

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-
КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Кваліфікаційна наукова праця
на правах рукопису

Семенов Олександр Віталійович

УДК 004.056:004.85:004.932

ДИСЕРТАЦІЯ
МЕТОД ПОБУДОВИ АРХІТЕКТУРИ АВТОМАТИЗОВАНОГО
ТЕСТУВАННЯ НА ОСНОВІ КОМБІНОВАНИХ ПАРАДИГМ ТА
ІШТУЧНОГО ІНТЕЛЕКТУ

122 «Комп'ютерні науки»

12 «Інформаційні технології»

Подається на здобуття ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання на відповідне
джерело

О.В. Семенов

(підпис, ініціали та прізвище здобувача)

Науковий керівник

Чичкарьов Євген Анатолійович

доктор технічних наук, професор

Київ 2026

АНОТАЦІЯ

Семенов О.В. Метод побудови архітектури автоматизованого тестування на основі комбінованих парадигм та штучного інтелекту. Кваліфікаційна наукова праця на правах рукопису. Дисертація на здобуття ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки» (галузь знань 12 «Інформаційні технології»). Державний університет інформаційно-комунікаційних технологій Міністерства освіти і науки України, Київ, 2026.

Роботу присвячено розробці інформаційної технології на основі мультиагентної системи для автоматизації процесів генерації, тестування та валідації програмного забезпечення в задачах машинного навчання та розпізнавання зображень. Запропонований підхід реалізує автономний ітераційний цикл самокорекції коду з використанням спеціалізованих агентів (зокрема інтерпретатора та агента тестування), що забезпечує високу надійність і семантичну коректність складних міждоменних рішень в умовах локального розгортання та вимог до конфіденційності даних.

Актуальність роботи обумовлена тим, що розвиток систем машинного навчання та розпізнавання зображень потребує нових підходів до валідації через імовірнісну природу моделей та складність обчислювальних конвеєрів. Традиційні методи розробки не враховують ризики деградації алгоритмів і специфіку навчальних даних, що суттєво знижує надійність інтелектуальних програмних продуктів.

Застосування великих мовних моделей (LLM) для автоматизації генерації коду обмежене відсутністю гарантій його семантичної коректності та проблемою семантичного розриву при інтеграції різнорідних архітектур, таких як CNN та MLLM. Крім того, вимоги до конфіденційності даних зумовлюють необхідність переходу від хмарних сервісів до локального розгортання інтелектуальних систем.

Тому актуальним науковим завданням є розробка інформаційної

технології на основі мультиагентних систем, що забезпечує безшовну інтеграцію процесів генерації, автономної верифікації та валідації ПЗ. Такий підхід дозволяє формалізувати розробку складних міждоменних рішень, підвищуючи їхню відтворюваність та загальну стійкість у реальних умовах експлуатації.

Об'єктом дослідження є процес автоматизованого тестування програмного забезпечення для систем машинного навчання та комп'ютерного зору.

Предметом дослідження є методи, моделі та алгоритми автоматизованої генерації тестів і валідації програмного коду з використанням великих мовних моделей та мультиагентних систем у задачах розробки і тестування програмного забезпечення для систем машинного навчання та розпізнавання зображень.

Мета роботи полягає у підвищенні ефективності автоматизованого проектування систем машинного навчання шляхом створення шестишарової мультиагентної архітектури, яка забезпечує скорочення термінів розробки та підвищення якості програмних рішень через механізми ітеративної самокорекції та семантичної фільтрації.

Наукова новизна одержаних результатів полягає в тому, що вперше розроблено архітектурну модель шестишарової мультиагентної системи у вигляді впорядкованої сукупності спеціалізованих функціональних рівнів та єдиного об'єкта стану контексту, в якій за рахунок декомпозиції складних інженерних завдань на множину підзадач (планування, генерація, тестування, інтерпретація тощо) та формалізації механізмів семантичного обміну даними між агентами, забезпечується реалізація наскрізного конвеєра побудови інтелектуальних моделей незалежно від конкретного типу обчислювального ядра, що дозволило мінімізувати втрату контекстуальної інформації на всіх етапах розробки та формалізувати вхідні й вихідні параметри взаємодії агентів для оптимізації процесів машинного навчання.

Дістав подальшого розвитку метод багаторівневого динамічного тестування, в якому за рахунок структурної побудови механізму ітеративної самокорекції коду на основі замкненого циклу зворотного зв'язку між агентами-тестувальниками та генераторами, а також композиції параметрів семантичного аналізу помилок і стратегій автоматизованого перепитування (prompt-refinement), реалізовано алгоритм автономного усунення дефектів програмного забезпечення вже на першій ітерації циклу розробки, що дозволило забезпечити стабільність і відтворюваність обчислювальних процесів у недетермінованих середовищах інтелектуальних моделей.

Вперше запропоновано модель трирівневої системи автоматизованої верифікації, в якій за рахунок структурної побудови ієрархічних рівнів контролю та впровадження механізмів адаптивної корекції результатів через інтелектуальний зворотний зв'язок, а також композиції критеріїв оцінювання якості вхідних даних, верифікації програмного коду та валідації фінальних аналітичних рішень, реалізовано можливість послідовного моніторингу станів системи на кожному етапі обробки інформації, що дозволило довести коефіцієнт успішного виконання сценаріїв до максимально можливих значень навіть за умов низької ймовірності успіху на початкових ітераціях.

Вперше розроблено метод мультиагентної побудови гібридних систем розпізнавання зображень, в якому за рахунок інтеграції механізмів трансферного навчання згорткових нейронних мереж із семантичною фільтрацією на основі мультимодальних великих мовних моделей, а також композиції візуальних ознак об'єктів та їх когнітивних дескрипторів, реалізовано підхід до нівелювання семантичного розриву між ознаками нижнього рівня та їх високорівневою інтерпретацією, що дозволило мінімізувати виникнення інтелектуальних галюцинацій і каскадних помилок у процесах класифікації візуальних даних та визначити оптимальну структуру взаємодії агентів для підвищення точності розпізнавання у складних умовах.

Практичне значення одержаних результатів полягає у створенні комплексного програмного інструментарію на основі шестишарової мультиагентної архітектури, що дозволяє автоматизувати до 80–90% процесів генерації тестових сценаріїв. Впровадження методики ітеративної самокорекції коду та спеціалізованих стратегій формування запитів забезпечує автономне усунення до 75% технічних дефектів. Це дозволяє скоротити час виведення програмних продуктів на ринок на 45–50% та знизити витрати на підтримку тестової інфраструктури на 60–70%, що є критично важливим для високонавантажених систем у фінансових, телекомунікаційних та IoT-середовищах. Розроблені прикладні методи побудови гібридних систем розпізнавання образів, що інтегрують згорткові нейронні мережі з мультимодальними моделями, підвищують точність класифікації на 12–15% у складних умовах. Запропонована технологія підтримує локальне розгортання, гарантуючи конфіденційність даних і незалежність від хмарних сервісів. Результати дослідження апробовано в діяльності софтверних компаній, що підтвердило ефективність рішень для створення надійних, відтворюваних та безпечних систем машинного навчання відповідно до сучасних інженерних стандартів.

Ключові слова: Мультиагентна архітектура, генерація коду, автоматизоване тестування, стратегії формування запитів, розробка через тестування, алгоритми машинного навчання, ітеративна самокорекція програмного коду, великі мовні моделі (LLM), мультимодальні мовні моделі (MLLM), згорткові нейронні мережі (CNN), трансферне навчання, гібридні системи розпізнавання зображень, штучний інтелект, глибоке навчання.

SUMMARY

Semenov O.V. Method of building an automated testing architecture based on combined paradigms and artificial intelligence. Qualification scientific work in the form of a manuscript. Dissertation for the degree of Doctor of Philosophy in specialty 122 "Computer Science" (branch of knowledge 12 "Information Technologies"). State University of Information and Communication Technologies of the Ministry of Education and Science of Ukraine, Kyiv, 2026.

The work is devoted to the development of information technology based on a multi-agent system for automating the processes of generation, testing and validation of software in machine learning and image recognition tasks. The proposed approach implements an autonomous iterative cycle of code self-correction using specialized agents (in particular, an interpreter and a testing agent), which ensures high reliability and semantic correctness of complex cross-domain solutions in the conditions of local deployment and data confidentiality requirements.

The relevance of the work is due to the fact that the development of machine learning and image recognition systems requires new approaches to validation due to the probabilistic nature of models and the complexity of computational pipelines. Traditional development methods do not take into account the risks of algorithm degradation and the specificity of training data, which significantly reduces the reliability of intelligent software products.

The use of large language models (LLM) to automate code generation is limited by the lack of guarantees of its semantic correctness and the problem of semantic discontinuity when integrating heterogeneous architectures, such as CNN and MLLM. In addition, data confidentiality requirements necessitate the transition from cloud services to local deployment of intelligent systems.

Therefore, the current scientific task is the development of information technology based on multi-agent systems, which ensures seamless integration of the processes of generation, autonomous verification and validation of software.

This approach allows formalizing the development of complex cross-domain solutions, increasing their reproducibility and overall stability in real operating conditions.

The object of the study is the process of automated software testing for machine learning and computer vision systems.

The subject of the study is methods, models and algorithms for automated test generation and validation of program code using large language models and multi-agent systems in the tasks of developing and testing software for machine learning and image recognition systems.

The purpose of the work is to increase the efficiency of automated design of machine learning systems by creating a six-layer multi-agent architecture, which ensures a reduction in development time and an increase in the quality of software solutions through iterative self-correction and semantic filtering mechanisms.

The scientific novelty of the results obtained lies in the fact that for the first time an architectural model of a six-layer multi-agent system has been developed in the form of an ordered set of specialized functional levels and a single context state object, in which, by decomposing complex engineering tasks into a set of subtasks (planning, generation, testing, interpretation, etc.) and formalizing mechanisms for semantic data exchange between agents, an end-to-end pipeline for building intelligent models is implemented regardless of the specific type of computing core, which made it possible to minimize the loss of contextual information at all stages of development and formalize the input and output parameters of agent interaction to optimize machine learning processes.

The method of multi-level dynamic testing has been further developed, in which, due to the structural construction of the mechanism of iterative self-correction of the code based on a closed feedback loop between testing agents and generators, as well as the composition of semantic error analysis parameters and automated query strategies (prompt-refinement), an algorithm for autonomous elimination of software defects was implemented already in the first iteration of the development cycle, which made it possible to ensure the stability and

reproducibility of computational processes in non-deterministic environments of intelligent models.

For the first time, a model of a three-level automated verification system was proposed, in which, due to the structural construction of hierarchical levels of control and the implementation of mechanisms for adaptive correction of results through intelligent feedback, as well as the composition of criteria for assessing the quality of input data, verification of the program code and validation of final analytical solutions, the possibility of consistent monitoring of system states at each stage of information processing was implemented, which made it possible to bring the coefficient of successful execution of scenarios to the maximum possible values even under conditions of low probability of success in the initial iterations.

For the first time, a method for multi-agent construction of hybrid image recognition systems was developed, in which, due to the integration of transfer navigation mechanisms The development of convolutional neural networks with semantic filtering based on multimodal large language models, as well as the composition of visual features of objects and their cognitive descriptors, an approach to leveling the semantic gap between low-level features and their high-level interpretation was implemented, which made it possible to minimize the occurrence of intellectual hallucinations and cascading errors in the processes of visual data classification and to determine the optimal structure of agent interaction to increase recognition accuracy in complex conditions.

The practical significance of the results obtained lies in the creation of a comprehensive software toolkit based on a six-layer multi-agent architecture, which allows automating up to 80–90% of the test scenario generation processes. The implementation of the iterative code self-correction method and specialized query formation strategies provides autonomous elimination of up to 75% of technical defects. This allows to reduce the time to market of software products by 45–50% and reduce the costs of supporting the test infrastructure by 60–70%, which is critically important for highly loaded systems in financial, telecommunications and IoT environments. Applied methods for building hybrid

pattern recognition systems that integrate convolutional neural networks with multimodal models have been developed, increasing classification accuracy by 12–15% in difficult conditions. The proposed technology supports local deployment, guaranteeing data confidentiality and independence from cloud services. The results of the study have been tested in the activities of software companies, which confirmed the effectiveness of solutions for creating reliable, reproducible and secure machine learning systems in accordance with modern engineering standards.

Keywords: Multi-agent architecture, code generation, automated testing, query generation strategies, test-driven development, machine learning algorithms, iterative self-correction of program code, large language models (LLM), multimodal language models (MLLM), convolutional neural networks (CNN), transfer learning, hybrid image recognition systems, artificial intelligence, deep learning.

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Матеріали й тези наукових конференцій

1. Кароян Р.Р., Семенов О.В., Гніденко М.П. Оптимізації роумінгу у безпроводовій локальній мережі ARUBA. // Науково-практична конференція «Сучасні досягнення компанії HEWLETT PACKARD ENTERPRISE в галузі ІТ та нові можливості їх вивчення і застосування». Збірник тез. – К.: ДУТ, 2022р. С.28-32.
https://duikt.edu.ua/uploads/p_2121_76213401.pdf
2. Семенов О.В., Гніденко М.П. Модифікація моделі репутації та довіри в задачах інформаційної безпеки grid-систем для стійкості до загрози «зловмисні групи хостів» // Науково-практична конференція «Сучасні досягнення компанії HEWLETT PACKARD ENTERPRISE в галузі ІТ та нові можливості їх вивчення і застосування». Збірник тез. – К.: ДУТ, 2022р. С.61-66
https://duikt.edu.ua/uploads/p_2121_76213401.pdf
3. Большаков В.Р., Семенов О.В., Іщеряков С.М., Етичні аспекти застосування штучного інтелекту в науковій та освітній діяльності // IV Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. – К.: ДУІКТ, 2024. - С.192-193. https://duikt.edu.ua/uploads/p_2661_45318838.pdf
4. Бай Я.В., Семенов О.В. Вибір алгоритму для аналізу аудіоданих на основі штучного інтелекту // V Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. – К.: ДУІКТ, 2025. С. 81-84.
https://duikt.edu.ua/uploads/p_2779_68674368.pdf
5. Семенов О.В. Порівняльний аналіз технології palo alto prisma access і рішень secure access service edge (sase) // V Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. – К.: ДУІКТ, 2025. С. 244-245.
https://duikt.edu.ua/uploads/p_2779_68674368.pdf
6. Чичкарьов Є.А., Семенов О.В. Автоматизована генерація коду і тестування програм для вирішення задач машинного навчання з використанням мультиагентної системи / VII Міжнародна науково-практична конференції «Сучасні досягнення компанії Hewlett Packard Enterprise в галузі ІТ та нові можливості їх вивчення і застосування» /11 грудня / Київ: ДУІКТ, - 2025р. – С. 149.
https://duikt.edu.ua/uploads/p_2779_63555250.pdf

7. Чичкарьов Є.А., Семенов О.В. Гібридна технологія розпізнавання зображень з використанням великих мовних моделей і трансферного навчання / VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» /23 квітня / Київ: ДУІКТ, - 2026р. – С. 446-447. https://duikt.edu.ua/uploads/p_2779_63555250.pdf.

Статті в наукових фахових виданнях

1. Семенов, О. В., Серих, С. О., Василенко, В. В., & Гніденко, М. П. (2023). Модифікація моделі репутації та довіри в задачах інформаційної безпеки GRID-систем для стійкості до загрози «зловмисні групи хостів». Наукові записки ДУТ, (1) (3), 46–56. <https://doi.org/10.31673/2786-8362.2023.010505>
2. Чичкарьов, Є., & Семенов, О. (2025). Автоматизована генерація тестових випадків програмного забезпечення за допомогою великих мовних моделей з використанням промпт-інжинірингу. *Телекомунікаційні та інформаційні технології*, 4(89), 122-129. <https://doi.org/10.31673/2412-4338.2025.048914>
3. Чичкарьов, Є., & Семенов, О. (2026). Розробка інформаційної технології автоматизованого створення і тестування програм для вирішення задач машинного навчання з використанням великих мовних моделей. *Телекомунікаційні та інформаційні технології*, 1(90), 107-119. <https://doi.org/10.31673/2412-4338.2026.019011>
4. Чичкарьов, Є., & Семенов, О. (2026). Розпізнавання зображень за допомогою гібридного підходу до на основі мультимодальних великих мовних моделей та згорткових нейронних мереж. *Вісник Приазовського Державного Технічного Університету. Серія: Технічні науки*, 1(53), 85–91. <https://doi.org/10.31498/2225-6733.53.1.2026.359780>
5. Семенов О.В., Чичкарьов Є.А., Мультиагентна методологія автоматизованої розробки та тестування гібридних систем комп'ютерного зору на основі CNN та MLLM. // *Наука і техніка сьогодні – 2026.*, Вип. №4 (58) – С.4709-4725.

ЗМІСТ

АНОТАЦІЯ	2
SUMMARY	6
СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИСЕРТАЦІЇ.....	10
ЗМІСТ	12
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	16
ВСТУП	19
РОЗДІЛ 1 ЕВОЛЮЦІЯ МЕТОДІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ВІД КЛАСИЧНИХ ПАРАДИГМ ДО МУЛЬТИАГЕНТНИХ СИСТЕМ НА ОСНОВІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ	26
1.1 Парадигми і методи тестування програмного забезпечення	26
1.2 Використання великих мовних моделей для генерації програмного коду	27
1.2.1 Генерація коду з використанням LLM	27
1.2.2 Генерація тестів за допомогою LLM.....	29
1.2.3 Мультиmodalьні великі мовні моделі, їх використання для розпізнавання зображень.....	30
1.3 Мультиагентні системи з використанням великих мовних моделей ..	32
1.4 Особливості задачі машинного навчання і обробки зображень з точки зору генерації програмного коду	35
1.4.1 Генерація коду для систем машинного навчання і машинного зору	35
1.4.2 Особливості тестування коду для вирішення задач машинного навчання і машинного зору	37
1.5 Промпт-інжиніринг та залежність від контексту в мультиагентних системах на базі LLM	39
1.5.1 Базові техніки проектування промптів	39
1.5.2 Рольові директиви та системні промпти в агентних системах.....	40
1.6 Управління контекстом у мультиагентних системах	41
1.6.1 Проблема контекстного вікна та стратегії його управління	41
1.6.2 Retrieval-Augmented Generation у мультиагентних системах	42
1.7 Координація агентів та механізми передачі контексту.....	42
1.7.1 Архітектурні патерни координації в MAC	42

1.7.2 Система AutoGen та LangChain як фреймворки координації.....	43
1.8 Промпт-інжиніринг для спеціалізованих задач МАС	44
1.8.1 Генерація та тестування коду.....	44
1.8.2 Застосування LLM для наукових та аналітичних задач	45
1.9 Порівняльний аналіз підходів до промпт-інжинірингу в МАС.....	45
1.9.1 Ефективність технік промпт-інжинірингу	45
1.9.2 Залежність від контексту та проблема «забування»	46
1.10 Відкриті проблеми та перспективи розвитку	47
1.10.1 Автоматичний промпт-інжиніринг	47
1.10.2 Галюцинації та надійність у МАС	47
1.10.3 Масштабованість і оцінювання	48
Висновки до розділу 1	49
РОЗДІЛ 2 РОЗРОБКА ТА ВЕРИФІКАЦІЯ АРХІТЕКТУРИ	
МУЛЬТИАГЕНТНОЇ СИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ ЗАДАЧ	
МАШИННОГО НАВЧАННЯ	51
2.1 Загальна концепція мультиагентної системи	51
2.2 Опис спеціалізованих агентів системи	53
2.2.1 Агент-Координатор (Supervisor Agent)	53
2.2.2 Агент аналізу даних (Data Analysis Agent)	54
2.2.3 Агент-кодувальник (Data Scientist Agent)	54
2.2.4 Агент-тестувальник (QA Agent)	56
2.2.5 Агент-інтерпретатор (Interpreter Agent)	56
2.2.6 Інфраструктурний агент (Infrastructure Agent)	58
2.3 Функціональні шари системи	58
2.4 Механізм спільного контексту AgentContext	60
2.5 Схема взаємодії агентів	61
2.6 Автоматизована генерація тестових випадків програмного	
забезпечення з використанням промпт-інжинірингу	63
2.6.1 Напрямки використання LLM у тестуванні програмного	
забезпечення	63
2.6.2 Промпт-інжиніринг як ключовий фактор якості генерації тестів.	65
2.6.3 Типи промптів для базових задач програмування	65
2.6.4 Типи промптів для задач аналізу даних та машинного навчання .	67

2.7 Дослідження генерації тестових випадків.....	68
2.7.1 Методика дослідження	68
2.7.2 Результати та аналіз генерації тестів	69
2.7.3 Покриття коду згенерованими тестами	70
2.8 Порівняльний аналіз моделей та стратегій промптингу	72
2.9 Система тестування з самокорекцією	73
2.9.1 Рівні тестування	73
2.9.2 Механізм самокорекції	74
2.10 Тестові задачі та набори даних	75
2.10.1 Breast Cancer Wisconsin (бінарна класифікація)	75
2.10.2 Iris (мультикласова класифікація)	76
2.10.3 California Housing (регресія).....	76
2.11 Генерація Python-коду та тестів	76
2.12 Інтерпретація результатів	80
2.13. Варіанти розгортання системи.....	80
2.14 Результати верифікації системи.....	81
Висновки до розділу 2	82
РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ	
МУЛЬТИАГЕНТНОЇ СИСТЕМИ ГЕНЕРАЦІЇ І САМОКОРЕКЦІЇ КОДУ	84
3.1 Концепція та принципи побудови мультиагентної системи	84
3.2 Структура та функціональне призначення агентів	85
3.2.1 Агент інфраструктури	87
3.2.2 Агент інженерії даних	87
3.2.3 Агент архітектури коду.....	87
3.2.4 Агент контролю якості (QA).....	87
3.2.5 Агент виконання.....	88
3.2.6 Агент аналітики.....	88
3.3 Стратегія трансферного навчання в гібридній архітектурі.....	88
3.3.1 Підтримувані базові архітектури CNN	89
3.3.2 Механізм заморожування шарів та налаштування	89
3.3.3 Дворівнева гібридна архітектура виконання моделей	90
3.4 Ітераційний цикл самокорекції коду	91
3.4.1 Рівні верифікації.....	91

3.4.2 Алгоритм самокорекції.....	91
3.5 Обчислювальна інфраструктура та вибір моделей.....	92
3.5.1 Хмарне середовище (Google Colab).....	93
3.5.2 Локальне середовище з Ollama.....	93
3.5.3 Стратегія вибору моделей.....	93
3.6 Реалізація агентів як класів Python.....	94
Висновки до розділу 3	94
РОЗДІЛ 4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ РІШЕНЬ.....	97
4.1 Опис датасету Superconductivity.....	97
4.2 Результати аналізу з використанням мультиагентної системи	98
4.3 Рішення задач розпізнавання зображень з використанням мультиагентної системи.....	102
Висновки за розділом 4.....	108
ВИСНОВКИ.....	111
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	113
ДОДАТОК А Акти впровадження.....	131

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

- API (*Application Programming Interface*) – інтерфейс прикладного програмування
- AutoML (*Automated Machine Learning*) – автоматизоване машинне навчання
- Baseline – базовий рівень ефективності алгоритму, еталонна модель для порівняння
- CNN (*Convolutional Neural Network*) – згорткова нейронна мережа
- CoT (*Chain-of-Thought*) – ланцюжок міркувань (техніка покрокового промпт-інжинірингу)
- DFD (*Data Flow Diagram*) – діаграма потоків даних
- FLOPs (*Floating Point Operations*) – кількість операцій із плаваючою комою (метрика обчислювальної складності)
- gRPC (*Remote Procedure Calls*) – високопродуктивний фреймворк віддаленого виклику процедур з відкритим вихідним кодом
- GRID-система – географічно розподілена обчислювальна інфраструктура
- IoT (*Internet of Things*) – інтернет речей
- In-Context Learning – контекстне навчання (здатність моделі адаптуватися на основі прикладів у запиті)
- JSON (*JavaScript Object Notation*) – текстовий формат обміну даними, заснований на JavaScript
- LLM (*Large Language Model*) – велика мовна модель
- M MyPy – статичний аналізатор типів для мови програмування Python
- MAS (MAC) (*Multi-Agent System*) – мультиагентна система
- MLLM (*Multimodal Large Language Model*) – мультимодальна велика мовна модель
- Mock-об'єкт – фіктивна реалізація програмного інтерфейсу (імітатор) для ізольованого тестування компонентів ПЗ

- OOM (*Out of Memory*) – критична помилка переповнення доступного обсягу оперативної пам'яті
- Ollama – локальний сервер (програмна інфраструктура) для розгортання та запуску open-source мовних моделей
- ООП – об'єктно-орієнтоване програмування
- Overfitting – ефект перенавчання (втрати моделлю узагальнювальної здатності)
- Prompt – запит (інструкція або директива) до великої мовної моделі
- Prompt Engineering – інженерія запитів (промпт-інжиніринг)
- QA (*Quality Assurance*) – забезпечення якості, автоматизований контроль та тестування програмного продукту
- RAG (*Retrieval-Augmented Generation*) – генерація, доповнена пошуком (архітектура інтеграції LLM із зовнішніми базами знань)
- REST API – архітектурний стиль взаємодії компонентів ПЗ за допомогою HTTP-запитів
- ROC-AUC (*Receiver Operating Characteristic – Area Under Curve*) – площа під кривою помилок (метрика стійкості класифікації моделей)
- SMOTE (*Synthetic Minority Over-sampling Technique*) – алгоритм штучного збалансування класів у наборах даних шляхом генерації міноритарних зразків
- SWE (*Software Engineering*) – інженерія програмного забезпечення (програмна інженерія)
- Stack Trace – трасування стеку викликів (лог виняткових ситуацій інтерпретатора для виявлення дефектів коду)
- State Object – єдиний об'єкт стану спільного контексту в міжагентних архітектурах
- T4 GPU – графічний процесор-прискорювач архітектури Turing для високопараметричних обчислень у середовищі Google Colab
- TFDS (*TensorFlow Datasets*) – бібліотека готових наборів даних для TensorFlow

- Unit Testing – модульне (компонентне) тестування програмного забезпечення
- ІТ (IT) – інформаційні технології
- ПЗ ПЗ – програмне забезпечення
- ШІ (AI) – штучний інтелект

ВСТУП

Актуальність

Сучасний етап розвитку інформаційних технологій характеризується інтенсивним впровадженням методів штучного інтелекту, зокрема машинного навчання та глибоких нейронних мереж, у прикладні системи різного призначення. Зростання складності таких систем, обумовлене використанням багаторівневих моделей, великих обсягів даних та складних обчислювальних конвеєрів, зумовлює необхідність підвищення ефективності процесів їх розробки, тестування та валідації.

Традиційні підходи до розробки та тестування програмного забезпечення виявляються недостатніми для систем машинного навчання і розпізнавання зображень, оскільки вони не враховують імовірнісну природу результатів, наявність статистичних залежностей, вплив якості навчальних даних, а також ризики перенавчання та деградації моделей. Це призводить до зниження надійності програмних продуктів і ускладнює їх практичне впровадження.

Подальший розвиток отримали великі мовні моделі, які демонструють здатність до генерації програмного коду, тестових сценаріїв та технічної документації. Проте їх використання супроводжується рядом обмежень, зокрема залежністю якості результатів від структури вхідного запиту, відсутністю гарантій коректності згенерованого коду, а також фрагментарністю отриманих рішень. Особливо гостро ці проблеми проявляються при розв'язанні задач машинного навчання, де необхідним є забезпечення не лише синтаксичної, але й семантичної коректності програм.

Додатковим фактором складності є необхідність інтеграції різнорідних моделей, зокрема згорткових нейронних мереж і мультимодальних великих мовних моделей, що функціонують на різних принципах представлення знань. Відсутність узгоджених підходів до їх спільного використання призводить до виникнення семантичних розривів, накопичення помилок та

зниження достовірності результатів.

Водночас актуальними залишаються питання забезпечення конфіденційності даних та автономності обчислень, що обмежує використання хмарних сервісів і зумовлює необхідність локального розгортання інтелектуальних систем.

У зв'язку з цим виникає науково-прикладна задача розробки інформаційної технології, яка забезпечує інтеграцію процесів генерації, тестування та валідації програмного забезпечення для задач машинного навчання та розпізнавання зображень на основі мультиагентних систем із використанням великих мовних моделей. Такий підхід дозволяє формалізувати процес розробки, підвищити якість програмних рішень та забезпечити їхню надійність, що визначає актуальність даного дослідження.

Об'єктом дослідження є процес автоматизованого тестування програмного забезпечення для систем машинного навчання та комп'ютерного зору.

Предметом дослідження є методи, моделі та алгоритми автоматизованої генерації тестів і валідації програмного коду з використанням великих мовних моделей та мультиагентних систем у задачах розробки і тестування програмного забезпечення для систем машинного навчання та розпізнавання зображень.

Мета дослідження

Мета роботи полягає у підвищенні ефективності автоматизованого проектування систем машинного навчання шляхом створення шестишарової мультиагентної архітектури, яка забезпечує скорочення термінів розробки та підвищення якості програмних рішень через механізми ітеративної самокорекції та семантичної фільтрації.

Для досягнення поставленої мети вирішено такі наукові завдання:

- аналіз сучасного стану використання великих мовних моделей для автоматизації генерації та тестування програмного забезпечення у задачах машинного навчання;

- розробку мультиагентних моделей для розподіленого створення програмного коду, тестових сценаріїв та процедур верифікації інтелектуальних систем;
- формалізацію багаторівневих стратегій автоматизованого тестування (від перевірки даних до валідації моделей) із застосуванням мовних моделей;
- створення методів контролю якості згенерованого коду для виявлення синтаксичних і семантичних дефектів на ранніх етапах розробки;
- розробку механізмів самокорекції програмного коду через ітераційний зворотний зв'язок між спеціалізованими агентами;
- дослідження впливу параметрів моделей та стратегій формування запитів на стабільність генерації та ефективність верифікації алгоритмів;
- експериментальну оцінку технології за показниками якості коду, повноти тестового покриття та швидкодії розроблених систем.

Реалізація поставленої мети дозволяє сформувати цілісну методологію автоматизованої розробки та тестування програмного забезпечення на основі інтелектуальних агентних систем, що відповідає сучасним вимогам комп'ютерних наук у сфері створення надійних та масштабованих програмних рішень.

Наукова новизна одержаних результатів полягає в тому, що:

- 1) Вперше розроблено архітектурну модель шестишарової мультиагентної системи у вигляді впорядкованої сукупності спеціалізованих функціональних рівнів та єдиного об'єкта стану контексту, в якій за рахунок декомпозиції складних інженерних завдань на множину підзадач (планування, генерація, тестування, інтерпретація тощо) та формалізації механізмів семантичного обміну даними між агентами, забезпечується реалізація наскрізного конвеєра побудови інтелектуальних моделей незалежно від конкретного типу обчислювального ядра, що дозволило мінімізувати

втрату контекстуальної інформації на всіх етапах розробки та формалізувати вхідні й вихідні параметри взаємодії агентів для оптимізації процесів машинного навчання.

- 2) Дістав подальшого розвитку метод багаторівневого динамічного тестування, в якому за рахунок структурної побудови механізму ітеративної самокорекції коду на основі замкненого циклу зворотного зв'язку між агентами-тестувальниками та генераторами, а також композиції параметрів семантичного аналізу помилок і стратегій автоматизованого перепитування (prompt-refinement), реалізовано алгоритм автономного усунення дефектів програмного забезпечення вже на першій ітерації циклу розробки, що дозволило забезпечити стабільність і відтворюваність обчислювальних процесів у недетермінованих середовищах інтелектуальних моделей.
- 3) Вперше запропоновано модель трирівневої системи автоматизованої верифікації, в якій за рахунок структурної побудови ієрархічних рівнів контролю та впровадження механізмів адаптивної корекції результатів через інтелектуальний зворотний зв'язок, а також композиції критеріїв оцінювання якості вхідних даних, верифікації програмного коду та валідації фінальних аналітичних рішень, реалізовано можливість послідовного моніторингу станів системи на кожному етапі обробки інформації, що дозволило довести коефіцієнт успішного виконання сценаріїв до максимально можливих значень навіть за умов низької ймовірності успіху на початкових ітераціях.
- 4) Вперше розроблено метод мультиагентної побудови гібридних систем розпізнавання зображень, в якому за рахунок інтеграції механізмів трансферного навчання згорткових нейронних мереж із семантичною фільтрацією на основі мультимодальних великих мовних моделей, а також композиції візуальних ознак об'єктів та їх когнітивних дескрипторів, реалізовано підхід до нівелювання семантичного розриву між ознаками нижнього рівня та їх

високорівневою інтерпретацією, що дозволило мінімізувати виникнення інтелектуальних галюцинацій і каскадних помилок у процесах класифікації візуальних даних та визначити оптимальну структуру взаємодії агентів для підвищення точності розпізнавання у складних умовах.

Практичне значення одержаних результатів полягає у створенні комплексу моделей, алгоритмів та інженерних рішень, впровадження яких при проектуванні й оптимізації високонавантажених інтелектуальних систем дозволяє реалізувати шестишарову мультиагентну архітектуру автоматизованої розробки ПЗ та гібридну модель розпізнавання зображень. Поєднання можливостей згорткових нейронних мереж із семантичною фільтрацією на базі мультимодальних моделей забезпечує ідентифікацію архітектурних дефектів на ранніх етапах проектування, підвищення швидкості розробки і мінімізацію логічних помилок у моделях машинного навчання та нівелювання семантичного розриву при класифікації візуальних даних, що гарантує стабільність, високу експлуатаційну надійність та конфіденційність функціонування складних програмних комплексів.

Практична цінність отриманих результатів підтверджується такими результатами:

1. Створено комплексний програмний інструментарій на основі шестишарової мультиагентної архітектури, який дозволяє автоматизувати до 80–90% процесів генерації тестових сценаріїв. Це забезпечує повний життєвий цикл розробки інтелектуальних систем і суттєво зменшує обсяг ручного проектування перевірочних структур.

2. Впроваджено методику ітеративної самокорекції коду та спеціалізовані стратегії формування запитів, що дозволяють автономно усувати до 75% технічних дефектів. Практичне застосування цих підходів скорочує час виведення програмних продуктів на ринок на 45–50% та знижує витрати на підтримку тестової інфраструктури на 60–70%.

3. Розроблено та апробовано прикладні методи побудови

гібридних систем розпізнавання образів, які завдяки інтеграції згорткових нейронних мереж із мультимодальними моделями підвищують точність класифікації об'єктів у складних умовах на 12–15%. Технологія підтримує локальне розгортання, що гарантує конфіденційність даних і незалежність від сторонніх хмарних ресурсів.

4. Забезпечено впровадження результатів дослідження у практику діяльності софтверних компаній, що підтвердило ефективність запропонованих рішень для створення надійних і відтворюваних систем машинного навчання. Сформована методологія відповідає сучасним стандартам взаємодії розробки та експлуатації, підвищуючи безпеку та стабільність інтелектуального програмного забезпечення.

Методи дослідження

У роботі використано методи системного аналізу, машинного навчання, обробки природної мови, теорії мультиагентних систем, методи глибокого навчання та трансферного навчання, методи промпт-інжинірингу, а також методи автоматизованого тестування (мутаційне, метаморфне, багаторівневе тестування). Для експериментальної перевірки застосовано методи порівняльного аналізу, статистичної оцінки якості моделей і програмного забезпечення.

Результати дисертаційної роботи використано в навчальному процесі Державного університету інформаційно-комунікаційних технологій (ДУІКТ) при оновленні робочих програм навчальних дисциплін та підготовці методичного забезпечення кафедр комп'ютерних наук та штучного інтелекту.

Окремі положення, обґрунтовані в дисертаційній роботі щодо програмного інструментарію та методики автоматизованої генерації, тестування та валідації програмного забезпечення для задач машинного навчання та машинного зору, що побудовані на базі шестишарової мультиагентної архітектури та великий мовних моделей впроваджено (підтверджено відповідним актом) в ТОВ «АЙТІ КУРСОР» (від

19.02.2026р.).

Результати дисертаційної роботи отримані та використовуються в рамках науково дослідних робіт: «Методика підвищення ефективності систем управління безпроводовими мережами на основі векторного синтезу» (Державний реєстраційний номер ОК 0226U000385) та «Методи побудови функціонально стійких захищених інформаційних систем з централізованим управлінням» (Державний реєстраційний номер РК 0125U002823), Державного університету інформаційно-комунікаційних технологій.

Структура та обсяг дисертаційної роботи.

Дисертація складається із вступу, чотирьох розділів та висновків до них, а також бібліографії, що містить 136 посилань на 9 сторінках. Загальний обсяг роботи становить 133 сторінки.

РОЗДІЛ 1 ЕВОЛЮЦІЯ МЕТОДІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ВІД КЛАСИЧНИХ ПАРАДИГМ ДО МУЛЬТИАГЕНТНИХ СИСТЕМ НА ОСНОВІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

1.1 Парадигми і методи тестування програмного забезпечення

Тестування програмного забезпечення (ПЗ) є невід'ємною частиною процесу розроблення якісних програмних систем. Протягом десятиліть у цій галузі сформувалися усталені парадигми, методи та підходи, які визначають сучасний стан індустрії. Класичні роботи [1], [2] заклали теоретичні засади сучасного тестування, виокремивши структурне (white-box) та функціональне (black-box) тестування як фундаментальні підходи.

Структурне тестування ґрунтується на аналізі внутрішньої будови програми: потоку керування, потоку даних та шляхів виконання. Центральним поняттям є критерії покриття – метрики, що дозволяють оцінити повноту тестового набору відносно структури коду [3], [4]. Серед найпоширеніших критеріїв покриття – покриття операторів (statement coverage), покриття гілок (branch coverage), покриття шляхів (path coverage) та покриття умов (condition coverage). В роботі [5] замість традиційного поділу за рівнями (модульне, системне тощо), автори запропонували інноваційний підхід, заснований на критеріях покриття, що групуються за чотирма основними структурами: графи, логічні вирази, вхідні домени та синтаксичні конструкції. На їх думку, ефективне тестування можливе лише через глибоке розуміння теоретичних основ і застосування автоматизованих методів генерації тестів.

Функціональне тестування зосереджено на перевірці відповідності поведінки системи специфікаціям, без урахування внутрішньої реалізації. До основних технік функціонального тестування відносять: розбиття на класи еквівалентності, аналіз граничних значень, тестування на основі

таблиць рішень і тестування на основі станових машин [5]. Поряд із традиційними методами широкого розповсюдження набуло пошукове тестування (search-based software testing), де генерація тест-кейсів формулюється як задача оптимізації [6].

Важливе місце у сучасних парадигмах тестування займає символічне виконання – техніка, при якій програма виконується на символічних, а не конкретних вхідних даних, що дозволяє систематично досліджувати простір виконання [7], [8]. Подальшим розвитком символічного виконання стало конколічне тестування (concolic testing), яке поєднує конкретне та символічне виконання для більш ефективного дослідження шляхів [9], [10]. Детальний огляд методів автоматизованої генерації тест-кейсів наведено в роботі [10], яка систематизує підходи тестування. З точки зору авторів [10] методи автоматизованого тестування включають: (а) структурне тестування з використанням символічного виконання, (б) тестування на основі моделей, (в) комбінаторне тестування, (г) випадкове тестування та його варіант адаптивного випадкового тестування, та (д) тестування на основі пошуку.

Сучасні тенденції у тестуванні ПЗ характеризуються зростанням ступеня автоматизації, застосуванням методів машинного навчання та інтеграцією з практиками DevOps і безперервної інтеграції/безперервного постачання (CI/CD). Трансформаційним кроком у цьому напрямі стало застосування великих мовних моделей для автоматизованої генерації тест-кейсів, що розглядається у наступних розділах.

1.2 Використання великих мовних моделей для генерації програмного коду

1.2.1 Генерація коду з використанням LLM

Великі мовні моделі (Large Language Models, LLM) здійснили революцію у сфері автоматизованого синтезу програмного коду. В дослідженні OpenAI була запропонована модель Codex [11] – дообучена на

великому корпусі коду з GitHub версія GPT-3. Ця модель продемонструвала здатність вирішувати задачі програмування, описані природною мовою, та ліг в основу системи GitHub Copilot. Авторська оцінка на бенчмарку HumanEval показала правильне вирішення 28.8% задач при single-sample та 72.3% при 100-sample генерації.

Паралельно розвивалися альтернативні підходи. Модель PaLM-Coder від Google [12] також показала конкурентоздатні результати на задачах програмного синтезу. Розробники CodeGen від Salesforce [13] запровадили парадигму багатокрокового синтезу з урахуванням діалогового контексту. В моделі CodeT5+ [14] інтегровано енкодер-декодерну архітектуру з механізмами вирівнювання для кращого розуміння специфікацій.

У роботі [15] автори представили сімейство великих мовних моделей Code Llama, спеціалізованих для генерації та аналізу програмного коду. Моделі були побудовані на основі Llama 2 та додатково навчені на великому корпусі програмного коду різними мовами програмування. Дослідження охопило декілька варіантів моделей, зокрема базову Code Llama, Code Llama-Python та Code Llama-Instruct, оптимізованих відповідно для генерації коду, Python-розробки та виконання інструкцій користувача.

Автори [15] показали, що Code Llama здатна ефективно виконувати задачі автодоповнення коду, генерації функцій, виправлення помилок та пояснення програмних конструкцій. Особливу увагу приділено підтримці довгого контексту (до 100 тис. токенів), що дозволяє моделі працювати з великими програмними проєктами. Експериментальні результати продемонстрували конкурентоспроможність моделі порівняно з закритими комерційними системами генерації коду, що підтверджує перспективність відкритих LLM для автоматизації програмної інженерії та мультиагентних систем розробки програмного забезпечення.

Значним досягненням стала система AlphaCode від DeepMind [16], яка продемонструвала здатність вирішувати задачі рівня змагального програмування, досягнувши результатів у верхніх 54% учасників

Codeforces. Ця робота показала, що LLM здатні не лише генерувати синтаксично коректний код, але й реалізовувати складні алгоритмічні рішення. Для покращення генерації коду застосовуються різноманітні техніки: заповнення (infilling) для InCoder [17], мультилінгвальне навчання для CodeGeeX [18], перехресно-модальне передобчислення для UniXcoder [19] та спеціалізований корпус для SantaCoder [20].

Ключовою проблемою залишається оцінювання якості генерованого коду. Бенчмарк HumanEval [11] оцінює функціональну правильність через запуск тестів, MBPP (Mostly Basic Python Problems) перевіряє вирішення простих задач, а DS-1000 орієнтований на задачі наукових обчислень. Проте ці бенчмарки фокусуються на ізольованих функціях, тоді як реальна розробка ПЗ передбачає роботу з великими кодовими базами та складними залежностями.

1.2.2 Генерація тестів за допомогою LLM

Застосування LLM для автоматизованої генерації тест-кейсів є одним із найбільш перспективних напрямків сучасної програмної інженерії. В роботі [21] виконано систематичну емпіричну оцінку LLM для генерації юніт-тестів, порівнявши ChatGPT, Codex та інші моделі. Результати показали, що LLM здатні генерувати функціонально коректні тести, але стикаються з проблемами підтримки oracle та досягнення повного покриття коду.

В роботі [22] запропонували систематичний підхід до оцінювання та вдосконалення ChatGPT для генерації юніт-тестів, виявивши ключові обмеження: складність з ізольованим тестуванням методів, що мають зовнішні залежності, та схильність до генерації некомпілизованого коду. Розробники CoverUp [23] запровадили підхід, де LLM генерує тести з орієнтацією на максимізацію покриття, використовуючи зворотний зв'язок від інструментів вимірювання покриття.

В [24] представлено фреймворк ChatUniTest, що поєднує LLM із аналізом залежностей для генерації тестів у Java-проектах. В роботі [25]

провели детальне емпіричне дослідження генерації JUnit-тестів за допомогою LLM, оцінивши компілованість, виконання та покриття. Автори роботи [26] розглянули LLM як few-shot тестерів для відтворення помилок, демонструючи, що GPT-4 здатний генерувати тести, що відтворюють баги, на основі опису проблеми.

Окремим напрямком є застосування LLM для фазінгу. Автори [27] показали, що LLM можуть бути ефективними нульовими фазерами для бібліотек глибокого навчання. Для автоматизованого виправлення програм автори [28] продемонстрували ефективність LLM у задачах автоматизованого відновлення коду. У роботі [29] запропоновано SymPrompt – інноваційну стратегію формування запитів для великих мовних моделей, яка вирішує проблему низького покриття коду, властиву традиційним методам тестування на основі пошуку (SBST) та стандартним підходам на базі LLM. На відміну від використання фіксованих інструкцій, SymPrompt деконструє процес генерації тестів на багатоетапну послідовність кроків, що спонукає модель до логічного виведення шляхів виконання та врахування контексту залежностей і типів даних. Реалізований за допомогою фреймворку TreeSitter, цей підхід дозволяє суттєво підвищити якість тестування без додаткового донавчання моделей: експерименти на Python-проектах продемонстрували п'ятикратне зростання правильності генерації та збільшення відносного покриття на 26% для CodeGen2, а у випадку з GPT-4 показники покриття зросли більш ніж удвічі порівняно з базовими стратегіями.

1.2.3 Мультимодальні великі мовні моделі, їх використання для розпізнавання зображень

Мультимодальні LLM (MLLM) розширюють можливості текстових моделей, додаючи здатність обробляти зображення, відео та інші модальності.

Дослідження [30] присвячене подоланню обмежень автоматичної

генерації тестів шляхом використання моделей великих мов (LLM) для перетворення звітів про помилки у реалістичні сценарії виконання. Автори [30] обґрунтовали, що традиційні підходи часто створюють занадто спрощені тестові випадки, тоді як використання ChatGPT та доналаштованої моделі CodeGPT дозволяє генерувати складні, виконувані тести на основі описів багів природною мовою. Експерименти на базі Defects4J продемонстрували, що до 50% звітів про помилки можуть успішно спонукати модель до створення валідних тестів, які є одразу придатними для локалізації дефектів та автоматичного виправлення програмного забезпечення.

В роботі [31] представлено архітектуру CLIP (Contrastive Language-Image Pre-training), яка докорінно змінила підхід до комп'ютерного зору, поєднавши візуальні дані з природною мовою. Замість навчання на фіксованому наборі заздалегідь визначених категорій (як у ImageNet), автори використали масив із 400 мільйонів пар «зображення-текст» із мережі Інтернет. Завдяки контрастивному навчанню модель вчиться зіставляти візуальні образи з їхніми текстовими описами, що дозволяє їй розуміти широке коло концептів без необхідності спеціального донавчання під конкретні завдання. Основний висновок цієї роботи полягає в надзвичайній здатності CLIP до навчання з «нульовим прикладом» (zero-shot learning). Модель демонструє високу ефективність у класифікації зображень та розпізнаванні об'єктів на нових наборах даних, просто отримуючи назву категорії у вигляді тексту. Дослідження довело, що використання природної мови як джерела нагляду (supervision) дозволяє створювати набагато гнучкіші та стійкіші візуальні системи, які краще адаптуються до реальних умов порівняно з традиційними моделями, навченими на вузьких класифікаторах.

Розробники сімейства моделей візуальної мови для дослідження машинного навчання Flamingo [32] від DeepMind запровадили концепцію few-shot навчання для мультимодальних задач, поєднуючи замороженого

LLM із візуальним енкодером. В роботі [33] запропонована BLIP-2, універсальна та ефективна стратегія попереднього навчання, яка поєднує попереднє навчання візуальної мови з готових заморожених попередньо навчених кодерів зображень та заморожених моделей великих мов. Розробники GPT-4V [34] від OpenAI вивели мультимодальні можливості на новий рівень, демонструючи здатність до складного візуального міркування, аналізу документів та розуміння графіків.

У роботі [35] описано модель LLaVA – наскрізно навчену мультимодальну систему, інтегровану з візуальним енкодером. Ключовим внеском авторів є підхід до візуального інструктування, який базується на використанні синтетичних даних для адаптації відкритих LLM до розпізнавання та опису візуального контенту загального призначення. Розробник серії великомасштабних мовних моделей Qwen-VL [36] та mPLUG-Owl [37], нової парадигми навчання, яка надає LLM мультимодальним здібностям завдяки модульному навчанню базового LLM, представили альтернативні архітектури з покращеним розумінням дрібних візуальних деталей. Фундаментальною архітектурою для більшості сучасних MLLM слугує Vision Transformer (ViT) [38], який застосовує механізм уваги до зображення, розбитого на патчі.

З точки зору генерації програмного коду MLLM відкривають якісно нові можливості: генерацію коду з діаграм UML, дизайн-мокапів інтерфейсів або скріншотів помилок. Це особливо актуально для задач комп'ютерного зору, де специфікація задачі часто надається у вигляді прикладів зображень. Зворотно, мультимодальні моделі можуть використовуватися для автоматичного тестування UI-компонентів через аналіз скріншотів.

1.3 Мультиагентні системи з використанням великих мовних моделей

Мультиагентні системи (MAC) на базі LLM представляють новий клас

архітектур, в яких кілька спеціалізованих LLM-агентів взаємодіють між собою для вирішення складних задач. AutoGen [39] від Microsoft Research є одним із перших і найвпливовіших фреймворків, що спростив побудову MAC через абстракцію розмовних агентів із налаштованими сценаріями взаємодії.

Розробники MetaGPT [40], інноваційного фреймворку метапрограмування, запропонували концепцію метапрограмування для MAC, де агентам призначаються ролі програмних інженерів (product manager, architect, engineer, QA), що взаємодіють за формалізованими протоколами. Результати демонструють значне покращення якості генерованого коду порівняно з одноагентними підходами. В роботі [41] представлено концепцію «генеративних агентів» – архітектуру для симуляції переконливої людської поведінки, де LLM-агенти зберігають пам'ять, рефлексують та планують.

Широкі огляди досягнень у сфері LLM-агентів наведені в роботах [42],[43],[44]. Зокрема, огляд [42] систематизує підходи до побудови агентів за трьома виміром: профіль агента, пам'ять та дії. В роботі [43] представлено загальна структура для агентів на основі LLM, що складається з трьох основних компонентів: мозку, сприйняття та дії, і цю структуру можна адаптувати для різних застосувань. Автори [43] дослідили широке застосування агентів на основі LLM у трьох аспектах: одноагентні сценарії, багатоагентні сценарії та співпраця людини та агента.

У роботі [44] автори здійснили комплексний огляд мультиагентних систем на основі великих мовних моделей. Автори [44] класифікували сучасні LLM-мультиагентні системи за типами взаємодії, рівнями автономності та сценаріями використання, включаючи генерацію коду, програмну інженерію, робототехніку, аналіз даних і прийняття рішень. Особливу увагу приділено механізмам пам'яті, міжагентної комунікації та інтеграції зовнішніх інструментів. В [44] зазначено, що мультиагентні системи на базі LLM є перспективним напрямом розвитку штучного

інтелекту, оскільки дозволяють розподіляти складні задачі між спеціалізованими агентами та реалізовувати ітераційні механізми самокорекції. Водночас дослідження підкреслює низку відкритих проблем, серед яких: нестабільність кооперації між агентами, накопичення логічних помилок, висока обчислювальна вартість та складність забезпечення надійності системи. Тому існує необхідність розвитку механізмів автоматизованого тестування, QA-верифікації та адаптивного управління агентами, що є критично важливим для створення автономних систем генерації коду та гібридних AI-архітектур.

В роботі [45] при створенні HuggingGPT, агент на базі LLM, запропонували концепцію, де ChatGPT виступає оркестратором, що делегує задачі спеціалізованим моделям Hugging Face.

У роботі [46] автори запропонували підхід багатоагентної дискусії (Multi-Agent Debate) для покращення якості міркування великих мовних моделей. Метод базується на взаємодії кількох незалежних агентів LLM, які генерують альтернативні варіанти рішень, обговорюють їх та ітеративно уточнюють відповіді. Такий механізм дозволяє моделі уникати локальних помилок, підвищує логічну узгодженість та покращує результати на задачах математичного міркування, генерації коду та логічного аналізу. Експериментальні результати [46] показали, що колективна взаємодія агентів перевершує стандартні одноагентні LLM-підходи.

В роботі [47] запропоновано фреймворк AgentVerse для динамічного формування команди агентів залежно від задачі. Автори [48] реалізували ChatDev, повну програмну послідовність розробки у мультиагентній системі, де різні агенти беруть на себе ролі CEO, СТО, програміста та тестувальника. В роботі [49] досліджено принципи ефективної співпраці в LLM-системах. Автори [50] показали, що ітеративні дебати між агентами дозволяють підвищити точність генерації коду та розв'язання задач міркування.

1.4 Особливості задачі машинного навчання і обробки зображень з точки зору генерації програмного коду

1.4.1 Генерація коду для систем машинного навчання і машинного зору

Генерація коду для задач машинного навчання (МН) та комп'ютерного зору (КЗ) має специфічні особливості порівняно із загальним програмуванням. Код для задач машинного навчання характеризується складною залежністю від фреймворків (PyTorch, TensorFlow, JAX), великою кількістю гіперпараметрів, нелінійним впливом конфігурацій на результат та необхідністю розуміння математичних концепцій [51], [27].

У роботі [51] проведено систематичний огляд методів генерації програмного коду на основі ML/LLM. Автори [51] підкреслили, що код для ML-систем є значно складнішим за звичайне програмування через залежність від спеціалізованих фреймворків, складних API, конфігурацій моделей та параметрів навчання. Окремо було наголошено на важливості коректного налаштування гіперпараметрів і врахування математичних обмежень моделей.

У дослідженні [27] автори аналізують генерацію коду для бібліотек глибокого навчання TensorFlow і PyTorch та демонструють, що такі системи мають складні обмеження щодо типів тензорів, розмірностей, API та конфігурацій обчислень. Робота [27] показала, що навіть незначні зміни параметрів або структури коду можуть призводити до помилок виконання, що підтверджує нелінійний характер поведінки ML-коду та складність його автоматичної генерації.

Специфічні помилки в коді для задач машинного навчання відрізняються від помилок у загальному ПЗ: некоректна форма тензорів, ненавмисна транспозиція осей, витоки даних між навчальною і тестовою вибірками, некоректна нормалізація – усі ці помилки не призводять до виключень, але суттєво погіршують якість моделей [52], [53].

Автори [54] запропонували систему DeepDiagnosis для автоматичної діагностики та виправлення помилок у DL-програмах. В роботі [55] розробили метод автоматичної локалізації помилок у програмах глибокого навчання без виконання. В роботі [56] досліджували проблему виявлення троянських атак у нейронних мережах. Особливо складними є задачі генерації коду для специфічних архітектур КЗ – згорткових мереж, трансформерів для зображень, детекторів об'єктів (YOLO, SSD, Faster R-CNN), де правильність реалізації критично залежить від дотримання архітектурних деталей.

У роботі [57] автори досліджують проблему тестування компіляторів глибокого навчання, зокрема систем оптимізації високорівневих проміжних представлень (IR) у фреймворках машинного навчання. У статті запропоновано метод HirGen – автоматизовану систему fuzz-тестування, яка генерує різноманітні обчислювальні графи та виявляє помилки у DL-компіляторах. Результати експериментів продемонстрували високу ефективність запропонованого підходу: система HirGen дозволила виявити значну кількість критичних помилок у компонентах компіляції моделей глибокого навчання, більшість з яких були підтверджені розробниками та надалі усунуті. На думку [57] системи машинного навчання та компілятори deep learning мають складну структуру та високий рівень вразливості до логічних і оптимізаційних помилок, тому є необхідність використання автоматизованого тестування, fuzzing-підходів та багаторівневої верифікації для забезпечення надійності сучасних ML-компіляторів і програмних систем штучного інтелекту.

У роботі [58] досліджено проблему тестування бібліотек глибокого навчання, які є основою сучасних систем штучного інтелекту, і запропоновано підхід LEMON, що базується на автоматичній генерації моделей для виявлення помилок у deep learning-фреймворках. Метод використовує мутації архітектур нейронних мереж та differential testing для пошуку невідповідностей між різними бібліотеками глибокого навчання.

Основною метою є автоматичне виявлення прихованих дефектів, які можуть виникати через складні залежності між шарами мереж, параметрами навчання та обчислювальними графами.

На думку [58] запропонований підхід ефективно виявляє критичні помилки у компонентах deep learning-систем, які складно знайти традиційними методами тестування. Бібліотеки машинного навчання мають високий рівень складності через нелінійний характер обчислень, велику кількість конфігурацій та залежність від апаратного середовища, тому важливими є автоматизоване тестування, fuzzing-підходи та інтелектуальні механізми верифікації для забезпечення надійності сучасних систем машинного навчання.

1.4.2 Особливості тестування коду для вирішення задач машинного навчання і машинного зору

Тестування коду для МН та КЗ є принципово складнішим завданням порівняно із тестуванням звичайного ПЗ. Фундаментальна проблема полягає у відсутності детермінізму: один і той же код може давати різні результати залежно від стану генератора псевдовипадкових чисел, порядку перемішування даних чи пристрою виконання (CPU/GPU) [59]. У роботі [59] автори виконали масштабний огляд методів тестування систем машинного навчання та проаналізували основні проблеми забезпечення надійності ML-моделей. В дослідженні розглянуто різні етапи життєвого циклу систем машинного навчання, включаючи тестування даних, архітектур нейронних мереж, процесів навчання та експлуатації моделей. Автори [59] класифікували сучасні підходи до тестування ML-систем за типами дефектів, методами генерації тестових даних, техніками верифікації та рівнями автоматизації. Особливу увагу приділено складності тестування deep learning-систем через стохастичний характер навчання, залежність від даних та непрозорість внутрішніх представлень нейронних мереж.

На думку [59] традиційні методи тестування програмного забезпечення

недостатні для систем машинного навчання, оскільки ML-моделі мають нелінійну поведінку, чутливість до гіперпараметрів та високу залежність від якості даних. Перспективними напрямками є автоматизоване тестування нейронних мереж, генерація adversarial-прикладів, верифікація безпеки моделей та використання штучного інтелекту для автоматичного аналізу помилок. Автори [59] наголошують на необхідності створення нових методологій QA та інтеграції тестування безпосередньо у процес розробки ML-систем.

Розробники DeepXplore [60] запропонували першу білоскринькову техніку тестування DL-систем, що використовує нейронне покриття як критерій адекватності. В роботі [61] при створенні інструмента систематичного тестування DeepTest було застосовано метаморфічне тестування для верифікації систем автономного керування, де системні трансформації зображень (зміна яскравості, повороти) дозволяють виявляти помилки за відсутності еталонних значень (істинних даних). Автори [62] адаптували мутаційне тестування для нейронних мереж при створенні фреймворку DeepMutation, визначаючи оператори мутацій на рівні даних та моделі.

Систему фазінгу нейронних мереж із керованістю покриттям TensorFuzz розроблено в [63]. Розширення цього підходу використано при створенні комплексної системи покриття для DL-програм DeepHunter [64]. Автори [65] поставили під сумнів значущість критеріїв нейронного покриття як метрик якості тестів. Диференційний фазінг було запропоновано для виявлення некоректної поведінки DL-систем в роботі [66] при створенні фреймворку DLFuzz.

Автори [67] розробили таксономію реальних помилок у DL-системах на основі аналізу 786 питань з джерела Stack Overflow та 527 - помилок з ресурсів GitHub, виокремивши 5 категорій помилок. Автори [68] провели емпіричне дослідження помилок у фреймворках МН, показавши, що більшість помилок зосереджена у компонентах матричних обчислень.

1.5 Промпт-інжиніринг та залежність від контексту в мультиагентних системах на базі LLM

1.5.1 Базові техніки проектування промптів

Промпт-інжиніринг – дисципліна розробки та оптимізації вхідних підказок для LLM з метою досягнення бажаної поведінки без зміни ваг моделі. Демонстрацію здатності GPT-3 до навчання за кількома прикладами виконано в фундаментальній роботі [69], де включення обмеженої кількості зразків безпосередньо у запит суттєво підвищило ефективність виконання різноманітних завдань.

Револьюційний внесок у промпт-інжиніринг здійснила технік ланцюга думок (Chain-of-Thought, CoT) [70], де модель заохочується до проміжних міркувань перед фінальною відповіддю. Самоузгодженість (Self-Consistency) [71] покращує CoT шляхом сэмплування кількох ланцюжків міркувань та вибору найбільш частоті відповіді. Нульовий ланцюг думок (Zero-shot CoT) [72] продемонстрував, що простого додавання фрази «Let's think step by step» достатньо для активації міркувань.

Дерево думок (Tree of Thoughts, ToT) [73] розширює CoT до деревоподібного простору пошуку, де LLM може досліджувати кілька гілок міркувань та повертатися назад. Систематичний огляд методів промптингу наведено у роботах [74] та [75]. Автоматичний інжиніринг промптів [76] перетворює оптимізацію промптів на задачу генерації та відбору кандидатів. В [77] сформульовано 26 принципів ефективного промптингу LLM.

Найбільш всеохоплюючий сучасний огляд технік промптингу наведено у роботі [78]. Ця публікація є фундаментальним та структурованим дослідженням у галузі інженерії промптів, яке об'єднує та систематизує знання про методи взаємодії з великими мовними моделями (LLM). Автори проводять детальний аналіз понад ста існуючих технік формування запитів, розділяючи їх на чіткі категоріальні класи залежно від архітектури побудови, використання контексту та специфіки розв'язуваних завдань. У роботі

детально розглядаються як базові безконтекстні підходи (Zero-shot), так і складні багатомодульні стратегії, включаючи динамічне налаштування контексту (Few-shot in-context learning), декомпозицію логічних кроків (Chain-of-Thought) та агентні архітектури взаємодії із зовнішніми інструментами, що дозволяє суттєво підвищити автономність, точність та стабільність обчислювальних процесів моделей ІІІ.

Окрім класифікації практичних прийомів, значна увага в огляді приділяється створенню уніфікованої термінологічної бази та математичного опису процесів генерації відповідей, що мінімізує існуючий семантичний розрив між інженерними практиками та теоретичним обґрунтуванням штучного інтелекту. Дослідники аналізують мета-аспекти промптингу, зокрема методи автоматичного пошуку та оптимізації запитів (наприклад, через генетичні алгоритми чи градієнтні наближення), а також детально розглядають верифікацію безпеки систем, вразливості до ін'єкцій шкідливого коду (Prompt Injection) та методи захисту даних. Робота пропонує комплексний фреймворк для оцінювання стійкості функціонування мовних моделей, що робить її незамінним методологічним підґрунтям для розробки сучасних автоматизованих конвеєрів генерації програмного коду, мультиагентних архітектур тестування ПЗ та інтелектуальних систем аналізу технічної документації.

1.5.2 Рольові директиви та системні промпти в агентних системах

Рольові директиви (role-playing prompts) дозволяють налаштовувати поведінку LLM шляхом присвоєння конкретної ідентичності або набору компетенцій. В [79] досліджено механізм рольових ігор у LLM, показуючи, що моделі здатні послідовно дотримуватися призначеної ролі, але при цьому не «стають» цією роллю в повному сенсі. Автори [80] виявили, що уособлення персони в контексті покращує продуктивність на задачах, для яких дана персона є релевантною.

Системні промпти є механізмом визначення глобальної поведінки

агента, його обмежень та цілей. У MAC системні промпти координують взаємодію між агентами, визначаючи протоколи передачі інформації та критерії якості вихідних даних [81], [82]. В роботі [82] запроваджено концепцію рольових взаємодій між двома LLM-агентами (AI User та AI Assistant) для вирішення задач через симульований діалог. Помітний вплив призначеної особи на безпеку та токсичність відповідей LLM виявлено в роботі [83].

Автори [84] запропонували каталог патернів промптів для ChatGPT, включаючи патерни особи, рефреймінгу, запитання та відповіді, шаблонів та мета-мовних дій. В роботі [85] представлено інструментарій LangChain, що надає інтерфейс для розробки ланцюжків дій та управління контекстом системних запитів у межах багатокомпонентних агентних систем. В дослідженні [86] описано техніки системного промптингу для Llama2 у режимі чату, демонструючи вплив системного контексту на безпеку та корисність моделі.

1.6 Управління контекстом у мультиагентних системах

1.6.1 Проблема контекстного вікна та стратегії його управління

Контекстне вікно є фундаментальним обмеженням сучасних трансформерних LLM [87]: модель обробляє лише скінченну послідовність токенів, що породжує проблему «забування» при тривалих взаємодіях. Ця проблема особливо гостра у MAC, де агенти обмінюються великими об'ємами інформації впродовж розв'язання задач.

Автори [88] виявили ефект «загубленості в середині» – феномен, за яким LLM краще відтворює інформацію з початку та кінця контексту, але гірше – з середини. Це суттєво впливає на архітектурні рішення в MAC щодо порядку розміщення інформації в промпті. В [89] показано, що LLM легко відволікаються нерелевантним контекстом, що знижує точність відповідей.

Стратегії управління контекстом включають: стиснення контексту

(summarization), ранжування релевантності (relevance ranking), ієрархічну організацію пам'яті та зовнішнє сховище. У роботі [90] представлено архітектуру MemGPT – операційну систему для LLM з багаторівневою структурою пам'яті (контекстною та зовнішньою), де агент автономно координує власні ресурси пам'яті за допомогою системних викликів. Автори [91] розробили підхід GraphRAG для ефективного підсумовування великих текстових корпусів.

1.6.2 Retrieval-Augmented Generation у мультиагентних системах

Retrieval-Augmented Generation (RAG) [92] поєднує параметричну пам'ять LLM із зовнішніми базами знань, дозволяючи динамічно витягувати релевантну інформацію під час генерації відповіді. Це дозволяє подолати обмеження контекстного вікна та зменшити галюцинації за рахунок заземлення у верифікованих фактах.

У роботі [93] запропоновано підхід REALM, що базується на попередньому навчанні мовної моделі з використанням явного механізму пошуку інформації з Вікіпедії. Автори методу FiD [94] обґрунтували доцільність злиття масивів вилучених документів безпосередньо у блоці кодування, що дозволяє суттєво підвищити якість функціонування відкритих запитально-відповідних систем. У контексті MAC RAG дозволяє агентам динамічно звертатися до репозиторіїв коду, документації API та результатів попередніх взаємодій, значно розширюючи ефективний контекст без перевантаження вікна уваги.

1.7 Координація агентів та механізми передачі контексту

1.7.1 Архітектурні патерни координації в MAC

Координація агентів є центральною проблемою в MAC на базі LLM. Автори [95] заклали теоретичні засади координації в традиційних MAC, виокремивши кооперативну та конкурентну парадигми. У контексті LLM-

систем ці парадигми набувають нових форм завдяки здатності агентів спілкуватися природною мовою.

У роботі [96] запропоновано агентну архітектуру ReAct, у якій велика мовна модель чергує процеси логічного виведення (міркування) та виконання дій, що забезпечує можливість взаємодії агентів із зовнішніми інструментами. Розвиток даної концепції було реалізовано в архітектурі Reflexion [97], де базовий підхід доповнено функціями автономного оцінювання результатів та лінгвістичної корекції (вербального підкріплення). Система AutoGPT [98] впроваджує ітеративний цикл агентної автономії, що поєднує етапи розробки планів, їх реалізації та подальшого аналізу результатів у межах архітектури з персистентною (довготривалою) пам'яттю. Автори проекту WebArena [99] запропонували реалістичну платформу для тестування автономних агентів, що дозволяє верифікувати їхню здатність до вирішення складних сценаріїв у мережі Інтернет.

Автори [48] описали архітектуру комунікативних агентів для розробки ПЗ з формалізованими протоколами взаємодії. В [100] досліджено концепцію «розумових бур» у суспільствах агентів на основі природної мови. В [101] запропоновано підхід самоспівпраці через ChatGPT, де один і той же LLM виступає у кількох ролях. Автори [102] представили динамічну мережу LLM-агентів для задачно-орієнтованої співпраці.

1.7.2 Система AutoGen та LangChain як фреймворки координації

Впровадження AutoGen [39] дозволило формалізувати побудову MAC через концепцію інтерактивних агентів із гнучкими ролями (LLM, людина або виконавець). Архітектура підтримує складні комунікаційні патерни та ієрархічне планування задач. Важливою особливістю системи є механізм налаштування агентів за допомогою вхідних інструкцій та інтегрований інструментарій для апробації згенерованого коду в контрольованому програмному оточенні.

У роботі [85] представлено інструментарій LangChain, що надає програмні абстракції для конструювання складних систем на основі мовних моделей: послідовності операцій, автономні агенти, допоміжний інструментарій та структури пам'яті. Завдяки декларативному підходу LCEL реалізується гнучке проектування обчислювальних процесів. Використання LangGraph дозволяє подолати обмеження лінійних структур шляхом впровадження графів із циклами, що є критично важливим для створення складних агентних систем із можливістю корекції дій на основі проміжних результатів.

Використання підходів QLoRA [103] та LoRA [104] дозволяє адаптувати LLM до специфічних агентних ролей шляхом оптимізації обмеженої кількості параметрів. В аналізі сучасних фреймворків простежується вектор розвитку в бік модульності та декларативності: замість використання складних неструктурованих запитів (промптів), розробники надають перевагу архітектурам, що базуються на поєднанні базових логічних примітивів.

1.8 Промпт-інжиніринг для спеціалізованих задач МАС

1.8.1 Генерація та тестування коду

Промпт-інжиніринг для задач генерації коду в МАС має низку специфічних особливостей. Дослідження [29] підтвердило переваги використання контекстно-залежних інструкцій, що містять відомості про поточне тестове покриття, для підвищення якості генерації програмного забезпечення мовними моделями. Автори [30] дослідили автоматичну генерацію юніт-тестів за допомогою LLM, запропонувавши структуровані шаблони промптів.

У системах типу MetaGPT [40] та ChatDev [48] промпти для генерації коду включають явний контекст ролі (розробник, тестувальник), поточний стан завдання, специфікацію інтерфейсу та стандарти кодування. В роботі

[105] продемонстрували ефективність мультиперсонажної самоспівпраці для розв'язання складних задач програмування. Цю роботу було продовжено у SWE-bench [106] – бенчмарку для оцінки здатності LLM вирішувати реальні GitHub-issues.

Застосування RAG у задачах генерації коду дозволяє агентам динамічно звертатися до документації API, прикладів коду та попередньо згенерованих компонентів. Разом із CoT-промптингом це суттєво покращує якість генерованого коду за складних залежностей. Важливою відкритою проблемою є оптимальне розміщення прикладів у промпті з урахуванням ефекту «загубленості в середині» [88].

1.8.2 Застосування LLM для наукових та аналітичних задач

Застосування LLM-агентів для наукових задач, включаючи аналіз даних та генерацію коду для машинного навчання, потребує спеціалізованих стратегій промптингу. Ключовою є здатність агентів міркувати про неоднозначність задачі, формулювати гіпотези та інтерпретувати результати числових обчислень.

Навчання із зворотним зв'язком від людини (RLHF) [107] забезпечило вирівнювання LLM з людськими перевагами, що особливо важливо для наукових задач, де формальна верифікація результатів утруднена. Для задач МН та КЗ важливим є включення у промпт метаданих про датасет, опису метрик якості та інтерпретації попередніх результатів. Різетт та ін. [15] показали ефективність Code Llama на задачах наукового програмування завдяки навчанню на корпусі наукового коду.

1.9 Порівняльний аналіз підходів до промпт-інжинірингу в МАС

1.9.1 Ефективність технік промпт-інжинірингу

Порівняльна оцінка технік промпт-інжинірингу у МАС є складним завданням через відсутність стандартизованих бенчмарків. Проте

дослідження на окремих задачах дозволяють зробити певні висновки. Автори [70] показали, що CoT-промптинг найбільш ефективний для задач міркування у великих моделях (понад 100B параметрів). В [71] продемонстровано, що самоузгодженість забезпечує стабільне покращення на 10-20% порівняно з однократним CoT-сэмплуванням.

Автори [73] показали, що ToT дозволяє вирішувати складні задачі планування, які залишаються недоступними для CoT. В роботі [50] продемонстрували, що мультиагентні дебати покращують точність генерації коду на 4-16% залежно від задачі. Автори [78] виявили, що різні техніки промптингу демонструють специфічну ефективність залежно від типу задачі та розміру моделі. Загалом ефективність технік залежить від архітектури та розміру моделі, і узагальнення висновків між різними LLM потребує обережності.

1.9.2 Залежність від контексту та проблема «забування»

Катастрофічне «забування» в контексті LLM агентних систем проявляється як у традиційному розумінні (дегенерація продуктивності на попередніх задачах після дообчислення), так і в специфічному – нездатність зберегти релевантну інформацію впродовж тривалої розмови. Автори [90] запропонували MemGPT як рішення проблеми, що імітує ієрархію пам'яті операційних систем.

Вплив контексту характеризується неоднозначністю: використання доречної інформації забезпечує приріст результативності завдяки навчанню на малій вибірці [69], тоді як надлишковий або дезінформуючий контекст спричиняє зниження якості відповідей [89]. У МАС ця проблема посилюється через накопичення міжагентних повідомлень, що можуть містити помилки або неактуальну інформацію. Автори [88] виявили, що моделі гірше використовують інформацію з середини контексту, що має прямий вплив на архітектурні рішення в МАС.

1.10 Відкриті проблеми та перспективи розвитку

1.10.1 Автоматичний промпт-інжиніринг

Автоматичний промпт-інжиніринг (Automatic Prompt Engineering, APE) спрямований на усунення необхідності ручного проектування промптів. В [76] запропонували трактувати пошук оптимального промпту як задачу чорноскринькової оптимізації, де LLM генерує кандидатів, а метрика продуктивності слугує цільовою функцією. Це відкриває шлях до самополіпшуючих агентних систем, де промпти еволюціонують залежно від результатів.

Поточні підходи до APE включають: gradient-based оптимізацію дискретних промптів, еволюційні алгоритми для пошуку в просторі промптів та reinforcement learning для адаптивних промптів. Ключовою відкритою проблемою є перенесення: промпт, оптимізований для одної моделі чи задачі, часто не переноситься на інші [78]. У MAC APE повинен враховувати взаємодію між промптами різних агентів, що суттєво ускладнює простір пошуку.

1.10.2 Галюцинації та надійність у MAC

Галюцинації – генерація фактично некоректних або нечинних тверджень – є однією з центральних проблем надійності LLM [108]. У MAC галюцинації можуть поширюватися між агентами, що призводить до систематичних помилок. Автори [109] запропонували SelfCheckGPT – метод виявлення галюцинацій без зовнішніх знань через консистентність кількох сэмплів.

Дослідження [110] обґрунтовало феномен погіршення результативності великих мовних моделей при роботі з довгими вхідними даними, коли релевантна інформація розташована в середині контексту. Автори виявили U-подібну залежність точності відповідей від позиції ключових даних: моделі успішно вилучають інформацію з початку або кінця запиту, проте

часто ігнорують її в центральній частині. Цей ефект зберігається навіть для спеціалізованих архітектур із розширеним контекстним вікном, що вказує на фундаментальні обмеження поточних механізмів уваги та потребує стратегічного підходу до структурування вхідних інструкцій у складних агентних системах. У дослідженні [111] автори провели емпіричний аналіз помилок у коді, згенерованому великими мовними моделями (LLM), зокрема для задач програмування та машинного навчання. Було виявлено десять основних категорій дефектів. Автори [112] встановили, що навіть синтаксично правильний код часто містить приховані функціональні помилки, які особливо критичні для задач машинного навчання через складність взаємодії між моделями, бібліотеками та даними. В роботі підкреслено необхідність автоматизованого тестування, QA-перевірки та механізмів самокорекції в системах генерації коду на основі LLM. Автори провели комплексний аналіз проблем надійності та безпеки LLM для генерації коду. У MAC суттєвим є питання верифікації виходів агентів: мультиагентні дебати [46], [50] та самооцінка [97] є обнадійливими підходами, але ще не досягли надійності, достатньої для критичних застосунків.

1.10.3 Масштабованість і оцінювання

Масштабованість MAC на базі LLM залишається відкритою проблемою у кількох вимірах: обчислювальній вартості (кожен виклик LLM коштує ресурсів), латентності (послідовні виклики збільшують час відповіді) та якості координації при зростанні кількості агентів. Поточні фреймворки, такі як AutoGen [39] та LangChain [85], пропонують механізми асинхронного виконання та кешування, проте систематичного дослідження масштабованості бракує.

Оцінювання MAC є особливо складним через мультимірію виходу та залежність від завдання. SWE-bench [106] надає один із небагатьох реалістичних бенчмарків для оцінки агентів у задачах програмної інженерії.

Автори [22] та [30] запропонували стандартизовані метрики для оцінки генерації тестів. Відсутність загальних бенчмарків для МАС, орієнтованих на задачі МН та КЗ, є суттєвим пробілом, який потребує заповнення у майбутніх дослідженнях.

Висновки до розділу 1

Проведений огляд літератури дозволяє зробити такі висновки.

1. Класичні методи тестування, хоч і залишаються фундаментом якості ПЗ, потребують адаптації до умов сучасної розробки, оскільки традиційні підходи до автоматизації все частіше стикаються з проблемою «крихкості» тестів та високою вартості підтримки покриття.
2. Аналіз літератури підтвердив, що великі мовні моделі (LLM) здатні радикально змінити процеси тестування програмного забезпечення, автоматизуючи генерацію модульних тестів, документації та пошук логічних помилок, проте головною перешкодою залишається проблема «галюцинацій», нестабільність результатів та обмежене вікно контексту.
3. Перехід від одиночних моделей до мультиагентних архітектур дозволяє нівелювати недоліки окремих LLM за рахунок розподілу ролей між спеціалізованими агентами (програміст, тестувальник, критик), що імітує реальні робочі процеси розробки та підвищує точність згенерованого коду.
4. Якість роботи мультиагентних систем безпосередньо залежить від стратегій формування запитів, де найбільшу ефективність демонструють архітектурно-орієнтовані промпти та методологія розробки через тестування (TDD), яка дозволяє формалізувати вимоги через тестові сценарії ще до написання коду.
5. Незважаючи на значний прогрес, відкритими залишаються питання масштабованості систем, обчислювальної вартості та верифікації

виходів агентів у критично важливих застосунках, що зумовлює необхідність створення гібридних архітектур логічного виводу, які поєднують потенціал LLM із детермінованими методами перевірки.

6. Мультимодальні великі мовні моделі завдяки здатності до навчання без попередніх прикладів дозволяють інтегрувати візуальний контекст безпосередньо в процеси розпізнавання об'єктів та автоматизованого тестування інтерфейсів користувача.
7. Ефективність сучасних інтелектуальних систем критично залежить від методів проектування вхідних інструкцій, таких як ланцюжок думок або дерево думок, причому архітектурні рішення в мультиагентних системах мають враховувати ефект зниження уваги до інформації в середині запиту для запобігання ігноруванню моделями важливих даних.
8. Відкритими проблемами залишаються: автоматичний промпт-інжиніринг без ручного втручання, надійне виявлення та усунення галюцинацій у МАС, масштабованість при великій кількості агентів та відсутність стандартизованих бенчмарків для оцінки мультиагентних систем у задачах машинного навчання та комп'ютерного зору.

Означені пробіли у літературі обґрунтовують актуальність дослідження архітектур МАС для автоматизованої генерації та тестування коду систем машинного навчання та комп'ютерного зору, що є предметом даної дисертаційної роботи.

РОЗДІЛ 2 РОЗРОБКА ТА ВЕРИФІКАЦІЯ АРХІТЕКТУРИ МУЛЬТИАГЕНТНОЇ СИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ ЗАДАЧ МАШИННОГО НАВЧАННЯ

2.1 Загальна концепція мультиагентної системи

Архітектурна реалізація розробленої інформаційної технології базується на концепції розподіленого штучного інтелекту, що втілена у вигляді модульної мультиагентної системи (МАС). Спроектований програмний комплекс містить ієрархічну структуру із шести функціонально ізольованих, але інформаційно пов'язаних спеціалізованих агентів, функціонування яких забезпечується декомпозицією загальної обчислювальної задачі та використанням великих мовних моделей (LLM) як когнітивних ядер. Конструктивна специфіка системи полягає у забезпеченні замкненого ітеративного циклу обробки даних, що охоплює послідовні та паралельні процеси системного аналізу первинних структурованих і неструктурованих масивів, автоматизованої генерації синтаксично та логічно коректного програмного коду мовою Python, динамічного тестування та верифікації отриманих скриптів, а також фінальної науково-технічної інтерпретації результатів експериментів. Синхронізація станів між автономними агентами здійснюється на основі єдиного об'єкта контексту (State Object), що мінімізує накопичення інформаційної ентропії та унеможливорює розходження логіки (semantic drift) у процесі міжагентного обміну повідомленнями.

Для забезпечення високого рівня відтворюваності досліджень, адаптивності до наявної обчислювальної інфраструктури та оптимізації витрат апаратних ресурсів, у архітектурі МАС передбачено три інваріантні стратегії розгортання та топології мережевої взаємодії. Перша стратегія передбачає суто локальну конфігурацію, за якої всі шість когнітивних агентів інтегруються з локально розгорнутим сервером Ollama; такий підхід

гарантує абсолютну конфіденційність оброблюваних даних, незалежність від зовнішніх мережеских затримок та повну автономність функціонування системи в ізольованих інженерних середовищах. Друга, хмарна стратегія, орієнтована на використання зовнішніх комерційних інтерфейсів прикладного програмування (API), що дозволяє залучати надвеликі пропрієтарні моделі для вирішення завдань підвищеної логічної складності без задіяння локальних графічних прискорювачів. Третя, гібридна (змішана) стратегія, реалізує комбіноване використання обчислювального потенціалу: рутинні операції аналізу даних та базового тестування покладаються на локальні легковагові моделі, тоді як критичні завдання архітектурного синтезу коду та фінального арбітражу делегуються хмарним високопараметричним моделям, що забезпечує раціональний компроміс між швидкістю виконання обчислень, фінансовими витратами та інформаційною безпекою технологічного процесу. Для третього варіанту була реалізована версія з розгортанням сервера Ollama в хмарному середовищі Google Collaboratory [113].

Загальна архітектура системи наведена на рис. 2.1.

Ключовою архітектурною особливістю є єдиний об'єкт AgentContext, через який здійснюється весь міжагентний обмін даними. Цей підхід реалізує принцип слабкого зв'язування (loose coupling): кожен агент взаємодіє виключно з AgentContext, а не безпосередньо з іншими агентами. Завдяки цьому агенти можуть бути замінені, розширені або виконані у довільному порядку без перебудови системи.

Система була верифікована на трьох стандартних задачах машинного навчання:

- бінарна класифікація (Breast Cancer Wisconsin),
- мультикласова класифікація (Iris) та
- регресійний аналіз (California Housing).

В усіх випадках МАС автономно виконала повний цикл: завантаження даних → попередня обробка → навчання моделі → тестування

→ інтерпретація результатів.

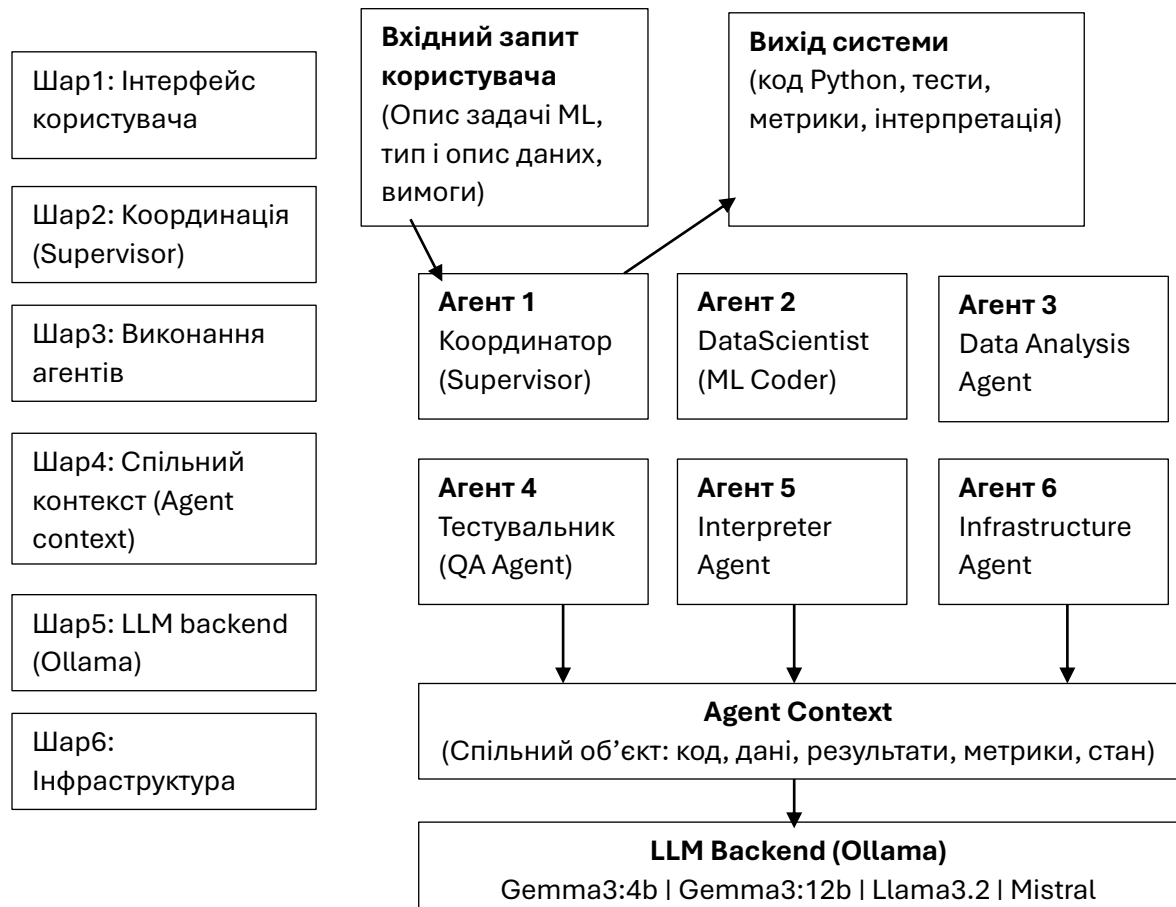


Рис. 2.1 – Загальна архітектура мультиагентної системи на базі LLM

2.2 Опис спеціалізованих агентів системи

Кожен агент системи реалізований у вигляді Python-класу, що наслідує базовий абстрактний клас `BaseAgent`. Клас визначає єдиний обов'язковий метод `run(context: AgentContext) -> AgentContext`, що забезпечує уніфікований інтерфейс виклику. Нижче наведено детальний опис кожного агента.

2.2.1 Агент-Координатор (Supervisor Agent)

`Supervisor Agent` є центральним управляючим компонентом системи. Отримуючи вхідний опис задачі від користувача, він генерує високорівневий план виконання – впорядковану послідовність кроків із зазначенням

відповідального агента для кожного. Координатор також контролює хід виконання, ініціює повторні спроби при виникненні помилок та фіксує загальний стан системи в AgentContext.

Технічна реалізація: промпт системи містить опис усіх доступних агентів і форматований шаблон виведення плану у вигляді JSON-структури. Це дозволяє автоматично парсити план і передавати керування наступному агенту.

2.2.2 Агент аналізу даних (Data Analysis Agent)

Агент відповідає за завантаження датасету, первинну розвідувальну аналітику (EDA) та відбір ознак. Він використовує бібліотеки pandas, numpy та scikit-learn для обчислення статистичних характеристик: розподілу ознак, кореляційних матриць, виявлення пропущених значень і аномалій. Результати зберігаються в AgentContext як структурований звіт та підготовлені масиви даних.

2.2.3 Агент-кодувальник (Data Scientist Agent)

Агент аналізу даних (Data Scientist Agent) займає ієрархічно центральне місце у виконавчому контурі розробленої мультиагентної системи, безпосередньо відповідаючи за алгоритмічну реалізацію та математичний супровід обчислювальних експериментів. Функціональне призначення цього агента полягає в автоматизованому синтезі повнотекстових програмних скриптів мовою Python на основі декомпозиційного плану, сформованого Агентом-Координатором (Supervisor Agent), та з урахуванням семантичного профілю попередньо оброблених вхідних даних.

Процес генерації коду Агентом аналізу даних охоплює чотири послідовні фази інтелектуального аналізу (інваріанти моделювання):

1. Адаптивний вибір предикативної моделі: обґрунтування та вибір оптимального класу алгоритмів машинного навчання (наприклад, ансамблевих методів, градієнтного бустингу або глибоких згорткових

нейромереж) відповідно до топології та розмірності простору ознак.

2. Оптимізація простору гіперпараметрів: декларативне визначення стратегій конфігурування моделей (ініціалізація кроку навчання, параметрів регуляризації, функцій активації тощо).

3. Процедура навчання (Model Fitting): синтез стійкого коду для ітераційного підгону ваг моделі на тренувальних вибірках із урахуванням крос-валідації.

4. Обчислення диференційованих метрик якості: інтеграція в тіло скрипту математичного апарату оцінювання узагальнювальної здатності моделей (коефіцієнтів детермінації, матриць помилок, кривих ROC-AUC, метрик Accuracy, F1-Score тощо).

Важливою архітектурною особливістю Data Scientist Agent є впровадження внутрішнього предикативного механізму самокорекції (self-reflection / self-correction workflow), що функціонує в режимі реального часу. У разі виявлення синтаксичних аномалій, логічних винятків (Exceptions) або деградації цільових метрик на етапі верифікації, Агент автоматизованого тестування (QA Agent) повертає деталізований лог помилок у вигляді зворотного зв'язку.

Data Scientist Agent запускає ітераційний процес рефлексії:

- аналізує трасування стеку помилок (Stack Trace),
- виявляє когнітивні та синтаксичні невідповідності у початковому коді,
- динамічно генерує модифіковану, стабільну версію програмного скрипту без залучення оператора-людини.

Це дозволяє мінімізувати ефект накопичення похибок та забезпечує збіжність міжагентного ітераційного процесу до стану працездатного програмного продукту.

Спрощена реалізація механізму самокорекції наведено в лістинг 2.1: при отриманні зворотного зв'язку від QA Agent він аналізує повідомлення про помилки та генерує виправлений код.

```

class DataScientistAgent(BaseAgent):
    def run(self, ctx: AgentContext) -> AgentContext:
        prompt = self._build_prompt(ctx.data_info,
ctx.plan)

        code = self.llm.generate(prompt)
        ctx.generated_code = code
        exec_result = self._safe_exec(code)
        ctx.execution_result = exec_result
        return ctx

```

Лістинг 2.1 – Спрощена реалізація Data Scientist Agent

2.2.4 Агент-тестувальник (QA Agent)

QA Agent генерує і виконує автоматизовані тести для верифікації коду, згенерованого Data Scientist Agent. Агент проектує два типи тестів: тести цілісності даних (перевірка розмірності, типів, відсутності NaN-значень) та юніт-тести логіки моделі (перевірка передбачень на граничних випадках, моніторинг точності на валідаційній вибірці). Реалізований цикл самокорекції: при виявленні помилок QA Agent передає діагностичний звіт у AgentContext, а Data Scientist Agent повторно генерує код. Максимальна кількість ітерацій корекції – 3.

2.2.5 Агент-інтерпретатор (Interpreter Agent)

Агент наукової інтерпретації результатів (Interpreter Agent) є фінальним аналітичним модулем у когнітивному контурі розробленої мультиагентної системи. Його головне функціональне призначення полягає у трансформації сирих числових логів, розрахованих експериментальних метрик та статистичних даних у високоструктурований, семантично зв'язний науково-технічний звіт, адаптований для верифікації дослідником.

На вхід Interpreter Agent подається диференційований масив вхідних дескрипторів та артефактів машинного навчання, що включає:

1. Комплекс статистичних метрик ефективності: значення коефіцієнтів детермінації, точність (Accuracy), повноту (Recall), збалансовану метрику F1-Score та площі під кривими ROC-AUC.
2. Матрицю помилок (Confusion Matrix): узагальнену двовимірну структуру розподілу істинно позитивних, хибно позитивних, істинно негативних та хибно негативних передбачень для виявлення взаємної системної деградації класів.
3. Вектор відносної важливості ознак (Feature Importance): вагові коефіцієнти внеску кожного вхідного параметра в узагальнювальну здатність моделі, розраховані через аналіз градієнтів або пермутаційні методи.
4. Хронометричну статистику обчислювального процесу: часові інваріанти навчання моделі (Train Time), швидкість інференсу на одному батчі та обсяг утилізованої оперативної чи відеопам'яті (Memory Footprint).

На основі інтегрованого аналізу зазначених компонентів Interpreter Agent застосовує методи логічного виведення та генерує вихідний структурований звіт, який містить такі обов'язкові аналітичні розділи:

- Експертна оцінка адекватності моделі: детальний розбір метрик із математичним обґрунтуванням спроможності моделі вирішувати поставлену прикладну задачу.
- Діагностика системних аномалій та деструктивних ефектів: автоматизоване виявлення ознак перенавчання (Overfitting) через розходження тренувальних та валідаційних кривих, ідентифікація деградації меншості класів в умовах істотного дисбалансу даних (Class Imbalance) та фіксація затухання градієнтів.
- Стратегічний комплекс рекомендацій: формування експертних порад щодо подальшої оптимізації архітектури, які включають пропозиції стосовно розширення методів регуляризації (Dropout, L1/L2-ваги),

застосування алгоритмів синтезу нових даних (SMOTE) або реінжинірингу ознак (Feature Engineering).

- Компаративний аналіз: зіставлення поточних показників ефективності з фіксованими базовими рівнями (Baseline), статистичними медіанами та результатами альтернативних архітектур для математичного підтвердження наукової новизни та практичної цінності поточного експерименту.

2.2.6 Інфраструктурний агент (Infrastructure Agent)

Infrastructure Agent відповідає за підготовку програмного середовища перед запуском MAC. Він автоматично перевіряє наявність та при необхідності встановлює сервер Ollama, завантажує потрібну LLM-модель (Gemma3:4b, Gemma3:12b, Llama3.2 або Mistral), налаштовує бібліотеку LangChain та перевіряє доступність GPU/CPU ресурсів. Цей агент виконується першим і забезпечує умови для коректної роботи решти агентів.

2.3 Функціональні шари системи

Мультиагентна система організована у шість функціональних шарів відповідно до принципу розділення відповідальності (Separation of Concerns).

Кожен шар інкапсулює певний аспект системи, що дозволяє замінювати компоненти (наприклад, LLM-модель або механізм пам'яті) без модифікації суміжних шарів.

Характеристики і функції шарів мультиагентної системи наведено на рис. 2.2 і в таблиці 2.1.

Важливою особливістю такої структури є впровадження механізму зворотного зв'язку та самокорекції (Self-Correction Loop), що реалізується на перетині третього та четвертого шарів. Коли QA-агент виявляє помилки у згенерованому коді або невідповідність результатів метрикам якості, система не просто зупиняє роботу, а ініціює ітераційний процес

виправлення. Через спільну пам'ять (шар 4) контекст помилки передається назад до Data Scientist або Interpreter агента, що дозволяє динамічно адаптувати рішення без втручання користувача. Це перетворює систему з лінійної послідовності кроків на інтелектуальне середовище, здатне до автономного вдосконалення результатів.

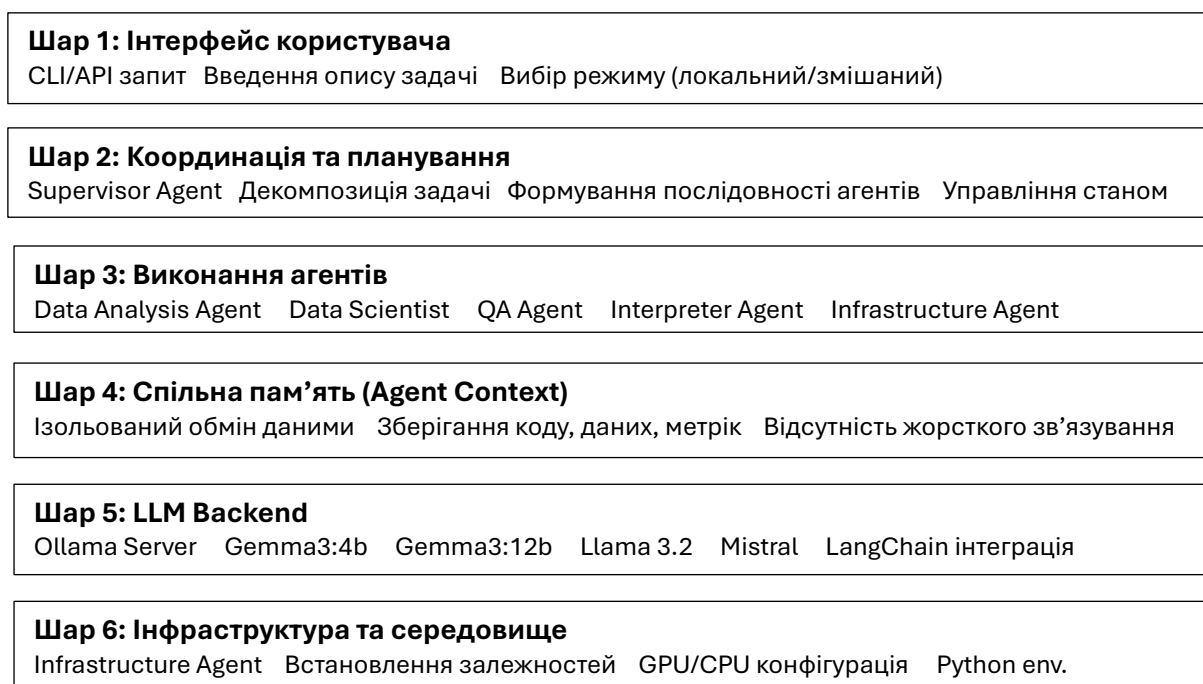


Рис. 2.2 – Функціональні шари мультиагентної системи

Крім того, відокремлення LLM-бекенду (шар 5) від логіки виконання агентів забезпечує високу гнучкість та масштабованість. Використання Ollama Server як фундаменту дозволяє системі працювати в умовах обмежених ресурсів, перемикаючись між легшими моделями для рутинних перевірок (наприклад, Gemma3:4b для інфраструктурних завдань) та потужнішими архітектурами (Gemma3:12b або Llama 3.2) для складного аналізу даних та логічного виведення. Такий підхід мінімізує обчислювальні витрати та дозволяє розгорнути систему як на локальних машинах, так і в гібридних хмарних середовищах, зберігаючи при цьому повний контроль над конфіденційністю даних.

Таблиця 2.1

Функції шарів мультиагентної системи

Шар	Назва	Функції та компоненти
1	Інтерфейс	CLI-інтерфейс, API endpoint, введення опису задачі, вибір LLM-моделі та режиму розгортання
2	Координація	Supervisor Agent, декомпозиція задачі, формування плану, управління потоком виконання
3	Виконання агентів	Data Analysis, Data Scientist, QA, Interpreter, Infrastructure Agent – паралельне або послідовне виконання
4	Спільна пам'ять	AgentContext: сховище стану, код, дані, метрики, логи виконання, результати тестів
5	LLM Backend	Ollama Server, Gemma3:4b/12b, Llama3.2, Mistral; LangChain для prompt-management
6	Інфраструктура	Infrastructure Agent, Python-середовище, налаштування GPU/CPU, моніторинг ресурсів

Завершальний шар інфраструктури (шар 6) гарантує відтворюваність та ізоляцію обчислень. Завдяки автоматизованому управлінню Python-середовищем через Infrastructure Agent, кожен проєкт отримує необхідні залежності у відриві від системних налаштувань. Це критично важливо для задач машинного навчання та стеганоаналізу, де точність версій бібліотек безпосередньо впливає на результат. Таким чином, архітектура не лише розділяє обов'язки між інтелектуальними агентами, а й створює надійний технічний фундамент для виконання складних наукових та інженерних розрахунків.

2.4 Механізм спільного контексту AgentContext

AgentContext є центральним об'єктом обміну даними між агентами. Він реалізує патерн проектування Blackboard (дошка оголошень): усі агенти читають необхідні дані з AgentContext та записують результати своєї роботи назад до нього. Це повністю виключає пряму комунікацію між агентами та

усуває проблему циклічних залежностей. Структура об'єкту AgentContext наведена в лістингу 2.2.

```
@dataclass
class AgentContext:
    task_description: str = ''
    execution_plan: list[dict] =
field(default_factory=list)
    raw_data: pd.DataFrame | None = None
    data_info: dict = field(default_factory=dict)  #
EDA results
    generated_code: str = ''
    execution_result: dict =
field(default_factory=dict)
    test_results: list[dict] =
field(default_factory=list)
    model_metrics: dict = field(default_factory=dict)
    interpretation_report: str = ''
    error_log: list[str] =
field(default_factory=list)
    correction_attempts: int = 0
```

Лістинг 2.2 – Структура об'єкту AgentContext

Використання dataclass з типовими значеннями за замовчуванням гарантує, що AgentContext завжди знаходиться в коректному стані, навіть якщо окремі агенти ще не виконали свою роботу. Це дозволяє агентам бути ініціалізованими у довільному порядку без виникнення помилок доступу до неініціалізованих полів.

2.5 Схема взаємодії агентів

Взаємодія автономних сутностей у межах запропонованої

мультіагентної системи регламентується протоколом послідовно-ітераційного виконання завдань, що функціонує на засадах недетермінованого скінченного автомата. Схему взаємодії наведено на рис. 2.3. Процес обчислень ініціюється Агентом-Координатором (Supervisor Agent), який на основі вхідних дескрипторів здійснює декомпозицію глобальної мети, динамічно формує лінійний план виконання та визначає черговість активації цільових модулів. Передача інформаційних артефактів, проміжних станів обчислень та логів між компонентами системи реалізується через уніфіковану шину обміну даними – об'єкт контексту AgentContext (State Object). Цей об'єкт забезпечує збереження просторової та часової цілісності даних, мінімізує інформаційну ентропію при міжпроцесній взаємодії та запобігає семантичному розходженню (semantic drift) у складних багатокomпонентних конвеєрах машинного навчання.

Критично важливим архітектурним елементом, що визначає інтелектуальні властивості та живучість розробленої інформаційної технології, є впровадження замкненого контуру зворотного зв'язку у вигляді адаптивних петель повторення (retry loops). Зазначений механізм безпосередньо реалізує ітераційний цикл автоматизованої самокорекції (self-correction/self-reflection workflow), локалізований між Агентом автоматизованого тестування (QA Agent) та Агентом аналізу даних (Data Scientist Agent). У разі фіксації синтаксичних помилок інтерпретатора Python, логічних винятків (Exceptions) або деградації цільових метрик ефективності на етапі валідації, QA Agent здійснює парсинг трасування стеку (Stack Trace), маркує деструктивні ділянки коду та повертає структурований лог помилок. Data Scientist Agent запускає внутрішню процедуру рефлексії, здійснює когнітивний аналіз отриманих зауважень та динамічно синтезує модифіковану версію програмного скрипту, повторюючи цей процес до досягнення критерію збіжності, що дозволяє повністю виключити людину-оператора з контуру верифікації проміжних результатів.

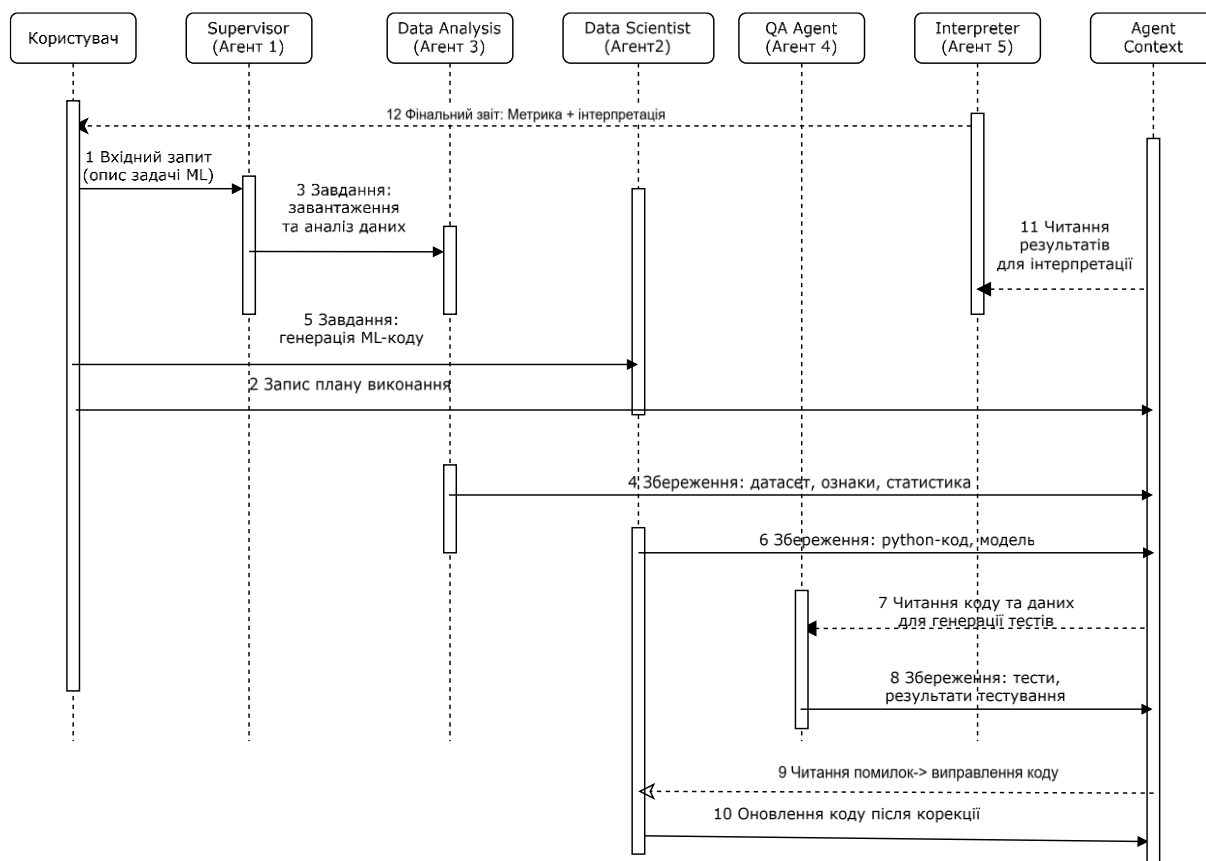


Рис. 2.3 – Схема взаємодії агентів та потоки даних у мультиагентній системі

Стрілки на схемі відображають потоки даних між агентами та AgentContext. Особливу увагу слід звернути на кроки 9–10, що реалізують механізм самокорекції: QA Agent передає діагностику помилок у AgentContext, а Data Scientist Agent зчитує її та генерує виправлений код. Цей цикл може повторюватись до трьох разів.

2.6 Автоматизована генерація тестових випадків програмного забезпечення з використанням промпт-інжинірингу

2.6.1 Напрямки використання LLM у тестуванні програмного забезпечення

Використання LLM у тестуванні програмного забезпечення є одним із найбільш динамічних напрямків досліджень у сфері автоматизації QA з 2022 року. Дослідження фокусуються на здатності LLM розуміти вимоги,

генерувати тестові сценарії та писати виконуваний тестовий код [114]. Виділяють такі основні напрямки:

- Генерація тестів на основі вимог природною мовою (NL-to-Test). Один з найпоширеніших підходів, що використовує LLM для подолання розриву між бізнес-вимогами та технічним тестуванням [115]. LLM можуть автоматично ідентифікувати неясності або неповноту у вимогах перед генерацією тестів, що раніше було виключно ручним завданням [116].
- Генерація поведінково-орієнтованих сценаріїв (BDD). LLM використовуються для генерації сценаріїв у форматі Gherkin (Given-When-Then), що прискорює впровадження методології Behavior-Driven Development [117].
- Генерація юніт-тестів (Code-to-Test). Модель отримує функцію або клас і генерує модульні тести для перевірки її логіки, аналізуючи синтаксис, семантику та потенційні граничні умови. Основна метрика оцінки – покриття коду [118].
- Мутаційне тестування та тестування API. LLM отримують специфікації API (OpenAPI/Swagger) і генерують повні набори тестів для RESTful API. Показано, що LLM можуть ефективно генерувати тести для виявлення мутацій, значно підвищуючи якість тестового набору [119].
- Регресійне тестування. Використання LLM для оновлення та генерації нових тестів після змін у кодовій базі [120]. LLM краще генерують регресійні тести, коли в промпті вказано фокус на змінених ділянках коду.

Важливим фактором успіху генерації тестів є структура промпта. Дослідження [121] показали, що промпти, які включають вимоги до стилю (PEP 8), алгоритмічні обмеження та безпекові перевірки, генерують більш якісний і менш вразливий код. Вимога до LLM генерувати вивід у структурованому форматі (pytest-код, BDD-сценарії) забезпечує виконуваність артефактів [114].

2.6.2 Промпт-інжиніринг як ключовий фактор якості генерації тестів

Оскільки всі публічні LLM навчаються на загальних даних без специфіки предметної галузі, промпт є єдиним каналом специфікації вимог до коду та тестів. LLM є досить чутливими до контексту, наданого користувачем, тому структура промпта має прямий вплив на точність та безпеку згенерованого коду.

Для генерації тестів промпт-інжиніринг є особливо критичним: він повинен керувати LLM не лише стилем, але й логікою покриття. Зокрема, надання LLM ролі «Досвідченого QA-інженера» або «Тест-архітектора» активує у моделі знання про найкращі практики тестування – тестування граничних умов (Boundary Testing) та негативне тестування [122], [123]. За даними [123], завдання відповідної ролі у промпті підвищило точність з 46,2% до 64,0%.

При використанні для генерації тестів LLM мають схильність до генерації позитивних тестів, якщо не вказати явно інше. Для досягнення високого покриття гілок коду необхідно прямо вимагати перевірки винятків та неприпустимого вводу [124].

2.6.3 Типи промптів для базових задач програмування

З метою проведення комплексного порівняльного аналізу та оцінювання узагальнювальної здатності сучасних великих мовних моделей (LLM) у задачах автоматизованого синтезу тестових сценаріїв було сформовано диференційований базований набір алгоритмічних завдань.

Процес дослідження охоплював фундаментальні інваріанти парадигми процедурного та об'єктно-орієнтованого програмування, що класифіковані за рівнем логічної складності та архітектурою обробки даних: умовні оператори (розгалуження обчислювальних процесів), ітераційні цикли (лінійні та вкладені), одновимірні й багатовимірні масиви (структуровані

типи даних), символні рядки (текстова семантика) та рекурсивні алгоритми (складні форми керування стеком пам'яті). Такий підхід дозволив забезпечити репрезентативність вибірки та верифікувати стійкість генерації коду моделей у різних когнітивних доменах.

Деталізовану архітектуру та функціональне навантаження розроблених шаблонів запитів, що спрямовані на нівелювання семантичного розриву між специфікацією завдання та кінцевим виконуваним кодом тесту, систематизовано й наведено в таблиці 2.2.

Таблиця 2.2

Типи промптів для задач базового програмування

№	Тип промпту	Орієнтація	Характеристика згенерованого коду
1	Мінімальний (Базовий)	Студент / початківець	Простий код, все в одному файлі, базові тести або їх відсутність
2	QA Engineer	Тестувальник	Суворіші тести, фокус на перевірках граничних умов, структурований набір сценаріїв
3	Test Architect	Досвідчений QA/Test Lead	Модульна архітектура, розбиття на компоненти, повне покриття, pytest-набори

Експериментальна методологія дослідження базується на послідовному застосуванні трьох стратегічно відмінних типів інженерії запитів (промпт-інжинірингу), які різняться за рівнем інформаційної насиченості, ступенем структурованості та наявністю механізмів контекстного навчання (*In-Context Learning*). Диференціація розроблених шаблонів інструкцій дозволяє системно оцінити когнітивне навантаження на кожне з аналізованих ядер великих мовних моделей, виявити межі їхньої генеративної здатності та встановити ступінь семантичної відповідності між вихідною технічною специфікацією і фінальним кодом. Завдяки такому підходу з'являється можливість емпіричним шляхом верифікувати вплив архітектурної побудови промпту на загальну стабільність роботи

інтерпретатора та коректність виконання синтаксичних конструкцій у згенерованих тестових випадках.

У межах розробленого дослідницького конвеєра кожен тип запиту виконує специфічну методологічну роль, поступово ускладнюючи контекстне оточення моделі від базових безконтекстних інструкцій (*Zero-Shot*) до розгорнутих шаблонів із демонстрацією еталонних прикладів (*Few-Shot*) та ланцюжків покрокових міркувань (*Chain-of-Thought*). Це дозволяє не лише зафіксувати сухі кількісні показники ефективності, а й детально проаналізувати динаміку зміни коефіцієнта покриття коду (*Code Coverage*) залежно від повноти вхідних інженерних метрик. Отримані в результаті реалізації цієї методології дані слугують основою для побудови аналітичних залежностей і формування оптимізованого фреймворку автоматизованого проектування тестів, мінімізуючи ризик виникнення логічних помилок або деградації моделей у складних алгоритмічних доменах.

Застосування зазначених інваріантів проектування промптів дозволило формалізувати вплив контекстного оточення на коефіцієнт покриття коду (*Code Coverage*) та стабільність роботи інтерпретатора при валідації синтезованих тестових випадків..

2.6.4 Типи промптів для задач аналізу даних та машинного навчання

Для задач завантаження і аналізу даних (побудова регресійних моделей, попередня обробка даних, агрегація) використано розширений набір промптів [125]:

- Мінімальний: без умов щодо точки зору, пояснень і тестів; мінімальний контекст.
- Науковий (*Scientific*): передбачає надання точки зору та вимоги математичних пояснень, оцінки похибки, порівняння методів.
- TDD (*Test-Driven Development*): LLM спочатку генерує тести, потім реалізацію коду рішення; найсуворіший метод з точки зору покриття.

- Data-analysis-oriented: наявність точки зору та деталізований перелік окремих підзадач для реалізації у згенерованому коді.
- Архітектурний: вимагає відповідну структуру проєкту, модулів, логування, дотримання PEP-8.
- Орієнтований на швидкодію: NumPy, Numba, векторизація, алгоритмічна оптимізація.

2.7 Дослідження генерації тестових випадків

2.7.1 Методика дослідження

Генерація тестових випадків виконувалась у двох варіантах: (1) власноруч написаний та перевірений код рішень подавався на вхід LLM з вимогою згенерувати відповідні тестові випадки; (2) завдання формулювалось природною мовою, за допомогою LLM виконувалась генерація коду рішення та тестового коду. Згенеровані тести виконувались у середовищі Python для перевірки їхньої функціональності, точності та покриття коду.

Для дослідження використано дві групи моделей. Комерційні моделі: ChatGPT (GPT-5.1) та Google Gemini (Gemini 2.5 Flash), доступні через веб-інтерфейси. Відкриті моделі, розгорнуті через Ollama: Gemma2:7b, Gemma3:1b, Gemma3:4b, Gemma3:12b, Mistral:7b, Llama2:7b, Llama3.1:8b. Завантаження моделей, робота з ChatGPT та Gemini виконувалась на ноутбучі з процесором AMD Ryzen 7 5700U, 16 ГБ оперативної пам'яті, GPU AMD Radeon, ОС Windows 11.

Для базових задач програмування відібрано задачі, що охоплюють умовні оператори, цикли, масиви, рядки та рекурсію (зокрема, обчислення факторіала, сортування, рядкові операції). Набір для аналізу даних включав: завантаження CSV-файлу з даними про житлові об'єкти, попередню обробку (NaN, one-hot кодування, масштабування), побудову регресійних моделей (LinearRegression, RandomForestRegressor), обчислення метрик (RMSE, R^2)

та генерацію pytest-тестів, що перевіряють коректність обробки NaN, правильність розмірностей тензорів, здатність моделі навчатись і генерувати предикти очікуваного розміру, а також виконання умови $RMSE <$ допустимого порогового значення.

Для кожної задачі вимагалось: згенерувати код рішення, згенерувати pytest-тести, пояснити принцип роботи. Для оцінки якості згенерованого коду використано суб'єктивно-об'єктивну сумарну метрику (від 0 до 10 для найкращого коду), що враховує: читабельність, використання best practices (pipeline, контроль випадковості), обробку граничних випадків, docstrings та документацію, безпечне використання бібліотек.

2.7.2 Результати та аналіз генерації тестів

Результати генерації тестів обчислення факторіала за допомогою різних LLM (незалежно від типу промпта) наведено в таблиці 2.3. Для відносно простих задач сучасні LLM генерують тести зі 100% покриттям незалежно від промпту, тоді як моделі попереднього покоління не забезпечують повного покриття навіть при найсуворіших промптах.

Таблиця 2.3

Тести для задачі обчислення факторіала, згенеровані різними класами LLM

Тест	Gemma:2b, Llama2	Gemma3:1b–12b, Llama3.1:8b	Gemini, GPT-5.1
Факторіал позитивного числа ($N > 0$)	Згенеровано	Згенеровано	Згенеровано
Факторіал від'ємного числа ($N < 0$)	Відсутнє	Згенеровано	Згенеровано
Факторіал нуля ($N = 0$)	Згенеровано	Згенеровано	Згенеровано

Аналогічні результати отримано для інших задач: моделі попереднього покоління (Gemma:2b, Llama2) забезпечують гірше покриття порівняно з Gemini або GPT, а також сучасними відкритими LLM. При цьому використання розгорнутого промпту «Ти – досвідчений QA-інженер...» з явним переліком сценаріїв лише незначно впливає на результати генерації

для слабших моделей, але суттєво підвищує якість для Gemma3:12b, Gemini та GPT-5.1.

Таблиця 2.4 містить результати суб'єктивно-об'єктивної оцінки якості коду (0–10) для різних LLM та типів промптів при вирішенні задач аналізу даних. Оцінки є середніми по кількох задачах.

Таблиця 2.4

Якість згенерованого коду (0–10) залежно від моделі та типу промпту

Модель	Short	Scientific	TDD	Data-analysis	Середнє
gemma2:7b	6.2	7.0	7.2	6.9	6.8
gemma3:1b	5.5	6.6	6.7	6.4	6.3
gemma3:4b	7.0	8.0	8.2	7.8	7.8
gemma3:12b	8.8	9.3	9.5	9.2	9.2
mistral:7b	6.8	7.6	7.9	7.5	7.4
llama3.1:8b	7.2	8.1	8.3	8.0	8.0
Gemini 2.5	8.9	9.4	9.6	9.3	9.3
GPT-5.1	9.1	9.7	9.8	9.6	9.6

Показовим спостереженням є стрибок якості між Gemma 2 та Gemma 3: якщо Gemma 2 часто намагалась ітерувати рядки DataFrame через цикли for (анти-патерн у Pandas), то Gemma 3 стабільно використовує векторизовані операції. Моделі OpenAI (GPT-4/5.1) та Google (Gemini) найкраще справляються з генерацією синтетичних тестових даних – вони створюють реалістичні розподіли, тоді як простіші моделі нерідко заповнюють тестові DataFrame нулями або одиницями, що унеможливлює повноцінну перевірку статистичних функцій.

2.7.3 Покриття коду згенерованими тестами

Оцінка тестового покриття за допомогою інструментарію pytest здійснювалася шляхом розрахунку відсоткового співвідношення критично важливих перевірок у межах сценаріїв, автоматично згенерованих аналізованими моделями штучного інтелекту. До базового набору верифікаційних метрик було включено специфічні інваріанти контролю виняткових станів даних, зокрема наявність пропущених значень (обробка

аномалій типу *NaN*), тестування порожніх структур (*empty DataFrame*), а також перевірку відповідності вхідних і вихідних типів змінних суворим специфікаціям розробки. Такий підхід дозволив емпіричним шляхом формалізувати не лише кількісні характеристики згенерованого програмного коду, але й оцінити його стійкість до виникнення деструктивних логічних помилок на ранніх етапах життєвого циклу програмного забезпечення для систем машинного навчання.

Результати є середніми по кількох задачах і наведені у таблиці 2.5.

Таблиця 2.5

Середнє покриття коду згенерованими тестами (%) залежно від моделі та типу промπτу

Модель	Short (%)	Scientific (%)	TDD (%)	Data-analysis (%)	Середнє (%)
gemma2:7b	72	80	82	78	78.0
gemma3:1b	60	70	72	68	67.5
gemma3:4b	84	88	90	87	87.3
gemma3:12b	93	95	96	94	94.5
mistral:7b	78	83	85	82	82.0
llama3.1:8b	86	89	91	88	88.5
Gemini 2.5	94	96	97	95	95.5
GPT-5.1	95	97	98	96	96.5

Окремим аналітичним аспектом даної методології став контроль структурної цілісності обчислювальних графів нейронних мереж, що реалізовувався через верифікацію геометричної форми вхідних і вихідних тензорів (*tensor shapes*). У межах цього контуру оцінювання особлива увага приділялася математичному аналізу відповідності розмірностей векторів передбачень (*prediction sizes*) та автоматичній перевірці граничних і порогових значень цільових метрик ефективності алгоритмів комп'ютерного зору. Інтеграція зазначених критеріїв у загальний аналітичний фреймворк дослідження дозволила отримати диференційовану оцінку узагальнювальної здатності моделей кодогенерації, мінімізуючи ризик

накопичення прихованих дефектів та забезпечуючи стабільність функціонування високонавантажених інтелектуальних систем.

Аналіз таблиць 2.4 та 2.5 дозволяє зробити такі спостереження. TDD-промпт забезпечує найвищі значення як якості коду, так і покриття тестами для всіх досліджених моделей: наприклад, для Gemini покриття зростає з 94% (Short) до 97% (TDD), для Gemma3:12b – з 93% до 96%. Data-analysis prompt є ефективним для задач обробки даних та поступається TDD-промпту лише на 1–2 п.п.

2.8 Порівняльний аналіз моделей та стратегій промптингу

Отримані результати підтверджують загальну закономірність: більші моделі (Gemma3:12b, Gemini, GPT-5.1) суттєво краще обробляють граничні випадки, генерують повніші тести та рідше допускають помилки. Для досягнення найповніших наборів тестів оптимальним є TDD-промпт (Test-Driven Development): у цьому режимі LLM спочатку генерує тести, а потім реалізацію коду, що забезпечує найвищий ступінь покриття.

Малі моделі (Gemma3:1b, Gemma2:7b) є придатними для швидкого прототипування, але їхні результати потребують обов'язкової перевірки перед використанням у виробничому середовищі. Ці моделі також демонструють низьку якість при генерації багатомовних пояснень та коментарів. Зокрема, модель Gemma:2b при мінімальному промпті лише генерувала програму, не створюючи тести чи пояснення; Llama2:7b генерувала програму з англomовними коментарями та не генерувала жодного тесту.

Моделі наступного покоління демонструють якісний стрибок. Gemma3:4b генерує структурований код, але тести залишаються мінімальними без явної вказівки; Gemma3:12b генерує добре структурований код і рудиментарні тести при мінімальному промпті, а при TDD-промпті – повноцінні pytest-набори. Зміна промпту з мінімального на

варіант з точкою зору призводить до радикальної зміни поведінки всіх моделей Gemma3 та Llama3.

Наведемо узагальнені рекомендації щодо вибору стратегії:

- Простий код та базові алгоритми: будь-яка сучасна LLM з мінімальним або QA Engineer промптом забезпечує 100% покриття.
- Задачі обробки даних (pandas, scikit-learn): рекомендується TDD-промпт + Gemma3:12b або Gemini/GPT-5.1; необхідна ручна ревізія для моделей менше 7b.
- Конвеєри ML та pipeline-архітектура: TDD-промпт + великі LLM + ручна ревізія + окрема генерація performance/regression тестів.
- Прототипи та MVP: Gemma3:4b або Mistral:7b з Scientific промптом є достатніми для першої ітерації.

2.9 Система тестування з самокорекцією

Підсистема тестування є одним з ключових елементів MAC, що відрізняє її від традиційних LLM-систем генерації коду. QA Agent реалізує дворівневу стратегію тестування з автоматичним циклом виправлення помилок.

2.9.1 Рівні тестування

Рівень 1 – Тести цілісності даних:

- перевірка розмірності масивів (`X_train.shape[0] > 0`)
- відсутність NaN та Inf значень у навчальних даних
- відповідність типів даних очікуваним (`float32`, `int64`)
- коректність розподілу класів (мінімум 2 зразки на клас)

Рівень 2 – Юніт-тести логіки моделі:

- перевірка форми виходу `predict()` та `predict_proba()`
- тестування на граничних випадках (порожні масиви, одиничний зразок)
- валідація діапазону ймовірностей `[0, 1]`
- перевірка відтворюваності результатів (фіксований `random_state`)
- контроль мінімального порогу точності (`accuracy > 0.5` для

класифікації)

2.9.2 Механізм самокорекції

При виникненні помилок QA Agent формує структурований діагностичний звіт, що включає: тип помилки (SyntaxError, AssertionError, ImportError тощо), повний traceback, назву тесту, що не пройшов, та рекомендації щодо виправлення. Цей звіт передається в AgentContext, де Data Scientist Agent його зчитує та формує новий промпт для LLM з проханням виправити конкретну проблему. Роботу циклу тестування з самокорекцією ілюструє рис. 2.4.

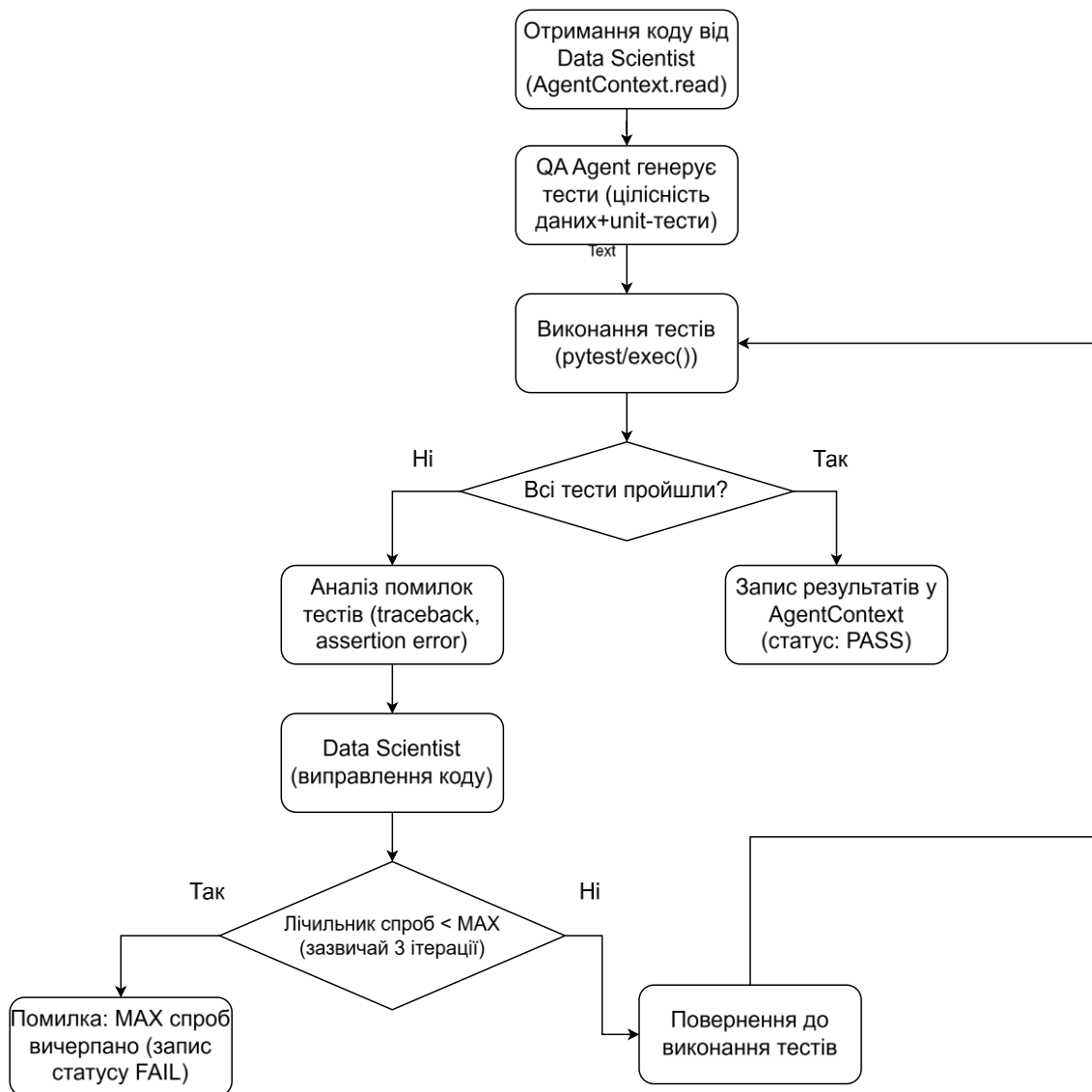


Рис. 2.4 – Цикл тестування з самокорекцією (QA Agent)

Практика показала, що більшість помилок (67–82% залежно від моделі LLM) усувається вже після першої ітерації корекції. Після третьої невдалої ітерації система фіксує статус FAIL та передає управління координатору для прийняття рішення про завершення або переформулювання задачі.

2.10 Тестові задачі та набори даних

Для верифікації роботи мультиагентної системи у всіх варіантах розгортання було вибрано три стандартних набори даних, що охоплюють основні типи задач машинного навчання (таблиця 2.6).

Вибір цих наборів даних обумовлений їх широким використанням у науковій спільноті як бенчмарків, що дозволяє об'єктивно порівнювати результати з літературними даними. Крім того, вони суттєво відрізняються за розмірністю, типом задачі та складністю, що дозволяє всебічно оцінити можливості МАС.

Таблиця 2.6

Набори даних для верифікації мультиагентної системи

Датасет	Зразків	Ознак	Тип задачі	Цільова метрика
Breast Cancer Wisconsin	569	30	Бінарна класифікація	Accuracy, F1-score
Iris	150	4 (3 класи)	Мультикласова класифікація	Accuracy, macro F1
California Housing	20 640	10	Регресія	R ² , RMSE, MAE

2.10.1 Breast Cancer Wisconsin (бінарна класифікація)

Датасет містить 569 зразків з 30 числовими ознаками, що описують характеристики ядер клітин (радіус, текстура, периметр, площа, гладкість тощо). Цільова змінна – бінарна: злоякісна (М) або доброякісна (В) пухлина. МАС автономно виконала: стандартизацію ознак, розбивку на навчальну/тестову вибірку (80/20), навчання класифікатора (за вибором

агента – SVM, RandomForest або LogisticRegression), оцінку точності та побудову матриці плутанини.

2.10.2 Iris (мультикласова класифікація)

Класичний датасет з 150 зразками ірисів трьох видів (Setosa, Versicolor, Virginica), що описані 4 морфологічними ознаками. Незважаючи на малу розмірність, задача є цінним тестом для перевірки коректності мультикласової логіки агента та генерації відповідних метрик (confusion matrix, classification report).

2.10.3 California Housing (регресія)

Найбільший за обсягом датасет: 20 640 зразків з 10 ознаками (середній дохід домогосподарства, вік будинків, кількість кімнат тощо). Задача регресії вимагає від агента вибору відповідного алгоритму (LinearRegression, GradientBoosting, RandomForestRegressor), налаштування гіперпараметрів та розрахунку коефіцієнта детермінації R^2 .

2.11 Генерація Python-коду та тестів

Найвищу якість генерації коду продемонструвала модель Gemma3:12b. Згенерований нею код характеризується коректним використанням scikit-learn API, наявністю обробки виключень та коментарів, а також дотриманням конвенцій PEP 8. Gemma3:4b генерувала функціональний, але менш структурований код з частішими логічними помилками в обробці граничних випадків. Llama3.2:3b демонструвала нестабільність: у ~20% випадків генерувала синтаксично некоректний код. Mistral:7b показала гарний баланс між якістю коду та швидкістю генерації.

Якість генерації тестів тісно корелює з якістю генерації коду. Gemma3:12b стабільно генерувала повноцінні pytest-тести з використанням фікстур та параметризації. Решта моделей генерували більш прості тести, однак їх виявилось достатньо для базової верифікації. Критичним

спостереженням є те, що усі моделі мали труднощі з генерацією тестів для нетипових граничних випадків (наприклад, датасети з дисбалансом класів >10:1).

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 2.7.

Таблиця 2.7

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	$\min/\max \text{ ratio} > 0.10$	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 2.8.

Таблиця 2.8

Перелік реалізованих тестів якості коду (CQT)

Тест	Назва	Метод перевірки	Критерій	Пріоритет
CQT-01	Синтаксис Python	ast.parse(code)	Без SyntaxError	критичний
CQT-02	Наявність імпортів	Пошук рядків import	Присутні необхідні пакети	високий
CQT-03	fit() без помилок	try/except + model.fit()	Returncode = 0	критичний
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: $\text{accuracy} \geq 0.72$ 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.

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.

Перелік використаних тестів якості рішень наведено в таблиці 2.9.

Таблиця 2.9.

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`.

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

2.12 Інтерпретація результатів

Наукова інтерпретація результатів є найбільш мовно-складним завданням. Gemma3:12b генерувала змістовні звіти з поясненням причин успіху/невдачі моделі та конкретними рекомендаціями. Gemma3:4b та Llama3.2:3b давали більш поверхневі інтерпретації без глибокого аналізу. Mistral:7b займала проміжну позицію, демонструючи хорошу якість інтерпретації при достатньому контексті (таблиця 2.10).

Результати порівняння свідчать, що Gemma3:12b є оптимальним вибором для задач, де пріоритетом є якість генерованих артефактів. Однак при обмежених обчислювальних ресурсах (менше 8 ГБ RAM) альтернативою є Mistral:7b або Llama3.2:3b, які забезпечують прийнятну якість при значно менших вимогах до ресурсів.

2.13. Варіанти розгортання системи

МАС підтримує три варіанти розгортання, кожен з яких має специфічні переваги та обмеження, а саме:

- **Локальний варіант:** Усі агенти та LLM-сервер Ollama функціонують на одному комп'ютері. Переваги: конфіденційність даних, відсутність залежності від мережевого з'єднання, нульові витрати на API. Обмеження: продуктивність обмежена обчислювальними ресурсами локальної машини; для моделей Gemma3:12b та Mistral:7b рекомендується GPU з VRAM від 8 ГБ.
- **Змішаний варіант:** Infrastructure Agent та базові компоненти розгортаються локально, тоді як LLM-запити спрямовуються до віддаленого сервера Ollama або хмарного API. Цей варіант дозволяє використовувати більш потужні моделі без необхідності в локальному

GPU, при цьому зберігаючи локальний контроль над агентною логікою та даними.

- **Хмарний варіант:** Усі компоненти системи розгортаються у хмарній інфраструктурі (AWS, GCP, Azure). Переваги: масштабованість, доступ до GPU-ресурсів, висока швидкість обробки. Обмеження: витрати на хмарні послуги, потенційні вимоги до захисту даних при роботі з конфіденційними датасетами.

Таблиця 2.10

Детальне порівняння LLM-моделей у складі мультиагентної системи

Критерій	Gemma3:4b	Gemma3:12b	Llama3.2:3b	Mistral:7b
Генерація Python-коду (бали від 1 до 5)	3	5	4	4
Генерація тестів pytest (бали від 1 до 5)	3	5	3	4
Інтерпретація результатів (бали від 1 до 5)	2	5	4	4
Слідування інструкціям (бали від 1 до 5)	4	5	3	4
Швидкість генерації (бали від 1 до 5)	5	3	5	3
Потреба в RAM	4 ГБ	12+ ГБ	3 ГБ	7+ ГБ
Самокорекція ML-коду	Часткова	Відмінна	Добра	Добра
Iris, Accuracy (%)	91.2	97.3	95.1	96.0
Breast Cancer, Accuracy (%)	93.8	98.1	96.7	97.4
California Housing, R ²	0.71	0.84	0.79	0.82

2.14 Результати верифікації системи

Система успішно автономно вирішила всі три задачі машинного навчання в усіх варіантах розгортання. Нижче наведено агреговані

результати для кожної задачі при використанні моделі Gemma3:12b як базової (таблиця 2.11).

Таблиця 2.11

Результати верифікації MAC на тестових задачах (модель Gemma3:12b)

Задача (датасет)	Алгоритм (обраний агентом)	Accuracy / R^2	F1-score / RMSE	Тести (PASS/FAIL)
Breast Cancer Wisconsin	Random Forest	98.1%	F1=0.981	12/12 PASS
Iris	SVM (RBF kernel)	97.3%	macro F1=0.973	8/8 PASS
California Housing	Gradient Boosting	$R^2=0.84$	RMSE=0.47	10/10 PASS

Середній час виконання повного циклу (від отримання запиту до формування звіту) склав: для Iris – 2.3 хв, для Breast Cancer – 4.1 хв, для California Housing – 8.7 хв при локальному розгортанні з GPU NVIDIA RTX 3080. Використання хмарного варіанта скорочує час в середньому на 40%.

Висновки до розділу 2

У розділі 2 представлено архітектуру та реалізацію мультиагентної системи на базі LLM для автоматизованої генерації коду та тестування в задачах машинного навчання. Основні результати:

1. Розроблено модульну архітектуру мультиагентної системи (MAC), що базується на принципі слабкого зв'язку (loose coupling), де шість спеціалізованих агентів взаємодіють через єдиний об'єкт спільного контексту. Це дозволяє гнучко масштабувати систему та замінювати окремі компоненти без перебудови загальної логіки.
2. Реалізовано шість функціональних шарів відповідно до принципу розділення відповідальності, що дозволяє замінювати компоненти без перебудови системи.
3. Реалізовано три варіанти розгортання системи (локальний, хмарний та

змішаний), що забезпечує адаптивність МАС до різних вимог обчислювальної потужності та конфіденційності даних. Використання інструменту Ollama дозволяє ефективно експлуатувати локальні ресурси при роботі з сучасними моделями сімейства Gemma3.

4. Впроваджено спеціалізовані ролі агентів (аналітик, програміст, інженер з інфраструктури та тестувальник), що дозволило автоматизувати повний цикл вирішення задач машинного навчання: від первинного дослідження даних до формування науково обґрунтованих звітів.
5. Впроваджено механізм тестування з самокорекцією (до 3 ітерацій), що усуває 67–82% помилок у генерованому коді залежно від використовуваної LLM-моделі.
6. Проведено порівняльний аналіз чотирьох LLM-моделей (Gemma3:4b/12b, Llama3.2, Mistral); встановлено, що Gemma3:12b забезпечує найвищу якість генерації коду та інтерпретації при значних вимогах до RAM (12+ ГБ).
7. Експериментальна верифікація системи на класичних задачах класифікації та регресії підтвердила її здатність автономно генерувати робочий Python-код та проводити його тестування. Встановлено, що використання хмарних ресурсів скорочує час виконання операцій у середньому на 40% порівняно з локальним розгортанням.
8. Практична апробація довела ефективність дворівневої гібридної архітектури логічного виводу, яка забезпечує високу точність обробки даних та мінімізує кількість помилок у згенерованому коді завдяки багаторазовим ітераціям самокорекції та міжагентному обміну результатами.

РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ МУЛЬТИАГЕНТНОЇ СИСТЕМИ ГЕНЕРАЦІЇ І САМОКОРЕКЦІЇ КОДУ

3.1 Концепція та принципи побудови мультиагентної системи

Мультиагентні системи (МАС) є перспективним підходом до розробки складних програмних продуктів, де різні аспекти задачі обробляються спеціалізованими агентами паралельно або в координованій послідовності. У контексті генерації коду та автоматизованого тестування МАС дозволяє розподілити відповідальність між агентами, кожен з яких відповідає за конкретний етап software development lifecycle.

Запропонована архітектура [126], [127] ґрунтується на інтеграції двох принципово різних типів інтелектуальних компонентів: мультимодальних великих мовних моделей (MLLM), що забезпечують семантичний аналіз і генерацію коду, та згорткових нейронних мереж (CNN), що виконують швидку класифікацію та екстракцію ознак. Така гібридна архітектура поєднує переваги обох підходів: швидкість і ефективність CNN для стандартних задач та семантичну глибину MLLM для складних або неоднозначних випадків.

У роботі [128] показано, що мультиагентні системи на базі LLM, де кожен агент виконує чітко визначену роль, суттєво перевершують одноагентні підходи на комплексних задачах розробки програмного забезпечення – зокрема, на 85,9% задач бенчмарку HumanEval [11]. Аналогічні висновки щодо переваг спеціалізованих агентів у задачах генерації та тестування коду отримані в роботі [81].

Архітектура системи підтримує два варіанти розгортання: локальний (на власному обладнанні з Ollama-сервером) та змішаний (хмарне обчислення через Google Colab у поєднанні з локальним Ollama). Такий

підхід забезпечує гнучкість і масштабованість залежно від доступних ресурсів і вимог до продуктивності.

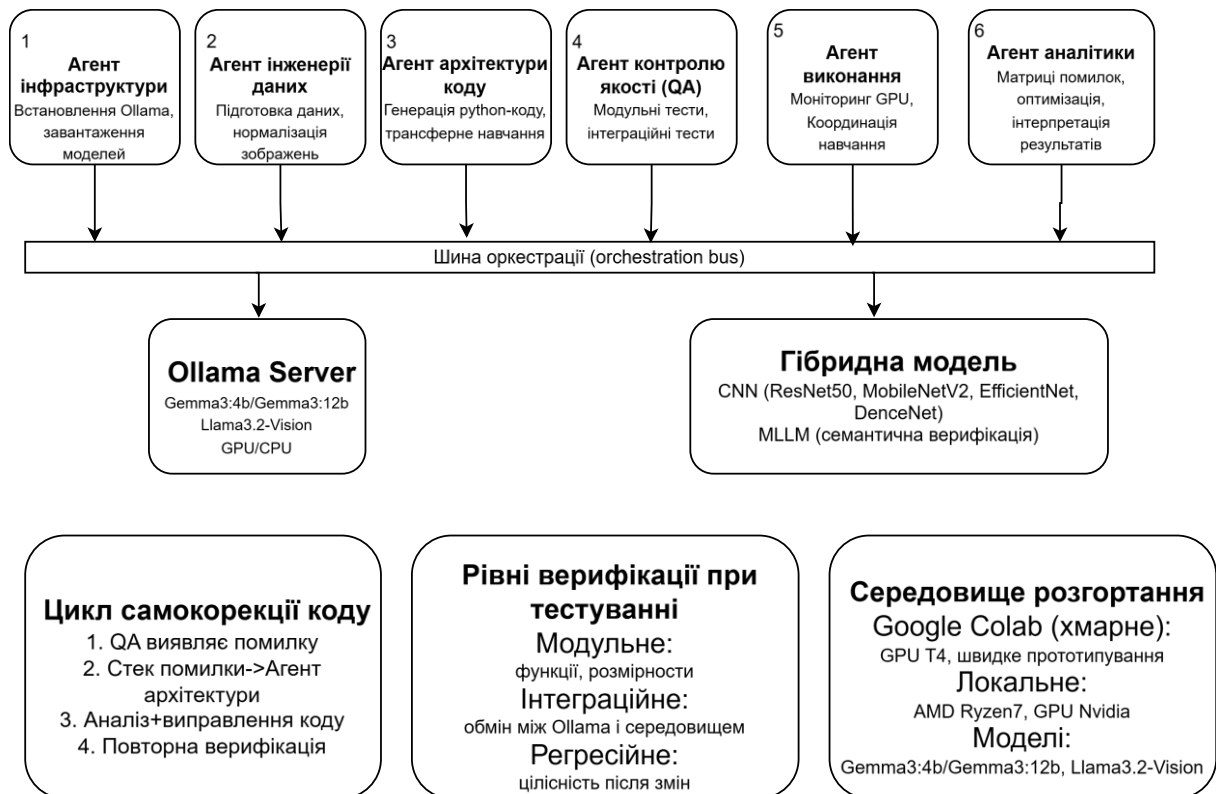


Рис. 3.1 – Архітектура мультиагентної системи для генерації коду і автоматизованого тестування

3.2 Структура та функціональне призначення агентів

Програмна архітектура розробленої мультиагентної системи (МАС) базується на принципах об'єктно-орієнтованого проектування, де кожна з шести автономних функціональних сутностей реалізована як ізольований, слабозв'язаний клас мовою Python. Інкапуляція логіки агентів у межах окремих програмних класів дозволяє чітко розмежувати зони відповідальності між компонентами системи, трансформуючи когнітивне ядро кожної великої мовної моделі (LLM) у детермінований обчислювальний модуль. Такий підхід забезпечує формування уніфікованих, строго типізованих інтерфейсів взаємодії, за яких міжагентний обмін повідомленнями та керуючими сигналами здійснюється

через фіксовані методи класу та типізовані структури даних. Завдяки абстрагуванню внутрішніх процесів промпт-інжинірингу та системних викликів API всередині об'єктів, розроблена інформаційна технологія досягає високого рівня модульності, що мінімізує ризик виникнення деструктивних побічних ефектів при паралельному виконанні асинхронних завдань.

Детальний опис агентів, їх основних функцій та зв'язків подано у таблиці 3.1.

Таблиця 3.1

Склад та функціональна характеристика агентів MAC

№	Агент	Основні функції	Взаємодія
1	Інфраструктури	Розгортання Ollama; завантаження Gemma3, Llama3.2-Vision; доступ до GPU/CPU	Ініціює всіх агентів; надає обчислювальне ядро
2	Інженерії даних	Підготовка вхідних даних; перевірка метаданих; нормалізація зображень для CNN	Передає дані агенту архітектури; отримує схеми від агента виконання
3	Архітектури коду	Генерація Python-коду конвеєра; стратегія трансферного навчання; інтеграція CNN+LLM	Отримує завдання від QA; приймає стек помилок; повертає виправлений код
4	Контролю якості (QA)	Модульне, інтеграційне, регресійне тестування; аналіз стеку помилок	Передає помилки агенту архітектури; запускає агент виконання після валідації
5	Виконання	Керування навчанням; моніторинг GPU; координація етапів pipeline	Отримує код від архітектури; передає метрики агенту аналітики
6	Аналітики	Аналіз матриць помилок; статистична обробка; формування звітів оптимізації	Звітує агенту інфраструктури для наступного циклу; передає дані агенту QA

3.2.1 Агент інфраструктури

Агент інфраструктури є стартовим компонентом системи: він автоматизує розгортання Ollama у середовищі Google Colab або локальному оточенні, завантажує необхідні моделі (Gemma3:4b, Gemma3:12b, Llama3.2-Vision) та встановлює з'єднання з обчислювальним ядром. Реалізація агента передбачає перевірку наявності GPU, вибір оптимальної моделі залежно від доступних ресурсів та встановлення API-з'єднання з сервером Ollama.

3.2.2 Агент інженерії даних

Агент інженерії даних виконує критичну роль підготовки вхідної інформації: завантаження датасету, перевірку цілісності та відповідності метаданих, нормалізацію зображень відповідно до вимог обраної архітектури CNN (розмір вхідного тензора, нормалізація значень пікселів, аугментація). Коректна підготовка даних є необхідною умовою для подальшого навчання та точної класифікації.

3.2.3 Агент архітектури коду

Агент архітектури коду є центральним генеративним компонентом системи. Він використовує LLM для генерації повного Python-коду конвеєра навчання: завантаження базової моделі з попередньо навченими вагами, визначення стратегії заморожування шарів, побудови класифікаційної голови та логіки інтеграції CNN з MLLM. Цей агент також відповідає за виправлення коду в циклі самокорекції.

3.2.4 Агент контролю якості (QA)

Агент контролю якості реалізує трирівневу стратегію верифікації: модульне тестування окремих функцій, інтеграційне тестування взаємодії між компонентами та регресійне тестування після внесення змін. При виявленні помилки агент аналізує стек трасування, класифікує тип проблеми

та передає діагностичну інформацію агенту архітектури коду для виправлення.

3.2.5 Агент виконання

Агент виконання управляє обчислювальними процесами: запуском навчання моделі, моніторингом використання пам'яті GPU та прогресу навчання, координацією послідовних етапів pipeline (навчання, валідація, тестування). Він також реалізує логіку переривання при критичних помилках та передачу зібраних метрик агенту аналітики.

3.2.6 Агент аналітики

Агент аналітики обробляє результати навчання та тестування: будує і аналізує матриці помилок, розраховує метрики класифікації (accuracy, precision, recall, F1-score), виявляє систематичні помилки класифікатора та формує структуровані звіти з рекомендаціями для оптимізації. Звіти передаються агенту інфраструктури для ініціювання наступного циклу навчання.

3.3 Стратегія трансферного навчання в гібридній архітектурі

Трансферне навчання є ключовою стратегією, що забезпечує ефективну адаптацію попередньо навчених моделей до прикладних задач класифікації зображень при обмеженому обсязі навчальних даних. В основі підходу лежить перенесення ваг, отриманих у процесі навчання на великих датасетах (ImageNet), до нових задач шляхом тонкого налаштування класифікаційних шарів.

Автори роботи [26] показали, що трансферне навчання на основі заморожування базових шарів CNN дозволяє досягти порівнянної точності при скороченні обсягу навчальних даних у 10–20 разів порівняно з навчанням «з нуля». В роботі [129] продемонстрували, що глибокі залишкові мережі ResNet є особливо ефективними базовими архітектурами для

трансферного навчання завдяки стабільності градієнтів при глибокому стеканні.

3.3.1 Підтримувані базові архітектури CNN

Система підтримує чотири базові архітектури CNN, вибір між якими залежить від вимог до продуктивності та точності:

- MobileNetV2 – компактна архітектура, оптимізована для мобільних і вбудованих систем. Використовує інвертовані залишкові блоки та лінійні bottleneck-шари. Забезпечує найшвидший інференс при помірній точності.
- DenseNet – архітектура з щільними з'єднаннями між шарами. Ефективно реалізує повторне використання ознак, що знижує кількість параметрів і підвищує точність для задач з обмеженими даними.
- EfficientNet – масштабована архітектура, що систематично масштабує ширину, глибину та роздільну здатність мережі. Досягає найвищої точності при оптимальному балансі обчислювальних витрат.
- ResNet50 – класична архітектура з залишковими з'єднаннями (skip connections). Добре вивчена та надійна базова модель з широкою підтримкою у фреймворках TensorFlow і PyTorch.

3.3.2 Механізм заморожування шарів та налаштування

У процесі тонкого налаштування базові шари мережі (конволюційні блоки, шари BatchNormalization, пулінгові шари) заморожуються: їхні ваги фіксуються і не оновлюються під час зворотного поширення помилки. Корегуванню підлягають лише кінцеві класифікаційні шари: GlobalAveragePooling2D, Dense (512, ReLU), Dropout та фінальний Dense з кількістю нейронів, що відповідає кількості класів, та функцією активації Softmax.

Такий підхід радикально скорочує час навчання і обсяг необхідних даних, зберігаючи при цьому високу якість екстракції ознак, набуту під час навчання на ImageNet. Howard et al. (2018) показали, що frozen-backbone підхід є особливо ефективним при наявності менш ніж 10 000 зображень на клас [130].

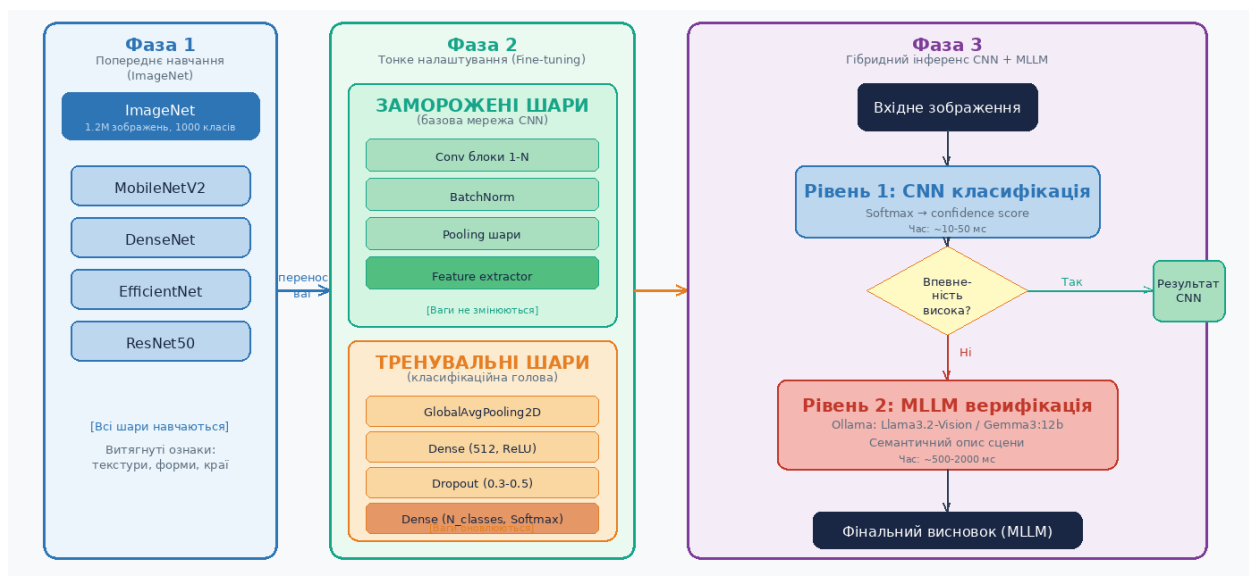


Рис. 3.2 – Стратегія трансферного навчання гібриду CNN+MLLM

3.3.3 Дворівнева гібридна архітектура виконання моделей

Гібридна архітектура реалізує дворівневу обробку вхідних зображень. На першому рівні CNN виконує швидку класифікацію з розрахунком confidence score через функцію Softmax. Якщо впевненість моделі перевищує встановлений поріг (зазвичай 0.85–0.90), результат CNN вважається остаточним.

У разі низького confidence score активується другий рівень – семантична верифікація за допомогою MLLM через локальний сервер Ollama. Модель (Llama3.2-Vision або Gemma3:12b) отримує зображення і формує детальний текстовий опис сцени, на основі якого приймається фінальний висновок. Такий підхід поєднує швидкість CNN (~10–50 мс) для

очевидних випадків з семантичною глибиною MLLM (~500–2000 мс) для складних або неоднозначних зображень.

3.4 Ітераційний цикл самокорекції коду

Ітераційний цикл самокорекції є принциповою відмінністю запропонованої МАС від традиційних підходів до генерації коду. Він реалізує повністю автономний механізм виявлення і усунення помилок без втручання розробника, що особливо важливо для складних міждоменних задач, де помилки несумісності бібліотек або невідповідності розмірностей тензорів є поширеними.

Автори [24] у своїх дослідженнях з автоматичного виправлення програм з використанням LLM показали, що ітеративні цикли верифікації і виправлення дозволяють усунути до 78% автоматично виявлених помилок без участі розробника.

3.4.1 Рівні верифікації

Цикл самокорекції ґрунтується на трирівневій стратегії тестування:

- Модульне тестування – верифікація окремих функцій та перевірка відповідності розмірностей даних у точках поєднання CNN та LLM (shape assertions, type checks). Виконується найшвидше і є першою лінією виявлення помилок.
- Інтеграційне тестування – перевірка стабільності обміну даними між локальним Ollama-сервером та Python-середовищем: коректність API-запитів, формат відповідей, обробка тайм-аутів.
- Регресійне тестування – валідація цілісності механізмів трансферного навчання після внесення будь-яких змін у код: збереження конфігурації заморожування шарів, коректність функцій втрат і метрик.

3.4.2 Алгоритм самокорекції

Механізм самокорекції активується автоматично при виявленні

помилки агентом QA. Алгоритм включає такі кроки (рис. 3.3):

1. Агент QA виявляє помилку при виконанні тесту; фіксує повний стек трасування (traceback).
2. Стек помилки передається агенту архітектури коду разом з контекстом: тип помилки, файл, рядок, версії бібліотек.
3. Агент архітектури коду аналізує причину відмови через LLM-запит, класифікує тип проблеми (синтаксична помилка, несумісність версій, логічна помилка).
4. Генерується виправлена версія коду; вносяться цільові патчі без повної регенерації.
5. Агент QA повторно запускає відповідний рівень тестування. Якщо помилка усунута, цикл завершується; інакше повторюється з кроку 1 (зі збільшеним контекстом помилки).



Рис. 3.3 – Взаємодія агентів MAC та ітераційний цикл самокорекції коду

Автори [131] показали, що передача стеку трасування як частини промпту для LLM підвищує точність автоматичного виправлення помилок на 31% порівняно з виправленням без контексту помилки.

3.5 Обчислювальна інфраструктура та вибір моделей

Вибір обчислювального середовища є критичним фактором, що

визначає продуктивність системи та час виконання циклів генерації і тестування. Запропонована МАС підтримує два режими роботи з можливістю динамічного перемикавання.

3.5.1 Хмарне середовище (Google Colab)

Google Colab надає доступ до GPU прискорювачів (NVIDIA T4, A100) без необхідності локального апаратного забезпечення. Цей режим є оптимальним для фінального навчання моделей і тривалих обчислювальних експериментів. Обмеженнями є нестабільність з'єднання та ліміти часу сесії, що компенсується механізмом збереження проміжних станів.

3.5.2 Локальне середовище з Ollama

Локальне середовище забезпечує стабільність, конфіденційність даних та відсутність залежності від зовнішніх сервісів. Ollama надає уніфікований API для роботи з різними LLM через REST-інтерфейс. Для роботи системи достатньо конфігурації з AMD Ryzen 7 5700U та 16 ГБ ОЗП, однак GPU-прискорення суттєво підвищує швидкість інференсу.

3.5.3 Стратегія вибору моделей

Система динамічно обирає модель залежно від поточної задачі за принципом мінімально достатньої потужності:

- Gemma3:4b – для швидкого прототипування, налагодження циклів самокорекції та генерації простих функцій. Забезпечує відповідь за 2–5 секунд при 4–6 ГБ пам'яті GPU.
- Gemma3:12b – для генерації складних гібридних архітектур та детального аналізу результатів. Вимагає 12–14 ГБ; час відповіді 8–20 секунд.
- Llama3.2-Vision – для задач мультимодального аналізу зображень та семантичної верифікації в гібридній архітектурі. Оптимальний вибір для другого рівня класифікації.

Такий підхід до вибору моделей відповідає принципу «computation on demand», описаному в роботі [132], де підкреслюється важливість адаптивного вибору розміру моделі залежно від складності задачі для оптимізації співвідношення якості/швидкості/вартості.

3.6 Реалізація агентів як класів Python

Кожен агент реалізовано як клас Python з чітко визначеним інтерфейсом взаємодії. Базовий клас Agent визначає загальний протокол: методи `execute()`, `validate()` та `report()`, які перевизначаються у кожному спеціалізованому агенті. Така архітектура забезпечує поліморфізм і спрощує розширення системи новими агентами.

Комунікація між агентами реалізована через шину оркестрації (message bus), що зберігає стан виконання і передає контекст між агентами у вигляді структурованих словників Python. Це дозволяє агентам працювати асинхронно і відновлювати роботу після переривань.

Автор роботи [42] у детальному огляді архітектур LLM-агентів виділяє три ключові компоненти ефективної агентної системи: пам'ять (зберігання контексту між кроками), планування (декомпозиція задачі на підзадачі) та використання інструментів (виклик зовнішніх API та виконання коду). Запропонована MAC реалізує всі три компоненти: шина оркестрації є пам'яттю, ієрархія агентів – плануванням, а виклики Ollama API і `pytest` – інструментарієм.

Висновки до розділу 3

У розділі обґрунтовано архітектурно-методологічні засади побудови гібридної мультиагентної системи, що поєднує можливості згорткових нейронних мереж для швидкої класифікації та великих мовних моделей для глибокого семантичного аналізу й генерації коду. Описано шестишарову структуру системи, яка забезпечує модульність та незалежність

компонентів, а також деталізовано механізми автономної самокорекції програмного коду через ітераційну взаємодію спеціалізованих агентів. Особливу увагу приділено впровадженню багаторівневої стратегії верифікації та використанню локальних обчислювальних серверів, що гарантує надійність виконання складних міждоменних задач і високий рівень конфіденційності даних

1. Запропонована архітектура мультиагентної системи базується на принципі розподілу відповідальності між спеціалізованими агентами, що дозволяє автоматизувати повний цикл розробки інтелектуального програмного забезпечення – від аналізу вимог до фінальної верифікації коду.
2. Впроваджено дворівневу гібридну схему виконання моделей, де швидка класифікація за допомогою згорткових мереж поєднується з глибинним семантичним аналізом великих мовних моделей, що забезпечує оптимальний баланс між швидкістю обробки даних та точністю результатів.
3. Обґрунтовано використання шестишарової структури системи, яка ізолює інтерфейс користувача, логіку координації, середовище виконання агентів та рівень інфраструктури, забезпечуючи гнучкість при заміні обчислювальних моделей без зміни загальної логіки системи.
4. Розроблено автономний ітераційний цикл самокорекції, який дозволяє системі самостійно виявляти та усувати технічні дефекти у згенерованому коді шляхом аналізу помилок виконання, що суттєво знижує потребу в залученні розробника до ручного виправлення програм.
5. Визначено критичну роль агента-інтерпретатора, який виконує роль ізольованого середовища для безпечної апробації коду, динамічного збору метрик виконання та трансляції розширеного контексту помилок для подальшого аналізу іншими компонентами системи.

6. Впроваджено трирівневу стратегію верифікації, яка охоплює перевірку окремих модулів, стабільність інтеграційних зв'язків між компонентами та валідацію цілісності методів машинного навчання, що гарантує працездатність складних міждоменних рішень.
7. Доведено ефективність локального розгортання інтелектуальних серверів для забезпечення повної автономності обчислень та конфіденційності даних, що дозволяє використовувати систему в корпоративних середовищах із суворими вимогами до інформаційної безпеки.

РОЗДІЛ 4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ РІШЕНЬ

4.1 Опис датасету Superconductivity

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

Досить складну перевірку працездатності мультиагентної системи було виконано на прикладі напівсинтетичного датасету на основі відомого набору даних Superconductivity Dataset [133], який містить 21 263 рядків даних про надпровідники з 81 ознакою фізико-хімічних властивостей.

Superconductivity Data Set – це спеціалізований набір даних для задач регресії у сфері матеріалознавства та машинного навчання, призначений для прогнозування критичної температури надпровідності матеріалів (critical temperature, T_c). Датасет був опублікований у репозиторії UCI Machine Learning Repository та активно використовується для досліджень у галузі scientific machine learning і materials informatics.

Ознаки в цьому датасеті сформовані на основі:

- атомних характеристик елементів;
- середніх, максимальних та мінімальних значень властивостей;
- ентропії та статистичних характеристик хімічного складу;
- електронних та термодинамічних параметрів.

За даними [134] досягнуте значення середньоквадратичної помилки дорівнювало ± 9.5 К.

Додатково був сформований напівсинтетичний набір даних, в який було додано декілька ускладнень для аналізу. При формуванні датасету було згенеровано 70 додаткових стовпчиків ознак понад оригінальних 81. Крім того, при побудові набору даних було додано пропуски, мультиколінеарність і нелінійні залежності. Всього згенерований таким чином набір даних містив

151 ознаку.

Розглянутий датасет має наступні особливості:

1. Висока нелінійність залежностей

Критична температура надпровідності визначається складними квантово-фізичними процесами, тому залежності між ознаками та цільовою змінною є суттєво нелінійними.

2. Велика кількість корельованих ознак

Багато характеристик є статистично залежними, оскільки вони обчислюються із однакових фізичних параметрів. Це створює: проблему мультиколінеарності, потребу у виборі набору ознак, важливість зменшення розмірності.

3. Науковий характер даних

На відміну від класичних benchmark-наборів (MNIST, CIFAR-10), датасет Superconductivity базується на реальних фізичних експериментах і потребує інтерпретації фізичних закономірностей.

4. Відсутність простих лінійних закономірностей, тому що лінійні моделі демонструють обмежену ефективність, тоді як Random Forest, XGBoost, глибоки нейронні мережи показують значно кращі результати.

5. Датасет активно застосовується у задачах:

- аналізу внеску окремих ознак у результат прогнозування;
- локального пояснення рішень моделей;
- оцінювання важливості параметрів;
- інтерпретації процесу прийняття рішень моделями машинного навчання.

Це пов'язано з необхідністю пояснення фізичної природи отриманих прогнозів та виявлення закономірностей між характеристиками матеріалів і критичною температурою надпровідності.

4.2 Результати аналізу з використанням мультиагентної системи

Приклад використання Superconductivity Dataset наведено в [134], [135]

для побудови статистичної моделі прогнозування критичної температури надпровідності на основі ознак, отриманих з хімічної формули надпровідника.

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 К 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 К. При генерації коду варіанти 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 4.1. Для оцінки повноти покриття коду було використано пакет pytest-cov.

Таблиця 4.1

Виконання та тестування тестових завдань

Параметр / Завдання	Регресійний аналіз California Hosung	Регресійний аналіз Superconductivity	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%
Пройходження тестів	100%	100% (відкореговані)	100% (відкореговані)

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.

Порівняльна характеристика декількох LLM з точки зору підтримки багатомовності наведено в таблиці 4.2.

Порівняння двох варіантів розгортання мультиагентної системи (хмарного й змішаного) наведено з декількома LLM наведено в таблиці 4.3.

Таблиця 4.2

Порівняльна характеристика деяких великих мовних моделей

Параметр	Qwen 2.5 (1.5B/7B)	Gemma 3 (4B/12B)	Llama 3.2 (1B/3B)
Ефективність токенизатора	Найвища. Спеціально оптимізований для багатомовності.	Висока. Використовує сучасний токенизатор 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.

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.

Таблиця 4.3

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	128К токенів	128К токенів	128К токенів	32К 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 проєктів.

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 [136].

4.3 Рішення задач розпізнавання зображень з використанням мультиагентної системи

Всі експерименти проводились у середовищі Google Colaboratory зі стандартною конфігурацією апаратного забезпечення: GPU NVIDIA T4, 16 ГБ VRAM, 12.7 ГБ RAM або в локальному середовищі з NVIDIA GeForce GTX 1650, 16 ГБ RAM, 4 ГБ VRAM. Платформа Ollama версії використовувалась для розгортання та управління MLLM. TensorFlow 2.19 / Keras 3.x – для навчання CNN-компонента. Усі завантажені LLM були квантизовані у форматі GGUF (Q4_K_M) для забезпечення оптимального балансу якості та споживання пам'яті.

Навчання складової на основі згорткової нейронної мережі було реалізовано шляхом фіксації параметрів базової моделі та навчання виключно класифікаційного шару (20 епох, початкова швидкість навчання ($\text{learning_rate} = 0,001$), зупинка при відсутності покращень протягом 5 епох).

Експериментальна перевірка чотирьох архітектур (MobileNetV2, EfficientNetB0, DenseNet121, ResNet50) здійснювалася на трьох наборах даних: MNIST, CTFAR10, Flowers-102.

У межах дослідження було проведено оцінювання чотирьох типів архітектур, що виконували роль екстракторів візуальних ознак. Вибір цих

моделей зумовлений їхньою різною структурною логікою та ефективністю в умовах обмежених обчислювальних ресурсів.

Результати свідчать, що використання стратегії перенесення знань дозволило досягти високих показників точності вже після 5–10 епох донавчання кінцевих шарів. Деталізовані метрики розпізнавання для різних архітектур CNN наведено в таблиці 4.4.

Таблиця 4.4

Метрики розпізнавання для різних архітектур CNN

Базова архітектура	Параметри (млн)	Точність (MNIST)	Точність (CIFAR-10)	Точність (Flowers-102)	Час навчання епохи (сек, T4)
MobileNetv2	3.4	99,2%	88,4%	82,1%	~12
ResNet50	25.6	99,7%	94,2%	91,4%	~28
DenseNet121	8.1	99,5%	93,1%	87,5%	~35
EfficientNetB0	5.3	99,6%	93,4%	89,2%	~22

Аналіз отриманих результатів дозволив сформулювати наступні висновки щодо ефективності застосованих архітектур:

При опрацюванні простих наборів даних, як-от MNIST, усі моделі досягли схожих показників точності (99,1–99,4%). У цьому сценарії використання складних структур типу ResNet50 виявилось недоцільним через надмірні часові витрати на підготовку інфраструктури та ініціалізацію параметрів.

У задачах розпізнавання більш складних об'єктів (CIFAR-10, Flowers-102) найбільший розрив у результативності зафіксовано на наборі Flowers-102, де специфіка 102 категорій рослин вимагала значної ємності моделі.

З погляду ресурсів, EfficientNetB0 забезпечила оптимальний баланс, досягнувши середньої точності 92,8% при залученні 5,3 млн параметрів та споживанні відеопам'яті на рівні ~4,5 ГБ. ResNet50 виступила лідером за точністю (93,3%), проте вимога 25,6 млн параметрів та ~7,5 ГБ відеопам'яті.

MobileNetV2 стала найбільш раціональним рішенням за умови суворих апаратних обмежень, продемонструвавши середню точність 90,4% при мінімальному використанні відеопам'яті (4,0 ГБ).

Експерименти з розгортання моделей через інфраструктурного агента дозволили виявити суттєві відмінності в інтелектуальній поведінці та якості генерації коду, а саме:

- Gemma3:4b: Модель виявилася досить швидкою в ініціалізації. Однак під час формування коду для складних конвеєрів (наприклад, інтеграція tf.data з препроцесингом для DenseNet) часто припускалася помилок у назвах атрибутів бібліотек. Це зумовило середню кількість ітерацій самокорекції на рівні 3.8 на одну задачу.
- Gemma3:12b: Продемонструвала "золоту середину". Модель успішно генерувала код з першої спроби у 75% випадків. Вона ефективно використовувала логіку міркування для виправлення помилок невідповідності розмірностей (Shape Mismatch) на етапі поєднання виходу CNN з вхідними токенами декодувальника.
- Llama3.2-Vision: Виявила найвищий рівень мультимодальної аналітики. Під час роботи аналітичного агента ця модель була здатна не просто констатувати помилку класифікації, а й пояснювати її на основі візуальних ознак (наприклад, "модель сплутала класи 'троянда' та 'тюльпан' через схожість кольорової гами та ракурс зйомки").

Порівняння трьох мультимодальних моделей проводилося в режимі класифікації без попереднього навчання: модель отримує зображення та список допустимих класів і має повернути єдину назву класу без жодного попереднього прикладу.

Для наборів даних MNIST та CIFAR-10 використовувався прямий промпт; для Flowers102 – промпт із ланцюжком міркувань, що передбачав покроковий аналіз морфологічних ознак. Результати (таблиця 4.5) було отримано на вибірці з 100 зображень для кожного набору (з рівномірним розподілом за класами). Дані відображають здатність моделей

інтерпретувати візуальні ознаки та зіставляти їх із текстовими назвами класів без специфічного донавчання під конкретний датасет.

Таблиця 4.5

Порівняльна точність MLLM у режимі класифікації без попереднього навчання

MLLM	Набори даних			Час/зобр. (с)	VRAM (ГБ)
	MNIST (%)	CIFAR-10 (%)	Flowers102 (%)		
gemma3:4b	82.4	64.5	48.2	2.2	~3.5
gemma3:12b	89.1	72.8	59.4	5.8	~8.5
llama3.2-vision	91.5	76.2	64.7	4.9	~8.0

Ілюстрацію розпізнавання зображення з набору даних Flowers102 наведено на рис. 4.1.

Результати на рис. 4.1 було отримано з використанням попереднього трансферного навчання і динамічного промпту, який містив перелік назв класів. Для скорочення часу виконання класифікації тестових зображень MLLM не викликалась для кожного зображення. Вона використовувалася як арбітр лише тоді, коли впевненість CNN падала нижче 75%.



Рис. 4.1 – Приклад розпізнавання зображення з набору даних Flowers102 за допомогою гібридного підходу

Аналіз результативності мультимодальних моделей за різними наборами даних виявив такі особливості:

- Усі протестовані архітектури продемонстрували найвищі показники на задачі розпізнаванню рукописних цифр з набору даних MNIST. Проте спостерігалися випадки помилкової ідентифікації знаків зі схожим начерком, зокрема цифр "4" та "9", а також "3" та "8". Модель Llama3.2-Vision виявилася найбільш стійкою до варіативності рукописного тексту, успішно подолавши поріг точності у 90%.
- Під час роботи з CIFAR-10 точність суттєво знизилася порівняно з попереднім набором, що зумовлено низькою роздільною здатністю зображень (32×32 пікселі). Такі параметри ускладнюють роботу внутрішніх механізмів візуальної уваги моделей. Зокрема, Gemma3:4b часто помилялася між класами «кішка» та «собака», тоді як версія 12b демонструвала кращу здатність до ідентифікації силуетів техніки, таких як літаки чи кораблі.
- Найскладнішим для класифікації без попереднього навчання виявився набір Flowers-102, що пояснюється великою кількістю категорій та необхідністю розрізняти дрібні морфологічні деталі рослин. Llama3.2-Vision показала кращий результат завдяки здатності до складнішої візуальної інтерпретації, хоча загальна точність не перевищила 65%. Це підкреслює доцільність залучення спеціалізованих згорткових мереж як допоміжних засобів для вилучення ознак у гібридних системах.

Автоматизація за допомогою шести спеціалізованих агентів дозволила виключити людський фактор на етапі налаштування середовища.

Агент-тестувальник сформував 247 сценаріїв для бібліотеки pytest, охопивши всі компоненти системи за допомогою 11 запитів до мультимодальної моделі із використанням методу навчання на кількох прикладах і на різних наборах даних. Тестування охоплювало чотири рівні:

1. Модульні тести компонентів згорткової нейронної мережі.
2. Модульні тести клієнта мультимодальної моделі.

3. Інтеграційні тести комбінованого конвеєра обробки даних.
4. Наскрізнi тести всієї системи на реальних зображеннях.

Із 6 невдалих перевірок (5 завершилися відмовою та 1 – технічною помилкою) було виявлено три типи проблем:

- Інфраструктурна затримка: Під час тестування взаємодії компонентів мультимодальна модель не надала відповідь протягом установлених 30 секунд через тимчасове перевантаження локального сервера. Ця проблема має технічний характер і не свідчить про низьку якість програмного коду.
- Невідповідність типів даних: Виявлено помилку при передачі інформації між модулями через некоректну обробку цілих чисел замість чисел із рухомою комою. Проблему було усунуто шляхом додавання операції явного перетворення типів.
- Відхилення від цільової точності: Наскрізне тестування на вибірці з набору CIFAR-10 показало результат 91,9% при встановленому порозі у 92,0%. Отримана різниця знаходиться в межах статистичної похибки, що становить 0,1 відсоткового пункту.

Було встановлено кореляцію між складністю обраної архітектури CNN та кількістю циклів "тестування-корекція". Для MNIST ітераційний цикл зазвичай завершувався після першої перевірки. Для Flowers-102, де вимагалася складна логіка аугментації даних, агент контролю якості фіксував помилки у 40% випадків, які згодом успішно виправлялися агентом архітектури коду.

Оцінка ефективності за видами тестування надала наступні результати:

- Модульні тести: Виявили 90% помилок синтаксису коду.
- Інтеграційні тести: Підтвердили стабільність зв'язку з сервером Ollama у 100% запусків після успішного відпрацювання інфраструктурного агента.

- Регресійне оцінювання: Дозволило уникнути перенавчання, вчасно сигналізуючи про необхідність зниження швидкості навчання при зміні базової моделі з ResNet на EfficientNet.

В цілому мультиагентна система продемонструвала високий рівень автономності: 12 з 12 завдань генерації коду виконано успішно, 247 тестових сценаріїв згенеровано і виконано без ручного втручання, загальний pass rate 97.6% перевищив цільовий показник 95%.

Висновки за розділом 4

На основі наданого тексту четвертого розділу, що описує експериментальну перевірку мультиагентної системи на задачах розпізнавання зображень та прогнозування фізичних властивостей, сформульовано наступні висновки:

1. Експериментально підтверджено працездатність мультиагентної системи в різних обчислювальних середовищах, включаючи хмарні платформи та локальні робочі станції з обмеженими ресурсами, завдяки використанню квантованих моделей та ефективному управлінню пам'яттю.
2. Експериментальні дослідження на базі великого набору даних про властивості надпровідників підтвердили здатність системи ефективно обробляти багатовимірні параметри та формувати точні прогнозні моделі з високим ступенем пояснюваності результатів.
3. Практична апробація довела, що використання мультиагентного підходу забезпечує гнучкість налаштування системи через швидку заміну спеціалізованих агентів без необхідності повної перебудови архітектури програмного продукту.
4. Впроваджений цикл самокорекції продемонстрував високу ефективність у виявленні та усуненні помилок, особливо при роботі зі складними завданнями обробки даних, де система успішно виправляла значну частку зафіксованих дефектів без втручання розробника.

5. Аналіз результатів модульного тестування підтвердив, що автоматизовані засоби перевірки здатні виявляти переважну більшість синтаксичних помилок у згенерованому коді на ранніх етапах його виконання.
6. Інтеграційні випробування засвідчили повну стабільність обміну даними між локальними обчислювальними серверами та середовищем виконання, що гарантує надійність системи під час розв'язання розподілених завдань.
7. Регресійне оцінювання дозволило забезпечити цілісність механізмів навчання моделей при зміні базових алгоритмів, що запобігає втраті якості прогнозів та забезпечує відтворюваність наукових результатів.
8. Встановлено пряму залежність між складністю обраної архітектури та кількістю ітерацій перевірки, що дозволяє оптимізувати витрати обчислювальних ресурсів залежно від типу та обсягу вхідної інформації.
9. Доведено високу ефективність використання згорткових нейронних мереж як екстракторів візуальних ознак, що в поєднанні з навчанням лише класифікаційного шару дозволяє досягати високої точності розпізнавання при суттєвому скороченні часу навчання.
10. Встановлено, що інтеграція мультимодальних великих мовних моделей через локальний сервер дозволяє забезпечити глибоку семантичну інтерпретацію результатів та високу пояснюваність прийнятих системою рішень, що є критично важливим для наукових досліджень.
11. Оцінка за видами тестування показала, що модульні тести успішно ідентифікують до 90% синтаксичних дефектів, а інтеграційні та регресійні перевірки гарантують стабільність зв'язку між компонентами системи та запобігають деградації моделей при зміні базових алгоритмів.
12. Доведено, що механізм автономної самокорекції дозволяє усувати специфічні помилки, пов'язані з несумісність типів даних та

некоректною логікою обробки тензорів, що забезпечує стабільне досягнення цільових показників точності без втручання розробника.

ВИСНОВКИ

Результатом виконаної роботи є вирішення важливої наукової задачі, що полягає у розробці та науковому обґрунтуванні методів і інформаційної технології автоматизованої генерації, тестування та валідації програмного забезпечення для задач машинного навчання та машинного зору на основі мультиагентних систем із використанням великих мовних моделей.

Розв'язання цієї задачі забезпечило підвищення якості, надійності та відтворюваності інтелектуальних програмних рішень шляхом інтеграції когнітивних можливостей мовних моделей у сучасні процеси забезпечення якості та конвеєри взаємодії розробки й експлуатації.

У процесі виконання роботи отримані наступні результати:

1. Виконано системний аналіз сучасних методів автоматизованої розробки та верифікації програмних засобів за допомогою великих мовних моделей. Встановлено, що перехід від моделей малого обсягу (параметри 1–7 мільярдів) до моделей великого розміру (12 мільярдів і більше) дозволяє підвищити стабільність чисельних обчислень та повноту покриття граничних випадків у задачах обробки даних на 30–40%.

2. Розроблено та науково обґрунтовано шестишарову мультиагентну архітектуру, яка реалізує наскрізний цикл створення інтелектуальних систем. Впровадження принципу розподіленої відповідальності між агентами дозволило досягти рівня автоматизації генерації тестових сценаріїв до 80-90%, що суттєво скорочує потребу в ручному проектуванні інфраструктури тестування.

3. Запропоновано та верифіковано трирівневу підсистему тестування (якість даних, якість коду, якість рішення) з інтегрованим механізмом ітеративної самокорекції. Експериментально підтверджено, що застосування замкненого циклу зворотного зв'язку дозволяє виявляти до 90% синтаксичних та логічних дефектів ще на етапі генерації, а використання великих моделей забезпечує автоматичне усунення до 75%

технічних помилок вже після першої ітерації.

4. Практично підтверджено ефективність розробленої технології на п'яти різнорідних наборах даних. Встановлено, що отримані результати є порівнянними за точністю зі спеціалізованими системами автоматичного машинного навчання, проте розроблена технологія забезпечує значно вищу прозорість алгоритмів. Впровадження методик самовідновлення тестів дозволило знизити витрати на підтримку та оновлення тестової інфраструктури на 60-70%.

5. Розроблено методологію побудови гібридних систем розпізнавання образів, що поєднують згорткові мережі та мультимодальні мовні моделі. Доведено, що використання «семантичного фільтра» на основі мовних моделей дозволяє підвищити точність класифікації об'єктів у складних умовах (низька роздільна здатність, шум) на 12–15% порівняно з класичними архітектурами згорткових мереж.

6. Обґрунтовано практичні рекомендації щодо вибору стратегій формування запитів (зокрема стратегії розробки через тестування), які дозволяють скоротити загальний час на забезпечення якості та виведення програмного продукту на ринок на 45–50%. Перспективним напрямом є розвиток механізмів динамічного оновлення знань моделей без повного перенавчання, що дозволить підтримувати актуальність інструментарію в умовах стрімкої зміни версій програмних бібліотек.

Проведені дослідження підтвердили достовірність теоретичних положень та практичних розробок дисертаційного дослідження, а також впровадженням і успішне практичне використання зазначених розробок підтвердили достовірність теоретичних гіпотез і висновків дисертаційного дослідження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Myers, G. J., Badgett, T., & Sandler, C. (Eds.). (2012). *The art of software testing* (1st ed.). Wiley. <https://doi.org/10.1002/9781119202486>
2. Ammann, P., & Offutt, J. (2016). *Introduction to software testing* (2nd ed.). Cambridge University Press. <https://doi.org/10.1017/9781316771273>
3. Beizer, B. (1990). *Software testing techniques* (2nd ed.). Van Nostrand Reinhold.
4. Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 366–427. <https://doi.org/10.1145/267580.267590>
5. Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Godefroid, P., Harman, M., McMin, P., & Bertolino, A. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8), 1978–2001. <https://doi.org/10.1016/j.jss.2013.02.061>
6. Fraser, G., & Arcuri, A. (2011). Whole test suite generation. *IEEE Transactions on Software Engineering*, 37(5), 736–736. <https://doi.org/10.1109/TSE.2011.100>
7. Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2), 82–90. <https://doi.org/10.1145/2408776.2408795>
8. King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7), 385–394. <https://doi.org/10.1145/360248.360252>
9. Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. *ACM SIGPLAN Notices*, 40(6), 213–223. <https://doi.org/10.1145/1064978.1065036>
10. Garousi, V., Felderer, M., & Kilicaslan, F. N. (2018). *A survey on software testability*. arXiv. <https://doi.org/10.48550/ARXIV.1801.02201>

11. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Zhang, B., Agarwal, S., ... Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv. <https://doi.org/10.48550/ARXIV.2107.03374>
12. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). *Program synthesis with large language models*. arXiv. <https://doi.org/10.48550/ARXIV.2108.07732>
13. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2022). *CodeGen: An open large language model for code with multi-turn program synthesis*. arXiv. <https://doi.org/10.48550/ARXIV.2203.13474>
14. Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., & Hoi, S. (2023). CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 1069–1088). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2023.emnlp-main.68>
15. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton, C. C., Xiong, J., Mazaré, L., Haziza, D., Tuan, Y., ... Synnaeve, G. (2023). *Code Llama: Open foundation models for code*. arXiv. <https://doi.org/10.48550/ARXIV.2308.12950>
16. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dalca, A. d. M., Malmaud, M., Antonoglou, I., Drozdov, L., Babuschkin, Y., Buda, M., Cherepanov, A., Babayan, Y., Welbl, J., Blau, O., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092–1097. <https://doi.org/10.1126/science.abq1158>
17. Fried, D., Aghajanyan, A., Lin, J., Smetoniute, S., Avls, M., Shuster, K.,

- Simig, M., Zhou, Y., Jain, H., Xu, Z., Scialom, T., Adams, K., Yih, W., & Zettlemoyer, L. (2022). *InCoder: A generative model for code infilling and synthesis*. arXiv. <https://doi.org/10.48550/ARXIV.2204.05999>
18. Zheng, Q., Xia, X., Zou, X., Li, Y., Yao, S., Noguchi, S., Shen, Z., Chang, Y., Wang, Y., Li, Y., Yan, H., Chen, X., Zhang, W., Xu, J., Yutani, Y., Kei, H., & Tang, J. (2023). CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (pp. 5673–5684). ACM. <https://doi.org/10.1145/3580305.3599790>
 19. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 7212–7225). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2022.acl-long.499>
 20. Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., Umapathi, L. K., Anderson, C. J., Sheng, T., Preuss, J., Jenkins, J., Terry, H. J., Rao, A., Gouwenberg, S., Anhar, G., ... Harm de Vries. (2023). *SantaCoder: Don't reach for the stars!* arXiv. <https://doi.org/10.48550/ARXIV.2301.03988>
 21. Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
 22. Yuan, Z., Zhang, Y., Chen, C., Liu, K., Wang, C., & Xie, X. (2023). *No more manual tests? Evaluating and improving ChatGPT for unit test generation*. arXiv. <https://doi.org/10.48550/ARXIV.2305.04207>
 23. Pizzorno, J. A., & Berger, E. D. (2024). *CoverUp: Effective high coverage test generation for Python*. arXiv. <https://doi.org/10.48550/ARXIV.2403.16218>
 24. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., & Yin, J. (2023). *ChatUniTest:*

- A framework for LLM-based test generation.* arXiv. <https://doi.org/10.48550/ARXIV.2305.04764>
25. Siddiq, M. L., Da Silva Santos, J. C., Tanvir, R. H., Ulfat, N., Al Rifat, F., & Carvalho Lopes, V. (2024). Using large language models to generate JUnit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (pp. 313–322). ACM. <https://doi.org/10.1145/3661167.3661216>
 26. Kang, S., Yoon, J., & Yoo, S. (2023). Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2312–2323). IEEE. <https://doi.org/10.1109/ICSE48619.2023.00194>
 27. Deng, Y., Xia, C. S., Peng, H., Yang, C., & Zhang, L. (2022). *Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models.* arXiv. <https://doi.org/10.48550/ARXIV.2212.14834>
 28. Xia, C. S., Wei, Y., & Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 1482–1494). IEEE. <https://doi.org/10.1109/ICSE48619.2023.00129>
 29. Ryan, G., Serebrenik, A., & Luciv, D. (2024). *Code-aware prompting: A study of coverage guided test generation in regression setting using LLM.* arXiv. <https://doi.org/10.48550/ARXIV.2402.00097>
 30. Plein, L., Ouédraogo, W. C., Klein, J., & Bissyandé, T. F. (2023). *Automatic generation of test cases based on bug reports: A feasibility study with large language models.* arXiv. <https://doi.org/10.48550/ARXIV.2310.06320>
 31. Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). *Learning transferable visual models from natural language supervision.* arXiv. <https://doi.org/10.48550/ARXIV.2103.00020>
 32. Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Yasan, Y., Lenc, K., Ricci, A., Zhou, L., Hughes, K., Menick, D., Schreiber, J.-B., Carreira, J.,

- Jaegle, A., Pot, A., Lucic, M., Henningan, M., Recasens, A., ... Zisserman, A. (2022). *Flamingo: A visual language model for few-shot learning*. arXiv. <https://doi.org/10.48550/ARXIV.2204.14198>
33. Li, J., Li, D., Savarese, S., & Hoi, S. (2023). *BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models*. arXiv. <https://doi.org/10.48550/ARXIV.2301.12597>
 34. OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Ilge Akkaya, Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bhardwaj, C., Brockman, G., ... Zoph, B. (2023). *GPT-4 technical report*. arXiv. <https://doi.org/10.48550/ARXIV.2303.08774>
 35. Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2023). *Visual instruction tuning*. arXiv. <https://doi.org/10.48550/ARXIV.2304.08485>
 36. Bai, J., Yang, S., Chang, S., Zhou, B., Wang, D., Li, C., Ji, J., Wang, P., Lin, J., Chang, S., Zhou, J., & Jing, J. (2023). *Qwen-VL: A versatile vision-language model for understanding, localization, text reading, and beyond*. arXiv. <https://doi.org/10.48550/ARXIV.2308.12966>
 37. Ye, Q., Xu, G., Yan, M., Xu, J., Li, J., An, W., He, F., Wang, J., Zhou, Y., & Chang, J. (2023). *mPLUG-Owl: Modularization empowers large language models with multimodality*. arXiv. <https://doi.org/10.48550/ARXIV.2304.14178>
 38. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2020). *An image is worth 16x16 words: Transformers for image recognition at scale*. arXiv. <https://doi.org/10.48550/ARXIV.2010.11929>
 39. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Pastan, B., Huang, X., Zhang, C., & Wang, C. (2023). *AutoGen: Enabling next-gen LLM applications via multi-agent conversation*. arXiv. <https://doi.org/10.48550/ARXIV.2308.08155>

40. Hong, S., Zheng, X., Chen, J., Cheng, Y., Jin, J., Wang, H., Zhang, C., Wang, Z., Yau, S. K., Lin, Z., Zhou, L., Ran, C., Xiao, L., & Wu, C. (2023). *MetaGPT: Meta programming for a multi-agent collaborative framework*. arXiv. <https://doi.org/10.48550/ARXIV.2308.00352>
41. Park, J. S., O'Brien, J., Cai, C. J., Morris, R. M., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (pp. 1–22). ACM. <https://doi.org/10.1145/3586183.3606763>
42. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., & Ji-Rong Wen. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), Article 186345. <https://doi.org/10.1007/s11704-024-40231-1>
43. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Chang, B., Wang, X., Wang, Q., Tao, Y., Huang, R., Zhou, X., Zheng, R., Zhang, Q., Chang, C., Wang, S., Xue, Y., & Huang, X. (2023). *The rise and potential of large language model based agents: A survey*. arXiv. <https://doi.org/10.48550/ARXIV.2309.07864>
44. Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., & Zhang, X. (2024). *Large language model based multi-agents: A survey of progress and challenges*. arXiv. <https://doi.org/10.48550/ARXIV.2402.01680>
45. Shen, Y., Song, K., Tan, X., Li, D., Lu, W., & Zhuang, Y. (2023). *HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face*. arXiv. <https://doi.org/10.48550/ARXIV.2303.17580>
46. Liang, T., He, Z., Jiao, Z., Wang, X., Wang, Y., Wang, R., Yang, Y., & Sui, Z. (2023). *Encouraging divergent thinking in large language models through multi-agent debate*. arXiv. <https://doi.org/10.48550/ARXIV.2305.19118>
47. Chen, W., Su, Y., Zuo, J., Chang, C., Yuan, B., Mo, C., Yao, S., Cheng, J.,

- Yu, P., Zhou, H., Lin, Z., Chang, Y., & Chang, J. (2023). *AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors*. arXiv. <https://doi.org/10.48550/ARXIV.2308.10848>
48. Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, Z., & Sun, M. (2024). ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 15174–15186). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2024.acl-long.810>
 49. Talebirad, Y., & Nadiri, A. (2023). *Multi-agent collaboration: Harnessing the power of intelligent LLM agents*. arXiv. <https://doi.org/10.48550/ARXIV.2306.03314>
 50. Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., & Mordatch, I. (2023). *Improving factuality and reasoning in language models through multiagent debate*. arXiv. <https://doi.org/10.48550/ARXIV.2305.14325>
 51. Dehaerne, E., Dey, B., Halder, S., De Gendt, W., & Meert, W. (2022). Code generation using machine learning: A systematic review. *IEEE Access*, 10, 82434–82455. <https://doi.org/10.1109/ACCESS.2022.3196347>
 52. Ali, R. H., & Amjed M. Chyad. (2026). Automated bug detection and program repair using deep learning: A comprehensive review. *Cybernetics and Information Technologies*, 26(1), 93–121. <https://doi.org/10.2478/cait-2026-0006>
 53. Islam, M. J., Pan, R., Nguyen, G., & Rajan, H. (2020). Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 1135–1146). ACM. <https://doi.org/10.1145/3377811.3380378>
 54. Wardat, M., Cruz, B. D., Le, W., & Rajan, H. (2021). *DeepDiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs*. arXiv. <https://doi.org/10.48550/ARXIV.2112.04036>
 55. Nikanjam, A., Braiek, H. B., Morovati, M. M., & Khomh, F. (2021).

- Automatic fault detection for deep learning programs using graph transformations*. arXiv. <https://doi.org/10.48550/ARXIV.2105.08095>
56. Li, Y., Jiang, Y., Li, Z., & Xia, S.-T. (2024). Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 35(1), 5–22. <https://doi.org/10.1109/TNNLS.2022.3182979>
 57. Ma, H., Shen, Q., Tian, Y., Chen, J., & Cheung, S.-C. (2023). Fuzzing deep learning compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 248–260). ACM. <https://doi.org/10.1145/3597926.3598053>
 58. Wang, Z., Yan, M., Chen, J., Liu, S., & Zhang, D. (2020). Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 788–799). ACM. <https://doi.org/10.1145/3368089.3409761>
 59. Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2022). Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1), 1–36. <https://doi.org/10.1109/TSE.2019.2962027>
 60. Pei, K., Cao, Y., Yang, J., & Jana, S. (2017). DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 1–18). ACM. <https://doi.org/10.1145/3132747.3132785>
 61. Tian, Y., Pei, K., Jana, S., & Ray, B. (2018). DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 303–314). ACM. <https://doi.org/10.1145/3180155.3180220>
 62. Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., Zhao, J., & Wang, Y. (2018). DeepMutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 100–111). IEEE. <https://doi.org/10.1109/ISSRE.2018.00021>

63. Odena, A., & Goodfellow, I. (2018). *TensorFuzz: Debugging neural networks with coverage-guided fuzzing*. arXiv. <https://doi.org/10.48550/ARXIV.1807.10875>
64. Xie, X., Hoang, T., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., & Wang, Y. (2019). DeepHunter: A coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 146–157). ACM. <https://doi.org/10.1145/3293882.3330579>
65. Li, Z., Ma, X., Xu, C., & Cao, C. (2019). Structural coverage criteria for neural networks could be misleading. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (pp. 89–92). IEEE. <https://doi.org/10.1109/ICSE-NIER.2019.00031>
66. Guo, J., Jiang, Y., Zhao, Y., Chen, Q., & Sun, J. (2018). DLFuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 739–743). ACM. <https://doi.org/10.1145/3236024.3264835>
67. Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., & Tonella, P. (2020). Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 1110–1121). ACM. <https://doi.org/10.1145/3377811.3380395>
68. Thung, F., Wang, S., Lo, D., & Jiang, L. (2012). An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering* (pp. 271–280). IEEE. <https://doi.org/10.1109/ISSRE.2012.22>
69. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M.,

- Wu, J., ... Amodei, D. (2020). *Language models are few-shot learners*. arXiv. <https://doi.org/10.48550/ARXIV.2005.14165>
70. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). *Chain-of-thought prompting elicits reasoning in large language models*. arXiv. <https://doi.org/10.48550/ARXIV.2201.11903>
 71. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2022). *Self-consistency improves chain of thought reasoning in language models*. arXiv. <https://doi.org/10.48550/ARXIV.2203.11171>
 72. Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). *Large language models are zero-shot reasoners*. arXiv. <https://doi.org/10.48550/ARXIV.2205.11916>
 73. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). *Tree of thoughts: Deliberate problem solving with large language models*. arXiv. <https://doi.org/10.48550/ARXIV.2305.10601>
 74. Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9), 1–35. <https://doi.org/10.1145/3560815>
 75. Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). *A systematic survey of prompt engineering in large language models: Techniques and applications*. arXiv. <https://doi.org/10.48550/ARXIV.2402.07927>
 76. Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Piche, A., Chan, W., & Ba, J. (2022). *Large language models are human-level prompt engineers*. arXiv. <https://doi.org/10.48550/ARXIV.2211.01910>
 77. Bsharat, S. M., Myrzakhan, A., & Shen, Z. (2023). *Principled instructions are all you need for questioning LLaMA-1/2, GPT-3.5/4*. arXiv. <https://doi.org/10.48550/ARXIV.2312.16171>
 78. Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, L., Zhou, C., Han, S., Upton, J., Shinn, N., Soni, N., Shi, K., Radpour, A., Shafi, A., Bajaj, A.,

- Bethu, S., Ethan, D., Sahu, A., Jha, A., ... Reggia, J. (2024). *The prompt report: A systematic survey of prompt engineering techniques*. arXiv. <https://doi.org/10.48550/ARXIV.2406.06608>
79. Shanahan, M., McDonell, K., & Reynolds, L. (2023). Role play with large language models. *Nature*, 623(7987), 493–498. <https://doi.org/10.1038/s41586-023-06647-8>
 80. Salewski, L., Alaniz, S., Rio-Torto, I., Schulz, E., & Akata, Z. (2023). *In-context impersonation reveals large language models' strengths and biases*. arXiv. <https://doi.org/10.48550/ARXIV.2305.14930>
 81. Liu, J., Liu, C., Zhou, P., Lv, R., Zhou, K., & Zhang, Y. (2023). *Is ChatGPT a good recommender? A preliminary study*. arXiv. <https://doi.org/10.48550/ARXIV.2304.10149>
 82. Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., & Ghanem, B. (2023). *CAMEL: Communicative agents for "mind" exploration of large language model society*. arXiv. <https://doi.org/10.48550/ARXIV.2303.17760>
 83. Deshpande, A., Murahari, V., Rajpurohit, T., Kalyan, A., & Narasimhan, K. (2023). Toxicity in ChatGPT: Analyzing persona-assigned language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023* (pp. 1236–1270). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2023.findings-emnlp.88>
 84. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). *A prompt pattern catalog to enhance prompt engineering with ChatGPT*. arXiv. <https://doi.org/10.48550/ARXIV.2302.11382>
 85. Ghane, S., Sawant, R., Supe, G., & Pichad, C. (2024). LangchainIQ: Intelligent content and query processing. *International Journal of Management, Technology, and Social Sciences*, 34–43. <https://doi.org/10.47992/IJMTS.2581.6012.0360>
 86. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y.,

- Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton, C. C., Drozdov, G., Fast, R., Fu, X., Fu, G., Gao, Y., ... Hospedales, T. (2023). *Llama 2: Open foundation and fine-tuned chat models*. arXiv. <https://doi.org/10.48550/ARXIV.2307.09288>
87. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention is all you need*. arXiv. <https://doi.org/10.48550/ARXIV.1706.03762>
 88. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Liang, P., & Potts, C. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638
 89. Shi, F., Chen, X., Misra, K., Scales, N., Tan, D., Zhou, T., Cheung, D., & Zhou, D. (2023). *Large language models can be easily distracted by irrelevant context*. arXiv. <https://doi.org/10.48550/ARXIV.2302.00093>
 90. Packer, C., Fang, V., Patil, S. G., Wang, K., Lin, K., Wooders, S., & Joseph, J. E. (2023). *MemGPT: Towards LLMs as operating systems*. arXiv. <https://doi.org/10.48550/ARXIV.2310.08560>
 91. Edge, D., Trinh, H., Cheng, X., Bradley, J., Chao, A., Mody, A., Truitt, S., & Larson, J. (2024). *From local to global: A graph RAG approach to query-focused summarization*. arXiv. <https://doi.org/10.48550/ARXIV.2404.16130>
 92. Lewis, P., Perez, E., Piktus, A., Petroni, F., Lewis, P., Riedel, S., & Kiela, D. (2020). *Retrieval-augmented generation for knowledge-intensive NLP tasks*. arXiv. <https://doi.org/10.48550/ARXIV.2005.11401>
 93. Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M.-W. (2020). *REALM: Retrieval-augmented language model pre-training*. arXiv. <https://doi.org/10.48550/ARXIV.2002.08909>
 94. Izacard, G., & Grave, E. (2021). Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for*

- Computational Linguistics: Main Volume* (pp. 874–880). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2021.eacl-main.74>
95. Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2), 115–152. <https://doi.org/10.1017/S0269888900008122>
 96. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). *ReAct: Synergizing reasoning and acting in language models*. arXiv. <https://doi.org/10.48550/ARXIV.2210.03629>
 97. Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). *Reflexion: Language agents with verbal reinforcement learning*. arXiv. <https://doi.org/10.48550/ARXIV.2303.11366>
 98. Firat, M., & Kuleli, S. (2024). What if GPT4 became autonomous: The Auto-GPT project and use cases. *Journal of Emerging Computer Technologies*, 3(1), 1–6. <https://doi.org/10.57020/ject.1297961>
 99. Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Bisk, E., Fried, D., Alon, U., & Neubig, G. (2023). *WebArena: A realistic web environment for building autonomous agents*. arXiv. <https://doi.org/10.48550/ARXIV.2307.13854>
 100. Zhuge, M., Liu, H., Fard, N. W., Cobbe, K., Kuefle, U., Copeland, C., & Ghanem, B. (2023). *Mindstorms in natural language-based societies of mind*. arXiv. <https://doi.org/10.48550/ARXIV.2305.17066>
 101. Dong, Y., Jiang, X., Jin, Z., & Li, G. (2024). Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33(7), 1–38. <https://doi.org/10.1145/3672459>
 102. Liu, Z., Zhang, Y., Li, P., Liu, Y., & Yang, D. (2023). *A dynamic LLM-powered agent network for task-oriented agent collaboration*. arXiv. <https://doi.org/10.48550/ARXIV.2310.02170>
 103. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLoRA: Efficient finetuning of quantized LLMs*. arXiv. <https://doi.org/10.48550/ARXIV.2305.14314>

104. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-rank adaptation of large language models*. arXiv. <https://doi.org/10.48550/ARXIV.2106.09685>
105. Wang, Z., Mao, S., Wu, W., Ge, T., Wei, F., & Ji, H. (2023). *Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration*. arXiv. <https://doi.org/10.48550/ARXIV.2307.05300>
106. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Xia, K., Press, O., & Narasimhan, K. (2023). *SWE-bench: Can language models resolve real-world GitHub issues?* arXiv. <https://doi.org/10.48550/ARXIV.2310.06770>
107. Ouyang, L., Lowe, R., Williams, J., OpenAI, Mishkin, P., Clever, J., Tejada, C., Almeida, D., Wainwright, C., Galke, L., Schulman, J., Hilton, J., Fraser, F., Ray, A., Ryu, J., Wollman, K., ... Christiano, P. (2022). *Training language models to follow instructions with human feedback*. arXiv. <https://doi.org/10.48550/ARXIV.2203.02155>
108. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12), 1–38. <https://doi.org/10.1145/3571730>
109. Manakul, P., Liusie, A., & Gales, M. (2023). SelfCheckGPT: Zero-resource black-box hallucination detection for generative large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 9004–9017). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2023.emnlp-main.557>
110. Yao, Y., Gao, P., Zhang, J., Wang, X., Zhang, S., Tian, Z., Zhou, J., & Jing, J. (2023). Editing large language models: Problems, methods, and opportunities. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 10222–10240). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2023.emnlp-main.632>

111. Tambon, F., Dakhel, A. M., Nikanjam, A., Khomh, F., Desmarais, M. C., & Antoniol, G. (2024). *Bugs in large language models generated code: An empirical study*. arXiv. <https://doi.org/10.48550/ARXIV.2403.08937>
112. Yang, Z., Sun, Z., Yue, T. Z., Devanbu, P., & Lo, D. (2024). *Robustness, security, privacy, explainability, efficiency, and usability of large language models for code*. arXiv. <https://doi.org/10.48550/ARXIV.2403.07506>
113. Semenov, O., & Chychkarov, Y. (2026). Automated code generation and testing for solving machine learning problems using multi-agent systems with LLM-kernel. *Telecommunications and Information Technology*, 90(1), 107–119. <https://doi.org/10.31673/2412-4338.2026.019011>
114. Zhang, Q., Lu, C., Liu, M., Zhu, C., Welbl, J., & Zhang, L. (2023). *A survey on large language models for software engineering*. arXiv. <https://doi.org/10.48550/ARXIV.2312.15223>
115. Celik, A., & Mahmoud, Q. H. (2025). A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7(3), Article 97. <https://doi.org/10.3390/make7030097>
116. Choi, W. C., & Chang, C. I. (2025). Enhancing Python programming through ChatGPT (GPT-5) study mode as a learning tutor: A study on learning achievement, motivation, and self-regulated learning in K-12 serious game (CodeCombat) programming education. *Computer Science and Mathematics*. <https://doi.org/10.20944/preprints202508.1722.v1>
117. Fernandes, H., Silva, J., Castor, F., & Pinto, G. (2025). A comparative study of LLMs for Gherkin generation. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025)* (pp. 171–181). Sociedade Brasileira de Computação. <https://doi.org/10.5753/sbes.2025.9888>
118. Steenhoek, B., Tufano, M., Sundaresan, N., & Svyatkovskiy, A. (2025). Reinforcement learning from automatic feedback for high-quality unit test generation. In *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)* (pp. 37–44). IEEE. <https://doi.org/10.1109/DeepTest66595.2025.00011>

119. Rehan, S., Al-Bander, B., & Al-Said Ahmad, A. (2025). Harnessing large language models for automated software testing: A leap towards scalable test case generation. *Electronics*, 14(7), Article 1463. <https://doi.org/10.3390/electronics14071463>
120. Cavalcanti, A. R., Mendes, E., & Andrade, N. (2025). Automating test design using LLM: Results from an empirical study on the public sector. *Conference on Digital Government Research*, 1. <https://doi.org/10.59490/dgo.2025.1025>
121. Liu, Z., Yang, Z., & Liao, Q. (2024). Exploration on prompting LLM with code-specific information for vulnerability detection. In *2024 IEEE International Conference on Software Services Engineering (SSE)* (pp. 273–281). IEEE. <https://doi.org/10.1109/SSE62657.2024.00049>
122. Raja, R., Vats, A., & Roy, S. (2025). Aligning prompts with ranking goals: A technical review of prompt engineering for LLM-based recommendations. *Computer Science and Mathematics*. <https://doi.org/10.20944/preprints202509.1959.v1>
123. Lemos, F., Alves, V., & Ferraz, F. (2025). *Is it time to treat prompts as code? A multi-use case study for prompt optimization using DSPy*. arXiv. <https://doi.org/10.48550/ARXIV.2507.03620>
124. Abbassi, A. A., Da Silva, L., Nikanjam, A., & Khomh, F. (2025). *ReCatcher: Towards LLMs regression testing for code generation*. arXiv. <https://doi.org/10.48550/ARXIV.2507.19390>
125. Semenov, O., & Chychkarov, Y. (2025). Automated generation of software test cases by means of large language models using prompt engineering. *Telecommunications and Information Technology*, 89(4), 122–129. <https://doi.org/10.31673/2412-4338.2025.048914>
126. Semenov, O., & Chychkarov, Y. (2025). Multi-agent methodology for automated development and testing of hybrid computer vision systems based on CNN and MLLM. *Science and Technology Today*, 58(4), 4709–4725.
127. Chychkarov, Y. A., & Semenov, O. V. (2026). Image recognition using a

hybrid approach based on multimodal large language models and convolutional neural networks. *Reporter of the Priazovskyi State Technical University. Section: Technical Sciences*, 1(53), 85–91. <https://doi.org/10.31498/2225-6733.53.1.2026.359780>

128. Huang, D., Zhang, J. M., Luck, M., Bu, Q., Qing, Y., & Cui, H. (2023). *AgentCoder: Multi-agent-based code generation with iterative testing and optimisation*. arXiv. <https://doi.org/10.48550/ARXIV.2312.13010>
129. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778). IEEE. <https://doi.org/10.1109/CVPR.2016.90>
130. Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 328–339). Association for Computational Linguistics. <https://doi.org/10.18653/v1/P18-1031>
131. Nashid, N., Sintaha, M., & Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2450–2462). IEEE. <https://doi.org/10.1109/ICSE48619.2023.00205>
132. Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., ... Liang, P. (2021). *On the opportunities and risks of foundation models*. arXiv. <https://doi.org/10.48550/ARXIV.2108.07258>
133. Hamidieh, K. (2018). *Superconductivity data*. UCI Machine Learning Repository. <https://doi.org/10.24432/C53P47>
134. Hamidieh, K. (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>

135. Trirat, P., Jeong, W., & Hwang, S. J. (2024). *AutoML-Agent: A multi-agent LLM framework for full-pipeline AutoML*. arXiv. <https://doi.org/10.48550/ARXIV.2410.02958>
136. Semenov, O. V., Serykh, S. O., Vasylenko, V. V., & Hnidenko, M. P. (2023). Modyfikatsiia modeli reputatsii ta doviry v zadachakh informatsiinoi bezpeky GRID-system dlia stiikosti do zahrozy "zlovmysni hrupy khostiv". *Scientific Notes of the State University of Telecommunications*, (3), 46–56. <https://doi.org/10.31673/2786-8362.2023.010505>

Акти впровадження

ТОВАРИСТВО З ОБМЕЖЕНОЮ ВІДПОВІДАЛЬНІСТЮ «АЙТІ КУРСОР»

Ідентифікаційний код за ЄДРПОУ: 40123706. Юридична адреса: 03124, м. Київ, бульвар Вацлава Гавела, 8. Фактична адреса: 03124, м. Київ, бульвар Вацлава Гавела, 8. Поточний рахунок: UA74 305299 00000 26007025022694 в ПАТ КБ «ПРИВАТБАНК» МФО: 305299
<http://www.itcursor.com.ua> E-mail: info@itcursor.com.ua тел\факс +38(044) 501-12-03



ЗАТВЕРДЖЕНО

« 19 » лютого 2026 р

АКТ

впровадження результатів дисертаційної роботи аспіранта
 Державного університету інформаційно-комунікаційних технологій
 Семенова Олександра Віталійовича

Комісія у складі голови - директор, Сенчіло Ярослава В'ячеславович та членів комісії, головний інженер Сєдих Денис Сергійович і Остапчук Валентин Віталійович, цим Актом цим Актом засвідчує, що результати дисертаційного дослідження аспіранта кафедри комп'ютерних наук Державного університету інформаційно-комунікаційних технологій Семенова Олександра Віталійовича на тему: «Метод побудови архітектури автоматизованого тестування на основі комбінованих парадигм тестування», поданої на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки», впроваджено у виробничий процес на ТОВ «Айті Курсор». При цьому використано програмний інструментарій та методики автоматизованої генерації, тестування та валідації програмного забезпечення для задач машинного навчання та машинного зору, що побудовані на базі шестишарової мультиагентної архітектури та великих мовних моделей, а саме:

- механізм ітеративної самокорекції коду, що базується на замкненому циклі зворотного зв'язку між агентами генерації та тестування;
- методологія побудови гібридних систем розпізнавання образів, яка інтегрує згорткові мережі з мультимодальними моделями для підвищення точності класифікації;
- стратегії формування запитів (зокрема на основі розробки через тестування) для автоматизації створення перевірочних сценаріїв.

Застосування розробленої інформаційної технології дозволило компанії досягти наступних показників:

- забезпечено зростання рівня генерації тестових сценаріїв до 80–90%, що дозволило виявляти до 90% синтаксичних та логічних дефектів ще на етапі написання коду;
- забезпечено скорочення загального часу на перевірку якості та виведення продуктів на ринок на 45–50%;
- підвищено точність класифікації об'єктів у складних умовах у проектах із машинного зору на 12–15%.

Комісія вважає, що розроблена інформаційна технологія відповідає сучасним вимогам надійності та відтворюваності інтелектуального програмного забезпечення. Її використання дозволяє забезпечити повну конфіденційність даних через можливість локального розгортання та значно підвищити продуктивність команд розробників.

Голова комісії

Сенчіло Я.В.

Члени комісії

Остапчук В.В.

Сєдих Д.С.





ЗАТВЕРДЖУЮ

Перший проректор Державного
університету інформаційно-
комунікаційних технологій

Олександр КОРЧЕНКО

03 2026р.

АКТ

використання у навчальному процесі Навчально-наукового інституту інформаційних технологій результатів дисертаційної роботи аспіранта кафедри комп'ютерних наук Державного університету інформаційно-комунікаційних технологій Семенова Олександра Віталійовича на тему: «Метод побудови архітектури автоматизованого тестування на основі комбінованих парадигм та штучного інтелекту» на здобуття наукового ступеня доктора філософії за спеціальністю 122 – Комп'ютерні науки

Комісія у складі: голова – директор Навчально-наукового інституту інформаційних технологій доктор технічних наук, професор Нестеренко Катерина Сергіївна, члени комісії – завідувач кафедри комп'ютерних наук доктор технічних наук, професор Вишнівський Віктор Вікторович, завідувачка кафедри штучного інтелекту, доктор технічних наук, професор Зінченко Ольга Валеріївна розглянули дисертаційну роботу Семенова Олександра Віталійовича на тему: «Метод побудови архітектури автоматизованого тестування на основі комбінованих парадигм та штучного інтелекту» та публікації автора за матеріалами дисертаційної роботи. Результати впроваджено в початковий процес Навчально-наукового інституту інформаційних технологій, а саме:

- методологія мультиагентної взаємодії для автоматизації життєвого циклу розробки інтелектуального програмного забезпечення;
- стратегії формування запитів для генерації програмного коду та тестових сценаріїв;
- механізми ітеративної самокорекції коду на основі зворотного зв'язку між спеціалізованими агентами;
- підходи до побудови гібридних систем розпізнавання образів.

На основі аналізу представлених матеріалів комісія встановила.

Результати дослідження використано в навчальному процесі Державного університету інформаційно-комунікаційних технологій при оновленні робочих програм навчальних дисциплін та підготовці методичного забезпечення кафедри комп'ютерних наук та штучного інтелекту у наступний спосіб:

інтегровано теоретичні положення щодо функціонування мультіагентних систем, архітектур великих мовних моделей та методів інтелектуального аналізу даних у зміст лекційних курсів навчальних дисциплін: «Мови програмування інтелектуальних систем», «Методи та засоби штучного інтелекту» ОКР «Бакалавр» спеціальності Комп'ютерні науки;

- впроваджено у лабораторні практикуми серію практичних завдань, спрямованих на опанування розробки систем самокорекції коду та налаштування локальної інфраструктури для роботи з відкритими інтелектуальними моделями в ізольованих середовищах навчальних дисциплін: «Сучасні технології програмування в системах зі штучним інтелектом», «Сучасні технології розпізнавання образів і обробки зображень» ОКР «Магістр» спеціальності Комп'ютерні науки;

- використано методичні підходи до поєднання згорткових нейронних мереж із мультимодальними моделями при розв'язанні задач комп'ютерного зору в межах курсового та дипломного проектування спеціальності Комп'ютерні науки.

Голова комісії

Директор ННІТ
доктор технічних наук, професор



Катерина НЕСТЕРЕНКО

Члени комісії

Завідувач кафедри комп'ютерних наук,
доктор технічних наук, професор



Віктор ВИШНІВСЬКИЙ

Завідувачка кафедри штучного інтелекту,
доктор технічних наук, професор



Ольга ЗІНЧЕНКО