УДК 004.4′2 DOI https://doi.org/10.35546/kntu2078-4481.2025.2.2.36

В. С. РАЛЕНКО
викладач кафедри інженерії програмного забезпечення
Чорноморський національний університет імені Петра Могили
ORCID: 0009-0009-4161-8468

К. О. АНТІПОВА
доктор філософії,
доцент (б.в.з.) кафедри інженерії програмного забезпечення
Чорноморський національний університет імені Петра Могили
ORCID: 0000-0002-9012-5290
Scopus-Author ID: 57212609599

АНАЛІЗ ПЕРЕВАГ ТА НЕДОЛІКІВ ГЕНЕРАЦІЇ КОДУ ІЗ ВИКОРИСТАННЯМ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ В СУЧАСНИХ IDE

У роботі розглянуто способи використання великих мовних моделей для генерації програмного коду. Основна увага приділяється IDE та вбудованим в них системам штучного інтелекту для використання в таких завданнях, як створення проєктів, побудова архітектури програмних застосунків та прототипування застосунків за запитами до моделей. Коду, згенерованому штучним інтелектом, бракує відстежуваності, він часто потребує ітеративного доопрацювання – виклики, з якими існуючі IDE не здатні впоратися. На відміну від коду, створеного людиною, згенерований код погано інтегрується зі структурованими робочими процесами розробки. Як наслідок, розробники борються з галюцинаціями ШІ, коли моделі генерують правдоподібний, але некоректний код, який важко перевірити. У рамках цього дослідження виявлено ситуації некоректної роботи великих мовних моделей як неправильна інтерпретація намірів користувача, галюцинація моделі, неправильне використання API, пов'язане з відсутніми елементами, та використання надлишкової інформації. Зловживання сторонніми API у згенерованому коді є серйозною проблемою у розробці програмного забезпечення. Хоча моделі продемонстрували вражаючі можливості генерації коду, їхня взаємодія зі складними бібліотечними API залишається дуже схильною до появи помилок, що потенційно може призвести до збоїв у роботі програмного забезпечення та вразливостей у безпеці. Результати дослідження демонструють, що найкраще використовувати автозаповнення коду, так як воно надає можливість згенерувати код за запитом із дотриманням заданих обмежень та з меншою ймовірністю появи помилок. Але, серед недоліків такого підходу можна виявити відсутність можливості генерувати код відразу в декількох файлах. Отримані результати сприятимуть оптимальному вибору варіантів використання штучного інтелекту та великих мовних моделей в розробці застосунків. Ця робота спрямована на покращення розуміння сценаріїв використання великих мовних моделей в роботі з програмним кодом.

Ключові слова: *генерація коду, великі мовні моделі, інтегроване середовище розробки, ШІ-помічник.*

V. S. RALENKO
Lecturer at the Department of Software Engineering
Petro Mohyla Black Sea National University
ORCID: 0009-0009-4161-8468

K. O. ANTIPOVA
Ph.D., Associate Professor at the Department of Software Engineering
Petro Mohyla Black Sea National University
ORCID: 0000-0002-9012-5290
Scopus-Author ID: 57212609599

ADVANTAGES AND DISADVANTAGES OF GENERATING CODE USING LARGE LANGUAGE MODELS IN MODERN IDE

In this paper we discuss ways to use large language models to generate program code. The focus is on IDEs and their built-in artificial intelligence for such tasks as creating projects, building software application architectures, and prototyping applications based on prompts. AI-generated code lacks traceability and often requires iterative refinement, challenges that existing IDEs are unable to address. Unlike human-created code, generated code does not integrate well with structured development workflows. As a result, developers struggle with AI hallucinations, when models generate plausible but incorrect code that is difficult to verify. This research has identified situations of incorrect output of LLMs as intent misuse, hallucination, missing items misuse, and redundancy misuse. Misuse of third-party APIs in generated code

is a serious problem in software development. Although LLMs have demonstrated impressive code generation capabilities, their interaction with complex library APIs remains highly error-prone, potentially leading to software malfunctions and security vulnerabilities. The obtained results show that code autocomplete is more useful, as it makes it possible to generate code on demand in compliance with the specified restrictions and with a lower probability of errors. However, among the disadvantages of this approach is the lack of the ability to generate code in several files at once. The results obtained will contribute to the optimal choice of options for using AI and LLMs in application development. This work is aimed at improving the understanding of scenarios for using LLMs in working with program code.

Key words: *AI-assistant, code generation, IDE, LLM.*

Постановка проблеми

Багато великих мовних моделей (ВММ) мають можливість генерувати код, якщо дані, на яких вони навчаються, містять приклади коду. Інструменти ВММ добре підходять для розв'язання вступних завдань з програмування і добре показують себе у вступних курсах з інформатики. Зокрема, за наявності підказки, яка описує програму природною мовою, інструменти штучного інтелекту (ШІ) можуть генерувати функціональний код після однієї або декількох спроб.

Сучасні інтегровані середовища розробки (integrated development environment, IDE) були створені для коду, написаного людиною, де розробники контролюють процеси маніпулювання кодом, правки структуруються вручну, а зміни є наслідками передбачуваних робочих процесів. Однак код, згенерований ШІ, створюється динамічно, йому бракує відстежуваності, і він часто потребує ітеративного доопрацювання – виклики, з якими існуючі IDE не здатні впоратися. Поточні реалізації, такі як Copilot від GitHub, ChatGPT від OpenAI, Gemini від Google тощо, позиціонують ШІ як асистента або агента, який працює пліч-о-пліч з розробником. На відміну від коду, створеного людиною, код, згенерований ШІ, може бути досить непередбачуваним, йому не притаманне відстеження походження і він погано інтегрується зі структурованими робочими процесами розробки. Як наслідок, розробники борються з галюцинаціями ШІ, коли моделі генерують правдоподібний, але некоректний код, який важко перевірити.

Такі галюцинації часто пов'язані з послідовними та логічними міркуваннями або процесами глибокого пошуку, які керуються ШІ, але залишаються непрозорими для розробника. Низький рівень прозорості та відстежуваності може привести до втрати контролю розробниками, що ускладнює довіру до коду, згенерованого ШІ. Відсутність прозорості змушує розробників покладатися на пропозиції ШІ без повного розуміння їхньої логіки та аргументації рішень і того, чи відповідає код намірам розробника, що підвищує ризик інтеграції помилкового або неоптимального коду. Без ретельного контролю розробникам може бути складно спрямовувати згенерований ШІ код на створення якісних рішень, які можна в подальшому підтримувати.

Залишається незрозумілим, чи роблять ВММ ті ж типи помилок у використанні прикладних програмних інтерфейсів (application programming interface, API), що й звичайні розробники. Більше того, дослідження шаблонів неправильного використання API у моделях з відкритим та закритим кодом є дуже важливим, оскільки моделі з відкритим кодом часто відстають від своїх закритих аналогів за продуктивністю. Сучасні підходи до запобігання зловживанню API в основному зосереджені на автоматичному виявленні за допомогою попередньо визначених типів зловживань та документації до API. Однак ці методи стикаються зі значними обмеженнями при застосуванні до генерації коду на основі ВММ, що включає мільйони API та різноманітні контексти.

Аналіз останніх досліджень і публікацій

У широкому контексті ефективність GitHub Copilot була оцінена у вирішенні численних тестів з комп'ютерних наук, включаючи питання з множинним вибором та вправи з програмування на Python [1]. Їхні висновки показують, що хоча модель не змогла пройти курс, вона досягла високих результатів у більшості вправ і змогла покращити відповіді на основі зворотного зв'язку від автооцінювача. В іншому підході ефективність Github Copilot була оцінена при розв'язуванні задач Парсонса, а саме програм, де програмний код подано в неправильному порядку [2]. Показано, що Copilot може вирішити 80 % проблем на Python, якщо ігнорувати помилки відступів.

Автори [3] досліджували здатність GitHub Copilot вирішувати завдання на великому наборі простих вправ з програмування на Python. Коли ВММ не могла згенерувати правильний розв'язок, автори намагалися надати більш чіткі інструкції природною мовою (*prompt engineering*). Загалом Copilot не міг надати правильне рішення для 20 % завдань. Автори [4] оцінили здатність GitHub Copilot вирішувати вступні завдання з програмування. Їх робота включає оцінку декількох згенерованих програм для набору тестів з програмування у відповідному курсі. Їхні висновки показують, що GitHub Copilot отримав високі бали на тестах з програмування, перевершивши більшість здобувачів вищої освіти.

Існуючі дослідження зловживань використання API у коді, згенерованому ВММ, зосереджувалися переважно на контрольованих, синтетичних сценаріях. Наприклад, у роботі [5] автори досліджували якість коду, згенерованого кількома моделями на основі питань у стилі StackOverflow, і виявили, що GPT-4 генерував код із зловживанням API у 62 % випадків при тестуванні на 18 широко використовуваних Java API. Аналогічно, автори [6]

досліджували зловживання API шляхом вивчення питань програмування на Java, пов'язаних з безпекою, і виявили, що ВММ зловживали API приблизно в половині завдань. Нещодавній аналіз помилок 1000 зразків з існуючих бенчмарків виявив три основні типи помилок: AttributeError, TypeError і ValueError, які зазвичай асоціюються з неправомірним використанням API у коді, що генерується ВММ. Хоча ці дослідження продемонстрували, що певні моделі можуть генерувати код із неправильним використанням API на обмеженій кількості бібліотек, API та мов програмування, їхні висновки не можуть бути узагальнені для реальних контекстів програмування через їхню залежність від питань та сценаріїв, створених людиною.

Формулювання мети дослідження

Метою дослідження є спостереження, як інструменти ВММ справляються із генерацією коду на основі запиту з певними обмеженнями. Зокрема, ми зосередилися на завданнях з програмування для здобувачів третього курсу університету, щоб дослідити, як інструменти ВММ справляються з ними. Питання полягає в тому, чи можуть інструменти ВММ розшифрувати описане природною мовою завдання на програмування, розпізнати задачу, яку потрібно розв'язати, та встановлені умови її щодо розв'язання, і згенерувати функціональний код, який вирішує цю задачу.

Викладення основного матеріалу дослідження

Традиційні інтегровані середовища розробки, такі як Eclipse та Visual Studio Code, докорінно змінили підхід розробників до кодування. Вони надають розробникам потужні інструменти, які полегшують практику модульного кодування, від написання коду на різних рівнях до рефакторингу, збірки та тестування.

Розробники використовують IDE для роботи зі складними проєктами завдяки таким функціям, як редагування коду з підсвічуванням синтаксису та автозавершенням. Модульний підхід до кодування ще більше посилюється дизайном інтерфейсу IDE, який організовує код у логічні секції, полегшуючи розробникам керування та маніпулювання різними частинами їхніх проєктів. Така структура не лише сприяє ефективному кодуванню, але й допомагає у налагодженні та доопрацюванні кодової бази за потреби.

Інтеграція в традиційні IDE асистентів на основі ШІ призвела до значних змін в практиці розробки програмного забезпечення. Однією з найпомітніших функцій є вбудовані підказки, які надають допомогу розробникам у режимі реального часу під час написання коду або його ітерацій. Ця функція не лише завершує рядки коду, але й пропонує контекстно-релевантні пропозиції, засновані на вимогах проєкту та найкращих практиках. Інтерфейс чату дозволяє розробникам вести діалог зі штучним інтелектом для створення або рефакторингу вихідного коду. Цей підхід особливо корисний для створення коментарів до документації або юніт-тестів, створюючи контекстно-релевантні результати, які покращують зрозумілість проєкту.

Як і інші методи ШІ, інструменти генерації коду страждають від невизначеності та недосконалості, притаманних їм за своєю природою. Вони можуть генерувати код з помилками або навіть код, який кардинально відрізняється від очікувань користувачів. Однак, на відміну від інших областей, генерація коду вимагає набагато вищого рівня коректності: код або компілюється, або ні, і він або правильний, або містить помилки, такі як логічні помилки та вразливості безпеки [7].

ВММ зазвичай роблять типові помилки, коли пропонують завершити або вставити код [8]:

– модель генерує неповні виклики методів та/або з помилковими параметрами;
– модель генерує виклики методів до схожих, але не пов'язаних між собою API;
– модель генерує сторонні виклики методів, які не потрібні в контексті цільового використання;
– модель не дотримується необхідної послідовності викликів методів для правильного налаштування та використання цільової бібліотеки через складність її API;
– модель має викликати виклики API до декількох бібліотек, але не може правильно використовувати ці різні бібліотечні API разом.

Всі ці помилки може бути дуже складно виявити і виправити, як статично, так і динамічно.

Поширеною схемою взаємодії є використання ШІ як замінника пошуку в Інтернеті. Однак, на відміну від прикладів коду з вебресурсу Stack Overflow, які перевіряються програмістами-людьми, код, згенерований ШІ, може містити помилки. Оскільки ШІ генерує лише одне рішення за раз, програмісти не можуть порівняти кілька альтернативних рішень і оцінити їхню якість, як це часто роблять під час пошуку в Інтернеті.

Хоча генерування коду за допомогою ШІ відбувається швидше, ніж копіювання коду з Інтернету, згенерований код може містити помилки, що призводить до збільшення часу, витраченого на налагодження. Тоді як код з Інтернету, як правило, не містить помилок, супроводжується поясненнями та обговореннями і може бути змінений відповідно до поточного завдання шляхом внесення незначних правок. ШІ також надає користувачам корисну відправну точку для початку роботи, навіть якщо згенерований код був неправильним. Це особливо корисно для користувачів, які застрягли в проблемі або не знають, як підійти до виконання завдання.

Зловживання сторонніми API у коді, згенерованому ВММ, є серйозною проблемою у розробці програмного забезпечення. Хоча ВММ продемонстрували вражаючі можливості генерації коду, їхня взаємодія зі складними бібліотечними API залишається дуже схильною до появи помилок, що потенційно може призвести до збоїв у роботі програмного забезпечення та вразливостей у безпеці.

Неправильна інтерпретація намірів виникає, коли модель обирає API або метод, який є синтаксично правильним, але не відповідає запланованій функціональності або завданню. Це призводить до неправильної поведінки через нерозуміння моделлю призначення методу.

Галюцинація моделі виникає, коли модель вводить абсолютно неправильний метод API або параметр, який не існує або не потрібен для виконання завдання. Модель може створити або «галюцинувати» виклик API, який здається правдоподібним, але є недійсним у контексті, що призводить до помилкової поведінки або помилки компіляції/виконання.

Неправильне використання, пов'язане з відсутнім елементом відбувається, коли модель пропускає необхідний метод або параметр API, що призводить до неповної або некоректної поведінки. Цей тип зловживань зазвичай виникає, коли модель не повністю розуміє вимоги API або ігнорує важливі деталі, які необхідні для правильного використання.

Використання надлишкової інформації виникає, коли модель включає непотрібні ланцюжки методів або повторювані виклики API, які не сприяють виконанню завдання і призводять до неефективності або потенційних помилок.

В якості прикладу згенерованого коду розглянемо новий інструмент прототипування на базі Firebase Studio. Firebase Studio – новий продукт, представлений компанією Google 9 квітня 2025 року для використання Google Platform в якості середовища розробки різних видів застосунків, таких як веб-застосунки (front-end та fullstack) і мобільні застосунки (кросплатформні та нативні).

Для тесту введемо наступний запит: «Згенеруй Android-застосунок, який буде мати інтерфейс з використанням AntD та працювати з запитами згідно документації до API: People Data Generator API». Також в запит були додані деталі API які мають працювати локально на іншому проєкті. Відповідно, очікуваний результат – прототип Android-застосунку, який буде містити в собі поля згідно опису параметрів API та матиме стилізацію згідно AntDesign. AntDesign – це бібліотека для React застосунків, тому стиль має бути запозичений з цієї бібліотеки. Результат запиту наведений нижче.

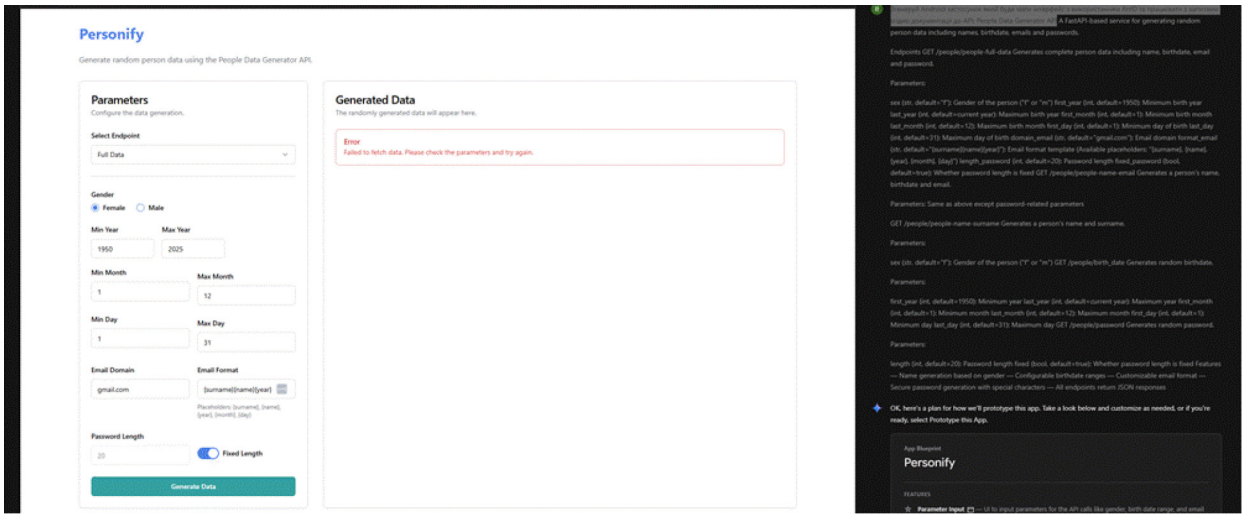

Рис. 1. Результат першого запиту

На рис. 1 наведений приклад галюцинації моделі. В коді можна побачити, що створений вебзастосунок на основі бібліотеки React, але при цьому AntDesign компоненти не застосовуються. В запиті використання інтерфейсу AntD було пріоритетним, але оскільки модель володіє інформацією лише про React, тому був створений саме вебзастосунок. Проблема, яка виникла, була вирішена лише за допомогою ручного вибору для створення проєкту Android та використання автозавершення коду.

На рис. 2 показаний результат генерації коду за допомогою автозавершення, який дозволяє автоматично заповнювати необхідні частини коду всередині файлу, та, за необхідністю, за допомогою запиту для інших файлів можна зробити робочий проєкт. Недоліком такого підходу є відсутність одночасної генерації коду в різних файлах.

На практиці розробники часто не приймають початкові пропозиції асистентів ШІ-програмування. Розробникам важко зрозуміти, які вхідні дані впливають на генерацію коду, вони відмовляються від використання отриманого коду і не можуть контролювати модель, щоб вона генерувала корисні пропозиції щодо коду. Найчастішими причинами, чому користувачі відмовляються від використання отриманого коду, є те, що код не виконує правильної дії або не відповідає функціональним чи нефункціональним вимогам.

```
class MainActivity : ComponentActivity() {
    }
}

@Composable
fun PersonDataScreen(modifier: Modifier = Modifier) {
    var personData by remember { mutableStateOf<String?>(null) }
    var isLoading by remember { mutableStateOf(true) }
    var hasError by remember { mutableStateOf(false) }

    LaunchedEffect(Unit) {
        try {
            val client = OkHttpClient()
            val request = Request.Builder()
                .url("https://api.example.com/people/people-full-data")
                .get()
                .build()

            val response: Response = client.newCall(request).execute()
            if (response.isSuccessful) {
                val jsonData = response.body?.string()
                val jsonObject = JSONObject( json: jsonData ?: "")
                personData = jsonObject.toString( indentSpaces: 4) // Format JSON
                isLoading = false
            } else {
                hasError = true
                isLoading = false
            }
        } catch (e: Exception) {
            hasError = true
            isLoading = false
        }
    }

    if (isLoading) {
        Text(text = "Loading...", modifier = modifier)
    } else if (hasError) {
        Text(text = "Error fetching data", modifier = modifier)
    } else {
        personData?.let { Text(text = it, modifier = modifier) }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
```

Рис. 2. Результат другого запиту

Висновки

У цій статті наведено результати досліджень можливостей ВММ для розв'язання задач програмування. Наші результати показують, що такі інструменти мають труднощі з генерацією правильного вихідного коду. ВММ часто здатні розпізнати алгоритм, необхідний для розв'язання завдання, однак моделі мають труднощі з адаптацією коду до конкретних обмежень і часто схильні застосовувати популярні рішення замість того, щоб слідувати заданим інструкціям. Хоча сучасні помічники для програмування зі штучним інтелектом є високопродуктивними, існує розрив між потребами розробників і можливостями інструментів, наприклад, щодо врахування нефункціональних вимог при створенні коду.

Список використаної літератури

1. Šavelka, J., Agarwal, A., Bogart, C., Song, Y., & Sakr, M. F. (2023). Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses? *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. DOI: https://doi.org/10.48550/arXiv.2303.09325

2. Reeves, B., Sarsa, S., Prather, J., Denny, P., Becker, B.A., Hellas, A., Kimmel, B., Powell, G., & Leinonen, J. (2023). Evaluating the performance of code generation models for solving parsons problems with small prompt variations.

Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. DOI: https://doi.org/10.1145/3587102.3588805

3. Denny, P., Kumar, V., & Giacaman, N. (2023). Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. DOI: https://doi.org/10.1145/3545945.3569823

4. Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., & Prather, J. (2022). The robots are coming: Exploring the implications of openai codex on introductory programming. *Proceedings of the 24th Australasian Computing Education Conference*. DOI: https://doi.org/10.1145/3511861.3511863

5. Zhong, L., Wang, Z. (2024). Can LLM replace stack overflow? A study on robustness and reliability of large language model code generation. *Proceedings of the AAAI Conference on Artificial Intelligence, vol. 38, no. 19*. DOI: https://doi.org/10.48550/arXiv.2308.10335

6. Mousavi, Z., Islam, C., Moore, K., Abuadbba, A. & Babar, M. A. (2024). An investigation into misuse of java security apis by large language models. *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. DOI: https://doi.org/10.48550/arXiv.2404.03823

7. Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *Chi conference on human factors in computing systems extended abstracts*. https://doi.org/10.1145/3491101.3519665

8. Zhuo, T.Y., He, J., Sun, J., Xing, Z., Lo, D., Grundy, J., & Du, X. (2025). Identifying and Mitigating API Misuse in Large Language Models. DOI: https://doi.org/10.48550/arXiv.2503.22821