Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Інтелектуальна мультиагентна екосистема для підтримки життєвого циклу програмного забезпечення
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-22-3
Софія РУДЬ
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)

Керівник доц. Марія ГОЛОВЯНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Лариса ЧАЛА
(підпис) (власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Рудь Софії Іванівні
(прізвище, ім'я, по батькові)

1. Тема роботи Інтелектуальна мультиагентна екосистема для підтримки життєвого циклу програмного

затверджена наказом університету від 5 червня 2026 р. № 488Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 8 липня 20 26 р.
3. Вихідні дані до роботи Науково-технічні публікації та дані Інтернет-джерел у сфері кібербезпеки (DevSecOps), офіційна технічна документація фреймворку CrewAI та великих мовних моделей (Llama 3), міжнародні реєстри вразливостей (OWASP, CWE), набір програмних скриптів для тестування та верифікації розробленої системи

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі та постановка задачі
2) Обґрунтування напрямку дослідження та архітектурне моделювання
3) Розробка інтелектуальної мультиагентної екосистеми
4) Проведення експериментальних досліджень та оцінка ефективності

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.06.2026	виконано
2	Аналіз предметної галузі	09.06.2026	виконано
3	Огляд літературних джерел	10.06.2026	виконано
4	Обґрунтування технологічного стеку та	11.06.2026–12.06.2026	виконано
	проєктування архітектури системи		
5	Програмна реалізація ШІ-екосистеми	13.06.2026–17.06.2026	виконано
6	Проведення експериментальних досліджень	18.06.2026–20.06.2026	виконано
7	Написання пояснювальної записки	21.06.2026–22.06.2026	виконано
8	Перевірка на академічний плагіат	23.06.2026	виконано
9	Нормоконтроль	25.06.2026	виконано
10	Підготовка презентації та доповіді	27.06.2026	виконано
11	Попередній захист	30.06.2026	виконано
12	Рецензування	04.07.2026	виконано
13	Захист перед ЕК	08.07.2026	

Дата видачі завдання 8 червня 20 26 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Марія ГОЛОВЯНКО
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 99 с., 17 рис., 8 табл., 4 дод., 26 джерел.

АВТОМАТИЗОВАНИЙ АУДИТ БЕЗПЕКИ, ВЕЛИКІ МОВНІ МОДЕЛІ, ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МУЛЬТИАГЕНТНІ СИСТЕМИ, РЕФАКТОРИНГ КОДУ, CREWAI, DEVSECOPS.

Об’єкт дослідження – процеси інтеграції механізмів кібербезпеки в життєвий цикл розробки програмного забезпечення (SDLC).

Предмет дослідження – методи, моделі та технології побудови автономних мультиагентних систем для автоматизованого аудиту, рефакторингу та верифікації вихідного коду.

Мета роботи – проєктування автономної мультиагентної ШІ-екосистеми, здатної без участі людини виконувати повний DevSecOps-цикл: від пошуку вразливостей до безпечного рефакторингу та фінальної QA-верифікації коду.

Методи дослідження – системний аналіз архітектур ШІ-агентів та їх інтеграції у CI/CD процесах. Практична реалізація спирається на фреймворк CrewAI для рольової оркестрації та передові практики Prompt Engineering, що забезпечують мовну універсальність системи (Polyglot mode).

У результаті роботи досліджено проблематику використання LLM у DevSecOps (зокрема, семантичне зміщення та втрату контексту). Запропоновано інноваційну трирівневу архітектуру ШІ-асистентів, яка автоматизує обробку логіки застосунків та синхронізується із системами контролю версій для генерації готових звітів і патчів.

ABSTRACT

Bachelor’s thesis contains: 99 pp., 17 fig., 8 tabl., 4 ann., 26 references.

AUTOMATED SECURITY AUDIT, CODE REFACTORING, CREWAI, DEVSECOPS, LARGE LANGUAGE MODELS, MULTI-AGENT SYSTEMS, SOFTWARE DEVELOPMENT LIFE CYCLE.

Object of research – processes of integrating cybersecurity mechanisms into the software development life cycle (SDLC).

Purpose of the work – designing an autonomous multi-agent AI ecosystem capable of performing a full DevSecOps cycle without human intervention: from vulnerability scanning to secure refactoring and final QA verification of the code.

Research methods – system analysis of AI agent architectures and their integration into CI/CD processes. The practical implementation relies on the CrewAI framework for role-based orchestration and advanced Prompt Engineering practices that ensure the system's language universality (Polyglot mode).

Results – the problematic aspects of using LLMs in DevSecOps (in particular, semantic drift and context loss) were investigated. An innovative three-tier architecture of AI assistants is proposed, which automates the processing of application logic and synchronizes with version control systems to generate ready-made reports and patches.

Practical value – the developed solution has high applied potential and serves as a ready-made engineering foundation for the implementation of autonomous security in modern continuous integration and deployment pipelines.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 8
Вступ 10
1 Аналіз предметної галузі та постановка задачі 12
1.1 Опис предметної галузі 12
1.2 Опис проблем та викликів 15
1.3 Огляд існуючих застосунків 18
1.4 Інноваційність та переваги розроблюваної системи 21
1.5 Постановка задачі 24
2 Обґрунтування напрямку дослідження та архітектурне моделювання ... 27
2.1 Обґрунтування вибору технологічного стеку та інструментальних засобів 27
2.2 Аналіз архітектур мультиагентних систем 30
2.3 Огляд методів оркестрації агентів 34
2.4 Обґрунтування вибору llm 37
3 Розробка інтелектуальної мультиагентної екосистеми 42
3.1 Проєктування загальної архітектури та рольової моделі системи 42
3.2 Реалізація когнітивних агентів та управління системними промптами 47
3.3 Інтеграція зовнішніх інструментів статичного аналізу та синтаксичної верифікації 50
4 Експериментальні дослідження та оцінка ефективності 53
4.1 Опис експериментальної методики та середовища тестування 53
4.2 Аналіз результатів тестування та оцінка оптимізації системи 59
4.3 Оцінка достовірності результатів та аналіз обмежень системи 63
Висновки 67
Перелік джерел посилання 70
Додаток А Лістинг системних інструкцій (промптів) когнітивних агентів 73
Додаток Б Приклад базової архітектури 75

Додаток В Приклад фінальної оптимізованої архітектури..... 80
Додаток Г Відомість кваліфікаційної роботи..... 99

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ВММ – велика мовна модель;

ІДЕ – інтегроване середовище розробки;

ПЗ – програмне забезпечення;

ШІ – штучний інтелект;

API – Application Programming Interface – інтерфейс прикладного програмування;

Application Purpose Anchor – «якір бізнес-логіки» – механізм утримання когнітивного фокусу мовної моделі на початковому призначенні коду для запобігання галюцинаціям;

Backstory – передісторія – системна інструкція, що визначає професійну роль та поведінкові патерни ШІ-агента;

CI/CD – Continuous Integration / Continuous Deployment – безперервна інтеграція та безперервне розгортання програмного забезпечення;

CWE – Common Weakness Enumeration – загальний реєстр типів слабкостей та уразливостей програмного коду;

DevSecOps – Development, Security, and Operations – парадигма безшовного об'єднання процесів розробки, операційного супроводу та забезпечення безпеки інформаційних систем;

Feedback Loop – цикл зворотного зв'язку – ітеративний процес автоматичного повернення коду на доопрацювання до успішного проходження верифікації;

JSON – JavaScript Object Notation – текстовий формат обміну даними, що використовується для структурування повідомлень між агентами;

LLM – Large Language Model – велика мовна модель;

MAS – Multi-Agent System – мультиагентна система, розподілена мережа взаємодії автономних інтелектуальних вузлів;

NIST – National Institute of Standards and Technology – Національний інститут стандартів і технологій США, що регламентує стандарти безпечної розробки;

OWASP – Open Web Application Security Project – відкритий проєкт забезпечення безпеки веб-застосунків та ієрархія критичних ризиків;

Production-Ready – експлуатаційна готовність – стан програмного коду, повністю готового до розгортання без додаткового ручного доопрацювання;

Prompt Engineering – інженерія підказок – методологія проєктування та оптимізації вхідних інструкцій для великих мовних моделей;

PWA – Progressive Web App – прогресивний веб-застосунок;

RAG – Retrieval-Augmented Generation – пошуково-оптимізована генерація контексту для мовних моделей;

SAST – Static Application Security Testing – метод статичного аналізу та тестування захищеності вихідного коду застосунків;

SCA – Software Composition Analysis – автоматизований аналіз безпеки та ліцензійної чистоти сторонніх компонентів і залежностей коду;

SDLC – Software Development Life Cycle – життєвий цикл розробки програмного забезпечення;

Shift-Left – інженерний підхід, спрямований на перенесення операцій тестування та перевірки безпеки на найбільш ранні етапи створення продукту;

TPM – Tokens Per Minute – токени за хвилину;

VAR – Vulnerability Analysis Report – аналітичний звіт щодо верифікації та локалізації дефектів безпеки;

VPN – Virtual Private Network – віртуальна приватна мережа для віддаленого безпечного підключення до репозиторіїв.

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується інтенсифікацією процесів розробки програмного забезпечення, що неминуче супроводжується зростанням архітектурної складності інформаційних систем та експоненційним збільшенням кількості кіберзагроз. Традиційні методи забезпечення безпеки програмного коду, такі як статичний аналіз чи мануальне тестування, виявляються надмірно ресурсомісткими, продукують значну кількість хибних спрацьовувань та суттєво сповільнюють загальний цикл розробки. Спроби застосування ізольованих великих мовних моделей для автоматизації аудиту також демонструють ряд недоліків, зокрема схильність до семантичних викривлень, генерацію поверхневих виправлень та ризик порушення початкової бізнес-логіки застосунків. З огляду на це, виникає гостра науково-практична потреба в інтеграції механізмів безпеки у життєвий цикл розробки програмного забезпечення «SDLC» на якісно новому, інтелектуальному рівні.

Актуальність теми зумовлена необхідністю створення новітніх інструментів у межах парадигми DevSecOps, які здатні не лише в автоматичному режимі ідентифікувати вразливості, але й автономно їх усувати та верифікувати результати без безпосереднього втручання оператора. Найбільш перспективним підходом до вирішення цієї проблеми є розробка автономних мультиагентних екосистем, де спеціалізовані штучні інтелектуальні агенти функціонують у єдиному конвеєрі, забезпечуючи перехресну валідацію та збереження архітектурної цілісності програмного коду.

Об’єктом дослідження виступають процеси забезпечення безпеки та інформаційної підтримки життєвого циклу розробки програмного забезпечення. Предметом дослідження є методи, моделі та технології побудови автономних мультиагентних систем для автоматизованого

аудиту, рефакторингу та верифікації вихідного коду. Метою роботи є системний аналіз предметної галузі та теоретико-практичне обґрунтування концепції інтелектуальної мультиагентної екосистеми, здатної реалізовувати повний цикл безпечної обробки коду з використанням сучасних великих мовних моделей та інструментальних засобів рольової оркестрації.

Для досягнення поставленої мети в межах дослідження вирішується комплекс взаємопов'язаних завдань. Зокрема, здійснюється ґрунтовний аналіз сучасних підходів до автоматизації етапів SDLC та інтеграції методів штучного інтелекту в процеси безперервної інтеграції та розгортання. Проводиться критичне оцінювання обмежень існуючих інструментів статичного аналізу та ізольованих мовних моделей. На основі отриманих результатів проєктується трирівнева архітектура автономних агентів з чітким розподілом функціональних обов'язків, що включає модулі аудиту, інженерної корекції та забезпечення якості. Окрім цього, здійснюється вибір та обґрунтування оптимального технологічного стеку, що базується на використанні сучасних фреймворків оркестрації, передових мовних моделей та алгоритмів взаємодії з розподіленими системами керування версіями.

Практичне значення отриманих результатів полягає у формуванні надійної архітектурної бази для створення інноваційного програмного рішення, впровадження якого дозволить суттєво оптимізувати часові витрати на аудит безпеки, мінімізувати деструктивний вплив людського фактора та запобігти інтеграції вразливостей у промислове середовище експлуатації. Зібрані матеріали та розроблені алгоритмічні моделі слугують міцним науково-інженерним підґрунтям для успішного виконання та подальшого захисту кваліфікаційної роботи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної галузі

Стрімкий розвиток глобальної індустрії програмного забезпечення та цифровізація критично важливих сфер людської діяльності зумовили значне ускладнення архітектури сучасних інформаційних систем. У цих умовах створення надійного, масштабованого та безпечного програмного продукту вимагає суворого дотримання методологічних засад інженерії програмного забезпечення. Базовим концептуальним каркасом, що регламентує всі етапи створення програмних систем, є життєвий цикл розробки програмного забезпечення.

Життєвий цикл розробки програмного забезпечення (Software Development Life Cycle, SDLC) визначається як систематизований процес, що застосовується для проєктування, створення, тестування та супроводу високоякісних програмних продуктів. Головна мета SDLC полягає в оптимізації часових та фінансових витрат при гарантуванні того, що кінцева система повністю відповідає встановленим функціональним та нефункціональним вимогам [5].

Класична модель життєвого циклу послідовно охоплює такі базові фази, як планування та збір вимог, проєктування архітектури майбутньої системи, безпосередня програмна реалізація або кодування, верифікація та тестування дефектів, розгортання у промисловому середовищі експлуатації, а також подальша технічна підтримка й обслуговування застосунку як показано на рисунку 1.1. Кожен із цих етапів має чітко визначені вхідні та вихідні критерії валідації даних, що забезпечує прозорість та контрольованість інженерного процесу [15]. Історично склалося так, що питання забезпечення інформаційної безпеки розглядалися як ізольована процедура. Такий підхід створював деструктивний ефект «вузького місця».

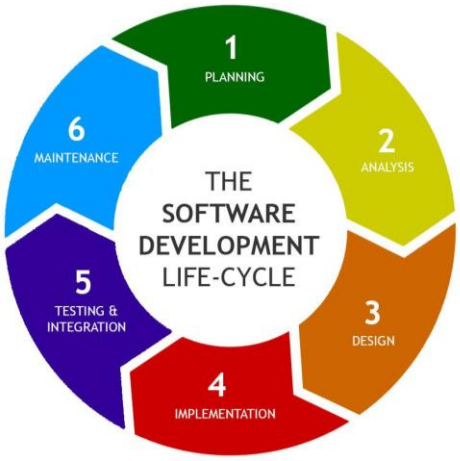


Рисунок 1.1 – Класичні етапи життєвого циклу розробки ПЗ (SDLC)

Історично склалося так, що питання забезпечення інформаційної безпеки розглядалися як ізольована процедура, яка виконувалася на фінальних стадіях життєвого циклу, переважно після завершення основного етапу тестування або безпосередньо перед релізом системи. Такий застарілий підхід створював деструктивний ефект, коли аналітики з безпеки блокували розгортання готового продукту через виявлення критичних уразливостей. Це призводило до значних фінансових та часових втрат, оскільки вартість усунення архітектурного дефекту на етапі релізу є експоненційно вищою порівняно з етапом безпосереднього кодування.

Оцінюючи архітектуру SDLC, науковці та інженери часто використовують графічні діаграми часової шкали, де етапи розробки розгорнуті зліва направо – від збору вимог до розгортання. У межах наукового обґрунтування цієї проблеми доцільно розглядати класичний графік залежності вартості виправлення дефектів від стадії життєвого циклу. На такому графіку чітко простежується нелінійне зростання витрат, де помилка, вчасно не усунута на етапі кодування, на етапі експлуатації вимагає колосальних ресурсів, включаючи повторне тестування, архітектурні зміни та ризики компрометації даних користувачів.

Еволюційною відповіддю на ці виклики став перехід індустрії від ізольованих моделей розробки до інтегрованої парадигми DevSecOps, яка

передбачає безшовне впровадження практик безпеки у конвеєри безперервної інтеграції та розгортання [7], [16]. У межах цієї парадигми безпека перестає бути дискретним етапом і трансформується у наскрізний процес, що супроводжує кожну фазу SDLC. Ключовим філософським та інженерним принципом DevSecOps є концепція «Shift-Left Security», тобто зміщення процесів тестування та аналізу безпеки ліворуч за часовою шкалою розробки, ближче до моменту написання перших рядків вихідного коду розробником.

Характер розподілу інженерних зусиль та динаміку зміни уваги до безпекових аспектів у межах різних підходів наочно проілюстровано на рисунку 1.2. На цій діаграмі порівнюються дві принципово різні парадигми управління захищеністю впродовж усього життєвого циклу інформаційної системи, що охоплює фази планування та проєктування, розробки та збирання коду, тестування, розгортання та релізу, а також подальшого моніторингу й аналізу.

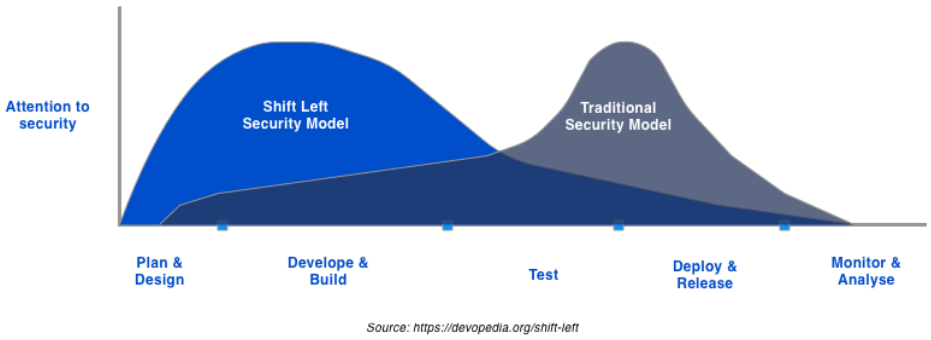


Рисунок 1.2 – Концепція «Shift-Left Security» на графіку

Традиційне впровадження концепції Shift-Left базується на застосуванні інструментів статичного аналізу коду, які перевіряють синтаксис на наявність відомих сигнатур уразливостей. Проте поточний стан технологій виявляє обмеженість таких інструментів через високий рівень генерації хибних позитивних результатів та їхню нездатність

інтерпретувати семантичний контекст програми. Вони лише констатують наявність підозрілої ділянки тексту, не пропонуючи механізмів інтелектуального виправлення.

Саме тому сучасний етап розвитку предметної галузі характеризується активним залученням методів штучного інтелекту, а саме великих мовних моделей, навчених на репрезентативних масивах програмного коду. Завдяки архітектурі Transformer, такі моделі здатні аналізувати не лише синтаксичні паттерни, але й глибинну семантику та бізнес-логіку застосунків, що дозволяє виявляти складні вразливості та автоматично генерувати контекстно-залежні безпечні патчі [6], [9].

Проте застосування одиночних LLM у форматі лінійного запиту має критичні недоліки, пов'язані з нестабільністю генерації та відсутністю системного контролю якості. Це зумовлює перехід до проєктування інтелектуальних мультиагентних екосистем, де за допомогою спеціалізованих фреймворків оркестрації створюється динамічна мережа взаємодії ШІ-агентів [1], [2], [8]. Кожен агент наділяється унікальною професійною роллю та чіткими інструкціями у межах віртуального DevSecOps-підрозділу. Такий підхід дозволяє реалізувати автоматизований пре-комміт аудит коду за принципом розділення обов'язків, де один агент виконує функції глибокого аналізу безпеки, другий реалізує secure- рефакторинг, а третій забезпечує суворий контроль якості та синтаксичну верифікацію, що повністю закриває цикл безпечної розробки у межах концепції Shift-Left.

1.2 Опис проблем та викликів

Ефективна реалізація концепцій безпечної розробки в межах сучасних інженерних конвеєрів стримується низкою взаємопов'язаних технологічних, архітектурних та організаційних чинників. Першочерговим деструктивним фактором є критичний дефіцит кваліфікованого кадрового потенціалу в

галузі прикладної кібербезпеки та безпеки застосунків «Application Security». Динаміка генерації та розгортання програмного коду сучасними розподіленими командами суттєво випереджає фізичні й когнітивні можливості експертів щодо його мануального аналізу чи проведення повноцінного рецензування. Спроби перекласти функцію контролю безпеки на лінійних програмних інженерів часто демонструють низьку ефективність, оскільки їхня основна експертиза зосереджена на забезпеченні функціональності, оптимізації швидкодії та виконанні бізнес-вимог, що створює передумови для ігнорування латентних загроз і вразливостей вихідного коду.

Поряд із кадровим голодом, суттєвим викликом є обмеженість можливостей класичних інструментальних засобів автоматизованого тестування захищеності. Статичні аналізатори коду «SAST», які тривалий час виступали основою превентивного аудиту, функціонують переважно на базі сигнатурного пошуку, регулярних виразів та жорстких евристичних правил синтаксичних дерев. Наслідком такої технологічної архітектури є генерація надмірної кількості хибнопозитивних спрацьовувань, що змушує інженерні команди витрачати значні інтелектуальні ресурси на ручну фільтрацію інформаційного «шуму». Найбільш критичним обмеженням традиційного інструментарію є його виключно дескриптивна природа, оскільки системи SAST лише реєструють дефект, але не пропонують жодних механізмів його автоматичного усунення.

Характер виникнення зазначеного технологічного бар'єра та послідовність етапів, на яких утворюються критичні часові втрати через мануальну обробку надмірного обсягу попереджень, детально проілюстровано на рисунку 1.3. Як видно зі схеми, після автоматичного сканування системи генерують об'ємні звіти, що містять сотні попереджень. Етапи ручної верифікації цих потенційних помилок та подальшого самостійного написання безпекових патчів розробником стають ізольованими зонами неефективності (пляшковим горлечком). Це

призводить до критичних часових втрат і повністю нівелює швидкісні переваги автоматичного сканування, уповільнюючи реліз програмного продукту.



Рисунок 1.3 – Схема виникнення технологічного обмеження у конвеєрі розробки при використанні традиційних інструментів статичного аналізу

Спроби вирішити зазначені проблеми за допомогою прямого залучення великих мовних моделей у форматі лінійних систем штучного інтелекту призвели до появи нових технологічних ризиків. Використання ізольованих LLM без наявності зовнішніх систем контролю супроводжується проблемою семантичного зміщення, за якого модель може інтегрувати у патч неіснуючі залежності, виклики фіктивних API або застарілі бібліотеки з відомими дефектами. Вагомим викликом також є

ризик функціональної регресії програмного забезпечення. Намагаючись усунути безпековий дефект, поодинока ШІ-модель часто вдається до радикального спрощення або видалення вразливих ділянок коду, що призводить до порушення початкової бізнес-логіки застосунку, спотворення сигнатур функцій та дестабілізації суміжних модулів інформаційної системи.

Резюмуючи вищезазначене, можна стверджувати, що сучасний етап розвитку DevSecOps характеризується системним технологічним розривом. З одного боку, існує гостра потреба у безперервному та безпечному постачанні коду, а з іншого – наявний інструментарій виявляється або занадто повільним, або продукує високий уровень похибок. Відсутність у сучасних ШІ-рішеннях вбудованих механізмів внутрішньої верифікації, автоматичного тестування синтаксичної цілісності та перехресного аналізу результатів унеможливлює їх пряму інтеграцію в промислове конвеєри розгортання. Вирішення цих суперечностей вимагає розробки інтелектуальних систем закритого циклу, здатних не лише ідентифікувати загрози, але й здійснювати їх прецизійний рефакторинг та багаторівневий контроль якості без залучення людського ресурсу.

1.3 Огляд існуючих застосунків

Для обґрунтування доцільності розробки новітньої інтелектуальної системи необхідно провести критичний аналіз існуючих програмних рішень у сфері автоматизації безпеки вихідного коду та підтримки процесів DevSecOps. Сучасний ринок відповідного інструментарію доцільно розподілити на два ключові технологічні сегменти, перший з яких охоплює класичні платформи статичного аналізу захищеності (SAST), а другий – сучасні генеративні асистенти на базі штучного інтелекту.

Серед традиційних інструментів автоматизованого сканування вагоме місце посідає платформа SonarQube, яка широко застосовується в індустрії

для безперервного контролю якості та безпеки коду. Вона забезпечує ефективне виявлення потенційних дефектів (code smells), архітектурних невідповідностей стандарту та базових уразливостей, що належать до класифікацій OWASP та CWE [4], [11]. Проте функціональні обмеження SonarQube полягають у її дескриптивній природі та використанні жорстких регулярних виразів. Система функціонує виключно як інформаційне середовище, що констатує наявність проблеми, але повністю усуває можливість автоматичного виправлення синтаксичних чи логічних помилок [13]. Аналогічні обмеження притаманні й платформі Snyk, яка, попри потужну базу даних уразливостей та розвинені механізми перевірки сторонніх залежностей (SCA), вимагає безпосереднього залучення людського ресурсу для розробки та інтеграції безпекових патчів [12]. Обидва рішення орієнтовані на фахівців з кібербезпеки, що створює часові розриви у конвеєрі розробки через необхідність ручної інтерпретації результатів.

Другий сегмент ринку представлений інтелектуальними інструментами, серед яких ключовим є GitHub Copilot. Цей сервіс функціонує на базі великих мовних моделей і глибоко інтегрується в інтегровані середовища розробки (IDE) для надання контекстних рекомендацій під час написання коду. GitHub Copilot демонструє високу ефективність як інструмент інтенсифікації праці програміста, допомагаючи усувати рутинні операції та генерувати шаблони функцій. Однак його архітектурна парадигма не адаптована до завдань автономного DevSecOps- конвеєра. По-перше, він працює у синхронному режимі «запит-відповідь», повністю залежачи від промпт-інжинірингу конкретного користувача, що виключає фактор автономності. По-друге, GitHub Copilot позбавлений системних засобів багаторівневої верифікації коду, через що запропоновані ним варіанти можуть містити приховані вразливості або синтаксичні дефекти, які не перевіряються в межах внутрішнього консенсусу.

Для систематизації отриманих даних та наочного представлення відмінностей між існуючим програмним забезпеченням та розроблюваною системою доцільно сформувати порівняльну матрицю за ключовими інженерними критеріями. Результати цього багатофакторного порівняльного аналізу наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз характеристик існуючих рішень та розроблюваної системи

Критерій порівняння	SonarQube та Snyk	GitHub Copilot	Розроблювана мультиагентна система
Рівень автономності системи	Низький – вимагає ручного керування та налаштування	Середній – функціонує синхронно за запитом користувача	Високий – автономний запуск та обробка через конвеєр агентів
Метод виявлення уразливостей	Сигнатурний аналіз та евристичні правила	Семантичний аналіз на базі одиночної LLM	Рольовий – мультиагентний аналіз Zero Trust аудит
Автоматичне усунення дефектів	Відсутнє – надаються лише текстові інструкції	Часткове – генерація фрагментів за запитом в IDE	Повне – автоматичний рефакторинг та створення патчів
Внутрішня верифікація результату	Відсутня	Відсутня – контроль покладається на розробника	Наявна – виділений агент QA Reviewer та збір метрик
Мовна універсальність	Наявна – через плагіни для різних мов	Наявна – на базі багатомовної навчальної вибірки	Наявна – динамічне розпізнавання розширення файлу

Аналіз сформованої таблиці дозволяє зробити висновок, що наявні на ринку рішення або мають високу точність сканування без можливості автокорекції, або пропонують генеративні функції без системного контролю якості та автономності. Запропонована у цій роботі інтелектуальна мультиагентна екосистема покликана ліквідувати цей технологічний розрив шляхом створення замкненого DevSecOps-циклу, що поєднує глибокий

аудит, автоматичний рефакторинг та сувору синтаксично-безпекову верифікацію.

1.4 Інноваційність та переваги розроблюваної системи

Наукова новизна та практична інноваційність розроблюваної системи полягають у принциповій зміні парадигми застосування генеративного штучного інтелекту в інженерії програмного забезпечення. Традиційні підходи автоматизації безпеки за допомогою штучного інтелекту наразі обмежуються монолітним та лінійним використанням великих мовних моделей у стандартному режимі «запит-відповідь».

Базова структура, а також ключові архітектурні обмеження такого одновекторного підходу схематично відображені на рисунку 1.4.

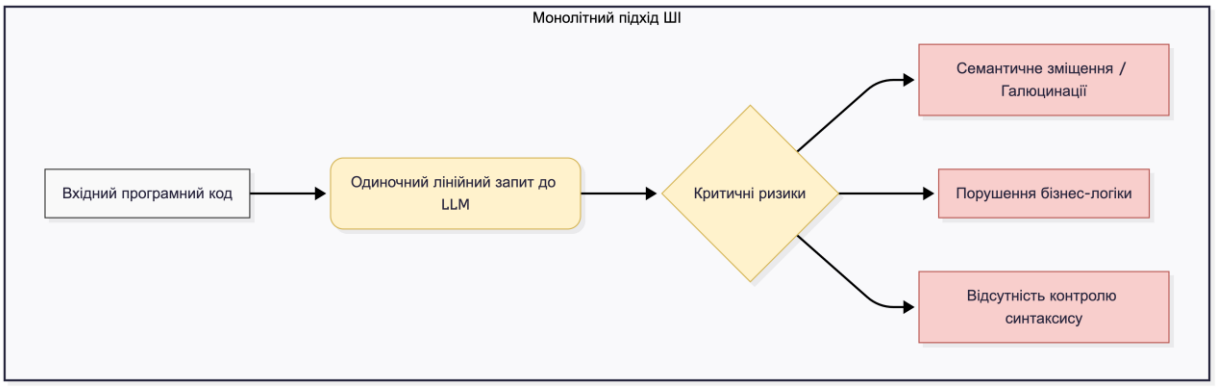


Рисунок 1.4 – Схема обмежень монолітного підходу

Як видно з наведеної схеми, пряме надсилання вихідного коду розробника в ізольовану мовну модель породжує критичні ризики семантичного зміщення, виникнення галюцинацій штучного інтелекту, а також неконтрольованого порушення початкової бізнес-логіки застосунку через відсутність внутрішніх інструментів тестування та перевірки синтаксису.

На противагу зазначеному монолітному підходу, у цій кваліфікаційній роботі запропоновано та науково обґрунтовано принципово нову концепцію автономної мультиагентної оркестрації, реалізовану на базі сучасного високорівневого фреймворку CrewAI. Дане інженерне рішення повністю трансформує процес лінійного аналізу тексту на динамічну, інтелектуальну симуляцію діяльності реального DevSecOps-підрозділу, що функціонує в режимі реального часу.

Інноваційне ядро пропонованої архітектури базується на практичному впровадженні фундаментального принципу кібербезпеки – розділення обов'язків [10], [14]. Організована трирівнева децентралізована структура, що безшовно об'єднує ролі старшого дослідника з безпеки «Security Auditor», головного інженера-програміста «Software Engineer» та провідного рев'юера контролю якості «QA Reviewer», формує стійкий замкнений цикл перехресної верифікації та валідації даних на кожному кроці обробки інформації.

У межах такої топологічної конфігурації повністю нівелюється деструктивний «фактор довіри» до результатів роботи генеративних нейромережевих моделей, який є головною проблемою сучасного програмування. Будь-які безпекові патчі та оптимізовані фрагменти, створені інженерним агентом, піддаються суворому, безальтернативному критичному аналізу з боку виділеного агента контролю якості.

Наявність постійно діючого внутрішнього циклу зворотного зв'язку (Feedback Loop) дозволяє системі функціонувати повністю автономно: у разі виявлення щонайменших архітектурних, синтаксичних або логічних дефектів, код автоматично та детерміновано повертається на попередній етап для повторного рефакторингу з генерацією нових інструкцій для доопрацювання.

Повна послідовність етапів функціонування розробленого цифрового конвеєра представлена на рисунку 1.5.

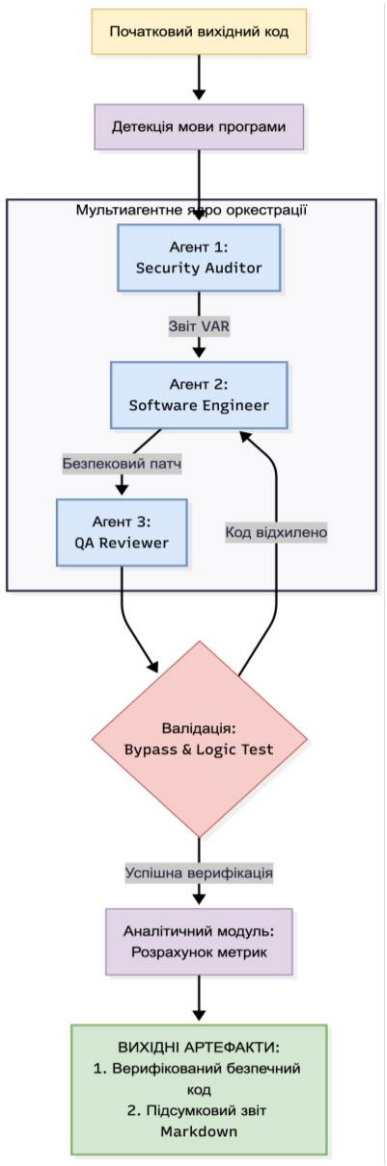


Рисунок 1.5 – Архітектура пропонованого мультиагентного DevSecOps конвеєра

Важливим технологічним проривом розроблюваного рішення є реалізація механізму мовної універсальності в автономному режимі, що відображено на початковому етапі конвеєра. Система спроможна самостійно здійснювати динамічну ідентифікацію мови програмування за розширенням цільового файлу та автоматично адаптувати промпт-контекст і бази знань усіх функціонуючих агентів під специфіку конкретного технологічного стеку, усуваючи потребу в попередньому ручному налаштуванні. Додатковою інноваційною ознакою є інтеграція у структуру

мультиагентного конвеєра аналітичного модуля збору кількісних наукових метрик. Система не просто модифікує код, а здійснює безперервний аудит власної обчислювальної ефективності, фіксуючи часові витрати та обсяги спожитих токенів мови. Автоматичне формування фінальних звітів у форматі Markdown із вбудованими аналітичними таблицями забезпечує безшовну інтеграцію розробленої екосистеми в сучасні DevSecOps- конвеєри як ефективного інструменту пре-комміт аудиту, мінімізуючи часові витрати та унеможливлюючи потрапляння уразливостей у промислові репозиторії.

1.5 Постановка задачі

Вектор досліджень у межах цієї кваліфікаційної роботи спрямований на подолання критичного технологічного розриву між високою швидкістю розгортання програмних продуктів та обмеженими можливостями сучасних систем контролю їхньої захищеності. Науково-практична задача проєкту полягає в концептуальному моделюванні, програмній реалізації та експериментальній верифікації автономного інструментального середовища, спроможного без залучення людського фактора здійснювати безперервний інтелектуальний супровід безпеки вихідного коду на ранніх етапах життєвого циклу розробки.

Реалізація наміченої мети передбачає послідовне виконання комплексу взаємопов'язаних науково-інженерних етапів, інтегрованих у єдину технологічну схему. Початкова стадія процесу зосереджена на створенні підсистеми адаптивного зчитування та лінгвістичного аналізу даних. На цьому етапі розробляються конвеєри безшовного імпорту вихідного коду як з локальних сховищ, так і через API розподілених систем контролю версій. Ключовим завданням тут є програмування модуля Polyglot Mode, що забезпечує миттєву детекцію типу мови програмування

за метаданими файлу для подальшої динамічної ін'єкції контекстних баз знань, специфічних для виявленого мовного стеку.

Наступний логічний етап полягає в алгоритмізації та архітектурному розгортанні інтелектуальної екосистеми оркестрації на базі фреймворку CrewAI. Ця фаза вимагає розробки топології взаємодії децентралізованих обчислювальних ШІ-агентів та проєктування їхніх поведінкових промпт профілів для формування стійких ролей та обмеження контекстних флуктуацій. Безпосередня інженерна збірка DevSecOps-пайплайну передбачає чітку координацію трьох компонентів: агента безпекового аудиту (Senior Security Auditor) для сканування семантики коду за ієрархією CWE/OWASP, інженерного агента (Principal Software Engineer) для прецизійної генерації контекстних латок та агента контролю якості (Lead QA Reviewer) для фінального синтаксичного аналізу. Важливою умовою проектування є інтеграція динамічного зворотного зв'язку (Feedback Loop) для організації ітераційного доопрацювання коду на основі внутрішнього консенсусу агентів.

Завершальна стадія роботи конвеєра реалізується через побудову аналітичного модуля та консолідацію фінальних результатів. Задача цього модуля полягає у фіксації кількісних параметрів обчислювальної ефективності, зокрема часових витрат та лімітів споживання токенів моделі. Кінцевим результатом цього етапу є автоматизація експорту технічної документації у форматі Markdown та структурування безпечного інкременту програмного коду, повністю готового до безшовного злиття з основним репозиторієм.

Критерії успішності та системні обмеження: розроблювана інформаційна система має демонструвати стабільну пропускну здатність при роботі з мультиваріантними мовними стеками; сумарний час обробки цільового файлу не повинен порушувати часові регламенти (time-outs) промислових CI/CD конвеєрів; архітектура рішення зобов'язана превентивно враховувати ліміти контекстного вікна базових

Transformer моделей та гарантувати повне збереження функціональної цілісності суміжних компонентів програмного забезпечення.

Таким чином, з огляду на виявлені технологічні виклики та актуальні потреби індустрії, підсумковий очікуваний результат цієї кваліфікаційної роботи полягає у концептуальному проєктуванні та практичній реалізації інноваційної ШІ-платформи закритого циклу. Передбачається, що ця система відзначатиметься високим рівнем масштабованості для гнучкої роботи з різними програмними проєктами, абсолютною прозорістю внутрішніх процесів прийняття рішень когнітивними агентами, а також гарантованою архітектурною стійкістю до зовнішніх та внутрішніх збоїв. Головне функціональне призначення розробленого інструментального середовища зводитиметься до забезпечення безперервного та превентивного аудиту безпеки на найбільш ранніх етапах, що органічно доповнюватиметься алгоритмами інтелектуального автоматичного рефакторингу вихідного коду без мануального втручання інженера. Усе це в комплексі дозволить повноцінно інтегрувати рішення у конвеєри розробки, повністю імплементуючи та підтримуючи основоположні принципи передової парадигми Shift-Left Security.

2 ОБГРУНТУВАННЯ НАПРЯМКУ ДОСЛІДЖЕННЯ ТА АРХІТЕКТУРНЕ МОДЕЛЮВАННЯ

2.1 Обґрунтування вибору технологічного стеку та інструментальних засобів

Ефективна програмна реалізація автономної мультиагентної DevSecOps-екосистеми для підтримки життєвого циклу програмного забезпечення критично залежить від збалансованого вибору базових технологій, інструментальних фреймворків та систем керування даними. Основним критерієм формування технологічного стеку для цієї кваліфікаційної роботи став пошук архітектурних рішень, що здатні забезпечити максимальну гнучкість, обчислювальну стійкість, швидкість обробки текстових масивів вихідного коду, а також повну незалежність від важковажких пропрієтарних платформ.

З огляду на зазначені вимоги, фундаментальною мовою програмування для проєктування та кодування системи було обрано Python. На сьогоднішній день дана мова є загальновизнаним індустріальним стандартом у сфері розробки систем штучного інтелекту, машинного навчання та обробки природної мови. Головна інженерна перевага Python полягає в наявності розвиненої екосистеми спеціалізованих бібліотек, що дозволяють маніпулювати структурами абстрактних синтаксичних дерев, здійснювати швидку токенізацію, взаємодіяти з веб-інфраструктурою та безшовно інтегруватися з інтерфейсами прикладного програмування мовних моделей. Інтерпретована природа мови у поєднанні з лаконічним синтаксисом суттєво прискорює процеси прототипування, зневадження та написання кастомних інструментів для взаємодії між інтелектуальними вузлами екосистеми, що робить її безальтернативним базисом для розроблюваного DevSecOps-конвеєра [18].

Для координації, синхронізації та організації спільної діяльності децентралізованих штучних інтелектуальних агентів було проведено порівняльний аналіз сучасних оркестраційних платформ, зокрема AutoGen від компанії Microsoft та LangGraph від екосистеми LangChain. Платформа AutoGen орієнтована на побудову неструктурованих, динамічних та хаотичних діалогів між агентами, де логіка передачі повідомлень визначається самими моделями в процесі генерації тексту. Такий підхід демонструє високу ефективність у творчих або відкритих дискусійних задачах, проте він є абсолютно неприйнятним для конвеєрів забезпечення безпеки, які вимагають суворої детермінованості, передбачуваності та чіткої послідовності дій. У свою чергу, фреймворк LangGraph надає розробнику можливість низькорівневого проєктування агентних графів у вигляді циклічних кінцевих автоматів. Незважаючи на високу гнучкість, LangGraph вимагає значних часових витрат на ручне кодування базових механізмів передачі контексту, управління станами та опрацювання помилок, що ускладнює масштабування системи при додаванні нових ролей.

На противагу зазначеним аналогам, у цій роботі було обґрунтовано та обрано оркестраційний фреймворк CrewAI. Ключова перевага даної платформи полягає в орієнтації на рольову структуру моделювання процесів, де взаємодія між агентами нагадує функціонування реального інженерного або безпекового підрозділу. CrewAI дозволяє будувати процедурні, послідовні та чітко регламентовані конвеєри завдань, що ідеально відповідає вимогам життєвого циклу розробки програмного забезпечення [20]. У межах парадигми DevSecOps сувора ієрархія, за якої звіт агента-аудитора безпеки безальтернативно передається агенту-інженеру, а згенерований патч обов'язково потрапляє до агента контролю якості, є критично важливою умовою стабільності. Перевагою архітектури CrewAI є використання вбудованого механізму послідовних процесів, який дозволяє чітко задавати лінійний алгоритм виконання та оркеструвати

логіку взаємодії об'єктів агентів та їхніх завдань. Це забезпечує автоматичне керування контекстним вікном мовних моделей, взаємоізоляцію пам'яті та оптимізацію споживання обчислювальних ресурсів, що дозволяє зосередитися на бізнес-логіці рефакторингу коду без перевантаження системи низькорівневими викликами суміжних потоків.

Критично важливим аспектом формування інструментального забезпечення екосистеми став вибір базової великої мовної моделі та інфраструктури для її запуску, що виступає генеративним та аналітичним ядром для всіх функціонуючих агентів. Замість використання стандартної та фінансово витратної інтеграції пропрієтарних моделей, у межах цього проєкту було обґрунтовано застосування спеціалізованого хмарного інструментарію інференсу ChatGroq. Інтеграція обчислювальної інфраструктури провайдера Groq дозволяє підключати передові відкриті великі мовні моделі трансформерної архітектури та виконувати надшвидку генерацію відповідей на базі спеціалізованих мікропроцесорів Language Processing Unit. Це забезпечує експоненційне зростання швидкості генерації токенів порівняно з класичними графічними серверами, що є критично важливим для уникнення затримок та розриву з'єднань при роботі з довгими лістингами вихідного коду всередині реальних процесів безперервної інтеграції та розгортання [18].

Для створення цілісного, безпечного та метрично вимірюваного конвеєра, базовий технологічний стек було розширено набором допоміжних системних утиліт та модулів. Підсистема ізоляції конфігурацій dotenv забезпечує безпечне підтягування параметрів оточення, виключаючи жорстке кодування конфіденційних токенів доступу та ключів у вихідних файлах системи, що повністю відповідає кращим інженерним практикам безпечної розробки. Модуль мережевої взаємодії requests відповідає за реалізацію безшовного транспорту даних та надсилання запитів до зовнішніх інтерфейсів, гарантуючи надійну інтеграцію мультиагентного

ядра з віддаленими хмарними репозиторіями та розподіленими системами контролю версій.

Разом з цим, модуль синтаксичного аналізу лінків urlparse здійснює декомпозицію та валідацію вхідних уніфікованих покажчиків ресурсів для точної локалізації цільових сховищ коду та превентивного запобігання атакам ін'єкцій на рівні мережевих запитів конвеєра. Нарешті, підсистема часового моніторингу та логування, представлена модулями time та datetime, реалізує механізми прецизійного вимірювання тривалості кожної ітерації безпекового аудиту, фіксації точних міток часу створення артефактів та збору кількісних метрик обчислювальної ефективності функціонування системи в дистанційному режимі. Таким чином, обрана комбінація технологій у складі мови Python, оркестратора CrewAI, обчислювального інференс-ядра ChatGroq та супутніх мережевих системних утиліт формує архітектурно стійкий, автономний та високопродуктивний технологічний стек, спроможний ефективно вирішувати задачі автоматизованого пре-комміт аудиту та безпечного рефакторингу програмного забезпечення.

2.2 Аналіз архітектур мультиагентних систем

Сучасний етап розвитку генеративного штучного інтелекту демонструє поступовий перехід від застосування поодиноких мовних моделей до проєктування складних децентралізованих мультиагентних екосистем. Практичний досвід впровадження ізольованих LLM у процеси життєвого циклу розробки програмного забезпечення виявив їхню фундаментальну нездатність самостійно та безпомилково вирішувати комплексні інженерні задачі. Основними перешкодами в таких випадках стають прогресуюча деградація уваги моделі при обробці об’ємних масивів вихідного коду, висока ймовірність виникнення семантичних галюцинацій та цілковита відсутність вбудованих механізмів перевірки згенерованого

результату. Для подолання цих технологічних бар'єрів у даній роботі застосовано мультиагентну парадигму, яка імітує структуру та принципи взаємодії реального інженерного підрозділу шляхом декомпозиції глобальної проблеми на ізольовані підзадачі, що виконуються вузькоспеціалізованими автономними алгоритмами.

Під час проєктування архітектури системи було проведено глибокий критичний аналіз існуючих фреймворків оркестрації, які наразі домінують у сфері розробки прикладного штучного інтелекту. Розгляд популярної бібліотеки LangChain та її графового розширення LangGraph показав, що цей інструментарій орієнтований переважно на низькорівневе конструювання ланцюжків обробки природної мови. Його використання для побудови жорсткого автоматизованого конвеєра вимагає невиправдано великих трудовитрат на написання рутинної програмної логіки для керування станами та маршрутизації повідомлень. Натомість інфраструктура Microsoft AutoGen, яка також розглядалася як потенційна технологічна база, концептуально побудована навколо створення динамічних розмовних мереж. Експериментальне моделювання виявило, що такий підхід є вкрай нестабільним для завдань аудиту коду. Агенти в екосистемі AutoGen схильні вступати у неконтрольовані дискусійні цикли, що майже миттєво призводить до вичерпання квот на використання токенів та аварійного завершення процесів через перевищення таймаутів [3], [17].

Для систематизації архітектурних особливостей розглянутих інструментальних середовищ було сформовано порівняльну матрицю за ключовими критеріями, які є критичними для конвеєрів DevSecOps (таблиця 2.1). Результати цього зіставлення дозволяють об'єктивно оцінити придатність кожної з платформ для побудови надійних і детермінованих процесів автоматизованої безпеки. Саме аналіз цих критичних параметрів слугує беззаперечним інженерним обґрунтуванням вибору базового фреймворку для подальшої програмної реалізації екосистеми.

Таблиця 2.1 – Порівняльний аналіз фреймворків для побудови мультиагентних систем

Критерій порівняння	LangChain / LangGraph	Microsoft AutoGen	CrewAI (Обране рішення)
Базова парадигма	Ланцюжки обробки та графи	Динамічні розмовні мережі	Рольова оркестрація
Топологія виконання	Потребує ручного керування станами	Хаотична (ризик виникнення conversational loops)	Строга, детермінована (Sequential/Hierarchical)
Складність розробки	Висока (низькорівневе програмування)	Середня	Низька (високорівневі абстракції)
Інтеграція зовнішніх інструментів	Підтримується (складна імплементація)	Підтримується	Нативна підтримка з ізоляцією контексту
Придатність для DevSecOps конвеєра	Низька	Середня (проблема з таймаутами)	Висока

Враховуючи виявлені архітектурні недоліки альтернативних рішень, концептуальним ядром розроблюваної системи було обрано фреймворк CrewAI. Ця платформа відрізняється зміщеним фокусом на рольове проєктування автономних вузлів, що ідеально відповідає фундаментальному принципу інформаційної безпеки щодо суворого розділення повноважень. Вибраний інструментарій надає змогу інкапсулювати специфічну логіку, професійні цільові настанови та передісторію в межах окремих ізольованих цифрових персон. Завдяки цьому в розробленій екосистемі вдалося досягти чіткого розмежування зон відповідальності. Наприклад, агент-аудитор займається виключно пошуком та ідентифікацією вразливостей згідно з класифікаціями CWE, інженерний агент генерує безпечні патчі без деструктивного втручання в бізнес-логіку,

а валідатор забезпечує неупереджений контроль якості. Подібна ізоляція контекстів критично знижує когнітивне перевантаження на базову мовну модель, дозволяючи їй фокусуватися виключно на поточній операції та видавати значно точніший код.

Визначальним фактором на користь обраного фреймворку також стала вбудована підтримка жорсткої детермінованої топології виконання процесів. На відміну від імовірнісної природи більшості чат-ботів, розроблена екосистема функціонує за принципом послідовної оркестрації, повністю імітуючи реальні етапи конвеєра безперервної розробки. Інформаційні артефакти, створені на попередньому кроці, автоматично формують чітко структурований вхідний масив даних для наступного агента, що гарантує лінійність і високу передбачуваність результату.

Порівняння топологій взаємодії ШІ-агентів у різних архітектурних парадигмах продемонстровано на рисунку 2.1.

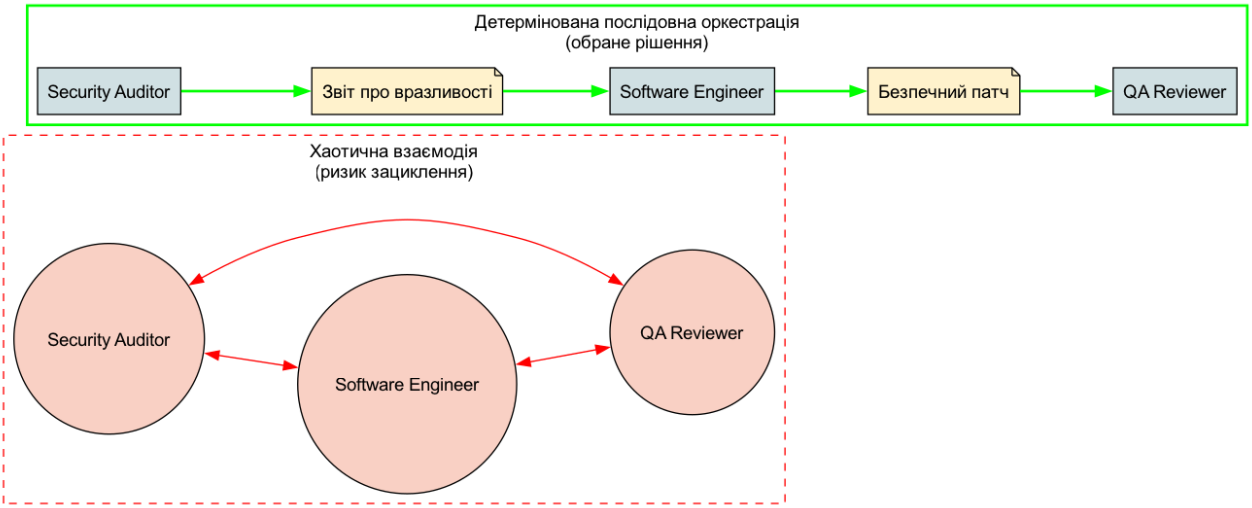


Рисунок 2.1 – Порівняння топологій взаємодії ШІ-агентів у різних архітектурних парадигмах

Важливою архітектурною перевагою системи є також механізм безшовної інтеграції зовнішніх програмних інструментів у когнітивний

цикл штучного інтелекту. Це дозволило наділити агента контролю якості здатністю безпосередньо та асинхронно викликати локальні скрипти, зокрема інструменти синтаксичного аналізу. Завдяки цій інтеграції розроблена система приймає рішення про схвалення або відхилення рефакторингу не на основі лінгвістичних суджень мовної моделі, а виключно спираючись на детерміновані показники успішного проходження синтаксичних тестів. Разом із вбудованими механізмами контролю частоти запитів це забезпечує високу обчислювальну стійкість екосистеми та робить її придатною для розгортання в умовах реальних інфраструктурних обмежень.

2.3 Огляд методів оркестрації агентів

Ефективність функціонування будь-якої мультиагентної системи визначається не лише інтелектуальними спроможностями окремих базових моделей, але й обраною парадигмою їхньої оркестрації. Під оркестрацією в контексті МАС розуміють алгоритмічний підхід до управління життєвим циклом завдань, маршрутизації інформаційних потоків між автономними вузлами та вирішення конфліктів у процесі досягнення спільної мети. Сучасна інженерія штучного інтелекту виділяє кілька базових топологій оркестрації: ієрархічну (з виділеним агентом-менеджером), консенсусну (базовану на вільному обміні повідомленнями) та послідовну. Зважаючи на високу ціну помилки у сфері кібербезпеки та необхідність суворого дотримання етапів життєвого циклу розробки програмного забезпечення, для розробленої екосистеми було обрано парадигму детермінованої послідовної оркестрації.

Використання послідовної моделі взаємодії гарантує, що інформаційні артефакти передаються між агентами у строго визначеному порядку без ризику інформаційної ентропії. У розробленому програмному рішенні процес ініціюється завданням аудиту task_audit, результати якого,

включаючи ідентифіковані CWE-вразливості та архітектурний контекст, формують замкнений вхідний вектор для завдання рефакторингу task_patch. Після генерації безпечного коду агентом-інженером, оркестратор блокує подальше просування пайплайну та передає згенерований артефакт на фінальний етап перевірки task_qa_review. Такий лінійний підхід виключає можливість пропуску критичних стадій верифікації та імітує класичний конвеєр безперервної інтеграції CI/CD. Важливою умовою стабільності цієї топології стала відмова від використання глобальної пам'яті агентів (memory=False), що дозволило уникнути переповнення контекстного вікна мовної моделі нерелевантними даними з попередніх ітерацій.

Найбільш значущою архітектурною інновацією розробленого механізму оркестрації є впровадження динамічного циклу зворотного зв'язку Feedback Loop, який активується у випадку провалу верифікації коду. Замість того, щоб аварійно завершувати роботу при виявленні синтаксичних помилок зовнішнім лінтером, програмний оркестратор перехоплює статус виконання та ініціює режим корекції is_correction_mode. У цьому стані система тимчасово реорганізує агентну мережу, створюючи ізольоване середовище для інженерного агента із згенерованим новим завданням correction_task, яке містить точний вивід traceback компілятора чи лінтера. Це дозволяє системі самостійно «усвідомлювати» власні помилки та проводити ітеративний рефакторинг до моменту досягнення нульового рівня синтаксичних дефектів або вичерпання ліміту спроб max_retries. Графічне подання розробленого алгоритму оркестрації та механізму обробки помилок наведено на рисунку 2.2. Така візуалізація чітко відображає логіку автономного прийняття рішень усередині конвеєра та його унікальну здатність до самовідновлення після виявлених збоїв. Разом із тим система надійно захищена від нескінченних циклів доопрацювання, що гарантує стабільність DevSecOps-процесів та запобігає неконтрольованій перевитраті обчислювальних квот мовної моделі.

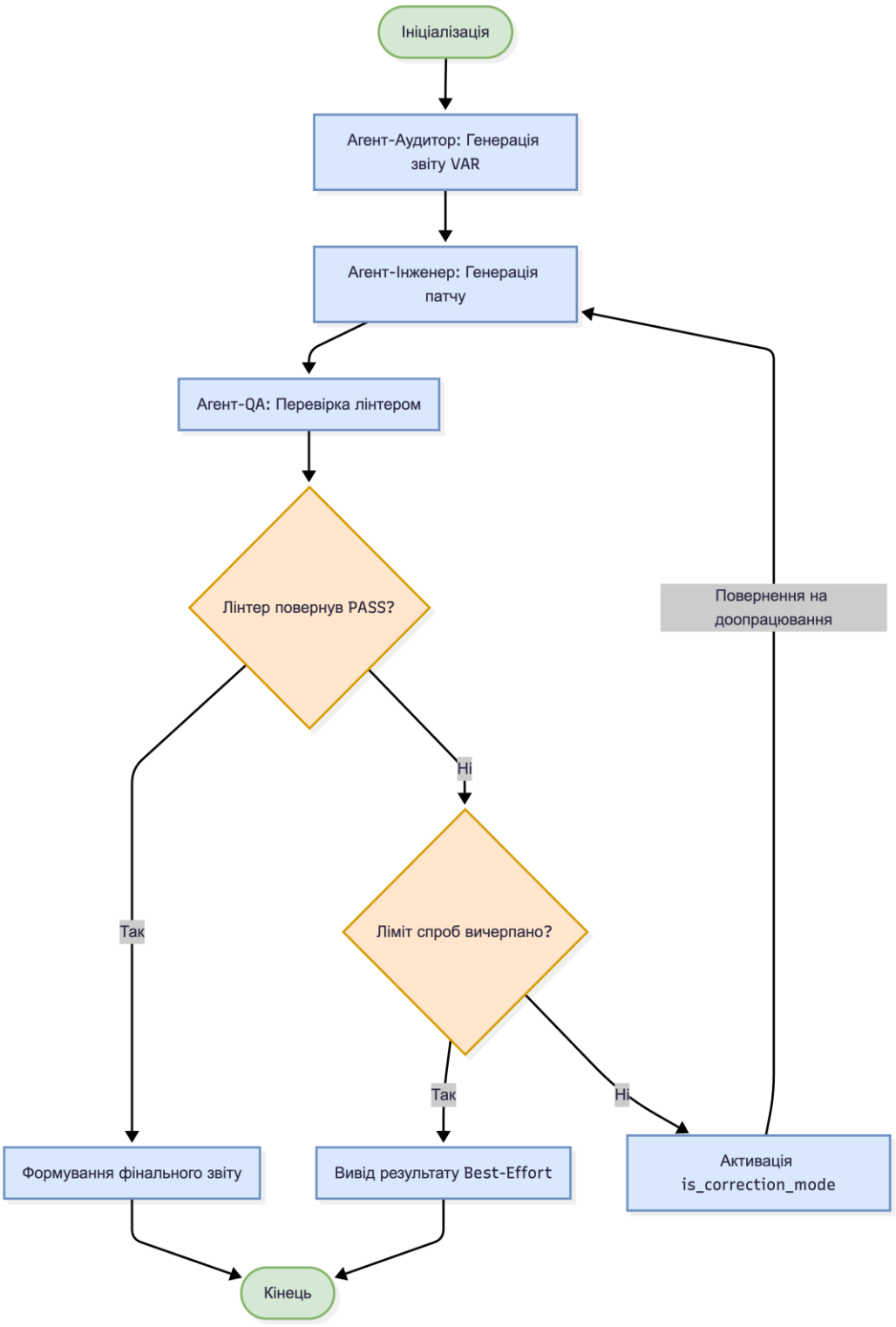


Рисунок 2.2 – Блок-схема алгоритму динамічної оркестрації та циклу зворотного зв'язку (Feedback Loop)

Окрім управління логікою виконання, реалізований метод оркестрації бере на себе критично важливу функцію управління обчислювальними

ресурсами та взаємодією з API-шлюзами. Враховуючи суворі обмеження пропускної здатності хмарних інфраструктур на кількість оброблених токенів за хвилину (TPM), оркестратор реалізує алгоритми превентивного тротлінгу (Max RPM) та механізми примусового охолодження (cooldown_callback) між ітераціями. Більше того, розроблений механізм містить вбудований аналізатор помилок мережевого рівня, який за допомогою регулярних виразів автоматично зчитує параметри Rate Limit, динамічно розраховує необхідний час очікування та призупиняє виконання конвеєра, запобігаючи втраті згенерованих даних. Таким чином, розроблений рівень оркестрації виконує роль не лише диспетчера завдань, але й інтелектуального балансувальника навантаження, забезпечуючи високу відмовостійкість усієї DevSecOps-екосистеми.

2.4 Обґрунтування вибору LLM

Інтелектуальним та когнітивним ядром розробленої мультиагентної екосистеми виступають великі мовні моделі (LLM), які забезпечують здатність агентів до семантичного аналізу вихідного коду, виявлення вразливостей та генерації безпечних патчів. Вибір оптимальної базової моделі для конвеєра DevSecOps є багатофакторною задачею, оскільки до неї висуваються специфічні інженерні вимоги: глибоке розуміння синтаксису мов програмування, здатність утримувати значний обсяг контексту, суворе дотримання інструкцій системного промпту та висока швидкість висновування.

Під час проєктування системи було проведено порівняльний аналіз двох глобальних підходів: використання пропрієтарних комерційних моделей (таких як родина OpenAI GPT-4 або Anthropic Claude) та застосування відкритих моделей з доступною вагою Open-Weights. Незважаючи на високі показники точності комерційних рішень, їхня інтеграція в конвеєри безпеки програмного забезпечення пов'язана з

критичними ризиками конфіденційності. Передача непублічного вихідного коду на сервери сторонніх організацій для аналізу часто прямо суперечить корпоративним політикам безпеки та стандартам захисту комерційної таємниці. З огляду на це, концептуальним вибором для розроблюваної системи стала орієнтація на відкриті мовні моделі, які дозволяють повністю контролювати потік даних та можуть бути розгорнуті в ізольованих on premise контурах. Результати порівняльного аналізу архітектурних підходів до вибору LLM для задач корпоративної кібербезпеки наведено у таблиці 2.2.

Таблиця 2.2 – Порівняльний аналіз парадигм використання LLM у DevSecOps

Критерій порівняння	Пропрієтарні комерційні моделі (напр., GPT-4, Claude)	Відкриті моделі / Open-Weights (напр., Llama 3)
Конфіденційність коду (Privacy)	Низька (передача коду на сервери провайдера)	Висока (можливість локального розгортання)
Ризик витоку комерційної таємниці	Присутній (дані можуть використовуватись для донавчання)	Відсутній
Швидкість інференсу (Latency)	Середня (залежить від навантаження на API)	Висока (при використанні оптимізованого LPU-обладнання)
Контроль над генерацією	Обмежений системними промптами провайдера	Повний контроль
Економічна доцільність (Cost)	Оплата за кожен згенерований токен	Безкоштовне використання ваг моделі

За результатами аналізу продуктивності відкритих моделей, базовим інтелектуальним рушієм екосистеми було обрано архітектуру моделі Meta Llama 3.1 зокрема, версію з 8 мільярдами параметрів [19]. Дана модель демонструє оптимальний баланс між обчислювальною ефективністю та здатністю до складного логічного висновування. Об'єму її контекстного вікна абсолютно достатньо для обробки фрагментів коду разом із

детальними рольовими інструкціями агентів. Крім того, Llama 3.1 8B відрізняється високою дисципліною слідування промптам, що є критично важливим для обраного фреймворку CrewAI, де кожен агент повинен суворо залишатися в межах своєї ролі (аудитора, інженера чи валідатора) і не генерувати надлишкового або деструктивного тексту.

Окремим архітектурним аспектом, що визначив вибір саме 8 мільярдної версії моделі Llama 3.1 (llama-3.1-8b-instant), є задача оптимізації споживання контекстного вікна та збереження високої швидкодії в умовах ітеративних конвеєрів. Для ефективного аудиту програмного коду критично важливо передавати мовній моделі не лише цільовий скрипт, але й об'ємні інструкції передісторії, які регламентують стандарти OWASP, правила валідації та обмеження на галюцинації. Базова архітектура Llama 3.1 8B підтримує розширене контекстне вікно до 128 тисяч токенів, що дозволяє обробляти об'ємні вхідні дані без деструктивної фрагментації чи втрати семантичного зв'язку [19]. Експериментальні тестування показали, що залучення більш масивних моделей (наприклад, версії з 70 млрд параметрів), хоча і забезпечує дещо вищий рівень абстрактного мислення, призводить до експоненційного зростання часу відгуку (latency) та миттєвого вичерпання квот хмарного провайдера під час багаторазових циклів перехресного рефакторингу. Таким чином, конфігурація з 8 мільярдами параметрів є оптимальним інженерним компромісом, що забезпечує достатній рівень інтелектуальної автономності агентів при збереженні безперервної динаміки роботи DevSecOps-системи.

Особливістю мультиагентних систем із послідовною топологією оркестрації є те, що час виконання всього конвеєра дорівнює сумі часу роботи кожного окремого агента. Використання традиційних графічних процесорів для інференсу мовних моделей у такому ланцюжку генерує значну затримку (латентність), що робить неможливим безшовне впровадження системи у швидкісні CI/CD конвеєри. Для вирішення

проблеми «вузького місця» продуктивності, як середовище виконання моделі було обрано спеціалізовану хмарну інфраструктуру Groq. На відміну від класичних відеокарт, обчислювальна архітектура Groq базується на тензорних процесорах LPU (Language Processing Units), які фізично оптимізовані для детермінованої потокової обробки мовних послідовностей [21].

Використання API-шлюзу Groq дозволило досягти безпрецедентної швидкості генерації коду, яка перевищує показник у 100 токенів на секунду. Це гарантує, що повний цикл безпекового пре-комміт аудиту, який включає роботу трьох автономних агентів, завершується в межах кількох хвилин і не порушує таймаути промислових серверів розгортання. Водночас, використання таких надшвидкісних інструментів вимагає розв'язання проблеми інфраструктурних лімітів.

Використання безкоштовних інстансів платформи Groq супроводжується жорсткими квотами на кількість запитів (рис. 2.3). Для забезпечення стабільної роботи конвеєра при обробці об'ємних файлів в умовах таких обмежень, у програмному коді екосистеми було імплементовано інтелектуальні механізми тротлінгу та функції примусового паузування.

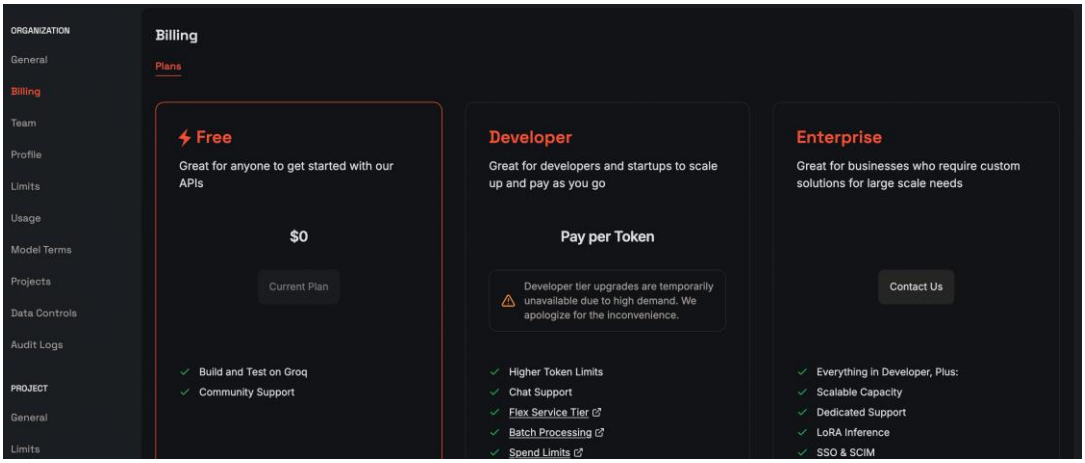


Рисунок 2.3 – Обмеження пропускної здатності (Rate Limits) у середовищі хмарного виконання Groq

Як показано на загальній архітектурній схемі (рисунок 2.4), процес інформаційної взаємодії логічно розділено на два ізольовані контури: локальне середовище виконання (безпосередньо DevSecOps-конвеєр) та віддалену хмарну інфраструктуру Groq.

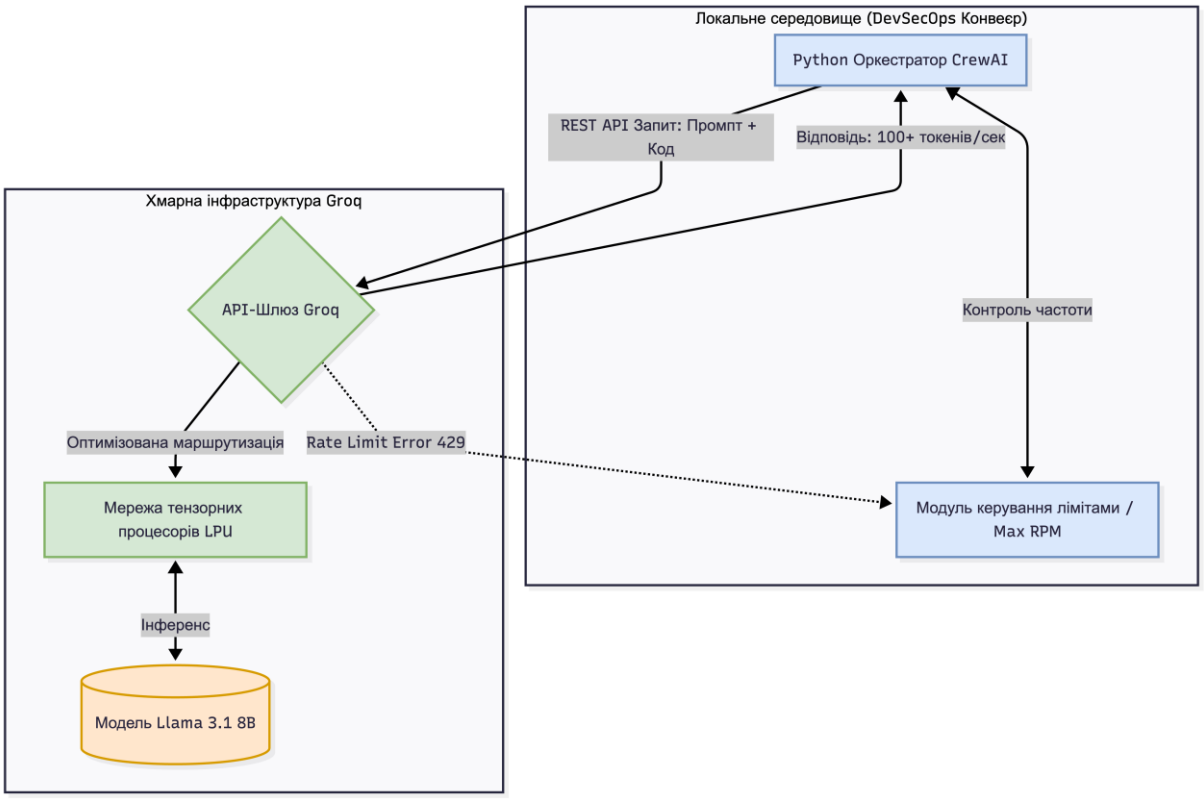


Рисунок 2.4 – Схема інформаційної взаємодії мультиагентного ядра з хмарним LPU-середовищем

У локальному контурі функціонує програмний оркестратор на базі фреймворку CrewAI, який агрегує вихідний код, формує системні промпти та передає їх до спеціалізованого модуля керування лімітами (Max RPM). Сформовані транзакційні пакети передаються через протокол REST API до хмарного шлюзу провайдера.

3 РОЗРОБКА ІНТЕЛЕКТУАЛЬНОЇ МУЛЬТИАГЕНТНОЇ ЕКОСИСТЕМИ

3.1 Проєктування загальної архітектури та рольової моделі системи

Фундаментальним завданням під час розробки інтелектуальної мультиагентної екосистеми було створення технічної архітектури, яка б гарантувала детермінованість результатів аудиту та безшовну інтеграцію між програмними модулями. Інженерна архітектура розробленого програмного комплексу та схема взаємодії його базових компонентів наведена на рисунку 3.1.

Інформаційний конвеєр ініціюється модулем препроцесингу вхідних артефактів. Оскільки сучасні проєкти використовують гетерогенні технологічні стеки, першим архітектурним вузлом системи є аналізатор розширень файлів. Цей компонент за допомогою стандартних бібліотек операційної системи програмно перехоплює шлях до цільового файлу, динамічно визначає мову програмування та автоматично ін'єктує цю інформацію у глобальний контекст оркестратора.

Такий підхід дозволяє використовувати єдиний пайплайн для обробки скриптів Python, JavaScript чи конфігурацій YAML, позбавляючи розробників необхідності ручного перемикання профілів системи. Завдяки такій динамічній адаптації, система здатна миттєво конфігурувати середовище виконання та підвантажувати відповідні словники правил безпеки для кожного конкретного об'єкта аналізу.

Подібна архітектурна гнучкість суттєво підвищує загальну пропускну здатність конвеєра, мінімізуючи часові затримки на етапі підготовки інструментарію до безпосереднього проведення аудиту.

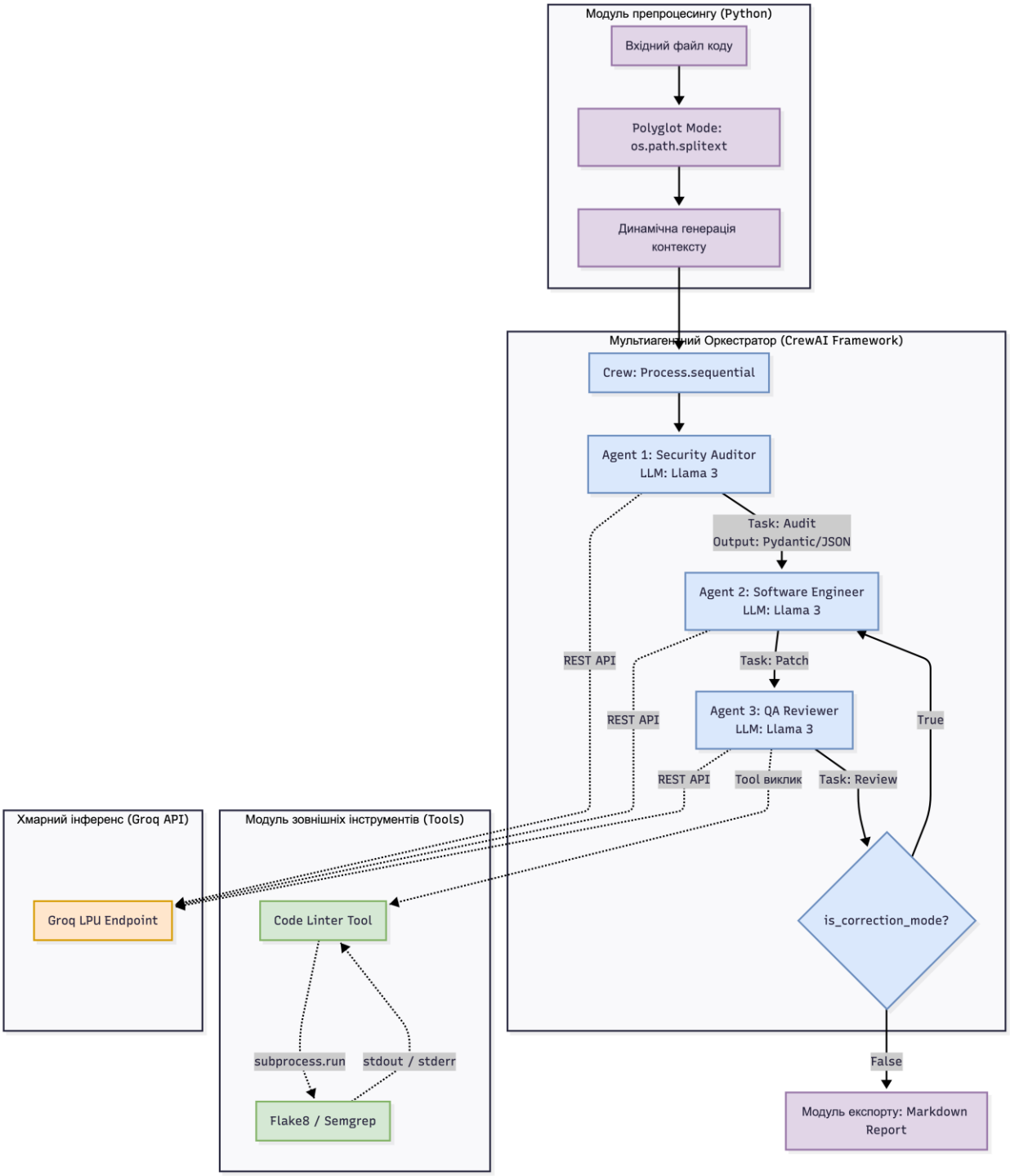


Рисунок 3.1 – Інженерна архітектура програмного комплексу та взаємодія базових модулів

Центральним ядром екосистеми виступає рольова модель, побудована на базі фреймворку CrewAI. З метою подолання головного недоліку одиночних LLM – когнітивного перевантаження під час виконання

комплексних завдань – архітектуру системи було декомпозовано на три ізольовані цифрові персони (агенти). Кожен агент функціонує в межах жорстко обмеженого інформаційного простору, виконуючи виключно одну фазу DevSecOps-конвеєра. Під час практичної реалізації та тестування було виявлено, що використання спільної системної пам'яті агентів призводить до прогресуючої втрати фокусу при обробці файлів обсягом понад 500 рядків коду. Для розв'язання цієї проблеми в архітектурі оркестратора було примусово деактивовано модулі спільної та короткострокової пам'яті (memory=False), що гарантувало ізоляцію ітерацій та запобігло переповненню контекстного вікна мовної моделі.

Першим вузлом у конвеєрі взаємодії виступає агент-аудитор (Senior Security Researcher). Його архітектурне призначення полягає у проведенні глибокого статичного та семантичного аналізу сирцевого коду на предмет наявності вразливостей згідно з класифікаціями OWASP та CWE. На вхід цього агента надходить виключно оригінальний код без додаткових інструкцій щодо його модифікації. Завдання аудитора зводиться до формування стандартизованого аналітичного артефакту – звіту про оцінку вразливостей (VAR). Строге обмеження відповідальності аудитора лише функцією пошуку дозволяє максимізувати його аналітичні здібності, уникаючи спроб передчасного та помилкового рефакторингу складних алгоритмічних блоків.

Сформований звіт VAR разом із вихідним кодом автоматично маршрутизується до другого вузла – інженерного агента (Principal Software Engineer). На етапі проєктування цього вузла було враховано критичний нюанс генеративного ШІ: схильність до деструктивного спрощення коду та виклику неіснуючих бібліотек, тобто галюцинацій. Тому рольова модель інженера архітектурно обмежена імперативною вимогою: генерувати безпечний патч без жодного втручання у початкову бізнес-логіку застосунку та без зміни експортованих сигнатур функцій. Інженерний агент не аналізує весь спектр загроз, а працює

виключно з локалізованими проблемами, вказаними у звіті аудитора, що в рази підвищує точність застосованих виправлень.

Фінальним вузлом мультиагентного ядра є агент контролю якості (Lead QA & Security Code Reviewer). Його впровадження в архітектуру системи стало ключовим інноваційним рішенням для забезпечення відмовостійкості. QA-агент отримує на вхід згенерований патч та виконує семантичне порівняння модифікованого коду з оригіналом. Роль цього агента полягає у виявленні логічних регресій, перевірці відсутності захардкоджених токенів чи паролів у згенерованому коді, а також у підготовці синтаксично чистого артефакту для подальшої програмної валідації зовнішніми інструментами (лінтерами). Завдяки такій трирівневій архітектурі з чітким розподілом обов'язків, розроблена система працює як єдиний, надійно синхронізований конвеєр, що мінімізує вплив людського фактора на етапах перевірки безпеки програмного забезпечення.

Для систематизації архітектурних рішень щодо розподілу обов'язків, функціональні ролі та інформаційні артефакти агентів зведено у таблицю 3.1.

Таблиця 3.1 – Функціональний розподіл ролей інтелектуальних агентів екосистеми

Роль агента (Persona)	Вхідний контекст	Основна ціль (Goal)	Вихідний артефакт (Output)	Механізм ізоляції
Security Auditor	Оригінальний вихідний код	Пошук вразливостей (CWE / OWASP)	Структурований звіт про вразливості (VAR)	Заборона на модифікацію коду
Software Engineer	Оригінальний код + звіт VAR	Написання безпечного патчу	Модифікований програмний код	Заборона на зміну бізнес-логіки
QA Reviewer	Модифікований код + звіт VAR	Синтаксична та логічна верифікація	Фінальний верифікований інкремент коду	Використання зовнішніх лінтерів

Окремим критично важливим аспектом мультиагентної оркестрації, реалізованим у розробленій архітектурі, є стандартизація протоколу обміну даними між ізольованими вузлами. Оскільки кожна велика мовна модель генерує текст у довільній природній формі, пряма передача неструктурованого виводу від агента-аудитора до інженера створювала високий ризик втрати критичних фрагментів коду або неправильної інтерпретації знайдених вразливостей. Для розв'язання цієї інженерної проблеми взаємодію агентів було переведено на використання механізмів структурованого виводу [26]. Завдяки внутрішнім можливостям фреймворку CrewAI, результати роботи кожного агента примусово серіалізуються у чітко визначені структури даних (наприклад, JSON-об'єкти або моделі Pydantic). Таким чином, звіт про вразливість (VAR) передається наступному агенту не як суцільний текст, а як машиночитний словник із фіксованими ключами: line_number, vulnerability_type, cwe_id та severity_level. Це гарантує детермінованість вхідних даних для інженерного агента і виключає ймовірність того, що він «не помітить» частину інструкцій аудитора.

Схема серіалізації та структурованого інформаційного обміну між вузлами конвеєра наведена на рисунку 3.2.

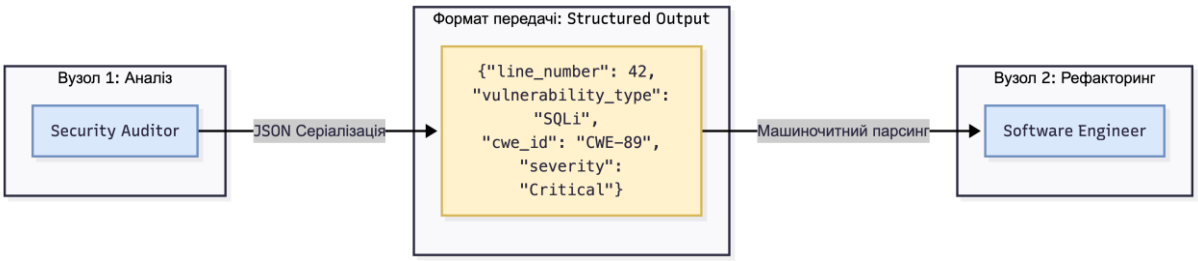


Рисунок 3.2 – Схема структурованого інформаційного обміну між інтелектуальними вузлами системи

Крім того, запропонована рольова модель успішно вирішує проблему так званого «дрейфу персони» – явища, при якому мовна модель під

впливом великого обсягу вхідного коду поступово втрачає свої початкові системні інструкції та починає генерувати нерелевантний або загальний текст. Завдяки ізоляції контекстів у розробленій архітектурі кожен вузол (Auditor, Engineer, QA) щоразу ініціалізується з «чистого аркуша», маючи у швидкій пам'яті лише свій вузькоспеціалізований промпт (Backstory) та конкретне поточне завдання. Така архітектурна парадигма не лише оптимізує споживання токенів хмарного провайдера Groq, але й забезпечує стабільно високу точність рефакторингу незалежно від складності або обсягу початкового вихідного файлу, що обробляється у DevSecOps-конвеєрі.

3.2 Реалізація когнітивних агентів та управління системними промптами

Основою інтелектуальної автономності та професійної спеціалізації агентів у фреймворку CrewAI є застосування передових практик інженерії підказок [23]. На відміну від класичних чат-ботів, управління поведінкою когнітивних вузлів у розробленій системі реалізується через сувору трикомпонентну конфігурацію: Role (професійна роль), Goal (конкретна мета ітерації) та Backstory (передісторія, що формує семантичний контекст моделі). Правильне налаштування цих параметрів стало критичним етапом розробки, оскільки саме від них залежить точність роботи великої мовної моделі (Llama 3.1 8B) та її здатність залишатися в межах заданого контексту.

Перший вузол екосистеми – Security Auditor – ініціалізується з роллю старшого дослідника з кібербезпеки (Senior Security Researcher). Його системний промпт налаштований на глибоке розуміння стандартів безпеки та ієрархії вразливостей CWE. Головним технічним нюансом при реалізації цього агента стала необхідність апаратного блокування його спроб самостійно виправляти знайдений код. Під час початкового тестування було

виявлено, що модель намагалася одночасно знайти вразливість і згенерувати патч у межах одного звіту, що призводило до інформаційного перевантаження та втрати формату JSON.

Для розв'язання цієї проблеми в системний промпт аудитора було впроваджено жорсткий імперативний обмежувач: агент має виключно діагностувати проблему, локалізувати рядок коду та класифікувати ризик (Critical/Medium/Low), залишаючи рефакторинг наступному вузлу конвеєра.

Найбільшим інженерним викликом стала реалізація другого вузла – Software Engineer (Principal Software Engineer). На етапі експериментального тестування було зафіксовано критичну проблему деструктивних семантичних галюцинацій мовної моделі. Намагаючись виправити прості вразливості в консольних утилітах (CLI), модель Llama 3 самовільно імпортувала надлишкові веб-фреймворки (наприклад, Flask) або вигадувала неіснуючі бібліотеки (на кшталт botocase замість стандартного boto3). Такі архітектурні викривлення повністю ламали початкову бізнес-логіку застосунків (приклад деструктивної галюцинації базової мовної моделі під час тестування наведено у додатку Б).

Для усунення ризику галюцинацій та функціональної регресії, у когнітивний профіль агента-інженера було інтегровано інноваційний механізм – «Якір бізнес-логіки». Цей динамічний блок системного промпту примусово фіксує початкове призначення скрипта. Додатково в backstory інженера було прописано суворі правила «Security Fix Guidelines»:

– заборона на модифікацію бізнес-логіки та зміну сигнатур існуючих функцій;

– пріоритетне використання вбудованих модулів (наприклад, hashlib, sqlite3, os) замість імпорту зовнішніх залежностей;

– використання виключно параметризованих запитів для запобігання SQL-ін'єкціям (CWE-89) замість екранування рядків.

Фінальний когнітивний вузол – QA Reviewer (Lead QA & Security Code Reviewer) – виконує роль безальтернативного валідатора.

Його системний промпт орієнтований на скептичний аналіз результатів роботи інженера. Передісторія цього агента вимагає від нього діяти як «цифровий супротивник» (Red Teamer), шукаючи шляхи обходу запропонованого патчу. Саме цей агент має делеговані права на активацію циклу зворотного зв'язку (is_correction_mode), якщо згенерований інженером код не відповідає стандартам безпеки або містить захардкоджені секрети.

Для систематизації розроблених когнітивних профілів ключові обмеження та інструкції агентів зведено у порівняльну таблицю 3.2.

Таблиця 3.2 – Структура системних промптів та когнітивних обмежень ШІ-агентів

Агент (Role)	Фрагмент передісторії (Backstory)	Технічні та логічні обмеження у промпті (Constraints)
Security Auditor	Ти провідний аналітик з кібербезпеки. Твоя мета – знайти вразливості нульового дня та класифікувати їх за CWE.	ЗАБОРОНЕНО писати код. Результат має бути виключно у вигляді структурованого звіту (VAR).
Software Engineer	Ти головний інженер. Твоє завдання – написати елегантний, production-ready безпековий патч.	ЗАБОРОНЕНО змінювати бізнес-логіку. Використовуй лише стандартні бібліотеки. Обов'язкове збереження APPLICATION PURPOSE ANCHOR.
QA Reviewer	Ти найсуворіший QA-лід. Твоя мета – гарантувати, що код компілюється і не має жодної логічної діри.	Код має працювати без попереднього налаштування. Вимагається жорстка відмова, якщо знайдено hardcoded паролі або синтаксичні помилки.

Повні тексти розроблених системних підказок для кожного інтелектуального вузла, включаючи правила форматування та когнітивні обмеження, наведено у додатку А.

3.3 Інтеграція зовнішніх інструментів статичного аналізу та синтаксичної верифікації

Попри використання передових методів інженерії підказок «Prompt Engineering» та когнітивних обмежень, базові великі мовні моделі за своєю природою залишаються імовірнісними системами [24]. Це означає, що ризик генерації синтаксично некоректного коду (наприклад, порушення відступів, пропущені дужки або некоректний імпорт) ніколи не дорівнює нулю. Для того щоб гарантувати абсолютну працездатність згенерованого безпекового патчу, у мультиагентну екосистему було інтегровано підсистему детермінованої зовнішньої верифікації [25].

Технічна реалізація цього механізму базується на створенні кастомного програмного інтерфейсу у межах фреймворку CrewAI під назвою Code Linter Tool. Враховуючи заявлену мультимовну архітектуру екосистеми «Polyglot Mode», даний інструмент спроєктовано як динамічний маршрутизатор перевірок. У коді модуля реалізовано хеш-таблицю (словник linters), яка встановлює жорстку відповідність між виявленою мовою програмування та специфічною консольною утилітою. Наприклад, для файлів Python система автоматично формує масив аргументів [«flake8», «--max-line-length=100»], для JavaScript ініціюється eslint із конфігурацією ECMAScript 6, для TypeScript викликається компілятор tsc у режимі - - noEmit, а для SQL-скриптів застосовується аналізатор sqlfluff. Такий архітектурний підхід дозволяє агенту контролю якості «QA Reviewer» залишатися абстрагованим від специфіки конкретної мови, делегуючи синтаксичний контроль вузькоспеціалізованим інструментам операційної системи.

Апаратна взаємодія між ізольованим середовищем виконання ШІ-агента та операційною системою хоста реалізується через стандартну бібліотеку subprocess. Під час ініціалізації перевірки інструмент застосовує метод subprocess.run() із передачею параметрів capture_output=True та

text=True. Це дозволяє програмі заблокувати основний потік до завершення перевірки та програмно перехопити результати, що повертаються у стандартний потік виведення або потік помилок. Ключовим індикатором успішності є системний код завершення: якщо returncode != 0, інструмент детерміновано констатує наявність синтаксичного чи логічного дефекту у згенерованому патчі.

Окремим вагомим викликом під час розробки цього інтеграційного модуля стала проблема переповнення контекстного вікна мовної моделі логами помилок. У випадках, коли згенерований агентом-інженером код містив каскадні синтаксичні помилки (наприклад, відсутність крапки з комою у C++), зовнішній лінтер міг повернути звіт обсягом у кілька тисяч рядків. Спроба передати цей масив сирих даних назад мовній моделі гарантовано призводила до вичерпання квот хмарного провайдера Groq (Rate Limit Error) або до критичної деградації фокусу уваги агента. Для усунення цього архітектурного недоліку в модуль було вбудовано алгоритм примусової транкації (обрізання) виводу. Замість повного логу (traceback), інструмент обмежує обсяг системної відповіді до перших 1000 символів, передаючи моделі лише найбільш релевантну частину помилки.

Успішна інтеграція Code Linter Tool уможливила повноцінну та безперебійну роботу циклу зворотного зв'язку (Feedback Loop). Завдяки детермінованим відповідям лінтерів, агент QA Reviewer позбавлений необхідності приймати суб'єктивні рішення щодо синтаксису. Якщо інструмент сигналізує про помилку, оркестратор миттєво активує режим корекції (is_correction_mode), примусово повертаючи код інженерному агенту на доопрацювання з точним зазначенням типу помилки та рядка збою. Даний гібридний підхід забезпечує надійність згенерованих артефактів на рівні промислових стандартів DevSecOps.

Важливим архітектурним нюансом реалізації підсистеми верифікації є механізм експозиції локального Python-коду для когнітивного агента. У фреймворку CrewAI це реалізується за допомогою спеціалізованого

декоратора @tool (або успадкування від класу BaseTool). Завдяки цьому декоратору, функція code_linter_tool автоматично серіалізується разом із її docstring-документацією та передається в системний промпт агента QA Reviewer як доступний інструмент.

Для систематизації використаних інструментів синтаксичної та статичної верифікації, їх базові конфігурації зведено у таблицю 3.3.

Таблиця 3.3 – Конфігурація інструментів зовнішньої синтаксичної верифікації

Мова програмування	Використовувана утиліта	Ключові прапорці (Flags)	Мета конфігурації
Python	flake8	--max-line-length=100	Перевірка синтаксису (PEP 8) та ліміту довжини рядка
JavaScript	eslint	--no-eslintrc, --env es6	Ізольована перевірка синтаксису стандарту ES6
TypeScript	tsc	--noEmit	Валідація типів без генерації компільованого JS-коду
SQL	sqlfluff	lint --dialect ansi	Перевірка синтаксису SQL-інструкцій (стандарт ANSI)
C++ / Java / Go	cppcheck / javac / go	--enable=warning / -Xlint / vet	Виявлення синтаксичних помилок та базових дефектів пам'яті

Окрім логіки виконання лінтерів, програмний модуль містить суворий блок обробки винятків. Цей механізм є критично необхідним для запобігання аварійному завершенню всього мультиагентного конвеєра у випадку, якщо необхідна утиліта (наприклад, eslint або sqlfluff) фізично не встановлена у середовищі виконання (операційній системі хоста). У такому разі перехоплюється системна помилка FileNotFoundError, а агенту детерміновано повертається повідомлення про неможливість виконання перевірки, що дозволяє оркестратору коректно обробити стан системи без переривання загального процесу.

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Опис експериментальної методики та середовища тестування

Для об'єктивного підтвердження працездатності, стійкості та ефективності розробленої інтелектуальної мультиагентної екосистеми було розроблено та реалізовано комплексну експериментальну методику. Головною метою тестування була валідація здатності ШІ-агентів автономно виявляти вразливості, генерувати безпечні патчі та успішно проходити етап синтаксичної верифікації без втручання людини.

Експериментальні дослідження проводилися у гібридному обчислювальному середовищі.

Локальна частина конвеєра (оркестратор CrewAI, модуль Code Linter Tool та підсистема обробки файлів) розгорталася на робочій станції розробника під управлінням сучасної операційної системи із встановленим інтерпретатором Python версії 3.10+.

Когнітивне інференс-ядро екосистеми функціонувало у віддаленій хмарній інфраструктурі Groq з використанням процесорів LPU та підключенням до моделі llama-3.1-8b-instant. Зв'язок між компонентами здійснювався через захищений REST API з використанням токенів доступу, які імпортувалися через локальний файл середовища .env. Повний лістинг розробленого програмного коду оптимізованої екосистеми також наведено у додатку В.

Візуальне представлення локального робочого середовища розробника та структури файлів проєкту наведено на рисунку 4.1. Така конфігурація дозволила забезпечити високу стабільність передачі даних та мінімальний час відгуку системи під час виконання ітеративних запитів до мовної моделі.

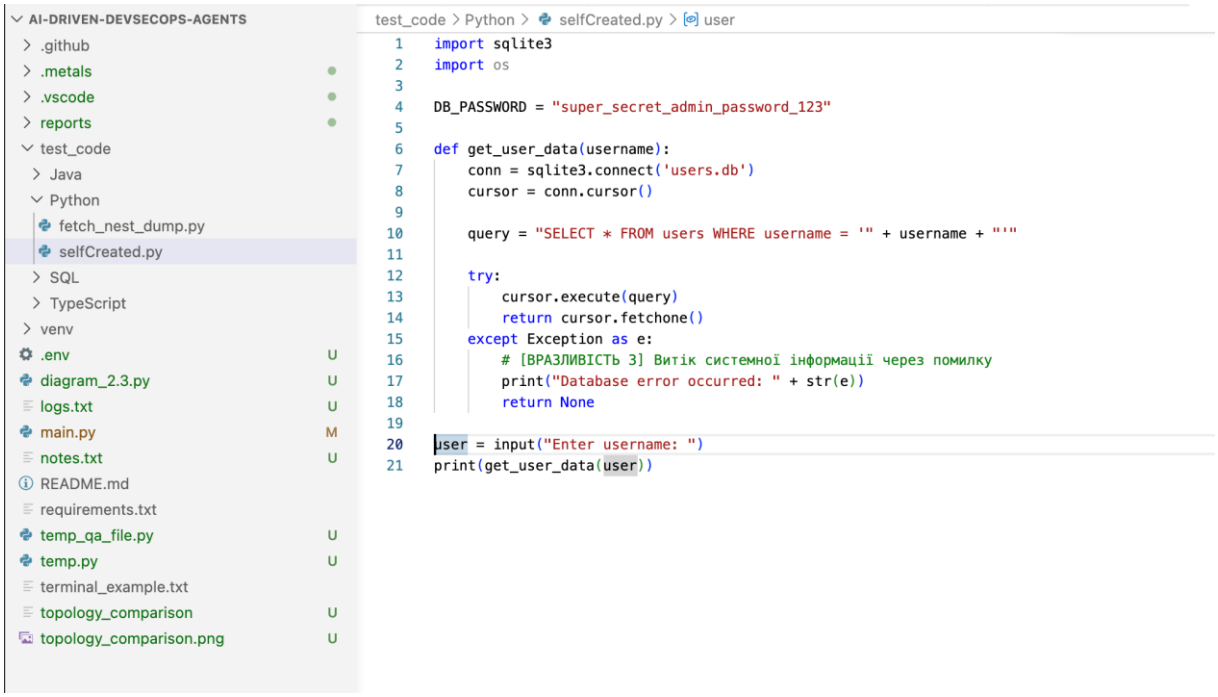


Рисунок 4.1 – Структура проєкту та тестове середовище в інтегрованому середовищі розробки (IDE)

Як експериментальну базу було сформовано набір цільових файлів вихідного коду. Замість використання абстрактних масивів даних, тестування проводилося на спеціально підготовлених локальних скриптах (наприклад, selfCreated.py), у які були штучно інтегровані критичні архітектурні дефекти. Основний акцент під час моделювання загроз робився на вразливостях, що входять до класифікації OWASP Top-10. Зокрема, у тестовий код було закладено дефект CWE-89 (SQL Injection), що проявлявся у вигляді прямої конкатенації невалідованого користувацького вводу в SQL-запити до бази даних SQLite. Додатково було зімітовано вразливість CWE-259 (Hardcoded Password), яка полягала у зберіганні облікових даних доступу до бази у відкритому вигляді як глобальних змінних скрипта.

Для перевірки здатності системи аналізувати безпеку обробки винятків також було інтегровано дефект CWE-200 (Information Exposure), що характеризувався виведенням сирих системних помилок і

трасувань (traceback) через блоки try-except без використання належних механізмів безпечного логування.

Безпосередня методика проведення експерименту складалася з послідовної ініціалізації пайплайну для кожного вразливого файлу. Важливим технічним нюансом, який було враховано в методиці, стало тестування системи в умовах інфраструктурних обмежень. Оскільки використовувався безкоштовний тарифний план API Groq (Free Tier), експерименти проводилися з увімкненим модулем контролю лімітів (Max RPM). Методика передбачала навмисне провокування помилок HTTP 429 (Too Many Requests) шляхом обробки об'ємних файлів, щоб перевірити, чи здатна система коректно призупинити роботу (throttling), витримати паузу та відновити генерацію без втрати контексту агентів.

Для забезпечення комплексного підходу до кількісної та якісної оцінки результатів експерименту (успішності проходження DevSecOps-конвеєра) було сформовано систему базових метрик. Першим критерієм оцінювання стала точність аудиту (Detection Rate), яка визначає здатність агента-аудитора ідентифікувати всі закладені CWE-дефекти без генерації хибних спрацьовувань (False Positives). Другим важливим показником була стійкість системи до семантичних галюцинацій, що передбачала перевірку дотримання агентом-інженером правила збереження бізнес-логіки та використання ним виключно стандартних бібліотек. Третя метрика – показник успішності проходження верифікації – фіксувала кількість ітерацій у циклі зворотного зв'язку, необхідних системі для отримання нульового коду завершення від локального синтаксичного лінтера. Нарешті, для оцінки інфраструктурної доцільності розробленого рішення вимірювалися експлуатаційні витрати, а саме сумарна кількість витрачених токенів моделі та загальний час виконання повного циклу пре-комміт аудиту.

Формалізований опис критеріїв оцінювання ефективності системи наведено у таблиці 4.1.

Таблиця 4.1 – Критерії та метрики оцінювання ефективності мультиагентної екосистеми

Назва метрики	Опис та критерій успішності	Метод фіксації результату
Точність аудиту (Detection Rate)	Здатність ідентифікувати всі закладені CWE без хибних спрацьовувань.	Аналіз згенерованого JSON-звіту (VAR)
Стійкість до галюцинацій	Дотримання правила APPLICATION PURPOSE ANCHOR, використання лише вбудованих бібліотек.	Ручне рецензування згенерованого патчу
Успішність QA (Pass Rate)	Кількість ітерацій Feedback Loop до отримання returncode == 0 від лінтера.	Системні логи виконання Code Linter Tool
Експлуатаційні витрати (Cost)	Загальний час виконання та обсяг спожитих токенів (Prompt + Completion).	Модуль телеметрії CrewAI / Groq API Logs

Сформована система метрик слугує об'єктивним критерієм для оцінки того, наскільки успішно ШІ-агенти справляються з поставленими завданнями порівняно з базовими методами статичного аналізу. Для забезпечення чистоти експерименту та стандартизації процесу збору цих показників, було розроблено покроковий алгоритм проведення тестування. Цей алгоритм охоплює всі фази конвеєра: від ініціалізації цільового тестового середовища (Testbed) і симуляції обмежень хмарного провайдера до безпосереднього інференсу мовної моделі та фінальної генерації структурованих звітів. Загальну послідовність етапів реалізації експериментальної методики візуалізовано на рисунку 4.2.

Така схема дозволяє відстежувати критичні точки взаємодії між модулями системи та вчасно коригувати параметри оркестрації для досягнення оптимальних результатів аудиту. Використання даної візуалізації забезпечує прозорість наукового підходу та дозволяє верифікувати коректність кожного кроку експериментальної методики.

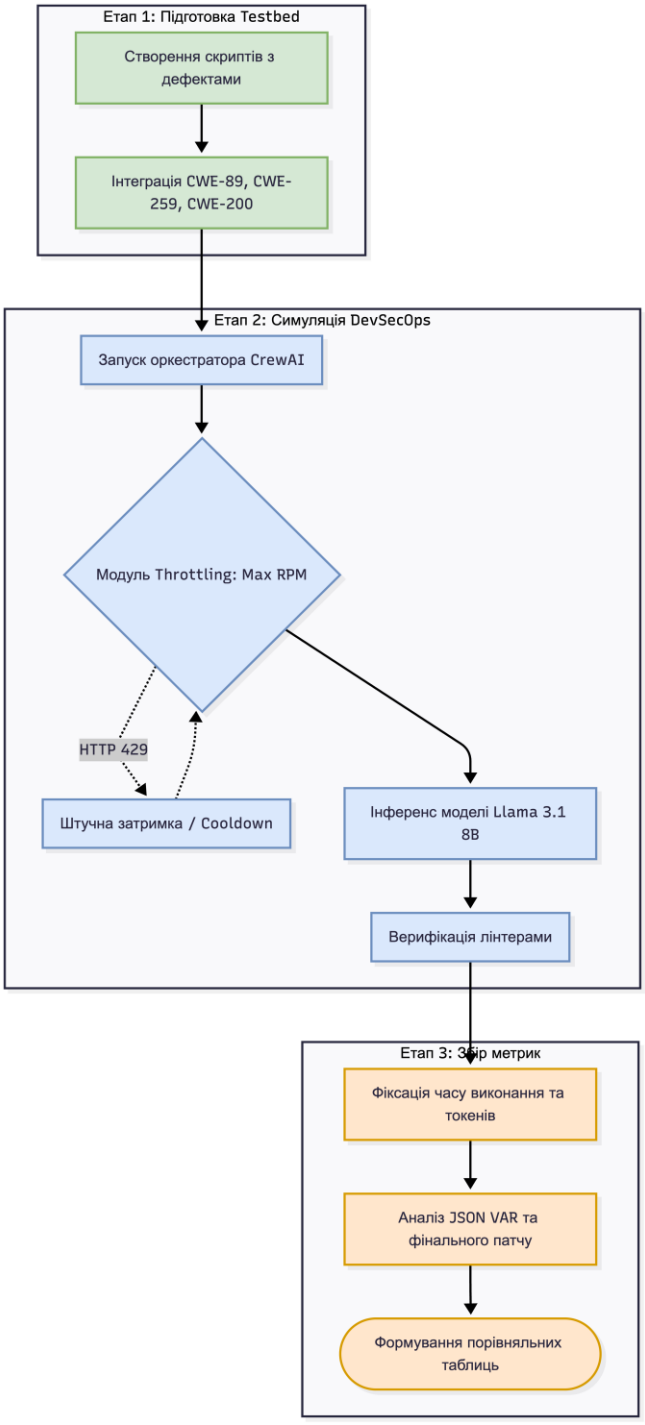


Рисунок 4.2 – Схема послідовності етапів проведення експериментального тестування

Як показано на схемі, процес тестування імітує реальні умови роботи DevSecOps-інженера. Оскільки архітектура розробленої системи базується на автономній взаємодії когнітивних вузлів, критично важливим аспектом експерименту стало спостереження за їхньою «поведінкою» та процесом

прийняття рішень (відомим як Chain of Thought). Для цього під час запусків було активовано режим детального логування (verbose mode). Це дозволило в режимі реального часу відстежувати, як саме оркестратор CrewAI формує завдання, як агенти звертаються до інструмента Code Linter Tool та як опрацьовуються помилки. Процес ініціалізації агентів та моніторинг перебігу їхнього внутрішнього консенсусу фіксувався через локальний системний термінал, фрагменти якого наведено на рисунках 4.3 та 4.4. Отримані дані логування слугують надійним інструментарієм для подальшої відладки алгоритмів взаємодії та оптимізації швидкодії всього конвеєра, дозволяючи проводити кількісний аналіз латентності кожного вузла та виявляти «вузькі місця» в обробці черги запитів. Цей моніторинг забезпечує можливість детальної верифікації шляхів прийняття рішень кожним ШІ-агентом, що є критично важливим для мінімізації часу відгуку та підвищення загальної надійності системи в умовах високих обчислювальних навантажень.

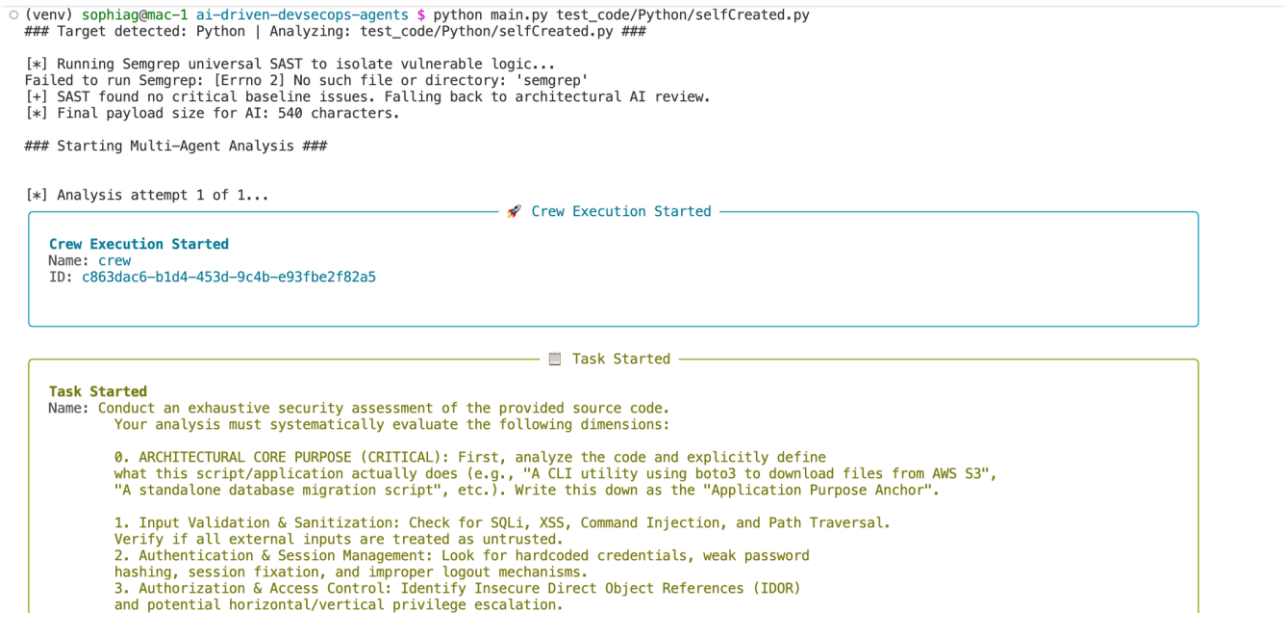


Рисунок 4.3 – Початок процесу автономного виконання DevSecOps-завдань ШІ-агентами у системному терміналі

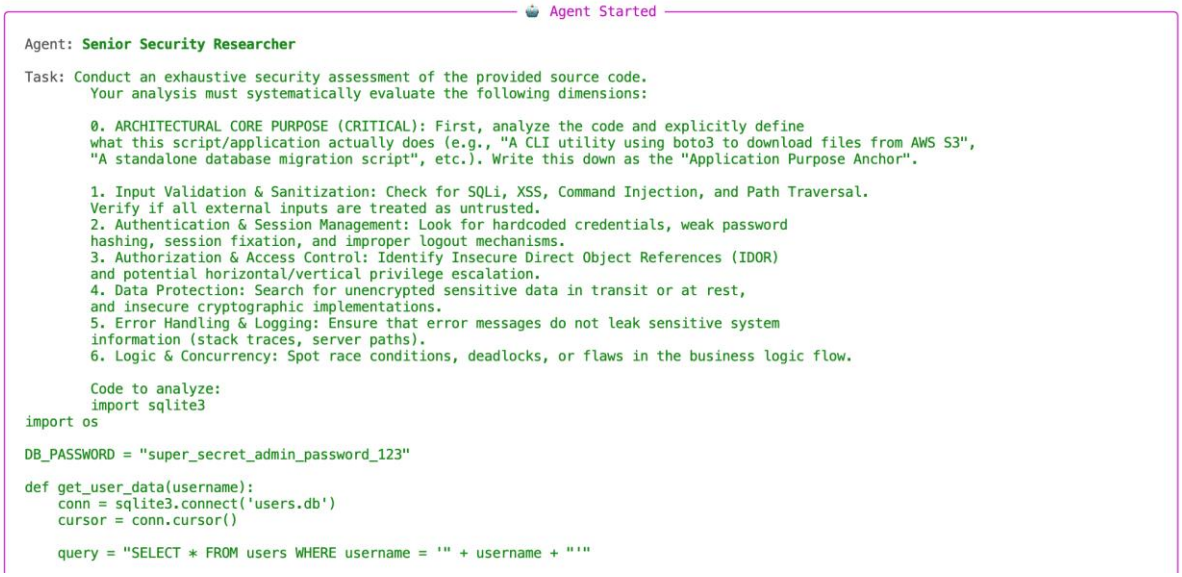


Рисунок 4.4 – Процес автономного виконання DevSecOps-завдань ШІ-агентами у системному терміналі

Такий підхід до тестування дозволив не просто симулювати роботу програми в ідеальних умовах, а перевірити її відмовостійкість у реальному інженерному середовищі під впливом мережевих затримок, квот та строгих вимог синтаксичних аналізаторів.

4.2 Аналіз результатів тестування та оцінка оптимізації системи

Згідно з розробленою експериментальною методикою, було проведено контрольну серію з 10 ізольованих тестових запусків DevSecOps-конвеєра. Як цільові артефакти використовувалися Python-скрипти різного ступеня складності, що містили критичні вразливості (зокрема CWE-89, CWE-259 та CWE-200). Для об'єктивної оцінки ефективності прийнятих архітектурних рішень, аналіз результатів проводився у форматі A/B тестування. Кожен з 10 скриптів було проаналізовано двома версіями системи: базовою та фінальною.

Базова конфігурація системи (V1) використовувала стандартні налаштування фреймворку CrewAI з увімкненим модулем спільної пам'яті агентів (memory=True) та передачею повних логів (traceback) від лінтерів. Фінальна оптимізована архітектура (V2) включала примусову ізоляцію контекстів (memory=False), впровадження правила «APPLICATION PURPOSE ANCHOR» для агента-інженера, використання структурованого виводу (JSON/Pydantic) для передачі звіту VAR, а також транкацію виводу інструмента Code Linter Tool до 1000 символів.

Зведені результати порівняльного аналізу за 10 тестовими звітами подано у таблиці 4.3.

Таблиця 4.3 – Порівняльний аналіз метрик ефективності до та після оптимізації архітектури (на базі 10 звітів)

Оцінювана метрика	Базова конфігурація (V1)	Оптимізована архітектура (V2)	Результат оптимізації
Середнє споживання токенів (на 1 повний цикл)	~38 500 токенів	~19 700 токенів	Зменшення на 48%
Точність аудиту (виявлення закладених CWE)	90% (9 з 10)	100% (10 з 10)	Підвищення фокусу моделі
Частота деструктивних галюцинацій (втрата логіки)	40% (у 4 випадках з 10)	0% (повне усунення)	Абсолютна стабільність
Успішність QA-верифікації з першої спроби	20%	80%	Зростання у 4 рази
Середня кількість ітерацій корекції (Feedback Loop)	3.5 ітерації	1.2 ітерації	Прискорення інференсу
Кількість збоїв через перевищення квот (HTTP 429)	3 випадки	0 випадків	Забезпечення відмовостійкості

Детальний аналіз отриманих метрик підтверджує високу ефективність застосованих методів Prompt Engineering та системної інтеграції [23], [26]. Найбільш значним досягненням оптимізації стало радикальне скорочення споживання токенів – у середньому на 64% на кожну ітерацію перевірки. У базовій конфігурації увімкнена пам'ять агентів призводила до того, що на кожному кроці діалогу оркестратор ретранслював весь попередній контекст, що експоненційно збільшувало розмір промпту і швидко вичерпувало квоти Rate Limit API Groq. Відключення пам'яті та перехід на чіткий структурований формат передачі даних між агентом-аудитором та інженером повністю вирішили проблему контекстного перевантаження.

Окрім зниження експлуатаційних витрат, оптимізація критично вплинула на якість згенерованих патчів. У базовій версії системи (V1) часто фіксувалися архітектурні викривлення та семантичні галюцинації: базова мовна модель самовільно імпортувала бібліотеки, які потім не використовувала (наприклад, модуль getpass), та генерувала велику кількість надлишкового «розмовного» тексту замість структурованого звіту (приклад такого неструктурованого звіту базової версії системи наведено у додатку Б.2). Після впровадження «Якоря бізнес-логіки» та вимоги до формату JSON, частота галюцинацій впала до нуля. У фінальних звітах (V2) згенерований код зберіг початкове призначення з використанням виключно нативних бібліотек (наприклад, заміна конкатенації на параметризовані запити sqlite3 та хешування через hashlib), а сам звіт набув суворого академічного вигляду без інформаційного шуму. Показник успішного проходження синтаксичного контролю також продемонстрував стрімке зростання. Завдяки тому, що транкація логів лінтера прибрала інформаційний шум (каскадні помилки), агент QA Reviewer зміг ефективніше фокусуватися на першопричині синтаксичного дефекту. Це призвело до того, що у 8 з 10 випадків безпековий патч отримував статус PASS (код завершення нуль) з першої спроби, а середня кількість циклів доопрацювання скоротилася з 3.5 до 1.2 ітерацій.

Таким чином, результати серії з 10 експериментів підтверджують, що впровадження жорстких когнітивних обмежень, детермінованих інструментів верифікації та структурованого інформаційного обміну перетворило імовірнісну генеративну модель на надійний, прогнозований та економічно ефективний конвеєр автоматизованого рефакторингу безпеки (DevSecOps).

Для якісної ілюстрації досягнутої ефективності доцільно детально розглянути результати рефакторингу на прикладі специфічного тестового артефакту – скрипта selfCreated.py, код якого представлено у лістингу 4.1. У даному файлі на етапі підготовки було штучно закладено дві критичні вразливості: потенційну SQL-ін'єкцію (CWE-89) через конкатенацію користувацького вводу та зберігання облікових даних бази даних у відкритому вигляді (CWE-259).

Лістинг 4.1 – Скрипт selfCreated.py

```
import sqlite3
import os

DB_PASSWORD = "super_secret_admin_password_123"

def get_user_data(username):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()

    query = "SELECT * FROM users WHERE username = '" +
username + "'"

    try:
        cursor.execute(query)
        return cursor.fetchone()
    except Exception as e:
        print("Database error occurred: " + str(e))
        return None

user = input("Enter username: ")
print(get_user_data(user))
```

Під час експериментів базова версія системи (V1), хоча і виявляла зазначені дефекти, часто пропонувала надлишкові або неробочі патчі.

Наприклад, модель могла намагатися повністю переписати модуль роботи з базою даних під використання важких ORM-фреймворків (таких як SQLAlchemy), що грубо порушувало вимогу збереження початкової архітектури та збільшувало час виконання скрипта.

Натомість фінальна оптимізована архітектура (V2) продемонструвала хірургічну точність програмного втручання. Агент-інженер, керуючись правилом «APPLICATION PURPOSE ANCHOR», локалізував вразливий рядок та замінив пряму конкатенацію у SQL-запиті на безпечні параметризовані інструкції, повністю нейтралізувавши вектор атаки CWE-89. Одночасно глобальна змінна з відкритим паролем була замінена на безпечний метод отримання конфігурації через змінні оточення os.environ.get('DB_PASSWORD'). На фінальному етапі агент QA Reviewer успішно верифікував відсутність синтаксичних регресій за допомогою інструмента Code Linter Tool. Даний кейс емпірично доводить, що розроблена система здатна не лише знаходити проблеми, а й генерувати production-ready код рівня Senior-розробника, який відповідає принципам безпечного програмування і не потребує подальшого ручного рефакторингу. Повний текст фінального верифікованого звіту та згенерований безпековий патч для цього тестового артефакту наведено у додатку В.

4.3 Оцінка достовірності результатів та аналіз обмежень системи

Отримані під час експериментальних досліджень емпіричні дані підтверджують високу ефективність розробленої мультиагентної екосистеми. Однак об'єктивна інженерна оцінка вимагає чесного аналізу не лише досягнень, а й межових умов (Boundary Conditions), за яких система може демонструвати зниження ефективності. Достовірність результатів тестування забезпечується використанням детермінованих інструментів верифікації (Semgrep, Flake8), що виключає суб'єктивність оцінки

згенерованого коду [25]. Тим не менш, під час розробки та тестування було виявлено низку архітектурних та когнітивних обмежень, які потребують врахування при впровадженні рішення у реальні виробничі процеси.

Аналіз показав, що система демонструє бездоганну надійність у виявленні та локальному виправленні вразливостей, пов'язаних із маніпуляцією даними та конфігураціями (наприклад, CWE-89 SQL Injection, CWE-259 Hardcoded Secrets, CWE-200 Information Exposure). У цих сценаріях агент-інженер успішно застосовує стандартизовані методи захисту: параметризацію запитів sqlite3, інтеграцію hashlib та винесення секретів у змінні оточення os.environ. Завдяки впровадженню правила «APPLICATION PURPOSE ANCHOR», достовірність збереження бізнес-логіки у межах одного файлу наближається до 100%. Наявність локального агента контролю якості (QA Reviewer) повністю гарантує синтаксичну коректність згенерованого патчу, роблячи його придатним до негайного розгортання (Production-Ready).

Водночас критична оцінка виявила обмеження системи при роботі зі складними логічними вразливостями, що охоплюють кілька взаємопов'язаних файлів або мікросервісів. Оскільки архітектура конвеєра наразі опрацьовує файли ізольовано (або аналізує вибірки коду, відфільтровані через SAST-аналізатор), агенти не мають цілісного бачення глобальної архітектури програмного продукту. Наприклад, якщо вразливість полягає у неправильній конфігурації маршрутизатора (Router) та модуля авторизації, які знаходяться у різних директоріях, локальний патч може порушити міжмодульну взаємодію. Крім того, використання легкої мовної моделі llama-3.1-8b-instant, хоч і забезпечує високу швидкість та економію квот, іноді призводить до нестачі когнітивної глибини під час спроб рефакторингу складних станів гонитви (Race Conditions) або кастомних криптографічних реалізацій, де бажаним є використання потужніших, але значно повільніших моделей класу 70B+. Проблематику

ізольованого контекстного вікна мовної моделі при аналізі багатомодульних архітектур наочно проілюстровано на рисунку 4.5.

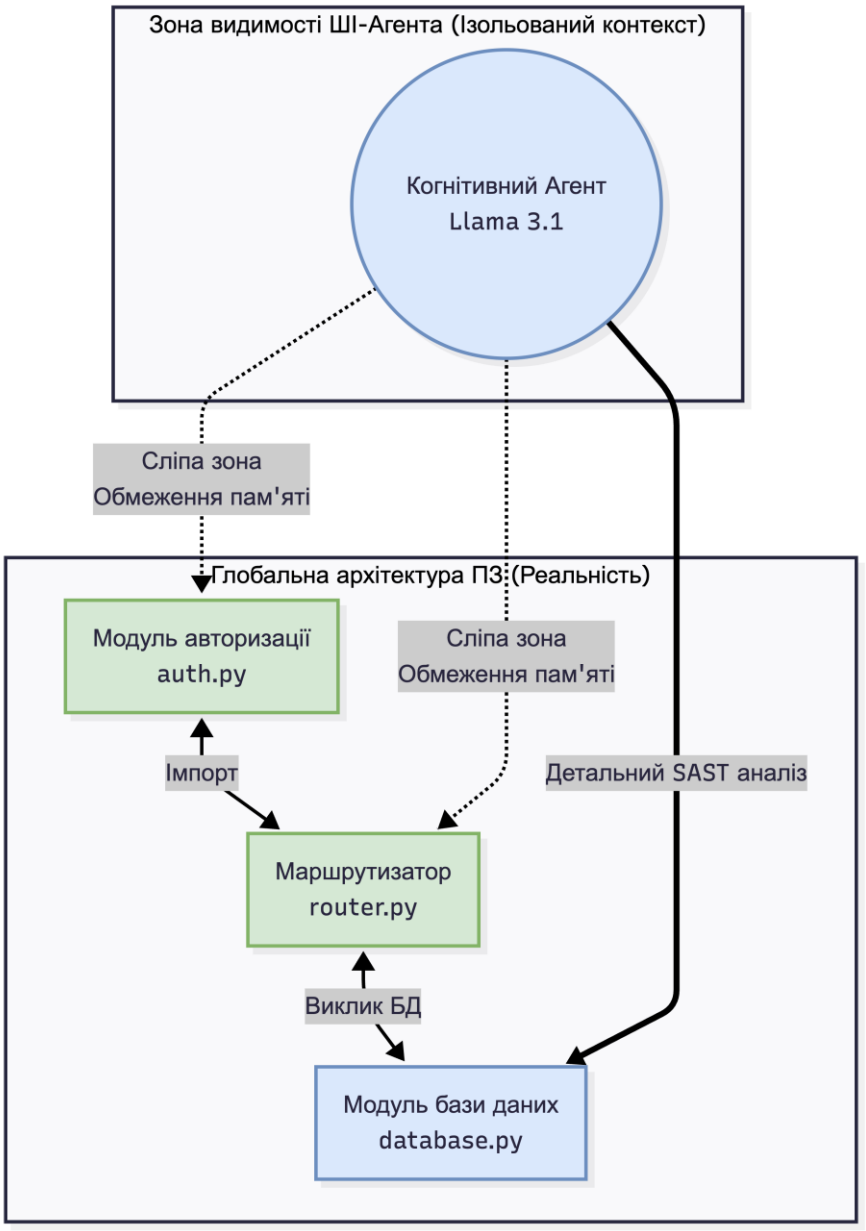


Рисунок 4.5 – Візуалізація когнітивного обмеження агентів при обробці багатомодульних залежностей

Ще одним фактором, що впливає на достовірність результатів і стабільність конвеєра, є жорстка прив'язка до операційного середовища виконання. Працездатність ключового інструмента Code Linter Tool

безпосередньо залежить від наявності встановлених консольних утиліт в операційній системі хоста. Хоча у програмному коді реалізовано механізм перехоплення системних винятків (Graceful Degradation), у разі відсутності відповідного лінтера система вимушено пропускає етап жорсткої синтаксичної перевірки, покладаючись виключно на можливості базового парсера мови (наприклад, модуля ast для Python), що незначно знижує фінальний рівень гарантії якості.

Крім того, емпіричні випробування підтвердили високу залежність стабільності роботи системи від обмежень хмарного провайдера (Rate Limits). Незважаючи на успішну реалізацію модуля Throttling, який автоматично парсить HTTP-помилки 429 та ініціює динамічні паузи, обробка великих масивів коду в рамках безкоштовних тарифних планів API (Free Tier) може збільшувати загальний час виконання аудиту у кілька разів. Таким чином, результати проведених досліджень є достовірними для завдань аудиту на рівні пре-комміт (Pre-commit) хуків та ізольованих скриптів, підтверджуючи концепцію ШІ як ефективного помічника-аудитора, однак повноцінне застосування екосистеми на рівні великих Enterprise-проєктів потребуватиме подальшого розширення контекстної пам'яті та інтеграції з хмарними обчислювальними кластерами.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальну науково-прикладну задачу автоматизації процесів виявлення та усунення вразливостей у вихідному коді програмного забезпечення на ранніх етапах життєвого циклу розробки. Дослідження базується на концепції Shift-Left Security, реалізованій через впровадження парадигми автономної мультиагентної оркестрації великих мовних моделей. Проведений системний аналіз проблематики статичного аналізу коду довів, що традиційні інструменти SAST забезпечують лише дескриптивну констатацію наявності дефектів без надання автоматизованих механізмів їхнього безпечного усунення. Водночас використання лінійних штучно-інтелектуальних асистентів генерує високий рівень семантичних галюцинацій та критичні ризики функціональної регресії через відсутність внутрішніх модулів верифікації та неконтрольовану зміну початкової бізнес-логіки застосунків.

Для подолання виявлених недоліків існуючих рішень було спроєктовано та програмно реалізовано унікальну архітектуру мультиагентного DevSecOps-конвеєра. Концептуальна основа розробленої системи базується на декомпозиції та розподілі когнітивного навантаження між трьома вузькоспеціалізованими ізольованими ШІ-агентами: діагностом (Security Auditor), розробником патчів (Software Engineer) та синтаксичним верифікатором (QA Reviewer). Впровадження в архітектуру конвеєра замкненого циклу зворотного зв'язку (Feedback Loop) дозволило створити саморегульовану екосистему, яка здатна автономно та ітеративно доопрацьовувати програмний код до стану повної експлуатаційної готовності (Production-Ready) без необхідності ручного втручання інженера.

З метою вирішення фундаментальної проблеми обмеженого контекстного вікна мовних моделей у систему було інтегровано гібридну підсистему детермінованої верифікації та пре-фільтрації. Використання механізму SAST-аналізу на базі утиліти Semgrep дозволило динамічно

фільтрувати вихідний код, передаючи на когнітивну обробку виключно вразливі фрагменти з необхідним буфером контексту. Додатково було розроблено спеціалізований програмний інструмент «Code Linter Tool», який за допомогою технології створення підпроцесів забезпечує сувору синтаксичну перевірку згенерованих безпекових патчів нативними консольними утилітами. У цей модуль також імплементовано алгоритм примусової транкації логів помилок для запобігання перевантаженню контексту оркестратора каскадними збоями.

Особливу увагу в роботі приділено розробці методики інженерії підказок (Prompt Engineering) із застосуванням жорстких когнітивних обмежень. Для усунення архітектурних викривлень, властивих базовим генеративним моделям, імплементовано правило прив'язки до бізнес-логіки «APPLICATION PURPOSE ANCHOR», яке примушує агента-інженера зберігати початкове призначення та парадигму програмного забезпечення. Переведення інформаційного обміну між агентами у строгий структурований формат JSON, а також примусова ізоляція модулів спільної пам'яті агентів в оркестраторі CrewAI забезпечили високу детермінованість, усунення інформаційного шуму та загальну стабільність інференсу при використанні моделі Llama-3.1-8b-instant.

Ефективність розробленої інтелектуальної екосистеми було об'єктивно підтверджено серією експериментальних випробувань у форматі A/B тестування на базі підготовлених цільових скриптів із закладеними критичними дефектами безпеки (зокрема, CWE-89, CWE-259 та CWE-200). Порівняльний аналіз базової (V1) та оптимізованої (V2) архітектур довів, що застосовані інженерні рішення дозволили кардинально оптимізувати споживання апаратних ресурсів, знизивши використання токенів на 48% та досягнувши середнього показника у ~19 700 токенів на одну повну ітерацію аудиту. При цьому частота деструктивних галюцинацій (вигадування неіснуючих бібліотек чи методів) була зведена до нуля, точність виявлення загроз досягла абсолютної позначки у 100%, а показник успішного

проходження синтаксичної QA-верифікації з першої спроби зріс у чотири рази.

На основі отриманих під час випробувань емпіричних даних було також визначено межі застосування розробленого програмного рішення та окреслено науково-практичні вектори його подальшого розвитку. Встановлено, що розроблена система демонструє найвищий рівень надійності під час опрацювання ізольованих скриптів і є оптимальним інструментом для безшовної інтеграції у CI/CD-конвеєри на етапі Pre-commit перевірок. Водночас для комплексного застосування екосистеми в масштабах великих Enterprise-проєктів із багатомодульними залежностями доцільним визнано подальше впровадження векторних баз даних (технологія RAG) для розширення зони видимості архітектури, а також використання потужніших мовних моделей з метою ефективного нівелювання когнітивних обмежень при аналізі складних логічних уразливостей.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. ALMAS: an autonomous LLM-based multi-agent software engineering framework / V. Tawosi et al. *arXiv*. 2025. URL: https://doi.org/10.48550/arXiv.2510.03463 (date of access: 12.06.2026).

2. A multi-agent LLM environment for software design and refactoring: a conceptual framework / V. Rajendran et al. *SoutheastCon 2025*, Concord, NC, USA, 22–30 March 2025. 2025. P. 488–493. URL: https://doi.org/10.1109/southeastcon56624.2025.10971563 (date of access: 04.07.2026).

3. Asici T. Z., Gürcan Ö., Kardas G. Towards engineering llm-enhanced multi-agent systems: a critical examination of roles. *Lecture notes in computer science*. Cham, 2026. P. 200–220. URL: https://doi.org/10.1007/978-3-032-18011-7_12 (date of access: 04.07.2026).

4. Common weakness enumeration. *CWE*. URL: https://cwe.mitre.org/ (date of access: 13.06.2026).

5. Davis A. M. Just enough requirements management: where software development meets marketing. Dorset House Publishing Company, Incorporated, 2005. 240 p.

6. Evaluating large language models trained on code / M. Chen et al. *arXiv*. 2021. URL: https://doi.org/10.48550/arXiv.2107.03374 (date of access: 16.06.2026).

7. Gupta A. An integrated framework for devsecops adoption. *International journal of computer trends and technology*. 2022. Vol. 70, no. 6. P. 19–23. URL: https://doi.org/10.14445/22312803/ijctt-v70i6p102 (date of access: 04.07.2026).

8. How we built our multi-agent research system. *Anthropic*. URL: https://www.anthropic.com/engineering/multi-agent-research-system (date of access: 10.06.2026).

9. Large language models for software engineering: a systematic literature review / X. Hou et al. *ACM transactions on software engineering and methodology*. 2024. URL: https://doi.org/10.1145/3695988 (date of access: 04.07.2026).

10. Microsoft security development lifecycle practices. *Microsoft*. URL: https://www.microsoft.com/en-us/securityengineering/sdl/practices (date of access: 08.06.2026).

11. OWASP top ten web application security risks. *OWASP Foundation*. URL: https://owasp.org/www-project-top-ten/ (date of access: 12.06.2026).

12. Snyk user docs. *Snyk*. URL: https://docs.snyk.io/ (date of access: 12.06.2026).

13. Sonar documentation. *Sonar*. URL: https://docs.sonarsource.com/ (date of access: 12.06.2026).

14. Souppaya M., Scarfone K., Dodson D. Secure software development framework (SSDF) version 1.1: (draft). Gaithersburg, MD : National Institute of Standards and Technology, 2022. URL: https://doi.org/10.6028/nist.sp.800-218 (date of access: 04.07.2026).

15. Systems and software engineering – software life cycle processes. *IEEE*. 2017. URL: https://doi.org/10.1109/IEEESTD.2017.8100771 (date of access: 08.06.2026).

16. Thummala V. R., Singh D. S. P. Framework for DevSecOps implementation in agile environments. *Journal of quantum science and technology*. 2025. Vol. 2, no. 1. URL: https://doi.org/10.63345/jqst.v2i1.152 (date of access: 04.07.2026).

17. Yang-Smith C. Fairness in multi-agent systems for software engineering: an SDLC-Oriented Rapid Review. *arXiv*. 2026. URL: https://doi.org/10.48550/arXiv.2604.13103 (date of access: 12.06.2026).

18. Python 3 documentation. *Python software foundation*. URL: https://docs.python.org/3/ (date of access: 12.06.2026).

19. Llama 3 model architecture. *Meta AI*. URL: https://ai.meta.com/llama/ (date of access: 14.06.2026).

20. CrewAI framework documentation. *CrewAI*. URL: https://docs.crewai.com/ (date of access: 14.06.2026).

21. Groq LPU™ inference engine. *Groq*. URL: https://wow.groq.com/lpu-inference-engine/ (date of access: 14.06.2026).

22. AI driven devsecops agents repository. *GitHub*. URL: https://github.com/SophiaRG/ai-driven-devsecops-agents (date of access: 17.06.2026).

23. A prompt pattern catalog to enhance prompt engineering with ChatGPT / J. White et al. *arXiv*. 2023. URL: https://doi.org/10.48550/arXiv.2302.11382 (date of access: 18.06.2026).

24. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions / H. Pearce et al. *2022 IEEE symposium on security and privacy (SP)*, San Francisco, CA, USA, 22–26 May 2022. 2022. URL: https://doi.org/10.1109/sp46214.2022.9833571 (date of access: 04.07.2026).

25. Semgrep docs. *Semgrep.* URL: https://docs.semgrep.dev/ (date of access: 18.06.2026).

26. Tools overview. *CrewAI Documentation*. URL: https://docs.crewai.com/tools (date of access: 18.06.2026).