МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Державне некомерційне підприємство

«Державний університет «Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій

Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувачка кафедри

_____ Олена ГРІНЕНКО

«___» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)
ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Методика використання великих мовних моделей для генерації модульних тестів на основі вихідного коду програми

Виконавець: Мельник Тарас Васильович

Керівник: к.ф.-м.н., доцент Татаринов Євген Олександрович

Нормоконтролер: ас. Андреєва Тетяна Василівна

Київ, 2025

Державне некомерційне підприємство
«Державний університет» Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій
Кафедра інженерії програмного забезпечення
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувачка кафедри
_____ Олена ГРІНЕНКО
«___» _____ 2025 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи студента
Мельника Тараса Васильовича

1. Тема кваліфікаційної роботи: «Методика використання великих мовних моделей для генерації модульних тестів на основі вихідного коду програми» затверджена наказом президента від 17 листопада 2025 р. №2450/ст.

2. Термін виконання проєкту: з 29.09.2025 р. по 21.12.2025 р.

3. Вихідні дані до роботи: результати дослідження використання великих мовних моделей для генерації модульних тестів на мові програмування C#.

4. Зміст пояснювальної записки:
1) Аналіз предметної області.
2) Дослідження генерації модульних тестів великими мовними моделями.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1) Таблиця із порівняльним аналізом великих мовних моделей для відбору в участі у дослідженні.

2) Таблиця зі зведеними результатами дослідження.

3) Результати дослідження у вигляді гістограм.

6. Календарний план-графік

№	Завдання	Термін виконання	Відмітка про виконання
1	Розробка та затвердження графіка роботи	29.09.2025–01.10.2025	Виконано
2	Ознайомлення з постановкою задачі, вивчення інформаційних джерел та складання плану роботи	02.10.2025–12.10.2025	Виконано
3	Підготовка 1 розділу та подання його керівнику	13.10.2025–02.11.2025	Виконано
4	Підготовка 2 розділу та подання його керівнику	03.11.2025–23.11.2025	Виконано
5	Загальне редагування пояснювальної записки, графічного матеріалу	24.11.2025–30.11.2025	Виконано
6	Представлення роботи для перевірки на академічну доброчесність, проходження нормоконтролю	01.12.2025–07.12.2025	Виконано
7	Отримання відгуку керівника, підготовка презентації та тексту доповіді	08.12.2025–14.12.2025	Виконано
8	Попередній захист	08.12.2025–14.12.2025	Виконано
9	Рецензування кваліфікаційної роботи	15.12.2025–22.12.2025	Виконано
10	Здача секретарю екзаменаційної комісії пояснювальної записки, електронної версії кваліфікаційної роботи, презентації доповіді, відгуку керівника, рецензії, результату проходження перевірки на плагіат, довідки про успішність, декларації про академічну доброчесність	15.12.2025–22.12.2025	Виконано
11	Захист кваліфікаційної роботи перед екзаменаційною комісією	26.12.2025	Виконано

Дата видачі завдання 29.09.2025 р.

Керівник кваліфікаційної роботи: к.ф.-м.н., доцент Євген ТАТАРИНОВ

Завдання прийняв до виконання: Тарас МЕЛЬНИК

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи "Методика використання великих мовних моделей для генерації модульних тестів на основі вихідного коду програми": 121 сторінка, 11 рисунків, 1 таблиця, 148 використаних джерел, 4 додатки.

Об'єкт дослідження – процес забезпечення якості програмного забезпечення шляхом автоматизації створення модульних тестів.

Мета кваліфікаційної роботи – дослідження ефективності застосування сучасних великих мовних моделей для автоматизованої генерації модульних тестів на основі вихідного коду, написаного мовою C#, порівняння їх якісних та кількісних характеристик та визначення оптимальних моделей і підходів для практичного впровадження у процес розробки програмного забезпечення.

Методи дослідження – аналіз науково-технічної літератури та джерел, порівняльний аналіз, експериментальний метод, методи програмної інженерії, мутаційне тестування, структурний аналіз коду, метод експертних оцінок із залученням ШІ-агентів, статистичні методи.

Результати роботи можуть бути використані для обґрунтованого вибору та впровадження LLM у промислові процеси автоматизованої генерації модульних тестів для C#/.NET з урахуванням якості, вартості й конфіденційності; для формування відтворюваної бази для подальших наукових досліджень завдяки наданим артефактам проведеного експерименту; для проєктування спеціалізованих мульти-агентних інструментів, що поєднують генерацію тестів, автоматичне вимірювання метрик та ітеративне поліпшення результату.

Розробка та дослідження проводилися під управлінням ОС Windows 11. Розробка програмного засобу для автоматизації експерименту та формування набору даних здійснювалися у середовищі JetBrains Rider на мові програмування C#.

ВЕЛИКА МОВНА МОДЕЛЬ, МОДУЛЬНЕ ТЕСТУВАННЯ, C#, .NET, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ДОСЛІДЖЕННЯ, PROMPT-ІНЖЕНЕРІЯ, БЕНЧМАРК, ГЕНЕРАЦІЯ МОДУЛЬНИХ ТЕСТІВ, МУТАЦІЙНЕ ТЕСТУВАННЯ.

ABSTRACT

Explanatory note to the qualification thesis "Methodology for using large language models to generate unit tests based on program source code": 121 pages, 11 figures, 1 table, 148 references, 4 appendices.

The object of the study is the software quality assurance process through the automation of unit test creation.

The purpose of the qualification thesis is to investigate the effectiveness of applying modern large language models to the automated generation of unit tests based on source code written in C#, to compare their qualitative and quantitative characteristics, and to determine the optimal models and approaches for practical integration into the software development process.

Research methods include the analysis of scientific and technical literature and sources, comparative analysis, experimental method, software engineering methods, mutation testing, structural code analysis, an expert assessment method involving AI agents, and statistical methods.

The results of the work can be used for a well-grounded selection and adoption of LLMs in industrial processes of automated unit test generation for C#/.NET, taking into account quality, cost, and confidentiality; for building a reproducible foundation for further scientific research thanks to the provided artifacts of the conducted experiment; for designing specialized multi-agent tools that combine test generation, automated metric measurement, and iterative improvement of the outcome.

Development and research were carried out under Windows 11. The development of the software tool for experiment automation and dataset formation was performed in JetBrains Rider using the C# programming language.

LARGE LANGUAGE MODEL, UNIT TESTING, C#, .NET, SOFTWARE QUALITY, RESEARCH, PROMPT ENGINEERING, BENCHMARK, UNIT TEST GENERATION, MUTATION TESTING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ПЗ - програмне забезпечення.

ШІ - штучний інтелект.

Модульне тестування (unit testing, юніт-тестування) – вид тестування, під час якого перевіряють коректність роботи окремих найменших частин програми (методів, функцій, класів) ізольовано від інших компонентів.

Модульний тест (юніт-тест) – автоматизований тест, що перевіряє очікувану поведінку конкретного програмного компонента (метода, функції, властивості) для заданих вхідних даних.

Велика мовна модель (Large Language Model, LLM, модель) – модель штучного інтелекту, навчена на великих обсягах текстових даних, яка здатна генерувати та аналізувати текст.

Пропрієтарна модель (proprietary model, закрита модель) – модель, доступ до якої обмежений власником: зазвичай користувачі мають доступ через сервіс/API, але ваги, навчальні дані та внутрішні деталі не є публічними.

Відкрита модель (open-weight model, модель з відкритими вагами) – модель, для якої опубліковані ваги, що дозволяє запускати її локально на власній інфраструктурі.

Промпт (prompt) – вхідний текст/інструкція, що подається моделі для отримання відповіді або виконання задачі.

Промпт-інженерія (prompt-інженерія) – підбір і структуризація промптів (інструкцій, прикладів, обмежень, формату відповіді) для підвищення якості та керованості результатів моделі.

Бенчмарк – стандартизований набір задач, метрик і правил оцінювання, призначений для порівняння якості або продуктивності моделей.

Датасет – впорядкований набір даних (наприклад, кодові фрагменти, тексти, мітки), який використовується для навчання, донавчання або оцінювання моделей.

Токен – мінімальна умовна одиниця тексту, на яку модель розбиває ввід та вивід під час обробки (може бути словом, частиною слова, символом тощо).

Донавчання (fine-tuning) – додаткове навчання вже попередньо натренованої моделі на спеціалізованому наборі даних для адаптації під конкретну задачу або домен.

Агент (agent, ШІ-агент) – система на базі ШІ, яка може планувати кроки та виконувати послідовність дій для досягнення мети (інколи із використанням інструментів, API, пам’яті тощо).

API (Application Programming Interface) – інтерфейс взаємодії між програмами, який визначає способи виклику функцій/сервісів і формат обміну даними.

Мутаційне тестування (mutation testing) – метод оцінювання якості тестів, коли у вихідний код вносять невеликі штучні помилки (мутації) і перевіряють, чи "ловлять" їх тести (тобто чи падають), тим самим визначаючи мутаційну оцінку.

Асерт (assert) – перевірка в тесті, що фактичний результат відповідає очікуваному; у разі невідповідності тест завершується з помилкою.

ВСТУП

Сучасна індустрія розробки програмного забезпечення перебуває на етапі стрімкої трансформації, що характеризується постійним зростанням складності програмних систем та підвищенням вимог до швидкості випуску продуктів на ринок. У цьому контексті забезпечення надійності та якості коду стає критично важливим завданням, фундаментом якого є тестування. Особливе місце у стратегії забезпечення якості займає модульне тестування – підхід, що передбачає ізольовану перевірку окремих компонентів системи, таких як функції або класи. Відповідно до концепції піраміди тестування, запропонованої Майком Коуном, саме модульні тести мають складати найширший базовий шар перевірок, оскільки вони є найшвидшими у виконанні та дозволяють отримувати миттєвий зворотний зв’язок щодо коректності коду [119, 121].

Попри беззаперечну цінність модульного тестування, його практична реалізація супроводжується значними витратами ресурсів. Процес написання якісних тестів, що відповідають принципам FIRST (Fast, Independent, Repeatable, Self-Validating, Timely) [41], вимагає від розробників значних зусиль та високої кваліфікації. За наявними даними, інженери можуть витрачати понад 15% свого робочого часу виключно на створення та підтримку модульних тестів [61]. Ця діяльність часто є монотонною та трудомісткою, що нерідко призводить до зниження мотивації, неповного покриття коду тестами та появи так званих "запахів коду тестів" (test smells) – проблем, що ускладнюють подальший супровід системи [111].

Спроби автоматизувати процес написання тестів здійснювалися протягом останніх десятиліть. Традиційні інструменти, такі як EvoSuite [39, 40], що базуються на еволюційних алгоритмах та методах пошуку, здатні генерувати набори тестів із високим рівнем покриття коду (до 80%) [79]. Однак, як свідчить аналіз предметної області, такі тести часто страждають на низьку читабельність, містять складну для сприйняття людиною логіку та позбавлені семантичного зв’язку з бізнес-вимогами [79]. У реальній практиці це робить їх малопридатними для довгострокової

підтримки, перетворюючи згенерований код тестів на "мертвий вантаж", якому розробники не довіряють.

Якісний стрибок у вирішенні цієї проблеми став можливим завдяки стрімкому розвитку технологій штучного інтелекту, зокрема появі великих мовних моделей [131]. Сучасні моделі, такі як GPT-5.1 [85], Gemini 3 Pro [46], Claude Sonnet 4.5 [56], що навчені на великих масивах вихідного коду, демонструють здатність не лише синтаксично правильно відтворювати конструкції мов програмування, але й "розуміти" контекст, семантику та наміри розробника. Це відкриває принципово нові можливості для автоматичної генерації тестів, які за своєю структурою, іменуванням змінних та логікою перевірок наближаються до коду, написаного людиною.

Проте інтеграція LLM у процес розробки ПЗ породжує низку нових викликів, які потребують детального вивчення. По-перше, стохастична природа мовних моделей призводить до ризику генерації некоректного коду або виникнення "галюцинацій" [66] – посилань на неіснуючі бібліотеки чи методи, що створює загрози для безпеки ланцюга постачання ПЗ [93, 5, 120]. По-друге, використання хмарних LLM-сервісів актуалізує питання конфіденційності та захисту інтелектуальної власності, оскільки передача вихідного коду третій стороні може порушувати умови угод про нерозголошення (NDA) [5, 13, 128]. По-третє, існує потреба в об’єктивній оцінці якості згенерованих тестів, яка б виходила за межі примітивних метрик покриття і враховувала реальну здатність тестів виявляти дефекти та їхню читабельність для людини.

Враховуючи динамічну появу нових моделей у 2024–2025 роках та відсутність вичерпних порівняльних досліджень їхньої ефективності саме для екосистеми .NET (C#), тема роботи є надзвичайно актуальною. Необхідність систематичного аналізу можливостей сучасних пропрієтарних та відкритих LLM, розробки ефективних стратегій prompt-інженерії [138] та оцінювання згенерованих рішень за комплексом якісних та кількісних показників зумовлює доцільність даного дослідження для розвитку галузі програмної інженерії.

Метою роботи являється дослідження ефективності застосування сучасних великих мовних моделей для автоматизованої генерації модульних тестів на основі вихідного коду, написаного мовою C#, порівняти їх якісні та кількісні характеристики та визначити оптимальні моделі й підходи для практичного впровадження у процес розробки програмного забезпечення.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

1) Проаналізувати сучасний стан предметної області, зокрема підходи до модульного тестування, наявні методи автоматизації та особливості застосування великих мовних моделей у програмній інженерії.

2) Дослідити питання безпеки та конфіденційності даних при використанні LLM, виявити ризики витоку інтелектуальної власності через хмарні сервіси та загрози, пов’язані з вразливостями у згенерованому коді.

3) Обґрунтувати вибір репрезентативного набору великих мовних моделей для експериментального дослідження, що включає як провідні пропрієтарні рішення, так і сучасні відкриті моделі, доступні через уніфікований інтерфейс OpenRouter [90].

4) Створити промпти для генерації модульних тестів та визначення індексу читабельності із використанням найкращих практик prompt-інженерії, які були попередньо проаналізовані та засвоєні, зокрема із застосуванням технік zero-shot, system, contextual та role prompting.

5) Сформувати спеціалізований датасет, що містить різнопланові фрагменти вихідного коду на мові програмування C# (від простих класів із допоміжними методами до складних алгоритмів та бізнес-логіки), який забезпечить репрезентативність експерименту.

6) Розробити та реалізувати програмний засіб для автоматизації експерименту, який забезпечить взаємодію з моделями через OpenRouter API [91], пакетну обробку запитів, автоматизований збір метрик та визначення індексу читабельності.

7) Провести експериментальне дослідження генерації тестів обраними моделями та виконати комплексний порівняльний аналіз отриманих результатів за

показниками успішності компіляції, валідності, покриття коду, мутаційної оцінки, читабельності та економічної ефективності.

Об'єктом дослідження є процес забезпечення якості програмного забезпечення шляхом автоматизації створення модульних тестів.

Предметом дослідження є методи, інструменти та метрики якості генерації модульних тестів для програмного забезпечення на мові C# із використанням сучасних великих мовних моделей.

Для розв'язання поставлених завдань у роботі використано комплекс взаємодоповнювальних методів дослідження:

1) Аналіз науково-технічної літератури та джерел – для вивчення поточного стану розвитку LLM, методів prompt-інженерії, наявних підходів до оцінювання якості тестів та питань безпеки й конфіденційності.

2) Порівняльний аналіз – для відбору моделей-кандидатів на основі результатів авторитетних бенчмарків (SWE-Bench [108, 107], LiveCodeBench [15, 65], LiveBench [63], MMLU-Pro [17], HLE [51, 52, 14]) та для зіставлення результатів експерименту між різними моделями.

3) Експериментальний метод – для безпосереднього отримання емпіричних даних шляхом генерації тестових наборів для 20 прикладів коду за допомогою п'яти різних LLM через OpenRouter API [91].

4) Методи програмної інженерії – для розробки консольного застосунку автоматизації експерименту, формування датасету та інтеграції інструментів тестування (фреймворк xUnit [146], бібліотека NSubstitute [80]).

5) Мутаційне тестування – для об’єктивної оцінки здатності згенерованих тестів виявляти дефекти у коді; реалізовано за допомогою інструменту Stryker.NET [106].

6) Структурний аналіз коду (code coverage analysis) – для визначення повноти покриття інструкцій програми тестами; реалізовано за допомогою інструменту JetBrains dotCover [57].

7) Метод експертних оцінок із залученням ШІ-агентів – для визначення індексу читабельності згенерованого коду, де в ролі незалежних експертів виступали п’ять інших високопродуктивних мовних моделей.

8) Статистичні методи – для обробки, агрегації та усереднення отриманих кількісних показників (часу виконання, кількості токенів, вартості, значень метрик).

У роботі отримано нові наукові результати, які полягають у наступному:

1) Дослідження застосування генеративного штучного інтелекту в задачах верифікації ПЗ отримало подальший розвиток. На відміну від наявних робіт, проведено комплексний порівняльний аналіз найновіших моделей 2025 року саме в контексті екосистеми .NET/C#, що дозволило виявити залежність якості тестів від архітектури та типу розповсюдження моделі (пропрієтарна/відкрита).

2) Вперше застосовано комбіновану методику оцінювання якості автоматично згенерованих тестів, яка інтегрує об’єктивні динамічні метрики (мутаційна оцінка, покриття коду) із суб’єктивною метрикою читабельності, розрахованою автоматизовано за допомогою мульти-агентного підходу. Це дозволило отримати багатовимірну оцінку, що враховує не лише технічну коректність, але й придатність коду до супроводу людиною, що є критичним фактором для промислового застосування.

3) Отримано чітку об’єктивну оцінку можливостей сучасних великих мовних моделей у контексті генерації саме програмного коду модульних тестів. Завдяки проведеному дослідницькому експерименту було оцінено якість згенерованих тестів кожної моделі та, на основі отриманих показників, визначено фаворитів та їх сильні й слабкі сторони.

4) Розроблено спеціалізований програмний засіб (консольний застосунок) для автоматизації процесу дослідження, вихідний код якого доступний у відкритому репозиторії [67]. Інструмент реалізує взаємодію з OpenRouter API [91], забезпечує пакетну обробку запитів, збір метрик та збереження результатів. Це рішення може бути модифіковане та адаптоване для використання у подальших експериментах.

5) Сформовано та опубліковано відкритий датасет ("LlmUnitTestGenerationArtifacts.Dataset"), що містить 20 репрезентативних,

самодостатніх фрагментів коду на C# різного рівня складності (від простих допоміжних методів до складних алгоритмів та бізнес-сервісів) [67]. Цей набір даних може слугувати відправною точкою для подальших досліджень якості генерації модульних тестів великими мовними моделями.

Отримані в роботі результати мають безпосереднє практичне значення для інженерів з якості, розробників ПЗ та технічних керівників, які планують впровадження інструментів на базі ШІ у свої процеси:

1) Надано обґрунтовані рекомендації щодо вибору оптимальної моделі залежно від проєктних обмежень. Експериментально доведено, що для задач, де пріоритетом є максимальна якість та покриття, найкращим вибором є моделі Gemini 3 Pro [46] та GPT-5 mini [82]. Водночас для проєктів із суворими вимогами до конфіденційності даних або обмеженим бюджетом рекомендовано використання відкритої моделі DeepSeek R1 0528 [33], яка демонструє результати, співставні з комерційними лідерами, при можливості локального розгортання.

2) Систематизовано ризики безпеки, пов'язані з використанням LLM для генерації коду, зокрема виявлено проблему "галюцинацій" залежностей (slopsquatting) [120]. Сформульовані рекомендації щодо очищення коду та верифікації бібліотек дозволяють мінімізувати загрози безпеки при використанні ШІ-асистентів у розробці ПЗ.

3) Визначено найоптимальніші моделі та сформовано якісний детальний промпт для генерації модульних тестів на основі вихідного коду програми. Ці напрацювання можна використати для побудови прикладного інструменту для автоматизації процесу створення тестів.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Модульне тестування в розробці програмного забезпечення

Модульне тестування – це підхід до перевірки якості програмного забезпечення, коли тестуються окремі невеликі фрагменти програми (наприклад, окремі функції чи класи) ізольовано від інших частин системи. Мета модульного тестування – переконатися, що кожен компонент працює відповідно до вимог і задуму розробників. Завдяки тестуванню модулів можна виявляти дефекти на ранніх етапах розробки та виправляти їх до інтеграції компонентів, що економить час і зусилля на наступних етапах. Модульне тестування є базовим шаром піраміди автоматизованого тестування (рисунок 1.1) – концепції, запропонованої Майком Коуном, де найширший фундамент складають численні швидкі юніт-тести, а вище розташовані менші за кількістю інтеграційні та кінцеві тести [119, 121]. Такий розподіл означає, що більшість перевірок якості коду мають виконуватись саме на рівні модулів, оскільки ці тести найшвидші, найізольованіші і дають миттєвий зворотний зв’язок розробникам.

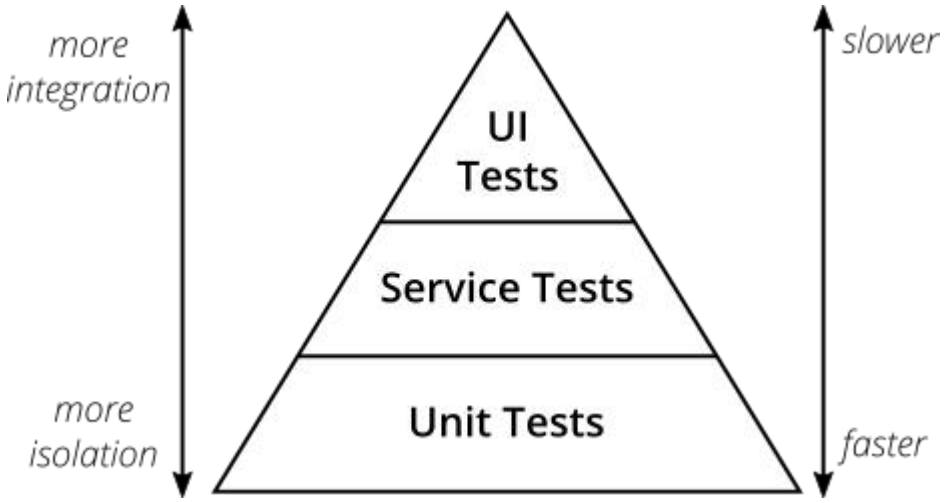

Рис. 1.1. Піраміда тестування

Важливість модульного тестування складно переоцінити. Воно вже стало стандартною практикою і навіть обов’язковою вимогою у багатьох проєктах, адже слугує наріжним каменем забезпечення якості і надійності ПЗ. Наявність якісного набору юніт-тестів дає впевненість у правильній роботі компонентів та спрощує подальшу підтримку коду [116]. У контексті гнучких методологій та CI/CD, автоматизовані модульні тести прискорюють цикл розробки: швидкий зворотний зв’язок дозволяє "ловити" помилки за секунди чи хвилини, а не витрачати дні на ручні перевірки. Завдяки цьому команди можуть вносити зміни в кодову базу значно впевненіше і швидше, знаючи, що тестовий набір одразу сповістить про ненавмисні помилки [126].

Попри очевидну користь, написання модульних тестів потребує значних зусиль. Розробники витрачають значну частину часу – за деякими даними, понад 15% свого робочого часу – саме на створення юніт-тестів [61]. Цей процес часто монотонний і трудомісткий: потрібно придумати вхідні дані, очікуваний результат, налаштувати оточення для кожного тесту. Через брак часу чи мотивації буває, що покриття тестами залишається неповним – деякі гілки коду не перевіряються взагалі, або ж тести не охоплюють крайні випадки. Інша проблема – супровід тестів: коли розвивається функціонал, доводиться відповідно змінювати і тестові сценарії. Недостатньо підтримувані або застарілі тести можуть заважати рефакторингу – розробники починають їх ігнорувати чи вимикати. З людського боку, існує ризик помилок при написанні тестів: тест може містити хибне припущення щодо поведінки коду і тим самим або не ловити баг, або навпаки завжди "червоніти" через невірний очікуваний результат. Всі ці фактори збільшують вартість модульного тестування і нерідко спонукають розробників шукати способи автоматизувати або спростити створення тестів.

Автоматизація генерації тестів має на меті зменшити ручну працю і покрити якомога більше коду корисними перевірками. Починаючи ще з 2000-х років науковці й інженери пропонували інструменти, що автоматично генерують тестові випадки на основі програмного коду. Зокрема, застосовувалися такі підходи: символічне виконання (аналіз шляхів виконання програм з символічними вхідними

значеннями), генетичні й еволюційні алгоритми пошуку тестових даних (еволюційне тестування), методи моделювання і перевірки моделей (model checking) тощо [72]. Один із найвідоміших інструментів, EvoSuite, розглядає генерування тестів як задачу пошуку і застосовує генетичний алгоритм для досягнення високого покриття коду тестами [39, 40]. Такі традиційні підходи в цілому здатні згенерувати великий набір тестів з досить високим відсотком покриття. Однак якість і корисність цих тестів нерідко поступається тестам, написаним вручну. Автоматично згенеровані тести часто страждають на так звані "запахи коду" – проблеми, що ускладнюють розуміння і підтримку тестів. Наприклад, інструменти еволюційного тестування можуть видавати тести з нечіткими іменами, загадковою логікою чи надмірною деталізацією, що робить їх важкими для сприйняття розробником. Дослідники відзначають, що хоча EvoSuite досягає приблизно 80% покриття, його тести вважаються малопридатними через погану читабельність та підтримуваність [79]. У реальній практиці розробникам складно безпосередньо використати такі тести – вони скоріше служать чернетками або додатковою перевіркою, ніж фінальним артефактом.

Таким чином, виникає потреба у нових підходах, які б поєднали високе покриття з якістю тестового коду, наближеною до людської. Важливу роль тут відіграють кращі практики модульного тестування, напрацьовані часом та спільнотою. Зокрема, якісний юніт-тест має бути Fast (швидким), Independent (незалежним від інших тестів), Repeatable (стабільно відтворювати результат), Self-Validating (автоматично перевіряти результат без ручного втручання) і Timely (написаним вчасно, паралельно з кодом) – ці п’ять принципів відомі як акронім FIRST [41]. Тест повинен мати чітку структуру Arrange-Act-Assert (підготовка даних, виконання дії, перевірка результату), щоб його легко було читати та розуміти [127]. Рекомендується приділяти увагу назвам тестів – вони мають ясно описувати, яку поведінку перевіряє сценарій. Якщо тести генеруються автоматично, бажано, щоб інструмент також генерував осмислені імена [125].

Також відомі й досить розповсюджені анти-патерни тестування, яких слід уникати. Один із них – "рулетка" тверджень (assertion roulette), коли в одному тесті

виконується багато перевірок підряд без належних повідомлень: у разі падіння тесту незрозуміло, яке саме твердження не пройшло [111]. Хорошою практикою є забезпечити один логічний асерт, тобто перевірку, на тест або принаймні додавати пояснення до кожного асерту, щоб у разі помилки тест повідомляв причину [105]. Інший анти-патерн – надмірно специфічні тести (over-specified tests): такі тести прив’язуються до деталей реалізації замість перевірки зовнішньої поведінки. Вони часто ламаються при найменшому рефакторингу коду, хоч з точки зору специфікації функція працює правильно. Автоматичні засоби іноді нехтують цим, генеруючи перевірки на внутрішні значення, що робить тести крихкими [92]. Нестабільні, або "флейкі" (англ. flaky), тести – ще одна проблема: це тести, що час від часу випадково падають або проходять при одному й тому ж коді [130]. Причини можуть бути в неконтрольованих факторах (наприклад, залежність від поточного часу, зовнішніх сервісів, випадкового порядку). Нестабільний тест підриває довіру до тестового набору: розробники починають ігнорувати червоні прогони, списуючи їх на випадковість [9]. За даними Google, близько 1,5% усіх прогонів тестів у їхній інфраструктурі дають "flaky"-результат, і це призводить до значних витрат часу на аналіз – у проєкті з 1000 тестів в середньому 15 можуть бути нестабільними і потребуватимуть перевірки [42]. Тому при генеруванні тестів варто уникати недетермінованості: наприклад, не використовувати в тесті випадкові числа чи прив’язку до зовнішнього середовища. Автоматично створені тести повинні дотримуватися принципів ізоляції: працювати лише з модулем, який перевіряється, використовуючи заглушки чи макети для залежностей, щоб їх результат не залежав від сторонніх систем.

Підсумовуючи, модульне тестування є критично важливою практикою для забезпечення якості ПЗ, але його ручне впровадження потребує значних ресурсів і дисципліни. Проблеми вартості написання, підтримки тестів та людського фактору створюють підґрунтя для автоматизації цього процесу. Завдання, яке розглядається у даній роботі, полягає в автоматичній генерації коректних та корисних модульних тестів з вихідного коду програми, що відповідають кращим практикам (мають

змістовні перевірки, читабельний код, стабільно проходять) і допомагають виявити потенційні дефекти.

1.2 Великі мовні моделі у контексті програмування

Останніми роками в галузі штучного інтелекту стрімко розвивається потужний інструмент – великі мовні моделі (англ. Large Language Models, LLM). Це нейронні моделі з надвеликою кількістю параметрів, що натреновані на масштабних корпусах тексту для передбачення продовження текстової послідовності [131]. Ключовою технологією, що лежить в основі LLM, є архітектура трансформерів [135]. Трансформер використовує механізм самоуваги (self-attention), який дозволяє моделі визначати, які слова (або токени) в тексті найбільш впливають одне на одне. Це вирішує проблему великих залежностей у тексті – модель здатна врахувати контекст із сотень і навіть тисяч слів. Крім того, трансформери оперують на рівні токенів – спеціальному закодованому представленні тексту. Вся вхідна програма або текст перетворюється на послідовність токенів і подається моделі. Важливою характеристикою є контекстне вікно – максимальна довжина послідовності токенів, з якою модель може працювати одночасно. Сучасні LLM здатні опрацьовувати контекст величиною у сотні тисяч токенів, що еквівалентно приблизно 40-70 сторінкам коду. На етапі навчання великих моделей широко використовується підхід попереднього навчання (pre-training) – модель проганяють через терабайти текстових даних, навчаючи прогнозувати наступне слово в реченні [95]. Завдяки цьому модель набуває загального "розуміння" мови і навіть певних знань. Після цього часто застосовують точне налаштування (fine-tuning) – додаткове навчання на спеціалізованих даних для виконання конкретних задач [95]. Окремо останнім часом практикується інструктивне навчання (instruct-tuning): модель донавчають реагувати на текстові інструкції людини, що дозволяє їй коректніше і слухняніше виконувати поставлені завдання [95]. Саме завдяки інструктивному навчанню моделі типу ChatGPT почали адекватно відповідати на запити, сформульовані природною мовою.

Генеруючи відповіді, LLM можуть використовувати різні режими детермінованості, контрольовані параметрами temperature і top-p [31, 124]. Параметр temperature визначає "креативність" чи випадковість вибору слів: при значенні 0 модель буде завжди обирати найбільш ймовірне продовження (відповіді будуть більш одноманітними), а при значеннях близько 1 – частіше ризикує і може пропонувати неочікувані варіанти. Параметр top-p ("ядерна вибірка") встановлює поріг ймовірності: модель вибирає наступне слово не з усіх можливих, а лише з деякого найімовірнішого набору, сумарна ймовірність якого – p. Комбінація цих параметрів дозволяє балансувати між різноманітністю і якістю генерованого коду.

LLM загального призначення – такі як GPT-5 [83], GPT-5.1 [85], Gemini 3 Pro [46] – тренуються на змішаних текстових даних: книгах, статтях, сайтах, що можуть містити і фрагменти програмного коду. Вони демонструють здатність до написання коду, але це не їхня основна спеціалізація. Натомість з’явилася окрема категорія – мовні моделі, орієнтовані на код. Це моделі, які або донавчалися спеціально на великих масивах вихідного коду, або від початку архітектурно оптимізовані для роботи з програмами. Прикладами є GPT-5 Codex [81], GPT-5.1 Codex [84], Claude Sonnet 4.5 [56] та інші. Такі моделі засвоїли синтаксис багатьох мов програмування і можуть продовжувати код за заданим початком, генерувати цілі функції за описом, виправляти помилки та навіть пояснювати код. Моделі на кшталт Codex лягли в основу практичних інструментів, як-от GitHub Copilot – "розумного" автодоповнення для IDE. Особливість моделей, навчених на коді, – вміння враховувати структуру програм: відступи, дужки, ключові слова, назви змінних тощо. Вони можуть продукувати завершені фрагменти коду, синтаксично правильні у переважній більшості випадків. До можливостей таких LLM належать: генерація коду за описом (наприклад, створення функції за коментарем), автодоповнення під час написання коду (продовження рядка чи блоку, коли програміст призупинив ввід), рефакторинг (переписування коду з покращенням, як-от перейменувати змінні, розбити функцію на менші), статичний аналіз (виявлення потенційних багів чи вразливостей шляхом аналізу коду як тексту), генерація документації (опис функції на основі її коду) та, звісно, генерація тестів. Важливо розуміти, що LLM не

компілюють код і не виконують його при генерації – вони спираються лише на статистичні закономірності, вивчені з даних. Тому їхні можливості обмежені тим, що вони "зустріли" під час навчання, але обсяг навчальних даних наразі настільки великий, що часто модель "знає" типові патерни і рішення багатьох задач.

Для вирішення задач програмування, зокрема генерування тестів, важливу роль відіграє репрезентація коду у вхідних даних LLM. Оскільки модель оперує послідовністю токенів, їй можна подавати різні види інформації про програму. Найпростіше – вставити текст вихідного коду (наприклад, тіло функції) у промпт, тобто запит, до моделі. Проте якщо код дуже об’ємний, то можна надати лише стислі характеристики: назви методів (сигнатури), докладні коментарі, короткий опис функціональності. Деякі дослідники експериментують з передачею структурних уявлень: наприклад, перетворюють код у абстрактне синтаксичне дерево (Abstract Syntax Tree, AST) [3] чи граф потоку виконання (Control Flow Graph, CFG) [32] і стисло описують їх. Існують моделі, що спеціально враховують структурні зв’язки – наприклад, GraphCodeBERT будує граф зв’язків між токенами коду на основі AST, щоб краще розуміти контекст [47]. Але в загальному випадку сучасні LLM на вході приймають текст, тому навіть структуровані дані треба представити у текстовій формі (наприклад, список викликів функцій, сигнатур, JSON із описом). У задачі генерації тестів можна комбінувати декілька видів інформації: передати моделі і код функції, і короткий опис її призначення, і, можливо, специфікації чи приклади використання. Чим багатший контекст, тим більше шансів, що модель зрозуміє, які сценарії треба перевірити в тестах.

Процес взаємодії з LLM для отримання бажаного результату часто називають prompt-інженерією [138]. Для генерації тестів це особливо актуально: формулювання запиту значною мірою визначає, які тести будуть отримані. Виділяють різні техніки [97]. Zero-shot підхід – коли моделі просто надається інструкція на кшталт: "Створи модульні тести для наведеної функції", без жодних прикладів. Модель намагатиметься, спираючись на свій загальний досвід, одразу видати відповідні тести. Few-shot – коли у промпті наводяться приклади: наприклад, спочатку показуємо код функції A і відповідний добре написаний тест для неї, потім

просимо: "А тепер напиши тест для функції B". Модель, бачачи зразок, ймовірніше витримає такий самий стиль і формат. У задачі тестування часто застосовують техніку ланцюжка міркувань (Chain-of-Thought, CoT). Замість того, щоб модель одразу дала фінальний тест, її просять спочатку поміркувати: які можливі вхідні випадки, які граничні значення, що функція має робити в різних ситуаціях. Це ніби змушує модель виконати аналітичну роботу перед генерацією коду тестів. Іноді такий CoT-промпт покращує результати, хоча дослідження показують, що в генерації тестів ефективність CoT не гарантується і залежить від здатності моделі розуміти код. Ще одна тактика – надання шаблону тесту. Можна в промпті задати структуру: наприклад, написати заголовок тестового класу, імпорти, а модель повинна доповнити тіло тестових методів. Чіткі інструкції теж допомагають: можна вказати моделі, які саме аспекти перевіряти ("перевір і позитивні, і негативні випадки, включи перевірку винятку"), чи наголосити, щоб тест не залежав від зовнішнього стану ("не звертайся до файлової системи, натомість відповідні виклики заміни моками"). Оточення виконання теж варто врахувати: якщо модель цього явно не знає, вона може згенерувати тест, що звертається до бази даних, або викликає неіснуючий API. Тому часто в промпт додають обмеження: "тести запускаються в ізольованому середовищі, доступу до мережі немає, дані можна лише захардкодити". Такі підказки допомагають уникнути генерації непрактичних тестів.

Різноманіття технік породило класифікацію стратегій використання LLM для генерування коду і тестів. Найпростіший – prompt-only, або одноетапна генерація. Користувач один раз звертається до моделі: подає код і отримує тест без подальших уточнень. Більш просунуті – ітеративні підходи зі зворотним зв’язком. В них LLM використовується у циклі: спочатку модель створює тест, потім цей тест виконують (компілюють і запускають) на цільовому коді, а далі збирають інформацію про результати – наприклад, чи тест пройшов, або які рядки коду покриті. На основі цього моделі дають зворотний зв’язок і просять виправити або доповнити тести. Один з перших таких підходів реалізований в інструменті ChatUniTest – там використовується цикл "генерація-валідація-ремонт" за участю ChatGPT [26].

Модель генерує тест, потім автоматично виявляються помилки (синтаксичні чи падіння тестів), і після цього модель отримує повідомлення про ці помилки, щоб спробувати їх виправити. Це приклад динамічного зворотного зв'язку від реального виконання: модель крок за кроком доводить тест до робочого стану. Інший різновид зворотного зв'язку – керування покриттям. Якщо після першого набору тестів видно, що частина коду не виконується (низьке покриття), можна спонукати модель згенерувати додаткові тести саме для непокритих гілок. Дослідники з Майкрософту запропонували підхід CodaMosa [27]: інтегрувати LLM у цикл пошуку на основі покриття. Традиційний генетичний алгоритм швидко досягав певного рівня покриття, але не знаходив шлях до складних гілок; тоді в гру вступала LLM (Codex), яка намагалася згенерувати тест, враховуючи розуміння коду, щоб прорватися через ці гілки. Такий гібрид показав кращі результати за покриттям, ніж чисто пошуковий метод. Окремо варто згадати підхід з багатоагентною взаємодією (multi-agent): це коли декілька моделей або процесів діють спільно, обмінюючись результатами [134]. В контексті тестування можна уявити, що один агент (модель) генерує код, інший генерує тести до нього, третій аналізує результати, і так кілька раундів. Певною мірою, та ж схема генерація-валідація-ремонт є двоагентною: "генератор" – LLM, а "верифікатор" – середовище виконання коду, яке виявляє помилки. Є роботи, де застосовано дві різні моделі: одна пропонує рішення, інша перевіряє і критикує, а потім перша покращує свій варіант. У задачі тестів конкретні реалізації multi-agent підходу ще формуються, але концепція має потенціал підвищити якість за рахунок розподілу завдань між агентами. Ще одна перспективна схема – генерація з пошуком у зовнішніх джерелах (Retrieval-Augmented Generation, RAG) [140]. Моделі можуть підключатись до баз знань або документації: наприклад, при генеруванні тестів для певної бібліотеки модель могла б автоматично знайти в документації приклади використання функцій і на основі цього будувати тести. Ідея RAG – не обмежуватися лише тим, що в моделі "в голові", а довантажувати релевантну інформацію.

Окремо варто згадати можливості підсилення LLM інструментами (tool use) [132]. Сучасні великі моделі можна наділити здатністю виконувати певні дії,

викликати зовнішні сервіси, коли це потрібно. У нашій задачі такими інструментами є, перш за все, компіляція та запуск коду. Модель може сама згенерувати код і зупинитися, але якщо дати їй змогу дізнатися, що згенерований код не працює, то вона може покращити результат. Наприклад, в деяких роботах реалізоване автоматичне виконання згенерованого фрагмента: якщо він не компілюється або падає, помилка (текст помилки) додається до наступного промпта, і модель генерує виправлену версію. Такий цикл self-repair ("самовиправлення") дозволяє моделі поступово усунути синтаксичні помилки, несумісні типи, невизначені імена тощо, які могли виникнути через її "галюцинації". Інструментом зворотного зв’язку може бути і аналіз покриття коду: після запуску тестів збирається інформація про те, які рядки або гілки програми пройдені. Ці дані можна повернути моделі і попросити додати тести для непокритих ділянок. Ще один "інструмент" – Mutation Testing: спеціальна програма може внести дрібні помилки (мутанти) в код і прогнати тести; якщо тести не помітили підміну, це сигнал моделі, що потрібно покращити асерти [77, 78]. Робота MuTAP [36] застосувала мутаційний підхід у циклі генерації тестів і домоглася кращого виявлення помилок – тестові набори вийшли сильнішими, ніж при звичайному zero-shot або few-shot зверненні до моделі. Усі ці інструменти працюють як доповнення: LLM генерує гіпотези (код тестів), інструменти перевіряють або дають інформацію, LLM на основі цього генерує правки. Такий симбіоз дозволяє компенсувати слабкості моделі.

Необхідно розуміти, з якими викликами стикається людина, застосовуючи LLM для написання коду. По-перше, якість і правильність згенерованого коду не гарантована. Модель може робити помилки, бо не має справжнього "розуміння" виконання програми, а лише статистичні асоціації. Відоме явище – "галюцинації" моделі: коли впевненим тоном пропонується код, якого не існує, або функції, яких немає в бібліотеці [66]. Наприклад, LLM може використати ім’я методу, який логічно підходить, але насправді не визначений у даному класі. У тестах це проявляється як недійсні виклики або перевірки неіснуючих умов. Дослідження показали, що значний відсоток тестів від моделей типу GPT-4 виходить синтаксично некоректним або некомпільованим з першого разу [79]. Причина – все ті ж

галюцинації або змішування контекстів. По-друге, обмеження контекстного вікна: модель може просто не вмістити весь код модуля чи залежні класи у свій промпт. Якщо важливий для тестування аспект лишився за межами контексту, модель про нього "забуде". Це робить генерацію тестів для дуже великих класів чи сильно пов’язаних між собою функцій складною – доводиться якось стисло описувати або генерувати частинами. По-третє, нестабільність і чутливість до формулювання запиту. LLM – стохастичний механізм; невелика варіація у промпті або інші випадкові фактори (наприклад, різні параметри temperature) можуть призвести до істотно різних результатів. Дві спроби з одними і тими ж даними можуть видати різні тести: в одному будуть важливі перевірки, в іншому модель щось упустить. Це ускладнює відлагодження: важко передбачити, коли отриманий тестовий набір є оптимальним. Також є ефект, що модель може бути упередженою до певного стилю, залежно від того, на яких даних вона вчилась – наприклад, називати тест-методи певним чином або перевіряти тільки "щасливий" сценарій. Змінюючи інструкцію це можна відкоригувати, але потрібен час на підбір правильного промпта. І нарешті, практичні питання: вартість і продуктивність. Найпотужніші LLM (на кшталт GPT-5.1) – це комерційні і платні сервіси. Масштабне використання їх для генерації тестів у великому проєкті може коштувати дорого, оскільки моделі беруть плату за кожні тисячу токенів запиту і відповіді. Крім грошей є й часові витрати: якщо треба згенерувати тести для сотень модулів, навіть при швидкій відповіді моделі це може зайняти години. Вбудувати таке в CI/CD поки складно через затримки. До того ж, корпоративні користувачі інколи не можуть відправляти свій вихідний код у зовнішній сервіс з міркувань безпеки і приватності – дані можуть бути конфіденційними. Хоча існують відкриті моделі, які можна запускати локально, але вони зазвичай слабші. Ще один аспект – довіра до результатів: навіть якщо модель написала тест, розробник має його переглянути, переконатися, що він коректний і дійсно перевіряє потрібне. Поки що повністю автоматично додавати такі тести в репозиторій без рев’ю людиною ризиковано, адже існує шанс пропущеної помилки чи хибно-позитивного тесту.

Отже, великі мовні моделі відкривають нові можливості для автоматизації програмування, зокрема і написання модульних тестів. Вони здатні генерувати код близький до людського стилю і враховувати семантичний контекст, що було складно реалізувати традиційними алгоритмами. LLM, спеціально навчені на коді, вже вміють досить добре доповнювати та створювати функції, а тепер активно досліджується їх застосування у створенні тестових сценаріїв. Було розглянуто як працюють LLM, які є методи промпт-інженерії для покращення результатів, які підходи можна застосувати (ітеративний зворотний зв'язок, multi-agent тощо) та які зовнішні інструменти варто використовувати у поєднанні із моделями. Водночас, було окреслено виклики: від технічних (помилки, галюцинації, обмежений контекст) до організаційних (вартість, інтеграція в процеси, необхідність перевірки). Ці фактори потрібно враховувати при побудові системи генерації тестів на основі LLM.

1.3 Метрики та критерії оцінювання якості згенерованих тестів

Щоб визначити, наскільки вдало модель генерує модульні тести, необхідні чіткі метрики якості. Вимірювання якості тестів важливе як у наукових дослідженнях (для порівняння підходів), так і на практиці (щоб розуміти, чи достатньо добре покрита і перевірена система). Оцінка має бути комплексною – включати як кількісні показники (що можна автоматично порахувати), так і якісні критерії (що відображають змістовність тестів). Жодна одна метрика не дає повної картини, тому зазвичай використовують набір показників, кожен з яких висвітлює різний аспект. Важливо також розуміти, що високі числові показники не завжди гарантують високу якість – наприклад, 100% покриття коду не гарантує відсутності багів. Тому потрібно збалансовано інтерпретувати метрики і доповнювати їх ручним аналізом або експертною оцінкою.

До основних кількісних метрик якості тестів належить, звичайно, покриття коду (code coverage) [136, 28]. Це відсоток коду програми, який виконується при прогоні всіх тестів. Покриття буває різних видів:

- покриття рядків/операторів (line/statement coverage) – частка виконаних рядків коду;
- покриття гілок (branch/condition coverage) – частка виконаних логічних розгалужень (true/false у кожному розгалуженні);
- інколи в критичних сферах застосовують MC/DC (Modified Condition/Decision Coverage) – більш суворий критерій, що вимагає перевірки впливу кожної умови на результат складних логічних виразів.

У контексті генерованих тестів зазвичай беруть базові метрики: відсоток рядків і гілок, покритих тестами. Якщо модель генерує багато тестів і охопила майже весь код – це добрий знак, адже невиконані ділянки можуть ховати невиявлені помилки. В дослідженнях часто порівнюють покриття, досягнуте автоматичними тестами, із золотим стандартом (ручними тестами) або між різними моделями. Проте покриття не відображає, що саме перевіряють тести. Можна уявити ситуацію: тест викликає всі функції, але взагалі не містить assert-ів – покриття 100%, але сенсу від такого тесту нема. Тому вводять інші метрики.

Ще одна кількісна метрика – mutation score, показник, пов’язаний із мутаційним тестуванням [77, 78]. Як згадувалося, мутаційне тестування навмисно вносить дрібні помилки в код і перевіряє, чи тестовий набір їх виявить. Наприклад, у виразі замінюють > на < або число 0 на 1; якщо жоден тест не помітив зміни (тести як проходили, так і проходять) – значить, у тестах є помилка: певна умова коду не перевіряється належним чином. Mutation score – це відсоток "вбитих" мутантів до загальної кількості внесених. Високий mutation score означає, що тестовий набір здатен ловити більшість потенційних дефектів: тобто тести не просто виконують код (покриття), а й фіксують неправильну поведінку. Для оцінки згенерованих тестів mutation score часто більш показовий, ніж покриття, адже він ближчий до реальної здатності тестів знаходити баги. Якщо модель згенерувала тести з високим покриттям, але низьким mutation score – це сигнал, що перевірки слабкі (тести можуть містити асерти "на всяк випадок" або перевіряти тривіальні умови). У наукових статтях нерідко вимірюють mutation score для порівняння згенерованих тестів з існуючими або між різними моделями.

Поряд із кількісними показниками, дуже важливі якісні критерії – те, що важко автоматично виміряти, але що визначає корисність тестів. Один такий критерій – коректність тестів. Коректним вважається тест, який перевіряє очікувану поведінку програми, тобто відповідає специфікації. Якщо згенерований тест очікує, що функція поверне 5 для певного вводу, але насправді за вимогами вона мала б повернути 6 – тест некоректний, він буде хибно сигналізувати про дефект. На жаль, автоматично знати очікувану правильну відповідь можна тільки якщо є специфікація або оракул, чого у загальному випадку немає (і це одна з причин, чому генерація тестів важка). Тому коректність зазвичай оцінюють вручну: експерт читає тест і вирішує, чи логічно те, що цей тест перевіряє. У дослідженнях інколи підставляють згенеровані тести до реально випущеної версії програми: якщо тест падає на коректному коді, значить, він очікує неправильного результату (некоректний тест) [61].

Поряд із кількісними показниками, дуже важливі якісні критерії – те, що важко автоматично виміряти, але що визначає корисність тестів. Один такий критерій – коректність тестів. Коректним вважається тест, який перевіряє очікувану поведінку програми, тобто відповідає специфікації. Якщо згенерований тест очікує, що функція поверне 5 для певного вводу, але насправді за вимогами вона мала б повернути 6 – тест некоректний, він буде хибно сигналізувати про дефект. На жаль, автоматично знати очікувану правильну відповідь можна тільки якщо є специфікація або оракул, чого у загальному випадку немає (і це одна з причин, чому генерація тестів важка). Тому коректність зазвичай оцінюють вручну: експерт читає тест і вирішує, чи логічно те, що цей тест перевіряє. У дослідженнях інколи підставляють згенеровані тести до реально випущеної версії програми: якщо тест падає на коректному коді, значить, він очікує неправильного результату (некоректний тест) [10].

Ще суб’єктивний, але суттєвий критерій – змістовність асертів. Тест може мати перевірки, але що саме вони перевіряють? Якісні тести перевіряють суттєві умови роботи коду. Асерти мають перевіряти саме результат функції, чи дорівнює він очікуваному. Натомість слабкі тести можуть робити поверхневі перевірки –

наприклад, що функція не викинула виключення (тобто просто викликають її і все). Інколи можна побачити згенерований тест: "try { foo(); assertTrue(true); } catch (Exception e) { fail(); }". Він пройде, якщо функція не впаде – але чи цього достатньо? Звісно, ні, адже не перевірено правильність виходу. Тому для оцінки якості дивляться на тип і зміст перевірок: чи використовуються твердження рівності результату очікуваному, чи перевіряються всі ключові аспекти (наприклад, і основний результат, і побічні ефекти, якщо такі є). Цей критерій тісно пов’язаний з mutation score: якщо асерти малозмістовні, тести не ловлять мутантів, то як наслідок mutation score низький.

Технічна придатність тестів – ціла група критеріїв, що визначають, чи тест взагалі придатний для використання. Перший мінімальний показник – синтаксична правильність і компільованість. Згенерований тестовий код має хоча б компілюватись без помилок [10]. Якщо модель видає тест з помилками або викликає неіснуючі методи, треба такий результат доопрацьовувати – одразу викидати не обов’язково (його можна поправити, особливо в ітеративному режимі), але в ідеалі модель повинна одразу писати коректний код. У дослідах фіксують Compilation Success Rate (CSR) – відсоток згенерованих тест-кейсів чи класів, що скомпілювались успішно [10]. Далі, тест має проходити (green) на коректній версії коду і, по можливості, падати на помилковій. Якщо тест сам по собі падає на правильній програмі – він некоректний (або надто суворий). Якщо тест призначений виявити баг, але не ловить його – він недостатньо сильний. Дослідження відзначають, що тести, згенеровані LLM, поки що мають обмежену здатність ловити реальні баги, порівняно зі спеціалізованими методами чи навіть ручними тестами [10]. Інший технічний параметр – час виконання тестів. Модульні тести за задумом мають працювати швидко. Якщо згенерований тест, скажімо, містить внутрішній цикл на мільйон ітерацій, він уповільнить прогін. Звісно, модель навряд чи вигадає настільки неефективний тест навмисно, але треба стежити за такими моментами. Ізольованість тестів: кожен тестовий метод не має залежати від результатів попередніх. Генеровані тести можуть порушувати це: наприклад, спершу один тест створює файл, а наступний очікує, що він уже існує. Такі приховані залежності –

погана практика. Їх виявляють запускаючи тести у випадковому порядку і перевіряючи, що порядок не впливає на результат. Стабільність (детермінізм) – вже згадувались flaky-тести. Тут метрика – відсоток нестабільних тестів або flaky rate. В ідеалі він має бути 0. На практиці можна кілька разів запустити згенеровані тести і перевірити, чи є розбіжності. Причини флаків треба усувати (наприклад, якщо модель зробила тест залежним від поточної дати – зафіксувати значення дати в тесті).

Окрім функціональної якості є поняття якості коду тестів. Тести – це теж код, який потрібно читати і підтримувати. Тут вступають класичні метрики чистоти коду. Читабельність і зрозумілість – суб'єктивний показник, але важливий. Чи легко розібратись, що робить тест? Чи він не містить надлишкової складності? Можна також мати і формальні метрики: наприклад, цикломатична складність тестових методів – бажано, щоб вона була низькою (тест не повинен мати складної логіки, умов, циклів – він має бути лінійним сценарієм Arrange-Act-Assert). Довжина тестів: занадто довгі тести важко підтримувати, краще мати кілька менших, кожен з яких перевіряє один аспект. "Запахи" тестів: у літературі описано декілька типових проблем тестового коду (багато з них задокументовано в "xUnit Test Patterns" Г. Мезароса [145]). Деякі вже згадувались – "рулетка" тверджень, нестабільні (flaky) тести, надмірно специфічні тести. Сюди можна ще додати "масивні" тести, що перевіряють кілька речей відразу (Eager Test), дублювання коду між тестами. Ідеально, щоб згенерований інструментом тестовий код дотримувався принципів DRY (Don't Repeat Yourself). Звісно, повної відповідності чистому коду очікувати важко, але хоча б найгрубіші "запахи" варто не допускати. Наприклад, якщо модель називає всі тести test1, test2, ... – це погано (неінформативні назви – типовий "smell"). Якщо модель всюди копіює один і той же код налаштування замість використання механізму setUp, – теж неідеально. У наукових роботах зустрічаються спроби кількісно оцінити тестовий код, наприклад, підрахувати кількість "запахів" у згенерованих тестах порівняно з еталонними [10]. В одному дослідженні зазначено, що автогенеровані тести часто мають проблему з нечіткими іменами та іншими стилістичними вадами, що ускладнює їх сприйняття розробниками [10].

Ще один вимір – продуктивність та економічність генерації тестів. Якщо метод генерує прекрасні тести, але за 2 години на кожен клас – практична цінність сумнівна. Тому варто враховувати час генерації. Так само, якщо використовується платний API, можна порахувати кількість токенів і вивести орієнтовну грошову вартість генерації всіх тестів проєкту. У промислових умовах це важливо: інколи дешевше дати завдання молодшому розробнику написати тести вручну, ніж платити за тисячі викликів моделі. Є і менш очевидні витрати – людський час на доведення тестів "до зеленого стану". Якщо після автогенерації розробнику потрібно ще годину виправляти кожен тест, то автоматизація не дала виграшу. В ідеалі, цей час має прямувати до нуля (повністю автоматично), але поки реальні сценарії передбачають обов'язкове рев'ю та доопрацювання. Тим не менш, різні підходи можна оцінити за обсягом доопрацювання: можливо, одна модель створить тест, де необхідно буде тільки дрібницю виправити, а інша – створить зовсім некоректний тест, який простіше переписати.

У сучасних дослідженнях інколи вводять комплексні метрики, які агрегують кілька простих. Наприклад, можуть визначити, що "якість тесту" = (покриття + mutation score) / 2 – це спрощено, але дає один номер для порівняння. Або ж будують рейтинги моделей, враховуючи і покриття, і дефекти, і читабельність (останнє через опитування користувачів). Варто зазначити, що в науковій літературі з'явилися бенчмарки саме для оцінки генерування тестів LLM. Наприклад, TestEval – набір з 210 Python-завдань для перевірки здатності моделі генерувати тести з певним покриттям [112]. При оцінці учасників цього бенчмарку аналізують, чи досягає модель заданого рівня line, branch, path coverage. Дослідники Zhang і співавт. порівнювали різні метрики і з'ясували, що статичні метрики подібності (на кшталт BLEU, CodeBLEU) – тобто наскільки текст згенерованого тесту схожий на еталонний тест – майже не корелюють із динамічними метриками якості (покриттям, mutation score) [61]. Тобто тест може словесно бути зовсім іншим, ніж очікувався, але виконувати роботу не гірше. Висновок – основну увагу слід приділяти динамічним метрикам адекватності тестів (покриття, виявлення багів). Водночас потрібно завжди дивитися комплексно: наприклад, якщо один набір тестів має

coverage 90% і mutation score 50%, а інший – coverage 80% і mutation 80%, то який кращий? Перший охопив майже весь код, але пропустив половину помилок, другий – менше рядків, зате тести там більш змістовні. Такий баланс оцінюється з огляду на контекст. Можливо, більш важливо вловити логічні баги (тоді mutation score цінніший), або ж покрити все хоча б якимись базовими тестами (тоді coverage). Тому часто наводять обидва числа, а остаточне рішення приймає аналітик.

Оцінювання генерованих LLM-тестів має і свої виклики. По-перше, якщо відсутня специфікація, повну коректність автоматично не перевіриш – доводиться залучати розробників для рев'ю тестів. Людина, прочитавши тест, може сказати: "Цей тест безглуздий, він перевіряє те, що гарантується бібліотекою, а не логікою нашого коду" або "Тут тест очікує неправильну поведінку". Таку експертну оцінку складно формалізувати. По-друге, coverage ≠ якість: на практиці траплялись випадки, коли добивалися високого покриття, але продукт все одно падав у користувачів, бо тести не перевіряли реальних сценаріїв. З іншого боку, 100% mutation score теж не гарантія – можна уявити логічну помилку, яку жоден "дрібний мутант" не моделює, і її тести теж пропустять. Тому бажано комбінувати кілька методів оцінки. Скажімо, якщо тести мають і високий mutation score, і знаходять кілька реальних відомих багів – це вже сильний аргумент у бік того, що вони якісні. Якщо ще й розробники переглянули і схвалили їхній зміст – то, можна сказати, мета досягнута. В рамках дослідження в даній праці висновки робитимуться з урахуванням комплексу показників, аби обрати модель з найкращим балансом кількісних і якісних характеристик. Зрештою, людство цікавить така генерація, яка максимально наближена до роботи досвідченого інженера-тестувальника: тести мають бути повні, коректні, зрозумілі й ефективні.

1.4 Огляд попередніх напрацювань та інструментів

Тема автоматичної генерації модульних тестів за допомогою ШІ дуже активно обговорюється в науковій спільноті. Дуже "свіжим" є огляд Zhang та ін. (2025), який проаналізував 105 наукових праць щодо застосування LLM у тестуванні,

опублікованих до березня 2025 року [61]. Це свідчить про бурхливий інтерес і активні дослідження в цій сфері. Нижче будуть розглянуті ключові результати попередніх робіт, а також наявні інструменти – як нові, LLM-орієнтовані, так і класичні.

Ще до появи великих мовних моделей існувало кілька напрямків автоматизації тестування. Один з них – вже згадане символічне виконання (symbolic execution) [110]. Інструменти на кшталт KLEE (для C) [58, 59] або Microsoft Pex (для .NET) [94] аналізують програму, розглядаючи вхідні параметри як символи, і намагаються знайти такі їх значення, щоб пройти різні гілки коду. Символічні аналізатори можуть автоматично породити набір вхідних даних, що забезпечують високий відсоток покриття, і навіть виявити ситуації, де відбудеться збій або порушиться assert в коді. Але їхній недолік – складність із багатьма шляхами (комбінаторний вибух) і необхідність мати засоби для вирішення складних математичних умов. Інший напрям – генерація тестів на основі пошуку (Search-Based Software Engineering, SBSE) [104, 103]. Тут до задачі підходять як до оптимізації: треба знайти таку послідовність викликів або такий набір значень, щоб оптимізувати певну метрику (наприклад, максимізувати покриття). EvoSuite – яскравий представник SBSE – застосовує генетичний алгоритм, еволюціонуючи множину тестів так, щоб поступово збільшувати покриття і знаходити нові шляхи в програмі [39, 40]. EvoSuite досяг значних успіхів у покритті: часто його тести покривають 70-80% коду автоматично. Проте, як уже обговорювалось, ці тести страждають на низьку практичність: мають погані імена, інколи безглузді асерти. Розробники нерідко не довіряють таким тестам і не приймають їх до основної кодової бази. Існують й інші методи: модельне тестування (на основі перевірки моделей) формує модель поведінки (скінченний автомат) і генерує тести як послідовності переходів, але це вимагає моделювання системи вручну або напівавтоматично [115]. Фаззінг (fuzz testing) – надзвичайно успішний у сфері пошуку вразливостей та багів для програм на C/C++: він випадково або спрямовано генерує масу вхідних даних з метою спричинити збій програми [44, 43]. Фаззери (наприклад, AFL [4], libFuzzer [62]) традиційно не продукують юніт-тести у вигляді

коду, але видають конкретні вхідні дані, на яких програма падає. Це більше системне чи інтеграційне тестування, ніж модульне, хоча існують й фаззери для окремих функцій. Генерація на основі специфікацій (property-based testing) – підхід, де розробник задає властивості, які повинні виконуватися, а інструмент (наприклад, QuickCheck [101], Hypothesis [53]) генерує багато випадкових входів, перевіряючи ці властивості [139]. Це напівавтоматичний метод: він автоматизує лише частину – добір різних кейсів, але критерії перевірки задає людина.

Усі ці традиційні методи мають свої переваги та обмеження. Вони, як правило, гарантують певну строгість – наприклад, символічне виконання не пропустить помилку арифметичного переповнення, якщо зможе розв’язати відповідну умову; фаззер знайде "креш", якщо він можливий достатньо явно. Також результати цих інструментів часто відтворювані і прогнозовані: якщо EvoSuite сказав, що покриття 85%, то так і є; якщо KLEE не зміг пройти далі певної гілки, то хоч відомо, що він справді дослідив всі прості комбінації умов до певної глибини. На противагу, LLM – стохастичний та ймовірнісний інструмент. Він може здогадатись протестувати щось, про що не було явного сигналу в коді (наприклад, додати тест на null-параметр, навіть якщо в коді немає явної перевірки на null – просто тому, що "зазвичай це роблять"). Це цікава семантична можливість, але й ризик: модель може припустити таку поведінку, якої програма не має, і написати тест "від себе". Тобто LLM діє більше як "статистичний досвід", а не формальний алгоритм. Проте потенційна перевага LLM-підходу – генерувати тести, ближчі до того, як пише людина, з осмисленими перевірками й сценаріями. Крім того, LLM може використовувати знання з широкого контексту: скажімо, побачивши назву функції computeTax(), вона може додати тест на від’ємне значення (чи не нараховується податок для від’ємної суми) навіть якщо програмно це не випливає – просто розуміючи реальний сенс. Такі інтуїтивні речі не доступні традиційним інструментам, якщо явно не закладені. З іншого боку, класичні методи (символічні, пошукові) краще гарантовано покриють усі дрібні гілки та виняткові ситуації – LLM може про якісь "не здогадатися", особливо якщо вони не типові.

Останні дослідження намагались об’єднати сильні сторони традиційних підходів і LLM. Вже згадувалось про CodaMosa (Lemieux та ін., 2023) – це інструмент, що поєднав еволюційний фаззинг з мовною моделлю Codex [27]. Класична евристика MOSA (Many-Objective Sorting Algorithm) генерує тести, націлені на покриття різних гілок, і, коли застрягає, викликає LLM, аби той "підказав" тест для непокритої гілки з урахуванням коду. Результат – підвищення покриття там, де звичайний пошук не справляється, бо LLM може розуміти семантику умов (наприклад, що для проходження "if (specialFlag.equals("ADMIN"))" треба передати рядок "ADMIN"). Подібно, інша робота LangSym (Xu та ін., 2024) використовує LLM, щоб згенерувати тестові вхідні дані, а тоді запускає символічне виконання для розширення покриття [109]. Це вирішує проблему символіки з комплексними введеннями: модель пропонує декілька осмислених стартових сценаріїв, далі symbolic engine від них відштовхується і досліджує суміжні випадки. В результаті поєднання LLM + символіки дало краще покриття і масштабованість, ніж кожен підхід окремо. Ще один напрям – мутаційне підсилення. Алгоритм MuTAP (Dakhel та ін., 2023) покращив генерацію тестів LLM шляхом використання мутаційного тестування як механізму зворотного зв’язку: ті мутанти, які не "вбито" початковим набором тестів, додавались у промпт як контекст, щоб спонукати модель згенерувати нові тести, що їх виявлятимуть [36]. Це змушувало LLM генерувати більш точні перевірки, підвищуючи mutation score. MuTAP показав, що така комбінація перевершує базові запити до моделей за якістю тестів (синтаксичних помилок менше, виявлення багів краще).

Деякі наукові роботи присвячені безпосередньо LLM для генерації юніт-тестів без класичних методів, але з детальним аналізом як покращити результат. Серед піонерів – дослідження команди з Мічиганського університету під керівництвом Xie (2023), які представили ChatUniTest [26]. В їхньому підході модуль ChatGPT використовується не одноразово, а в циклі. Спершу генерується базовий тестовий код (використовуючи в промпті опис методу, його код та, можливо, приклади). Потім код аналізується – ChatUniTest містить механізм "generation-validation-repair" (генерація-валідація-ремонт). Якщо згенерований тест не компілюється або падає,

алгоритм застосовує набір правил (наприклад, виправити імпорт, додати відсутню залежність) або знов звертається до ChatGPT з повідомленням про помилку, щоб модель виправила свій код. Також ChatUniTest приділяє увагу фокусуванню контексту: при великому класі вони навчають модель виділяти лише релевантні частини. У підсумку ChatUniTest зміг перевершити EvoSuite за покриттям на половині проєктів, а за сумарним покриттям показав найкращий результат серед порівнюваних інструментів. Це важливий сигнал, що LLM при належній інженерії здатні не тільки писати гарні тести, а й досягати високої технічної ефективності. Інша робота – ChatTester від Yuan та співав. (2023) [79]. Вони поставили питання: "Чи можемо ми змусити ChatGPT самовдосконалювати свої тести?". Їхній підхід: спочатку згенерувати тести простим запитом, потім прогнати їх, виявити місця, де не вистачає перевірок або тести невірні, і також за допомогою ChatGPT спробувати поліпшити. Замість зовнішніх правил, ChatGPT виступає і генератором, і "рецензентом". Зокрема, вони впровадили у розмові з моделлю крок: "Оціни свої власні тести: чи всі гілки коду покриті, чи результати перевіряються?", а потім: "Виправ свій тест з урахуванням цих зауважень". Такий ланцюжок думок (CoT) специфічно для тестів допоміг отримати на ~34% більше компільованих тестів і на ~18% більше правильних асертів порівняно з одноразовим викликом моделі. Тобто ChatTester підвищив якість, використавши саму ж модель для аналізу своїх прогалин. Це цікавий прийом, що зменшує потребу в складній зовнішній логіці.

Компанії теж не стоять осторонь. Наприклад, в Meta презентували інструмент TestGen-LLM (Alshahwan та співав., 2024), який застосували не для створення тестів з нуля, а для покращення наявних ручних тестів [20]. У великій кодовій базі Meta є багато тестів, які могли пропустити якісь гілки. TestGen-LLM брав такий тест, дивився які умови коду не покриті, і за допомогою LLM дописував нові перевірки або нові тест-кейси, що розширювали покриття. По суті, модель використовувалася як асистент: "згенеруй додатковий тест, який би охопив оцю умову в коді, бо наявні тести її не покривають". Це доволі прагматичний підхід, коли AI не замінює, а доповнює працю інженера-тестувальника.

Порівнюючи традиційні та LLM-орієнтовані підходи варто підкреслити тенденцію: йде не заміна, а синтез. Дослідники все частіше роблять висновок, що найкращий результат дає комбінування методів. LLM чудово пише зрозумілий код тесту і придумує нетривіальні кейси, але може щось пропустити; натомість класичний інструмент може просканувати, що пропущено, і підказати це моделі. Така інтеграція – напрям розвитку технології. В перспективі можна очікувати появу комплексних рішень: наприклад, multi-agent систем для тестування, де один агент (LLM) грає роль "розробника" і пише код чи тести, другий агент – "QA-інженера" і перевіряє їх, третій – "системного інженера", який запускає і збирає метрики, і разом вони імітують командну роботу, покращуючи якість. Уже існують прототипи, наприклад, TestChain – система, де генерація вхідних даних і генерація очікуваних результатів розділені між двома моделями, що покращило точність тестів [60].

Також великі компанії інтегрують AI у свої конвеєри розробки (CI/CD). Наприклад, в роботі Santos та співав. зазначено, що близько 48% опитаних інженерів вже експериментують з залученням LLM на різних етапах тестування [12]. Це не лише генерація тестового коду, а й аналіз логів, написання тест-планів, генерація тестових даних. У майбутньому очікується, що AI-асистенти будуть пропонувати тести під час code review. Інша перспектива – гібридні фреймворки: наприклад, розширення EvoSuite, яке перед запуском буде звертатись до LLM, щоб той згенерував стартовий набір тестів, або навпаки – після генетичного пошуку передаватиме чорнові тести в LLM на "доведення до ладу" (перейменувати змінні, додати логічні асерти). Такі комбінації здатні об’єднати формальну строгість та семантичну гнучкість.

Отже, на даний момент вже є ґрунтовна база досліджень про LLM у тестуванні. Модель може автоматично генерувати модульні тести, які наближаються за якістю до людських, а іноді перевершують традиційні засоби за показниками метрик. Проте також зрозуміло, що найкращі результати дає інтеграція LLM з наявними підходами та інструментами, використання зворотного зв’язку від виконання тестів.

1.5 Безпека та конфіденційність даних

Використання великих мовних моделей (LLM) для генерації коду й модульних тестів породжує низку питань щодо безпеки даних та конфіденційності. Передача вихідного коду до зовнішнього AI-сервісу може створювати ризики витоку інтелектуальної власності та чутливої інформації. Ці ризики стали предметом уваги дослідників і експертів з кібербезпеки – зокрема, у переліку топ 10 вразливостей LLM за версією OWASP фігурують проблеми витоку даних і вразливостей ланцюга постачання програмного забезпечення [93]. Нижче будуть розглянуті основні аспекти безпеки та приватності даних при генеруванні тестів для коду за допомогою LLM, а також методи їхньої мінімізації.

1.5.1 Ризики передачі інтелектуальної власності через хмарні API

Під час використання пропрієтарних хмарних LLM-моделей вихідний код надсилається на сервери провайдера, що несе загрозу несанкціонованого доступу до інтелектуальної власності. Хоча передача даних зазвичай здійснюється через захищені канали (шифрування TLS), сам факт зберігання коду у стороннього постачальника викликає занепокоєння. Зокрема, за відсутності чіткої політики видалення даних провайдер може зберігати надісланий код практично безстроково [5]. Таким чином, конфіденційна інформація або фрагменти коду, що є комерційною таємницею, можуть залишатися в чужому середовищі довше, ніж це потрібно, підвищуючи ризик витоку. Це особливо критично для корпоративного сектору: передача закритого коду сторонньому сервісу може потенційно порушувати умови NDA (угода про нерозголошення) та ставити під удар охорону комерційної таємниці [13]. Іншими словами, усе, що розробник вводить у хмарний AI-сервіс, слід розглядати як таке, що може стати відомим третій стороні, якщо провайдер не гарантує повної конфіденційності і негайного видалення цих даних.

1.5.2 Проблема зберігання даних та донавчання моделей на користувацькому коді

Окремим викликом є політика провайдерів щодо використання отриманих даних для донавчання LLM. Більшість публічних LLM-сервісів за замовчуванням зберігають і аналізують промпти користувачів для подальшого поліпшення моделей. Наприклад, аналіз політик конфіденційності шести провідних розробників LLM показав, що усі вони за умовчанням використовують дані чатів користувачів для тренування моделей, причому деякі компанії залишають ці дані у себе без визначеного строку [128]. Таким чином, фрагменти приватного вихідного коду, надіслані в публічну LLM, можуть бути включені до тренувального набору даних і стати частиною "знання" моделі. У гіршому разі це призводить до того, що модель мимовільно відтворює фрагменти цього закритого коду у відповідях іншим користувачам, фактично спричиняючи витік інтелектуальної власності. Існує також ризик цілеспрямованої атаки на модель з метою витягнення інформації (data extraction attack), коли зловмисник через ряд спеціальних запитів намагається відновити дані, що були використані при навчанні моделі [133].

Для організацій, що працюють з чутливим кодом, вирішенням цієї проблеми є використання спеціальних корпоративних (enterprise) версій LLM, які гарантують режим без збереження даних. Такі сервіси обіцяють не використовувати введені користувачем дані для тренування загальних моделей і видаляти або ізолювати ці дані одразу після отримання результату. Наприклад, у рамках корпоративного сервісу OpenAI запроваджено політику, за якою всі введені бізнес-дані клієнта не використовуються для навчання моделей, а сам клієнт повністю контролює строки їх зберігання [38]. Подібні гарантії "Zero Data Retention" (нульового зберігання даних) [49] вже стали стандартом для багатьох постачальників LLM-рішень для бізнесу, що дозволяє дотримуватися вимог NDA та зберігати комерційну таємницю.

1.5.3 Локальні LLM як альтернатива хмарним сервісам

Найрадикальніший підхід до захисту даних – розгортання мовної моделі локально, у межах власного IT-інфраструктурного контуру (на приватних серверах, у хмарі компанії або навіть в ізольованому середовищі без виходу в інтернет). Такий on-premise сценарій гарантує, що вихідний код та інші дані не покидають межі організації. Відсутність передачі даних третім особам фактично усуває ризики, пов'язані з довірою до зовнішнього провайдера: дані залишаються під повним контролем компанії [25]. Це особливо важливо для індустрій з підвищеними вимогами до конфіденційності (банківська сфера, охорона здоров'я тощо) та для випадків, коли законодавство вимагає зберігати дані на території певної країни.

Однак переваги приватності при локальному розгортанні LLM мають свою ціну. По-перше, сучасні моделі з мільярдами параметрів потребують надзвичайно великих ресурсів – продуктивних GPU, значних обсягів пам'яті і потужного серверного обладнання. Забезпечення такої інфраструктури та її підтримка лягають на плечі організації. По-друге, якість генерування коду локальними (особливо відкритими) моделями може поступатися якості хмарних гігантів. Провідні пропрієтарні LLM, такі як GPT-5.1, треновані на колосальних обсягах даних і тонко налаштовані, часто демонструють кращі результати у складних завданнях, ніж компактніші моделі, які можна розгорнути самостійно. Для досягнення співставної якості локальну модель можливо доведеться додатково донавчати під конкретний домен, що знову ж потребує обчислювальних ресурсів і експертизи. Таким чином, між локальною та хмарною LLM існує компроміс: локальна модель забезпечує максимальну приватність і відповідність вимогам безпеки, тоді як хмарна модель надає вищу якість генерації та не потребує від користувача підтримки інфраструктури. Вибір залежить від пріоритетів організації – контролю над даними чи доступу до найпотужніших можливостей ШІ.

1.5.4 Методики захисту даних: санітизація та обфускація коду

За відсутності власної локальної моделі і необхідності звертатися до хмарного LLM-сервісу важливо мінімізувати обсяг чутливої інформації, що передається моделі. Один з підходів – попередня обробка (sanitization) вихідного коду перед відправкою до LLM. Розробникам рекомендується обфускувати та анонімізувати ті частини коду, що можуть видати конфіденційні відомості. Наприклад, можна тимчасово перейменувати імена класів, методів і змінних на узагальнені (замінивши назви, що містять назву компанії чи продукту, на нейтральні), видалити коментарі, в яких описана бізнес-логіка або згадуються деталі реалізації, а також замаскувати будь-які літеральні константи, що несуть чутливі дані (такі як URL внутрішніх сервісів, ключі API, паролі тощо). Такий підхід відповідає принципу мінімізації даних та псевдонімізації, рекомендованому фахівцями з приватності – зокрема, у практичних порадах Європейської ради із захисту даних (EDPB) зазначено доцільність маскування або анонімізації конфіденційних полів перед тим, як подавати їх на вхід LLM [5].

Водночас надмірна анонімізація може мати і негативні наслідки. Якщо з коду вилучити занадто багато контекстної інформації, модель перестане "розуміти" специфіку завдання, що може різко знизити якість згенерованих тестів. Наприклад, перейменування всіх доменних сутностей на абстрактні назви ускладнить для LLM визначення предметної області та взаємозв’язків у програмі – відповідно, згенеровані тести можуть бути занадто тривіальними або взагалі нерелевантними. Тому у питаннях санітизації важливо дотримуватися балансу: забезпечити видалення суто конфіденційних деталей, але зберегти достатньо інформації для того, щоб модель могла коректно зрозуміти структуру і призначення коду.

1.5.5 Безпека згенерованого тестового коду

Навіть якщо питання конфіденційності вхідних даних вирішено, залишається аспект якості та безпечності самого коду, що генерується LLM. На жаль, великі

мовні моделі не гарантують, що згенеровані ними тест-кейси або допоміжний код будуть відповідати усім принципам secure coding. Навпаки, дослідження показують, що LLM часто порушують найкращі практики безпеки в своєму коді. Зокрема, виявлено випадки, коли згенерований тестовий код містив критичні уразливості – наприклад, відсутність перевірки входів, слабку аутентифікацію сесій чи некоректну роботу з конфіденційними даними [117]. Більш того, були зафіксовані ситуації, коли модель вводила прямі уразливі конструкції: наприклад, в одному з експериментів декілька різних LLM-моделей згенерували код із жорстко зашитими обліковими даними (паролями), що є грубим порушенням безпеки і могло б призвести до компрометації системи [19].

Ще одна потенційна небезпека генерованого коду – галюцинації залежностей. LLM у відповідях на програмні запити інколи впевнено посилаються на бібліотеки або пакети, яких насправді не існує в екосистемі. Модель може "вигадати" ім’я неіснуючого пакету, якщо вважає його логічним розв’язанням задачі. Для недосвідченого розробника або в умовах поспіху така відповідь виглядає правдоподібно – людина може спробувати підключити чи встановити цей пакет. Це породило новий тип атаки на ланцюг постачання: зловмисники відстежують подібні вигадані назви пакетів і реєструють їх в публічних репозиторіях із шкідливим вмістом. Як наслідок, якщо хтось спробує встановити "галюцинований" пакет, він отримає шкідливий код замість очікуваної бібліотеки. Такий сценарій отримав назву "slopsquatting". Свіжі дослідження підтверджують масштаб проблеми: в середньому близько 20% назв пакетів, запропонованих LLM при генерації коду, не існують у реальності [120]. Більш того, відкриті моделі схильні до таких галюцинацій значно більше, ніж комерційні – у тестах відкриті LLM генерували неіснуючі залежності приблизно вчетверо частіше (21,7% випадків), ніж пропрієтарні моделі (5,2%) [120]. Дослідники зафіксували понад 205 тисяч унікальних вигаданих найменувань пакетів, що були продуковані різними моделями [120]. Потенційно кожен з них може стати носієм атаки, якщо буде зареєстрований зловмисником. Таким чином, при використанні AI-генерованих підказок щодо залежностей, особливо у середовищах мов Python, JavaScript та ін., критично важливо перевіряти, чи існує зазначений

пакет і чи він є легітимним. У корпоративних процесах розробки доцільно впровадити автоматизовані сканери залежностей та політики перевірки (allow-list) схвалених бібліотек, щоби відфільтровувати потенційно небезпечні або невідомі компоненти, запропоновані LLM.

Підсумовуючи, розвиток великих мовних моделей відкриває нові можливості для автоматизації написання коду та тестів, проте разом з цим породжує нові виклики у сфері безпеки даних. При використанні LLM-асистентів розробникам і організаціям слід приділяти особливу увагу захисту інтелектуальної власності: забезпечувати, щоб конфіденційний код не покидав меж безпечного контуру, або користуватися послугами з гарантіями невикористання даних. Політики безпеки повинні враховувати ризики витоку через зберігання даних у провайдера та потенційного порушення NDA. У випадках, коли це неприйнятно, варто розглянути перехід на локальні LLM-моделі, особливо для критично важливих або чутливих проєктів – попри вищі витрати, вони дають повний контроль над даними. Якщо ж застосовується публічний LLM, необхідно впровадити заходи санітизації коду, аби мінімізувати витік важливих деталей. Генерованому коду не можна сліпо довіряти: його потрібно перевіряти на безпечність і коректність, проводити код-рев'ю та тестування з урахуванням можливих уразливостей. В цілому, підтримка належного рівня безпеки та конфіденційності при роботі з LLM вимагає комбінування технічних рішень (шифрування, обфускація, локальний хостинг) з організаційними заходами (політики використання AI, навчання персоналу з кібергігієни) та постійного аудиту нових загроз.

1.6 Висновки

Модульні тести є надзвичайно важливими для забезпечення якості ПЗ, але їх ручне написання потребує значних зусиль і дисципліни [116]. Наявні засоби автоматизації на кшталт EvoSuite [39, 40] досягають високого покриття коду (близько 80%) за допомогою еволюційних алгоритмів, проте якість згенерованих ними тестів часто поступається ручним – такі тести бувають складними в читанні,

підтримці і практично непридатними без доопрацювання розробником [79]. Це підсилює потребу в нових підходах, які поєднали б високе покриття з якістю тестового коду, наближеною до написаного людиною.

Сучасні великі мовні моделі, особливо спеціалізовані на програмному коді, відкривають нові можливості для автоматизації тестування [131, 135]. Такі моделі здатні генерувати змістовний, зрозумілий тестовий код і придумувати нетривіальні тестові сценарії, спираючись на контекст програми. На відміну від традиційних генераторів, LLM можуть враховувати семантику і намір функцій, завдяки чому згенеровані тести ближчі до інженерних стандартів. Це дозволяє очікувати підвищення корисності автогенерованих тестів – вони потенційно краще відображають реальні вимоги до поведінки коду та легше сприймаються розробниками.

Щоб максимально реалізувати потенціал LLM у генерації модульних тестів, застосовуються спеціальні методики prompt-інженерії [138, 97] та ітеративного вдосконалення відповіді. Зокрема, інструмент ChatUniTest використовує цикл "генерація – валідація – ремонт" за участю ChatGPT: модель генерує базовий тест, потім виявляються помилки (синтаксису чи падіння тестів) і LLM отримує зворотний зв’язок для їх виправлення [26]. Подібно, підхід ChatTester пропонує моделі самостійно проаналізувати власні тести і поліпшити їх у наступному запиті [79]. Перспективним напрямом є multi-agent схеми, де кілька LLM-агентів співпрацюють у ролях розробника, тестувальника, аналізатора тощо [134]. Такі багатоетапні стратегії дозволяють LLM поступово виправляти помилки, охоплювати більше гілок коду і генерувати більш надійні тести. Крім того, інтеграція LLM із зовнішніми інструментами підсилює результат: наприклад, гібридний підхід CodaMosa поєднав еволюційний пошук (MOSA) з підказками від Codex, що допомогло "прорватися" через важкодоступні гілки програми [27]. Отже, комбінування методів (LLM + класичні алгоритми) здатне компенсувати недоліки кожного з них і суттєво покращити якість тестів.

Для об’єктивного аналізу результатів автогенерації необхідно застосовувати набір взаємодоповнювальних метрик і критеріїв. Кількісні метрики включають

покриття коду (ліній, гілок тощо) – базовий показник, що свідчить, яка частина програми перевіряється тестами [136, 28], а також mutation score – відсоток штучно внесених мутантів коду, що були виявлені тестами [77, 78]. Високий mutation score означає, що тестовий набір не лише виконує код, а й ловить потенційні дефекти, і тому є більш показовим для оцінки здатності тестів знаходити реальні баги. Якісні критерії охоплюють коректність перевірок (наскільки очікувані результати співпадають зі специфікацією поведінки), змістовність асертів (чи перевіряють тести саме ключові умови роботи коду, а не тривіальні факти), технічну придатність (синтаксична правильність, успішна компіляція та виконання на еталонній версії програмного коду, відсутність flaky-поведінки [130, 9]), а також читабельність і структурованість тестового коду. Жоден окремий показник не дає повної картини, тому важливо аналізувати сукупність метрик у комплексі та інтерпретувати їх з урахуванням контексту. Наприклад, 100% покриття коду мало важить, якщо твердження в тестах не перевіряють суттєву логіку програми; навпаки, низьке покриття при високому mutation score вказує, що тести перевіряють критичні випадки, хоча і не охопили весь код. Таким чином, успіх LLM-генерації тестів доцільно вимірювати багатогранно – як кількісно, так і якісно – і за потреби залучати експертну перевірку складних аспектів (коректність, стиль тощо).

Аналіз сучасних робіт показує життєздатність підходу генерації тестів за допомогою LLM. З’явилися прототипи, що вже демонструють конкурентні результати порівняно з традиційними засобами. Зокрема, ChatUniTest [26] у ряді проєктів досяг покриття вищого, ніж EvoSuite [39, 40], при цьому продукуючи зрозуміліший код тестів. Інший інструмент – MuTAP [36] – підвищив ефективність LLM-тестів за рахунок зворотного зв’язку від мутаційного аналізу: модель отримувала інформацію про невиявлені мутанти і генерувала додаткові тести для їх "вбивства", що помітно підвищило mutation score. Такі системи, як і ChatTester [79], засвідчили, що залучення LLM до аналізу та покращення власних же тестів збільшує відсоток правильних асертів і синтаксично коректних тестів. CodaMosa [27] продемонструвала переваги комбінованого підходу: поєднання еволюційного алгоритму з мовною моделлю дозволило обійти обмеження кожного з них і суттєво

розширити охоплення тестами складних ділянок коду. Узагальнюючи досвід попередніх робіт, дослідники доходять висновку, що найкращі результати дає синтез методів: LLM генерує семантично осмислені тести, а класичні інструменти або додатковий аналіз допомагають виявити пропущені кейси та помилки. Це підтверджує перспективність напряму і вказує, в якому руслі варто розробляти нові рішення.

Широке залучення хмарних LLM-сервісів до генерації коду породжує нові виклики у сфері безпеки даних. Передача приватного вихідного коду зовнішній моделі містить ризик витоку інтелектуальної власності та чутливої інформації: відправлений на сервер код може бути збережений постачальником послуги і теоретично стати доступним третім особам [93, 5]. Більше того, більшість публічних LLM використовують отримані від користувачів дані для донавчання моделей, що створює шанс мимовільного відтворення фрагментів чужого коду в відповідях іншим запитувачам [128]. Для захисту від цього ризику організаціям варто розглядати локальні розгорнення LLM або корпоративні версії сервісів із гарантіями невикористання введених даних. Ще один аспект – надійність самого згенерованого коду: моделі можуть допускати небезпечні практики або помилки [117, 19]. Відомі випадки, коли в згенерованих тестах з’являлися небезпечні конструкції (наприклад, жорстко прописані облікові дані) або галюциновані залежності – посилання на неіснуючі пакети. Останнє явище відкрило новий вектор атак: зловмисники реєструють вигадані моделлю назви бібліотек у публічних репозиторіях, і розробник, нічого не підозрюючи, може випадково завантажити шкідливий код [120]. Таким чином, при використанні LLM-асистентів у програмуванні потрібно впроваджувати заходи безпеки: мінімізувати передачу секретного коду (санітизація, обфускація чутливих фрагментів), перевіряти рекомендації моделей (особливо щодо залежностей), застосовувати політики контролю (використання дозволених бібліотек, обов’язковий код-рев’ю згенерованих тестів) і за можливості утримувати генерацію в межах безпечного контуру (локальні моделі або ізольовані середовища). Дотримання цих принципів є обов’язковим, щоб впровадження LLM у процес тестування не створило додаткових ризиків для проєкту.

Використання великих мовних моделей для генерації модульних тестів є перспективним напрямом, здатним суттєво знизити трудомісткість написання тестів і підвищити рівень покриття та якості тестового набору. Водночас успішне застосування цього підходу залежить від врахування згаданих викликів: необхідності інженерії запитів, ретельної валідації результатів, комбінування з перевіреними методами та забезпечення безпеки даних.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ГЕНЕРАЦІЇ МОДУЛЬНИХ ТЕСТІВ ВЕЛИКИМИ МОВНИМИ МОДЕЛЯМИ

2.1 Пошук та підбір моделей

Ринок сучасних великих мовних моделей надзвичайно насичений – існують сотні доступних моделей різних розробників. Для коректного і репрезентативного експерименту необхідно обрати підмножину моделей, на яких буде проводитися подальше дослідження генерації модульних тестів. З огляду на невеликий ресурс часу, було вирішено обмежити вибір 5 LLM. Відбір цих моделей здійснювався на основі репрезентативності: до вибірки мали увійти як пропрієтарні (закриті), так і відкриті моделі; як моделі загального призначення, так і спеціалізовані на генеруванні коду; крім того, бажано, щоб моделі походили від різних розробників. Такий підхід забезпечує різноманітність та усуває ризик взяти 5 схожих флагманських моделей, що могли б показати майже однакові результати.

Ще однією важливою вимогою було забезпечення єдиних умов тестування для всіх моделей. Щоб уникнути розбіжностей у використанні різних API чи інтерфейсів, доступ до всіх моделей планувався через єдиний шлюз. Для цього обрано платформу OpenRouter (рисунок 2.1), що є агрегатором LLM-моделей різних провайдерів [90]. OpenRouter надає стандартизований API для понад 500 моделей від 60+ постачальників, який повністю сумісний із протоколом OpenAI. Це означає, що можна змінювати модель фактично зміною значення одного параметра, не переписуючи код під кожен окремий API. Такий уніфікований підхід значно знижує інженерні зусилля: не потрібно реалізовувати окремий клієнтський код для кожної моделі чи отримувати різні облікові ключі, оскільки всі запити проходять через єдину точку доступу [91]. Окрім зручності інтеграції, OpenRouter забезпечує й інші переваги, важливі для дослідження – зокрема, оптимізацію витрат шляхом маршрутизації запитів через дешевші провайдери за потреби та консолідовану систему оплати [91]. Таким чином, вибір OpenRouter як платформи доступу

обґрунтований тим, що він надає єдине уніфіковане середовище для експериментів з різними моделями при мінімальних накладних витратах на інтеграцію.

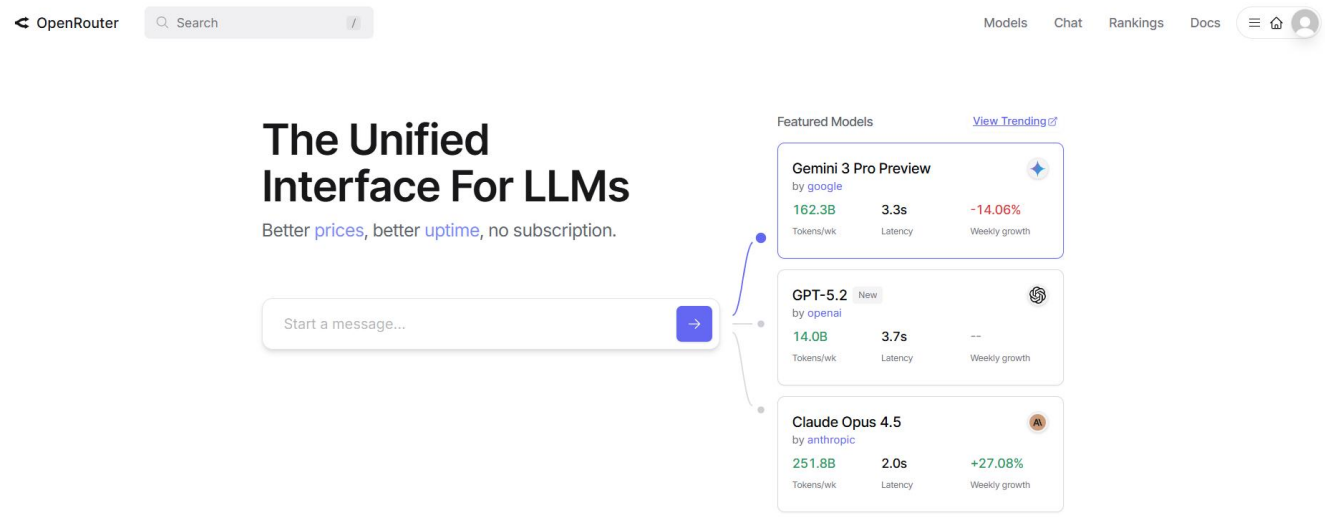

Рис. 2.1. Головна сторінка OpenRouter

2.1.1 Первинний пошук кандидатів

Зважаючи на тисячі існуючих LLM, перед остаточним відбором п’ятірки було проведено попередній огляд та фільтрацію кандидатів. На цьому етапі проаналізовано інформацію з авторитетних джерел – результати незалежних бенчмарків (LiveCodeBench [15, 65] та SWE-Bench [108, 107]), рейтинги продуктивності моделей та огляди спільноти (в тому числі публікації на профільних сайтах та обговорення на форумах). Зокрема, використовувалися рейтинги моделей штучного інтелекту (наприклад, Artificial Analysis LLM Leaderboard [18, 16]), а також наявні дослідження [141, 8] та аналітичні статті щодо найкращих моделей [2, 22]. У результаті було сформовано список із 21 моделі, які потенційно відповідали критеріям дослідження та заслуговували на детальніший аналіз. До цього списку ввійшли:

- флагманські моделі провідних компаній: GPT-5 [83], GPT-5 Codex [81], GPT-5 mini [82], GPT-5.1 [85], GPT 5.1 Codex [84], o3 [88], o4-mini [89] від OpenAI, Gemini 2.5 Pro [45] та Gemini 3 Pro [46] від Google, Claude Sonnet 4.5 [56] та Claude

Opus 4.5 [55] від Anthropic, Grok 4 [143] та Grok 4 Fast [142] від xAI – ці моделі очолюють рейтинги багатьох бенчмарків і демонструють найвищі результати на стандартних наборах задач [15, 108, 16];

- передові відкриті моделі: DeepSeek R1 0528 [33] від DeepSeek (за деякими показниками бенчмарків наближається до Gemini 2.5 Pro), gpt-oss-20b [87] та gpt-oss-120b [86] від OpenAI (відкриті аналоги GPT), GLM-4.6 [148] від Zhipu AI, Kimi K2 Thinking [75] від Moonshot AI – дані моделі демонструють одні з найкращих результатів на бенчмарках, але в той же час вони мають велику перевагу у вигляді відкритої ліцензії;
- моделі, рекомендовані AMD: за даними випробувань, проведених інженерами AMD, більшість менших локальних моделей не справляється з комплексними завданнями кодування [141, 8] – фактично єдиною реально дієздатною локальною моделлю виявилася Qwen3-Coder 30B [7] від Alibaba, а за наявності великих обчислювальних ресурсів рекомендується GLM-4.5-Air [147] від Zhipu AI. Також дослідники відзначили модель Magistral Small 1.2 [74] від Mistral AI. Ці висновки стали приводом включити зазначені три відкриті моделі до переліку кандидатів.

Сформувавши список із 21 претендента необхідно визначити набір критеріїв для їх порівняльного аналізу. На практиці існує багато стандартних бенчмарків, за якими зіставляються моделі, тому спершу з них було відібрано еталонні.

2.1.2 Відбір релевантних бенчмарків

Оцінюючи якість моделей з погляду генерації коду, сучасні дослідники покладаються на низку загальноприйнятих бенчмарків. Було здійснено огляд наявних наборів задач та метрик, особливо тих, що стосуються програмування та інженерії ПЗ [122, 1, 123]. Було розглянуто 19 бенчмарків (HumaEval/HumanEval+, MBPP/MBPP+, SWE-Bench, BIG-Bench/BIG-bench Lite/BIG-Bench Hard/BIG-Bench Extra Hard, MMLU/MMLU-Pro, CodeXGLUE, Humanity’s Last Exam, Code Lingua, LiveBench, Aider’s polyglot benchmark, LiveCodeBench/LiveCodeBench Pro, MASK),

серед яких як класичні тести на генерацію коду, так і новітні комплексні екзамени для LLM:

1) HumanEval – набір із 164 програмних завдань на Python, запропонований OpenAI у 2021 році для оцінювання здатності моделі писати правильний код за текстовим описом [50]. Основна метрика – Pass@1, відсоток завдань, правильно розв'язаних моделлю з першої спроби. HumanEval став де-факто стандартом оцінки перших кодових LLM (наприклад, Codex), однак на сьогодні його складність уже недостатня: провідні моделі досягли на ньому показників понад 90% успішності. Іншими словами, тест фактично "пройдено" сучасними LLM, тому опиратись лише на HumanEval уже не інформативно. Існує розширена версія HumanEval+ (EvalPlus), де кількість тест-кейсів на кожне завдання збільшено у приблизно 80 разів для більш суворої перевірки [21]. Утім, вона також втрачає свою актуальність.

2) MBPP (Mostly Basic Python Problems) – ще один популярний набір задач з 2021 р., орієнтований на базові навички програмування на Python [70]. Він містить 500 простих завдань з відповідями і використовувався для оцінки моделей нарівні з HumanEval. Розширена версія MBPP+ (EvalPlus) так само пропонує більше тестів для кожного завдання [21]. Як і HumanEval та HumanEval+, ці бенчмарки нині вважаються застарілими: провідні моделі легко розв'язують більшість задач, тож різниця між ними майже не проявляється.

3) SWE-Bench – сучасний бенчмарк, який зосереджується на реальних завданнях розробки ПЗ [108, 107]. Він побудований на основі виправлення реальних Issues з репозиторіїв GitHub і вимагає від моделі не лише згенерувати код, але й вирішити практичну проблему (виправити баг, реалізувати новий функціонал тощо) у контексті існуючого проєкту. Результати вимірюються як відсоток успішно вирішених задач (закритих Issue). SWE-Bench є одним із найактуальніших тестів, оскільки наближений до повноцінної інженерної діяльності: він перевіряє здатність моделі інтегрувати зміни в багато-файловий проєкт, розуміти існуючий код і проходити інтеграційні тести. Висока складність цього бенчмарку робить його корисним для визначення відмінності між передовими моделями.

4) BIG-Bench (Beyond the Imitation Game Benchmark) – велика збірка з понад 200 різнорідних завдань, призначена для всебічної оцінки можливостей LLM (від простих питань до складних логічних викликів) [24, 23]. Існують його варіації: BIG-Bench Hard (BBH) – піднабір із 23 найскладніших завдань; BIG-Bench Lite (BBL) – облегшена версія, що складається із 24 різноманітних задач; BIG-Bench Extra Hard (BBEH) – найновіша версія, де кожне завдання BBH замінено на ще складніше аналогічної тематики. Ці бенчмарки перевіряють загальні навички міркування та "здоровий глузд" моделі. На жаль, вони слабо корелюють саме із інженерією програмного забезпечення. До того ж, провідні моделі вже близькі до стелі на BBH – власне через це і з'явилась версія "Extra Hard". Варто зазначити, що відсутність публічно доступного агрегованого рейтингу за цими бенчмарками ускладнює їх використання.

5) MMLU-Pro (Massive Multi-Task Language Understanding, Professional) – розширена версія відомого академічного тесту MMLU [71, 137, 17]. Включає приблизно 12 000 запитань експертного рівня з різних галузей (STEM, гуманітарні, соціальні науки тощо) з підвищеною складністю – 10 варіантів відповіді замість 4, усунені тривіальні питання, більше акценту на міркування. MMLU-Pro було запроваджено, оскільки базовий MMLU став надто "легким" для сучасних моделей.

6) CodeXGLUE – комплексний бенчмарк від Microsoft (2020 р.), який об'єднує 14 датасетів для 10 різних завдань з аналізу та генерування коду [30]. Зокрема, він містить задачі таких типів: генерація коду з опису, генерація опису коду, виявлення дефектів, автодоповнення коду, переклад коду на іншу мову тощо. CodeXGLUE свого часу став важливим кроком уніфікації оцінки "розуміння коду" для моделей. Проте нині він вважається нерелевантним для новітніх моделей, оскільки охоплює відносно прості сценарії і не відображає здібностей моделей до складних багатокрокових завдань.

7) Humanity's Last Exam (HLE) – так званий "останній екзамен людства", надскладний тест загальної ерудиції та мислення, створений у 2025 р. Центром безпеки ШІ та Scale AI [51, 52, 14]. Містить 2500 питань рівня експерта з понад 100 дисциплін, покликаний стати новим рубежем, оскільки моделі вже досягли

людського рівня на попередніх екзаменах на кшталт MMLU. HLE цікавий тим, що охоплює і питання з програмування/інформатики, і перевіряє глибину міркування. Наразі навіть найпотужніші LLM дають вкрай низькі результати на HLE.

8) Code Lingua – спеціалізований бенчмарк з перекладу вихідного коду між мовами програмування [29]. Він перевіряє, наскільки модель розуміє семантику коду і здатна відтворити ту саму логіку іншою мовою. Наприклад, завдання – перекласти код Java на Python, або код C++ на JavaScript. Code Lingua акцентує увагу саме на трансляції коду, а не на його створенні з нуля. Однак рейтинг цього бенчмарка містить дуже обмежену кількість моделей, більшість з яких вже є застарілими.

9) MASK (Model Alignment between Statement and Knowledge) – це бенчмарк, що оцінює правдивість та чесність відповідей моделей, відокремлюючи показник чесності від простої фактичної точності [118, 69]. Розроблений Центром безпеки ШІ (CAIS) у 2025 році, MASK містить питання, на які моделі можуть навмисне збрехати, і перевіряє узгодженість їхніх тверджень із внутрішніми "знаннями". Хоча він є важливим для оцінки надійності моделей (виявлено, наприклад, що більші моделі мають вищу точність, але не обов’язково більше прагнуть до правди), до генерації програмного коду цей тест прямого стосунку не має.

10) Aider’s polyglot benchmark – набір тестів, запроваджений платформою Aider для оцінки вбудованих можливостей моделей редагувати та дописувати код [6]. Містить 225 складних практичних задач із ресурсу Exercism на різних мовах (C++, Go, Java, JavaScript, Python, Rust). Модель повинна внести правки у наданий код так, щоб пройти всі тести. Цей бенчмарк зосереджується на сценарії "AI-помічник, що редагує код у середовищі IDE" і є дуже показовим для інтерактивних можливостей LLM. Недоліком є те, що даний рейтинг поки містить результати лише обмеженого кола моделей.

11) LiveBench – "живий" бенчмарк, що щомісяця оновлюється новими задачами з нещодавніх джерел (наукових статей, новин, конкурсів) і автоматично оцінює відповіді моделей за об’єктивними критеріями [63]. Він складається з категорій: математика, кодування, логічне мислення, аналіз даних, виконання

інструкцій, мовні задачі тощо. Особливість LiveBench – захист від витоку тестових даних у тренування (data contamination) шляхом постійного додавання нових завдань, актуальних "на сьогодні". Таким чином, LiveBench показує актуальну працездатність моделі на нових викликах.

12) LiveCodeBench – сучасний динамічний бенчмарк від команди DeepMind/Google, спрямований на оцінку здатності моделі писати та виправляти код у режимі "реального часу" [15, 65]. Він використовує реальні зміни в open-source проєктах: тобто моделі пропонується вирішити задачу з відкритого репозиторію, після чого її відповідь виконується і перевіряється на актуальних тестах проєкту. Важлива відмінність LiveCodeBench – боротьба з "контамінацією" даних: використовуються задачі, які з'явилися після дати навчання моделей, щоб виключити можливість, що правильне рішення було в тренувальному наборі. Цей бенчмарк надзвичайно складний, він показує розрив між умінням моделі вирішувати ізольовані алгоритмічні задачі і вмінням "інженерити" – працювати з живим кодом, системним оточенням, багатофайловими проєктами.

13) LiveCodeBench Pro – спеціалізований піднабір LiveCodeBench, що фокусується на завданнях рівня олімпіад з програмування (Codeforces, ICPC, IOI) і керується командою з медалістів цих олімпіад [64]. Це фактично експертний стрес-тест алгоритмічних здібностей моделей. Завдання групуються по рівнях складності (Medium, Hard). Однак LiveCodeBench Pro містить оцінки лише небагатьох моделей-лідерів.

Провівши такий огляд, з 19 бенчмарків було обрано 5 основних, які покривають різні аспекти завдань і є актуальними на 2025 рік: SWE-Bench, MMLU-Pro, Humanity's Last Exam, LiveBench і LiveCodeBench. Решта були відхилені через те, що вони: або застарілі та нерелевантні (HumaEval/HumanEval+, MBPP/MBPP+, MMLU, CodeXGLUE, Code Lingua), або мають вузьку вибірку моделей (Aider's polyglot benchmark, LiveCodeBench Pro), або не мають загальнодоступного рейтингу (BIG-Bench/BIG-bench Lite/BIG-Bench Hard/BIG-Bench Extra Hard), або не стосуються сфери програмної інженерії (MASK). Вибрані п'ять бенчмарків

відповідають поставленій задачі – оцінити якість коду модульних тестів згенерованого великими мовними моделями.

2.1.3 Порівняльний аналіз моделей та фінальний відбір

Маючи на руках список кандидатів (21 модель) та визначені критерії (5 бенчмарків + додаткові характеристики), було складено порівняльну таблицю (рисунок 2.2). У ній для кожної моделі зібрано наступні дані:

1) Результати на обраних бенчмарках: відсоткові показники успішності на SWE-Bench, MMLU-Pro, HLE, LiveBench, LiveCodeBench. Ці цифри взято з відкритих рейтингів [15, 65, 108, 107, 17, 14, 63]. Для деяких моделей показники певних бенчмарків відсутні, в таблиці такі поля залишилися порожніми.

2) Вартість використання: ціна моделі за 1 млн токенів (вхідних і вихідних) згідно з тарифами OpenRouter. Наприклад, для моделі GPT-5 стандартна вартість складала $1.25 за 1M токенів вводу та $10 за 1M токенів виводу, тоді як менша модель GPT-5 mini були значна дешевшою ($0.25 та $2 відповідно). Цей показник важливий з практичної точки зору – оцінити, чи є модель економічно доцільною для масового застосування (генерації сотень тестів).

3) Підтримка донавчання (fine-tuning): наявність можливості донавчити модель під специфічні дані. Більшість закритих API-моделей (OpenAI, Anthropic, Google) на 2025 р. не дозволяють користувачам донавчати свої великі моделі, тоді як деякі відкриті моделі (наприклад, DeepSeek або Qwen) можуть бути донавчені на власному обладнанні.

4) Наявність функції міркування: чи підтримує модель спеціальний режим або архітектурні особливості для багатокрокового розв’язування задач (reasoning). Наприклад, відомо, що серія моделей OpenAI o1–o4 має вбудований механізм поетапного міркування під час генерації відповіді. Інші моделі (зазвичай більш відкриті) такого явного механізму не мають і генерують код "в один прохід". У таблиці це враховано для розуміння, як на результати впливає наявність chain-of-thought.

5) Розмір контекстного вікна: максимальна довжина промпта (і окремо виходу), з якою модель може працювати. Новітні моделі суттєво збільшили контекст – до 100k–1M токенів у Google Gemini, GPT-5 тощо. Це важливо при генеруванні тестів для великих файлів або аналізі довгого коду.

6) Тип та ліцензія: узагальнено зазначено, чи модель являється загального призначення (general) чи оптимізованою для програмного коду (code), а також відкрита вона чи закрита. Ці атрибути допомагають переконатися, що вибрано збалансований набір.

Model	Creator	API name	Type	License	OpenRouter provider	Price per 1M (in/out)	Context window (in/out)	Fine-tuning support	Reasoning support	SWE-bench	MMLU-Pro	HLE	LiveBench	LiveCodeBench
Gemini 3 Pro (high)	Google	google/gemini-3-pro-preview	general	proprietary	google-vertex	$2/$12	1M/64k	No	Yes	**71,6%**	**89,8%**	**37,2%**	**79,70%**	**91,7%**
GPT-5.1 (high)	OpenAI	openai/gpt-5.1	general	proprietary	openai	$1.25/$10	400k/128k	No	Yes	67,2%	87,0%	**26,5%**	78,79%	86,8%
GPT-5.1 Codex	OpenAI	openai/gpt-5.1-codex	code	proprietary	openai	$1.25/$10	400k/128k	No	Yes	**70,4%**			75,10%	85,5%
GPT-5 (high)	OpenAI	openai/gpt-5	general	proprietary	openai	$1.25/$10	400k/128k	No	Yes	68,8%	87,1%	**26,5%**	**79,33%**	84,6%
GPT-5 Codex (high)	OpenAI	openai/gpt-5-codex	code	proprietary	openai	$1.25/$10	400k/128k	No	Yes	69,4%	86,5%	25,6%	78,24%	84,0%
GPT-5 mini (high)	OpenAI	openai/gpt-5-mini	general	proprietary	openai	$0.25/$2	400k/128k	No	Yes	59,8%	83,7%	19,7%	75,31%	83,8%
Gemini 2.5 Pro	Google	google/gemini-2.5-pro	general	proprietary	google-vertex/global	$1.25/$10	1M/64k	Yes	Yes	46,8%	86,2%	21,1%	71,92%	80,1%
o4-mini (high)	OpenAI	openai/o4-mini-high	general	proprietary	openai	$1.10/$4.40	200k/100k	Yes	Yes	33,4%	83,2%	17,5%		85,9%
o3	OpenAI	openai/o3	general	proprietary	openai	$2/$8	200k/100k	No	Yes	49,8%	85,3%	20,0%		80,8%
gpt-oss-120B (high)	OpenAI	openai/gpt-oss-120b	general	open-weight	google-vertex	$0.09/$0.36	131,1k/131,1k	Yes	Yes	25,6%	80,8%	18,5%	55,56%	**87,8%**
gpt-oss-20B (high)	OpenAI	openai/gpt-oss-20b	general	open-weight	hyperbolic	$0.04/$0.04	131,1k/131,1k	Yes	Yes		74,8%	9,8%		77,7%
Qwen3 Coder 30B	Alibaba	qwen/qwen3-coder-30b-a3b-instruct	code	open-weight	nebius/fp8	$0.10/$0.30	262,1k/262,1k	Yes	No	51,6%	70,6%	4,0%		40,3%
GLM 4.5 Air	Zhipu AI	z-ai/glm-4.5-air	general	open-weight	novita/bf16	$0.104/$0.68	131,1k/98,3k	Yes	Yes		81,5%	6,8%	60,53%	68,4%
GLM-4.6 (Reasoning)	Zhipu AI	glm-4.6	general	open-weight	novita/bf16	$0.48/$1.76	204,8k/131,1k	Yes	Yes	56,0%	82,9%	13,3%	71,22%	69,5%
Magistral Small 1.2	Mistral AI	magistral-small-2509	general	open-weight		$0.5/$1.5	128k	Yes	Yes		76,8%	6,1%		72,3%
Kimi K2 Thinking	Moonshot AI	kimi-k2-thinking	general	open-weight	google-vertex	$0.60/$2.50	262,1k/262,1k	Yes	Yes	57,0%	84,8%	22,3%	71,56%	85,3%
Claude Sonnet 4.5 (Thinking)	Anthropic	claude-sonnet-4.5	code	proprietary	google-vertex	$3/$15	1m/64k	No	Yes	69,8%	**87,5%**	17,3%	78,26%	71,4%
Claude Opus 4.5 (Thinking)	Anthropic	claude-opus-4.5	code	proprietary	anthropic	$5/$25	200k/32k	No	Yes	**74,2%**	**89,5%**	**28,4%**	**80,14%**	**87,1%**
DeepSeek R1 0528	DeepSeek	deepseek/deepseek-r1-0528	general	open-weight	nebius/fp8	$0.80/$2.40	163,8k/163,8k	Yes	Yes	57,6%	84,9%	14,9%	69,41%	77,0%
Grok 4	xAI	grok-4	general	proprietary	xai	$3/$15	256k/256k	No	Yes	58,6%	86,6%	23,9%	72,84%	81,9%
Grok 4 Fast (Reasoning)	xAI	grok-4-fast	general	proprietary	xai	$0.20/$0.50	2m/30k	No	Yes	52,4%	85,0%	17,0%	70,10%	83,2%

Date	26.11.2025

Benchmarks used	
SWE-Bench	https://www.vals.ai/benchmarks/swebench
	https://www.swebench.com
MMLU-Pro	https://artificialanalysis.ai/evaluations/mmlu-pro
HLE	https://artificialanalysis.ai/evaluations/humanitys-last-exam
LiveBench	https://livebench.ai/
LiveCodeBench	https://artificialanalysis.ai/evaluations/livecodebench
	https://www.vals.ai/benchmarks/lcb

Notes
Price and context window were taken from official documentation or from OpenRouter
Empty cells mean that no values were found for these models
Magistral Small 1.2 is not available in the OpenRouter
The models marked in green were selected for furher research

Рис. 2.2. Порівняльний аналіз моделей

На основі зведеного порівняння було прийнято рішення щодо 5 фінальних моделей, які залучаються до експерименту. Ними стали:

1) Google Gemini 3 Pro [46] – новітня флагманська модель від Google (станом на грудень 2025 р.), що демонструє найвищі результати майже на всіх бенчмарках. Зокрема, вона серед лідерів у генерації коду (понад 91% на LiveCodeBench, понад 71% на SWE-Bench), а також має величезний контекст (до 1 млн токенів) і підтримує багатомодальність. Хоча Gemini 3 пропрієтарна і дорога у використанні, вона слугує орієнтиром "верхньої планки" якості.

2) OpenAI GPT-5 mini [82] – модель від OpenAI, що представляє компроміс між продуктивністю і вартістю. За даними бенчмарків, ця "молодша" модель п’ятого покоління набирає дещо менше балів, ніж топові GPT-5.1 чи Gemini, але все ще

перевершує більшість попередників і конкурентів. Важливо, що її цінова доступність набагато вища: вартість генерації токенів у кілька разів нижча, ніж у флагманів. Отже GPT-5 mini дозволить перевірити, чи можна отримати прийнятну якість автогенерації тестів без використання найдорожчих ресурсів.

3) Alibaba Qwen3 Coder 30B [7] – спеціалізована модель для програмування, відкрита і доступна для розгортання локально. За результатами дослідження AMD, Qwen3 Coder 30B визначена як найкраща локальна LLM для коду на середину 2025 р. [141, 8]. Хоча на обраних бенчмарках її показники досить низькі (40% на LiveCodeBench, 51% на SWE-Bench), включення Qwen3 забезпечує репрезентацію відкритих локальних моделей. Цікаво й те, що Qwen3 не має явного режиму міркування chain-of-thought, тож її приклад покаже, як модель "традиційної" архітектури справляється з генеруванням тестів.

4) Zhipu AI GLM-4.5-Air [147] – ще одна відкрита модель для коду, рекомендована дослідженням AMD [141, 8]. GLM-4.5-Air має 106 млрд параметрів і оптимізована для запуску на споживчому обладнанні (через 8-бітні стискання). У бенчмарках вона зазвичай дещо випереджає Qwen3 Coder 30B, однак поступається закритим системам (68% на LiveCodeBench, 60% на LiveBench).

5) DeepSeek DeepSeek R1 0528 [33] – модель з відкритими вагами від компанії DeepSeek (версія релізу від 28.05.2025). Ця LLM цікава тим, що позиціонується як відкрита альтернатива комерційним флагманам, у деяких задачах вона наближається до показників Gemini 2.5 Pro та інших лідерів. Включення DeepSeek R1 0528 до п’ятірки дозволяє побачити, наскільки відкритий підхід виправдав себе і чи може така модель перевершити пропрієтарних конкурентів у генеруванні модульних тестів.

Обрані моделі повністю відповідають початковим вимогам до репрезентативності: серед них є і загального призначення (Gemini 3 Pro, GPT-5 mini, DeepSeek R1 0528, GLM-4.5-Air), і спеціалізована на коді (Qwen3 Coder 30B); є закриті (Google, OpenAI) і відкриті проєкти (DeepSeek, Alibaba, Zhipu AI); усі походять від різних розробників і мають різні архітектурні особливості. На жаль, до фінальної вибірки не увійшли деякі інші сильні моделі, які могли б бути цікавими –

зокрема, Moonshot AI Kimi K2 Thinking, Anthropic Claude Sonnet 4.5, Anthropic Claude Opus 4.5, Zhipu AI GLM-4.6, xAI Grok 4, xAI Grok 4 Fast. Вони також демонструють високі результати на профільних бенчмарках і мали б науковий інтерес. Однак обмежені ресурси часу і коштів змусили звузити коло експерименту. Ці моделі залишаються як напрям для можливого майбутнього дослідження.

Підсумовуючи, на цьому етапі роботи було виконано великий обсяг підготовчої аналітики: досліджено стан галузі LLM у задачах кодування, сформовано критерії оцінки та на їх основі відібрано п’ять конкретних моделей для подальшого експериментального вивчення. Таким чином, закладено основу для наступних кроків – конструювання промптів, генерування тестів обраними моделями та оцінювання отриманих результатів. Ретельність відбору моделей і метрик гарантує, що подальші висновки праці матимуть надійну та об’єктивну основу.

2.2 Побудова промптів

Для успішного застосування великих мовних моделей у генеруванні модульних тестів необхідно сформувати якісний текстовий запит (prompt), який чітко визначає завдання для моделі. Prompt-інженерія – це нова галузь, що поєднує мистецтво і науку розробки оптимальних запитів з метою спрямування моделей на отримання бажаних результатів [99, 100, 98]. Сам prompt являє собою текстовий вхід, наданий моделі, який може бути реалізований у вигляді питання, інструкції, фрагмента коду тощо – від якості цього вводу безпосередньо залежить релевантність і точність відповіді моделі. Відповідно до загальноприйнятого принципу, "якісний запит = якісний результат": добре продумані й детальні промпти дають змогу великим мовним моделям краще зрозуміти наміри користувача і згенерувати коректнішу відповідь. Саме тому розробці промпта в межах даного дослідження передував аналіз актуальних рекомендацій та підходів у сфері prompt-інженерії [99, 100, 98, 96, 113, 76].

2.2.1 Підходи до побудови промптів

На сьогодні існує низка усталених підходів і технік розробки запитів для LLM, кожна з яких має свої особливості та сфери застосування. Нижче розглянуто основні з них:

1) Zero-shot (нульовий приклад)

Модель отримує лише інструкцію або запитання без надання будь-яких додаткових прикладів чи контексту. Це найпростіший тип запиту, коли система повинна виконати завдання, спираючись лише на своє загальне навчання. Zero-shot підхід часто застосовується для тривіальних або добре визначених завдань, де зайвий контекст не потрібен.

2) One-shot/Few-shot (один або декілька прикладів)

Моделі надаються один або кілька прикладів входу-виходу перед формулюванням основного запиту. Таким чином система отримує зразок бажаного результату, що допомагає їй налаштуватися на виконання завдання. Few-shot підказки особливо корисні для більш складних задач, оскільки демонструють формат відповіді і зменшують імовірність неоднозначностей у розумінні завдання.

3) Ланцюжок міркувань (Chain-of-Thought, CoT)

Ця техніка заохочує модель генерувати проміжні кроки логічних міркувань перед формуванням фінальної відповіді. Розбиваючи складне завдання на послідовність простіших кроків, модель підвищує вірогідність отримати правильну та обґрунтовану відповідь. Відомо, що CoT-підказки допомагають LLM вирішувати задачі, які потребують багатоетапних розрахунків або логічного висновку, оскільки модель ніби "пояснює сама собі" хід рішення.

4) Системний промпт

Це підхід, при якому ще до основного запиту моделі надається системне повідомлення – високорівнева інструкція щодо бажаної поведінки, стилю або ролі моделі. Системний промпт встановлює контекст і "правила гри" для подальшої розмови, причому такі інструкції мають пріоритет над наступними запитами користувача. Наприклад, через системний промпт можна задати моделі тон

спілкування ("ти – експерт з тестування") або обмеження ("ігноруй незначні деталі безпеки, фокусуйся на функціональності"), що залишатиметься в силі протягом усієї сесії.

5) Рольовий промпт

У цій техніці модель явно інструктується прийняти на себе певну роль або персону при генерації відповіді. Рольова підказка дозволяє контролювати тон, стиль і поведінку моделі, наближаючи її відповіді до бажаного формату. Наприклад, моделі можна задати роль "досвідченого інженера-тестувальника" – тоді її відповіді будуть більш формальними і професійними, ніж за замовчуванням. Завдяки гнучкості LLM, рольові інструкції можуть задавати будь-який образ (вчителя, критика, системного аналітика тощо), що допомагає адаптувати вихід моделі під конкретні потреби.

6) Контекстуальний промпт

Цей підхід передбачає надання моделі детального контексту, необхідного для виконання завдання. До запиту включається релевантна інформація про середовище або вхідні дані – наприклад, опис API, витяг з документації або сам код програми – щоб модель краще розуміла умови задачі. Забезпечення достатнього контексту у prompt-інструкції істотно підвищує точність і доречність згенерованих відповідей. Інакше кажучи, замість того щоб очікувати, що модель "здогадається" про необхідні деталі, їх явно повідомляють у тексті запиту.

7) Дерево думок (Tree of Thoughts, ToT)

Ця просунута техніка є розвитком ідеї ланцюжка міркувань. Дерево думок надає моделі можливість розгалужувати хід рішення на декілька альтернативних шляхів, оцінювати їх і відбирати найперспективніший. По суті, модель вибудовує структуру у вигляді дерева, де вузли – це проміжні "думки" (кроки розв’язання), а гілки відповідають різним варіантам розвитку рішення. Такий підхід імітує стратегії, які людина застосовує при вирішенні складних проблем: паралельно продумуються різні можливості, неперспективні відкидаються, після чого обрана гілка доводиться до кінця. Метод ToT дозволяє LLM поетапно досліджувати і відшукувати рішення

складних задач у структурований спосіб, що підвищує шанси знайти правильну відповідь навіть у багатокрокових сценаріях.

8) Step-back prompting ("крок назад")

Це техніка, яка заохочує модель спочатку зробити крок вбік від конкретного питання і проаналізувати його на більш абстрактному, узагальненому рівні, перш ніж намагатися дати відповідь. Іншими словами, модель спочатку формулює загальні принципи чи високорівневі концепції, релевантні до задачі, і лише потім переходить до її безпосереднього розв’язання. Такий "крок назад" допомагає уникнути поверхневих помилок та забезпечує більш глибоке розуміння проблеми. Дослідження показують, що step-back prompting покращує здатність моделей дотримуватися правильного шляху міркування, зменшуючи кількість логічних хиб. Цей метод особливо ефективний для завдань, що потребують складних доводів або роботи з великими обсягами конкретних деталей – абстрагування на початку дозволяє моделі побачити "загальну картину" і організувати знання перед пошуком рішення.

9) Self-consistency (самоузгодженість)

Self-consistency не стільки визначає вміст промпта, скільки стосується стратегії отримання кінцевої відповіді від моделі. Ідея полягає в тому, щоб згенерувати декілька відповідей (або ланцюжків міркувань) на один і той самий запит, а потім обрати серед них найбільш узгоджену або найчастіше повторювану відповідь. Такий підхід ґрунтується на припущенні, що складне завдання може мати кілька різних правильних шляхів розв’язання, але кінцевий результат (відповідь) зазвичай єдина. Вибір відповіді більшістю ("голосування" серед кількох незалежно згенерованих рішень) підвищує точність моделі на задачах логічного та арифметичного характеру. Метод самоузгодженості фактично усуває залежність від одного, можливо хибного, ланцюжка думок і зменшує вплив випадковості при виборі наступного токена, що позитивно позначається на якості кінцевого результату.

10) ReAct (Reason + Act)

Підхід ReAct об’єднує в рамках одного промпта два аспекти: міркування моделі та виконання дій у зовнішньому середовищі. За цією методикою модель генерує не лише вербальні пояснення (thoughts) до задачі, але й спеціальні дії, які дозволяють їй запитувати додаткову інформацію або виконувати команди в ході міркувань. Вихід моделі при ReAct-підказці складається з послідовності "Думка → Дія → Спостереження → … → Відповідь". Міркування допомагають моделі формувати і оновлювати план розв’язання, а дії дають змогу звертатися до зовнішніх джерел даних (наприклад, баз знань, пошукових систем) для отримання потрібної інформації. Таким чином, ReAct-підхід перетворює LLM на своєрідного агента, що одночасно роздумує і взаємодіє зі світом. Це дозволяє розв’язувати складні питання, де одних лише внутрішніх знань моделі недостатньо: під час експериментів ReAct продемонстрував вищу точність і кращу інтерпретованість рішень у завданнях запитання-відповідь та інтерактивного планування порівняно з традиційними методами без явних дій.

Перераховані техніки нерідко комбінуються для досягнення оптимального результату. Зокрема, існують гібридні підходи, такі як Zero-shot CoT – коли до прямого запиту без прикладів додається фраза на кшталт "Давай міркувати покроково", що поєднує zero-shot і CoT ефекти. У практичних застосуваннях вибір стратегії формується з урахуванням специфіки завдання: наприклад, для генерації програмного коду чи тестів важливо надати контекст (код, середовище) і чіткі вимоги, тому доречними будуть contextual prompting і system/role prompting; водночас для перевірки логіки або розв’язання алгоритмічних задач може знадобитися CoT або навіть ToT для перебору можливих сценаріїв.

2.2.2 Побудова промпта для генерації тестів

Спираючись на вищезгадані найкращі практики і техніки, було розроблено спеціалізований prompt для проведення дослідження. Зважаючи на цільове завдання – генерацію модульних тестів за фрагментом C#-коду – в промпті вирішено застосувати базові, але ефективні підходи: zero-shot (жодних готових прикладів

тестів не надається, модель сама продукує рішення з нуля), а також system, role та contextual prompting (чітко задається роль моделі, високорівневі інструкції щодо стилю роботи та детальний контекст). Вибір було зроблено на користь цих методик, оскільки вони є фундаментальними і відносно простими у реалізації – фактично, слугують відправною точкою для більш складних сценаріїв. Такий початковий "базовий" промпт закладає основу для подальших експериментів і легко розширюється іншими техніками у разі потреби. При його створенні дотримано загальних рекомендацій щодо формулювання запитів: вимоги сформульовані максимально чітко, присутній необхідний контекст, задано формат відповіді та інші обмеження, щоб модель однозначно зрозуміла поставлене завдання.

Розроблений промпт написано англійською мовою з використанням синтаксису Markdown [68] – це підвищує ймовірність того, що різні моделі правильно інтерпретують його структуру. Для зручності сприйняття та відповідності кращим практикам, текст запиту було поділено на шість логічних блоків, кожен з яких виконує окрему функцію:

1) Роль – в цьому блоці модель налаштовується на виконання певної ролі. Зокрема, їй вказується діяти як помічник-розробник, експерт із розробки модульних тестів. Такий вступ задає тон подальшої роботи і фактично є реалізацією рольового промптингу (model act as unit testing assistant).

2) Завдання – містить опис завдання, яке модель повинна виконати. В даному випадку вказано, що потрібно згенерувати модульні тести (unit tests) для наведеного фрагмента C#-коду.

3) Середовище – уточнює контекст програмування: зазначено, що код написано мовою C# під платформу .NET [73], а для тестування слід використовувати фреймворк xUnit [146] і бібліотеку підстановок NSubstitute [80]. Цей блок забезпечує модель необхідною технічною інформацією про середовище виконання тестів і доступні інструменти, аби згенеровані тести були сумісні з реальними умовами.

4) Вимоги – містить перелік правил та критеріїв, яких модель має дотримуватися при генерації тестів. Сюди входять вимоги щодо змісту і стилю

тестового коду: наприклад, надавати осмислені імена тестових методів, дотримуватися кращих практик написання модульних тестів, уникати тривалого виконання тощо. Чітко сформульовані вимоги гарантують, що модель не відійде від теми і не згенерує нерелевантний або неправильний код.

5) Формат відповіді – визначає, в якому вигляді модель повинна надати результати. В нашому випадку зазначено, що відповідь має містити лише вихідний код згенерованих тестів (без зайвих пояснень), оформлений у відповідному синтаксисі (весь код в межах одного блоку, щоб його зручно було скопіювати). Цей блок запобігає виведенню моделями будь-якого зайвого тексту (коментарів, міркувань) і забезпечує зручність інтеграції отриманого коду тестів у проєкт.

6) Код – заключний блок, що містить сам фрагмент вихідного коду C#, для якого потрібно згенерувати тести. Код надається повністю або у релевантній частині, щоб модель могла проаналізувати його функціональність. Це є основний вхід для моделі: на основі цього коду вона генеруватиме тестові методи. Важливо, що код оформлено у Markdown-блоці, аби модель коректно його сприйняла як вхідні дані, а не частину інструкції.

Повний текст даного промпта наведено у Додатку А.

2.2.3 Промпт для оцінки читабельності коду

Окрім промпта, призначеного безпосередньо для генерації тестів, було створено допоміжний запит для оцінювання читабельності згенерованих тестових сценаріїв. Цей крок виконується з використанням агентного підходу: інша мовна модель отримує згенерований код тестів та інструкцію розрахувати показник їх читабельності. Для автоматизації такої оцінки сформовано окремий prompt англійською мовою, також оформлений у Markdown-стилі і детально структурований. Він складається із семи логічних блоків:

1) Роль – задає моделі роль рецензента або аналітика коду, що буде оцінювати якість і читабельність поданих тестів.

2) Завдання – формулює, що саме потрібно зробити: обчислити індекс читабельності для наведеного коду модульних тестів та надати оцінку згідно з визначеною методикою.

3) Середовище – уточнює контекст, аналогічно до попереднього промпта (мова C#, платформа .NET), щоб модель розуміла, що йдеться про код на C# і могла застосувати відповідні знання про стиль кодування на цій мові.

4) Опис метрики – пояснює, що таке індекс читабельності (readability ndex), на основі якої формули або критеріїв він обчислюється, та які значення свідчать про "добру" чи "погану" читабельність коду. Таким чином, модель отримує формальне визначення метрики, щоб коректно виконати розрахунок.

5) Опис читабельного коду – наводить характеристики ідеального, з точки зору читабельності, коду. Тут перераховано ознаки "хорошого" коду (зрозумілі імена змінних, достатнє коментування, простота і ясність логіки, дотримання стилістичних конвенцій тощо), щоб модель мала еталон для порівняння. Цей блок допомагає нейромережі зрозуміти, на що звертати увагу при аналізі тестів.

6) Формат відповіді – визначає, в якому вигляді модель-рецензент повинна надати результат оцінювання. Як правило, це числове значення індексу та коротке пояснення щодо читабельності.

7) Код – містить сам згенерований код модульних тестів, які треба проаналізувати. Цей блок буде динамічно підставлятися в промпт після того, як цільова модель згенерує тестові сценарії. Таким чином, модель-рецензент отримує на вхід конкретний код для оцінки.

Даний prompt для оцінки читабельності застосовує ті самі техніки zero-shot, system, contextual та role prompting: модель не отримує прикладів оцінювання (лише формальне визначення метрики), їй задається роль аналітика, через системні інструкції пояснюються критерії аналізу, а також задається конкретний контекст у вигляді середовища та тестового коду. Важливо зазначити, що метрика індексу читабельності та підхід до її розрахунку будуть детальніше розглянуті у розділі "Проведення дослідження", оскільки це стосується методики оцінювання результатів. Повний текст промпта для оцінки читабельності наведено у Додатку Б.

Підбиваючи підсумки, на даному етапі було здійснено критичний огляд сучасних підходів prompt-інженерії та сформовано два робочі запити для потреб дослідження. Перший – основний промпт генерації модульних тестів – розроблено з урахуванням базових технік (zero-shot, system/contextual/role prompting) і структуровано на логічні блоки, що повністю визначають роль моделі, контекст, вимоги та формат відповіді. Другий – допоміжний промпт оцінювання читабельності коду – покликаний забезпечити об'єктивний вимір якості згенерованих тестів за встановленою метрикою. Обидва промпти ретельно деталізовано і підготовлено для подальшого використання в експериментальній частині роботи. Результатом цього етапу є наявність чітких інструкцій для мовних моделей, що закладає основу для успішного проведення дослідження та отримання релевантних результатів на наступних етапах.

2.3 Формування набору даних

Для об'єктивного оцінювання якості згенерованих модульних тестів потрібен відповідний набір вихідного коду, на основі якого проводитиметься експеримент над моделями. На даному етапі було сформовано спеціалізований датасет із прикладів коду на C#, що слугує основою для подальшого дослідження. Якість та репрезентативність цього датасету є критично важливими, адже тільки різноманітні та реалістичні сценарії дозволяють всебічно перевірити можливості великих мовних моделей у генерації тестів. Враховуючи обмежені часові рамки дослідження, обсяг датасету обмежено двадцятьма прикладами коду. Хоча така вибірка є порівняно невеликою, вона відповідає практиці створення компактних, але показових наборів даних у подібних дослідженнях. Кожний із обраних прикладів являє собою самодостатній фрагмент програмного коду, який успішно компілюється, що дозволяє уникнути помилок під час генерації та виконання тестів.

До складу датасету ввійшли 20 різноманітних фрагментів коду на C#. Кожен приклад є повноцінним, коректно компільованим фрагментом програми, який містить один основний клас для тестування. Для забезпечення працездатності цього

класу до прикладу можуть додаватися й інші необхідні компоненти (такі як інтерфейси чи допоміжні класи-моделі), проте весь код кожного прикладу є самодостатнім. Це відповідає рекомендованим підходам до відбору програмних проєктів для тестування, згідно з якими всі залежності повинні бути включені, а код має бути цілісним і без синтаксичних помилок.

Датасет сформовано з використанням двох основних джерел коду: власних розробок автора та зовнішніх відкритих проєктів. Значна частина прикладів узята з особистих напрацювань (зокрема, EmployeeAccountingSystem [37], QuickNoteVault [102], DirectoryHierarchy [48], Internet-Auction [54], WalksInfoViberBot [129]), що забезпечує відповідність коду предметній області дослідження. Доповненням до них стали фрагменти із відкритих репозиторіїв на платформі GitHub, які мають відкриті ліцензії та є загальнодоступними. При відборі зовнішніх проєктів перевага надавалася надійним репозиторіям, що підтримуються спільнотою, таким як Aqua.StringHelpers [11], TheAlgorithms/C-Sharp [114], dotnetrdf [35]. Використання відкритого вихідного коду з популярних репозиторіїв під відкритими ліцензіями узгоджується з практикою формування наборів даних у сфері програмної інженерії, що гарантує легальність використання коду та його високу якість. Крім того, включення різних джерел коду забезпечує різноманітність стилів програмування, що є корисним для перевірки здатності моделей адаптуватися до різних стилістичних особливостей реалізації алгоритмів.

Для підвищення якості дослідження особливу увагу приділено репрезентативності датасету. До нього увійшли приклади коду різного рівня складності та різних типів задач. Зокрема, присутні як елементарні допоміжні класи, що містять прості методи, так і складніші класи, які реалізують бізнес-логіку з численною валідацією вхідних даних, а також класи, що містять нетривіальні математичні алгоритми. Такий підбір покликаний охопити широкий спектр сценаріїв. З одного боку, прості допоміжні функції дозволяють перевірити базову здатність моделей генерувати тести для тривіального коду. З іншого боку, складніші приклади (наприклад, алгоритми чи класи з винятками) створюють для моделей додатковий виклик, оскільки містять граничні випадки та нелінійну логіку.

Забезпечення різноманітності вхідних даних і рівнів складності відповідає загальним рекомендаціям щодо побудови тестових наборів для LLM: необхідно включати різні реальні сценарії, варіювати складність та враховувати крайні випадки для повноцінного випробування можливостей моделі.

Усі відібрані фрагменти коду було об’єднано в окремий C#-проєкт "LlmUnitTestGenerationArtifacts.Dataset". Цей проєкт розміщено у відкритому доступі на платформі GitHub [67]. Публікація датасету у репозиторії забезпечує прозорість дослідження та можливість незалежного ознайомлення з використаними прикладами коду. Також це створює умови для відтворюваності експерименту іншими дослідниками, що є важливою вимогою академічних досліджень.

Для ілюстрації різноманітності датасету нижче наведено два контрастні приклади з нього. Вихідний код фрагментів надано у Додатку В та Додатку Г відповідно. Перший приклад – клас "AuctionService", який реалізує сервіс керування аукціонами. Цей клас містить кілька публічних методів ("Bet", "OpenAuction", "CloseAuction", "GetAuction", "GetAuctions") і використовує патерн Unit of Work через інтерфейс "IUnitOfWork" для взаємодії з базою даних аукціонів. "AuctionService" виконує значну кількість перевірок вхідних даних: наприклад, якщо ідентифікатор аукціону неприпустимий (<=0) або об’єкт аукціону не знайдений, генерується виняток "InvalidIdException"; якщо ім’я покупця не задано (null) – генерується "InvalidNameException"; якщо ж робиться спроба зробити ставку на аукціон, який ще не розпочато, вже завершено або якщо запропонована ставка не перевищує поточну, генеруються відповідні винятки "InvalidAuctionException" з інформативними повідомленнями. Після проходження всіх перевірок метод "Bet" оновлює стан аукціону (фіксує нову ставку та лідера) і зберігає зміни через виклик "_database.Commit()". Подібним чином методи "OpenAuction" та "CloseAuction" змінюють стан аукціону (початок або завершення торгів) лише за умови валідності операції, інакше генерують винятки, що сигналізують про некоректний стан. Таким чином, клас "AuctionService" демонструє типову бізнес-логіку з правилами та обмеженнями, де важливо перевірити в тестах як нормальні сценарії роботи (правильне відкриття, закриття аукціону, прийом ставки), так і реакцію на

некоректні дії (впевнитись, що заборонені ситуації спричиняють винятки). Наявність у датасеті такого класу дозволяє оцінити, наскільки добре модель генеруватиме тести для коду, що містить багато умовних конструкцій та винятків.

Другий приклад – статичний клас "LUPDecomposition", який реалізує алгоритм факторизації матриці методом LU з частковим вибором, тобто з використанням матриці перестановок (LUP-розкладання). Цей алгоритм полягає у представленні заданої матриці як добутку нижньої і верхньої трикутних матриць з врахуванням перестановок рядків. Клас "LUPDecomposition" надає метод "Decompose(double[,] matrix, out double[,] L, out double[,] U, out int[] P)", що виконує розкладання: на виході формується нижня трикутна матриця L, верхня трикутна U та масив індексів P, який уособлює матрицю перестановок. Алгоритм передбачає пошук максимального елемента в кожному стовпці для забезпечення чисельної стійкості (частковий вибір головного елемента) та обмін рядків матриці відповідно до цього вибору. Якщо на деякому кроці алгоритму в стовпці виявляється нульовий максимальний елемент, метод генерує виняток "InvalidOperationException" з повідомленням українською мовою "Матриця є виродженою." (що означає, що матриця не може бути розкладена). Окрім основного методу "Decompose", клас містить допоміжні методи "GetMatrix" та "GetPermutationMatrix", які формують текстове представлення матриці або матриці перестановок у вигляді рядків (для зручного виведення результатів розкладання). Цей приклад значно відрізняється від попереднього: він належить до галузі чисельних алгоритмів і містить вкладені цикли, математичні обчислення та спеціальні випадки (вироджена матриця). Наявність у датасеті подібного алгоритмічного фрагмента коду дозволяє перевірити здатність моделей генерувати тести для обчислювальних задач. Зокрема, модель повинна згенерувати тести, що перевіряють правильність роботи алгоритму на різних матрицях (наприклад, на звичайній матриці, на одиничній чи нульовій матриці, на виродженій матриці, для якої очікується виникнення винятку), а також перевірити коректність формування вихідних матриць L та U і вектора P. Такий приклад сприяє оцінці можливостей моделі щодо розуміння та тестування складних логічних структур і числових крайніх випадків.

У підсумку, на даному етапі було підготовлено та структуровано набір із 20 фрагментів вихідного коду на C#, призначених для подальшої генерації модульних тестів. Датасет сформовано з урахуванням принципів репрезентативності та різноманітності: до нього ввійшли приклади з різних джерел (власні проєкти та перевірені відкриті репозиторії) і різного рівня складності (від простих функцій до складних алгоритмів). Усі приклади коду є коректними, самодостатніми та готовими до написання тестів для них. В результаті виконання цього етапу отримано цілісний відкритий датасет, який слугуватиме базою для експериментального порівняння можливостей великих мовних моделей у генерації модульних тестів.

2.4 Проведення дослідження

Щоб об’єктивно порівняти різні великі мовні моделі, необхідно було попередньо окреслити набір чітких критеріїв (метрик) якості згенерованих тестів. Після цього було розроблено засоби автоматизації експерименту: створено спеціальний консольний застосунок для автоматизованого отримання результатів від моделей і збору значень обраних метрик. Далі було покроково проведено експеримент для кожної моделі – від генерування тестів та їх корекції до розрахунку всіх показників якості. Нижче наведено опис використаних метрик та інструментів, процес автоматизації збору даних і детальний перебіг експерименту.

2.4.1 Метрики оцінювання якості згенерованих тестів

Для оцінювання якості модульних тестів, згенерованих великими мовними моделями, було визначено кілька взаємодоповнювальних метрик. Кожна метрика відображає певний аспект якості тестового набору – від здатності виявляти дефекти в коді до зручності підтримки тестів. Вибір саме цих метрик зумовлений прагненням отримати комплексну і об’єктивну оцінку, тому вони охоплюють як ефективність тестування, так і технічну якість тестового коду та продуктивність самої моделі. Нижче подано опис кожної метрики та обґрунтування її використання у дослідженні.

1) Mutation score (мутаційна оцінка) [77, 78]

Ця метрика характеризує ефективність тестів у виявленні потенційних дефектів. Вона визначається як відсоток штучно внесених у програмний код "мутантів" (відхилень у коді), які були "вбиті" (тобто виявлені) тестами. Інакше кажучи, mutation score – це відношення кількості виявлених мутантів до загальної кількості внесених мутантів, виражене у відсотках. Чим вище цей показник (на шкалі 0–100%), тим якіснішим вважається набір тестів, оскільки він ловить більше прихованих помилок. Мутаційне тестування є відомим підходом для оцінки якості тестів, що фактично "тестує самі тести" шляхом внесення помилок у код і перевірки, чи здатні тести їх виявити. У даному дослідженні мутаційна оцінка розглядається як ключова метрика, адже вона відображає реальну спроможність згенерованих тестів знаходити дефекти. Для вимірювання цього показника використано інструмент Stryker.NET – спеціалізований фреймворк для мутаційного тестування у середовищі .NET [106]. Вибір Stryker.NET обумовлений тим, що це відкритий, загальнодоступний інструмент, який автоматично генерує мутанти в коді і аналізує результати тестування. Окрім того, Stryker.NET є наразі провідним рішенням для C# у цій галузі, широко застосовується в .NET-спільноті та надає наочні звіти про мутовані ділянки коду. Таким чином, мутаційна оцінка та обраний інструмент Stryker.NET дозволяють кількісно оцінити здатність тестів виявляти помилки у програмному коді (чим ближче показник до 100%, тим ефективніші тести з точки зору знаходження дефектів).

2) Code coverage (покриття коду) [136, 28]

Дана метрика відображає повноту тестового покриття і визначається як відсоток рядків вихідного коду, що виконуються під час прогону тестів. Іншими словами, це частка коду, перевірена тестами. Хоча високий відсоток покриття сам по собі не гарантує виявлення всіх помилок, він свідчить про те, що тестовий набір принаймні виконує більшість логіки програми. У парі з мутаційною оцінкою coverage дає більш повну картину: якщо покриття високе, але mutation score низький – це ознака слабких перевірок у тестах (тести виконують код, але не фіксують помилки). Навпаки, низьке покриття навіть за високої якості окремих тестів означає,

що частина коду взагалі не перевіряється. Тому покриття коду було включено як базова метрика якості. Для її визначення використано інструмент JetBrains dotCover – утиліту для аналізу покриття у .NET [57]. Вибір dotCover зумовлений його надійністю та популярністю: цей інструмент розроблено компанією JetBrains, він легко вбудовується в процес розробки і широко використовується для оцінки покриття в .NET-проєктах. dotCover дозволяє отримати відсоток покриття (0–100%) – чим більше, тим краще, оскільки це свідчить про більшу частку коду, перевіреного тестами.

3) Compilability index (індекс компільованості)

Даний показник відображає технічну придатність згенерованих тестів до виконання, тобто чи проходять вони компіляцію без помилок. Очевидно, що тестовий код, який не компілюється, не може бути запущений і не приносить користі, тому важливо оцінити цей аспект. Індекс компільованості задається у відсотках від 0% до 100%, причому 100% означає, що згенеровані тести скомпілювалися одразу без помилок. Розрахунок цього показника було реалізовано згідно з таким правилом: якщо всі тести для даного прикладу коду скомпілювались без проблем, то індекс = 100%; якщо в коді тестів були помилки компіляції, то від 100% віднімається по 1% за кожну хвилину, витрачену на виправлення цих помилок вручну. Наприклад, якщо на доопрацювання тестів пішло 11 хвилин, індекс компільованості для цього прикладу становить 89%. Бувають випадки, коли помилки виправити неможливо без повного переписування тесту – такі невдалі тести вилучаються і за кожен з них віднімається 10% від значення метрики. Наприклад, якщо із згенерованого набору 2 тести взагалі не вдалося виправити, то загальний індекс для прикладу буде 80% (100% - 2 × 10%). До уваги при цьому не бралися незначні правки, що не стосуються логіки: зміна простору імен ("namespace") для успішного запуску, видалення зайвих "using"-директив, а також наявність лише попереджень компілятора (warning). Таким чином, compilability index відображає якість синтаксису та структури згенерованих тестів: значення, близьке до 100%, означає, що модель генерує здебільшого коректний з точки зору

компіляції код тестів, а нижчі значення свідчать про часті синтаксичні або структурні помилки у відповідях моделі.

4) Validity index (індекс валідності)

Ця метрика характеризує коректність самих тестів з точки зору проходження – тобто відсоток тестів, які виконуються і проходять (стають "зеленими"). Оскільки для дослідження було сформовано спеціальний еталонний датасет (кожен фрагмент програмного коду, для якого генерувалися тести, є завідомо правильним і містить коректну бізнес-логіку), усі згенеровані тести повинні проходити успішно. Іншими словами, якщо модель правильно зрозуміла завдання, її тести не мають провалюватися на коректному коді. На практиці деякі згенеровані тести не проходили – це може означати, що модель неправильно інтерпретувала вимоги до коду або додала зайві перевірки, які не відповідають реальній логіці. Індекс валідності обчислюється як відношення кількості тестів, що пройшли успішно, до загальної кількості виконаних тестів, помножене на 100%. Шкала 0–100%, де 100% означає, що всі тести пройшли, а нижчі значення свідчать про наявність тестів, які "падають" (не виконуються успішно). Ця метрика доповнює попередні: наприклад, модель може згенерувати тести з високим покриттям, але якщо значна їх частина не проходить, то практична користь таких тестів сумнівна. Validity index, таким чином, оцінює коректність прогнозів моделі щодо поведінки еталонного коду.

5) Readability index (індекс читабельності)

Окрім технічної правильності, важливим аспектом якості тестів є зручність їх читання і підтримки. Індекс читабельності покликаний відобразити, наскільки легко іншому розробнику зрозуміти згенеровані тести, чи дотримані в них принципи чистого коду, зрозумілі імена, логічна структура тощо. Показник вимірюється у відсотках від 1 до 100%, розподілених по інтервалах:

- 1–20% – дуже погана читабельність (код тестів майже неможливо читати і підтримувати);
- 21–40% – погана читабельність (багато безладу, погане форматування, складно зрозуміти призначення тестів);

- 41–60% – посередня читабельність (загалом зрозуміло, але є суттєві недоліки);
- 61–80% – хороша читабельність (структура і найменування здебільшого зрозумілі, недоліки некритичні);
- 81–100% – відмінна читабельність (тести легко читати, вони логічні, добре структуровані і прості в підтримці).

Оцінювання читабельності коду є доволі суб’єктивним завданням, тому для визначення цього індексу застосовано агентний підхід. Було залучено п’ять великих мовних моделей-агентів і доручено їм проаналізувати якість коду тестів. Такий підхід дозволяє отримати усереднену оцінку від декількох "експертів" і знизити суб’єктивність окремої моделі. В якості агентів було відібрано 5 флагманських моделей від різних виробників, які не брали участі в основному дослідженні: DeepSeek DeepSeek-V3.2 [34], OpenAI GPT-5.1 [85], Anthropic Claude Sonnet 4.5 [56], xAI Grok 4.1 Fast [144], Moonshot AI Kimi K2 Thinking [75]. Критерій відбору – високі показники цих моделей у відкритому рейтинговому бенчмарку LiveCodeBench [15]. Таким чином, обрані агент-моделі є сучасними рішеннями з відмінними навичками аналізу коду, що забезпечує надійність їх оцінок.

Для визначення індексу читабельності було підготовлено окремий промпт, в якому детально описано роль і завдання агента. У ньому моделям-агентам пропонувалося проаналізувати наданий фрагмент коду тестів і видати чисельну оцінку читабельності (від 1 до 100) разом з поясненнями. Даний промпт було розроблено заздалегідь, його зміст докладно розглядався у розділі "Побудова промптів". В ході експерименту для кожного блоку згенерованих тестів (тобто для кожного вхідного прикладу програмного коду) послідовно виконувалося 5 запитів – по одному до кожної з п’яти агент-моделей. Отримані від них п’ять оцінок читабельності усереднювалися (вираховувалося середнє арифметичне), і цей середній бал брався як підсумковий readability index для даного блоку тестів. Слід зазначити, що індекс читабельності не враховує правильність бізнес-логіки тестів чи повноту покриття – він оцінює лише якість стилю і структури коду (чи легко читати і розуміти тести). Таким чином, навіть абсолютно логічно некоректний тест може

отримати високий бал читабельності, якщо написаний зрозуміло; і навпаки, коректний за перевірками тест може мати низьку читабельність через поганий стиль написання. Врахування цієї метрики є важливим з огляду на те, що в реальних умовах супроводу проєкту тестовий код має бути зрозумілим для команди розробників – код має не лише працювати правильно, а й бути легким для читання іншими програмістами.

6) Execution time (час виконання)

Ця метрика стосується продуктивності самих великих мовних моделей. Вона вимірює, скільки часу проходить від моменту відправлення запиту до моделі до моменту отримання від неї повної згенерованої відповіді (коду тестів). Очевидно, що менший час відповіді є кращим, адже свідчить про швидкість моделі і потенційно вищу продуктивність при інтеграції в інструменти розробки. Час виконання вимірювався в мілісекундах. Цей показник не впливає на якість самих тестів, але впливає на загальну ефективність використання моделі: надто повільна модель навіть за високої якості тестів може бути непрактичною. Отримані значення часу виконання кожного окремого запиту надалі усереднювалися для оцінки середнього часу відповіді конкретної мовної моделі.

7) Prompt tokens (токени запиту), completion tokens (токени відповіді), total tokens (всі токени)

Ці три кількісні метрики характеризують обсяг вхідних і вихідних даних при генеруванні тестів, виміряний у токенах (уніфікованих одиницях тексту, з якими оперує модель). Prompt tokens – кількість токенів у тексті запиту, надісланому моделі (тобто розмір підказки, що включає опис завдання і фрагмент вихідного коду). Completion tokens – кількість токенів у відповіді, згенерованій моделлю (сюди входять як власне токени коду тестів, так і службові токени, які модель використала під час міркування). Total tokens – сумарна кількість токенів, що є сумою prompt + completion. Ці метрики, окрім відображення довжини запиту і відповіді, мають практичне значення з точки зору вартості та обмежень моделі. Більшість API великих мовних моделей тарифікують використання саме за кількістю вхідних і вихідних токенів. Тому розмір промпта та об’єм відповіді впливають на вартість

генерації тестів. До того ж різні моделі мають різні ліміти контексту – якщо згенерований тест надто великий (багато токенів), це може стати проблемою для подальшої обробки або для моделі-агента, яка оцінює читабельність. У рамках даного дослідження кількість токенів і для запиту, і для відповіді, і загальна автоматично фіксувалися за допомогою функцій OpenRouter API [91], через який здійснювались запити. Зокрема, дана платформа надає можливість отримати детальну статистику щодо використання токенів – скільки було у prompt, скільки у completion, – а також обрахувати суму і навіть вартість запиту. Таким чином, для кожного згенерованого блоку тестів було отримано числові характеристики обсягу інформації, з яким працювала модель.

8) Total cost (вартість запиту)

Це похідна метрика, яка безпосередньо залежить від попередніх (кількості токенів) і тарифів моделі. Вартість вимірюється у доларах США і відображає, у скільки обходиться один запит до моделі з певним числом токенів. Деякі моделі є безкоштовними для дослідницького використання, інші – комерційні і стягують плату за кількість опрацьованих токенів. OpenRouter, що використовувався у роботі, надавав можливість отримувати інформацію про вартість кожного запиту в режимі реального часу. Дана метрика є важливою при виборі оптимальної моделі: інколи модель, що генерує якісніші тести, може виявитися значно дорожчою в експлуатації, ніж трохи слабша, але дешевша модель. Тому порівняння total cost між моделями дозволяє врахувати також економічну доцільність їх використання.

Значення кожної з наведених метрик у ході експерименту визначалися для кожного окремого фрагменту коду, на основі якого генерувалися тести. Таким чином, для кожного прикладу програмного коду і кожної моделі було отримано набір показників. Після цього для узагальненої оцінки по моделі обчислювалося середнє арифметичне значення кожної метрики за всіма прикладами. Іншими словами, для кожної моделі підсумковий показник, наприклад, покриття коду – це середнє значення відсотка покриття по всіх згенерованих нею тестових наборах. Аналогічно обчислювалися середні значення часу виконання, вартості тощо. Такий підхід дозволяє згладити випадкові коливання на окремих прикладах і об’єктивніше

порівняти моделі між собою за кожним критерієм. Усі обрані метрики разом дають всебічну характеристику згенерованих тестів: метрики на кшталт mutation score і coverage відображають ефективність та повноту тестування, показники compilability та validity – технічну правильність, readability – якість стилю, а час, токени і вартість – продуктивність та економічність використання моделі. Саме тому в сукупності вони були обрані для дослідження – щоб жоден важливий аспект не залишився поза увагою.

2.4.2 Автоматизація процесу збору результатів

Для проведення експерименту, що охоплював п’ять різних мовних моделей і десятки прикладів коду, було критично важливим максимально автоматизувати процес генерації тестів і збору метрик. З цією метою було розроблено спеціальний консольний застосунок на C#, який виконував роль виконавця експерименту і автоматичного збирача даних. Даний додаток реалізує два режими роботи: режим дослідження (основний, для генерування тестів) та режим оцінки читабельності (для отримання оцінок від моделей-агентів). Вихідний код програми опубліковано на GitHub [144].

У режимі дослідження застосунок виконує такі кроки:

1) Завантажує з файлу заздалегідь підготовлений текст промпта для генерації тестів. Він описує, що необхідно зробити – згенерувати модульні тести певного формату на основі наведеного коду C#. Вміст промпта був розроблений на етапі "Побудова промптів" і залишався незмінним для всіх моделей піддослідної групи, щоб забезпечити коректність порівняння.

2) Завантажує датасет прикладів коду – набір файлів, кожен з яких містить фрагмент вихідного коду програми на C#, для якого потрібно згенерувати тести. Датасет є спільним для всіх моделей і підбирався таким чином, щоб охопити різні сценарії (різні структури коду, алгоритми, класи тощо).

3) Далі додаток ітерується по кожному прикладу коду з датасету. Для кожного прикладу програма:

a) Підставляє поточний фрагмент коду у шаблон промпта (в потрібне місце, передбачене для вставки вихідного коду).

b) Відправляє сформований повний запит (prompt з кодом) до поточної тестованої мовної моделі через API. З'єднання з різними моделями здійснюється централізовано за допомогою сервісу OpenRouter – уніфікованого API-шлюзу, що надає доступ до сотень моделей від різних провайдерів через єдиний інтерфейс. Кожна з п'яти моделей досліджуваної групи запускалася окремо: тобто програма мала конфігурацію, в якій зазначалася цільова модель (наприклад, GPT-5 mini), після чого виконувала цикл запитів для всіх прикладів. Таких запусків було п'ять – по одному для кожної моделі, – що дозволило ізолювати результати різних моделей. Для всіх мовних моделей, що підтримують функцію міркування (reasoning mode), було застосовано найвищий доступний рівень міркування з метою забезпечення максимальної якості згенерованих тестів.

c) Очікуючи на відповідь, програма автоматично засікає час виконання запиту (в мілісекундах). Таймер стартує перед надсиланням запиту і зупиняється, коли повна відповідь від моделі отримана.

d) Отримавши відповідь (згенерований код тестів), програма зберігає її у файл, а також записує значення метрик, надані OpenRouter API [91].

4) Після проходження всіх прикладів для однієї моделі, програма генерує набір файлів, в кожному з яких збережено ідентифікатор генерації, згенеровані тести для прикладу коду, час виконання, токени запиту/відповіді/сумарно та вартість запиту.

Варто підкреслити, що написання окремого додатка дозволило повністю уніфікувати й автоматизувати збір базових метрик, усунувши ризик людської помилки або суб'єктивності. Після запуску та завершення роботи програми у режимі дослідження були отримані сирі результати генерації для кожної моделі, що включають згенеровані тести й записані інструментом метрики (execution time, tokens, cost).

Другий режим роботи програми – режим визначення індексу читабельності – використовується після того, як основні результати вже отримані і збережені. Алгоритм дій у цьому режимі такий:

1) Програма завантажує із файлу шаблон промпта для оцінки читабельності.

2) Завантажує результати згенерованих тестів (файли з кодом модульних тестів, які видала кожна модель на кожен приклад).

3) У циклі програма проходить по кожному збереженому блоку тестів і для нього послідовно робить 5 запитів – по одному до кожної з п’яти моделей-агентів. Тобто, береться текст тестів, підставляється у промпт оцінки читабельності, і надсилається запит, наприклад, до моделі DeepSeek-V3.2 (агент). Отримана від неї відповідь (яка містить числову оцінку та пояснення) зберігається у файл. Потім той самий тестовий код надсилається моделі GPT-5.1, і т.д. Відповіді всіх агентів зберігаються.

Даний процес – оцінка читабельності – також проводився окремо для кожної тестованої моделі. Тобто, запускаючи режим оцінки, у програмі можна вказати, результати якої моделі слід обробити. У підсумку виконано 5 запусків (для тестів, згенерованих кожною з п’яти основних моделей). Загальний обсяг звернень у цьому режимі – 5 (агентів) × 5 (піддослідних моделей) × 20 (кількість прикладів у датасеті).

Автоматизація оцінювання читабельності суттєво прискорила процес отримання відповідних показників й уніфікувала його. Адже ручний аналіз читабельності кожного тесту експертами-людьми зайняв би багато часу і міг бути непослідовним. Використання ж LLM-агентів дозволило отримати швидкі оцінки. Звісно, якість таких оцінок залежить від точності моделей-агентів, проте завдяки вибору топ-моделей і усередненню можна очікувати прийнятного рівня об’єктивності.

2.4.3 Процедура проведення експерименту

Після визначення метрик і розробки програмного інструментарію було виконано саме дослідження – серію експериментальних кроків для кожної

піддослідної мовної моделі. Загалом процедура для кожної LLM складалася з наступних етапів:

1) Автоматичне генерування тестів та збір базових метрик

Спочатку виконано запуск консольного застосунку у режимі дослідження, як описано вище, з вказівкою на конкретну модель. В результаті для кожної моделі було отримано комплект згенерованих модульних тестів (по одному файлу тестів на кожен приклад коду) та базові метрики: час відповіді, кількість токенів (prompt, completion, total) та вартість на кожен запит. Ці проміжні дані зберігалися для подальшого аналізу.

2) Перевірка та корекція згенерованих тестів (аналіз компіляції)

На цьому кроці всі згенеровані тести певної моделі імпортувалися до окремого проєкту "LlmUnitTestGenerationArtifacts" – підготовленого тестового проєкту в середовищі .NET, що містив вихідний код (еталонні приклади) і мав налаштування для запуску тестів. Спочатку перевірялося, чи кожен з отриманих файлів тестів компілюється без помилок. Якщо компіляція проходила успішно, для цього прикладу відразу фіксувалося 100% компільованості. Якщо ж були виявлені помилки компіляції, виконувалося ручне виправлення коду тесту. Час, витрачений на виправлення проблем, замірявся (у хвилинах) – ці дані потім використовувалися для розрахунку compilability index за описаною раніше формулою.

3) Виконання тестів та визначення індексу валідності

Коли всі тестові файли були приведені до компільованого стану, відбувся їх запуск (через тестовий фреймворк, що використовувався в проєкті – xUnit [146]). Для кожного набору тестів фіксувалося, чи пройшли вони успішно. Оскільки еталонний код працює правильно, очікувалося, що всі згенеровані тести мають проходити. Підрахувавши кількість успішних та провалених тестів, для кожного прикладу обчислено validity index: (кількість тестів що пройшли / загальна кількість тестів) × 100%.

4) Аналіз покриття коду (code coverage)

Наступним кроком оцінено, яку частку коду покривають згенеровані тести. Для цього використано інструмент JetBrains dotCover [57]. У проєкті

"LlmUnitTestGenerationArtifacts", де вже були наявні всі вихідні приклади і виправлені тести, було запущено аналіз покриття: прогін усіх модульних тестів під моніторингом dotCover. Інструмент згенерував звіт, що показує відсоток покриття коду тестами. З отриманого звіту були взяті показники coverage для кожного прикладу.

5) Мутаційне тестування (обчислення mutation score)

На цьому етапі кожен набір тестів перевірявся на здатність виявляти навмисно внесені помилки у вихідний код. За допомогою інструмента Stryker.NET [106] було проведено мутаційне тестування для кожного прикладу та отримано значення mutation score.

6) Оцінка індексу читабельності

Після того як всі технічні метрики зібрано, настав заключний крок – визначення readability index для кожного згенерованого блоку тестів. Для цього було використано другий режим консольного застосунку. Програма послідовно проходила по кожному файлу з кодом тестів (для даної моделі) і запитувала п’ять моделей-агентів, отримуючи від них оцінки читабельності. Отримані п’ять значень (у відсотках) усереднювалися, і цей середній показник записувався як індекс читабельності для відповідного блоку тестів. Таким чином, для кожного прикладу і кожної моделі отримано оцінку читабельності.

7) Агрегація та збереження результатів

Після проходження всіх вищезазначених кроків для кожної з п’яти моделей, було сформовано підсумкову таблицю з результатами. У цій таблиці по рядках вказані моделі, а по стовпцях – середні значення всіх метрик: час виконання (с.), вартість ($), токени (prompt, completion, total), індекс компільованості (%), індекс валідності (%), покриття коду (%), мутаційна оцінка (%), індекс читабельності (%). Така структура даних дозволяє на наступному етапі (аналізі результатів) порівняти моделі між собою за кожним критерієм, побудувати відповідні графіки, а також виявити сильні та слабкі сторони кожної моделі.

Усі результати експерименту – згенеровані тести, звіти dotCover і Stryker, відповіді моделей-агентів щодо читабельності, а також зведені таблиці – збережені

та викладені у відкритий репозиторій на GitHub для верифікації та подальшого аналізу [67]. Це забезпечує прозорість дослідження: зацікавлені особи можуть ознайомитися з артефактами і пересвідчитися у правильності обчислення метрик.

Підсумовуючи, у цьому розділі було детально описано процес практичного проведення дослідження з генерування модульних тестів великими мовними моделями. По-перше, визначено низку метрик якості тестів – мутаційну оцінку, покриття коду, індекси компільованості, валідності та читабельності, а також показники продуктивності (час, токени, вартість). Надано обґрунтування вибору цих метрик та зазначено інструменти й підходи для їх вимірювання (Stryker.NET для mutation score, dotCover для coverage, агентний метод для читабельності тощо). По-друге, представлено засоби автоматизації експерименту: створено C#-застосунок, який у стандартизований спосіб надсилає запити до моделей (через OpenRouter) і фіксує отримані відповіді разом із метриками, а також автоматично оцінює читабельність тестів за допомогою зовнішніх моделей-агентів. По-третє, покроково розглянуто процедуру обробки результатів: перевірку і виправлення згенерованих тестів, запуск їх на виконання, проведення аналізу покриття та мутаційного тестування, опитування агентів щодо читабельності. В результаті реалізовано повний цикл дослідження: від отримання тестів від моделей – до збору всіх кількісних показників якості цих тестів. Отримані чисельні результати для кожної моделі зведено у таблиці та підготовлено для наступного етапу – аналізу та інтерпретації, що буде виконано в наступному розділі роботи.

2.5 Огляд результатів

У таблиці 2.1 надано результати дослідження генерації великими мовними моделями модульних тестів на основі вихідного коду програми C#.

Усі відібрані моделі дотрималися заданого формату відповіді – кожна згенерувала лише код тестів (блоки модульних тестів) без будь-яких текстових пояснень. Згенеровані тестові набори виявилися придатними до виконання: їх запуск проходив дуже швидко – на рівні секунд або навіть мілісекунд – що цілком

відповідає вимогам до швидкодії модульних тестів. Жодна з моделей не згенерувала тестів, виконання яких суттєво б перевищувало стандартний час, отже всі результати є прийнятними з практичної точки зору.

Таблиця 2.1

Результати дослідження

	Cmpl	**Vldt**	**Cvrg**	**Mut**	**Exc**	**Ptkns**	**Ctkns**	**Ttkns**	**Tcst**	**Rdblt**
Gemini 3 Pro	99,9	**96,7**	**99,6**	**88,3**	66	5749,75	5826,8	7620,95	0,0735	86,9
GPT-5 mini	99,85	93,7	99,2	87,6	39,9	1578,4	2869,65	4448,05	0,0061	85,84
DeepSeek R1 0528	**99,95**	91,2	98,6	83,6	204,5	1914,55	5221,1	7135,65	0,0141	**87,08**
Qwen3 Coder 30B	**99,95**	83,7	96,6	72,9	**8,2**	1557,4	**1185,65**	**2743,05**	**0,0005**	84,21
GLM-4.5-Air	97,45	82,3	92,15	82,3	14,8	**1553,8**	1572,7	3126,5	0,0012	81,86

2.5.1 Синтаксична коректність та помилки компіляції.

За показником compilability index (відсоток успішної компіляції тестів) майже всі моделі продемонстрували результат близький до 100% (рисунок 2.3). Зокрема, Gemini 3 Pro досягла 99,9%, DeepSeek R1 0528 та Qwen3 Coder 30B – 99,95%, GPT-5 mini – 99,85%. Лише модель GLM-4.5-Air мала дещо нижчий показник компіляції (97,45%), що свідчить про наявність у частини згенерованих нею тестів синтаксичних помилок. Аналіз цих випадків показав, що помилки здебільшого були невеликими і типовими для автоматично згенерованого коду. Наприклад, Qwen3 Coder 30B в 4 прикладі не додала необхідну директиву імпорту "using AutoMapper;", а Gemini 3 Pro пропустила "using NSubstitute.ExceptionExtensions;" у першому. GPT-5 mini іноді використовувала неоднозначні виклики методів – зокрема, у одному тесті застосовано конструкцію "Arg.Is<int?>(null)" замість правильної "Arg.Is((int?)null)", що заважало компілятору визначити необхідне перевантаження методу; також в іншому випадку ця модель помилково ініціалізувала не ту змінну, а в окремому тесті створила зайвий допоміжний клас. Модель GLM-4.5-Air припускалася найрізноманітніших синтаксичних помилок: у кількох тестах вона намагалася звертатися до елементів колекцій за індексом там, де це недоречно, або використовувати в атрибутах "[InlineData]" некоректні літерали (наприклад,

створення об’єкта "DateTime" чи "String"). Також у тестах від GLM-4.5-Air траплялися випадки, коли не було вказано тип даних для масивів у початкових значеннях, або здійснювалися звернення до приватних полів класу, що тестувався. В одному з прикладів GLM-4.5-Air навіть намагалася використати неіснуючий тип даних при оголошенні змінної, що також призводило до помилки компіляції. Відкрита модель DeepSeek R1 0528 продемонструвала майже бездоганну синтаксичну коректність: єдиний зафіксований недолік – відсутність директиви "using System.Linq.Expressions;" у одному тесті. Зазначені помилки пояснюють незначне зниження compilability index для окремих моделей, однак в цілому переважна більшість згенерованих тестів успішно компілювалися. Це підтверджує, що LLM здатні генерувати синтаксично правильний код тестів з високою ймовірністю, хоча інколи можуть траплятися дрібні пропуски чи неточності.

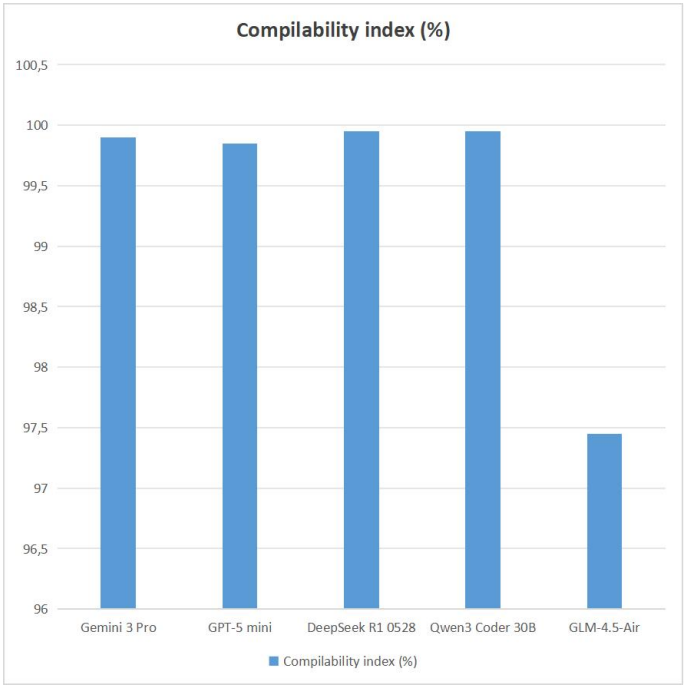

Рис. 2.3. Результати дослідження: індекс компільованості

2.5.2 Коректність виконання тестів

Після компіляції тестових наборів оцінювалася їхня семантична правильність – наскільки тести проходять без "падінь", тобто чи відповідають очікувані

результати фактичній поведінці коду. Цей показник відображає відсоток тестів, що виконалися успішно. У ході експериментів validity index виявився високим для всіх моделей, але найкращим – у флагманських (рисунок 2.4). Зокрема, Gemini 3 Pro досягла 96,7% валідних тестів, GPT-5 mini – 93,7%. Відкрита модель DeepSeek R1 0528 також показала гідний результат (91,2%), лише трохи поступившись пропрієтарним аналогам. Натомість у моделей Qwen3 Coder 30B та GLM-4.5-Air validity index був помітно нижчим (83,7% та 82,3% відповідно). Це означає, що в тестах від цих моделей частіше траплялися логічні помилки – наприклад, неправильні очікувані значення в перевірках або невраховані граничні випадки, через що деякі згенеровані тести не проходили успішно. Незважаючи на це, навіть найслабші моделі забезпечили понад 80% семантично коректних тестів, що є доволі прийнятним показником, хоча й гіршим за результати лідерів.

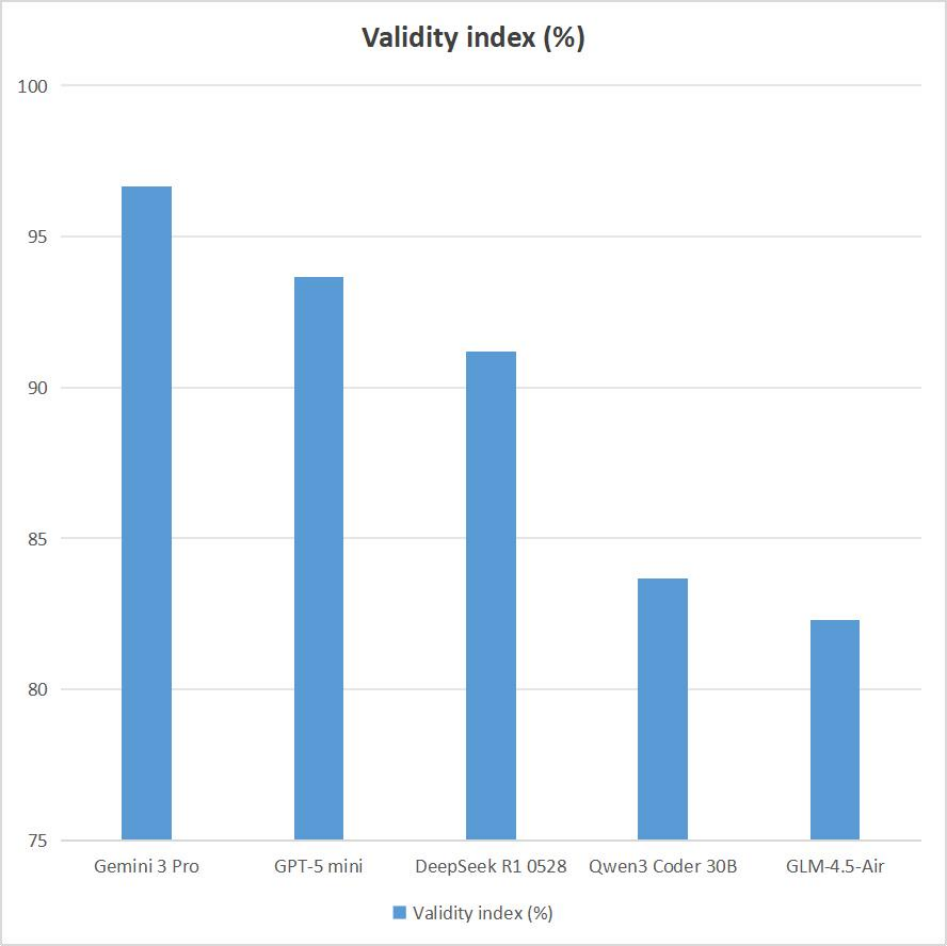

Рис. 2.4. Результати дослідження: індекс валідності

2.5.3 Повнота тестування: покриття коду

Важливим критерієм якості тестів є частка коду, яку вони перевіряють. Отримані результати продемонстрували, що дві найпотужніші моделі забезпечили майже повне покриття коду (рисунок 2.5): Gemini 3 Pro досягнула 99,6% покриття інструкцій, а GPT-5 mini – 99,2%. Дуже близьким до них був показник і DeepSeek R1 0528 (98,6%), тобто відкритій моделі вдалося охопити майже весь код не гірше комерційних продуктів. Помітно менше покриття продемонстрували Qwen3 Coder 30B (96,6%) та особливо GLM-4.5-Air (92,2%). Це означає, що тести, згенеровані останніми двома моделями, залишили неперевіреними деякі фрагменти програм – ймовірно, більш складні гілки логіки або рідкісні випадки, які інші моделі змогли врахувати. В цілому ж високі значення code coverage для топ-моделей свідчать про їхню здатність генерувати досить комплексні набори тестів, що перевіряють більшість поведінки вихідного коду.

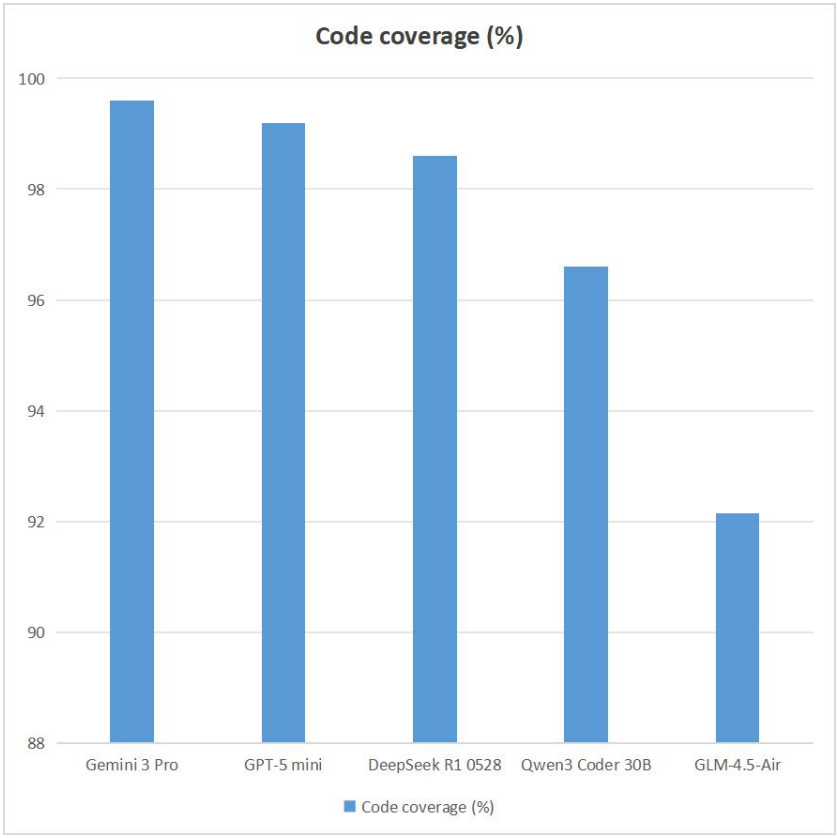

Рис. 2.5. Результати дослідження: покриття коду

2.5.4 Здатність виявляти дефекти: мутаційний аналіз

Окрім покриття, важливою є здатність тестів виявляти потенційні помилки в коді. Для цього використано метрику mutation score. За результатами експерименту найбільшу ефективність за цією метрикою знову показали Gemini 3 Pro та GPT-5 mini (рисунок 2.6): їхні показники становлять 88,3% і 87,6% відповідно. Це підтверджує, що тести, згенеровані цими моделями, виявляють переважну більшість можливих дефектів у коді. Відкрита модель DeepSeek R1 0528 досягла дещо меншого показника (83,6%), однак це все одно дуже високий результат, близький до рівня лідерів. Натомість у GLM-4.5-Air значення mutation score склало 82,3%, що нижче порівняно з топ-моделями. Найгірше себе зарекомендувала Qwen3 Coder 30B – її mutation score близько 72,9% став найнижчим серед усіх. Такий низький показник означає, що значна частина дефектів (мутантів) лишилася непоміченою тестами Qwen, тобто ці тести недостатньо ретельно перевіряють поведінку програмного коду. Таким чином, за здатністю "ловити" помилки моделі можна розташувати від найкращих (Gemini 3 Pro, GPT-5 mini, трохи відстає DeepSeek R1 0528) до найгірших (GLM-4.5-Air та особливо Qwen3 Coder 30B).

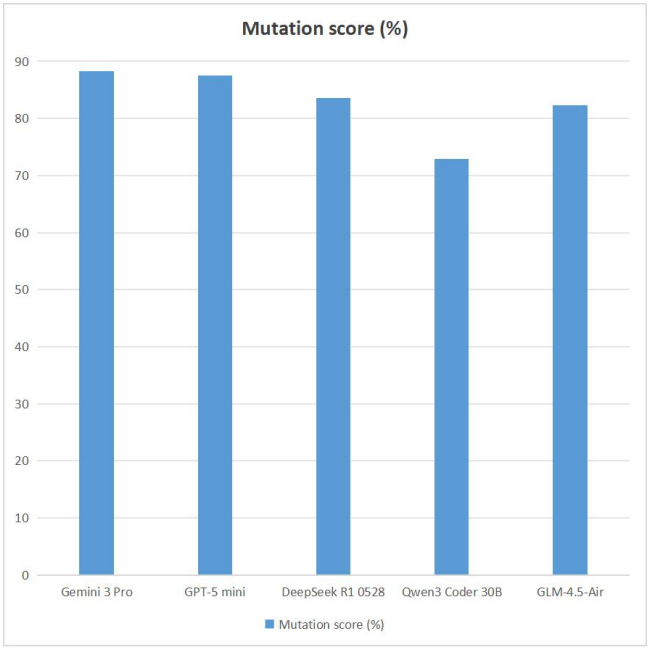

Рис. 2.6. Результати дослідження: мутаційна оцінка

2.5.5 Якість та читабельність згенерованого коду тестів

Окрім коректності та повноти перевірок, важливо, щоб самі тестові сценарії були якісно написані – адже вони мають бути зрозумілими для розробників і зручними у підтримці. Для цього було введено readability index – показник, що відображає читабельність згенерованого тестового коду. Найвище значення продемонструвала модель DeepSeek R1 0528 – 87,1%, що навіть трохи перевищило показник флагманської Gemini 3 Pro (86,9%). Це означає, що тести, згенеровані DeepSeek, структуровані особливо добре, відповідають усталеним шаблонам (можливо, завдяки навчання моделі на open-source коді) і їх відносно легко розуміти. Високу читабельність також показали тести від GPT-5 mini – 85,8%. Натомість у відкритих моделей Qwen3 Coder 30B та GLM-4.5-Air показники читабельності були найнижчими (84,2% та 81,9% відповідно). Це свідчить, що код тестів від Qwen і особливо GLM містив більше стилістичних недоліків чи менш вдалих рішень з точки зору чистоти коду. Втім, навіть 81–84% – доволі прийнятний рівень, але все ж помітно нижчий, ніж у решти учасників. Отже, за критерієм підтримуваності та якості коду найкращі результати знову належать моделям Gemini 3 Pro, GPT-5 mini та DeepSeek R1 0528 (рисунок 2.7).

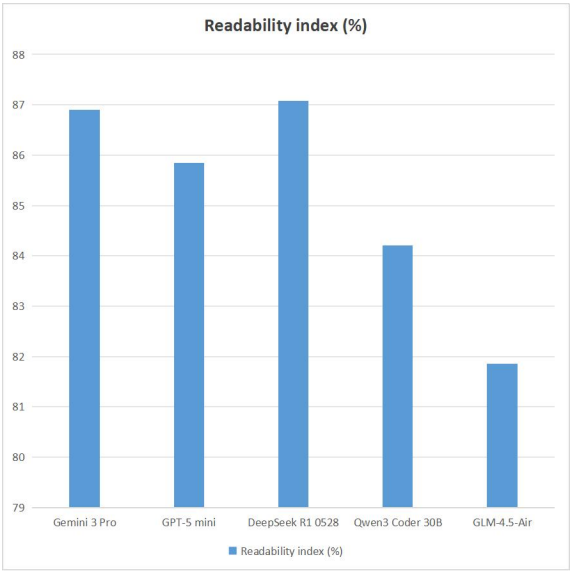

Рис. 2.7. Результати дослідження: індекс читабельності

2.5.6 Показники продуктивності

Для практичного застосування автоматичного генерування тестів важливі не лише якість, а й швидкодія (рисунок 2.8). Найшвидшою виявилася модель Qwen3 Coder 30B: середній час генерації і виконання всіх тестів для одного фрагмента коду становив усього 8,2 с. Модель GLM-4.5-Air також працювала дуже швидко – близько 14,8 с. Пропрієтарні GPT-5 mini та Gemini 3 Pro потребували більше часу (39,8 с. та 65,9 с. відповідно) для формування відповіді і прогону тестів, що можна пояснити їхнім більшим розміром та складністю. Найдовшим був час відповіді відкритої моделі DeepSeek R1 0528 – у середньому 204,5 с., тобто понад 3 хвилини. У контексті застосування як інструменту розробника, навіть максимальний час у кілька хвилин не є критичним для одноразового генерування тестів; проте швидкодія може мати значення при масштабному використанні або інтеграції в CI/CD, де перевагу матимуть моделі з меншими затримками.

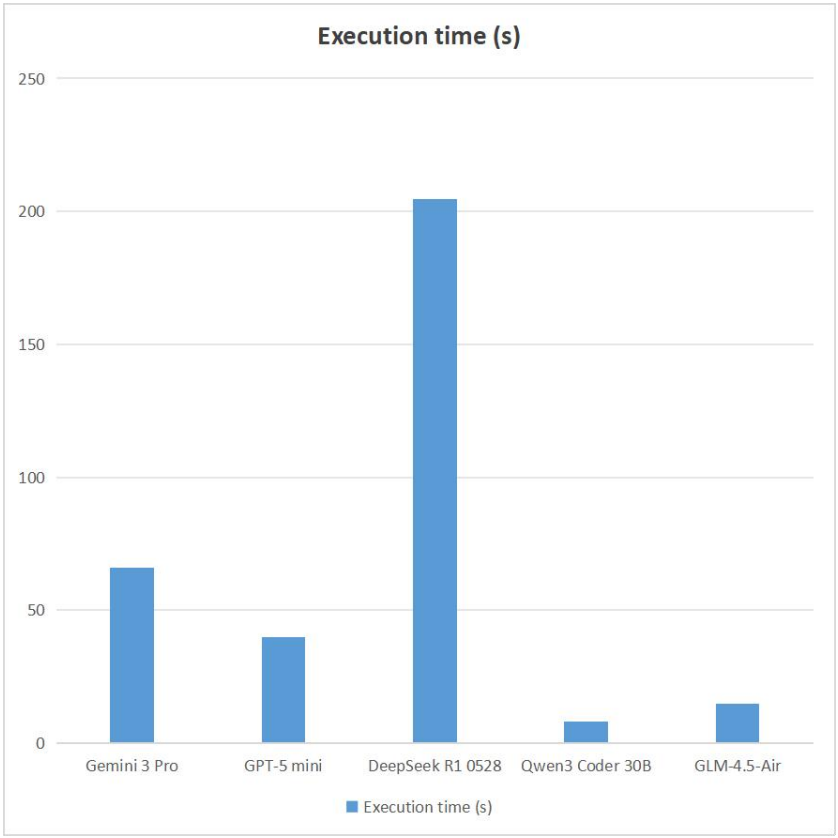

Рис. 2.8. Результати дослідження: час генерації

2.5.7 Вартість генерації

Вартість використання моделей (розрахована на основі кількості опрацьованих токенів у запиті та відповіді) також суттєво відрізняється (рисунки 2.9, 2.10). Найдешевшими виявилися відкриті моделі: Qwen3 Coder 30B та GLM-4.5-Air, які генерували відносно короткі відповіді (в середньому 2,7–3,1 тисячі токенів на один запит) і мали умовну вартість менше 1 цента ($0,0005–$0,001) за запит. Ненабагато дорожчим був GPT-5 mini (близько 4,45 тис. токенів, вартість $0,006). Натомість потужні моделі потребують значно більше токенів: відповідь DeepSeek R1 0528 в середньому містила 7,14 тис. токенів, а Gemini 3 Pro – понад 7,62 тис. токенів, що відповідно оцінюється в $0,014 та $0,073 за один запит. Таким чином, запит до Gemini 3 Pro обходиться приблизно в 5 разів дорожче, ніж до DeepSeek, і в десятки разів дорожче, ніж до найлегших моделей. Слід зауважити, що DeepSeek R1 0528 як відкрита модель може бути розгорнута локально, що усуває ризики витоку даних і проблеми конфіденційності, притаманні використанню закритих API, а також потенційно дозволяє взагалі уникнути прямих витрат на сторонні сервіси (окрім витрат на власні обчислювальні ресурси).

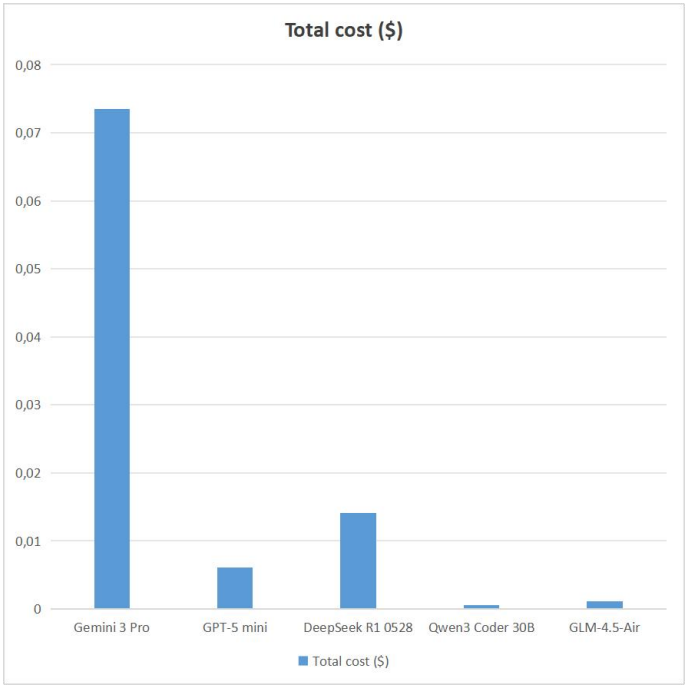

Рис. 2.9. Результати дослідження: вартість запиту

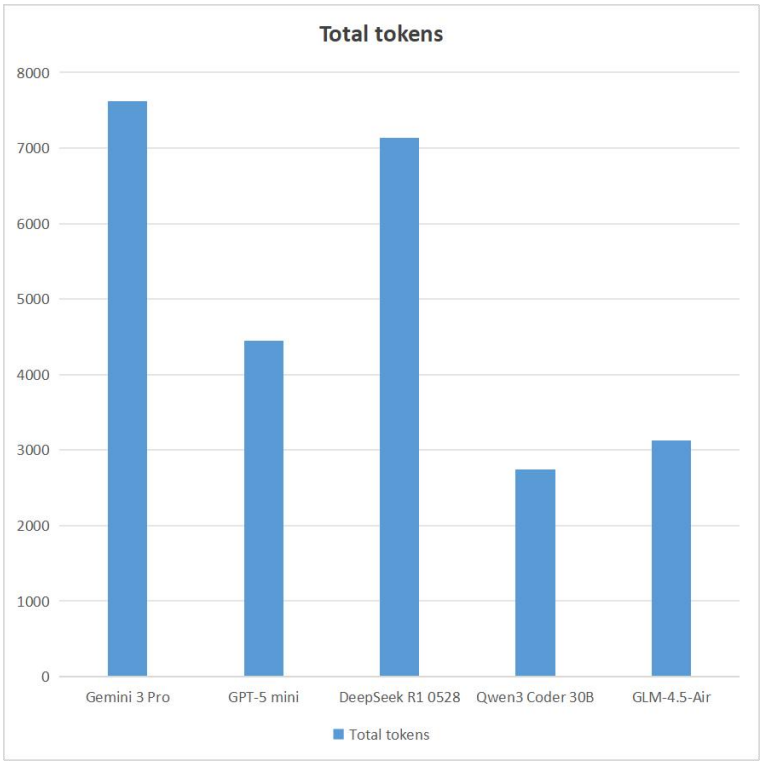

Рис. 2.10. Результати дослідження: кількість токенів

2.5.8 Порівняльний аналіз та підсумки

Загальна картина результатів така: дві пропрієтарні моделі нового покоління – Gemini 3 Pro та GPT-5 mini – продемонстрували найвищу якість генерованих тестів. Вони забезпечили майже максимальне покриття коду, виявлення переважної більшості дефектів і дуже високий відсоток коректних тестів, що узгоджується з їхніми провідними позиціями у відповідних бенчмарках. Відкрита модель DeepSeek R1 0528 виявилася здатною конкурувати з флагманами: за основними метриками (code coverage, mutation score, validity index) вона лише трохи відстає від лідерів, а подекуди навіть перевершує їх (compilability index, readability index). Такий результат особливо важливий з огляду на переваги відкритих LLM – відсутність залежності від закритих платформ, кращий контроль над даними та нижчу ціну використання. Таким чином, DeepSeek R1 0528 демонструє, що сучасні відкриті моделі можуть наблизитися за ефективністю до комерційних систем і стати реальними альтернативами для задач генерування тестів. Водночас дві інші відкриті

моделі – Qwen3 Coder 30B та GLM-4.5-Air – показали значно гірші результати. Хоча вони відзначилися мінімальними витратами і швидким генеруванням, їхні тести мають низькі показники покриття і mutation score, тобто пропускають багато сценаріїв та можливих дефектів. Їхня синтаксична і семантична коректність теж на межі прийнятного мінімуму. Отже, у практичному плані Qwen3 Coder 30B та GLM-4.5-Air наразі суттєво поступаються більш просунутим моделям і генерують тести помітно нижчої якості.

Проведене дослідження підтверджує, що застосування великих мовних моделей для автоматичної генерації модульних тестів на C# є дієвим підходом. Найпотужніші LLM (такі як Gemini 3 Pro, GPT-5 mini) здатні створювати високоякісні тести, які майже не поступаються ручним за повнотою та ефективністю перевірки коду. Відкрита модель DeepSeek R1 0528 продемонструвала близький до них рівень, водночас маючи переваги у вигляді локального розгортання та відсутності плати за API. Отримані в роботі результати є корисними для практики, оскільки дозволяють обґрунтовано обрати оптимальну модель залежно від потреб: якщо пріоритетом є максимальна якість тестів – доцільно використати флагманські моделі, якщо важлива конфіденційність даних або обмежені ресурси – хорошим компромісом може стати відкрита модель на кшталт DeepSeek. Таким чином, результати дослідження підтверджують потенційну користь залучення сучасних LLM у процес розробки програмного забезпечення (зокрема для автоматизації тестування) та окреслюють сильні і слабкі сторони різних підходів, що стане підґрунтям для подальших досліджень у цій галузі.

2.6 Висновки

Проведене в межах даного розділу дослідження дало змогу комплексно оцінити потенціал застосування великих мовних моделей для автоматизованої генерації модульних тестів та сформулювати обґрунтовані рекомендації щодо їх впровадження у процеси розробки програмного забезпечення. У ході роботи було реалізовано повний цикл перевірки гіпотез: від відбору інструментарію та

формування методології до проведення автоматизованого експерименту і глибокого аналізу отриманих метрик. Отримані результати підтверджують, що сучасні LLM досягли рівня розвитку, який дозволяє їм виступати ефективними асистентами у створенні тестового покриття, забезпечуючи високу якість коду та економію часових ресурсів.

Для забезпечення об'єктивності та репрезентативності результатів експерименту було здійснено ретельний відбір піддослідних моделей. Спираючись на показники авторитетних бенчмарків, таких як LiveCodeBench [15, 65], SWE-Bench [108, 107], LiveBench [63], MMLU-Pro [17], HLE [14], до фінальної вибірки було включено п'ять моделей, що відображають різні сегменти ринку: комерційні флагмани Gemini 3 Pro [46] та GPT-5 mini [82], передова модель з відкритими вагами DeepSeek R1 0528 [33], а також спеціалізовані локальні рішення Qwen3 Coder 30B [7] та GLM-4.5-Air [147]. Такий підхід дозволив порівняти ефективність пропрієтарних та відкритих технологій, а також моделей загального призначення та спеціалізованих на коді. Важливим методичним рішенням стало використання платформи OpenRouter [90], що забезпечило єдине уніфіковане середовище для взаємодії з усіма моделями, нівелюючи технічні відмінності в API різних провайдерів та гарантуючи рівні умови тестування.

Методологічною основою дослідження стала розробка структурованих промптів, які базуються на передових практиках prompt-інженерії [99, 100, 98, 96, 113]. Застосування комбінації технік zero-shot, role prompting, system prompting та contextual prompting дозволило чітко окреслити роль моделі, контекст завдання та вимоги до вихідного коду, мінімізуючи ймовірність галюцинацій. Для експериментальної перевірки було сформовано унікальний датасет, що складається з 20 фрагментів коду C# [67]. До набору увійшли як прості допоміжні методи, так і складні алгоритми та елементи бізнес-логіки, що дозволило всебічно оцінити адаптивність моделей до різних сценаріїв програмування та рівнів складності завдань.

Ключовим фактором успішного виконання роботи стала повна автоматизація процесу експерименту. Було розроблено спеціалізоване програмне забезпечення, яке

забезпечило автоматизовану генерацію тестів та збір широкого спектра метрик якості. Використання інструментів Stryker.NET [106] для мутаційного тестування та JetBrains dotCover [57] для аналізу покриття коду дозволило отримати об'єктивні показники ефективності згенерованих тестів (mutation score, code coverage). Окрім того, було впроваджено оцінку індексів валідності (validity), компільованості (compilability) та читабельності (readability). Це дало змогу комплексно охарактеризувати якість роботи моделей не лише з технічної точки зору, а й з позиції підтримки коду.

Головним результатом проведеного дослідження є емпіричне підтвердження доцільності використання LLM для автоматизації юніт-тестування. Встановлено, що беззаперечними лідерами за якістю генерації є комерційні моделі Gemini 3 Pro [46] та GPT-5 mini [82], які забезпечують найвищі показники покриття коду та мутаційної стійкості. Водночас, важливим відкриттям стала висока ефективність відкритої моделі DeepSeek R1 0528 [33], яка за ключовими метриками наблизилася до пропрієтарних аналогів, демонструючи потенціал open-weight рішень як повноцінної альтернативи дорогим комерційним сервісам. Це свідчить про те, що технологічний розрив між закритими та відкритими моделями скорочується, відкриваючи нові можливості для локального розгортання інструментів генерації коду.

Отримані результати та напрацьована методологія створюють надійне підґрунтя для подальших наукових пошуків у цьому напрямку. Перспективним вбачається розширення датасету, що дозволить провести більш глибоку статистичну оцінку та нівелювати вплив випадкових факторів. Збільшення вибірки до сотень фрагментів із різних предметних областей сприятиме виявленню кореляцій між типами алгоритмів та ефективністю конкретних моделей. Також логічним продовженням роботи є збільшення кількості піддослідних моделей шляхом включення нових флагманів, що з'являються на ринку, та спеціалізованих моделей, донавчених виключно на завданнях генерації коду.

Важливим вектором майбутніх досліджень є застосування більш складних технік промпт-інженерії. Зокрема, експерименти з методами Chain-of-Thought

(ланцюжок міркувань) та few-shot prompting (надання прикладів) можуть суттєво підвищити здатність моделей розв’язувати нетривіальні задачі та генерувати тести для складної логіки. Окремої уваги заслуговує дослідження впливу мови запиту на якість коду, що передбачає порівняльний аналіз ефективності промптів англійською, українською та іншими мовами.

Необхідно також розширити експерименти на інші мови програмування (Python, Java, JavaScript) та середовища розробки, що дозволить перевірити універсальність виявлених закономірностей. Дослідження впливу параметрів генерації, таких як temperature, top-p, top-k, на варіативність результатів допоможе знайти баланс між креативністю моделі та стабільністю тестів. Нарешті, для верифікації автоматичних метрик читабельності та якості коду доцільно залучити експертну оцінку від людей-розробників, що дозволить відкалібрувати застосовані підходи та наблизити критерії оцінювання до реальних вимог індустрії. Реалізація окреслених напрямків дозволить поглибити розуміння ролі штучного інтелекту в інженерії програмного забезпечення та розробити комплексні методики його ефективного застосування.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було проведено комплексне дослідження ефективності застосування сучасних великих мовних моделей для автоматизованої генерації модульних тестів у середовищі розробки .NET, що дозволило повністю досягти поставленої мети. На основі отриманих теоретичних відомостей, розробленої методології та результатів емпіричного експерименту було визначено, що передові моделі штучного інтелекту, представлені станом на 2025 рік, здатні автоматизувати рутинний процес написання тестового коду та забезпечити якість, яка наближається до рівня кваліфікованого розробника. Головним підсумком виконаної праці є експериментально підтверджений факт, що сучасні великі мовні моделі трансформують парадигму модульного тестування, переходячи від механічного забезпечення покриття коду до створення семантично змістовних сценаріїв перевірки, які перевершують за якістю та читабельністю результати традиційних інструментів генерації. Встановлено, що інтеграція генеративного штучного інтелекту у виробничі процеси є дієвим засобом підвищення надійності програмних продуктів та оптимізації ресурсів команд інженерії якості.

Глибокий аналіз предметної області дозволив виявити суттєві обмеження традиційних методів автоматизації тестування, таких як EvoSuite [39, 40], які, хоча й здатні досягати високих показників формального покриття коду, часто генерують складні для сприйняття та підтримки тести [79]. На противагу цьому, теоретичний аналіз підтвердив ключову перевагу великих мовних моделей, яка полягає у їхній здатності розуміти семантику, контекст та наміри, закладені у вихідному коді. Це дозволяє генерувати осмислені перевірки, що відповідають бізнес-логіці програми, а не просто механічно покривають шляхи виконання. Водночас аналіз питань безпеки виявив серйозні ризики використання хмарних сервісів, зокрема загрози витоку персональної інформації, проблеми збереження даних провайдерами та феномен "галюцинацій" залежностей, що може призвести до атак на ланцюг постачання. Це обґрунтовує необхідність застосування політик очищення даних або переходу на

використання локальних моделей, що працюють в ізольованому контурі, для мінімізації кіберзагроз.

Методологічна частина роботи базувалася на обґрунтованому виборі репрезентативної вибірки з п'яти моделей, що охоплюють різні сегменти ринку: пропрієтарні лідери Gemini 3 Pro [46] та GPT-5 mini [82], передова відкрита модель DeepSeek R1 0528 [33], а також менші спеціалізовані моделі Qwen3 Coder 30B [7] та GLM-4.5-Air [147]. Такий підхід дозволив провести багатовимірне порівняння між закритими та відкритими екосистемами, а також оцінити співвідношення вартості та якості. Ефективність дослідження була забезпечена використанням платформи OpenRouter [90], яка дозволила уніфікувати технічну взаємодію з різними моделями через єдиний інтерфейс. Важливим методологічним досягненням стала розробка ефективної структури промпта, що поєднує техніки zero-shot, role, system та contextual prompting. Застосування цієї комбінованої методики дозволило досягти показників успішної компіляції коду на рівні понад 99 відсотків для провідних моделей без необхідності додаткового донавчання, що підтверджує критичну важливість правильного формулювання контексту та обмежень у запиті.

Інженерна складова роботи втілилася у створенні низки програмних артефактів, що забезпечили автоматизацію та репрезентативність дослідження. Було сформовано спеціалізований датасет із 20 різнопланових фрагментів коду на мові C#, які включають допоміжні методи, алгоритми та бізнес-логіку, що дозволило всебічно перевірити адаптивність моделей. Розроблений консольний застосунок повністю автоматизував цикл "запит – генерація – отримання результату – збір метрик – оцінка читабельності". Комплексність оцінювання була забезпечена поєднанням технічних метрик, таких як успішність компіляції, валідність, покриття коду та мутаційна оцінка, з якісними показниками, зокрема індексом читабельності, визначеним з використанням ШІ-агентів, та економічними характеристиками, такими як кількість токенів, вартість та час виконання. Такий підхід дозволив отримати об'єктивну та багатогранну картину ефективності кожного досліджуваного інструменту.

Емпіричні результати порівняльного аналізу дозволили чітко диференціювати моделі за рівнем їхньої ефективності. Беззаперечними лідерами за якістю стали пропрієтарні моделі Gemini 3 Pro та GPT-5 mini, які продемонстрували найвищі показники покриття коду (99,6% та 99,2%), мутаційної стійкості (88,3% та 87,6%) та валідності тестів (96,7% та 93,7%). Гарні результати показала відкрита модель DeepSeek R1 0528, яка впритул наблизилася до показників комерційних лідерів: покриття коду – 98,6%, мутаційна оцінка – 83,6%, валідність – 91,2%. Водночас менші локальні моделі, такі як Qwen3 Coder 30B та GLM-4.5-Air, поки що суттєво поступаються у здатності генерувати складні логічні перевірки, демонструючи нижчі показники якості, хоча і є значно швидшими та економнішими. Економічний аналіз підкреслив компроміс між вартістю та якістю: найдорожчі моделі забезпечують найвищу надійність, тоді як дешеві або безкоштовні відкриті моделі можуть бути використані для менш критичних задач або в умовах обмеженого бюджету.

Практичне значення отриманих результатів полягає у формуванні обґрунтованих рекомендацій для індустрії. Для систем, де критично важливою є максимальна якість та повнота тестування, рекомендовано використання моделей Gemini 3 Pro або GPT-5 mini. У випадках, що вимагають суворої конфіденційності через локальне розгортання або економії бюджету, оптимальним вибором є модель DeepSeek R1 0528. Створені в рамках роботи артефакти, включаючи датасет, код інструменту та результати експериментів, доступні у відкритому репозиторії для використання спільнотою [67]. Крім того, результати вказують на перспективність побудови спеціалізованих застосунків на основі найкращих моделей, які можуть реалізовувати мульти-агентний підхід: генерація тестів однією моделлю, визначення метрик та подальше ітеративне покращення коду на основі отриманих даних іншою моделлю, що дозволить створити замкнений цикл підвищення якості.

Наукова новизна дослідження полягає у тому, що вперше проведено такий комплексний порівняльний аналіз найновіших моделей 2025 року саме для екосистеми .NET та мови C#, заповнюючи прогалину в дослідженнях, які переважно фокусуються на інших мовах. Акцент роботи зроблено саме на генерації модульних

тестів, що відрізняє її від досліджень генерації звичайного коду та вимагає оцінки логічної коректності перевірок. Вперше застосовано комбіновану методику оцінювання, що включає автоматизований розрахунок індексу читабельності за допомогою мульти-агентного підходу, де одні моделі оцінюють якість коду, згенерованого іншими. Це дослідження надає фундамент для подальших напрацювань у напрямку автоматизації забезпечення якості програмного забезпечення та формалізації оцінки результатів роботи генеративного штучного інтелекту.

Перспективи подальших досліджень у цьому напрямку охоплюють розширення експериментальної бази шляхом збільшення датасету та кількості досліджуваних моделей. Важливим вектором є застосування складніших технік prompt-інженерії [99, 100], таких як Chain-of-Thought та few-shot, що може покращити результати у складних сценаріях. Також перспективним є дослідження генерації тестів для інших мов програмування та платформ, а також вивчення ефективності багатомовних промптів, зокрема українською мовою. Окремої уваги потребує залучення людей-експертів для калібрування якісних метрик, що дозволить підвищити точність автоматизованих оцінок та наблизити їх до стандартів індустрії.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 15 LLM coding benchmarks. URL: https://www.evidentlyai.com/blog/llm-coding-benchmarks.

2. 9 Top Open-Source LLMs for 2026 and Their Uses. URL: https://www.datacamp.com/blog/top-open-source-llms.

3. Abstract Syntax Tree (AST) - Explained in Plain English. URL: https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38.

4. AFL. URL: https://github.com/google/AFL.

5. AI Privacy Risks & Mitigations – Large Language Models (LLMs). URL: https://www.edpb.europa.eu/system/files/2025-04/ai-privacy-risks-and-mitigations-in-llms.pdf.

6. Aider polyglot coding leaderboard. URL: https://aider.chat/docs/leaderboards.

7. Alibaba Qwen3 Coder 30B A3B Instruct. URL: https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct.

8. AMD tested 20+ local models for coding & only 2 actually work (testing linked). URL: https://www.reddit.com/r/LocalLLaMA/comments/1nufu17/amd_tested_20_local_models_for_coding_only_2.

9. An Empirical Analysis of Flaky Tests. URL: https://www.cs.cornell.edu/courses/cs5154/2021sp/resources/LuoETAL14FlakyTestsAnalysis.pdf.

10. An Empirical Study of Unit Test Generation with Large Language Models. URL: https://arxiv.org/html/2406.18181v1.

11. Aqua.StringHelpers. URL: https://github.com/ualehosaini/Aqua.StringHelpers.

12. Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing. URL: https://ieeexplore.ieee.org/abstract/document/10638556.

13. Are You Allowed To Share Information with ChatGPT if You Work Under NDA? URL: https://loio.com/guides/business/internet-law/are-you-allowed-to-share-information-with-chat-gpt-if-you-work-under-nda.

14. Artificial Analysis Humanity's Last Exam Benchmark Leaderboard. URL: https://artificialanalysis.ai/evaluations/humanitys-last-exam.

15. Artificial Analysis LiveCodeBench Benchmark Leaderboard. URL: https://artificialanalysis.ai/evaluations/livecodebench.

16. Artificial Analysis LLM Leaderboard - Comparison of over 100 AI models from OpenAI, Google, DeepSeek & others. URL: https://artificialanalysis.ai/leaderboards/models.

17. Artificial Analysis MMLU-Pro Benchmark Leaderboard. URL: https://artificialanalysis.ai/evaluations/mmlu-pro.

18. Artificial Analysis: Independent analysis of AI. URL: https://artificialanalysis.ai.

19. Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis. URL: https://arxiv.org/pdf/2508.14727.

20. Automated Unit Test Improvement using Large Language Models at Meta. URL: https://dl.acm.org/doi/abs/10.1145/3663529.3663839.

21. Benchmarks @ EvalPlus. URL: https://evalplus.github.io.

22. Best LLMs for coding: developer favorites. URL: https://codingscape.com/blog/best-llms-for-coding-developer-favorites.

23. BIG-Bench Benchmark. URL: https://www.emergentmind.com/topics/big-bench-benchmark.

24. BIG-Bench. URL: https://github.com/google/BIG-bench.

25. Building vs. Buying an LLM: Key Decision Factors. URL: https://www.rohan-paul.com/p/building-vs-buying-an-llm-key-decision.

26. ChatUniTest: A Framework for LLM-Based Test Generation. URL: https://arxiv.org/abs/2305.04764.

27. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. URL: https://www.carolemieux.com/codamosa_icse23.pdf.

28. Code Coverage Best Practices. URL: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

29. Code Lingua Leaderboard. URL: https://github.com/codetlingua/codetlingua.

30. CodeXGLUE. URL: https://github.com/microsoft/CodeXGLUE.

31. Complete Guide to Prompt Engineering with Temperature and Top-p. URL: https://promptengineering.org/prompt-engineering-with-temperature-and-top-p.

32. Control Flow Graph (CFG) - Software Engineering. URL: https://www.geeksforgeeks.org/software-engineering/software-engineering-control-flow-graph-cfg.

33. DeepSeek DeepSeek R1 0528. URL: https://huggingface.co/deepseek-ai/DeepSeek-R1-0528.

34. DeepSeek DeepSeek-V3.2. URL: https://huggingface.co/deepseek-ai/DeepSeek-V3.2.

35. dotnetrdf. URL: https://github.com/dotnetrdf/dotnetrdf.

36. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. URL: https://arxiv.org/abs/2308.16557.

37. Employee accounting system. URL: https://github.com/DubBro/EmployeeAccountingSystem.

38. Enterprise privacy at OpenAI. URL: https://openai.com/enterprise-privacy.

39. EvoSuite. URL: https://www.evosuite.org.

40. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. URL: https://www.evosuite.org/wp-content/papercite-data/pdf/esecfse11.pdf.

41. F.I.R.S.T. Principles. URL: https://romainbrunie.medium.com/f-i-r-s-t-principles-4eec4b9c1cde.

42. Flaky Tests at Google and How We Mitigate Them. URL: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

43. Fuzzing. URL: https://en.wikipedia.org/wiki/Fuzzing.

44. Fuzzing: Progress, Challenges, and Perspectives. URL: https://openurl.ebsco.com/EPDB%3Agcd%3A1%3A24042401/detailv2?sid=ebsco%3Aplink%3Ascholar&id=ebsco%3Agcd%3A175291522&crl=c.

45. Google Gemini 2.5 Pro. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro.

46. Google Gemini 3 Pro. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-pro.

47. GraphCodeBERT: Pre-training Code Representations with Data Flow. URL: https://arxiv.org/abs/2009.08366.

48. Hierarchical directory structure. URL: https://github.com/DubBro/DirectoryHierarchy.

49. How It Works: Zero Data Retention. URL: https://vlex.com/news/How-It-Works-Zero-Data-Retention.

50. HumanEval: When Machines Learned to Code. URL: https://runloop.ai/blog/humaneval-when-machines-learned-to-code.

51. Humanity's Last Exam. URL: https://agi.safe.ai.

52. Humanity's Last Exam. URL: https://scale.com/leaderboard/humanitys_last_exam.

53. Hypothesis. URL: https://hypothesis.works.

54. Internet-Auction. URL: https://github.com/DubBro/Internet-Auction.

55. Introducing Claude Opus 4.5. URL: https://www.anthropic.com/news/claude-opus-4-5.

56. Introducing Claude Sonnet 4.5. URL: https://www.anthropic.com/news/claude-sonnet-4-5.

57. JetBrains dotCover. URL: https://www.jetbrains.com/dotcover.

58. KLEE symbolic execution engine in 2019. URL: https://link.springer.com/article/10.1007/s10009-020-00570-3.

59. KLEE Symbolic Execution Engine. URL: https://klee-se.org.

60. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. URL: https://arxiv.org/abs/2404.13340.

61. Large Language Models for Unit Testing: A Systematic Literature Review. URL: https://www.arxiv.org/abs/2506.15227.

62. libFuzzer – a library for coverage-guided fuzz testing. URL: https://llvm.org/docs/LibFuzzer.html.

63. LiveBench: A Challenging, Contamination-Free LLM Benchmark. URL: https://livebench.ai.

64. LiveCodeBench Pro. URL: https://livecodebenchpro.com.

65. LiveCodeBench. URL: https://www.vals.ai/benchmarks/lcb.

66. LLM Hallucinations Explained. URL: https://medium.com/@nirdiamant21/llm-hallucinations-explained-8c76cdd82532.

67. LlmUnitTestGenerationResearch. URL: https://github.com/DubBro/LlmUnitTestGenerationResearch.

68. Markdown. URL: https://en.wikipedia.org/wiki/Markdown.

69. MASK. URL: https://scale.com/leaderboard/mask.

70. MBPP. URL: https://airank.dev/benchmarks/mbpp.

71. Measuring Massive Multitask Language Understanding. URL: https://arxiv.org/abs/2009.03300.

72. Methods for automated test value generation. URL: https://symflower.com/en/company/blog/2022/methods-for-automated-test-value-generation.

73. Microsoft .NET. URL: https://dotnet.microsoft.com.

74. Mistral AI Magistral Small 1.2. URL: https://huggingface.co/mistralai/Magistral-Small-2509.

75. Moonshot AI Kimi K2 Thinking. URL: https://moonshotai.github.io/Kimi-K2/thinking.html.

76. Must Known 4 Essential AI Prompts Strategies for Developers. URL: https://reykario.medium.com/4-must-know-ai-prompt-strategies-for-developers-0572e85a0730.

77. Mutation testing. URL: https://en.wikipedia.org/wiki/Mutation_testing.

78. Mutation Testing: How to Ensure Code Coverage Isn’t a Vanity Metric. URL: https://about.codecov.io/blog/mutation-testing-how-to-ensure-code-coverage-isnt-a-vanity-metric.

79. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. URL: https://arxiv.org/abs/2305.04207.

80. NSubstitute. URL: https://nsubstitute.github.io.

81. OpenAI GPT-5 Codex. URL: https://platform.openai.com/docs/models/gpt-5-codex.

82. OpenAI GPT-5 mini. URL: https://platform.openai.com/docs/models/gpt-5-mini.

83. OpenAI GPT-5. URL: https://platform.openai.com/docs/models/gpt-5.

84. OpenAI GPT-5.1 Codex. URL: https://platform.openai.com/docs/models/gpt-5.1-codex.

85. OpenAI GPT-5.1. URL: https://platform.openai.com/docs/models/gpt-5.1.

86. OpenAI gpt-oss-120b. URL: https://platform.openai.com/docs/models/gpt-oss-120b.

87. OpenAI gpt-oss-20b. URL: https://platform.openai.com/docs/models/gpt-oss-20b.

88. OpenAI o3. URL: https://platform.openai.com/docs/models/o3.

89. OpenAI o4-mini. URL: https://platform.openai.com/docs/models/o4-mini.

90. OpenRouter. URL: https://openrouter.ai.

91. OpenRouter: The Universal API for AI Development in 2025. URL: https://www.codegpt.co/blog/openrouter-universal-api-ai-development.

92. Over Specification in Tests. URL: https://osherove.com/blog/2008/7/12/over-specification-in-tests.html.

93. OWASP Top 10 for LLM Applications 2025. URL: https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf.

94. Pex and Moles – Isolation and White Box Unit Testing for .NET. URL: https://www.microsoft.com/en-us/research/project/pex-and-moles-isolation-and-white-box-unit-testing-for-net.

95. Pretraining vs Fine-Tuning vs Instruction Tuning | Simplified Guide. URL: https://medium.com/@mkmanjula96/pretraining-vs-fine-tuning-vs-instruction-tuning-simplified-guide-9132c7f2f4ac.

96. Prompt design strategies. URL: https://ai.google.dev/gemini-api/docs/prompting-strategies.

97. Prompt engineering techniques. URL: https://www.ibm.com/think/topics/prompt-engineering-techniques#7281536.

98. Prompt engineering. URL: https://drive.google.com/file/d/1AbaBYbEa_EbPelsT40-vj64L-2IwUJHy/view.

99. Prompt engineering. URL: https://platform.openai.com/docs/guides/prompt-engineering.

100. Prompt engineering: overview and guide. URL: https://cloud.google.com/discover/what-is-prompt-engineering?hl=uk.

101. QuickCheck: Automatic testing of Haskell programs. URL: https://hackage.haskell.org/package/QuickCheck.

102. QuickNote Vault. URL: https://github.com/DubBro/QuickNoteVault.

103. Search-Based Software Engineering and AI Foundation Models: Current Landscape and Future Roadmap. URL: https://arxiv.org/abs/2505.19625.

104. Search-based software engineering. URL: https://en.wikipedia.org/wiki/Search-based_software_engineering.

105. Stop requiring only one assertion per unit test: Multiple assertions are fine. URL: https://stackoverflow.blog/2022/11/03/multiple-assertions-per-test-are-fine.

106. Stryker Mutator. URL: https://stryker-mutator.io.

107. SWE-bench. URL: https://www.swebench.com.

108. SWE-bench. URL: https://www.vals.ai/benchmarks/swebench.

109. Symbolic Execution with Test Cases Generated by Large Language Models. URL: https://ieeexplore.ieee.org/abstract/document/10684633.

110. Symbolic execution. URL: https://en.wikipedia.org/wiki/Symbolic_execution.

111. Test Smell Types. URL: https://testsmells.org/pages/testsmells.html.

112. TestEval: Benchmarking Large Language Models for Test Case Generation. URL: https://aclanthology.org/2025.findings-naacl.197.

113. The 2025 Guide to Prompt Engineering. URL: https://www.ibm.com/think/prompt-engineering#605511093.

114. The Algorithms - C#. URL: https://github.com/TheAlgorithms/C-Sharp.

115. The bounded model checker LLBMC. URL: https://ieeexplore.ieee.org/abstract/document/6693138.

116. The Economics of Unit Testing. URL: https://www.researchgate.net/publication/42790485_The_Economics_of_Unit_Testing.

117. The Hidden Risks of LLM-Generated Web Application Code: A Security-Centric Evaluation of Code Generation Capabilities in Large Language Models. URL: https://ar5iv.labs.arxiv.org/html/2504.20612v1.

118. The MASK Benchmark: Disentangling Honesty from Accuracy in AI Systems. URL: https://www.mask-benchmark.ai.

119. The Practical Test Pyramid. URL: https://martinfowler.com/articles/practical-test-pyramid.html.

120. The Rise of Slopsquatting: How AI Hallucinations Are Fueling a New Class of Supply Chain Attacks. URL: https://socket.dev/blog/slopsquatting-how-ai-hallucinations-are-fueling-a-new-class-of-supply-chain-attacks.

121. The test automation pyramid. URL: https://www.ontestautomation.com/the-test-automation-pyramid.

122. The Ultimate 2025 Guide to Coding LLM Benchmarks and Performance Metrics. URL: https://www.marktechpost.com/2025/07/31/the-ultimate-2025-guide-to-coding-llm-benchmarks-and-performance-metrics.

123. Understanding LLM Code Benchmarks: From HumanEval to SWE-bench. URL: https://runloop.ai/blog/understanding-llm-code-benchmarks-from-humaneval-to-swe-bench.

124. Understanding Temperature, Top-k, and Top-p Sampling in Generative Models. URL: https://codefinity.com/blog/Understanding-Temperature%2C-Top-k%2C-and-Top-p-Sampling-in-Generative-Models.

125. Unit testing best practices for .NET. URL: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices.

126. Unit Testing for Better Software Quality and Fewer Bugs. URL: https://www.frugaltesting.com/blog/unit-testing-for-better-software-quality-and-fewer-bugs.

127. Unit Testing Principles. URL: https://www.tmap.net/building-blocks/unit-testing-principles.

128. User Privacy and Large Language Models: An Analysis of Frontier Developers' Privacy Policies. URL: https://arxiv.org/abs/2509.05382.

129. Walks Info Viber Bot. URL: https://github.com/DubBro/WalksInfoViberBot.

130. What are Flaky Tests? URL: https://www.jetbrains.com/teamcity/ci-cd-guide/concepts/flaky-tests.

131. What are LLMs? URL: https://www.ibm.com/think/topics/large-language-models.

132. What Are Tools in the Scope of LLMs and Why Are They So Important. URL: https://alnutile.medium.com/what-are-tools-in-the-scope-of-llms-and-why-are-they-so-important-f57f76190e58.

133. What Is a Data Extraction Attack? URL: https://troj.ai/blog/data-extraction-attack.

134. What is a multi-agent system? URL: https://www.ibm.com/think/topics/multiagent-system.

135. What is a transformer model? URL: https://www.ibm.com/think/topics/transformer-model.

136. What is code coverage? URL: https://www.sonarsource.com/resources/library/code-coverage.

137. What is MMLU? LLM Benchmark Explained and Why It Matters. URL: https://www.datacamp.com/blog/what-is-mmlu.

138. What is prompt engineering? URL: https://www.ibm.com/think/topics/prompt-engineering.

139. What is Property Based Testing? URL: https://hypothesis.works/articles/what-is-property-based-testing.

140. What is retrieval augmented generation (RAG)? URL: https://www.ibm.com/think/topics/retrieval-augmented-generation.

141. Which local models actually work with Cline? AMD tested them all. URL: https://cline.bot/blog/local-models-amd.

142. xAI Grok 4 Fast. URL: https://x.ai/news/grok-4-fast.

143. xAI Grok 4. URL: https://x.ai/news/grok-4.

144. xAI Grok 4.1 Fast. URL: https://x.ai/news/grok-4-1-fast.

145. xUnit Test Patterns. URL: https://martinfowler.com/books/meszaros.html.

146. xUnit.net. URL: https://xunit.net.

147. Zhipu AI GLM-4.5-Air. URL: https://huggingface.co/zai-org/GLM-4.5-Air.

148. Zhipu AI GLM-4.6. URL: https://z.ai/blog/glm-4.6.

ДОДАТОК А

Промпт для генерації модульних тестів

```
**Role:**
You are a Senior C# developer with 15 years of experience and an
expert in unit testing.

---

**Task:**
Write high-quality unit tests for the given code.

---

**Environment:**
- Language: C# 12.0.
- Platform: .NET 8.
- Test framework: xUnit.
- Tool for creating test doubles: NSubstitute.

---

**Requirements:**
- The tests must cover `public` methods.
- The tests must cover `public` properties that contain logic (for
example, validation or data conversion).
- The tests must not directly test `private`, `protected`, `private
protected`, `internal`, or `protected internal` methods --- internal
logic must be tested indirectly via `public` methods.
- If a class under test implements interface members that are only
accessible through the interface type, e.g., explicit interface
implementations or default interface methods, test these members via a
variable of the interface type, for example: `IUserService service =
new UserService();`.
- All tests must follow the AAA pattern, i.e., consist of three blocks
--- Arrange (prepare the environment), Act (perform the action),
Assert (verify the result of the action).
- All tests must follow the FIRST principles:
  - Fast: tests should run quickly; do not use real I/O operations in
test code (file system, database, HTTP calls, etc.); do not use
`Thread.Sleep`/`Task.Delay` in test code.
  - Isolated: each test prepares its own state in the Arrange step and
does not depend on the results of other tests or on shared mutable
static state.
  - Repeatable: the same test with the same initial state must always
produce the same result; do not use
`DateTime.Now`/`DateTime.UtcNow`/`DateTime.Today`, `Random`, or
dependencies on the current environment in test code.
  - Self-verifying: each test must contain explicit asserts and must
not rely on `Console.WriteLine` or manual inspection of output.
  - Timely: write tests only for the current behaviour that follows
from the given code; do not add new functionality or public methods.
```

- External dependencies (DB, file system, HTTP, external services, etc.) must be replaced with test doubles (fakes, mocks, stubs) using NSubstitute, for example: `var someServiceSubstitute = Substitute.For<ISomeService>();`.
- For assertions, use xUnit's built-in assertion methods, for example: `Assert.Equal(expectedResult, actualResult);`, `Assert.Throws<Exception>(action);`.
- If the given code specifies a `namespace`, use the same `namespace` for the tests with the `.UnitTests` suffix added, for example, for `MyApp.Services` it must be `MyApp.Services.UnitTests`; if the `namespace` is absent, use `UnitTests`.
- The test code must follow standard C# and .NET naming conventions.
- The names of test classes must follow the `ClassUnderTestTests` pattern, for example: `UserServiceTests`, `ExcelFileExportProviderTests`.
- The names of tests must follow the `MethodUnderTest_Scenario_ExpectedResult` pattern, for example: `GetUser_PassingInvalidId_ThrowsUserNotFoundException`, `GetUsers_WhenCalled_ReturnsListUserModel`.
- If a scenario tests an asynchronous method, the test signature must be `async Task`, not `async void`.
- The tests must cover the key positive, negative, and edge scenarios that logically follow from the given code.
- A single test must verify only one scenario, i.e., follow the Single Responsibility principle.
- Use `[Fact]` for single scenarios and `[Theory]` (with `[InlineData]`) to test multiple input variations of the same scenario.
- Do not modify the production code.

Response format:
- The response must contain only one `csharp` block with the test code.
- Do not add any explanatory text before the code block.
- Do not add any explanatory text after the code block.
- The test code must be a complete C# file with the necessary `using` directives, `namespace`, and one or more test classes.
- Do not include production code (the classes under test) in the response; generate only the test code.
- Do not add comments in the test code, except for `// Arrange`, `// Act`, `// Assert`; assume that no other comments are critically necessary.

Code:
```csharp
{{CODE}}
```

ДОДАТОК Б

Промпт для оцінювання читабельності модульних тестів

```
**Role:**
You are a Senior C# Developer with 15 years of experience and an
expert in Unit Testing, focused on the quality, readability, and
maintainability of test code.

---

**Task:**
Evaluate the readability of the provided test code and determine its
Readability Index.

---

**Environment:**
- Language: C# 12.0.
- Platform: .NET 8.
- Testing Framework: xUnit.

---

**Evaluation description:**
- The Readability Index assesses only how easy the tests are to read
and maintain.
- The Readability Index does not account for:
  - correctness of the tests' business logic;
  - completeness or level of code coverage;
  - correctness of the assertions themselves regarding business logic.
- The Readability Index takes a value from 1 to 100, where:
  - 1--20 --- very poor readability: almost completely unreadable and
very difficult to maintain code;
  - 21--40 --- poor readability: a lot of chaos, bad formatting,
difficult to understand the purpose of the tests;
  - 41--60 --- mediocre readability: generally understandable, but
there are significant flaws;
  - 61--80 --- good readability: structure and naming are mostly clear,
flaws are not critical;
  - 81--100 --- very high readability and maintainability: tests are
easy to read, logical, well-structured, and simple to maintain.

---

**Description of readable code:**
- Names of classes, methods, variables, fields, and properties are
clear, meaningful, and reflect their purpose.
- Test class names follow the `ClassUnderTestTests` pattern, e.g.,
`UserServiceTests`, `ExcelFileExportProviderTests`.
- The test class does not have unnecessary nested classes.
- Test names follow the `MethodUnderTest_Scenario_ExpectedResult`
pattern, e.g., `GetUser_PassingInvalidId_ThrowsUserNotFoundException`,
`GetUsers_WhenCalled_ReturnsListUserModel`.
```

- If a scenario tests an asynchronous method, the test signature must be `async Task`, not `async void`.
- The test follows the AAA pattern, meaning it contains three logical blocks: Arrange (setup), Act (execution), Assert (verification). These blocks may be marked with comments `// Arrange`, `// Act`, `// Assert` and separated by empty lines.
- The code contains no comments other than `// Arrange`, `// Act`, `// Assert`, which serve as separators for the respective blocks within the test body.
- The code adheres to standard C# and .NET naming conventions.
- The `[Fact]` attribute is used for single scenarios, and `[Theory]` (with `[InlineData]`) is used for testing multiple input variations of the same scenario to avoid code duplication.

The description above defines ideally readable test code. Violations of individual recommendations do not automatically mean a low score. Evaluate the code based on the aggregate of signs. Assess how close the actual code comes to this ideal.

Response format:
- The response must be strictly in the following JSON format:
```json
{
     "readabilityIndex": 0,
     "reasons": [
          "reason 1",
          "reason 2"
     ]
}
```
- The `readabilityIndex` field must contain an integer value of the readability index from 1 to 100.
- The `reasons` field must contain a list (up to 10 elements) of concise reasons for the given score, each describing a specific aspect of readability and maintainability of the test code (naming, structure, formatting, code duplication, etc.).
- Provide only the raw JSON object in the response, without markdown wrappers (```json```) and without any additional text before or after.

Code:
```csharp
{{CODE}}
```

ДОДАТОК В

Код прикладу №10 із набору даних дослідження

```
using AutoMapper;

namespace Dataset.Sample10;

public class AuctionService : IAuctionService
{
    private readonly IUnitOfWork _database;
    private readonly IMapper _mapper;

    public AuctionService(IUnitOfWork database, IMapper mapper)
    {
        _database = database;
        _mapper = mapper;
    }

    public void Bet(int id, string customerName, int bid)
    {
        if (id <= 0)
        {
            throw new InvalidIdException();
        }

        if (customerName == null)
        {
            throw new InvalidNameException();
        }

        var auction = _database.Auctions.Get(id);

        if (auction == null)
        {
            throw new InvalidIdException();
        }

        if (!auction.Started)
        {
            throw new InvalidAuctionException("ERROR: Auction has not
started yet");
        }

        if (auction.Ended)
        {
            throw new InvalidAuctionException("ERROR: Auction is
over");
        }

        if (auction.Bid >= bid)
        {
            throw new InvalidAuctionException("ERROR: Invalid bid");
        }
```

```
        auction.Bid = bid;
        auction.Leader = customerName;

        _database.Auctions.Update(auction);
        _database.Commit();
    }

    public void CloseAuction(int id)
    {
        if (id <= 0)
        {
            throw new InvalidIdException();
        }

        var auction = _database.Auctions.Get(id);

        if (auction == null)
        {
            throw new InvalidIdException();
        }

        if (!auction.Started)
        {
            throw new InvalidAuctionException("ERROR: Auction has not
started yet");
        }

        if (auction.Ended)
        {
            throw new InvalidAuctionException("ERROR: Auction has
already finished");
        }

        auction.Ended = true;

        _database.Auctions.Update(auction);
        _database.Commit();
    }

    public AuctionDTO GetAuction(int id)
    {
        if (id <= 0)
        {
            throw new InvalidIdException();
        }

        var auction = _mapper.Map<Auction,
AuctionDTO>(_database.Auctions.Get(id))
            ?? throw new InvalidIdException();

        return auction;
    }
```

```
    public IEnumerable<AuctionDTO> GetAuctions()
    {
        return _mapper.Map<IEnumerable<Auction>,
IEnumerable<AuctionDTO>>(_database.Auctions.GetAll());
    }

    public void OpenAuction(int id)
    {
        if (id <= 0)
        {
            throw new InvalidIdException();
        }

        var auction = _database.Auctions.Get(id);

        if (auction == null)
        {
            throw new InvalidIdException();
        }

        if (auction.Ended)
        {
            throw new InvalidAuctionException("ERROR: Auction has
already finished");
        }

        if (auction.Started)
        {
            throw new InvalidAuctionException("ERROR: Auction has
already started");
        }

        auction.Started = true;

        _database.Auctions.Update(auction);
        _database.Commit();
    }
}

public interface IAuctionService
{
    AuctionDTO GetAuction(int id);
    IEnumerable<AuctionDTO> GetAuctions();
    void OpenAuction(int id);
    void CloseAuction(int id);
    void Bet(int id, string customerName, int bid);
}

public interface IUnitOfWork : IDisposable
{
    IAuctionRepository Auctions { get; }
```

```
    void Commit();
}

public interface IAuctionRepository
{
    Auction Get(int id);
    IEnumerable<Auction> GetAll();
    void Update(Auction entity);
}

public class AuctionDTO
{
    public int ID { get; set; }
    public int Bid { get; set; }
    public string Leader { get; set; }
    public bool Started { get; set; }
    public bool Ended { get; set; }
}

public class Auction
{
    public virtual int ID { get; set; }
    public virtual int Bid { get; set; }
    public virtual string Leader { get; set; }
    public virtual bool Started { get; set; }
    public virtual bool Ended { get; set; }
}

public class InvalidAuctionException : Exception
{
    public InvalidAuctionException()
        : base()
    {
    }

    public InvalidAuctionException(string message = "ERROR: Invalid
auction")
        : base(message)
    {
    }

    public InvalidAuctionException(string message = "ERROR: Invalid
auction", Exception? innerException = null)
        : base(message,  innerException)
    {
    }
}

public class InvalidIdException : Exception
{
    public InvalidIdException()
        : base()
    {
```

```
    }

    public InvalidIdException(string message = "ERROR: Invalid ID")
        : base(message)
    {
    }

    public InvalidIdException(string message = "ERROR: Invalid ID",
Exception? innerException = null)
        : base(message, innerException)
    {
    }
}

public class InvalidNameException : Exception
{
    public InvalidNameException()
        : base()
    {
    }

    public InvalidNameException(string message = "ERROR: Invalid name")
        : base(message)
    {
    }

    public InvalidNameException(string message = "ERROR: Invalid name",
Exception? innerException = null)
        : base(message, innerException)
    {
    }
}
```

ДОДАТОК Г

Код прикладу №12 із набору даних дослідження

```
using System.Text;

namespace Dataset.Sample12;

public static class LUPDecomposition
{
    public static void Decompose(double[,] matrix, out double[,] L,
out double[,] U, out int[] P)
    {
        int n = matrix.GetLength(0);
        L = new double[n, n];
        U = new double[n, n];
        P = new int[n];

        for (int i = 0; i < n; i++)
            P[i] = i;

        for (int i = 0; i < n; i++)
        {
            double max = 0;
            int row = i;
            for (int k = i; k < n; k++)
            {
                if (Math.Abs(matrix[k, i]) > max)
                {
                    max = Math.Abs(matrix[k, i]);
                    row = k;
                }
            }

            if (max == 0)
                throw new InvalidOperationException("Матриця є
вироджено.");

            int temp = P[i];
            P[i] = P[row];
            P[row] = temp;

            for (int j = 0; j < n; j++)
            {
                double tmp = matrix[i, j];
                matrix[i, j] = matrix[row, j];
                matrix[row, j] = tmp;
            }

            for (int j = i; j < n; j++)
            {
                double sum = 0;
                for (int k = 0; k < i; k++)
                    sum += L[i, k] * U[k, j];
```

```
                U[i, j] = matrix[i, j] - sum;
            }

            for (int j = i + 1; j < n; j++)
            {
                double sum = 0;
                for (int k = 0; k < i; k++)
                    sum += L[j, k] * U[k, i];

                L[j, i] = (matrix[j, i] - sum) / U[i, i];
            }

            L[i, i] = 1;
        }
    }

    public static string GetMatrix(double[,] matrix)
    {
        StringBuilder result = new StringBuilder();

        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
                result.Append($"{matrix[i, j]}\t");
            result.Append('\n');
        }

        return result.ToString();
    }

    public static string GetPermutationMatrix(int[] P)
    {
        StringBuilder result = new StringBuilder();

        int n = P.Length;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
                result.Append(j == P[i] ? "1\t" : "0\t");
            result.Append('\n');
        }

        return result.ToString();
    }
}
```