Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра прикладної математики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Підходи до використання моделей LLM
у процесі тестування програмного забезпечення
(тема)

Виконав:
здобувач _2_ року навчання, групи _САУм-23-2
Богатов В.О.
(прізвище, ініціали)
Спеціальність
124 Системний аналіз
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма
Системний аналіз і управління
(повна назва освітньої програми)
Керівник доц. Поляков А.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ПМ ____________ Сидоров М.В.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет _інформаційно-аналітичних технологій та менеджменту
Кафедра _прикладної математики
Рівень вищої освіти _другий (магістерський)
Спеціальність _124 Системний аналіз
(код і повна назва)
Тип програми _освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма _Системний аналіз і управління
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри ПМ ____________
(підпис)
“_25_” _листопада_ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві _Богатову Віталію Олеговичу
(прізвище, ім’я, по батькові)

1. Тема роботи Підходи до використання моделей LLM у процесі програмного забезпечення

затверджена наказом по університету від 22 листопада 2024 р. № 1228 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 06 січня 2025 р.

3. Вихідні дані до роботи набір вихідних даних буде отримано від розробленої системи автоматизованої генерації, оцінки якості та формування аналітичних звітів

4. Перелік питань, що потрібно опрацювати в роботі
   1. Системний аналіз предметної області
   2. Вибір і обґрунтування методу розв’язання
   3. Програмна реалізація
   4. Результати обчислювального експерименту
   5. Аналіз можливих застосувань

5. Перелік графічного матеріалу із зазначенням кресленників, схем, плакатів, комп'ютерних ілюстрацій

1. Актуальність теми роботи
2. Постановка задачі
3. Системний аналіз предметної області
4. Метод чисельного аналізу
5. Результати обчислювального експерименту

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Підбір та вивчення технічної літератури за темою роботи	25 листопада – 1 грудня 2024 р.	виконано
2	Вибір та обґрунтування методу	2 – 8 грудня 2024 р.	виконано
3	Розробка алгоритму і програми	9 – 22 грудня 2023 р.	виконано
4	Проведення аналітичних досліджень та розрахунків	23 – 29 грудня 2024 р.	виконано
5	Робота над текстом пояснювальної записки	30 грудня 2024 р. – 9 січня 2025 р.	виконано
6	Представлення роботи на рецензію в ЕК	10 січня 2025 р.	виконано

Дата видачі завдання _25_ листопада 2024 р.

Здобувач ____________________
(підпис)

Керівник роботи ______________ доц. Поляков А.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 95 с., 19 табл., 18 рис., 2 дод., 41 джерел.

ВЕЛИКІ МОВНІ МОДЕЛІ, МОДУЛЬНЕ ТЕСТУВАННЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ТЕСТУВАННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

Об'єкт дослідження – процеси тестування мікросервісної архітектури.

Мета роботи – розробка та обґрунтування математичної моделі оцінки ефективності використання генеративного штучного інтелекту для оптимізації процесів тестування при переході від монолітної до мікросервісної архітектури.

Методи дослідження – системний аналіз, математичне моделювання, статистичний аналіз, порівняльний аналіз та методи оцінки ефективності процесів тестування програмного забезпечення.

У роботі проведено детальний аналіз особливостей тестування програмного забезпечення в монолітній та мікросервісній архітектурах. Досліджено можливості застосування великих мовних моделей для автоматизації процесів тестування. Розроблено комплексну систему метрик для оцінки якості згенерованих модульних тестів, що включає: базові метрики (кількість тестів, щільність перевірок), метрики покриття коду (покриття методів, граничних випадків, граничних значень), показники якості (описовість назв, різноманітність тверджень) та читабельності (використання документації, структурованість коду).

Створено програмну систему для порівняльного аналізу ефективності різних LLM моделей (ChatGPT, Gemini, CodeLlama) у задачі генерації Unit-тестів. Система забезпечує автоматизовану генерацію тестів, оцінку їх якості та формування аналітичних звітів. Проведено експериментальне дослідження на наборі синтетичних тестів, що охоплюють різні аспекти розробки: від простих алгоритмічних задач до складних асинхронних операцій та управління станами.

За результатами дослідження визначено оптимальні стратегії формування промптів для різних типів задач тестування. Виявлено, що найбільш збалансовані результати забезпечує промпт Prompt_DEV_Detailed_Focused з емуляцією ролі розробника та чіткими вказівками щодо застосування патернів тестування. При цьому CodeLLama показала найвищу сумарну оцінку, згенерувавши найбільшу кількість якісних тестів з високим покриттям граничних випадків.

Практична цінність роботи полягає у створенні методики ефективного використання LLM для автоматизації процесів тестування програмного забезпечення. Розроблені підходи та програмна система можуть бути застосовані для підвищення якості та швидкості розробки тестів у проектах з мікросервісною архітектурою.

ABSTRACT

Introductory note: 95 pages, 19 tables, 18 figures, 2 appendixes, 41 sources.

LARGE LANGUAGE MODELS, UNIT TESTING, SOFTWARE, TESTING, ARTIFICIAL INTELLIGENCE, SOFTWARE QUALITY.

Object of research – testing processes in microservices architecture.

Purpose of work – development and validation of a mathematical model for evaluating the effectiveness of generative artificial intelligence in optimizing testing processes during the transition from monolithic to microservices architecture.

Methods of research – systems analysis, mathematical modeling, statistical analysis, comparative analysis, and software testing efficiency evaluation methods.

The research presents a detailed analysis of software testing characteristics in monolithic and microservices architectures. The potential of large language models for test automation has been investigated. A comprehensive metric system for evaluating generated unit test quality has been developed, including: basic metrics (test count, assertion density), code coverage metrics (method coverage, edge cases, boundary values), quality indicators (naming descriptiveness, assertion variety), and readability measures (documentation usage, code structure).

A software system has been created to comparatively analyze the effectiveness of different LLM models (ChatGPT, Gemini, CodeLlama) in generating unit tests. The system provides automated test generation, quality assessment, and analytical reporting. Experimental research was conducted on synthetic tests covering various development aspects, from simple algorithmic tasks to complex asynchronous operations and state management.

The research identified optimal prompt strategies for different testing scenarios. The Prompt_DEV_Detailed_Focused approach, emulating a developer's role with clear testing pattern guidelines, proved most effective. CodeLLama achieved the highest total score, generating the largest number of quality tests with comprehensive

edge case coverage.

The practical value lies in establishing a methodology for effective LLM utilization in software testing automation. The developed approaches and software system can be applied to improve test development quality and speed in microservices architecture projects.

ЗМІСТ

С.

Перелік скорочень, умовних познак, одиниць і термінів 10

Вступ 11

1 Системний аналіз предметної області та постановка задач дослідження 13

1.1 Системний аналіз задачі оптимізації процесів тестування програмного забезпечення з використанням моделей LLM 13

1.2 Аналіз сценаріїв вирішення задачі оптимізації процесів тестування програмного забезпечення з використанням моделей LLM 25

1.3 Змістовна та формальна постановка задачі 27

1.4 Постановка задач дослідження 27

2 Вибір та обґрунтування методу розв’язання 29

2.1 Аналіз особливостей модульного тестування та його ролі у забезпеченні якості програмного забезпечення 29

2.2 Дослідження можливостей застосування генеративного штучного інтелекту в процесах тестування 31

2.3 Обґрунтування вибору LLM моделей 36

2.4 Критерії оцінювання якості Unit-тестів 38

Висновки за розділом 2 46

3 Програмна реалізація 48

3.1 Обґрунтування вибору мови Kotlin для дослідження використання LLM 48

3.2 Обґрунтування використання мови Python для аналізу даних 49

3.3 Обґрунтування вибору синтетичних тестів 50

3.4 Обґрунтування вибору налаштувань LLM 52

3.5 Алгоритм розв'язання задачі 54

3.6 Опис програми 55

Висновки за розділом 3 56

4 Результати обчислювального експерименту та їх аналіз 58

4.1 Аналіз результатів генерації Unit-тестів при різних налаштуваннях великих мовних моделей 58
4.2 Аналіз результатів синтетичних тестів 65
Висновки за розділом 4 72
Висновки 73
Перелік джерел посилання 75
Додаток А Лістинг програми 80
Додаток Б Діаграми порівняльного аналізу використання різних параметрів для генерації Unit-тестів 93

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАК, ОДИНИЦЬ І ТЕРМІНІВ

ПЗ - програмне забезпечення;

ШІ - штучний інтелект;

AI – artificial intelligence (штучний інтелект);

API – application programming interface (програмний інтерфейс додатку);

BC – branch coverage (покриття гілок);

BDD – behavior-driven development (розробка через поведінку);

BTC – boundary tests Coverage(покриття граничних умов) ;

CC – code coverage (покриття рядків коду);

CI/CD - continuous integration/continuous delivery (безперервна інтеграція/безперервне розгортання) ;

CoC – condition coverage (покриття умов) ;

DevOps – development and operations (методологія активної взаємодії спеціалістів з розробки та експлуатації);

E2E – end-to-end (наскрізне тестування від початку до кінця);

ECC – edge cases coverage (покриття граничних випадків) ;

GPT – generative pre-trained transformer (генеративний попередньо навчений трансформер);

LLM – large language models (великі мовні моделі);

MC – methods coverage (покриття методів) ;

PC – path coverage (покриття шляхів);

SC – statements coverage (покриття операторів);

UAT – user acceptance testing (тестування прийняття користувачем) ;

QA – quality assurance (забезпечення якості) ;

QM – quality metrics (оцінки якості).

ВСТУП

Актуальність теми. Стрімкий перехід індустрії розробки програмного забезпечення від монолітної до мікросервісної архітектури створює нові виклики в процесах тестування. Зростаюча складність та розподіленість мікросервісних систем вимагає переосмислення традиційних підходів до тестування та пошуку нових інструментів для підвищення його ефективності, одним з яких є генеративний штучний інтелект.

Розвиток технологій обробки природної мови, зокрема поява великих мовних моделей (Large Language Models, LLM), відкриває нові можливості для оптимізації процесів розробки програмного забезпечення. Особливо перспективним напрямком є застосування LLM для автоматизації та підвищення ефективності тестування, що є критично важливим етапом життєвого циклу програмного забезпечення. Дослідження показують, що використання LLM дозволяє скоротити час на створення тестів, підвищити покриття вимог та збільшити кількість виявлених дефектів. Це обумовлює високу актуальність розробки нових підходів та інструментів тестування програмного забезпечення на основі LLM.

Мета і завдання кваліфікаційної роботи. Метою кваліфікаційної роботи є розробка та обґрунтування підходів до використання моделей LLM для оптимізації процесів тестування програмного забезпечення. Для досягнення поставленої мети необхідно виконати наступні завдання:

– провести огляд і аналіз сучасного стану задачі тестування програмного забезпечення;

– дослідити можливості застосування генеративного штучного інтелекту в процесах тестування;

– розробити та верифікувати підходи до використання LLM для генерації тестів.

Результати роботи мають практичну цінність для підвищення ефективності процесів тестування програмного забезпечення. Розроблені підходи до використання LLM можуть бути впроваджені в процеси розробки для авто-

матизації генерації модульних тестів та оптимізації. Створена система метрик та методика оцінки якості згенерованих тестів дозволяє об'єктивно оцінювати ефективність різних LLM моделей та обирати оптимальні стратегії їх застосування для конкретних задач тестування. Результати дослідження також можуть бути використані в освітньому процесі при підготовці спеціалістів з тестування програмного забезпечення та для подальших наукових досліджень у напрямку вдосконалення методів автоматизації тестування з використанням штучного інтелекту.

Об'єктом дослідження є процеси тестування мікросервісної архітектури.

Предметом дослідження є методи та засоби підвищення ефективності процесів тестування при переході до мікросервісної архітектури з використанням генеративного штучного інтелекту.

Методи дослідження. У роботі використовуються методи системного аналізу, математичного моделювання, статистичного аналізу, порівняльного аналізу та методи оцінки ефективності процесів тестування програмного забезпечення.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Системний аналіз задачі оптимізації процесів тестування програмного забезпечення з використанням моделей LLM

Історичний розвиток методів забезпечення якості програмного забезпечення тісно пов'язаний зі зміною парадигм розробки та зростанням складності програмних систем. Початковий етап розвитку індустрії програмного забезпечення характеризувався відсутністю формалізованих підходів до тестування, коли перевірка якості здійснювалася здебільшого через налагодження програм розробниками [17]. З появою складних програмних систем та зростанням їх критичності для бізнес-процесів виникла потреба у створенні спеціалізованих методологій тестування.

Становлення процесів забезпечення якості програмного забезпечення як окремої дисципліни відбувалося паралельно з розвитком методологій розробки. Каскадна модель розробки, що домінувала протягом тривалого часу, розглядала тестування як окрему фазу життєвого циклу, яка слідувала за етапом розробки [19]. Такий підхід, хоча й забезпечував структурованість процесу, мав суттєві недоліки, пов'язані з пізнім виявленням дефектів та високою вартістю їх виправлення.

Значним кроком у розвитку практик забезпечення якості стало впровадження концепції тестування програмного забезпечення як безперервного процесу, інтегрованого в усі етапи розробки. Ця трансформація була зумовлена переходом до гнучких методологій розробки, які наголошують на важливості раннього та постійного зворотного зв'язку щодо якості продукту [18]. Зміна парадигми призвела до появи нових практик, таких як test-driven development (TDD) та behavior-driven development (BDD), які передбачають написання тестів перед реалізацією функціональності.

Еволюція підходів до забезпечення якості також відображає зміну розуміння самої природи дефектів програмного забезпечення. Якщо спочатку осно-

вна увага приділялася виявленню та виправленню функціональних помилок, то з часом фокус розширився на забезпечення нефункціональних вимог, таких як продуктивність, безпека та зручність використання [20]. Це призвело до появи спеціалізованих видів тестування та відповідних методологій їх проведення.

Сучасний світ розробки програмного забезпечення перебуває у стані постійної еволюції, рушійною силою якої є прагнення до підвищення ефективності, гнучкості та масштабованості систем [8]. Для забезпечення якості тестування використовуються різні підходи, включаючи:

– критичні процеси тестування - набір практик та процедур, спрямованих на виявлення та запобігання критичних дефектів на ранніх етапах розробки [19];

– test-driven development (TDD) - методологія розробки програмного забезпечення, яка спирається на повторення коротких циклів розробки, що починаються з написання тестів [17];

– domain-driven design (DDD) - підхід до розробки складних систем, що базується на глибокому розумінні предметної області та її відображенні в архітектурі програмного забезпечення [16].

В умовах зростаючої складності програмних продуктів та динамічних вимог ринку, традиційна монолітна архітектура все частіше поступається місцем мікросервісному підходу [9; 10]. Цей перехід, хоч і обіцяє низку переваг, водночас ставить перед розробниками нові виклики, особливо в сфері тестування та забезпечення якості програмного забезпечення.

Монолітна архітектура презентує класичний підхід до створення програмного забезпечення, де розробка додатку ведеться як єдиного цілого [1]. В рамках такого підходу життєвий цикл розробки програмного забезпечення традиційно слідує каскадній моделі, що суттєво впливає на процеси забезпечення якості. Характерною особливістю такого життєвого циклу є відокремлення фази тестування від процесу розробки, що призводить до суттєвого зростання витрат на виправлення дефектів. Дослідження показують [19], що вартість виправлення помилки експоненційно зростає на кожному наступному етапі життєвого циклу, досягаючи максимуму при виявленні дефектів на етапі промисло-

вої експлуатації. Представлення монолітної архітектури розробки ПЗ [33] наведено на рис. 1.1.

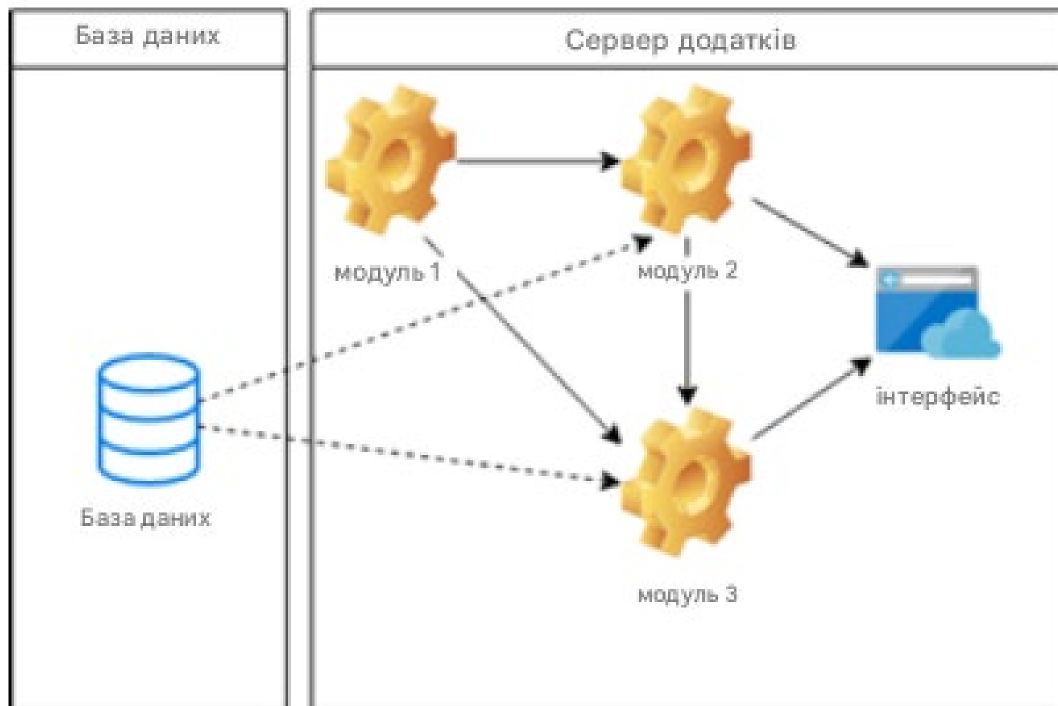

Рисунок 1.1 – Приклад монолітної архітектури розробки ПЗ

Життєвий цикл монолітної розробки чітко розділений на послідовні фази: аналіз вимог, проектування, реалізація, тестування та впровадження. На кожній фазі команда тестування має свої специфічні завдання. На етапі аналізу вимог проводиться їх валідація та верифікація. Під час проектування розробляється стратегія тестування та плануються необхідні ресурси. Етап реалізації супроводжується створенням тестових сценаріїв. Основне тестування починається тільки після завершення розробки, що створює значний розрив між написанням коду та виявленням дефектів. Така послідовність етапів, хоча і дозволяє чітко планувати ресурси, але призводить до пізнього виявлення критичних проблем у дизайні та архітектурі системи. Життєвий цикл розробки у монолітній архітектурі та етапи тестування [34] наведено на рис. 1.2.

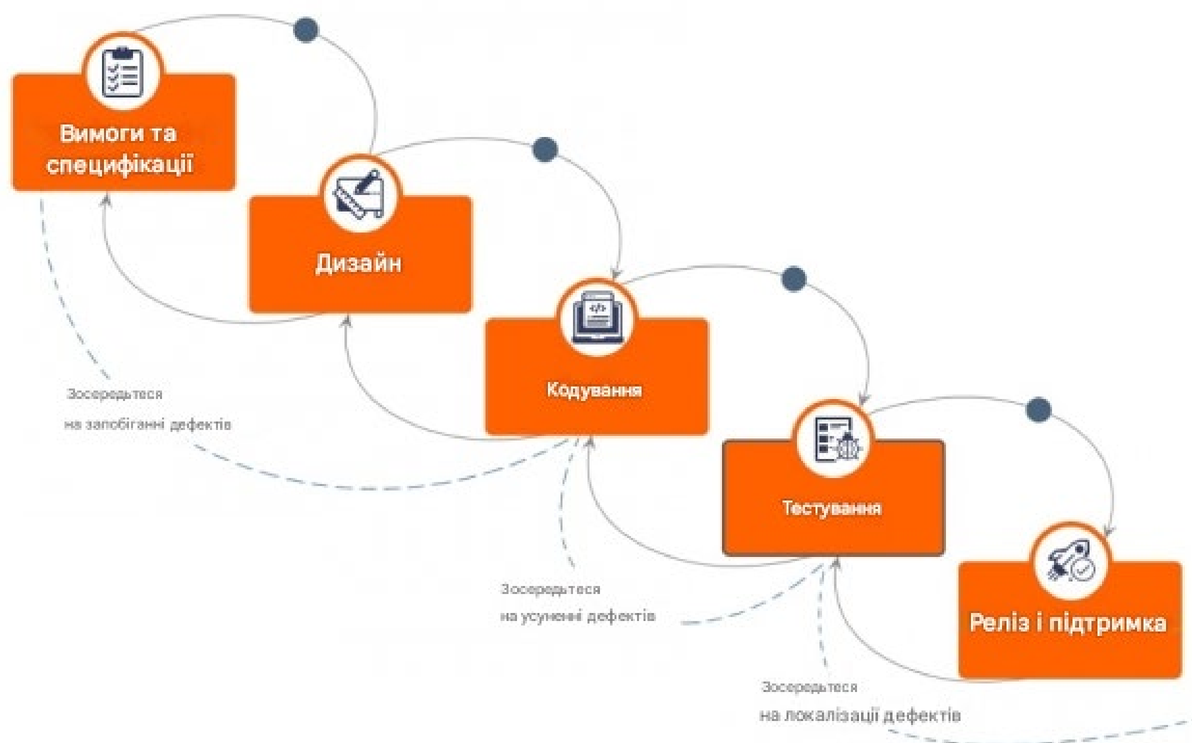

Рисунок 1.2 – Життєвий цикл розробки у монолітній архітектурі

Під час такої розробки всі компоненти та підсистеми функціонують в рамках однієї системи, тісно пов'язані між собою та мають загальну кодову базу [1]. Цей підхід розробки програмного забезпечення довгий час переважав над іншими через легкість розробки та керування процесами, спрощений процес відлагоджування. Процеси тестування в такій архітектурі характеризуються тяжінням до інтеграційних та end-to-end тестів замість модульного тестування [17; 18]. Всі тестові сценарії виконуються послідовно в одному середовищі розробки, що значно спрощує відтворення дефектів та відлагодження [19; 20]. Монолітні додатки можуть бути дуже гарно перевірені за допомогою традиційних інструментів, при цьому забезпечуючи достатній рівень покриття коду тестами та високу якість випускаємого додатку.

Також на популяризацію монолітної архітектури сильно впливали організаційні особливості. Під час розробки за каскадною методологію використовуються централізовані команди. Управління відбувається набагато ефективніше, а процес розгортання та моніторингу в таких системах значно простіше порівняно з розподіленими системами. Однак така централізація створює суттєві об-

меження для масштабування команд та паралельної розробки.

Проте у монолітній архітектурі проявляються значні обмеження через ріст складності розроблюваних програм. Масштабування монолітних додатків можливе тільки вертикально, що зумовлює значні фінансові та технічні обмеження. Залежність та зв’язок компонентів значно ускладнює внесення будь-яких змін та призводить до ризиків під час розгортання нових версій системи. Також суттєвим обмеженням стає неможливість використання різних технологічних комплексів у монолітних додатках для оптимального вирішення специфічних задач.

У контексті монолітної архітектури організаційна структура тестувальників традиційно формується за централізованим принципом, де група забезпечення якості (QA team) функціонує як окремий підрозділ [19]. Формування окремого QA-відділу в монолітній архітектурі зумовлює виникнення комунікаційних бар’єрів між розробниками та тестувальниками, що призводить до затримок у виявленні та виправленні дефектів. Тестувальники, працюючи як відокремлена команда, часто стикаються з необхідністю очікування завершення розробки функціональності перед початком тестування, що створює часові затримки та збільшує вартість виправлення знайдених дефектів [17; 20]. Окрім того, централізована структура команди тестування при монолітному підході до розробки ПЗ спричиняє виникнення «пляшкового горла» у процесах перевірки якості, оскільки всі зміни повинні проходити верифікацію через єдину команду тестувальників. Це суттєво впливає на швидкість випуску нових версій ПЗ та знижує ефективність процесів розробки в цілому [18]. Відсутність тісної інтеграції між командами розробки та тестування також ускладнює впровадження практик раннього тестування та запобігання дефектам на початкових етапах.

Таким чином роблячи аналіз недоліків і переваг підходів до тестування у монолітній архітектурі, виявляється що класичний підхід до організації процесів тестування не відповідає сучасним вимогам оцінки якості програмного забезпечення. З одного боку, підхід до розробки ПЗ як моноліту суттєве спрощення процесу відлагодження, що досягається завдяки єдиному середовищу

розробки та послідовному виконанню тестових сценаріїв [19]. Єдина системи та тісний взаємозв'язок між компонентами значно полегшують процес відтворення дефектів [17; 18], а централізований підхід до розгортання та моніторингу тестового середовища робить ці процеси більш керованими порівняно з розподіленими системами. Проте такий підхід має і суттєві недоліки. Централізована структура команди тестування призводить до значного сповільнення випуску нових версій ПЗ. Відокремленість QA-відділу від команд розробки призводить до проблем комунікації. А необхідність очікування завершення розробки функціоналу перед початком тестування в поєднанні з експоненціальним зростанням вартості виправлення помилок на кожному наступному етапі життєвого циклу, створює значні фінансові ризики для проекту. Характерне для монолітної архітектури тяжіння до інтеграційних та end-to-end тестів замість більш гнучкого модульного тестування також знижує ефективність процесів тестування в умовах сучасних вимог до швидкості та гнучкості розробки програмного забезпечення.

Принципова відмінність між мікросервісною та монолітною архітектурою ґрунтується у докорінно іншому підході до декомпозиції системи на частини [11]. У той час як традиційний підхід до розподілення системи на компоненти відбувається на рівні коду та виконання коду у єдиному середовищі, мікросервісна архітектура передбачає фізичне розділення компонентів на окремі додатки зі своїми вимогами, бізнес-процесами та зверненням до бази даних. Таке переосмислення парадигми виводить створення додатків на принципово інший рівень гнучкості розробки, масштабування та розгортання програмного забезпечення [6, 7], де з кожним сервісом можна оперувати незалежно від інших. Представлення мікросервісної архітектури розробки ПЗ [33] наведено на рис. 1.3.

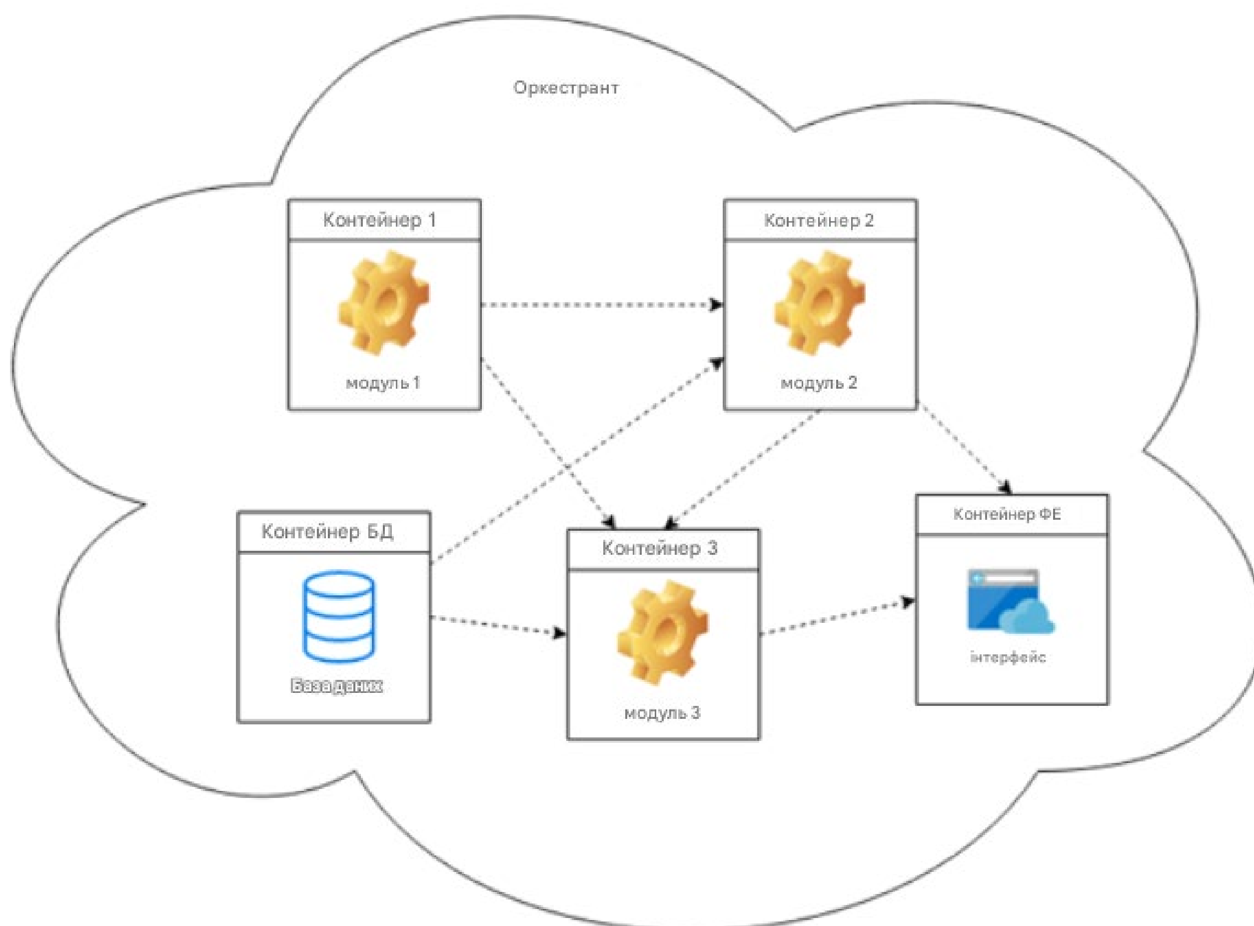

Рисунок 1.3 – Приклад мікросервісної архітектури розробки ПЗ

Окрім того, перехід до мікросервісної архітектури зумовлений також іншими важливими факторами. Серед яких використання сучасної методології розробки програмного забезпечення Agile, такі зміни суттєво модифікують підходи оцінки якості ПЗ. Традиційні практики тестування, що були розроблені для оптимізації роботи великих централізованих груп тестування, не можуть підтримувати швидкі темпи поставки, характерні для agile-команд розробки. В Agile-методології тестування стає невід'ємною частиною кожної ітерації розробки, де QA працюють разом з розробниками як єдина команда [29]. Цикл розробки ПЗ за Agile методологією [36] наведено на рис. 1.4.

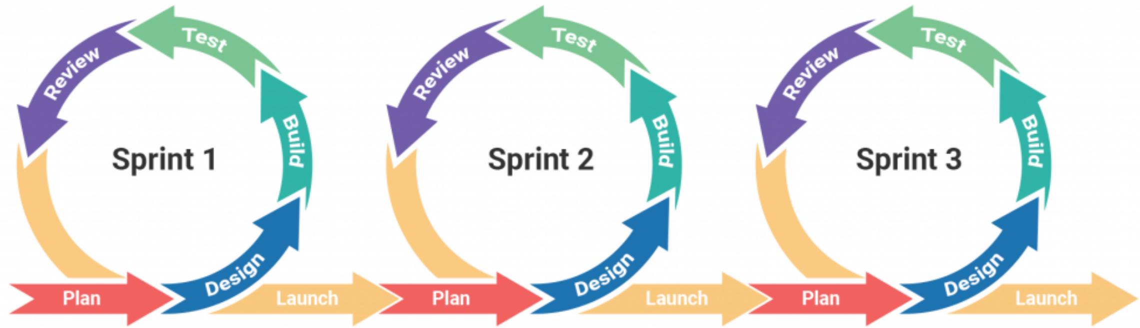

Рисунок 1.4 – Цикл розробки ПЗ за Agile метологією

Такий підхід отримав назву "shift-left testing" - концепція, що передбачає переміщення активностей з тестування на більш ранні етапи життєвого циклу розробки програмного забезпечення. На відміну від традиційного підходу, де тестування починається після завершення розробки, "shift-left" забезпечує раннє виявлення та запобігання дефектів. Тестувальник приймає активну участь на всіх стадіях розробки функціональності: у плануванні спринтів, декомпозиції задач та оцінці їх складності, розробці критеріїв прийняття (acceptance criteria) та проектуванні тестових сценаріїв ще до початку розробки. "Shift-left testing" - концепція [35] наведено на рис. 1.5.

Рисунок 1.5 – "shift-left testing" – концепція

Стрімкий розвиток технологій, зростання використання програмних рішень у всіх галузях, збільшення складності бізнес-процесів підприємств та необхідність швидкої адаптації до умов ринку разом з розумінням обмежень монолітної архітектури підштовхує організації до пошуку більш ефективних підходів до забезпечення якості програмного забезпечення, що призводить до переходу від монолітної до мікросервісної архітектури [9] та переосмислення процесів тестування з акцентом на ранню інтеграцію тестування у процеси розробки та автоматизацію процесів перевірки якості [13; 21].

Ключовим аспектом тестування в Agile стає автоматизація, яка забезпечує можливість швидкого отримання зворотного зв'язку щодо якості змін. Як показує практика [29], ручне тестування, навіть при залученні великої кількості тестувальників, не може забезпечити необхідну швидкість перевірок для щоденних збірок та безперервної інтеграції. Це особливо критично через складність відтворення всіх можливих сценаріїв взаємодії між сервісами вручну [14]. Практики безперервної інтеграції (CI) та безперервного розгортання (CD) вимагають створення комплексних наборів автоматизованих тестів, що можуть виконуватися регулярно та надавати швидкий фідбек команді [13; 21]. Це призводить до необхідності розвитку комплексної інфраструктури автоматизованого тестування, включаючи управління тестовими даними, моніторинг та аналіз результатів. Сучасні підходи до автоматизації тестування [13] демонструють значне підвищення ефективності порівняно з традиційними методами. Це набуває особливого значення в контексті мікросервісної архітектури, де кожен сервіс має власний життєвий цикл розробки та розгортання.

Автоматизація тестування в мікросервісній архітектурі вимагає комплексного підходу та специфічної інфраструктури. Кожен сервіс потребує власного набору автоматизованих тестів, які повинні виконуватися швидко та надавати чіткий фідбек. При цьому особливу увагу приділяють підготовці тестових даних та управлінню тестовим середовищем. Критично важливим стає впровадження практик continuous testing, коли автоматизовані тести ін-

тегруються в процеси безперервної інтеграції та розгортання (CI/CD), забезпечуючи швидку перевірку якості кожної зміни.

Мікросервісна архітектура суттєво змінює процеси тестування програмного забезпечення. Ефективні підходи до тестування, напрацьовані у монолітних додатках, виявляються не оптимальними та недостатніми у сучасній парадигмі [13; 21]. Це передусім пов'язано з переходом до окремих сервісів, які вимагають більш детальних і специфічних методів перевірки як окремих компонентів, так і їх інтеграції.

Характерною особливістю тестування мікропроцесорних додатків стає багаторівневий підхід до забезпечення якості розроблюваного продукту:

– контрактне тестування API та автоматизація API;

– модульне тестування кожного сервісу окремо;

– інтеграційне тестування взаємодії сервісів;

– тестування продуктивності системи;

– тестування відмовостійкості та відновлення після збоїв;

– системне тестування end-to-end;

– регресійне тестування;

– тестування прийняття користувачем (user acceptance testing UAT).

Функціональна розподіленість системи на окремі сервіси створює необхідність ретельного тестування взаємодії між компонентами, враховуючи можливі затримки та відмови мережі. Можливість незалежного розгортання сервісів зумовлює необхідність гарантування зворотної сумісності API та тестування різних версій сервісів в рамках однієї системи. Використання різних технологій при розробці сервісів ускладнює створення уніфікованого підходу до автоматизації тестування.

Особливу увагу при тестуванні у мікросервісній архітектурі необхідно приділяти перевірці контрактів, забезпечуючи взаємодію сервісів у відповідності до заздалегідь визначених інтерфейсів. Це дозволяє командам розробляти окремі сервіси незалежно одна від одної, спираючись на узгоджені специфікації взаємодії. В основі такого підходу лежать практики test-driven

development [17] та domain-driven design [16], які забезпечують якість розробки на рівні окремих компонентів.

Contract testing стає фундаментальним підходом у тестуванні взаємодії між сервісами. Контракти визначають очікувану поведінку та формат даних для кожного інтерфейсу, що дозволяє командам розробляти та тестувати сервіси незалежно. При цьому важливим є версіонування контрактів та перевірка зворотної сумісності при оновленні інтерфейсів. Це забезпечує стабільність системи при незалежному розгортанні різних версій сервісів.

Сучасна піраміда тестування в мікросервісній архітектурі відображає зміщення акцентів у бік модульного тестування та автоматизації. Якщо в монолітній архітектурі основу піраміди складало мануальне тестування, то в мікросервісному підході фундаментом стають Unit-тести. Успішним прикладом такої трансформації є досвід Міністерства фінансів США, де команда повністю відмовилась від традиційного центру тестування (TCOE) на користь поведінково-орієнтованої розробки (BDD) з використанням автоматизованих інструментів, що дозволило досягти значно вищого рівня автоматизації та швидкості тестування [29]. Жоден код не може бути випущений без належного покриття модульними тестами, що стає не просто рекомендацією, а обов'язковою вимогою та найкращою практикою розробки [18].

У цій новій парадигмі Unit-тести формують широку основу піраміди, забезпечуючи швидкий зворотний зв'язок та надійний фундамент для рефакторингу. Наступний рівень складають інтеграційні тести, які перевіряють взаємодію між компонентами одного сервісу. Контрактні тести займають важливе місце в середині піраміди, гарантуючи коректність взаємодії між сервісами. На вершині знаходиться обмежений набір end-to-end тестів, що перевіряють критичні бізнес-сценарії через всі сервіси системи.

Система оцінки якості тестування в мікросервісній архітектурі суттєво відрізняється від монолітного підходу та розширюється новими метриками, що виходять за межі традиційних показників тестового покриття. Важливими стають такі аспекти як час виявлення та локалізації дефектів у розподіленій систе-

мі, ефективність ізоляції дефектів у межах окремих сервісів, ступінь автоматизації тестування міжсервісної взаємодії та швидкість розгортання і верифікації змін. Особливу увагу приділяють показникам стабільності контрактів між сервісами та ефективності раннього виявлення дефектів у процесі розробки.

Таким чином, перехід до мікросервісної архітектури трансформує не лише підходи до розробки, але й процеси забезпечення якості програмного забезпечення. Раннє залучення тестувальників до життєвого циклу розробки, акцент на модульному тестуванні та автоматизації, разом з чітко визначеними метриками якості, створюють передумови для розробки більш надійних та масштабованих систем. Виявлені складнощі в тестуванні розподілених систем потребують впровадження новітніх рішень, зокрема тих, що базуються на використанні генеративного штучного інтелекту, що буде розглянуто в наступних розділах.

Для об'єктивної оцінки ефективності процесів тестування в мікросервісної архітектурі, особливо на рівні модульних тестів, важливо визначити ключові метрики [19]. Серед основних кількісних показників можна виділити:

– покриття коду модульними тестами (code coverage);

– час, необхідний на написання Unit-тестів;

– кількість виявлених дефектів на етапі модульного тестування;

– час на виправлення дефектів, знайдених під час Unit-тестування;

– відсоток проходження тестів (test pass rate).

Якісні метрики включають:

– відповідність тестів документації та вимогам;

– повнота перевірки граничних випадків;

– зрозумілість тестових сценаріїв;

– підтримуваність тестового коду;

– ефективність виявлення крайових випадків.

Аналіз цих метрик у контексті монолітної архітектури виявляє певні обмеження [17; 18]. Хоча монолітні додатки можуть демонструвати високі показники покриття коду тестами, часто виникають проблеми з якістю самих тестів та часом, необхідним на їх розробку та підтримку.

1.2 Аналіз сценаріїв вирішення задачі оптимізації процесів тестування програмного забезпечення з використанням моделей LLM

Життєвий цикл тестування програмного забезпечення суттєво відрізняється в монолітній та мікросервісній архітектурах, що зумовлено фундаментальними відмінностями в організації процесів розробки та розгортання. У монолітній архітектурі життєвий цикл тестування традиційно слідує послідовному підходу, де різні види тестування виконуються в чітко визначеній послідовності після завершення розробки функціональності.

Процес тестування в монолітній архітектурі починається з модульного тестування окремих компонентів, хоча його ефективність часто обмежена через високу зв'язаність коду [17]. Наступним етапом є інтеграційне тестування, яке в монолітній архітектурі зосереджується на перевірці взаємодії між різними модулями в рамках єдиної системи. Системне тестування в монолітній архітектурі є відносно простим з точки зору організації, оскільки вся система розгортається та тестується як єдине ціле.

У мікросервісній архітектурі життєвий цикл тестування набуває більш складної та розподіленої форми. Кожен мікросервіс має власний життєвий цикл розробки та тестування, що вимагає координації між різними командами та забезпечення узгодженості інтерфейсів взаємодії [13]. Модульне тестування в мікросервісній архітектурі набуває більшого значення, оскільки кожен сервіс повинен бути максимально автономним та надійним. Зміна піраміди тестування з переходом від традиційного підходу до Agile [37] наведено на рис. 1.6.

Особливу увагу в мікросервісній архітектурі приділяють тестуванню взаємодії між сервісами, що включає перевірку контрактів API, обробку мережевих затримок та відмов, а також забезпечення версійності інтерфейсів. Інтеграційне тестування в мікросервісній архітектурі вимагає створення складної інфраструктури, яка дозволяє тестувати різні комбінації версій сервісів та сценарії їх взаємодії.

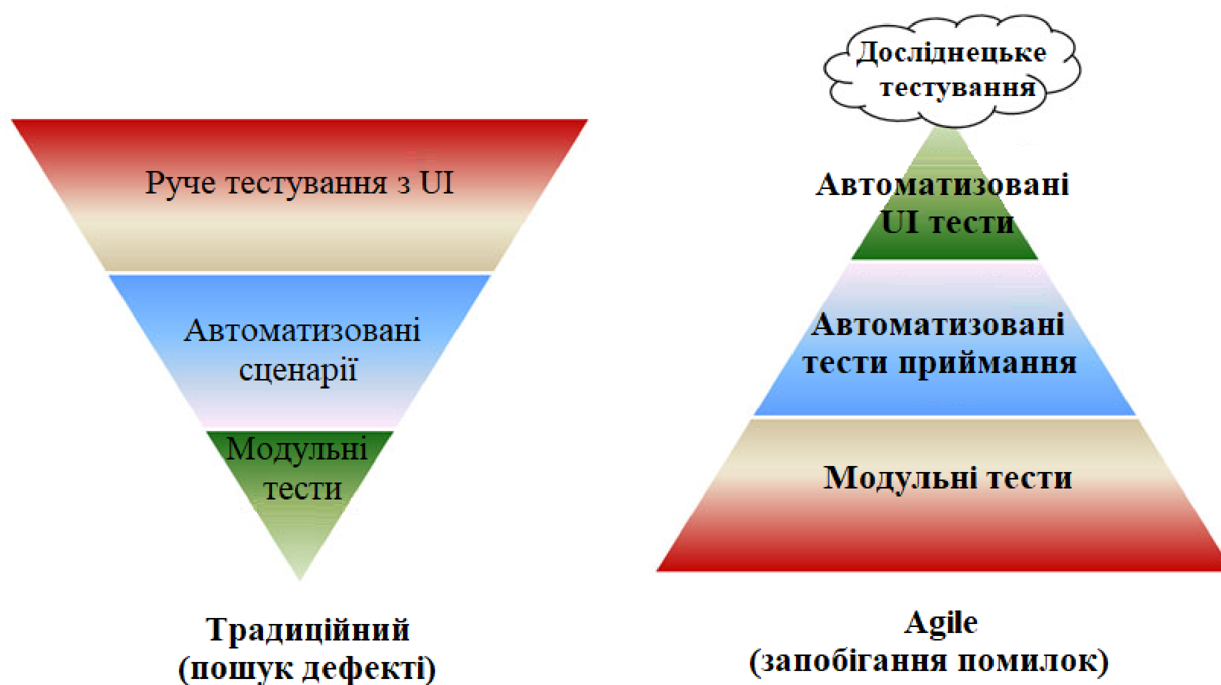

Рисунок 1.6 – Піраміда тестування з переходом від традиційного підходу до Agile

Сучасний етап розвитку індустрії програмного забезпечення характеризується низкою викликів у сфері забезпечення якості, які потребують нових підходів та інструментів. Масштабування процесів тестування стає критичним фактором у контексті зростання складності систем та прискорення циклів розробки. Традиційні підходи до організації тестування виявляються неефективними при роботі з розподіленими системами та мікросервісною архітектурою [40].

Одним з ключових викликів є забезпечення ефективної автоматизації тестування в умовах гетерогенного середовища мікросервісів. Різноманітність технологій, що використовуються в різних сервісах, створює складності у створенні уніфікованого підходу до автоматизації. Крім того, асинхронна природа взаємодії між сервісами вимагає специфічних підходів до організації тестових сценаріїв та валідації результатів.

Забезпечення якості в розподілених системах ставить нові вимоги до процесів моніторингу та діагностики проблем. Складність відстеження причинно-наслідкових зв'язків у розподіленій системі вимагає впровадження спеціалізованих інструментів трасування та аналізу логів. Особливої важливості набуває здатність швидко локалізувати та усувати проблеми в продуктивному середовищі.

1.3 Змістовна та формальна постановка задачі

Формальна постановка задачі оптимізації процесів тестування програмного забезпечення з використанням моделей LLM передбачає математичний опис процесу генерації модульних тестів. Вона включає визначення вхідних даних (вихідний код, вимоги до тестування, параметри LLM), формалізацію процесу генерації тестів (подання коду та вимог у векторному просторі, застосування LLM), формулювання критеріїв якості згенерованих тестів (метрики покриття, коректності, читабельності) та визначення обмежень (час генерації, обчислювальні ресурси, доступність навчальних даних).

Змістовна постановка задачі фокусується на практичній значимості дослідження. Вона передбачає аналіз проблем тестування у мікросервісній архітектурі (складність, часові витрати, потреба в автоматизації), оцінку потенціалу LLM для оптимізації процесів тестування (генерація тестових сценаріїв, аналіз коду, оптимізація покриття), дослідження факторів впливу на якість генерації тестів (вибір LLM моделі, стратегії формування промптів, параметри генерації) та верифікацію результатів (порівняння згенерованих тестів з написаними вручну, аналіз ефективності виявлення дефектів).

Метою дослідження є розробка методології використання LLM для оптимізації процесів тестування ПЗ при переході до мікросервісної архітектури.

1.4 Постановка задач дослідження

На основі проведеного системного аналізу предметної області та виявлених особливостей тестування в різних архітектурних підходах можна конкретизувати основні задачі дослідження. Першочерговим завданням є розробка комплексної системи метрик для оцінки якості згенерованих модульних тестів. Ця система має включати базові метрики, такі як кількість тестів та щільність перевірок, метрики покриття коду з акцентом на покриття методів, граничних ви-

падків та граничних значень, показники якості, що оцінюють описовість назв та різноманітність тверджень, а також метрики читабельності коду, які враховують наявність документації та структурованість тестів.

Наступною важливою задачею є створення ефективних підходів до використання LLM у процесах тестування. Це передбачає розробку різних стратегій формування промптів, дослідження впливу параметрів моделей на якість генерованих тестів та визначення оптимальних моделей для різних типів тестових задач. Особливу увагу необхідно приділити дослідженню впливу контексту та деталізації промптів на якість генерації.

Для практичної реалізації розроблених підходів необхідно створити програмну систему, що забезпечить взаємодію з API різних LLM моделей, реалізує модулі генерації та аналізу тестів, а також компоненти оцінки якості та формування аналітичних звітів. Система має підтримувати гнучке налаштування параметрів генерації та забезпечувати можливість порівняльного аналізу результатів різних моделей.

Завершальним етапом є проведення експериментального дослідження на основі спеціально підготовленого набору синтетичних тестів. Це дозволить виконати комплексний порівняльний аналіз ефективності різних LLM моделей, оцінити вплив різних стратегій формування промптів та сформувати практичні рекомендації щодо вибору оптимальних підходів для різних сценаріїв використання. Результати експериментів мають бути верифіковані та проаналізовані з точки зору практичної застосовності розроблених підходів у реальних проектах з розробки програмного забезпечення.

Вирішення цих задач у комплексі дозволить створити ефективну методику використання LLM для автоматизації процесів тестування, що матиме практичну цінність для оптимізації розробки програмного забезпечення.

2 ВИБІР ТА ОБҐРУНТУВАННЯ МЕТОДУ РОЗВ’ЯЗАННЯ

2.1 Аналіз особливостей модульного тестування та його ролі у забезпеченні якості програмного забезпечення

Модульне тестування (Unit testing) – це процес під час якого проводиться тестування найменшої функціональної одиниці коду. Такий тип тестування забезпечує підвищення якості програмного коду та на даний час є невід’ємною частиною розробки програмного забезпечення. Представлення модулю в межа мікросервісу [30], що наведено на рис. 2.1.

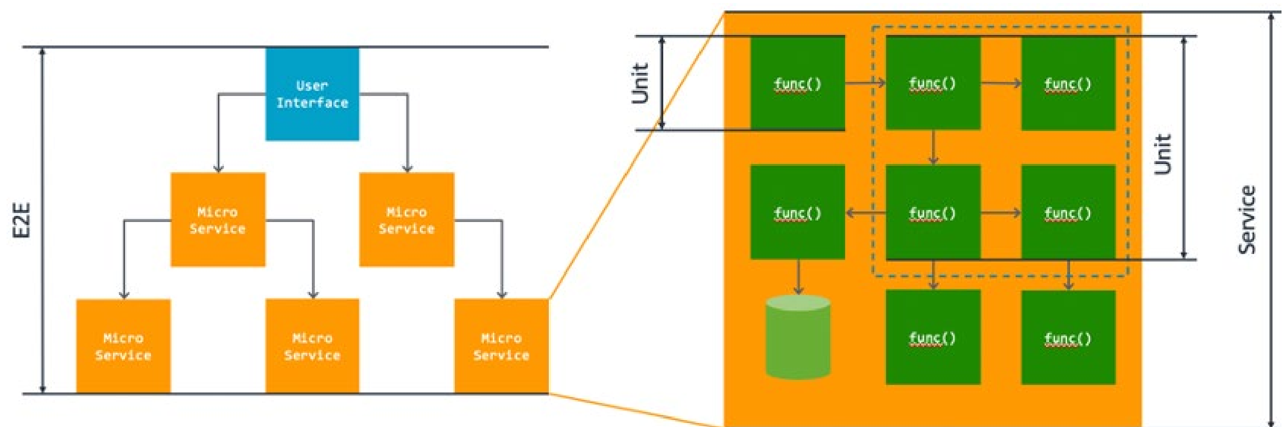

Рисунок 2.1 – Представлення модулю в рамах мікросервісу

Так наразі написання програмного коду у вигляді функціональних блоків та створення модульних текстів для кожної одиниці коду вважається не тільки правилом гарного тону, але й стандартом при розробці програмного забезпечення. Від розробника окрім самого коду вимагається писати і модульні тести у вигляді коду. Після написання модульних тестів цей код має автоматично запускатися щоразу, коли до програмного коду модуля вносяться зміни. В результаті такого підходу:

– виконується раннє виявлення помилок. Модульні тести виконуються на етапі збірки, ще до того як зміни в компоненті потраплять до тестувальників та кінцевих користувачів. Це значно підвищує швидкість виникнення помилок та

знижує витрати на їх виправлення;

– підвищується якість коду. У зв'язку з вимогою написання Unit-тестів розробники змушені писати модульний та зрозуміліший код. Це в свою чергу призводить до менш заплутаної та більш гнучкої системи яку легше підтримувати та масштабувати;

– спрощення рефакторінгу. Через наявність модульних тестів розробник може менше переживати за помилки допущені при змінах у модулі. Так як всі блоки вже покриті перевірками які виявлять помилки.

Однак не зважаючи на те що Unit тестування має безліч переваг, навіть це не переконує розробників усіляко слідувати їй. До цього є ряд причин. Так розробники вважають що написання Unit-тестів є зайвою роботою що потребує додаткових часу та зусиль яку не бачить та не оцінює менеджмент, особливо коли терміни виконання задач підтискають. Зайвий тиск з боку керівництва та неможливість оцінити витрати часу та ресурсів на написання модульних тестів призводять до нехтування програмістами цими вимогами та прихильності до написання нового функціоналу. Іншою проблемою з якою стикаються розробники є складність написання тестів. Так для складних заплутаних модулів складність модульних тестів значно зростає. Цей процес потребує налаштування додаткових бібліотек так інструментів. Додаткового часу на вивчення. Окрім цього розробники за своєю природою не звикли виконувати рутинні завдання в яких відсутній якійсь креатив та додаткові витати часу не ведуть до розвитку додатків. На відміну від тестувальників в програмістів відсутні знання про правила покриття тестами модулів, а також немає достатньої картини функціонування додатку. Які результат модульні тести можуть бути тривіальні та не покривати критичні функціональності. На останок наявність Unit-тестів не гарантує відсутності дефектів та стовідсоткового покриття кодом. Тому ця практика частиною розробників вважається непотрібною.

Як бачимо підхід з написанням Unit-тестів є дуже важливою практикою яка покращує швидкість виявлення помилок при збірках, якість самого коду, спрощення рефакторінгу, підтримку та розширюваність коду. Але при всіх цих

перевагах багато хто з програмістів не бачить переваг у використанні такої практики та навіть нехтує нею. Тому логічним кроком було б віддати виконання рутинних операції машині. Саме для виконання таких задач LLM мають революційний потенціал.

2.2 Дослідження можливостей застосування генеративного штучного інтелекту в процесах тестування

За минулий час відбувся стрімкий прогрес технологій штучного інтелекту (ШІ). Протягом останніх років значно покращилася якість генеративних моделей ШІ, і такі зміни відкривають нові можливості для використання технологій ШІ для вирішення комплексу проблем тестування у мікросервісній архітектурі та переосмислення процесів тестування. Генеративний ШІ являє собою тип моделей машинного навчання, які здатні створювати нові дані, подібні до тих, на яких вони були навчені. Вони використовуються для генерування тексту, зображень, звуку та інших видів контенту. Можливість створення контенту на основі навчальних даних робить генеративні моделі дуже перспективними для використання у тестуванні, а саме для генерації тестових сценаріїв, тестових даних та автоматизації [30].

В межах даного дослідження особливе зацікавлення ставлять великі мовні моделі (Large Language Models, LLM) [5, 24], такі як генеративний попередньо навчений трансформе (Generative Pre-trained Transformer, GPT) та їх аналоги, тобто модель штучного інтелекту, яка створює тексти, навчаючись на великому обсязі даних. Така модель побудована на архітектурі трансформерів і здатна генерувати зв'язний текст, відповіді на питання, виконувати переклади, аналізувати технічну документацію, писати код та багато іншого. Таким чином LLM відкривають нові можливості для автоматизації процесів тестування. Практика показує [4], що тестувальники, які володіють інструментами ШІ, демонструють значно вищу ефективність у порівнянні з традиційними підходами [41].

Сучасні моделі LLM, такі як GPT-4, Gemini, T5 тощо, демонструють здатність до розуміння природної мови, генерації осмисленого тексту, відповідей на запитання та інших складних задач обробки інформації [5]. Інтеграцію GPT у процеси тестування мікросервісних додатків можливо розділити на декілька ключових напрямків:

– автоматична генерація тестових сценаріїв. Генеративні моделі можуть аналізувати вимоги, API, документацію та програмний код, що дозволяє використовувати їх для розробки тестових сценаріїв. Такі моделі можуть пришвидшити створення тестів які покривають як позитивні так і негативні сценарії з використанням різних технік тестування, таких як класи еквівалентності, граничні значення та інші. Також завдяки використанню LLM у процесах тестування можливо підвищити покриття коду тестовими сценаріями, покращити перевірки міжсервісної взаємодії, генерувати тести що перевіряють різноманітні сценарії асинхронної комунікації;

– створення та підтримка тестових даних. Генеративний ШІ може автоматично продукувати узгоджені колекції тестових даних для перевірки різноманітних сервісів, враховуючи їх зв'язок та вимоги до даних. Це особливо важливо при тестуванні сценаріїв, що охоплюють кілька сервісів одночасно. Тестові дані мають відповідати вимогам та бізнес-логікі кожного сервісу та дотримуватись цілісності міжсервісних взаємозв'язків. Використання генеративного ШІ для створення наборів тестових даних суттєво зменшує час на тест-дизайну та підвищує якість тестування;

– оптимізація тестових наборів. Генеративні моделі здатні аналізувати існуючі сценарії та пропонувати поліпшення покриття прибиранням надлишкових чи додаванням відсутніх перевірок з метою оптимізації тестів та підвищення ефективності та якості покриття;

– автоматизація опрацювання виконання тестів. Генеративний ШІ може обробляти логи та результати проходження тестів, виявляти закономірності у виникаючих проблемах, формулювати причини виникнення проблем та передбачати потенційні місця виникнення нових помилок. Така особливість є дуже

важливою бо виявлення та локалізація причин помилок у розподілених системах є вузьким місцем, а аналіз і знаходження першопричин виконується вручну та займає багато часу;

– генерація документів та звітів. Генеративні моделі можуть як створювати так і підтримувати в актуальному стані тестову документацію , тестові сценарії та звіти про виконання тестів.

Також дослідження показують , що використання великих мовних моделей дуже ефективне при модульному та end-to-end тестуванні, тоді як для інтеграційного та приймального тестування їх застосування залишається обмеженим. При цьому основними викликами є досягнення високого покриття тестами, вирішення проблеми тестових оракулів, проведення ретельних оцінювань та застосування в реальних промислових умовах [25]. Особливо це стосується випадків, коли організації з міркувань безпеки даних не можуть використовувати комерційні LLM і змушені застосовувати відкриті моделі з додатковим навчанням на власних даних.

Інтеграція штучного інтелекту у процеси тестування [26] дозволяє значно покращити ефективність виявлення дефектів та автоматизації тестування [24, 27]. Важливим аспектом є правильний вибір стратегії тестування мікросервісів [12], що включає різні рівні та підходи до автоматизації [21, 22].

Окрім того до основних переваг застосування LLM у тестуванні можна також віднести підвищення швидкості розробки тестів за рахунок генерації сценаріїв і даних, збільшення покриття вимог і зменшення впливу людського фактору, підвищення якості тестування, можливість обробки великих обсягів документації і вихідного код [41].

Запровадження генеративного ШІ у процеси тестування супроводжується рядом значних викликів [38, 39]. Перш за все, слід звернути увагу на якість створеного коду, який, навіть за умови високої потужності сучасних моделей, все ще потребує уважної перевірки та доопрацювання спеціалістами. Крім того, важливим фактором залишається залежність ефективності генеративних моделей від якості та адекватності навчальних даних. Окрему проблему становить

обмежена здатність моделей повністю розуміти складну бізнес-логіку, через що часто не враховуються важливі деталі предметної області, що потребує постійного контролю з боку експертів. Ще одним серйозним викликом є забезпечення безпеки та конфіденційності даних, особливо при використанні зовнішніх сервісів генеративного ШІ, де необхідний ретельний контроль обсягу та характеру інформації, яка передається для обробки.

Для того щоб ефективно впроваджувати генеративний штучний інтелект у процеси тестування програмного забезпечення необхідно використовувати системний підхід у якому будуть враховуватися наступні аспекти:

– формування стратегії використання ШІ в залежності від специфіки галузі та проекту;

– розробка процесів валідації та верифікації тестових артефактів генеруємих ШІ;

– навчання команд розробки, а саме тестувальників, правилам ефективного використання інструментів ШІ у повсякденній роботі;

– інтеграція та впровадження інструментів генеративного ШІ у існуючу процеси автоматизації;

Дослідження показують [3], що застосування ШІ в процесах тестування дозволяє суттєво скоротити час на виявлення та локалізацію дефектів. При цьому важливо забезпечити правильну стратегію впровадження таких інструментів [15, 27]. Аналіз сучасних досліджень також показує перспективність поєднання LLM з традиційними техніками тестування для підвищення різноманітності та якості генерованих тестів [28]. Це дозволяє компенсувати обмеження кожного з підходів та досягти кращих результатів.

Роботи, проведені в області застосування LLM для створення програмної документації [28], підтверджують значний потенціал для підвищення ефективності розробки. Наприклад, експериментальне оцінювання використання CodeLlama та GPT-4 для генерації документації вимог до програмного забезпечення (SRS) виявило, що:

– якість документації, згенерованої LLM, наближається до рівня документів, створених фахівцями початкового рівня;

– застосування LLM дозволяє зменшити час генерації специфікації вимог у 7-47 разів порівняно з традиційним підходом;

– GPT-4 показує кращі результати у валідації та виправленні вимог, тоді як CodeLlama генерує більш детальні, але менш точну документацію;

– головними складнощами при використанні LLM для генерації документації є: потреба у додатковому налаштуванні промптів, можливість галюцинацій та необхідність верифікації згенерованого контенту.

Такі результати доводять доцільність використання LLM не лише безпосередньо для процесів тестування програмного забезпечення, але й для супутніх процесів розробки, а саме створення специфікацій вимог. При цьому важливо зазначити, що участі кваліфікованих експертів не оминути, тому що найкращі результати від використанні LLM досягаються під контролем досвідчених фахівців.

Таким чином, беручи до уваги всі переваги та недоліки, можемо зробити висновки що генеративний штучний інтелект уявляє собою доволі потужний інструмент який має перспективи стрімкого розвитку і вже зараз має братися до уваги на будь якому проекті. LLM являє собою ефективний інструмент оптимізації процесів тестування програмного забезпечення у мікросервісній архітектурі. Використання ШІ дозволяє суттєво спростити рутині процеси тестувальників, підвищити якість розробляєм тестових артефактів та створюємого продукту, а також скоротити процес розробки у цілому. При цьому треба розуміти що інструменти генеративного ШІ не можуть замінити працівників, але вони дозволяють розширити можливості тестувальників і дозволити команді сконцентруватися на інших активностях та більш складних аспектах розробки програмного забезпечення.

2.3 Обґрунтування вибору LLM моделей

Вибір великих мовних моделей є критичним етапом, що безпосередньо впливає на якість генеруємих результатів, достовірність та практичну цінність отриманих відповідей. Наразі багато різних компаній пропонують свої рішення в області LLM навчених на великих об’ємах даних, що дозволяє спілкуватися з ними та генерувати відповіді. Всі ці моделі мають багато спільних характеристик таких як:

– архітектура Transformer. Ця архітектура є основою багатьох новітніх LLM і дозволяє ефективно опрацьовувати великі послідовності даних, наприклад текст чи код. Така можливість реалізована завдяки механізму уваги;

– здатність до діалогу. Так LLM можуть вести діалог з користувачем, відповідати на питання, уточнюючи деталі та аналізуючи контекст бесіди;

– великі розміри моделей. Ця властивість дає можливість зберігати та обробляти великі об’єми інформації;

– вільний перехід між різними мовами. LLM можуть перекладати тексти та вільно підтримують перехід з однієї мови до іншої під час діалогу. Таким чином на вхід моделі можливо подавати як різний код так і документацію різними мовами;

– доступність виклику через API. До великих мовних моделей можливо звертатися з використанням API, що дає можливість інтегрувати потужні можливості LLM у різні додатки.

При цьому завдяки специфіці навчальних даних велика мовна модель може бути орієнтована на роботу з текстами, написання віршів, генерацію малюнків, відео чи то коду. В цьому дослідженні увага приділяється саме можливості LLM читати, аналізувати та генерувати код різними мовами.

У рамках даної роботи досліджуються підходи використання великих мовних моделей для саме для генерації Unit-тестів, тобто можливість звільнення розробників від написання ізольованих тестів для окремих модулів коду. Такий підхід має пришвидшити написання модульних тестів, дозволити виявляти по-

милки на ранніх стадіях, покращити як якість коду так і якість написаних тестів та звільнити час розробників на написання нових модулів, що для менеджменту та замовників є більш зрозумілішим.

Для аналізу можливості використання LLM для генерації Unit-тестів було обрани три наступних великі мовні моделі:

– ChatGPT. Модель розроблена компанією OpenAI та побудована на архітектурі Transformer, тобто модель має механізм самоуваги (self-attention) що дозволяє моделі ефективно розуміти контекст розмови. Модель здатна розуміти природну мову, документацію, код та генерувати тексти;

– Gemini. Велика мовна модель від компанії Google, що має доволі велику кількість параметрів. На відміну від ChatGPT була розроблена з самого початку як мультимодальна, тобто має можливість обробляти інформацію з різних джерел, наприклад текст, аудіо та відео матеріал, зображення чи то код. Завдяки мультимодальності модель бо більш цілісно сприймати інформацію у форматі різних типів джерел. Також Gemini оптимізована до високої продуктивності при низькому рівні енергоспоживання;

– CodeLlama. Мовна модель розроблена компанією Meta. Також побудована на архітектурі Transformer і може розуміти контекст. Є у відкритому доступі і може використовуватися для розробки власних інструментів та застосунків. На відміні від ChatGPT та Gemini CodeLlama проходила навчання саме на великих об’ємах коду включаючи і GitHub, тобто ця модель спеціалізована сама на розумінні мов програмування. Велика мовна модель оптимізована для роботи з різними мовами програмування, розуміння семантики та структури коду. LLM здатна генерувати код базуючись на фрагментах коду чи навіть на основі текстового опису. Також CodeLlama може пояснювати згенерований код природною мовою.

Вибір саме цих мовних моделей був обумовлений наступною низкою факторів:

– висока здатність моделей до розуміння коду та його генерації. Всі три моделі мають значну здатність до аналізу коду, розуміння контексту та генерації коду, що є критичним у даній роботі;

– доступність API. Кожна з обраних моделей надає можливість її використання через API;

– різноманітність використовуємих моделей. Обрані великі мовні моделі розроблені різними технологічними гігантами, що дозволяє порівнювати їх ефективність.

2.4 Критерії оцінювання якості Unit-тестів

При розробці оцінювання якості згенерованих модульних тестів було застосовано комплексний підхід оцінки з використанням чотирьох основних категорій метрик. Для оцінювання якості згенерованих модульних тестів були використані наступні критерії:

– базові метрики (Base);

– метрики покриття кода (Cov);

– метрики якості (Qual);

– метрики читабельності (Read);

– метрики часу генерування тестів (Time).

Загальна оцінка якості генеруємих Unit-тестів розраховується за формулою.

$$score = q_1 \times Base + q_2 \times Cov + q_3 \times Qual + q_4 \times Read, \qquad (2.1)$$

де q_1, q_2, q_3, q_4 – вагові коефіцієнти, які можуть задаватися в залежності від того, які метрики є більш важливими.

Базові метрики відіграють головну роль у формуванні оцінки і складають 30% від загальної оцінки. Обрані базові метрики та їх вагові коефіцієнти засновані на загальноприйнятих практиках модульного тестування та досвіді розробки програмного забезпечення. Базова метрика забезпечує оцінку фундаментальних характеристик тестового набору, що впливають на ефективність та підтримуваність.

До базових метрик відносяться загальна кількість тестів ($totalTests$), що

розраховується шляхом підрахунку анотації ($@Test$) у згенерованому коді. Ця метрика має максимальну оцінку п'ятнадцять балів та базується на наступних факторах:

– достатнє покриття функціональності – кожен публічний метод має бути перевірений хоча б одним тестом;

– уникнення надлишкового тестування – велика кількість тестів збільшує складність підтримки та часу виконання;

– збалансованість тестового набору – оптимальна кількість тестів забезпечує баланс між повнотою перевірок та зусиллями на підтримку.

$$totalTests = \sum @Test, \tag{2.2}$$

де $@Test$ – методи з анотацією Test.

Також до базових метрик відноситься середня кількість перевірок на один тест (averageAssertionsPerTest), що обчислюється як відношення кількості перевірок до кількості тестів. Ця метрика оцінюється у п'ятнадцять балів.

$$averageAssertionsPerTest = \frac{totalAssertions}{totalTests}, \tag{2.3}$$

де $totalAssertions$ – загальна кількість перевірок;

$totalTests$ – загальна кількість згенерованих тестів.

Щоб створити збалансовану формулу для оцінки якості покриття коди тестами, важливо вибрати такі критерії, які найкраще відображають ключові аспекти покриття. Метрики покриття становлять 30% загальної оцінки та включають три ключові показники, кожен з яких відповідає за різні аспекти якості тестування. Розглянемо основні наявні покриття коду:

а) покриття рядків коду (Code Coverage - CC). Це основний критерій, який дає загальне уявлення про те, скільки коду було протестовано. Така метрика

може використовуватися для простих проектів щоб впевнитися що всі частину коду покриваються тестами;

$$CC = \frac{N_{ecp}}{N_{tcp}} \times 100, \qquad (2.4)$$

де N_{ecp} – кількість елементів коду (наприклад, рядків, гілок чи умов), які були виконані під час тестування;

N_{tcp} – загальна кількість елементів коду (рядків, гілок, умов тощо) в програмі або модулі, який тестується.

б) покриття гілок (Branch Coverage – BC). Дана метрика забезпечує перевіряє покриття усіх логічних гілок тестами. Така метрика є більш детальною та допомагає виявити потенційні помилки в логіці. Критерій важливий на проєктах із розгалуженою умовною логікою, де необхідно впевнитись, що всі можливі гілки програми протестовані;

$$BC = \frac{N_{ebl}}{N_{tbl}} \times 100, \qquad (2.5)$$

де N_{ebl} – це кількість гілок, які були виконані хоча б один раз під час тестування (як позитивні, так і негативні);

N_{tbl} – це загальна кількість гілок у всіх умовах в коді, які можна виконати (включаючи всі if-else вирази, цикли тощо).

в) покриття умов (Condition Coverage – ConC). Данна метрика перевіряє як виконання всіх гілок так і покриття окремих умов (істинних/хибних умов). Така метрика корисна для проєктів із складними умовними виразами або розгалуженими логічними конструкціями, але досягти повного покриття умов складно;

$$Con = \frac{N_{ec}}{N_{tc}} \times 100, \qquad (2.6)$$

де N_{ec} – кількість умов у коді, які були протестовані на обидва можливі варіанти (істина та хиба) ;

N_{tc} – загальна кількість умов в коді, які потрібно протестувати.

г) покриття операторів (Statements Coverage – SC). Являє собою базову метрику що вимірює, скільки операторів коду було виконано під час тестування. Такий критерій важливий для загальної оцінки того, що більша частина коду не залишається без перевірки, але якщо оцінюється покриття гілок або шляхів;

$$SC = \frac{N_{ecl}}{N_{tc}l} \times 100, \qquad (2.7)$$

де N_{ecl} – це кількість рядків коду, які були виконані хоча б один раз під час тестування;

N_{tcl} – це загальна кількість рядків у програмі або перевіреному модулі, які мають бути покриті тестами.

д) покриття шляхів (Path Coverage – PC). Ця метрика є найбільш оновлюючою та перевіряє всі можливі шляхи виконання коду, включаючи всі комбінації умов і гілок. Вона важливо для складних програм з численними умовами та логічними шляхами, однак досягти повного покриття шляхів може бути надзвичайно складно для великих систем, оскільки кількість шляхів експоненційно зростає.

$$PC = \frac{N_{ep}}{N_{tp}} \times 100, \qquad (2.8)$$

де N_{ep} – кількість шляхів, які були фактично виконані під час тестування;

N_{tp} – загальна кількість усіх шляхів виконання програми, яку можна отри-

мати, комбінуючи різні умови, гілки та варіанти виконання.

е) покриття методів (Methods Coverage – MC). Ця метрика оцінює кількість методів модулю що були покриті тестами, забезпечує повноту тестування всіх публічних класів, виявляє не відтестований функціонал та запобігає пропуску важливих компонентів:

$$MC = \frac{testMethods}{originalMethods}, \tag{2.9}$$

де $testMethods$ – кількість методів, які були покриті під час тестування;

$originalMethods$ – загальна кількість усіх методів у модулі.

ж) покриття граничних випадків (Edge Cases Coverage– ECC). Ця метрика оцінює перевірку нульових та від'ємних значень, покриття пустих колекції та обробки null.

$$ECC = \sum edgeTest, \tag{2.10}$$

де $edgeTest$ – кількість граничних випадків перевіряємих у згенерованому коді.

з) покриття граничних умов (Boundary Tests Coverage– BTC). Ця метрика оцінює перевірку модуля на межах допустимих значень, тестування мінімальних та максимальних значень та ліміти.

$$BTC = \sum boundaryTest, \tag{2.11}$$

де $boundaryTest$ – кількість граничних умов перевіряємих у згенерованому коді.

У рамках дослідження було проаналізовано ефективність різних метрик покриття коду в контексті модульного тестування. Хоча загальноприйнятою практикою є використання покриття рядків коду, гілок та шляхів, для аналізу згенерованих модульних тестів ці метрики виявилися не оптимальними через

кілька факторів:

Покриття методів (methodsCovered) було обрано замість покриття рядків коду через відсутність можливості виконання коду при оцінці якості згенерованих тестів, необхідність статичного аналізу тексту тестів без їх запуску та можливість швидко оцінити базове покриття функціональності через відслідковування викликів методів

Покриття граничних випадків (edgeCasesCovered) використовується замість покриття гілок оскільки: аналіз гілок вимагає побудови графу потоку керування, що складно реалізувати без виконання коду. Граничні випадки часто є джерелом помилок та важливі для тестування. Можливість автоматизованої перевірки наявності тестів для типових проблемних ситуацій.

Тестування граничних умов (boundaryTests) обрано замість покриття шляхів через: надмірну складність обчислення покриття шляхів під час статичного аналізу, важливість перевірки поведінки на межах дозволених значень та простоту автоматизованого пошуку відповідних тестів у згенерованому коді

Таким чином, вибір даних метрик був обумовлений специфікою задачі аналізу згенерованих тестів, де немає можливості отримати метрики виконання коду. При цьому обрані метрики дозволяють оцінити основні аспекти якості згенерованих Unit-тестів:

– загальне покриття функціональності через аналіз викликів методів;

– тестування граничних випадків;

– перевірку поведінки на межах через пошук тестів граничних умов.

У подальших дослідженнях доцільно розглянути можливість інтеграції більш складних метрик покриття, таких як покриття рядків коду, гілок та шляхів, але це потребуватиме розробки механізмів виконання згенерованих тестів та збору відповідних метрик під час виконання.

Таким чином враховуючи особливості кожної метрики покриття описаної вище покриття коду можливо розрахувати:

$$Cov = q_1 \times MC + q_2 \times ECC + q_3 \times BTC \text{, де } \sum_{i=1}^{3} q_i \text{,} \qquad (2.12)$$

де q_1, q_2, q_3 – вагові коефіцієнти, які можуть задаватися в залежності від того, які аспекти покриття є більш важливими.

Для оцінки якості (quality metrics – QM) написання модульних тестів у системі реалізовано чотири метрики якості:

– hasDescriptiveNames (DNS). Ця функція є частиною аналізу якості тестів, що перевіряє, чи достатньо описові назви тестових методів. Довші, більш описові назви допомагають краще зрозуміти призначення тесту. Функція повертає true, якщо всі знайдені назви тестів містять 3 або більше слів, що вважається хорошою практикою для описових назв тестів;

– usesAssertionVariety (AVS). Ця функція перевіряє різноманітність використаних assert-тверджень у тесті. Повертає true, якщо використано 3 або більше різних типів перевірок. Функція є важливою, бо різні типи assertions перевіряють різні аспекти поведінки коду та допомагають знаходити різні типи помилок;

– hasTestDocumentation (TDS). Функція перевіряє наявність документації у тестовому коді. Повертає true, якщо знайдено хоча б один з типів коментарів. Дана функція є простою у реалізації, але не перевіряє змістовність, якість та не відрізняє закоментований код від документації;

– followsNamingConventions (NCS). Функція перевіряє дотримання конвенцій іменування тестових методів, а саме використання backticks для їх назв. Дана функція уніфікує написання тестів, покращую читабельність та відповідає конвенціям.

Загальна оцінка якості розраховується за наступною формулою, де кожна функція отримує один бал, якщо умова виконується, або нуль, якщо не виконується.

$$Qual = q_1 \times DNS + q_2 \times AVS + q_3 \times TDS + q_4 \times NCS, \qquad (2.13)$$

де q_1, q_2, q_3, q_4 – вагові коефіцієнти, які можуть задаватися в залежності відтого, які аспекти якості є більш важливими.

Вибір даних метрик якості обумовлений необхідністю оцінки не лише те-

хнічних аспектів тестів, але й їх зрозумілості та підтримуваності. При генерації тестів за допомогою LLM важливо забезпечити не тільки функціональне покриття, але й відповідність стандартам написання тестового коду. Обрані метрики дозволяють комплексно оцінити якість згенерованих тестів з точки зору їх практичного використання в реальних проєктах.

Для оцінки читабельності (readability metric – RM) згенерованих модульних тестів реалізовано три метрики. Розглянемо детально кожну з них та їх реалізацію:

– usesBackticks (BS). Функція перевірки використання лапок, що перевіряє їх наявність. Використання дужок покращую читання тестів, дозволяє використання пробілів і спеціальних символів, а також є стандартом;

– hasComments (CS). Функція перевірки використання коментарів та перевіряє їх наявність чи відсутність. Наявність пояснює складні моменти в тестах, документувати бізнес логіку, полегшує розуміння тестів іншим розробникам;

– averageTestLength (TLS). Функція оцінює середню довжину метода. Вважається що короткі тести більш легші для розуміння. Оптимальною довжиною вважається тестовий метод довжиною від шости до двадцяті п’яти рядків. Враховуються тільки саме рядки методів.

$$averageTestLangth = \frac{\sum_{i=1}^{n} methodLength}{n}, \qquad (2.14)$$

де $methodLength$ – кількість рядків одного метода;

n – загальна кількість методів.

Загальна оцінка читабельності розраховується за формулою:

$$Read = q_1 \times BS + q_2 \times CS + q_3 \times TLS, \qquad (2.15)$$

де q_1, q_2, q_3 – вагові коефіцієнти, які можуть задаватися в залежності від того, які аспекти якості є більш важливими.

Обрані метрики читабельності та їх вагові коефіцієнти базуються на загальноприйнятих практиках написання модульних тесті. Вони дозволяють оцінити наскільки легко буде розуміти та підтримувати згенеровані тести в майбутньому.

Висновки за розділом 2

У розділі було проведено аналіз можливостей використання генеративного штучного інтелекту для тестування програмного забезпечення. Результати дослідження показали, що великі мовні моделі (LLM) мають значний потенціал для автоматизації процесу створення Unit-тестів. Це дозволяє розробникам зосередитися на більш важливих завданнях та підвищити якість тестового покриття.

Було визначено ключові напрямки використання LLM в тестуванні: автоматизоване формування тестових сценаріїв, генерація та оновлення тестових даних, оптимізація наборів тестів та автоматизація аналізу результатів тестування.

Для подальшого дослідження було обрано три моделі LLM: ChatGPT, Gemini та CodeLlama. Критерії вибору включали здатність розуміти та генерувати код та різноманітність підходів до навчання моделей. Особлива увага приділялася CodeLlama, яка спеціалізується на роботі з програмним кодом.

В рамках дослідження було створено комплексну систему метрик для оцінювання якості згенерованих Unit-тестів. Вона охоплює базові метрики, такі як кількість згенерованих тестів та середня кількість перевірок на один тест, метрики покриття коду, показники якості та читабельності, а також часові характеристики генерації тестів. Ці метрики лягли в основу математичної моделі для розрахунку загальної оцінки якості згенерованих тестів з використанням вагових коефіцієнтів. Завдяки цьому можна гнучко налаштовувати пріоритети оцінювання в залежності від особливостей проекту та вимог до тестування.

Важливим етапом дослідження стало обґрунтування вибору специфічних метрик покриття коду, які враховують особливості статичного аналізу згенерованих тестів. Замість стандартних метрик покриття рядків та гілок було запро-

поновано використовувати альтернативні метрики, що дозволяють оцінити якість без запуску тестів.

Було також визначено основні труднощі та обмеження, пов'язані з використанням LLM для генерації тестів, такі як необхідність перевірки згенерованого коду, залежність від якості даних для навчання та потреба в експертному контролі.

Проведена робота є теоретичною базою для експериментального дослідження ефективності різних LLM в генерації Unit-тестів та забезпечує інструменти для об'єктивної оцінки їх якості. Розроблені метрики та математична модель є основою для подальшого впровадження генеративного штучного інтелекту в процеси тестування програмного забезпечення.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Обґрунтування вибору мови Kotlin для дослідження використання LLM

Вибір мови програмування Kotlin для дослідження можливостей великих мовних моделей у генерації модульних тестів обумовлений кількома ключовими факторами.

Kotlin має чітку та регулярну структуру синтаксису, що спрощує роботу LLM з кодом. На відміну від мов з більш складним синтаксисом, Kotlin використовує послідовні правила форматування та структурування коду. Це особливо важливо при генерації тестів, оскільки моделі можуть більш точно відтворювати потрібні конструкції та дотримуватися стилістичних конвенцій.

Важливим фактором є широке представлення Kotlin у навчальних датасетах LLM. Оскільки Kotlin активно використовується для розробки під Android та серверних додатків, у відкритих репозиторіях доступна велика кількість прикладів коду та тестів. Це забезпечує моделям достатньо навчальних даних для розуміння типових патернів тестування та кращої генерації коду.

Kotlin має вбудовану підтримку конструкцій, що покращують читабельність тестів, наприклад, можливість використання рядків у backticks для назв тестів. Це дозволяє LLM генерувати більш описові та зрозумілі тестові сценарії. Також мова надає зручні засоби для роботи з колекціями та null-safety, що важливо для створення надійних тестів.

Для оцінки якості згенерованих тестів Kotlin надає можливість чіткого структурного аналізу коду. Завдяки регулярному синтаксису можна легко виділяти та аналізувати такі елементи як анотації тестів, assertions, документаційні коментарі. Це спрощує реалізацію метрик оцінки якості та робить результати більш надійними.

При використанні LLM для генерації тестів важлива можливість точного контролю контексту та форматування. Kotlin з його лаконічним синтаксисом

дозволяє ефективно передавати у промптах як вихідний код, так і очікувану структуру тестів, що покращує якість генерації.

3.2 Обґрунтування використання мови Python для аналізу даних

Python широко визнана мова розробки у науковому середовищі як потужний інструмент для аналізу та математичних обчислень. Популярність використання мови обумовлена багатьма характеристиками та можливостями, які роблять його оптимальним вибором для вирішення аналітичних задач.

Насамперед, Python виділяється наявністю великої кількості наукових бібліотек та інструментів. Мова надає доступ до потужних математичних бібліотек, таких як NumP та Matplotlib, які базовими для наукових обчислень та дають великі можливості при роботі з масивами даних. Також стандартні бібліотеки Python мають велику кількість вбудованих модулів що дозволяють розробникам зусередитися саме на вирішені основних задач дослідження, а не заглиблюватися у низькорівневі деталі реалізації.

Вбудовані типи даних високого рівня дозволяють виражати складні операції в компактній та інтуїтивно зрозумілій формі. Простий і лаконічний синтаксис Python дозволяє швидко реалізовувати ідеї, не витрачаючи зайвий час на написання складного коду. Динамічна типізація і інтерактивний режим роботи інтерпретатора дають можливість оперативно перевіряти фрагменти коду без необхідності явного оголошення типів змінних.

Крім того, підтримка Python модульності дозволяє легко розбивати код на окремі компоненти, які в подальшому можливо перевикористовувати. Це спрощує структурування та допомагає уникнути дублювання коду.

Для Python також існує широкий вибір спеціалізованих фреймворків і бібліотек, розроблених саме для тестування - pytest, Unittest, doctest, behave та інші. Вони надають готові засоби і функціональність для різних видів тестування - модульного, інтеграційного, системного, приймального тощо.

Вибір Python як мови для аналізу результатів генерації тестів різними LLM обумовлений кількома ключовими перевагами цієї мови у сфері обробки та візуалізації даних.

Python надає потужну екосистему бібліотек для аналізу даних. Вони дозволяють ефективно агрегувати та трансформувати дані про якість згенерованих тестів, обчислювати статистичні показники та готувати дані для візуалізації.

Нарешті, Python є кросплатформеним і може виконуватись на різних операційних системах, таких як Windows, Linux, macOS. Тож програми, написані на Python, не прив'язані до якоїсь конкретної платформи і можуть запускатись у різних середовищах.

Таким чином, Python є гнучким і доволі потужним інструментом. Це робить його зручним і ефективним засобом для дослідження, оцінки та вибору оптимальних стратегій і підходів. Python дозволяє швидко писати лаконічний, зрозумілий і модульний код, а також використовувати переваги численних бібліотек і фреймворків, розроблених спеціально для цих цілей [30].

3.3 Обґрунтування вибору синтетичних тестів

Для проведення експериментального дослідження ефективності використання LLM у генерації модульних тестів було обрано набір синтетичних тестів, що охоплюють різні аспекти розробки програмного забезпечення. Вибір синтетичних тестів замість реальних проектних кодів обумовлений кількома факторами: можливість контролювати складність коду та перевіряти конкретні аспекти генерації тестів, відсутність залежності від специфічного контексту проекту та бізнес-логіки, спрощення процесу оцінки якості згенерованих тестів завдяки чітко визначеним очікуваним результатам, можливість ізольованого тестування різних патернів програмування та граничних випадків.

Розглянемо обрані синтетичні тести та обґрунтування їх вибору:

– TimedCache - реалізація кешу з обмеженим часом життя елементів. Цей клас обраний для тестування здатності LLM генерувати тести для: конкурентного доступу до даних, роботи з часовими параметрами та їх граничними значеннями, перевірки правильності видалення застарілих елементів;

– ValidatedList - кастомна реалізація MutableList з валідацією елементів. Даний клас дозволяє оцінити здатність LLM створювати тести для: складних інтерфейсів з великою кількістю методів, функціонального програмування (використання lambda-функцій), обробки помилок та валідації даних;

– DataProcessor - асинхронний обробник даних з використанням корутин. Клас обраний для перевірки генерації тестів що охоплюють: асинхронне програмування та корутини, обробку складних бізнес-процесів; роботу з sealed класами та патерном Result;

– OrderStateMachine - реалізація патерну машини станів для управління замовленнями. Цей клас дозволяє оцінити якість тестів для: складної логіки переходів між станами, взаємодії з зовнішніми сервісами, обробки подій та валідації переходів;

– PalindromeChecker та FibonacciGenerator - прості алгоритмічні задачі, що дозволяють оцінити базові можливості LLM у генерації тестів для: перевірки граничних випадків, роботи з колекціями та строками, математичних обчислень;

– BoundaryValues - функція з умовною логікою для розрахунку знижок. Обрана для тестування здатності LLM визначати: граничні значення в умовних конструкціях, еквівалентні класи вхідних даних, обробку некоректних вхідних даних.

Такий набір синтетичних тестів забезпечує комплексну оцінку можливостей LLM у генерації Unit-тестів для різних шаблонів програмування, від простих алгоритмічних задач до складних асинхронних операцій та управління станами. Це дозволяє отримати об'єктивні метрики якості генерації тестів та порівняти ефективність різних моделей LLM.

3.4 Обґрунтування вибору налаштувань LLM

Для дослідження ефективності використання LLM у генерації модульних тестів було розроблено вісім типів початкових налаштувань LLM, що охоплюють різні підходи та стратегії взаємодії з моделями, а саме різні комбінації промптів та температур.

Базовий промпт (Simple) містить мінімальні інструкції для оцінки базових можливостей моделей. Представлення базового промпту наведено на рис. 3.1.

```
Provide me a set of unit test for the provided codebase
```

Рисунок 3.1 – Представлення базового промпту

Промпти з емуляцією ролей (LikeQA, LikeDEV) досліджують вплив контексту та професійної перспективи на якість генерації. Представлення LikeQA, LikeDEV промптів наведено на рис. 3.2.

```
##LikeQA
Provide me a set of unit test for the provided codebase
Think like QA

##LikeDEV
Provide me a set of unit test for the provided codebase
Think like Dev
```

Рисунок 3.2 – Представлення LikeQA, LikeDEV промптів

Для вивчення впливу параметру температури моделі на результати генеруємих тестів використовується базовий промпт з різними значеннями temperature: Focused (0.1) для отримання детермінованих результатів та Creative (0.9) для дослідження різноманітності генерації.

Детальні промпти містять розширені інструкції щодо структури тестів, використання патерну AAA (Arrange-Act-Assert) та вимоги до покриття граничних випадків та еквівалентних класів (Prompt_QA_Detailed_Focused,

Prompt_DEV_Detailed_Focused). Представлення Prompt_QA_Detailed_Focused, Prompt_DEV_Detailed_Focused промптів наведено на рис. 3.3.

```
##Prompt_QA_Detailed_Focused
Act like a software quality engineer
Write unit test for the provided class
Make sure AAA (Arrange-Act-Assert) pattern is applied
Make sure the Edge cases are covered (zero, negative tests, empty, null, large values)
Make sure the equivalence partitioning is covered
Make sure the boundary value analysis is covered

##Prompt_DEV_Detailed_Focused
Act like a developer
Write unit test for the provided class
Make sure AAA (Arrange-Act-Assert) pattern is applied
Make sure the Edge cases are covered (zero, negative tests, empty, null, large values)
Make sure the equivalence partitioning is covered
Make sure the boundary value analysis is covered
```

Рисунок 3.3 – Представлення промптів Prompt_QA_Detailed_Focused, Prompt_DEV_Detailed_Focused промптів

Промпт з розширеним бізнес-контекстом (Context) розроблений для оцінки впливу додаткової контекстної інформації на якість генерованих тестів. Представлення Context промпту наведено на рис. 3.4.

```
As a retail store manager,
I need an automated way to calculate customer discounts for our loyalty program
to ensure consistent discount application and increase customer satisfaction.
Business Context:
- Our store has noticed that large orders (₹1000+) are becoming more common
- We want to encourage medium-sized purchases (₹500-999) to compete with online retailers
- Customer feedback shows that predictable discounts increase repeat purchases
Acceptance Criteria:
- Apply 20% discount for purchases of ₹1000 or more to reward big spenders
- Apply 10% discount for purchases between ₹500-999 to encourage medium-sized purchases
- No discount for purchases below ₹500
- System must validate that purchase amounts are not negative
- Discounts should be calculated instantly at checkout
Technical Notes:
- Function should accept purchase amount as integer input
- Function should return discount percentage as integer
- Input validation should throw clear error message for negative amounts
- Implementation should be efficient for high-volume processing
Provide me a set of unit test for the provided codebase
```

Рисунок 3.4 – Представлення Context промпту

Кожен тип промпту створений для дослідження специфічних аспектів генерації та виявлення оптимальних підходів для різних моделей LLM, з особливою увагою до структури тестів та покриття граничних випадків.

3.5 Алгоритм розв'язання задачі

Алгоритм розв'язання задачі оцінки ефективності використання моделей LLM для генерації Unit-тестів включає наступні етапи:

Етап 1. Збір та підготовка даних:

– збір Kotlin коду для генерації тестів. Це може бути код з відкритих репозиторіїв, синтетичний код або код, написаний спеціально для дослідження;

– визначення критеріїв оцінки якості тестів. Вибір метрик, які будуть використовуватися для оцінки різних аспектів якості згенерованих Unit-тестів (покриття, якість, читабельність тощо);

– підготовка промптів (підказок) для LLM. Створення шаблонів запитів до LLM, які будуть містити інформацію про код, для якого потрібно згенерувати тести, та інструкції щодо формату і типу тестів.

Етап 2. Генерація Unit-тестів:

– вибір моделей LLM для дослідження (наприклад, ChatGPT, Gemini, CodeLlama);

– налаштування параметрів генерації (температура, максимальна кількість токенів тощо);

– запуск генерації Unit-тестів для кожної LLM та кожного фрагмента коду.

Етап 3. Аналіз згенерованих тестів:

– статичний аналіз коду тестів;

– оцінка якості згенерованих тестів за обраними метриками. Розрахунок значень метрик для кожного набору згенерованих тестів;

– порівняння результатів. Аналіз результатів оцінки якості тестів, згенерованих різними LLM, для визначення сильних та слабких сторін кожної моделі.

Етап 4. Формування звіту:

– візуалізація результатів. Створення графіків та діаграм для наочного представлення результатів оцінки та порівняння LLM;

– формування висновків. На основі аналізу результатів зробити висновки про ефективність використання різних LLM для генерації Unit-тестів.

3.6 Опис програми

В рамках дослідження було розроблено програмну систему для аналізу та порівняння ефективності різних моделей LLM у задачі генерації Unit-тестів для коду Kotlin. Система побудована на модульній архітектурі, що забезпечує гнучкість та розширюваність.

Основними складовими системи є три ключові модулі. Модуль генерації тестів містить базовий інтерфейс TestGenerator для взаємодії з різними LLM (ChatGPT, Gemini, CodeLlama), реалізує TestGenerator для кожної LLM через окремі класи, що інтегруються з відповідним API, включає TestGeneratorFactory для створення екземплярів генераторів та PromptProvider, що відповідає за формування промптів до LLM. Модуль аналізу якості тестів реалізований через TestQualityAnalyzer, що забезпечує комплексний підхід до оцінювання якості згенерованих тестів та використовує систему метрик, що охоплюють різні аспекти якості: BasicMetrics (кількість тестів, кількість тверджень), CoverageMetrics (покриття методів, покриття граничних випадків, покриття граничних значень), QualityMetrics (описовість назв тестів, різноманітність тверджень) та ReadabilityMetrics (використання лапок, наявність коментарів, середня довжина тесту). Модуль порівняння результатів представлений TestGeneratorComparator, який виконує порівняння результатів різних генераторів тестів, формує звіти у форматах json та txt, а результати зберігаються в об'єктах ComparisonResult.

Компоненти виконання включають MainProcess, що керує процесом гене-

рації тестів, створюючи окремі сесії для кожного експерименту, здійснює порівняльний аналіз згенерованих тестів та формує звіти. Система забезпечує збереження результатів у чіткій ієрархічній структурі: каталог result містить окремі підкаталоги для кожної LLM з результатами різних сесій, а каталог analysis_results містить аналітичні звіти з відміткою часу створення.

Результати зібраних аналітичних звітів обробляються та візуально подаються за допомогою LLMComparisonAnalyzer.

Основними перевагами розробленої системи є модульна архітектура, гнучка система оцінювання, детальний аналіз різних аспектів якості тестів та об'єктивне порівняння ефективності різних LLM. Система надає потужний інструментарій для дослідження якості генерації Unit-тестів різними LLM, сприяючи розвитку та вдосконаленню методів автоматизації тестування.

Висновки за розділом 3

У розділі було описано програмну реалізацію системи для дослідження ефективності використання моделей LLM у генерації Unit-тестів.

Обґрунтовано вибір мови Kotlin для проведення дослідження. Ключовими факторами стали чітка та регулярна структура синтаксису, що спрощує роботу LLM з кодом, широке представлення Kotlin у навчальних датасетах LLM, вбудована підтримка конструкцій для покращення читабельності тестів та можливості чіткого структурного аналізу коду для оцінки якості згенерованих тестів.

Обґрунтовано використання Python як мови для аналізу зібраних даних завдяки наявності потужних бібліотек для наукових обчислень та візуалізації даних, простому синтаксису для швидкої реалізації алгоритмів аналізу, багатим можливостям для обробки та трансформації даних, а також кросплатформеності розробленого рішення. Розроблено та обґрунтовано набір синтетичних тестів (TimedCache, ValidatedList, DataProcessor та інші), що охоплюють різні аспекти програмування - від базових алгоритмів до асинхронних операцій. Також ство-

рено вісім типів вхідних параметрів LLM для дослідження впливу промптів та температурних параметрів моделей на якість генерації.

Розроблено чіткий алгоритм розв'язання задачі, що включає етапи збору та підготовки даних, генерації Unit-тестів різними LLM моделями, аналізу згенерованих тестів та формування звітів з результатами дослідження. Для кожного етапу визначено конкретні кроки та очікувані результати.

Створено програмну систему з модульною архітектурою, що складається з трьох ключових модулів: генерації тестів, аналізу якості та порівняння результатів. Система забезпечує гнучкість через можливість розширення підтримуваних LLM моделей, комплексний аналіз якості згенерованих тестів за різними метриками та зручне збереження й візуалізацію результатів порівняння.

Розроблена система надає потужний інструментарій для об'єктивного порівняння ефективності різних LLM у задачі генерації Unit-тестів та може бути використана для подальших досліджень у цій галузі.

4 РЕЗУЛЬТАТИ ОБЧИСЛЮВАЛЬНОГО ЕКСПЕРИМЕНТУ ТА ЇХ АНАЛІЗ

4.1 Аналіз результатів генерації Unit-тестів при різних налаштуваннях великих мовних моделей

Для проведення порівняльного аналізу ефективності різних підходів до формування промптів та можливостей великих мовних моделей було обрано синтетичний тестовий сценарій - функцію розрахунку знижок calculateDiscount. Дана функція реалізує просту але показову бізнес-логіку нарахування знижок в залежності від суми покупки. Код реалізації функції calculateDiscount наведено на рис. 4.1.

```
fun calculateDiscount(purchaseAmount: Int): Int {
  require(purchaseAmount >= 0) { "Purchase amount cannot be negative." }
  return when {
    purchaseAmount >= 1000 -> 20 // 20% discount for purchases over 1000
    purchaseAmount >= 500 -> 10 // 10% discount for purchases over 500
    else -> 0 // No discount
  }
}
```

Рисунок 4.1 – Код реалізації функції calculateDiscount

Вибір даної функції обумовлений декількома важливими факторами. По-перше, функція має чітку бізнес-логіку розрахунку знижок з фіксованими граничними значеннями 500 та 1000, що дозволяє оцінити здатність LLM виявляти та тестувати граничні випадки. По-друге, наявність валідації вхідних даних через оператор require створює можливість для тестування обробки некоректних вхідних значень. По-третє, використання конструкції when для розгалуженої логіки дозволяє перевірити покриття різних шляхів виконання коду. Додатково, простота та зрозумілість функціоналу дає змогу зосередитись на оцінці якості згенерованих тестів, а не на складності розуміння бізнес-логіки.

В рамках дослідження було протестовано вісім різних стратегій формування промптів, кожна з яких мала свої особливості та цілі.

При використанні Simple промпту моделі демонстрували базовий рівень розуміння задачі тестування, але часто пропускали важливі граничні випадки. GPT показав середній результат з оцінкою 14.4, в той час як Gemini досяг кращих результатів з оцінкою 19.0, що свідчить про кращу базову здатність до розуміння контексту тестування. Загальні результати представлені у порівняльній таблиці (табл. 4.1).

Таблиця 4.1 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різни LLM та Simple промпту

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	7	7
Загальна кількість тверджень	17	11	10
Час генерації (мс)	6251	1429	17470
Базова оцінка	9.8	7.2	7
Оцінка покриття	3.2	3.6	2.8
Оцінка якості	2	3	2
Оцінка читабельносьті	4	4	2.6
Загальна оцінка	19	17.8	14.4

Промпт з емуляцією ролі тестувальника (LikeQA) покращив загальну якість тестів, особливо в частині перевірки граничних випадків. Gemini досяг оцінки 18.5, показавши хороші результати в метриках якості тестів. CodeLlama з оцінкою 17.5 продемонструвала помітне покращення в документуванні тестів та їх читабельності. Загальні результати представлені у порівняльній таблиці (табл. 4.2).

Таблиця 4.2 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різни LLM та LikeQA промпту

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	7	7
Загальна кількість тверджень	18	10	11
Час генерації (мс)	4565	7400	13978
Базова оцінка	9.9	7.1	7.3
Оцінка покриття	3.2	3.4	3.2
Оцінка якості	2	3	2
Оцінка читабельності	2.3	3	2.3
Загальна оцінка	18.5	17.5	15.9

При використанні промпту з емуляцією ролі розробника (LikeDEV) моделі показали близькі до попередніх результати з деякими відмінностями. GEMINI отримав найвищу оцінку 17.9, показавши особливо хороші результати в базових метриках 9.7 завдяки високій щільності тверджень на тест 3.3. CodeLLAMA досягла оцінки 17.2, продемонструвавши збалансовані показники з особливо хорошими результатами в покритті граничних випадків 3.6 та якості тестів 2.7. GPT показав найнижчий результат з оцінкою 14.7, хоча мав прийнятні показники базових метрик 7.3 та покриття 3.2, але поступився в якості 1.7 та читабельності 2.6 тестів. Цікаво відмітити, що GEMINI виявився найшвидшим у генерації тестів 4838 мс, тоді як GPT витратив майже вдвічі більше часу 9853 мс. Загальні результати представлені у порівняльній таблиці (табл. 4.3).

Промпт з емуляцією промпту (Focused) з параметром температури = 0.1 призвів до більш консистентних результатів серед всіх моделей. Gemini досяг оцінки 18.7, показавши високу якість базових метрик. GPT з оцінкою 16.7 продемонстрував покращення в частині документації тестів, а CodeLlama (16.0) показала стабільні результати в метриках якості. Загальні результати представлені у порівняльній таблиці (табл. 4.4).

Таблиця 4.3 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та LikeDEV промпту

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	7	6
Загальна кількість тверджень	17	11	10
Час генерації (мс)	4843	6198	9883
Базова оцінка	9.7	7.4	7.3
1	**2**	**3**	**4**
Оцінка покриття	3.2	3.6	3.2
Оцінка якості	2	2.7	1.7
Оцінка читабельносьті	3.1	3.5	2.6
Загальна оцінка	17.9	17.2	14.7

Таблиця 4.4 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та Focused промпту

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	6	7
Загальна кількість тверджень	17	12	10
Час генерації (мс)	5135	9007	6466
Базова оцінка	9.8	7.8	7
Оцінка покриття	3.3	3.1	3.6
Оцінка якості	2	2.3	2.3
Оцінка читабельносьті	3.5	3.5	3.1
Загальна оцінка	18.7	16.7	16

Використання креативного промпту (Creative) з параметром температури = 0.9 призвело до найбільш різноманітних результатів. Gemini досяг найвищої оцінки 19.6, згенерувавши найбільш повний набір тестових сценаріїв. CodeLlama з оцінкою 18.0 показала хороші результати в покритті граничних випадків, а GPT (16.1) продемонстрував стабільні показники в базових метриках. Загальні результати представлені у порівняльній таблиці (табл. 4.5).

Таблиця 4.5 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та Creative промпту

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	6	7	8
Загальна кількість тверджень	19	11	10
Час генерації (мс)	5710	6700	15341
Базова оцінка	10	7.4	7.3
Оцінка покриття	3.6	3.6	3.2
Оцінка якості	2	3	2
Оцінка читабельносьті	4	4	3.5
Загальна оцінка	19.6	18	16.1

Детальний промпт з акцентом на тестування та детермінованим результатом (Prompt_QA_Detailed_Focused) призвів до значного покращення якості тестів у всіх моделей. Gemini досяг оцінки 19.0, показавши відмінні результати в усіх метриках. CodeLlama (17.8) продемонструвала найкраще покриття граничних випадків, а GPT (14.4) покращив показники читабельності тестів. Загальні результати представлені у порівняльній таблиці (табл. 4.6).

Таблиця 4.6 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та промпту Prompt_QA_Detailed_Focused

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	7	7
Загальна кількість тверджень	17	11	10
Час генерації (мс)	6265	1437	17554
Базова оцінка	9.8	7.2	7
Оцінка покриття	3.2	3.6	2.8
Оцінка якості	2	3	2
Оцінка читабельносьті	4	4	2.6
Загальна оцінка	19	17.8	14.4

Деталізований промпт з позиції розробника та детермінованим результатом (Prompt_DEV_Detailed_Focused) показав найбільш збалансовані результати серед усіх стратегій. CodeLlama досягла найвищої оцінки 18.4, показавши відмінні результати в усіх метриках. GPT з оцінкою 17.8 продемонстрував найкраще покриття коду, а Gemini (16.8) показав високу якість базових метрик. Загальні результати представлені у порівняльній таблиці (табл. 4.7).

Таблиця 4.7 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та промпту Prompt_DEV_Detailed_Focused

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	8	9	8
Загальна кількість тверджень	11	11	10
Час генерації (мс)	6268	15999	5843
Базова оцінка	7.4	7.8	7.2
Оцінка покриття	4	4	3.6
Оцінка якості	3	2	2
Оцінка читабельносьті	4	4	4
Загальна оцінка	18.4	17.8	16.8

Промпт з розширеним бізнес-контекстом (Context) призвів до покращення релевантності тестових сценаріїв. Gemini досяг оцінки 18.6, показавши найкраще розуміння бізнес-вимог. GPT з оцінкою 16.0 продемонстрував покращення в документації тестів, а CodeLlama (15.7) показала стабільні результати в метриках якості. Загальні результати представлені у порівняльній таблиці (табл. 4.8).

Таблиця 4.8 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM та Context промпту.

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	6	6	7
Загальна кількість тверджень	11	17	13
Час генерації (мс)	14733	5533	4859
Базова оцінка	7.1	9.4	7.8
Оцінка покриття	3.3	3.2	3.6
Оцінка якості	2	2	1.7
Оцінка читабельносьті	3.5	4	2.6
Загальна оцінка	16	18.6	15.7

На основі проведеного аналізу можна зробити висновок, що найбільш ефективною стратегією є використання деталізованого промпту з позиції розробника (Prompt_DEV_Detailed_Focused), який забезпечує чіткі інструкції щодо патернів тестування та очікуваного покриття. Така стратегія дозволяє отримати найбільш збалансовані результати з високими показниками в усіх метриках якості. Додавання бізнес-контексту може бути корисним для покращення релевантності тестових сценаріїв, але потребує балансу для уникнення надмірного ускладнення тестів.

При виборі моделі LLM варто враховувати, що Gemini показує найкращі результати при роботі з бізнес-контекстом та креативними промптами, CodeLlama демонструє стабільно високе покриття граничних випадків та якість документації, а GPT забезпечує збалансовані результати з хорошою читабельністю тестів. Комбінація правильно обраної моделі та оптимальної стратегії формування промпту дозволяє досягти найкращих результатів у автоматизованій генерації модульних тестів.

Аналіз ефективності різних промптів для кожної моделі показав суттєві відмінності у їх результативності. Модель GEMINI продемонструвала найвищу ефективність при використанні Creative промптур з оцінкою 19.6 балів. Також високі результати було отримано при використанні Simple промпту та

Prompt_QA_Detailed_Focused, обидва з 19.0 балів. Така висока ефективність з різними типами промптів свідчить про здатність моделі добре розуміти контекст тестування та генерувати якісні тести навіть при мінімальних інструкціях.

CodeLLAMA показала найкращий результат при використанні Prompt_DEV_Detailed_Focused з оцінкою 18.4 бали. Дещо нижчі, але все ще високі результати було отримано з Creative промптом (18.0) та Simple промптом (17.8). Такий розподіл оцінок вказує на те, що модель найефективніше працює з детальними технічними інструкціями, особливо коли вони сформульовані з позиції розробника. Це підтверджує спеціалізацію CodeLLAMA на роботі з програмним кодом.

Для моделі GPT Prompt_DEV_Detailed_Focused виявився найефективнішим з оцінкою 17.8 балів. LikeDEV та Focused промпти показали однаковий результат у 16.7 балів, а Creative промпт забезпечив 16.1 бал. Такий розподіл результатів свідчить про те, що GPT потребує більш структурованого підходу та чітких інструкцій для генерації якісних тестів.

Цікавим спостереженням є те, що додавання бізнес-контексту через Context промпт не призвело до покращення результатів у жодної з моделей. Це може свідчити про те, що надмірна контекстуалізація не є необхідною для ефективної генерації модульних тестів. Натомість, ключовим фактором є правильний вибір рівня технічної деталізації та стилю формулювання промпту відповідно до особливостей конкретної моделі.

4.2 Аналіз результатів синтетичних тестів

На основі проведеного аналізу різних стратегій формування промптів для всіх трьох моделей, Prompt_DEV_Detailed_Focused демонструє найбільш стабільні та високі результати. При використанні цього промпту CodeLLAMA досягла оцінки 18.4, GPT отримав 17.8, а GEMINI показав результат 16.8 балів. Хоча для GEMINI це не найвищий показник, але сумарна оцінка всіх трьох моде-

лей з цим промптом є найвищою. Структура Prompt_DEV_Detailed_Focused включає емуляцію ролі розробника, чіткі вказівки щодо застосування патерну AAA (Arrange-Act-Assert), вимоги до покриття граничних випадків та граничних значень, а також інструкції щодо еквівалентного розбиття. Такий рівень деталізації забезпечує необхідний баланс між технічною точністю та зрозумілістю для всіх моделей. Важливо зазначити, що хоча для окремих моделей інші промпти можуть давати кращі результати (наприклад, Creative промпт для GEMINI з оцінкою 19.6), з точки зору універсальності та стабільності результатів Prompt_DEV_Detailed_Focused є найбільш оптимальним вибором для використання з будь-якою з розглянутих LLM моделей.

При тестуванні простої валідації та текстових операцій (NullPointerExceptions, getFullName) промпт показав стабільні результати. CodeLLAMA досягла оцінок 16.6 -18.4, продемонструвавши особливу ефективність у покритті граничних випадків. GPT показав схожі результати з оцінками 16.8 -17.4, маючи хороший баланс між покриттям коду та читабельністю тестів. GEMINI отримав оцінки 16.6 -17.0, зберігаючи високу якість базових метрик. Загальні результати представлені у порівняльній таблиця (табл. 4.9, табл. 4.10).

Таблиця 4.9 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на синтетичному тесті NullPointerExceptions

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	8	6	8
Загальна кількість тверджень	11	8	9
Час генерації (мс)	5821	1963	15047
Базова оцінка	7.4	6,2	7
Оцінка покриття	3.6	4.4	4.4
Оцінка якості	2	2	2
Оцінка читабельносьті	4	4	4
Загальна оцінка	17	16.6	17.4

Таблиця 4.10 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на синтетичному тесті getFullName

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	7	8	8
Загальна кількість тверджень	9	11	10
Час генерації (мс)	8503	3192	16138
Базова оцінка	6.6	7.4	7.2
Оцінка покриття	4	4	3.6
Оцінка якості	2	3	2
Оцінка читабельносьті	4	4	4
Загальна оцінка	16.6	18.4	16.8

Для більш складного алгоритмічного тестування (FibonacciGenerator, PalindromeChecker) всі моделі показали вищі результати. GPT досяг оцінки 21.4 з відмінними базовими метриками та якістю тестів. CodeLLAMA отримала 20.8-21.2, демонструючи найкраще покриття граничних випадків. GEMINI показав результат 20.6 - 21.4, з високою читабельністю згенерованих тестів. Загальні результати представлені у порівняльній таблицях (табл. 4.11, табл. 4.12).

Таблиця 4.11 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на тесті FibonacciGenerator

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	5	10	11
Загальна кількість тверджень	15	15	25
Час генерації (мс)	6405	1766	13453
Базова оцінка	9	9	11
Оцінка покриття	4.4	4.8	2.4
Оцінка якості	4	3	4
Оцінка читабельносьті	4	4	4
Загальна оцінка	21.4	20.8	21.4

Таблиця 4.12 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на синтетичному тесті PalindromeChecker

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	12	11	12
Загальна кількість тверджень	12	15	13
Час генерації (мс)	5925	12759	18868
Базова оцінка	9.2	9.2	9.2
Оцінка покриття	4.4	4	3.6
Оцінка якості	3	4	2
Оцінка читабельносьті	4	4	4
Загальна оцінка	20.6	21.2	18.8

При тестуванні асинхронної обробки даних (DataProcessor) моделі також показали високі результати. CodeLLAMA досягла оцінки 21.0 з відмінними базовими метриками. GEMINI отримав 19.0 балів, забезпечуючи хороший баланс між всіма метриками. GPT показав схожий результат 19.0, але з кращим покриттям граничних випадків. Загальні результати представлені у порівняльній таблицях (табл. 4.13).

Таблиця 4.13 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на синтетичному тесті DataProcessor

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	8	10	7
Загальна кількість тверджень	17	25	12
Час генерації (мс)	9931	13490	16026
Базова оцінка	9	11	7.6
Оцінка покриття	4	4	4.4
Оцінка якості	2	2	3
Оцінка читабельносьті	4	4	4
Загальна оцінка	19	21	19

Тестування складних структур даних (ValidatedList) виявило найбільшу варіативність результатів. CodeLLAMA досягла найвищої оцінки 28.2, показавши відмінні результати в усіх метриках. GPT отримав 22.2 бали з хорошим покриттям граничних випадків. GEMINI показав результат 20.0, маючи найкращі базові метрики. Загальні результати представлені у порівняльній таблицях (табл. 4.14).

Таблиця 4.14 – Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів на синтетичному тесті ValidatedList

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	10	26	9
Загальна кількість тверджень	29	40	23
Час генерації (мс)	10254	2901	15781
Базова оцінка	11.8	18.6	10.4
Оцінка покриття	1.2	1.6	4.8
Оцінка якості	3	4	3
Оцінка читабельносьті	4	4	4
Загальна оцінка	20	28.2	22.2

Найскладнішим виявилось тестування машини станів (OrderStateMachine), де результати були нижчими. CodeLLAMA отримала 18.0 балів, GPT - 14.6, а GEMINI - 13.6. Це вказує на складність генерації тестів для складних поведінкових патернів. Загальні результати представлені у порівняльній таблицях (табл. 4.15).

Загальний аналіз показує, що промпт Prompt_DEV_Detailed_Focused найбільш ефективний для тестування алгоритмічних задач та структур даних, де чітко визначені вхідні дані та очікувані результати. При цьому він демонструє деякі обмеження при тестуванні складних поведінкових патернів та асинхронних операцій. CodeLLAMA загалом показує найбільш стабільні результати, особливо в покритті граничних випадків, тоді як GPT та GEMINI мають більшу варіативність результатів в залежності від типу задачі. На основі загальних результатів аналізу можна зробити комплексний аналіз ефективності різних моделей

LLM при використанні промпту Prompt_DEV_Detailed_Focused. Порівняльний аналіз результатів оцінки якості генерації Unit-тестів LLM наведено на рис. 4.1.

Таблиця 4.15– Порівняльна таблиця загальних метрик оцінки генерації Unit-тестів з використанням різних LLM на синтетичному тесті OrderStateMachine

	Модель		
Метрика	**GEMINI**	**CODELLAMA**	**GPT**
Загальна кількість тестів	10	14	6
Загальна кількість тверджень	9	14	7
1	**2**	**3**	**4**
Час генерації (мс)	15659	5655	16351
Базова оцінка	7.8	10.4	5.8
Оцінка покриття	0.8	1.6	0.8
Оцінка якості	1	2	4
Оцінка читабельносьті	4	4	4
Загальна оцінка	13.6	18	14.6

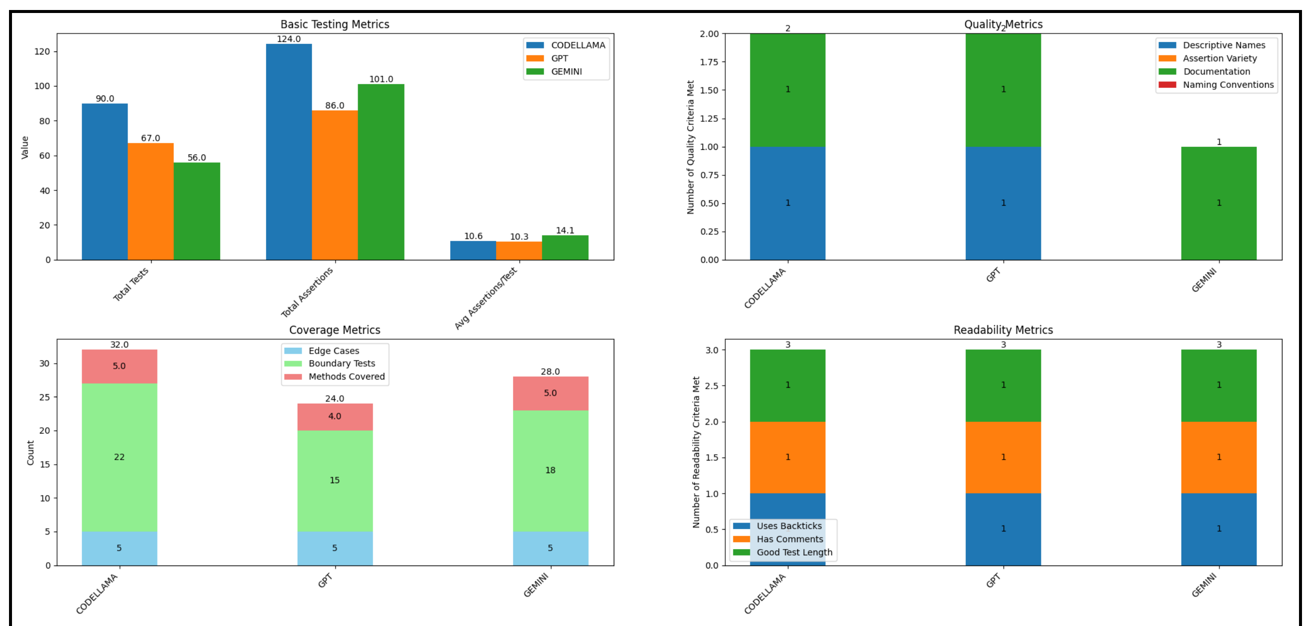

Рисунок 4.1 – Порівняльний аналіз результатів оцінки якості генерації Unit-тестів великими мовними моделями

CodeLLAMA досягла найвищої сумарної оцінки 102.0 бали, що підтвер-

джує її високу ефективність при генерації модульних тестів. Модель згенерувала найбільшу кількість тестів (90) та тверджень (124), забезпечивши високу щільність перевірок (10.63 тверджень на тест). Особливо варто відзначити значне покриття граничних випадків (25 сценаріїв) та граничних тестів (22 тести), що свідчить про глибоке розуміння тестованого коду. При цьому модель зберегла високу читабельність тестів із середньою довжиною методу 16 рядків.

GEMINI показав другий результат із загальною оцінкою 86.0 балів. Хоча модель згенерувала менше тестів (56), вона забезпечила найвищу щільність перевірок (14.13 тверджень на тест), що свідчить про комплексність кожного тестового сценарію. Модель також показала хороші результати в покритті граничних випадків (19 сценаріїв) та граничних тестів (18 тестів). Середня довжина тестового методу склала 21 рядок, що є найвищим показником серед всіх моделей.

GPT отримав загальну оцінку 82.6 бали, згенерувавши 67 тестів з 86 твердженнями (10.28 тверджень на тест). Модель показала дещо нижчі результати в покритті граничних випадків (15 сценаріїв) та граничних тестів (15 тестів), але забезпечила найкращу компактність тестових методів із середньою довжиною 14 рядків.

З точки зору часу генерації, найшвидшою виявилася CodeLLAMA (14160 мс), за нею слідує GEMINI (75552 мс), а GPT витратив найбільше часу (122488 мс). Такий розподіл швидкодії може бути важливим фактором при виборі моделі для практичного використання.

Результати підтверджують, що CodeLLAMA найкраще підходить для генерації модульних тестів, особливо коли важлива повнота покриття та якість тестів. GEMINI може бути хорошим вибором, коли потрібні комплексні тестові сценарії, а GPT доцільно використовувати для генерації компактних та читабельних тестів. При цьому всі моделі демонструють прийнятний рівень якості для практичного використання в процесах тестування програмного забезпечення.

Висновки за розділом 4

У розділі проведено комплексне експериментальне дослідження ефективності використання великих мовних моделей для генерації модульних тестів. Для забезпечення об'єктивності оцінки було розроблено набір синтетичних тестів, що охоплюють різні аспекти розробки програмного забезпечення: від простих алгоритмічних задач до складних асинхронних операцій та управління станами.

Порівняльний аналіз різних стратегій формування промптів показав, що найбільш збалансовані результати забезпечує Prompt_DEV_Detailed_Focused, який включає емуляцію ролі розробника та чіткі вказівки щодо застосування патернів тестування. При цьому встановлено, що надмірна контекстуалізація через додавання бізнес-контексту не призводить до покращення результатів генерації тестів.

При дослідженні ефективності різних LLM моделей виявлено їх специфічні особливості. GEMINI показує найкращі результати при роботі з креативними промптами (оцінка 19.6), CodeLLAMA демонструє стабільно високі показники з технічними промптами (оцінка 18.4), а GPT забезпечує збалансовані результати з детальними інструкціями (оцінка 17.8).

Аналіз результатів генерації тестів для різних типів задач виявив, що найвищу ефективність моделі демонструють при тестуванні алгоритмічних задач та структур даних (оцінки 20.6-28.2). При цьому найскладнішим виявилося тестування поведінкових патернів, таких як машини станів (оцінки 13.6-18.0), що вказує на необхідність подальшого вдосконалення підходів до генерації тестів для складних архітектурних рішень.

В цілому проведене дослідження підтверджує практичну цінність використання LLM для автоматизації процесів тестування та надає рекомендації щодо вибору оптимальної стратегії формування промптів в залежності від специфіки задачі та обраної моделі.

ВИСНОВКИ

У роботі проведено комплексне дослідження підходів до використання великих мовних моделей для оптимізації процесів тестування програмного забезпечення при переході від монолітної до мікросервісної архітектури.

На основі системного аналізу особливостей тестування в різних архітектурних підходах розроблено комплексну систему метрик для оцінки якості згенерованих модульних тестів. Система включає базові метрики (кількість тестів, щільність перевірок), метрики покриття коду (покриття методів, граничних випадків, граничних значень), показники якості (описовість назв, різноманітність тверджень) та читабельності (використання документації, структурованість коду).

Створено програмну систему для порівняльного аналізу ефективності різних LLM моделей (ChatGPT, Gemini, CodeLlama) у задачі генерації Unit-тестів. Система забезпечує автоматизовану генерацію тестів, оцінку їх якості та формування аналітичних звітів. Модульна архітектура системи та гнучкі налаштування параметрів генерації дозволяють легко розширювати функціональність та адаптувати її для різних проектів.

Експериментальне дослідження на наборі синтетичних тестів, що охоплюють різні аспекти розробки, показало найвищу ефективність моделі CodeLlama із сумарною оцінкою 102.0 бали. Модель згенерувала найбільшу кількість тестів (90) та тверджень (124), забезпечила найкраще покриття граничних випадків (25 сценаріїв) та показала найвищу швидкість генерації (14160 мс). GEMINI продемонстрував найвищу щільність перевірок (14.13 тверджень на тест), а GPT забезпечив найкращу компактність тестових методів.

За результатами дослідження визначено, що найбільш збалансовані результати забезпечує промпт Prompt_DEV_Detailed_Focused з емуляцією ролі розробника та чіткими вказівками щодо застосування патернів тестування. При цьому найвища ефективність спостерігається при тестуванні алгоритмічних задач (оцінки 20.6-28.2), тоді як тестування поведінкових патернів виявилося найскладнішим (оцінки 13.6-18.0).

Практична цінність роботи полягає у створенні методики ефективного використання LLM для автоматизації процесів тестування програмного забезпечення. Розроблені підходи та програмна система дозволяють значно прискорити розробку модульних тестів при збереженні їх високої якості. Результати дослідження показують, що CodeLlama найкраще підходить для генерації критичних тестів, GEMINI ефективний для створення комплексних сценаріїв, а GPT доцільно використовувати для генерації компактних та читабельних тестів.

Подальші дослідження можуть бути спрямовані на розширення підтримки інших типів тестування, інтеграцію з CI/CD pipeline, вдосконалення метрик оцінки якості тестів та адаптацію розроблених підходів для специфічних вимог мікросервісної архітектури, зокрема тестування міжсервісної взаємодії та розподілених транзакцій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Davis A. Monolithic vs Microservices Architecture: Pros, Cons and Which to Choose. OpenLegacy Blog. 2023. URL: https://www.openlegacy.com/blog/monolithic-application (дата звернення: 20.10.2024).

2. Dushenin O. Monolithic Architecture. Advantages and Disadvantages. Datamify. 2021. URL: https://datamify.medium.com/monolithic-architecture-advantages-and-disadvantages-e71a603eec89 (дата звернення: 15.10.2024).

3. Онищенко Р., Котенко Н., Жирова Т. Роль та ефективність засобів штучного інтелекту в тестуванні програмного забезпечення. *Інформаційні технології та суспільство.* 2024. № 2 (13). С. 66–70. DOI: https://doi.org/10.32689/maup.it.2024.2.10

4. Тестувальники, які володіють інструментами ШІ, замінять тих, хто їх не використовує. Anywhere Club. URL: https://aw.club/global/uk/blog/how-to-use-artificial-intelligence-in-testing (дата звернення: 06.10.2024).

5. Тестування програмного забезпечення з використанням штучного інтелекту. Delivering excellence with professionals at Brainberry.ua. URL: https://brainberry.ua/uk/newsroom/blog/software-testing-using-ai (дата звернення: 09.10.2024).

6. Свергун С., Жаровський Р. Тестування програмного забезпечення побудованого на мікросервісній архітектурі. *Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології»*, 7-8 груд. 2022 р. Тернопіль : ТНТУ, 2022. С. 92.

7. Свергун С., Жаровський Р. Тестування програмного продукту, побудованого на мікросервісній архітектурі на основі BDD. *Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології»*, 7-8 груд. 2022 р. Тернопіль : ТНТУ, 2022. С. 93.

8. Newman S. Building Microservices : Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 616 p.

9. Фаулер М., Льюїс Дж. Мікросервіси. URL: https://martinfowler.com/articles/microservices.html (дата звернення: 30.10.2024).

10. Уоттс С., Шифф Л. Огляд архітектури моноліту проти мікросервісів. URL: https://www.bmc.com/blogs/microservices-architecture/ (дата звернення: 07.10.2024).

11. Zhamak D. How to break a Monolith into Microservices. 2014. URL: https://martinfowler.com/articles/break-monolith-into-microservices.html (дата звернення: 29.09.2024).

12. Fowler M. Testing Strategies in a Microservice Architecture. URL: https://martinfowler.com/articles/microservice-testing/ (дата звернення: 01.10.2024).

13. Gos K., Zabierowski W. The Comparison of Microservice and Monolithic Architecture. *IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. Lviv, 2020. P. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514

14. Kankaria H. When to do Manual Testing and when to do Automated Testing. URL: https://automatedtesting.quora.com/When-to-do-Manual-Testing-and-when-to-do-Automated-Testing (дата звернення: 08.10.2024).

15. Як ШІ змінить тестування програмного забезпечення. Visure Solutions. URL: https://visuresolutions.com/uk/blog/ways-ai-will-change-software-testing/ (дата звернення: 10.10.2024).

16. Evans E. Domain-Driven Design : Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003. 560 p.

17. Beck K. Test Driven Development: By Example. Addison-Wesley Professional, 2021. 242 p.

18. Osherove R. The Art of Unit Testing : with examples in C#. Manning, 2013. 375 p.

19. Black R. Critical Testing Processes: Plan, Prepare, Perform, Perfect. Addison-Wesley Professional, 2003. 608 p.

20. Gill N. S. Software Quality Management Techniques and Best Practices. URL: https://www.xenonstack.com/insights/what-is-software-quality (дата звернення: 21.10.2024).

21. Kilonzi F. How AI Is Revolutionizing Software Testing and Test Automation. SauceLabs. URL: https://saucelabs.com/resources/blog/ai-test-automation (дата звернення: 23.10.2024).

22. Infosys Limited. Artificial Intelligence-led quality Insurance. URL: https://www.infosys.com/IT-services/validation-solutions/service-offerings/Documents/machine-learning-qa.pdf (дата звернення: 21.10.2024).

23. Elijah J. Automation of Requirement Analysis in Software Engineering. *International Journal on Recent and Innovation Trends in Computing and Communication*. 2017. Vol. 5. Is. 5. P. 1173–1188.

24. Штучний інтелект та програмне забезпечення: плюси від інтеграції. Об'єднання Intecracy Group. URL: https://intecracy.com/ua/news/shtuchnyi-intelekt-ta-prohramne-zabezpechennia-pliusy-vid-intehratsii.html (дата звернення: 11.10.2024).

25. Wang J., Huang Y., Chen C., Liu Z., Wang S., Wang Q. Software Testing with Large Language Models: Survey, Landscape, and Vision. URL: https://arxiv.org/pdf/2307.07221 (дата звернення: 10.10.2024).

26. Третякова М. С. Застосування штучного інтелекту в тестуванні програмного забезпечення. *Радіоелектроніка та молодь у XXI столітті : матеріали 23 Міжнар. молодіж. форуму*, 16–18 квітня 2019 р. Харків : ХНУРЕ, 2019. Т. 6. С. 37–38.

27. Власенко В. Що може робити ШІ на вашому проєкті – горизонтально і вертикально. Досвід архітектора. URL: https://dou.ua/forums/topic/44863 (дата звернення: 10.10.2024).

28. Madhava K., Gaur B., Verma A., Jalote P. Using LLMs in Software Requirements Specifications: An Empirical Evaluation. URL: https://arxiv.org/pdf/2404.17842 (дата звернення: 12.10.2024).

29. Ashton P. Why agile development races ahead of traditional testing. URL: https://www.computerweekly.com/feature/Why-agile-development-races-ahead-of-traditional-testing (дата звернення: 10.01.2025).

30. What is unit testing? https://aws.amazon.com/what-is/unit-testing/ (дата звернення: 09.01.2025).

31. Kotlin Native. Jetbrains. URL: https://kotlinlang.org/docs/native-overview.html (дата звернення: 05.01.2025).

32. Підручник з Python. URL: https://docs.python.org/uk/3/tutorial/index.html (дата звернення: 10.01.2025).

33. Остапов О. Складнощі тестування мікросервісів та що з ними робити. DOU. URL: https://dou.ua/lenta/articles/difficulties-in-microservices-testing/ (дата звернення: 03.01.2025).

34. Waterfall Model for Software Development and Testing Teams. QATestLab. URL: https://qatestlab.com/resources/knowledge-center/waterfall-process/ (дата звернення: 06.01.2025).

35. Bluein C. What is Agile Software Development? Openxcell. URL: https://www.openxcell.com/blog/what-is-agile-software-development/ (дата звернення: 02.01.2025).

36. Palamarchuk S. Why Shift-Left Testing? Pros and Cons. URL: https://www.testim.io/blog/shift-left-testing/ (дата звернення: 11.01.2025).

37. Brown B. The Agile Testing Pyramid. Agile Coach Journal. URL: https://www.agilecoachjournal.com/2014-01-28/the-agile-testing-pyramid (дата звернення: 11.01.2025).

38. Security and ethical aspects of using machine learning and artificial intelligence. *Публічне управління XXI століття: нові виклики і трансформації в умовах війни*. Харків : ХНУ імені В. Н. Каразіна, 2024. С.474 – 478.

39. Safe and ethical aspects of using machine learning and artificial intelligence. Social Development Towards Values Ethics – Technology. *Society: Proceedings of the 10th International Interdisciplinary Scientific Conference.* 24-29 Sep. 2024. Wisla : Silesian University of Technology, P. 37–38.

40. Using artificial intelligence to accelerate the errors root causes analysis in applications built on microservices architecture. Conference Proceedings of the III International Scientific & Practical Conference. Харків : ХНУРЕ, 2024. С. 198.

41. Зміни навичок тестувальників під час використання великих мовних моделей у разі оцінки якості програмного забезпечення. *Інформаційні технології в освітньому процесі ЗВО*. 27 лист. 2024, Харків : ХНАДУ, 2024. С. 474–478.