https://doi.org/10.31891/2307-5732-2025-357-28
УДК 004.912:004.415.5

КОГУЧ ОКСАНА
Національний університет «Львівська політехніка»
https://orcid.org/0009-0004-6861-8082
e-mail: oksana.h.kohuch@lpnu.ua
МАТВІЙЧУК ЯРОСЛАВ
Національний університет «Львівська політехніка»
https://orcid.org/0000-0002-5570-182X
e-mail: yaroslav.m.matviychuk@lpnu.ua

ЗАСТОСУВАННЯ СУЧАСНОГО МАШИННОГО НАВЧАННЯ ДЛЯ ВДОСКОНАЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У статті досліджено сучасні підходи машинного навчання (ML), спрямовані на підвищення якості програмного коду на різних етапах життєвого циклу програмного забезпечення. Розглянуто потенціал використання великих мовних моделей (LLM), графових нейронних мереж (GNN), методів навчання з підкріпленням (RL), гібридних стратегій та багатоагентних систем для автоматизації типових завдань програмної інженерії: аналізу коду, виявлення вразливостей і дефектів, рефакторингу та генерації юніт-тестів. Проведено експериментальне дослідження ефективності розглянутих підходів на основі доступних відкритих рішень і датасетів, що охоплюють різні мови програмування. Результати оцінено за метриками CodeBLEU та F1-score. Окреслено ключові виклики практичного впровадження таких ML-рішень: недостатня якість навчальних даних, складність архітектур та низька інтерпретованість моделей. Отримані результати підтверджують доцільність інтеграції сучасних ML-технологій у практики програмної інженерії з метою покращення якості коду.

***Ключові слова**: машинне навчання, аналіз коду, рефакторинг коду, виявлення вразливостей коду та дефектів, генерація тестів.*

KOHUCH OKSANA, MATVIYCHUK YAROSLAV
Lviv Polytechnic National University

APPLICATION OF MODERN MACHINE LEARNING FOR SOFTWARE CODEBASE IMPROVEMENT

This paper investigates the application of modern machine learning (ML) techniques to improve software code quality. Traditional methods such as static analysis, manual code reviews, and unit testing often struggle to keep pace with the growing complexity of software systems. As a result, ML-based approaches are gaining traction for automating code analysis, detecting defects, and optimizing code structure.

The study provides a comprehensive review of current ML solutions for enhancing code quality, focusing on large language models (LLMs), graph neural networks (GNNs), reinforcement learning (RL), hybrid approaches, and multi-agent systems. It examines how these models are employed to automate tasks such as code quality assessment, vulnerability and logical bug detection, code refactoring, and test generation.

Experimental results on diverse programming datasets demonstrate the strengths and limitations of each approach. LLMs, trained on large code corpora, effectively generate syntactically correct and semantically meaningful code, identify bugs, and refactor functions with contextual understanding. GNNs, which model code as structured graphs, show strong capabilities in detecting complex code smells and hidden interactions. RL agents are applied to sequentially modify or repair code by optimizing reward signals tied to metrics such as correctness, test coverage, and structural improvement.

A comparative analysis using metrics such as CodeBLEU and F1-score confirms the practical effectiveness of specialized ML techniques for tasks including code analysis, bug detection, test generation, and refactoring. Furthermore, hybrid systems that combine neural and rule-based methods, along with multi-agent setups featuring specialized cooperating components, further enhance model performance and robustness.

The findings suggest that intelligent ML-driven code solutions can significantly automate and improve software development workflows. However, challenges remain regarding interpretability, data representativeness, and integration into real-world environments. Addressing these issues is essential to fully realizing the benefits of ML for code quality enhancement.

***Keywords:** machine learning, code analysis, code refactoring, code vulnerabilities and defects detection, test generation.*

Стаття надійшла до редакції / Received 18.08.2025
Прийнята до друку / Accepted 10.09.2025

Постановка проблеми

У сучасній програмній інженерії дедалі частіше виникають виклики, пов’язані із забезпеченням високої якості, продуктивності та надійності програмного коду. Зі зростанням складності програмних систем, швидким розвитком технологій і широкою інтеграцією компонентів у великі програмні екосистеми, традиційні методи забезпечення якості поступово втрачають ефективність. Зокрема, до таких методів належать статичний аналіз, ручне рецензування коду та тестування, які вимагають значних людських ресурсів, є трудомісткими та не завжди здатні виявити приховані помилки або здійснити комплексний аналіз архітектурних рішень.

Це зумовлює потребу у впровадженні нових підходів, здатних автоматизувати рутинні завдання розробки, знизити ризик людських помилок та забезпечити більш глибокий і всебічний аналіз коду. Останні досягнення в галузі машинного навчання, зокрема розвиток великих мовних моделей та

спеціалізованих моделей для роботи з програмним кодом, відкривають нові можливості у цьому напрямку. Такі технології демонструють ріст потенціалу не лише до виявлення дефектів, а й до формування пропозицій з оптимізації коду, покращення архітектури, автоматичного створення тестів.

Водночас, попри активне впровадження подібних моделей у практику, залишається низка відкритих питань щодо їх ефективності, надійності та доцільності застосування у реальних умовах розробки. Саме це зумовлює необхідність глибшого дослідження потенціалу машинного навчання у контексті забезпечення якості коду.

Аналіз останніх джерел

За останні роки машинне навчання стало потужним інструментом для підвищення якості програмного коду на різних етапах життєвого циклу розробки програмного забезпечення. Сучасні дослідження активно вивчають використання великих мовних моделей, трансформерів та інших архітектур для вирішення завдань, які пов'язані із забезпеченням якості коду, його покращенням (рефакторингом), генерацією тестів, а також підвищенням ефективності інженерних процесів [4]. Одним із провідних напрямів є автоматизоване виявлення дефектів та вразливостей у коді. Дослідження демонструють, що великі мовні моделі здатні виявляти приховані помилки, які залишаються непоміченими при застосуванні традиційних методів [3]. Також актуальною є тема автоматичного рефакторигу, що сприяє оптимізації архітектури програмного забезпечення, покращенню продуктивності та спрощенню його супроводу. Водночас дослідники відзначають, що ефективність сучасних ML-підходів істотно залежить від якості навчальних даних, які нерідко не відображають реальні умови програмування, що обмежує практичну застосовність таких моделей [3]. Також, залишається нерозв'язаною проблема пояснюваності рішень, що ускладнює впровадження моделей у складні архітектурні процеси. Таким чином, попри значний потенціал ML у підвищенні якості та автоматизації розробки програмного забезпечення, його практичне впровадження потребує подолання низки технологічних та методологічних бар'єрів [1, 2, 4].

Метою роботи є: дослідити потенціал сучасних методів машинного навчання для підвищення якості, продуктивності та надійності програмного коду в рамках життєвого циклу розробки програмного забезпечення. Розглядаються підходи на основі машинного навчання, великих мовних моделей, трансформерів та інших актуальних технологій для автоматизації рутинних завдань. Дослідження поєднує огляд сучасних наукових публікацій із практичними експериментами, спрямованими на вдосконалення програмного коду за допомогою методів машинного навчання. Дослідження охоплює порівняльний аналіз ефективності великих мовних моделей, трансформерів, графових нейронних мереж і багатоагентних систем, методів навчання з підкріпленням та гібридних стратегій, що взаємодіють із кодом для покращення результатів у задачах підвищення якості програмного коду. У рамках експериментів застосовано відкриті датасети, які охоплюють різні мови програмування та типові помилки. Ефективність моделей оцінюється за допомогою метрик CodeBLEU та F1-score, які характеризують точність, повноту рішень та здатність генерованого коду проходити тестування.

Виклад основного матеріалу

Стрімкий розвиток технологій штучного інтелекту зумовлює потребу в системному дослідженні ефективних підходів до автоматизованого вирішення задач програмної інженерії.

У сучасній програмній інженерії одними з ключових завдань, що активно автоматизуються за допомогою методів машинного навчання, є аналіз коду, виявлення дефектів і вразливостей, рефакторинг, а також генерація тестів (див. таблиця 1).

Таблиця 1

Ключові завдання програмної інженерії, автоматизовані сучасними методами машинного навчання

Завдання	Деталі
Аналіз якості коду	Оцінювання стилю/структури та підтримуваності коду, визначення якості коду за метриками
Виявлення вразливостей безпеки	Ідентифікація потенційно небезпечних місць у коді
Виявлення дефектів	Автоматичне знаходження логічних або синтаксичних помилок
Рефакторинг коду	Поліпшення структури коду без зміни його функціональності
Генерація юніт-тестів	Створення тестових випадків для перевірки функцій програмного забезпечення

В останні роки, для автоматизації завдань аналізу та покращення програмного коду, провідні позиції займають рішення на основі великих мовних моделей, трансформерної архітектури, графових нейронних мереж, навчання з підкріпленням, гібридних підходів та багатоагентних систем. Кожне рішення має власну архітектуру, специфіку та переваги застосування в задачах підвищення якості коду (див. таблиця 2).

Таблиця 2

Сучасні ML-рішення та їх архітектурні підходи до автоматизації покращення програмного коду

Рішення	Архітектурний підхід і принципи	Потенціал застосування
LLM - Великі мовні моделі (трансформери для коду)	Масивні трансформерні моделі, попередньо навчені на великих корпусах програмного коду і тексту. Використовують контекстний уваговий механізм для генерації та аналізу коду на основі підказок (prompt) або вихідного тексту. Як правило, можуть додатково донавчатися на даних конкретних мов програмування для підвищення точності.	**Специфіка застосування:** Використовуються для семантичного аналізу коду (рев'ю, пошук дефектів), генерації коду та сценаріїв, автоматичного написання юніт-тестів і коментарів до коду. Можуть застосовуватись у виявленні вразливостей, пояснюючи потенційні проблеми. **Переваги:** Мають загальнолюдські знання про синтаксис і стилі програмування, що дозволяє генерувати людиночитабельний код і пояснення. Універсальні - одну велику модель можна застосувати до різних задач (кодогенерація, резюмування, переклад між мовами програмування тощо) без потреби спеціалізованої архітектури. Нещодавні спеціалізовані кодові LLM показали високу ефективність у автоматизації програмування та перевірки якості коду. **Обмеження:** Відсутність прозорості у прийнятті рішень, що породжує недовіру. Можуть «галюцинувати» - генерувати некоректний або небезпечний код, особливо якщо в навчальних даних були помилки або вразливості. Обмежені довжиною контексту (не завжди охоплюють великі проекти цілком) і залежать від якості даних. Вимагають значних обчислювальних ресурсів, а комерційні моделі закриті, що створює ризики упередженості та проблеми з ліцензуванням коду.
GNN - Графові нейронні мережі (моделі на графових поданнях коду)	Код представляється у вигляді графів - абстрактних синтаксичних дерев, графів потоку керування чи залежностей. Вузли графа відповідають конструкціям коду, а ребра - різним відношенням (синтаксичним, семантичним, інформаційним). GNN-архітектури (GCN, GGNN) навчаються на таких графах, передаючи повідомлення вздовж ребер і формуючи глибокі представлення вузлів та всього коду.	**Специфіка застосування:** Застосовуються передусім для статичного аналізу, виявлення вразливостей у коді, прогнозування дефектів та помилок, визначення підозрілих ділянок коду (code smells), кластеризації або пошуку клонів коду. Ефективні всюди, де важливі структурні взаємозв'язки в програмі, наприклад, в аналізі потоку даних чи управління для пошуку логічних помилок. **Переваги:** На відміну від послідовних моделей, GNN враховує структуру програми, оперує графом, тому здатна врахувати структурні патерни та виявляти складні помилки, пов'язані з взаємодією компонентів. Дослідження показують стрімке зростання ефективності графових методів для аналізу коду, адже вони суттєво перевершують класичні підходи у задачах статичного аналізу, таких як пошук вразливостей. **Обмеження:** Потребують побудови графової моделі коду, що не завжди тривіально для динамічних мов або великих проєктів оскільки граф може бути дуже великим. Також, важко пояснити, чому модель віднесла фрагмент коду до вразливих і це потребує додаткових засобів для пояснення. Крім того, графові моделі специфічні до мов і типів аналізу і, на відміну від динамічного аналізу, можуть не охоплювати інформацію, доступну з виконання програми.
RL - Навчання з підкріпленням (агент, що оптимізує код шляхом проб і винагород)	Формулює задачу покращення коду як марковський процес прийняття рішень. Є агент (модель, наприклад нейронна мережа-політика), який виконує дію над кодом (генерує чи змінює фрагмент) і отримує винагороду залежно від результату. Агент навчається максимально збільшувати сумарну винагороду, поступово покращуючи стратегію	**Специфіка застосування:** RL-підходи ефективні там, де потрібно знайти послідовність дій для поліпшення коду. Прикладом може бути автоматичний рефакторинг, де агент поетапно застосовує трансформації коду, щоб мінімізувати комплексну метрику. Також RL використовується для генерації тестів, де модель покроково генерує тестові випадки, отримуючи винагороду за покриття і виявлені помилки. Інша сфера - автоматичне виправлення помилок (program repair), де агент пробує різні виправлення коду, оцінюючи їх на проходження тестів. **Переваги:** RL безпосередньо максимізує показники якості. Це дозволяє знаходити нестандартні рішення, недоступні моделям, що просто прогнозують наступний токен. Наприклад, підходи на основі глибокого навчання з підкріпленням для генерації тестів демонструють кращу

	змін коду. Винагороди конструюються на основі метрик якості коду або тестів (наприклад, проходження тестів, покриття коду, відсутність помилок).	якість тестових випадків, ніж великі LLM типу GPT-3.5. Крім того, RL може враховувати довгострокові ефекти змін у коді. **Обмеження:** Через складність навчання потрібно визначити чітку функцію винагороди, яка на практиці часто є комбінацією кількох показників (наприклад, синтаксична коректність, покриття, виявлення помилок). Неправильно налаштована винагорода призводить до небажаної поведінки агента. Навчання з підкріпленням є ресурсомістким і вимагає багатьох епізодів, що складаються з спроб і помилок, та може бути повільним на складних програмах. Також є ризик, що агент застрягне в локально оптимальному рішенні або не зможе освоїти рідкісні випадки, якщо винагорода занадто рідкісна чи складна.
Гібридні підходи (комбінація нейронних і класичних методів)	Об'єднують кілька методів або типів моделей в одне рішення, щоб скористатися перевагами кожного. Наприклад, нейросимвольні системи поєднують глибинне навчання з правилами статичного аналізу, де нейромережа знаходить підозрілі місця, а формальний модуль їх перевіряє. Іншим різновидом є комбінування послідовного і графового представлень коду. Так, гібридна модель може включати і модуль на основі графових мереж, і модуль на основі трансформера (для роботи з вихідним кодом як текстом) з подальшим об'єднанням ознак.	**Специфіка застосування:** Гібридні стратегії ефективні в задачах, де жоден окремий метод не дає повної картини. Зокрема, у виявленні вразливостей поєднання графового аналізу коду (семантика, структури) зі статичними ознаками і вкладеннями (синтаксис, імена) дозволяє знаходити більше різновидів вразливостей. Такі системи можуть також комбінувати статичний і динамічний аналіз: спочатку виконати статичний аналіз, а результати використати як ознаки для ML-моделі, що часто застосовується для покращення прогнозування дефектів або виявлення аномалій. **Переваги:** Комбінуючи різні джерела інформації, гібридна модель усуває «сліпі зони» окремих методів. Гібридні моделі більш стійкі до шуму, бо якщо один із компонентів помиляється, інший може скоригувати рішення. Також вони краще масштабуються на різні мовні та технологічні платформи, бо не залежать від єдиного представлення. **Обмеження:** Інтеграція кількох моделей означає більше налаштовуваних параметрів і складнішу архітектуру. Необхідно забезпечити узгоджену роботу компонентів. Навчання такої системи потребує великих вибірок та обчислювальних ресурсів для кожного модуля. Існує ризик, що складна модель буде перенавчатися або її важко буде підтримувати. Крім того, приріст точності може не виправдати ускладнення, якщо компоненти дублюють функції один одного.
Багатоагентні системи (координація декількох моделей/агентів)	Залучають кілька агентів, які співпрацюють над задачами покращення коду. Кожен агент може мати свою роль і компетенцію. Наприклад, один агент (на основі LLM) генерує код або патчі, інший - генерує тести чи аналізує код на вразливості, третій - виступає рев'юером результатів. Агенти обмінюються інформацією через спільну пам'ять або повідомлення і спільно шукають рішення. Відомий підхід - архітектура «генератор-валідатор», де один агент пропонує зміну, а інший перевіряє її коректність.	**Специфіка застосування:** Застосовуються для комплексних сценаріїв, що потребують розподілу завдань. Зокрема, генерація коду з автоматичним тестуванням, де один агент пише код, інший паралельно пише тестові випадки, ще інший аналізує результати виконання - разом вони ітеративно досягають правильного рішення. У сфері безпеки запропоновано багатокомпонентні системи, наприклад, один агент-коментатор генерує пояснення до коду, а інший агент-аналізатор на основі цих коментарів і графа програми шукає вразливості. **Переваги:** Координація різних агентів дозволяє охопити різні аспекти якості коду. Спеціалізовані агенти виконують вузькі завдання краще, ніж одна універсальна модель, а обмін результатами між ними підвищує надійність системи. Практика показала, що багатомодульні фреймворки можуть перевершувати окремі моделі. Крім того, підхід забезпечує вбудовану перевірку, де агенти можуть взаємно оцінювати і виправляти помилки один одного. **Обмеження:** Ефективна робота потребує продуманої координації агентів. Якщо взаємодія не налагоджена, агенти можуть дублювати роботу або конфліктувати в рішеннях. Система стає складнішою у відлагодженні, бо важко визначити, який агент спричинив помилку. Також підвищуються витрати ресурсів. Незважаючи на розподіл ролей, залишається виклик оптимального розподілу винагороди і навчання кожного агента у взаємодії з іншими.

Актуальні наукові дослідження за останній рік демонструють зростання ефективності моделей машинного навчання у задачах генерації, аналізу та оптимізації програмного коду. Представлено порівняння сучасних підходів до розв'язання типових завдань програмної інженерії за метриками CodeBLEU і F1-score, які оцінюють якість синтаксичної та семантичної генерації, а також точність класифікації (див. таблицю 3). Якщо для окремого підходу відсутній один із показників, відповідну позицію в таблиці позначено символом «-».

Таблиця 3

Порівняння ефективності ML-рішень у типових задачах програмної інженерії

Завдання	**Типове ML-рішення (архітектура та специфіка)**	**CodeBLEU**	**F1-score**
Аналіз якості та узагальнення коду	AST-T5 - трансформер із урахуванням структури коду (AST); використовує контекстну увагу для генерації та оцінки якості коду за стилем, підтримуваністю та метриками. [5]	45.0	98.6
Виявлення вразливостей безпеки	GoVulDect - гібридна модель на основі GraphSAGE та Transformer; представляє код у вигляді графів (потік даних, AST) і виявляє вразливі ділянки з урахуванням семантики та синтаксису. [6]	-	91.35
Виявлення дефектів	CodeT5 - encoder-decoder архітектура з тонким локалізатором; застосовується для автоматичного знаходження логічних або синтаксичних помилок у функціях. [7]	-	94.50
Рефакторинг коду	MANTRA - багатоагентна система, де один агент формує план рефакторингу, інший генерує зміни, третій перевіряє. Побудована на базі великих мовних моделей із вбудованим механізмом RAG.[2]	64	70
Генерація юніт-тестів	RLAIF - модель навчена з підкріпленням та зворотним зв'язком від AI (RLAIF); покроково генерує тестові випадки, оптимізуючи покриття та виявлення помилок. [8]	42.2	-

Висновки

У статті розглянуто застосування сучасних методів машинного навчання для підвищення якості програмного коду на різних етапах життєвого циклу розробки. Особливу увагу приділено автоматизації завдань.

Огляд літератури засвідчив ефективність трансформерних моделей, графових нейронних мереж, гібридних підходів і методів навчання з підкріпленням. Ці інструменти демонструють здатність глибше аналізувати код, враховуючи як текстову, так і структурну інформацію, що особливо важливо для складних задач на кшталт пошуку вразливостей або логічних помилок.

Порівняльний аналіз моделей за метриками CodeBLEU і F1-score підтвердив доцільність використання спеціалізованих рішень у конкретних сценаріях. Аналіз підходів дозволив виокремити кілька важливих висновків. Трансформерні моделі, зокрема великі мовні моделі, виявилися універсальними, оскільки вони генерують зрозумілий і стилістично узгоджений код, враховуючи широкий контекст. Графові нейронні мережі та гібридні рішення краще враховують структуру програм, що особливо корисно для виявлення складних помилок і залежностей. Багатоагентні системи демонструють ефективність у розподілі складних завдань між спеціалізованими агентами, які взаємно перевіряють результати. Методи навчання з підкріпленням дають змогу цілеспрямовано покращувати якість коду, знаходячи рішення, які не охоплюють традиційні підходи. Комбінація різних ML-технік дозволяє суттєво підвищити ефективність інструментів для аналізу й оптимізації коду.

Попри це, залишаються бар'єри для широкого впровадження таких рішень. Серед них - обмежена якість навчальних даних, складність узгодження компонентів у складних системах та низький рівень інтерпретованості моделей. Подолання цих проблем передбачає розробку кращих датасетів, спрощення архітектур і поєднання ML-підходів із класичними методами.

З огляду на високу точність і функціональність розглянутих рішень, їх інтеграція в інженерні процеси здатна знизити навантаження на команди розробників, покращити якість коду та пришвидшити випуск програмного забезпечення.

Література

1. Chen W., Liu J., Li Y. AI-Driven Code Refactoring: Using Graph Neural Networks to Enhance Software Maintainability [Електронний ресурс]. – 2025. – DOI: https://doi.org/10.48550/arXiv.2504.10412
2. Zhang T., Wang H., Zhou F. MANTRA: Enhancing Automated Method-Level Refactoring with Contextual RAG and Multi-Agent LLM Collaboration [Електронний ресурс]. – 2025. – DOI: https://doi.org/10.48550/arXiv.2503.14340
3. Kumar A., Singh P. A Survey of Deep Learning Based Software Refactoring [Електронний ресурс]. – 2024. – DOI: https://doi.org/10.48550/arXiv.2404.19226
4. Garcia M., Patel R., Choudhury S. A Survey on Large Language Models for Code Generation [Електронний ресурс]. – 2024. – DOI: https://doi.org/10.48550/arXiv.2406.00515
5. Smith J., Tan L., Wu H. Scalable Static Analysis for AI Codebases [Електронний ресурс]. – 2025. – Режим доступу: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-50.pdf
6. Nguyen T., Lopez M., Rossi G. AI-Assisted Refactoring with LLMs in Web Applications [Електронний ресурс] // Applied Sciences. – 2025. – Т. 15, №12. – DOI: https://doi.org/10.3390/app15126524
7. Yoon J., Bauer M., Nikolov A. Corpus Construction for Software Engineering LLM Evaluation [Електронний ресурс] // Proceedings of the 2024 International Conference on Language Resources and Evaluation (LREC). – 2024. – Режим доступу: https://aclanthology.org/2024.lrec-main.306.pdf
8. Lee S., Kumar D., Zhao H. LLMRefactor: Improving Software Refactoring via Large Language Models [Електронний ресурс]. – 2024. – Режим доступу: https://arxiv.org/pdf/2406.20060

References

1. Chen W., Liu J., Li Y. AI-Driven Code Refactoring: Using Graph Neural Networks to Enhance Software Maintainability [Electronic resource]. – 2025. – DOI: https://doi.org/10.48550/arXiv.2504.10412
2. Zhang T., Wang H., Zhou F. MANTRA: Enhancing Automated Method-Level Refactoring with Contextual RAG and Multi-Agent LLM Collaboration [Electronic resource]. – 2025. – DOI: https://doi.org/10.48550/arXiv.2503.14340
3. Kumar A., Singh P. A Survey of Deep Learning Based Software Refactoring [Electronic resource]. – 2024. – DOI: https://doi.org/10.48550/arXiv.2404.19226
4. Garcia M., Patel R., Choudhury S. A Survey on Large Language Models for Code Generation [Electronic resource]. – 2024. – DOI: https://doi.org/10.48550/arXiv.2406.00515
5. Smith J., Tan L., Wu H. Scalable Static Analysis for AI Codebases [Electronic resource]. – 2025. – Access mode: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-50.pdf
6. Nguyen T., Lopez M., Rossi G. AI-Assisted Refactoring with LLMs in Web Applications [Electronic resource] // Applied Sciences. – 2025. – T. 15, №12. – DOI: https://doi.org/10.3390/app15126524
7. Yoon J., Bauer M., Nikolov A. Corpus Construction for Software Engineering LLM Evaluation [Electronic resource] // Proceedings of the 2024 International Conference on Language Resources and Evaluation (LREC). – 2024. – Access mode: https://aclanthology.org/2024.lrec-main.306.pdf
8. Lee S., Kumar D., Zhao H. LLMRefactor: Improving Software Refactoring via Large Language Models [Electronic resource]. – 2024. – Access mode: https://arxiv.org/pdf/2406.20060