

**МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ**

**ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І
СИСТЕМ**

КВАЛІФІКАЦІЙНА РОБОТА

**на тему: «РОЗРОБКА ІНСТРУМЕНТУ ДЛЯ ОЦІНКИ ЯКОСТІ
ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»**

Здобувача вищої освіти першого (бакалаврського) рівня,
IV курсу, групи КБ-22-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології (за галузями)»



(підпис)

Настаченка Артема
Сергійовича
(прізвище, ім'я, по батькові)

Науковий керівник



(підпис)

Філоненко Михайло
Миколайович, канд. фіз.-мат.
наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь,
вчене звання)

Керівник освітньо-
професійної програми



(підпис)

Ніжегородцев Владислав
Олександрович, канд. пед.
наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь,
вчене звання)

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних
технологій і систем протокол від «12» травня 2026 №20

Завідувач кафедри



(підпис)

Філоненко Михайло
Миколайович, канд. фіз.-мат.
наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь,
вчене звання)

Ірпінь – 2026

Реєстрація № 68

29.05.2026

(дата)



(підпис відповідальної особи)

Результати перевірки

до захисту

(до захисту, на доопрацювання)

12.05.2026

(дата)



(підпис)

Результат захисту

4 (добре)

(оцінка)

09.06.2026

(дата)



(підпис)

Голова комісії:



(підпис)

О.С. Дюбанов

Члени комісії:



(підпис)

М.М. Філоненко



(підпис)

В.О. Ніжегородцев



(підпис)

А.А. Омельчук

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії, секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст випускної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	22		09.06.2026 Філоненко М.М. Дюбанов О.С. Філоненко М.М. Ніжегородцев В.О. Омельчук А.А. Суховерхівка Л.П.
2. Дотримання вимог щодо оформлення	від 0 до 10	8		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	4		
4. Рецензент	від 0 до 5	4		
РАЗОМ БАЛІВ	від 0 до 50	38		

Оцінювання кваліфікаційної роботи науковим керівником – 36 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підписи)
74	C	4 (добре)	09.06.2026	Дюбанов О.С. Філоненко М.М. Ніжегородцев В.О. Омельчук А.А. Суховерхівка Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	
50-59	E	3 (задовільно)
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань: 12 «Інформаційні технології»
Спеціальність: 122 «Комп'ютерні науки»
Освітня програма: «Інформаційні управляючі системи і технології (за галузями)»

Затверджую

Завідувач кафедри  Філоненко М.М.

«01» грудня 2025

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Настаченку Артему Сергійовичу

1. Тема кваліфікаційної роботи Розробка інструменту для оцінки якості програмного забезпечення

Затверджена наказом ректора від «28» листопада 2025 № 1526

2. Строк здачі закінченої роботи «08» травня 2026

3. Мета роботи розробка інструменту для оцінки якості програмного забезпечення, який забезпечує аналіз програмного коду за допомогою зовнішнього AI API, відображення метрик якості, надання рекомендацій щодо виправлення помилок і покращення архітектури, а також збереження історії виконаних перевірок.

Об'єкт дослідження процес оцінювання якості програмного забезпечення на основі аналізу програмного коду.

Предмет дослідження програмні методи автоматизованого аналізу коду з використанням великих мовних моделей і локального збереження результатів перевірки.

Орієнтовна база дослідження Репозиторії на GitHub або GitLab із різним рівнем підготовки коду (наприклад, стабільні бібліотеки та нові аматорські проекти

4. Вихідні дані до роботи Локальний або віддалений каталог, що містить вихідний код програмного продукту на мові та конфігураційні файли залежностей. Конфігураційний файл у форматі JSON/YAML, який містить граничні значення метрик.

5. Перелік графічного матеріалу Таблиці, рисунки графіки

6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)

ВСТУП

1 Аналіз предметної області оцінювання якості програмного забезпечення

2 Проєктування інструменту для оцінки якості програмного забезпечення

3 Реалізація програмного інструменту

4 Тестування та оцінювання ефективності розробленого інструменту

5 Аналіз ризиків, безпеки даних та можливостей впровадження

Висновки

Перелік джерел посилання

Додатки

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 30.10.2025	До 30.10.2025
2	I розділ (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	05.02.2026	05.02.2026
3	II розділ (аналіз експериментальної частини про методики досліджень)	28.02.2026	28.02.2026
4	III розділ (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	15.04.2026	15.04.2026
5	IV розділ (проведені теоретичні і (або) експериментальні)	22.04.2026	22.04.2026
6	V розділ (проведені теоретичні і (або) експериментальні)	29.04.2026	29.04.2026
7	Висновки	01.05.2026	01.05.2026
8	Доопрацювання роботи з урахуванням зауважень наукового керівника	15.05.2026	15.05.2026
9	Отримання відгуку керівника	29.05.2026	29.05.2026
10	Отримання зовнішньої рецензії	01.06.2026	01.06.2026

8. Дата видачі завдання «01» грудня 2025

9. Керівник кваліфікаційної роботи


(підпис)

Філоненко М.М.
(прізвище, ініціали)

10. Керівник освітньо-професійної програми


(підпис)

Ніжегородцев В.
(прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти


(підпис)

Настаченко А.С.

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи присвячена розробці інструменту для оцінки якості програмного забезпечення. У роботі розглянуто поняття якості програмного забезпечення, роль аналізу програмного коду у процесі розробки, можливості автоматизації code review та використання великих мовних моделей для формування рекомендацій щодо виправлення помилок, покращення архітектури і підвищення читабельності коду.

Об'єктом дослідження є процес оцінювання якості програмного забезпечення на основі аналізу програмного коду. Предметом дослідження є методи та програмні засоби автоматизованого аналізу коду із застосуванням зовнішнього AI API. Метою роботи є створення настільного програмного інструменту, який дозволяє користувачеві вставити фрагмент коду, отримати виправлений варіант, переглянути метрики якості, отримати архітектурні рекомендації та поставити уточнювальні запитання в консультаційному чаті.

У процесі розробки використано мову програмування Python, фреймворк Flet для побудови графічного інтерфейсу, бібліотеку httpx для асинхронної взаємодії із зовнішнім API, python-dotenv для роботи з конфігурацією, SQLite для локального збереження історії перевірок. Як приклад зовнішнього AI-сервісу використано Mistral AI API, однак архітектура системи допускає підключення іншого сумісного API за наявності відповідного ключа доступу.

Ключові слова: якість програмного забезпечення, аналіз коду, code review, штучний інтелект, велика мовна модель, Python, Flet, SQLite, Mistral AI.

Рік виконання кваліфікаційної роботи: 2026

Рік захисту роботи: 2026

ABSTRACT

The bachelor's thesis is devoted to the development of a software quality assessment tool. The work considers the concept of software quality, the role of source code analysis in the development process, the automation of code review, and the use of large language models for generating recommendations on bug fixing, architecture improvement, and code readability enhancement.

The object of the study is the process of software quality assessment based on source code analysis. The subject of the study is methods and software tools for automated code analysis using an external AI API. The goal of the work is to develop a desktop tool that allows a user to insert a code fragment, receive a corrected version, review quality metrics, obtain architecture recommendations, and ask clarifying questions in a contextual consultation chat.

The developed tool uses Python, Flet for the graphical user interface, httpx for asynchronous API communication, python-dotenv for configuration management, and SQLite for local history storage. Mistral AI API is used as an example of an external AI service, while the system architecture allows using another compatible AI API key.

Keywords: software quality, code analysis, code review, artificial intelligence, large language model, Python, Flet, SQLite, Mistral AI.

Year of completion of the qualification paper: 2026

Year of defense of the work: 2026

ЗМІСТ

ВСТУП.....	5
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	7
1.1 Поняття якості програмного забезпечення.....	7
1.2 Роль аналізу програмного коду у забезпеченні якості ПЗ.....	8
1.3 Проблеми ручного рецензування програмного коду.....	8
1.4 Метрики оцінювання якості програмного коду.....	9
1.5 Автоматизовані інструменти аналізу програмного забезпечення.....	9
1.6 Використання штучного інтелекту в задачах аналізу коду.....	10
1.7 Постановка задачі розробки інструменту оцінки якості ПЗ.....	10
1.8 Розширений аналіз інструментів-конкурентів.....	11
1.9 Порівняльна характеристика розробленого інструменту.....	15
2 ПРОЄКТУВАННЯ ІНСТРУМЕНТУ ДЛЯ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	17
2.1 Формування вимог до програмного інструменту.....	17
2.2 Вибір архітектури програмного засобу.....	17
2.3 Дерево технологій розробки.....	18
2.4 Обґрунтування використання великих мовних моделей.....	19
2.5 Проєктування структури бази даних.....	20
2.6 Проєктування сценаріїв взаємодії користувача із системою.....	21
3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ІНСТРУМЕНТУ.....	23
3.1 Загальна структура програмного проєкту.....	23
3.2 Реалізація графічного інтерфейсу користувача.....	23
3.3 Реалізація модуля взаємодії з AI API.....	24
3.4 Реалізація режимів аналізу програмного коду.....	24
3.5 Реалізація локального збереження історії та експорту звіту.....	25
3.6 Обробка помилок і тестовий режим роботи.....	25
3.7 Особливості захисту даних користувача.....	25

4	ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ІНСТРУМЕНТУ.....	27
4.1	Методика тестування програмного засобу.....	27
4.2	Перевірка функціональних можливостей.....	27
4.3	Оцінювання швидкодії та зручності використання.....	27
4.4	Порівняння з існуючими рішеннями.....	28
4.5	Переваги, обмеження та напрями подальшого розвитку.....	28
4.6	Методика інтерпретації метрик якості.....	29
4.7	Сценарії практичного використання інструменту.....	30
4.8	Методика оцінювання достовірності AI-відповідей.....	32
4.9	Критерії приймального тестування програмного засобу.....	34
4.10	Оцінювання практичної ефективності використання інструменту.....	35
5	АНАЛІЗ РИЗИКІВ, БЕЗПЕКИ ДАНИХ ТА МОЖЛИВОСТЕЙ ВПРОВАДЖЕННЯ.....	39
5.1	Аналіз ризиків використання AI API.....	39
5.2	Захист API-ключів та конфігураційних даних.....	40
5.3	Конфіденційність програмного коду користувача.....	41
5.4	Практичне впровадження інструменту в навчальний процес.....	41
5.5	Перспективи розвитку розробленого програмного засобу.....	42
	ВИСНОВКИ.....	45
	СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	46
	ДОДАТКИ.....	48

ВСТУП

Якість програмного забезпечення є однією з ключових характеристик сучасних інформаційних систем. Програмні продукти використовуються у фінансовій сфері, освіті, медицині, промисловості, державному управлінні та повсякденному житті користувачів. Помилки у програмному коді можуть призводити не лише до некоректної роботи окремої функції, а й до втрати даних, зниження продуктивності, порушення безпеки, погіршення користувацького досвіду та фінансових збитків.

Традиційно якість програмного забезпечення забезпечується за допомогою тестування, ручного перегляду коду, статичного аналізу, дотримання стандартів програмування та використання інструментів безперервної інтеграції. Проте ручний аналіз коду потребує значних часових витрат і залежить від досвіду конкретного фахівця. У невеликих командах або навчальних проєктах якісний code review часто виконується нерегулярно або взагалі не виконується через нестачу часу.

Окремого значення набуває використання штучного інтелекту у програмній інженерії. Великі мовні моделі здатні аналізувати фрагменти коду, пояснювати помилки, пропонувати виправлення, формувати рекомендації щодо стилю, архітектури та безпеки. Це не замінює професійного розробника або тестувальника, однак може стати корисним інструментом підтримки, особливо на етапах первинного аналізу, навчання та швидкого виявлення очевидних проблем.

Актуальність роботи полягає у потребі створення зручного інструменту, який поєднує можливості аналізу програмного коду, оцінювання якості, формування рекомендацій і збереження результатів перевірок у локальному середовищі. Такий інструмент може використовуватися студентами, початківцями-розробниками та невеликими командами як допоміжний засіб для підвищення якості програмного забезпечення.

Метою роботи є розробка інструменту для оцінки якості програмного забезпечення, який забезпечує аналіз програмного коду за допомогою зовнішнього AI API, відображення метрик якості, надання рекомендацій щодо виправлення помилок і покращення архітектури, а також збереження історії виконаних перевірок.

Для досягнення поставленої мети необхідно виконати такі завдання:

проаналізувати предметну область оцінювання якості програмного забезпечення; дослідити підходи до автоматизованого аналізу коду; сформулювати вимоги до програмного інструменту; спроектувати архітектуру системи; реалізувати настільний застосунок; забезпечити взаємодію із зовнішнім AI API; реалізувати локальне збереження історії; провести тестування основних функцій.

Об'єктом дослідження є процес оцінювання якості програмного забезпечення на основі аналізу програмного коду.

Предметом дослідження є програмні методи автоматизованого аналізу коду з використанням великих мовних моделей і локального збереження результатів перевірки.

Практична цінність роботи полягає у створенні працездатного програмного інструменту, який демонструє можливість інтеграції AI-сервісу у настільний застосунок для підтримки процесу аналізу якості програмного забезпечення. Розроблене рішення може бути розширене шляхом підключення інших AI API, додавання аналізу репозиторіїв, інтеграції з GitHub або GitLab та впровадження розширених метрик якості.

Розділ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Поняття якості програмного забезпечення

Якість програмного забезпечення можна розглядати як сукупність властивостей програмного продукту, що визначають його здатність задовольняти встановлені та очікувані потреби користувача. У програмній інженерії це поняття не обмежується відсутністю помилок. Якісний програмний продукт повинен бути функціонально повним, надійним, зручним, безпечним, продуктивним і придатним до подальшого супроводу.

Відповідно до сучасних підходів, якість ПЗ має багатовимірний характер. Один і той самий застосунок може коректно виконувати функції, але бути складним для підтримки через заплутану архітектуру, дублювання коду або відсутність зрозумілих назв. Тому оцінювання якості повинно враховувати як зовнішню поведінку продукту, так і внутрішню структуру програмного коду.

Для розробника важливими є такі характеристики як читабельність, модульність, простота внесення змін, передбачуваність поведінки, мінімізація побічних ефектів і наявність обробки помилок. Для користувача якість проявляється через стабільність, швидкість роботи, зручність інтерфейсу та відсутність критичних збоїв.

У межах цієї роботи якість програмного забезпечення розглядається через призму аналізу програмного коду. Такий підхід дає змогу оцінити не лише кінцевий результат виконання програми, а й потенційні проблеми, які можуть виникнути під час розвитку проєкту.

1.2 Роль аналізу програмного коду у забезпеченні якості ПЗ

Аналіз програмного коду є важливим елементом життєвого циклу розробки програмного забезпечення. Його завданням є виявлення помилок, небезпечних конструкцій, порушень стилю, надмірної складності та архітектурних недоліків до того, як вони стануть причиною збоїв у роботі системи.

Code review дозволяє виявити проблеми, які не завжди фіксуються

автоматичними тестами. Наприклад, тести можуть підтвердити правильність результату для певного набору вхідних даних, але не вказати на те, що функція є надто великою, має дублювання логіки або порушує принцип єдиної відповідальності.

Аналіз коду також має навчальний ефект. Коли розробник отримує пояснення помилки та бачить виправлений варіант, він краще розуміє причину проблеми. Саме тому інструменти автоматизованого аналізу можуть використовуватися не лише у промисловій розробці, а й в освітньому процесі.

У сучасних умовах аналіз програмного коду дедалі частіше поєднується з інтелектуальними системами підтримки. AI-моделі можуть не лише знайти проблему, а й пояснити її природною мовою, запропонувати альтернативну реалізацію і відповісти на додаткові питання користувача.

1.3 Проблеми ручного рецензування програмного коду

Ручне рецензування коду є ефективним, але ресурсомістким процесом. Воно потребує участі досвідчених розробників, які здатні швидко оцінити логіку, стиль, безпеку та архітектуру запропонованих змін. У реальних умовах такий процес часто обмежується часом, пріоритетами команди та рівнем підготовки учасників.

Однією з проблем ручного code review є суб'єктивність. Різні рецензенти можуть по-різному оцінювати один і той самий фрагмент коду. Частина зауважень може стосуватися стилю, частина - архітектури, а частина - потенційних помилок. Без єдиної системи критеріїв результати перевірки можуть бути непослідовними.

Іншою проблемою є втома рецензента. Великі обсяги коду складно аналізувати уважно, особливо коли перевірка виконується наприкінці робочого дня або паралельно з іншими завданнями. Через це прості помилки можуть залишатися непоміченими.

Автоматизований інструмент не повинен повністю замінювати людину, але може зменшити навантаження на рецензента. Він здатен виконувати первинну перевірку, знаходити типові недоліки, формувати загальну оцінку та виділяти фрагменти, які потребують уваги.

1.4 Метрики оцінювання якості програмного коду

Метрики якості коду є кількісними або якісними показниками, які допомагають оцінити стан програмного продукту. У межах розробленого інструменту використовується узагальнений набір метрик: опрацьованість, продуктивність, оптимізація, безпека та чистота коду.

Опрацьованість характеризує ступінь завершеності та логічної цілісності фрагмента. Код з високою опрацьованістю має зрозумілу структуру, виконує заявлену задачу та не містить очевидних пропусків. Низька опрацьованість може свідчити про неповну реалізацію або неузгодженість окремих частин.

Продуктивність показує, наскільки ефективно код використовує ресурси. При цьому йдеться не лише про швидкість виконання, а й про вибір алгоритмів, уникнення зайвих циклів, повторних обчислень і нераціональної роботи з пам'яттю.

Оптимізація відображає можливість покращення структури коду без зміни його функціонального призначення. Безпека характеризує ризик наявності небезпечних конструкцій, відсутності валідації даних або некоректної роботи з конфіденційною інформацією. Чистота коду пов'язана з читабельністю, стилем, зрозумілими назвами та відсутністю зайвого дублювання.

1.5 Автоматизовані інструменти аналізу програмного забезпечення

Автоматизовані засоби аналізу програмного забезпечення можна умовно поділити на статичні аналізатори, лінтери, інструменти контролю стилю, системи виявлення вразливостей, платформи якості коду та AI-асистенти. Кожна група має власне призначення і обмеження.

Статичні аналізатори перевіряють код без його виконання. Вони ефективні для пошуку типових помилок, небезпечних шаблонів і порушень правил. Лінтери орієнтовані на стиль і синтаксичну коректність. Платформи на зразок SonarQube дозволяють аналізувати великі проєкти, відстежувати технічний борг і контролювати якість у командній розробці.

AI-інструменти відрізняються тим, що здатні працювати з фрагментом коду як з контекстом і формувати відповідь природною мовою. Вони можуть пояснити помилку, запропонувати виправлення або сформулювати рекомендації щодо

архітектури. Водночас їхні відповіді потребують критичного сприйняття, оскільки модель може помилятися.

Розроблений у роботі інструмент не претендує на повну заміну промислових систем статичного аналізу. Його призначення полягає у створенні зручного середовища для швидкого інтелектуального аналізу фрагментів коду з можливістю локального збереження результатів.

1.6 Використання штучного інтелекту в задачах аналізу коду

Великі мовні моделі навчаються на значних обсягах текстових і програмних даних, тому здатні розпізнавати типові шаблони програмування, пояснювати фрагменти коду та генерувати альтернативні варіанти реалізації. У задачах аналізу коду це відкриває можливості для автоматизованого пошуку помилок, рефакторингу та навчальної підтримки.

Перевагою AI-підходу є гнучкість. На відміну від класичного статичного аналізатора, який працює за наперед визначеними правилами, мовна модель може враховувати контекст, стиль запиту та бажаний формат відповіді. Наприклад, можна попросити модель повернути тільки виправлений код, JSON з оцінками або пояснення у форматі Markdown.

Разом з тим, AI-аналіз має ризики. Модель не виконує код і не гарантує абсолютної правильності результату. Вона може запропонувати синтаксично коректний, але логічно невдалий варіант. Тому результати AI-аналізу слід розглядати як рекомендаційні, а не як безумовно правильні.

У розробленому інструменті AI використовується саме як допоміжний модуль. Користувач отримує результат, але остаточне рішення щодо прийняття виправлень або рекомендацій залишається за ним.

1.7 Постановка задачі розробки інструменту оцінки якості ПЗ

На основі аналізу предметної області сформульовано задачу розробити настільний інструмент, який дозволяє виконувати інтелектуальне оцінювання якості програмного коду. Інструмент повинен мати простий інтерфейс, підтримувати

введення фрагмента коду, вибір мови програмування, запуск аналізу, перегляд виправленого коду, метрик і рекомендацій.

Система повинна забезпечувати взаємодію із зовнішнім AI API через ключ доступу. Як приклад у роботі використовується Mistral AI, оскільки він доступний для тестування і дозволяє виконувати запити до мовної моделі. При цьому архітектура не повинна бути жорстко прив'язана до одного постачальника: за потреби можна адаптувати модуль взаємодії до іншого сумісного API.

Важливою вимогою є локальне збереження історії перевірок. Це дозволяє користувачеві повернутися до попередніх фрагментів коду, повторно переглянути результати або використати їх для навчання. Історія має бути обмежена за кількістю записів, щоб не перевантажувати локальну базу даних.

Таким чином, розроблюваний інструмент має поєднати функції AI-аналізу, візуального відображення метрик, консультаційного чату, локальної історії та експорту результатів у звіт.

1.8 Розширений аналіз інструментів-конкурентів

SonarQube та платформи комплексного контролю якості

SonarQube належить до класу платформ безперервної інспекції коду. Такі рішення орієнтовані не на аналіз одного фрагмента, а на систематичну перевірку цілих репозиторіїв. Вони інтегруються з CI/CD-процесами, системами контролю версій і дозволяють відстежувати якість програмного продукту протягом тривалого часу.

Сильним боком SonarQube є наявність quality gates, тобто формалізованих правил прийнятності коду. Команда може визначити мінімальний рівень покриття тестами, допустиму кількість помилок, code smells або вразливостей. Якщо проєкт не проходить ці умови, зміни можуть бути заблоковані до виправлення проблем.

Для великих команд такий підхід є дуже корисним, оскільки якість коду контролюється не епізодично, а як частина інженерного процесу. Проте для невеликого навчального проєкту SonarQube може бути надмірним: потрібно налаштовувати сервер, сканери, профілі якості, підключення до репозиторію та

інтерпретувати досить великий обсяг технічних даних.

Розроблений у роботі інструмент займає іншу нішу. Він не конкурує з SonarQube у сфері корпоративного контролю якості, але є простішим для індивідуального використання. Його можна запустити як настільний застосунок, вставити фрагмент коду і швидко отримати пояснення, виправлення та метрики без повного розгортання платформи.

ESLint, Pylint та класичні літери

Літери є одними з найпоширеніших інструментів контролю якості коду. ESLint використовується для JavaScript та TypeScript-проектів, Pylint - для Python. Їхнє основне призначення полягає у перевірці коду за набором правил, виявленні потенційних помилок, стилістичних порушень і небажаних конструкцій.

Перевагою літерів є детермінованість. Якщо правило налаштоване, інструмент стабільно повідомляє про однакові проблеми в однакових умовах. Це дуже важливо для командної розробки, де потрібно підтримувати єдиний стиль і запобігати появі типових дефектів.

Недоліком літерів у контексті даної роботи є обмеженість пояснювальної складової. Вони добре повідомляють, яке правило порушене, але не завжди пояснюють архітектурний контекст або не пропонують розгорнуте навчальне пояснення. Крім того, літери зазвичай спеціалізовані на конкретній мові або екосистемі.

Розроблений інструмент не замінює ESLint або Pylint, а доповнює їх. Його перевага полягає у тому, що користувач отримує природномовне пояснення, може поставити уточнювальне запитання і побачити не тільки правило, а й можливий напрям покращення коду.

IDE-інспекції JetBrains та Visual Studio Code

Сучасні середовища розробки мають вбудовані або підключені механізми інспекції коду. JetBrains IDE надають широкий набір code inspections, які виявляють помилки, потенційні виключення, дублювання, проблеми зі стилем і неефективні конструкції ще під час редагування. Visual Studio Code підтримує інтеграцію з літерами, мовними серверами та розширеннями.

Перевага IDE-інспекцій полягає у тому, що вони працюють безпосередньо в середовищі розробника. Помилка відображається поруч із рядком коду, часто доступні quick fixes, навігація до документації та автоматичні рефакторинги.

Водночас IDE-інспекції зазвичай прив'язані до конкретного середовища і налаштувань проєкту. Якщо студент або користувач хоче швидко перевірити окремий фрагмент без створення повного проєкту, використання IDE може бути менш зручним.

Розроблений інструмент має інший сценарій: він працює як окрема програма для аналізу вставленого фрагмента. Це робить його придатним для навчальних ситуацій, демонстрацій і швидкого аналізу невеликих прикладів, коли запуск повноцінного IDE або налаштування літера є зайвим.

GitHub pull request review та командні процеси перевірки

GitHub pull request review є важливим елементом командної розробки. Він дозволяє переглядати зміни, залишати коментарі, пропонувати виправлення, погоджувати або відхиляти зміни перед злиттям у основну гілку. Такий підхід добре працює у команді, де є репозиторій, гілки, історія комітів і відповідальні рецензенти.

Сильним боком GitHub review є контекст змін. Рецензент бачить не ізольований фрагмент, а різницю між версіями, обговорення, історію комітів і пов'язані задачі. Це дозволяє приймати рішення не лише на рівні синтаксису, а й з урахуванням цілей розробки.

Однак pull request review не є повністю автоматичним аналізатором. Він потребує участі людей і не завжди підходить для індивідуальної самоперевірки або навчальних фрагментів коду. Студент, який не працює з репозиторієм, не отримає користі від pull request-процесу без додаткової організації проєкту.

Розроблений інструмент можна розглядати як попередній етап перед командним review. Розробник може перевірити фрагмент локально, виправити очевидні проблеми, переглянути метрики, а вже після цього відправляти код у репозиторій або на перевірку іншому учаснику.

Універсальні AI-чати як альтернативний підхід

Окремою групою аналогів є універсальні AI-чати. Вони дозволяють вставити

фрагмент коду і поставити запитання природною мовою. Такий підхід є гнучким і не потребує спеціального програмного інструменту.

Недоліком універсального чату є відсутність спеціалізованого робочого процесу. Користувач сам формулює промпт, сам просить метрики, сам зберігає відповідь і сам структурує результати. Якщо потрібно повторювати перевірки, порівнювати відповіді або формувати звіт, це доводиться робити вручну.

Розроблений інструмент формалізує цей процес. Він має окремі режими для виправлення коду, оцінювання метрик, архітектурних рекомендацій і консультаційного чату. Тобто AI використовується не як окрема розмова, а як частина спеціалізованого інтерфейсу для оцінки якості програмного забезпечення.

Саме ця спеціалізація є головною відмінністю від звичайного AI-чату. Користувач отримує не тільки відповідь моделі, а й структуроване середовище для роботи з результатами аналізу.

Таблиця 1.1 - Порівняльний аналіз аналогічних інструментів

Клас інструменту	Сильні сторони	Обмеження	Позиція розробленого інструменту
SonarQube-платформи	Глибокий аналіз репозиторіїв, quality gates, CI/CD	Потребують налаштування серверної або хмарної інфраструктури	Простіший інструмент для локального аналізу фрагментів
Лінтери	Швидкість, детерміновані правила, інтеграція з редакторами	Обмежене природномовне пояснення, прив'язка до мов	Доповнення через AI-пояснення і консультації
IDE-інспекції	Працюють прямо в середовищі розробника	Залежать від IDE та налаштувань проєкту	Окремий desktop-інструмент без потреби створювати проєкт
Pull request review	Командне обговорення змін у контексті	Потребує команди, репозиторію і	Попередня самоперевірка до командного review

	репозиторію	рецензентів	
Універсальні AI-чати	Гнучкість запитів і природномовні відповіді	Немає спеціалізованої історії, метрик і звіту	Структурований AI-workflow для оцінки якості коду

1.9 Порівняльна характеристика розробленого інструменту

Після аналізу аналогічних рішень можна зробити висновок, що розроблений інструмент займає проміжну позицію між класичним лінтером, універсальним AI-чатом і платформою контролю якості коду. Він не є найпотужнішим засобом для промислового аналізу великих репозиторіїв, але є зручним для швидкої локальної перевірки, навчання і пояснення помилок. Його головна особливість полягає у спеціалізації інтерфейсу. Замість того щоб залишати користувача сам на сам із порожнім чат-вікном, система пропонує готові режими аналізу. Користувачу не потрібно щоразу формулювати промпт для метрик, просити виправлений код або окремо створювати звіт.

Другою важливою перевагою є локальна історія. Універсальні AI-чати можуть зберігати діалоги у власному середовищі, але не завжди зручно повертатися саме до фрагментів коду. У розробленому інструменті історія орієнтована на перевірки і доступна безпосередньо в інтерфейсі.

Третьою перевагою є поєднання кількісного і якісного підходів. Метрики дають швидке числове уявлення про стан коду, а архітектурні рекомендації та чат пояснюють, що саме можна покращити. Це робить результат зрозумілішим для користувача, який навчається програмуванню або хоче швидко оцінити свій фрагмент.

Обмеження розробленого інструменту також є очевидними. Він не виконує код, не будує повний граф залежностей, не аналізує весь репозиторій і не має суворих формальних quality gates. Однак ці обмеження узгоджуються з метою бакалаврської роботи: створити працездатний інструмент оцінки якості ПЗ, який демонструє інтеграцію AI у практичний workflow.

Таким чином, конкурентний аналіз підтверджує актуальність розробки. Існуючі інструменти або надто спеціалізовані, або потребують складного налаштування, або не мають структурованого інтерфейсу для задачі оцінювання якості коду. Розроблений інструмент закриває нішу швидкого, доступного і пояснювального AI-аналізу програмних фрагментів.

Таблиця 1.2 - Порівняння ручного та автоматизованого аналізу коду

Критерій	Ручне code review	Автоматизований AI-аналіз
Швидкість	Залежить від зайнятості рецензента	Результат формується після виконання API-запиту
Суб'єктивність	Висока, залежить від досвіду людини	Нижча, але залежить від якості промпту та моделі
Пояснення	Може бути дуже якісним, але потребує часу	Формується автоматично у природній мові
Масштабування	Потребує залучення додаткових фахівців	Може масштабуватися через API та автоматизацію

Висновки до розділу 1

У першому розділі розглянуто поняття якості програмного забезпечення та показано, що якість не обмежується відсутністю помилок. Вона охоплює підтримуваність, безпеку, продуктивність, зрозумілість і придатність до розвитку.

Встановлено, що аналіз програмного коду є важливим засобом підвищення якості ПЗ. Ручне code review залишається цінним, але потребує значних ресурсів. Автоматизовані та AI-орієнтовані інструменти можуть виконувати первинну перевірку, пояснювати помилки та допомагати розробнику швидше знаходити проблеми.

На основі аналізу предметної області сформульовано задачу розробки інструменту оцінки якості програмного забезпечення, який поєднує AI-аналіз коду, оцінювання метрик, архітектурні рекомендації, консультаційний чат і локальне збереження історії.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ІНСТРУМЕНТУ ДЛЯ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Формування вимог до програмного інструменту

Проєктування програмного засобу починається з визначення вимог. Для розробленого інструменту основною групою користувачів є студенти, початківці-розробники та фахівці, яким потрібно швидко проаналізувати фрагмент коду без розгортання складної серверної інфраструктури.

Функціональні вимоги охоплюють введення коду, вибір мови програмування, запуск аналізу, отримання виправленого варіанта, формування метрик, перегляд архітектурних рекомендацій, роботу з консультативним чатом, збереження історії та експорт звіту.

Нефункціональні вимоги стосуються зручності інтерфейсу, простоти запуску, локального збереження даних, обробки помилок API, можливості роботи у тестовому режимі без справжнього ключа та потенційної заміни AI-провайдера.

2.2 Вибір архітектури програмного засобу

Для реалізації обрано модульну архітектуру настільного застосунку. У ній головний модуль відповідає за побудову інтерфейсу та управління станом, окремий модуль AI-аналізу відповідає за взаємодію із зовнішнім API, модуль бази даних забезпечує збереження історії, а модуль компонентів інтерфейсу містить повторно використовувані візуальні елементи.

Такий підхід є достатнім для бакалаврського проєкту, оскільки не потребує серверної частини, але водночас демонструє розділення відповідальності між компонентами. У майбутньому архітектуру можна розширити шляхом винесення логіки аналізу в окремий сервіс або додавання інтеграції з репозиторіями.

Важливою особливістю архітектури є асинхронне виконання запитів. Аналіз коду складається з кількох незалежних операцій: пошуку помилок, оцінювання метрик і формування архітектурних порад. Їх можна виконувати паралельно, що зменшує загальний час очікування.

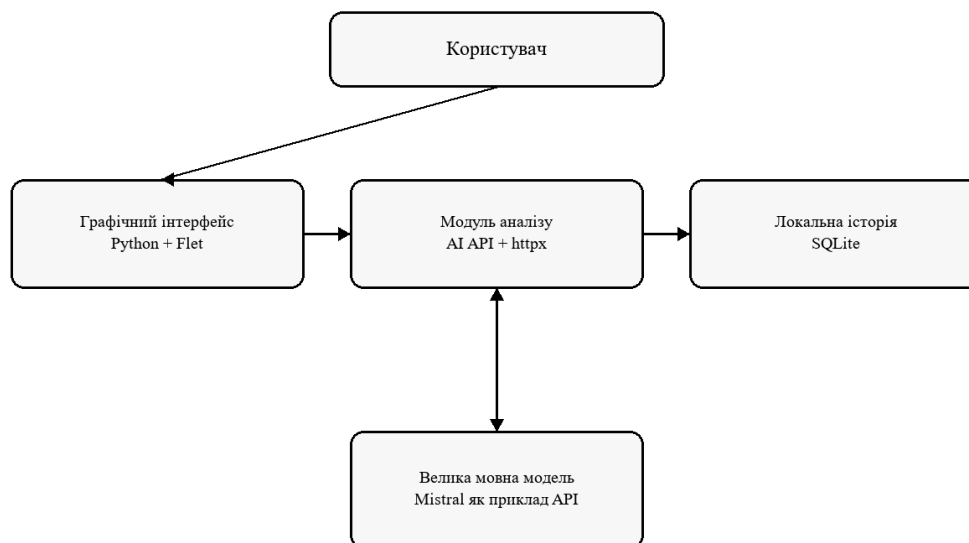


Рисунок 2.1 - Загальна архітектура інструменту оцінки якості ПЗ

2.3 Дерево технологій розробки

Основою програмного засобу є Python. Ця мова обрана через простоту синтаксису, розвинену екосистему бібліотек, підтримку асинхронного програмування та зручність швидкої розробки прототипів.

Для побудови графічного інтерфейсу використано Flet. Він дозволяє створювати desktop-застосунки мовою Python і використовувати компонентну модель, подібну до сучасних UI-фреймворків. Це спрощує розробку інтерфейсу без необхідності переходу до окремого JavaScript-фронтенду.

Для взаємодії із зовнішнім API використовується httpx, оскільки ця бібліотека підтримує асинхронні HTTP-запити. Конфігурація API-ключа виконується через python-dotenv, а локальна історія зберігається у SQLite.

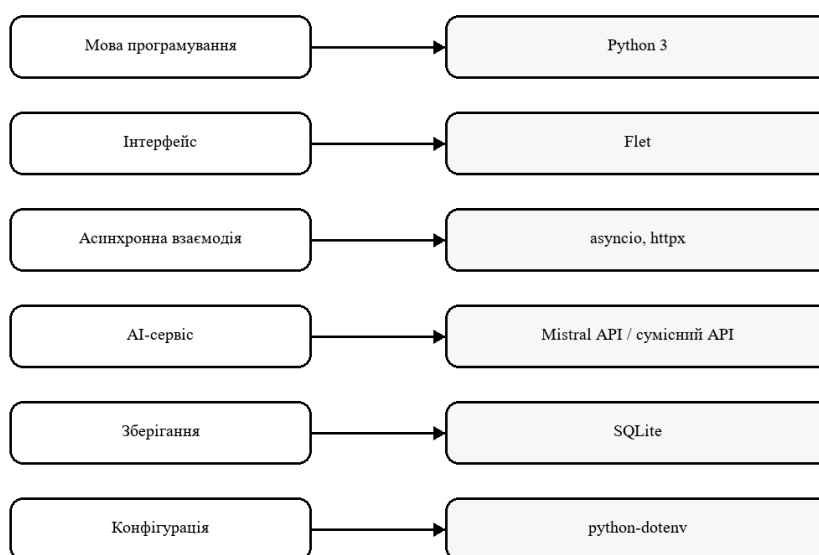


Рисунок 2.2 - Дерево технологій програмного засобу

2.4 Обґрунтування використання великих мовних моделей

Великі мовні моделі доцільно використовувати для задач, у яких потрібне поєднання аналізу тексту, коду та природномовного пояснення. У межах розробленого інструменту модель використовується для трьох основних типів відповідей: виправлений код, JSON-оцінка метрик і Markdown-рекомендації щодо архітектури.

У роботі як приклад використано Mistral AI API. Вибір пояснюється доступністю для тестування, наявністю chat completions endpoint і підтримкою формату відповіді JSON. Водночас архітектурно модуль AI-аналізу може бути адаптований до іншого API, якщо він підтримує подібну модель запитів.

Використання API-ключа дає можливість відокремити застосунок від конкретного користувацького облікового запису. Користувач може вказати власний ключ доступу, а в разі його відсутності система переходить у тестовий режим із демонстраційними відповідями.

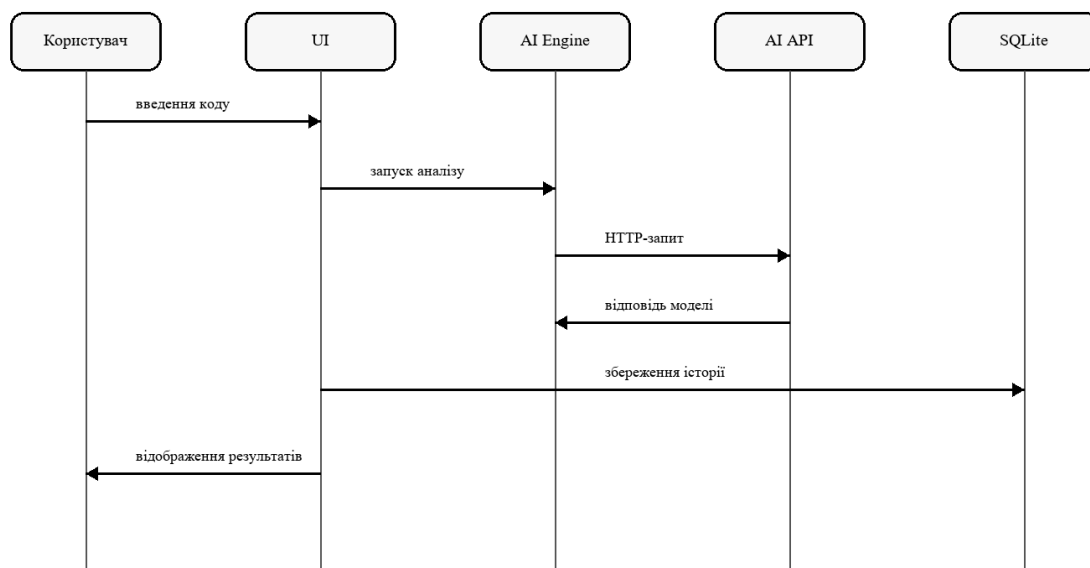


Рисунок 2.3 - Послідовність виконання аналізу програмного коду

2.5 Проєктування структури бази даних

Для локального збереження історії перевірок використовується SQLite. Це вбудована реляційна база даних, яка не потребує окремого серверного процесу і зберігає всі дані в одному файлі. Такий варіант добре підходить для настільного застосунку.

Структура бази даних мінімальна і містить одну таблицю history. У ній зберігаються ідентифікатор запису, текст коду та час створення запису. При додаванні нового запису система автоматично обмежує історію десятима останніми перевітками.

Обмеження історії є практичним рішенням. Воно зменшує ризик неконтрольованого накопичення даних і робить бічну панель інтерфейсу компактною. За потреби у майбутньому можна додати налаштування кількості записів або можливість ручного експорту історії.

history		
Поле	Тип	Призначення
id	INTEGER	PRIMARY KEY AUTOINCREMENT
code	TEXT	Фрагмент програмного коду
timestamp	DATETIME	DEFAULT CURRENT_TIMESTAMP

Рисунок 2.4 - ER-діаграма локальної бази даних

2.6 Проєктування сценаріїв взаємодії користувача із системою

Основний сценарій починається з введення програмного коду в текстове поле. Користувач може вибрати мову програмування або залишити режим автоматичного визначення. Після натискання кнопки аналізу система зберігає фрагмент в історію та запускає паралельні AI-запити.

Після завершення аналізу користувач переходить до області результатів. Він може переглянути виправлений код, оцінки метрик, архітектурні рекомендації або поставити запитання у консультативному чаті. Також доступний експорт звіту у Markdown-файл.

Додатковий сценарій пов'язаний з історією перевірок. Користувач може вибрати попередній запис, повернути код у поле введення та повторно виконати аналіз. Це корисно для порівняння різних варіантів реалізації або продовження роботи над попереднім фрагментом.

Таблиця 2.1 - Функціональні вимоги до системи

ID	Вимога	Опис
F1	Введення коду	Користувач повинен мати можливість вставити фрагмент програмного коду у текстове поле.
F2	Вибір мови	Система повинна підтримувати вибір мови програмування або режим Auto.
F3	AI-аналіз	Система повинна надсилати код до модуля аналізу та отримувати відповідь від AI API.
F4	Метрики	Система повинна відображати оцінки якості за п'ятьма критеріями.
F5	Чат	Користувач повинен мати можливість ставити уточнювальні запитання щодо коду.
F6	Історія	Система повинна зберігати останні перевірки в локальній базі даних.
F7	Експорт	Система повинна формувати Markdown-звіт з результатами аналізу.

Висновки до розділу 2

У другому розділі сформовано вимоги до програмного інструменту та обґрунтовано вибір архітектури. Запропонована структура поєднує графічний інтерфейс, модуль AI-аналізу, модуль бази даних і набір повторно використовуваних UI-компонентів.

Обґрунтовано використання Python, Flet, httpx, python-dotenv, SQLite та зовнішнього AI API. Mistral AI використовується як приклад доступного AI-сервісу, але архітектура не обмежує систему лише цим провайдером.

Спроектowano сценарій взаємодії користувача із системою, структуру локальної бази даних і основні функціональні вимоги, які реалізуються у програмному засобі.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ІНСТРУМЕНТУ

3.1 Загальна структура програмного проєкту

Програмний проєкт має компактну структуру і складається з головного файлу застосунку, папки модулів, бази даних історії, файлу залежностей і супровідної документації. Головний файл відповідає за запуск Flet-застосунку, ініціалізацію сторінки, побудову інтерфейсу та обробку подій користувача.

Модуль `ai_engine.py` містить функції взаємодії із зовнішнім AI API. У ньому реалізовано загальну функцію виклику моделі, режим пошуку помилок, оцінювання метрик, архітектурні рекомендації та консультаційний чат.

Модуль `database.py` відповідає за створення таблиці `history`, додавання нового запису, отримання історії та очищення бази. Модуль `ui_components.py` містить візуальні компоненти: контейнер із напівпрозорим стилем, анімований заголовок, стовпчикову діаграму метрик і повідомлення чату.

3.2 Реалізація графічного інтерфейсу користувача

Графічний інтерфейс реалізовано за допомогою Flet. Сторінка застосунку має темний режим, акцентний колір, заголовок, панель історії, область введення коду, кнопку запуску аналізу та область результатів.

Інтерфейс поділено на дві основні області: бічну панель історії та головну робочу область. У головній області користувач спочатку бачить текстове поле для введення коду. Після запуску аналізу вміст перемикається на область результатів із вкладками.

Для перемикання між різними результатами використано вкладки Bug Hunter, Code Metrics, Architecture та Consult. Такий поділ відповідає різним видам інформації: виправлення, кількісна оцінка, якісні рекомендації та інтерактивне уточнення.

Окремою деталлю інтерфейсу є лічильник рядків і символів. Він оновлюється під час введення коду і допомагає користувачеві оцінити обсяг фрагмента, який буде передано на аналіз.

3.3 Реалізація модуля взаємодії з AI API

Взаємодія з AI API реалізована асинхронно. Функція `call_mistral` формує HTTP POST-запит до `chat completions endpoint`, додає заголовок авторизації з API-ключем і передає список повідомлень у форматі, сумісному з чат-моделями.

Якщо API-ключ відсутній або має значення `token`, функція не виконує мережевий запит, а повертає `None`. Це дає можливість запускати застосунок у демонстраційному режимі без зовнішньої залежності.

Для різних сценаріїв аналізу використовуються різні системні промпти. У режимі пошуку помилок модель отримує інструкцію повернути тільки виправлений код. У режимі метрик модель повинна повернути JSON. У режимі архітектурного аналізу відповідь формується у Markdown.

Такий поділ промπτів дозволяє отримувати результати у форматі, зручному для конкретної вкладки інтерфейсу. Виправлений код відображається як текст, метрики перетворюються на словник і візуалізуються стовпчиками, а рекомендації відображаються як Markdown.

3.4 Реалізація режимів аналізу програмного коду

Режим `Bug Hunter` призначений для пошуку синтаксичних і логічних помилок. Він повертає виправлений варіант коду без додаткових пояснень. Такий формат зручний тоді, коли користувач хоче швидко отримати робочу версію фрагмента.

Режим `Code Metrics` оцінює код за п'ятьма критеріями: опрацьованість, продуктивність, оптимізація, безпека та чистота. Результат очікується у форматі JSON, після чого кожне значення відображається у вигляді вертикального стовпчика.

Режим `Architecture` формує рекомендації щодо покращення структури коду. Він не обмежується виправленням локальних помилок, а звертає увагу на розділення відповідальності, дублювання, іменування, тестованість і підтримуваність.

Режим `Consult` працює як контекстний чат. До повідомлень додається поточний фрагмент коду та історія діалогу, тому користувач може ставити уточнювальні запитання щодо останнього аналізу.

3.5 Реалізація локального збереження історії та експорту звіту

Історія перевірок зберігається у файлі `history.db` за допомогою SQLite. Під час запуску застосунку виконується ініціалізація таблиці, якщо вона ще не створена. Після кожного запуску аналізу поточний код додається до історії.

Щоб історія не зростала без обмежень, після додавання нового запису виконується SQL-запит, який залишає тільки десять останніх записів. Це просте рішення підтримує актуальність бічної панелі та не потребує складної логіки очищення.

Експорт звіту реалізовано шляхом запису Markdown-файлу `review_report.md`. До нього додається заголовок, час аналізу, виправлений код і архітектурні рекомендації. Такий формат легко читати, редагувати та переносити в інші документи.

3.6 Обробка помилок і тестовий режим роботи

Зовнішній AI API може бути недоступний через відсутність інтернету, неправильний ключ, перевищення лімітів або помилки сервера. Тому модуль взаємодії обгортає запит у блок обробки винятків і повертає спеціальний рядок `API_ERROR`.

На рівні інтерфейсу помилка API відображається через повідомлення користувачеві. Це важливо для зручності роботи: застосунок не повинен аварійно завершуватися через тимчасову недоступність зовнішнього сервісу.

Тестовий режим дає можливість продемонструвати роботу інтерфейсу без справжнього API-ключа. У цьому режимі система повертає mock-дані: виправлений код фактично повторює введений фрагмент, а метрики мають демонстраційні значення.

3.7 Особливості захисту даних користувача

Розроблений інструмент працює локально, але під час використання AI API текст коду передається зовнішньому сервісу. Тому користувач повинен розуміти, що конфіденційні фрагменти комерційного коду не варто передавати без відповідного дозволу.

API-ключ зберігається у файлі `.env` і не повинен потрапляти у відкритий репозиторій. Для промислового використання доцільно застосовувати змінні середовища операційної системи або захищене сховище секретів.

Локальна база `history.db` також може містити фрагменти коду, які користувач вставляв у систему. Тому файл бази даних слід розглядати як робочий артефакт, що може містити чутливу інформацію.

Таблиця 3.1 - Структура програмного проєкту

Файл	Призначення
<code>main.py</code>	Запуск застосунку, побудова UI, обробка подій і керування станом.
<code>modules/ai_engine.py</code>	Функції взаємодії з AI API та формування різних типів аналізу.
<code>modules/database.py</code>	Робота з SQLite: ініціалізація, запис, читання та очищення історії.
<code>modules/ui_components.py</code>	Повторно використовувані компоненти інтерфейсу.
<code>requirements.txt</code>	Перелік основних залежностей для запуску застосунку.

Висновки до розділу 3

У третьому розділі описано реалізацію програмного інструменту для оцінки якості програмного забезпечення. Показано структуру проєкту, призначення основних модулів і логіку взаємодії між ними.

Реалізовано графічний інтерфейс, модуль AI-аналізу, локальну базу історії, візуалізацію метрик, консультаційний чат і експорт звіту. Для підвищення зручності використано асинхронні запити, обробку помилок і `mock`-режим.

Розроблена реалізація демонструє практичну можливість поєднання настільного Python-застосунку з великими мовними моделями для підтримки процесу аналізу програмного коду.

РОЗДІЛ 4 ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ІНСТРУМЕНТУ

4.1 Методика тестування програмного засобу

Тестування розробленого інструменту виконується з урахуванням його функціональної структури. Окремо перевіряються введення коду, запуск аналізу, робота з API, відображення результатів, збереження історії, очищення історії, чат і експорт звіту.

Оскільки частина функцій залежить від зовнішнього AI API, доцільно використовувати два режими тестування: з реальним API-ключем і в mock-режимі. Mock-режим дозволяє перевірити інтерфейс, історію та експорт без мережевого з'єднання.

Під час тестування важливо оцінювати не лише коректність функцій, а й поведінку інтерфейсу: чи не блокується сторінка під час очікування відповіді, чи з'являється індикатор завантаження, чи зрозуміло відображаються помилки.

4.2 Перевірка функціональних можливостей

Перевірка основних функцій показує, що користувач може вставити код, вибрати мову програмування, запустити аналіз і отримати результати у відповідних вкладках. Лічильник рядків і символів оновлюється під час редагування поля введення.

Після виконання аналізу система додає запис до історії. Користувач може натиснути на попередній запис, повернути його в поле введення та повторно запустити аналіз. Також доступна кнопка очищення історії.

Експорт Markdown-звіту створює файл, який містить основні результати. Це дозволяє користувачеві зберегти підсумки перевірки та використати їх у документації або навчальних матеріалах.

4.3 Оцінювання швидкодії та зручності використання

Швидкодія системи значною мірою залежить від часу відповіді зовнішнього AI API. Використання `asyncio.gather` дозволяє запускати кілька запитів паралельно, тому

загальний час аналізу наближається до часу найповільнішого запиту, а не до суми всіх запитів.

Зручність використання забезпечується тим, що всі основні дії доступні з одного екрана: введення коду, запуск аналізу, перегляд результатів і повернення до попередніх перевірок. Вкладкова структура дозволяє не перевантажувати екран усією інформацією одночасно.

Для освітнього використання корисною є наявність консультативного чату. Користувач може не лише отримати результат, а й поставити запитання щодо причини помилки або способу покращення коду.

4.4 Порівняння з існуючими рішеннями

Розроблений інструмент не дублює повністю функціональність промислових платформ аналізу коду. Його сильна сторона полягає у простоті запуску, інтерактивності та можливості швидкого AI-пояснення для окремого фрагмента.

Порівняно з класичними літерами, система має більш гнучке пояснення результатів. Порівняно з веб-чатом, вона має спеціалізований інтерфейс для коду, метрик, історії та експорту. Порівняно з повноцінними платформами якості коду, вона простіша і не потребує складного розгортання.

Обмеженням є залежність від зовнішньої моделі та відсутність глибокого аналізу всього репозиторію. Проте для задач швидкого аналізу фрагментів, навчання та демонстрації AI-assisted review інструмент є доцільним.

4.5 Переваги, обмеження та напрями подальшого розвитку

До переваг розробленого інструменту належать простота використання, локальна історія перевірок, наявність кількох режимів аналізу, підтримка консультативного чату, можливість роботи в mock-режимі та зрозуміла модульна структура.

Основні обмеження пов'язані з тим, що AI-відповіді мають рекомендаційний характер. Система не виконує код, не запускає юніт-тести і не гарантує повної правильності виправленого варіанта. Також поточна реалізація працює з

фрагментами коду, а не з повними репозиторіями.

Подальший розвиток може включати інтеграцію з GitHub або GitLab, аналіз pull request, підтримку декількох AI-провайдерів через єдиний інтерфейс, додавання локальних правил статичного аналізу, кешування результатів, розширену систему метрик і формування звітів у DOCX або PDF.

4.6 Методика інтерпретації метрик якості

Оцінювання якості програмного коду за допомогою метрик потребує обережної інтерпретації. Числова оцінка не повинна розглядатися як абсолютний доказ високої або низької якості. Вона є узагальненим сигналом, який допомагає користувачеві швидко визначити, на які аспекти коду варто звернути увагу.

Метрика опрацьованості показує, наскільки код виглядає завершеним і логічно послідовним. Якщо значення низьке, це може означати, що фрагмент містить неповні конструкції, недостатню обробку умов, неузгодженість між частинами алгоритму або відсутність очевидних завершальних дій.

Метрика продуктивності вказує на потенційні проблеми ефективності. Вона не замінює профілювання і фактичного вимірювання часу виконання, але може привернути увагу до зайвих циклів, повторних обчислень, нераціонального використання структур даних або надмірної складності алгоритму.

Метрика оптимізації пов'язана з можливістю спростити код без зміни його поведінки. Висока оцінка означає, що фрагмент має достатньо зрозумілу структуру, тоді як низька може свідчити про дублювання, зайві проміжні змінні, невдалі умовні конструкції або складність, яку можна зменшити.

Метрика безпеки є особливо важливою, оскільки навіть невеликий фрагмент може містити небезпечні рішення: відсутність валідації введення, небезпечну роботу з файлами, логування конфіденційних даних, використання застарілих алгоритмів або пряме збереження секретів у коді.

Метрика чистоти коду відображає читабельність, зрозумілість назв, послідовність стилю та відсутність надмірного ускладнення. Вона важлива не лише для поточного автора, а й для майбутніх розробників, які будуть підтримувати або

змінювати код.

Для практичного використання доцільно розглядати не окреме число, а сукупність метрик. Наприклад, код може мати високу продуктивність, але низьку чистоту, якщо він оптимізований ціною складності. Інший фрагмент може бути добре читабельним, але мати проблеми з безпекою. Саме тому візуальне порівняння п'яти показників є корисним.

У межах розробленого інструменту метрики формуються мовною моделлю, тому вони мають рекомендаційний характер. Остаточне рішення повинен приймати користувач після аналізу виправленого коду, рекомендацій і власного розуміння задачі. Такий підхід відповідає ролі інструменту як помічника, а не автоматичного судді якості.

Таблиця 4.1 - Орієнтовна інтерпретація числових метрик

Діапазон оцінки	Умовна інтерпретація	Рекомендована дія користувача
0-39	Низький рівень показника	Потрібно уважно переглянути фрагмент і виправити основні проблеми.
40-69	Середній рівень показника	Доцільно проаналізувати рекомендації та покращити окремі частини коду.
70-89	Достатній рівень показника	Код загалом прийнятний, але може мати локальні недоліки.
90-100	Високий рівень показника	Код виглядає якісним за цим критерієм, але все одно потребує тестування.

4.7 Сценарії практичного використання інструменту

Перший сценарій використання пов'язаний із навчанням програмуванню. Студент виконує лабораторну роботу, вставляє фрагмент коду в інструмент і отримує первинний аналіз. Після цього він може порівняти власне рішення з виправленим варіантом і зрозуміти, чому певна конструкція є небажаною.

Другий сценарій стосується самоперевірки перед передаванням коду на

перевірку викладачу або керівнику. Інструмент дозволяє швидко знайти очевидні проблеми, покращити читабельність і підготувати більш акуратний варіант роботи.

Третій сценарій пов'язаний з роботою початківця-розробника. Коли розробник не впевнений у якості фрагмента, він може отримати швидку рекомендацію щодо структури, безпеки або продуктивності. Це особливо корисно тоді, коли поруч немає досвідченого рецензента.

Четвертий сценарій полягає у використанні інструменту для демонстрації принципів code review. Викладач або наставник може показати один і той самий фрагмент у різних режимах: виправлення помилок, метрики, архітектурні поради та пояснення через чат.

П'ятий сценарій стосується підготовки документації. Після аналізу користувач може експортувати Markdown-звіт і включити його до навчального звіту, технічного опису або матеріалів для обговорення. Це зменшує потребу вручну копіювати результати з інтерфейсу.

Шостий сценарій передбачає порівняння декількох варіантів реалізації. Користувач може проаналізувати початковий варіант, внести зміни, повторно виконати аналіз і порівняти метрики та рекомендації. Такий процес сприяє поступовому покращенню коду.

Сьомий сценарій пов'язаний із швидким аналізом чужого прикладу. Якщо користувач знаходить фрагмент коду у відкритому джерелі або навчальному матеріалі, він може перевірити його якість, отримати пояснення і краще зрозуміти сильні та слабкі сторони рішення.

У всіх перелічених сценаріях інструмент виконує роль допоміжного засобу. Він прискорює первинний аналіз і пояснення, але не скасовує необхідності тестування, критичного мислення і перевірки результату людиною.

Таблиця 4.2 - Практичні сценарії використання розробленого інструменту

Сценарій	Користувач	Очікувана користь
Навчання програмуванню	Студент	Отримання пояснень і виправлень для власного коду.
Самоперевірка	Студент або розробник	Зменшення кількості очевидних помилок перед здачею або review.
Демонстрація code review	Викладач або наставник	Наочне пояснення метрик, помилок і рекомендацій.
Порівняння реалізацій	Розробник	Оцінка впливу змін на якість фрагмента.
Підготовка звіту	Користувач інструменту	Швидке збереження результатів аналізу в Markdown.

4.8 Методика оцінювання достовірності AI-відповідей

Оскільки розроблений інструмент використовує велику мовну модель, важливо окремо розглянути питання достовірності отриманих відповідей. На відміну від класичного компілятора або статичного аналізатора, мовна модель не виконує формальної перевірки програми. Вона генерує відповідь на основі статистичних закономірностей, контексту запиту та інструкцій, сформульованих у промпті.

Першим критерієм достовірності є синтаксична коректність запропонованого коду. Якщо модель повертає виправлений фрагмент, користувач повинен перевірити, чи відповідає він синтаксису обраної мови програмування. У майбутньому цю перевірку можна автоматизувати шляхом підключення компілятора, інтерпретатора або мовного сервера.

Другим критерієм є логічна відповідність вихідній задачі. Навіть якщо код синтаксично правильний, він може змінити поведінку програми. Тому користувач повинен порівнювати запропонований варіант із початковим наміром: чи збереглася основна функціональність, чи не зникли важливі умови, чи не було додано небажаних побічних ефектів.

Третім критерієм є пояснюваність рекомендації. Якщо модель пропонує змінити

архітектуру, винести функцію або замінити алгоритм, така рекомендація повинна бути зрозумілою. Нечіткі поради на зразок 'покращити структуру' без конкретизації мають меншу практичну цінність, ніж рекомендації з прикладом і поясненням причини.

Четвертим критерієм є стабільність відповіді. Для однакових або схожих фрагментів модель бажано повинна давати узгоджені оцінки. Якщо при повторних запитах метрики сильно відрізняються без очевидної причини, користувач повинен сприймати результат обережно і покладатися на додаткову перевірку.

П'ятим критерієм є узгодженість метрик із текстовими рекомендаціями. Якщо система виставляє високий показник безпеки, але в архітектурних рекомендаціях вказує на відсутність валідації введення, це є сигналом до ручного перегляду. Такі суперечності можливі через імовірнісний характер генерації.

Для практичного використання доцільно застосовувати правило подвійної перевірки: AI-відповідь розглядається як початкова рекомендація, після чого користувач перевіряє її за допомогою запуску коду, тестів, лінера або власного аналізу. Це дозволяє отримати переваги AI-підходу без некритичного прийняття кожної відповіді.

У межах роботи розроблений інструмент позиціонується саме як система підтримки прийняття рішення. Він пришвидшує аналіз і допомагає сформулювати перше уявлення про якість коду, але остаточна відповідальність за використання виправлень залишається за розробником.

Таблиця 4.3 - Критерії оцінювання достовірності AI-відповідей

Критерій	Питання для перевірки	Практичне значення
Синтаксис	Чи компілюється або запускається запропонований код?	Дозволяє відсіяти очевидно некоректні відповіді.
Логіка	Чи збереглася початкова поведінка програми?	Запобігає прийняттю виправлення, яке змінює сенс алгоритму.
Пояснюваність	Чи зрозуміло, чому запропоновано зміну?	Підвищує навчальну цінність аналізу.

Стабільність	Чи узгоджені відповіді при повторному аналізі?	Показує надійність рекомендацій для схожих фрагментів.
Узгодженість	Чи не суперечать метрики текстовим порадам?	Допомагає виявляти слабкі або випадкові відповіді моделі.

4.9 Критерії приймального тестування програмного засобу

Приймальне тестування визначає, чи відповідає розроблений інструмент поставленим вимогам і чи може бути використаний за призначенням. Для даного проекту приймальні критерії мають охоплювати не лише технічний запуск програми, а й повний користувацький сценарій від введення коду до отримання звіту.

Першим критерієм є успішний запуск застосунку без додаткової серверної інфраструктури. Користувач повинен мати можливість встановити залежності, запустити головний файл і побачити графічний інтерфейс. Якщо для демонстрації потрібен API-ключ, система все одно повинна мати зрозумілий mock-режим.

Другим критерієм є коректна робота поля введення. Воно повинно підтримувати багаторядковий код, не втрачати форматування, дозволяти вставлення з буфера обміну і оновлювати статистику рядків та символів. Для інструменту аналізу коду це важливо, оскільки структура фрагмента має значення.

Третім критерієм є успішне виконання аналізу. Після натискання кнопки система повинна зберегти фрагмент в історію, показати індикатор очікування, виконати AI-запити або mock-відповіді та відобразити результат у вкладках. Користувач не повинен залишатися без зрозумілого стану системи.

Четвертим критерієм є коректна робота вкладок результатів. Виправлений код має бути доступним для перегляду і копіювання, метрики повинні відображатися у вигляді зрозумілих індикаторів, архітектурний аналіз має підтримувати Markdown, а консультаційний чат повинен зберігати контекст останнього коду.

П'ятим критерієм є робота з історією перевірок. Нові записи повинні додаватися до бази, старі записи мають обмежуватися встановленою кількістю, а очищення історії повинно оновлювати інтерфейс без перезапуску програми.

Шостим критерієм є експорт звіту. Після натискання відповідної кнопки повинен створюватися Markdown-файл із результатами аналізу. У звіті мають бути наявні основні блоки: заголовок, час аналізу, виправлений код і архітектурні рекомендації.

Сьомим критерієм є стійкість до помилок. Якщо API недоступний або ключ неправильний, застосунок не повинен аварійно завершуватися. Він має повідомити користувача про проблему і повернути інтерфейс у робочий стан.

Таблиця 4.4 - Приймальні критерії програмного засобу

Критерій	Умова приймання	Статус у реалізації
Запуск	Застосунок відкривається після запуску main.py	Реалізовано
Введення коду	Підтримується багаторядкове поле та статистика введення	Реалізовано
AI-аналіз	Виконуються три основні режими аналізу	Реалізовано
Історія	Останні перевірки зберігаються у SQLite	Реалізовано
Чат	Користувач може ставити уточнювальні питання	Реалізовано
Експорт	Формується Markdown-звіт	Реалізовано
Помилки API	Система не завершується аварійно при збої API	Реалізовано частково, потребує деталізації повідомлень

4.10 Оцінювання практичної ефективності використання інструменту

Практична ефективність розробленого інструменту визначається не лише наявністю функцій, а й тим, наскільки ці функції скорочують час первинного аналізу коду та роблять результат зрозумілішим для користувача. Для навчального і прикладного використання важливо, щоб інструмент не створював додаткового навантаження, а спрощував роботу з помилками і рекомендаціями.

У традиційному сценарії студент або розробник самотійно шукає помилку, читає документацію, звертається до викладача або колеги, виправляє код і лише після

цього оцінює, чи стало рішення кращим. Такий процес є корисним для навчання, але може займати багато часу, особливо якщо проблема типова і вже багато разів зустрічалася в інших прикладах.

Розроблений інструмент прискорює первинний етап. Користувач вставляє код і отримує одразу декілька типів результатів: виправлений фрагмент, числові метрики, архітектурні рекомендації та можливість поставити уточнювальне запитання. Це дозволяє швидше перейти від невизначеного стану 'код неякісний або не працює' до конкретного списку дій.

Ефективність особливо помітна у повторюваних навчальних ситуаціях. Наприклад, якщо студент часто допускає схожі помилки з обробкою винятків, назвами змінних або дублюванням логіки, інструмент може багаторазово показувати такі проблеми у різних фрагментах. З часом це формує навичку самостійного розпізнавання дефектів.

Ще одним аспектом ефективності є зменшення бар'єра для code review. Багато початківців сприймають перевірку коду як складний або критичний процес. Інструмент у desktop-форматі дозволяє виконати попередню перевірку без зовнішнього тиску, отримати рекомендації і підготувати кращий варіант перед показом іншій людині.

У командному середовищі подібний інструмент може виконувати роль попереднього фільтра. Розробник перевіряє фрагмент локально, виправляє очевидні проблеми, а вже потім передає зміни на повноцінний review. Це не усуває потребу в командному контролі, але зменшує кількість дрібних зауважень і дає рецензенту змогу зосередитися на складніших питаннях.

Для оцінювання практичної ефективності можна використовувати такі показники: час до першої рекомендації, кількість знайдених типових проблем, зрозумілість пояснень, частка рекомендацій, які користувач прийняв після ручної перевірки, та зручність повторного доступу до історії аналізу.

Таким чином, ефективність інструменту полягає не в абсолютній заміні людини або формальних аналізаторів, а в прискоренні первинного аналізу, підвищенні зрозумілості результатів і створенні зручного середовища для поступового

покращення програмного коду.

Таблиця 4.5 - Показники практичної ефективності інструменту

Показник ефективності	Спосіб оцінювання	Очікуваний ефект
Час до першого результату	Вимірювання часу від натискання кнопки аналізу до появи відповіді	Швидке отримання первинного зворотного зв'язку.
Зрозумілість рекомендацій	Оцінка користувачем після перегляду пояснень	Підвищення навчальної цінності інструменту.
Кількість знайдених проблем	Порівняння початкового фрагмента і рекомендацій	Виявлення типових помилок до ручного review.
Повторне використання історії	Перевірка звернення до попередніх записів	Зручне повернення до раніше проаналізованого коду.
Прийнятність виправлень	Ручна перевірка запропонованого коду користувачем	Оцінка практичної корисності AI-відповіді.

Таблиця 4.6 - Тестові сценарії програмного засобу

№	Сценарій	Очікуваний результат
1	Запуск застосунку	Відкривається головне вікно з полем введення та історією.
2	Введення коду	Оновлюється лічильник рядків і символів.
3	Запуск аналізу	Виконуються AI-запити, відображається індикатор очікування.
4	Перегляд метрик	П'ять значень відображаються у вигляді стовпчикових індикаторів.
5	Використання чату	Повідомлення користувача та відповідь AI додаються до списку.
6	Очищення історії	Таблиця history очищується, бічна панель оновлюється.
7	Експорт звіту	Створюється файл review_report.md.

Таблиця 4.7 - Порівняння підходів до аналізу коду

Інструмент	Основне призначення	Порівняння з розробленим засобом
Лінтери	Перевірка стилю та типових синтаксичних проблем	Менш гнучкі у поясненнях, але швидкі та детерміновані.
SonarQube	Комплексний контроль якості проєктів	Потужніший для команд, але потребує складнішого розгортання.
Веб-чат з LLM	Універсальна взаємодія з мовною моделлю	Не має спеціалізованої історії, метрик і вкладок аналізу.
Розроблений інструмент	Швидкий AI-аналіз фрагментів коду у desktop UI	Простий для запуску, спеціалізований під навчальні та прикладні задачі.

Висновки до розділу 4

У четвертому розділі розглянуто методику тестування та оцінювання ефективності розробленого інструменту. Перевірено основні сценарії використання: введення коду, запуск аналізу, перегляд результатів, роботу з історією, чат і експорт звіту.

Показано, що система є зручною для швидкого аналізу фрагментів коду і може використовуватися як допоміжний інструмент у навчанні та розробці. Її перевагою є поєднання AI-аналізу, метрик, рекомендацій і локальної історії в одному інтерфейсі.

Водночас визначено обмеження: залежність від AI API, рекомендаційний характер відповідей і відсутність аналізу повних репозиторіїв. Ці обмеження формують напрями подальшого розвитку системи.

РОЗДІЛ 5 АНАЛІЗ РИЗИКІВ, БЕЗПЕКИ ДАНИХ ТА МОЖЛИВОСТЕЙ ВПРОВАДЖЕННЯ

5.1 Аналіз ризиків використання AI API

Використання зовнішнього AI API дає змогу швидко інтегрувати можливості великої мовної моделі у прикладний програмний інструмент, однак таке рішення має власні ризики. Найважливішими серед них є залежність від доступності зовнішнього сервісу, обмеження тарифного плану, зміни у форматі API, можливість помилкових відповідей моделі та передавання користувацького коду за межі локального середовища.

Залежність від зовнішнього сервісу означає, що працездатність частини функцій інструменту визначається не лише якістю локального коду, а й станом мережевого з'єднання та стабільністю AI-платформи. Якщо сервіс тимчасово недоступний, основні режими аналізу не можуть повернути реальний результат. Саме тому в реалізації передбачено обробку помилок і тестовий mock-режим.

Окремий ризик пов'язаний з тим, що відповідь великої мовної моделі має ймовірнісний характер. Модель може сформулювати переконливу відповідь, яка не є повністю правильною. Для задач оцінювання якості коду це означає, що результати слід сприймати як рекомендації, а не як формальну верифікацію програмного продукту.

У межах бакалаврської роботи цей ризик знижується шляхом чіткого позиціонування інструменту як допоміжного засобу. Він не замінює тестування, статичний аналізатор або експертне code review, а доповнює їх швидким поясненням і первинними рекомендаціями.

Ще одним важливим аспектом є можливість зміни API з боку постачальника. Якщо формат запиту або відповіді зміниться, модуль взаємодії потребуватиме оновлення. Тому логіку роботи з AI API доцільно ізолювати в окремому модулі, що і було зроблено у розробленому проєкті.

Таким чином, використання AI API є виправданим для навчального та прикладного інструменту, але потребує обережного трактування результатів, обробки

помилки і можливості заміни постачальника AI-сервісу у майбутньому.

5.2 Захист API-ключів та конфігураційних даних

API-ключ є персональним засобом доступу до зовнішнього сервісу. Якщо він потрапить до сторонніх осіб, його можуть використати для несанкціонованих запитів, що призведе до витрат, блокування облікового запису або порушення правил використання сервісу.

У розробленому інструменті ключ зберігається у файлі `.env`, а доступ до нього здійснюється через бібліотеку `python-dotenv`. Такий підхід є зручним для локальної розробки, оскільки ключ не записується безпосередньо в програмний код і може змінюватися без редагування файлів застосунку.

Водночас файл `.env` не повинен передаватися разом із відкритою версією проєкту. Якщо проєкт буде розміщуватися у системі контролю версій, файл з ключем необхідно додати до `.gitignore`. Для прикладу або навчальної передачі проєкту краще створювати файл `.env.example`, у якому вказано тільки назву змінної без реального значення ключа.

Для промислового використання більш доцільним є збереження секретів у змінних середовища операційної системи або у спеціалізованому сховищі секретів. Це зменшує ризик випадкового розкриття ключа і дозволяє централізовано керувати доступом.

Окрему увагу потрібно приділити повідомленням про помилки. Система не повинна виводити API-ключ у логах, повідомленнях інтерфейсу або експортованих звітах. У поточному рішенні повідомлення про помилку має бути достатнім для користувача, але не повинно розкривати конфіденційні параметри підключення.

Захист конфігураційних даних є необхідною умовою безпечного використання будь-якого інструменту, що взаємодіє із зовнішніми сервісами. Навіть у навчальному проєкті це питання важливе, оскільки формує правильну культуру роботи з секретами.

5.3 Конфіденційність програмного коду користувача

Оскільки інструмент аналізує фрагменти програмного коду, потрібно враховувати питання конфіденційності. Код може містити комерційну логіку, фрагменти алгоритмів, налаштування, приклади ключів, адреси сервісів або інші дані, які не призначені для передавання третім сторонам.

Під час використання зовнішнього AI API текст фрагмента передається за межі локального комп'ютера. Це є принциповою відмінністю від класичного локального статичного аналізатора, який працює без мережевого обміну. Тому користувач має усвідомлювати, що для конфіденційного коду потрібні додаткові правила використання.

Для навчальних прикладів і невеликих демонстраційних фрагментів такий підхід є прийнятним. Однак для комерційного середовища потрібно або отримати дозвіл на передавання коду до AI-сервісу, або використовувати локальну модель, або додати попередню анонімізацію фрагмента.

Локальна історія перевірок також є джерелом потенційного ризику. Файл history.db містить тексти, які користувач вставляв у систему. Якщо цей файл буде скопійовано або передано разом з проєктом, у ньому можуть залишитися фрагменти чутливого коду.

Для зменшення цього ризику в інтерфейсі передбачено очищення історії. Крім того, у майбутньому доцільно додати налаштування, яке дозволить повністю вимикати збереження історії, обмежувати час зберігання або шифрувати локальну базу даних.

Отже, питання конфіденційності не заперечує доцільності розробленого інструменту, але визначає правила його відповідального використання. У роботі інструмент розглядається насамперед як навчальний і допоміжний засіб, а не як система для обробки закритого промислового коду без додаткових заходів безпеки.

5.4 Практичне впровадження інструменту в навчальний процес

Розроблений програмний інструмент має практичну цінність для навчального процесу на кафедрі інформаційних технологій. Його можна використовувати під час

вивчення програмування, основ програмної інженерії, якості програмного забезпечення, тестування та розробки користувацьких інтерфейсів.

Студент може вставити власний фрагмент коду і швидко отримати пояснення проблем, виправлений варіант та рекомендації щодо структури. Це не скасовує роль викладача, але дозволяє студенту отримувати первинний зворотний зв'язок до задачі лабораторної або курсової роботи.

Викладач може використовувати інструмент для демонстрації типових помилок. Наприклад, один і той самий фрагмент коду можна проаналізувати у різних режимах: спочатку показати виправлення, потім метрики, а потім архітектурні рекомендації. Такий підхід робить процес обговорення якості коду більш наочним.

Окремою перевагою є консультаційний чат. Він дозволяє ставити уточнювальні запитання: чому певна конструкція є проблемною, як скоротити функцію, як розділити відповідальність між модулями, чому потрібно обробляти помилки або як покращити назви змінних.

Інструмент також може використовуватися для самоперевірки перед захистом лабораторних робіт. Студент бачить не тільки факт помилки, а й пояснення, тому поступово формує навички самостійного аналізу коду.

Таким чином, розроблений інструмент може стати додатковим навчальним засобом, який підтримує практичне засвоєння принципів якості програмного забезпечення та сприяє підвищенню культури програмування.

5.5 Перспективи розвитку розробленого програмного засобу

Поточна реалізація є завершеним прототипом, однак вона залишає значний простір для розвитку. Найочевиднішим напрямом є підтримка декількох AI-провайдерів через єдиний інтерфейс. Користувач міг би обирати Mistral, OpenAI, Anthropic, Gemini або локальну модель залежно від наявного ключа, вартості та вимог до конфіденційності.

Другим напрямом є аналіз не лише окремого фрагмента коду, а й повного проекту. Для цього потрібно додати вибір папки або репозиторію, обхід файлів, фільтрацію за розширеннями, обмеження розміру запиту і формування підсумкового

звіту за декількома файлами.

Третім напрямом є інтеграція з системами контролю версій. Інструмент може аналізувати зміни у pull request, порівнювати стару і нову версії коду, виявляти ризикові зміни та формувати коментарі для розробника.

Четвертим напрямом є додавання локальних правил статичного аналізу. Це дозволить поєднати детерміновані перевірки з AI-рекомендаціями. Наприклад, локальний аналізатор може знаходити занадто довгі функції, небезпечні виклики або відсутність обробки винятків ще до звернення до AI API.

П'ятим напрямом є покращення системи звітності. Замість Markdown-файлу можна формувати DOCX або PDF-звіт з таблицями метрик, прикладами виправлень, графіками та підсумковими рекомендаціями. Це зробить інструмент кориснішим для командної роботи та навчальної звітності.

Отже, розроблений інструмент має потенціал для перетворення з навчального desktop-застосунку на повноцінну систему підтримки code review. Основна перевага поточної архітектури полягає у тому, що вона вже розділяє інтерфейс, AI-логіку та збереження даних, тому її можна розширювати без повного переписування проєкту.

Таблиця 5.1 - Основні ризики використання інструменту та способи їх зменшення

Ризик	Можливий наслідок	Спосіб зменшення
Недоступність AI API	Неможливість отримати реальний результат аналізу	Обробка помилок, mock-режим, підтримка альтернативних API
Витік API-ключа	Несанкціоноване використання облікового запису	Зберігання ключа поза кодом, .gitignore, змінні середовища
Передавання конфіденційного коду	Розкриття комерційної або приватної логіки	Попередження користувача, анонімізація, локальні моделі
Помилкова відповідь моделі	Некоректне виправлення або неправильна рекомендація	Критична перевірка користувачем, тестування, порівняння з локальними правилами

Накопичення історії	Збереження чутливих фрагментів у history.db	Очищення історії, обмеження записів, можливість вимкнення історії
---------------------	---	---

Висновки до розділу 5

У п'ятому розділі розглянуто ризики, пов'язані з використанням зовнішнього AI API, збереженням API-ключів, конфіденційністю програмного коду та локальною історією перевірок.

Показано, що ці ризики не заперечують доцільності використання інструменту, але потребують правильних організаційних і технічних заходів: обробки помилок, захисту ключів, очищення історії та критичного ставлення до AI-рекомендацій.

Також визначено можливості впровадження інструменту в навчальний процес і сформульовано напрями подальшого розвитку, зокрема підтримку декількох AI-провайдерів, аналіз репозиторіїв, інтеграцію з системами контролю версій і розширену звітність.

ВИСНОВКИ

У кваліфікаційній роботі розглянуто задачу розробки інструменту для оцінки якості програмного забезпечення. У процесі виконання роботи проаналізовано поняття якості ПЗ, роль аналізу програмного коду, проблеми ручного code review, можливості автоматизованих інструментів і перспективи використання великих мовних моделей.

На основі аналізу предметної області сформовано вимоги до програмного засобу. Було визначено, що інструмент повинен підтримувати введення коду, вибір мови програмування, запуск AI-аналізу, отримання виправленого коду, оцінювання метрик, архітектурні рекомендації, консультаційний чат, локальне збереження історії та експорт результатів.

Для реалізації обрано стек технологій Python, Flet, httpx, python-dotenv і SQLite. Як приклад зовнішнього AI-сервісу використано Mistral AI API, однак архітектура допускає підключення іншого сумісного API через відповідний ключ доступу.

Розроблено настільний програмний інструмент, який виконує інтелектуальний аналіз фрагментів коду та надає користувачеві результати у декількох формах: виправлений код, метрики якості, архітектурні рекомендації та відповіді консультаційного чату. Реалізовано локальну історію перевірок і можливість експорту Markdown-звіту.

Проведене тестування підтвердило працездатність основних функцій системи. Інструмент може використовуватися для навчальних задач, самоперевірки коду та як допоміжний засіб первинного аналізу якості програмного забезпечення.

Подальший розвиток системи доцільно спрямувати на підтримку декількох AI-провайдерів, аналіз повних репозиторіїв, інтеграцію з системами контролю версій, додавання локальних правил статичного аналізу та формування розширених звітів у форматах DOCX і PDF.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Python Software Foundation. Python 3 Documentation. URL: <https://docs.python.org/3/>
2. Flet. Official documentation. URL: <https://flet.dev/docs/>
3. SQLite Consortium. SQLite Documentation. URL: <https://www.sqlite.org/docs.html>
4. Mistral AI. API Documentation. URL: <https://docs.mistral.ai/>
5. Encode OSS. HTTPX Documentation. URL: <https://www.python-httpx.org/>
6. Python-dotenv project documentation. URL: <https://pypi.org/project/python-dotenv/>
7. ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE).
8. SonarSource. SonarQube Documentation. URL: <https://docs.sonarsource.com/sonarqube/>
9. ESLint. Find and fix problems in your JavaScript code. URL: <https://eslint.org/>
10. Pylint. User Manual. URL: <https://pylint.pycqa.org/>
11. GitHub Docs. About pull request reviews. URL: <https://docs.github.com/>
12. OWASP Foundation. Source Code Analysis Tools. URL: <https://owasp.org/>
13. OWASP Foundation. Code Review Guide. URL: <https://owasp.org/www-project-code-review-guide/>
14. Microsoft. Visual Studio Code Documentation. URL: <https://code.visualstudio.com/docs>
15. JetBrains. Code inspections documentation. URL: <https://www.jetbrains.com/help/>
16. OpenAI. API documentation. URL: <https://platform.openai.com/docs/>
17. Anthropic. Claude API documentation. URL: <https://docs.anthropic.com/>
18. Google. Gemini API documentation. URL: <https://ai.google.dev/docs>
19. IEEE Computer Society. Software Engineering Body of Knowledge (SWEBOK).
20. PEP 8 - Style Guide for Python Code. URL: <https://peps.python.org/pep-0008/>
21. NIST. Secure Software Development Framework (SSDF). URL: <https://csrc.nist.gov/>

22. OWASP Foundation. Top 10 Web Application Security Risks. URL:
<https://owasp.org/www-project-top-ten/>

ДОДАТОК А. ГРАФІЧНІ МАТЕРІАЛИ

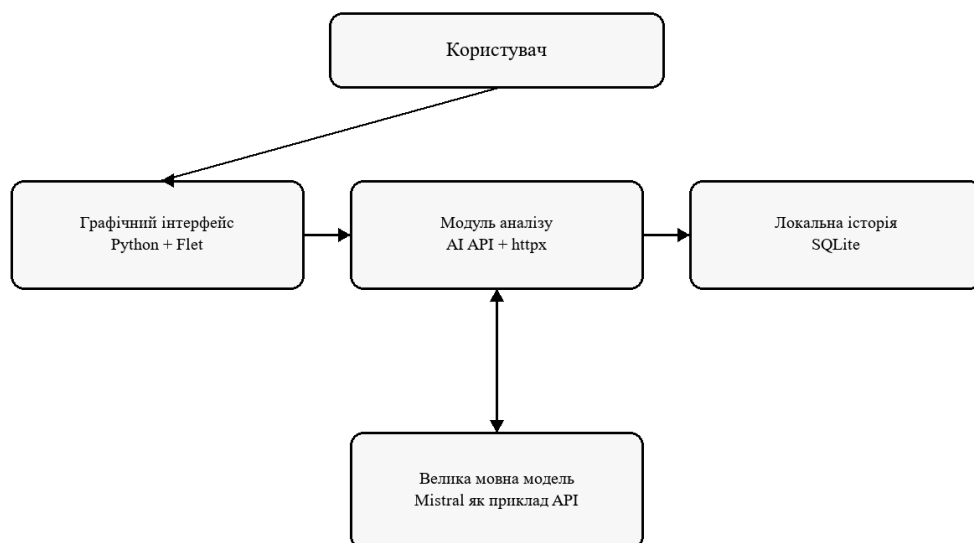


Рисунок А.1 - Архітектура програмного засобу

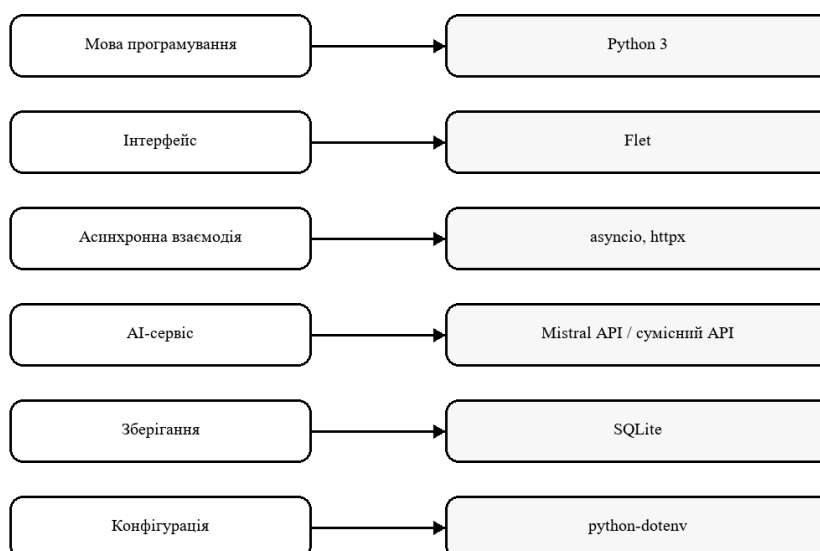


Рисунок А.2 - Дерево технологій

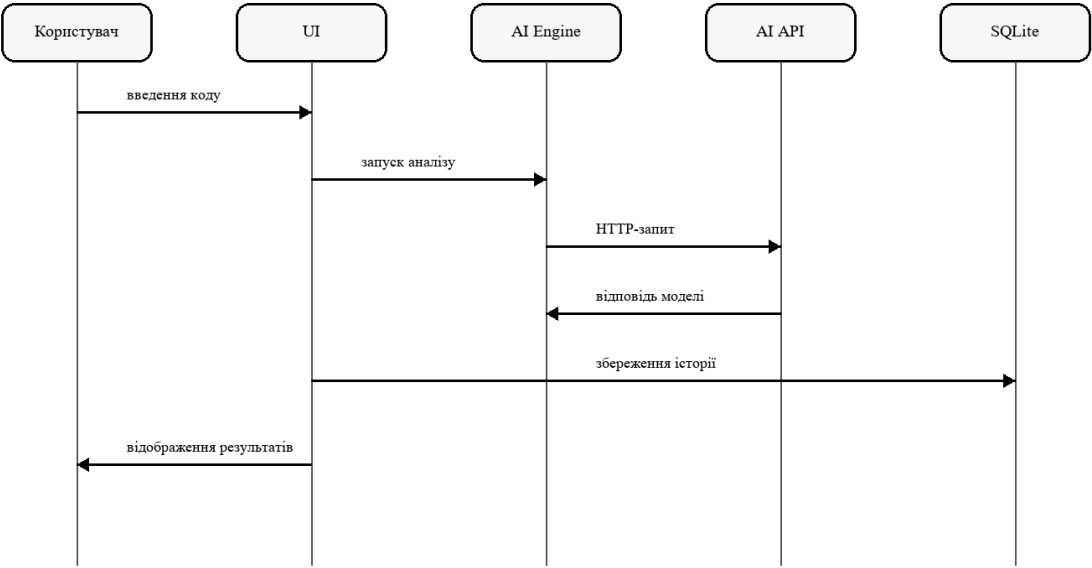


Рисунок А.3 - Діаграма послідовності

history		
Поле	Тип	Призначення
id	INTEGER	PRIMARY KEY AUTOINCREMENT
code	TEXT	Фрагмент програмного коду
timestamp	DATETIME	DEFAULT CURRENT_TIMESTAMP

Рисунок А.4 - ER-діаграма бази даних

ДОДАТОК Б. ФРАГМЕНТИ ПРОГРАМНОГО КОДУ

Б.1 Виклик зовнішнього AI API

```

    async with httpx.AsyncClient(timeout=30.0) as client:
resp = await client.post(url, headers=headers, json=payload)
resp.raise_for_status()
data = resp.json()
return data['choices'][0]['message']['content']

```

Б.2 Збереження історії перевірок

```

    cursor.execute('INSERT INTO history (code) VALUES (?)', (code,))
cursor.execute("""
DELETE FROM history
WHERE id NOT IN (
    SELECT id FROM history ORDER BY id DESC LIMIT 10
)
""")

```

Б.3 Паралельний запуск аналізу

```

    bug_res, met_res, arch_res = await asyncio.gather(
bug_hunter(code, lang, roast),
analyze_code_metrics(code, lang),
architecture_refactor(code, lang, roast)
)

```

ДОДАТОК В. ПРИКЛАДИ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМИ



Рисунок В.1 - Приклад екрана роботи програмного інструменту

Останні перевірки



2026-04-19 15:13
<!DOCTYPE html> <ht...

2026-04-19 15:10
import sys # Прос...

2026-04-19 15:05
import sys # Прос...

2026-04-19 15:00
import sys # Прос...

2026-04-19 14:58
import sys # Прос...

Рисунок В.2 - Приклад екрана роботи програмного інструменту

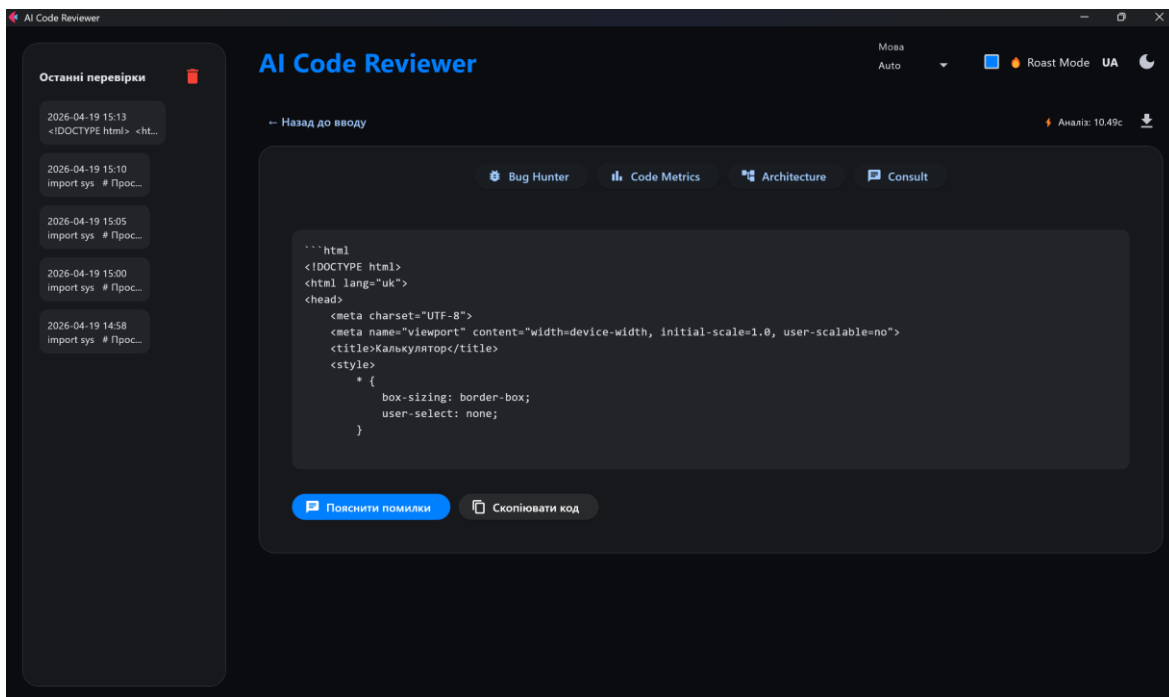


Рисунок В.3 - Приклад екрана роботи програмного інструменту



Рисунок В.4 - Приклад екрана роботи програмного інструменту

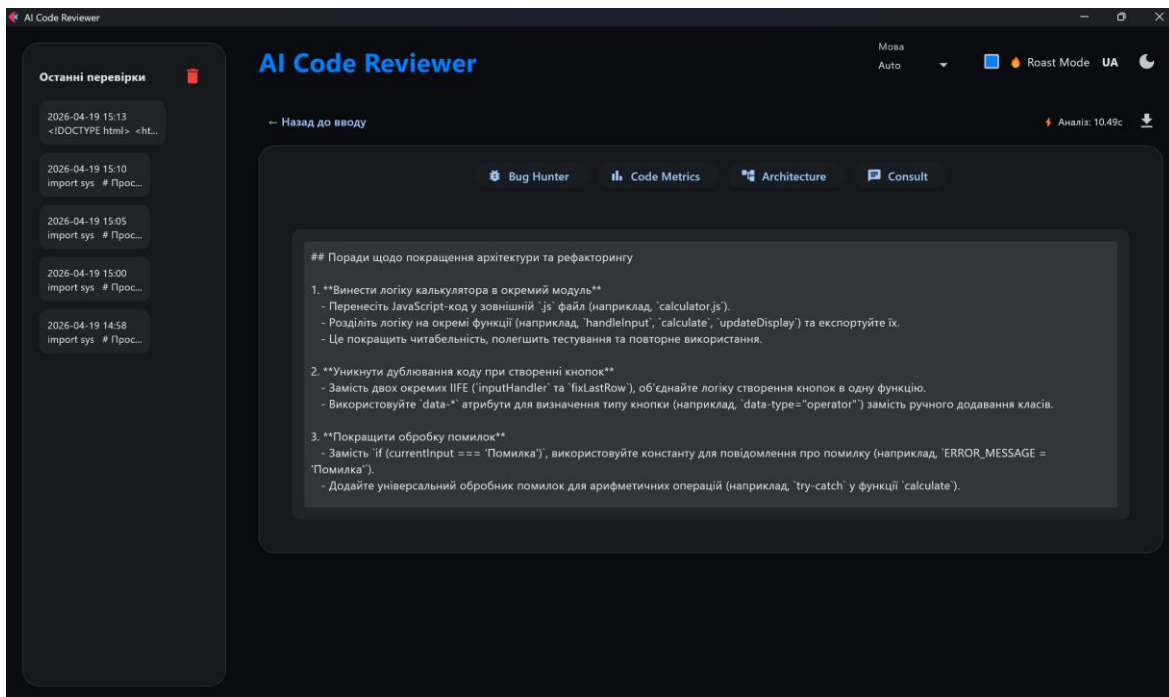


Рисунок В.5 - Приклад екрана роботи програмного інструменту

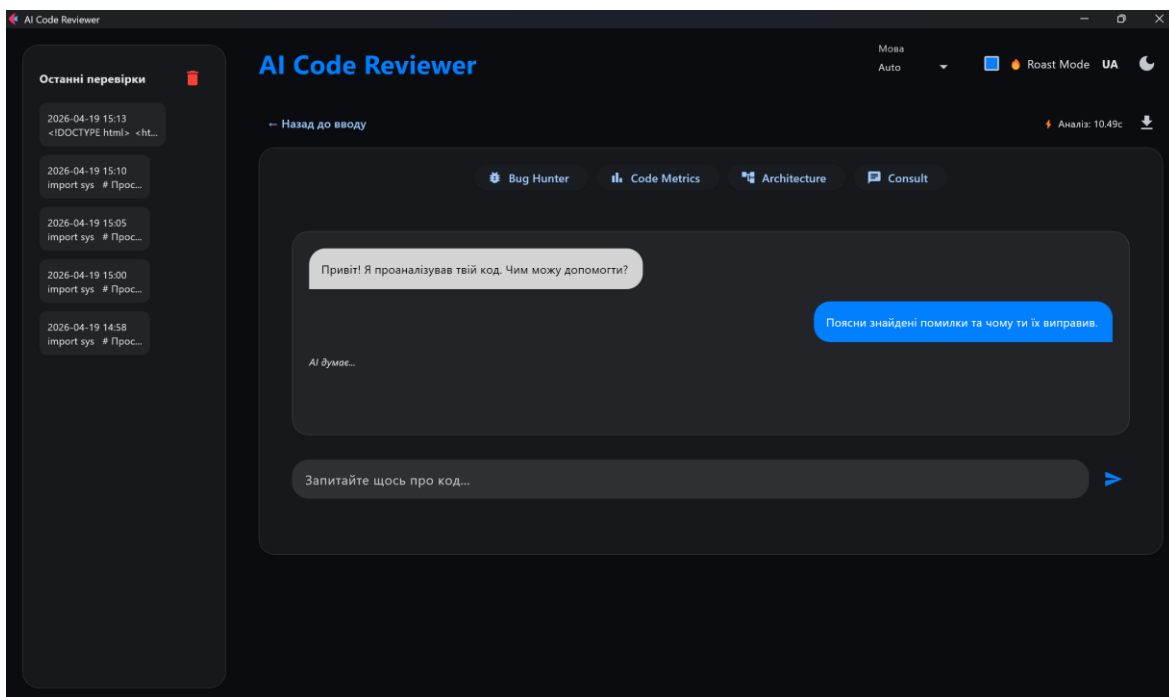


Рисунок В.6 - Приклад екрана роботи програмного інструменту

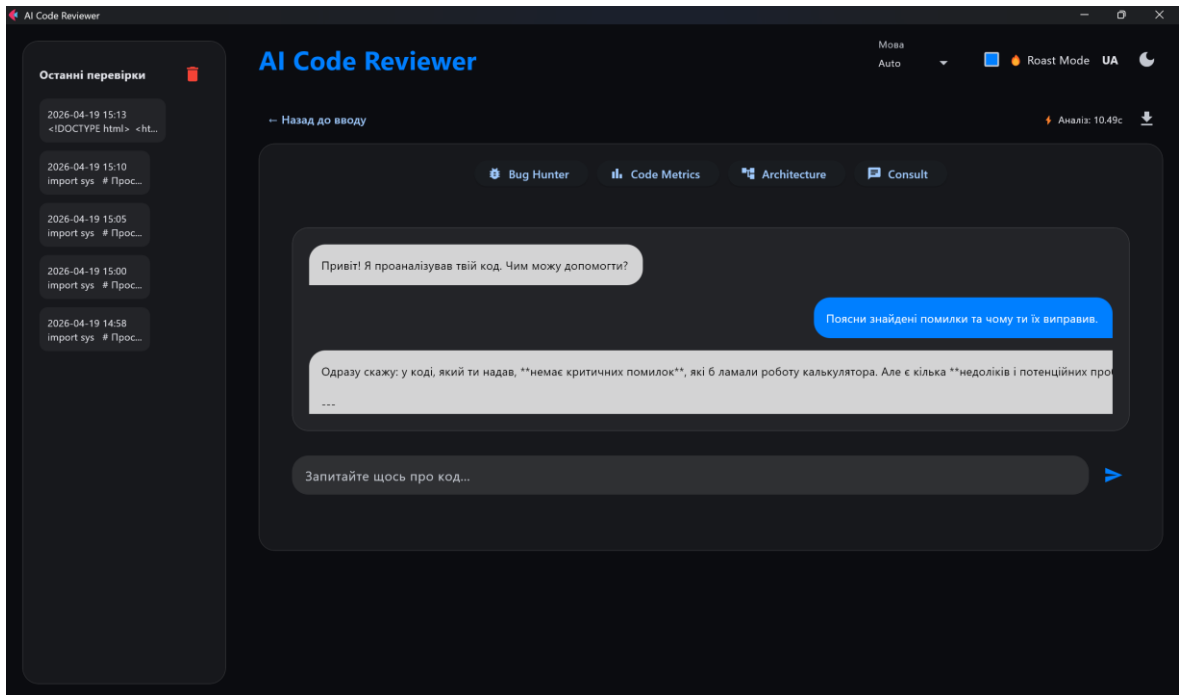


Рисунок В.7 - Приклад екрана роботи програмного інструменту

ДОДАТОК Г. РОЗШИРЕНЕ ОБҐРУНТУВАННЯ ПРОЄКТНИХ РІШЕНЬ

Обґрунтування вибору настільного формату застосунку

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів

і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів.

Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору настільного формату застосунку пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи настільний формат є доцільним для навчального і локального використання, оскільки не потребує розгортання серверної інфраструктури, спрощує демонстрацію результатів і дозволяє зберігати історію перевірок без зовнішньої бази даних. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого,

зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами

використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування вибору локальної бази даних пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи SQLite забезпечує достатню функціональність для збереження короткої історії перевірок, не потребує адміністрування і добре відповідає масштабу розробленого інструменту. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою

інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування

Обґрунтування використання асинхронного програмування пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого,

зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання асинхронного програмування пов'язане з

практичними умовами використання розробленого інструменту. У межах даної роботи асинхронна модель виконання дозволяє паралельно запускати декілька незалежних запитів до AI API і зменшувати сумарний час очікування результату. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку.

У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та

придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування використання метрик якості пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи метрики створюють зручний спосіб швидкої інтерпретації стану фрагмента коду, навіть якщо вони мають рекомендаційний характер і залежать від відповіді мовної моделі. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку.

У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування консультаційного режиму пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи чат із контекстом перевіреного коду перетворює інструмент із простого генератора відповіді на інтерактивний засіб навчання і уточнення технічних рішень. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та

придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку.

У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення

коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.

Обґрунтування експорту результатів пов'язане з практичними умовами використання розробленого інструменту. У межах даної роботи збереження результатів у Markdown дозволяє використовувати аналіз у звітах, навчальних матеріалах і подальшому документуванні роботи над програмним кодом. Це рішення не є випадковим, оскільки воно узгоджується з метою створення простого, зрозумілого та придатного до розширення програмного засобу для оцінки якості програмного забезпечення.

Під час проєктування враховувалося, що користувач не повинен виконувати складне налаштування середовища. Основний сценарій має складатися з введення коду, запуску аналізу, перегляду результатів і, за потреби, експорту звіту. Саме тому кожне технічне рішення оцінювалося з позиції простоти, підтримуваності та відповідності навчально-прикладному характеру роботи.

Також важливо, що обране рішення залишає можливість подальшого розвитку. У майбутньому систему можна доповнити новими режимами аналізу, підтримкою інших AI API, інтеграцією з репозиторіями або формуванням складніших звітів. Отже, поточна реалізація є не лише завершеним прототипом, а й основою для наступних ітерацій розвитку.