БАКАЛАВРСЬКА РОБОТА

БР.ІП – 42.00.00.000 ПЗ

Група ІП-22-2

Матіїшин Андрій

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Матіїшин Андрій Васильович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Автоматизація тестування за допомогою великих мовних моделей
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____А.В. Матіїшин_____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Бандура Вікторія Валеріївна, к.т.н., доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Матіїшин Андрій Васильович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Автоматизація тестування за допомогою великих мовних моделей"

керівник проекту (роботи) Бандура Вікторія Валеріївна, к.т.н., доцент _____
затверджені наказом вищого навчального закладу від “ 21_ ” _квітня 2026 р. № ___179/7____

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Розподіл вибраних досліджень за роками (рис. 2.2, ст. 29)
2 Найпопулярніші моделі в основних сімействах великих мовних моделей (рис. 2.5, ст. 32)
3 Процес створення бази даних. (рис. 2.6, ст. 34)
4 Основні ролі та функції (рис. 3.1, ст. 50)
5 Архітектура системи (рис. 3.4, ст. 54)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Матіїшин А.В.
(підпис)

Керівник роботи _____Бандура В.В.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 92 сторінки, 19 рисунків, 11 таблиць, список використаних джерел із 40 найменувань.

Метою роботи є дослідження методів автоматизації тестування програмного забезпечення із застосуванням великих мовних моделей та аналіз ефективності їх використання для підвищення якості.

Об'єкт дослідження: процеси тестування програмного забезпечення.

Предмет дослідження: методи, моделі та інструменти автоматизації тестування із використанням великих мовних моделей.

Результати дослідження: проведено аналіз сучасних підходів до тестування програмного забезпечення, досліджено можливості великих мовних моделей у генерації тестових сценаріїв, автоматизації тестування та порівняно їх ефективність із традиційними методами тестування.

У першому розділі розглянуто теоретичні основи тестування програмного забезпечення, основні види тестування, а також еволюцію інструментів автоматизації тестування та роль штучного інтелекту.

У другому розділі досліджено методи та інструменти автоматизації тестування із застосуванням великих мовних моделей, включаючи підходи до генерації тестів, аналізу вимог користувачів та моделювання сценаріїв використання (Use Case).

У третьому розділі розглянуто практичну реалізацію системи автоматизації тестування на основі великих мовних моделей, описано інструменти створення тестів, проведено оцінку ефективності.

Висновок: використання великих мовних моделей у процесі тестування дозволяє значно підвищити швидкість створення тестів, розширити покриття тестування та знизити витрати часу на розробку тестових сценаріїв.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ВЕЛИКІ МОВНІ МОДЕЛІ, ШТУЧНИЙ ІНТЕЛЕКТ, ГЕНЕРАЦІЯ ТЕСТІВ, QA, TESTING, MACHINE LEARNING, CI/CD.

ANNOTATION

The bachelor's thesis thesis of 92 pages, 19 figures, 11 tables, and a reference list with 40 sources.

The aim of the work is to study methods of test automation using large language models and to analyze their effectiveness in improving software quality, testing speed, and coverage.

Research object: software testing processes.

Research subject: methods, models, and tools for test automation using large language models, including test generation, code analysis.

Research results: modern approaches to software testing were analyzed, the capabilities of large language models in test generation and automation were studied, and their effectiveness was compared with traditional testing methods.

The first section describes the theoretical foundations of software testing, types of testing, and the evolution of automation tools, including the role of artificial intelligence in testing processes.

The second section analyzes methods and tools for test automation using large language models, including approaches to test generation, user requirements analysis, and use case modeling.

The third section covers the practical implementation of a testing automation system based on large language models, describes tools for test creation, presents experimental evaluation results, and compares AI-generated testing with manual testing approaches.

Conclusion: the use of large language models in testing significantly improves test generation speed, increases coverage, and reduces development time, but requires additional quality control and validation of results.

KEY WORDS: TEST AUTOMATION, LARGE LANGUAGE MODELS, ARTIFICIAL INTELLIGENCE, TEST GENERATION, QA, SOFTWARE TESTING, MACHINE LEARNING, CI/CD.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ 9
1.1 Основи тестування програмного забезпечення 9
1.2 Еволюція інструментів автоматизації тестування 14
1.3 Великі мовні моделі та їх застосування в тестуванні ПЗ 20
1.4 Висновки по розділу ……..…… 24

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ З ВИКОРИСТАННЯМ LLM 25
2.1 Методологія застосування великих мовних моделей у тестуванні 25
2.2 Методи генерації тестових сценаріїв та тестових даних 32
2.3 Аналіз вимог користувачів та моделювання тестових сценаріїв 43
2.4 Висновки по розділу ……..… 47

РОЗДІЛ 3. РОЗРОБКА, ІНТЕГРАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РІШЕНЬ НА ОСНОВІ LLM 48
3.1 Розробка інструменту автоматизації тестування на основі штучного інтелекту: технології та підходи 48
3.2 Реалізація та аналіз пілотного проєкту автоматизованого тестування 57
3.3 Оцінювання ефективності генеративного штучного інтелекту порівняно з ручним тестуванням 73
3.4 Висновки по розділу …… ..90

ВИСНОВКИ 91
СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА 93

ВСТУП

У сучасному світі програмне забезпечення відіграє ключову роль у функціонуванні більшості сфер діяльності - від бізнесу та фінансів до медицини та державного управління. Зі зростанням складності програмних систем та підвищенням вимог до їхньої якості особливої актуальності набуває ефективне тестування програмного забезпечення. Традиційні підходи до тестування часто вимагають значних часових і людських ресурсів, що ускладнює забезпечення повного покриття тестами та швидкого реагування на зміни в коді. У цьому контексті використання великих мовних моделей відкриває нові можливості для автоматизації тестування, підвищення продуктивності розробників і покращення якості програмних продуктів.

Актуальність теми зумовлена необхідністю підвищення ефективності процесів тестування в умовах швидкої розробки програмного забезпечення, зокрема в середовищах безперервної інтеграції та доставки (CI/CD). Існуючі інструменти автоматизації тестування, хоча й забезпечують значну підтримку, часто потребують ручного налаштування та написання тестових сценаріїв. Великі мовні моделі дозволяють автоматизувати генерацію тестів, аналіз вимог, виявлення помилок і навіть оптимізацію тестових наборів, що робить їх перспективним інструментом у сучасній інженерії програмного забезпечення.

Метою роботи є дослідження та розробка підходів до автоматизації тестування програмного забезпечення із використанням великих мовних моделей, а також оцінка ефективності їх застосування у порівнянні з традиційними методами тестування.

Завданнями дослідження є аналіз теоретичних основ тестування програмного забезпечення та сучасних підходів до його автоматизації, дослідження можливостей великих мовних моделей у генерації тестів і підтримці тестових процесів, визначення вимог до систем автоматизації тестування, розробка архітектури рішення, реалізація прототипу системи автоматизованого тестування, а також проведення експериментальної оцінки ефективності запропонованого підходу та порівняння його з традиційними методами.

Об’єктом дослідження є процеси тестування програмного забезпечення, що включають планування, створення тестів, їх виконання та аналіз результатів.

Предметом дослідження є методи, моделі та інструменти автоматизації тестування із застосуванням великих мовних моделей, зокрема генерація тестових сценаріїв, аналіз коду та підтримка тестових процесів.

Методи дослідження включають аналіз наукових джерел і сучасних практик тестування, порівняльний аналіз традиційних і інтелектуальних методів автоматизації, моделювання процесів тестування, експериментальні дослідження для оцінки ефективності запропонованого підходу, а також статистичний аналіз отриманих результатів.

У першому розділі роботи розглянуто теоретичні основи тестування програмного забезпечення, класифікацію видів тестування та еволюцію інструментів автоматизації. **У другому розділі** досліджено методи та інструменти автоматизації тестування із використанням великих мовних моделей, включаючи аналіз вимог користувачів і моделювання сценаріїв використання. **У третьому розділі** представлено реалізацію системи автоматизованого тестування, описано використані технології, проведено експериментальне дослідження та оцінено ефективність застосування великих мовних моделей у тестуванні.

Очікується, що результати роботи дозволять підвищити ефективність процесів тестування, скоротити витрати часу на створення тестів, покращити їх якість та забезпечити більш повне покриття програмного забезпечення. Використання великих мовних моделей сприятиме автоматизації рутинних завдань тестування та підвищенню продуктивності розробки програмних систем.

Бакалаврська робота містить 92 сторінок, 19 рисунків, 11 таблиць, 3 розділи, висновки та список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ТА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

1.1 Основи тестування програмного забезпечення

Тестування програмного забезпечення – це процес, який зосереджений на оцінці впровадження системи для виявлення потенційних збоїв та забезпечення надійності [1] . Тестування було критично важливою діяльністю з перших днів існування комп'ютерних систем, безпосередньо впливаючи на якість та надійність програмних продуктів і послуг [2] . Зі зростанням складності програмних систем у різних областях, потреба в надійному тестуванні стала ще більш очевидною [3] . Незважаючи на те, що недостатнє тестування є найдорожчим та найтривалішим етапом розробки програмного забезпечення [2] , воно значно збільшує ризик створення низькоякісного програмного забезпечення, що призводить до фінансових втрат, негативного досвіду користувачів та репутаційної шкоди організаціям [4] . Для вирішення цих проблем були розроблені комплексні підходи до тестування, включаючи модульне, інтеграційне та наскрізне тестування, щоб гарантувати, що сучасні програмні системи відповідають цим вимогам та залишаються бездефектними [5] . У відповідь на потребу в підвищенні ефективності, автоматизоване тестування викликало значний інтерес з боку дослідницької спільноти [6] та галузі [7] . З широким впровадженням гнучких методологій автоматизоване модульне тестування стало стандартною практикою серед розробників [7] . Крім того, очікується, що досягнення в генеративному штучному інтелекті покращать та розвинуть процеси автоматизованого тестування, пропонуючи нові можливості для покращення якості програмного забезпечення та оптимізації робочих процесів розробки [6].

Великі мовні моделі (LLM) викликали значний інтерес в останні роки та розглядаються як перспективні помічники в кодуванні, особливо в автоматизованій генерації тестових випадків, завдяки їхній здатності розуміти вимоги природної мови та генерувати відповідний код [8 , 9] . Завдяки своїй

постійній еволюції LLM продемонстрували виняткові можливості у генерації функціонального коду з описів природною мовою, значно підвищуючи ефективність розробки програмного забезпечення шляхом автоматизації завдань, зменшуючи людські помилки та дозволяючи розробникам зосередитися на більш складних та креативних аспектах програмування [10] .

У цій роботі представлено огляд використання LLM для автоматизованої генерації тестових випадків. Основний внесок цього дослідження полягає в наступному.

- Аналіз запропонованих методів: Наведено категоризацію методів використання LLM у генерації тестових випадків. Ці методи класифікуються на швидке проектування та інженерію, підходи на основі зворотного зв'язку, методи точного налаштування та попереднього навчання моделі, а також гібридні підходи, які пропонують розуміння їхніх сильних та обмежених сторін.
- Оцінка ефективності: Було проведено оцінку ефективності LLM відносно сучасних інструментів тестування, з акцентом на випадки, коли LLM перевершують та не досягають успіху порівняно з існуючими інструментами.
- Майбутні напрямки досліджень включають визначення ключових областей для вдосконалення, зокрема розширення застосовності LLM до кількох мов програмування, інтеграцію предметно-специфічних знань та використання гібридних методів для вирішення поточних проблем.

LLM використовують нейронні мережі, що містять велику кількість параметрів, яка може перевищувати мільярди. Їх можна навчати самостійно, а їх попереднє навчання може включати великі корпуси з Інтернету. Вони можуть вивчати складні шаблони, мовну семантику та нюанси, використовуючи свої методології навчання. Ці здібності роблять їх чудовими кандидатами для вирішення різних мовних завдань, таких як переклад, реферування тексту та аналіз настроїв. Важливо, що їх можна точно налаштувати для спеціалізованих завдань, що є ключовою особливістю, яка дозволяє їм досягати кращих результатів [11].

Методи LLM суттєво вплинули на галузь моделювання мови. Вони розвинулися з ранніх мовних моделей та нейронних мереж. У минулому

використовувалися статистичні підходи та n-грамні моделі, але вони мали труднощі з вираженням довгострокової взаємозалежності та контексту в мові. Нещодавні досягнення в нейронних мережах призвели до розробки рекурентних нейронних мереж (RNN), які можуть моделювати послідовні дані. Однак RNN мають обмеження через свої зникаючі градієнти та довгострокові залежності. Більш недавній прорив у LLM ознаменувався розробкою архітектури трансформатора, яка ефективно керує довгостроковими залежностями та розуміє зв'язки між словами за допомогою механізму самоуваги [11]. Ця розробка відкрила двері до моделі генеративного попередньо навченого трансформатора (GPT) та привернула значний інтерес з боку промисловості, наукових кіл та широкої громадськості [12].

Хоча методи LLM широко застосовуються та виявляються переконливими в задачах традиційних мов програмування, їх застосування в програмній інженерії також привернуло значну увагу. LLM досліджуються та вивчаються на всіх етапах життєвого циклу розробки програмного забезпечення, включаючи інженерію вимог, проектування, розробку, тестування та підтримку [12].

Згідно з Міжнародним стандартом 24765-2017-ISO/IEC/IEEE «Системна та програмна інженерія – Термінологія» [13], «тестування» визначається як «діяльність, під час якої система виконується за певних умов, результати спостерігаються або записуються, а також проводиться оцінка певного аспекту системи або компонента». Зі швидким розвитком індустрії програмного забезпечення, потреба в покращенні цієї діяльності за допомогою ефективних методів та зменшенні ризику дороговартісних катастроф, які можуть завдати шкоди населенню, стала пріоритетною [14].

Тестування може виконуватися на різних етапах життєвого циклу програмного продукту та охоплювати всю систему або її частину. Модульне, інтеграційне, системне та регресійне тестування є добре відомими стратегіями тестування. Яка б стратегія не використовувалася в будь-який момент часу, ймовірно, потрібно буде генерувати тестові випадки, що, можливо, є однією з найбільш інтенсивних видів тестування програмного забезпечення [15]. Отже,

генерація тестових випадків була ретельно вивчена, що призвело до розробки різних методів та інструментів для автоматичної або напівавтоматичної генерації наборів тестів [16].

Генерація тестових випадків – це процес створення набору тестів з інформаційного артефакту за допомогою механізму генерації, який зазвичай керується критерієм адекватності тестових даних. Після генерації тестового випадку він виконувався на тестованій системі та оцінювався за допомогою тестового оракула, щоб визначити, чи пройшов він перевірку, чи ні. Інформаційні артефакти можуть мати кілька форм і є джерелом змістовних тестових випадків. Це може бути формальна специфікація, проектний документ або вихідний код. Механізм генерації – це алгоритм або стратегія, яка створює тестові випадки на основі інформаційних артефактів. Критерій адекватності тестових даних вимірює та керує якістю згенерованих тестових випадків. Нарешті, тестовий оракул використовувався для оцінки правильності результату виконання заданого тесту на тестованій системі [16].

У багатьох дослідженнях досліджувався взаємозв'язок між LLM та генерацією тестових випадків, де LLM часто слугують основним механізмом генерації. Хоча це може здатися простим завданням генерації коду, у контексті тестування програмного забезпечення генерація різноманітного набору змістовних тестових даних для покращеного покриття коду та необхідність перевірки згенерованих тестових випадків мають унікальні характеристики, що призвело до інноваційних гібридних підходів, у яких LLM поєднуються з різними інструментами та методологіями тестування [9] .

Інтеграція LLM у систему забезпечення якості програмного забезпечення розширила можливості автоматизації завдань, таких як генерація тестів, пропонуючи значні покращення ефективності та надійності [17]. Одним з основних напрямків досліджень тестування програмного забезпечення була автоматизована генерація наборів тестів, яка встановлює цінні орієнтири для порівняння нових методів на основі LLM та гібридних методів [12]. У сфері генерації тестових випадків традиційні методи, такі як стратегії на основі пошуку, обмежень та випадкових варіантів, спрямовані на максимізацію

покриття, але часто створюють тести, яким бракує різноманітності та значущості, тоді як LLM, з їх продемонстрованим успіхом у генерації коду, пропонують перспективну альтернативу, дозволяючи створювати різноманітні тестові випадки, покращуючи покриття та сприяючи співпраці через генерацію тестових випадків на основі природної мови [9] .

Наприклад, роль проектування підказок досліджувалася в таких дослідженнях, як [18], в яких досліджувалися можливості ChatGPT (GPT-3.5) у створенні наборів модульних тестів за допомогою простих підказок. Це дослідження підкреслює готовність моделі до використання без врахування зворотного зв'язку чи коригувань, проливаючи світло на її потенціал та обмеження в охопленні тестових випадків, коректності та виявленні помилок. Окрім проектування підказок, підходи, засновані на зворотному зв'язку, такі як ChatUniTest [19], включають валідацію та виправлення, що підвищує якість тестових випадків шляхом динамічного їх уточнення на основі виявлених помилок.

Інші підходи використовують точне налаштування моделі та попереднє навчання для узгодження LLM з вимогами тестування програмного забезпечення. Одним із прикладів є CAT-LM [20], спеціалізований LLM, попередньо навчений на проектах Python та Java, який фіксує зв'язок між кодом та пов'язаними з ним тестами, що дозволяє генерувати висококонтекстуалізовані та точні тестові випадки. Крім того, гібридні методи, такі як CODAMOSA [21], поєднують сильні сторони LLM та тестування програмного забезпечення на основі пошуку (SBST), використовуючи LLM для генерації тестових випадків, коли тести, згенеровані SBST, досягають плато покриття, що демонструє, як ці дві парадигми можуть працювати синергетично. Ці різноманітні стратегії ілюструють, як LLM використовуються для вирішення проблем у генерації тестових випадків, а також відкривають шляхи для їх інтеграції з існуючими інструментами та робочими процесами.

Відображаючи цей зростаючий імпульс, також з'явилися зусилля щодо обстеження, спрямовані на консолідацію досліджень у цій галузі. Ван та ін. [9] надають комплексний огляд 102 досліджень, що охоплюють широкий спектр

завдань тестування програмного забезпечення, від генерації модульних тестів до ремонту програм. Зовсім недавно провели систематичний огляд, зосереджений на модульному тестуванні, проаналізувавши 105 досліджень станом на березень 2025 року (arXiv:2506.15227). Поява таких опитувань підкреслює як швидке зростання цієї галузі, так і зростаючу потребу в синтезованих перспективах для керівництва майбутніми дослідженнями. У сукупності ці розробки вказують на галузь, яка швидко розвивається, але продовжує стикатися з відкритими викликами.

Незважаючи на свою універсальність та перспективність, поточні дослідження мають кілька обмежень. Наприклад виявили, що тести, згенеровані ChatGPT, часто демонструють покращену читабельність, але не завжди перевершують тести, згенеровані EvoSuite, за оцінкою. Натомість повідомили про досягнення 100% охоплення за допомогою ChatGPT; однак оцінці бракувало повноти, що ускладнювало узагальнення результатів для різних сценаріїв. Крім того, підкреслили занепокоєння щодо обчислювальних витрат та продуктивності, які впливають на практичність використання LLM для генерації тестових випадків. Ці проблеми підкреслюють необхідність подальшого вдосконалення та адаптованих стратегій, що спонукає до огляду для консолідації існуючих досліджень, виявлення тенденцій та спрямування майбутньої роботи в цій новій галузі.

1.2 Еволюція інструментів автоматизації тестування

Швидкий розвиток різних програмних технологій, що використовують штучний інтелект (ШІ), в останні роки розкрив значний потенціал у сфері освіти [1 , 2] . ШІ пропонує численні методи, технології та інструменти для навчання та оцінювання, а також для адміністрування освітніх процесів. Це включає використання таких технологій, як обробка природної мови за допомогою великих мовних моделей [3, 4] , технології розпізнавання зображень за допомогою комп'ютерного зору [5, 6] , аналіз даних та прогнозування за допомогою нейронних мереж [7, 8] для покращення процесу навчання тощо.

Зростаючий інтерес д нових тенденцій використання штучного інтелекту в освіті підтверджується зростанням кількості наукових публікацій на цю тему [9, 10] . Ці публікації охоплюють широкий спектр питань - від технічних деталей, пов'язаних з інтеграцією штучного інтелекту в програмні додатки [11] , до етичних проблем, ризиків та загроз, пов'язаних з використанням штучного інтелекту [12] . Дослідження показують, що штучний інтелект має потенціал для підвищення якості освіти, роблячи її більш доступною та адаптованою до потреб окремих учнів [13 , 14] .

Штучний інтелект знаходить широке застосування в різних галузях освіти - соціальних науках [15] , інженерній освіті [16] , природничих науках [17] , медичній освіті [18] , науках про життя [19] , вивченні мов [20] та інших [21].

Загалом, виділилися п'ять основних напрямків досліджень: оцінювання/евалюація, прогнозування, асистент на основі штучного інтелекту, інтелектуальна система навчання (ІТС) та управління навчанням студентів [22], кожен з яких розкриває окрему галузь зі значним потенціалом для інновацій в освітній сфері.

Використання ШІ охоплює різні освітні види діяльності та процеси. Інструменти ШІ використовуються для створення інтелектуальних систем навчання, систем профілювання та прогнозування, а також для адаптивного навчання [23]. Дослідження демонструють позитивну роль ШІ у покращенні персоналізованого навчального досвіду, виявленні студентів з групи ризику та автоматизації адміністративних завдань [24]. Різні алгоритми та технології ШІ також застосовуються для забезпечення якості вищої освіти. Аналіз даних з різних показників, що надаються внутрішніми системами якості навчальних закладів, разом з оцінкою їхньої відповідності цільовим показникам, дозволяє на ранній стадії виявити слабкі місця та прогалини [25].

Чат-боти на основі штучного інтелекту знаходять дедалі ширше застосування в освіті. Вони пропонують значні можливості для підтримки досліджень, автоматизованого оцінювання та покращення взаємодії людини з комп'ютером [26]. Інструменти віртуальних асистентів надають учням

інтерактивні та захопливі способи опанування складних теорій та проведення самостійних досліджень [27, 28].

Одним із найпопулярніших чат-ботів на базі штучного інтелекту є ChatGPT. Його можна використовувати для викладання, навчання та проведення наукових досліджень [29]. ChatGPT має потенціал для написання домашніх завдань, есе та курсових робіт [30, 31]. У деяких випадках неможливо визначити, чи була дана робота створена студентом чи ChatGPT [32]. Використовуючи її без встановлення відповідних правил і процедур, це може призвести до руйнування традиційних механізмів оцінювання студентських екзаменаційних робіт. Більше того, деякі дослідники стверджують, що чат-бот може зробити певні традиційні форми завдань, такі як написання есе, застарілими [33]. Відсутність навичок критичного мислення та труднощі в оцінці інформації, що генерується ChatGPT, є іншими проблемами [34]. Це може призвести до різних проблем в академічному розвитку учнів [35].

ChatGPT може використовуватися викладачами для різних видів діяльності, включаючи створення навчальних матеріалів, екзаменаційних матеріалів та тестів, аналіз та надання відгуків щодо есе та інших завдань, створення навчальної документації, такої як навчальні плани, програми та розклади, а також пропонування персоналізованих пояснень до складного освітнього контенту. Він особливо популярний для створення тестів, оскільки викладачі можуть або надавати конкретний контент, або покладатися на базу знань чат-бота [36].

Інтеграція можливостей ChatGPT у програмну систему для автоматизованого створення та управління тестами є природним кроком вперед в оптимізації навчальних процесів. Переваги автоматичного створення тестових питань на основі наданого освітнього контенту, пропонування можливих відповідей та оцінювання правильних, управління виконанням тестів та оцінювання відповідей студентів як альтернативи ручній роботі зі створення питань, структурування тестів та оцінювання студентських тестів очевидні. Однак використання програмних застосунків на базі штучного інтелекту для управління тестами приносить додаткові переваги. Вони можуть запропонувати

більшу адаптивність, створюючи тести з різним рівнем складності для оцінки різних когнітивних навичок, підтримуючи сучасні освітні парадигми, зосереджені на диференційованому та адаптивному навчанні. Здатність ChatGPT створювати різні типи питань - від питань з множинним вибором до есе - дозволяє створювати тести, що заохочують критичне та творче мислення. Здатність чат-бота надавати диференційований, негайний зворотний зв'язок з короткими або детальними поясненнями правильних та неправильних відповідей допомагає учням зрозуміти свої помилки та сприяє їхньому прогресу.

У цій роботі представлено підхід до розробки програмного застосунку, який використовує можливості ChatGPT для створення та управління тестами. Мета полягає в тому, щоб полегшити роботу викладачів шляхом створення тестових питань на основі заданих текстів, структурування тестів та автоматизації процесу проведення тестового оцінювання. Тести зберігаються в базі даних для подальшого використання та автоматично оцінюються після завершення. У цьому контексті розроблений застосунок є інноваційним інструментом, який оптимізує процеси оцінювання та підтримує персоналізоване навчання.

Створення тестів зазвичай вимагає значного часу та людських ресурсів. Крім того, тести необхідно оцінювати на надійність та валідність, а пілотне тестування є важливим для виявлення слабких місць та прогалин, а також для їх усунення [37]. Це мотивує освітян активно шукати способи автоматизації процесу, включаючи генерацію тестових питань, побудову та оцінку тестів [38]. Спочатку додатки розроблялися з використанням традиційних методів та технологій.

Перші такі програми спиралися на ручне введення запитань та відповідей. Викладачі створювали запитання вручну за допомогою текстових редакторів або спеціалізованих програмних застосунків для створення тестів, що вимагало значного часу та зусиль. З часом з'явилися нові розробки, які пропонували шаблони та попередньо підготовлені анкети, які можна було змінювати та адаптувати відповідно до потреб користувачів [39]. Шаблони визначають структуру різних типів тестових запитань та включають речення з фіксованим

текстом, у яких є заповнювачі. Для створення конкретного запитання ці заповнювачі замінюються відповідним текстом [40].

Прагнення до швидкого створення тестів призвело до розробки застосунків із фіксованими питаннями. Ці застосунки часто використовують статичні бази даних із заздалегідь створеними питаннями та відповідями. Викладачі можуть вибирати питання з цих баз даних для створення тестів. У деяких випадках питання класифікуються та маркуються відповідно до теми, рівня складності та типу питання (з вибором однієї правильної відповіді, з відкритою відповіддю тощо), що значно полегшує створення збалансованих тестів із випадково згенерованими питаннями [41].

Потреба в більш гнучких рішеннях для онлайн-навчання призвела до розробки складних систем управління навчанням (LMS), таких як Moodle, Blackboard, Google Classroom, Cornerstone, OpenEDX та інші [42]. Ці платформи підтримують різноманітні інструменти для створення та управління тестами, включаючи базові функції для створення питань, встановлення правил та відстеження результатів. Більш просунуті системи використовують алгоритми для створення адаптивних тестів на основі попередньо визначених правил, де питання вибираються на основі попередніх відповідей користувача. Однак ці системи вимагають складних алгоритмів та значного ручного налаштування.

Деякі автоматизовані системи генерації тестів дотримуються методології, подібної до ручного створення тестів. Для створення тестового питання спочатку вибирається речення, а потім ідентифікується ключове слово або фраза. Потім речення перетворюється на формат питання на основі вибраного ключа, і визначаються можливі відповіді. Формалізацію цих кроків можна приблизно розділити на шість етапів: (1) Попередня обробка, (2) Вибір речення, (3) Вибір ключа, (4) Генерація питання, (5) Генерація відволікаючого
фактора та (6) Постобробка.

У багатьох випадках застосовується попередня обробка тексту, з якого генеруються питання. Для цього використовуються різні методи, такі як нормалізація тексту, структурний аналіз, спрощення речень, лексичний аналіз, статистичний аналіз, синтаксичний аналіз та інші. Вибір речень для побудови

тестових питань також є предметом численних досліджень. Використовуються різні критерії, такі як довжина речення, вживання певного слова, інформація про частини мови, інформація про синтаксичний аналіз, семантична інформація та інші. Для побудови певних типів тестових питань ключові слова вибираються на основі частоти, інформації про частини мови, інформації про синтаксичний аналіз, семантичної інформації та інших. Старіші програми покладаються переважно на базову статистичну та синтаксичну інформацію, тоді як новіші все частіше використовують семантичну інформацію.

Генерацію тестових питань можна виконувати за допомогою різних алгоритмів. У найпростішому випадку одне або кілька слів з вихідного речення видаляються, створюючи таким чином питання типу «заповнення пропусків». Інші підходи включають формування питань шляхом вибору відповідного слова з Wh, аналіз зв'язків між підметом, дієсловом та додатком, шаблони на основі залежностей, синтаксичну трансформацію та інші. У науковій літературі описано різні методи генерації дистракторів для тестових питань. До них належать методи, засновані на інформації про частини мови, частоті слів, онтології предметної області, семантичному аналізі та інші. Детальні огляди технологій генерації тестів.

Класичні підходи до створення тестів мають деякі суттєві обмеження:

- Традиційні системи дуже залежать від попередньо визначених правил і шаблонів, що обмежує можливості генерування різноманітних питань, особливо коли потрібні складніші або адаптивніші формати тестів.
- Ці системи вимагають значного ручного налаштування, як для створення правил, так і для вибору відповідних ключів та відволікаючих факторів, що робить їх трудомісткими та трудомісткими.
- Адаптація обмежена, зазвичай ґрунтується на попередніх відповідях і нелегко пристосовується до динамічно мінливих умов.
- Додатки використовують обмежену обробку семантичної інформації, що зменшує їхню здатність розуміти та інтерпретувати глибші контексти та взаємозв'язки в тексті.

Сучасні технології, засновані на штучному інтелекті та моделях великих

мов програмування (LLM), надають багатий набір інструментів, що дозволяють легко подолати вищезгадані обмеження. Завдяки цим технологіям процес автоматизованого створення персоналізованих тестів може бути повністю інноваційним, підвищуючи не лише ефективність, але й якість самих тестів.

Штучний інтелект та LLM можуть аналізувати та розуміти тексти на глибшому рівні, що дозволяє генерувати запитання з високим ступенем точності та релевантності. Вони можуть створювати різноманітні запитання, адаптовані до індивідуальних потреб та рівня знань учнів, використовуючи контекстуальну інформацію та семантичні зв'язки в тексті.

Цікаві можливості в цьому напрямку пропонують чат-боти на базі штучного інтелекту, такі як ChatGPT, Gemini, Claude та інші. Їх можна інтегрувати в освітні платформи та проводити динамічні тести. Ці чат-боти не лише генерують запитання, але й пропонують відповіді, що відволікають увагу, але є достатньо складними та релевантними, що підвищує рівень складності та запобігає легкому вгадуванню.

1.3 Великі мовні моделі та їх застосування в тестуванні ПЗ

Протягом останніх місяців ми провели численні експерименти, за допомогою яких тестували різні варіанти використання ChatGPT (версія gpt-3.5turbo) для освітніх цілей. Наш досвід був позитивним, і ми бачимо великий потенціал для подальшого розвитку.

У 2023 році ми протестували здатність ChatGPT створювати завдання, спрямовані на навчання та оцінку навичок мислення вищого порядку відповідно до таксономії Блума. Завдання та навчальні ресурси були успішно створені для курсу з нереляційних баз даних, зі специфічними вимогами до
навичок залучення, таких як аналіз, синтез та оцінка [36].

Представлено наш досвід використання інструментів штучного інтелекту у викладанні теми «Електричний струм» у фізиці учням 7-го класу. Особлива увага приділяється ChatGPT у підтримці учнів під час самостійного навчання та розробки курсових робіт.

В іншому експерименті з курсу «Бази даних» у Moodle було інтегровано три генеративні інструменти штучного інтелекту: «Генератор тексту на основі штучного інтелекту» для автоматичної генерації тестів, «Текст на основі зображення» для генерації зображень на основі заданого тексту та «Блок чату OpenAI» - віртуальний помічник на основі ChatGPT. Студентам було доручено послідовно виконати п’ять завдань: виконати одне завдання самостійно без використання штучного інтелекту; самостійно дослідити нову технологію, використовуючи інтегрований віртуальний чат-бот на основі штучного інтелекту; відповісти на короткий тест, згенерований штучним інтелектом, з п’ятьма питаннями, що оцінюють їхнє розуміння нової технології; продемонструвати свою здатність використовувати нову технологію, розширивши своє рішення до першого завдання. Фінальним завданням було написати короткий есе без використання технологій штучного інтелекту, в якому було б пояснено переваги та технології, пов’язані з використанням нової технології. Експеримент був успішним.

Наша подальша робота зосереджена на створенні навчальних ігор за допомогою ChatGPT. Описано процес розробки навчальних ігор за допомогою ChatGPT. Він є ітеративним і включає кілька кроків: початкове формулювання вимог, тестування, додавання нових функцій або елементів дизайну, вдосконалення та виправлення помилок. Запропоновано систематичний підхід до створення навчальних ігор за допомогою ChatGPT. Досліджено приклади різних типів ігор - картки, вікторини, головоломки, ігри на зіставлення та ігри на заповнення пропусків - з акцентом на певні специфічні особливості створення підказок та ітеративного процесу вдосконалення гри та виправлення помилок.

Усі наші експерименти однозначно довели, що ChatGPT можна ефективно використовувати в освіті. Саме тому ми зосередилися на розробці програмної системи для автоматизованого створення тестів. Наша ідея полягає в персоналізації всього процесу тестування, водночас надаючи вчителю можливість встановлювати різноманітні специфічні критерії для тестових питань, щоб підтримувати свої стратегії та підходи до навчання.

Розробляючи тести, викладачі зазвичай враховують багато факторів, таких як вивчаний матеріал, вікова група учнів, їхній рівень знань, культурні особливості та відмінності та багато інших. Цю складність можна перенести на

ChatGPT шляхом визначення численних специфічних вимог для створення тестів. До них можуть належати, наприклад:

- Мета тесту: конкретна мета тесту, яку слід враховувати під час створення питань (наприклад, оцінка розуміння, самопідготовка, підготовка до змагань);
- Охоплення теми: розподіл питань за різними підтемами (наприклад, п'ять питань для кожної з чотирьох підтем у межах однієї основної теми);
- Рівень складності: рівень складності питань (наприклад, легкий, середній, важкий);
- Тип питання: тип питань, що генеруються (наприклад, питання з вибором однієї правильної відповіді, питання з відкритою відповіддю, питання «правда/неправда», питання з пропуском);
- Формат питань: додаткові вимоги до формату питань (наприклад, використання діаграм або схем, включення тематичних досліджень або сценаріїв);
- Стиль питання: тип висловлювання (наприклад, академічний, розмовний, технічний);
- Кількість відповідей: скільки відповідей мають містити питання з вибором однієї правильної відповіді (наприклад, чотири можливі відповіді, одна правильна).
- Зворотній зв'язок: включення пояснень або коментарів до правильних та неправильних відповідей (наприклад, коротке пояснення кожної правильної відповіді);
- Конкретні рекомендації: додаткові інструкції або критерії для створення питань (наприклад, уникнення питань з оманливими відповідями, включаючи питання, що вимагають критичного мислення);
- Використання джерел: включення питань, що ґрунтуються на конкретних джерелах або текстах (наприклад, питання, що базуються на

конкретній статті, підручнику чи дослідженні);

- Контекст і приклади: включення запитань з контекстом або прикладами (наприклад, запитань, що використовують реальні життєві ситуації або тематичні дослідження);
- Формат тесту: вихідний формат, у якому ChatGPT має надавати запитання та відповіді (наприклад, таблиця, текстовий файл, файл Excel);
- Цільова аудиторія: вікова група або рівень освіти учнів (наприклад, учні старших класів, студенти університетів, фахівці) тощо.

Виконуючи цей запит, чат-бот створить тест і відобразить його в робочій області. Викладач може скопіювати та використовувати його належним чином. Незручність у цьому випадку полягає в тому, що викладачеві буде важко керувати тестами, створеними таким чином. Його роботу можна полегшити, розробивши спеціалізований застосунок для генерації тестів на основі ChatGPT. Потенційні переваги:

- Повторне використання: Запитання та тести можна зберігати в базі даних та використовувати повторно багато разів. Це усуває необхідність для викладача зберігати тести у файлах, що ускладнює пошук та повторне використання тестів.
- Персоналізація: Додаток може включати певні критерії для створення тестів, адаптованих до певної предметної області, викладача чи навчального закладу. Це дозволяє створювати тести, адаптовані до конкретних потреб та стилів викладання.
- Гнучкість та адаптивність: Розробка власного застосунку може забезпечити більшу гнучкість у дизайні та функціях, що дозволяє адаптуватися до різних форматів тестів та типів питань.
- Захист даних: Використання власного програмного продукту дозволяє контролювати безпеку та конфіденційність даних учнів. Це допомагає уникнути ризиків, пов’язаних із використанням конфіденційних даних третіми сторонами.
- Специфічні освітні методики: Викладач може використовувати унікальні методи навчання, які не підтримуються існуючими програмами, а

також можна створити користувацьку програму. Розроблені таким чином, щоб підтримувати їх використання .

Інтеграція з іншими системами: Користувацьку програму можна легше інтегрувати з іншими внутрішніми системами, які навчальний заклад або вчитель вже використовує

1.4 Висновки по розділу

У першому розділі було розглянуто теоретичні основи автоматизації тестування програмного забезпечення та використання великих мовних моделей. Проаналізовано базові принципи тестування ПЗ, його роль у забезпеченні якості та надійності програмних систем. Досліджено еволюцію інструментів автоматизації тестування, що відображає перехід від ручних підходів до інтелектуальних рішень. Також розглянуто великі мовні моделі та їх потенціал у задачах тестування, зокрема генерації тестів та аналізі вимог. Отримані результати формують теоретичну основу для подальшого дослідження методів застосування LLM у тестуванні.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ З ВИКОРИСТАННЯМ LLM

2.1 Методологія застосування великих мовних моделей у тестуванні

Для проведення та звіту про цей огляд було дотримано рекомендацій щодо систематичних оглядів літератури. Ця структура забезпечує комплексний та структурований підхід до виявлення та аналізу досліджень щодо генерації тестових випадків за допомогою LLM. Застосовуючи цю методологію, було досліджено різноманітні стратегії, інструменти, методології та тенденції. Крім того, цей підхід сприяє виявленню прогалин у дослідженнях та можливостей для подальшого дослідження.

У цьому розділі представлені результати огляду, структуровані навколо трьох дослідницьких питань, які були основою цього дослідження.

- RQ-1: Які методи були запропоновані для використання LLM в автоматизованому тестуванні

генерація справ?

- RQ-2: Наскільки ефективні LLM у покращенні якості та ефективності тестових випадків

покоління порівняно з традиційними методами?

- RQ-3: Які майбутні напрямки використання LLM в автоматизованому тестуванні були визначені

генерація справ?

Цей огляд зосереджений переважно на дослідженнях, які використовують LLM для генерації тестових випадків. Щоб визначити відповідні дослідження на перетині цих областей, були проведені експерименти з використанням різних рядків пошуку. Одне з основних джерел, IEEE Xplore, надає функцію «пошуку команд», яка дозволяє поєднувати терміни з логічними операторами, такими як AND та OR. Використовуючи цю функцію, було створено наступний рядок пошуку: ((«генерація тестів» OR «автоматизована генерація модульних тестів» OR «автоматизована генерація тестових випадків» OR «тестування програмного забезпечення») AND («LLM» OR «моделі великих мов» OR «GPT» OR «ChatGPT» OR «T5»)).

Розробка цього рядка пошуку була зумовлена експериментальними пошуками, проведеними в IEEE Xplore. Ці експериментальні пошуки включали тестування низки поширених термінів, акронімів та специфічних ключових слів,

пов'язаних з LLM та генерацією тестових випадків. Цей ітеративний процес гарантував, що рядок пошуку всебічно охоплює відповідну літературу, не пропускаючи ключових досліджень, які могли використовувати альтернативну термінологію. Це уточнення дозволило систематично ідентифікувати та переглянути дослідження, що стосуються генерації тестових випадків за допомогою LLM, забезпечуючи міцну основу для огляду.

Конкретні назви LLM, такі як GPT, ChatGPT та T5, були включені,оскільки експерименти показали, що деякі статті прямо посилаються на ці терміни у своїх назвах. Без цих критеріїв включення ключові дослідження могли бути пропущені. Лапки навмисно використовувалися, щоб гарантувати, що пошук повертатиме точні збіги для таких фраз, як «тестування програмного забезпечення», тим самим уникаючи нерелевантних результатів, які просто містять слова «програмне забезпечення» та «тестування» окремо.

Окрім пошуку в структурованій базі даних, було переглянуто 20 статей, у категорії «Генерація одиничних тестів», і 14 було відібрано для включення до основних досліджень через їхню актуальність. Кілька з них були взяті з Arxiv та перетиналися з результатами, отриманими з IEEE Xplore. Дослідження, визначені за допомогою основного рядка пошуку, були виключені з цього додаткового набору, щоб уникнути дублювання.

Потім Litmaps було використано для розширення набору релевантних досліджень. Початкові 26 досліджень, визначених за допомогою IEEE Xplore та ручного скринінгу, було імпортовано до Litmaps. Функція «Дослідити пов’язані статті» була використана для пошуку додаткових статей на основі зв’язків цитування та тематичної схожості. Для подальшого уточнення пошуку певні статті були позначені як «Схожі статті», що спонукало Litmaps надавати пріоритет подібним дослідженням. Кожна нещодавно знайдена стаття оцінювалася за тими ж критеріями включення та виключення, що й ті, що застосовувалися до початкового набору. Цей процес повторювався ітеративно, доки не було виявлено жодних додаткових релевантних досліджень, що призвело до набору з 76 первинних досліджень. Litmaps використовувався для

доповнення, а не заміни, структурованої стратегії пошуку для забезпечення прозорості та методологічної узгодженості.

Щоб забезпечити додаткове охоплення нещодавніх робіт, було проведено цільовий додатковий пошук у цифровій бібліотеці ACM та Scopus, обмежений статтями, опублікованими у 2025 році. Запит, який використовувався для цифрової бібліотеки ACM («llm» ТА «тестовий випадок»), повернув 261 запис, а запит для Scopus - «генерація тестового випадка llm», повернув 77 записів. Назви та анотації були перевірені на релевантність, а потім, за необхідності, проведено повнотекстовий огляд. Після застосування тих самих критеріїв включення та виключення, що й в оригінальному робочому процесі, та видалення дублікатів було визначено 8 унікальних досліджень та додано до основного набору. Це довело кінцеву кількість включених досліджень до 84.

Критерії відбору в цьому огляді були встановлені для вибору досліджень, що відповідали цілям дослідження, включаючи як критерії включення, так і критерії виключення.

- Тип публікації: Прийняті публікації включали доповіді конференцій, статті з журналів та доповіді препринтів.
- Мова: Розглядалися лише дослідження, написані англійською мовою.
- Актуальність: Дослідження повинні бути чітко зосереджені на застосуванні LLM у генерації тестових випадків.
- Часові рамки: Огляд обмежувався дослідженнями, опублікованими між 2022 і 2025 роками. Одне дослідження 2021 року [28] також було включено через його особливу актуальність для теми.
- Нерелевантність: Дослідження, які не були зосереджені переважно на генерації тестових випадків за допомогою LLM, були виключені.

Після виконання визначеного рядка пошуку було застосовано фільтрацію даних для уточнення результатів пошуку. Для включення розглядалися лише дослідження, опубліковані між 2022 та 2025 роками. Цей часовий проміжок був обраний тому, що, хоча LLM вивчаються вже деякий час, їхня популярність різко зросла після випуску ChatGPT у листопаді 2022 року. Ця тенденція

відображається в літературі, оскільки більшість відповідних статей були опубліковані в цей період. Щоб забезпечити охоплення оглядом останніх розробок, було включено дослідження, опубліковані між січнем 2022 року та липнем 2025 року. Також було включено попереднє дослідження через його значну актуальність для теми, незважаючи на те, що воно виходить за межі заданих часових рамок.

Застосування фільтра за роком зменшило кількість результатів до 38. Цей набір було додатково звужено шляхом перегляду назв, анотацій та ключових розділів кожного дослідження для оцінки їхньої релевантності. За допомогою цього процесу було визначено та відібрано 12 досліджень, які безпосередньо відповідали критеріям включення, як первинні.

Потім було включено 14 додаткових досліджень, які класифікували 20 досліджень за темою «Генерація одиничних тестів». Дослідження, вже знайдені в пошуку IEEE Xplore, були виключені, щоб уникнути дублювання.

Для розширення початкового набору з 26 досліджень було використано Litmaps. Дослідження було імпортовано на платформу, а її функції «Переглянути пов’язані статті» та «Ще подібне» були застосовані для визначення додаткових релевантних статей на основі цитування та тематичної схожості. Кожне рекомендоване дослідження було оцінено за заздалегідь визначеними критеріями включення та виключення. Цей ітеративний процес тривав доти, доки не було визначено жодних подальших відповідних досліджень. В результаті загальна кількість включених первинних досліджень зросла до 76.

Щоб доповнити цей набір найновішими роботами, було проведено цільовий додатковий пошук у цифровій бібліотеці ACM та базах даних Scopus, зосереджуючись виключно на дослідженнях, опублікованих у 2025 році. Для ACM використовувався запит ("llm" ТА "test case"), який повернув 261 результат. Для Scopus використовувався запит "llm test case generation", який повернув 77 результатів. Після застосування тих самих критеріїв включення та виключення, а також видалення дублікатів, було відібрано та включено до остаточного набору 8 додаткових досліджень.

Кінцевий результат складався з 84 первинних досліджень. На рисунку 2.1 наведено огляд процесу пошуку та включення літератури.

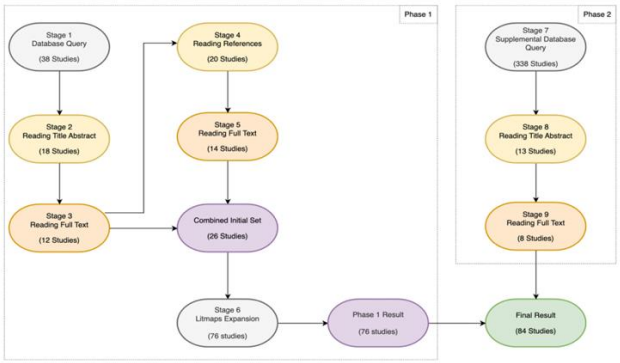

Рисунок 2.1 - Процедура збору даних.

Пошук у кількох базах даних призвів до виявлення 84 первинних досліджень, що стосуються генерації тестових випадків на основі LLM. Ці дослідження розглядали різні аспекти використання LLM, включаючи швидку інженерію, ітеративне уточнення, точне налаштування та гібридні підходи. Для кращої ясності вибрані дослідження були класифіковані на основі їх методологічної спрямованості.

На рисунку 2.2 показано розподіл вибраних досліджень з 2021 по 2025 рік, що відображає помітне зростання дослідницької активності після випуску ChatGPT. Ця візуалізація підкреслює зростаючий інтерес до генерації тестів на основі LLM та розкриває ключові тенденції досліджень у цій галузі.

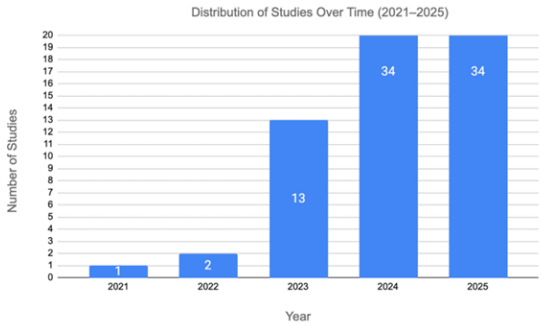

Рисунок 2.2 - Розподіл вибраних досліджень за роками.

Вибрані дослідження поділяються на чотири основні групи: швидке проектування, підходи на основі зворотного зв'язку, точне налаштування та попереднє навчання моделі, а також гібридні підходи. Ця класифікація забезпечує структурований погляд на дослідницький ландшафт і дозволяє проводити більш цілеспрямований аналіз з використанням різних методологічних підходів.

- Інженерія підказок: Дослідження в цій категорії досліджують методи побудови та вдосконалення підказок LLM для покращення результатів генерації тестів.
- Підходи, що керуються зворотним зв'язком: ця категорія включає дослідження, що включають ітеративні взаємодії та механізми для подальшого вдосконалення та підвищення релевантності й якості згенерованих тестових випадків.
- Точне налаштування та попереднє навчання моделі: Дослідження в цій групі досліджували підходи до точного налаштування та попереднього навчання, спрямовані на оптимізацію продуктивності LLM для завдань генерації тестів.
- Гібридні підходи: Дослідження в цій категорії вивчали, як генерацію тестів на основі LLM можна поєднувати з традиційними інструментами та методологіями тестування.

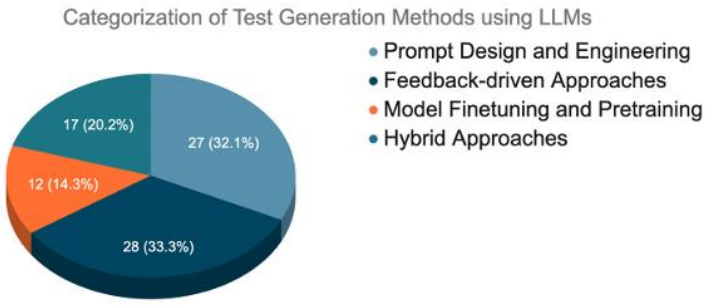

Рисунок 2.3 - Класифікація методів генерації тестових випадків на основі великих мовних моделей, виявлених у первинних дослідженнях

На рисунку 2.4 показано найчастіше використовувані набори даних у розглянутих дослідженнях. Варіанти наборів даних (наприклад, розширені або мовно-специфічні версії HumanEval) були згруповані під єдиною назвою для узгодженості. Власні та спеціально створені набори даних, а також ті, що з'являлися менш ніж у чотирьох дослідженнях, були виключені, щоб підкреслити ширше прийняті контрольні показники. Хоча діаграма не є вичерпною, вона виділяє набори даних, які стали стандартом для навчання та оцінки підходів до генерації тестів на основі LLM.

Фігура На рисунку 2.5 представлено стислий огляд найчастіше використовуваних моделей у п'яти основних сімействах LLM, визначених у первинних дослідженнях: GPT (OpenAI), LLaMA, DeepSeek, CodeGen та Codex. Щоб зменшити візуальну складність та підкреслити ключові тенденції, на діаграмі відображено три найпопулярніші моделі з кожного сімейства.

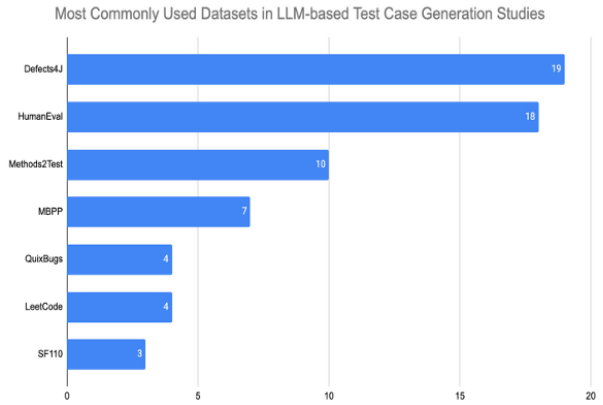

Рисунок 2.4 - Частота використання наборів даних у дослідженнях з генерації тестових випадків на основі великих мовних моделей

Сімейство GPT від OpenAI явно домінує в розподілі, причому gpt-3.5-turbo є найчастіше використовуваною окремою моделлю, за нею йдуть GPT-4 та GPT-4o. Сімейство LLaMA посідає друге місце за частотою, з помітним використанням як моделей загального призначення LLaMA, так і спеціалізованих на коді варіантів, таких як CodeLLaMA. DeepSeek слідує за ним, включивши кілька своїх моделей Coder, тоді як CodeGen та Codex мають нижчий, але порівнянні рівні використання. Серед варіантів Codex до діаграми було включено лише code-davinci-002, оскільки його частота перевищила частоту

інших варіантів, включаючи code-cushman-001, code-cushman-002 та code-davinci-001. Хоча в дослідженнях також використовувалися інші моделі, такі як Gemini, Claude, BART та T5, вони застосовувалися рідше.

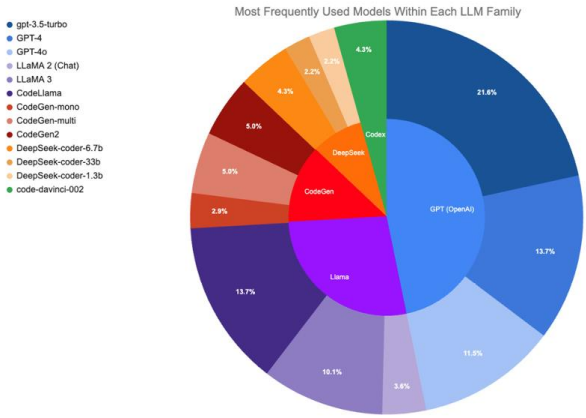

Рисунок 2.5 - Найпопулярніші моделі в основних сімействах великих мовних моделей

Важливо зазначити, що частота використання моделі не означає кращу продуктивність. Наприклад, gpt-3.5-turbo зустрічається частіше, ніж GPT-4o, але це може відображати її ранню доступність та ширший доступ до API на момент проведення багатьох досліджень. Оскільки новіші та більш потужні моделі стають більш доступними, можна очікувати зміни розподілу частот у майбутніх дослідженнях.

2.2 Методи генерації тестових сценаріїв та тестових даних

Це дослідження було зосереджено на використанні розширених можливостей LLM, які добре справляються з генерацією та обробкою коду, особливо коли код добре структурований і не надто складний. Хоча ці моделі демонструють високу ефективність у виконанні більш доступних завдань, вони, як правило, стикаються з труднощами зі складнішим або багатошаровим кодом [12] . Зосереджуючись на кодах, які відповідають сильним сторонам наявних LLM, цей підхід максимізує їхній потенціал у генерації тестового коду.

Щоб розпочати дослідження можливостей LLM у написанні тестів коду, необхідно було спочатку створити колекцію валідацій, що містить фрагменти коду з відповідними тестами. Для досягнення цієї мети було проведено пошук на сайтах репозиторіїв (здебільшого Github) проектів, написаних за методологією Test-Driven-Development (TDD) [13] на Python. На жаль, на наш жаль, багато проектів, навіть дуже відомих, мали неадекватні тести; отже, виникла необхідність створити набір даних з нуля. Вихідний та тестовий коди з репозиторіїв спочатку були відібрані на основі зручності використання (були відібрані відносно простіші коди, а коди, які потребували додаткових файлів, таких як бази даних або зображення, не були включені). Далі, відібрані коди були якісно проаналізовані членами нашої команди на функціональність та якість, і, нарешті, скориговані або переписані за необхідності, щоб забезпечити їх високу якість. Наприклад, якщо тест був написаний з використанням методу операторів assert і не містив додаткової функції, яка б забезпечила виконання тесту, її довелося вручну додати до сценарію тестового коду.

Існує підозра, що вихідний та тестовий коди, знайдені на сайтах репозиторіїв, були не найкращої якості через недосвідченість авторів у програмуванні (більшість знайдених проектів були створені для навчальних, а не комерційних чи дослідницьких цілей), а інші проекти не були завершені їхніми авторами. Це не відображає справжніх людських здібностей у написанні тестових кодів, оскільки досвідчені програмісти розробляють тестові коди, які є одночасно виконуваними та високоякісними; з цієї причини ми вирішили вручну налаштувати вихідний та тестовий коди, що призвело до нашого остаточного набору даних із 50 пар вихідного та тестового кодів. Схема збору даних у базі даних показана на рисунку 2.6.

Щоб максимально використати потенціал LLM у генерації тестового коду, це дослідження спочатку зосереджувалося на добре структурованих та простих вихідних кодах. Однак простіші коди не завжди відображають складність коду, з якою програмісти стикаються в реальних сценаріях. Для точніше розуміння продуктивності LLM у простіших та складніших кодових

базах вихідні коди були класифіковані на основі складності генерації модульних тестів.

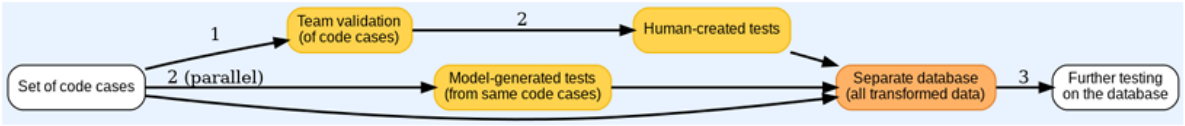

Рисунок 2.6 - Процес створення бази даних.

Після ретельного пошуку в різних репозиторіях вдалося виділити лише 46 вихідних випадків коду. Після перегляду було визначено 2 випадки, для яких написання тестів було неможливим, а решту 44 випадків було проаналізовано на предмет складності. Усі вони виявилися досить простими, і для забезпечення більш повного набору даних наша команда вирішила зібрати складніший код.

Не знайшовши складніших кодів у репозиторіях, наша дослідницька група, що складалася з фахівців з обробки даних, а не з професійних програмістів чи тестувальників, використала GPT4 у поєднанні з людським аналітичним досвідом для ручного отримання складніших кодів, зокрема 5 складних кодів та 1 середнього коду.

Вихідні коди потім були розділені на чотири категорії залежно від складності написання відповідних модульних тестів:

- Складно: Ці коди вимагають глибокого розуміння складних математичних концепцій, таких як здатність проектувати та тестувати фрактальну розмірність, а також розширених навичок реалізації з використанням бібліотек Python, адаптованих для складних обчислень та візуалізацій.
- Середній: Ці коди вимагають знань, специфічних для предметної області, для створення відповідних тестових випадків. Наприклад, тестування логіки шахової гри вимагає розуміння правил гри. Крім того, ці коди часто містять імітацію певних частин логіки, таких як операції вводу/виводу, щоб забезпечити ізольоване виконання модульних тестів відповідно до найкращих практик [14] .

• Легко та дуже легко: решта кодів були класифіковані на основі використання в них концепцій програмування, таких як абстракція, успадкування або шаблони проектування.

Після цієї категоризації остаточний набір даних містив 20 дуже простих, 16 легких, 9 середніх та 5 складних кодів. Цей ширший діапазон складності забезпечив краще розуміння того, як LLM працюють на різних рівнях складності генерації тестів.

Набір даних містить пари вихідних кодів та тестів, написаних людиною. Коди є основою для генерації тестів - вихідні коди - це переважно коди простих функцій з простою логікою, наприклад, код рядів Фібоначчі. Однак ми також включили деякий складніший код, написаний в об'єктно-орієнтованому програмуванні (ООП), такий як шахова логіка або коди клієнт-серверної архітектури. Кожен вихідний код був вручну протестований та виправлений за потреби, щоб забезпечити максимально можливу ймовірність для LLM генерувати виконувані тести на основі вихідного коду.

Вихідні коди структуровані в такому форматі:

1. Імпорт бібліотек, що використовуються в програмі.

2. Функції або класи: основна логіка програми. Кожна з них визначена з чіткою метою виконання певного завдання.

Набір даних містить створені людиною тестові випадки, написані двома різними методами тестування, щоб відобразити різноманітність реального світу розробки програмного забезпечення. Два основні підходи – pytest та unitest. Тестові коди структуровані в такому форматі:

1. Імпорт
- Бібліотеки (такі як unittest або pytest), необхідні для тестування;
- Бібліотеки, що використовуються в тесті;
- Вихідний код.
2. Тестові випадки:
- У випадку використання бібліотеки unittest:
- Стиль: Формальний та об'єктно-орієнтований.
- Структура: Тести організовані в класи, що успадковуються від

unittest.TestCase. Кожен метод у класі відповідає певному випадку використання.

- Твердження: Використання методів тверджень, таких як self.assertEqual, self.assertTrue тощо.
- Переваги: Добре організовані та згруповані тести, що підходять для складних наборів тестів, із вбудованою підтримкою виявлення тестів та звітності.
- Недоліки: Більш багатослівний та вимагає додаткової шаблонної документації.

• У випадку використання бібліотеки pytest або простих операторів assert:

- Стиль: Лаконічний та декларативний.
- Структура: Тести пишуться як окремі функції, кожна з яких починається з префікса test_.
- Твердження: Використання вбудованого оператора assert в Python для перевірки очікуваних результатів.
- Переваги: Мінімалістичний шаблон, легкий для читання та написання.
- Недоліки: Менш структурований та не має формальності, необхідної для масштабних проектів.

В останні роки ми спостерігаємо динамічне зростання в розробці моделей великих мов [1] . Такий швидкий випуск нових моделей призвів до їх величезної різноманітності. Наша дослідницька група мала можливість вибирати між моделями з відкритим або закритим кодом, базовими або налаштованими для дослідницьких завдань. Через обмеження в часі та відносно невелику кількість членів команди ми вирішили використовувати моделі, розміщені у зовнішнього постачальника з доступом до API.

Це рішення звузило кількість потенційних моделей, виключивши практично всі моделі з відкритим кодом, налаштовані на генерацію коду, такі як CodeLLlama або StarCoder. Наша команда обрала сервіс Amazon Bedrock серед доступних хмарних сервісів, оскільки він пропонує значну кількість сучасних базових моделей. З доступних LLM ми зрештою використали такі:

- Llama3 70B: За словами авторів [15] , цей LLM досягає найсучасніших результатів у поширених тестах, включаючи генерацію коду. Він має перевагу в тому, що є відкритим кодом, що забезпечує прозорість з точки зору того, як він працює, і його потенційно можна розміщувати локально.
- Mistral Large: Моделі Mistral, подібно до моделей Llama, мають відкритий вихідний код, а їхня ліцензія дозволяє комерційне використання. Автори стверджують, що ці моделі, включаючи найменшу доступну модель, продуктивність якої порівнянна з продуктивністю більших моделей [16] , можуть досягати найсучасніших результатів. На практиці найменша модель має лише 7 мільярдів параметрів, що може бути більш економічно ефективним у великомасштабних проектах. Згідно з зовнішніми дослідженнями [17] , Mistral 7B також майже на дві третини дешевший, ніж конкурентна Llama 8B у випадку використання Bedrock, що забезпечує кращу доступність.
- Клод 3 Сонет: За словами авторів [18] , ця LLM здатна досягти найсучасніших результатів; у контрольних характеристиках авторів зазначено, що вона перевершує одну з найвідоміших LLM - GPT4.

Наша команда вирішила використовувати лише найбільші доступні версії моделей для встановлення базової лінії. Обґрунтування цього полягає в тому, що якщо найдосконаліші та найскладніші моделі демонструють обмеження у створенні автоматичних модульних тестів, ймовірно, що менші та менш складні моделі зіткнуться з аналогічними або більшими проблемами продуктивності. Більше того, основною метою цього дослідження є оцінка можливостей автоматизації тестування, а не пошук найефективнішої моделі; однак з практичної точки зору варто розглядати виключно моделі з сімейств моделей, які потенційно можуть бути використані у виробничому середовищі. Нарешті, через обмеження в часі та обмежені ресурси було вирішено відкласти тестування GitHub Copilot, одного з найпопулярніших інструментів на базі GPT4 для створення модульних тестів.

Обмеження в часі також обмежувало нашу команду використанням моделей у сценаріях з нульовим результатом, тобто без будь-якого точного налаштування підготовленого набору даних, тому моделі, що використовуються

для дослідження, використовують лише загальні знання. Тим не менш, можна було вплинути на їхні кінцеві результати, підготувавши відповідні інструкції для моделей (підказки).

У цьому розділі детально описано методологію, яка використовується для оцінки продуктивності моделей GenAI під час генерації тестового коду разом із тестовим кодом, написаним людиною, з використанням двох метрик: покриття коду та кількість збоїв виконання.

- Покриття коду - це кількісний показник того, яка частина вихідного коду була виконана під час запуску тестового коду. Він дає уявлення про ефективність тестового коду у виконанні різних частин коду. Це широко використовувана метрика в дослідженнях програмної інженерії [19] , а також у дослідженнях щодо ефективності LLM у генерації модульних тестів [7] . У цьому експерименті тестові коди тестових кодів, згенерованих GenAI та написаних людиною, були інструменталізовані за допомогою Coverage.py у поєднанні з pytest на Python. Згенеровані та написані тести були перевірені на той самий вихідний код, і інструмент покриття генерував звіти з детальним описом відсотка охоплених рядків вихідного коду. Потім відсотки покриття були додатково проаналізовані за допомогою статистики (середнє значення, медіана та стандартне відхилення) для кожного методу запиту, а також кожної моделі GenAI/автора-людини та складності.
- Збій виконання - окрім покриття коду, підраховувалася кількість збоїв. Ця метрика відображає кількість тестових випадків, які не вдалися виконати через помилки в самому тестовому коді. Вона використовувалася в дослідженнях для оцінки продуктивності LLM [20] . Збій виконання вимірювався за допомогою того ж процесу, що й міра покриття, описана вище. Якщо міра покриття не повертала результат для жодного тестового випадку або якщо покриття коду дорівнювало 0, це було пов'язано з помилками в тестовому коді, і тестовий випадок для конкретної моделі позначався як такий, що показав збій виконання.

Щоб виміряти подібність між тестовими кодами моделей та тестовими кодами авторів-людей, коди спочатку були перетворені за допомогою моделі

Claude 3, якій було доручено замінити кожен тестовий код рядок за рядком на звичайний текст, дотримуючись інструкцій, які вказували на необхідність перейменування певних елементів коду за допомогою певних семантичних категорій слів, як показано на рисунку. 2.7. Наприклад, оператори імпорту мали бути перейменовані на фрукти, змінні – на предмети одягу тощо. Метою цього перетворення було допомогти в узагальненні тестових кодів, щоб їх було легше порівнювати, зберігаючи при цьому важливу інформацію щодо структури коду . Ці перетворені коди потім були перетворені на вбудовування за допомогою Amazon Titan Embeddings. Перетворення мовного тексту в числові представлення, відоме як вбудовування, – це практика, яка широко використовується для завдань обробки мови та часто застосовується до фрагментів коду [21] . Нарешті, вбудовування можна було порівняти за допомогою трьох мір: евклідової відстані, косинусної подібності та скалярного добутку.

- Евклідова відстань - вимірює різницю по прямій між двома точками у багатовимірному просторі вбудовувань. Вона фіксує, наскільки далеко вбудовування знаходяться одне від одного - чим далі вони, тим більше вони відрізняються. Евклідова відстань використовується як базова метрика для вимірювання подібності вбудовування тексту, серед іншого [22] .
- Косинусна подібність - для підтвердження міри подібності також було розраховано косинусну подібність між вбудовуваннями. Вона вимірює косинус кута між двома векторами вбудовування та є однією з найчастіше використовуваних метрик для порівняння подібності тексту на основі семантичних вбудовування [23]. Вона дає уявлення про те, наскільки схожі напрямки вбудовування. Косинусна подібність коливається від -1 до 1, де 1 означає однакові напрямки, а -1 - протилежні. Таким чином, вищі значення косинусної подібності свідчать про те, що порівнювані вбудовування більш схожі.
- Скальковий добуток - окрім косинусної подібності, використовується подібна міра - крапковий добуток

добуток. Це вимірює величину перекриття вбудовування. Це схоже на косинусну подібність, але також враховує довжину векторів [24]. Більший скалярний добуток свідчить про те, що два порівнювані вбудовування більш схожі.

Рисунок 2.7 - Метод отримання значень подібності тестів.

Для обраних LLM завдання генерації модульних тестів було описано чотирма різними типами інструкцій (підказок), кожен з яких відповідає різному підходу до підказок, що зазвичай зустрічається в літературі [25]. Перша спроба, показана в лістингу. 1 , є «базовим», оскільки він містить найпростішу обов'язкову частину: інструкції для генерації тестового коду для заданого вихідного коду.

Лістинг 2.1 - Базовий запит.

```
You are a programming assistant.  Your task is to provide a unit test code
snippet to the source code below.  Please import the source code as
'source_code' module in the code snippet.
{source code}
```

Базова підказка також використовує метод призначення ролей. Згідно з сучасними дослідженнями [26], опис ролі моделей у підказці допомагає досягти бажаного стилю та тону, або, що ще важливіше, спрямувати їх на виконання поставленого завдання. Команда вирішила призначити моделям роль помічника програмування, очікуючи, що вони будуть ефективнішими у генерації коду. У попередніх експериментах було помічено, що згенеровані тести мали проблеми з імпортом вихідного коду. Моделі не могли визначити належну назву вихідного

модуля: імпорт вихідного коду не був включений або було додано імпорт заповнювачів. Приклад такої несправності показано в лістингу 2.2.

Лістинг 2.2 - Приклад проблеми з імпортом вихідного коду.

```
from your_module import x, y, z # Replace with the actual module name
```

Щоб вирішити цю проблему, моделі були підкріплені прямими інструкціями щодо правильного імпорту вихідного коду («Будь ласка, імпортуйте вихідний код як модуль *source_code* у фрагменті коду»). Після додавання цього пункту було помічено, що всі моделі правильно включають імпорт вихідного коду. Недоліком цього виправлення є те, що вихідний код має бути розміщений у модулі з такою ж назвою.

Наступна спроба (Лістинг 2.3) було зосереджено на наданні моделі більш конкретних інструкцій. Окрім базового підходу, ми додали кілька порад щодо написання якісних модульних тестів на основі найкращих практик.

Лістинг 2.3 - Запит із розширеними інструкціями.

```
Act as a Python software test engineer.  Your task is to provide a unit
test code snippet to the source code below.  Please import the source code
as 'source_code' module in the code snippet.  Each test must have the
following parts:
(1) Arrange:  create and set necessary objects / data for the test
(2) Act:  call the tested method from source code and get the actual value
(3) Assert:  check the expected and actual values.
Don't forget to test edge cases and all possible exceptions which can be
raised.
<source_code >
{source_code}
</source_code >
```

Основними покращеннями було додавання інструкції щодо використання шаблону «впорядкувати, діяти, стверджувати» та нагадування про граничні випадки та тестування винятків. Роль моделі було змінено на інженера-тестувальника програмного забезпечення Python, що є більш доречним у ситуації, коли ми тестуємо лише код Python.

Наступний експеримент включав детальні інструкції та приклад модульного тесту. Цей підхід називається підказкою з кількома спробами [27]. Шаблон підказки показано в лістингу 2.4 ; код замінено позначкою прикладу. Приклад містить вихідний код із простою функцією, яка повертає результат ділення першого аргументу на другий. Потім розміщуються еталонні модульні тести, що містять «щасливий шлях» (правильні аргументи) та граничні випадки: другий аргумент дорівнює 0, а аргументи мають неправильний тип. Очікувалося, що приклад допоможе моделям покращити тестування граничних випадків, показавши, як вибрати проблемні.

Лістинг 2.4 - Запрошення з кількома пострілами.

```
Act as a Python software test engineer.  Your task is to provide a unit
test code snippet to the source code below.  You also get a example of a
unit test.  Please import the source code as 'source_code' module in the
code snippet.  Each test must have the following parts:
(1) Arrange:  create and set necessary objects / data for the test.
(2) Act:  call the tested method from source code and get the actual value.
(3) Assert:  check expected and actual values.  Don't forget to test edge
cases and all possible exceptions which can be raised.
<example >
{example}
</example >
<source_code >
{source_code}
</source_code >
```

Лістинг 2.5 показано останній протестований підхід до підказок, який називається ланцюжком думок [28]; фрагменти коду були замінені символами { } (повну підказку див. у репозиторії). Основною метою було допомогти зі складними міркуваннями, необхідними для моделі, шляхом підтримки проміжних кроків міркування. Інструкція починається з призначення ролі моделі та надання ідентичного прикладу вихідного коду, як і в експерименті з підказками кількома кроками. Потім модель навчають, як правильно включати вихідний код та писати модульний тест, враховуючи найкращі практики. Моделі також надаються поради щодо розпізнавання граничних випадків. Нарешті,

інструкції підкріплені прикладом модульного тесту, написаного відповідно до заданих рекомендацій. Як і в попередньому експерименті,

Метою було покращити якість згенерованого модульного тесту шляхом вибору правильних граничних випадків для тестування.

Лістинг 2.5 - Підказка «Ланцюг думок».

```
Act as a Python software test engineer.  Your task is to provide the unit
test code snippet to the source code below:
<source_code >
{example of the source code}
</source_code >
The source code is located in the separate file, so you must remember to
import the source code as 'source_code' module in the code snippet.
First you must check if the tested function works correctly in the
following way:
(1) Arrange:  create and set necessary objects / data for the test.
(2) Act:  call the tested method from source code and get the actual value.
(3) Assert:  check expected and actual values.
Then you must also check the edge cases and all possible exceptions which
can be raised.  Edge cases depend on the tested source code logic.  In this
case, it's wise to check the wrong type of the argument and if the written
exception is raised correctly.  Other cases worth checking are when the
input values are outside the range, a null value or an empty list/dict.
To sum up, the final unit test code snippet should look like this:
<example >
{example of the unit tests }
</example >
Your task is to provide a unit test code snippet to the source code below:
<source_code >
{source code}
</source_code >
```

2.3 Аналіз вимог користувачів та моделювання тестових сценаріїв

На рисунку 2.6 представлено високорівневу архітектуру TEST PILOT , яка складається з п'яти основних компонентів: маючи PUT на вхід, *API Explorer* визначає функції для тестування; *documentation miner* витягує метадані про них; а *генератор запитів, валідатор тестів* та *уточнювач запитів* співпрацюють для створення запитів для генерації тестів, збирають повні тести з відповіді LLM, запускають їх, щоб визначити, чи пройшли вони перевірку, та створюють подальші запити для генерації більшої кількості тестів. Тепер ми обговоримо кожен з цих компонентів більш детально.

API Explorer: Цей компонент аналізує PUT, щоб визначити його API,

тобто набір функцій, методів, констант тощо, які пакет надає клієнтам. У JavaScript дуже важко визначити API статично через дуже динамічну природу мови. Тому, подібно до інших робіт з генерації тестів JavaScript [10] , [11] , ми використовуємо підхід, заснований на динамічному аналізі. Зокрема, ми завантажуємо основний пакет програми та застосовуємо інтроспекцію для проходження результуючого графа об'єктів та визначення властивостей, пов'язаних з функціями. Для кожної функції ми записуємо її шлях доступу (тобто послідовність властивостей, які необхідно пройти, щоб дістатися до неї з основного модуля), її сигнатуру (яка за відсутності статичної інформації про тип є просто списком імен параметрів) та її визначення (тобто її вихідний код). Виходом API Explorer є список функцій, описаних їхніми шляхами доступу, сигнатурами та визначеннями; інші елементи API ігноруються.

Documentation Miner: Цей компонент витягує фрагменти коду та коментарі з документації, що додається до PUT, та пов'язує їх з функціями API, до яких вони належать. Мета полягає в тому, щоб зібрати для кожної функції API коментарі та приклади, що описують її призначення та передбачуване використання. У базах коду JavaScript документація зазвичай надається у вигляді файлів Markdown (.md), в яких фрагменти коду вбудовані як огороджені блоки коду (тобто блоки, оточені потрійними зворотними позначеннями). Ми знаходимо всі такі блоки у всіх файлах Markdown у базі коду та пов'язуємо з кожною функцією набір усіх фрагментів коду, які текстово містять назву функції. Хоча це проста евристика, приклади коду можуть бути неповними або синтаксично коректними, тому більш складний підхід, що спирається на парсинг або статичний аналіз, навряд чи працюватиме добре. Ми також пов'язуємо кожну функцію API з коментарем документа (/**. . . */), який безпосередньо передує їй, якщо такий є.

Решта три компоненти – це *генератор запитів*, *валідатор тестів* та *уточнювач запитів* , які працюють разом для створення та перевірки тестів для всіх функцій API, визначених API Explorer, використовуючи інформацію, надану Documentation Miner. Функції обробляються по одній, і для кожної функції генерується лише один тест за раз (на відміну від генерації цілого набору тестів

одночасно). Це дозволяє нам перевіряти кожен тест окремо без перешкод з боку інших тестів.

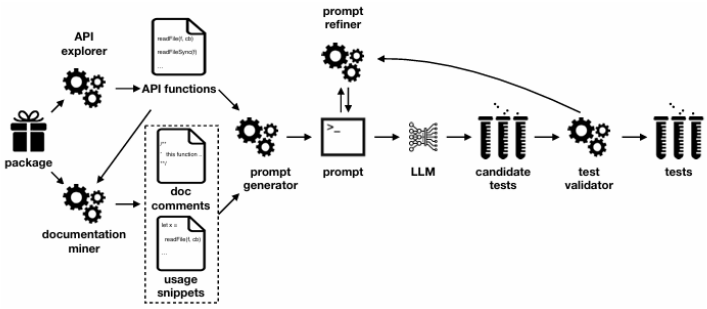

Рисунок 2.6 - Огляд методу генерації адаптивних тестів, який ми використовуємо в TESTPILOT

Генератор підказок: Цей компонент створює початковий підказку для надсилання до LLM для створення тесту для заданої функції f. Як згадувалося вище, спочатку ми маємо (щонайбільше) чотири частини інформації про f: її сигнатуру, її визначення, її коментар до документації та фрагменти її використання, отримані з документації. Хоча може здатися природним створити підказку, що містить всю цю інформацію, на практиці іноді може статися так, що складніші підказки призводять до гіршого завершення, оскільки LLM заплутується в додатковій інформації. Тому ми дотримуємося іншої стратегії: ми починаємо з дуже простої початкової підказки, яка не містить метаданих, окрім сигнатури функції, а потім дозволяємо уточнювачу підказок крок за кроком розширювати її додатковою інформацією.

Запит (а) не містить фрагментів, і згенерований на його основі тест не проходить. Запит (б) містить один фрагмент, і згенерований тест проходить.

Валідатор тестів: Далі ми надсилаємо згенеровані запити до LLM та чекаємо на завершення рисунок 2.7. Ми використовуємо лише стільки токенів, скільки потрібно для формування синтаксично коректного тесту .

```
let mocha = require('mocha');
let assert = require('assert');
let countries_and_timezones = require('countries-and-timezones');
// countries-and-timezones.getCountry(id)
describe('test countries_and_timezones', function() {
  it('test countries-and-timezones.getCountry', function(done) {
    let country = countries_and_timezones.getCountry('US');
    assert.equal(country.name, 'United States'); // fails
    assert.equal(country.timezones.length, 2);
    assert.equal(country.timezones[0], 'America/New_York');
    assert.equal(country.timezones[1], 'America/Chicago');
    done();
  })
})
```

(a)

```
let mocha = require('mocha');
let assert = require('assert');
let countries_and_timezones = require('countries-and-timezones');
// usage #1
// const ct = require('countries-and-timezones');
//
// const country = ct.getCountry('DE');
// console.log(country);
//
// /*
// Prints:
//
// {
//   id: 'DE',
//   name: 'Germany',
//   timezones: [ 'Europe/Berlin', 'Europe/Zurich' ]
// }
//
// */
// countries-and-timezones.getCountry(id)
describe('test countries_and_timezones', function() {
  it('test countries-and-timezones.getCountry', function(done) {
    let country = countries_and_timezones.getCountry('DE');
    assert.equal(country.id, 'DE');
    assert.equal(country.name, 'Germany');
    assert.equal(country.timezones[0], 'Europe/Berlin');
    assert.equal(country.timezones[1], 'Europe/Zurich');
    done();
  })
})
```

(b)

Рисунок 2.7 - Приклади запитів (виділені) та відповіді, надані великою мовною моделлю (LLM), що містять повні тести.

Оскільки немає гарантії, що запропоновані моделлю завершення є синтаксично коректними, валідатор тестів намагається виправити прості синтаксичні помилки, такі як відсутні дужки, а потім аналізує результуючий код, щоб перевірити його синтаксично коректність. Якщо ні, тест негайно позначається як невдалий. В іншому випадку він запускається за допомогою тестового виконавця Mocha, щоб визначити, чи він пройдений, чи ні (через помилку твердження або якусь іншу помилку під час виконання).

Кожне повернене завершення можна об'єднати з підказкою, щоб отримати тест-кандидат. Однак, щоб ми могли виключити дублікати тестів, згенерованих з різних підказок, ми обробляємо тести-кандидати наступним чином: ми видаляємо Генератор підказок: Цей компонент створює початковий підказку для надсилання до LLM для створення тесту для заданої функції f. Як згадувалося вище, спочатку ми маємо (щонайбільше) чотири частини інформації про f: її сигнатуру, її визначення, її коментар до документації та фрагменти її використання, отримані з документації. Хоча може здатися природним створити підказку, що містить всю цю інформацію, на практиці іноді може статися так, що складніші підказки призводять до гіршого завершення, оскільки LLM заплутується в додатковій інформації. Тому ми дотримуємося іншої стратегії: ми починаємо з дуже простої початкової підказки, яка не містить метаданих, окрім

сигнатури функції, а потім дозволяємо уточнювачу підказок крок за кроком розширювати її додатковою інформацією.

Валідатор тестів: Далі ми надсилаємо згенеровані запити до LLM та чекаємо на завершення. Ми використовуємо лише стільки токенів, скільки потрібно для формування синтаксично коректного тесту. Оскільки немає гарантії, що запропоновані моделлю завершення є синтаксично коректними, валідатор тестів намагається виправити прості синтаксичні помилки, такі як відсутні дужки, а потім аналізує результуючий код, щоб перевірити його синтаксично коректність. Якщо ні, тест негайно позначається як невдалий. В іншому випадку він запускається за допомогою тестового виконавця Mocha, щоб визначити, чи він пройдений, чи ні (через помилку твердження або якусь іншу помилку під час виконання).

Кожне повернене завершення можна об'єднати з підказкою, щоб отримати тест-кандидат. Однак, щоб ми могли виключити дублікати тестів, згенерованих з різних підказок, ми обробляємо тести-кандидати наступним чином: ми видаляємо

2.4 Висновки по розділу

У другому розділі було досліджено методи та інструменти автоматизації тестування з використанням великих мовних моделей. Розглянуто методологію застосування LLM у процесах тестування програмного забезпечення, що дозволяє підвищити рівень автоматизації та ефективності. Проаналізовано підходи до генерації тестових сценаріїв і тестових даних із використанням генеративного штучного інтелекту. Також досліджено можливості аналізу вимог користувачів і моделювання тестових сценаріїв на основі LLM. У результаті визначено ефективні підходи до інтеграції великих мовних моделей
у процеси тестування ПЗ.

РОЗДІЛ 3. РОЗРОБКА, ІНТЕГРАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РІШЕНЬ НА ОСНОВІ LLM

3.1 Розробка інструменту автоматизації тестування на основі штучного інтелекту: технології та підходи

Основними технологіями, що використовуються для розробки застосунку, є хмарна платформа Google Firebase та ChatGPT API для генерації тестових питань.

Google Firebase - це платформа для розробки мобільних та веб-додатків, що надається Google. Вона пропонує набір сервісів та інструментів, які допомагають розробникам створювати, тестувати, керувати та масштабувати свої додатки. До них належать система автентифікації користувачів, бази даних NoSQL, хостинг для веб-додатків, серверні сервіси для запуску бекенд-коду в хмарі, інструменти для аналізу поведінки користувачів, інструменти для моніторингу та оптимізації продуктивності додатків тощо.

Для управління даними в застосунку використовується NoSQL система керування базами даних Firebase Realtime Database. Окрім стандартних функцій, вона включає вбудовані механізми безпеки, шифрування даних та механізми автоматичного створення резервних копій, що забезпечує надійність та безпеку даних. Вона підтримує можливості масштабування ресурсів відповідно до потреб застосунку. Адаптація ресурсів на основі кількості активних користувачів, обсягу даних тощо забезпечує оптимальну ефективність та продуктивність. Firebase Database легко інтегрується з іншими сервісами, що дозволяє здійснювати подальшу підтримку та розвиток освітньої платформи.

API OpenAI - це платформа, яка надає доступ до різних моделей OpenAI, таких як GPT-3.5, GPT-4, DALL-E, Whisper, Codex та інших. API ChatGPT є частиною API OpenAI, що дозволяє розробникам інтегрувати можливості ChatGPT у додатки, веб-сайти, сервіси тощо. Завдяки цьому API розробники мають доступ до різних мовних моделей, можуть створювати інтерактивні чат-боти, генерувати тексти для різних цілей, використовувати їх для аналізу та

інтерпретації тексту тощо. Для використання API OpenAI потрібна реєстрація в OpenAI, з якої можна отримати ключ API для доступу до сервісу. Запити до API здійснюються через запити HTTP POST. Запит зазвичай містить текстове введення (підказку), на яке модель повинна відповісти. Відповіді від API повертаються як об'єкти JSON, що містять згенерований текст.

Основні ролі в системі управління вікторинами – це адміністратор, викладач та учень. Важливим елементом системи є модуль для автоматизованої генерації тестів, який у поточному рішенні використовує API OpenAI, але в майбутньому може інтегрувати інші інструменти штучного інтелекту. Ключові дії, пов’язані з роботою з вікторинами, що виконуються цими учасниками:

- Введення тексту з навчальним змістом: Викладач вводить текст - навчальний зміст, на основі якого мають бути створені тестові питання та відповіді. Правильна відповідь позначається як така.
- Генерація запитань та відповідей: Система використовує спеціалізований модуль та ChatGPT API для генерації кількох запитань та відповідей на основі введеного тексту.
- Затвердження тестових питань: Викладач переглядає згенеровані питання та відповіді й затверджує відповідні на власний розсуд. Затверджені відповіді зберігаються в базі даних. Існує можливість подальшого редагування питань та відповідей.
- Створення вікторин: Викладач може створювати вікторини, вибираючи з тестових питань, що зберігаються в базі даних.
- Проходження тестів: Учні проходять тести за завданням викладача.
- Оцінювання тестів: Система автоматично оцінює тести, надані учнями.
- Перегляд тестів: Учні мають можливість переглянути оцінені тести та допущені ними помилки.
- Керування користувачами: Адміністратор може керувати обліковими записами користувачів, зокрема створювати, змінювати та видаляти облікові записи.

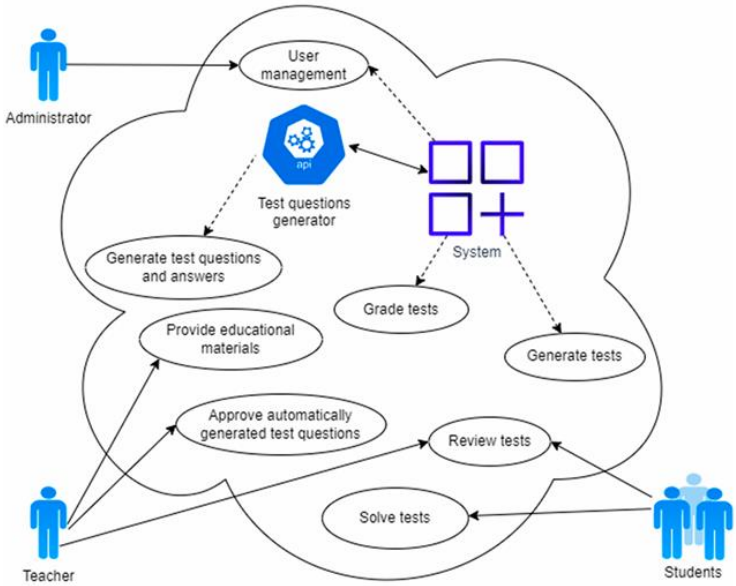

Рисунок 3.1 - Основні ролі та функції.

Під час створення тестових питань (питань, на додаток до стандартних аспектів, що стосуються тему та цільову групу, викладач має можливість вказати такі характеристики для створення тестових питань:

- Рівень таксономії Блума, наприклад, знання, розуміння, застосування, аналіз, синтез, оцінювання
- Складність питань, наприклад, *легкі* , середні, важкі
- (Стиль питань, наприклад, академічний, розмовний, технічний
- Зворотній зв'язок: наприклад, немає зворотного зв'язку, правильна/неправильна (тобто вказується лише правильна чи неправильна відповідь), *коротке* пояснення кожної правильної відповіді тощо.
- Конкретні рекомендації щодо запитань, наприклад, уникнення запитань з оманливими відповідями, створення запитань, що вимагають критичного мислення, включаючи реальні приклади, що включають міждисциплінарні $_{\text{питання}}$, короткі та точні формулювання тощо.

Тренування специфічних навичок, наприклад, обчислювальних навичок, розв'язання задач, розуміння прочитаного , навичок інтерпретації даних, мовних навичок, дослідницьких навичок тощо.

Слід зазначити, що ці характеристики тесту обирає конкретний викладач. Будь-які інші характеристики можна додати аналогічним чином.

Модель великої мови (LLM) – це тип моделі машинного навчання, навченої розуміти та генерувати текст природною мовою. LLM базуються на глибоких нейронних мережах і зазвичай містять мільярди параметрів, що дозволяє їм обробляти складні текстові завдання, такі як переклад, підсумовування, відповіді на запитання, генерація тексту, переклад іноземними мовами тощо. LLM широко використовуються в чат-ботах, системах автозаповнення, пошукових системах та інших сервісах обробки природної мови.

Процес створення та вдосконалення текстових підказок, які подаються до LLM, таких як GPT-3, GPT-4 та інших, для досягнення бажаних результатів, відомий як інженерія підказок. Це вирішальний аспект взаємодії з LLM, оскільки добре сформульовані підказки можуть значно покращити якість і точність відповідей, що генеруються моделлю.

Зв'язок між клієнтським застосунком та LLM вимагає ретельного формулювання питань, що ставить перед моделлю. Звичайний підхід, коли користувач задає питання природною мовою та отримує відповідь у вільному тексті, непридатний для подальшої автоматизованої обробки цього тексту застосунком. Якщо результати повертаються у структурованому форматі, це значно полегшить їх автоматизовану обробку клієнтським застосунком.

JSON – це широко використовуваний стандарт для передачі даних в інтернет-комунікації між програмами. У контексті LLM, генерація виводу у форматі JSON забезпечує чітку та організовану структуру даних, яку клієнтська програма може легко інтерпретувати та обробляти. Багато останніх версій провідних LLM API, включаючи OpenAI та Gemini, пропонують можливість генерувати структурований вивід JSON. Однак різні API використовують для цієї мети різні функції та параметри, що може ускладнити логіку клієнтської програми та перешкодити майбутній інтеграції нових LLM у неї.

Наше дослідження показує, що всі основні LLM, такі як ChatGPT, Gemini, Llama, Claude, Mistral, Cohere, Reka, DeepSeek та інші, можуть генерувати структурований вивід JSON, якщо запит правильно сформульовано та містить інструкції для бажаного формату виводу. Це дозволяє клієнтській програмі

однаково працювати з різними LLM API, використовуючи один і той самий запит, та отримувати результати у форматі, визначеному розробниками.

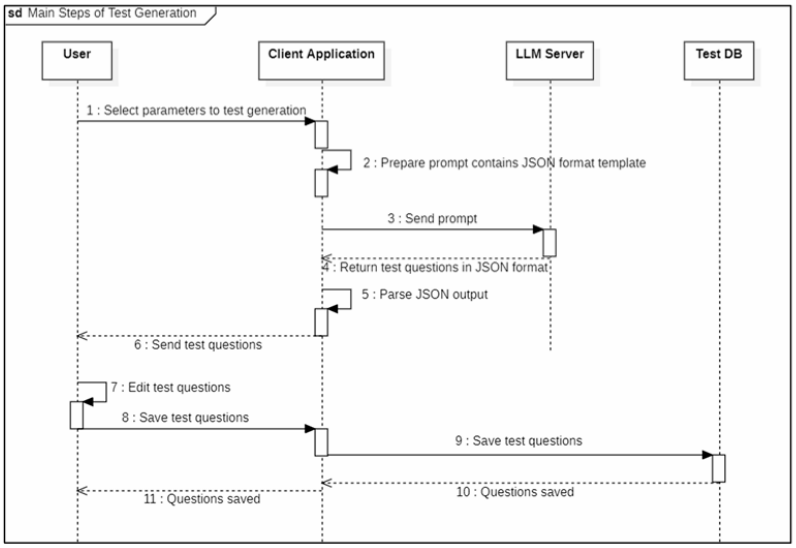

Рисунок 3.2 - Основні етапи процесу формування тестів.

На рисунку 3.2 представлено діаграму послідовності UML, що ілюструє основні кроки процесу автоматизованої генерації тестових питань за допомогою LLM. Ці кроки включають:

- Конфігурація вчителя: Користувач з роллю вчителя налаштовує параметри для створення тестових питань, такі як тема, цільова група, складність тесту, типи питань та інші конкретні вимоги.
- Генерація запиту: Після запуску процесу програма генерує запит, який містить параметри, визначені вчителем, та шаблон JSON для структури відповіді.
- Надсилання запиту: Клієнтська програма надсилає цей запит до LLM через API відповідної моделі.
- Відповідь LLM: LLM повертає вихід, який містить згенеровані тестові питання та відповіді у форматі JSON.
- Обробка виводу: Клієнтська програма обробляє отриманий вивід JSON та відображає запитання та відповіді у зручному для користувача форматі.

Перевірка та зберігання вчителем: Вчитель перевіряє та редагує питання, після чого вони зберігаються в базі даних програми.

Шаблон запиту, який можна використовувати для всіх розглянутих LLM, представлено на рисунку 3.3. Він повертає структурований результат JSON, що містить тестові питання та відповіді.

```
Create <n> sample test questions with <m> answers in the field of «<subject area>' in the follow-
ing JSON format:
    { "questions": [
    {
    "text": "Question text",
    "answers": [
    {
    "text": "Answer text",
    "isCorrect": true or false,
    "feedback": "Explanation of why the answer is correct or incorrect"
    }
    ]
    }
    ]
    }
```

Рисунок 3.3 - Промпт, розроблений для великої мови (LLM), що визначає формат виводу у форматі JSON.

Приклади значень для параметрів <n>, <m> та <предметна область> – це "5", "4" та "Бази даних" відповідно.

Представлена JSON-структура складається з масиву питань, кожне з яких має два піделементи: текст питання та масив відповідей. Відповіді, у свою чергу, мають три піделементи: текст відповіді, логічне поле, що вказує, чи є відповідь правильною чи неправильною, та зворотний зв'язок, що пояснює, чому відповідь правильна чи неправильна.

Використовуючи контекст, наданий шаблоном JSON, LLM можуть визначити логіку структури та заповнити відповідні елементи JSON відповідним вмістом. Якщо вказано лише типи даних без детальних пояснень для полів, це може ускладнити для LLM "розуміння" та правильне застосування логіки шаблону JSON.

Хоча JSON є широко використовуваним та зручним форматом, можна також обрати інші формати для структурованого виводу, такі як XML, YAML, Markdown, Protobuf, SQL тощо.

Додаток для створення та керування вікторинами може працювати незалежно або бути інтегрованим у навчальну платформу. Навчальна платформа, у свою чергу, може бути інтегрована в комплексну університетську систему, забезпечуючи стандартизований доступ користувачів до послуг усіх університетських систем. Узагальнена архітектура навчальної платформи представлена на рисунку 3.4 і включає рівень клієнта, інтерфейс користувача, рівень компонентів, рівень доступу до бази даних та зовнішніх служб, системи управління базами даних та інші зовнішні служби.

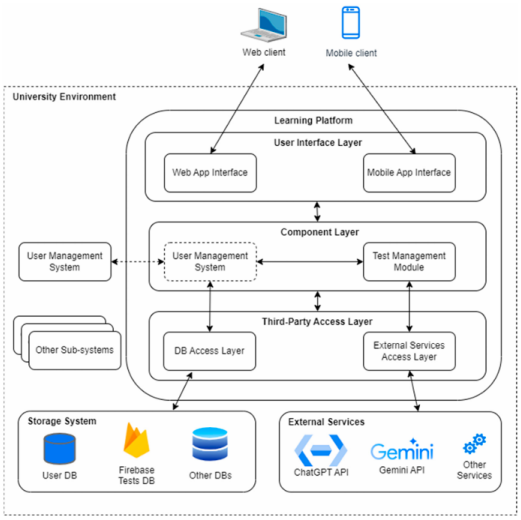

Рисунок 3.4 - Архітектура системи.

Користувачі отримують доступ до функцій системи через інтерфейс, що надається веб- або мобільним додатком. Незалежно від використовуваного інтерфейсу, користувач повинен мати можливість вільного доступу до всіх функцій. *Ключовою* перевагою використання мобільних додатків є легкий доступ з будь-якого місця, тоді як для веб-додатків це краща організація робочих екранів.

Важливі функції для різних ролей користувачів (адміністратори, викладачі та учні) на рівні презентації:

- Керування інформацією профілю користувача
- Введення даних і тексту для створення тестових питань; затвердження та редагування тестових питань і відповідей; налаштування тестів

- Проходження тестів, перегляд отриманих результатів та багато іншого.

На другому рівні розташовані компоненти, що реалізують основну бізнес-логіку системи. Модуль керування користувачами повинен координувати свої дані із зовнішнім середовищем. Модуль керування тестуванням надає можливості для:

- Генерація тестових питань та налаштування тестів
- Виконання та автоматичне оцінювання тестів
- Зберігання тестів у базі даних
- Персоналізація для користувачів за допомогою Системи керування користувачами.

Важливою частиною системи управління тестуванням є модуль генерації тестових питань, який розроблено з використанням технології Node.js. У поточній версії тестові питання та відповіді на них генеруються за допомогою API ChatGPT на основі текстових матеріалів, наданих викладачем. Згенеровані питання затверджуються відповідним викладачем, а потім зберігаються в базі даних. Їх можна редагувати та використовувати для створення тестів.

Комплексна система управління навчанням може підтримувати багато інших компонентів, таких як «Управління курсом та навчальними матеріалами», «Інтерактивні уроки», «Форуми» тощо.

Компоненти другого рівня взаємодіють із системами керування базами даних та іншими зовнішніми системами й службами через модулі стороннього рівня доступу.

Система керування базами даних Google Firebase забезпечує надійне та ефективне зберігання всіх важливих даних, включаючи інформацію про вікторини, їхні запитання, профілі користувачів та результати завершених вікторин. Зв'язок між усіма компонентами здійснюється через RESTful запити, що забезпечує безпеку та ефективність комунікації системи. Система пропонує можливості для майбутнього розширення шляхом інтеграції інших баз даних та зовнішніх сервісів, включаючи інші системи штучного інтелекту, такі як Gemini, Llama, Claude тощо.

Основні сутності в базі даних для зберігання згенерованих тестових питань, складених тестів та їх виконання представлені на рисунку 3.5. Для візуального представлення бази даних під час процесу проектування за допомогою MySQL Workbench було створено розширену діаграму сутностей-зв'язків .

Основними об'єктами в базі даних вікторин є:

- Користувачі: Містить інформацію про всіх користувачів системи, включаючи адміністраторів, викладачів та учнів. Атрибути включають унікальний ідентифікатор, ім'я користувача, пароль та роль у системі.
- Освітні ресурси: Зберігає ресурси у вигляді тексту або файлів, які використовуються для створення тестових питань. Містить унікальний ідентифікатор ресурсу, текст, ім'я файлу та ідентифікатор автора, який є зовнішнім ключем для користувача, що надав ресурс.
- Категорії тестів: Зберігає категорії тестів, які визначають як окремі питання, так і тест загалом. Атрибути включають ідентифікатор категорії та її назву.
- Запитання: Містить питання, згенеровані за допомогою OpenAI API. Включає ідентифікатор питання, ідентифікатор категорії питання, вміст питання та додатковий зовнішній ключ до освітнього ресурсу, оскільки питання можна створювати без прив’язки до конкретних освітніх ресурсів. Додаткові атрибути для додаткової категоризації питання, що використовуються під час його генерації з OpenAI API, включають рівень Блума, складність, рекомендації, стиль питання та тренування навичок.

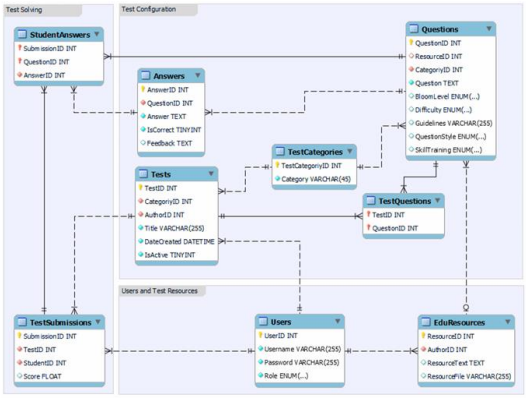

Рисунок 3.5 - Розширена діаграма «сутності-зв’язки» моделі даних.

- Відповіді: Зберігає всі відповіді, пов'язані з питаннями. Складається з ідентифікатора відповіді, ідентифікатора питання, тексту відповіді, зворотного зв'язку до відповіді та логічного прапорця, який вказує на правильність відповіді.
- Тести: Зберігає інформацію про створені тести, включаючи унікальний ідентифікатор тесту, назву тесту, ідентифікатор автора тесту, ідентифікатор категорії тесту, дату створення тесту та логічний прапорець, який вказує, чи є тест активним.
- Тестові питання: Встановлює зв'язок між тестами та питаннями та містить ідентифікатори тесту та питання.
- "Надіслані тестові завдання: Записує надіслані тестові рішення, які знаходять учні, включаючи унікальний ідентифікатор надісланого завдання, ідентифікатор тесту, ідентифікатор студента та бал."
- Відповіді тесту: Містить відповіді, дані учнями на тестові запитання, включаючи ідентифікатор завдання, ідентифікатор запитання та необов'язковий ідентифікатор наданої відповіді.

3.2 Реалізація та аналіз пілотного проєкту автоматизованого тестування

Показано кількість тестів, які генерує TEST PILOT для кожного пакета, кількість (і частку) тестів, що пройшли успішне проходження, та відповідне покриття, досягнуте за допомогою тестів, що пройшли успішне проходження. Показують покриття, отримане шляхом простого завантаження пакета *(покриття завантаженням).* Це покриття, яке ми отримуємо «безкоштовно» без будь-якого набору тестів, який ми надаємо як точку відліку для інтерпретації наших результатів. Загалом, 9,9%-80,0% тестів, згенерованих TEST PILOT , є тестами, що пройшли успішне проходження, з медіаною 48,0% для всіх пакетів. Тепер ми обговоримо різні вимірювання покриття цих тестів, що пройшли успішне проходження.

Покриття операторів: Покриття операторів для кожного пакета, досягнуте

в результаті успішного проходження тестів, коливається від 33,9% до 93,1%, з медіаною 70,2%. Зазначимо, що у всіх пакетах досягнуте покриття операторів значно вище, ніж покриття завантаження, з різницею 19,1% - 88,2% та медіанною різницею 53,7% .

Включення витягнутих фрагментів прикладів із цього зовнішнього репозиторію збільшує досягнутий рівень покриття до 43,6%, що свідчить про важливість включення прикладів використання до підказок. Ми детально розглядаємо вплив інформації, що міститься в підказках, у RQ5 .

Варто зазначити, що покриття T EST PILOT для проектів GitLab, перелічених у нижніх 5 рядках, коливається від 51,4% до 78,3%. Це демонструє, що T EST PILOT ефективно генерує модульні тести з високим покриттям для пакетів, які не зустрічалися у його навчальному наборі.

Покриття гілок: Ми також показуємо покриття гілок, досягнуте шляхом успішного проходження тестів. Ми виявили, що покриття гілок на пакет становить від 16,5% до 71,3%, з медіаною 52,8%. Подібно до покриття операторів, досягнуте покриття гілок також значно вище, ніж покриття завантаження, з різницею 15,9% - 71,3% та медіанною різницею 50,0%.

Оскільки досягнення покриття гілок, як правило, складніше, ніж досягнення покриття операторів, очікується, що покриття гілок для згенерованих тестів буде нижчим, ніж покриття операторів. Однак, ми відзначаємо цікавий випадок у gitlab-js , де ця різниця здається більш помітною (51,7% проти 16,5%). Після подальшого дослідження його вихідного коду та документації ми виявляємо, що gitlab-js пропонує різні опції конфігурації та параметри для визначення репозиторію GitLab для підключення та використання/запиту (наприклад, його URL-адреса, токен автентифікації, параметри пошуку для використання для запиту). Обробка цих опцій відображається в основній логіці розгалуження в коді. Хоча T EST PILOT намагається генерувати розумні тести, які викликають різні кінцеві точки з різними опціями, іноді йому важко знайти правильний виклик функції для використання, що призводить до помилок типу. Загалом, значна частина тестів, які T EST PILOT генерує для цього пакета, зазнає невдачі, і тому не враховуються в наших показниках покриття. Варто також

зазначити, що належне тестування такого пакета вимагатиме імітації, але ми не спостерігали жодного з них.

Покриття для кожної функції: На рисунку 3.5 показано розподіл покриття операторів для кожної функції для кожного пакета. Кожен блок відповідає одному з наших контрольних пакетів, а кожна точка даних у блоку представляє покриття операторів для функції в цьому пакеті. Медіанне покриття операторів для кожної функції для кожного пакета показано червоним кольором.

Загалом, медіана покриття операторів на функцію для даного проєкту коливається від 0,0% до 100,0%, з медіаною 77,1%. Щоб переконатися, що T EST PILOT не генерує тести з високим покриттям лише для менших функцій, ми проводимо тест кореляції Пірсона між покриттям операторів на функцію та відповідним розміром функції (в операторах). Ми не виявили статистично значущої кореляції між покриттям та розміром, що вказує на те , що T EST PILOT добре працює не лише для менших функцій .

Як і очікувалося, на рисунку 3.5 показано , що для більшості пакетів T EST PILOT добре справляється з деякими функціями, але досягає низького покриття для інших. Візьмемо як приклад jsonfile . У таблиці 2 ми побачили, що покриття операторів на рівні пакета становить 38,3%. З рисунка 5 ми бачимо, що покриття операторів для кожної функції коливається від 0% до 100%, з медіаною майже 50%. Заглиблюючись у дані, ми виявляємо, що є дві функції, які T EST PILOT не може охопити, оскільки відповідні згенеровані тести не виконуються або через посилання на неіснуючі файли, які T EST PILOT включає в тести , або через те, що вони мають тайм-аут. Однак функції, які T EST PILOT може охопити, мають покриття операторів у діапазоні від 58% до 100%. Ми можемо спостерігати подібну поведінку з іншими пакетами, залежними від файлової системи, такими як grace-fs або fs-extra . На іншому кінці спектра ми бачимо zip-a-folder , де TEST PILOT досягає як високого покриття операторів на рівні пакета (84%), так і високого покриття операторів на рівні функції на рисунку 3.5 , де мінімальне покриття на функцію становить 75%.

Тести з унікальним внеском: Щоб краще зрозуміти різноманітність згенерованих тестів, скільки тестів, що генеруються T EST PILOT , мають

унікальний внесок, тобто вони охоплюють принаймні одне твердження, яке не охоплюють інші тести. Медіана 10,5% тестів, що пройшли тестування , є такого типу, а деякі пакети мають показник сягає 100,0%. Ці результати є багатообіцяючими, оскільки вони показують, що T EST PILOT може генерувати тести, які охоплюють граничні випадки, але серед згенерованих тестів явно існує певна надлишковість. Звичайно, ми не можемо просто виключити всі 89,5% тестів, що залишилися, без втрати покриття, оскільки деякі твердження можуть бути охоплені кількома тестами неоднозначно. Дослідження методів мінімізації набору тестів [54] для зменшення розміру згенерованого набору тестів є цікавим напрямком для майбутньої роботи.

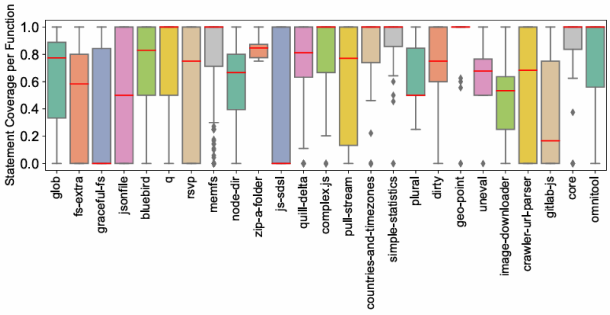

Рисунок 3.5 - Розподіл охоплення тестових висловлювань за функціями для тестів, згенерованих TESTPILOT з використанням GPT-3.5-Turbo.

Додатковий підхід, спрямований на зворотний зв'язок . Для кожного пакета Nessie генерує 1000 тестів, для яких ми вимірюємо покриття операторів та гілок так само, як і для T EST PILOT . Потім ми повторюємо ці вимірювання 10 разів і беремо медіану покриття для 10 запуску, щоб дотримуватися налаштування, подібного до оцінювання T EST PILOT . Ми використовуємо парний ранговий тест Вілкоксона, щоб визначити, чи є статистично значущі відмінності між покриттям, досягнутим обома інструментами.

Показано покриття операторів та гілок для Nessie. Зазначимо, що Nessie не міг працювати на uneval , оскільки єдиним експортом модуля є функція, яку Nessie не підтримує. Для решти 24 пакетів Nessie досяг покриття операторів від 4,7% до 96,0% з медіаною 51,3%. На противагу цьому, медіана покриття операторів T EST PILOT набагато вища і становить 70,2%. Різниця в покритті

гілок ще більша: 52,8% для T EST PILOT проти 25,6% для Nessie . Обидві ці відмінності є статистично значущими (p-значення 0,002 та 0,027 відповідно) з великим розміром ефекту, 0,493 для покриття операторів та середнім (0,431) для покриття гілок . [10] Зверніть увагу, що Nessie завжди генерує 1000 тестів на пакет, тоді як T EST - PILOT зазвичай генерує набагато менше тестів, за винятком memfs та omnitool . Варто також підкреслити, що Nessie (та інші методи генерації тестів, такі як LambdaTester) повідомляють про покриття *всіх* згенерованих тестів, незалежно від того, чи пройшли вони успішно, чи ні, тоді як наші звітні показники покриття стосуються лише успішних тестів.

Тепер заглибимося в результати на рівні пакета. Для кожного пакета в показано вище покриття за допомогою двох методів, виділених жирним шрифтом. TEST PILOT перевершує Nessie у 17 з 24 пакетів (glob, fs-extra, blue bird, q, rsvp, memfs, js-sdsl, quill-delta, complex.js, pull-stream, simple-statistics, plural, dirty, geo-point, image-downloader, core, omnitool), збільшивши покриття на 3,6% до 74,5%, із медіаною збільшення 30,0%. Для 7 з решти пакетів (graceful-fs, jsonfile, node-dir, zip-a-folder, countries-and-timezones, crawler-url-parser, gitlab-js), TEST PILOT досягає нижчого покриття , ніж Nessie. Для цих пакетів покриття зменшується на 0,5% - 53,2%, із медіаною зниження на 3,6%. Ми також зазначаємо, що Nessie не досягає жодного покриття гілок у 3 проектах (dirty, geo-point, core), тоді як покриття операторів для цих проектів не дорівнює нулю. Після подальшого дослідження та консультацій з авторами Nessie ми виявили, що Nessie не може генерувати тести, які створюють екземпляри класів, а це означає, що покриття операторів трохи перевищує покриття навантаження для пакетів з API на основі класів, тоді як покриття гілок дорівнює нулю.

```
let manuelmhtr_countries_and_timezones = require("../../TEST_REPO_manuelmhtr_countries_and_timezones");
module.exports = function() {
        let ret_val_manuelmhtr_countries_and_timezones_1;
        try {
                ret_val_manuelmhtr_countries_and_timezones_1 = manuelmhtr_countries_and_timezones.getAllCountries();
                Promise.resolve(ret_val_manuelmhtr_countries_and_timezones_1).catch(e => { console.log({"error_1": true}); });
        } catch(e) {
                console.log({"error_1": true});
        }
        let ret_val_manuelmhtr_countries_and_timezones_2;
        try {
                ret_val_manuelmhtr_countries_and_timezones_2 =
                  manuelmhtr_countries_and_timezones.getCountry({"k": -293.76984807333383,
                                                                 "rHMR": -17.71151399309167,
                                                                 "vSF6": 721.0602634375625,
                                                                 "1": 497.17371230897766,
                                                                 "EnL": -611.9090030925536});
                Promise.resolve(ret_val_manuelmhtr_countries_and_timezones_2).catch(e => { console.log({"error_2": true}); });
        } catch(e) {
                console.log({"error_2": true});
        }
}
```

Рисунок 3.6 - Приклад тесту, згенерованого Nessie. Виділені рядки призначені виключно для налагодження і не впливають на тестування пакета, що перевіряється.

Окрім різниці в охопленні, досягнутої Nessie та T EST PILOT , тести , згенеровані Nessie, як правило, виглядають досить відмінно від тих, що згенеровані T EST - PILOT , що випливає з випадкового підходу Nessie до генерації тестів. Щоб проілюструвати це, на рисунку 6 показано приклад тесту, згенерованого Nessie, який виконує функцію getCountry для countries-and-timezones . Як видно на рисунку, тест використовує довгі імена змінних, такі як ret_val_manuelmhtr_countries_and_timezones_1 , що ускладнює читабельність . Крім того, тест викликає getCountry у рядках 13-17 з об'єктним літералом, який пов'язує випадкові значення з деякими випадково названими властивостями, що не відображає передбачуваного використання API. Крім того, тести, згенеровані Nessie, не містять жодних тверджень. На противагу цьому, тести , згенеровані T EST PILOT для того самого пакета (див. рисунок 3), зазвичай використовують імена змінних, подібні до тих, що вибирають програмісти, викликають API з розумними значеннями та часто містять твердження.

RQ3 : Нетривіальні твердження

Ми визначаємо *нетривіальне твердження* як твердження, яке залежить принаймні від однієї функції з тестованого пакета. Щоб ідентифікувати нетривіальні твердження, ми спочатку використовуємо CodeQL [57] для обчислення зворотного зрізу програми з кожного твердження у згенерованих тестах. Ми розглядаємо твердження, зворотний зріз яких містить імпорт

тестованого пакета, як нетривіальні твердження. Потім ми повідомляємо про згенеровані тести, які містять принаймні одне нетривіальне твердження.

Ми спостерігаємо, що існує лише один пакет, image-downloader, де T EST PILOT генерує лише тривіальні тести. Хоча згенеровані тести для image-downloader містили виклики до його API, у всіх них були відсутні оператори assert. Для решти пакетів медіана від 9,1% до 94,6% згенерованих тестів T EST PILOT на пакет є нетривіальними. Медіана 61,4% згенерованих тестів для даного пакета є нетривіальними. Порівняно з усіма згенерованими тестами, ми також бачимо, що лише трохи менша частка нетривіальних тестів проходить успішно (медіана 48,0% для загального успішного проходження тестів проти 43,7% для нетривіальних успішних тестів) . Обидва ці результати показують, що T EST PILOT зазвичай генерує тести з твердженнями , які реалізують функціональність цільового пакета.

Покриття, досягнуте нетривіальними тестами, також підтверджує цей висновок. Зокрема, порівнюючи покриття операторів для всіх згенерованих тестів з покриттям для нетривіальних тестів, ми виявляємо, що різниця - коливається від 0,0% до 84,0%, з медіанною різницею лише 7,5%. Це означає, що досягнуте покриття для більшості пакетів в основному пов'язане з використанням функціональності API, яка тестується згенерованими оракулами. Однак зазначимо, що є 4 пакети (jsonfile, node-dir, zip-a-folder, image-downloader), де нетривіальні тести досягають 0% покриття операторів, що спричиняє більші відмінності. Окрім image-downloader, про який йшлося вище, три інші пакети не мають жодних успішно пройдених нетривіальних тестів. Оскільки ми розраховуємо покриття лише для успішно пройдених тестів, це призводить до 0% покриття операторів для нетривіальних тестів.

Таблиця 3.1

Кількість (у %) нетривіальних тестів TESTPILOT, згенерованих за допомогою GPT-3.5-Turbo, та відповідний рівень покриття операторів, отриманий за результатами успішного виконання нетривіальних тестів.

Проект	Небанальні тести	Проходження	Стейтмент-

	(%)	небанальних тестів	покриття (Stmt Cov)
Назва	**К-сть (%)**	**Тести (%)**	**Покриття (%)**
glob	37 (54.4%)	3 (8.1%)	50.1%
fs-extra	142 (30.1%)	70 (49.5%)	28.0%
graceful-fs	64 (18.4%)	27 (42.5%)	41.5%
jsonfile	4 (32.0%)	0 (0.0%)	0.0%
bluebird	227 (61.4%)	137 (60.4%)	61.6%
q	235 (72.6%)	136 (58.0%)	66.4%
rsvp	68 (62.4%)	48 (70.6%)	67.6%
memfs	758 (73.1%)	356 (47.0%)	77.4%
node-dir	7 (16.5%)	0 (0.0%)	0.0%
zip-a-folder	1 (9.1%)	0 (0.0%)	0.0%
js-sdsl	349 (85.3%)	44 (12.6%)	33.9%
quill-delta	92 (60.5%)	27 (28.8%)	59.7%
complex.js	190 (90.9%)	104 (54.6%)	62.7%
pull-stream	60 (72.3%)	29 (47.5%)	64.7%
countries-and-timezones	22 (78.6%)	7 (31.8%)	73.5%
simple-statistics	189 (53.6%)	115 (60.6%)	46.9%
plural	12 (92.3%)	8 (66.7%)	73.8%
dirty	29 (41.7%)	13 (44.8%)	66.0%
geo-point	60 (78.9%)	34 (56.7%)	64.6%
uneval	4 (57.1%)	2 (50.0%)	68.8%
image-downloader	0 (0.0%)	—	0.0%
crawler-url-parser	6 (42.9%)	1 (16.7%)	49.5%
gitlab-js	104 (73.8%)	12 (11.5%)	49.3%
core	64 (74.7%)	12 (18.9%)	75.5%
omnitool	977 (94.6%)	319 (32.6%)	73.8%
Медіана	**61.4%**	**43.7%**	**61.6%**

RQ4 : Характеристики невдалих тестів

Помилки твердження виникають, коли очікуване значення в твердженні не відповідає фактичному значенню, отриманому в результаті виконання коду. Помилки файлової системи включають такі помилки, як неможливість знайти файли або каталоги, які ми ідентифікуємо, перевіряючи коди помилок, пов'язані з файловою системою, у трасуванні стека помилок. Помилки коректності включають усі помилки типу, синтаксичні помилки, помилки посилань, неправильні виклики done та помилки нескінченної рекурсії/стеку викликів. Помилки тайм-ауту виникають, коли тести перевищують максимально дозволений час виконання (2 с/тест). Нарешті, ми групуємо всі інші помилки, пов'язані з програмою, які ми спостерігаємо, в розділі "Інше".

На рисунку 3.7 показано кількість невдалих тестів для кожного пакета, а також розподіл причин невдачі.

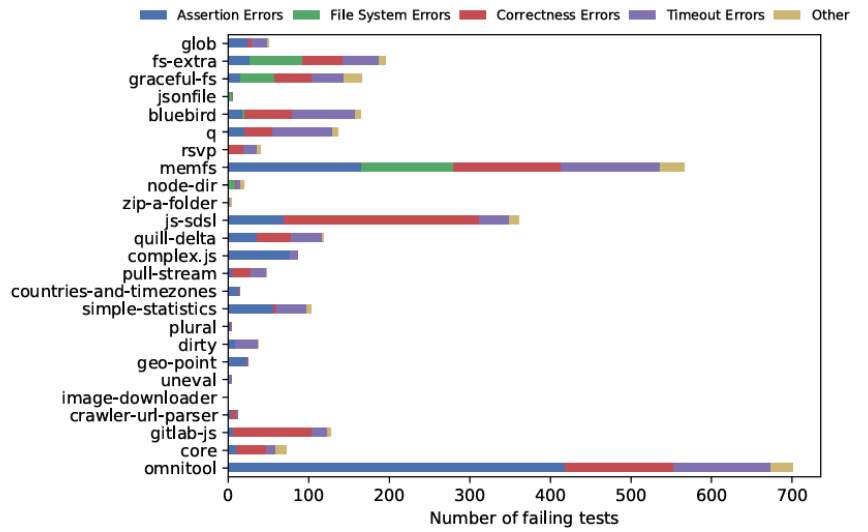

Рисунок 3.7 - Типи помилок у тестах, що завершилися невдачею, згенерованих TESTPILOT із використанням gpt3.5-turbo.

Ми виявили, що найпоширенішою причиною невдачі є тайм-аути з медіаною 22,7% невдалих тестів, далі йдуть помилки коректності (особливо помилки типу) з медіаною 20,0% невдалих тестів. Більшість тайм-аутів пов'язані з пропущеними викликами done , що змушує Mocha продовжувати очікувати виклику. Зазначимо, що в середньому уточнювач *RetryWithError* зміг виправити 15,4% таких помилок тайм-ауту, причому модель часто просто додаючи виклик до done.

Ми виявили, що медіана 19,2% невдач є помилками твердження, що вказує на те, що в деяких випадках *gpt3.5-turbo* не може визначити правильне очікуване значення для тестового оракула. Це особливо актуально, коли тестований пакет не використовується широко, і жодна інформація, яку ми надаємо моделі, не може допомогти їй визначити правильні значення. Наприклад, в одному з тестів для geo -point , T EST PILOT зміг використати координати з наданого фрагмента прикладу, щоб правильно побудувати дві географічні координати як вхідні дані для функції calculateDistance , яка обчислює відстань між двома координатами. Однак, T EST PILOT неправильно

згенерував 131.4158102876726 як очікуване значення відстані між цими двома точками, тоді як правильне очікуване значення становить 130584.05017990958; це призвело до невдалого завершення тесту з помилкою твердження. Зазначимо, що в цьому конкретному випадку, коли TEST PILOT повторно запросив модель з повідомленням про невдалий тест та помилку, він зміг створити успішний тест з виправленим оракулом. В середньому по пакетах ми виявили, що уточнювач *RetryWithError* зміг виправити 11,1% помилок тверджень.

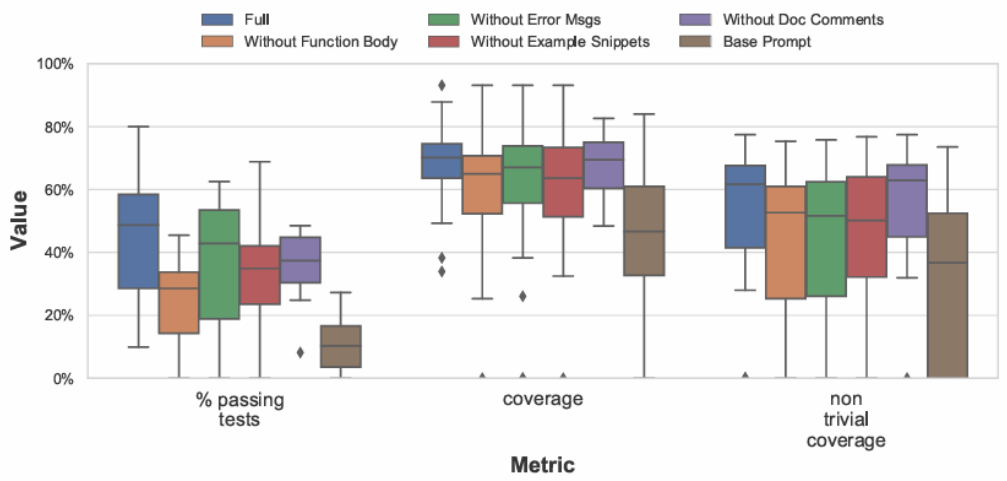

Рисунок 3.8 - Вплив вимкнення вбудованих механізмів уточнення запитів у TESTPILOT при використанні моделі gpt3.5-turbo. У режимі «Full» враховуються всі механізми уточнення, тоді як у режимі «Base» - лише сигнатура функції та каркас Mochas.

Зрештою, зазначимо, що помилки файлової системи є специфічними для домену. Згенеровані тести для пакетів у домені файлової системи, такі як fs-extra або memfs , мають високу частку невдалих тестів через такі помилки. Це не дивно, враховуючи, що ці тести можуть спиратися на файли, які можуть не існувати або вимагати певного вмісту. Пакети в інших доменах не стикаються з цією проблемою, що повторне викликання моделі повідомленням про помилку невдалих тестів (незалежно від причини невдачі) дозволяє TEST PILOT створити послідовний результат успішного проходження тесту у 15,6% випадків.

RQ5 : Вплив швидких переробників

Наші результати наразі включають тести, згенеровані за допомогою всіх чотирьох швидких уточнювачів, обговорених у розділі 2. У цьому запиті ми досліджуємо вплив кожного з цих уточнювачів на якість згенерованих тестів. Зокрема, ми проводимо дослідження абляції, де ми вимикаємо один уточнювач за раз. Вимкнення

Хоча додавання відсутніх викликів до done може здаватися простим і тому підлягає автоматичному виправленню, знайти правильні місця для вставки таких викликів може бути напрочуд складно, а правильна обробка цього вимагатиме застосування статичного аналізу до згенерованого тесту. Тому ми обрали автоматизований підхід, який спирається виключно на LLM, але розглянемо використання статичного аналізу для відновлення згенерованих тестів у майбутньому.

```
let mocha = require('mocha');
let assert = require('assert');
let complex_js = require('complex.js');
// complex.js.ZERO.valueOf()
describe('test suite', function() {
    it('test case', function(done) {
        assert.equal(complex_js.ZERO.valueOf(), 0);
        done();
    })
})
```

(a)

```
let mocha = require('mocha');
let assert = require('assert');
let complex_js = require('complex.js');
// complex.js.ZERO.valueOf()
// function() {
//   if (this['im'] === 0) {
//     return this['re'];
//   }
//   return null;
// }
describe('test complex_js', function() {
    it('test complex.js.ZERO.valueOf', function(done) {
        assert.equal(complex_js.ZERO.valueOf(), null);
        done();
    })
})
```

(b)

Рисунок 3.9 - Приклад того, як уточнення негативно впливає на генерацію тестів.

У підказці (а) немає жодної інформації, крім сигнатури методу, і згенерований тест проходить успішно. У підказці (б) додано тіло методу, але згенерований тест завершується з помилкою.

Уточнення означає, що ми більше не генеруємо запити, які містять надану ним інформацію. Наприклад, вимкнення *DocCommentIncluder* означає, що жодний із запитів, які ми генеруємо, не міститиме коментарів до документації. Потім ми порівнюємо відсоток успішного проходження тестів, досягнуте покриття, а також покриття нетривіальними тестами *(нетривіальне покриття)*.

На рисунку 3.8 показано наші результати. Вісь x показує показники, які ми порівнюємо для різних конфігурацій, показаних у легенді. Вісь y показує значення для кожного показника (усі відсотки). Кожна точка даних на коробковій діаграмі представляє результати конкретного показника для заданого пакета, використовуючи відповідну конфігурацію уточнення. Чорна лінія посередині кожного поля представляє медіанне значення для кожного показника для всіх пакетів. Повна конфігурація - це конфігурація, яку ми представили досі (тобто всі уточнення ввімкнено). Інші конфігурації показують результати виключення лише одного з уточнень. Наприклад, червона коробкова діаграма показує результати після вимкнення SnippetIncluder *(* тобто *без прикладів фрагментів).* Базова конфігурація запиту містить лише сигнатуру функції та тестове скарбування (тобто вимкнено всі уточнення). Однак зауважте, що лише 8 пакетів у нашій оцінці містять коментарі до документації. Немає сенсу порівнювати вплив вимкнення DocCommentIncluder *на* пакети, які взагалі не містять коментарів doc. Таким чином, хоча розподіли, показані на всіх блокових діаграмах, представляють 25 пакетів, конфігурація *«Без коментарів до документації»* містить дані лише для 8 пакетів.

Загалом, ми бачимо, що повна конфігурація виконує всі інші конфігурації за всіма трьома метриками, що означає, що вся інформація, яку ми включаємо, сприяє створенню ефективніших тестів. Ми виявили, що не було жодного пакета, де вимкнення уточнення призвело до кращих результатів за будь-якою метрикою. За винятком 4 пакетів, де вимкнення одного з уточнень не вплинуло на результати *(SnippetIncluder* на crawler-url-parser та dirty ; та *RetryWithError* на gitlab-js та zip-a-folder), вимкнення уточнення *завжди* призводило до нижчих значень принаймні за однією метрикою.

Внесок уточнень особливо помітний для відсотка успішних тестів, де вимкнення будь-якого з уточнень (наприклад, *FnBodyIncluder* або *SnippetIncluder*) призводить до значного падіння відсотка успішних тестів. Це свідчить про те, що уточнення ефективно спрямовують модель до генерації більшої кількості успішних тестів, навіть якщо це не обов'язково призводить до додаткового покриття. Ми виявили, що у *всіх* пакетах повна конфігурація завжди призводить

до вищого відсотка успішних тестів для даного API, зберігаючи при цьому високий рівень покриття.

Щоб зрозуміти, чи є відмінності між розподілами, які ми спостерігаємо на рисунку 8, статистично значущими, ми порівнюємо результати кожної пари конфігурацій для всіх трьох метрик, використовуючи парні парні тести Вілкоксона на знакові ранги. Зауважте, що під час порівняння з *DocCommentIncluder* ми порівнюємо розподіли лише для 8 пакетів, які містять коментарі doc.

Ми виявили статистично значущу різницю між повною конфігурацією та кожною конфігурацією, яка вимикає *будь-яке* уточнення, а також між базовою конфігурацією та кожною з інших конфігурацій. Порівняно з повною конфігурацією, найбільший розмір ефекту, який ми спостерігали для вимкнення уточнення, був при проходженні тестів, коли *FnBodyIncluder* або *DocCommentIncluder* були вимкнені (дельта Кліффа 0,582 та 0,531 відповідно).

Окрім відмінностей між повною та базовою конфігурацією, ми не знаходимо статистично значущих відмінностей між парами інших конфігурацій, за винятком наступних випадків: Ми виявили, що як для успішного проходження тестів, так і для покриття існує статистично значуща різниця між конфігурацією, яка вимикає *FnBodyIncluder* , і тією, яка вимикає *RetryWithError* (середній та незначний розміри ефекту відповідно). Для успішного проходження тестів ми також виявили статистично значущу різницю між вимкненням *FnBodyIncluder* та вимкненням кожного з *SnippetIncluder* та *DocCommentIncluder* (малий та середній розміри ефекту відповідно). Однак, ми зазначаємо, що розмір вибірки 8 занадто малий, щоб зробити будь-які обґрунтовані висновки для *DocCommentIncluder* . Особливо цікаво бачити, що не було статистично значущої різниці між вимкненням *SnippetIncluder* та вимкненням будь-якого з інших уточнень. Це свідчить про те, що відсутність прикладів фрагментів не обов'язково впливає на метрики більше, ніж відсутність будь-якої інформації, наданої іншими уточненнями. Оскільки на рисунку 8 видно, що ми все ще отримуємо високе медіанне покриття навіть при вимкненні *SnippetIncluder* , це свідчить про те, що наявність прикладів фрагментів не є обов'язковим для

створення ефективних наборів тестів з високим покриттям, і що TEST PILOT застосовується навіть у випадках , коли немає прикладів документації.

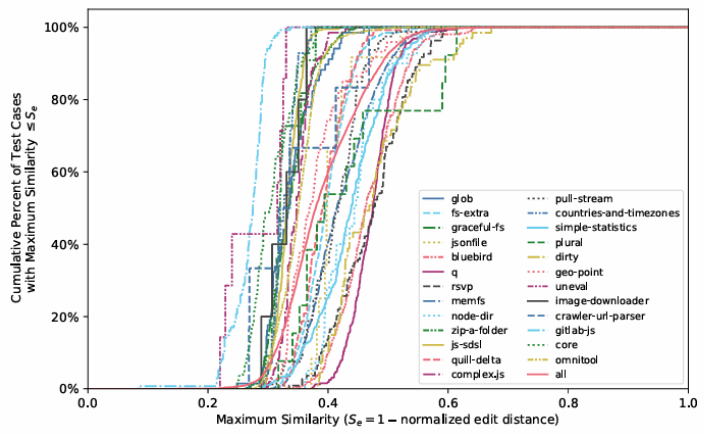

Рисунок 3.10 - Кумулятивний відсоток тестових випадків, згенерованих TESTPILOT із використанням gpt3.5-turbo, для яких максимальна схожість менша за значення схожості, вказане на осі X.

Зрештою, зазначимо, що хоча загальні результати для даного пакета показують, що уточнення завжди покращують або принаймні підтримують покриття та відсоток успішних тестів, це не означає, що уточнення завжди покращує результати для окремої функції API. Ми спостерігали ситуації, коли додавання такої інформації, як реалізація функції, до запиту, який її не включає, заплутує модель, що призводить до генерації тесту, що не пройшов. На рисунку 9 показано приклад для пакета complex.js : враховуючи базовий запит ліворуч, *gpt3.5-turbo* здатний створити (дуже простий) тест на проходження для методу valueOf константи НУЛЬ, експортованої пакетом; додавання тіла функції дає запит праворуч, що, здається, заплутує модель, що призводить до генерації тесту, що не пройшов. У всіх пакетах було згенеровано 5367 запитів шляхом застосування одного з уточнень, і лише у 394 випадках (7,3%) уточнений запит був менш ефективним, ніж оригінальний запит, у тому сенсі, що тест на проходження був згенерований з оригінального запиту, але не з уточненого запиту.

RQ6 : Запам'ятовування

Оскільки *gpt3.5-turbo* було навчено на коді GitHub, деякі з існуючих тестів, включених до наших бенчмарків, могли бути частиною його навчального набору. Це викликає занепокоєння, що TEST PILOT може запам'ятовувати існуючі тести, а не генерувати нові, що обмежує його корисність для пакетів, на яких він не був навчений. Щоб дослідити потенційний вплив запам'ятовування, ми вимірюємо подібність між кожним згенерованим тестом та існуючими тестами в бенчмарках (кількість існуючих тестів показана в таблиці 1) . Нещодавно Lemieux та ін. [27] повідомили, що методи плагіату коду або виявлення клонів неефективні для виявлення запам'ятовування коду LLM. Натомість вони виявили, що вимірювання подібності за допомогою відстані редагування дає більш значущі результати. Вони визначають *максимальну подібність* як метрику, яка вимірює нормалізовану найвищу подібність між заданим згенерованим тестом та всіма існуючими тестами наступним чином: ma x_{tpE} *TP* 1 - *max(tn (*)'!d(tp*))), де *TP* - це набір існуючих тестових функцій у пакеті, *t** – це заданий згенерований тест, а *dist* – це відстань редагування між згенерованим тестом та існуючим тестом. Ми дотримуємося того ж методу для обчислення максимальної подібності для кожного згенерованого тесту, використовуючи пакет npm Levenstein для обчислення *dist* .

```
it('test case', function(done) {
    bluebird.resolve().then(function() {
        throw new Error('test');
    }).catchThrow().catch(function(err) {
        assert.equal(err.message, 'test');
    }).finally(done);
});
```

(a)

```
it("1 level", function() {
    return Promise.resolve().then(function() {
        throw new Error();
    }).then(assert.fail, function(e) {
        assertLongTrace(e, 1 + 1, [1]);
    });
});
```

(b)

Рисунок 3.11 - Приклад тестового випадку, згенерованого TESTPILOT для bluebird (а), та існуючого тестового випадку (б) зі ступенем схожості 0,62

На рисунку 3.10 показано кумулятивний відсоток згенерованих тестових випадків для кожного проекту, де максимальна подібність менша за значення на осі x. Ми також показуємо цей кумулятивний відсоток для *всіх* згенерованих тестових випадків у всіх проектах. Ми виявили, що 6,2% згенерованих тестових випадків T EST PILOT мають максимальну подібність до існуючого тесту менше ніж < 0,3%, 60,0% мають подібність < 0,4, 92,8% мають < 0,5, 99,6% мають < 0,6 , тоді як 100,0% згенерованих тестових випадків мають < 0,7. Це означає, що T EST PILOT ніколи не генерує точні копії існуючих тестів. На противагу цьому, хоча 90% згенерованих тестів Python, розроблених Лем'є та ін. [27] , мають подібність < 0,4, 2% їхніх тестових випадків є точними копіями. Тим не менш, враховуючи відмінності між фреймворками для тестування в Python та JavaScript (наприклад, Mocha вимагає більше шаблонного коду, ніж pytest), подібні показники ity не можна безпосередньо порівнювати між цими двома мовами.

Для подальшої ілюстрації отриманих чисел подібності, на рисунку 3.11 показано приклад тестового випадку з bluebird зі схожістю 0,62 до існуючого тестового випадку. Хоча відстань редагування тут низька, що призводить до високої подібності, ми можемо бачити, що тести мають семантичні відмінності. Наприклад, згенерований тест просто перевіряє, чи є викинутий виняток помилкою типу, тоді як існуючий тест перевіряє певні значення в трасі. Таким чином, 7,2% тестових випадків, які ми генеруємо зі схожістю > 0,5, не викликають занепокоєння щодо того, що T EST PILOT генерує запам'ятовані тестові випадки. Нарешті, ми очікуємо, що згенеровані тести для проектів, розміщених на GitLab, матимуть меншу схожість з існуючими тестами, оскільки, наскільки нам відомо, навчальний набір для моделей OpenAI включає лише проекти з GitHub, тому модель менш імовірно бачила існуючі тести під час навчання. Наші результати дійсно показують, що три з п'яти проектів мають максимальну подібність < 0,4, а решта два мають максимальну подібність 0,5. Це дає нам впевненість у тому, що метрика подібності, яку ми використовуємо, дає

значущі результати.

3.3 Оцінювання ефективності генеративного штучного інтелекту порівняно з ручним тестуванням

У цьому розділі представлені результати оцінки ефективності моделей генеративного штучного інтелекту (Generative AI) у генерації тестових кодів. Обрані моделі GenAI (Claude, Llama та Mistral) були оцінені на основі кількості збоїв виконання та покриття їх коду. З набору даних було видалено два тестові випадки, для яких як люди, так і LLM надали тестовий код, який не був виконуваним або був виконуваним з мінімальним покриттям коду. Остаточний набір даних складався з п'ятдесяти зразків тестових випадків. Відсоток збоїв виконання, а також середнє значення, медіана та стандартне відхилення вихідного коду, охопленого тестовим кодом, наведено для базового рівня, методів збагачених інструкцій, прикладів та ланцюга думок. Результати можна знайти в таблицях.

Таблиця 3.2

Статистика помилок виконання та покриття коду для моделей із базовим підказками у порівнянні з результатами роботи людини.

Стратегія / Модель	**Рівень складності**	**Claude**	**Llama**	**Mistral**	**Людина (Human)**
Baseline (Базовий)	Very Easy	91.63%	**97.83%**	87.42%	96.04%
	Hard	**29.15%**	28.70%	27.31%	55.40%
Rich Instructions	Very Easy	79.63%	**92.11%**	82.05%	96.04%
(Детальні інструкції)	Hard	30.05%	26.98%	**31.36%**	55.40%
Example (З прикладом)	Very Easy	81.08%	**87.14%**	84.93%	96.04%
	Hard	**28.70%**	27.56%	27.31%	55.40%
Chain-of-Thought	Very Easy	88.16%	88.57%	**89.35%**	96.04%
(Ланцюжок думок)	Hard	28.79%	27.03%	**29.93%**	55.40%

-Таблиця 3.2 підсумовано результати метрик невдач виконання та покриття коду для тестових кодів моделей LLM, написаних людиною та з базовим підказуванням. Як тестовий код людини, так і тестовий код моделі Клода не зазнавав збоїв виконання, тоді як тестові коди Лами та Містраля мали збої виконання, що спочатку свідчить про те, що модель Клода з базовим підказуванням є високопродуктивною моделлю.

Таблиця 3.3

Статистика помилок виконання та покриття коду для моделей із підказками на основі розширених інструкцій у порівнянні з результатами роботи людини.

Модель	Складність	Помилки виконання [%]	Середнє значення	Покриття коду [%] (Медіана)	Стандартне відхилення (SD)
Claude	Дуже легко	0.00	79.63	100.00	32.04
	Легко	0.00	79.94	82.13	18.32
	Середньо	0.00	57.50	51.02	26.53
	Важко	0.00	30.05	30.88	5.10
Llama	Дуже легко	0.00	92.11	100.00	12.84
	Легко	0.00	87.54	91.95	14.06
	Середньо	0.00	56.59	52.54	16.93
	Важко	0.00	26.98	29.09	8.66
Mistral	Дуже легко	5.00	82.05	100.00	32.04
	Легко	0.00	83.93	86.05	14.10
	Середньо	11.11	40.51	46.01	29.93
	Важко	0.00	31.36	30.48	4.78
Людина	Дуже легко	0.00	96.04	100.00	9.61
	Легко	0.00	86.53	100.00	18.71
	Середньо	0.00	81.97	79.59	11.96
	Важко	0.00	55.40	43.64	28.45
Модель	Складність	Помилки виконання [%]	Середнє значення	Покриття коду [%] (Медіана)	Стандартне відхилення (SD)

Що стосується покриття коду з базовим підказуванням, результати свідчать про те, що моделі Клода та Лами працювали краще, ніж модель Містраля. Тим не менш, жодна з моделей, здається, не згенерувала тестовий код, який би працював краще з вихідним кодом, ніж коди, написані людиною, для будь-якої з категорій коду, незалежно від складності.

Таблиця 3.3 порівнюються результати невдалого виконання та покриття коду для тестових кодів, згенерованих моделями LLM, написаними людиною, та моделями LLM з підказками, що містять багато інструкцій. З цим типом підказок , з точки зору невдалого виконання, моделі Claude та Llama працювали так само добре, як і програмісти-люди, без жодних невдач виконання, тоді як деякі тести, згенеровані Mistral, не виконувалися. Підказки з багатою інструкцією призвели до того, що модель Llama показала найкращі результати серед трьох моделей з точки зору відсотка покриття коду, за нею йшли Mistral та Claude. Однак жодна з них не показала кращих результатів, ніж програмісти. Як показали наші результати, хоча Claude, здається, показав найкращі результати з базовими підказками, він показав набагато гірші результати з підказками з багатою інструкцією. З іншого боку, модель Mistral та Claude. Однак жодна з них не показала кращих результатів, ніж програмісти. Як показали наші результати, хоча Claude, здається, показав найкращі результати з базовими підказками, він показав набагато гірші результати з підказками з багатою інструкцією. З іншого боку, модель Mistral, здається, виграла від того, що в її підказці було надано багато інструкцій, принаймні з точки зору покриття коду.

Таблиця 3.4

Статистика помилок виконання та покриття коду для моделей із прикладами підказки у порівнянні з результатами роботи людини.

Модель	**Складність**	**Помилки виконання [%]**	**Середнє значення**	**Покриття коду [%] (Медіана)**	**Стандартне відхилення (SD)**
Claude	Дуже легко	10.00	81.08	100.00	33.07
	Легко	0.00	84.41	90.98	16.28
	Середньо	11.11	46.04	52.94	25.36
	Важко	0.00	28.70	29.09	6.07
Llama	Дуже легко	5.00	87.14	100.00	25.53
	Легко	0.00	80.04	81.25	15.01
	Середньо	0.00	65.49	56.78	23.36
	Важко	0.00	27.56	25.00	6.19
Mistral	Дуже легко	0.00	84.93	100.00	23.78
	Легко	0.00	80.04	81.25	15.01
	Середньо	0.00	53.33	50.85	22.83
	Важко	0.00	27.31	29.09	7.53
Людина	Дуже легко	0.00	96.04	100.00	9.61

	Легко	0.00	86.53	100.00	18.71
	Середньо	0.00	81.97	79.59	11.96
	Важко	0.00	55.40	43.64	28.45

Таблиця 3.4 наведеній нижче таблиці наведено результати невдач виконання та покриття коду для тестових кодів моделей LLM, що запитуються людиною та прикладами. Серед моделей Mistral досяг найнижчого рівня невдач виконання, що дорівнює показнику у людей, – 0%, тоді як Llama та Claude зазнали деяких невдач. Хоча Claude чудово показав себе у створенні виконуваних тестів з базовими та розширеними інструкціями, коли в запиті додатково надавався приклад, деякі з згенерованих ним тестів не були виконуваними. Що стосується покриття коду, то, схоже, моделі Claude та Llama показали кращі результати, ніж Mistral; однак, знову ж таки, вони не показали кращих результатів, ніж код, запитуваний людиною.

Таблиця 3.5 порівнюються показники невдач виконання та покриття коду для тестових кодів, згенерованих моделями LLM, написаними людиною та моделями з підказками за допомогою ланцюга думок. Схоже, що всі моделі LLM зазнали труднощів зі створенням виконуваного коду з цим типом підказок, причому Claude, Mistral та Llama мали невдачі виконання для коду середньої складності. Незважаючи на високий рівень невдач виконання, Claude зміг згенерувати тестові коди з високим покриттям у категоріях дуже легкої та легкої складності. Дві інші моделі, хоча також мали високі показники невдач виконання для коду середньої складності, не так добре працювали в простому або складному коді. Як і у випадку з попередніми методами підказок, моделі з підказками за допомогою ланцюга думок також не працювали краще, ніж програмісти.

Як повідомлялося, частка збоїв виконання в різних моделях ШІ, методах підказок та труднощах коливалася від 0% до 44% тестових випадків.

Таблиця 3.5

Статистика помилок виконання та покриття коду для моделей із підказками на основі ланцюжка міркувань у порівнянні з результатами роботи людини

Модель	Складність	Помилки виконання [%]	Середнє значення	Покриття коду [%] (Медіана)	Стандартне відхилення (SD)
Claude	Дуже легко	0.00	88.16	100.00	22.78
	Легко	0.00	80.53	90.44	21.78
	Середньо	**44.44**	37.58	38.77	40.28
	Важко	0.00	28.79	30.48	7.41
Llama	Дуже легко	0.00	88.57	100.00	22.07
	Легко	0.00	73.71	75.26	18.03
	Середньо	11.11	45.58	48.85	20.79
	Важко	0.00	27.03	29.09	7.14
Mistral	Дуже легко	0.00	89.35	100.00	18.55
	Легко	0.00	80.04	85.65	18.77
	Середньо	22.22	39.22	38.78	30.17
	Важко	0.00	29.93	29.09	5.47
Людина	Дуже легко	0.00	96.04	100.00	9.61
	Легко	0.00	86.53	100.00	18.71
	Середньо	0.00	81.97	79.59	11.96
	Важко	0.00	55.40	43.64	28.45

Ці збої могли бути наслідком неправильного імпорту або інших помилок у згенерованому коді, які могли б перешкодити виконанню коду у вихідному коді. Загальне припущення тестових кодів полягає в тому, що вони виконаються успішно; тому будь-які значення вище 0 вважаються високими. Таким чином, результати цього аналізу вказують на те, що тести, розроблені LLM, були відносно неефективними.

Якісні тести повинні тестувати 100% вихідного коду, щоб врахувати всі можливі тестові випадки, включаючи екстремальні. Хоча ні покриття людським кодом, ні покриття LLM-коду не досягло 100% для всіх рівнів складності, очевидно, що тести, написані людьми, були набагато ближчими до того, щоб вважатися якісними, ніж ті, що написані LLM. Медіана покриття коду моделі нижча за медіану людського коду майже для всіх випадків підказок та рівня складності. Одним з помітних моментів є розбіжність між середнім та медіанним значеннями - це значною мірою було спричинено окремими випадками, коли кілька вихідних кодів покривалися лише невеликим відсотком людського або штучного коду. Враховуючи відносно невеликий розмір вибірки, це значно знизило середні значення покриття коду моделями та людьми, а також вплинуло

на стандартне відхилення. Значення покриття коду в діапазоні від дуже низьких (як у згаданому тестовому випадку) до дуже високих (випадки покриття коду 100%) призвели до того, що стандартне відхилення покриття тестового коду як моделей, так і людей коливалося в діапазоні від 4,78 до 33,07, що є особливо високим показником для стандартного відхилення.

Ці результати не дають чіткої відповіді на питання, чи був певний метод чи модель підказок найуспішнішим у генерації тестового коду. Моделі відрізняються тим, як вони виконували поставлене завдання. Загалом, серед моделей, здається, що Claude з базовими підказками має найвищі значення покриття коду за всіма рівнями складності та 0 збоїв виконання. Llama демонструє подібну закономірність, при цьому базові підказки генерують коди з найвищим покриттям коду. На відміну від Claude та Llama, з точки зору медіани, Mistral найкраще показав результати з підказками з багатими інструкціями. Початкові припущення полягали в тому, що збільшення деталізації, наданої в підказці, призведе до створення кращих тестів моделями. Це не відображається в результатах, оскільки найпростіші методи підказок, базові та багаті інструкції, призвели до того, що LLM створювали тести з нижчими показниками збоїв виконання та вищим покриттям коду, ніж у більш детальному прикладі та в умовах підказок ланцюжка думок. Тим не менш, ці моделі, незалежно від методу підказок, не працювали так добре, як код, написаний людиною.

Середнє тестове покриття моделей GenAI демонструє тенденцію до зменшення зі зростанням складності. Тобто, як люди, так і LLM досягли високого успіху в написанні дуже простих та простих тестів, а їхня продуктивність дещо знизилася для тестів середньої складності та значно знизилася для тестів високої складності. Для кращої ілюстрації ці результати візуалізовано на рисунку.

На діаграмах візуалізовано продуктивність як тестів за участю людини, так і тестів з моделями для кожного з методів підказок. Основний висновок полягає в тому, що хоча LLM добре працюють з дуже простим і легким кодом - майже так само добре, як і люди, або в деяких випадках трохи краще, ніж люди - у випадку середнього та складного коду, тоді як люди працюють трохи гірше,

ніж LLM, з дуже простим або легким кодом, LLM працюють значно гірше. Це падіння продуктивності очевидне на основі випадків, використаних у цьому дослідженні, щодо відносно простих кодів, які вимагають лише певних знань предметної області та математики, макетів та реалізації.

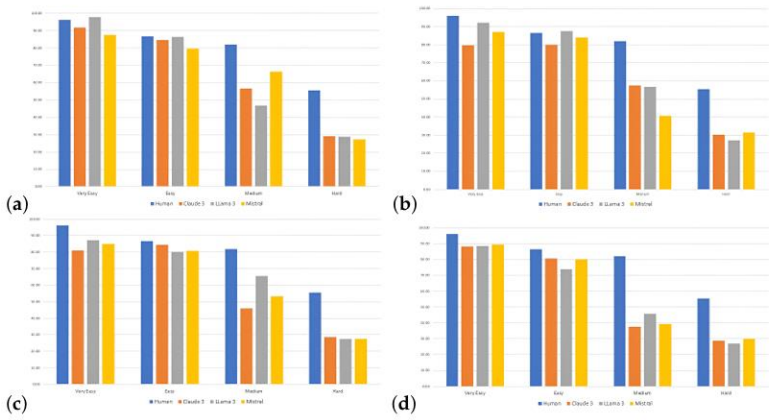

Рисунок 3.12 - Результати тестування людини та ШІ (середній рівень покриття коду) за категоріями складності тестів для різних стратегій формування підказок. (а) Базова підказка. (б) Підказка з розгорнутими інструкціями. (в) Підказка з прикладом. (г) Підказка з ланцюжком міркувань.

Можна очікувати, що продуктивність ще більше знизиться зі збільшенням складності написання тестів, що, ймовірно, буде характерно для більшості практичних застосувань. Наші результати свідчать про те, що хоча LLM виявляються ефективними інструментами для автоматизації генерації тестів для простих завдань, вони все ще мають труднощі з більш вимогливими сценаріями кодування, з якими люди справляються ефективніше, що підкреслює розрив між штучним інтелектом та людською експертизою в розробці програмного забезпечення.

Як доповнення до оцінки ефективності вибраних моделей GenAI окремо, ми також дослідили, наскільки схожими були згенеровані моделями GenAI тестові коди на написані людиною. Можна припустити, що написані людиною тестові коди виступають «золотим стандартом» для порівняння, і, як видно з попередніх таблиць, кількість збоїв виконання становила 0, а медіанне покриття коду було високим для написаного людиною коду, що свідчить про високу якість

коду. Крім того, було б корисно дослідити подібність між самими згенерованими кодами.

Щоб встановити, чи генеративний ШІ дає загалом подібні результати, незалежно від моделі, чи існують суттєві відмінності між генерацією коду моделями, тестові коди моделей GenAI та тестові коди, написані людиною, порівнювалися шляхом спочатку перетворення тестових кодів у семантичні представлення, потім перетворення їх у вбудовування та, нарешті, обчислення метрик подібності вбудовування: евклідова відстань, косинусна подібність та скалярний добуток. Вони були розраховані шляхом вимірювання індивідуальної подібності коду для тестових кодів моделі та написаних людиною для кожного випадку вихідного коду та для кожного методу підказки окремо. Потім ці значення були усереднені для всіх випадків вихідного коду для кожного методу підказки, щоб отримати кінцеві середні показники подібності вбудовування. Результати цього порівняння можна побачити в таблицях.

Для інтерпретації цих результатів необхідно враховувати, що подібність вбудовування зростає зі зменшенням евклідової відстані (більша відстань означає меншу подібність) та зростає зі збільшенням косинусної подібності та скалярного добутку (вищі значення метрики означають вищу подібність).

Таблиця 3.6 представлені результати подібності вбудовування тестів, згенерованих базовими моделями, з результатами один одного та з результатами, написаними людиною. Аналізуючи метрики евклідової відстані, косинусної подібності та скалярного добутку, очевидно, що Клод, Лама та Моделі Mistral створили тести, схожі один на одного, особливо тести Llama та Mistral з найменшою відстанню 8,19. Подібність Claude-Llama та Claude-Mistral була високою та майже рівною - наприклад, порівнюючи косинусне подібність, подібність становила 0,79 та 0,80. З іншого боку, тести моделей, здається, всі однаково відрізнялися від тестів, написаних людьми. Серед них тести Claude були найбільш схожими на тести, написані людьми. Таким чином, можна сказати, що з базовими підказками тести моделей, здається, були згенеровані дуже схожим чином, але разюче відмінно від тестів, написаних людьми. Claude

створив найбільш схожі тести з тестами, написаними людьми, а Llama згенерував тести, які були найменш схожі на тести програмістів.

Таблиця 3.6

Середні показники схожості вбудованих векторів тестового коду, написаного моделями з базовими підказками та людьми: евклідова відстань, косинусна схожість та скалярний добуток

	Claude	**Llama**	**Mistral**	**Людина**
Claude	—	8.40, 0.80, 150.80	8.50, 0.79, 148.64	8.80, 0.77, 136.41
Llama	—	—	8.19, 0.82, 156.49	9.33, 0.75, 139.83
Mistral	—	—	—	9.07, 0.76, 138.47
Людина	—	—	—	—

Таблиця 3.7 підсумовано результати вбудовування подібності в моделях LLM з підказками, виконаними багатим набором інструкцій, та тестах програмістів. Подібно до результатів базового підказування, тести моделей Клода та Містраля більш схожі на тести інших моделей LLM, ніж ті, що були написані людиною. Порівнюючи одну з метрик подібності, евклідову відстань, можна побачити, що подібність Клода-Людина та Містраля-Людина досягає високих значень, тоді як відстані Клода-LLM та Містраля-LLM нижчі. Тим не менш, можна спостерігати різницю у згенерованих тестах для Llama, де з підказками, виконаними багатим набором інструкцій, були згенеровані тести, які були дуже схожими на тести, виконані людиною, і не так сильно відрізнялися від тестів інших моделей

Таблиця 3.7

Середні показники схожості вбудованих моделей тестового коду, написаного моделями з використанням підказок із розгорнутими інструкціями та людьми: евклідова відстань, косинусна схожість та скалярний добуток.

	Claude	**Llama**	**Mistral**	**Людина**

Claude	—	8.57, 0.78, 141.60	8.99, 0.76, 139.15	9.00, 0.76, 133.62
Llama	—	—	8.30, 0.80, 147.58	8.32, 0.80, 143.36
Mistral	—	—	—	8.65, 0.78, 142.82
Людина	—	—	—	—

Таблиця 3.8 порівнюється подібність між тестами моделей програмістів та моделей, що використовують підказки на прикладах. Зрозуміло, що за допомогою цього методу підказок згенеровані тести були більш несхожими на тести, згенеровані за допомогою інших методів підказок, а також на тести, написані людиною, причому евклідова відстань знаходилася в діапазоні від 8,99 до 10,34, що значно вище, ніж значення подібності для базового методу та методу з розширеними інструкціями, які коливалися від 8,19 до 9,33 та від 8,30 до 9,00 відповідно. Можна зробити висновок, що всупереч логіці того, що LLM створюватимуть код, подібний до людського, після отримання тесту, написаного людиною, здається, що це не було правдою для жодної з трьох моделей. Стіл На рисунку 3.9 наведено результати дослідження подібності вбудовування тестових кодів, згенерованих за допомогою LLM, написаних людиною, та кодів, що генеруються за допомогою ланцюга думок. На відміну від інших методів підказок , підказки за допомогою ланцюга думок призвели до того, що моделі LLM створили більше тестів, подібних до тестів, створених людьми, ніж до інших LLM у випадку порівнянь Клода-Ллами та Клода-Містраля.

Таблиця 3.8

Середні показники схожості вбудованих векторів тестового коду, написаного моделями на основі прикладів підказок та людьми: евклідова відстань, косинусна схожість та скалярний добуток.

	Claude	**Llama**	**Mistral**	**Людина**
Claude	—	9.48, 0.73, 129.88	10.34, 0.70, 126.98	9.30, 0.75, 131.36
Llama	—	—	8.99, 0.77, 145.90	9.23, 0.76, 138.43
Mistral	—	—	—	9.44, 0.75, 139.26

Людина	—	—	—	—

Це свідчить про те, що ланцюг думок є ідеальним методом підказок для створення семантично подібних тестів для людей. Тим не менш, як видно з попередніх порівнянь невдач виконання та покриття коду, жодна з моделей не змогла згенерувати код такий же хороший, як код, написаний людиною. Таким чином, можна зробити висновок, що сама по собі схожість коду на код, написаний людиною, не гарантує кращої якості чи функціональності

По-перше, під час аналізу подібності між згенерованими та написаними людиною тестовими кодами, евклідова відстань між моделями та методами підказок коливалася від 8,32 до 9,48. Ця різниця особливо очевидна у випадку методу підказок з багатими інструкціями, де значення для всіх моделей були відносно низькими, що вказує на високу подібність. Для моделей Llama та Mistral значення, пов'язані з цим методом підказок, були найнижчими, що свідчить про те, що ці моделі краще виконують завдання, коли їм надаються багаті інструкції, більше, ніж, наприклад, з ланцюжком думок або базовим підказуванням.

Таблиця 3.9

Середні показники схожості вбудованих векторів тестового коду, написаного моделями з підказками «ланцюжка думок» та людьми: евклідова відстань, косинусна схожість та скалярний добуток.

	Claude	**Llama**	**Mistral**	**Людина**
Claude	—	9.48, 0.73, 129.88	10.34, 0.70, 126.98	9.30, 0.75, 131.36
Llama	—	—	8.99, 0.77, 145.90	9.23, 0.76, 138.43
Mistral	—	—	—	9.44, 0.75, 139.26
Людина	—	—	—	—

Клод згенерував тестовий код, найбільш схожий на людський код, згідно з базовими інструкціями. Дві інші метрики, косинусної подібності та скалярного

добутку, пропонують подібні висновки, де найвища косинусної подібність була виміряна для моделей Llama та Mistral у методі підказок з багатими інструкціями, тоді як Клод однаково добре показав себе в методах з багатими інструкціями та базовому методі. Значення косинусної подібності коливалися від -1 до 1; таким чином, усі подібності вбудовування між моделями можна вважати відносно високими, що вказує на те, що тести, створені людьми та моделями, є досить схожими. Однак, слід враховувати, що тести, згенеровані LLM та написані людиною, спочатку були перетворені на кодові представлення, як детально описано в підрозділі «Подібність тестового коду GenAI та написаного людиною» розділу «Експерименти», що, ймовірно, призвело до зростання схожості тестів.

По-друге, під час аналізу подібності між самими згенерованими тестовими кодами, загалом згенеровані тестові коди були більш схожими один на одного, ніж на коди, написані людиною. Наприклад, враховуючи метод підказок ланцюжка думок, Таблиця На рис. 3.9 показано, що хоча значення подібності моделі та людини для евклідової відстані становили 9,48, 9,30 та 9,44, подібність моделі до моделі була нижчою, зі значеннями 7,75, 8,81 та 9,14. Це показує, як тип підказок вплинув на те, наскільки схожим був згенерований код на код, написаний людиною. Загалом, згенерований код був найбільш схожим на людський код для методу підказок з багатими інструкціями, за яким йшли базові підказки, підказки з прикладами та підказки з ланцюжком думок. Це вказує на те, що для досягнення мети створення тестів, семантично подібних до коду, написаного людиною, слід використовувати багаті інструкції, а не, наприклад, підказки з ланцюжком думок, які в цьому випадку показали найгірші результати.

Також необхідно враховувати той факт, що тести, згенеровані моделями ШІ, завжди писалися за допомогою unittest, тоді як людський код писався за допомогою операторів unittest, pytest або assert. Ці різні методи впливають на структуру коду, що, можливо, сприяло тому, що згенеровані коди були схожі один на одного, більше, ніж на людський код. Більше того, хоча генеративний ШІ надавав перевагу генерації кодів за чітким покроковим шаблоном (щоб окремо організувати призначення значень, акт запуску коду та присвоєння його результату та ствердження очікуваного результату протягом кількох рядків

коду), людський код часто був структурований ефективніше, де всі три дії можна було виконати в простому операторі лише в одному рядку коду. Ці два пункти дають певне уявлення про можливі причини, чому згенеровані коди були більш схожі один на одного, ніж на людський код. Однак це не дає чіткого уявлення про їхню якість.

Щоб оцінити статистичну значущість відмінностей у евклідових відстанях між результатами, згенерованими моделлю, та результатами, написаними людиною, ми провели тест знакових рангів Вілкоксона для різних стратегій підказок. Ми порівняли пари моделей з людськими еталонами за чотирма методами підказок. Результати показали, що за всіма методами підказок існувала значна різниця між більшістю пар, що вказує на те, що вибір Клода, Лами, Містраля або людського агента має значення, коли справа доходить до генерації тестових кодів. Однак суттєвих відмінностей для пар Клод-Людина та Лама-Людина не було виявлено за базових ($p = 0,77$) та ланцюжків думок ($p = 0,09$) інструкцій, що свідчать про те, що за цими інструкціями не має значення, хто вибраний – Клод чи Лама, а не людський агент. Одним з можливих пояснень може бути те, що і Клод, і Лама однаково узгоджені з людським мисленням за мінімальних або структурованих підказок. З іншого боку, для пар Клод-Людина та Містраль-Людина відсутність статистичної різниці спостерігається між методами підказок із багатою інструкцією ($p = 0,16$) та прикладом ($p = 0,37$), що свідчить про те, що з цими типами стилів підказок різниця між використанням Клода або Містраля замість людського агента не буде суттєвою. Причиною цього може бути те, що більш багаті або засновані на прикладах підказки допомагають стандартизувати структуру виводу в різних моделях, зменшуючи варіабельність. Нарешті, у парах Клод-Ллама та Клод-Містраль лише в підказці з ланцюжком думок мало значення, хто був обраний Лама чи Містраль замість Клода ($p = 0,25$), що можна пояснити підвищеним когнітивним навантаженням та глибиною міркування, необхідними для цього типу підказки, що потенційно посилює специфічні для моделі відмінності. Загалом, ці висновки підкреслюють, що вибір агента (чи то Клод, Лама, Містраль чи людина) відіграє першочергову роль у визначенні якості та узгодженості генерації тестового коду, тоді як стиль

підказки слугує вторинним фактором, який може модулювати притаманні відмінності між агентами.

Таблиця 3.10 представлені результати H-тесту Краскела-Уолліса, виконаного на евклідовій відстані між вбудовуваннями зразків, отриманих за допомогою різних методів підказок. Тест було проведено для того, щоб визначити, чи розподіл відстаней між вбудовуваннями суттєво змінюється залежно від комбінації задіяних агентів, включаючи моделі Клод, Лама, Містраль - та людей. Для кожного методу підказок - базового рівня, збагачених інструкцій, прикладу та ланцюга думок - відстані вбудовування були згруповані за парами моделей (наприклад, Клод-Лама, Клод-Людина), і ми провели непараметричний H-тест для оцінки статистичних відмінностей між цими групами. Для базового рівня, збагачених інструкцій та ланцюга думок багаторазові порівняння дали p-значення нижче порогу 0,05, що вказує на статистично значущу варіацію у відстанях вбудовування між комбінаціями моделей. На противагу цьому, приклад підказки виділявся як єдина умова, де не було виявлено значних відмінностей, що свідчить про те, що цей тип підказки призводить до більш узгоджених представлень вбудовування в різних моделях. Цю подібність можна пояснити характеристикою використаної техніки підказок - усі моделі з прикладом підказки дотримуються шаблону «влаштувати-діяти-ствердити». Найбільші розбіжності спостерігалися в порівняннях, що включали людські відповіді, які послідовно давали вищі значення H. Це свідчить про те, що згенеровані моделлю вбудовування помітніше відрізняються від вбудов, створених людиною, ніж одне від одного.

Таблиця 3.10

Значення p за тестом Вілкоксона для знакових рангів для евклідової відстані між парами модельного та написаного людиною коду.

Промпт (Prompt)	**Перша пара**	**Друга пара**	**p-Value**
Базовий (Baseline)	Claude–Людина	Llama–Людина	0.77
	Claude–Людина	Mistral–Людина	<0.001
	Llama–Людина	Mistral–Людина	<0.001
	Claude–Llama	Claude–Mistral	<0.001

Розширені інструкції (Rich Instructions)	Claude–Людина	Llama–Людина	< 0.001
	Claude–Людина	Mistral–Людина	0.16
	Llama–Людина	Mistral–Людина	<0.001
	Claude–Llama	Claude–Mistral	<0.001
Приклад (Example)	Claude–Людина	Llama–Людина	0.02
	Claude–Людина	Mistral–Людина	0.37
	Llama–Людина	Mistral–Людина	0.03
	Claude–Llama	Claude–Mistral	0.01
Ланцюжок думок (Chain-of-Thought)	Claude–Людина	Llama–Людина	0.09
	Claude–Людина	Mistral–Людина	0.01
	Llama–Людина	Mistral–Людина	<0.001
	Claude–Llama	Claude–Mistral	0.25

У цьому дослідженні досліджувалися можливості LLM у генерації тестових кодів та порівнювалися з продуктивністю людей. Основним висновком є явне зниження ефективності LLM зі збільшенням складності завдання. Для простих завдань генерації, що потребують невеликої експертизи в предметній області, LLM працювали порівнянно з людьми. Однак їхня продуктивність значно знизилася для складніших проблем, що підкреслює
розрив у продуктивності між GenAI та тестувальниками-людьми.

Важливо визнати, що це дослідження не моделювало реальних промислових сценаріїв. Вихідний код, який використовувався, був значною мірою адаптований з навчальних прикладів або непрофесійних внесків, а іноді навіть за допомогою штучного інтелекту. На практиці виробничий код, як правило, значно складніший, і LLM може не узагальнюватися для такої складності. Подальша робота повинна розширити ці висновки, оцінивши продуктивність GenAI на реальних кодових базах.

Таблиця 3.11

Результати H-критерію Крускала–Уолліса для вибірок щодо відстані між вбудовуваннями.

Промпт (Prompt)	**Вибірки з (Samples from)**	**Значення**	**p-Value**
Базовий	ШІ–ШІ (Claude, Llama, Mistral)	62.72	0.02
	ШІ–Людина	76.15	<0.001

	Усі вищезазначені	84.11	<0.001
Розширені інструкції	ШІ–ШІ (Claude, Llama, Mistral)	68.86	<0.001
	ШІ–Людина	52.04	0.22
	Усі вищезазначені	57.95	0.04
Приклад	ШІ–ШІ (Claude, Llama, Mistral)	33.34	0.12
	ШІ–Людина	50.87	0.25
	Усі вищезазначені	29.50	0.24
Ланцюжок думок	ШІ–ШІ (Claude, Llama, Mistral)	47.13	0.01
	ШІ–Людина	77.21	<0.001
	Усі вищезазначені	44.39	0.03

Одним із найперспективніших напрямків є інтеграція людського зворотного зв'язку як в оцінювання, так і в генерацію тестів. Розробники можуть оцінювати тести, згенеровані штучним інтелектом, на предмет коректності, покриття та виявлення помилок, особливо в граничних випадках, та ітеративно керувати моделями для покращення результатів. Цей підхід, заснований на зворотному зв'язку, узгоджується з новими практиками спільної розробки програмного забезпечення, де генерація коду та тестування утворюють тісний цикл зворотного зв'язку.

Хоча моделі, що використовуються в цьому дослідженні, не були точно налаштовані для конкретного завдання, вони дотримувалися правил кодування, ймовірно, через попереднє навчання на великомасштабних репозиторіях. Хоча це дозволяє їм імітувати поширені шаблони, це обмежує їхню ефективність для предметно-специфічної або незвичайної логіки.

На практиці, генерацію тестів GenAI можна вбудувати в середовища розробників за допомогою таких інструментів, як GitHub Copilot, надаючи не лише пропозиції коду, але й генерацію тестів у режимі реального часу. Це може зробити процес тестування більш гнучким та інтегрованим.

Зрештою, потрібні додаткові дослідження щодо інтеграції GenAI у реальні робочі процеси програмування, особливо для складних або критично важливих систем.

У цьому дослідженні оцінено потенціал генеративного штучного інтелекту, зокрема методів навчання з ліцензування (LLM), в автоматизації створення модульних тестів на Python. Python було обрано через його

популярність та широке використання в навчальних корпусах LLM, що означає, що генерація коду цією мовою повинна бути цілком у межах можливостей моделей. Тим не менш, наші результати виявляють чіткі обмеження.

Ми використовували найсучасніші базові моделі з Amazon Bedrock та досліджували різні підходи до розробки запитань. Якість та подібність тестових випадків, згенерованих штучним інтелектом, аналізували за допомогою вбудовування.

Наші результати показують, що LLM здатні створювати ефективні тести лише для дуже простих і нескладних випадків коду. Їхня продуктивність значно падає при зіткненні з проблемами середнього або складного рівня, як з точки зору успішності виконання, так і з точки зору покриття коду. Збільшення складності запитань не завжди покращує якість тестування, що свідчить про те, що більший контекст не гарантує кращих результатів.

Важливим висновком з нашого аналізу подібності на основі вбудовування є відсутність різноманітності в результатах тестів, згенерованих штучним інтелектом. Хоча ці тести структурно схожі один на одного, вони семантично відрізняються від тестів, написаних людиною, що обмежує їхню адаптивність у різних сценаріях тестування.

Таким чином, для підвищення практичної корисності LLM у тестуванні програмного забезпечення необхідні подальші вдосконалення у трьох основних сферах: точне налаштування моделей для завдань тестування, розробка більш цілеспрямованих підказок та вбудовування циклів зворотного зв'язку від людини. До того часу LLM слід розглядати як додатковий інструмент для базового прототипування, а не як окреме рішення для забезпечення якості у складних системах.

3.4 Висновки по розділу

У третьому розділі було розглянуто практичну розробку, інтеграцію та оцінку ефективності рішень автоматизації тестування на основі великих мовних моделей. Виконано проєктування та реалізацію інструменту автоматизованого

тестування з використанням штучного інтелекту, а також описано застосовані технології та архітектурні підходи. Проведено аналіз пілотного проєкту автоматизованого тестування, що дозволило оцінити практичну ефективність запропонованого рішення. Також виконано порівняльну оцінку ефективності генеративного ШІ та ручного тестування, яка підтвердила переваги використання LLM у підвищенні швидкості, якості та масштабованості процесів тестування програмного забезпечення.

ВИСНОВКИ

У ході виконання бакалаврської роботи було досліджено та реалізовано підхід до автоматизації тестування програмного забезпечення із використанням великих мовних моделей. Робота є комплексним процесом, спрямованим на підвищення ефективності, якості та швидкості тестування програмних систем у сучасних умовах розробки. У дослідженні проаналізовано існуючі підходи до тестування, визначено їхні переваги та обмеження, а також обґрунтовано доцільність використання інтелектуальних моделей для автоматизації ключових етапів тестового процесу.

На початковому етапі було проведено аналіз предметної області, що дозволило визначити основні функціональні та нефункціональні вимоги до системи автоматизації тестування. Особливу увагу приділено таким аспектам, як зменшення часу на створення тестів, підвищення їх покриття, забезпечення якості та адаптивність до змін у програмному коді. Моделювання бізнес-процесів тестування дало змогу структурувати взаємодію між компонентами системи та визначити ключові сценарії використання, включаючи генерацію тестових сценаріїв, аналіз коду та оцінку результатів тестування.

У рамках реалізації було розроблено прототип системи автоматизованого тестування, який використовує великі мовні моделі для генерації тест-кейсів,

створення тестових даних та часткової автоматизації аналізу результатів. Було забезпечено інтеграцію з сучасними інструментами розробки та тестування, що дозволило реалізувати повний цикл тестування — від формування вимог до виконання тестів і збору метрик. Значна увага приділялася обробці помилок, валідації результатів та забезпеченню стабільності роботи системи.

Проведене тестування включало модульні, інтеграційні та експериментальні дослідження, спрямовані на оцінку ефективності використання великих мовних моделей у тестуванні. У результаті було виявлено, що автоматизована генерація тестів дозволяє суттєво скоротити час їх створення та підвищити покриття програмного коду. Водночас визначено певні обмеження, зокрема залежність якості результатів від вхідних даних, можливість генерації некоректних або надлишкових тестів, а також потребу в додатковій валідації з боку розробника.

Загалом, отримані результати підтверджують доцільність використання великих мовних моделей для автоматизації тестування програмного забезпечення. Запропонований підхід дозволяє підвищити продуктивність процесу тестування, зменшити навантаження на розробників і тестувальників, а також покращити якість програмних продуктів. Основними перевагами є гнучкість, масштабованість та здатність адаптуватися до різних типів програмних систем.

Разом з тим, існують певні виклики, пов’язані з інтеграцією таких рішень у реальні проєкти, зокрема питання точності, контролю якості згенерованих тестів та безпеки використання штучного інтелекту. У подальшому можливе вдосконалення системи шляхом використання комбінованих підходів, інтеграції з системами безперервної інтеграції та доставки (CI/CD), а також розширення функціональності за рахунок глибшого аналізу коду та поведінки програм.

Таким чином, розроблена система та проведене дослідження демонструють перспективність використання великих мовних моделей у сфері автоматизації тестування та відкривають нові можливості для підвищення ефективності розробки програмного забезпечення.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. OpenAI. (2025). GPT Models Documentation. https://platform.openai.com/docs
2. Microsoft. (2025). Azure OpenAI Service Documentation. https://learn.microsoft.com/en-us/azure/ai-services/openai/
3. Google. (2025). Generative AI Documentation. https://cloud.google.com/ai/generative-ai/docs
4. Anthropic. (2025). Claude AI Documentation. https://docs.anthropic.com/
5. Hugging Face. (2025). Transformers Documentation. https://huggingface.co/docs/transformers
6. Vaswani, A., et al. (2017). Attention Is All You Need. https://arxiv.org/abs/1706.03762
7. Brown, T. B., et al. (2020). Language Models are Few-Shot Learners. https://arxiv.org/abs/2005.14165
8. Chen, M., et al. (2021). Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374
9. OpenAI. (2023). GPT-4 Technical Report. https://arxiv.org/abs/2303.08774
10. Li, Y., et al. (2023). Code Generation with Large Language Models: A Survey. https://arxiv.org/abs/2308.12950
11. IEEE. (2024). AI in Software Testing: Trends and Applications. https://ieeexplore.ieee.org/
12. ACM. (2024). Automated Software Testing using AI Techniques. https://dl.acm.org/
13. Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing. Wiley.
14. Ammann, P., & Offutt, J. (2016). Introduction to Software Testing. Cambridge University Press.
15. ISTQB. (2024). Software Testing Foundation Level Syllabus. https://www.istqb.org/

16. Selenium. (2025). Documentation. https://www.selenium.dev/documentation/

17. Cypress. (2025). End-to-End Testing Guide. https://docs.cypress.io/

18. Playwright. (2025). Testing Framework Documentation. https://playwright.dev/docs/intro

19. Postman. (2025). API Testing Guide. https://learning.postman.com/docs/

20. REST Assured. (2025). API Testing Documentation. https://rest-assured.io/

21. Microsoft. (2025). .NET Testing Documentation. https://learn.microsoft.com/en-us/dotnet/core/testing/

22. NUnit. (2025). Unit Testing Framework. https://nunit.org/

23. xUnit. (2025). Testing Framework Documentation. https://xunit.net/

24. Jenkins. (2025). CI/CD Automation Server. https://www.jenkins.io/doc/

25. GitHub. (2025). GitHub Actions Documentation. https://docs.github.com/en/actions

26. Fowler, M. (2018). Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html

27. Humble, J., & Farley, D. (2010). Continuous Delivery. Addison-Wesley.

28. Rahman, M. M., et al. (2023). AI-Based Test Case Generation: A Systematic Review. IEEE Access.

29. Panichella, A., et al. (2024). Machine Learning for Software Testing. Springer.

30. Shahin, M., et al. (2022). AI for DevOps: A Systematic Review. Journal of Systems and Software.

31. Kumar, S., & Singh, R. (2024). Automated Test Generation using NLP Techniques. Elsevier.

32. Wang, J., et al. (2023). Deep Learning for Software Engineering. ACM Computing Surveys.

33. OWASP. (2024). Top Ten Security Risks. https://owasp.org/www-project-top-ten/

34. Microsoft. (2025). Azure DevOps Pipelines. https://learn.microsoft.com/en-us/azure/devops/pipelines/

35. Google. (2025). Vertex AI for Developers. https://cloud.google.com/vertex-ai/docs

36. IBM. (2024). AI in Software Testing. https://www.ibm.com/topics/software-testing

37. Deloitte. (2025). AI-Driven Software Quality Engineering. https://www2.deloitte.com/

38. Gartner. (2024). Trends in AI for Software Development. https://www.gartner.com/

39. McKinsey. (2024). The Impact of Generative AI on Software Engineering. https://www.mckinsey.com/

40. Zhang, H., et al. (2024). Large Language Models for Test Automation: Opportunities and Challenges. IEEE Software.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Автоматизація тестування за допомогою великих мовних моделей"

Обсяг пояснювальної записки: 92 аркушів

Дата закінчення роботи 10 червня 2026р.

Підпис студента _____