УДК 004.4:004.8 DOI: 10.31673/2412-9070.2026.318119

О. Б. ПРИДИБАЙЛО[1], ст. викл.;
ORCID: 0009-0003-7967-5827
Р. В. ПРИДИБАЙЛО[1], аспірант;
ORCID: 0009-0003-1747-7518
В. О. ЯСКЕВИЧ[2], канд. техн. наук, доцент;
ORCID: 0000-0002-5796-2521
Ю. В. ЯСКЕВИЧ[2], ст. викл.,
ORCID: 0009-0005-6084-5229
[1]Державний університет інформаційно–комунікаційних технологій, Київ
[2]Київський університет імені Бориса Грінченка

ЕВОЛЮЦІЯ ПАРАДИГМ ПРОГРАМУВАННЯ ТА ФЕНОМЕН VIBE CODING: ОЦІНКА МОЖЛИВОСТЕЙ, РИЗИКІВ ТА ПЕРСПЕКТИВ ДОСЛІДЖЕНЬ

Стрімкий розвиток систем штучного інтелекту (ШІ) та великих мовних моделей (LLM) зумовив фундаментальний зсув у підходах до створення програмного забезпечення, породивши нові парадигми взаємодії людини та обчислювальних машин. Дана стаття присвячена глибокому аналізу історичної еволюції програмування — від безпосереднього маніпулювання апаратним забезпеченням через машинний код та перші компілятори до високорівневих мов, об'єктно-орієнтованого програмування та сучасних підходів, керованих штучним інтелектом. Особлива увага у дослідженні приділяється концепції «vibe coding» (вайб-кодинг) — новітньому підходу до розробки, який передбачає створення програмних продуктів переважно шляхом ітеративної взаємодії з ШІ-асистентами за допомогою природної мови, з мінімальним ручним написанням коду або навіть без повного розуміння розробником згенерованих алгоритмів та внутрішньої структури системи.

У процесі дослідження детально та всебічно оцінюються можливості, переваги та недоліки цього підходу. Встановлено, що використання генеративного ШІ у форматі вайб-кодингу суттєво знижує поріг входження в IT-індустрію для осіб без профільної технічної освіти, дозволяє досягти безпрецедентного прискорення процесів прототипування та розробки мінімально життєздатних продуктів (MVP). Зокрема, результати систематичних оглядів демонструють підвищення загальної продуктивності виконання завдань на 26–30% та зростання швидкості ітерацій для розробників початкового рівня. Разом з тим, виявлено та систематизовано критичні недоліки і приховані ризики вайб-кодингу, які ставлять під загрозу довгострокову життєздатність проєктів. Доведено існування так званого «парадоксу продуктивності», згідно з яким досвідчені інженери (Senior developers) можуть витрачати на 19% більше часу на виконання складних завдань через необхідність аудиту та налагодження прихованих логічних помилок і «галюцинацій» ШІ-моделей.

Окремо розглядаються надзвичайно гострі проблеми кібербезпеки: згідно зі звітом лабораторії Veracode (2025), під час тестування понад 100 моделей було виявлено, що 45% згенерованого ШІ коду провалили базові тести, містячи критичні вразливості, що підпадають під класифікацію OWASP Top 10. Крім того, доведено, що надмірна та некритична довіра до ШІ без належного суворого архітектурного контролю призводить до накопичення специфічного «темного» технічного боргу (Dark Debt), прогресуючої архітектурної фрагментації (відомої як Context Rot) та поступової деградації фундаментальних інженерних навичок у розробників (Skill Atrophy).

На основі проведеного синтезу та аналізу обґрунтовано висновок, що вайб-кодинг має абсолютний сенс як потужний інструмент для надшвидкого створення прототипів, автоматизації рутинних завдань та генерації шаблонного коду. Проте цей підхід не здатний повністю замінити класичну дисципліну інженерії програмного забезпечення у контексті створення критичних, масштабованих, високонавантажених та безпеково-орієнтованих корпоративних систем. Визначено ключові напрямки для подальших наукових та прикладних досліджень, серед яких: розробка і впровадження методів «інженерії контексту» (Context Engineering), створення інтегрованих систем агентної кібербезпеки (Agentic AppSec) для семантичної перевірки коду в режимі реального часу, а також глибоке вивчення соціально-технічного впливу штучного інтелекту на процеси навчання та професійного становлення майбутніх інженерів.

Ключові слова: парадигми програмування, штучний інтелект, вайб-кодинг, технічний борг, кібербезпека, генерація коду, великі мовні моделі, інженерія програмного забезпечення.

Вступ

Постановка проблеми. Розробка програмного забезпечення історично сформувалася як складна інженерна дисципліна, що вимагає від спеціалістів глибоких когнітивних зусиль, суворого дотримання синтаксичних і семантичних правил специфічних мов програмування, розуміння патернів проєктування, алгоритмізації та принципів раціонального управління пам'яттю [1]. Протягом десятиліть індустрія рухалася шляхом поступового підвищення рівня абстракції, прагнучи наблизити машинну логіку до людського мислення. Однак з появою та стрімким вдосконаленням великих мовних моделей (LLM), здатних не лише аналізувати текст, але й генерувати синтаксично правильний та семантично функціональний програмний код на основі запитів природною мовою, парадигма створення програмного забезпечення зазнала найрадикальнішого зсуву з моменту винайдення компіляторів [2]. Цей зсув породив абсолютно нову практику розробки, яка отримала неформальну, але вже загальновизнану назву «vibe coding» (вайб-кодинг) [4].

Вайб-кодинг передбачає створення ПЗ шляхом ітеративного, високоагентного діалогу зі штучним інтелектом, де розробник або користувач формулює свої наміри природною мовою, а ШІ самостійно генерує, рефакторить та імплементує код, часто без глибокого аудиту чи розуміння його внутрішньої механіки з боку людини [5]. Хоча цей підхід забезпечує безпрецедентну швидкість перетворення ідей на робочі продукти та максимально демократизує сферу технологій, він одночасно кидає виклик фундаментальним принципам програмної інженерії: надійності, кібербезпеці, системній масштабованості та, що найважливіше, здатності людини повністю розуміти та контролювати створені цифрові системи [6]. Відповідно, виникає гостра наукова та практична необхідність об'єктивно оцінити, чи є раціональний сенс у впровадженні вайб-кодингу, де пролягають межі його безпечного застосування, та які невідворотні наслідки він несе для архітектури ПЗ і професії розробника в цілому.

Аналіз останніх досліджень і публікацій. Питання впливу систем генеративного штучного інтелекту на продуктивність розробників та якість створюваного коду останнім часом перебуває в епіцентрі уваги провідних наукових установ, університетів та аналітичних підрозділів технологічних корпорацій. Аналіз масиву досліджень, опублікованих у 2024–2026 роках, демонструє вкрай складну та багатовимірну картину.

З одного боку, низка систематичних оглядів літератури (SLR) та емпіричних оцінок вказують на значне зростання продуктивності. Наприклад, дослідження фіксують прискорення розробки на 26–30%, значне скорочення часу на пошук інформації та підвищення частоти компіляції коду на 38,4%, що особливо яскраво проявляється серед розробників молодшого рівня (Junior developers) [8].

З іншого боку, з'являються переконливі емпіричні дані високого рівня доказовості (рандомізовані контрольовані дослідження — RCT), що вказують на критичні ризики підходу

«інтуїтивного кодування». Найбільший резонанс викликало дослідження організації METR (Model Evaluation & Threat Research), проведене у 2025 році. Воно виявило так званий «парадокс продуктивності»: досвідчені інженери, які використовували ШІ-інструменти для роботи над складними реальними завданнями у зрілих кодових базах, витрачали на 19% більше часу на їх вирішення порівняно з роботою без ШІ, хоча суб'єктивно вірили, що працюють швидше [9].

Тривожні результати демонструють звіти з кібербезпеки. Згідно з масштабним звітом «2025 GenAI Code Security Report» від Veracode, під час тестування понад 100 великих мовних моделей було виявлено, що 45% згенерованих зразків коду містять критичні вразливості, що відповідають списку найнебезпечніших загроз OWASP Top 10 [10]. Дослідники з Anthropic у своєму RCT (2025) виявили деградацію інженерних навичок: розробники, які покладалися на ШІ-асистентів, демонстрували зниження рівня розуміння власного коду та засвоєння нових концепцій на 17% порівняно з тими, хто писав код власноруч [11]. Додатково, у фаховому середовищі активно вивчаються феномени «темного боргу» (Dark Debt) та фрагментації контексту систем, повністю згенерованих ШІ без належного людського нагляду [5].

Як результат аналізу джерел обов'язково виокремлюються раніше невирішені частини загальної проблеми, яким присвячена стаття: незважаючи на величезний обсяг даних щодо продуктивності, існує гострий брак комплексного, збалансованого розуміння довгострокових наслідків застосування парадигми вайб-кодингу на всьому життєвому циклі розробки ПЗ. Відсутня чітка артикуляція меж, де генерація коду стає небезпечною, а також бракує методологічних рекомендацій щодо безпечної інтеграції високорівневих ШІ-систем в архітектурно складні проєкти.

Мета статті. Метою статті є комплексне дослідження еволюції парадигм програмування, що історично призвела до появи концепції vibe coding, а також об'єктивна оцінка переваг та прихованих ризиків цього підходу у розробці ПЗ. На основі цього аналізу стаття має визначити, чи існує раціональний сенс у застосуванні вайб-кодингу в сучасній інженерії, окреслити безпечні межі його використання та сформулювати ключові напрямки, які потребують першочергового наукового дослідження.

Основна частина

Еволюція парадигм програмування: від машинного коду до ШІ-керованих систем

Повноцінне розуміння феномену вайб-кодингу як нової епохи у розробці неможливе без ретроспективного аналізу того, як людство взаємодіяло з обчислювальними машинами протягом останніх десятиліть. Історія програмування — це фундаментальний процес безперервного підвищення рівня абстракції. Кожна нова парадигма створювалася як відповідь на експоненційне зростання складності програмних систем, прагнучи делегувати низькорівневі завдання інструментам-посередникам (компіляторам, інтерпретаторам), щоб звільнити когнітивний ресурс людини для вирішення вищих архітектурних завдань [1].

1. Епоха апаратного маніпулювання та машинного коду (1940-ві – 1950-ті роки). На зорі обчислювальної техніки програмісти перших комп'ютерів (наприклад, ENIAC) фізично перемикали кабелі та тумблери [1]. Згодом перейшли до використання перфокарт, де інформація кодувалася нулями та одиницями. Машинний код був єдиною мовою, яку безпосередньо розумів процесор. Програмування було надзвичайно повільним, схильним до фатальних людських помилок і повністю позбавленим портативності. Першим кроком до абстракції стала розробка мови Assembly (Асемблер) у 1949 році, яка замінила бінарні послідовності на мнемонічні скорочення (наприклад, ADD для додавання). Проте Асемблер залишався мовою дуже низького рівня, жорстко прив'язаною до апаратної архітектури [1].

2. Виникнення високорівневих мов та процедурна парадигма (кінець 1950-х – 1970-ті роки). Першими спробами створення справжніх високорівневих мов, що абстрагували програміста від заліза, стали мова Short Code (Джон Моклі, 1949) та перший компілятор для системи A-0, створений Грейс Гоппер у 1951–1952 роках [1]. Однак справжній комерційний прорив стався у 1957 році, коли команда Джона Бекуса в корпорації IBM випустила FORTRAN, що дозволив записувати математичні формули у звичному вигляді.[1] Наступним викликом

стала проблема контролю потоку виконання. Відповіддю стала парадигма структурованого (процедурного) програмування, втілена в мовах ALGOL, Pascal та C (Денніс Рітчі, 1972).

3. Об'єктно-орієнтоване програмування (1980-ті – 1990-ті роки). Процедурне програмування виявилося вразливим при управлінні глобальними станами даних у великих інформаційних системах. Це зумовило перехід до об'єктно-орієнтованого програмування (ООП), піонером якого стала мова Smalltalk (1980). ООП запропонувало моделювати програму як сукупність «об'єктів», які інкапсулюють у собі стан та поведінку. Це багаторазово підвищило можливості повторного використання коду (C++, Java) [1].

4. Мультипарадигмальність, веб-епоха та управління пам'яттю (2000-ні – 2010-ті роки). З розвитком Інтернету та хмарних обчислень виникла потреба у мовах для швидкої розробки (JavaScript, Python). Одночасно, виклики багатоядерних процесорів повернули інтерес до функціонального програмування та породили мови, орієнтовані на безпеку пам'яті без використання «збирачів сміття» (Rust) [1].

5. Software 2.0 та Software 3.0 (Сучасний етап та ШІ). Історично всі попередні парадигми належали до категорії «Software 1.0», де програміст безпосередньо прописує детерміновану логіку поведінки системи. У концепції «Software 2.0» код замінюється ваговими коефіцієнтами нейронної мережі: людина проектує архітектуру нейромережі, а алгоритм «пише себе сам» шляхом оптимізації на основі даних. Сьогодні, з появою LLM, ми спостерігаємо перехід до «Software 3.0» або «Chat-Oriented Programming». У цій парадигмі інтерфейсом взаємодії з комп'ютером стає природна мова. Саме на вістрі цієї еволюційної хвилі сформувалося явище вайб-кодингу [2].

Феномен Vibe Coding: концептуалізація та відмінність від класичної інженерії

Сам термін «vibe coding» увійшов у широкий вжиток після допису Андрея Карпаті (Andrej Karpathy) у лютому 2025 року. Він описав цей процес як програмування, де розробник *«повністю віддається вайбу і забуває, що код взагалі існує»*, використовуючи функцію тотального прийняття згенерованого коду («Accept All») та уникаючи ручного налагодження [4].

З наукової точки зору, вайб-кодинг — це процес *радикальної реконфігурації медіації намірів (intent mediation)* [2]. У традиційній розробці людина має детально декомпозувати свій намір у формальні структури та синтаксис. У вайб-кодингу людина передає більшу частину епістемічної (пізнавальної) та імплементаційної праці машині [2]. Втім, ця передача не є абсолютною: людина бере на себе важливу роботу з валідації результату та коригування наступних запитів, що вимагає когнітивного навантаження іншого типу (оркестрація замість написання) [3].

Слід чітко розмежовувати «AI-Assisted Engineering» (ШІ-асистовану інженерію) та «Vibe Coding». Як зазначає експерт Саймон Віллісон (Simon Willison): *«Якщо LLM написала кожен рядок вашого коду, але ви перевірили, протестували і повністю зрозуміли його весь, це не вайб-кодинг — це використання LLM як асистента для набору тексту»* [12]. Вайб-кодинг характеризується відмовою від мікроменеджменту логіки. Його визначальними рисами є зосередженість на результаті («Що», а не «Як»), ітеративність замість детермінізму та ігнорування структурного аудиту з боку людини [6].

Методика дослідження

Дослідження базується на методі систематичного огляду літератури (Systematic Literature Review) та якісного синтезу даних. Для досягнення мети було проведено пошук у провідних наукометричних базах даних та архівах препринтів (зокрема IEEE Xplore, ACM Digital Library, SSRN, arXiv) за ключовими пошуковими запитами: «vibe coding», «AI-assisted programming», «LLM code generation», «technical debt in AI», «AI code security» за період з початку 2024 до березня 2026 року.

Критеріями включення джерел до фінального аналізу були: наявність кількісних або якісних емпіричних даних (зокрема результатів рандомізованих контрольованих досліджень або опитувань розробників); розгляд архітектурних, когнітивних або безпекових аспектів застосування генеративного ШІ в інженерії ПЗ. Додатково, для забезпечення повноти картини у швидкозмінній галузі, до аналізу було залучено авторитетні галузеві звіти профільних лабораторій

(Veracode, Black Duck, Faros AI, CSET) та аналітичних центрів (METR, Anthropic). Усього до фінального синтезу було відібрано ключові наукові статті та масиви даних, результати яких піддавалися порівняльному аналізу для виявлення як переваг продуктивності, так і інфраструктурних ризиків застосування підходу вайб-кодингу.

Результати дослідження

Впровадження вайб-кодингу та автономних агентних систем ШІ в життєвий цикл розробки програмного забезпечення формує глибоко поляризований ландшафт. Результати досліджень чітко демонструють дихотомію: феноменальні показники швидкості на початкових етапах життєвого циклу супроводжуються прихованими, але критичними руйнівними процесами на стадіях масштабування, підтримки та забезпечення безпеки.

1. Оцінка можливостей та переваг вайб-кодингу

Безпрецедентне прискорення прототипування (Time-to-Market). Головною рушійною силою популярності вайб-кодингу є його здатність драматично стискати час розробки. Систематичні огляди «сірої літератури» та поведінкових звітів свідчать, що 62% розробників називають «швидкість та ефективність» основним мотивом використання цього підходу [6]. У 64% випадків користувачі повідомляють про досвід «миттєвого успіху та потоку» (Instant Success & Flow) — стан, коли природно-мовний опис проблеми миттєво конвертується у функціональний інтерфейс або скрипт [6]. Для розробників-початківців дослідження фіксують стабільне зростання продуктивності: обсяг виконаних завдань зростає на 26–40%, а частота ітерацій компіляції — на 38,4%.[8] ШІ ефективно бере на себе написання шаблонного (boilerplate) коду. Як узагальнено у Таблиці 1, цей підхід докорінно змінює інтерфейс взаємодії з машиною.

Демократизація створення ПЗ та зміна ролей. Вайб-кодинг радикально знижує бар'єр входження в розробку програмного забезпечення. Особи без профільної інженерної освіти — підприємці, аналітики даних, дизайнери — отримують інструментарій для створення робочих програмних інструментів, використовуючи лише свої знання предметної області [6]. У цьому новому середовищі цінність фахівця зміщується від знання специфічного синтаксису до системного мислення, концептуального бачення продукту (product vision) та здатності оркерструвати ШІ-агентами [3].

Таблиця 1

Порівняльна матриця традиційної інженерії ПЗ та вайб-кодингу

Характеристика	Традиційна інженерія ПЗ	Vibe Coding (ШІ-керована розробка)
Основний інтерфейс взаємодії	Строгий синтаксис та семантика мов програмування	Природна мова (промптинг), ітеративний діалог
Фокус когнітивних зусиль	Логіка імплементації, управління пам'яттю, архітектура	Формулювання результату (intent), валідація, UI/UX
Крива навчання	Крута; вимагає тривалого академічного або практичного вивчення алгоритмів	Полога; вимагає базових навичок комунікації та доменної експертизи

Швидкість на етапі ідеації	Помірна; потребує початкового архітектурного планування	Надвисока; створення робочих прототипів за хвилини
Процес налагодження (Debugging)	Аналітичний: читання стеку викликів, аудит змінних, розуміння коду	Реактивний: копіювання помилки назад у ШІ-асистента для виправлення

2. Критичні недоліки, ризики та довгострокові загрози

Незважаючи на потужний потенціал для прототипування, використання вайб-кодингу у розробці реальних, продуктових систем (production-grade) без належного нагляду викриває серйозні вади (див. Таблицю 2).

Кібербезпека: масова ін'єкція вразливостей. Фундаментальною проблемою генеративних ШІ-моделей є те, що вони навчаються на гігантських масивах відкритого коду (Open Source), які історично містять застарілі патерни, помилки та небезпечні архітектурні рішення. Як наслідок, моделі часто реплікують ці недоліки [14]. Звіт лабораторії Veracode «2025 GenAI Code Security Report», що базується на масштабному тестуванні понад 100 великих мовних моделей на специфічних наборах завдань, містить тривожну статистику: згідно з їхнім дослідженням, 45% зразків згенерованого ШІ коду провалили базові тести на безпеку, безпосередньо впроваджуючи вразливості зі списку найнебезпечніших загроз OWASP Top 10 [10]. Найбільш поширеними загрозами є ігнорування санітизації даних (призводить до XSS та SQL-ін'єкцій), помилки переповнення буфера (C/C++) та жорстке кодування секретних ключів [10]. Ситуація критично погіршується через явище «упередженості до автоматизації» (Automation Bias): розробники схильні сліпо довіряти авторитету ШІ, навіть коли його код об'єктивно містить вразливості [16].

Парадокс продуктивності для досвідчених інженерів. Хоча вайб-кодинг дає потужний поштовх початківцям, для Senior-інженерів він створює перешкоди. Строге рандомізоване дослідження (RCT), проведене групою METR (2025 р.), виявило, що досвідчені інженери, використовуючи ШІ для вирішення складних завдань у своїх зрілих репозиторіях, витрачали на 19% більше часу, ніж контрольна група без ШІ [9]. Дослідження галузевої аналітичної платформи Faros AI підтверджує цю тенденцію в промислових умовах: впровадження ШІ-асистентів призвело до збільшення розміру Pull Request'ів (PR) на 154% (оскільки ШІ генерує багатослівний код), що спричинило зростання часу на код-рев'ю на 91% і збільшення кількості багів на 9% на розробника [17].

Математично цей парадокс можна концептуалізувати через формулу ефективної продуктивності (P_{eff}) розробника:

$$P_{eff} = \frac{V_{code}}{T_g + T_r + (T_d \cdot P(E))}, \quad (1)$$

де V_{code} — корисний обсяг функціонального коду (який вимірюється кількістю завершених, логічно валідованих бізнес-функцій, а не просто рядками згенерованого тексту); T_g — час на формулювання запиту та генерацію; T_r — час на первинний аудит коду людиною; T_d — час на глибокий дебагінг прихованої архітектурної помилки; $P(E)$ — ймовірність того, що ШІ згенерує критичну логічну галюцинацію. Для простих MVP знаменник малий, тому продуктивність висока. Однак у складних системах час на розуміння чужого (машинного) коду (T_d) зростає експоненційно [9].

Архітектурна деградація та «Темний борг» (Dark Debt). Масштабування вайб-кодингу за межі простих проєктів часто призводить до явища "Context Rot" (гноїння контексту) [5]. На 3-й чи 4-й день ітерацій модель починає плутатися у файловій структурі, руйнуючи існуючі

зв'язки. Близько 11% спроб вайб-кодингу закінчуються повним закиданням проєкту (Code Abandonment) [6]. Це породжує «Темний борг» (Dark Debt) — масиви коду, логіку яких ніхто у команді не може пояснити, що робить систему непідтримуваною [5].

Втрата навичок (Skill Atrophy). Постійне когнітивне розвантаження несе загрозу професійному розвитку. Дослідження Anthropic (2025) показало статистично значуще падіння майстерності: на тестуванні з нових концепцій група, що використовувала ШІ, набрала на 17% менше балів порівняно з тими, хто кодував власноруч [11]. Найбільший провал фіксувався у навичках налагодження.

Таблиця 2

Спектр ризиків вайб-кодингу та рекомендації з їх пом'якшення

Категорія ризику	Механізм виникнення у Vibe Coding	Стратегії мінімізації та протидії (Guardrails)
Кібербезпека (AppSec)	Реплікація застарілих патернів; відсутність санітизації (CWE-89)	Інтеграція Agentic AppSec; обов'язковий SAST/DAST контроль перед коммітом
Технічний борг (Dark Debt)	Накопичення масивів коду без усвідомленого архітектурного задуму	Застосування правила "Skeleton First" (людина задає структуру); дрібноблочні PR
Деградація навичок (Skill Atrophy)	Менталітет "Accept All"; когнітивне делегування розуміння логіки машині	Трансформація культури: ШІ як "молодший асистент"; фокус на System Design

Висновки

Еволюція програмування завжди була спрямована на створення вищих рівнів абстракції. Поява великих мовних моделей, здатних синтезувати код на основі запитів природною мовою, стала найрадикальнішим кроком у цій історії. Концепція «vibe coding» маніфестувала цей перехід.

Проведене дослідження дозволяє дійти висновку, що вайб-кодинг має абсолютний, раціональний сенс, але виключно у суворо обмеженому спектрі застосувань: для надшвидкого створення MVP, прототипування, автоматизації рутини та генерації шаблонного коду. Проте, як виробнича парадигма для розробки масштабних систем, чистий вайб-кодинг (із його менталітетом сліпого прийняття коду) є деструктивним. Емпіричні дані доводять, що він призводить до суттєвих ризиків безпеки, архітектурної деградації та парадоксу продуктивності для досвідчених інженерів. Майбутнє інженерії полягає у ШІ-асистованій розробці, де людина еволюціонує від простого «кодера» до системного архітектора.

Перспективою подальших досліджень є: дослідження методів семантичного структурування проєктів, управління RAG-системами та динамічної оркестрації промптів; розробка мультимодельних ШІ-агентів, здатних в режимі реального часу виявляти вразливості (SAST) та блокувати небезпечні коммітів; розробка нових фреймворків, які вимірюватимуть якість, дов-

гострокову ремонтопридатність та рівень «темного боргу», а також академічне вивчення впливу ШІ на формування навичок у Junior-фахівців для уникнення проблеми «загубленого покоління» розробників.

Внесок авторів

Оксана ПРИДИБАЙЛО – концептуалізація; Роман ПРИДИБАЙЛО – аналіз джерел, підготовка огляду літератури; Владислав ЯСКЕВИЧ – збір і перевірка емпіричних даних; Юрій ЯСКЕВИЧ – емпіричне дослідження.

Декларація про штучний інтелект

Автори використовували ChatGPT (версія GPT-4) для редагування англійської версії анотації.

Конфлікт інтересів

Автори заявляють про відсутність конфлікту інтересів та підтверджують, що під час підготовки цієї роботи не існувало жодних комерційних, фінансових чи інших взаємовідносин, які могли б бути розцінені як такі, що здатні вплинути на результати дослідження або їх інтерпретацію. Робота виконана відповідно до принципів академічної доброчесності, етичних норм проведення наукових досліджень та вимог редакційної політики щодо запобігання конфлікту інтересів.

Список використаної літератури

1. Valverde, S., & Solé, R. V. (2015). Punctuated equilibrium in the large-scale evolution of programming languages. Journal of The Royal Society Interface, 12(107), 20150249. https://doi.org/10.1098/rsif.2015.0249

2. Meske, C., Hermanns, T., & von der Weiden, E. (2025). Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. IEEE Access, 13, 213242-213259. https://doi.org/10.1109/ACCESS.2025.3645466

3. Drosos, I., Sarkar, A., & others. (2025). Vibe coding: Programming through conversation with artificial intelligence. arXiv preprint arXiv:2506.23253. https://doi.org/10.48550/arXiv.2506.23253

4. Karpathy, A. (2025, February). Vibe coding. X. https://x.com/karpathy

5. Waseem, M., Ahmad, A., & Kemell, K.-K. (2025). Vibe coding in practice: Flow, technical debt, and guidelines for sustainable use. arXiv preprint arXiv:2512.11922. https://doi.org/10.48550/arXiv.2512.11922

6. Fawzy, A., Tahir, A., & Blincoe, K. (2026). Vibe coding in practice: Motivations, challenges, and a future outlook – a grey literature review. In Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26). ACM. https://doi.org/10.1145/3786583.3786866

7. Gómez, O. S. (2025). The twilight of software engineering in the age of vibe coding. ResearchGate.

8. Mohamed, A., Assi, M., & Guizani, M. (2025). The impact of LLM-assistants on software developer productivity: A systematic literature review. arXiv preprint arXiv:2507.03156. https://doi.org/10.48550/arXiv.2507.03156

9. Model Evaluation & Threat Research (METR). (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. METR.

10. Veracode. (2025). 2025 GenAI code security report. Veracode.

11. Anthropic. (2025). AI assistance and coding skills: A randomized controlled trial. Anthropic Research.

12. Willison, S. (2025, March 19). Not all AI-assisted programming is vibe coding. Simon Willison's Weblog. https://simonwillison.net/2025/Mar/19/vibe-coding/

13. Samsyudin, I. (2025). Vibe coding and AI-led conversational programming: Emerging trends in software development. SSRN. https://doi.org/10.2139/ssrn.5469367

14. Center for Security and Emerging Technology (CSET). (2024). Cybersecurity risks of AI-generated code. Georgetown University.

15. Black Duck. (2026). Vibe coding and its implications for application security. Black Duck Blog.

16. Klemmer, J. H., Horstmann, S. A., Patnaik, N., et al. (2024). Using AI assistants in software development: A qualitative study on security practices and concerns. In Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24). ACM.

17. Faros AI. (2025). Research on AI code quality and PR sizes: Industry observations.

O. Prydybailo, R. Prydybailo, V. Yaskevych, Y. Yaskevych

EVOLUTION OF PROGRAMMING PARADIGMS AND THE VIBE CODING PHENOMENON: EVALUATION OF OPPORTUNITIES, RISKS, AND RESEARCH PERSPECTIVES

The rapid development of Artificial Intelligence (AI) systems and Large Language Models (LLMs) has led to a fundamental shift in software development approaches, engendering new paradigms of human-computer interaction. This article is devoted to an in-depth analysis of the historical evolution of programming — from direct hardware manipulation via machine code and early compilers to the establishment of high-level languages, object-oriented programming, and contemporary AI-driven methodologies. Special attention is paid to the concept of "vibe coding" — an emergent development approach that involves creating software products primarily through highly agentic, iterative interactions with AI assistants using natural language prompts, often with minimal manual coding or even without the developer's complete understanding of the generated algorithms and internal system architecture.

Throughout the research, the capabilities, advantages, and drawbacks of this approach are comprehensively evaluated. It has been established that deploying generative AI in a vibe coding format significantly lowers the barrier to entry into the IT industry for individuals without formal technical backgrounds. It facilitates unprecedented acceleration in prototyping and the development of Minimum Viable Products (MVPs). Empirical studies reveal a 26–30% overall increase in task completion productivity and a surge in iteration speed, particularly benefiting junior developers. Conversely, the study identifies and systematizes critical shortcomings and hidden risks associated with vibe coding that jeopardize long-term project viability. The existence of a "productivity paradox" is demonstrated, revealing that highly experienced engineers (Senior developers) can spend up to 19% more time executing complex tasks due to the cognitive overhead required to audit and debug hidden logical errors and AI model "hallucinations."

Furthermore, acute cybersecurity vulnerabilities are scrutinized: according to the Veracode report (2025), aggregated statistical data indicates that up to 45% of tested AI-generated code snippets contain critical vulnerabilities that align with the OWASP Top 10 classification. Additionally, it is proven that excessive, uncritical reliance on AI without strict architectural oversight leads to the accumulation of a specific "Dark Debt," progressive architectural fragmentation (Context Rot), and the gradual degradation of fundamental engineering skills among developers (Skill Atrophy).

Based on the synthesis and analysis conducted, it is substantiated that vibe coding possesses absolute utility as a powerful tool for ultra-fast prototyping, automating routine tasks, and generating boilerplate code. However, this paradigm is fundamentally incapable of completely replacing the rigorous discipline of classical software engineering in the context of building critical, highly scala-

ble, and security-oriented enterprise systems. Key directions for future scientific and applied research are delineated, including the development and implementation of "Context Engineering" methodologies, the creation of integrated Agentic AppSec systems for real-time semantic code verification, and a profound investigation into the socio-technical impact of AI on the educational and professional progression of future engineers.

Keywords: programming paradigms, artificial intelligence, vibe coding, technical debt, cybersecurity, code generation, large language models, software engineering.

Надійшла до редакції: 03.04.2026
Прийнята до друку: 22.05.2026
Опубліковано: 29.06.2026