УДК 004.415.53:004.8 DOI: 10.31673/2412-4338.2025.048914

Чичкарьов Євген Анатолійович
Державний університет інформаційно-комунікаційних технологій, Київ
ORCID: 0000-0002-4362-5129

Семенов Олександр Віталійович
Державний університет інформаційно-комунікаційних технологій, Київ
ORCID: 0009-0005-9749-3343

АВТОМАТИЗОВАНА ГЕНЕРАЦІЯ ТЕСТОВИХ ВИПАДКІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗА ДОПОМОГОЮ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ З ВИКОРИСТАННЯМ ПРОМПТ-ІНЖИНІРІНГУ

Анотація. *Тестування програмного забезпечення є критично важливою фазою життєвого циклу розробки, що забезпечує надійність та коректність програмних систем. Традиційна генерація тестових випадків може бути трудомісткою та трудомісткою, часто вимагаючи значних ручних зусиль. Зі швидким розвитком генеративного штучного інтелекту, такі інструменти, як великі мовні моделі, пропонують нові можливості для автоматизації цього процесу. У цій статті досліджується застосування цих інструментів на основі штучного інтелекту для генерації тестових випадків, оцінюючи їхню ефективність у досягненні повного покриття коду для різноманітних програмних задач, в тому числі задач обробки даних. У цій статті досліджується застосування різних великих мовних моделей, для автоматизації генерації програмного кода і тестових випадків при створенні програмного забезпечення. Встановлено, що розмір LLM-моделі суттєво впливає на якість згенерованих коду та тестів, тому моделі Gemma3:12B та GPT-5.1 та Gemini показують найкращі результати. Малі моделі (Gemma3:1B, Gemma2:7B) більш схильні до помилок і гірше покривають граничні випадкі у порівнянні з великими. Встановлено, що покриття тестів і якість коду для задач обробки даних значно підвищують Test-Driven-Development та Data-analysis промпти. Показано, що надійність чисельних обчислень також корелює з розміром LLM.. Ці результати підкреслюють потенціал генеративного ШІ для оптимізації робочих процесів тестування програмного забезпечення, звільняючи розробників від необхідності зосередитися на вирішенні проблем вищого порядку. Однак дослідження також підкреслює необхідність подальшого вдосконалення цих інструментів для підвищення їхньої надійності та стійкості. Ця робота є фундаментальним кроком до використання генеративного ШІ для трансформації практик розробки та тестування програмного забезпечення.*

Ключові слова: *Автоматизована генерація тестових випадків, Тестування програмного забезпечення за допомогою ШІ, Генеративний ШІ для тестування, Автоматизація тестування на базі ШІ, Генерація тестових випадків на основі ШІ*

Yevhen Chychkarov
State University of Information and Communication Technologies, Kyiv, Ukraine
ORCID: 0000-0002-4362-5129

Oleksandr Semenov
State University of Information and Communication Technologies, Kyiv, Ukraine.
ORCID: 0009-0005-9749-3343

AUTOMATED GENERATION OF SOFTWARE TEST CASES BY MEANS OF LARGE LANGUAGE MODELS USING PROMPT ENGINEERING

Abstract. *Software testing is a critical phase of the development lifecycle, ensuring the reliability and correctness of software systems. Traditional test case generation can be time-consuming and labor-intensive, often requiring significant manual effort. With the rapid development of generative artificial intelligence, tools such as large language models offer new opportunities to automate this process. This paper investigates the application of these artificial intelligence-based tools for test case generation, evaluating their effectiveness in achieving complete code coverage for a variety of software tasks, including data processing tasks. This paper investigates the application of various large language models to automate the generation of software code and test cases in software development. It is found that the size of the LLM model significantly affects the quality of the generated code and tests, so the Gemma3:12B and GPT-5.1 and Gemini models show the best results. Small models (Gemma3:1B, Gemma2:7B) are more error-prone and have poorer edge case coverage than large ones. Test coverage and code quality for data processing tasks are found to*

significantly improve Test-Driven-Development and Data-analysis prompts. The robustness of numerical computations is also shown to correlate with LLM size. These results highlight the potential of generative AI to streamline software testing workflows, freeing developers from the need to focus on solving higher-order problems. However, the study also highlights the need for further improvements to these tools to increase their robustness and robustness. This work is a fundamental step towards using generative AI to transform software development and testing practices.

Keywords: *Automated test case generation, AI-powered software testing, Generative AI for testing, AI-based test automation, AI-based test case generation*

1 Вступ

Забезпечення якості програмного забезпечення (QA) традиційно вимагає значних ручних зусиль, особливо на етапі створення тестових випадків.

В останні роки штучний інтелект (ШІ) суттєво вплинув на розробку програмного забезпечення, зокрема завдяки досягненням у машинному навчанні та обробці природної мови.

Моделі великих мов (LLM) – це моделі штучного інтелекту (ШІ), навчені на величезних текстових даних для розуміння та генерації людської мови, зазвичай з використанням архітектур глибокого навчання, таких як трансформери. Поява великих мовних моделей (LLM), таких як GPT-4, Gemini та Claude, відкрила нові горизонти для автоматизації QA. Ці моделі здатні розуміти контекст, граматику та наміри, викладені природною мовою, що робить їх ідеальними інструментами для перетворення неструктурованих вимог безпосередньо у високоякісні тестові сценарії [1].

Точне налаштування моделей великих мов значно покращує їхні можливості синтезу програм, досягаючи високої продуктивності в синтезі програм Python з описів природною мовою [2].

Оскільки всі публічні LLM навчаються на величезному корпусі загальних даних без будь-якої специфікації предметної області, можна вважати, що LLM – це генератор відповідей загального призначення, де запит може бути з будь-якої області, в якій були отримані навчальні дані LLM. Наразі більшість LLM використовуються в таких завданнях, як переклад, генерація контенту, підсумовування, аналіз настроїв питань і відповідей тощо. LLM останнім часом досягли значного прогресу в багатьох областях, і більшість популярних LLM доступні у вигляді додатків або з простими інтерфейсами користувача, які допомагають використовувати модель як сервіс.

Очікується, що LLM, навчені на достатньо великому корпусі даних, дають значущі результати в різних завданнях життєвого циклу розробки програмного забезпечення. Генерація тестових випадків є одним із таких завдань, яке, хоча й специфічне у своєму виконанні, може бути значною мірою залежним від предметної області в численних програмних застосунках.

Однак застосування штучного інтелекту, особливо моделей великих мов програмування (LLM), для верифікації програмного забезпечення залишається недостатньо дослідженим. Верифікація програмного забезпечення охоплює низку видів діяльності, включаючи тестування, під час якого програмне забезпечення виконується з різними вхідними даними для оцінки його поведінки та підтвердження його коректності. Загальні методології, що використовуються в цьому процесі, включають модульне тестування, інтеграційне тестування, системне тестування та методи формальної верифікації, особливо для критичних систем. Мета полягає у виявленні та виправленні дефектів на ранніх етапах життєвого циклу розробки, щоб забезпечити надійність та якість.

2 Огляд літератури

Використання LLM у тестуванні програмного забезпечення (Software Testing) є одним із найбільш динамічних напрямків досліджень у сфері автоматизації QA (Quality Assurance) з 2022 року. Література фокусується на здатності LLM розуміти вимоги, генерувати тестові сценарії та писати виконуваний код тестів. [3]

Генерація тестів на основі вимог природною мовою (NL-to-Test) – це один з найпоширеніших підходів, що використовує LLM для подолання розриву між бізнес-вимогами та технічним тестуванням [4]. Великі мовні моделі можуть автоматично ідентифікувати неясності або неповноту у вимогах перед генерацією тестів, що раніше було виключно ручним завданням [5].

Важливим завданням для LLM-моделей є генерація орієнтованих на поведінку сценаріїв (BDD-сценаріїв). В межах цього підходу LLM використовуються для генерації сценаріїв у форматі Gherkin (Given-When-Then). Це прискорює впровадження методології BDD (Behavior-Driven Development) [6].

Інший важливий напрямок використання LLM-моделей при тестування програмного забезпечення - генерація юніт-тестів (Code-to-Test). В межах цього підходу модель отримує функцію або клас і генерує модульні тести для перевірки її логіки. LLM-модель аналізує синтаксис, семантику та потенційні граничні випадки (boundary conditions) функції, щоб створити повний набір тестів, часто

використовуючи фреймворки, як-от JUnit, Pytest, або Jest. При порівнянні різних LLM-моделей основна метрика оцінки — покриття коду (Code Coverage). LLM здатні досягати високого рівня покриття, але можуть мати проблеми з генерацією тестів для складних, неявних граничних випадків [7].

Ще один важливий напрямок покращення генерації тестів програмного забезпечення за допомогою LLM – це генерація мутаційних тестів та тестів негативних сценаріїв. В межах цього напрямку виконують й тестування API. LLM можуть отримувати специфікації API (наприклад, OpenAPI/Swagger) і генерувати повні набори тестів для перевірки RESTful API, включаючи валідацію статус-кодів, тіл відповідей та заголовків. Наприклад, дослідження [8] продемонструвало, що LLM можуть ефективно генерувати тести, здатні виявляти мутації, значно підвищуючи якість тестового набору.

Важливим напрямком використання LLM в сфері тестування програмних засобів – створення тестів для регресійного тестування. Його основна мета — забезпечити, що система залишається стабільною, надійною та функціональною після будь-якої модифікації. Відомо досить багато відомостей про використання LLM для оновлення та генерації нових тестів після змін у кодовій базі (регресійне тестування). Приклади створення регресійних тестів за допомогою LLM наведено в [9].

Важливим фактором, що визначає успіх генерації коду та тестів, є наявність і особливості вхідної інструкції — підказки контексту або промпта (prompt). Великі мовні моделі (LLM), хоч і потужні, є досить чутливими щодо контексту, наданого користувачем, тому структура промпта має прямий вплив на точність та безпеку згенерованого коду.

Дослідження [10] показали, що промпти, які включають вимоги до стилю (наприклад, PEP 8 для Python), алгоритмічні обмеження (наприклад, "використовуй ітерацію, а не рекурсію") та безпекові перевірки, генерують більш якісний і менш вразливий код.

Для генерації тестів промпт-інжиніринг стає ще більш критичним, оскільки він повинен керувати LLM не лише стилем, але й логікою покриття. Наприклад, надання LLM ролі "Досвідченого QA-інженера" або "Тест-архітектора" змушує модель активувати знання про найкращі практики тестування, такі як тестування граничних умов (Boundary Testing) та негативне тестування [11-12]. За даними [12], завдання критерію оцінки підказок продемонструвало суттєве підвищення продуктивності, збільшивши точність з 46,2% до 64,0%. У випадку агента маршрутизатора було досліджено можливість покращення погано продуктивної підказки та меншої моделі, що відповідає сильнішій, за допомогою оптимізованих підказок. Хоча уточнення підказок збільшило точність з 85,0% до 90,0%, використання оптимізованої підказки з дешевшою моделлю не покращило продуктивність. Загалом, результати цього дослідження свідчать про те, що систематична оптимізація підказок може підвищити продуктивність LLM, особливо коли налаштування інструкцій та вибір прикладів оптимізуються разом.

При використанні для генерації тестів LLM мають схильність до генерації позитивних тестів, якщо не вказати явно інше. Для досягнення високого покриття гілок логіки коду необхідно прямо вимагати від ШІ перевірки винятків та неприпустимого вводу. Наприклад, автори [13] виявили, що LLM краще генерують ефективні регресійні тести, коли в промпті вказано фокус на змінених ділянках коду та потреба у виявленні помилок у цих ділянках.

Вимога до LLM генерувати вивід у чітко структурованому форматі (BDD-сценарії Gherkin, Pytest-код) забезпечує виконуваність артефактів. В огляді [14] зазначено, що LLM є особливо ефективними, коли використовуються як перекладачі з вимог природної мови на формальні, структуровані формати тестування, що мінімізує двозначність.

3 Завдання дослідження

Таким чином, сучасна література підтверджує, що LLM є цінним інструментом в арсеналі QA, здатним значно підвищити швидкість і покриття тестуванням.

Проте, їхнє використання не є універсальною заміною. Розширення використання LLM при тестуванні програмного забезпечення потріює вирішення наступних проблем:

- Якість тестування LLM суттєво залежить від якості наданого контексту (вимог, коду, документації), Як тип промпта та вибір LLM впливають на якість згенерованого коду?
- Не зрозуміла повністю можливість і необхідність використання комерційних LLM (GPT, Gemini) проти відкритих моделей (Llama, Mistral, Gemma) у завданні генерації різноманітних тестів.

Значний інтерес становлять питання генерації коду та тестування стосовно завдань обчислень, обробки даних та машинного навчання.

Для дослідження було відібрано низку базових задач програмування на python, які охоплюють

ключові структури (цикли, масиви, умовні оператори, рекурсія), а також обчислювальні завдань та завдань обробки даних з використанням пакетів numpy, scipy, pandas, scikit-learn. На відміну від класичного програмування, задачі Data Science часто оперують недетермінованими даними, NaN-значеннями та статистичними викидами. Це створює додаткові виклики для LLM: модель повинна не просто написати синтаксично вірний код, а й зрозуміти семантику даних.

Метою даної роботи є проведення системного вимірюваного дослідження якості коду і тестів, що генерують LLM, та обґрунтування оптимальних стратегій промптінгу для типових задач розробника програмного забезпечення та дата-аналітика.

4 Методологія дослідження

Генерація тестових випадків виконувалась в двох варіантах:

1. Власноруч написаний та перевірений код рішень для цих проблем був поданий на вхід LLM з вимогою згенерувати відповідні тестові випадки.
2. Завдання було сформульовано природною мовою, за допомогою LLM виконувалась генерація кода, що відповідає умовам завдання, і коду для виконання тестів.

Згенеровані ШІ тести виконувались в середовищі python для перевірки їхньої функціональності, точності та покриття коду.

Базові задачі, обрані для цього дослідження, охоплюють широкий спектр концепцій програмування, включаючи умовні оператори, цикли, масиви, рядки та рекурсію (див. нижче). Критерії відбору мали на меті забезпечити збалансоване поєднання складності, практичної застосовності та варіативності логічних структур. Обрані задачі варіюються від базових перевірок на відповідність та арифметичних операцій до складніших рекурсивних та сортувальних алгоритмів. Цей різноманітний набір дозволяє провести комплексну оцінку генеративних інструментів ШІ для створення ефективних тестових випадків у різних парадигмах програмування.

Генерація коду і тестів виконувалась за допомогою чата для роботи з ChatGPT (використано модель GPT-5.1) і також чата для роботи з Google Gemini (використано версію Gemini 2.5 Flash).

Для завантаження інших мовних моделей використано інтерфейс Ollama з попередним завантаженням моделей Gemma3:4b, Gemma3:12b, Llama2, Llama3.1 та інших.

Для більш змістовного аналізу процесів генерації кода і створення тестів було використано наступний набір задач аналізу даних (побудова моделей регресії, підготовка даних, їх групування і агрегація та інші).

Для кожної задачі вимагалося згенерувати код, згенерувати pytest-тести, пояснити принцип роботи.

В залежності від класу задач генерації коду і тестів було використано декілька різних типів промптів.

Для вирішення базових задач програування використовувались 3 типи промптів:

- мінімальний,
- з точки зору спеціаліста з тестування (з точки зору Test Architect),
- з точки зору інженер з контролю якості (з точки зору QA Engineer).

Для генерації обчислювального коду або обробки даних використано більш різноманітні запити:

- мінімальний (без умов стосовно точки зору, пояснень, тестів);
- науковий (вимагає математичних пояснень, оцінки похибки, порівняння методів);
- орієнтований на тестування (модель спочатку генерує тести, потім код; test-driven development);
- архітектурний (вимагає відповідну труктура проєкту, модулі, логування, PEP-8);
- сфокусований на швидкодії (NumPy, Numba, векторизація, алгоритмічна оптимізація);
- з акцентом на тестування (TDD - Test-Driven Development prompt);
- орієнтований на аналіз даних (Data-analysis-oriented prompt).

Операції з генерації коду та тестів, завантаження моделей, робота з ChatGPT та Gemini виконувались на ноутбуці з процесором AMD Ryzen 7 5700U з 16ГБ оперативної памяти і графічним ядром AMD Radeon graphics під управлінням Windows 11.

5 Результати і аналіз

Результат генерації тесту обчислення факторіалу за допомогою різних типів LLM-моделей (незалежно від промпта) наведено в таблиці 1. Як видно з таблиці, для відносно простих задач незалежно від промпту сучасні LLM генерують тести зі 100% покриття, для більш слабких моделей

попереднього покоління незалежно від промпта результат генерації не забезпечує повного покриття кода.

Таблиця 1

Характеристики теста для задачі обчислення факторіала, який згенеровано LLM

Тест	Класи LLM		
	Gemma:2b, llama2	Gemma3:1b, 4b, 12b, Llama3.1:8b	Gemini, GPT
Обчислення факториала позитивного числа	Згенеровано	Згенеровано	Згенеровано
Обчислення факториала негативного числа(N < 0)	Відсутне	Згенеровано	Згенеровано
Обчислення факториала числа 0 (N=0)	Згенеровано	Згенеровано	Згенеровано

Аналонгічні результати отримано і для інших варіантів генерації програм і тестів для них з допомогою різноманітних LLM-моделей: моделі попереднього покоління забезпечують гірше покриття у порівнянні з Gemini або GPT, а також вільними LLM сучасного покоління. Використання досконалого промпта «Ти — досвідчений QA-інженер. Створи функцію Python calculate_factorial(n) для обчислення факторіалу n з використанням циклу for. Обов'язково включи перевірку на від'ємні числа. Потім згенеруй вичерпний модульний тест Pytest, який повинен включати: 1. Позитивні сценарії (N=5, N=7); 2. Граничний випадок (N=0, N=1); 3. Негативний сценарій (перевірка, що виникає виняток ValueError для N=-1)» досить слабко вплинуло на результати генерації програм і тестів.

З заявленої точки зору Google Gemini створив відповідну функцію і тест зі 100% покриттям коду.

Якщо використати більш простий запит «Створи функцію Python calculate_factorial(n) для обчислення факторіалу n з використанням циклу for. Створити відповідні тести», який не надає явно точку зору, то за допомогою ChatGPT або Gemini створено як функцію розрахунку фактопіалу, так і модульний тест, але рівень покриття тесту погіршується.

Приклад вирішення більш складного завдання демонструє суттєвий вплив промпта на результат генерації теста:

1. Найпростіший варіант запиту: «Створити програму, що обчислює рішення лінійної системи рівнянь з матрицею NxN. Матриця і вектор правих частин генеруються вирадково. Використати numpy. Надрукувати розмір N, час генерації матриці і час рішення. Створити тести для перевірки функціонування. Додати графічну ілюстрацію залежності часу генерації і часу вішення від розміра матриці.»
2. Варіант запиту з врахуванням точки зору (наприклад, до попереднього формулювання додати «З точки зору спеціаліста з тестування програмного забезпечення треба: …».
3. Варіант завдання з врахуванням точки зору і міркувань з реалізації тестів (наприклад, вказати точку зору і вимогу генерувати теси для pytest).

Характеристика запитів (промптів) наведено в таблиці 2.

Результати аналізу згенерованого коду продемонстрували, що якість коду, рівень коментарів і підказок, повнота покриття тесту покращуються при переході першого до третього варіантів.

Таблиця 2

Характеристика промптів для вирішення задач тестування

№	Варіант	Для кого	Характеристика згенерованого кода
1	Базовий	Студент / початківець	Простий код, все в одному файлі
2	QA Engineer	Тестувальник	Більш суворі тести, фокус на перевірках
3	Test Architect	Досвідчений QA/Test Lead	Модульна архітектура, розбиття на компоненти

Дослідження поведінки інших великих мовних моделей продемонструвало суттевий вплив промпту на результати генерації коду і тестів.

З найпростішим промптом модель Gemma:2b лише згенерувала програму, не створюючи тести або пояснення. Спроба генерувати підказки для друку в коді українською не дали вдалого результату.

З тим же промптом модель Llama2:7b згенерувала програму з англомовними повідомденнями і

коментарями, і не згенерувала єдиного теста.

Моделі наступного покоління продемонстрували значно кращі можливості генерації коду.

Модель Gemma3:4b згенерувала структуровану програму з англомовними повідомленнями і коментарями, але не згенерувала єдиного теста.

Модель Gemma3:12b згенерувала добре структуровану програму з англомовними повідомленнями і коментарями, и згенерувала в коді рудіментарні тести.

Зміна тексту промпта на варіант з додаванням точки зору призвела до досить радікальної зміни поведінки моделей ШІ.

Модель Gemma3:12b згенерувала добре структурований код з якісними коментарями і підказками англійськоюмовою, але засоби тестування залишились на початковому рівні.

Модель Gemma3:4b згенерувала добре структурований код з якісними коментарями і підказками англійськоюмовою, але засоби тестування залишились на початковому рівні, але в поясненнях до коду надала рекомендацію: «Unit Tests: You could create unit tests using a testing framework (like unittest or pytest) to automate the testing process». Генерація тестів за допомогою команди «Створити тести для цього коду з використанням pytest» надала гарний зрозумілий результ.

Модель Llama3:8b також згенерувала добре структуровану програму з англомовними повідомленнями і коментарями, и згенерувала в коді рудіментарні тести.

Більш складне завдання містило питання завантаження і аналізу даних. Опис задачі (ціль для всіх промптів):

Дано CSV з даними про житлові об’єкти (area, bedrooms, bathrooms, year_built, neighborhood, price). Потрібно: завантажити дані (pandas); провести попередню обробку: відсутні значення, one-hot кодування категорій, масштабування числових ознак; побудувати просту модель регресії (LinearRegression / RandomForestRegressor) для прогнозу price; повернути метрики: RMSE, R^2; створити набор pytest-тестів, що перевіряють: коректність обробки NaN, правильність розмірностей, що модель навчається і видає предикти очікуваного розміру, та що RMSE < деякого допустимого значення на синтетичних даних.

Нижче наведено шаблони промптів, які використовуються для генерації коду/тестів:

1. Short prompt (скорочений)

2. Scientific prompt (науковий). Цей промпт передбачає надання точки зору і перелік умов генерації.

3. TDD prompt. Цей варіант генерації передбачає генерацію сперше тестів, а потім пов’язаного з ними коду.

4. Data-analysis oriented prompt передбачає наявнястьточки зору і перелік окремих задач, які треба вирішити за допомогою згенерованого коду.

Для оцінки якості згенерованого коду була використана сумарна суб’єктивно-об’єктивна метрика (від 0 ло 10 для найкращого кода), яка враховувала декілька параметрів: читабельність, використання best practices (pipeline, randomness control), обробку edge-cases, докстрінги і документацію, а також безпечне використання бібліотек. Оцінки отримані як узагальнення по прикладах і зведені в таблицю 4.

Таблиця 4

Якість згенерованого коду (0–10)

Модель	Оцінка якості коду (0 – 10) в залежності від варианта промпту				
	Short	Scientific	TDD	Data-analysis	Середнє
gemma2:7b	6.2	7.0	7.2	6.9	6.8
gemma3:1b	5.5	6.6	6.7	6.4	6.3
gemma3:4b	7.0	8.0	8.2	7.8	7.8
gemma3:12b	8.8	9.3	9.5	9.2	9.2
mistral:7b	6.8	7.6	7.9	7.5	7.4
llama3.1:8b	7.2	8.1	8.3	8.0	8.0
Gemini	8.9	9.4	9.6	9.3	9.3
GPT-5.1	9.1	9.7	9.8	9.6	9.6

Оцінка покриття тестів (pytest-покриття) була розрахована як відсоток важливих перевірок (NaN, empty DF, wrong types, shape, model predict shape, metric thresholds) у тестах, сгенерованих моделлю. Результати було отримано як середні по декількох задачах і наведено в таблиці 5.

Таблиця 5

Середне значення покриття кода згенерованими тестами

Модель	Покриття коду, %, для LLM-моделі для типу промпту				
	Short	Scientific	TDD	Data-analysis	Середнє
gemma2:7b	72	80	82	78	78
gemma3:1b	60	70	72	68	68
gemma3:4b	84	88	90	87	87
gemma3:12b	93	95	96	94	94.5
mistral:7b	78	83	85	82	82
llama3.1:8b	86	89	91	88	88.5
Gemini	94	96	97	95	95.5
GPT-5.1	95	97	98	96	96.5

6 Обговорення результатів

З проведених досліджень випливає, що генеративний ШІ має потенціал трансформувати розробку програмного забезпечення і вирішення задач аналізу даних, звільнивши спеціалістів від рутинних задач і забезпечити їм можливості зосередитися на творчому вирішенні проблем та інноваціях. Майбутня робота має бути спрямована на подальше вдосконалення цих інструментів, що дозволить їм вирішувати складніші завдання та розширити їхнє застосування на інші галузі комп'ютерних наук.

Отримані результати підтверджують загальну закономірність: більші моделі (gemma3:12b, Gemini, GPT-5.1) суттєво краще опрацьовують граничні випадки, генерують більш повні тести і рідше генерують помилки.

Для отримання найбільш повних наборів тестів, треба формулювати Test-Driven Development промпт (TDD-промпт). В цьому випадку спочатку LLM згенерує тести, потім вже потім реалізацію кода рішення завдання.

Для створення трохи спрощених прототипів корисні найменші за кількістю параметрів і об'ємом моделі (gemma3:1b, gemma2:7b), але для отримання кінцевих програмних продуктів або розрахунків їх результати треба перевіряти. Крім того, ці моделі не дуже вдало генерують багатомовні пояснення, коментарі і підказки при друку. Одним із цікавих спостережень став стрибок якості між Gemma 2 та прототипом Gemma 3. Нова версія демонструє значно краще розуміння контексту бібліотек. Якщо Gemma 2 часто намагалася ітерувати рядки DataFrame через цикли for (що є анти-патерном у Pandas), то Gemma 3 стабільно використовувала векторизовані операції.

За усіма метриками найкращі результати забезпечили моделі Gemini і GPT-5.1. Можливою альтернативою є достатньо потужні і об'ємні моделі gemma3:12b або Llama3.1:8b. Крім того, було виявлено, що моделі OpenAI (GPT-4/5.1) та Google (Gemini) найкраще справляються з генерацією синтетичних даних для тестів. Вони створюють реалістичні розподіли, тоді як простіші моделі часто заповнювали DataFrame нулями або одиницями, що заважало повноцінно протестувати статистичні функції. У складних промптах, де навмисно були сформульовані суперечливі умови задачі, GPT-5.1 додавала відповідні коментарі в код.

Висновки

1. Встановлено, що розмір LLM-моделі суттєво впливає на якість згенерованих коду та тестів, тому Gemma3:12B та GPT-5.1 або Gemini показують найкращі результати.
2. Встановлено, що малі моделі (Gemma3:1B, Gemma2:7B) більш схильні до помилок і гірше покривають граничні випадки у порівнянні з веикими.
3. Показано, що покриття тестів і якість коду для задач обробки даних значно підвищують Test-Driven-Development та Data-analysis промпти.
4. Встановлено, що надійність чисельних обчислень також корелює з розміром LLM.
5. Якщо вимоги природною мовою є двозначними, неповними або суперечливими, для генерації коректного коду і тестів з використанням LLM може потребувати виконання декількох ітерацій з уточненням вимог.
6. Показано, що для послідовностей команд обробки даних (piplines) є сенс використовувати Test-Develioment-Driven-prompt + великі LLM, потім додавати ручну ревізію і окремо генерувати performance/regression tests.

Список використаних джерел

1. R. A. Poldrack, T. Lu, and G. Beguš, "AI-assisted coding: Experiments with GPT-4," arXiv preprint, vol. 2304.13187, 2023. [Online]. Available: https://arxiv.org/abs/2304.13187.
2. 2. J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program Synthesis with Large Language Models," arXiv preprint arXiv:2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732.
3. Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., & Chen, Z. (2023). A Survey on Large Language Models for Software Engineering. *ArXiv, abs/2312.15223*.
4. Celik, A., & Mahmoud, Q. H. (2025). A Review of Large Language Models for Automated Test Case Generation. *Machine Learning and Knowledge Extraction*, *7*(3), 97. https://doi.org/10.3390/make7030097
5. Choi, Wan Chong & Chang, Chi In. (2025). Enhancing Python Programming through ChatGPT-5 Study Mode: A Study on Learning Achievement, Motivation, and Self-Regulated Learning in Serious Game Learning with CodeCombat. 10.20944/preprints202508.1722.v1.
6. Fernandes, Hiago & Perkusich, Mirko & Albuquerque, Danyllo & Silva, Izabella & Santos, Danilo & Gorgônio, Kyller & Perkusich, Angelo. (2025). A Comparative Study of LLMs for Gherkin Generation. 171-181. 10.5753/sbes.2025.9888.
7. Steenhoek, B., Tufano, M., Sundaresan, N., & Svyatkovskiy, A. (2023). Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, 37-44.
8. Rehan, S., Al-Bander, B., & Al-Said Ahmad, A. (2025). Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation. *Electronics*, *14*(7), 1463. https://doi.org/10.3390/electronics14071463
9. Cavalcantia, Artur & Accioly, Luan & Valença, George & Nogueira, Sidney & Morais, Ana & Oliveira, Antônio & Gomes, Sérgio. (2025). Automating Test Design Using LLM: Results from an Empirical Study on the Public Sector. Conference on Digital Government Research. 1. 10.59490/dgo.2025.1025.
10. Liu, Zhihong & Yang, Zezhou & Liao, Qing. (2024). Exploration On Prompting LLM With Code-Specific Information For Vulnerability Detection. 273-281. 10.1109/SSE62657.2024.00049.
11. Raja, Rahul & Vats, Arpita & Roy, Sudipta. (2025). Aligning Prompts with Ranking Goals: A Technical Review of Prompt Engineering for LLM-Based Recommendations. 10.20944/preprints202509.1959.v1.
12. Lemos, Francisca & Alves, Victor & Ferraz, Filipa. (2025). Is It Time To Treat Prompts As Code? A Multi-Use Case Study For Prompt Optimization Using DSPy. 10.48550/arXiv.2507.03620.
13. Abbassi, A.A., Silva, L.D., Nikanjam, A., & Khomh, F. (2025). ReCatcher: Towards LLMs Regression Testing for Code Generation. *ArXiv, abs/2507.19390*.
14. Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., & Chen, Z. (2023). A Survey on Large Language Models for Software Engineering. ArXiv, abs/2312.15223.