УДК 004.85:004.415.53:004.89 DOI: 10.31673/2412-4338.2026.019011

Чичкарьов Євген Анатолійович
доктор технічних наук, професор, професор кафедри штучного інтелекту
Державний університет інформаційно-комунікаційних технологій, Київ, Україна
ORCID 0000-0002-4362-5129
e.chychkarov@duikt.edu.ua

Семенов Олександр Віталійович
Державний університет інформаційно-комунікаційних технологій, Київ, Україна
ORCID 0009-0005-9749-3343
sasha.sem.fti@gmail.com

АВТОМАТИЗОВАНА ГЕНЕРАЦІЯ І ТЕСТУВАННЯ КОДУ ДЛЯ ВИРІШЕННЯ ЗАДАЧ МАШИННОГО НАВЧАННЯ ЗАСОБАМИ МУЛЬТИАГЕНТНИХ СИСТЕМ З LLM-ЯДРОМ

Анотація. *У статті розглянуто проблему автоматизації процесів розробки та тестування програмного забезпечення для задач машинного навчання в умовах стрімкого розвитку великих мовних моделей та інтелектуальних інструментів програмування. Зростання складності систем штучного інтелекту та обсягів даних зумовлює необхідність створення нових підходів до організації життєвого циклу розробки ML-систем, які б поєднували можливості автоматизованої генерації коду, аналізу даних та забезпечення якості програмного забезпечення. Метою дослідження є підвищення ефективності створення та валідації програм для задач машинного навчання шляхом розробки інформаційної технології автоматизованої генерації та тестування програмних рішень на основі мультиагентних систем і великих мовних моделей.*

У роботі виконано аналіз сучасних досліджень у галузі AI-Augmented Software Engineering, тестування ML-систем та застосування агентних архітектур для автоматизації програмування. Запропоновано концептуальну архітектуру мультиагентної інформаційної системи, що реалізує повний конвеєр розробки програм для машинного навчання: від підготовки інфраструктури та аналізу даних до генерації програмного коду, тестування і наукової інтерпретації результатів. Система складається з декількох спеціалізованих агентів, серед яких координатор виконання, агент аналізу даних, агент розробки моделей машинного навчання, агент тестування та агент інтерпретації результатів. Для забезпечення надійності розробленої технології запропоновано трирівневу підсистему тестування, яка включає перевірку якості вхідних даних, тестування згенерованого програмного коду та оцінювання якості отриманих моделей машинного навчання за статистичними метриками.

Реалізацію системи виконано у середовищі Google Colab або на локальному комп'ютері з локальним сервером Ollama із використанням бібліотек Python для аналізу даних та машинного навчання, зокрема scikit-learn, pandas та matplotlib, а також із підтримкою великих мовних моделей через локальне або хмарне розгортання. Проведено експериментальну перевірку працездатності системи на стандартних наборах даних для задач класифікації та регресії. Отримані результати демонструють ефективність запропонованого підходу та підтверджують можливість використання мультиагентних систем і великих мовних моделей для автоматизації створення та тестування програмних рішень у галузі машинного навчання.

Ключові слова: *машинне навчання, мультиагентні системи, великі мовні моделі, автоматизація програмування, тестування програмного забезпечення, AI-augmented software engineering.*

Yevhen Chychkarov
Doctor of Technical Sciences, Professor, Professor of Department of Artificial Intelligence
State University of Information and Communication Technologies, Kyiv, Ukraine
ORCID 0000-0002-4362-5129
e.chychkarov@duikt.edu.ua

Oleksandr Semenov
Academic degree, Academic title, position
State University of Information and Communication Technologies, Kyiv, Ukraine
ORCID 0009-0005-9749-3343
sasha.sem.fti@gmail.com

AUTOMATED CODE GENERATION AND TESTING FOR SOLVING MACHINE LEARNING PROBLEMS USING MULTI-AGENT SYSTEMS WITH LLM-KERNEL

Abstract. *The article considers the problem of automating the processes of developing and testing software for machine learning tasks in the context of the rapid development of large language models and intelligent programming tools. The increasing complexity of artificial intelligence systems and the growing volume of data require the development of new approaches to organizing the lifecycle of ML systems that combine automated code generation, data analysis, and software quality assurance. The aim of this study is to improve the efficiency of developing and validating software for machine learning tasks by designing an information technology for automated generation and testing of software solutions based on multi-agent systems and large language models.*

The article analyzes recent research in the fields of AI-augmented software engineering, testing of machine learning systems, and the application of agent-based architectures for programming automation. A conceptual architecture of a multi-agent information system is proposed that implements the complete pipeline for developing machine learning programs, ranging from infrastructure preparation and data analysis to program code generation, testing, and scientific interpretation of results. The system consists of several specialized agents, including an execution coordinator, a data analysis agent, a machine learning model development agent, a testing agent, and a results interpretation agent. To ensure the reliability of the developed technology, a three-level testing subsystem is proposed, which includes validation of input data quality, testing of the generated program code, and evaluation of the quality of the obtained machine learning models using statistical metrics.

The system implementation is performed in the Google Colab environment or on a local computer using a local Ollama server with Python libraries for data analysis and machine learning, including scikit-learn, pandas, and matplotlib, as well as support for large language models deployed locally or in cloud environments. An experimental evaluation of the system was conducted using standard datasets for classification and regression tasks. The obtained results demonstrate the effectiveness of the proposed approach and confirm the feasibility of using multi-agent systems and large language models to automate the development and testing of software solutions in the field of machine learning..

Keywords: *machine learning, multi-agent systems, large language models, programming automation, software testing, AI-augmented software engineering.*

1. Вступ

Впровадження машинного навчання та штучного інтелекту продовжує прискорюватися в усьому світі завдяки нещодавнім досягненням у галузі штучного інтелекту та зростаючому попиту підприємств на інтелектуальну автоматизацію та зниження витрат. Дефіцит робочої сили та кваліфікованих кадрів ще більше прискорює впровадження цих технологій, і приблизно кожна четверта компанія впроваджує штучний інтелект для вирішення обмежень у робочій силі. З огляду на це, прогнозується, що світові ринки машинного навчання та штучного інтелекту різко зростатимуть протягом наступного десятиліття завдяки швидкому розвитку в таких галузях, як автономні агенти, GenAI, обробка природної мови, комп'ютерний зір та зрозумілий штучний інтелект [1].

Світовий ринок машинного навчання стабільно зростає і, за прогнозами, збільшиться з 91,31 мільярда доларів у 2025 році до 1,88 трильйона доларів до 2035 року.

Сегмент машинного навчання як послуги (MLaaS) також очікує швидкого зростання, збільшившись з 45,76 млрд доларів у 2025 році до приблизно 209,63 млрд доларів у 2030 році, що відображає середньорічний темп зростання (CAGR) на рівні 35,58%.

На сучасному етапі розвитку глобальної IT-індустрії забезпечення якості програмного забезпечення для систем машинного навчання (ML) постає як одна зі складних та наукоємних задач. При впровадженні ML-моделей у програмні системи забезпечення якості ML-систем має першорядне значення, і дослідники докладають значних зусиль для визначення специфічних викликів забезпечення якості ML-систем [2-3].

На відміну від традиційного імперативного програмування, де логіка чітко детермінована вихідним кодом, ML-системи базуються на динамічних моделях, поведінка яких залежить від трьох взаємопов'язаних чинників: якості вхідних даних, обраної архітектури алгоритму та параметрів навчання. Це створює «трикутник невизначеності», який традиційні методи автоматизованого тестування (наприклад, unit-тести або інтеграційні тести) не здатні повністю покрити [4].

Стрімкий розвиток методів штучного інтелекту (ШІ) та зростання обсягів даних зумовили необхідність автоматизації процесів машинного навчання. Автоматизоване машинне навчання (AutoML) — напрям, який охоплює автоматичний вибір алгоритмів, налаштування гіперпараметрів, трансформацію ознак і оцінювання якості моделей [5]. Проте сучасні рішення AutoML, як-от Auto-sklearn, H2O.ai або Google AutoML, здебільшого є монолітними системами з обмеженими можливостями адаптації до специфічних задач і предметних областей.

Поява і розповсюдження великих мовних моделей (Gemini, GPT, Gemma, Llama та інші) суттєво змінили уявлення про життєвий цикл розробки програмного забезпечення. Штучний інтелект виступає

не просто як допоміжний інструмент, а як повноправний когнітивний партнер інженера, що призвело до появи парадигми AI-Augmented Software Engineering (AIASE) [6-7].

Взаємозв'язок між нею та завданнями машинного навчання проявляється у двох ключових аспектах.

По-перше, методи машинного навчання використовуються для автоматизації та оптимізації традиційних інженерних завдань [8]. Зокрема, великі мовні моделі демонструють здатність автоматизувати рутинні завдання програмування, що дозволяє розробникам зосередитись на концептуальних аспектах, які потребують критичного мислення та креативності [9].

По-друге, розробка самих ML-систем вимагає специфічних інженерних практик, які суттєво відрізняються від традиційної розробки ПЗ. Як зазначають дослідники, інженерія ML-систем стикається з унікальними викликами, пов'язаними з управлінням даними, валідацією моделей, забезпеченням відтворюваності результатів, та специфічними вимогами до тестування компонентів, що навчаються [10]. Це породжує необхідність у нових інженерних практиках, які б ефективно поєднували традиційні підходи розробки ПЗ з особливостями машинного навчання.

2. Постановка проблеми

Сучасний етап розвитку штучного інтелекту характеризується стрімким зростанням складності архітектур машинного навчання та обсягів даних, що потребує залучення висококваліфікованих фахівців на кожному етапі життєвого циклу розробки. Традиційні підходи до створення ML-рішень передбачають значні витрати часу на рутинні операції: підготовку даних, вибір гіперпараметрів, написання шаблонного коду та формування тестових сценаріїв. Це створює «вузьке місце» у процесах цифровізації, де попит на інтелектуальні сервіси суттєво перевищує можливості команд розробників щодо швидкої ітерації та підтримки якості програмного забезпечення.

Особливої гостроти набуває проблема надійності та валідації згенерованих програмних засобів. Специфіка задач машинного навчання полягає в тому, що некоректність коду може проявлятися не лише через синтаксичні помилки, а й через статистичну нестабільність моделей, перенавчання або невідповідність метрик якості бізнес-вимогам. Існуючі методи автоматизованого тестування в багатьох випадках виявляються неефективними для ML-систем, оскільки не враховують імовірнісну природу їх результатів, — що зумовлює необхідність розробки нових підходів до інтелектуальної валідації в режимі реального часу.

Поява великих мовних моделей (LLM) відкрила нові можливості для автоматизації кодування, проте їх пряме застосування часто призводить до генерації фрагментарного коду, який потребує ручної доробки і не гарантує цілісності фінального продукту. Відсутність інформаційної технології, яка б органічно поєднувала когнітивні можливості LLM із потужністю мультиагентних систем для розподіленого проектування, генерації та багаторівневого тестування ML-програм, визначає важливість і необхідність даного дослідження.

Таким чином, створення середовища, в якому спеціалізовані агенти здатні не лише генерувати код, а й автономно оцінювати його якість та стабільність із застосуванням сучасних непараметричних методів, моделей є актуальною задачею і відповідає на нагальні потреби індустрії.

3. Аналіз останніх досліджень і публікацій

Перехід від ізольованих великих мовних моделей (LLM) до агентних систем (LLM-based Agents) представляє фундаментальну зміну парадигми в автоматизації програмування. На відміну від традиційної генерації коду, агентні системи характеризуються автономністю, розширеним охопленням усього життєвого циклу розробки та підвищеною інженерною практичністю [11-12]. Реалізація парадигми AIASE як теоретичної основи застосування ШІ при розробці та тестуванні ПЗ зумовила широке впровадження мультиагентних систем, у межах яких спеціалізовані агенти взаємодіють між собою: один генерує код, інший тестує, третій документує [12].

Класичні монолітні підходи до побудови ML-конвеєрів демонструють суттєві обмеження з точки зору адаптивності та пояснюваності. Мультиагентні системи пропонують альтернативу, засновану на принципі розподіленої відповідальності та кооперативної взаємодії автономних агентів [13]. Попри активні дослідження інтеграції MAC з AutoML [14], більшість підходів фокусується на оптимізації окремих компонентів конвеєра, тоді як комплексна міжагентна координація залишається відкритою проблемою. Запропонована шестишарова архітектура системно вирішує її через чітке розмежування відповідальності між шарами і агентами.

Поява великих мовних моделей (LLM) як когнітивного ядра агентів суттєво розширила можливості MAC: агенти набули здатності розуміти природномовні інструкції, генерувати програмний код, інтерпретувати статистичні результати та забезпечувати обґрунтування прийнятих рішень [15].

Яскравим прикладом агентного підходу є Gemini Data Science Agent — спеціалізований інструмент Google, інтегрований у Google Colab, який автоматизує повний цикл аналізу даних на базі моделей Gemini 2.0. З весни 2025 року він став доступний широкому колу користувачів. Інші провідні компанії також розвивають власні агентні платформи.

Хоча Gemini Data Science Agent забезпечує високу швидкість розробки, для проектів з критично важливими конфіденційними даними (медицина, фінанси, державний сектор) є ризик витоку конфіденційної інформації через хмарну обробку. Крім того, виникають питання гарантій юридичного захисту та ризики порушення локального законодавства щодо захисту персональних або конфіденційних даних [16]. Тому доцільніше використовувати локальні архітектури на базі локальній LLM-моделі (наприклад, Llama3 або Gemma3). Це дозволяє реалізувати парадигму AIASE без компромісів у безпеці.

Відсутність витоку конфіденційних даних на інші сервери забезпечує для локального розгортання великих мовних моделей без використання хмарних API було застосовано фреймворк Ollama [17], який забезпечує запуск сучасних LLM у середовищі з обмеженими ресурсами. Перспективи використання локальних великих мовних моделей qwen2.5, gemma3, llama3.2 в системах кібербезпеки підтверджується результатами [18].

В огляді [19] доведено, що через використання спільних та спеціалізованих можливостей множинних агентів, мультиагентні LLM-системи забезпечують автономне вирішення проблем, покращують надійність та надають масштабовані рішення для управління складністю реальних програмних проектів.

Забезпечення якості програмного забезпечення залишається однією з найбільш критичних та ресурсномістких фаз життєвого циклу розробки. На думку авторів огляду [20], який присвячено використанню LLM у тестуванні програмного забезпечення, LLM виявляються особливо ефективними у середній та пізній фазах життєвого циклу тестування.

За даними [20] протягом середньої фази тестування LLM успішно застосовуються для різноманітних завдань підготовки тестових випадків, включаючи генерацію модульних тестів, створення тестових оракулів та формування системних тестових вхідних даних. У пізніх фазах, таких як виправлення помилок та підготовка звітів про тести/помилки, LLM використовуються для аналізу багів, налагодження та автоматичного ремонту коду. Найчастіше використовуваною моделлю у зібраних в [20] дослідженнях є ChatGPT, що визнається за виняткову продуктивність у різноманітних завданнях.

На думку [21], нещодавні досягнення у великих мовних моделях трансформують те, як розробляються та оцінюються конвеєри машинного навчання. Отримані результати дозволили скористатись новим типом робочого навантаження – агентним пошуком по конвеєру, в якому автономні або напівавтономні агенти генерують, перевіряють та оптимізують повні конвеєри машинного навчання. Ці агенти переважно працюють над популярними бібліотеками машинного навчання на Python та демонструють високий рівень дослідницької поведінки.

На думку [22], великі мові моделі загального призначення часто мають труднощі у спеціалізованих контекстах програмування, де необхідно використовувати специфічні для предметної області бібліотеки, API або конвенції. Налаштування менших моделей з відкритим кодом пропонує економічно ефективну альтернативу опорі на великі власницькі системи.

Мультиагентні системи з LLM-ядром останнім часом активно досліджуються. На думку [23], актуальним завданням є створення інтелектуального агента, здатного допомагати користувачам виконувати завдання AutoML за допомогою інтуїтивно зрозумілих, природних розмов без необхідності глибоких знань базових процесів машинного навчання. Автори [23] запропонували ідею інтелектуального агента на основі ChatGPT для створення природного інтерфейсу між користувачами та моделями машинного навчання (Scikit-Learn). Аналогічний підхід використано розробниками пакету Scikit-LLM [24], який поєднав можливості надійних мовних моделей, таких як ChatGPT, з універсальною функціональністю scikit-learn. Ще одним прикладом мультиагентної системи, яка направлена на вирішення задач машинного навчання, є AutoML-Agent [25]. Однак більшість з відомих прикладів орієнтовані на платні хмарні API та не реалізують механізмів мутаційного тестування або bootstrap-оцінювання якості.

На думку [26], існує необхідність покращення інфраструктури тестування, якості документації та практик обслуговування для забезпечення довгострокової надійності та стійкості при створенні мультиагентних інформаційних систем.

Системи на основі машинного навчання привносять серйозні виклики для забезпечення якості, оскільки їх поведінка визначається не тільки кодом, що їх реалізує, але й даними, використаними для навчання.

Дослідження [2] виявило необхідність створення спеціалізованих підходів та нових метрик для нейромережевих компонентів систем машинного навчання. Залишаються відкритими питання щодо необхідної кількості навчальних даних, потреби у перенавчанні, масштабованості методів та відсутності стандартизованих бенчмарків.

Незважаючи на прогрес у автоматизованій генерації та тестуванні коду з LLM, залишаються критичні виклики:

1. Згенерований код може містити логічні дефекти, проблеми продуктивності або вразливості безпеки, які важко виявити стандартними тестами.
2. Існує розрив між академічними досягненнями та промисловою практикою. Дослідження проводяться на спрощених бенчмарках, що не відображають складність реальних систем з застарілими залежностями та застарілим кодом.
3. Ефективна генерація коду вимагає врахування документації, API-специфікацій та додаткових вимог або правил, які можуть бути недосяжні або погано формалізовані.
4. При побудові мультиагентних систем необхідні архітектурні рішення, що забезпечують баланс між якістю досягнутих рішень та обчислювальною ефективністю, вимогами безпеки.

Огляд літератури показує, що автоматизоване створення та тестування програм з великими мовними моделями інтенсивно розвивається. Еволюція від автодоповнення коду до мультиагентних систем, що управляють повним життєвим циклом розробки, відображає зміну парадигми у програмній інженерії.

Особливої уваги заслуговує специфіка тестування систем машинного навчання, які вносять унікальні виклики через подвійну природу їх поведінки, що визначається як кодом, так і даними. Традиційні методи забезпечення якості виявляються недостатніми для таких систем, що вимагає розробки спеціалізованих підходів та метрик.

Розробка інформаційної технології автоматизованого створення і тестування програм машинного навчання з використанням великих мовних моделей є актуальною задачею, що може революціонізувати програмну інженерію через органічне поєднання людського та штучного інтелекту.

4. Мета і задачі дослідження

Мета дослідження полягає у підвищенні ефективності процесів розробки та забезпечення якості програмного забезпечення для систем машинного навчання шляхом створення інформаційної технології автоматизованої генерації та валідації програм для вирішенн завдань машинного навчання на основі мультиагентних систем з використанням великих мовних моделей.

Для досягнення цієї мети необхідно вирішити наступні завдання:

1. Виконати систематичний огляд методів генерації та тестування ПЗ з LLM;
2. Провести аналіз особливостей розробки та тестування програм для вирішення завдань машинного навчання;
3. Розробити раціональні варіанти мультиагентної архітектури для автоматизованої генерації коду;
4. Створити підходи до валідації і тестування з урахуванням специфіки машинного навчання;
5. Розробити декілька версій програмного комплексу, що інтегрує всі методи і варіанти розгортання;
6. Виконати практичну перевірку ефективності розроблених підходів і програмного забезпечення та порівняння з існуючими рішеннями.

5. Результати дослідження

5.1 Архітектура мультиагентної системи

Для досягнення мети дослідження було розроблено концептуальну модель мультиагентної інформаційної технології, що базується на інтеграції спеціалізованих агентів для автоматизації повного життєвого циклу розробки програмного забезпечення у сфері машинного навчання.

Запропоновану систему було розгорнуто в трьох варіантах з метою порівняльного аналізу ефективності різних підходів до автоматизації створення та тестування згенерованого коду та рішень:

1. Варіант з цілком хмарним розгортанням передбачав використання інтегрованого ШІ Gemini у середовищі Google Colab з можливістю безпосередньої інтеграції у Jupyter Notebook.

2. Варіант з цілком локальним розгортанням передбачав використання локального сервера Ollama з відкритими великими мовними моделями для забезпечення повного контролю над даними та незалежності від зовнішніх сервісів.
3. Змішаний варіант передбачав розгортання сервера Ollama з великими мовними моделями (Gemma3, Llama3, Mistral і т.п.) в контейнері Google Colab.

Для локального і змішаного варіантів уніфікований інтерфейс для взаємодії агентів із LLM, пам'яттю та інструментами було забезпечено за допомогою пакету LangChain. реалізують задачі аналізу даних та машинного навчання. Для вирішення власне задач машинного навчання було використано Python-бібліотеки scikit-learn, pandas, matplotlib і seaborn; для виконання і створення тестів – pytest.

Середовище Google Colab забезпечувало оперативну пам’ять 12.7 GB (обмеження безкоштовної версії), GPU Tesla T4 (15 GB VRAM, доступний періодично), обсяг тимчасового сховища 107.7 GB.

З грудня 2024 року платформа Google Colab надає вбудований доступ до моделі Gemini через спеціальний синтаксис @ai_model у кодових комірках або через чат-інтерфейс у правій панелі. Це усуває необхідність у API-ключах та зовнішніх бібліотеках, забезпечуючи нативну інтеграцію AI-асистента безпосередньо в робочий процес розробки.

Для локального розгортання було використано ноутбук з Ubuntu 25.10, 1024 GB дискового простору, 16 GB оперативної памяти, процесором Intel Core i5, GPU NVIDIA GTX 1650 (4 GB GDDR6).

Архітектура системи для локального або змішаного вариантів (на сервері Ollama) складалася з шести спеціалізованих агентів, які формувались у вигляді класів на Python:

- Агент 1 – Координатор (Supervisor): Аналізує опис задачі або вхідний запит користувача та формує ь послідовність виконання завдань (високорівневий план).
- Агент 2 – Експерт в галузі машинного навчання, кодувальник (Data Scientist): Перетворює план у виконуваний Python-код, обирає моделі і алгоритми машинного навчання, виконує навчання моделей.
- Агент 3 – Експерт в галузі завантаження і попередньої обробки даних (Data Analysis Agent) Завантажує дані, виконує обрання ознак та попередній аналіз даних.
- Агент 4 – Тестувальник (QA Agent): Генерує тести для перевірки цілісності даних та юніт-тести для коду, проектує тестові сценарії, забезпечення тестового покриття, валідація кінцевих випадків.
- Агент 5 – Науковий співробітник з машинного навчання (Interpreter Agent): Аналізує вивід моделі, інтерпретує результати, виконує аналіз важливості ознак, виконує наукове обґрунтування.
- Агент 6 - Інфраструктурний агент (Infrastructure Agent): завантаження і встановлення Ollama, LLM, LangChain.

Усі агенти читали та записували дані через єдиний об'єкт AgentContext. Це усунуло жорстке зв'язування між агентами і дозволяє їм працювати незалежно від порядку ініціалізації.

В локальній та змішаній версіях мультиагентна система була організована у шість функціональних шарів, кожен з яких відповідає за визначений аспект роботи системи. Розподіл на шари відповідає принципу розділення відповідальності та дозволяє замінювати компоненти без перебудови всієї системи.

В хмарній версії інфраструктурний агент і відповідний шар було вилучено. Важливою технічною особливістю системи є використання Gemini безпосередньо у середовищі Google Colab через механізм автентифікації облікового запису Google. Бібліотека google.generativeai дозволяє налаштувати Gemini без явного введення API-ключа, використовуючи Application Default Credentials (ADC) — токен, що автоматично генерується при виконанні google.colab.auth.authenticate_user().

Огляд шарів мультиагентної системи та їх відповідність агентам наведено в таблиці 1.

Шари L-0 та L-1 є підготовчими та управлінськими: вони забезпечували готовність інфраструктури та координацію виконання. Шари L-2 — L-5 є виконавчими, вони утворювали послідовний ML-конвеєр: дані → модель → перевірка → інтерпретація.

5.2 Архітектура підсистеми тестування

При розробці агента-тестувальника (QA Agent) було реалізовано трирівневу піраміду тестування, де кожен рівень перевіряє окремий аспект надійності системи. Тести виконувались послідовно: спочатку тести якості даних (без якісних даних навчання неможливе), потім тести якості коду (перевірка коректності коду), і нарешті тести якості рішень (перевірка якості отриманого рішення задачі машинного навчання).

Тести якості даних (Data Quality Tests — DQT) перевіряли вхідний набір даних до виконання будь-якого моделювання. Невдача будь-якого критичного тесту (DQT-01, DQT-04) зупиняла конвеєр і повертала управління SupervisorAgent із повідомленням про помилку. Перелік тестів якості даних наведено в таблиці 2.

Таблиця 1

Шари мультиагентної системи та їх відповідність агентам

№ шару	Назва шару	Агент	Основна функція	Технології
L-0*	Інфраструк-турний	Infrastructure Agent (A6)	Встановлення Ollama, завантаження LLM, налаштування LangChain, перевірка GPU/CPU	subprocess, requests, CUDA
L-1	Оркестра-ційний	Supervisor Agent (A1)	Приймає запит, формує план, маршрутизує задачі між агентами, збирає результат	AgentContext, або LangChain + LLM, або ШІ Gemini
L-2	Аналізу даних	DataAnalysis Agent (A3)	Завантаження даних, EDA, відбір ознак, попередня обробка	pandas, seaborn, sklearn
L-3	Моделювання	DataScientist Agent (A2)	Генерація Python-коду, вибір алгоритмів, навчання моделей ML	sklearn, LangChain або Gemini+exec()
L-4	Тестування та забезпечення якості	QA Agent (A4)	Тести даних (DQT), тести коду (CQT), юніт-тести, граничні випадки	pytest, unittest, coverage
L-5	Інтерпретації	Interpreter Agent (A5)	Аналіз метрик, важливість ознак, наукове обґрунтування висновків	matplotlib, SHAP, LLM або ШІ Gemini

* - локальний або змішаний варіант розгортання системи

Таблиця 2

Перелік тести якості даних (DQT)

Тест	Назва	Критерій	Пріоритет	Дія при провалі
DQT-01	Мінімальний розмір	≥ 100 записів	критичний	Зупинити конвеєр
DQT-02	Рівень пропусків	<50% у будь-якому стовпці	високий	Попередження + авто-імпутація
DQT-03	Дублікати рядків	<10% від загального обсягу	середній	Видалити дублікати
DQT-04	Цільова змінна	Стовпець target присутній	критичний	Зупинити конвеєр
DQT-05	Баланс класів	min/max ratio > 0.10	високий	Попередження
DQT-06	Константні ознаки	Відсутні стовпці з 1 унікальним значенням	середній	Автовидалення
DQT-07	Типи даних	Числові ознаки не є object-типом	низький	Авто-перетворення
DQT-08	Аномалії (IQR)	Викиди <5% у кожній ознаці	низький	Логування

Тести якості коду (Code Quality Tests — CQT) перевіряли Python-код, згенерований DataScientistAgent або написаний вручну. Тести застосовували модуль ast (статичний аналіз) та виконавче середовище (динамічний аналіз). Результатом був відсоток покриття та перелік виявлених проблем. Перелік тестів якості коду наведено в таблиці 3.

Таблиця 3

Перелік реалізованих тестів якості коду (CQT)

Тест	Назва	Метод перевірки	Критерій	Пріоритет
CQT-01	Синтаксис Python	ast.parse(code)	Без SyntaxError	крітичний
CQT-02	Наявність імпортів	Пошук рядків import	Присутні необхідні пакети	високий
CQT-03	fit() без помилок	try/except + model.fit()	Returncode = 0	крітичний
CQT-04	Розмір predict()	len(preds)==len(y_test)	Форми співпадають	високий
CQT-05	Відтворюваність	predict(), порівняння	np.array_equal==True	високий
CQT-06	Відсутність NaN	np.isnan(preds)	NaN = 0	високий
CQT-07	Граничні випадки	X_test із 1 рядком	Без IndexError	середній
CQT-08	Пустий вхід	X_test = DataFrame()	Коректний виняток	низький

Тести якості рішень (Solution Quality Tests — SQT) оцінювали якість навченої моделі машинного навчання за статистичними критеріями.

Ці тести є найбільш інформативним рівнем тестування з наукової точки зору: вони безпосередньо вимірюють придатність навченої ML-моделі для вирішення поставленої задачі. Порогові значення метрик обґрунтовані практикою ML-інженерії: accuracy ≥ 0.72 відповідає рівню, при якому модель перевершує випадкове вгадування принаймні у 72% випадків; перевірка перенавчання (різниця метрик якості менш, ніж 12% - overfit gap) є добре відомою оцінкою [3-4].

Bootstrap-тест SQT-05 використовує непараметричну оцінку стабільності: при ширині 95% довірчого інтервалу менше 0.10 вважається, що модель демонструє стабільну поведінку і не є чутливою до конкретного складу тестової вибірки [3-4].

Перелік використаних тестів якості рішень наведено в таблиці 4.

Таблиця 4

Перелік використаних тестів якості рішень (SQT)

Тест	Метрика	Поріг	Обґрунтування порогу	Пріоритет
SQT-01	Accuracy	≥ 0.72	Мінімум для виробничого використання	високий
SQT-02	F1-macro	≥ 0.65	Стійкість до дисбалансу класів	високий
SQT-03	ROC-AUC	≥ 0.75	Якість ранжування (бінарна кл.)	високий
SQT-04	Overfit gap	< 0.12	Допустима різниця train/test accuracy	критичний
SQT-05	Bootstrap CI	< 0.10	Стабільність оцінки (30 ітерацій)	середній
SQT-06	Precision	≥ 0.65	Для задач із дорогими хибнопозитивними	середній
SQT-07	Recall	≥ 0.65	Для задач із дорогими хибнонегативними	середній

В створеній мультиагентній системі було реалізовано цикл автоматичної корекції. Якщо агент тестування отримував помилку при запуску кода, виконувалася повторна корекція коду. При використанні простіших LLM можливе виникнення галюцинацій, коли помилку неможливо було виправити автоматично. Найчастіше виникало питання помилкового імпорту пакетів-компонентів scikit-learn, коли розробники scikit-learn вже поміняли розміщення пакету, але локальна LLM про це ще не знає. При використанні більш потужної моделі подібні питання або не виникали, або їх було значно менше. Один зі шляхів їх вирішення - надання прямої вказівки при формулюванні завдання, наприклад:

- Використати імпорт from sklearn.model_selection import learning_curve для завантаження learning_curve.
- Використати імпорт from sklearn.preprocessing import StandardScaler для завантаження StandardScaler.

5.3. Алгоритм прийняття рішень за результатами тестування

Після завершення всіх тестів QAAgent обчислює інтегральний бал тестового покриття та рекомендує дію для SupervisorAgent (наведено приклад коду відповідного методу):

```
def evaluate_test_results(ctx: AgentContext) -> str:
    """
    Алгоритм прийняття рішень на основі результатів тестування.
    Повертає: 'PASS' | 'WARNING' | 'FAIL'
    """
    all_tests = {**ctx.dqt_report, **ctx.cqt_report, **ctx.sqt_report}
    total     = len(all_tests)
    passed    = sum(1 for t in all_tests.values() if t['passed'])
    tcs       = passed / total * 100  # Test Coverage Score

    # Критичні тести — будь-який провал = FAIL
    critical_keys = ['DQT-01 Розмір >=100 рядків',
                     'DQT-04 Цільова змінна існує',
                     'CQT-01 Синтаксис Python',
                     'CQT-03 fit()+predict() без помилок',
                     'SQT-04 Overfit gap <0.15']
    critical_failed = [k for k in critical_keys
                       if k in all_tests
                       and not all_tests[k]['passed']]
```

```
if critical_failed:
    print(f'[QA] FAIL — критичні тести провалено: {critical_failed}')
    return 'FAIL'
elif tcs >= 85:
    print(f'[QA] PASS — TCS={tcs:.1f}% ({passed}/{total})')
    return 'PASS'
elif tcs >= 70:
    print(f'[QA] WARNING — TCS={tcs:.1f}% ({passed}/{total})')
    return 'WARNING'
else:
    print(f'[QA] FAIL — TCS={tcs:.1f}% нижче порогу 70%')
    return 'FAIL'
```

5.4 Результати перевірки роботи мультиагентної системи

Для перевірки роботи мультиагентної системи в усіх варіантах розгортання за її допомогою було вирішено декілька стандартних задач:

- Бінарна класифікація набору даних Breast Cancer Wisconsin (569 зразків, 30 ознак);
- Мультикласова класифікація набору даних Iris (150 зразків, 4 ознаки, 3 класи);
- Дослідження впливу різних факторів на вартість житла за допомогою регресійного аналізу набора даних California Housing (20640 зразків, 10 ознак).

За результатами тестування системи в варианті розгортання в Google Colab (або локальному) встановлено, що поведінка Gemma3:12b з точки зору генерації коду краща у порівнянні з Gemma3:4b.

Після вирішення питань правильного імпорту пакетів усі типи тестів системи (DQT + CQT + SQT) проходили успішно. Час виконання повного конвеєра (без встановлення моделі) — 4-8 хв. на Google Colab Free (T4).

Слід відмітити, що при використанні повністю хмарної системи з ШІ Gemini питання невірного імпорту пакетів scikit-learn практично не виникало.

Запропоновані мультиагентною системою рішення демонструють гарну пояснюваність результатів завдяки LLM-інтерпретації та гнучкість налаштування через заміну агентів.

Більш складну перевірку було виконано на прикладі напівсинтетичного датасету на основі відомого набору даних Superconductivity Dataset [27], який містить 21 263 рядків даних про надпровідники з 81 ознакою фізико-хімічних властивостей. Ці дані було використано в [25] для побудови статистичної моделі прогнозування критичної температури надпровідності на основі ознак, отриманих з хімічної формули надпровідника. За даними [28] досягнуте значення середньоквадратичної помилки дорівнювало ±9.5 K. Для підвищення складності задачі було згенеровано 70 додаткових стовпчиків ознак понад оригінальних 81. Крім того, при побудові набора даних було додано пропуски, мультиколінеарність і нелінійні залежності. Всього згенерований таким чином набір даних містив 151 ознаку.

Для порівняння поведінки згенерованого коду на тому ж самому наборі даних було виконано побудова моделі за допомогою підходу Automated Machine Learning (було використано пакет flaml). Розрахунки було виконано в середовищі google colab в обох випадках.

Найкращий результат, який було досягнуто за допомогою AutoML(flaml). Середньоквадратична помилка склала ±8.995 K при використанні ансамблевої оцінки, використаний ресурс часу – 600 с.

При використанні мультиагентної системи отримані результати залежали від обраного алгоритму побудови моделі: при використанні RandomForestRegressor середньоквадратична помилка склала ±9.03 K. При генерації коду варианти алгоритмів побудови і навчання моделі, які необхідно реалізувати, треба прописувати явно.

Зведені результати розв'язання задач машинного навчання за допомогою розробленої мультиагентної системи наведено в таблиці 5. Для оцінки повноти покриття коду було використано пакет pytest-cov.

В більшості випадків цикл автоматичної корекції виконувався продуктивно. При перегляді згенерованих версій коду для аналізу Superconductivity було виявлено наступні причини помилок: неправильний синтаксис UCI fetch API або помилки при побудові синтетичних стовпців з мультиколінеарністю.

Для вимірювання адекватності тестових наборів було виконано мутаційне тестування, яке вносило синтаксичні мутації до вихідного коду та перевіряючи, чи виявляють тести ці зміни. Для виконання мутаційного тестування було використано пакет mutmut. Отримані звіти mutmut для всіх

задач підтверджують, що код є не лише робочим, а й "некрихким" — він витримує перевірку на зміну логіки.

Таблиця 5

Виконання та тестування тестових завдань

Параметр / Завдання	Регнесійний аналіз California Hosung	Регресійний аналіз Supercon-ductivity	Генерація і класифікація синтетичних даних
Ітерації коду (A2)	1	3	2
Ітерації тестів (A4)	1	3	2
IPR (Initial Pass Rate)	100%	0%	0%
Статус виконання	Успішно (з 1-ї спроби)	Успішно (після виправлень)	Успішно (після виправлень)
CSR (Corrected Success Rate)	100%	100%	100%
Проходження тестів	100%	100% (відкореговані)	100% (відкореговані)

Наявність інфраструктурного агента спрощує зміну використаної LLM. Було виконано дослідження поведінки різних типів LLM (Gemma3, Llama 3.2, Mistral 7B, Phi-4, Qwen 2.5, DeepSeek-R1, CodeLlama). Встановлено, що поведінка моделей розрізняється в першу чергу в залежності від кількості параметрів. Наприклад, для серії моделей Gemma3:1B, Gemma:4B, Gemma3:12B якість коду, якість аналізу і підтримка кирилиці покращується зі зростанням кількості параметрів, на яких було навчено модель.

Порівняльна характеристика декількох LLM з точки зору підтримки багатомовності наведено в таблиці 6.

Порівняння двох варіантів розгортання мультиагентної системи (хмарного ы змішаного) наведено з декількома LLM наведено в таблиці 7.

Таблиця 6

Порівняльна характеристика деяких великих мовних моделей

Параметр	Qwen 2.5 (1.5B/7B)	Gemma 3 (4B/12B)	Llama 3.2 (1B/3B)
Ефективність токенізатора	Найвища. Спеціально оптимізований для багатомовності.	Висока. Використовує сучасний токенізатор Google (256к).	Середня. Орієнтована переважно на англійську та західні мови.
Розуміння української	Відмінне, майже без акценту.	Гарне, але інколи зустрічаються кальки з англійської.	Базове/Середнє. Може робити помилки в складних відмінках.
Логіка в коді (Python)	Дуже висока.	Висока (традиційна сильна сторона Google).	Добра для малих моделей.

Таблиця 7

Технічні характеристики чотирьох варіантів розгортання MAC

Характеристика	B1: Gemini	B2: Gemma 3	B3: Llama 3	B4: Mistral 7B
LLM-модель	gemini-1.5-flash	gemma3:4b	llama3.2:3b	mistral:7b
Кількість параметрів	~150B (хмара)	4B (локально)	3B (локально)	7B (локально)
VRAM Colab T4	0 GB	~3.5 GB	~2.8 GB	~5.5 GB
Розмір завантаження	0 GB	~2.7 GB	~2.1 GB	~4.1 GB
Час підготовки (L-0)	~30 с	~8–12 хв	~6–9 хв	~12–18 хв
Швидкість генерації	~500 tok/s	~22 tok/s	~30 tok/s	~15 tok/s
Час відгуку L-1 (план)	~2–4 с	~25–45 с	~18–35 с	~40–70 с
Час відгуку L-5 (звіт)	~4–8 с	~40–70 с	~30–55 с	~60–100 с
Контекстне вікно	128K токенів	128K токенів	128K токенів	32K токенів
Підтримка кирилиці	Відмінна	Добра	Добра	Добра
Ліміт запитів (Free)	60 запитів/хв.	Необмежено	Необмежено	Необмежено

В умовах обмеження пам’яті GPU (4 ГБ – локальний варіант розгортання), модель Gemma3 продемонструвала кращу адаптивність. Вона швидше завантажується та вивантажується з пам'яті.

Питання конфіденційності та приватності даних є ключовим диференціатором при виборі варіанту для реальних проектів.

Варіант B1 (Gemini) вимагає передачі промптів – включаючи фрагменти даних, згенерований код та ML-результати – на сервери Google. Це може суперечити корпоративним політикам захисту даних або умовам безпеки для конфіденційних датасетів (медичні, фінансові, персональні дані).

Варіанти B2–B4 (Ollama) виконують всі LLM-виклики локально у межах Colab-сесії. Дані не покидають середовище виконання, що відповідає найсуворішим вимогам з конфіденційності. Ця перевага є особливо суттєвою при обробці датасетів, що містять персональну ідентифікаційну інформацію (PII).

Реалізація варіантів B2-B4 (Ollama) за умовами цілком локального розгортання повністю відповідає умовам безпеки та конфіденційності даних.

6. Висновки та перспективи подальших досліджень

У статті запропоновано, обґрунтовано та практично перевірено технологію автоматизованого створення і тестування програм машинного навчання на основі мультиагентних систем із LLM-ядром. Отримані результати дозволяють сформулювати такі висновки:

1. Розроблена шестишарова мультиагентна архітектура (L-0 – L-5) із шістьма спеціалізованими агентами реалізує повний ML-конвеєр – від розгортання інфраструктури до наукової інтерпретації результатів – через принцип розподіленої відповідальності. Єдиний об'єкт стану контексту усуває жорстке зв'язування між агентами і дозволяє замінювати окремі компоненти без перебудови всієї системи. Наявність окремого інфраструктурного агента (A6) забезпечує LLM-агностицизм: заміна мовної моделі або цілком хмарне розгортання не потребує модифікації коду базових агентів.

2. Запропонована трирівнева підсистема тестування (DQT: 8 тестів, CQT: 8 тестів, SQT: 7 тестів) послідовно верифікує якість на кожному рівні ML-конвеєру: вхідні дані, згенерований код і отримане ML-рішення. Механізм автоматичної корекції коду на основі зворотного зв'язку агента-тестувальника підвищує виправлений коефіцієнт успіху до 100% навіть при нульовому початковому коефіцієнті успіху.

3. Верифікація трьох варіантів розгортання виявила чіткий розподіл їх застосовності. Хмарний варіант (Gemini у Google Colab) перевершує локальні рішення за швидкістю (~500 токенів/с проти 15–30 токенів/с) і якістю генерованого коду, але вимагає передачі даних на зовнішні сервери, що є неприйнятним для конфіденційних предметних областей. Локальний і змішаний варіанти (Ollama) забезпечують повну конфіденційність оброблюваних даних і необмежену кількість запитів, однак в умовах обмеження VRAM 4–6 ГБ вимагають ретельного вибору моделі. Для задач із кириличним виводом раціональним вибором є Qwen 2.5, для задач генерації коду — Gemma3 або Llama3.

4. Експериментальна перевірка на п’яти різнорідних наборах даних (Iris, Breast Cancer Wisconsin, California Housing, напівсинтетичний Superconductivity із 151 ознакою, класифікація синтезованого набору даних) підтвердила працездатність системи. На задачі регресії надпровідності розроблена система досягла середньоквадратичної помилки ±9.03 К при використанні RandomForestRegressor, що є порівнянним із результатом спеціалізованої AutoML-системи (FLAML, ±8.995 К за 600 с), але при значно вищій пояснюваності результатів і явному контролі над кожним кроком конвеєру. Мутаційне тестування (пакет mutmut) підтвердило «некрихкість» згенерованого коду щодо логічних мутацій.

5. Виявлено специфічний клас помилок, пов'язаний із застарілими знаннями локальних LLM щодо поточного API бібліотек (зокрема, зміна розташування модулів у scikit-learn). Цей клас помилок ефективно вирішується через явні директиви в системних промптах і значно рідше виникає при використанні хмарних моделей. Відкритим залишається питання автоматичного оновлення контекстних знань локальних LLM без повного перенавчання.

6. Перспективним напрямом є реалізація гібридного варіанту координації: використання хмарних LLM для агентів планування і інтерпретації (де критична якість тексту і актуальність знань) та локальних моделей для агентів генерації коду (де визначальними є відтворюваність і конфіденційність). Також доцільним є розширення підсистеми тестування методами метаморфічного тестування і диференційного тестування для виявлення семантичних дефектів, що не виявляються стандартними CQT-тестами.

Внесок авторів Євген Чичкарьов – концептуалізація; аналіз джерел, підготовка огляду літератури; Олександр Семенов – методика; програмне забезпечення; виконання досліджень; узагальнення результатів.

Декларація про штучний інтелект

При написанні статті штучний інтелект не використовувався.

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] Itransition. (2026, January 27). *Machine learning statistics for 2026: The ultimate list*. Itransition. https://www.itransition.com/machine-learning/statistics

[2] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., & Tonella, P. (2020). Testing machine learning–based systems: A systematic mapping. Empirical Software Engineering, 25(6), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0

[3] Al Alamin, M. A., & Uddin, G. (2021). Quality assurance challenges for machine learning software applications during software development life cycle phases. In 2021 IEEE International Conference on Autonomous Systems (ICAS). IEEE. https://doi.org/10.1109/ICAS49788.2021.9551151

[4] Breck, E., Cai, S., Nielsen, E., Salib, M., & Sculley, D. (2017). The ML test score: A rubric for ML production readiness and technical debt reduction. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 1123–1132). https://doi.org/10.1109/BigData.2017.8258038.

[5] Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated machine learning: Methods, systems, challenges*. Springer. https://doi.org/10.1007/978-3-030-05318-5

[6] Schieferdecker, I. K. (2024). Augmenting software engineering with AI and developing it further towards AI-assisted model-driven software engineering. arXiv. https://arxiv.org/abs/2409.18048

[7] Akhtar, S., & Aftab, S. (2025). Towards AI-augmented software engineering: A theoretical framework. ICCK Journal of Software Engineering, 1, 124. https://doi.org/10.62762/JSE.2025.407864

[8] Nyaga, F. (2025). AI-driven software engineering: A systematic review of machine learning's impact and future directions. Preprints. https://doi.org/10.20944/preprints202504.0174.v1

[9] Yang, Y., Xia, X., Lo, D., & Grundy, J. (2022). A survey on deep learning for software engineering. ACM Computing Surveys, 54(10s), Article 206. https://doi.org/10.1145/3505243

[10] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software engineering for machine learning: A case study. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19) (pp. 291–300). IEEE. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[11] Liu, Y., Wang, Z., & Zhang, L. (2025). A survey on code generation with LLM-based agents. arXiv. https://arxiv.org/abs/2508.00083

[12] Alenezi, M., & Akour, M. (2025). AI-driven innovations in software engineering: A review of current practices and future directions. Applied Sciences, 15. https://doi.org/10.3390/app15031344

[13] Wang, L., Ma, C., Feng, X. *et al.* A survey on large language model based autonomous agents. *Front. Comput. Sci.* **18**, 186345 (2024). https://doi.org/10.1007/s11704-024-40231-1

[14] Wang, Z., Su, K., Zhang, J., Jia, H., Ye, Q., Xie, X., & Lu, Z. (2023). Multi-agent automated machine learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 11960–11969).

[15] Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N., Wiest, O., & Zhang, X. (2024). Large Language Model based Multi-Agents: A Survey of Progress and Challenges. *International Joint Conference on Artificial Intelligence.* https://doi.org/10.48550/arXiv.2402.01680

[16] Coleman, S. & Wilson, D. A Comprehensive Evaluation of Privacy-Preserving Mechanisms in Cloud-Based Big Data Analytics: Challenges and Future Research Directions. Preprints 2026, 2026011025. https://doi.org/10.20944/preprints202601.1025.v1

[17] Huang, D., & Wang, Z. (2025). LLMs at the edge: Performance and efficiency evaluation with Ollama on diverse hardware. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. https://doi.org/10.1109/IJCNN64981.2025.11228317

[18] Palma, G., Cecchi, G., Caronna, M., & Rizzo, A. (2025). Leveraging large language models for scalable and explainable cybersecurity log analysis. *Journal of Cybersecurity and Privacy, 5*(3), 55. https://doi.org/10.3390/jcp5030055

[19] Jiang, N., Liu, K., Chen, T., & Liang, J. (2025). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3712003

[20] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2024.3368208

[21] Phani, A., Strauss, E., & Schelter, S. (2026). stratum: A System Infrastructure for Massive Agent-Centric ML Workloads. https://doi.org/10.48550/arXiv.2603.03589

[22] Freire, L.A., Andal'o, F.A., & Detlefsen, N.S. (2026). Exploring different approaches to customize language models for domain-specific text-to-code generation. https://doi.org/10.48550/arXiv.2603.16526

[23] Hassan, M., Knipper, R., & Karmaker, S. (2023). ChatGPT as your Personal Data Scientist. ArXiv, abs/2305.13657. https://doi.org/10.48550/arXiv.2305.13657

[24] Kulkarni, A., Shivananda, A., Kulkarni, A., Gudivada, D. (2023). Implement LLMs Using Sklearn. In: Applied Generative AI for Beginners. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-9994-4_6

[25] Trirat, P., Jeong, W., & Hwang, S.J. (2024). AutoML-Agent: A Multi-Agent LLM Framework for Full-Pipeline AutoML. *ArXiv, abs/2410.02958*. https://doi.org/10.48550/arXiv.2410.02958.

[26] Liu, D., Upadhyay, K., Chhetri, V., Siddique, A.B., & Farooq, U. (2025). A Large-Scale Study on the Development and Issues of Multi-Agent AI Systems. *2025 IEEE International Conference on Big Data (BigData)*, 7785-7792. https://doi.org/10.48550/arXiv.2601.07136

[27] Hamidieh, K. (2018). Superconductivty Data [Data set]. UCI Machine Learning Repository. https://doi.org/10.24432/C53P47

[28] Kam Hamidieh. (2018). A data-driven statistical model for predicting the critical temperature of a superconductor. Computational Materials Science, 154, 346–354. https://doi.org/10.1016/j.commatsci.2018.07.052

Надійшла до редакції: 01.12.25
Прийнята до друку: 17.03.26
Опубліковано: 30.03.26