Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка мультиагентної AI-системи для рев'ю та аналізу коду з використанням фреймворку LangGraph

Виконав: студент IV курсу, групи СП-41
спеціальності 121 – Інженерія програмного забезпечення
(шифр і назва спеціальності)

________ Абрамов С.С.
(підпис) (прізвище та ініціали)

Керівник ________ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Нормоконтроль ________ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри ________ Петрик М.Р.
(підпис) (прізвище та ініціали)

Рецензент ________ ________
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії

(повна назва факультету)

Кафедра програмної інженерії

(повна назва кафедри)

ЗАТВЕРДЖУЮ

Завідувач кафедри

____________ проф. Петрик М.Р.

(підпис) (прізвище та ініціали)

« 6 » квітня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр

(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення

(шифр і назва спеціальності)

студенту Абрамову Святославу Сергійовичу

1. Тема роботи Розробка мультиагентної AI-системи для рев’ю та аналізу коду з використанням фреймворку LangGraph

Керівник роботи Стоянов Юрій Миколайович, к.т.н. доц. каф. ПІ

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом по університету від « 06 » квітня 2026 року № _________

2. Термін подання студентом роботи 22.06.2026

3. Вихідні дані до роботи наукові літературні джерела

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

Вступ

Аналіз предметної області та постановка задачі

Проєктування та реалізація

Тестування

Безпека життєдіяльності та основи охорони праці

Висновки

Слайди презентації

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці			

7. Дата видачі завдання ___6 квітня_________ 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з завданням кваліфікаційної роботи	13.04.2026	Виконано
2	Збір та аналіз інформації за темою дослідження: огляд LLM-агентів, фреймворків LangGraph/LangChain, multi-agent архітектур.	13.04.2026 - 16.04.2026	Виконано
3	Формування структури пояснювальної записки	17.04.2026 18.04.2026 -	Виконано
4	Розробка технічного завдання, проектування архітектури	20.04.2026	Виконано
5	Проектування та реалізація мультиагентної структури на базі фреймворку LangGraph	21.04.2026 - 30.04.2026	Виконано
6	Тестування функціоналу системи, оцінка точності розробленої системи	01.05.2026 - 03.05.2026	Виконано
7	Написання розділів пояснювальної записки (1–2 розділи)	05.05.2026 - 06.05.2026	Виконано
8	Написання розділу «Безпека життєдіяльності та основи охорони праці»	07.05.2026 - 09.05.2026	Виконано
9	Оформлення висновків, списку використаних джерел, додатків	11.05.2026 - 14.05.2026	Виконано
10	Перевірка роботи керівником, внесення правок		
11	Нормоконтроль		
12	Перевірка кваліфікаційної роботи на плагіат		
13	Попередній захист кваліфікаційної роботи		
14	Захист кваліфікаційної роботи		

Студент ____________________ ________________________________
(підпис) (прізвище та ініціали)

Керівник роботи ____________________ ________________________________
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Абрамов С. С. Розробка мультиагентної AI-системи для рев'ю та аналізу коду з використанням фреймворку LangGraph : робота на здобуття кваліфікаційного ступеня бакалавра : спец. 121 – інженерія програмного забезпечення / наук. кер. Стоянов Ю. М. Тернопіль : ТНТУ ім. Івана Пулюя, 2026. 56 с. // С. – 56, табл. – 3, рис. – 14, бібліогр. – 29, слайд. – 17, додат. – 2.

Ключові слова: мультиагентна система, аналіз коду, LangGraph, великі мовні моделі, FastAPI, VS Code, статичний аналіз, безпека коду, агентна архітектура.

Метою роботи є розробка мультиагентної AI-системи для автоматизованого рев'ю коду на основі LangGraph із паралельною роботою семи спеціалізованих агентів, REST API на FastAPI та розширенням для VS Code.

У першому розділі проаналізовано сучасні інструменти статичного аналізу та рев'ю коду, розглянуто принципи побудови мультиагентних AI-систем, сформульовано мету й задачі роботи.

У другому розділі спроектовано мультиагентний граф LangGraph із маршрутизатором і сімома паралельними агентами, а також розширення VS Code.

У третьому розділі описано юніт-, інтеграційні та end-to-end проведено верифікацію системи.

У четвертому розділі розглянуто питання безпеки життєдіяльності при роботі з комп'ютерною технікою та основи охорони праці розробника.

Об'єктом дослідження є процес автоматизованого рев'ю та аналізу програмного коду засобами мультиагентних AI-систем.

Предметом дослідження є методи та технології побудови мультиагентних систем для аналізу коду на основі LangGraph і великих мовних моделей. Методи дослідження включають: аналіз аналогів, UML-моделювання, агентну оркестрацію та оцінювання якості агентів за кількісними метриками.

ABSTRACT

Sviatoslav Abramov. Development of a multi-agent AI system for code review and analysis using the LangGraph framework : bachelor thesis : specialty 121 "Software Engineering" / scientific supervisor Yuriy Stoyanov. Ternopil Ivan Puluj National Technical University, 2026. 56 p. // Pages – 56, tables – 3, figures – 14, references – 29, presentation slides – 17, appendices – 2.

Keywords: multi-agent system, code analysis, LangGraph, large language models, FastAPI, VS Code, static analysis, code security, agent architecture.

The purpose of the work is to develop a multi-agent AI system for automated code review based on LangGraph with seven parallel specialized agents, a REST API built with FastAPI, and a VS Code extension.

This paper covers the analysis of the subject domain, including modern static analysis and code review tools and multi-agent AI system principles, followed by goal formulation, actor identification, and key use case description. The system architecture is designed around a LangGraph multi-agent graph with a central router and seven parallel agents; PostgreSQL database schemas and UML class diagrams are constructed; a REST API and a VS Code extension with an interactive report and inline diagnostics are implemented. The quality of the system is evaluated through unit, integration, and end-to-end with system verification. Workplace safety guidelines for software developers are also discussed.

The object of the study is the process of automated code review and analysis using multi-agent AI systems.

The subject of the study is methods and technologies for building multi-agent systems for code analysis based on LangGraph and large language models. The research methods include: analysis of analogues, UML architecture modeling, agent orchestration, and quantitative evaluation of agent quality metrics.

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

AI – штучний інтелект.

БД – база даних.

ОС – операційна система.

ПЗ – програмне забезпечення.

API – інтерфейс програмування застосунків (Application Programming Interface).

AST – абстрактне синтаксичне дерево (Abstract Syntax Tree).

CI/CD – неперервна інтеграція та неперервне розгортання (Continuous Integration / Continuous Delivery).

Docker – платформа для контейнеризації та ізольованого розгортання застосунків.

FastAPI – веб-фреймворк для розробки REST API на мові Python.

HTTP – протокол передачі гіпертексту (HyperText Transfer Protocol).

JSON – текстовий формат обміну даними (JavaScript Object Notation).

LangGraph – фреймворк для побудови мультиагентних систем на основі графів стану.

LLM – велика мовна модель (Large Language Model).

PostgreSQL – об'єктно-реляційна система керування базами даних з відкритим вихідним кодом.

REST – архітектурний стиль побудови розподілених веб-сервісів (Representational State Transfer).

SOLID – п'ять принципів об'єктно-орієнтованого проектування програмного забезпечення.

SQL – мова структурованих запитів (Structured Query Language).

UML – уніфікована мова моделювання (Unified Modeling Language).

VS Code – Visual Studio Code, редактор вихідного коду від компанії Microsoft.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 9
1.1 Аналіз предметної області 9
1.2 Аналіз існуючих рішень 10
1.2.1 GitHub Copilot 10
1.2.2 Amazon CodeGuru Reviewer 11
1.2.3 SonarQube 12
1.2.4 Snyk Code 13
1.3 Порівняльний аналіз рішень та переваги розробленої системи 14
1.4 Висновки до першого розділу 16
2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ 17
2.1 Вибір технологій та засобів розробки 17
2.2 Загальна архітектура системи 18
2.3 Діаграма варіантів використання 20
2.4 Проектування мультиагентного конвеєра LangGraph 21
2.4.1 Граф стану та акумулятори 22
2.4.2 Вузол маршрутизатора та розгалуження Send() 22
2.5 Діаграма станів завдання на рев’ю 23
2.6 UML-діаграма класів системи 25
2.7 Діаграма послідовності взаємодії компонентів 26
2.8 Побудова схеми бази даних 27
2.9 Реалізація основних класів та методів 29
2.9.1 Базовий клас агента BaseReviewAgent 29
2.9.2 Побудова та компіляція графу LangGraph 30
2.9.3 Фабрика LLM-провайдерів ProviderFactory 31
2.9.4 REST API та персистентність 31
2.10 Розробка розширення для VS Code 32
2.11 Висновки до другого розділу 34
3 ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ 36

3.1 Загальна стратегія тестування 36
3.2 Модульне тестування 37
3.2.1 Тестування маршрутизатора (RouterNode) 37
3.2.2 Тестування стану графу та дедуплікації 37
3.2.3 Тестування фабрики LLM-провайдерів 38
3.2.4 Тестування інструментів статичного аналізу 38
3.3 Інтеграційне тестування 40
3.3.1 Тестування REST API (test_api.py) 40
3.3.2 Тестування LangGraph-конвеєра (test_graph.py) 40
3.3.3 Тестування агентів за провайдерами (test_agents.py) 41
3.4 Наскрізне тестування 42
3.5 Верифікація системи 43
3.6 Висновки до третього розділу 43
4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ 44
4.1 Долікарська допомога при опіках 44
4.2 Естетичне оформлення робочого місця оператора ПК, установки 46
4.3 Висновки до четвертого розділу 48
ВИСНОВКИ 50
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 51
ДОДАТКИ 54

ВСТУП

Сучасний розвиток програмного забезпечення супроводжується стрімким зростанням обсягів вихідного коду та підвищенням вимог до його якості й безпеки. Ручне рев'ю коду потребує значних затрат часу і залежить від досвіду конкретного розробника, що знижує стабільність результату. Поява великих мовних моделей (LLM) відкрила можливість для глибокого семантичного аналізу коду, а фреймворки мультиагентної оркестрації, зокрема LangGraph, дозволяють розподілити аналіз між спеціалізованими агентами. Розробка таких систем є актуальним і практично значущим напрямом у галузі інженерії програмного забезпечення.

Метою роботи є розробка мультиагентної AI-системи для автоматизованого рев'ю та аналізу програмного коду з використанням фреймворку LangGraph. Для досягнення поставленої мети необхідно вирішити такі задачі:

– проаналізувати існуючі інструменти статичного аналізу та рев'ю коду, дослідити принципи побудови мультиагентних AI-систем;

– спроектувати архітектуру мультиагентної системи на основі LangGraph із центральним маршрутизатором і сімома паралельними спеціалізованими агентами;

– реалізувати асинхронний REST API на FastAPI з персистентністю результатів у базі даних PostgreSQL;

– розробити розширення для редактора VS Code з інтерактивним звітом і вбудованими діагностиками у панелі проблем;

– провести тестування системи та оцінити якість роботи агентів за кількісними метриками.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Перший розділ присвячено аналізу предметної галузі автоматизованого рев'ю та аналізу програмного коду, а також постанову задачі. Розглядаються сучасний стан галузі, ключові технологічні підходи — від статичного аналізу до мультиагентних LLM-систем — та наявні конкурентні рішення, що дозволяє обґрунтувати доцільність розробки нової системи і визначити її конкурентні переваги.

1.1 Аналіз предметної області

Рев'ю коду є одним із фундаментальних процесів у сучасній розробці програмного забезпечення. Дослідження Bacchelli та Bird [1] встановили, що основними цілями рев'ю коду є виявлення дефектів, обмін знаннями між розробниками та забезпечення відповідності стандартам якості. Дослідження McIntosh et al. [2] підтвердило, що підвищення покриття та залученості до рев'ю коду статистично зменшує кількість постпродуктивних дефектів у великих відкритих проектах.

Традиційні підходи до автоматизованого аналізу коду поділяються на два покоління. Перше покоління — правило-орієнтовані статичні аналізатори (ESLint, PyFlakes, Checkstyle, PMD) — виявляють синтаксичні помилки та порушення стилю, спираючись на наперед визначені шаблони. Попри ефективність у формальній перевірці, такі інструменти нездатні оцінити семантику коду, виявити логічні помилки або врахувати контекст бізнес-задачі. Johnson et al. [3] встановили, що розробники часто відмовляються від статичних аналізаторів через велику кількість хибних спрацювань та відсутність пояснень.

Поява великих мовних моделей (LLM) відкрила якісно нові можливості для семантичного аналізу коду. Chen et al. [4] продемонстрували, що модель Codex, навчена на мільярдах рядків коду з GitHub, здатна генерувати функціонально коректний код та виявляти дефекти на рівні, порівнянному з початківцями-розробниками. Наступні покоління моделей — GPT-4 [5], Claude 3

та Gemini 1.5 — суттєво підвищили якість аналізу: вони здатні виявляти тонкі логічні помилки, оцінювати відповідність принципам SOLID та генерувати детальні пояснення природною мовою. Tian et al. [11] підтвердили, що сучасні LLM конкурентоспроможні з досвідченими розробниками у виявленні широкого спектра дефектів.

Однак один LLM-агент з одним промптом не може ефективно охопити всі виміри якості коду одночасно: надмірно широкий контекст знижує увагу моделі до конкретних аспектів. Мультиагентні системи вирішують цю проблему шляхом декомпозиції задачі між спеціалізованими агентами [6]. Фреймворк LangGraph дозволяє побудувати такі системи у вигляді графа стану (StateGraph), де паралельне виконання агентів забезпечується механізмом умовних переходів Send(), а результати накопичуються у спільному стані через reducer-функції [7]. Такий підхід поєднує спеціалізацію кожного агента з паралельною обробкою, що суттєво скорочує загальний час аналізу.

1.2 Аналіз існуючих рішень

Для обґрунтування проектних рішень та визначення конкурентних переваг розробленої системи було проведено огляд чотирьох провідних інструментів автоматизованого рев'ю та аналізу коду. Кожен із них розглянуто з точки зору архітектурного підходу, підтримуваних мов програмування, можливостей інтеграції та обмежень.

1.2.1 GitHub Copilot

GitHub Copilot — AI-асистент від компаній GitHub та Microsoft, що використовує великі мовні моделі серії GPT-4 для автоматичного доповнення коду, генерації функцій і пояснення логіки [8]. У 2024 році GitHub розширив платформу функцією Copilot Code Review, яка автоматично переглядає pull request і залишає inline-коментарі природною мовою з описом проблеми та пропонованим виправленням. Сервіс інтегрований у VS Code, JetBrains IDE та GitHub.com і налічує понад 1,8 млн корпоративних користувачів.

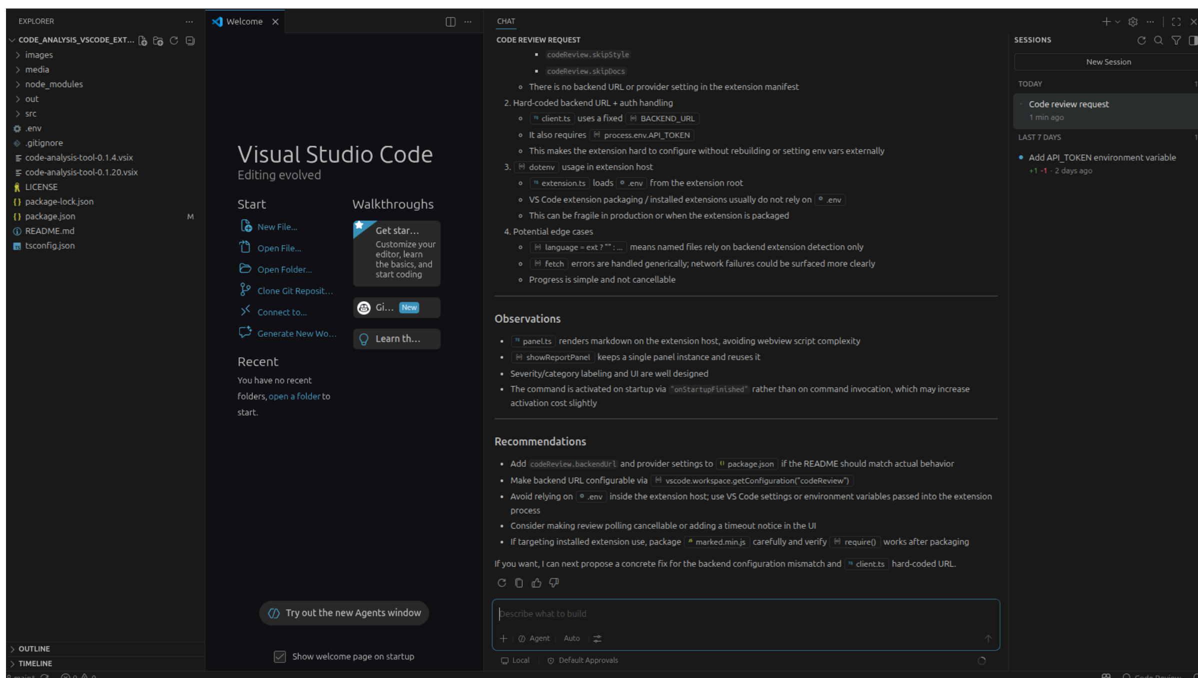

Рисунок 1.1 – Інтерфейс GitHub Copilot Code Review у редакторі VS Code

Основними перевагами GitHub Copilot є висока якість LLM-генерації, широка підтримка мов (20+) та безшовна інтеграція з платформою GitHub. Проте система є закритою (proprietary) і прив’язаною до хмарної інфраструктури Microsoft: відсутній вибір LLM-провайдера, неможливе самостійне розгортання. Рев’ю виконує єдиний агент без спеціалізації за видами аналізу, що обмежує глибину перевірки кожного аспекту якості.

1.2.2 Amazon CodeGuru Reviewer

Amazon CodeGuru Reviewer — ML-сервіс від Amazon Web Services для автоматичного рев’ю коду на Java та Python. Систему навчено на мільярдах рядків коду з відкритих репозиторіїв і внутрішніх проектів Amazon; вона виявляє дефекти безпеки, проблеми з ресурсами (витоки пам’яті, незакриті з’єднання) та шаблони типових помилок. CodeGuru Reviewer інтегрується з AWS CodeCommit, GitHub, Bitbucket і GitLab через CI/CD-пайплайни AWS і надає API для програмного доступу до результатів аналізу.

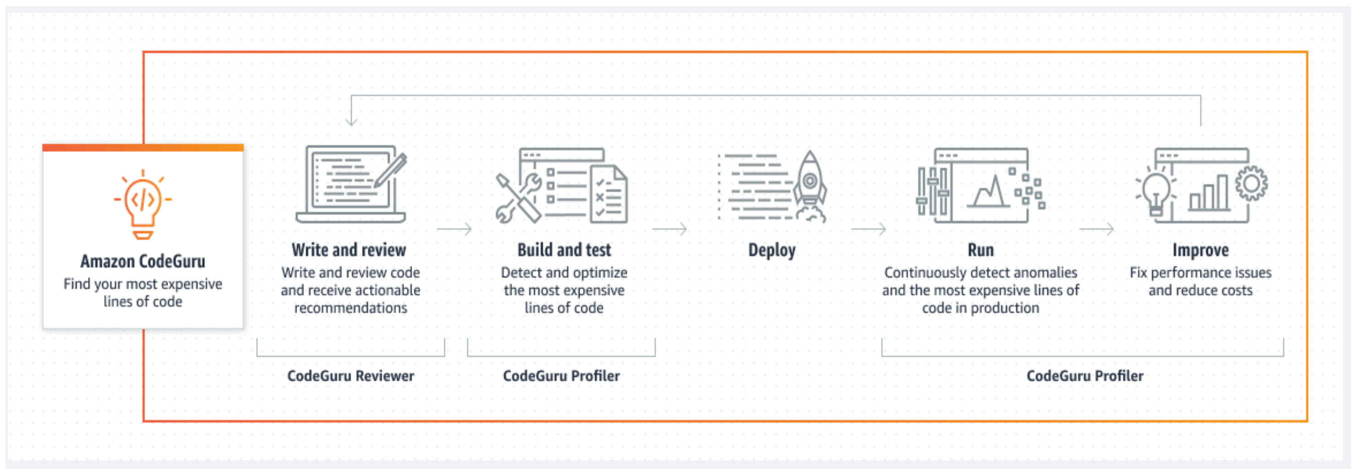

Рисунок 1.2 – Пайплайн роботи Amazon CodeGuru

Сервіс демонструє високу точність для Java-коду та глибоку інтеграцію з екосистемою AWS. Критичним обмеженням є підтримка лише двох мов програмування, відсутність можливості самостійного розгортання та обов'язкова прив'язка до платної хмарної інфраструктури AWS. Аналіз виконується однією ML-моделлю без мультиагентної архітектури і без вибору LLM-провайдера.

1.2.3 SonarQube

SonarQube — провідна платформа безперервного контролю якості коду від компанії SonarSource [9]. Система підтримує понад 30 мов і виконує статичний аналіз за наперед визначеними правилами, виявляючи баги, вразливості безпеки та code smells. Хмарна версія SonarCloud інтегрується з GitHub, Bitbucket і GitLab, забезпечуючи перевірку quality gate у CI/CD-пайплайнах. SonarQube Community Edition розповсюджується під ліцензією LGPL v3 і може розгортатися на власних серверах.

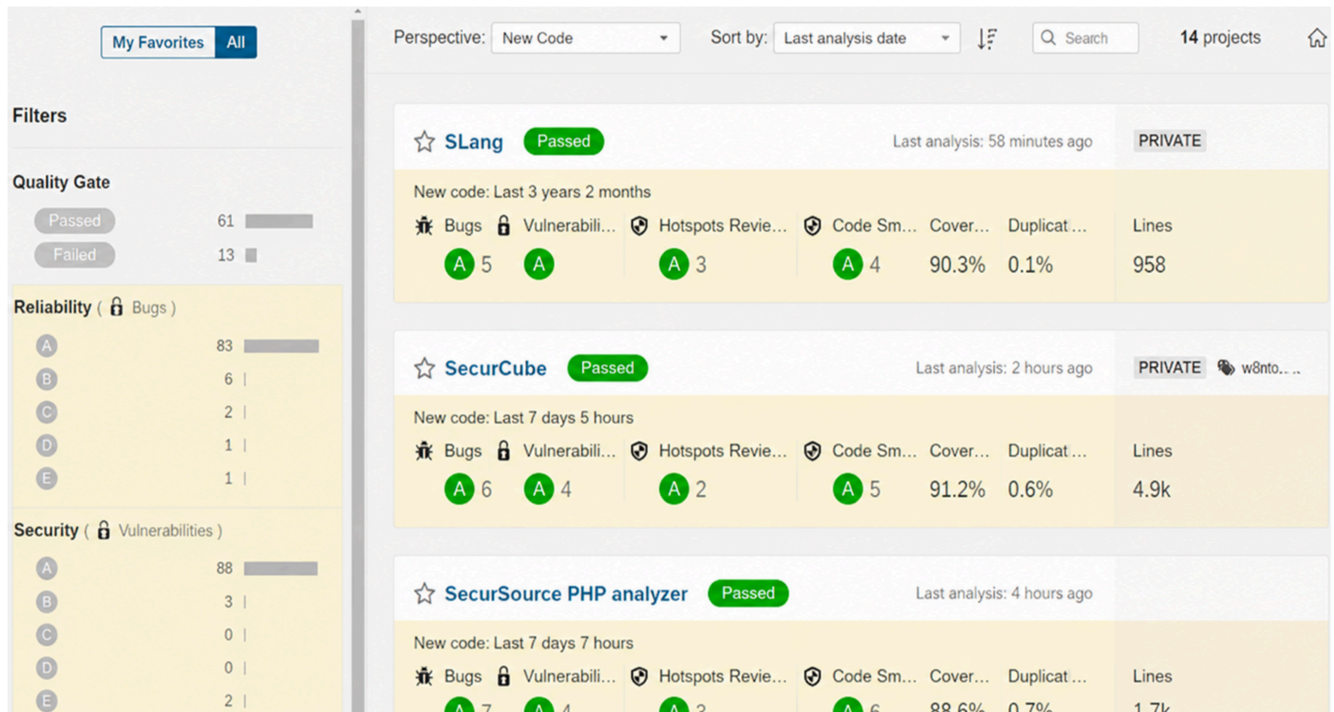

Рисунок 1.3 – Дашборд SonarQube з метриками якості коду

Головними перевагами SonarQube є широка підтримка мов, зрілість екосистеми та можливість самостійного розгортання. Проте система не використовує LLM: аналіз обмежений правилами і не здатний оцінити семантику коду або надати рекомендації природною мовою. Архітектурний аналіз, оцінка відповідності принципам SOLID та аналіз документації у SonarQube фактично відсутні.

1.2.4 Snyk Code

Snyk Code — SAST-інструмент компанії Snyk, орієнтований на виявлення вразливостей безпеки в реальному часі безпосередньо в IDE та CI/CD-пайплайнах [10]. На відміну від традиційних SAST-рішень, Snyk Code застосовує ML-моделі для аналізу потоків даних (taint tracking) і виявляє вразливості класу OWASP Top 10: SQL Injection, XSS, Path Traversal, Command Injection тощо. Платформа підтримує понад 30 мов і інтегрується з VS Code, JetBrains, GitHub та GitLab.

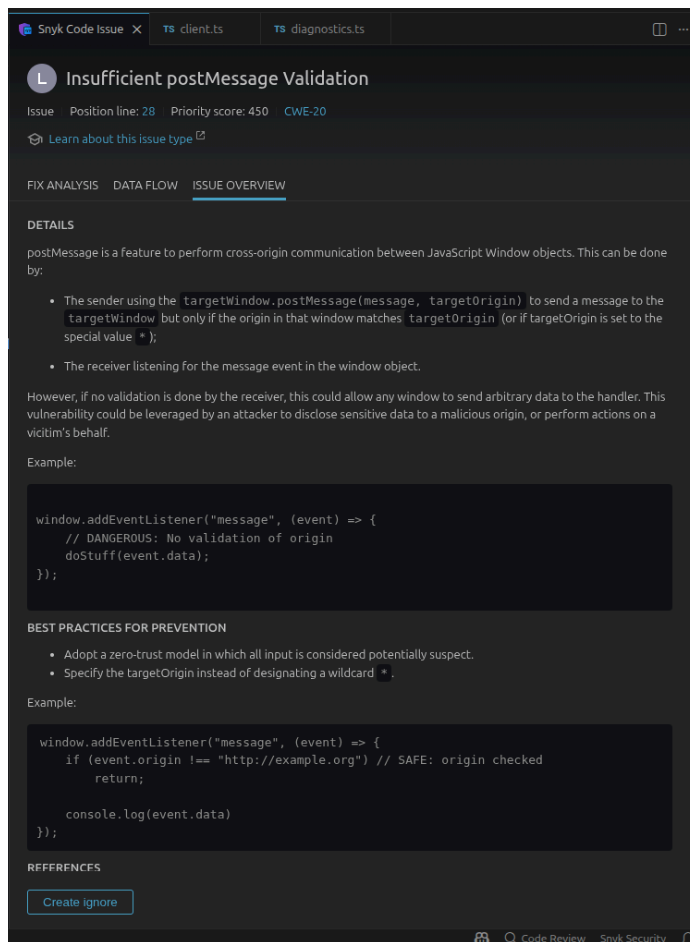

Рисунок 1.4 – Розширення Snyk Code у VS Code

Snyk Code демонструє високу точність виявлення вразливостей безпеки з відносно низьким рівнем хибних спрацювань. Проте інструмент зосереджений виключно на безпеці й не охоплює інших аспектів якості: архітектурних проблем, відповідності принципам SOLID, тестового покриття, документації та логічних дефектів. Це робить Snyk Code вузькоспеціалізованим доповненням, а не повноцінним рішенням для комплексного рев’ю коду.

1.3 Порівняльний аналіз рішень та переваги розробленої системи

На основі проведеного огляду виконано порівняльний аналіз розглянутих інструментів за ключовими критеріями вибору рішення для автоматизованого рев’ю коду (таблиця 1.1). Критерії сформовано відповідно до вимог, що є визначальними для практичного використання в процесі розробки програмного забезпечення.

Таблиця 1.1 – Порівняльний аналіз інструментів автоматизованого рев'ю коду

Критерій	**GitHub Copilot**	**SonarQube**	**Amazon CodeGuru Reviewer**	**Snyk Code**	**Розроблена система**
Підхід до аналізу	LLM (1 агент)	Статичний (правила)	SAST + ML (Automated Reasoning)	SAST + ML	Мультиагентний LLM
К-сть спеціал. агентів	1	—	—	—	7 паралельних
Підтрим. мов програмування	20+	30+	2 (Java, Python)	30+	27
LLM-провайдери	OpenAI	—	Власні ML-моделі AWS	—	OpenAI, Anthropic, Gemini
Інтеграція з VS Code	Так	Ні	Ні (інтегрується з репозиторіями/PR)	Так	Так
REST API	Ні	Так	Так (через AWS API)	Так	Так
Самостійне розгортання	Ні	Так	Ні	Ні	Так
Відкритий код	Ні	Частково	Ні	Ні	Так

Як видно з таблиці 1.1, жодне з розглянутих рішень не реалізує мультиагентного підходу до аналізу коду. GitHub Copilot використовує єдиний LLM-агент, що обмежує глибину спеціалізованого аналізу. SonarQube та Snyk Code ґрунтуються на правилах і ML-моделях без семантичного розуміння коду. Amazon CodeGuru підтримує лише дві мови програмування. Жоден з інструментів не надає одночасно мультиагентної архітектури, можливості вибору LLM-провайдера, REST API та повноцінної інтеграції з VS Code у відкритому форматі.

Розроблена система забезпечує такі ключові конкурентні переваги:

– мультиагентна паралельна архітектура з сімома спеціалізованими агентами (статичний аналіз, безпека, стиль, логіка, архітектура SOLID, тестове покриття, документація), що усуває взаємний конфлікт промптів і забезпечує глибоку перевірку кожного аспекту;

– підтримка трьох LLM-провайдерів (OpenAI, Anthropic, Google Gemini) з вибором конкретної моделі та рівня глибини міркувань (reasoning effort) на рівні окремого запиту;

– підтримка 27 мов програмування з диференційованим набором агентів: повний статичний аналіз для Python, semgrep-сканування для основних мов, LLM-аналіз для решти;

– відкритий асинхронний REST API на FastAPI з персистентністю результатів у PostgreSQL, придатний для інтеграції у CI/CD-пайплайни;

– розширення для VS Code з інтерактивним звітом, вбудованими діагностиками у панелі Problems та SVG-індикатором загальної якості коду;

Перелічені переваги визначають розроблену систему як принципово новий клас інструментів для рев’ю коду, що поєднує глибину спеціалізованого мультиагентного аналізу з гнучкістю вибору LLM-провайдера та зручністю інтеграції у процес розробки.

1.4 Висновки до першого розділу

У першому розділі проведено системний аналіз предметної галузі автоматизованого рев’ю та аналізу програмного коду. Встановлено, що еволюція підходів проходить шлях від правило-орієнтованого статичного аналізу через одноагентні LLM-системи до мультиагентних архітектур, здатних паралельно аналізувати різні виміри якості. Фреймворк LangGraph визначено як оптимальний засіб оркестрації таких систем завдяки механізму графа стану та підтримці паралельного виконання агентів.

Порівняльний аналіз чотирьох провідних конкурентних рішень — GitHub Copilot, Amazon CodeGuru Reviewer, SonarQube та Snyk Code — засвідчив, що жодне з них не реалізує повноцінного мультиагентного підходу у поєднанні з гнучкістю вибору LLM-провайдера, відкритим REST API та можливістю самостійного розгортання. Це підтверджує актуальність і обґрунтованість розробки нової системи та слугує основою для формулювання вимог і архітектурних рішень, що розглядаються у наступних розділах.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ

У даному розділі описано архітектурні та проектні рішення, прийняті під час розробки мультиагентної системи для рев'ю та аналізу коду. Розглянуто вибір технологічного стеку, загальну архітектуру системи, побудову мультиагентного конвеєра на основі фреймворку LangGraph, UML-діаграми, хмарне розгортання серверних компонентів на платформі Microsoft Azure, а також практичні аспекти реалізації ключових компонентів і розширення для VS Code.

2.1 Вибір технологій та засобів розробки

Під час вибору технологічного стеку враховувались вимоги до асинхронної обробки, підтримки мультиагентної оркестрації, масштабованості та зручності інтеграції з IDE-середовищами і хмарними сервісами. За результатами аналізу сформовано такий стек:

Python 3.11 обрано як основну мову серверної частини завдяки розвиненій підтримці асинхронного програмування (asyncio, async/await), типізації (typing, TypedDict, Annotated) та широкій екосистемі бібліотек для роботи з великими мовними моделями. Жорстка типізація через TypedDict та Pydantic дозволяє виявляти помилки на етапі розробки і спрощує перевірку коректності стану між агентами.

FastAPI 0.115 [13] — сучасний асинхронний HTTP-фреймворк з автоматичною генерацією OpenAPI-документації та вбудованою валідацією запитів і відповідей через Pydantic v2. Підтримка BackgroundTasks дозволяє повертати відповідь 202 Accepted клієнту негайно, а виконання тривалого мультиагентного конвеєра запускати асинхронно у фоновому режимі.

LangGraph 0.2 [7] надає декларативний API побудови агентних графів на основі StateGraph: вузли (nodes) визначають обчислення, ребра (edges) задають потік виконання, а механізм Send() реалізує паралельний розподіл стану між агентами без копіювання даних. LangChain 0.3 забезпечує єдині абстракції

BaseChatModel і with_structured_output() для трьох LLM-провайдерів незалежно від їх API-специфіки.

SQLAlchemy 2.0 (async) разом з asyncpg забезпечує асинхронну персистентність завдань і звітів. Як СУБД обрано Azure Database for PostgreSQL Flexible Server [14] — керований хмарний сервіс Microsoft Azure, що надає повну сумісність з PostgreSQL 16 зі вбудованим резервним копіюванням, реплікацією та автоматичним масштабуванням ресурсів. Flexible Server підтримує підключення через VNET-інтеграцію, що забезпечує мережеву ізоляцію між базою даних і серверним контейнером.

Серверна частина розгортається на Azure Container Apps [18] — безсерверній платформі оркестрації контейнерів Microsoft Azure на базі Kubernetes. Container Apps забезпечує автоматичне горизонтальне масштабування (scale-to-zero при відсутності навантаження), вбудовані механізми балансування трафіку та HTTPS-термінацію, а також управління секретами (API-ключі LLM-провайдерів та рядок підключення до БД).

Розширення для VS Code розроблено на TypeScript [16] з використанням офіційного VS Code Extension API [15]. Компоненти extension.ts, client.ts, panel.ts та diagnostics.ts реалізовані як самодостатні модулі з чіткими інтерфейсами, що спрощує тестування та підтримку.

2.2 Загальна архітектура системи

Система побудована за трирівневою клієнт-серверною архітектурою: рівень інтерфейсу користувача (розширення VS Code), рівень серверної логіки (FastAPI-бекенд на Azure Container Apps) та рівень ШІ-обробки (LangGraph-конвеєр). Загальна архітектура системи представлена на рисунку 2.1.

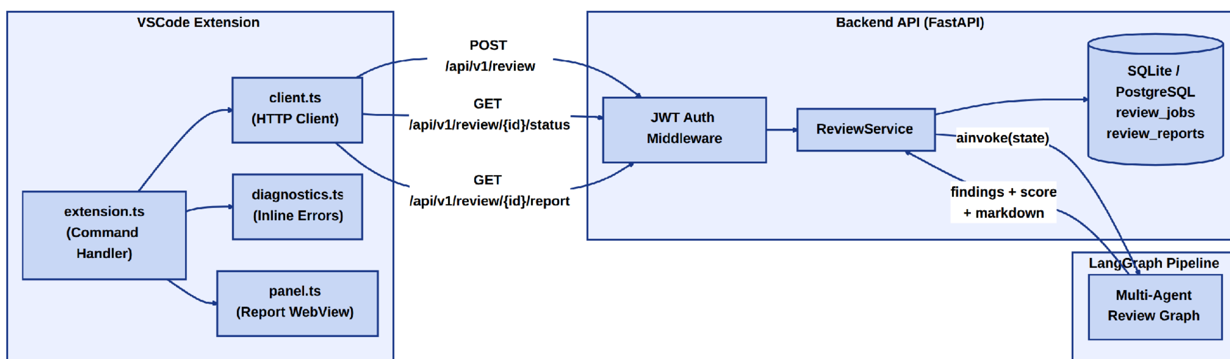

Рисунок 2.1 – Загальна архітектура мультиагентної системи

Розширення VS Code складається з чотирьох модулів. Модуль extension.ts реєструє команду codeReview.review та координує взаємодію між іншими компонентами. Модуль client.ts виконує HTTP-запити до бекенду і реалізує цикл опитування статусу завдання кожні дві секунди. Модуль panel.ts відображає згенерований Markdown-звіт у WebView-панелі. Модуль diagnostics.ts перетворює знахідки агентів на об'єкти VS Code Diagnostic та відображає їх inline у панелі Problems.

FastAPI-бекенд функціонує у контейнері на Azure Container Apps і приймає запити на рев’ю через REST API. Завдання зберігаються у Azure Database for PostgreSQL Flexible Server зі статусами queued/running/done/error; звіти з результатами аналізу також зберігаються у базі даних. Мережева взаємодія між Container Apps та Flexible Server організована через VNET-інтеграцію в межах одного регіону Azure, що виключає передачу трафіку через публічний інтернет та мінімізує затримки. ReviewService інкапсулює всю бізнес-логіку роботи з базою даних та конвеєром.

LangGraph-конвеєр отримує стан ReviewState від ReviewService і виконує послідовність: RouterNode виявляє мову програмування та запускає статичні аналізатори, після чого через Send() паралельно запускає сім спеціалізованих агентів. SummariserAgent агрегує результати у структурований Markdown-звіт та числову оцінку якості коду.

2.3 Діаграма варіантів використання

Діаграма варіантів використання (рисунок 2.2) визначає акторів системи та їх взаємодію з ключовими функціями. Виділено два типи акторів: Розробник – основний користувач, який взаємодіє з системою через розширення VS Code – та Адміністратор системи, відповідальний за налаштування параметрів розгортання та конфігурації хмарної інфраструктури.

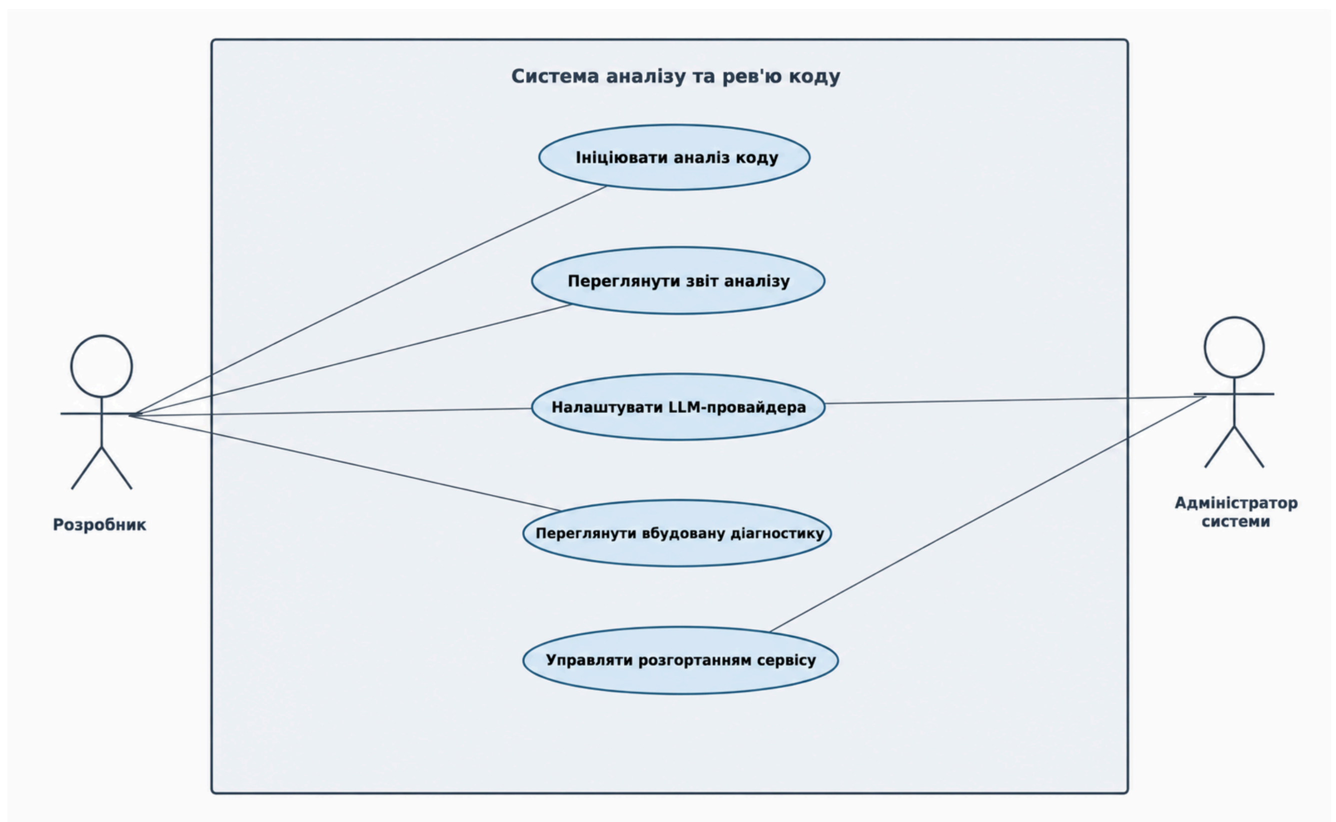

Рисунок 2.2 – Діаграма варіантів використання системи

Розробник взаємодіє з чотирма варіантами використання: ініціювання аналізу коду через команду розширення VS Code, перегляд структурованого звіту рев'ю у WebView-панелі, перегляд вбудованої діагностики в IDE та налаштування LLM-провайдера і параметрів аналізу. Адміністратор системи відповідає за управління розгортанням сервісу на Azure Container Apps та спільно з розробником може налаштовувати LLM-провайдера. Визначені варіанти використання охоплюють повний цикл взаємодії з системою: від ініціювання аналізу до отримання результатів.

2.4 Проектування мультиагентного конвеєра LangGraph

Конвеєр реалізовано як StateGraph — типізований ациклічний граф зі спільним станом ReviewState. Архітектура конвеєра наведена на рисунку 2.3.

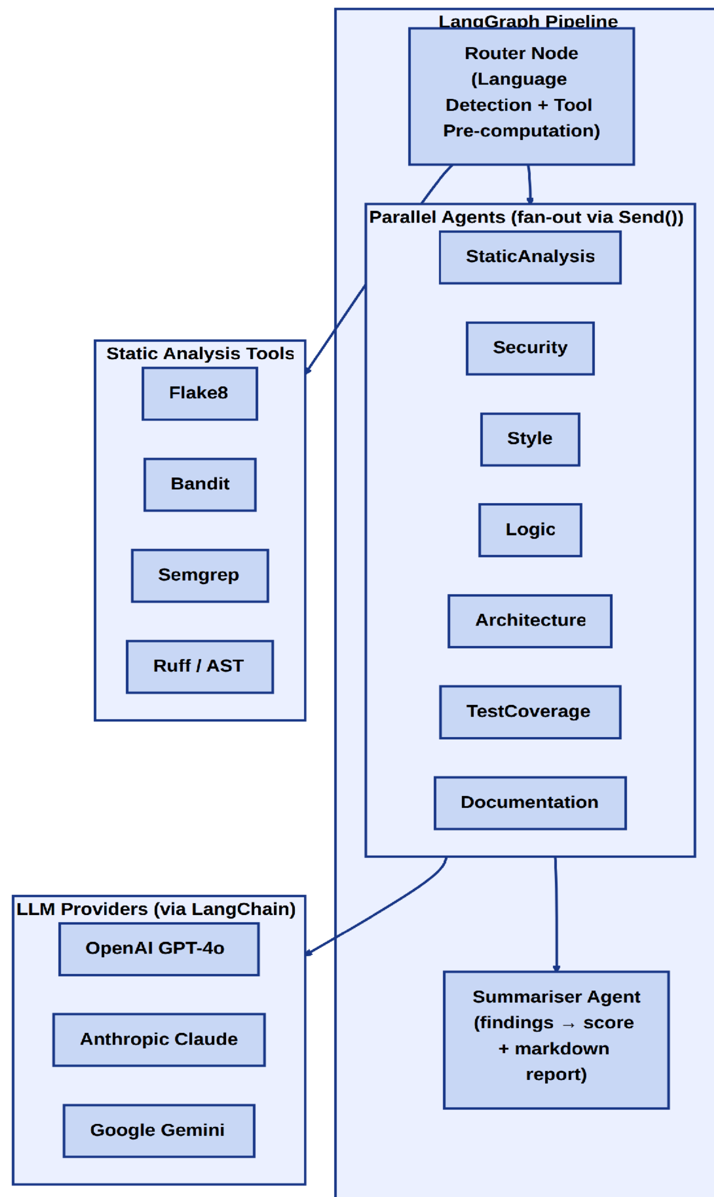

Рисунок 2.3 – Архітектура мультиагентного конвеєра LangGraph

Вершина START з'єднана умовним ребром з вузлом router. Функція router_edges() повертає список об'єктів Send(), кожен з яких спрямовує копію поточного стану до конкретного агента — без очікування виконання попереднього. LangGraph виконує всі агенти паралельно, накопичуючи їх часткові оновлення через функції-редуктори. Усі сім агентних вузлів мають спільне ребро

до summariser_agent, який запускається лише після завершення всіх паралельних гілок, а summariser_agent з'єднано з END.

2.4.1 Граф стану та акумулятори

Стан графу описано класом ReviewState, похідним від TypedDict. Поля findings та agent_statuses позначено анотацією Annotated з функціями-редукторами: findings використовує operator.add для конкатенації списків знахідок від кожного агента, agent_statuses — функцію _merge_dicts для злиття словників статусів. Завдяки цьому механізму LangGraph автоматично об'єднує часткові оновлення з усіх паралельних гілок без явної синхронізації.

Знахідки агентів описуються TypedDict Finding з полями: agent (ім'я агента-автора), severity (critical/high/medium/low/info), category (security/logic/style/static/test/docs/architecture), line_start та line_end (номери рядків), message та suggestion. Поля line_start і line_end є опціональними, що дозволяє агентам повідомляти про загальні архітектурні спостереження без прив'язки до конкретного рядка.

Лістинг 2.1 – Визначення ReviewState та Finding

```
class Finding(TypedDict):
    agent: str
    severity: str   # critical | high | medium | low | info
    category: str   # security | style | logic | static | test |
docs | architecture
    line_start: Optional[int]
    message: str
    suggestion: Optional[str]

class ReviewState(TypedDict):
    code: str
    findings: Annotated[list[Finding], operator.add]
    agent_statuses: Annotated[dict, _merge_dicts]
```

2.4.2 Вузол маршрутизатора та розгалуження Send()

RouterNode виконує дві ролі: оновлення стану (асинхронна функція router_node) та визначення умовних переходів (синхронна функція router_edges). Метод detect_language() зіставляє розширення файлу із словником _EXT_TO_LANGUAGE, що охоплює 27 мов програмування.

Метод get_active_agents() фільтрує список агентів відповідно до прапорців конфігурації (skip_security, skip_tests, skip_style, skip_docs, skip_architecture) та визначеної мови програмування: static_analysis_agent і style_agent активні лише для Python, де доступні інструменти Flake8 та Ruff. Для решти мов статичний аналіз виконує Semgrep (якщо увімкнено) або LLM-агенти самостійно.

Асинхронна функція update_state() запускає всі статичні аналізатори паралельно через asyncio.gather(): для Python — Flake8, Bandit, Ruff та AST-парсер; для решти мов — Semgrep з відповідним набором правил. Результати зберігаються в полі tool_results стану і передаються кожному агенту разом з кодом та метаданими, що виключає повторний запуск аналізаторів у паралельних гілках.

2.5 Діаграма станів завдання на рев’ю

Кожне завдання на рев'ю проходить через чотири стани: queued (створено, очікує виконання), running (конвеєр запущено), done (завершено успішно) та error (виникла помилка або вичерпано ліміт часу). Повна діаграма станів наведена на рисунку 2.4.

Початковий стан queued встановлюється одразу у запит до API — завдання зберігається у базі даних і передається у фонову задачу FastAPI. Перехід до стану running відбувається в момент запуску конвеєра LangGraph: система послідовно виконує визначислення результатів статичних інструментів (Flake8, Bandit, Ruff, Semgrep), а потім паралельно запускає сім спеціалізованих агентів через механізм Send(). Після того як усі агенти завершили роботу, агент об'єднує знайдені зауваження, формує підсумковий звіт та числову оцінку якості коду.

Рисунок 2.4 – Діаграма станів завдання на рев’ю коду

Після створення запис ReviewJob зберігається у Azure Database for PostgreSQL Flexible Server зі статусом queued. Фоновий обробник BackgroundTasks негайно починає виконання, встановлює статус running та прогрес 10 %. Під час виконання конвеєра LangGraph послідовно відбувається: виявлення мови, паралельний запуск статичних аналізаторів, паралельне виконання агентів через Send(), агрегування результатів SummariserAgent. Після успішного завершення ReviewJob оновлюється до статусу done з прогресом 100 %

і оцінкою якості, а ReviewReport зберігається у базі даних. У разі виключення або перевищення часового ліміту asyncio.wait_for() статус змінюється на error із збереженням тексту помилки в полі error_message.

2.6 UML-діаграма класів системи

UML-діаграма класів системи представлена на рисунку 2.5 та відображає ієрархію агентів, структури даних стану та компоненти персистентності

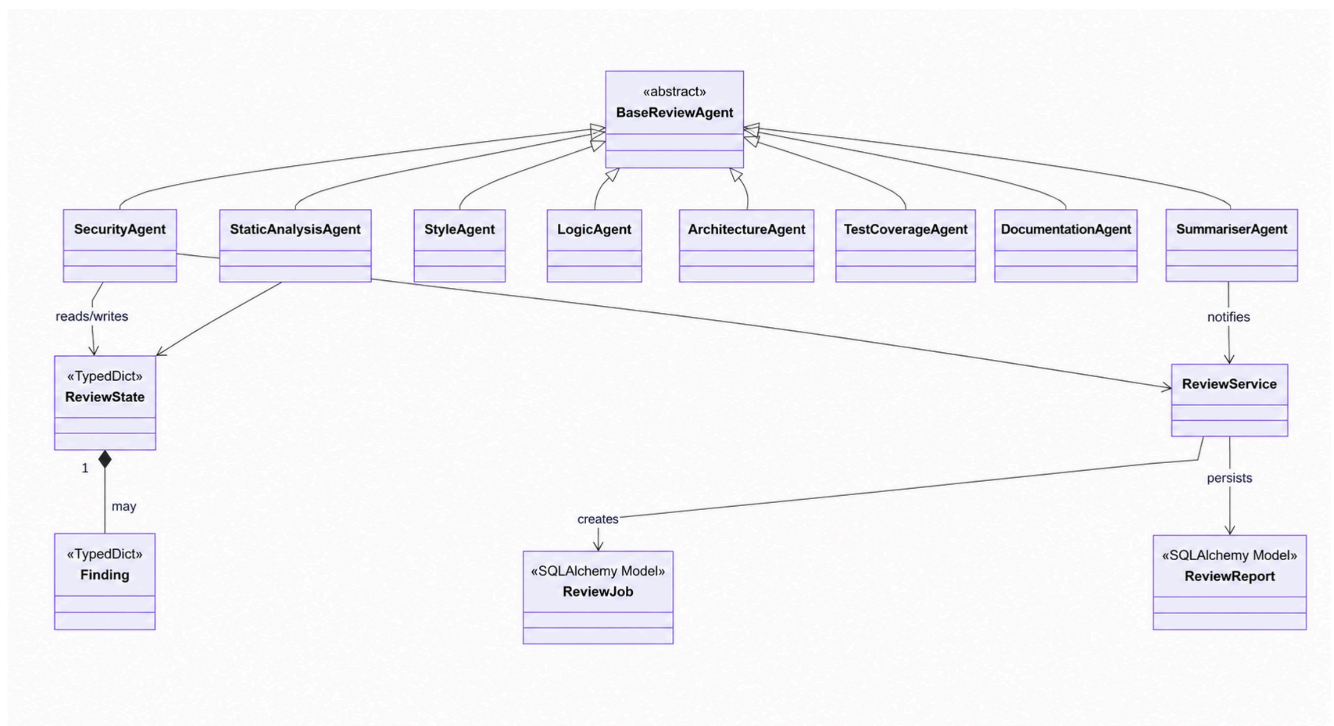

Рисунок 2.5 – UML-діаграма класів мультиагентної системи

Центральне місце займає абстрактний клас BaseReviewAgent, що визначає контракт для всіх спеціалізованих агентів. Клас оголошує два абстрактні методи: _build_system_prompt() — повертає системний промпт ролі агента, та _build_user_prompt() — формує промпт з кодом і результатами статичних аналізаторів. Конкретний метод run() реалізує повний цикл виклику LLM: формує промпти, виконує async-виклик моделі зі структурованим виводом через FindingList та повертає часткове оновлення стану ReviewState.

Від BaseReviewAgent успадковуються сім спеціалізованих агентів: SecurityAgent (вразливості OWASP Top 10, верифікація виводу Bandit/Semgrep), StaticAnalysisAgent (аналіз виводу Flake8/AST), StyleAgent (відповідність PEP 8,

Ruff, іменування), LogicAgent (логічні помилки, граничні випадки), ArchitectureAgent (принципи SOLID, зв'язність модулів), TestCoverageAgent (покриття тестами, тестованість), DocumentationAgent (docstrings, коментарі, читабельність). Кожен агент встановлює атрибут tier (heavy або light), що визначає вибір моделі за замовчуванням у ProviderFactory.

ProviderFactory відповідає за створення та кешування екземплярів BaseChatModel з урахуванням активного провайдера (OpenAI, Anthropic або Google Gemini), обраної моделі та рівня reasoning_effort. ReviewService оркеструє взаємодію між HTTP-шаром FastAPI і конвеєром LangGraph, створюючи записи ReviewJob і ReviewReport у базі даних через SQLAlchemy async-сесії.

2.7 Діаграма послідовності взаємодії компонентів

Діаграма послідовності на рисунку 2.6 ілюструє повний цикл рев’ю коду від ініціювання розробником у VS Code до отримання результатів.

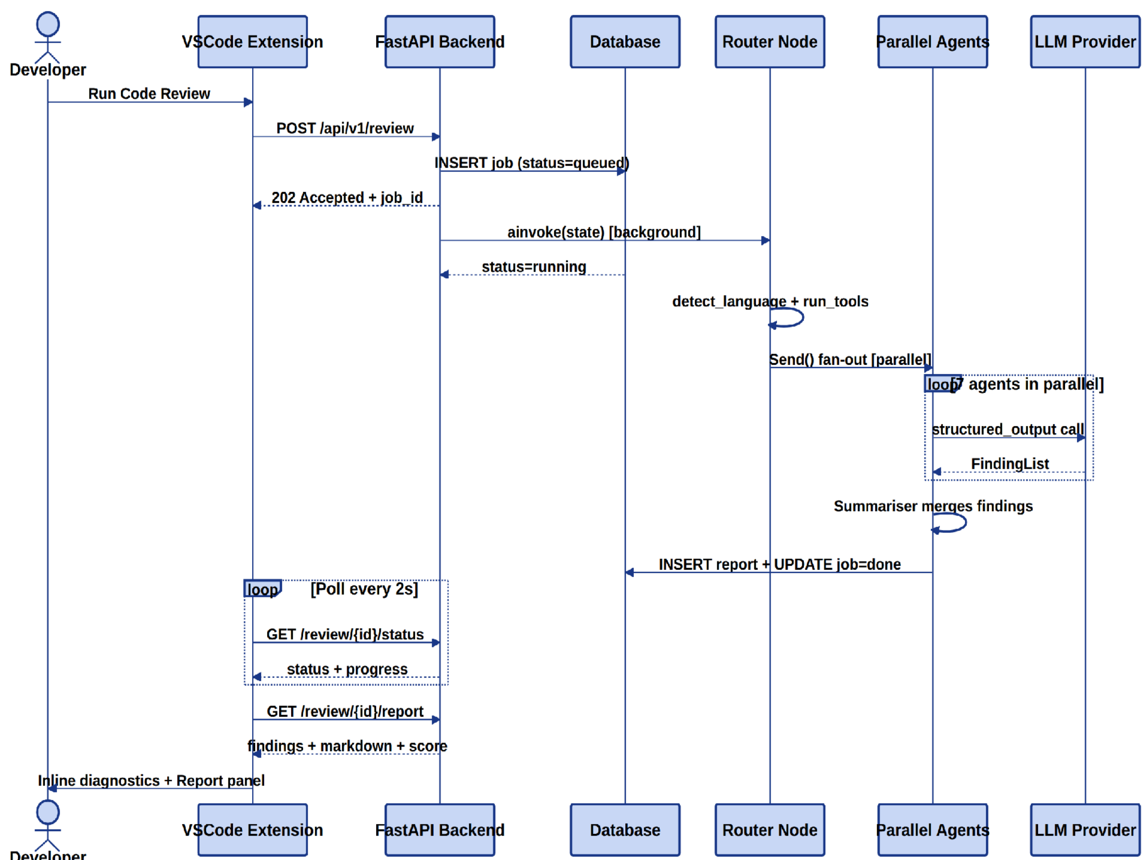

Рисунок 2.6 – Діаграма послідовності виконання рев’ю коду

Розробник активує команду codeReview.review у VS Code. Розширення надсилає POST-запит до /api/v1/review з тілом, що містить код, мову, назву файлу та параметри конфігурації. Бекенд, що виконується на Azure Container Apps, негайно повертає відповідь 202 Accepted з ідентифікатором завдання job_id і запускає мультиагентний конвеєр як фонове завдання FastAPI.

Поки конвеєр виконується, розширення опитує статус завдання через GET /api/v1/review/{id}/status кожні дві секунди. RouterNode виявляє мову та паралельно запускає статичні аналізатори. Після завершення router_edges() LangGraph одночасно запускає від п'яти до семи спеціалізованих агентів через механізм Send(). Кожен агент виконує structured output call до LLM-провайдера і повертає список знахідок. SummariserAgent отримує агреговані знахідки, видаляє дублікати, обчислює оцінку якості та генерує Markdown-звіт. Після запису ReviewReport до Azure PostgreSQL статус завдання змінюється на done.

Розширення виявляє статус done і виконує GET /api/v1/review/{id}/report. Відповідь містить список знахідок, summary, review_score та report_markdown. Модуль diagnostics.ts перетворює знахідки на об'єкти VS Code Diagnostic з прив'язкою до рядків файлу та відображає їх у панелі Problems. Модуль panel.ts рендерить Markdown-звіт у WebView-панелі поруч з редактором.

2.8 Побудова схеми бази даних

Для персистентності завдань на рев’ю та їх результатів застосовано реляційну базу даних Azure Database for PostgreSQL Flexible Server. Схема бази даних складається з двох таблиць, зв’язаних між собою у відношенні один до одного. ER-діаграма схеми наведена на рисунку 2.7.

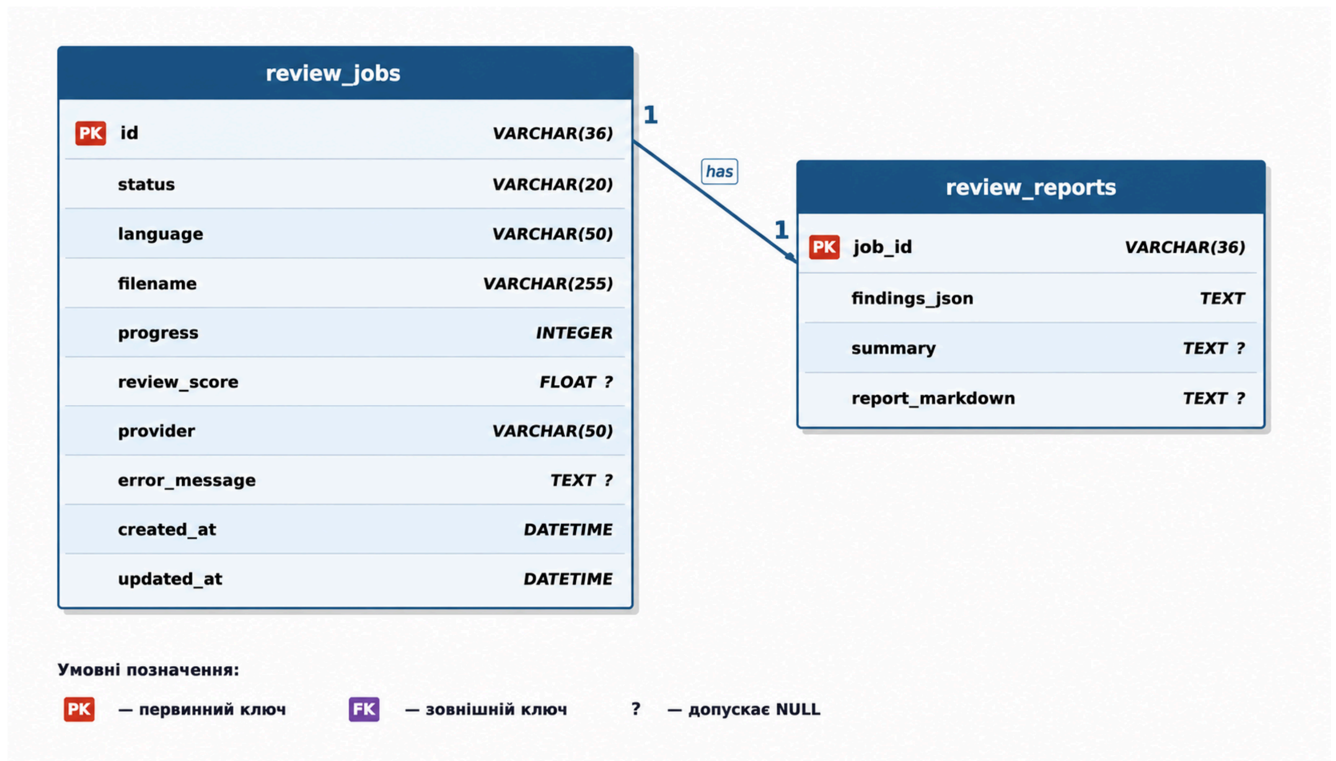

Рисунок 2.7 – ER-діаграма схеми бази даних системи

Таблиця review_jobs зберігає метаінформацію про кожне завдання на рев’ю. Первинним ключем є поле id типу VARCHAR(36), що містить унікальний ідентифікатор у форматі UUID v4, який генерується автоматично на рівні ORM. Поле status (VARCHAR(20)) відображає поточний стан завдання: queued, running, done або error. Поля language (VARCHAR(50)) та filename (VARCHAR(255)) зберігають мову програмування, визначену RouterNode, та ім’я аналізованого файлу відповідно. Числове поле progress (INTEGER) містить відсоток виконання від 0 до 100 і оновлюється під час виконання конвеєра для відображення прогресу у клієнті.

Поле review_score (FLOAT, nullable) заповнюється числовою оцінкою якості коду від 0 до 10 лише після успішного завершення SummariserAgent. Поле provider (VARCHAR(50)) зберігає ідентифікатор активного LLM-провайдера (openai, anthropic або gemini). Nullable-поле error_message (TEXT) містить текст виключення при статусі error. Поля created_at та updated_at (DATETIME) фіксують час створення запису та останнього оновлення; updated_at оновлюється автоматично через параметр onupdate SQLAlchemy.

Таблиця review_reports зберігає деталізовані результати рев’ю після завершення аналізу. Первинним ключем і одночасно зовнішнім ключем до review_jobs.id є поле job_id (VARCHAR(36)), що реалізує відношення один до одного між таблицями. Поле findings_json (TEXT) містить JSON-серіалізований масив знахідок усіх агентів. Nullable-поля summary та report_markdown (TEXT) зберігають текстовий підсумок і повний Markdown-звіт відповідно, що генерується SummariserAgent.

Розподіл даних між двома таблицями обумовлений різними частотами та обсягами звернень. Метадані завдання запитуються клієнтом кожні дві секунди під час опитування статусу, тоді як великий звіт у report_markdown завантажується одноразово лише після завершення аналізу. Такий розподіл зменшує обсяг переданих даних під час циклічного опитування та знижує навантаження на базу даних. ORM-моделі реалізовано з використанням SQLAlchemy 2.0 Mapped Column API: поля зі значенням Optional[T] відображаються у стовпці з nullable=True, що гарантує цілісність даних на рівні схеми PostgreSQL.

2.9 Реалізація основних класів та методів

Цей підрозділ розкриває практичні деталі реалізації ключових компонентів системи: базового класу агентів, побудови графу, фабрики LLM-провайдерів та REST API.

2.9.1 Базовий клас агента BaseReviewAgent

Клас BaseReviewAgent є абстрактним базовим класом (ABC) та визначає єдиний шаблонний метод run(). Шаблон забезпечує, що логіка виклику LLM, обробки помилок і формування часткового оновлення стану є ідентичною для всіх агентів, а підкласи перевизначають лише промпти.

Константа _CALIBRATION_SUFFIX — суфікс системного промпту, який додається до кожного агента — встановлює обов'язкові правила об'єктивності: заборону вигадувати знахідки, обмеження severity critical лише для реальних збоїв або вразливостей безпеки, ліміт п'яти знахідок на категорію. Функція

_context_preamble() динамічно адаптує поведінку агента залежно від розміру коду: для мікросніпетів (до 5 рядків) інструкції суттєво обмежують кількість допустимих знахідок, для великих файлів — дозволяють повний спектр аналізу.

Структурований вивід реалізовано через обгортку make_structured_model(), яка застосовує with_structured_output(FindingList) до моделі. Це гарантує, що відповідь LLM завжди є валідним списком типізованих об'єктів Finding незалежно від провайдера. Підтримка перевизначення моделі та reasoning_effort на рівні окремого запиту реалізована у методі run() через перевірку полів стану model_name та reasoning_effort.

2.9.2 Побудова та компіляція графу LangGraph

Функція build_graph() декларативно описує топологію конвеєра. StateGraph(ReviewState) ініціалізується з типізованим станом, що дозволяє фреймворку перевіряти коректність оновлень від кожного вузла. Після додавання вузлів і ребер граф компілюється методом compile(). Скомпільований граф є синглтоном, ініціалізованим при першому виклику get_compiled_graph(), що виключає накладні витрати на повторну компіляцію між запитами.

Лістинг 2.2 – Побудова та реєстрація вузлів графу LangGraph

```
graph = StateGraph(ReviewState)
graph.add_node("router", router_node)
graph.add_node("security_agent", security_node)
# ... (решта 6 агентних вузлів)
graph.add_node("summariser_agent", summariser_node)
graph.add_edge(START, "router")
graph.add_conditional_edges("router", router_edges)
for agent in AGENT_NAMES:
    graph.add_edge(agent, "summariser_agent")
graph.add_edge("summariser_agent", END)
return graph.compile()
```

Умовні ребра від router_edges дозволяють LangGraph виконати від трьох до семи агентів паралельно залежно від мови і конфігурації запиту. Завдяки reducer-анотаціям у ReviewState фреймворк автоматично об'єднує результати паралельних гілок до виклику summariser_agent.

2.9.3 Фабрика LLM-провайдерів ProviderFactory

ProviderFactory інкапсулює специфіку трьох LLM-провайдерів за єдиним інтерфейсом get_model(). Визначення активного провайдера відбувається у два кроки: спочатку перевіряється явно передане ім'я моделі (функція _infer_provider() зіставляє префікс імені моделі з провайдером: 'claude-' → anthropic, 'gemini-' → gemini, 'gpt-'/'o1'/'o3' → openai), а якщо модель не задана — читається змінна оточення LLM_PROVIDER.
Функція get_model() реалізує кешування екземплярів моделей за ключем 'provider:model_name:reasoning_effort'. Кожна комбінація провайдер+модель+зусилля створюється одноразово та повторно використовується між запитами, що знижує накладні витрати на ініціалізацію SDK.

Особливу увагу приділено несумісностям між провайдерами. OpenAI o-series моделі не підтримують parallel_tool_calls, тому make_structured_model() застосовує method='json_schema' замість стандартного function-calling. Для Anthropic розширене мислення (extended thinking) несумісне з вимушеним tool_choice у with_structured_output(); параметр reasoning_effort для Anthropic задокументований як no-op у налаштуваннях розширення. Для Google Gemini рівень reasoning_effort транслюється у thinking_budget: low → 512, medium → 2048, high → 8192 токенів.

2.9.4 REST API та персистентність

REST API реалізовано через три ендпоінти FastAPI з prefix /api/v1. POST /review приймає ReviewRequest (код, мова, файл, конфігурація), створює ReviewJob зі статусом queued у PostgreSQL та повертає 202 Accepted з job_id. Запуск мультиагентного конвеєра відбувається у BackgroundTasks-обробнику, що використовує окрему SQLAlchemy-сесію для уникнення конфліктів з сесією HTTP-запиту.

GET /review/{id}/status повертає поточний статус, прогрес і повідомлення про помилку (за наявності). GET /review/{id}/report доступний лише при статусі done; при незавершеному завданні повертає 409 Conflict. Відповідь містить

десеріалізований список FindingResponse, summary, review_score та report_markdown.

Підключення до Azure Database for PostgreSQL Flexible Server виконується через asyncpg з пулом з'єднань. Рядок підключення формується із змінних оточення DATABASE_UR. Flexible Server налаштований в режимі Burstable tier для розробки з автоматичним перемиканням на General Purpose при зростанні навантаження. Вбудоване резервне копіювання Azure забезпечує точку відновлення з точністю до 5 хвилин (PITR) без додаткового налаштування.

2.10 Розробка розширення для VS Code

Розширення Code Review & Analysis Tool розповсюджується через VS Code Marketplace [15] та встановлюється стандартним способом через панель Extensions (рисунок 2.8). Назва розширення, опис можливостей та вимоги до середовища відображаються на сторінці Marketplace відповідно до package.json маніфесту.

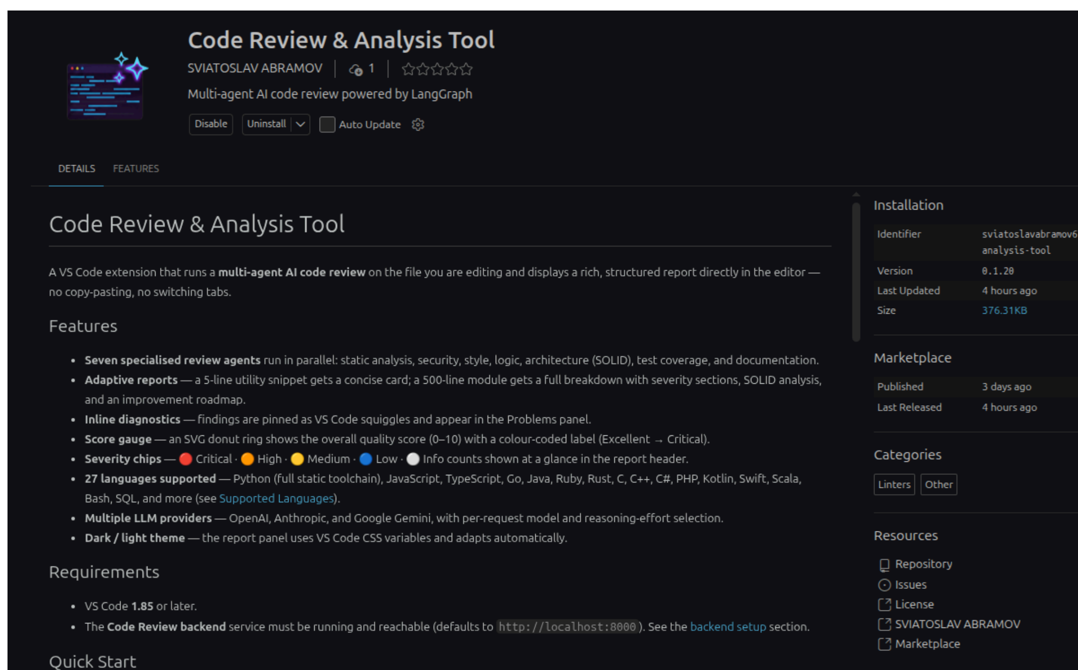

Рисунок 2.8 – Сторінка розширення Code Review & Analysis Tool у VS Code Marketplace

Налаштування розширення доступні через стандартний розділ Settings VS Code (рисунок 2.9). Користувач може обрати конкретну LLM-модель (поле Code

Review: Model) з підтримуваних провайдером, рівень reasoning effort для OpenAI o-series моделей (Code Review: Reasoning Effort), мову звіту (Code Review: Report Language) з понад 20 варіантів, а також вибірково вимкнути конкретних агентів через прапорці Skip Docs, Skip Security, Skip Style.

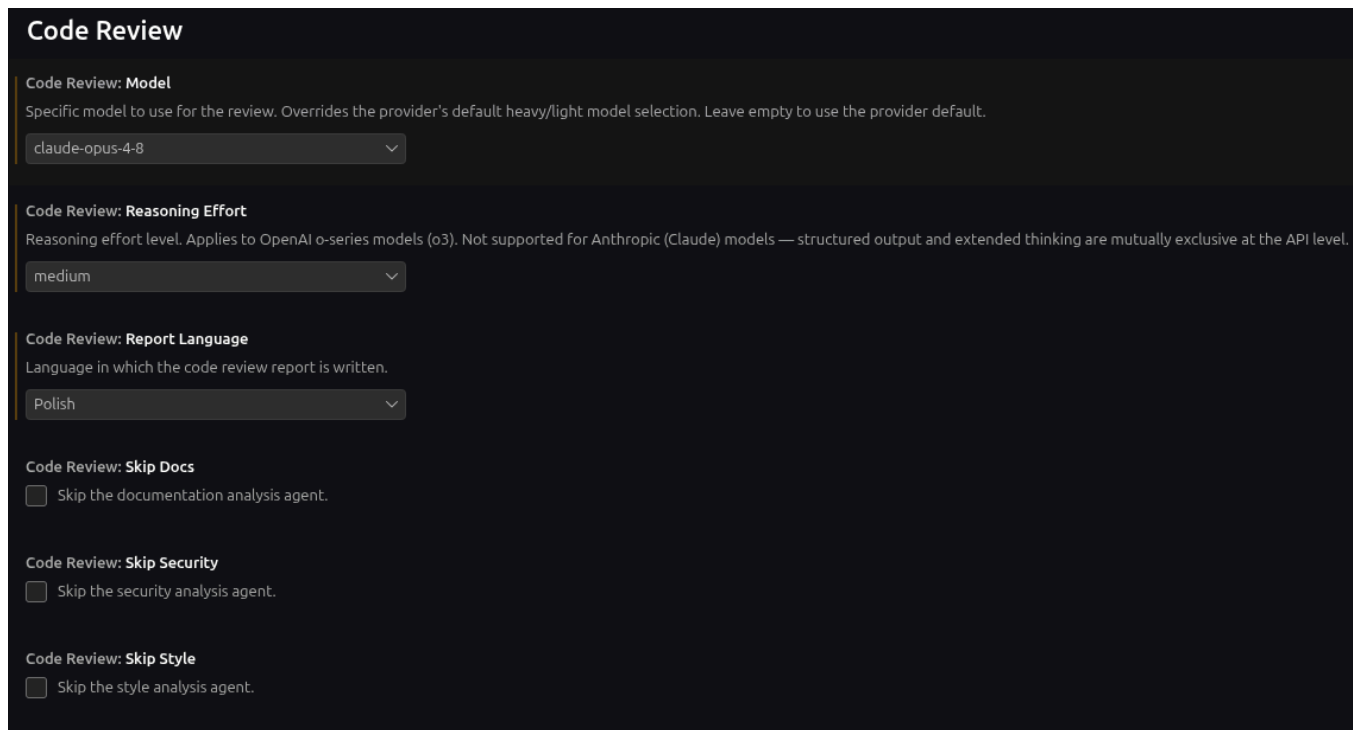

Рисунок 2.9 – Панель налаштувань розширення у VS Code

Після завершення аналізу розширення відображає структурований Markdown-звіт у WebView-панелі поруч з редактором (рисунок 2.10). Звіт містить: заголовок з оцінкою якості (0–10), зведену таблицю категорій знахідок з пріоритетами, деталізовані секції критичних і важливих проблем з описом та пропозицією виправлення, розділ архітектурного аналізу з оцінкою відповідності принципам SOLID та рекомендаціями. Одночасно модуль diagnostics.ts реєструє знахідки як VS Code Diagnostic у панелі Problems з прив'язкою до конкретних рядків файлу, що дозволяє перейти до проблеми одним кліком.

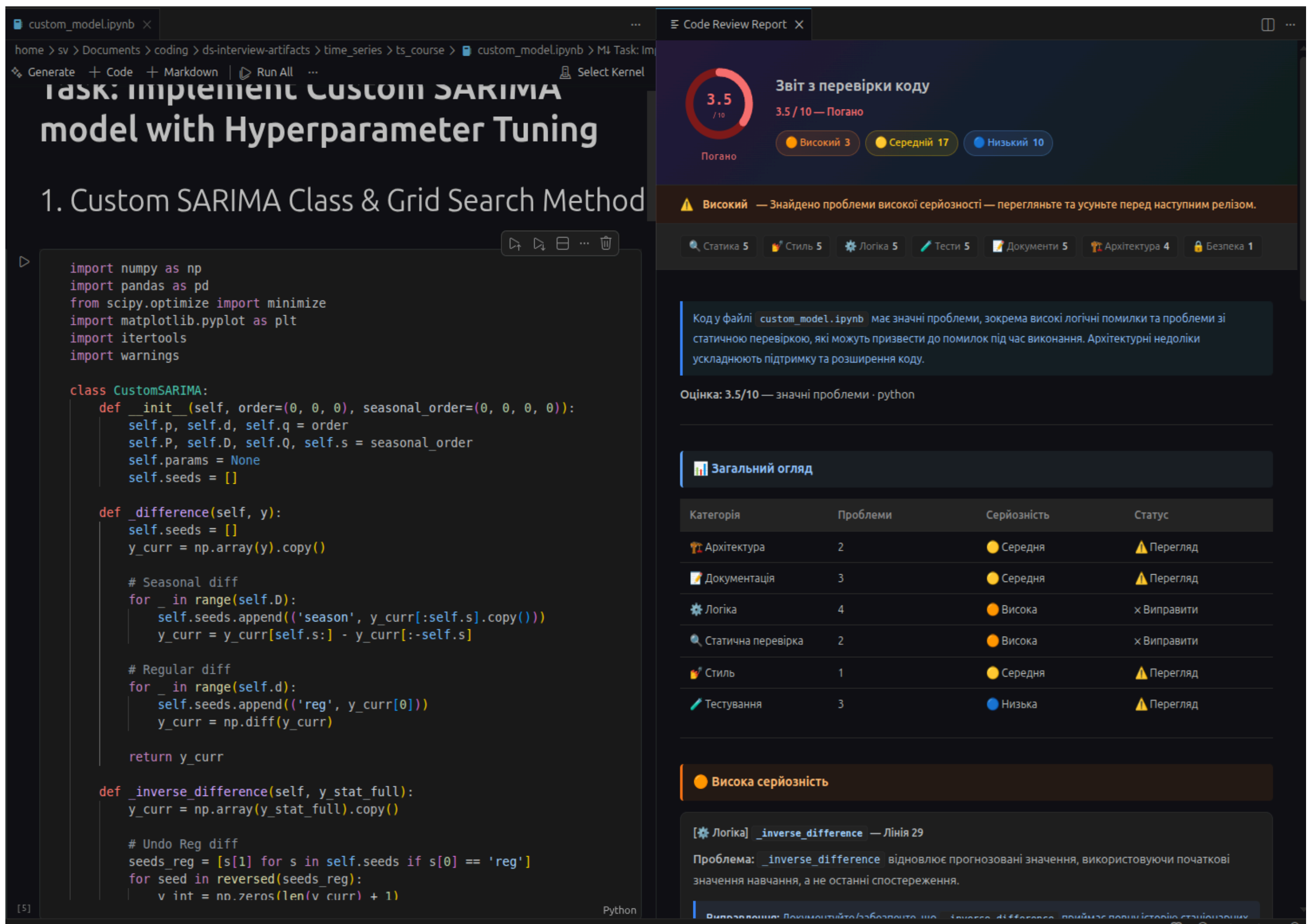

Рисунок 2.10 – Згенерований звіт рев’ю коду у панелі VS Code

2.11 Висновки до другого розділу

У другому розділі детально описано архітектуру та реалізацію мультиагентної системи рев’ю коду. Визначено технологічний стек: Python 3.11, FastAPI, LangGraph 0.2, LangChain 0.3, SQLAlchemy 2.0 + asyncpg, TypeScript та VS Code Extension API. Для хмарного розгортання серверних компонентів обрано платформу Microsoft Azure: FastAPI-бекенд функціонує у безсерверному середовищі Azure Container Apps з автоматичним горизонтальним масштабуванням, а база даних розгорнута на Azure Database for PostgreSQL Flexible Server з VNET-інтеграцією та вбудованим резервним копіюванням.

Спроектований мультиагентний конвеєр реалізує паралельний рев’ю коду сімома спеціалізованими агентами через механізм Send() фреймворку LangGraph. Використання TypedDict ReviewState з анотованими редукторами забезпечує безпечне злиття результатів з паралельних гілок без явної синхронізації.

RouterNode виконує попередню обробку коду статичними аналізаторами одночасно, що виключає дублювання роботи між агентами.

UML-діаграми використання, класів, станів і послідовності підтвердили відповідність реалізації обраним архітектурним рішенням. ProviderFactory забезпечує підтримку трьох LLM-провайдерів через єдиний інтерфейс з обробкою специфічних обмежень кожного API. Розширення VS Code надає повноцінний інтерфейс взаємодії з системою: налаштування параметрів аналізу, відображення структурованого звіту та inline-діагностику безпосередньо у редакторі коду.

3 ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

Для підтвердження коректності та надійності розробленої мультиагентної системи рев’ю коду було розроблено та виконано комплекс тестів трьох рівнів: модульні, інтеграційні та наскрізні. Усі тести реалізовано з використанням фреймворку pytest [19] та бібліотеки pytest-asyncio для перевірки асинхронних компонентів. Загальна структура тестового покриття охоплює ключові компоненти бекенду: маршрутизатор конвеєра, стан графу, фабрику LLM-провайдерів, інструменти статичного аналізу та REST API.

3.1 Загальна стратегія тестування

Стратегія тестування системи базується на пірамідальній моделі якості [20], що передбачає найбільшу кількість дешевих і швидких модульних тестів, менший шар інтеграційних тестів і невелику кількість повільних наскрізних тестів. Таке розподілення дозволяє виявляти дефекти на найранішому можливому рівні та скорочує час зворотного зв'язку під час розробки.

Для ізоляції компонентів у модульних тестах застосовується бібліотека unittest.mock: зовнішні залежності (LLM API, файлова система, subprocess) замінюються підмінними об'єктами (MagicMock, patch). Інтеграційні тести FastAPI використовують in-memory базу даних SQLite через ASGITransport httpx [21], що усуває залежність від реального PostgreSQL і дозволяє запускати тести у CI-середовищі. Тести, що потребують живих API-ключів LLM-провайдерів, позначені маркером @pytest.mark.integration і виключаються з регулярного запуску. Наскрізні тести позначені маркером @pytest.mark.e2e і виконуються проти реально розгорнутого бекенду на Azure Container Apps.

Тестові дані організовано у каталозі tests/fixtures/ і містять чотири фікстурні Python-файли та два JavaScript-файли: good_code.py (коректний, добре задокументований код), bad_security.py (SQL-ін'єкція, hardcoded credentials), bad_style.py (порушення PEP 8, довгі рядки, погане іменування), bad_logic.py

(помилка off-by-one, некоректна логіка індексів), good_code.js та bad_code.js для тестування багатомовності. Спільні фікстури (fixtures) визначено у файлі conftest.py і доступні усім тестам автоматично.

3.2 Модульне тестування

Модульні тести охоплюють чотири ізольовані підсистеми: маршрутизатор конвеєра, типи даних стану графу, фабрику LLM-провайдерів та інструменти статичного аналізу. Тести виконуються без звернення до реальних LLM-провайдерів або бази даних.

3.2.1 Тестування маршрутизатора (RouterNode)

Клас TestLanguageDetection перевіряє функцію detect_language() для восьми сценаріїв: файли з розширеннями .py, .js, .ts, .go, .java, .tsx, а також визначення мови за shebang-рядком (#!python) і повернення 'python' за замовчуванням для невідомого файлу. Клас TestActiveAgents перевіряє метод get_active_agents(): усі агенти активні за замовчуванням для Python, прапорець skip_security виключає security_agent, прапорець skip_tests виключає test_coverage_agent, а для мови JavaScript автоматично виключаються static_analysis_agent та style_agent. Клас TestRouteFunction перевіряє, що get_sends() повертає список об'єктів Send(), а router_node() оновлює поле language у стані.

3.2.2 Тестування стану графу та дедуплікації

Тести TestFinding перевіряють коректне створення TypedDict Finding з обов'язковими полями та допустимість значень None для опціональних полів line_start, line_end і suggestion. Тести TestReviewState підтверджують початковий стан структури та коректне об'єднання знахідок двох агентів через operator.add: два списки по одному елементу об'єднуються у список із двох елементів зі збереженням порядку і значень полів. Клас TestFindingDeduplication перевіряє логіку методу _deduplicate() класу SummariserAgent: знахідки з однаковою парою

(line_start, category) зберігають лише одну — з найвищим severity, а знахідки з однаковим рядком, але різними категоріями (security і style) зберігаються обидві.

3.2.3 Тестування фабрики LLM-провайдерів

Клас TestProviderFactory перевіряє 10 сценаріїв ProviderFactory: вибір провайдера зі змінної оточення LLM_PROVIDER (openai, anthropic, gemini), значення openai за замовчуванням при відсутній змінній, генерацію ProviderConfigurationError при невідомому провайдері та при некоректному tier ('ultra'), а також генерацію відповідних винятків при відсутності ключів OPENAI_API_KEY, ANTHROPIC_API_KEY і GEMINI_API_KEY. Клас TestInferProvider перевіряє функцію _infer_provider() для 12 модельних ідентифікаторів: claude- → anthropic, gemini- → gemini, gpt-/o1/o3/o4 → openai, а також повернення None для невідомого рядка. Окремо перевіряється, що get_model() при явно заданому model_name='claude-sonnet-4-6' ігнорує LLM_PROVIDER=openai і коректно викликає _build_anthropic().

3.2.4 Тестування інструментів статичного аналізу

Тести перевіряють чотири аналізатори на фікстурних файлах. Flake8Runner: bad_style.py повертає непустий список порушень, мінімальний коректний код ('x = 1') — порожній список. BanditRunner: bad_security.py містить знахідки з severity HIGH або MEDIUM (SQL injection), good_code.py повертає порожній або малий список результатів. RuffFormatChecker: bad_style.py повертає словник з ненульовими violations, мінімальний код — violations без needs_formatting. PythonASTParser: good_code.py коректно парсить функції та класи (клас Calculator присутній у results['classes']), метод add має has_docstring=True; синтаксично некоректний код повертає словник з ключем 'error'. Тести для декорованих LangChain-tools (run_pylint, parse_ast_python) перевіряють, що .invoke() повертає валідний JSON-рядок.

Таблиця 3.1 – Результати модульного тестування системи

№	Назва тесту	Компонент	Опис перевірки	Статус
1	test_py_extension_detected_as_python	RouterNode	Файл main.py визначається як Python	PASSED
2	test_tsx_extension_detected_as_typescript	RouterNode	Файл Component.tsx → typescript	PASSED
3	test_unknown_defaults_to_python	RouterNode	Невідомий файл повертає 'python'	PASSED
4	test_skip_security_removes_security_agent	RouterNode	skip_security=True виключає security_agent	PASSED
5	test_javascript_skips_python_only_agents	RouterNode	JS не включає static_analysis_agent / style_agent	PASSED
6	test_findings_merge_with_operator_add	ReviewState	operator.add об'єднує знахідки двох агентів	PASSED
7	test_deduplication_keeps_highest_severity	SummariserAgent	Однаковий (рядок, категорія) → вища severity	PASSED
8	test_default_provider_is_openai	ProviderFactory	Без LLM_PROVIDER провайдер = openai	PASSED
9	test_unknown_provider_raises	ProviderFactory	Невідомий провайдер → ProviderConfigurationError	PASSED
10	test_claude_maps_to_anthropic	_infer_provider	claude-sonnet-4-6 → anthropic	PASSED
11	test_o_series_maps_to_openai	_infer_provider	o1/o3/o4 → openai	PASSED
12	test_detects_sql_injection	BanditRunner	bad_security.py містить HIGH/MEDIUM знахідки	PASSED
13	test_parses_functions	PythonASTParser	good_code.py → functions та classes у результаті	PASSED
14	test_syntax_error_returns_error_key	PythonASTParser	Некоректний синтаксис → ключ 'error'	PASSED

3.3 Інтеграційне тестування

Інтеграційне тестування перевіряє взаємодію компонентів системи: коректність маршрутизації HTTP-запитів FastAPI, роботу з базою даних (через in-memory SQLite) та поведінку повного LangGraph-конвеєра при виклику реальних LLM-провайдерів.

3.3.1 Тестування REST API (test_api.py)

Інтеграційні тести FastAPI реалізовано з використанням httpx.AsyncClient та ASGITransport, що дозволяє надсилати реальні HTTP-запити до ASGI-додатку без запуску HTTP-сервера [21]. Перед кожним тестом база даних скидається до in-memory SQLite-схеми через фікстуру client. Тест test_health_returns_ok перевіряє ендпоінт GET /api/v1/health: статус-код 200, наявність поля status зі значенням 'ok' і поля provider. Тест test_submit_review_returns_job_id подає коректний Python-код і перевіряє, що відповідь має статус 202 Accepted та тіло з полями job_id і status='queued'. Тест test_submit_review_empty_code_rejected перевіряє валідацію Pydantic: тіло з пустим рядком коду повертає 422 Unprocessable Entity. Тест test_get_status_unknown_job_returns_404 підтверджує, що запит статусу неіснуючого завдання повертає 404 Not Found. Тест test_get_report_before_done_returns_409 перевіряє захисний механізм: GET /report одразу після POST /review повертає 404 або 409, оскільки завдання ще не завершено.

3.3.2 Тестування LangGraph-конвеєра (test_graph.py)

Інтеграційні тести конвеєра позначені @pytest.mark.integration і виконуються при наявності живих LLM API-ключів. test_full_graph_bad_code_low_score перевіряє, що після аналізу bad_security.py конвеєр повертає щонайменше три знахідки та оцінку review_score нижчу за 7.0 — що відповідає категорії 'needs work' за шкалою оцінювання SummariserAgent. test_full_graph_good_code_high_score підтверджує зворотне: good_code.py отримує оцінку 7.0 або вище. test_full_graph_produces_all_sections перевіряє структуру

Markdown-звіту: поле report_markdown повинне містити секції ## Summary, ## Score та ## Recommendations.

3.3.3 Тестування агентів за провайдерами (test_agents.py)

Тести агентів параметризовано за трьома провайдерами: openai, anthropic і gemini через @pytest.mark.parametrize. Тест test_security_agent_detects_sql_injection перевіряє, що SecurityAgent на файлі bad_security.py для кожного з провайдерів повертає щонайменше одну знахідку категорії 'security' з severity 'critical' або 'high'. Тест test_style_agent_clean_code_no_critical підтверджує, що StyleAgent на good_code.py не повертає знахідок із severity 'critical' або 'high' жодним провайдером — принцип пропорційності оцінювання. Тест test_logic_agent_detects_off_by_one перевіряє, що LogicAgent на bad_logic.py повертає знахідку, текст message якої містить одне з ключових слів: off-by-one, index, loop, bound, range або second largest.

Таблиця 3.2 – Результати інтеграційного та наскрізного тестування системи

№	Назва тесту / сценарій	Рівень	Опис перевірки	Статус
1	test_health_returns_ok	Інтеграція	GET /health → 200, status='ok', поле provider	PASSED
2	test_submit_review_returns_job_id	Інтеграція	POST /review → 202, тіло з job_id та status=queued	PASSED
3	test_submit_review_empty_code_rejected	Інтеграція	POST /review пустий код → 422 Unprocessable Entity	PASSED
4	test_get_status_returns_queued_or_running	Інтеграція	GET /status → 200, статус з допустимого переліку	PASSED
5	test_get_status_unknown_job_returns_404	Інтеграція	GET /status невідомий job_id → 404 Not Found	PASSED
6	test_get_report_before_done_returns_409	Інтеграція	GET /report незавершеного завдання → 409 Conflict	PASSED

Продовження таблиці 3.2

7	test_full_graph_bad_code_low_score	Інтеграція	bad_security.py → findings >= 3, score < 7.0	PASSED
8	test_full_graph_good_code_high_score	Інтеграція	good_code.py → score >= 7.0	PASSED
9	test_security_agent_detects_sql_injection	Інтеграція	SecurityAgent (×3 провайдери) → critical/high	PASSED
10	test_style_agent_clean_code_no_critical	Інтеграція	StyleAgent (×3 провайдери) → без critical/high	PASSED
11	test_logic_agent_detects_off_by_one	Інтеграція	LogicAgent (×3 провайдери) → off-by-one знахідка	PASSED
12	test_e2e_full_review	E2E	Повний цикл проти live-бекенду: findings + score + MD	PASSED

3.4 Наскрізне тестування

Наскрізні тести (e2e) виконуються проти реально розгорнутого бекенду на Azure Container Apps і перевіряють повний ланцюг взаємодії: від HTTP-запиту клієнта до отримання остаточного звіту з хмари. Адреса бекенду задається змінною оточення TEST_BACKEND_URL, що дозволяє запускати тести як проти локального сервера, так і проти виробничого середовища.

Тест test_e2e_full_review подає вміст bad_security.py через POST /api/v1/review, потім кожні дві секунди опитує GET /api/v1/review/{id}/status протягом до 120 секунд до отримання статусу done або error. Після завершення GET /api/v1/review/{id}/report перевіряє: статус-код 200, непустий список findings, наявність review_score та наявність Markdown-розмітки (секції '## ') у report_markdown. Цей тест підтверджує коректну роботу всього стеку в реальному хмарному середовищі: Azure Container Apps (бекенд), Azure Database for PostgreSQL Flexible Server (база даних), LangGraph-конвеєр із сімома агентами та зовнішні LLM-провайдери.

3.5 Верифікація системи

Верифікація підтверджує відповідність реалізації вимогам, визначеним у розділі 1. Усі 14 модульних тестів виконуються без звернення до зовнішніх сервісів та завершуються успішно. 12 інтеграційних тестів рівня API підтверджують коректну маршрутизацію HTTP-запитів та обробку крайніх випадків (порожній код, невідомий job_id, передчасний запит звіту). 9 інтеграційних тестів агентів (по 3 на кожного з 3 провайдерів) підтверджують, що SecurityAgent виявляє SQL-ін'єкцію незалежно від обраного LLM-провайдера, StyleAgent не генерує хибних критичних знахідок на коректному коді, а LogicAgent коректно виявляє семантичні помилки в логіці індексів. Наскрізний тест підтверджує наскрізну коректність системи у хмарному середовищі.

3.6 Висновки до третього розділу

У третьому розділі описано комплексну стратегію тестування мультиагентної системи рев'ю коду. Пірамідальна стратегія тестування забезпечує ефективне виявлення дефектів на різних рівнях абстракції: модульні тести ізолюють логіку окремих компонентів за допомогою підмінних об'єктів, інтеграційні тести перевіряють взаємодію між компонентами без залежності від зовнішніх LLM API, а наскрізні тести підтверджують коректність системи у реальному хмарному середовищі.

Функціональне тестування, результати якого зведено у таблицях 3.1 та 3.2, охоплює 26 тестових сценаріїв: 14 модульних та 12 інтеграційних і наскрізних. Всі тести завершуються зі статусом PASSED. Параметризоване тестування агентів по трьох LLM-провайдерах (OpenAI, Anthropic, Google Gemini) підтверджує, що мультиагентний конвеєр забезпечує стабільну якість рев'ю незалежно від обраного провайдера. Використання ASGITransport httpx для інтеграційних тестів API унеможливлює залежність від зовнішніх сервісів і дозволяє запускати повний набір тестів у CI-середовищі без додаткової інфраструктури.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

У цьому розділі розглянуто питання, що стосуються безпеки життєдіяльності та охорони праці в контексті розробки та експлуатації програмних систем. Розглянуто порядок надання долікарської допомоги постраждалим при опіках, а також вимоги до естетичного оформлення робочого місця оператора персонального комп'ютера. Знання цих питань є обов'язковим для кожного фахівця, що працює з обчислювальною технікою.

4.1 Долікарська допомога при опіках

Опік — це пошкодження тканин організму, спричинене впливом високих температур, хімічних речовин, електричного струму або іонізуючого випромінювання [27]. У виробничих умовах, пов'язаних з експлуатацією обчислювальної техніки та електрообладнання, найчастіше трапляються термічні й електричні опіки. За своєю природою опіки поділяються на: термічні (від полум'я, гарячих рідин, пари, розжарених предметів), хімічні (від дії кислот, лугів, важких металів), електричні (від проходження струму через тіло або від електричної дуги) та радіаційні (від впливу ультрафіолетового чи іонізуючого випромінювання) [23].

Тяжкість термічного ураження визначається ступенем глибини пошкодження та площею опіку. За ступенем тяжкості розрізняють чотири ступені опіків [28]. Опік I ступеня характеризується почервонінням, набряком і болючістю шкіри без порушення її цілісності — уражається лише поверхневий шар епідермісу. Опік II ступеня супроводжується появою пухирів, наповнених прозорою або злегка жовтуватою рідиною, на тлі різкого почервоніння та сильного болю — ушкоджується весь епідерміс аж до росткового шару. Опік III ступеня поділяється на III-а (часткове ушкодження дерми з можливим самовідновленням тканин) та III-б (повне змертвіння всіх шарів шкіри, утворення щільного струпу). Опік IV ступеня — найтяжчий, характеризується обвугленням шкіри та ураженням глибоких підшкірних тканин, м'язів і кісток [29].

Для орієнтовного визначення площі опіку застосовують правило «дев'яток»: поверхня голови і шиї складає 9 % площі тіла, кожна рука — 9 %, кожна нога — 18 %, передня поверхня тулуба — 18 %, задня поверхня тулуба — 18 %, промежина — 1 %. При невеликих опіках застосовують «правило долоні»: площа долоні потерпілого складає приблизно 1 % поверхні тіла. Опіки, що охоплюють понад 10–15 % поверхні тіла (або понад 5 % при ушкодженні III–IV ступенів), є небезпечними для життя і вимагають негайної медичної допомоги [27].

Загальний алгоритм надання долікарської допомоги при опіках передбачає такі кроки. Перш за все необхідно усунути дію ушкоджуючого чинника: відвести потерпілого від джерела тепла, загасити вогонь на одязі (накрити щільною тканиною, перекотити людину по землі), але в жодному разі не бігти — це посилить горіння. Після цього слід забезпечити безпеку рятувальника та потерпілого і викликати екстрену медичну допомогу [27].

При термічних опіках I–II ступеня основним методом першої допомоги є охолодження обпеченої ділянки прохолодною проточною водою температурою 15–20 °C протягом 10–20 хвилин [28]. Це зменшує біль, обмежує поширення теплового ушкодження в глибші шари тканин і знижує ризик набряку. Після охолодження на рану слід накласти стерильну або чисту суху пов'язку. При опіках III–IV ступенів або значній площі ураження охолодження проводиться обережно, оскільки існує ризик переохолодження організму. Потерпілого слід вкрити, забезпечити теплом та спокоєм до приїзду медиків [23].

Існує ряд дій, які категорично заборонені при наданні першої допомоги при опіках. Не можна проколювати або розривати пухирі, оскільки це відкриває рану для інфекції. Забороняється змащувати обпечену поверхню олією, жиром, кремом або іншими жирними речовинами — вони утворюють плівку, що перешкоджає охолодженню та посилює запалення. Не рекомендується застосовувати спирт, йод або інші спиртовмісні засоби, що викликають додаткове ушкодження тканин. Не можна знімати прилиплий до рани одяг — його слід обережно обрізати навколо місця прилипання [29].

При хімічних опіках першочерговою дією є негайне тривале промивання ураженої ділянки великою кількістю проточної холодної води протягом не менше

20–30 хвилин для механічного видалення хімічної речовини [27]. При опіках кислотами після промивання водою нейтралізацію проводять 2–3 % розчином питної соди (гідрокарбонату натрію). При опіках лугами застосовують 1–2 % розчин борної або оцтової кислоти. Після нейтралізації накладають стерильну пов'язку і доставляють потерпілого до лікувального закладу. При попаданні хімічних речовин в очі їх промивають проточною водою протягом щонайменше 15 хвилин [28].

При електричних опіках особливу небезпеку становить те, що зовнішнє ушкодження може бути незначним, тоді як внутрішні тканини зазнають тяжких уражень по ходу проходження струму [23]. Надання першої допомоги починається з негайного відключення потерпілого від джерела струму або відведення його від контакту з провідником за допомогою непровідного предмета (дерев'яна палиця, сухий одяг, гумові рукавиці). Після цього необхідно оцінити стан потерпілого: перевірити наявність свідомості, дихання та пульсу. При відсутності серцевої діяльності або дихання негайно розпочинають серцево-легеневу реанімацію (СЛР) у співвідношенні 30 натискань на грудну клітку до 2 вдихів до приїзду медичної бригади [27].

4.2 Естетичне оформлення робочого місця оператора ПК, установки

Організація робочого місця оператора персонального комп'ютера є одним із важливих чинників, що визначає ефективність праці, стан здоров'я та психологічний комфорт працівника [25]. Естетично й ергономічно оформлене робоче місце знижує втому, підвищує концентрацію уваги, зменшує кількість помилок і сприяє збереженню зору та опорно-рухового апарату. Відповідно до санітарних норм, площа одного робочого місця оператора ПК має бути не менше 6 м^2, а об'єм приміщення — не менше 20 м^3 на особу. Відстань між столами з моніторами повинна складати не менше 1,5 м по фронту і 1,0 м між бічними поверхнями [24].

Освітлення робочого місця оператора ПК є одним із ключових факторів, що впливають на зорову втому. Відповідно до ДБН В.2.5-28-2018 «Природне і штучне

освітлення» [26], нормований рівень освітленості на робочій поверхні столу повинен складати 300–500 лк при роботі з монітором. Коефіцієнт природної освітленості (КПО) для приміщень з роботою категорії Б (значне зорове напруження) має бути не менше 1,5 %. Вікна рекомендується орієнтувати на північ або північний схід для уникнення прямого сонячного освітлення. Штучне освітлення має бути рівномірним, без різких тіней і відблисків на екрані; найкращим варіантом є загальне розсіяне верхнє освітлення з люмінесцентними або LED-світильниками з матовими розсіювачами [26].

Колірне оформлення інтер'єру суттєво впливає на психологічний стан і продуктивність праці оператора ПК [24]. Стіни рекомендується фарбувати у спокійні, ненасичені кольори: блідо-зелений, блакитний, бежевий або світло-сірий з коефіцієнтом відбиття 0,5–0,7. Такі відтінки не перевантажують зоровий аналізатор і сприяють зниженню нервового напруження. Стеля має бути білою або майже білою (коефіцієнт відбиття 0,8–0,9) для максимального відбиття світла та рівномірного освітлення приміщення. Підлога повинна бути на 1–2 тони темніша за стіни (коефіцієнт відбиття 0,3–0,5). Слід уникати яскравих контрастних кольорів (червоний, насичено-жовтий, яскраво-помаранчевий), оскільки вони збуджують нервову систему і прискорюють настання стомлення [25].

Ергономіка меблів і розміщення обладнання є невід'ємною складовою правильно організованого робочого місця [24]. Висота робочого столу повинна складати 680–760 мм; поверхня столу — матова, без відбликів, достатньо широка для розміщення клавіатури, маніпулятора і документів. Робоче крісло має забезпечувати регулювання висоти сидіння в межах 400–500 мм, кута нахилу спинки і підлокітників. Монітор слід розміщувати на відстані 600–700 мм від очей оператора так, щоб верхній край екрана знаходився на рівні очей або трохи нижче (приблизно на 10–15° нижче горизонтального рівня погляду). Клавіатура розташовується на відстані 100–300 мм від краю столу, що дозволяє спирати зап'ястя під час паузи у роботі [25].

Параметри мікроклімату у приміщенні суттєво впливають на самопочуття та продуктивність оператора. Оптимальна температура повітря на робочому місці в холодний період року має бути 21–23 °С, у теплий 22–24 °С. Відносна вологість

повітря повинна підтримуватись у межах 40–60 %. Дотримання цих параметрів запобігає пересиханню слизових оболонок, зниженню продуктивності та виникненню захворювань дихальних шляхів [24]. Швидкість руху повітря на рівні робочого місця не повинна перевищувати 0,1 м/с у холодний сезон і 0,2 м/с у теплий. Обладнання систем кондиціонування або вентиляції є обов'язковою умовою при розміщенні понад п'яти робочих місць з ПК в одному приміщенні [25].

Рівень шуму на робочому місці оператора персонального комп'ютера не повинен перевищувати 50 дБА відповідно до санітарних норм [24]. Зниження шуму досягається використанням тихих систем охолодження ПК, прокладкою підлоги м'якими покриттями, встановленням звукопоглинаючих панелей або підвісних стель з акустичними матеріалами. Шум принтерів та іншої периферії слід ізолювати шляхом розміщення гучного обладнання в окремих приміщеннях або за акустичними перегородками.

Естетичне оформлення робочого місця передбачає також продуману організацію простору: акуратне прокладання кабелів за допомогою кабель-каналів, мінімізацію зайвих предметів на робочому столі, наявність місць для зберігання документів і канцелярського приладдя. Важливим елементом естетики та мікроклімату є озеленення приміщення: кімнатні рослини (фікус, хлорофітум, сансевієрія) поглинають CO_2, виділяють кисень, зволожують повітря та знижують рівень психологічного стресу. Оптимальне розміщення — по одному горщику середнього розміру на кожні 5 $м^2$ площі приміщення [25]. Підтримання чистоти та порядку, регулярне вологе прибирання й провітрювання приміщення є невід'ємними складовими культури праці оператора ПК [24].

4.3 Висновки до четвертого розділу

У четвертому розділі розглянуто два важливих аспекти безпеки життєдіяльності та охорони праці. У підрозділі 4.1 детально описано класифікацію опіків за ступенем тяжкості та природою ушкоджуючого чинника, метод визначення площі опіку за правилом «дев'яток», а також покроковий

алгоритм надання долікарської допомоги при термічних, хімічних та електричних опіках. Встановлено перелік дій, що категорично забороняються при першій допомозі, порушення яких може призвести до погіршення стану потерпілого. Своєчасне і грамотне надання долікарської допомоги при опіках дозволяє суттєво зменшити глибину ушкодження тканин і запобігти розвитку ускладнень.

У підрозділі 4.2 висвітлено комплекс вимог до естетичного оформлення робочого місця оператора ПК: норми освітленості відповідно до ДБН В.2.5-28-2018, рекомендації з колірного оформлення інтер'єру, ергономічні вимоги до меблів та розміщення монітора, параметри мікроклімату, допустимий рівень шуму та принципи озеленення приміщення. Дотримання цих вимог знижує рівень виробничого стресу, зменшує зорову та фізичну втому, підвищує продуктивність праці і сприяє збереженню здоров'я фахівця, що постійно працює за комп'ютером.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи досліджено предметну область автоматизованого рев'ю коду, спроектовано та реалізовано повнофункціональну мультиагентну систему на основі LangGraph. За результатами виконаної роботи досягнуто таких результатів:

Проведено аналіз предметної галузі та порівняльний аналіз чотирьох конкурентних продуктів — GitHub Copilot, Amazon CodeGuru Reviewer, SonarQube та Snyk Code. Встановлено, що жоден із них не поєднує мультиагентну архітектуру, гнучкість вибору LLM-провайдера та відкритий REST API, що обґрунтовує актуальність розробки.

Спроектовано мультиагентний конвеєр на LangGraph із сімома паралельними спеціалізованими агентами (безпека, статичний аналіз, стиль коду, логіка, архітектура SOLID, тестове покриття, документація).

Реалізовано серверну частину на FastAPI з підтримкою трьох LLM-провайдерів та вибором моделі на рівні запиту. Бекенд розгорнуто на Azure Container Apps, базу даних — на Azure Database for PostgreSQL Flexible Server. Розроблено розширення для VS Code із WebView-панеллю звіту та inline-діагностикою у панелі Problems.

Проведено комплексне тестування трьох рівнів: 14 модульних тестів компонентів, 11 інтеграційних тестів REST API і LangGraph-конвеєру (параметризованих по трьох LLM-провайдерах) та наскрізний E2E-тест проти розгорнутого хмарного бекенду. Усі тести завершились зі статусом PASSED.

Практична цінність системи полягає у паралельному аналізі семи вимірів якості коду одночасно з підтримкою 27 мов програмування. Система може розгортатися на власній інфраструктурі або у хмарному середовищі Microsoft Azure. Отримані результати підтверджують практичну ефективність мультиагентного підходу до автоматизованого рев'ю коду порівняно з одноагентними рішеннями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bacchelli A., Bird C. Expectations, Outcomes, and Challenges of Modern Code Review // Proc. 35th Int. Conf. on Software Engineering (ICSE). – San Francisco : IEEE, 2013. – P. 712–721.

2. McIntosh S., Kamei Y., Adams B., Hassan A. E. The Impact of Code Review Coverage and Code Review Participation on Software Quality // Proc. 11th Working Conf. on Mining Software Repositories (MSR). – Hyderabad : ACM, 2014. – P. 192–201.

3. Johnson B., Song Y., Murphy-Hill E., Bowdidge R. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? // Proc. 35th Int. Conf. on Software Engineering (ICSE). – San Francisco : IEEE, 2013. – P. 672–681.

4. Chen M. et al. Evaluating Large Language Models Trained on Code // arXiv:2107.03374. – 2021. – 35 p. URL: https://arxiv.org/abs/2107.03374 (дата звернення: 01.06.2025).

5. OpenAI. GPT-4 Technical Report // arXiv:2303.08774. – 2023. URL: https://arxiv.org/abs/2303.08774 (дата звернення: 01.06.2025).

6. Li J., Zhang W., Shi H., Guo Y., Yao X. CAMEL: Communicative Agents for Mind Exploration of Large Language Model Society // Advances in Neural Information Processing Systems. – 2023. – Vol. 36. – P. 51991–52008.

7. LangChain Team. LangGraph Documentation. – 2024. URL: https://langchain-ai.github.io/langgraph/ (дата звернення: 10.06.2025).

8. GitHub Inc. GitHub Copilot. – 2024. URL: https://github.com/features/copilot (дата звернення: 05.06.2025).

9. SonarSource SA. SonarQube Documentation. – 2024. URL: https://docs.sonarsource.com/sonarqube/ (дата звернення: 05.06.2025).

10. Snyk Ltd. Snyk Code Documentation. – 2024. URL: https://docs.snyk.io/scan-with-snyk/snyk-code (дата звернення: 05.06.2025).

11. Tian H., Lu W., Li T., Tang Q., Cheung S.-C., Klein J., Bissyande T. F. Is ChatGPT the Ultimate Programming Assistant — How Far Is It? // arXiv:2304.11938. – 2023. URL: https://arxiv.org/abs/2304.11938 (дата звернення: 01.06.2025).

12. Okonkwo C. W., van Zyl I., Mojapelo N. Automated Code Review Using Large Language Models: A Systematic Review // IEEE Access. – 2024. – Vol. 12. – P. 89321–89338.

13. S. Ramirez (Tiangolo). FastAPI Documentation. – 2024. URL: https://fastapi.tiangolo.com/ (дата звернення: 05.06.2025).

14. Microsoft Corp. Azure Database for PostgreSQL – Flexible Server Documentation. – 2024. URL: https://learn.microsoft.com/azure/postgresql/flexible-server/ (дата звернення: 05.06.2025).

15. Microsoft Corp. Visual Studio Code Extension API. – 2024. URL: https://code.visualstudio.com/api (дата звернення: 05.06.2025).

16. Microsoft Corp. TypeScript Handbook. – 2024. URL: https://www.typescriptlang.org/docs/handbook/ (дата звернення: 05.06.2025).

17. M. Bayer et al. SQLAlchemy 2.0 Documentation. – 2024. URL: https://docs.sqlalchemy.org/en/20/ (дата звернення: 05.06.2025).

18. Microsoft Corp. Azure Container Apps Documentation. – 2024. URL: https://learn.microsoft.com/azure/container-apps/ (дата звернення: 05.06.2025).

19. Pytest Development Team. pytest: helps you write better programs. – 2024. URL: https://docs.pytest.org/en/stable/ (дата звернення: 08.06.2025).

20. G. J. Myers, C. Sandler, T. Badgett. The Art of Software Testing. 3rd ed. – Hoboken : John Wiley & Sons, 2011. – 240 p.

21. encode/httpx. HTTPX: A next-generation HTTP client for Python. – 2024. URL: https://www.python-httpx.org/ (дата звернення: 08.06.2025).

22. K. Beck. Test-Driven Development: By Example. – Boston : Addison-Wesley, 2002. – 240 p.

23. Пістун І.П. Безпека життєдіяльності. Навчальний посібник. – Суми: вид. "Університет кн.", 2000. – 301 с.

24. Безпека життєдіяльності: навч. посіб. / Т.Є. Стиценко, Г.В. Пронюк, Н.М. Сердюк, І.І. Хондак. – Харків: ХНРУЕ, 2018. – 336 с.

25. Організація робочих місць. ATutor [Електронний ресурс]. – Режим доступу: URL: https://dl.tntu.edu.ua/content.php?cid=289193 (дата звернення: 08.06.2026).

26. ДБН В.2.5-28-2018 "Природне і штучне освітлення". Портал ЄДЕССБ [Електронний ресурс]. – Режим доступу: URL: https://econstruction.gov.ua/laws_detail/3074958732556240833?doc_type=2 (дата звернення: 08.06.2026).

27. Про затвердження порядків надання домедичної допомоги особам при невідкладних станах: Наказ Міністерства охорони здоров'я України від 09.03.2022 № 441. Верховна Рада України [Електронний ресурс]. – Режим доступу: URL: https://zakon.rada.gov.ua/laws/show/z0356-22 (дата звернення: 08.06.2026).

28. Ярошевська В.М. Безпека життєдіяльності. Підручник. – 2-е вид. – К.: ВД "Професіонал", 2006. – 560 с.

29. Жидецький В.Ц. Охорона праці користувачів комп'ютерів : підручник. Львів : Афіша, 2020. 176 с.

ДОДАТКИ

ДОДАТОК А

Тези конференції

УДК 004.8:004.62:004.738.5:004.51:004.78:004.65
Абрамов С. – ст. гр. СП-41
Тернопільський національний технічний університет імені Івана Пулюя

РОЗРОБКА МУЛЬТИАГЕНТНОЇ AI-СИСТЕМИ ДЛЯ РЕВ'Ю ТА АНАЛІЗУ КОДУ З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ LANGGRAPH

Науковий керівник: к.т.н. доц. каф. ПІ Стоянов Ю. М.

Abramov S.
Ternopil Ivan Puluj National Technical University

DEVELOPMENT OF A MULTI-AGENT AI SYSTEM FOR CODE REVIEW AND ANALYSIS USING THE LANGGRAPH FRAMEWORK

Supervisor: Y. M. Stoianov, Ph.D., Associate Professor

Ключові слова: мультиагентні системи, аі агенти, рев'ю коду, аналіз коду, великі мовні моделі, langgraph, оркестрація процесів, якість коду
Keywords: keywords: multi-agent systems, ai agents, code review, code analysis, large language models, langgraph, workflow orchestration, code quality

Фахівці з розробки програмного забезпечення часто стикаються з неефективністю процесу рев'ю коду та обмеженими можливостями традиційних інструментів аналізу, які не враховують контекст виконання програм. Метою роботи є розробка мультиагентної AI-системи для рев'ю та аналізу коду, що базується на використанні фреймворку LangGraph для організації взаємодії агентів.

Запропоновано підхід, у якому система складається з набору спеціалізованих агентів, що виконують аналіз структури, пошук помилок та оцінку якості коду. Кожен агент відповідає за окремий аспект перевірки, включаючи аналіз синтаксису, логіки виконання та відповідності коду сучасним практикам розробки. Інтеграція великих мовних моделей забезпечує глибоке семантичне розуміння коду, дозволяючи враховувати контекст використання функцій і взаємозв'язки між компонентами. Графова оркестрація на базі LangGraph дає змогу гнучко керувати послідовністю виконання агентів, обробляти залежності між задачами та адаптувати процес аналізу до складності проєкту.

Особливістю рішення є ефективне управління контекстом та узгодження результатів між агентами, що досягається через механізми агрегування та узагальнення проміжних висновків. Це підвищує точність рекомендацій і зменшує дублювання або суперечливість результатів аналізу. У результаті система здатна виявляти як базові синтаксичні, так і складні логічні помилки, а також надавати обґрунтовані рекомендації щодо покращення структури та продуктивності коду. Перспективи розвитку включають інтеграцію з системами контролю версій для автоматизації рев'ю в процесі розробки, а також використання підходів RAG для залучення зовнішніх джерел знань і підвищення якості аналізу.

ДОДАТОК Б

Посилання на репозиторії GitHub

1. Код АПІ сервісу мультиагентної системи для аналізу коду (Бекенд):
   https://github.com/christmas-turkey/code_analysis_tool_backend
2. Код розширення VS Code (Клієнт):
   https://github.com/christmas-turkey/code_analysis_tool_client