

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Факультет програмних систем та прикладної математики
Кафедра системного програмування і спеціалізованих
комп'ютерних систем**

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«__» _____ 20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Системне програмування та спеціалізовані комп'ютерні системи»

спеціальності 123 «Комп'ютерна інженерія»

**на тему: «Система автоматизованої генерації unit-тестів на основі великих
мовних моделей»**

Виконав:

студент IV курсу, групи КВ-21

Ярич Роман Русланович _____

Керівник:

ст. викладач каф. СП і СКС, к.т.н.

Коляда Константин Вячеславович _____

Консультант з нормконтролю:

доцент каф. СП і СКС, к.т.н., доцент

Клятченко Ярослав Михайлович _____

Рецензент:

доцент каф. ОТ, к.т.н., доцент

Марковський Олександр Петрович _____

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет програмних систем та прикладної математики
Кафедра системного програмування і спеціалізованих
комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«__» _____ 20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Ярич Роман Русланович

1. Тема проєкту «Система автоматизованої генерації unit-тестів на основі великих мовних моделей», керівник проєкту Коляда К.В., кандидат технічних наук, ст. викл. кафедри системного програмування і спеціалізованих комп'ютерних систем, затверджені наказом по університету від «28» травня 2026р. № 1984-с

2. Термін подання студентом проєкту 15 червня 2026 р.

3. Вихідні дані проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки

- аналіз існуючих рішень;
- розробка системи розпаралелення алгоритмів на 3 рівні;
- тестування;
- керівництво користувача.

Перелік графічного матеріалу (із зазначенням обов'язкових креслень, плакатів, презентацій, тощо): Структурні схеми і схеми алгоритмів

5. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., к.т.н., доцент каф. СП і СКС		

6. Дата видачі завдання: 24 жовтня 2025р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Терміни виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	24.10.2025	
2.	Розроблення та узгодження технічного завдання	11.11.2025	
3.	Аналіз існуючих рішень	17.01.2026	
4.	Підготовка матеріалів першого розділу дипломного проєкту	15.02.2026	
5.	Розробка апаратного забезпечення	17.03.2026	
6.	Підготовка матеріалів другого розділу дипломного проєкту	19.03.2026	
7.	Розробка програмного забезпечення	26.03.2026	
8.	Підготовка матеріалів третього розділу дипломного проєкту	07.04.2026	
9.	Підготовка графічної частини дипломного проєкту	23.04.2026	
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	29.04.2026	
11.	Оформлення документації дипломного проєкту	15.05.2026	
12.	Попередній огляд матеріалів диплому на кафедрі	27.05.2026	

Студент _____

Роман ЯРИЧ

Керівник проєкту _____

Костянтин КОЛЯДА

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (55 с., 13 рис., 4 додатки, 20 слайдів презентації).

Об'єкт розробки вебсервіс для автоматизованої генерації unit-тестів на основі великих мовних моделей.

Вебсервіс дозволяє: здійснювати автоматичний парсинг і витяг сигнатур функцій Python-коду; формувати структуровані запити до штучного інтелекту із переліком сценаріїв тестування; генерувати unit-тести за допомогою API; автоматично верифікувати коректність генерованого коду та аналізувати його покриття метриками.

В процесі розробки були використані мова програмування Python, вбудований модуль ast, інструменти pytest та pytest-cov.

В ході розробки:

- проведено огляд існуючих інструментів автогенерації тестів
- реалізовано модуль аналізу та парсингу Python-коду на основі вбудованого модуля ast;
- розроблено систему формування контекстних промптів та інтегровано API для генерації тестів;
- реалізовано модуль автоматичної верифікації тестів через pytest та аналіз покриття через pytest-cov;
- розроблено вебінтерфейс із можливістю завантаження файлів і перегляду результатів;
- проведено експериментальне дослідження ефективності на практичних прикладах.

Упровадження цієї системи дозволить розробникам зменшити навантаження на написання тестів, мінімізувати кількість рутинних операцій та підвищити підсумкову якість програмного продукту.

Ключові слова: ВЕЛИКІ МОВНІ МОДЕЛІ, UNIT-ТЕСТИ, АВТОГЕНЕРАЦІЯ ТЕСТІВ, AST-ПАРСИНГ, PYTEST.

ABSTRACT

Qualifying work includes an explanatory note (55 p., 13 fig., 4 applications, presentation of 20 slides).

The object of development is a web service for automated generation of unit tests based on large language models.

The web service allows users to automatically parse Python code and extract function signatures, generate structured prompts for artificial intelligence with a list of testing scenarios, generate unit tests through an API, automatically verify the correctness of the generated code, and analyze code coverage using appropriate metrics.

During the development process, the Python programming language, the built-in ast module, and the pytest and pytest-cov tools were used.

In the course of development:

- an overview of existing tools for automated test generation was carried out;
- a module for analyzing and parsing Python code based on the built-in ast module was implemented;
- a system for forming contextual prompts was developed, and an API for test generation was integrated;
- a module for automatic verification of tests using pytest and coverage analysis using pytest-cov was implemented;
- a web interface with the ability to upload files and view results was developed;
- an experimental study of the system's effectiveness on practical examples was conducted.

The implementation of this system will allow developers to reduce the workload associated with writing tests, minimize the number of routine operations, and improve the overall quality of the software product.

Keywords: LARGE LANGUAGE MODELS, UNIT TESTS, AUTOMATED TEST GENERATION, AST PARSING, PYTEST.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	A4	IАЛЦ.045420.002 ТЗ	Система автоматизованої генерації unit-тестів на основі велики мовних моделей. Технічне завдання.	4		
	A4	IАЛЦ.045420.003 ТП	Система автоматизованої генерації unit-тестів на основі велики мовних моделей. Відомість технічного проекту.	1		
	A4	IАЛЦ.045420.004 ПЗ	Система автоматизованої генерації unit-тестів на основі велики мовних моделей. Пояснювальна записка	53		
IАЛЦ.045420.001 ОА						
Змін.	Арк.	№ докум.	Підпис	Дата		
Розробив	Ярич Р.Р.				Літ.	Аркуші Аркушів
Перевірив	Коляда К.В.					1 2
Консульт.						
Н. контроль	Клятченко Я.М.				КПП ім. Ігоря Сікорського, ФПСІМ, KB-21	
Зав. каф.	Романкевич В.О.					

[illegible]

ЗМІСТ

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ	2
2. ПІДСТАВА ДЛЯ РОЗРОБКИ	2
3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ	2
4. ДЖЕРЕЛА РОБОТИ	2
5. ТЕХНІЧНІ ВИМОГИ.....	3
5.1 Вимоги до системи, що розробляється	3
5.2 Рекомендовані вимоги до апаратного забезпечення	3
5.3 Вимоги до мінімального програмного забезпечення	3
6. ЕТАПИ РОЗРОБКИ	4

					ІАЛЦ.045420.002 ТЗ					
Зм.	Арк.	№ докум.	Підп.	Дата	Система автоматизованої генерації unit-тестів на основі великих мовних моделей Технічне завдання.			Літ.	Аркуш	Аркушів
Розроб.	Ярич Р.Р.									
Перевір.	Коляда К.В.								1	4
								КПІ ім. Ігоря Сікорського, ФПСМ KB-21		
Н. контр.	Клятченко Я.М.									
Затв.	Романкевич В.О.									

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування проєкту - “Система автоматизованої генерації unit-тестів на основі великих мовних моделей”.

Область застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп’ютерних систем Національного технічного університету України “Київський Політехнічний Інститут імені Ігоря Сікорського”.

3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою дипломного проєкту є розробка системи автоматизованої генерації unit-тестів на основі великих мовних моделей для Python-коду з можливістю формування тестових сценаріїв, генерації pytest-тестів, їх запуску та оцінювання покриття коду.

4. ДЖЕРЕЛА РОБОТИ

Джерелами інформації для розроблення є технічна література, публікації у періодичних виданнях та Інтернет ресурси.

					ІАЛЦ.045420.002 ТЗ	Арк.
						2
Зм	Лист	№ докум.	Підп.	Дата		

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до системи, що розробляється

Система повинна забезпечувати такі функції:

- приймання Python-коду файлу;
 - статичний аналіз структури вхідного коду;
 - визначення основних елементів коду: функцій, класів, параметрів, імпортів, умов, циклів та винятків;
 - формування тестових сценаріїв для основних випадків роботи програми;
 - генерацію unit-тестів у форматі pytest;
 - перевірку згенерованих тестів;
 - автоматичний запуск тестів;
 - визначення покриття коду тестами;
 - формування звіту з результатами роботи системи.

5.2 Рекомендовані вимоги до апаратного забезпечення

- Процесор: Intel Core i5 або аналогічний.
- Оперативна пам'ять: 4 Гб.
- Простір на диску: 1 Гб.
- Доступ до мережі Інтернет: необхідний у разі використання зовнішнього API великої мовної моделі.

5.3 Вимоги до мінімального програмного забезпечення

- Операційна система Windows, Linux або macOS.
- Встановлений інтерпретатор Python версії 3.10 або вище.
- Встановлена бібліотека для взаємодії з API великої мовної моделі, наприклад google-gemini або anthropic.
- Наявність API-ключа Gemini API або Claude API.

					ІАЛЦ.045420.002 ТЗ	Арк.
						3
Зм	Лист	№ докум.	Підп.	Дата		

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проєкту	Терміни виконання етапів проєкту
1.	Вивчення літератури за тематикою проєкту	24.10.2025
2.	Розроблення та узгодження технічного завдання	11.11.2025
3.	Аналіз існуючих рішень	17.01.2026
4.	Підготовка матеріалів першого розділу дипломного проєкту	15.02.2026
5.	Розробка апаратного забезпечення	17.03.2026
6.	Підготовка матеріалів другого розділу дипломного проєкту	19.03.2026
7.	Розробка програмного забезпечення	26.03.2026
8.	Підготовка матеріалів третього розділу дипломного проєкту	07.04.2026
9.	Підготовка графічної частини дипломного проєкту	23.04.2026
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	29.04.2026
11.	Оформлення документації дипломного проєкту	15.05.2026
12.	Попередній огляд матеріалів диплому на кафедрі	27.05.2026

					ІАЛЦ.045420.002 ТЗ	Арк.
						4
Зм	Лист	№ докум.	Підп.	Дата		

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	A4	ІАЛЦ.045420.004 ПЗ	Система автоматизованої генерації unit-тестів на основі велики мовних моделей. Пояснювальна записка	53		
	A1	ІАЛЦ.045420.005 Д1	Система автоматизованої генерації unit-тестів. Схема структурна.	1		
	A1	ІАЛЦ.045420.006 Д2	Користувацький сценарій роботи з вебсервісом. Схема структурна.	1		
	A1	ІАЛЦ.045420.007 Д3	Перевірка згенерованих unit-тестів. Схема алгоритма.	1		
	A1	ІАЛЦ.045420.008 Д4	Обробка сценаріїв тестування. Схема алгоритма.	1		
				</		

Пояснювальна записка до дипломного проєкту

на тему: Система автоматизованої генерації unit-тестів на основі великих мовних моделей

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ.....	4
ВСТУП.....	5
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ.....	6
1.1 Актуальність автоматизованої генерації модульних тестів	6
1.2 Класичні підходи до автоматизованої генерації тестів	7
1.3 Великі мовні моделі у задачах генерації модульних тестів	9
1.4 Сучасні наукові підходи до покращення якості згенерованих тестів.....	10
1.5 Огляд існуючих інструментів і практичних рішень	13
1.6 Порівняльний аналіз підходів.....	15
1.7 Виявлення недоліків існуючих рішень	16
1.8 Постановка задачі та обґрунтування теми	17
2. АРХІТЕКТУРА ТА АЛГОРИТМІЧНІ ОСНОВИ СИСТЕМИ.....	20
2.1 Загальна архітектура системи	20
2.2 Вхідні та вихідні дані системи.....	22
2.3 Потік даних і логіка обробки	23
2.4 Статичний аналіз Python-коду.....	24
2.5 Двоетапне формування сценаріїв і тестового коду	26
2.6 Формування структурованих запитів до великої мовної моделі	27
2.7 Модуль взаємодії з LLM та режими генерації.....	29
2.8 Очищення, статична перевірка та підготовка pytest-тестів	30
2.9 Автоматичний запуск тестів, аналіз покриття та оцінювання якості	31
2.10 Формування звіту та пояснюваність результатів	32

					ІАЛЦ.045420.004 ПЗ			
Зм.	Арк.	№ докум.	Підп.	Дата				
Розроб.	Ярич Р.Р.				Система автоматизованої генерації unit-тестів на основі великих мовних моделей Пояснювальна записка.	Літ.	Аркуш	Аркушів
Перевір.	Коляда К.В.						1	53
Н. контр.	Клятчєнко Я.М.					КПІ ім. Ігоря Сікорського, ФПСІМ, КВ-21		
Затв.	Романкевич В.О.							

3. СИСТЕМА ГЕНЕРАЦІЇ UNIT-ТЕСТІВ	33
3.1 Призначення системи	33
3.2 Структура програмного продукту	33
3.3 Запуск вебсервісу та робочий сценарій користувача.....	34
3.3 Реалізація приймання вхідного коду у вебінтерфейсі.....	35
3.4 Реалізація серверного маршруту	38
3.5 Реалізація попередньої перевірки коду	39
3.6 Реалізація основного ланцюга.....	40
3.7 Формування запитів до LLM.....	42
3.8 Очищення відповіді LLM і валідація згенерованих тестів	43
3.9 Запуск pytest, збір покриття і формування звіту	44
3.10 Представлення результатів у вебінтерфейсі	45
4. ТЕСТУВАННЯ ТА ДОСЛІДЖЕННЯ	47
Тест 1 – неправильний синтаксис Python-коду	47
Тест 2 – невизначене ім'я у коді	47
Тест 3 – код без функцій і класів	48
Тест 4 – API-ключ не введено	49
Тест 5 – неправильний API-ключ	49
Тест 6 – відсутнє інтернет-з'єднання	50
ВИСНОВКИ.....	51
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	52

ДОДАТКИ

Додаток А. Копії графічних матеріалів

- ІАЛЦ.045420.005 Д1. Система автоматизованої генерації unit-тестів.
Схема структурна.
- ІАЛЦ.045420.006 Д2. Користувацький сценарій роботи з
вебсервісом. Схема структурна.

- ІАЛЦ.045420.007 Д3. Перевірка згенерованих unit-тестів. Схема алгоритма.
- ІАЛЦ.045420.008 Д4. Обробка сценаріїв тестування. Схема алгоритма.

Додаток Б. Фрагменти програмного коду

Додаток В. Презентація

					ІАЛЦ.045420.004 ПЗ	Арк.
						3
Зм	Лист	№ докум.	Підп.	Дата		

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

AST – Abstract Syntax Tree, абстрактне синтаксичне дерево; структуроване подання програмного коду у вигляді дерева, де вузли відповідають синтаксичним конструкціям мови програмування.

CLI – Command Line Interface, інтерфейс командного рядка; спосіб взаємодії з програмою через текстові команди.

GWT – Given–When–Then, «задано – коли – тоді»; структура опису тестового сценарію, де Given визначає початкові умови, When – дію, що виконується, а Then – очікуваний результат.

RAG – Retrieval-Augmented Generation, генерація з доповненням контексту; підхід, за якого мовна модель перед генерацією отримує додаткову релевантну інформацію із зовнішніх джерел.

RL – Reinforcement Learning, навчання з підкріпленням; підхід машинного навчання, за якого модель навчається на основі винагороди за виконані дії.

UTRL – Unit Test Reinforcement Learning; підхід до навчання генерації unit-тестів із використанням навчання з підкріпленням.

Unit-тест – модульний тест; тест, призначений для перевірки окремої функції, методу, класу або невеликого модуля програми.

Test oracle – тестовий оракул; критерій, за яким визначається правильність результату виконання тесту.

Coverage – покриття коду; метрика, що показує, яку частину програмного коду було виконано під час запуску тестів.

Mutation score – мутаційна оцінка; показник, що відображає здатність тестового набору виявляти штучно внесені помилки в код.

Prompt – структурований запит; текстова інструкція для мовної моделі, що містить опис задачі, контекст, обмеження та вимоги до формату відповіді.

Pipeline – ланцюг обробки; послідовність етапів, у якій результат одного модуля передається на наступний етап.

Repair-loop – цикл виправлення; повторне виправлення або уточнення згенерованого тесту після виявлення помилки під час запуску.

ВСТУП

Сучасне розроблення програмного забезпечення потребує швидкого внесення змін без втрати якості та стабільності системи. Одним із основних засобів перевірки коректності програм є unit-тестування, яке дає змогу ізольовано перевіряти окремі функції, методи або модулі. Наявність якісних unit-тестів допомагає своєчасно виявляти помилки, зменшувати ризик регресій і спрощує подальший супровід програмного продукту.

Разом з тим написання unit-тестів вручну є трудомістким процесом. Розробник має проаналізувати логіку коду, визначити тестові сценарії, підібрати вхідні дані та сформулювати коректні перевірки очікуваних результатів. Через це частина коду часто залишається недостатньо протестованою або покривається тестами формально.

Великі мовні моделі відкривають нові можливості для автоматизації цього процесу. Вони здатні аналізувати програмний код, враховувати назви функцій, параметри, коментарі та генерувати тестовий код у зрозумілому для розробника стилі. Водночас такі результати потребують перевірки, оскільки модель може створити некоректні твердження, неіснуючі імпорти або тести, що не виконуються.

Отже, завданням дипломного проєкту є створення системи автоматизованої генерації unit-тестів на основі великих мовних моделей для Python-коду. Система має аналізувати вхідний код, формувати тестові сценарії, генерувати pytest-тести, запускати їх, визначати помилки та оцінювати покриття коду. Це дозволить отримувати не лише згенерований тестовий код, а й перевірений результат, придатний для подальшого використання та доопрацювання розробником.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Перший розділ присвячено аналізу предметної області та існуючих підходів до автоматизованої генерації модульних тестів. Метою розділу є обґрунтування потреби у створенні власної системи, визначення її місця серед наявних інструментів і формулювання вимог, що впливають із сучасного стану досліджень та практики програмної інженерії.

1.1 Актуальність автоматизованої генерації модульних тестів

Модульне тестування є одним із найважливіших рівнів забезпечення якості програмного забезпечення, оскільки воно дозволяє перевіряти окремі програмні одиниці до того, як вони будуть інтегровані у більшу систему. У мовах високого рівня такою одиницею найчастіше є функція, метод класу або невеликий модуль. У випадку Python-проектів модульні тести особливо важливі, оскільки мова має динамічну типізацію, а частина помилок може проявлятися лише під час виконання програми. Наявність тестів допомагає швидше виявляти некоректні припущення щодо типів даних, граничних значень, порожніх структур, винятків та взаємодії між функціями.

Якісний модульний тест виконує не лише технічну, а й документаційну функцію. Він показує, як саме повинна використовуватися функція, які вхідні дані вважаються допустимими, який результат очікується у типовому випадку та як програма має реагувати на помилкові умови. Тому тестовий набір можна розглядати як виконувану специфікацію поведінки окремого модуля. Якщо тести написані зрозуміло, вони полегшують супровід коду, рефакторинг і навчання нових учасників проєкту.

Разом з тим написання модульних тестів потребує значних часових витрат. Розробник має визначити набір тестових сценаріїв, підготувати вхідні дані, написати виклик тестованої функції, сформулювати очікуваний результат і перевірити його за допомогою твердження перевірки. У науковій літературі

					ІАЛЦ.045420.004 ПЗ	Арк.
						6
Зм	Лист	№ докум.	Підп.	Дата		

структуру тесту часто описують через три складові: конфігурацію вхідних даних оператор виклику (invocation statement - рядок, у якому виконується тестована функція) та тестовий оракул [1]. Найскладнішою складовою є саме тестовий оракул, оскільки для його формування потрібно правильно зрозуміти очікувану поведінку програми.

Автоматизована генерація тестів покликана зменшити трудомісткість цієї роботи. Проте автоматичне створення тесту не повинно обмежуватися простим виконанням коду. Тест, який не містить осмисленої перевірки, може штучно підвищувати покриття, але не підтверджувати коректність поведінки. Саме тому сучасні системи генерації тестів мають оцінювати не лише кількість створених тестів або відсоток покриття, а й їхню виконуваність, наявність змістовних тверджень перевірки, читабельність і придатність для подальшого використання розробником.

Актуальність теми посилюється тим, що сучасні великі мовні моделі здатні створювати код у стилі, близькому до людського, проте не гарантують його правильності. У дослідженнях показано, що тести, згенеровані мовними моделями, можуть досягати порівнюваного покриття з традиційними інструментами, але при цьому мати проблеми з правильністю тверджень перевірки [2], [3]. Отже, практична система має поєднувати генерацію з автоматичним запуском, фільтрацією та аналізом результатів.

1.2 Класичні підходи до автоматизованої генерації тестів

До активного поширення великих мовних моделей автоматизована генерація тестів розвивалася переважно в межах трьох груп методів: випадкова генерація, пошукове тестування та підходи на основі символічного виконання. Ці методи мають значну історію розвитку й використовуються в низці відомих інструментів, зокрема Randoop, EvoSuite та Pynguin. Їхньою спільною рисою є орієнтація на дослідження простору можливих вхідних даних і шляхів виконання програми.

					ІАЛЦ.045420.004 ПЗ	Арк.
						7
Зм	Лист	№ докум.	Підп.	Дата		

Випадкова генерація тестів ґрунтується на автоматичному створенні великої кількості вхідних значень або послідовностей викликів. Удосконаленням такого підходу є генерація з урахуванням зворотного зв'язку, коли система зберігає комбінації, що призводять до нових станів або помилок. Перевагою випадкового підходу є простота та здатність знаходити неочікувані комбінації вхідних даних. Недоліком є те, що такі тести часто не мають чіткої семантичної мети й можуть бути малозрозумілими для розробника.

Пошукове тестування використовує цільові функції, наприклад максимізацію покриття рядків або гілок. EvoSuite є одним із відомих представників такого напрямку для Java-програм. Він застосовує еволюційні алгоритми для добору таких тестових наборів, які виконують якомога більшу частину коду. Для Python подібну роль виконує Pynguin – інструмент автоматизованої генерації модульних тестів, який орієнтується на створення тестів для фреймворку pytest [4], [5].

Таблиця 1.1 - Порівняння основних підходів до генерації модульних тестів

Підхід	Переваги	Обмеження
Ручне написання тестів	висока точність, повне розуміння логіки	значні часові витрати, залежність від досвіду розробника
Випадкова генерація	простота, можливість швидко створити багато вхідних даних	низька семантична якість, слабка читабельність
Пошукове тестування	високе структурне покриття, системний пошук шляхів	складні для читання тести, обмежене розуміння наміру коду
Символічне виконання	точне проходження окремих шляхів виконання	вибух кількості шляхів, складність зовнішніх залежностей
LLM-генерація	читабельність, використання семантики назв і коментарів	ризик галюцинацій, неправильних тверджень і нестабільного формату
Гібридний підхід	поєднання генерації, запуску, покриття та фільтрації	потребує складнішої архітектури та контролю виконання

Символічне виконання намагається побудувати умови, за яких виконуються різні шляхи програми. Якщо такі умови можна розв'язати, система отримує конкретні вхідні дані для проходження певної гілки коду. Цей підхід є потужним для програм із чіткою логікою, однак він може стикатися з проблемою вибуху кількості шляхів, складними залежностями, викликами зовнішніх ресурсів і високою обчислювальною вартістю.

Класичні методи мають важливу перевагу: вони системно досліджують структуру програми й можуть досягати високого покриття. Водночас їхнім обмеженням є слабше розуміння семантики коду. Автоматично створений тест може добре виконувати рядки програми, але залишатися важким для читання або мати слабкий тестовий оракул. Саме цей недолік став одним із чинників появи підходів, що використовують великі мовні моделі.

1.3 Великі мовні моделі у задачах генерації модульних тестів

Великі мовні моделі (Large Language Models, LLM) суттєво змінили підхід до автоматизації задач комп'ютерної інженерії. На відміну від класичних генераторів тестів, мовна модель може враховувати не лише формальну структуру коду, а й семантичні ознаки: назви функцій, параметрів, документаційні рядки, коментарі та типовий стиль використання бібліотек. Це дає змогу створювати тести, які є більш схожими на написані розробником.

У задачі генерації модульних тестів LLM може виконувати кілька функцій. По-перше, модель може визначати тестові сценарії, тобто описувати, які ситуації потрібно перевірити. По-друге, вона може генерувати сам тестовий код. По-третє, модель може пропонувати твердження перевірки, обирати очікувані винятки та підбирати вхідні дані. По-четверте, у складніших системах LLM може використовуватися для виправлення тестів після помилок запуску.

Проте використання мовної моделі саме по собі не усуває складність задачі тестування. Модель може створити синтаксично правильний код, який не виконується через неправильний імпорт, відсутній об'єкт, некоректну сигнатуру

					ІАЛЦ.045420.004 ПЗ	Арк.
						9
Зм	Лист	№ докум.	Підп.	Дата		

функції або хибне очікуване значення. У роботі Yuan та співавторів показано, що тести, згенеровані ChatGPT, мають перспективу з погляду читабельності й покриття, але суттєво страждають від помилок компіляції та неправильних тверджень перевірки [3]. Це означає, що LLM потрібно розглядати не як самодостатнє джерело істини, а як генератор кандидатних тестів.

Сучасний напрям досліджень поступово переходить від простого одноетапного запиту до керованих процесів генерації. Замість інструкції “напиши тести для цього коду” дедалі частіше використовуються структуровані запити поетапне формування сценаріїв, запуск тестів, аналіз помилок та повторне уточнення. Такий підхід дозволяє зменшити кількість невалідних тестів і зробити результат більш контрольованим.

Для цієї дипломної роботи важливою є ідея, що мовна модель має бути вбудована в інженерний ланцюг обробки. Такий ланцюг має включати аналіз вхідного Python-коду, формування контексту, генерацію сценаріїв, створення pytest-тестів, очищення відповіді, синтаксичну перевірку, запуск тестів і аналіз покриття. Саме поєднання генерації та валідації підвищує практичну цінність системи.

1.4 Сучасні наукові підходи до покращення якості згенерованих тестів

Аналіз сучасних досліджень показує, що головною проблемою LLM-генерації тестів є не лише створення коду, а забезпечення його практичної придатності. У систематичному огляді літератури з LLM-орієнтованої генерації модульних тестів виділяються кілька основних задач: генерація повного тесту, генерація вхідних даних, генерація тестового оракула та еволюція тестів після змін у програмі [1]. Така класифікація показує, що якісний тест складається з кількох взаємопов’язаних елементів, і помилка в будь-якому з них може зробити тест непотрібним або навіть шкідливим.

Одним із важливих напрямів є сценарно-орієнтована генерація. У підході Two-Step Zero-Shot Prompting спочатку визначаються тестові сценарії, а вже

					ІАЛЦ.045420.004 ПЗ	Арк.
						10
Зм	Лист	№ докум.	Підп.	Дата		

потім для них генерується тестовий код [2]. Такий підхід наближений до логіки роботи розробника: перед написанням тесту потрібно зрозуміти, що саме має бути перевірено. У контексті цієї роботи доцільно використовувати сценарії “задано – коли – тоді” (Given-When-Then).

Іншим важливим напрямом є ітеративне виправлення. ChatTester використовує ідею попереднього з’ясування наміру тестованого методу та подальшого виправлення тестів на основі помилок компіляції або виконання [3]. Для Python така ідея може бути адаптована через запуск pytest і передавання повідомлення про помилку назад до мовної моделі. Це дає змогу реалізувати цикл виправлення (repair-loop - повторна генерація або корекція тесту після виявлення помилки).

Промисловий досвід Meta у системі TestGen-LLM демонструє важливість фільтрів якості. У цьому підході мовна модель генерує кандидатні тести, але до рекомендацій потрапляють лише ті, що проходять перевірки: успішно збираються, стабільно виконуються та дають вимірюване покращення покриття [6]. Така логіка є надзвичайно важливою для практичної системи: результат мовної моделі не повинен прийматися без перевірки.

Окрему групу становлять підходи, у яких зворотний зв’язок від покриття використовується для покращення генерації. TestCTRL поєднує покрокове міркування (Chain-of-Thought - проміжний опис наміру та можливих вхідних даних) із навчанням із підкріпленням за сигналом покриття [6]. Повна реалізація такого підходу потребує значних ресурсів, однак сама ідея є корисною для бакалаврського проєкту: після запуску тестів система може використовувати непокриті рядки як підставу для додаткової генерації.

Окремої уваги в цьому напрямі заслуговує робота Dongjun Lee, Changho Hwang та Kimin Lee “Learning to Generate Unit Test via Adversarial Reinforcement Learning”, у якій запропоновано підхід UTRL (Unit Test Reinforcement Learning - навчання генерації модульних тестів за допомогою навчання з підкріпленням). На відміну від підходів, що лише застосовують готову мовну модель для

створення тестів, автори розглядають задачу саме як навчання моделі генерувати якісні тести.

Основна ідея полягає в тому, що тест має бути не просто синтаксично правильним і не лише підвищувати покриття, а й відрізняти коректну реалізацію програми від помилкових або неповних реалізацій. У підході UTRL використовується змагальна схема навчання. Одна модель виконує роль генератора модульних тестів, а інша - роль генератора програмного коду. Генератор тестів отримує винагороду тоді, коли створені ним тести проходять на еталонній реалізації, але виявляють помилки у варіантах коду, згенерованих іншою моделлю. Така винагорода називається дискримінаційною. Водночас генератор коду навчається створювати такі розв'язання, які здатні проходити все складніші тести [7].

Важливість цієї роботи для даного дипломного проєкту полягає не в безпосередньому відтворенні повного навчального процесу, оскільки adversarial reinforcement learning потребує значних обчислювальних ресурсів, спеціально підготовлених наборів даних і складної інфраструктури навчання моделей.

Цінною є сама постановка критеріїв якості тесту. Тест повинен бути валідним, тобто проходити на правильно реалізованій програмі, і водночас дискримінативним, тобто здатним виявляти неправильні реалізації або помилки логіки. У практичній системі для Python ці ідеї можуть бути спрощено адаптовані через запуск згенерованих pytest-тестів, перевірку тверджень перевірки, аналіз покриття та, у перспективі, застосування мутаційного тестування.

Ще одним перспективним напрямом є пошук релевантного контексту. RefTest розширює генерацію з доповненням контексту (Retrieval-Augmented Generation, RAG - підхід, коли модель отримує додаткову інформацію з зовнішніх джерел) для задачі тестування, використовуючи наявні тести пов'язаних методів [9]. У межах даної роботи повний репозиторний пошук може бути надмірним, але сама ідея використання релевантного контексту узгоджується з AST-аналізом Python-коду та формуванням стислого опису функцій, класів і залежностей.

					ІАЛЦ.045420.004 ПЗ	Арк.
						12
Зм	Лист	№ докум.	Підп.	Дата		

1.5 Огляд існуючих інструментів і практичних рішень

Для обґрунтування теми дипломного проєкту необхідно розглянути не лише наукові підходи, а й практичні інструменти. Вони демонструють реальні сценарії використання автоматичної генерації тестів та допомагають визначити вимоги до власної системи. Найбільш релевантними для цієї роботи є Pynguin, EvoSuite, AthenaTest, Qodo Cover, Diffblue Cover, Tusk, ChatTester, TestGen-LLM та підходи, що використовують структуровані запити до мовних моделей.

Pynguin є найближчим технічним аналогом для Python-екосистеми. Він автоматично генерує тести для Python-коду та використовує pytest як цільовий формат [10], [11]. Перевагою Pynguin є спеціалізація на Python і можливість застосування як базового інструменту для порівняння. Водночас Pynguin належить до класичних генераторів і не використовує семантичні можливості великої мовної моделі, тому його тести можуть бути менш природними для розробника.

EvoSuite є відомим інструментом для Java, що застосовує пошукове тестування та еволюційні алгоритми. Він часто використовується як базовий інструмент у дослідженнях автоматичної генерації тестів. Його сильна сторона – здатність досягати високого структурного покриття. Обмеження полягає у тому, що згенеровані тести можуть бути важкими для читання й не завжди добре відображати намір розробника.

AthenaTest представляє ранній напрям використання моделей глибокого навчання для генерації тестів. У порівнянні з класичними інструментами він демонструє більшу схожість із тестами, написаними людиною, однак також стикається з проблемами неправильних тверджень перевірки та неповного контексту. Це підтверджує загальний висновок: мовна модель може покращити читабельність тестів, але потребує інструментальної перевірки.

Qodo Cover, Diffblue Cover і Tusk можна розглядати як приклади промислової автоматизації тестування. Такі інструменти орієнтовані на практичне використання в середовищах розробки, інтеграцію з репозиторіями,

автоматизованими перевірками та процесами безперервної інтеграції (Continuous Integration, CI - автоматичний запуск перевірок після змін у коді). Для дипломної роботи вони важливі як підтвердження практичного попиту на системи, що не лише генерують код тестів, а й оцінюють його якість.

Проте значна частина промислових рішень є закритою або орієнтованою на конкретну екосистему. Це ускладнює дослідження внутрішньої логіки генерації, структури запитів, критеріїв фільтрації та помилок. Для бакалаврського проєкту доцільніше реалізувати власну прозору систему, у якій кожен етап – від аналізу Python-коду до запуску pytest - може бути описаний, перевірений і пояснений [12].

Таблиця 1.2 - Узагальнення існуючих інструментів і практичних рішень

Інструмент / підхід	Технологічний фокус	Ключова ідея
Pynguin	Python / pytest	автоматична генерація тестів із фокусом на покриття
EvoSuite	Java / JUnit	пошукове тестування та еволюційні алгоритми
AthenaTest	Java / JUnit	нейромережеве створення тестів, схожих на написані людиною
ChatTester	Java / LLM	генерація з ітеративним виправленням
TestGen-LLM	промисловий контекст	фільтрація кандидатних тестів за збіркою, стабільністю та покриттям
Qodo Cover / Tusk	CI / практичні сервіси	інтеграція генерації тестів у процес розроблення
UTRL	LLM / навчання з підкріпленням	змагальне навчання генератора тестів і генератора коду з використанням дискримінаційної винагороди

1.6 Порівняльний аналіз підходів

Порівняння існуючих підходів показує, що жоден із них не є універсальним. Ручне написання тестів забезпечує найвищий рівень розуміння логіки, але потребує значного часу. Класичні генератори можуть забезпечувати високе покриття, проте часто поступаються за читабельністю. Мовні моделі здатні створювати більш природний тестовий код, але мають ризик помилок і потребують перевірки. Тому найбільш доцільним є комбінований підхід, у якому LLM використовується для генерації кандидатних тестів, а стандартні інструменти Python - для їх виконання та оцінювання.

Основним критерієм для цієї роботи є підтримка Python і pytest. З цієї позиції Pynguin є найкращим традиційним базовим інструментом для порівняння. Проте Pynguin не вирішує проблему семантичної інтерпретації коду. Велика мовна модель, навпаки, може краще працювати з назвами функцій, документаційними рядками та логікою очікуваної поведінки, але її відповідь не можна приймати без перевірки.

Важливим критерієм є наявність автоматичного запуску тестів. Якщо система лише повертає текст згенерованого тесту, користувач не знає, чи він має коректний синтаксис, чи правильно імпортує тестований модуль, чи проходить на початковому коді. Саме тому архітектура власної системи має передбачати запуск pytest, збір стандартного виведення та помилок, класифікацію помилок і формування зрозумілого звіту.

Ще одним критерієм є аналіз покриття. Покриття рядків (line coverage - частка виконаних рядків коду) та покриття гілок (branch coverage - частка виконаних напрямів умовної логіки) не гарантують повної правильності тестів, але вони дають корисний кількісний сигнал. У поєднанні з перевіркою тверджень і результатів виконання покриття допомагає оцінити повноту тестового набору.

З огляду на ці критерії власна система має бути побудована як модульний і прозорий процес. Її призначення полягає не лише у створенні тестового файлу, а в організації повного циклу: приймання Python-коду, аналіз, формування

					ІАЛЦ.045420.004 ПЗ	Арк.
						15
Зм	Лист	№ докум.	Підп.	Дата		

сценаріїв, генерація pytest-тестів, перевірка, запуск, аналіз покриття та підготовка результату для користувача.

1.7 Виявлення недоліків існуючих рішень

Першим істотним недоліком існуючих підходів є надмірна орієнтація на покриття коду. Покриття є важливою метрикою, але воно не показує, чи справді тест перевіряє правильність результату. Наприклад, тест може виконати всі рядки функції, але містити лише загальну перевірку на відсутність помилки. Такий тест не є достатньо змістовним. Тому якість тестів потрібно оцінювати разом із наявністю та змістовністю тестового оракула.

Другим недоліком є некоректність тверджень перевірки. Мовна модель може помилитися у значенні, яке має повернути функція, або неправильно зрозуміти виняткову ситуацію. У результаті тест може падати не через дефект програми, а через помилкове очікування. Такі тести не можна автоматично приймати до фінального набору без запуску та аналізу результату.

Третім недоліком є галюцинації моделі (hallucination - генерація правдоподібної, але фактично неправильної інформації). У контексті програмування це може проявлятися як використання неіснуючих функцій, неправильних імен класів, зайвих бібліотек або неіснуючих параметрів. Для зменшення цього ризику система має надавати моделі структурований контекст і після генерації перевіряти тестовий код інструментально.

Четвертим недоліком є низька пояснюваність частини інструментів. Користувач часто бачить лише кінцевий тестовий файл, але не розуміє, які сценарії були враховані, які ділянки коду залишилися непокритими та чому певні тести були відхилені. Для навчального й дослідницького застосування важливо показувати проміжні етапи: виділені функції, сформовані сценарії, результат запуску й показники покриття.

П'ятим недоліком є складність роботи з багатофайловими проєктами, зовнішніми залежностями та станом об'єктів. Якщо функція залежить від

					ІАЛЦ.045420.004 ПЗ	Арк.
						16
Зм	Лист	№ докум.	Підп.	Дата		

файлової системи, мережі, бази даних або часу, автоматична генерація тестів стає складнішою. У таких випадках потрібні підміни залежностей (mock - об'єкт, який імітує поведінку реальної залежності), фікстури (fixture - підготовлений стан або дані для тесту) та ізоляція виконання.

Виявлені недоліки визначають ключові вимоги до системи: вона має не лише генерувати тести, а й аналізувати код, пояснювати сценарії, перевіряти тестовий оракул, запускати тести, вимірювати покриття, фільтрувати невалідні результати та показувати користувачу зрозумілий звіт.

1.8 Постановка задачі та обґрунтування теми

Проведений аналіз показав, що автоматизована генерація модульних тестів залишається актуальною задачею програмної інженерії. З одного боку, ручне створення тестів потребує значних часових витрат і залежить від досвіду розробника. З іншого боку, наявні автоматизовані інструменти не завжди забезпечують одночасно високе покриття, читабельність, коректність тверджень перевірки та прозорість процесу генерації.

Класичні генератори тестів ефективно досліджують структуру програми та можуть підвищувати покриття коду, однак їхні результати часто потребують додаткового аналізу з боку розробника. Великі мовні моделі, навпаки, здатні враховувати семантику назв функцій, параметрів, коментарів і документаційних рядків, завдяки чому можуть створювати більш зрозумілі тестові випадки. Водночас такі моделі не гарантують правильності результату: згенеровані тести можуть містити помилкові твердження перевірки, неіснуючі імпорти, некоректні виклики або слабкі перевірки очікуваної поведінки.

З огляду на це, у межах дипломного проєкту доцільно розробити систему, у якій велика мовна модель використовується не як самостійне джерело готового результату, а як один із компонентів керованого процесу генерації. Такий процес має поєднувати аналіз Python-коду, формування тестових сценаріїв, генерацію pytest-тестів, їх запуск, перевірку результатів і обчислення покриття. Саме

					ІАЛЦ.045420.004 ПЗ	Арк.
						17
Зм	Лист	№ докум.	Підп.	Дата		

поєднання генерації та інструментальної перевірки дозволяє зменшити ризики, характерні для LLM-орієнтованих підходів.

Таблиця 1.3 - Основні функціональні вимоги до системи

Вимога	Зміст
Приймання Python-коду	Користувач передає файл або фрагмент коду для подальшого аналізу.
Статичний аналіз	Система визначає функції, класи, параметри, імпорти, винятки та інші елементи структури коду.
Формування сценаріїв	Система визначає типові, граничні та помилкові випадки, які доцільно перевірити.
Генерація pytest-тестів	Мовна модель створює тестовий код з урахуванням сформованих сценаріїв і вимог до формату.
Перевірка згенерованого коду	Система перевіряє синтаксис тестів, наявність тестових функцій і змістовних тверджень перевірки.
Автоматичний запуск тестів	Згенеровані тести виконуються за допомогою pytest, після чого фіксується результат запуску.
Аналіз покриття	Система визначає, які рядки або гілки коду були виконані під час тестування.
Формування звіту	Користувач отримує згенеровані тести, результати запуску, повідомлення про помилки та показники якості.

Задача дипломного проєкту полягає у розробленні системи автоматизованої генерації модульних тестів для Python-коду на основі великих мовних моделей. Система має приймати вхідний Python-файл або фрагмент коду, виконувати його попередній аналіз, визначати потенційні об'єкти тестування, формувати тестові сценарії, генерувати тести для фреймворку pytest, запускати

їх у контрольованому середовищі, аналізувати помилки та формувати звіт про отримані результати.

Основна ідея запропонованого підходу полягає у побудові послідовного ланцюга обробки, де кожен етап має окреме призначення. Статичний аналіз коду забезпечує виділення функцій, класів, параметрів, імпортів і виняткових ситуацій. Етап формування сценаріїв визначає, які типові, граничні та помилкові випадки потрібно перевірити. Мовна модель генерує кандидатні тести, а модуль валідації перевіряє їхню синтаксичну коректність, наявність змістовних тверджень перевірки та результат виконання. Завершальним етапом є аналіз покриття й підготовка зрозумілого звіту для користувача.

					ІАЛЦ.045420.004 ПЗ	Арк.
						19
Зм	Лист	№ докум.	Підп.	Дата		

2. АРХІТЕКТУРА ТА АЛГОРИТМІЧНІ ОСНОВИ СИСТЕМИ

2.1 Загальна архітектура системи

Другий розділ присвячено опису архітектури та алгоритмічних основ системи автоматизованої генерації unit-тестів для Python-коду. На відміну від першого розділу, де було розглянуто предметну область, наявні інструменти та наукові підходи, у цьому розділі розглядається власний підхід до побудови системи. Основна увага приділяється тому, як окремі модулі взаємодіють між собою, які дані передаються між етапами та яким чином велика мовна модель вбудовується у контрольований процес генерації тестів.

Розроблювана система побудована як консольне backend-ядро, тобто програмна частина, яка виконує всю основну логіку без прив'язки до конкретного користувацького інтерфейсу. Такий вибір є доцільним на етапі бакалаврського проєктування, оскільки спочатку необхідно забезпечити працездатність основного алгоритму: приймання Python-файлу, аналіз його структури, формування запитів до мовної моделі, отримання тестів, їх запуск і побудову звіту. Після стабілізації цього ядра до нього можна підключити простий вебінтерфейс або іншу оболонку без перебудови всієї системи.

Архітектура системи має модульний характер. Кожен модуль відповідає за окремий етап обробки: завантаження коду, створення робочого каталогу, статичний аналіз, формування першого структурованого запиту, отримання тестових сценаріїв, формування другого запиту, генерацію pytest-тестів, очищення відповіді моделі, статичну перевірку тестового файлу, запуск тестів, аналіз покриття та формування підсумкового звіту. Завдяки такому розподілу відповідальності систему легше супроводжувати, тестувати і розширювати.

Важливо, що велика мовна модель у запропонованій архітектурі не є єдиним джерелом результату. Вона використовується як генератор сценаріїв і кандидатних тестів, однак остаточна придатність отриманого коду перевіряється засобами самої системи. Згенерований код очищується, перевіряється на наявність тестових функцій і змістовних тверджень перевірки, виконується через

					ІАЛЦ.045420.004 ПЗ	Арк.
						20
Зм	Лист	№ докум.	Підп.	Дата		

pytest, а покриття визначається за допомогою pytest-cov. Така організація відповідає висновкам першого розділу про необхідність поєднання LLM-генерації з інструментальною перевіркою результату.

На рисунку 2.1 наведено загальну структурну схему системи. Вона демонструє, що користувач взаємодіє лише з верхнім рівнем програми, передаючи Python-файл. Усі подальші дії виконуються автоматично: створюється робочий простір, аналізується код, формуються запити, викликається LLM або fallback-режим, виконуються перевірки та формується звіт.

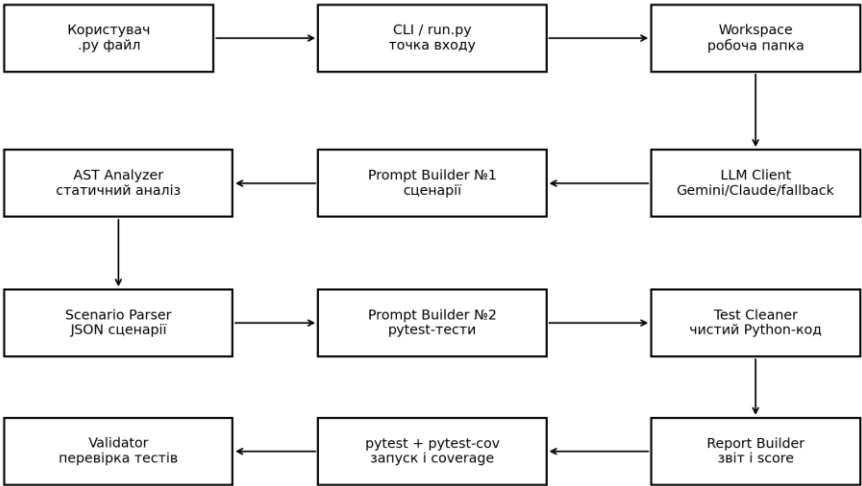


Рисунок 2.1 - Структурна схема системи автоматизованої генерації unit-тестів

Таблиця 2.1 - Основні модулі підсистеми та їх призначення

Підсистема	Призначення	Результат
Приймання вхідних даних	Отримання Python-коду від користувача у вигляді файлу або тексту.	Підготовлений текст коду та назва модуля.
Попередній аналіз	Перевірка синтаксису, структури та базових статичних помилок.	Структурований опис коду або повідомлення про помилку.
Генерація через LLM	Формування сценаріїв і pytest-тестів за допомогою Gemini API або Claude API.	Сценарії тестування та кандидатний тестовий код.

Продовження таблиці 2.1.

Виконання тестів	Запуск pytest і збір результатів виконання.	Статус проходження тестів і повідомлення про помилки.
Оцінювання результату	Визначення покриття, оцінка якості та підготовка звіту.	Звіт, файл тестів і метрики якості.

2.2 Вхідні та вихідні дані системи

Вхідними даними системи є Python-код, який користувач може передати двома способами: завантажити файл з розширенням .py або вставити фрагмент коду у текстове поле вебінтерфейсу. Обидва способи розглядаються системою як джерело одного Python-модуля, для якого необхідно сформувати unit-тести. На поточному етапі система орієнтована на аналіз одного модуля за один запуск, що спрощує контроль імпортів, виконання pytest і формування звіту.

Об'єктом тестування в межах системи є функції, методи або класи, що мають чітко визначену поведінку. Код, який не містить функцій або класів, не є придатним для генерації unit-тестів у межах реалізованого прототипу. У такому випадку система повинна повідомити користувача про відсутність об'єктів тестування, а не звертатися до мовної моделі.

Користувач також задає режим взаємодії з великою мовною моделлю. У фінальній користувацькій версії передбачено два варіанти: Gemini API та Claude API. Для роботи з обраним постачальником користувач вводить API-ключ і, за потреби, назву моделі. Ключ використовується лише для поточного запиту й не повинен записуватися до проміжних файлів або звіту.

Першим проміжним результатом є структурований опис вхідного коду. Він містить інформацію про знайдені функції, класи, методи, параметри, імпорти, винятки та керуючі конструкції. Цей опис використовується для побудови запиту до мовної моделі, а також відображається у звіті, щоб користувач бачив, які саме елементи програми були враховані системою.

Основними вихідними даними є згенерований файл unit-тестів у форматі pytest і текстовий звіт. Крім того, система формує дані про успішність запуску тестів, покриття коду, попередження та евристичну оцінку якості. Вихідний результат має бути корисним не лише для автоматичного виконання, а й для подальшого ручного перегляду розробником.

2.3 Потік даних і логіка обробки

Потік даних у системі організований як послідовний процес. Спочатку користувацький код потрапляє до модуля приймання вхідних даних. На цьому етапі система визначає, чи було надіслано файл, чи текстовий фрагмент, і формує внутрішнє подання модуля. Після цього код передається до етапу попередньої перевірки.

Попередня перевірка складається з двох частин. Перша частина - синтаксичний і структурний аналіз через AST. Вона дозволяє визначити, чи є код валідним Python-кодом, а також які програмні одиниці в ньому присутні. Друга частина - перевірка Ruff за правилом F821, що дозволяє знайти невизначені імена. Якщо на будь-якому з цих етапів виявлено критичну проблему, подальша генерація не виконується.

Якщо код пройшов попередні перевірки, система формує перший запит до LLM для визначення тестових сценаріїв. Після отримання сценаріїв формується другий запит, у якому модель повинна згенерувати pytest-тести. Отриманий код не вважається одразу готовим результатом: він проходить очищення, синтаксичну перевірку та аналіз наявності тестових функцій і тверджень перевірки.

Після валідації система запускає тести й збирає результат виконання. Якщо тести проходять успішно, це свідчить про те, що згенерований тестовий набір є технічно виконуваним. Якщо тести падають, система зберігає повідомлення про помилку, щоб користувач міг зрозуміти причину. Додатково система визначає покриття коду, що дає кількісну оцінку повноти тестового набору.

Завершальним етапом є формування звіту. У звіті поєднуються результати всіх попередніх етапів: аналіз коду, сценарії, тестовий файл, результат pytest, покриття та евристична оцінка якості. Саме така структура робить результат пояснюваним і придатним для подальшого використання або редагування.

```
workspaces/run_YYYYMMDD_HHMMSS/
├── __pycache__/
├── .pytest_cache/
├── .coverage
├── ast_analysis.json
├── code_quality_check.json
├── coverage_result.json
├── coverage.json
├── pytest_result.json
├── quality.json
├── raw_scenarios.txt
├── raw_tests.txt
├── report.txt
├── scenario_prompt.txt
├── scenarios.json
├── test_code_4.py
├── test_generated.py
├── test_prompt.txt
└── validation.json
```

Рисунок 2.2 - Приклад структури робочого каталогу одного запуску

2.4 Статичний аналіз Python-коду

Статичний аналіз є першим інструментальним етапом системи. Його призначення полягає в тому, щоб отримати структуровану інформацію про вхідний Python-код без його виконання. Це важливо з міркувань безпеки: система не повинна запускати користувацький код до того, як буде сформовано контрольоване середовище для тестування.

Для первинного аналізу використовується абстрактне синтаксичне дерево, або AST. AST подає програмний код у вигляді дерева синтаксичних конструкцій. За допомогою цього дерева можна визначити, які функції та класи оголошені в

модулі, які параметри мають функції, чи містять вони оператори `return`, `raise`, `if`, `for` або `while`. Такі ознаки не описують повну поведінку програми, але вони є корисними для планування тестів [13].

Наприклад, наявність оператора `raise` є сигналом для формування сценарію з очікуваним винятком. Наявність умовного оператора `if` означає, що функція може мати кілька гілок виконання. Наявність циклу може вказувати на потребу перевірити порожні та непорожні колекції. Таким чином, AST-аналіз створює структурний контекст для мовної моделі.

Окремо застосовується перевірка Ruff. На відміну від AST-парсингу, який визначає синтаксичну правильність, Ruff може виявляти частину помилок, що проявляються під час виконання. У межах системи використовується правило F821, яке відповідає за пошук невизначених імен. Наприклад, код `return a + bbb` має правильний Python-синтаксис, але ім'я `bbb` не визначене, тому тестування такого коду без попередження є некоректним [14].

Поєднання AST і Ruff дозволяє розмежувати два типи перевірок. AST відповідає за базову структуру та синтаксис, а Ruff - за виявлення помилок невизначених імен. Якщо код не проходить ці перевірки, звернення до великої мовної моделі не виконується. Це робить систему економнішою, стабільнішою та зрозумілішою для користувача.

Інструмент	Що перевіряє	Приклад проблеми
AST	Синтаксичну правильність і структуру коду.	Незакрита дужка, неправильний відступ, некоректне оголошення функції.
AST	Наявність функцій, класів, методів і параметрів.	Код містить лише інструкції верхнього рівня без об'єкта тестування.
Ruff F821	Використання невизначених імен.	<code>return a + bbb</code> , якщо <code>bbb</code> не оголошено.
Внутрішня логіка системи	Придатність коду для unit-тестування.	Відсутність функцій або класів у вхідному коді.

2.5 Двоетапне формування сценаріїв і тестового коду

Ключовою алгоритмічною ідеєю системи є двоетапна генерація. Замість того щоб одразу просити мовну модель створити тестовий код, система спочатку формує запит на визначення тестових сценаріїв. Лише після цього на основі сценаріїв створюється другий запит для генерації pytest-тестів. Такий підхід відповідає логіці Two-Step Zero-Shot Prompting, описаній у сучасних роботах з генерації unit-тестів.

Перший етап спрямований на планування тестування. Модель отримує вихідний код і структурний опис, після чого визначає, які випадки доцільно перевірити. До таких випадків належать типові сценарії, граничні значення, помилкові вхідні дані та крайові ситуації. Сценарії подаються у формі Given-When-Then: Given описує початкові умови, When - дію, Then - очікуваний результат.

Другий етап спрямований на синтез тестового коду. Модель отримує вже не лише вихідний код, а й сформований список сценаріїв. Це зменшує ризик того, що важливі випадки будуть пропущені, і робить процес генерації більш контрольованим. Система вимагає, щоб результатом другого етапу був виконуваний Python-код у форматі pytest.

Перевага такого підходу полягає в розділенні двох різних задач: визначення того, що потрібно тестувати, і написання коду, який це перевіряє. Якщо ці задачі змішати в одному запиті, модель може створити тести без достатнього обґрунтування сценаріїв або пропустити важливі граничні умови. Двоетапна схема робить процес більш пояснюваним.

У контексті дипломного проєкту сценарії мають також навчальну цінність. Вони дозволяють користувачу зрозуміти, чому система згенерувала саме такі тести. Якщо сценарій некоректний, це можна помітити до аналізу самого коду тестів. Отже, сценарії виступають проміжним шаром між аналізом програми та генерацією тестового набору.

Двоетапне формування результату: від сценаріїв до коду

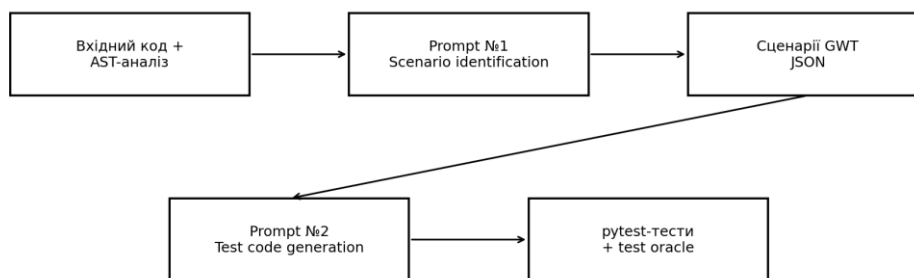


Рисунок 2.3 - Двоетапний підхід до генерації unit-тестів

Таблиця 2.4 - Ролі двох prompt-ів у системі

Етап	Зміст	Результат	Етап
Етап 1	LLM визначає тестові сценарії на основі коду та структурного опису.	Список сценаріїв Given-When-Then.	Етап 1
Етап 2	LLM генерує pytest-тести за сформованими сценаріями.	Кандидатний файл unit-тестів.	Етап 2

2.6 Формування структурованих запитів до великої мовної моделі

Формування структурованого запиту (prompt) є одним із центральних етапів системи. Prompt визначає, яку саме задачу повинна виконати мовна модель, у якому форматі повернути результат і яких обмежень дотримуватися. У цій роботі всі prompt-и сформульовано англійською мовою, оскільки технічні інструкції щодо програмного коду зазвичай стабільніше сприймаються моделями саме англійською. Водночас пояснення результатів у звіті може бути подано українською.

Перший prompt створюється модулем ScenarioPromptBuilder. Він містить роль моделі, заборону одразу писати тестовий код, опис двоетапного підходу, перелік типів сценаріїв і вимогу повернути лише JSON. До prompt додається

результат AST-аналізу у форматі JSON і повний текст Python-коду. Такий склад контексту дозволяє моделі бачити як фактичну реалізацію функцій, так і стислий структурний опис. Структурований запит до великої мовної моделі є засобом керування її поведінкою. У системі prompt не розглядається як довільний текст. Він містить роль моделі, формулювання задачі, обмеження, опис формату відповіді та контекст, сформований на попередніх етапах. Якість prompt-у безпосередньо впливає на якість отриманого тестового коду.

Для першого етапу prompt має змусити модель не писати код одразу, а сформулювати план тестування. Тому в ньому зазначається, що результатом мають бути сценарії, а не pytest-файл. Також задаються типи сценаріїв, які потрібно враховувати: нормальні, граничні, помилкові та крайові випадки. Це допомагає моделі працювати системніше [15].

Для другого етапу prompt містить вимоги до тестового коду. Зокрема, тести мають бути написані для pytest, мати функції з префіксом `test_`, містити `assert` або `pytest.raises`, не використовувати формальні перевірки `assert True`, не вигадувати неіснуючі функції та не змінювати тестований модуль для приховування помилок. Такі обмеження потрібні, тому що мовна модель може генерувати правдоподібний, але некоректний код.

Окрема увага приділяється тестовому оракулу. Тестовий оракул визначає, за якою ознакою система вважає результат правильним. Якщо тест лише викликає функцію без перевірки результату, він має низьку цінність навіть за високого покриття. Тому prompt формулюється так, щоб модель створювала конкретні очікувані значення або перевіряла очікувані винятки.

Усі prompt-и формуються англійською мовою, оскільки більшість мовних моделей стабільніше обробляє технічні інструкції з програмування англійською. Водночас інтерфейс, звіт і повідомлення про помилки можуть бути подані українською мовою, що робить систему зручнішою для кінцевого користувача в межах навчального проєкту.

2.7 Модуль взаємодії з LLM та режими генерації

Модуль взаємодії з LLM відповідає за надсилання сформованого запиту до великої мовної моделі та отримання відповіді. У користувацькій версії системи передбачено два режими генерації: Gemini API та Claude API. В обох випадках решта модулів системи не залежить від конкретного постачальника моделі, оскільки звернення до API приховано за єдиним інтерфейсом взаємодії.

Користувач обирає постачальника моделі у вебінтерфейсі та вводить API-ключ. API-ключ використовується лише для поточного запиту й не зберігається у звіті або проміжних файлах. Такий підхід зменшує ризик випадкового розкриття ключа під час перегляду результатів або передавання звіту іншій особі.

Наявність абстрактного модуля взаємодії з LLM дає змогу підтримувати кілька моделей без зміни загальної логіки системи. Для pipeline не має значення, чи відповідь отримано від Gemini, чи від Claude. На вхід модуль отримує prompt, а на виході повертає текстову відповідь. Подальша обробка цієї відповіді виконується однаково для обох режимів [16].

Поведінка великих мовних моделей є недетермінованою. Навіть за однакового prompt відповідь може відрізнятися за кількістю тестів, стилем назв, коментарями або способом побудови перевірок. Тому архітектура системи не покладається на припущення, що відповідь завжди буде ідеальною. Після отримання відповіді обов'язково виконуються очищення, валідація, запуск тестів і аналіз покриття.

У випадку помилки зовнішнього API система має повернути користувачу зрозуміле повідомлення. До таких помилок належать неправильний API-ключ, недоступна модель, відсутність інтернет-з'єднання або тимчасова недоступність сервісу. Обробка таких ситуацій є частиною загальної надійності системи.

2.8 Очищення, статична перевірка та підготовка pytest-тестів

Відповідь мовної моделі не можна безпосередньо вважати готовим тестовим файлом. Модель може повернути код у Markdown-блоці, додати пояснення, службові символи або текст, який не є Python-кодом. Тому після отримання відповіді система виконує етап очищення. Його мета - виділити саме виконуваний Python-код.

Після очищення тестовий код проходить статичну перевірку. Спочатку перевіряється синтаксис за допомогою `ast.parse`. Якщо тестовий код не є валідним Python-кодом, запуск `pytest` не виконується, а користувачу показується повідомлення про помилку. Це дозволяє відокремити помилки формату відповіді LLM від помилок виконання самих тестів.

Далі перевіряється наявність тестових функцій. Для `pytest` стандартним способом виявлення тестів є назви функцій, що починаються з префікса `test_`. Якщо таких функцій немає, файл не буде корисним як тестовий набір. Крім того, система перевіряє наявність `assert` або `pytest.raises`, оскільки без таких конструкцій тест може лише виконати код, але не підтвердити правильність результату.

Особливу увагу приділено слабким тестовим оракулам. Наприклад, конструкція `assert True` формально є перевіркою, але вона завжди проходить і не має змістовної цінності. Такі випадки повинні знижувати оцінку якості. Це важливо, оскільки високе покриття без змістовних тверджень не гарантує коректності програмної поведінки.

Етап очищення і перевірки виконує роль захисного шару між недетермінованою відповіддю LLM і детермінованим запуском `pytest`. Завдяки цьому система може не тільки створити тестовий код, а й оцінити його технічну придатність перед виконанням.

2.9 Автоматичний запуск тестів, аналіз покриття та оцінювання якості

Після проходження статичної перевірки згенеровані тести виконуються за допомогою `pytest`. Запуск тестів є принципово важливим етапом, оскільки лише він показує, чи можна реально виконати тестовий файл на вхідному коді. Якщо тестовий набір не запускається або містить неправильні очікувані значення, система повинна зафіксувати це у звіті [17].

Результат `pytest` містить статус успішності, код повернення, стандартне виведення та потік помилок. Якщо всі тести проходять, система отримує підтвердження технічної виконуваності тестового набору. Якщо хоча б один тест падає, користувач бачить відповідне повідомлення. Такий підхід дозволяє не приховувати проблеми LLM-генерації, а робити їх видимими.

Окремо система вимірює покриття коду. Покриття показує, яка частина рядків програми була виконана під час запуску тестів. Ця метрика не є абсолютним показником якості, але вона дає кількісний сигнал про повноту тестового набору. Наприклад, якщо тести проходять, але покривають лише невелику частину коду, користувач має розуміти, що набір потребує розширення.

Для узагальнення результатів використовується евристична оцінка якості. Вона враховує синтаксичну коректність тестів, наявність тестових функцій, наявність осмислених перевірок, успішність запуску `pytest` і рівень покриття. Такий бал не є математично строгою метрикою, однак він корисний для швидкого інженерного аналізу результату.

Важливо підкреслити, що система не ототожнює покриття з правильністю. Можлива ситуація, коли `coverage` є високим, але тест містить неправильне очікуване значення. Саме тому у звіті одночасно подаються і результат запуску `pytest`, і покриття, і попередження щодо тестового оракула.

2.10 Формування звіту та пояснюваність результатів

Підсумковий звіт є важливою складовою системи, оскільки користувач повинен отримати не лише файл тестів, а й пояснення того, як цей файл було отримано. Звіт поєднує результати всіх етапів: аналізу коду, формування сценаріїв, генерації тестів, статичної перевірки, запуску `pytest`, аналізу покриття та оцінки якості.

Перший блок звіту містить інформацію про модуль і результати структурного аналізу. Користувач бачить, які функції, класи та методи були виявлені, які з них мають `return`, `raise`, `if` або цикли. Це дозволяє переконатися, що система правильно розпізнала об'єкти тестування.

Другий блок звіту містить тестові сценарії. Це робить процес генерації прозорішим, оскільки користувач може оцінити не лише код тестів, а й логіку їх появи. Якщо певний сценарій виглядає зайвим або неточним, це можна помітити ще до аналізу тестового файлу.

Третій блок звіту містить результати перевірки та запуску. У ньому зазначається, чи є тестовий код синтаксично правильним, чи містить він `test_` функції, `assert` або `pytest.raises`, чи пройшов `pytest` і яке покриття було отримано. Такий формат дає змогу швидко знайти джерело проблеми: у вхідному коді, у відповіді LLM, у тестовому оракулі або у виконанні тестів.

Завершальний блок звіту містить повний текст згенерованих тестів. Це дає користувачу можливість одразу скопіювати або завантажити отриманий тестовий файл. У межах вебсервісу звіт також використовується як артефакт, який можна зберегти й додати до результатів експериментального дослідження.

Пояснюваність результатів є однією з переваг запропонованої системи порівняно з підходом, де користувач отримує лише готовий код. Завдяки звіту видно весь шлях від вхідного Python-коду до згенерованого тестового набору.

3. СИСТЕМА ГЕНЕРАЦІЇ UNIT-ТЕСТІВ

3.1 Призначення системи

Реалізований програмний продукт є локальним вебсервісом для автоматизованої генерації unit-тестів для Python-коду. Користувач працює з програмою через браузер: обирає постачальника великої мовної моделі, вводить API-ключ, завантажує .py файл або вставляє Python-код у текстове поле, після чого запускає генерацію. У відповідь система відображає сформовані тестові сценарії, згенерований pytest-код, результат виконання pytest, відсоток покриття коду та текстовий звіт.

Система не розгортається як публічний сайт для необмеженої кількості користувачів, а запускається локально на комп'ютері. Такий підхід обрано з міркувань безпеки, оскільки програма виконує згенеровані тести над користувацьким Python-кодом.

Система автоматизованої генерації unit-тестів

1. Вхідні дані

Велика мовна модель

Геміні API

Модель

Геміні 2.5 Flash

API key

Встав API key для gemini

Ключ використовується тільки для поточного запиту і не зберігається у звіт або файли workspace.

Завантажити .py файл

Виберіть файл

Файл не вибран

або встав Python-код нижче

Python-код

```
def add(a: int, b: int) -> int:
    return a + b

def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Згенерувати unit-тести

Рисунок 3.1 - Головна сторінка вебсервісу

3.2 Структура програмного продукту

Програмний продукт має модульну структуру. Верхній рівень проєкту містить файли запуску, папку з backend-модулями, папку вебінтерфейсу,

приклади Python-коду та каталоги для збереження результатів. Такий поділ спрощує супровід системи, оскільки інтерфейс користувача, логіка генерації, аналіз коду і запуск тестів розміщені в окремих частинах проєкту.

Основна логіка розміщена у пакеті `utg`. Файл `web_app.py` створює застосунок і описує HTTP-маршрути. Файл `pipeline.py` керує повним ланцюгом обробки від отримання коду до формування звіту. Підкаталог `core` містить модулі аналізу, формування `prompt`-ів, очищення, валідації, оцінювання якості та звітності. Підкаталог `llm` відповідає за звернення до Gemini API або Claude API. Підкаталог `runtime` містить модуль запуску `pytest` і збору `coverage`.

Файли `templates/index.html`, `static/app.js` і `static/style.css` реалізують клієнтську частину вебсервісу. HTML-файл описує структуру сторінки, JavaScript-файл відповідає за взаємодію з формою та відображення результатів, а CSS-файл визначає зовнішній вигляд інтерфейсу.



Рисунок 3.2 - Узагальнена структура програмного продукту

3.3 Запуск вебсервісу та робочий сценарій користувача

Вебсервіс запускається локально командою `python run_web.py`. У цьому файлі імпортується `uvicorn` і запускається застосунок, визначений у модулі `utg.web_app`. Після запуску користувач відкриває адресу `http://127.0.0.1:8000` у браузері. Це дозволяє працювати із системою через графічний інтерфейс, не вводячи команди в терміналі.

Основний сценарій роботи користувача складається з кількох кроків. Спочатку користувач обирає Gemini API або Claude API. Далі він вводить API-ключ і вибирає конкретну модель. Після цього можна або завантажити файл з розширенням .py, або вставити Python-код у текстове поле. Якщо користувач вибрав файл помилково, він може натиснути кнопку “×” біля назви файлу, після чого файл буде видалено з форми, а користувач зможе вибрати інший файл або скористатися текстовим полем.

Після натискання кнопки “Згенерувати unit-тести” JavaScript формує FormData і відправляє POST-запит на серверний маршрут /api/generate. Сервер перевіряє дані, запускає pipeline і повертає результат у форматі JSON. На сторінці відображаються коротке резюме, сценарії тестування, згенерований pytest-код, результат pytest, coverage, quality score, повний звіт і кнопки для скачування report.txt та test_generated.py.

1. Вхідні дані

Велика мовна модель

Модель

Gemini API

Gemini API

Claude API

Gemini 2.5 Flash

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

або встав Python-код нижче

Выберите файл

Файл не выбран

Python-код

```
def add(a: int, b: int) -> int:
    return a + b

def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Згенерувати unit-тести

Рисунок 3.3 - Вибір LLM, моделі та введення API-ключа

3.3Реалізація приймання вхідного коду у вебінтерфейсі

У вебінтерфейсі реалізовано два способи введення Python-коду. Перший спосіб полягає у завантаженні файлу з розширенням .py. Другий спосіб дозволяє вставити код вручну в текстове поле. При цьому файл має вищий пріоритет, ніж текстове поле. Така логіка зроблена для того, щоб користувач міг працювати як із

готовими файлами, так і з невеликими фрагментами коду, не змінюючи загальний сценарій використання системи.

У цій частині системи використовуються три основні файли клієнтської частини: `templates/index.html`, `static/app.js` та `static/style.css`. Файл `index.html` визначає структуру сторінки. У ньому розміщені поля вибору LLM-постачальника, вибору моделі, введення API-ключа, завантаження .ру файлу, текстове поле для коду та кнопка запуску генерації. Файл `style.css` відповідає за візуальне оформлення сторінки, а файл `app.js` реалізує поведінку інтерфейсу.

Основним у цьому підрозділі є саме файл `static/app.js`. Він отримує посилання на елементи сторінки через `document.getElementById`, реагує на зміну обраного постачальника LLM, оновлює список моделей, показує або приховує вибраний файл, формує `FormData` та надсилає дані на серверний маршрут `/api/generate`. Таким чином, JavaScript-модуль виступає проміжною ланкою між HTML-формою користувача та backend-сервісом.

Функція `updateModels()` відповідає за оновлення списку доступних моделей залежно від того, який постачальник LLM обраний користувачем. Якщо у полі `provider` вибрано `Gemini`, у списку моделей відображаються варіанти `Gemini`. Якщо вибрано `Claude`, список перебудовується відповідно до моделей `Claude`. Це дозволяє не показувати користувачу несумісні варіанти моделей.

Внутрішня логіка функції складається з кількох послідовних кроків. Спочатку зчитується поточне значення поля `provider`. Далі список `modelSelect` очищується, щоб прибрати моделі попередньо вибраного постачальника. Після цього функція проходить по масиву `models[provider]`, створює для кожної моделі HTML-елемент `option`, задає його `value` та текстову назву, а потім додає його до списку. Наприкінці функція активує поле введення API-ключа та змінює `placeholder`, щоб користувач бачив, для якого саме API потрібно ввести ключ.

Функція `updateSelectedFileBox` призначена для відображення інформації про вибраний файл. Стандартне поле завантаження файлу не завжди зручно показує користувачу поточний вибір, тому в інтерфейсі реалізовано окремий візуальний блок із назвою файлу та кнопкою видалення.

На початку функція перевіряє, чи існує у `fileInput` хоча б один вибраний файл. Якщо файл вибрано, його назва записується у `selectedFileName`, а контейнер `selectedFileBox` стає видимим. Якщо файл не вибрано або його було видалено, назва очищується, а блок приховується через CSS-клас `hidden`. Завдяки цьому користувач завжди бачить, який файл буде передано на `backend`.

Обробник натискання кнопки видалення файлу реалізує скасування вибору файлу без перезавантаження сторінки. Після натискання кнопки значення `fileInput` очищується, а потім повторно викликається `updateSelectedFileBox`. Це важливо, тому що користувач може випадково вибрати неправильний файл і після цього захотіти вставити код вручну або вибрати інший файл.

Функції `showError` і `hideError` відповідають за відображення помилок у вебінтерфейсі. Перша функція записує текст помилки у відповідний блок і робить його видимим. Друга функція, навпаки, приховує блок помилки та очищує його вміст. Ці функції використовуються під час надсилання форми, коли `backend` повертає повідомлення про помилку або коли перед новим запуском потрібно прибрати попередній результат.

Основним обробником клієнтської частини є обробник події `submit`. Він перехоплює стандартну поведінку HTML-форми, щоб сторінка не перезавантажувалася після натискання кнопки генерації. Далі приховується попередня помилка, очищується попередній результат, показується індикатор завантаження, а кнопка запуску тимчасово блокується. Це запобігає повторному надсиланню одного й того самого запиту.

Після цього створюється об'єкт `FormData`, який автоматично включає всі поля форми: `provider`, `model`, `api_key`, `source_code` та `file`. Далі виконується `fetch`-запит до `/api/generate` методом `POST`. Після отримання JSON-відповіді JavaScript перевіряє поле `ok`. Якщо `ok` має значення `false`, генерується помилка і користувач бачить її в `errorBox`. Якщо запит успішний, сторінка заповнюється результатами генерації: сценаріями, `pytest`-кодом, звітом, результатом `pytest` і посиланнями для скачування файлів.

3.4 Реалізація серверного маршруту

Серверний маршрут є проміжною ланкою між вебінтерфейсом і основним ланцюгом генерації unit-тестів. Його завдання полягає не в тому, щоб самостійно аналізувати Python-код або формувати тести, а в тому, щоб прийняти HTTP-запит, перевірити його параметри, підготувати дані до подальшої обробки та передати їх у `UnitTestPipeline`.

Основна логіка цього рівня реалізована у модулі `utg/web_app.py`. У ньому створюється застосунок `FastAPI`, підключаються статичні файли та HTML-шаблони, а також описуються маршрути для головної сторінки, запуску генерації і скачування результатів. Таким чином, `web_app.py` виконує роль зовнішнього серверного шару, який приймає запити від браузера та повертає відповідь у зручному для frontend-частини форматі.

Функція `index` обробляє звичайний GET-запит до головної сторінки. Вона не запускає генерацію тестів, а лише повертає HTML-шаблон інтерфейсу. Основна обробка зосереджена у функції `generate`, що відповідає маршруту `/api/generate`. Саме вона приймає значення `provider`, `model`, `api_key`, `source_code` та `file`, які надходять із форми вебінтерфейсу.

На початку `generate` виконується нормалізація параметрів. Значення `provider` переводиться у нижній регістр, із `model` та `api_key` прибираються зайві пробіли, а порожні значення перетворюються на відсутні. Після цього маршрут перевіряє, чи належить `provider` до підтримуваних варіантів Gemini API або Claude API. Якщо користувач не ввів API-ключ або вибрав непідтримуваний режим, система одразу повертає повідомлення про помилку і не створює `pipeline`.

Окремий блок логіки відповідає за визначення джерела Python-коду. Якщо користувач завантажив файл, перевіряється його розширення `.py` та можливість прочитати вміст у кодуванні UTF-8. Для такого файлу формується безпечна назва модуля за допомогою функції `safe_module_name`. Якщо файл не завантажено, система бере код із текстового поля. У випадку, коли обидва джерела порожні, користувач отримує повідомлення про необхідність передати Python-код.

					ІАЛЦ.045420.004 ПЗ	Арк.
						38
Зм	Лист	№ докум.	Підп.	Дата		

Після успішної перевірки вхідних даних `generate` створює об'єкт `UnitTestPipeline` і викликає метод `run_from_source`. Результат роботи `pipeline` містить дані аналізу, сценарії, згенеровані тести, результат запуску `pytest`, покриття, оцінку якості та шлях до робочого каталогу. На основі цих даних функція формує JSON-відповідь для сторінки.

Допоміжні функції `workspace_id_from_path` і `safe_workspace_path` використовуються для роботи з результатами запуску. Перша виділяє короткий ідентифікатор робочого каталогу, а друга перевіряє, що цей ідентифікатор справді відповідає допустимому каталогу системи. Це потрібно для безпечного скачування файлів `report.txt` і `test_generated.py` через маршрути `download_report` та `download_tests`.

3.5 Реалізація попередньої перевірки коду

Перед зверненням до великої мовної моделі система виконує локальну перевірку вхідного Python-коду. Цей етап потрібний для того, щоб не надсилати до LLM код, який не може бути виконаний або не містить об'єктів для unit-тестування. Завдяки цьому система зменшує кількість зайвих API-запитів і швидше повідомляє користувача про очевидні помилки.

Попередня перевірка реалізована двома модулями: `utg/core/ast_analyzer.py` та `utg/core/code_quality_checker.py`. Перший модуль відповідає за синтаксичний і структурний аналіз Python-коду, другий - за додаткову перевірку коректності імен за допомогою `Ruff`. Обидва модулі працюють без виконання користувацького коду, що є важливим для безпеки системи.

Головною функцією AST-рівня є `AstAnalyzer.analyze`. Вона приймає текст Python-коду та назву модуля, після чого намагається побудувати абстрактне синтаксичне дерево. Якщо `ast.parse` виявляє синтаксичну помилку, функція повертає результат із ознакою невдалої перевірки та текстом помилки. У такому випадку подальша генерація тестів не виконується.

Якщо синтаксис коректний, `analyze` переходить до збирання структурної інформації. Допоміжна функція `_extract_imports` формує список імпортів, а `_extract_function` створює опис кожної знайденої функції або методу. До такого опису входять назва, параметри, документаційний рядок, номер рядка, а також ознаки наявності `return`, `raise`, `if` і циклів. Ці ознаки надалі використовуються під час формування `prompt`-ів до мовної моделі.

Модуль `CodeQualityChecker` доповнює AST-аналіз. Його основна функція `check` записує код у тимчасовий файл і запускає `Ruff` з правилом `F821`. Це правило дозволяє знайти імена, які використовуються у коді, але не були оголошені. Наприклад, вираз `return a + bbb` має правильний синтаксис, однак ім'я `bbb` не визначене, тому виконання такого коду призведе до помилки.

Результати `Ruff` не передаються користувачу у службовому вигляді. Функція `_parse_diagnostics` розбирає JSON-виведення аналізатора, а `_translate_diagnostic` перетворює його на зрозуміле повідомлення українською мовою. Якщо у повідомленні є конкретне невизначене ім'я, функція `_extract_undefined_name` виділяє його з тексту діагностики.

Таким чином, попередня перевірка поєднує два рівні контролю: AST визначає, чи можна розібрати код і які об'єкти в ньому є, а `Ruff` виявляє помилки, що не порушують синтаксис, але можуть зупинити виконання. Якщо хоча б один із цих етапів завершується помилкою, `pipeline` не переходить до LLM-генерації.

3.6 Реалізація основного ланцюга

Основний ланцюг обробки об'єднує всі окремі модулі системи в один послідовний процес. У програмі цю роль виконує клас `UnitTestPipeline`, розміщений у модулі `utg/pipeline.py`. Він не замінює собою аналізатор, LLM-клієнт, `runner` або генератор звіту, а лише координує їхню роботу та передає результат одного етапу на наступний.

Головним методом цього класу є `run_from_source`. Він отримує текст Python-коду та назву модуля, після чого запускає повний цикл обробки:

					ІАЛЦ.045420.004 ПЗ	Арк.
						40
Зм	Лист	№ докум.	Підп.	Дата		

створення робочого каталогу, збереження вхідного файлу, аналіз структури, перевірку Ruff, формування сценаріїв, генерацію тестового коду, перевірку, запуск pytest, обчислення покриття і створення звіту. Саме цей метод викликається серверним маршрутом `/api/generate`.

На початку роботи pipeline створює окремий робочий каталог для конкретного запуску. У ньому зберігається вхідний Python-модуль, згенерований файл `test_generated.py`, звіт `report.txt` та службові дані покриття. Така організація дозволяє ізолювати різні запуски один від одного і спрощує подальше скачування результатів через вебінтерфейс.

Далі `UnitTestPipeline` виконує локальні перевірки. Спочатку викликається `AstAnalyzer.analyze`, після чого перевіряється, чи знайдено у коді хоча б одну функцію або клас. Потім запускається `CodeQualityChecker.check`. Якщо синтаксис неправильний, об'єкти тестування відсутні або Ruff знаходить невизначене ім'я, метод зупиняє роботу і повертає помилку на рівень серверного маршруту.

Після успішної попередньої перевірки pipeline переходить до LLM-частини. Він формує перший prompt для сценаріїв, передає його в `LlmClient`, отримує відповідь і перетворює її на список тестових сценаріїв. Потім на основі цих сценаріїв створюється другий prompt для генерації pytest-коду. Такий поділ дозволяє відокремити планування тестування від безпосереднього написання тестів.

Отриманий від моделі тестовий код проходить очищення і перевірку. За це відповідають `TestCleaner` і `TestValidator`. Якщо тестовий файл не має правильного синтаксису або не містить тестових функцій, pytest не запускається. Якщо перевірка успішна, pipeline передає файл до `TestRunner`, після чого збирає результат виконання та coverage.

Завершальним етапом є формування об'єкта результату. Він містить усі основні дані запуску: аналіз коду, сценарії, тестовий файл, результат pytest, показники покриття, якісну оцінку і шлях до workspace. Завдяки цьому

					ІАЛЦ.045420.004 ПЗ	Арк.
						41
Зм	Лист	№ докум.	Підп.	Дата		

web_app.py може перетворити результат pipeline на JSON-відповідь для сторінки без повторного виконання будь-яких обчислень.

3.7 Формування запитів до LLM

У системі велика мовна модель використовується не як самостійний генератор довільного тексту, а як частина керованого процесу. Для цього перед зверненням до LLM формується структурований prompt, у якому задаються роль моделі, вхідний Python-код, результати аналізу та вимоги до формату відповіді. Такий підхід зменшує ймовірність отримати випадковий або непридатний результат.

На цьому рівні використовуються три основні групи модулів: builder-и prompt-ів, LlmClient і парсер сценаріїв. ScenarioPromptBuilder формує перший запит, TestPromptBuilder - другий запит, а LlmClient приховує деталі взаємодії з конкретним API. Якщо відповідь моделі містить сценарії у структурованому форматі, ScenarioParser перетворює їх на внутрішні об'єкти системи.

Перший етап генерації починається з ScenarioPromptBuilder.build. Ця функція отримує вхідний код і результат AST-аналізу, після чого формує інструкцію для моделі: не писати тести одразу, а спочатку визначити сценарії перевірки. У запиті зазначається, що сценарії мають охоплювати типові, граничні та помилкові випадки й бути подані у логіці Given-When-Then.

Після отримання сценаріїв запускається TestPromptBuilder.build. На відміну від першого запиту, другий prompt уже містить список сценаріїв і просить модель створити pytest-код. У ньому задаються обмеження: відповідь має бути лише Python-кодом, тести повинні імпортувати модуль під наперед визначеним іменем, містити test_ функції, assert або pytest.raises і не використовувати формальні перевірки на зразок assert True.

Модуль LlmClient забезпечує єдиний інтерфейс звернення до Gemini API або Claude API. Його основним методом є ask, який отримує готовий prompt і повертає текстову відповідь моделі. Для решти системи не має значення, який

саме постачальник LLM використовується, оскільки pipeline працює з однаковим методом і однаковим типом результату.

Важливою особливістю цього етапу є те, що результат LLM не приймається без перевірки. Сценарії проходять розбір через ScenarioParser, а згенерований тестовий код на наступному етапі очищується і перевіряється окремими модулями. Тому LLM у системі виконує роль джерела кандидатних сценаріїв і тестів, а не остаточного критерію правильності.

3.8 Очищення відповіді LLM і валідація згенерованих тестів

Відповідь великої мовної моделі не завжди є готовим Python-файлом. Навіть якщо prompt вимагає повернути лише код, модель може додати Markdown-блок, пояснювальний текст або службовий маркер. Тому перед запуском pytest система виконує окремий етап очищення і перевірки згенерованих тестів.

За очищення відповіді відповідає модуль TestCleaner. Його основна функція clean приймає сирий текст, отриманий від LLM, і приводить його до вигляду, придатного для запису у test_generated.py. Насамперед вона прибирає зайві пробіли на початку й наприкінці, видаляє службові маркери на зразок <ctrl63> або <control...>, а також виділяє код із Markdown-блоків, якщо модель повернула відповідь у такому форматі.

Крім службових маркерів, TestCleaner прибирає типові текстові префікси, наприклад короткі фрази перед кодом. Це потрібно для того, щоб у тестовий файл не потрапив текст, який не є Python-інструкцією. Після очищення результат завершується символом нового рядка, що робить файл коректним для подальшого читання та запуску.

Другий модуль цього етапу - TestValidator. Його функція validate виконує базову перевірку згенерованого тестового коду. Спочатку за допомогою ast.parse перевіряється синтаксис. Якщо тестовий код неможливо розібрати як Python-програму, pytest не запускається, а система повертає повідомлення про помилку генерації.

					ІАЛЦ.045420.004 ПЗ	Арк.
						43
Зм	Лист	№ докум.	Підп.	Дата		

Якщо синтаксис коректний, TestValidator аналізує структуру тестового файлу. Він перевіряє наявність функцій, назви яких починаються з `test_`, а також шукає змістовні перевірки через `assert` або `pytest.raises`. Такі умови потрібні тому, що файл без тестових функцій або без тверджень перевірки не можна вважати повноцінним набором unit-тестів.

Окремо перевіряється наявність слабкого тестового оракула. Прикладом є `assert True`: такий вираз формально є перевіркою, але він завжди проходить і не підтверджує правильності поведінки тестованої функції. У цьому випадку система може сформулювати попередження, щоб користувач бачив потенційну проблему якості тестів.

Результатом роботи TestValidator є об'єкт перевірки, у якому зберігаються ознаки синтаксичної коректності, наявності тестових функцій, наявності `assert` або `pytest.raises`, а також список помилок і попереджень. Саме цей результат визначає, чи можна переходити до запуску `pytest`, чи потрібно зупинити обробку і показати користувачу повідомлення.

3.9 Запуск `pytest`, збір покриття і формування звіту

Після успішної перевірки згенерованого тестового коду система переходить до виконання тестів. Цей етап показує, чи є створений LLM тестовий файл не лише синтаксично коректним, а й реально виконуваним на переданому Python-модулі. Для цього використовується `pytest`, а для визначення покриття - `pytest-cov`.

Запуск тестів реалізовано у модулі `runtime/test_runner.py`. Основна функція цього рівня формує команду запуску `python -m pytest` для файлу `test_generated.py` і виконує її у межах робочого каталогу конкретного запуску. Робочий каталог містить тестований модуль і файл згенерованих тестів, тому `pytest` може імпортувати код без звернення до сторонніх директорій.

Під час запуску `TestRunner` передає в середовище виконання змінну `PYTHONPATH`, що вказує на поточний `workspace`. Це потрібно для того, щоб

					ІАЛЦ.045420.004 ПЗ	Арк.
						44
Зм	Лист	№ докум.	Підп.	Дата		

рядок імпорту в тестах коректно знаходив тестований модуль. Крім того, запуск виконується з обмеженням часу. Якщо тести зависають або виконуються надто довго, процес завершується, а користувач отримує повідомлення про проблему.

Результат `pytest` зберігається у структурованому вигляді: ознака успішності, код повернення, стандартне виведення і потік помилок. Якщо всі тести пройшли, користувач бачить успішний статус. Якщо хоча б один тест упав, система не приховує це, а передає текст помилки у звіт і вебінтерфейс. Це дозволяє відрізнити технічну помилку генерації від можливої помилки тестового оракула.

Для збору покриття використовується окремий запуск `pytest` із параметрами `pytest-cov`. У результаті створюється файл `coverage.json`. Модуль аналізу покриття читає цей файл і виділяє основні показники: загальний відсоток `coverage`, виконані рядки та рядки, які залишилися непокритими. Ці дані надалі використовуються як у звіті, так і в короткій панелі результатів на сторінці.

Після запуску тестів і збору `coverage` система формує підсумковий звіт. За це відповідає `ReportBuilder`. Він об'єднує результати попередніх етапів: структурний аналіз коду, список сценаріїв, результат перевірки тестів, `stdout` і `stderr` `pytest`, показники покриття та сам текст згенерованого тестового файлу. Завдяки цьому користувач отримує не лише файл `test_generated.py`, а й пояснення того, як система дійшла до цього результату.

Звіт зберігається у файлі `report.txt` у робочому каталозі. Разом із `test_generated.py` він є основним артефактом роботи системи. Саме ці файли можуть бути завантажені через вебінтерфейс і використані користувачем для подальшого аналізу або ручного доопрацювання тестового набору.

3.10 Представлення результатів у вебінтерфейсі

Після завершення `pipeline backend` повертає JSON-відповідь. Вона містить коротке резюме, результати AST-аналізу, сценарії, валідацію, результат `pytest`, `coverage`, `quality score`, текст звіту, текст згенерованих тестів і посилання для

					ІАЛЦ.045420.004 ПЗ	Арк.
						45
Зм	Лист	№ докум.	Підп.	Дата		

скачування файлів. JavaScript на клієнтській стороні заповнює відповідні блоки сторінки.

Користувач бачить коротку панель результатів: назву модуля, обраний provider, кількість знайдених функцій і класів, кількість сценаріїв, статус pytest, coverage та quality score. Нижче розміщено список сценаріїв, результат pytest/coverage, згенерований тестовий код і повний report.txt. Також передбачено дві кнопки: скачати звіт і скачати файл тестів.

1. Вхідні дані

Велика мовна модель

Gemini API

Модель

Gemini 2.5 Flash

API key

.....

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

Выберите файл

test_code_4.py

test_code_4.py

або встав Python-код нижче

Python-код

Згенерувати код-тести

2. Результат

Модуль

test_code_4

Провайдер

gemini

Функцій знайдено

1

Класів знайдено

2

Сценаріїв

18

Pytest успішний

Так

Coverage

74.24%

Quality score

95/100 (good)

Скачати звіт

Скачати файл тестів

Рисунок 3.3 - Результат генерації

Користувач бачить коротку панель результатів: назву модуля, обраний provider, кількість знайдених функцій і класів, кількість сценаріїв, статус pytest, coverage та quality score. Нижче розміщено список сценаріїв, результат pytest/coverage, згенерований тестовий код і повний report.txt. Також передбачено дві кнопки: скачати звіт і скачати файл тестів.

4. ТЕСТУВАННЯ ТА ДОСЛІДЖЕННЯ

Тест 1 – неправильний синтаксис Python-коду

У першому тесті перевірялася поведінка системи у випадку синтаксично некоректного Python-коду. У вхідному фрагменті навмисно залишено незавершений арифметичний вираз. Такий код не може бути перетворений у абстрактне синтаксичне дерево, тому система повинна зупинити обробку на етапі AST-перевірки.

1. Вхідні дані

Велика мовна модель: Gemini API

Модель: Gemini 2.5 Flash

API key:

Ключ використовується тільки для поточного запуску і не зберігається у звіт або файли журналу.

Завантажити .py файл

Вибрати файл | Файл не вибран

або встав Python-код нижче

Python-код

```
def add(a, b):
    return a +
```

Згенерувати звіт-тести

Некоректний синтаксис Python-коду: invalid syntax at line 2

Рисунок 4.1 – повідомлення про некоректний синтаксис Python-код

Тест 2 – невизначене ім'я у коді

Другий тестовий випадок перевіряє код, який має правильний синтаксис, але містить помилку, що проявиться під час виконання. Вираз `return a + bbb` є синтаксично коректним, однак ім'я `bbb` не визначено. Для виявлення таких помилок у системі використовується Ruff з правилом F821.

1. Вхідні дані

Велика мовна модель

Gemini API

Модель

Gemini 2.5 Flash

API key

.....

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

Выберите файл

Файл не выбран

або встав Python-код нижче

Python-код

def add(a, b):

return a + bbb

Згенерувати unit-тести

У коді знайдено помилки статичного аналізу: pasted_code.py:3:16: невизначене ім'я 'bbb'. Перевірте назву змінної або оголосіть її перед використанням.

Рисунок 4.2 – повідомлення Ruff про невизначене ім’я bbb

Тест 3 – код без функцій і класів

Третій тест перевіряє ситуацію, коли користувач вводить синтаксично правильний Python-код, але в ньому немає функцій або класів. Оскільки система призначена для генерації unit-тестів саме для функцій, методів і класів, такий код не має очевидного об’єкта модульного тестування.

1. Вхідні дані

Велика мовна модель

Gemini API

Модель

Gemini 2.5 Flash

API key

.....

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

Выберите файл

Файл не выбран

або встав Python-код нижче

Python-код

x = 10

print(x)

Згенерувати unit-тести

У вхідному Python-коді не знайдено функцій або класів для unit-тестування. Додайте хоча б одну функцію або клас.

Рисунок 4.3 – повідомлення про відсутність функцій або класів

Тест 4 – API-ключ не введено

У цьому тесті користувач обирає модель, але залишає поле API-ключа порожнім. Така ситуація повинна бути виявлена ще до формування запиту до LLM. Система має повернути повідомлення про те, що для обраної моделі необхідно ввести API-ключ.

Рисунок 4.4 – повідомлення про відсутній API-ключ

Тест 5 – неправильний API-ключ

У цьому тесті користувач вводить недійсний API-ключ і запускає генерацію. Очікувана поведінка полягає в тому, що система перехоплює помилку зовнішнього API та показує зрозуміле повідомлення українською мовою. Повідомлення має вказувати, що API-ключ недійсний або не має доступу до обраної моделі.

1. Вхідні дані

Велика мовна модель

Геміні API

Модель

Геміні 2.5 Flash

API key

.....

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

Выберите файл

Файл не выбран

або встав Python-код нижче

Python-код

```
def add(a: int, b: int) -> int:
    return a + b

def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Згенерувати unit-тести

API-ключ недійсний. Перевірте правильність введеного ключа та повторіть запит.

Рисунок 4.5 – повідомлення про недійсний API-ключ

Тест 6 – відсутнє інтернет-з’єднання

Для перевірки цього сценарію інтернет-з’єднання було навмисно вимкнено перед запуском генерації. У такому випадку система не може звернутися до сервера Gemini або Claude. Очікуваний результат – повідомлення про відсутність підключення до інтернету або тимчасову недоступність сервісу великої мовної моделі.

1. Вхідні дані

Велика мовна модель

Геміні API

Модель

Геміні 2.5 Flash

API key

.....

Ключ використовується тільки для поточного запиту і не записується у звіт або файли workspace.

Завантажити .py файл

Выберите файл

Файл не выбран

або встав Python-код нижче

Python-код

```
def add(a: int, b: int) -> int:
    return a + b

def divide(a: float, b: float) -> float:
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Згенерувати unit-тести

Немає підключення до інтернету або сервіс великої мовної моделі тимчасово недоступний. Перевірте з’єднання з мережею та повторіть запит.

Рисунок 4.6 – повідомлення про відсутність інтернет-з’єднання

ВИСНОВКИ

У результаті виконання дипломного проєкту було розроблено систему автоматизованої генерації unit-тестів на основі великих мовних моделей. Реалізований програмний продукт дає змогу користувачу передавати Python-код через вебінтерфейс, виконувати його попередній аналіз, формувати тестові сценарії, генерувати pytest-тести, запускати їх та отримувати звіт із результатами виконання і показниками покриття.

У ході роботи було проаналізовано існуючі підходи до автоматизованої генерації модульних тестів, зокрема класичні інструменти, пошукове тестування та сучасні LLM-орієнтовані методи. На основі цього було обґрунтовано доцільність побудови системи, у якій велика мовна модель використовується не як єдине джерело результату, а як частина контрольованого ланцюга обробки.

У програмній реалізації було використано модульний підхід. Окремі компоненти відповідають за приймання вхідного коду, AST-аналіз, перевірку коректності коду за допомогою Ruff, формування prompt-ів, взаємодію з Gemini API або Claude API, очищення відповіді мовної моделі, перевірку згенерованих тестів, запуск pytest, збір покриття через pytest-cov і формування підсумкового звіту.

Під час тестування було перевірено як основні функції вебсервісу, так і його поведінку у помилкових ситуаціях. Система коректно реагує на синтаксично неправильний Python-код, невизначені імена, відсутність функцій або класів, відсутній чи неправильний API-ключ, а також проблеми з інтернет-з'єднанням. Це підтверджує, що реалізований вебсервіс не лише генерує тести, а й забезпечує базовий контроль вхідних даних та зрозуміле повідомлення користувача про помилки.

Отже, поставлену мету дипломного проєкту було досягнуто. Розроблена система демонструє практичну можливість використання великих мовних моделей для автоматизованої генерації unit-тестів у поєднанні з інструментальною перевіркою, запуском тестів і аналізом покриття коду.

					ІАЛЦ.045420.004 ПЗ	Арк.
						51
Зм	Лист	№ докум.	Підп.	Дата		

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] Zhang J. et al. Enhancing Automated Unit Test Generation with Large Language Models: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology. – 2026.

[2] Bhatia S., Gandhi T., Kumar D., Jalote P. Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools LLM4Code. – 2024.

[3] Yuan Z. et al. Evaluating and Improving ChatGPT for Unit Test Generation Proceedings of the ACM on Software Engineering. – 2024.

[4] Lukasczyk S., Fraser G. Pynguin: Automated Unit Test Generation for Python Proceedings of the IEEE/ACM International Conference on Software Engineering Companion. – 2022.

[5] Pynguin. PYthoN General UnIt test geNerator [Електронний ресурс]. – Режим доступу: <https://pynguin.readthedocs.io/>. – Назва з екрана. – Дата звернення: 10.06.2026.

[6] Zhang J. et al. Automated Unit Test Generation via Chain-of-Thought Prompt and Reinforcement Learning from Coverage Feedback ACM Transactions on Software Engineering and Methodology. – 2026.

[7] Lee D., Hwang C., Lee K. Learning to Generate Unit Test via Adversarial Reinforcement Learning [Електронний ресурс]. – 2025. – Accepted to ICLR 2026.

[8] Zhang Z. et al. Reference-Based Retrieval-Augmented Unit Test Generation ACM Transactions on Software Engineering and Methodology. – 2026.

[9] Diop A., Toure F., Badri M. From Scenario to Code: Structured Prompting for LLM-Based Unit Test Generation WSSE. – 2025.

[10] Altmayer Pizzorno J., Berger E. D. CoverUp: Effective High Coverage Test Generation for Python Proceedings of the ACM International Conference on the Foundations of Software Engineering. – 2025.

[11] Alshahwan N. et al. Automated Unit Test Improvement using Large Language Models at Meta Proceedings of the ACM International Conference on the Foundations of Software Engineering Companion. – 2024.

[12] Pan R., Kim M., Krishna R., Pavuluri R., Sinha S. ASTER: Natural and Multi-Language Unit Test Generation with LLMs IEEE/ACM International Conference on Software Engineering – Software Engineering in Practice. – 2025.

[13] Python Software Foundation. ast — Abstract Syntax Trees. Python Documentation [Електронний ресурс]. – Режим доступу: <https://docs.python.org/3/library/ast.html>. – Назва з екрана. – Дата звернення: 10.06.2026.

[14] Ruff. Ruff Documentation [Електронний ресурс]. – Режим доступу: <https://docs.astral.sh/ruff/>. – Назва з екрана. – Дата звернення: 10.06.2026.

[15] pytest. pytest Documentation [Електронний ресурс]. – Режим доступу: <https://docs.pytest.org/>. – Назва з екрана. – Дата звернення: 10.06.2026.

[16] Google. Gemini API Documentation [Електронний ресурс]. – Режим доступу: <https://ai.google.dev/gemini-api/docs>. – Назва з екрана. – Дата звернення: 10.06.2026.

[17] pytest-cov. pytest-cov Documentation [Електронний ресурс]. – Режим доступу: <https://pytest-cov.readthedocs.io/>. – Назва з екрана. – Дата звернення: 10.06.2026.