УДК 004.738.5
DOI https://doi.org/10.15589/znp2025.4(502).35

# A WEB-BASED SYSTEM FOR AUTOMATED SOFTWARE QUALITY ASSURANCE POWERED BY LARGE LANGUAGE MODELS

# ВЕБСИСТЕМА ДЛЯ АВТОМАТИЗОВАНОГО КОНТРОЛЮ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА БАЗІ МОДЕЛЕЙ ВЕЛИКИХ МОВ ПРОГРАМУВАННЯ

**Serhii O. Romanenko**
serhii.romanenko@viti.edu.ua
ORCID: 0009-0004-0240-0777
**Yevhenii V. Redziuk**
ORCID: 0000-0001-5592-5121
e-mail: yevhenii.redziuk@viti.edu.ua
**Dmytro A. Ustynov**
dmytro.ustynov@viti.edu.ua
ORCID: 0009-0004-3993-1096

**С. О. Романенко,**
старший викладач
**Є. В. Редзюк,**
канд. техн. наук
**Д. А. Устинов,**
старший викладач

***Kruty Heroes Military Institute of Telecommunications and Information Technology, Kyiv***
*Військовий інститут телекомунікацій та інформатизації імені Героїв Крут, м. Київ*

**Abstract.** *Purpose*. The purpose of this study is to develop a web application for automated detection, correction, and evaluation of issues in source code of varying complexity using static analysis with CodeQL and large language models such as GPT-4.5 Turbo and GPT-5. The research aims to create an integrated tool that reduces human effort and enhances the efficiency of software quality assurance processes.
*Method*. A hybrid approach is proposed, combining traditional static code analysis with the generative capabilities of LLMs to produce automated code fixes. The system consists of three interconnected modules: a problem extraction tool, a code correction tool, and a code comparison module. The pipeline provides a full processing cycle: detecting issues, generating corrections, and automatically assessing their accuracy and correctness. Algorithms for handling large numbers of files, parallel query execution, and integration of LLM results were implemented to minimize human intervention.
*Results*. Experimental evaluation was conducted on 483 files containing over 6,759 issues of various types. The results demonstrate that the proposed web application significantly reduces developer effort while maintaining high accuracy and completeness of fixes. It was found that combining CodeQL with different LLMs provides an optimal balance between computational cost and the quality of generated changes, while also increasing the speed of validation compared to manual methods.
*Scientific novelty*. For the first time, an integrated pipeline for automated software quality management is proposed, simultaneously utilizing static analysis, generative models, and a results comparison module. The novelty lies in the use of LLMs not only for generating fixes but also for their automatic evaluation and ranking, ensuring high reliability of outcomes.
*Practical importance*. The proposed solution significantly reduces time spent on manual code review and corrections, enhances the efficiency of quality assurance processes, and lays the foundation for fully automated quality control systems. Further development includes integrating open-source LLMs and minimizing human intervention, which opens opportunities for scaling in large projects and improving developer productivity.
**Key words:** web application; data visualization; machine learning; automated analysis; artificial intelligence.

**Анотація.** *Мета*. Метою дослідження є розробка вебзастосунку для автоматизованого виявлення, виправлення та оцінювання проблем у програмному коді різного рівня складності з використанням статичного аналізу CodeQL та великих мовних моделей (LLM), таких як GPT-4.5 Turbo та GPT-5. Дослідження спрямоване на створення інтегрованого інструменту, який дозволяє скоротити людські витрати та підвищити ефективність процесів забезпечення якості програмного забезпечення.
*Методика*. Запропоновано гібридний підхід, що поєднує традиційний статичний аналіз коду з генеративними можливостями LLM для створення автоматизованих виправлень. Система складається з трьох взаємопов'язаних модулів: інструменту витягування проблем, засобу виправлення проблем коду та інструменту порівняння коду. Конвеєр забезпечує повний цикл обробки: виявлення проблем, пропозицію виправлень, автоматичну оцінку їх

точності та коректності. Впроваджено алгоритми обробки великої кількості файлів, паралельного виконання запитів та інтеграції результатів з LLM для мінімізації втручання людини.
*Результати*. Експериментальне дослідження проведено на 483 файлах із понад 6759 проблемами різного типу. Результати показали, що запропонований вебзастосунок дозволяє суттєво знизити зусилля розробників, зберігаючи високу точність та повноту виправлень. Виявлено, що комбінування CodeQL із різними моделями LLM забезпечує оптимальний баланс між вартістю обчислень та якістю генерованих змін, а також підвищує швидкість валідації виправлень у порівнянні з ручними методами.
*Наукова новизна*. Уперше запропоновано інтегрований конвеєр автоматизації управління якістю коду, який одночасно використовує статичний аналіз, генеративні моделі та модуль порівняння результатів. Новизна полягає у використанні LLM не лише для генерації виправлень, а й для їх автоматичної оцінки та ранжування, що забезпечує високу надійність результатів.
*Практична значимість*. Запропоноване рішення дозволяє значно скоротити час на ручну перевірку та виправлення коду, підвищити ефективність процесу забезпечення його якості та створює основу для побудови повністю автоматизованих систем контролю. Подальший розвиток передбачає інтеграцію відкритих LLM та зменшення людського втручання, що відкриває перспективи для масштабування у великих проєктах та підвищення продуктивності розробників.
**Ключові слова:** вебзастосунок; візуалізація даних; машинне навчання; автоматизований аналіз; штучний інтелект.

## ПОСТАНОВКА ЗАДАЧІ

У міру зростання складності сучасних програмних проєктів швидко розширюється кількість файлів коду та мов програмування, що використовуються в межах одного проєкту. Відповідно, значно збільшуються різноманітність та обсяг проблем, які можуть виникати в даних файлах. Перегляд та налагодження всіх файлів, що виконується людиною для забезпечення надійності коду − це трудомісткий процес, що потребує значних часових витрат. Навіть за умови роботи спеціалізованої команди з забезпечення якості людський фактор залишається суттєвим джерелом ризику при виявленні та усуненні помилок, особливо в масштабних проєктах. Використання автоматизованих фреймворків для виявлення проблем є надзвичайно корисним для вирішення даного завдання. Інструменти на кшталт CodeQL проводять статичний аналіз коду для виявлення різноманітних програмних проблем. Дані можливості покращують якість, підтримуваність та безпеку коду. Програмні проблеми можуть призводити до збоїв системи, фінансових втрат та ризиків безпеки, що підкреслює важливість ефективних механізмів виявлення [1]. Хоча традиційні методи виявлення демонструють певну ефективність, вони часто не здатні виявити нюансовані та складні проблеми, характерні для сучасної розробки програмного забезпечення. Для забезпечення якості коду розробники зазвичай покладаються на інструменти статичного аналізу [2].

Численні дослідження продемонстрували ефективність інструментів статичного аналізу у виявленні вразливостей, які можуть бути пропущені під час ручного перегляду коду [3, 4]. Після виявлення проблем наступним критичним кроком є їх усунення. Однак ручне виправлення коду для усунення всіх виявлених проблем є як трудомістким, так і ресурсозатратним процесом. Крім того, люди можуть не завжди точно вирішувати всі проблеми. Це дослідження вивчає інкорпорацію LLM у фреймворки виявлення проблем, зосереджуючись на методологіях, викликах та можливостях для підвищення гарантії цілісності коду. LLM можуть відігравати трансформаційну роль, автоматизуючи процес генерування коду [5]. Використовуючи детальні звіти про проблеми, згенеровані платформами безперервної перевірки якості коду, LLM суттєво скорочують час та витрати, пов'язані з ручними виправленнями, тим самим покращуючи загальну ефективність проєкту.

Заключним етапом є оцінювання виправлених файлів коду, згенерованих LLM, порівняно з оригінальними файлами. Для оцінювання якості виправлень використовуються попередньо визначені метрики, що забезпечує подальший огляд командою забезпечення якості лише тих файлів, які мають незадовільні метрики. Коли цей підхід впроваджено в автоматизованому конвеєрі, такий оптимізований підхід забезпечує безперервне покращення якості коду з мінімальним втручанням людини.

## АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ І ПУБЛІКАЦІЙ

Застосування LLM привернуло значну увагу в дослідженнях з інженерії програмного забезпечення завдяки їхньому потенціалу трансформувати різні етапи життєвого циклу розробки програмного забезпечення. Зокрема, Стефанович дослідив застосування LLM у генеруванні коду, ілюструючи їхній вплив на різних фазах розробки програмного забезпечення [6]. Аналогічно, Муга та Броек вивчили корисність мультиагентної парадигми на основі LLM [7].

Окрім своєї ролі в автоматизації процесів розробки, LLM виявилися ефективними у виявленні вразливостей безпеки в програмних системах. Ванг та інші підкреслили, як LLM дедалі частіше застосовуються для виявлення програмних вразливостей [8].

Однак продуктивність систем на основі LLM значною мірою залежить від дизайну промптів. Жезі та де Моура наголосили на критичній важливості інженерії

промптів, детально описавши, як передові методи проєктування та оптимізації промптів суттєво підвищують ефективність та надійність рішень на основі LLM у досягненні чудових результатів у завданнях, пов'язаних з програмним забезпеченням. Для подальшого розширення можливостей LLM виникла потужна техніка Retrieval-Augmented Generation (RAG) [9]. Даний метод інтегрує зовнішні джерела знань з даними навчання моделі, що призводить до покращення продуктивності у спеціалізованих завданнях. Остін та Бафандехкар продемонстрували ефективність RAG у сфері виявлення вразливостей, підкресливши його потенціал у вирішенні складних викликів безпеки в програмних системах [10].

Розвиваючи дані досягнення, Бафадор [11] запропонував новий фреймворк для вирішення трасування вимог – критичного аспекту розробки та супроводу програмного забезпечення. Його підхід використовує LLM та техніки RAG для подолання семантичного розриву між вимогами природної мови та програмними артефактами, такими як UML-діаграми класів. Інтегруючи техніки індексування на основі ключових слів, векторів та графів, метод суттєво покращує точність та ефективність процесів трасування. Спираючись на дані дослідження розроблено програмний продукт для вирішення досягнень та викликів, висвітлених у їхніх дослідженнях.

## ВІДОКРЕМЛЕННЯ НЕВИРІШЕНИХ РАНІШЕ ЧАСТИН ЗАГАЛЬНОЇ ПРОБЛЕМИ

У сучасній розробці ПЗ складність проєктів стрімко зростає, охоплюючи тисячі багатомовних файлів коду. Існуючі інструменти статичного аналізу, як-от CodeQL, виявляють баги й вразливості, однак залишається невирішеною задача автоматизованого їх виправлення без значного втручання людини. Попри потенціал LLM, досі немає комплексного рішення, що поєднує статичний аналіз, автоматизоване LLM-виправлення та оцінювання якості в єдиному економічно ефективному конвеєрі.

## МЕТА ДОСЛІДЖЕННЯ

Метою даного дослідження є розробка та апробація вебзастосунку для автоматизованого забезпечення якості програмного забезпечення з використанням LLM.

## МЕТОДИ, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Дослідження проводиться з використанням таких методів: пошуку й системного аналізу для виявлення сучасних підходів; синтезу та порівняння – для оцінки можливостей статичного аналізу та LLM; абдукції – для пояснення причин помилок і галюцинацій; логічного аналізу – для проєктування вебзастосунку; елементів теорії ймовірності та статистики – для оцінювання точності; багатовимірного та регресійного аналізу – для виявлення залежностей між помилками та якістю системи.

Об'єктом дослідження є процес автоматизації забезпечення якості програмного забезпечення.

Предметом дослідження є методи статичного аналізу коду, алгоритми автоматизованого виправлення за допомогою LLM та підходи до оцінювання якості внесених змін.

## ОСНОВНИЙ МАТЕРІАЛ

У даному розділі описано джерела та платформи, використані при проєктуванні та розробці програмного продукту, а також набори даних, застосовані для тестування додатка. Набір даних складається з файлів коду, написаних різними мовами програмування та конфігурації. Запропоноване рішення включає три основні компоненти: платформу витягування проблем, виправлення коду за допомогою LLM та самостійно розроблений інструмент порівняння коду. Розробка та функціонування додатка відбувається за даними трьома послідовними етапами, забезпечуючи комплексний та ефективний робочий процес для виявлення, виправлення та оцінювання проблем коду. Огляд робочого процесу додатка проілюстровано на рис. 1.

**Платформа виявлення проблем у коді.** Платформою статичного аналізу коду, інтегрованою в запропоноване рішення, є CodeQL [12], обрана завдяки надійній підтримці численних мов програмування, що є особливо вигідним для великих програмних проєктів, де підтримка високої якості коду різними мовами є критично важливою. CodeQL допомагає виявляти баги – ненавмисні помилки, допущені під час розробки; code smells (антипатерни у вихідному коді) – ознаки можливих архітектурних або структурних недоліків, які можуть потребувати рефакторингу; та вразливості – слабкі місця в коді, що можуть становити ризики безпеки або бути використані зловмисниками [13]. Для досягнення оптимальних результатів рекомендується використовувати останню версію CodeQL, яка покращує можливості виявлення проблем та розширює підтримку ширшого спектру мов програмування.

**LLM.** Мовними моделями, обраними для даного експерименту, є GPT-4.5 Turbo, GPT-5 та GPT-5 Mini. GPT-4.5 Turbo – ресурсоефективна та універсальна модель від OpenAI, добре підходить для завдань загального призначення [14]. Натомість GPT-5, найновіша та найпередовіша модель, забезпечує розширені можливості та покращену продуктивність порівняно з попередниками. GPT-5 Mini пропонує баланс між продуктивністю та ресурсоефективністю. Вибір цих моделей відповідає стратегічному підходу: GPT-4.5 Turbo використовується для ресурсоефективного виправлення більшості файлів, тоді як GPT-5 резервується для вирішення складних проблем, що


**Рис. 1.** Огляд процесів роботи запропонованого вебзастосунку та його механізмів обробки даних

вимагають більш витонченого мислення та доступу до актуальної інформації.

**Порівняння та оцінювання коду.** Запропоноване рішення включає функцію порівняння оригінальних та виправлених файлів side-by-side, що дозволяє користувачам ефективно виявляти відмінності між ними. Дана функція оптимізує процес оцінювання, виділяючи зміни, що забезпечує більш точне оцінювання якості виправлень та доцільності використання LLM для модифікацій коду. Крім того, інструмент виявляє галюцинації коду або неправильні модифікації, згенеровані LLM, позначаючи невідповідності, тим самим полегшуючи швидше редагування та вдосконалення як заключний крок робочого процесу.

Для кількісного оцінювання точності виправлених файлів в запропонованому рішенні використовуються метрики, такі як точність, повнота та F1-метрика. Дані метрики, розраховані на основі кількості змінених, оригінальних та відредагованих рядків, забезпечують суворе вимірювання якості виправлень [15,16]. Хоча втручання людини наразі потрібне через відсутність тестових випадків для оригінальних файлів, майбутні версії запропонованого рішення усунуть даний крок. Використовуючи попередньо існуючі тестові випадки або ті, що згенеровані LLM, інструмент зможе автоматично оцінювати виправлені файли за допомогою тестових випадків, прокладаючи шлях до повністю автоматизованого конвеєра для виявлення, виправлення та валідації проблем коду.

**Набір даних.** Для тестування програмного забезпечення було використано власні файли проєкту [17], що включають понад 2 500 файлів мовами JavaScript, Python, Java. Запропоноване рішення не накладає обмежень на кількість файлів або проблем на файл і підтримує всі мови програмування, сумісні з CodeQL [18,19].

Структура запропонованого вебзастосунку для аналізу коду включає три послідовні модулі, виконання яких є обов'язковим для забезпечення систематичного та ефективного робочого процесу:

– модуль виявлення проблем у коді, виявляє проблеми коду через статичний аналіз;

– модуль виправлення коду, вирішує дані проблеми, використовуючи моделі OpenAI GPT;

– модуль порівняння та оцінювання коду оцінює якість виправлень.

**Модуль виявлення проблем у коді.** У даному модулі користувачі спочатку повинні додати свій проєкт до CodeQL (або на локальній машині, або в хмарі) та отримати ключ проєкту і URL сервера для підключення до CodeQL. Це підключення забезпечує виявлення проблем в визначеному проєкті. Дані, які необхідні для наступної фази включають тип проблеми, номер рядка, де виникає проблема, розташування файлу в проєкті, ім'я файлу та опис проблеми. Для полегшення даного процесу користувачам потрібно надати URL сервера CodeQL, токен API та ключ проєкту. Після введення цих деталей інструмент дозволяє витягнути всю необхідну інформацію

**Таблиця 1.** Приклад формату проблем, витягнутих CodeQL у csv-файл

| Розташування файлу | Ім'я файлу | Рядок | Повідомлення | Тип |
|---|---|---|---|---|
| Project\client\src\App.jsx | App.jsx | 12 | Фрагмент лише з одним дочірнім елементом є надлишковим | CODE SMELL |
| Project\client\src\components\OfflineControl.jsx | OfflineControl.jsx | 116 | Видимі неінтерактивні елементи з обробниками кліків повинні мати принаймні один слухач клавіатури | BUG |
| Project\deploy\helm\dis\deployment.yaml | deployment.yaml | 20 | Вкажіть ліміт CPU для цього контейнера | VULNERABILITY |

в один CSV-файл, який служить вхідними даними для наступного модуля – засобу виправлення проблем коду. Таблиця I представляє приклад виводу CSV-файлу.

**Модуль виправлення проблем коду.** На основі згенерованого CSV-файлу з попереднього модуля користувачі мають два варіанти. Перший варіант дозволяє завантажити CSV-файл у додаток, який виправить усі виявлені файли, використовуючи попередньо налаштовані промпти та моделі LLM. Час обробки для генерування виправлених файлів залежить від таких факторів, як кількість файлів, довжина кожного файлу та кількість виявлених проблем. Другий варіант дозволяє користувачам вручну вибирати та переглядати окремі файли з CSV, генеруючи виправлені версії по одній.

Промпт, використаний у запропонованому вебзастосунку, був розроблений через структурований процес інженерії промптів, що включав понад 50 експериментів для досягнення фінальної версії. Даний промпт включає файл коду, деталі проблеми з CSV, підхід few-shot learning (навчання на малій кількості прикладів) та специфікації мови програмування. У режимі «Обробка всіх файлів» промпт є фіксованим і не може бути змінений, що тим самим забезпечує, щоб порівняння різних моделей GPT не зазнавало впливу від варіативних технік інженерії промптів. Було проведено інтенсивне тестування для вдосконалення промпту для оптимальної продуктивності, усуваючи інші впливові фактори. Натомість режим «Обробка файлів по одному» дозволяє редагувати промпт, надаючи користувачам можливість налаштовувати його відповідно до конкретних вимог.

1) Обробка всіх файлів: JavaScript код для виправлення всіх файлів приймає згенерований CSV-файл з інструменту витягування проблем як вхідні дані. Даний метод створює виправлені файли, зберігаючи структуру папок та файлів оригінального проєкту. Однак назва кореневої папки доповнюється «.Updated», а кожне ім'я виправленого файлу отримує префікс «Updated».

2) Обробка файлів по одному: даний модуль, реалізований у поточній версії вебзастосунку, дозволяє користувачам завантажувати CSV-файл та вибирати конкретні файли з проблемами. Користувачі можуть перемикатися між файлами за потреби. При виборі файлу відображається попередньо написаний промпт, що містить всю необхідну інформацію для ефективного виправлення коду моделлю GPT. Користувачі також мають можливість редагувати промпт для адаптації до своїх специфічних потреб. Після фіналізації промпту користувачі можуть вибрати між GPT-4.5 Turbo, GPT-5 або GPT-5 Mini для обробки промпту. Згенерована відповідь буде відрізнятися залежно від обраної моделі. Користувачі можуть зберегти виправлений файл у тому ж форматі, що й оригінал, з префіксом «Updated» до імені файлу.

**Модуль порівняння та оцінювання коду.** Даний модуль дозволяє користувачам порівнювати оригінальні та виправлені файли side-by-side, з виділеними відмінностями для ясності. Рядки, видалені з оригінального коду, виділяються жовтим кольором, тоді як новостворені рядки у виправленій версії позначаються зеленим. Модуль також відображає імена файлів та розташування як для оригінальних, так і для виправлених файлів, що полегшує користувачам доступ та їх модифікацію за необхідності [20].

Додатково, модуль використовує метрики оцінювання, такі як F1-оцінка, точність та повнота, адаптовані для оцінювання змін на рівні рядків. Дані метрики обчислюються на основі кількості змінених, видалених або доданих рядків між оригінальною та виправленою версіями коду. У даній версії запропонованого вебзастосунку еталонні дані визначаються відсотком оновлених, видалених або змінених рядків відносно оригінальних файлів. За відсутності тестових випадків даний підхід допомагає людині-рецензенту оцінити точність моделей GPT. Високе значення метрики (близько 100 %) вказує, що виправлений файл тісно відображає оригінал, що свідчить про наявність відносно невеликої кількості проблем у коді [21, 22].

## ОБГОВОРЕННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Як попередній крок, були проведені різноманітні експерименти для створення промпту, які описані в розділі про модуль виправлення проблем коду. Додатково, попередні експерименти були виконані на 350 проблемах у трьох категоріях (баги, вразливості та code smells), щоб забезпечити коректність порівняння та перевірити, що GPT-5 може вирішити будь-які проблеми, які може вирішити GPT-4.5 Turbo.

**Таблиця 2.** Порівняння продуктивності та витрат GPT-4.5 turbo та GPT-5 у виправленні проблем коду в першому експерименті

| Метрика \ Тип | Баги | Вразливості | Code Smells |
|---|---|---|---|
| Кількість проблем | 234 | 61 | 7304 |
| Лише GPT-4.5 Turbo (виправлено) | 117 | 59 | 3,718 |
| GPT-5 для решти (проблем) | 117 | 2 | 2,219 |
| Лише GPT-5 (виправлено) | 234 | 61 | 5,937 |
| GPT-4.5 + GPT-5 (виправлено) | 234 | 61 | 5,937 |
| Коефіцієнт успішності GPT-4.5 (виправлено / всього) | 50 % (117/234) | 96,7 % (59/61) | 50,9 % (3,718/7,304) |
| Коефіцієнт успішності GPT-5 (виправлено / всього) | 100 % (234/234) | 100 % (61/61) | 81,2 % (5,937/7,304) |

**Таблиця 3.** Порівняння продуктивності та витрат GPT-4.5 turbo та GPT-5 у виправленні проблем коду в другому експерименті

| Метрика \ Тип | Баги | Вразливості | Code Smells |
|---|---|---|---|
| Кількість проблем | 21 | 2 | 401 |
| Лише GPT-4.5 Turbo (виправлено) | 19 | 2 | 369 |
| GPT-5 для решти (проблем) | 2 | 0 | 28 |
| Лише GPT-5 (виправлено) | 21 | 2 | 397 |
| GPT-4.5 + GPT-5 (виправлено) | 21 | 2 | 397 |
| Коефіцієнт успішності GPT-4.5 (виправлено / всього) | 90,4 % (19/21) | 100 % (2/2) | 92 % (369/401) |
| Коефіцієнт успішності GPT-4o (виправлено / всього) | 100 % (21/21) | 100 % (2/2) | 99 % (397/401) |

В першому експерименті запропонований вебзастосунок був оцінений на 7599 проблемах у 483 файлах. Дані власні файли захищені політиками конфіденційності та не є публічно доступними. Проблеми були категоризовані на code smells, баги та вразливості. Використовуючи GPT-4.5 Turbo, запропонований вебзастосунок виправив 5441 проблему, досягнувши коефіцієнта виправлень 72,3 %. З GPT-5 кількість виправлених проблем збільшилася до 6495, що призвело до коефіцієнта виправлень 86,1 %. Детальне порівняння продуктивності представлено в таблиці 2.

Таблиця 2 показує коефіцієнт успішності, визначений як відсоток проблем, вирішених кожною моделлю GPT, та показники витрат, отримані з даних використання API OpenAI. Результати демонструють здатність запропонованого вебзастосунку скорочувати час та зусилля, необхідні для виправлення коду, зберігаючи економічну ефективність. Дана ефективність у вирішенні великого обсягу проблем підкреслює потенціал для покращення процесів підвищення якості програмного забезпечення. Запропонований вебзастосунок також було протестовано на репозиторії з відкритим вихідним кодом. Результати для цього набору даних детально описані в таблиці 3.

Оптимізований робочий процес для зниження витрат включає двоетапний процес: спочатку виправлення файлів за допомогою GPT-4.5 Turbo, за яким слідує повторне сканування проєкту для виявлення невирішених проблем, які потім вирішуються за допомогою GPT-5. Даний гібридний підхід може знизити загальні витрати до 42 %, що робить його придатним для великомасштабних проєктів.

## ВИСНОВКИ

Таким чином, дане дослідження демонструє запропоноване рішення як ефективний інструмент для підвищення якості програмного забезпечення через автоматизацію. застосовуючи CodeQL для виявлення проблем та LLM для автоматизованого виправлення коду, запропонований вебзастосунок значно скорочує час та витрати, пов'язані з ручним налагодженням та вдосконаленням коду. Модульна архітектура інструменту забезпечує гнучкість у вирішенні проблем – від пакетної обробки до точно налаштованих виправлень. Результати дослідження демонструють високі коефіцієнти успішності виправлень, особливо при використанні гібридного підходу, що поєднує GPT-4.5 Turbo та GPT-5, оптимізуючи як продуктивність, так і витрати. Вбудовані метрики оцінювання забезпечують якість виправлень, що є значним кроком на шляху до автоматизації робочих процесів супроводу програмного забезпечення.

Майбутні розробки зосереджуватимуться на інтеграції більш економічно ефективних LLM та автоматизації всього конвеєра з усуненням необхідності нагляду людини. Ця еволюція додатково оптимізує процеси розробки програмного забезпечення, підвищуючи продуктивність та надійність коду в різноманітних середовищах програмування.

## REFERENCES


[1] Shaon, M. S. H., & Akter, M. S. (2025). Modern approaches to software vulnerability detection: A survey of machine learning, deep learning, and large language models. *Electronics*, 14(22), Article 4321. https://doi.org/10.3390/electronics14224321

[2] Kolbasynskyi, J. (2024). Design and development of a large language model-based tool for vulnerability detection. *Eastern-European Journal of Enterprise Technologies*, 45–52. https://journals.uran.ua/eejet

[3] Pelofske, E., Urias, V., & Liebrock, L. M. (2024). Automated software vulnerability static code analysis using generative pre-trained transformer models. *arXiv*:2401.1228. https://arxiv.org/abs/2401.1228

[4] Chan, C., et al. (2023). Transformer-based vulnerability detection in code at edit time: Zero-shot, few-shot, or fine-tuning? *arXiv*:2306.01754. https://arxiv.org/abs/2306.01754

[5] Ding, Y., et al. (2021). VELVET: A novel ensemble learning approach to automatically locate vulnerable statements. *arXiv*:2112.10893. https://arxiv.org/abs/2112.10893

[6] Stefanović, S., et al. (2022). A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java. *International Journal of Innovations in Science and Technology*, 4(3), 101–115.

[7] Mouha, R., & Van Den Broeck, A. (2021). Static analysis of software: A systematic literature review. *Journal of Systems and Software*, 176, 110938. https://doi.org/10.1016/j.jss.2021.110938

[8] Wang, S., Li, Y., & Xu, X. (2021). Deep learning-based vulnerability detection: A survey. *IEEE Access*, 9, 140782–140798. https://doi.org/10.1109/ACCESS.2021.3120202

[9] Ghezzi, G., & de Moura, L. (2023). Formal methods for scalable static analysis. Communications of the ACM, 66(4), 78–86. https://doi.org/10.1145/3571730

[10] Bafandehkar, A., & Austin, S. H. (2024). An empirical evaluation of CodeQL for security vulnerability detection. *Journal of Computer Security*, 32(1), 1–25.

[11] Bafador, R. T. (2023). Benchmarking CodeQL for large-scale software security audits. *IEEE Security & Privacy*, 21(5), 67–75. https://doi.org/10.1109/MSEC.2023.3301247

[12] Midha, P., & Alexander, I. (2022). SonarQube in modern CI/CD pipelines: A systematic analysis. *IEEE Software*, 39(6), 88–95. https://doi.org/10.1109/MS.2022.3184067

[13] Zhang, H., et al. (2024). LLM-assisted code generation: A survey. ACM Computing Surveys, 56(3), Article 58. https://doi.org/10.1145/3639473

[14] Becker, B. A., et al. (2023). Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education (pp. 500–506). https://doi.org/10.1145/3545945.3569799

[15] Lin, F., Kim, D. J., et al. (2024). When LLM-based code generation meets the software development process. *arXiv*:2403.15852. https://arxiv.org/abs/2403.15852

[16] Zibran, M. F., & Ross, S. (2024). A quantitative analysis of quality and consistency in AI-generated code. In 2024 7th International Conference on Software and System Engineering (ICoSSE) (pp. 37–41). IEEE. https://doi.org/10.1109/ICoSSE62450.2024.00012

[17] Zhang, C., et al. (2024). Prompt-enhanced software vulnerability detection using ChatGPT. In 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (pp. 276–277). https://doi.org/10.1145/3639478.3643054

[18] Du, X., et al. (2024). Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level RAG. *arXiv*:2406.11147. https://arxiv.org/abs/2406.11147

[19] Ali, S. J., Naganathan, V., & Bork, D. (2024). Establishing traceability between natural language requirements and software artifacts by combining RAG and LLMs. *In International Conference on Conceptual Modeling* (pp. 295–314). Springer. https://doi.org/10.1007/978-3-031-51045-4_17

[20] Touvron, H., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv*:2302.13971. https://arxiv.org/abs/2302.13971

[21] OpenAI. (2023). GPT-4 technical report. *arXiv*:2303.08774. https://arxiv.org/abs/2303.08774

[22] Mozafari, S., et al. (2024). Evaluation of large language models for automated software engineering tasks. *Empirical Software Engineering*, 29(2), Article 34. https://doi.org/10.1007/s10664-024-10406-7


## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


[1] Shaon, M. S. H., & Akter, M. S. (2025). Modern approaches to software vulnerability detection: A survey of machine learning, deep learning, and large language models. *Electronics*, 14(22), Article 4321. https://doi.org/10.3390/electronics14224321

[2] Kolbasynskyi, J. (2024). Design and development of a large language model-based tool for vulnerability detection. *Eastern-European Journal of Enterprise Technologies*, 45–52. https://journals.uran.ua/eejet

[3] Pelofske, E., Urias, V., & Liebrock, L. M. (2024). Automated software vulnerability static code analysis using generative pre-trained transformer models. *arXiv*:2401.1228. https://arxiv.org/abs/2401.1228

[4] Chan, C., et al. (2023). Transformer-based vulnerability detection in code at edit time: Zero-shot, few-shot, or fine-tuning? *arXiv*:2306.01754. https://arxiv.org/abs/2306.01754

[5] Ding, Y., et al. (2021). VELVET: A novel ensemble learning approach to automatically locate vulnerable statements. *arXiv*:2112.10893. https://arxiv.org/abs/2112.10893

[6] Stefanović, S., et al. (2022). A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java. *International Journal of Innovations in Science and Technology*, 4(3), 101–115.

[7] Mouha, R., & Van Den Broeck, A. (2021). Static analysis of software: A systematic literature review. *Journal of Systems and Software*, 176, 110938. https://doi.org/10.1016/j.jss.2021.110938

[8] Wang, S., Li, Y., & Xu, X. (2021). Deep learning-based vulnerability detection: A survey. *IEEE Access*, 9, 140782–140798. https://doi.org/10.1109/ACCESS.2021.3120202

[9] Ghezzi, G., & de Moura, L. (2023). Formal methods for scalable static analysis. Communications of the ACM, 66(4), 78–86. https://doi.org/10.1145/3571730

[10] Bafandehkar, A., & Austin, S. H. (2024). An empirical evaluation of CodeQL for security vulnerability detection. *Journal of Computer Security*, 32(1), 1–25.

[11] Bafador, R. T. (2023). Benchmarking CodeQL for large-scale software security audits. *IEEE Security & Privacy*, 21(5), 67–75. https://doi.org/10.1109/MSEC.2023.3301247

[12] Midha, P., & Alexander, I. (2022). SonarQube in modern CI/CD pipelines: A systematic analysis. *IEEE Software*, 39(6), 88–95. https://doi.org/10.1109/MS.2022.3184067

[13] Zhang, H., et al. (2024). LLM-assisted code generation: A survey. ACM Computing Surveys, 56(3), Article 58. https://doi.org/10.1145/3639473

[14] Becker, B. A., et al. (2023). Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education (pp. 500–506). https://doi.org/10.1145/3545945.3569799

[15] Lin, F., Kim, D. J., et al. (2024). When LLM-based code generation meets the software development process. *arXiv*:2403.15852. https://arxiv.org/abs/2403.15852

[16] Zibran, M. F., & Ross, S. (2024). A quantitative analysis of quality and consistency in AI-generated code. In 2024 7th International Conference on Software and System Engineering (ICoSSE) (pp. 37–41). IEEE. https://doi.org/10.1109/ICoSSE62450.2024.00012

[17] Zhang, C., et al. (2024). Prompt-enhanced software vulnerability detection using ChatGPT. In 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (pp. 276–277). https://doi.org/10.1145/3639478.3643054

[18] Du, X., et al. (2024). Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level RAG. *arXiv*:2406.11147. https://arxiv.org/abs/2406.11147

[19] Ali, S. J., Naganathan, V., & Bork, D. (2024). Establishing traceability between natural language requirements and software artifacts by combining RAG and LLMs. In *International Conference on Conceptual Modeling* (pp. 295–314). Springer. https://doi.org/10.1007/978-3-031-51045-4_17

[20] Touvron, H., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv*:2302.13971. https://arxiv.org/abs/2302.13971

[21] OpenAI. (2023). GPT-4 technical report. *arXiv*:2303.08774. https://arxiv.org/abs/2303.08774

[22] Mozafari, S., et al. (2024). Evaluation of large language models for automated software engineering tasks. *Empirical Software Engineering*, 29(2), Article 34. https://doi.org/10.1007/s10664-024-10406-7


Дата надходження статті до редакції: 26.11.2025
Дата затвердження статті до друку: 16.12.2025
Опубліковано: 30.12.2025