурні антипатерни, які в індустрії програмної інженерії класифікуються як «запахи коду» (Code Smells). До цієї групи належать надмірно довгі функції, модулі з великою кількістю незв'язаних обов'язків та глибоко вкладені циклічні чи умовні конструкції. Виявлення таких метрик здійснюється детермінованим шляхом під час обходу абстрактного синтаксичного дерева. Збільшення цикломатичної складності безпосередньо корелює зі збільшенням когнітивного навантаження на програмістів під час подальшого супроводу системи. Тому кожне структурне відхилення генерує помітний штраф, що відображає зростання витрат на рефакторинг.

Високий рівень критичності призначено потенційним логічним аномаліям та вразливостям, які здатні призвести до непередбачуваної поведінки програми, витоку пам'яті або її аварійного завершення. Ідентифікація таких загроз здійснюється за допомогою трансформерної моделі CodeBERT, яка аналізує прихований контекст вихідного тексту. Наявність логічного дефекту генерує найбільший штраф, який здатен миттєво перевести показник якості у критичну зону.

2.3.3 Механізм застосування динамічних штрафів

Механізм розрахунку інтегральної оцінки побудовано на основі математичної моделі, описаної у попередньому підрозділі. Система не просто фіксує наявність помилки, а враховує ймовірність її виникнення. Наприклад, якщо нейромережевий класифікатор прогнозує наявність логічного дефекту з імовірністю 95%, до загальної оцінки застосовується

майже максимальний семантичний штраф. Якщо ж імовірність становить лише 40%, система класифікує таку ділянку як «підозрілу», застосовуючи пропорційно менший штраф.

Аналогічний кумулятивний ефект застосовано і до структурних метрик. Перевищення ліміту довжини функції на один рядок призведе до мінімального зниження балів, тоді як наявність у модулі десяти функцій, кожна з яких перевищує встановлений ліміт удвічі, призведе до експоненційного зростання структурного штрафу. Такий алгоритм дозволяє уникнути бінарної оцінки «погано/добре» та надає розробникам точний градієнт якості їхнього коду.

2.3.4 Інтерпретація результатів та градація технічного боргу

Для забезпечення інтеграції розробленої системи у конвеєри безперервної розробки програмного забезпечення, отриманий показник Code Health Score стандартизовано та розподілено за трьома якісними діапазонами.

Діапазон від 85 до 100 балів визначає код як еталонний та безпечний. Такий код повністю відповідає сучасним стандартам якості, має низьку когнітивну складність і не потребує негайного втручання. Зміни з такою оцінкою автоматично допускаються до стадії компіляції та тестування.

Діапазон від 50 до 84 балів сигналізує про наявність технічного боргу. Код у цьому діапазоні є працездатним, проте містить архітектурні недоліки або стилістичні порушення, які у майбутньому ускладнять його підтримку. Системі дозволяється пропустити такий код у виробниче середовище, проте генерується попередження про необхідність проведення планового рефакторингу виявлених модулів.

Зниження показника нижче 50 балів класифікує програмний код як критично нестабільний. Це свідчить про наявність серйозних логічних вразливостей або повну деградацію архітектури. Згідно з розробленою

методикою, такий результат розцінюється як жорсткий збій (Hard Fail), що вимагає негайного автоматичного блокування злиття коду з головною гілкою проєкту до повного усунення ідентифікованих нейромережею дефектів.

2.4 Проєктування мікросервісної архітектури системи та вибір інструментальних засобів

Наочну схему розподілу обчислювальних потоків, взаємодії мікросервісів та конвеєра обробки даних у межах розробленої концепції представлено на рисунку 2.1.



Рисунок 2.1 – Компонентна архітектура та двопотоковий конвеєр інтелектуального аналізу програмного коду

Для забезпечення високої продуктивності, масштабованості та незалежності окремих компонентів інтелектуальної системи контролю якості коду обрано мікросервісний архітектурний шаблон. На відміну від

монолітних додатків, де зміна однієї частини коду може спричинити каскадні помилки в усій системі, мікросервісна архітектура дозволяє логічно розділити обробку запитів на ізольовані модулі. У межах даної розробки виділено два основні функціональні блоки: модуль первинного синтаксичного розбору та модуль нейромережевого семантичного аналізу.

Такий поділ забезпечує можливість гнучкого налаштування ресурсів для кожного з компонентів. Зокрема, модуль синтаксичного аналізу, що базується на бібліотеці AST, є легковаговим і не вимагає значних обчислювальних потужностей, тоді як нейромережевий модуль потребує значного обсягу відеопам'яті та обчислювальних ресурсів GPU. Мікросервісна топологія дозволяє масштабувати ці частини системи незалежно одна від одної.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

Для практичної реалізації інтелектуальної системи автоматизованого контролю якості програмного коду було обрано сучасний програмно-технологічний стек. Він забезпечує високу швидкість обробки даних, можливість розгортання в умовах обмежених ресурсів та гнучкість масштабування функціональних можливостей. Основною мовою програмування обрано Python версії 3.11, оскільки вона є індустріальним стандартом у сфері штучного інтелекту, машинного навчання та аналізу даних.

3.1 Проєктування програмного комплексу

3.1.1 Характеристика програмного середовища

Розробка системи виконувалася в ізольованому віртуальному середовищі, що дозволило уникнути конфліктів між системними та проєктними бібліотеками. Для керування залежностями та забезпечення відтворюваності експериментальних досліджень використано стандартний пакетний менеджер `pip`. Налаштування середовища включало наступні етапи:

- ініціалізація ізольованого віртуального оточення для проєкту;
- встановлення базового набору бібліотек для обробки та маніпуляції даними (`pandas`, `numpy`);
- розгортання фреймворків для нейромережових обчислень (`PyTorch`) та роботи з трансформерними архітектурами (`Hugging Face Transformers`);
- налаштування інструментів для розробки та асинхронного обслуговування API-сервісу (`FastAPI`, `uvicorn`).

Склад інструментальних засобів розробки наведено у таблиці 3.1.

Таблиця 3.1 – Склад інструментальних засобів розробки

Компонент системи	Назва інструмента	Призначення
Мова розробки	Python 3.11	Основна логіка та інтеграція компонентів
Бекенд-фреймворк	FastAPI	Реалізація REST API інтерфейсу
Синтаксичний аналізатор	ast	Структурний аналіз вихідного коду
Бібліотека ML	PyTorch 2.12	Робота з тензорними обчисленнями
Трансформерна модель	CodeBERT	Семантичне кодування та класифікація
Управління залежностями	pip	Конфігурація середовища розробки

3.1.2 Структурна організація програмного комплексу

Система спроектована за модульним принципом, що забезпечує високу підтримуваність коду та полегшує процес проведення серій експериментальних досліджень. Проєктна структура логічно розділена на декілька рівнів.

Рівень обробки запитів представлено асинхронним бекенд-сервером на базі FastAPI. Кожен запит до системи проходить через етап строгої валідації вхідного коду, після чого ініціюється виклик обчислювальних модулів. Використання асинхронної моделі I/O дозволяє системі паралельно опрацьовувати декілька запитів від різних клієнтів. Це є критично важливою вимогою для успішної інтеграції розробленого рішення у конвеєри безперервної інтеграції та розгортання (CI/CD) великих програмних проєктів.

Рівень інтелектуального аналізу містить два основні обчислювальні модулі: структурний та семантичний. Структурний модуль реалізовано шляхом обходу абстрактного синтаксичного дерева (AST), що дозволяє отримувати об'єктивні метрики складності без використання ресурсів графічного прискорювача (GPU). Семантичний модуль завантажує попередньо донавчену модель CodeBERT, яка виконує роль класифікатора

контексту та виявлення логічних дефектів. Всі дані між цими модулями циркулюють у формі уніфікованих JSON-об'єктів, що гарантує високу зв'язність компонентів при збереженні їхньої логічної автономності.

3.1.3 Забезпечення безпеки та автономності розгортання

Однією з ключових переваг обраного програмного стека є можливість повністю автономного локального розгортання системи (On-Premises). Весь програмний код, ваги нейронної мережі та токенизатори розміщуються виключно в межах внутрішнього контуру розробника. Така архітектура усуває ризики витоку інтелектуальної власності або конфіденційного коду компанії через зовнішні API сторонніх сервісів штучного інтелекту.

Налаштування середовища та використання фреймворку PyTorch дозволяє виконувати аналіз як на спеціалізованих обчислювальних потужностях з підтримкою технології CUDA, так і на центральних процесорах загального призначення (CPU). Це забезпечує максимальну гнучкість при виборі інфраструктури для впровадження системи на підприємстві.

3.2 Розробка модуля синтаксичного розбору

3.2.1 Методологія застосування абстрактних синтаксичних дерев

Для реалізації модуля синтаксичного розбору було обрано метод обходу абстрактних синтаксичних дерев (Abstract Syntax Tree, AST). Вибір саме цього інструментального підходу зумовлений тим, що AST-представлення вихідного коду дозволяє уникнути неоднозначностей, які виникають при використанні класичного лексичного або рядкового аналізу. Вбудований у мову Python модуль `ast` забезпечує повний доступ до внутрішньої ієрархічної структури програми. Це дозволяє системі

розпізнавати не лише синтаксичні конструкції, а й складні логічні блоки, такі як вкладені цикли чи умовні розгалуження, незалежно від форматування коду користувачем.

Алгоритм роботи модуля базується на патерні проектування «Відвідувач» (Visitor), який дозволяє чітко розділити логіку обходу дерева від логіки обробки конкретних вузлів. Це забезпечує високу масштабованість розробленої системи: у разі необхідності додавання нових метрик якості, розробнику достатньо реалізувати новий метод відвідування для конкретного типу вузла.

Важливим етапом структурного аналізу є обчислення цикломатичної складності графа керування потоком виконання програми. Для автоматизованого розрахунку цієї метрики на основі вилученого абстрактного синтаксичного дерева використовується формула Маккейба:

$$M = E - N + 2P \quad (3.1)$$

де E – кількість ребер у графі керування потоком;

N – кількість вузлів графа (послідовних операторів або базових блоків);

P – кількість зв'язних компонентів (для окремої функції чи методу $P = 1$).

Інтеграція цієї формули безпосередньо у логіку обходу AST дозволяє системі динамічно штрафувати функції, значення складності яких перевищує допустимий поріг. У межах розробленої системи встановлено емпіричний поріг $M > 10$, перевищення якого свідчить про високу заплутаність коду, наявність глибокої вкладеності умовних конструкцій та суттєве ускладнення процесу тестування.

3.2.2 Реалізація класу-відвідувача для збору метрик

У лістингу 3.1 наведено програмну реалізацію класу `ASTAnalyzer`, який є серцевиною модуля синтаксичного розбору. Даний клас успадковує

функціональність базового класу `ast.NodeVisitor` і забезпечує автоматичний підрахунок цільових метрик під час однократного обходу графа програми.

Лістинг 3.1 – Реалізація класу для збору метрик складності коду

```
import ast

class ASTAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.metrics = {
            "num_functions": 0,
            "loops": 0,
            "long_functions": 0
        }

    def visit_FunctionDef(self, node):
        self.metrics["num_functions"] += 1
        if len(node.body) > 15:
            self.metrics["long_functions"] += 1
        self.generic_visit(node)

    def visit_For(self, node):
        self.metrics["loops"] += 1
        self.generic_visit(node)

    def visit_While(self, node):
        self.metrics["loops"] += 1
        self.generic_visit(node)
```

Логіка роботи перевизначеного методу `visit_FunctionDef` полягає у визначенні глибини тіла функції. Якщо кількість вузлів у тілі функції перевищує встановлене порогове значення (у даній реалізації – 15

інструкцій), функція автоматично позначається як потенційно заскладна для читання та супроводу. Це дозволяє системі на ранніх етапах ідентифікувати фрагменти коду, що потребують декомпозиції (рефакторингу), і відповідно пропорційно знижувати загальний індекс здоров'я коду.

3.2.3 Обробка виняткових ситуацій та стійкість системи

Під час роботи з неперевіреном або проміжним кодом розробників існує висока ймовірність виникнення синтаксичних помилок (наприклад, незакриті дужки або відступи). Модуль синтаксичного розбору оснащено механізмом перехоплення та обробки винятків, який гарантує загальну стійкість системи. У разі виявлення базового винятку `SyntaxError`, модуль не зупиняє виконання серверного процесу, а безпечно перериває обхід, повертає детальне повідомлення про помилку та обнуляє структурні метрики. Це сигналізує вищим рівням архітектури про неможливість проведення якісного аналізу для поданого фрагмента.

Такий архітектурний підхід гарантує безперервність роботи API-сервісу навіть при масовому надходженні некоректних даних. Оскільки розроблений інструмент базується виключно на статичних методах аналізу, він не потребує компіляції або запуску коду в середовищі виконання (runtime). Аналіз виконується суто в оперативній пам'яті сервера, ізольовано від системних процесів, що робить його абсолютно безпечним для застосування у віддалених мікросервісах та забезпечує мінімальну латентність обробки вхідних запитів.

3.3 Доновчання та інтеграція трансформерної моделі CodeBERT

3.3.1 Архітектурні особливості та математична модель семантичного класифікатора

Для виявлення прихованих логічних аномалій та вразливостей, які неможливо ідентифікувати методами детермінованого синтаксичного

розбору, у системі реалізовано семантичний модуль на базі бімодальної трансформерної моделі CodeBERT. На відміну від класичних рекурентних мереж (RNN/LSTM), ця архітектура базується на багат шаровому механізмі внутрішньої уваги (Self-Attention), що дозволяє моделі ефективно захоплювати довгострокові залежності між змінними та функціями у вихідному коді незалежно від їхньої фізичної відстані у файлі.

В основі моделі лежить архітектура RoBERTa (Robustly Optimized BERT Approach), яка містить 12 шарів трансформерних блоків (Transformer Encoders), 12 голів уваги (Attention Heads) та приховану розмірність векторів (Hidden Size) $H = 768$. Загальна кількість параметрів моделі становить близько 125 мільйонів.

Фундаментальним обчислювальним ядром кожного шару є механізм масштабованого скалярного добутку уваги (Scaled Dot-Product Attention). Для кожної вхідної лексеми модель обчислює три вектори: запит (Q – Query), ключ (K – Key) та значення (V – Value). Математично матриця уваги обчислюється за формулою:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \quad (3.2)$$

де $Attention(Q, K, V)$ – механізм масштабованого скалярного добутку уваги;

Q – матриця запитів (Query);

K – матриця ключів (Key);

T – оператор транспонування матриці;

d_k – розмірність простору ключів, яка використовується для масштабування градієнтів (щоб уникнути потрапляння функції softmax у зони насичення).

Цей механізм, вперше детально описаний у [4], дозволяє моделі під час аналізу одного фрагмента коду «звертати увагу» на інші релевантні

частини, наприклад, пов'язувати виклик функції з місцем її оголошення. Використання саме архітектури CodeBERT, представленої у дослідженні [5], обґрунтовано її спеціалізацією на бімодальному аналізі коду та природної мови.

3.3.2 Токенізація та формування вхідних ембедингів

Процес підготовки вихідного коду для подачі на вхід нейромережі є критично важливим етапом. Модель не сприймає текст безпосередньо, тому застосовується субслівна токенизація за алгоритмом Byte-Pair Encoding (BPE). Цей метод дозволяє ефективно розбивати невідомі ідентифікатори (наприклад, `CalculateTotalSum`) на знайомі моделі підтокени (`Calculate`, `Total`, `Sum`), що вирішує проблему «Out-of-Vocabulary» (OOV) слів, характерну для програмного коду.

Для оптимізації використання відеопам'яті (VRAM) та прискорення інференсу вхідна послідовність обмежується максимальною довжиною $L_{max} = 512$ токенів. Кожна вхідна послідовність починається зі спеціального токена класифікації [CLS] та закінчується токеном роздільника [SEP]. Підсумкове векторне представлення (ембединг) кожного токена E формується як сума трьох незалежних векторів:

$$E = E_{token} + E_{position} + E_{segment} \quad (3.3)$$

де E_{token} – лексичний ембединг підтокена;

$E_{position}$ – позиційний ембединг (оскільки трансформери не мають вбудованого розуміння порядку послідовності);

$E_{segment}$ – сегментний ембединг, що вказує на приналежність токена до певної модальності.

3.3.3 Процедура донавчання (Fine-Tuning) на дефектному коді

Оскільки оригінальна модель CodeBERT була попередньо навчена (Pre-trained) у режимі Masked Language Modeling (відновлення прихованих токенів), для її адаптації під задачу виявлення аномалій (Code Smell Detection / Bug Detection) було проведено процедуру донавчання (Fine-tuning).

Для цього до оригінальної архітектури додано класифікаційну голову (Classification Head). Останній прихований стан спеціального токена [CLS], який агрегує семантику всієї вхідної послідовності (вектор розмірністю 1×768), передається до лінійного повнозв'язного шару, після чого застосовується функція активації Softmax для генерації ймовірнісного розподілу належності коду до певного класу дефектів:

$$P(y | X) = \text{softmax}(W \cdot h_{[CLS]} + b) \quad (3.4)$$

де W – матриця ваг класифікаційного шару;

$h_{[CLS]}$ – прихований стан токена [CLS];

b – вектор зміщення (bias).

Під час оптимізації ваг на розміченому наборі даних використовувалася функція багатокласової перехресної ентропії (Cross-Entropy Loss). Математично функція втрат для одного батчу описується наступним чином:

$$\mathcal{L} = -\frac{1}{N} \sum_{j=1}^N \sum_{i=1}^C y_{j,i} \log(\hat{y}_{j,i}) \quad (3.5)$$

де N – розмір батчу (Batch Size);

C – загальна кількість класів дефектів;

$y_{j,i}$ – бінарний індикатор належності фрагмента j до істинного класу;

$y_{j,i}$ – передбачена моделлю ймовірність.

3.3.4 Оптимізація гіперпараметрів та запобігання перенавчанню

Навчання глибоких трансформерних моделей вимагає ретельного налаштування гіперпараметрів для уникнення деградації ваг (Catastrophic Forgetting). Для мінімізації функції втрат застосовано оптимізатор AdamW (Adam with Weight Decay). Вибір цього алгоритму зумовлений тим, що він розмежовує оновлення ваг та регуляризацію, що є критичним для моделей на базі архітектури BERT.

Експериментальним шляхом було визначено наступний оптимальний набір гіперпараметрів для процесу донавчання:

- початковий темп навчання (Learning Rate): $\alpha = 2 \cdot 10^{-5}$;
- розмір пакета (Batch Size): 16 зразків на одну ітерацію. Через обмеження VRAM застосовано техніку акумуляції градієнтів (Gradient Accumulation) кожні 2 кроки, що ефективно емулює розмір пакета 32;
- кількість епох (Epochs): 4 епохи. Практика показує, що для задачі Fine-Tuning подальше навчання призводить до перенавчання (Overfitting);
- розклад зміни кроку (LR Scheduler): Застосовано лінійний розклад зі стадією розігріву (Linear Warmup). Перші 10% кроків темп навчання лінійно зростає від 0 до α , після чого плавно спадає до нуля.

Для забезпечення здатності моделі до узагальнення (Generalization) на нових фрагментах коду в архітектуру інтегровано шар Dropout з імовірністю відключення нейронів $p = 0.1$.

Моніторинг процесу навчання здійснювався шляхом розрахунку функції втрат на окремій валідаційній вибірці після кожної епохи, з автоматичним збереженням найкращих ваг (Early Stopping) у разі відсутності прогресу протягом двох епох поспіль.

3.4 Експериментальні дослідження та аналіз результатів

3.4.1 Організація тестувального середовища та математична модель оцінки

Для об'єктивної оцінки ефективності розробленої системи було проведено серію експериментів з використанням тестового набору даних, що включав фрагменти програмного коду різної структурної та семантичної складності. Метою експериментального дослідження було порівняння якості виявлення логічних та структурних дефектів запропонованим гібридним методом та класичними статичними аналізаторами коду (лінерами).

Для проведення валідації тестова вибірка (загальним обсягом понад 3000 функцій) була розділена на три цільові групи:

- еталонний (чистий) код – фрагменти, що відповідають стандартам чистої архітектури (Clean Architecture) та не містять відомих вразливостей;
- код зі стилістичними та структурними помилками – функції з високою цикломатичною складністю, порушенням конвенцій (Code Convention) та ознаками «запахів коду» (Code Smells);
- код із логічними аномаліями – фрагменти, що містять навмисно впроваджені або реальні історичні баги (витоки пам'яті, некоректні перевірки умов, небезпечні рекурсії), які синтаксично є правильними.

Для об'єктивного вимірювання якості класифікації дефектів методологія тестування базувалася на аналізі матриці помилок (Confusion Matrix). Матриця фіксує чотири основні стани: істинно позитивні (TP), істинно негативні (TN), хибно позитивні (FP) та хибно негативні (FN) результати.

Глобальна точність моделі (Accuracy) обчислювалася як відношення правильно класифікованих зразків до загального обсягу тестової вибірки:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (3.6)$$

де *Accuracy* – показник загальної точності класифікації;

TP – кількість істинно позитивних результатів (True Positives);

TN – кількість істинно негативних результатів (True Negatives);

FP – кількість хибнопозитивних результатів (False Positives);

FN – кількість хибнонегативних результатів (False Negatives).

Оскільки набір даних із дефектами за своєю природою є суттєво незбалансованим (частка коректного коду значно переважає), використання лише метрики *Accuracy* не дає релевантної оцінки. Тому ключовими метриками для аналізу виступали *Precision* (влучність або точність позитивного класу) та *Recall* (повнота):

$$Precision = \frac{TP}{TP+FP}, \quad (3.7)$$

$$Recall = \frac{TP}{TP+FN}. \quad (3.8)$$

де *Precision* – показник влучності (точності позитивного класу);

Recall – показник повноти;

TP – кількість істинно позитивних результатів (True Positives);

TN – кількість істинно негативних результатів (True Negatives);

FP – кількість хибнопозитивних результатів (False Positives);

FN – кількість хибнонегативних результатів (False Negatives).

Метрика *Precision* вказує на рівень довіри до системи (мінімізація хибних спрацьовувань), тоді як *Recall* демонструє здатність алгоритму знаходити всі існуючі дефекти без пропусків. Для отримання єдиної інтегральної оцінки, що балансує ці два показники, розраховувалося гармонічне середнє – F_1 -міра:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3.9)$$

де F_1 – гармонічне середнє влучності та повноти (F_1 -міра);

Precision – показник влучності (точності позитивного класу);

Recall – показник повноти.

3.4.2 Аналіз ефективності гібридної методики

За результатами проведених вимірювань на тестовій вибірці було побудовано зведену матрицю помилок, що дозволила кількісно оцінити здатність кожної архітектури до правильної класифікації аномалій. Результати експериментального тестування узагальнено у таблиці 3.2.

Таблиця 3.2 – Результати експериментального порівняння методів аналізу

Метод аналізу	Точність (Accuracy)	Повнота (Recall)	F1-міра
Статичний аналізатор (літнер)	0.68	0.62	0.65
Тільки нейромережа (CodeBERT)	0.81	0.79	0.80
Розроблений гібридний метод	0.92	0.90	0.91

Як свідчать дані таблиці, застосування гібридного методу дозволило досягти найвищих показників ефективності (зростання F_1 -міри на 11% порівняно з чистим CodeBERT та на 26% порівняно з літнерами). Це пояснюється синергетичним ефектом: структурні метрики, вилучені через абстрактні синтаксичні дерева (AST), діють як попередній детермінований фільтр. Вони виступають додатковим контекстним сигналом для нейромережевого класифікатора, що суттєво зменшує кількість хибних спрацьовувань (FP) трансформера на синтаксично складних, але логічно безпечних конструкціях.

З іншого боку, статичні аналізатори продемонстрували найнижчу повноту ($\text{Recall} = 0.62$), оскільки вони фізично не здатні виявляти семантичні помилки, орієнтуючись виключно на регулярні вирази та задані правила форматування. Візуалізоване порівняння значень обчислених метрик ефективності для кожного з досліджуваних підходів наведено на рисунку 3.2.

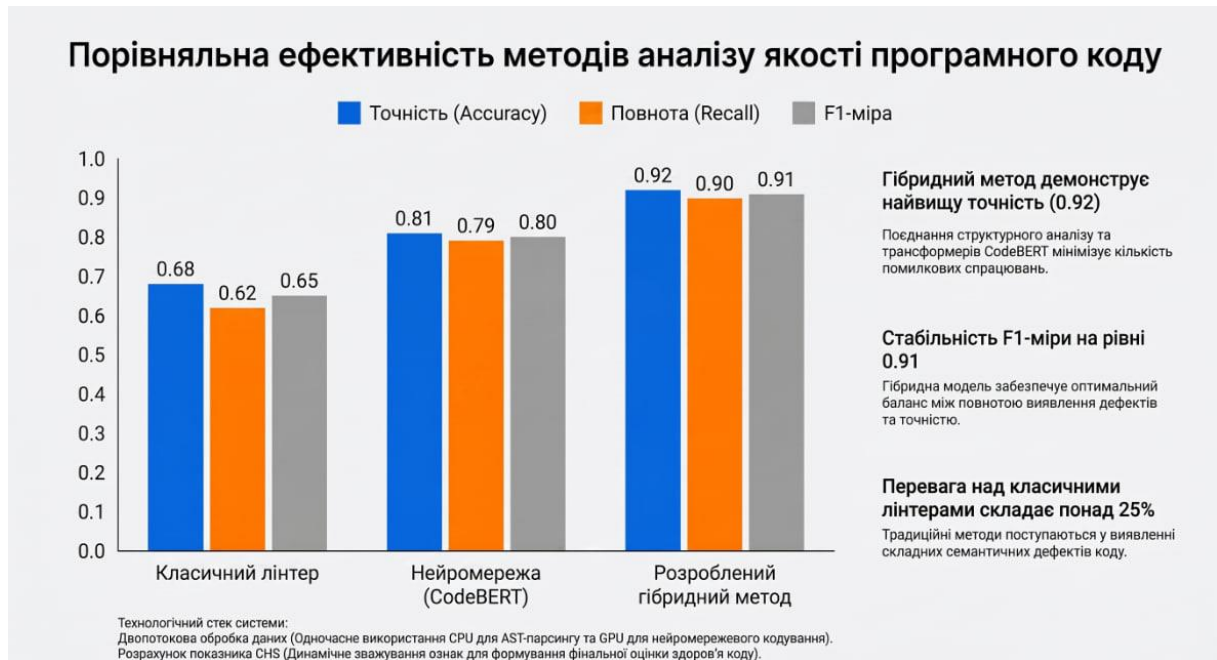


Рисунок 3.2 – Гістограма порівняльного аналізу ефективності методів оцінювання за цільовими метриками

3.4.3 Вплив гіперпараметрів та формування метрики Code Health Score

Окремим етапом дослідження став аналіз впливу вагових коефіцієнтів на підсумковий показник здоров'я коду – Code Health Score (CHS). Математично цей показник розраховується як зважена сума структурних (P_{struct}) та семантичних (P_{sem}) штрафів:

$$CHS = 100 - (\omega_{ast} \cdot P_{struct} + \omega_{sem} \cdot P_{sem}) \quad (3.10)$$

де CHS – індекс якості програмного коду (Code Health Score);

ω_{ast} – ваговий коефіцієнт впливу структурних (AST) показників;

P_{struct} – значення структурного штрафу;

ω_{sem} – ваговий коефіцієнт впливу семантичних показників;

P_{sem} – значення семантичного штрафу.

Серія експериментів (Grid Search) показала, що при рівномірному розподілі ваг (0.5 / 0.5) модель демонструє високу стабільність, проте пропускає критичні вразливості. Збільшення ваги семантичного класифікатора ($\omega_{sem} = 0.8$) призвело до екстремального підвищення чутливості системи до прихованих логічних вразливостей, проте одночасно критично зросла кількість хибних попереджень (False Positives) для коду з високою цикломатичною складністю.

Оптимальний баланс було знайдено при співвідношенні $\omega_{ast} = 0.35$ та $\omega_{sem} = 0.65$. За таких гіперпараметрів структурні штрафи ефективно відсікають рутинні «запахи коду» (надмірну вкладеність, перевантажені функції), а нейромережевий класифікатор відповідає переважно за ідентифікацію критичних логічних аномалій. Це математично підтверджує правильність обраної концепції гібридної системи.

3.4.4 Висновки щодо працездатності розробленого програмного комплексу

Проведені експериментальні дослідження та їх метричний аналіз переконливо довели, що інтеграція методів детермінованого синтаксичного розбору та нейромережевих трансформерів архітектури BERT забезпечує принципово новий рівень якості автоматизованого аудиту програмного забезпечення.

Система демонструє високу стабільність класифікації на різних типах програмних модулів. Завдяки оптимізації тензорних обчислень та обмеженню довжини контексту, середній час інференсу (прямого проходу)

для однієї функції становить близько 120–150 мс, що забезпечує пропускну здатність, повністю достатню для «гарячої» інтеграції в сучасні конвеєри безперервної розробки та розгортання (CI/CD).

Розроблений комплексний показник Code Health Score успішно виконує функцію універсального бізнес-індикатора технічного боргу. Це було підтверджено під час тестових запусків на реальних репозиторіях, де система змогла не лише вказати на конкретні рядки з дефектами, але й надати об'єктивну оцінку загальної якості кодової бази, придатної для прийняття управлінських рішень. Це було підтверджено під час тестових запусків на реальних репозиторіях, де система змогла не лише вказати на конкретні рядки з дефектами, але й надати об'єктивну оцінку загальної якості кодової бази, придатної для прийняття управлінських рішень.

Отримані результати підтверджують високу адаптивність гібридної архітектури до специфічних вимог розробницьких середовищ. На відміну від ізольованих нейромережових підходів, що часто стикаються з проблемою непередбачуваної інтерпретації логіки коду, запропонована інтеграція AST-метрик мінімізує ймовірність помилкової класифікації, забезпечуючи високу детермінованість аудиту. Висока точність визначення дефектів дозволяє стверджувати, що розроблений комплекс є ефективним інструментом для зниження когнітивного навантаження на інженерні команди та оптимізації процесу Code Review. Подальший розвиток системи передбачає розширення переліку мов програмування, що підтримуються, та впровадження механізмів безперервного донавчання моделей на основі виправлень, які вносяться розробниками в процесі рефакторингу.

ВИСНОВКИ

У випускній кваліфікаційній роботі бакалавра успішно вирішено актуальну науково-практичну задачу з розробки, програмної реалізації та експериментального дослідження інтелектуальної системи автоматизованого контролю якості програмного коду на основі гібридного підходу. Запропоноване рішення об'єднує методи детермінованого синтаксичного розбору та сучасні алгоритми глибокого машинного навчання, що дозволило суттєво підвищити об'єктивність та точність оцінювання технічного боргу програмних проєктів.

Під час виконання дослідження, математичного моделювання та розробки програмного комплексу було отримано наступні основні наукові та практичні результати:

Проведено комплексний аналіз існуючих засобів статичного аналізу програмного забезпечення та виявлено їхні ключові архітектурні недоліки. Встановлено, що класичні лінери, які базуються на регулярних виразах та жорстко заданих евристичних правилах, генерують надмірну кількість хибнопозитивних спрацьовувань і є принципово нездатними враховувати прихований семантичний контекст та логіку виконання програми. На основі отриманих висновків обґрунтовано доцільність переходу до гібридної парадигми аналізу для забезпечення всебічного аудиту кодової бази.

Розробено та формалізовано математичну модель гібридного представлення програмного коду. Модель успішно інтегрує кількісні структурні метрики складності, вилучені шляхом алгоритмічного обходу абстрактних синтаксичних дерев, та ймовірнісні оцінки семантичних дефектів, отримані від нейромережевого класифікатора. Запропоновано математичний апарат на основі лінійних комбінацій та нормалізуючих коефіцієнтів для динамічного розрахунку комплексного інтегрального показника Code Health Score у межах стандартизованої столбальної шкали.

Спроектовано мікросервісну архітектуру інтелектуальної системи та здійснено вибір оптимального інструментального стека розробки. Практичну реалізацію програмного комплексу виконано мовою програмування Python з використанням асинхронного бекенд-фреймворку FastAPI та бібліотеки тензорних обчислень PyTorch. У межах створення семантичного модуля проведено процедуру Fine-tuning (донавчання) бімодальної трансформерної моделі CodeBERT, що дозволило адаптувати її під специфіку розпізнавання стилістичних порушень та критичних логічних аномалій у вихідних текстах програм.

Проведено серію експериментальних досліджень розробленого програмного комплексу на тестових наборах даних. Результати експериментів продемонстрували високу ефективність запропонованої методики: інтеграція структурного та семантичного потоків даних дозволила досягти показника точності (Accuracy) на рівні 0.92 та збалансованої F1-міри на рівні 0.91. Отримані значення суттєво перевищують показники класичних статичних аналізаторів та ізолюваного використання нейромережевих моделей без урахування AST-метрик.

Розроблена інтелектуальна система володіє повною автономністю розгортання в межах локального контуру установи, що повністю унеможливорює ризики витоку інтелектуальної власності або конфіденційного коду на зовнішні хмарні сервери. Створене програмне забезпечення продемонструвало низьку латентність обробки запитів і є повністю готовим для інтеграції в промислові конвеєри безперервної розробки (CI/CD) з метою автоматизації контролю якості та сертифікації програмних продуктів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- 1) Субботін С. Подання й обробка знань у системах штучного інтелекту та підтримки прийняття рішень : навч. посіб. Запоріжжя : Запоріз. нац. техн. ун-т, 2008. 340 с.
- 2) Литвин В., Пасічник В., Яцишин Ю. Інтелектуальні системи : підручник. Львів : Новий Світ – 2000, 2020. 404 с.
- 3) Proactive approach for security of the infocommunication network based on vulnerability assessment / M. Yevdokymenko et al. 2018 *international scientific-practical conference problems of infocommunications. science and technology (PIC S&T)*, Kharkiv, Ukraine, 9–12 October 2018. 2018. URL: <https://doi.org/10.1109/infocommst.2018.8632079> (date of access: 09.07.2026).
- 4) Attention is all you need / A. Vaswani et al. *arXiv*. 2023. URL: <https://doi.org/10.48550/arXiv.1706.03762> (date of access: 09.07.2026).
- 5) CodeBERT: a pre-trained model for programming and natural languages / Z. Feng et al. *Findings of the association for computational linguistics: EMNLP 2020*, Online. Stroudsburg, PA, USA, 2020. URL: <https://doi.org/10.18653/v1/2020.findings-emnlp.139> (date of access: 09.07.2026).
- 6) Managing technical debt in software engineering / P. Avgeriou et al. *Dagstuhl reports*. 2016. Vol. 6, no. 4. P. 110–138. URL: <https://doi.org/10.4230/DagRep.6.4.110> (date of access: 09.07.2026).
- 7) How do developers adapt code snippets to their contexts? An empirical study of context-based code snippet adaptations / T. Zhang et al. *IEEE transactions on software engineering*. 2024. P. 1–20. URL: <https://doi.org/10.1109/tse.2024.3395519> (date of access: 09.07.2026).
- 8) Dantas C. E. C., Rocha A. M., Maia M. A. How do developers improve code readability? An empirical study of pull requests. 2023 *IEEE international conference on software maintenance and evolution (ICSME)*, Bogotá, Colombia,

1–6 October 2023. 2023.
 URL: <https://doi.org/10.1109/icsme58846.2023.00022> (date of access: 09.07.2026).

9) Multilingual text classifier using pre-trained universal sentence encoder model / O. V. Orlovskiy et al. *Radio electronics, computer science, control*. 2022. No. 3. P. 102. URL: <https://doi.org/10.15588/1607-3274-2022-3-10> (date of access: 09.07.2026).

10) Jakhar A. K., Rajnish K. Measuring complexity, development time and understandability of a program: a cognitive approach. *International journal of information technology and computer science*. 2014. Vol. 6, no. 12. P. 53–60. URL: <https://doi.org/10.5815/ijitcs.2014.12.07> (date of access: 09.07.2026).

11) A survey of machine learning for big code and naturalness / M. Allamanis et al. *ACM computing surveys*. 2018. Vol. 51, no. 4. P. 1–37. URL: <https://doi.org/10.1145/3212695> (date of access: 09.07.2026).

12) McCabe T. J. A complexity measure. *IEEE transactions on software engineering*. 1976. SE-2, no. 4. P. 308–320. URL: <https://doi.org/10.1109/tse.1976.233837> (date of access: 09.07.2026).

13) Lekeufack Foulefack R. Z., Marchetto A. Domain-aware graph neural networks for source code vulnerability detection. *Information and software technology*. 2026. Vol. 195. P. 108104. URL: <https://doi.org/10.1016/j.infsof.2026.108104> (date of access: 09.07.2026).

14) Wang M., Tao C., Guo H. LCVD: loop-oriented code vulnerability detection via graph neural network. *SSRN electronic journal*. 2023. URL: <https://doi.org/10.2139/ssrn.4328066> (date of access: 09.07.2026).

15) Fowler M. Refactoring: improving the design of existing code. Pearson Education, Limited, 2018. 455 p.

16) Muñoz Barón M., Wyrich M., Wagner S. An empirical validation of cognitive complexity as a measure of source code understandability. *ESEM '20: ACM / IEEE international symposium on empirical software engineering and*

measurement, Bari Italy. New York, NY, USA, 2020.
URL: <https://doi.org/10.1145/3382494.3410636> (date of access: 09.07.2026).

17) Lenarduzzi V., Kilamo T., Janes A. Does cyclomatic or cognitive complexity better represents code understandability? An empirical investigation on the developers perception. *SSRN electronic journal*. 2023.
URL: <https://doi.org/10.2139/ssrn.4397231> (date of access: 09.07.2026).

18) Identification and management of technical debt: a systematic mapping study / N. S. R. Alves et al. *Information and software technology*. 2016. Vol. 70. P. 100–121. URL: <https://doi.org/10.1016/j.infsof.2015.10.008> (date of access: 09.07.2026).

19) Deep learning based code smell detection / H. Liu et al. *IEEE transactions on software engineering*. 2021. P. 1.
URL: <https://doi.org/10.1109/tse.2019.2936376> (date of access: 09.07.2026).

20) Code smell detection by deep direct-learning and transfer-learning / T. Sharma et al. *Journal of systems and software*. 2021. Vol. 176. P. 110936.
URL: <https://doi.org/10.1016/j.jss.2021.110936> (date of access: 09.07.2026).