Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Розроблення застосунку для комплексного аналізу якості, безпеки та готовності до експлуатації програмного коду в Node.js-проєктах із використанням штучного інтелекту
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-22-3
Драненко О. А.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Любченко В. А.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Драненко Олексію Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення застосунку для комплексного аналізу якості, безпеки та готовності до експлуатації програмного коду в Node.js-проєктах із використанням штучного інтелекту

затверджена наказом університету від 26 травня 2026 року № 435Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 22 червня 2026 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література у сфері розробки та тестування програмного забезпечення, матеріали конференцій, дані інтернет-мережі, офіційна документація платформи Node.js та мови TypeScript, документація фреймворку Next.js та бібліотеки React, документація до великої мовної моделі Gemini 2.5 Flash від Google, специфікації API GitHub (REST та GraphQL), офіційна документація баз даних MongoDB (разом із Mongoose) та Redis, документація системи асинхронних черг BullMQ, звіти щодо стану індустрії (GitHub Octoverse, Snyk), сучасні галузеві стандарти безпеки вебзастосунків (зокрема OWASP Top 10).

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз сучасного стану методів контролю якості та безпеки програмного коду, зокрема коду, згенерованого за допомогою штучного інтелекту.

2. Дослідження існуючих інструментів статичного аналізу (SAST, SCA) та визначення їхніх обмежень і недоліків.

3. Проєктування архітектури веборієнтованої системи з асинхронним обробленням фонових завдань сканування.

4. Розроблення модуля детермінованого сканування для первинного виявлення архітектурних дефектів та вразливостей у Node.js-проєктах.

5. Проєктування та реалізація модуля інтелектуальної верифікації виявлених дефектів за допомогою моделі Gemini 2.5 Flash для мінімізації хибно-позитивних спрацьовувань.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Порівняльний аналіз існуючих інструментів контролю якості коду (плакат), високорівнева клієнт-серверна архітектура розробленої системи (схема), шарова архітектура серверної частини (схема), архітектура асинхронної обробки завдань через BullMQ і Redis (схема), діаграма «сутність-зв'язок» бази даних MongoDB (схема), структура маршрутизації клієнтської частини застосунку (схема), графічний інтерфейс користувача.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	13.04.2026	
2	Аналіз завдання, підбір літератури	14.04.26-16.04.26	
3	Аналіз літератури з проблеми, що вирішується	17.04.26-19.04.26	
4	Аналіз існуючих методів розпізнавання	20.04.26-22.04.26	
5	Формування кроків вирішення проблеми	23.04.26-05.05.26	
6	Програмна реалізація	26.04.26-20.05.26	
7	Аналіз отриманих результатів	21.05.26-04.06.26	
8	Оформлення пояснювальної записки	05.06.26-09.06.26	
9	Перевірка на нормоконтроль	11.06.26-12.06.26	
10	Перевірка на плагіат	12.06.26-13.06.26	
11	Рецензування	20.06.26-30.06.26	
12	Підготовка презентації та доповіді	20.06.26-30.06.26	
13	Занесення роботи в електронний архів	02.07.26	
14	Попередній захист кваліфікаційної роботи	02.07.26	

Дата видачі завдання 13 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Любченко В. А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 79 с., 9 табл., 21 рис., 26 джерел.

ВЕБЗАСТОСУНОК, ВЕРИФІКАЦІЯ, СТАТИЧНИЙ АНАЛІЗ, ШТУЧНИЙ ІНТЕЛЕКТ, GEMINI 2.5 FLASH, GITHUB, NEXT.JS, NODE.JS.

Об’єктом роботи є розроблення вебзастосунку для автоматизованого аналізу GitHub-репозиторіїв та оцінювання їхньої готовності до промислової експлуатації.

Метою роботи є розроблення вебзастосунку, що виконує статичний аналіз вихідного коду та верифікує виявлені дефекти за допомогою штучного інтелекту.

Під час роботи використано: методи статичного аналізу коду та штучного інтелекту, методи веброзробки

Практична цінність полягає в автоматизації виявлення вразливостей і проблем якості коду зі зниженням кількості хибнопозитивних спрацьовувань завдяки автономній верифікації.

Розроблено вебзастосунок, що забезпечує статичний аналіз репозиторіїв за трьома напрямами, обчислення оцінки готовності до промислової експлуатації та генерацію рекомендацій щодо виправлення виявлених проблем.

Область застосування: підготовка до виробничого розгортання, аудит безпеки коду.

ARTIFICIAL INTELLIGENCE, GEMINI 2.5 FLASH, GITHUB, NEXT.JS, NODE.JS, STATIC ANALYSIS, VERIFICATION, WEB APPLICATION.

The objective of this work is to develop a web application for automated analysis of GitHub repositories and assessment of their production readiness.

The goal of this work is to develop a web application that performs static code analysis and verifies detected defects using the large language model.

The following were used during the work: static code analysis and artificial intelligence methods, web development methods.

The practical value lies in automating the detection of vulnerabilities and code quality issues while reducing false positives through autonomous verification.

A web application has been developed that provides three-dimensional static analysis, production readiness score calculation, and fix guide generation.

Applications: production deployment preparation, source code security auditing, code review assistance

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект

API – Application Programming Interface (інтерфейс програмного застосунку, використовується для обміну даними між клієнтом і сервером)

CI/CD – Continuous Integration / Continuous Deployment (безперервна інтеграція / безперервне розгортання, автоматизований процес тестування та розгортання коду)

CRUD – Create, Read, Update, Delete (базові операції з даними: створення, читання, оновлення, видалення)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту, основний протокол для обміну даними в інтернеті)

JSON – JavaScript Object Notation (формат обміну даними, який використовується для передачі структурованої інформації)

OAuth – відкритий протокол авторизації, що дозволяє користувачам надавати доступ до свого облікового запису третій стороні без передачі пароля

ORM – Object-Relational Mapping (об’єктно-реляційне відображення, бібліотека для взаємодії з базою даних через об’єкти коду замість прямих SQL-запитів)

SAST – Static Application Security Testing (статичне тестування безпеки програми, аналіз вихідного коду без його виконання)

SCA – Software Composition Analysis (аналіз складу програмного забезпечення, виявлення уразливостей у залежностях та бібліотеках)

SDK – Software Development Kit (набір інструментів розробника, що містить бібліотеки, документацію та приклади коду для інтеграції сервісу)

ВСТУП

Автоматизований контроль якості програмного забезпечення на сучасному етапі розділяється за двома основними напрямками: детермінований статичний аналіз та інтелектуальна верифікація вихідного коду. Статичний аналіз це це перевірка програмного продукту без його безпосереднього виконання на основі аналізу синтаксичної структури, правил та заздалегідь сформованих шаблонів. Прикладом може бути автоматичний пошук вразливостей, виявлення застарілих залежностей бібліотек чи перевірка відповідності написаного тексту архітектурним стандартам безпосередньо у середовищах розроблення застосунків.

Основне завдання інтелектуальної верифікації це одержання контекстного семантичного оцінювання виявлених дефектів за допомогою моделей штучного інтелекту. Мета такої верифікації може бути різна: як детальний аналіз конкретного підозрілого фрагмента коду у контексті всього файла, так і комплексне відсікання хибно-позитивних спрацьовувань статичного сканера. Завдання інтелектуальної верифікації є доповнюючим та завершальним стосовно завдання первинного детермінованого сканування. Області застосування – хмарні системи автоматичного рецензування коду, інтелектуальні асистенти розробників та інструменти автоматичної генерації інструкцій з виправлення програмних помилок.

Актуальність роботи полягає у необхідності створення веборієнтованого інструменту, який об’єднує швидкість сканування абстрактного синтаксичного дерева та аналітичні можливості великих мовних моделей для забезпечення доступного контролю якості й безпеки вебзастосунків. Це дозволить авторам без глибокої технічної підготовки виявляти та усувати дефекти у вихідному коді без необхідності локального розгортання складного інфраструктурного програмного забезпечення.

1 АНАЛІЗ СУЧАСНОГО СТАНУ КОНТРОЛЮ ЯКОСТІ ТА БЕЗПЕКИ ПРОГРАМНОГО КОДУ

1.1 Аналіз предметної галузі

Сучасна індустрія розробки програмного забезпечення переживає трансформацію, безпрецедентну за своїми масштабами та темпами. Поява великих мовних моделей (Large Language Models, LLM), здатних генерувати синтаксично коректний та функціонально працездатний програмний код за описом звичайною мовою, призвела до формування принципово нової парадигми розробки. У лютому 2025 року відомий дослідник у галузі штучного інтелекту Андрій Карпатий запропонував термін «vibe coding» (укр. «vibe-програмування», або «інтуїтивне програмування»), який описує підхід, за якого автор повністю делегує написання програмного коду штучному інтелекту. Стрімкість поширення явища підтверджується тим, що словник Collins визнав термін «vibe coding» словом року 2025 [1], а за даними GitHub станом на 2026 рік близько 46% усього нового коду, що потрапляє до публічних репозиторіїв, згенеровано штучним інтелектом [2].

Ключовою особливістю нинішнього етапу є те, що процес розробки вийшов далеко за межі професійної спільноти. ШІ-асистована розробка дозволила особам без формальної технічної освіти (підприємцям, дизайнерам, студентам непрофільних спеціальностей) самостійно створювати повноцінні застосунки. Спеціалізовані платформи, такі як Lovable, Bolt.new та Cursor надали цій аудиторії інструменти для генерації застосунків з клієнтською, серверною частиною та базою даних на основі коротких описів, сформульованих без використання спеціалізованої термінології. Таким чином у галузі сформувалася принципово нова категорія авторів програмного забезпечення це люди, які ніколи не вивчали програмування як дисципліну, не читають згенерований код та не мають фундаментальної підготовки для оцінки його якості чи безпеки.

Саме ця зміна аудиторії визначає гостроту ситуації. Історично автор коду був одночасно його першим рецензентом: професійний розробник читав те, що пише, впізнавав типові антипатерни, знав базові практики безпеки. Користувач, який повністю передає написання коду ШІ-інструментам, у багатьох випадках взагалі не відкриває файли вихідного коду, а взаємодіє з проєктом лише через запити до штучного інтелекту та спостереження за поведінкою застосунку у браузері. Дослідження 2025-2026 років показують, що понад 40% розробників-початківців визнають факт розгортання ШІ-згенерованого коду, який вони не розуміють у повному обсязі [3]. Для суто нетехнічної аудиторії частка таких ситуацій наближається до стовідсоткової за визначенням. У результаті з процесу розробки повністю виключається традиційний захисний шар – людська перевірка коду перед його публічним запуском.

На тлі цієї зміни аудиторії питання безпеки набуло гостроти, якої галузь не знала раніше. За даними звіту CodeRabbit, код згенерований штучним інтелектом містить у 1,7 рази більше проблем порівняно з кодом написаним людиною, при цьому кількість проблем безпеки вища у 2,74 рази [4]. У незалежних дослідженнях фіксується, що близько 45% ШІ-згенерованого коду містить вразливості безпеки [5]. Окремою проблемою є «галюцинації пакетів», коли модель посилається на неіснуючі бібліотеки, та схильність ШІ пропонувати спрощені, але небезпечні архітектурні рішення: публічно доступні бази даних замість автентифікації, зберігання секретів у клієнтському коді, відсутність перевірки прав доступу до об’єктів. Показовим є те, що в одному з масових аудитів близько 10% обстежених ШІ-згенерованих застосунків містили критичні вразливості, які дозволяли отримати прямий доступ до персональних даних без автентифікації [3]. Реальні інциденти 2025-2026 років з платформами, побудованими на такому підході, вже продемонстрували повний набір наслідків: витік записів користувачів, компрометацію адміністративних панелей, публічне розкриття внутрішньої архітектури проєктів, втрату довіри

та репутаційні кризи. Звіт Snyk щодо стану безпеки відкритого коду за 2025 рік [6] фіксує суттєве зростання середнього часу усунення критичних вразливостей у відкритих проєктах, що частково пояснюється збільшенням частки авторів без досвіду роботи із засобами безпеки.

Переважна більшість ШІ-згенерованих застосунків належить до категорії веброзробки, яка залишається домінуючим напрямом створення програмних продуктів у сучасній індустрії. Це зумовлено зрозумілістю та універсальністю вебплатформи: готовий продукт доступний користувачеві через браузер без встановлення, а базові можливості вебзастосунків (форми, сторінки, API) добре співпадають із запитами, які нетехнічні автори здатні сформулювати без знання технічних термінів. Серед серверних технологій абсолютним лідером є середовище виконання Node.js разом із мовою TypeScript: за даними звіту GitHub Octoverse 2025, TypeScript вперше став найпопулярнішою мовою на GitHub, обігнавши Python та JavaScript, а переважна більшість сучасних фреймворків (Next.js, NestJS, Express) та платформ для ШІ-генерації застосунків використовують саме цю технологічну базу за замовчуванням [2]. Саме тому застосунки на базі Node.js є репрезентативним об’єктом дослідження у контексті розробки за допомогою штучного інтелекту.

Методи статичного аналізу коду ґрунтуються на аналізі абстрактних синтаксичних дерев (Abstract Syntax Tree, AST) – ієрархічних структур, що відображають синтаксичну будову програми без її виконання [7]. Перевагою AST-аналізу є здатність виявляти структурні патерни, які неможливо надійно знайти за допомогою регулярних виразів: наприклад, виклики функцій з конкретними аргументами, ланцюжки методів, патерни деструктуризації або умовні присвоєння. Водночас AST-аналізатори традиційно страждають від підвищеного рівня хибно-позитивних спрацьовувань, оскільки вони не розуміють семантичного контексту конкретної кодової бази. За різними оцінками, від 25 до 35 % попереджень SAST-інструментів є хибними [8, 19],

що призводить до так званої «втоми від попереджень» – систематичного ігнорування результатів аналізу розробниками.

Паралельно з описаною трансформацією процесу розробки існує розвинена екосистема інструментів контролю якості та безпеки програмного коду. Вона включає кілька основних категорій: інструменти синтаксичного аналізу та засоби уніфікації стилю коду (ESLint, Prettier), платформи статичного аналізу безпеки застосунків (Static Application Security Testing, SAST), інструменти аналізу складу програмного забезпечення (Software Composition Analysis, SCA) а також категорію ШІ-асистентів для перегляду коду, що активно розвивається (CodeRabbit, GitHub Copilot Code Review, Greptile).

Проте ключовою рисою всіх перелічених інструментів є те, що вони проєктувалися під професійну аудиторію. Вони передбачають, що користувач розуміє поняття статичного аналізу, здатен інтерпретувати повідомлення про вразливості, має технічну можливість налаштувати процес автоматизованої інтеграції та розгортання і самостійно написати виправлення після отримання звіту. Для нової аудиторії авторів застосунків, які за визначенням не володіють жодною з цих компетенцій, наявний інструментарій виявляється практично неприступним бо саме тут формується розрив між реальною потребою в контролі якості згенерованого коду та можливостями існуючих рішень.

Поряд із цим окремою сферою розглядається концепція виробничої придатності (production compliance) як цілісної характеристики готовності застосунку до промислової експлуатації. Запровадження такої метрики є необхідним для надання користувачеві простої й однозначної відповіді на ключове запитання: чи готовий застосунок до використання в промисловому середовищі. Характеристика об’єднує три взаємопов’язані виміри: безпеку, надійність, якість коду.

У контексті зазначених тенденцій розробка спеціалізованого інструменту автоматизованого аналізу виробничої придатності

Node.js-застосунків, орієнтованого на нову аудиторію нетехнічних авторів коду, є актуальним завданням. Такий інструмент має ліквідувати розрив між масовим поширенням розробки за допомогою штучного інтелекту серед людей без технічної підготовки та наявним професійним інструментарієм контролю якості, який цим людям фактично недоступний.

1.2 Виявлення проблем та актуалізація рішень

Після детального аналізу предметної галузі автоматизованого аналізу застосунків, що виконуються в середовищі Node.js, важливо дослідити існуючі рішення на ринку, щоб виявити їхні обмеження та визначити напрямки вдосконалення. Розглянемо три провідні інструменти, що представляють різні категорії підходів до контролю якості коду, та їхні особливості.

SonarQube є одним із найпоширеніших представників категорії платформ статичного аналізу коду та безпеки застосунків, розроблюваним компанією SonarSource. Він представляє собою комплексну платформу для аналізу якості коду з підтримкою понад тридцяти мов програмування, включаючи JavaScript та TypeScript. Ключовими перевагами SonarQube є широкий набір вбудованих правил, інтеграція з популярними CI/CD-системами, механізм відстеження технічного боргу у вигляді грошового еквіваленту часу на його усунення, а також розвинена система управління якістю через механізм Quality Gates. Однак, незважаючи на технічну зрілість платформи, вона має низку суттєвих недоліків у контексті ШІ-асистованої розробки. По-перше, SonarQube орієнтований на корпоративне використання та потребує складного розгортання: самостійне рішення вимагає окремого серверу з базою даних, Docker-інфраструктури та налаштування через конфігураційні файли. По-друге, безкоштовна версія платформи (SonarQube Community Build) не включає критично важливі

механізми, що знижує її ефективність для виявлення реальних вразливостей безпеки. Повноцінний функціонал доступний лише у платних ліцензіях з непропорційно високою вартістю для індивідуальних користувачів. По-третє, SonarQube не оцінює готовність застосунку до промислової експлуатації як цілісну характеристику, ігноруючи такі аспекти, як наявність механізмів перевірки працездатності системи (health check), коректне налаштування CORS чи відсутність зберігання стану у пам’яті процесу. Нарешті, інтерфейс платформи передбачає розуміння понять статичного аналізу та потребує від користувача технічної підготовки для інтерпретації результатів, що робить її непридатною для нетехнічної аудиторії.

Snyk, розроблений однойменною компанією, є одним із провідних інструментів категорії Software Composition Analysis з функціями SAST, широко використовуваним для виявлення вразливостей у залежностях проєкту та в коді застосунку. Його беззаперечними перевагами є постійно оновлювана база даних відомих вразливостей (Snyk Vulnerability Database), вбудована інтеграція з GitHub, GitLab та Bitbucket, можливість автоматичного створення pull request’ів із виправленнями залежностей, а також безкоштовна версія, доступна для індивідуальних розробників. Проте Snyk також має свої обмеження. Його основним фокусом є аналіз залежностей, а не власного коду застосунку: модуль Snyk Code, що відповідає за SAST, поступається спеціалізованим платформам у глибині аналізу та не розрізняє архітектурні антипатерни, зокрема відсутності обмеження кількості запитів (rate limiting) або некоректного налаштування політики CORS. Крім того, Snyk традиційно демонструє високу кількість хибно-позитивних спрацьовувань у модулі SAST, що створює додаткове навантаження на користувача у вигляді фільтрації нерелевантних результатів. Безкоштовна версія обмежена за кількістю тестів на місяць, що робить її непридатною для регулярного використання на активному проєкті. Платформа не надає автоматично генерованих виправлень коду для виявлених проблем застосунку (лише для залежностей), що залишає

користувача наодинці з необхідністю самостійно писати виправлення. Ареалом використання Snyk залишається корпоративне середовище з командами, що мають технічну кваліфікацію для роботи з результатами аналізу.

CodeRabbit позиціонується як ШІ-асистент для аналізу коду нового покоління, що використовує великі мовні моделі для перегляду запитів на злиття (pull request) у GitHub та GitLab. Його інтеграція безпосередньо з процесом аналізу коду, здатність залишати контекстні коментарі та генерувати читабельні пояснення проблем роблять CodeRabbit привабливим для сучасних команд розробки. Bacchelli та Bird [10] встановили, що контекстне коментування в процесі перегляду коду суттєво підвищує ефективність усунення дефектів у порівнянні з традиційними звітами статичного аналізу, що підтверджує цінність підходу CodeRabbit для досвідченої аудиторії. Однак, на відміну від його сильних можливостей у сфері аналізу на рівні окремих змін коду, функціональність CodeRabbit для цілісної оцінки виробничої придатності проєкту залишається обмеженою. По-перше, інструмент орієнтований на інкрементальний аналіз запитів на злиття (pull request), а не на комплексне сканування всієї кодової бази: для оцінки вже існуючого застосунку, який ніколи не проходив перевірку, CodeRabbit не є придатним. По-друге, стабільність результатів перевірки за допомогою штучного інтелекту є обмеженою: той самий фрагмент коду може отримати різні зауваження у різних запусках через стохастичну природу великих мовних моделей, що унеможливлює формальну відтворюваність аналізу. По-третє, подання повного контексту проєкту у засоби штучного інтелекту призводить до непрогнозованих результатів: модель може пропускати очевидні проблеми, вигадувати неіснуючі дефекти (галюцинації) або давати непослідовні оцінки у різних запусках. Вартість регулярного використання CodeRabbit для індивідуального користувача є значною через модель, орієнтовану на корпоративні команди. Порівняння інструментів наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз існуючих інструментів контролю якості коду

Інструмент	**Категорія**	**Ключові переваги**	**Обмеження для нетехнічних авторів**
SonarQube	SAST-платформа	Широкий набір правил (30+ мов); інтеграція з CI/CD; відстеження технічного боргу; механізм Quality Gates	Складне серверне розгортання; критичні функції безпеки лише в платній версії; не оцінює виробничу придатність.
Snyk	SCA + SAST	Актуальна база вразливостей залежностей; автоматичні PR.	Основний фокус на залежностях, а не на коді;
CodeRabbit	ШІ-асистент перегляду коду	Контекстні коментарі до pull request; читабельні пояснення проблем; інтеграція з GitHub/GitLab	Призначений для аналізу PR, а не повного сканування; нестабільність результатів через природу ШІ; висока вартість; не оцінює готовність до промислової експлуатації

Розглядаючи ширший ландшафт суміжних досліджень, слід відзначити роботу Brito та ін. [8], у якій систематизовано інструменти статичного аналізу для Node.js-пакетів та виявлено суттєві розбіжності між ними в повноті та точності виявлення вразливостей. Kula та ін. [11] досліджують практики оновлення залежностей і встановлюють, що переважна більшість проєктів залишає вразливі залежності без оновлення навіть через тривалий час після публікації виправлень. Thongtanunam та ін. [12] аналізують практики перегляду коду у вебзастосунках та демонструють, що вебспецифічні вразливості (некоректна авторизація, проблеми CORS, відсутність rate limiting) суттєво рідше виявляються під час звичайного перегляду коду, ніж загальні дефекти. Зазначені дослідження підтверджують, що поєднання детермінованого статичного аналізу з семантичним контекстом є перспективним напрямком підвищення ефективності автоматизованих систем безпеки.

Аналіз цих інструментів виявляє наступні проблеми, які потрібно вирішити:

- баланс між простотою використання та функціональною повнотою: існуючі рішення або вимагають складного налаштування та технічної підготовки користувача, або не забезпечують достатньої повноти аналізу для цілісної оцінки виробничої придатності застосунку;
- недостатня орієнтація на нетехнічну аудиторію. Більшість інструментів розраховані на досвідчених розробників у корпоративних середовищах та використовують інтерфейси, термінологію і процеси, непридатні для користувачів без формальної технічної освіти;
- відсутність комплексної оцінки виробничої придатності. Традиційні SAST-інструменти зосереджені на безпеці, SCA-інструменти на залежностях, статичні аналізатори на стилі коду. Жоден з них не оцінює готовність застосунку до промислової експлуатації як єдину характеристику, що об’єднує безпеку, надійність, продуктивність та якість коду;

- дилема точності та повноти у статичному аналізі. Традиційні SAST-інструменти страждають від великої кількості хибно-позитивних спрацьовувань (за різними оцінками, близько третини попереджень), що призводить до «втоми від попереджень» та втрати довіри користувача, тоді як повністю ШІ-орієнтовані рішення демонструють непрогнозовану поведінку та галюцинування;
- обмежена автоматизація виправлення проблем. Більшість інструментів обмежуються виявленням дефектів, залишаючи їх усунення на користувача, що для нетехнічної аудиторії є непереборною перешкодою.

На основі проведеного аналізу можна виділити ключові вимоги до нової програмної системи автоматизованого аналізу виробничої придатності Node.js-застосунків:

- простота використання та відсутність необхідності попереднього налаштування як головний пріоритет при розробці системи (на відміну від корпоративних SAST-платформ);
- використання сучасного вебінтерфейсу з інтуїтивно зрозумілою взаємодією через GitHub OAuth, без локального встановлення та конфігураційних файлів;
- комплексна оцінка виробничої придатності за трьома вимірами – безпекою, надійністю і продуктивністю, якістю коду – з єдиним інтегральним показником (Production Compliance Score);
- дворівнева архітектура виявлення проблем, що поєднує детермінований статичний аналіз для широкого охоплення потенційно підозрілих фрагментів із подальшою верифікацією за допомогою ШІ для усунення хибно-позитивних результатів;
- підтримка конкретних бібліотек та фреймворків Node.js-екосистеми через попередній етап контекстного сканування, що дозволяє застосовувати правила, адаптовані до технологічного стеку проєкту;
- автоматизована генерація структурованих інструкцій з виправлення у форматі Markdown, придатних для безпосередньої передачі ШІ-агентам.

1.3 Постановка задачі

Автоматизація статичного аналізу та контролю безпеки коду вебзастосунків є актуальним завданням через стрімкий розвиток інструментів штучного інтелекту для генерації програм. Зниження порогу входження в індустрію розробників без глибоких технічних знань призводить до зростання кількості вразливостей та архітектурних дефектів у програмних продуктах. Тому ставиться завдання розробити хмарний вебзастосунок для комплексного оцінювання відповідності вихідного коду вимогам промислової експлуатації.

Об’єктом роботи є процес автоматизованого статичного аналізу, інтелектуальної верифікації та багатовимірного оцінювання відповідності вихідного коду сучасних вебзастосунків встановленим галузевим стандартам і вимогам безпечної промислової експлуатації. Відповідно до стандарту CVSS v3.1 [9], серйозність виявлених вразливостей класифікується за чотирма рівнями (Low, Medium, High, Critical), що відображається у механізмі оцінювання розробленої системи.

Метою роботи є моделювання, архітектурне проєктування та програмна реалізація веборієнтованої системи для інкрементального аналізу вихідного коду на основі інноваційної дворівневої гібридної архітектури, яка поєднує детерміноване сканування абстрактного синтаксичного дерева з інтелектуальною контекстною верифікацією за допомогою великої мовної моделі Gemini 2.5 Flash для суттєвого підвищення точності аналізу та мінімізації хибних спрацьовувань. Дослідження Bohdan та ін. [13] та Suprun та ін. [14] засвідчують, що поєднання детермінованих методів аналізу з контекстними можливостями великих мовних моделей забезпечує якісно кращі результати, ніж застосування кожного підходу окремо, що підтверджує обґрунтованість обраної гібридної архітектури.

Для досягнення поставленої мети та забезпечення логічної повноти дослідження необхідно вирішити такі конкретні завдання:

- сформулювати детальні технічні вимоги до системи, визначити критерії вибору технологічного стеку та спроєктувати веборієнтований адаптивний інтерфейс користувача з інтеграцією протоколу безпечної авторизації GitHub Open Authorization для забезпечення безперешкодного доступу нетехнічних авторів безпосередньо до аналізу репозиторіїв;
- дослідити методи побудови та аналізу абстрактних синтаксичних дерев для мови програмування JavaScript та середовища Node.js, а також розробити перший рівень архітектури – швидкий детермінований сканер для первинного виявлення потенційно небезпечних конструкцій у коді;
- розробити другий рівень архітектури – інтелектуальний модуль верифікування на основі штучного інтелекту, що взаємодіє з API моделі Gemini 2.5 Flash через систему оптимізованих бінарних запитів, метою якого є детальний контекстний аналіз виявлених підозрілих фрагментів та ефективне відсікання хибно-позитивних результатів статичного сканування;
- реалізувати алгоритми контекстного сканування для автоматичного визначення технологічного стеку аналізованого проєкту (зокрема, специфіки взаємодії з об'єктно-реляційними та документо-орієнтованими бібліотеками Prisma, Mongoose, Express), що дозволить динамічно адаптувати каталог правил детекції під конкретну структуру коду;
- обґрунтувати, формалізувати та програмно реалізувати математичну модель розрахунку інтегрального показника виробничої придатності проєкту (Production Compliance Score);
- розробити інтелектуальний генератор структурованих покрокових інструкцій та рекомендацій з виправлення виявлених дефектів програмного коду, адаптованих для легкого сприйняття нетехнічними розробниками застосунків;
- спроєктувати та впровадити схему зберігання результатів інкрементального сканування у базі даних MongoDB із реалізацією механізмів підтримки детальної історії аналізів, що необхідно для

довготривалого моніторингу та відстеження динаміки змін виробничої придатності програмного проєкту;

– виконати комплексне розгортання, профілювання та тестування працездатності створеного програмного забезпечення, провести експериментальне оцінювання точності та ефективності розробленого гібридного підходу на базі реальних наборів вихідного коду, а також сформулювати шляхи подальшого розвитку отриманих результатів.

Практична цінність розробки полягає у тому, що вона безпосередньо адресує виявлені прогалини в існуючому інструментарії: надає безбар’єрний доступ до аналізу якості та безпеки коду через веб-інтерфейс, формує єдиний зрозумілий показник виробничої придатності та генерує конкретні покрокові інструкції з виправлення, придатні для безпосереднього використання нетехнічними авторами вебзастосунків. Таким чином, результати роботи безпосередньо знижують технічний бар’єр між масовою ШІ-асистованою розробкою та промисловою якістю кінцевих програмних продуктів.

2 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ЗАСТОСУНКУ

2.1 Вибір та обґрунтування архітектурного стилю системи

Перш ніж переходити до детального проєктування програмного забезпечення, необхідно визначити загальний архітектурний стиль системи, оскільки саме він задає базові обмеження та можливості, у межах яких приймається решта проєктних рішень. Архітектурний стиль визначає, у який спосіб система поділяється на компоненти, яким чином ці компоненти взаємодіють між собою, де проходять межі відповідальності та як забезпечуються нефункціональні вимоги: масштабованість, надійність, безпека та зручність супроводу.

Перед розробкою системи було сформульовано низку функціональних та нефункціональних вимог, які прямо вплинули на вибір архітектурного стилю. До функціональних вимог належить можливість підключення довільного публічного або приватного репозиторію GitHub, автоматичне визначення набору технологій проєкту, виконання статичного аналізу за трьома незалежними напрямами (безпека, продуктивність, якість коду), верифікування отриманих кандидатів на дефекти за допомогою великої мовної моделі, формування оцінки готовності до промислової експлуатації та генерація документа з рекомендаціями щодо виправлення знайдених проблем. До нефункціональних вимог віднесено стійкість до тривалих операцій сканування (до кількох хвилин), здатність обслуговувати кількох користувачів одночасно без блокування ресурсів сервера, відокремлення обчислювально-важких процесів від обробки HTTP-запитів, можливість підміни постачальника послуг штучного інтелекту без зміни ядра системи, а також забезпечення сучасних практик безпеки під час зберігання конфіденційних даних користувача (ключів доступу та зовнішніх сервісів).

При проєктуванні було розглянуто три потенційні архітектурні підходи. Першим є монолітна архітектура, у межах якої всі компоненти

системи виконуються в одному процесі та діляться спільним адресним простором. Її головною перевагою є простота розгортання та налагодження, проте під час тривалих сканувань, що можуть займати десятки секунд і більше, монолітний процес був би повністю заблокованим, що унеможливлює одночасне обслуговування інших користувачів. Другим розглянутим підходом була мікросервісна архітектура, у якій кожна функціональна область виконується як окремий сервіс із власною базою даних та незалежним розгортанням. Однак для системи описаного масштабу мікросервісний підхід вносить непропорційно високу складність: необхідність розподілених транзакцій, узгодження схем між сервісами, окрема інфраструктура спостережуваності без відповідної винагороди у вигляді покращення продуктивності або надійності. Третім підходом, який і було обрано, є клієнт-серверна архітектура з асинхронними фоновими обробниками. Вона поєднує простоту монолітного розгортання з відокремленням довготривалих обчислювальних операцій від синхронного HTTP-шару, що повністю задовольняє нефункціональні вимоги без надмірного ускладнення системи.

Обраний архітектурний стиль формально можна охарактеризувати як трирівневу клієнт-серверну архітектуру з фоновими виконавцями завдань (workers) та зовнішнім посередництвом до сервісів штучного інтелекту. Перший рівень – це клієнтський застосунок, який відповідає за подання інформації, маршрутизацію між сторінками, керування формами та локальним станом. Другий рівень – серверна частина у вигляді HTTP-API, яка відповідає за авторизацію, валідацію вхідних даних, операції створення, читання, оновлення, видалення, розміщення завдань у чергах та повернення сформованих об’єктів передачі даних. Третій рівень – фонові обробники, які споживають завдання з черги, виконують операції тривалого аналізу та зберігають результат у спільну базу даних. Окремо до архітектури включено зовнішні сервіси: ідентифікаційний постачальник GitHub, який одночасно є джерелом вихідного коду, та постачальник великих мовних моделей

Google Gemini, який виконує верифікування кандидатів на дефекти та генерацію рекомендацій.

У сукупності описаний архітектурний стиль доцільно характеризувати як шарову клієнт-серверну архітектуру з конвеєрною обробкою аналізу та модульною інтеграцією зовнішніх постачальників ШІ. Така характеристика заздалегідь визначає чотири ключові архітектурні рішення, які буде детально розглянуто далі: відокремлення синхронного HTTP-шару від асинхронного шару аналізу, шарова організація серверного коду, конвеєрна організація аналітичного рушія та підмінні постачальники ШІ за єдиним контрактом.

2.2 Огляд архітектури системи

Архітектура системи побудована навколо трьох незалежних виконуваних компонентів, що працюють паралельно та обмінюються даними через спільну базу даних і чергу повідомлень. Такий поділ є основою для забезпечення відмовостійкості та горизонтального масштабування окремих частин системи без впливу на інші. На рисунку 2.1 наведено високорівневу схему компонентів системи та зв’язків між ними.

Першим компонентом системи є клієнтський застосунок, розроблений із використанням технології Next.js, що забезпечує формування окремих сторінок на стороні сервера. Він взаємодіє із серверною частиною виключно через захищений канал HTTPS, передаючи запити у форматі JSON. Уся авторизація користувача виконується через cookie-сесії, які встановлюються після успішного проходження OAuth-процедури з постачальником послуг GitHub. Серверний стан передається в інтерфейс через бібліотеку Tanstack, яка забезпечує кешування та запити до API у фоновому режимі.

Другим компонентом є серверний HTTP-API, розроблений на платформі Node.js із використанням фреймворку Express 5.

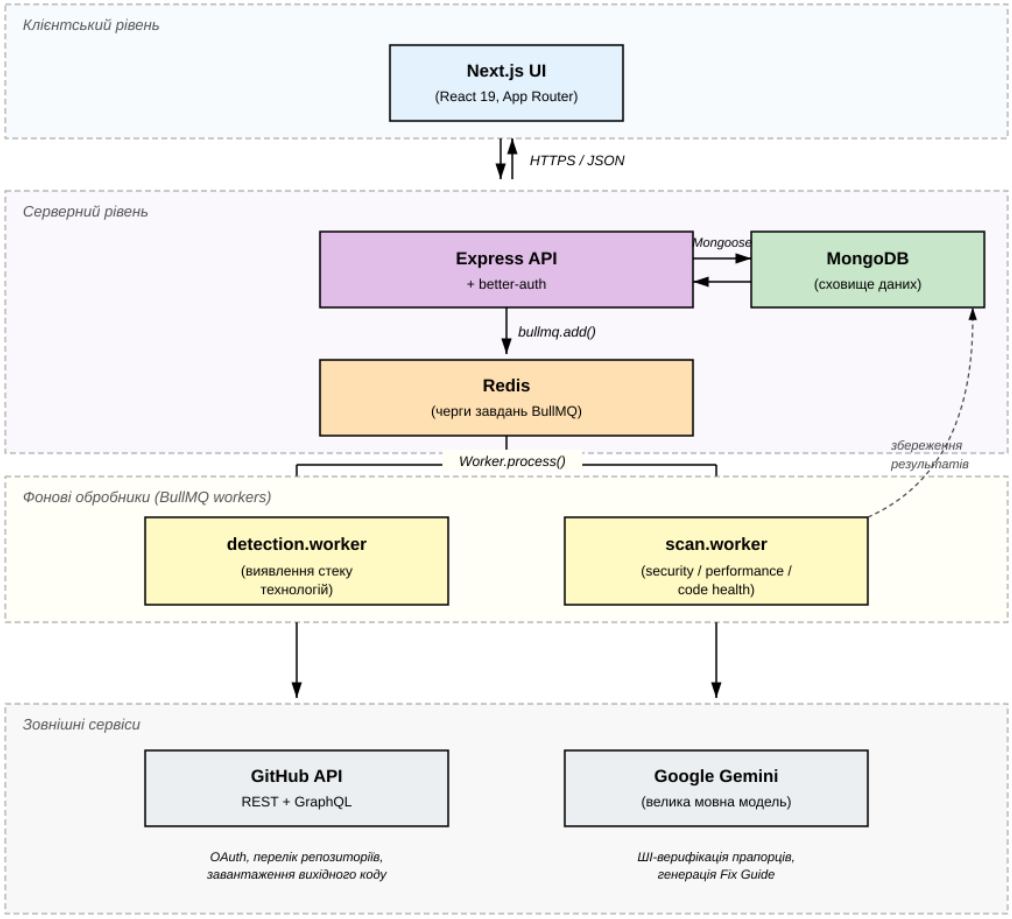

Рисунок 2.1 – Високорівнева архітектура системи

Цей компонент служить точкою входу для взаємодії системи з зовнішніми клієнтами та відповідає за маршрутизацію запитів, перевіряння вхідних даних, реалізацію бізнес-логіки застосунку, формування завдань для фонової обробки та повернення серіалізованих результатів клієнту. Серверна частина не виконує самостійно тривалих операцій аналізу – натомість делегує цю роботу спеціалізованим обробникам через систему черг повідомлень.

Третім компонентом є фонові обробники завдань, реалізовані на основі бібліотеки BullMQ, яка використовує базу даних Redis. У системі визначено два окремих обробники: визначення набору технологій та сканування репозиторію. На поточному етапі обидва обробники запускаються в тому ж процесі, що й HTTP-сервер, однак архітектура системи черг уже передбачає можливість їх перенесення в окремі контейнери чи процеси без будь-яких

змін у коді – це є важливою властивістю горизонтальної масштабованості системи.

Четвертим архітектурним елементом є сховище даних – нереляційна база MongoDB, яка зберігає всі сутності прикладної області: користувачі, проєкти, контексти проєктів, сканування, прапорці, рекомендації та зашифровані ключі API. Доступ до бази даних здійснюється виключно через бібліотеку Mongoose, яка надає типізовану обгортку над колекціями та перевіряє дані на рівні схеми перед записом.

П’ятим елементом є зовнішні служби, з якими взаємодіє система. До них належать GitHub (одночасно як ідентифікаційний постачальник OAuth, постачальник переліку репозиторіїв та джерело вихідного коду) і Google Gemini (як постачальник великої мовної моделі для верифікування прапорців та генерації документів з рекомендаціями). Контракти взаємодії з цими сервісами інкапсульовано в окремих інтерфейсах на серверному рівні, що дозволяє в майбутньому замінити їх іншими постачальниками без впливу на бізнес-логіку.

2.3 Проєктування серверної частини застосунку

Серверна частина є центральним компонентом усієї системи: вона приймає всі HTTP-запити від клієнта, виконує перевіряння вхідних даних, реалізує бізнес логіку, керує постановкою завдань у фонові черги та контролює доступ до сховища даних. У цьому розділі послідовно розглянуто набір технологій, обраний для серверної частини, та її внутрішнє проєктування: шарову архітектуру, файлову структуру, організацію маршрутизації, контролерів, сервісів і моделей даних.

Серверна частина побудована на платформі Node.js, що дозволяє використовувати ту саму мову програмування (TypeScript), що й на клієнті, та обмінюватися спільними типами. Як HTTP-фреймворк використано

Express 5 – мінімалістичний, гнучкий та широко поширений у промисловості фреймворк. Перевага Express полягає у тому, що він не нав’язує жодних архітектурних рішень, залишаючи розробнику повну свободу в організації коду. Це особливо корисно у поєднанні з власною шаровою архітектурою, яку описано далі в цьому розділі.

Як основне сховище даних обрано MongoDB разом із бібліотекою Mongoose. Документна модель MongoDB є природним вибором для системи з кількох причин. По-перше, ключовий об’єкт ProjectContext містить різнорідну метаінформацію (виявлені фреймворки, бібліотеки аутентифікації, бази даних, ORM-системи, бібліотеки логування тощо), яку важко змоделювати у вигляді жорсткої реляційної схеми. По-друге, прапорці (позначувачі помилок) однієї перевірки можуть мати різні додаткові поля: поле context може містити вміст файла, фрагмент рядків, тіло функції тощо. По-третє, операція з найбільшим навантаженням – це масова вставка прапорців, отриманих під час одного сканування, що в MongoDB виконується одним викликом «insertMany».

Для асинхронної обробки завдань використано Redis як сховище черг та BullMQ як прикладну бібліотеку для роботи з ними. BullMQ забезпечує надійну доставку завдань, автоматичні повторні спроби в разі тимчасових помилок, підтримку відкладеної обробки, ізоляцію обробників за іменами черг, а також можливість збору метрик про швидкість обробки та кількість завдань у черзі. Альтернативою могло б бути використання простих JavaScript-проміжків та setInterval або інтеграція повноцінного брокера на кшталт RabbitMQ, проте перший варіант не забезпечує надійності, а другий є надмірним для системи поточного масштабу.

Для аутентифікації використано бібліотеку better-auth із MongoDB-адаптером. Це сучасна типізована бібліотека, яка підтримує OAuth-постачальники з коробки, керує сесіями через cookie та надає клієнтський SDK для зручного використання у Next.js. Альтернативою могло б бути NextAuth, проте better-auth забезпечує тіснішу інтеграцію з кастомним

Express-сервером (вона надає express-middleware) і простіший доступ до моделі даних.

Для виконання HTTP-захисту використано набір стандартних посередників: helmet (для встановлення безпечних заголовків), cors (для контролю міждоменних запитів), express-rate-limit (для обмеження частоти запитів), compression (для стиснення відповідей gzip), hpp (для захисту від HTTP Parameter Pollution). Для валідування вхідних даних обрано бібліотеку Zod, яка дозволяє декларативно описувати схеми даних і отримувати з них типізовані парсери. Для тестування серверної частини обрано Jest – найпоширеніший у JavaScript-екосистемі фреймворк модульного тестування, який підтримує імітацію залежностей, паралельне виконання тестів та збір покриття коду.

Узагальнено технологічний набір серверної частини представлено у таблиці 2.1.

Таблиця 2.1 – Технологічний стек серверної частини

Технологія / бібліотека	**Призначення**
1	**2**
Node.js + Express 5	Платформа виконання та HTTP-фреймворк
TypeScript	Типова безпека на всіх рівнях
MongoDB + Mongoose	Основне сховище даних, типізована обгортка
Redis + BullMQ	Черги асинхронних завдань
better-auth + MongoDB-адаптер	Серверна автентифікація через GitHub OAuth

Продовження таблиці 2.1

1	2
@octokit/rest + GraphQL	Інтеграція з GitHub
@google/genai	Інтеграція з Google Gemini
helmet, cors, express-rate-limit, compression, hpp	HTTP-захист
zod	Валідування вхідних даних
jest	Модульне тестування

Серверна частина організована у чотири основні шари, через які проходить кожен HTTP-запит: маршрутизація (routes), контролери (controllers), сервіси (services) та моделі даних (models). Такий підхід забезпечує тестованість, спрощує локалізацію змін і дозволяє новим розробникам швидко орієнтуватися в коді. Графічно ця структура зображена на рисунку 2.2.

Рисунок 2.2 – Шарова архітектура серверної частини

Шар маршрутизації відповідає виключно за прив’язку HTTP-шляхів до відповідних обробників і застосування проміжного програмного забезпечення (middleware). У цьому шарі не міститься жодної бізнес-логіки - він складає лише декларативний опис того, який обробник відповідає за який маршрут і які проміжні обробники до нього застосовуються.

Шар контролерів отримує HTTP-запит і перевіряє вхідні дані за допомогою Zod-схем. Це критичний крок, який забезпечує, що сервіс працює лише з коректними даними. Після перевіряння контролер викликає відповідний метод сервісу, отримує результат, серіалізує його у JSON та повертає клієнту з відповідним HTTP-статусом. Така структура спрощує сервісний код, оскільки сервіс може покладатися на те, що отримані дані вже перевірені.

Шар сервісів є серцем серверної частини: саме тут міститься вся бізнес-логіка системи. Сервіс взаємодіє безпосередньо з моделями даних через бібліотеку Mongoose, виконує перевірки приналежності ресурсу користувачеві, ставить завдання в черги BullMQ та координує виклики до зовнішніх служб. Принципово сервіси абстраговані від HTTP – вони не взаємодіють безпосередньо з параметрами вебзапиту. Це означає, що той же сервіс легко перевикористовується в іншому контексті.

Шар моделей даних описує схеми колекцій MongoDB через бібліотеку Mongoose. Кожна модель експортує типізований клас із методами на кшталт find, findOne, insertMany, які гарантовано повертають об’єкти, що відповідають визначеному типу TypeScript. Це повністю усуває цілий клас помилок, пов’язаних із розбіжностями між фактичною формою даних у базі та типовою моделлю в коді.

У системі визначено п’ять основних модулів маршрутизації, що відповідають основним прикладним сценаріям користувача. Усі маршрути захищено проміжним програмним забезпеченням «requireAuth», за винятком маршрутів OAuth-обміну. Стислий перелік основних маршрутів подано у таблиці 2.2.

Таблиця 2.2 – Основні HTTP-маршрути серверної частини

Маршрут	**Призначення**	**Тип операції**
POST /projects	Створення проєкту з власних репозиторіїв	CRUD + черга
POST /projects/import	Імпорт публічного репозиторію за URL	CRUD + черга
GET /projects	Перелік проєктів користувача	Читання
GET /projects/:id	Детальна інформація про проєкт	Читання
GET /projects/:id/branches	Перелік гілок репозиторію	Зовнішнє API
POST /projects/:id/scans	Запуск сканування обраної гілки	CRUD + черга
GET /scans/:scanId	Стан і результат сканування	Читання
GET /scans/:scanId/flags	Пагінований перелік прапорців сканування	Читання
POST /projects/:id/scans/:scanId/fix-guides	Генерація документа з рекомендаціями	CRUD + ШІ
GET /projects/:id/fix-guides	Перелік документів з рекомендаціями	Читання
POST /api-keys	Збереження зашифрованого ключа ШІ-постачальник	CRUD
GET /api-keys	Перелік ключів користувача	Читання
DELETE /api-keys/:id	Видалення ключа	CRUD

Як видно з таблиці, частина маршрутів виконує лише операції CRUD, частина додатково ставить завдання у фонову чергу, частина залучає виклики до зовнішніх служб (GitHub або Gemini). Це відмінне розмежування дозволяє чітко прогнозувати, які запити будуть швидкими, а які потенційно повільними, та правильно проєктувати клієнтську частину з викликом довготривалих операцій.

Окремо слід зазначити, що перевірка приналежності ресурсу користувачеві виконується на рівні сервісного шару. Кожен метод автоматично перевіряє, що шуканий ресурс належить поточному користувачеві. Такий підхід відомий як авторизація на рівні власника ресурсу. Він гарантує, що користувач не може отримати доступ до чужих даних, навіть у разі помилки в іншій частині системи.

2.4 Проєктування клієнтської частини застосунку

Клієнтська частина реалізована як сучасний вебзастосунок, що забезпечує користувачеві зручний інтерфейс для роботи з GitHub-репозиторіями, керування скануваннями та перегляду результатів аналізу. У цьому розділі послідовно розглянуто технологічний стек клієнтської частини, її маршрутизацію, підхід до управління станом та організацію компонентів.

Як основний фреймворк було обрано Next.js із застосуванням маршрутизатора App Router. Цей вибір обґрунтовано кількома факторами. По-перше, Next.js забезпечує гібридний підхід до генерування контенту: окремі сторінки можуть генеруватися на сервері для покращення часу першого відображення, тоді як інтерактивні компоненти лишаються повноцінно клієнтськими. По-друге, App Router підтримує сучасну модель компонентів сервера, що дозволяє знизити обсяг JavaScript, який відправляється у браузер. По-третє, він забезпечує вбудовану файлову

маршрутизацію, що добре поєднується із функціонально-орієнтованою структурою папок.

Для стилізації використовується Tailwind у поєднанні з бібліотекою компонентів Shadcn. Tailwind було обрано як систему допоміжних класів, оскільки вона забезпечує передбачуваність стилів та мінімізує розмір CSS-файлів. Бібліотека shadcn/ui надає набір базових компонентів інтерфейсу, які копіюються у проєкт як вихідний код (а не як зовнішня залежність), що дозволяє при необхідності модифікувати їх без обмежень.

Узагальнено технологічний набір клієнтської частини представлено у таблиці 2.3.

Таблиця 2.3 – Технології клієнтської частини

Технологія / бібліотека	**Призначення**
1	**2**
Next.js 16 (App Router)	Фреймворк клієнтської частини з гібридним рендерингом
React 19	Бібліотека для побудови інтерфейсу користувача
TypeScript	Типова безпека та узгодженість контрактів із сервером
Tailwind CSS 4 + shadcn/ui	Стилізація та базові компоненти інтерфейсу
@tanstack/react-query	Управління серверним станом, кешування та запити до API
better-auth client SDK	Клієнтська частина автентифікації
axios	HTTP-клієнт для запитів до API

Продовження таблиці 2.3

1	2
react-markdown / streamdown	Відображення Fix Guide у форматі Markdown
lucide-react	Бібліотека векторних іконок

Next.js App Router використовує файлову систему як декларативний опис маршрутів: кожна папка у /client/app відповідає сегменту URL, а файл page.tsx у ній – конкретній сторінці. У проєкті виокремлено приватну зону маршрутів (private), доступну лише після автентифікації, та публічну зону (public), де розташовані сторінка входу та цільова сторінка. Усі приватні маршрути проходять через спільне обгортання, яке містить компонент AuthProvider – він зв’язує клієнт із сесією better-auth і надає всім дочірнім компонентам поточного користувача через React Context.

Принциповим архітектурним рішенням є відмова від глобальної бібліотеки управління станом (на кшталт Redux, Zustand, Jotai). Замість цього увесь стан умовно поділено на дві категорії: серверний стан (дані, які отримуються з API і кешуються) та локальний стан (тимчасові значення по типу активних вкладок, текстових полів форм). Серверний стан повністю покладено на «@tanstack/react-query», який забезпечує автоматичне кешування за ключем запиту, а також запити до API із заданим інтервалом. Це особливо важливо для довготривалих операцій сканування.

Клієнтська частина організована за принципом файлової маршрутизації App Router. Система включає публічні маршрути (вхід через OAuth), приватні маршрути з перегляду проєктів, сканувань та результатів, а також маршрути для управління даними (ключі API, перегляд файлів). Логіка кожної сторінки локалізована у відповідному компоненті, а спільні елементи UI винесено у /client/components. Структуру маршрутизації подано на рисунку 2.3.

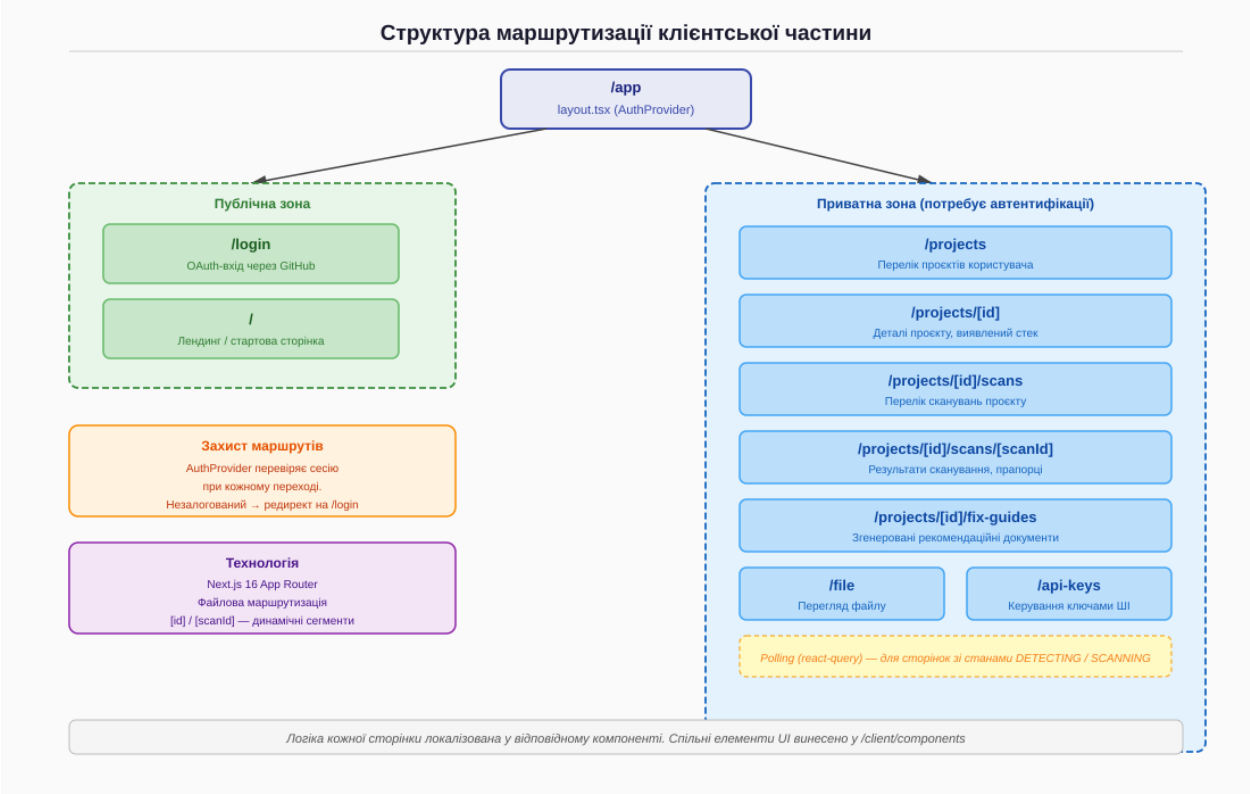

Рисунок 2.3 – Структура маршрутизації клієнтської частини

2.5 Проєктування моделі даних

Модель даних побудована на основі восьми основних колекцій MongoDB, кожна з яких відповідає чітко окресленій сутності. Центральними сутностями прикладної області є користувач (User), проєкт (Project) як зв’язаний репозиторій GitHub, контекст проєкту (ProjectContext) як результат визначення використаних в проєкті технологій, сканування (Scan) як окремий запуск аналізу однієї гілки репозиторію, прапорець (Flag) як одна виявлена проблема, документ рекомендацій (FixGuide) як згенерований ШІ Markdown-документ для виправлення обраної підмножини прапорців, та ключ API (ApiKey) як зашифрований доступ до постачальника ШІ. Окрім цього, бібліотека better-auth додає дві службові колекції – accounts (зв’язки OAuth-акаунтів) та sessions (активні сесії користувача).

Усі колекції описано через Mongoose-схеми з обов’язковою TypeScript-типізацією. Кожна модель визначає власний інтерфейс – IUser, IProject, IScan,

IFlag тощо. Інтерфейс одночасно використовується як на рівні схеми, так і на рівні об’єктів передачі даних, що повертаються сервісами. Завдяки цьому форма даних залишається узгодженою на всіх шарах системи. Зміна схеми колекції автоматично проявляється у вигляді помилок компіляції в усіх місцях, де відповідний тип використовується. Стратегія індексації побудована на основі профілю найчастіших запитів: у колекції projects визначено унікальний складений індекс за полями userId та repoId, у колекції flags – складений індекс за scanId та severity для перегляду даних порційними частинами з фільтрацією за рівнем серйозності, у колекції apiKeys – індекс за userId для швидкого пошуку активного ключа перед кожним скануванням.

Зв’язки між сутностями такі: один користувач має багато проєктів; кожен проєкт має один контекст (зв’язок 1:1) та багато сканувань (зв’язок 1:*N*); кожне сканування має багато прапорців і нуль чи більше документів рекомендацій; кожен документ рекомендацій посилається на конкретну підмножину прапорців через масив flagIds. Графічно цю структуру представлено на рисунку 2.4.

Стан проєкту моделюється явним скінченним автоматом через поле status у колекції projects. Кожен дозволений перехід зі стану в стан виконується конкретним сервісним методом, що запобігає неконсистентним переходам.

Проєкт створюється у стані «DETECTING» – у цей момент в чергу detection-queue ставиться завдання визначення набору технологій, використаних в проекті. Фоновий обробник detection.worker завантажує файлову структуру репозиторію, запускає детектори і за результатом переводить проєкт в один із трьох можливих наступних станів: «AWAITING_CONFIRMATION» (якщо набір технологій успішно визначено й систему підтримано), «UNSUPPORTED_ENVIRONMENT» (якщо репозиторій не містить підтримуваних мов та середовищ) або «DETECTION_FAILED» (у разі помилки виконання пошуку).

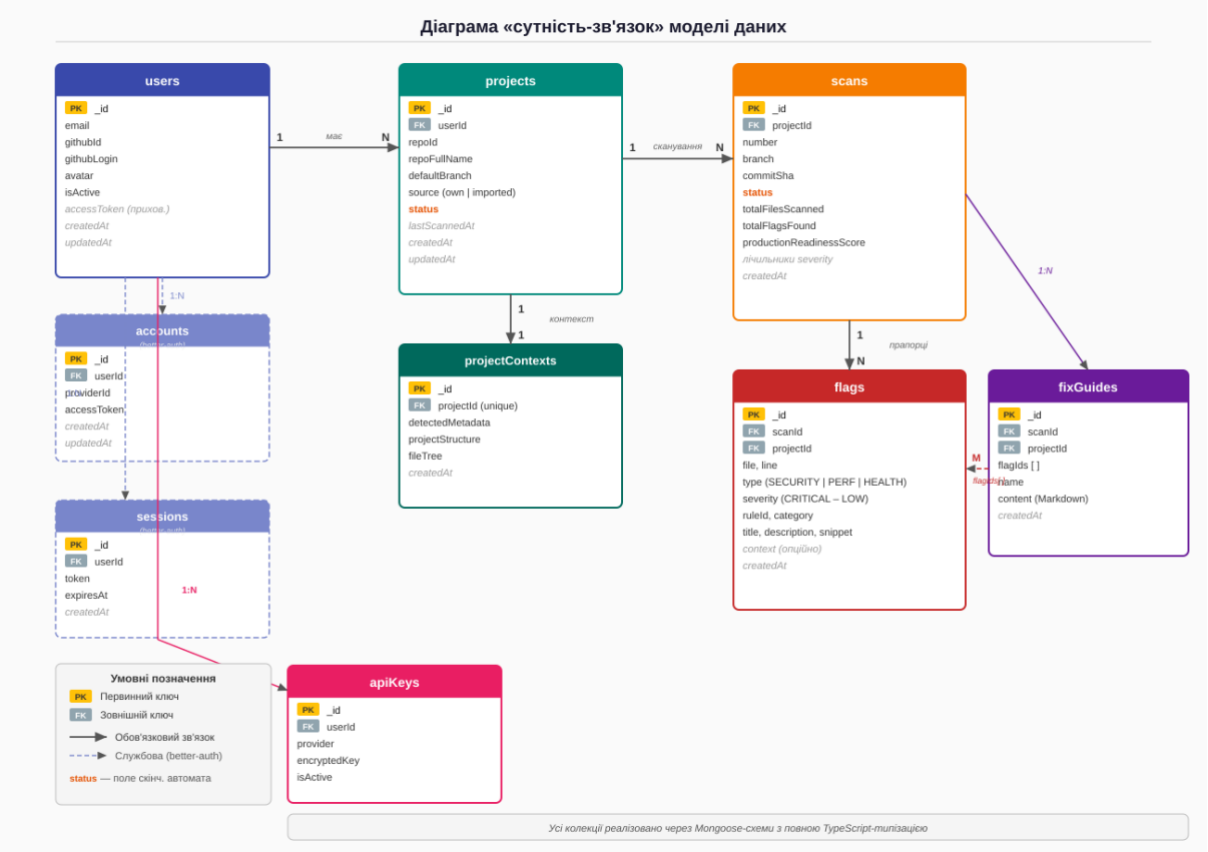

Рисунок 2.4 – Діаграма «сутність-зв’язок» моделі даних

Зі стану «AWAITING_CONFIRMATION» користувач обибирає гілку та запускає сканування – проєкт переходить у стан «SCANNING». Після завершення сканування він переходить у «COMPLETED» (успіх) або «SCAN_FAILED» (помилка). Зі стану «COMPLETED» новий запуск сканування знову переводить проєкт у «SCANNING», що дозволяє ітеративну роботу: користувач виправляє помилки, перезапускає сканування, отримує оновлений набір прапорців.

Окрім автомату станів проєкту, колекція scans має власне поле status, що моделює дрібніший автомат: «PENDING», «SCANNING», «COMPLETED» або «FAILED». Стан «PENDING» – це короткий початковий стан між створенням документа Scan та фактичним підняттям його обробником із черги. У стані «SCANNING» тривають завантаження файлів та виконання трьох конвеєрів аналізу. Перехід у «COMPLETED» виконується після успішного збереження всіх прапорців і обчислення оцінки готовності.

Перехід у «FAILED» у разі будь-якого винятку всередині фонового обробника. Окремо зазначається, що поле status проєкту та поле status сканування є двома різними автоматами і змінюються одночасно у відповідних сервісних методах.

2.6 Асинхронне оброблення завдань

Однією з ключових архітектурних особливостей системи є чітке відокремлення синхронного HTTP-шару від асинхронного шару виконання тривалих операцій. Основна операція системи (сканування репозиторію) за своєю природою є тривалою: вона включає десятки HTTP-запитів до GitHub для завантаження файлів, обробку отриманого вмісту регулярними виразами, паралельні виклики до Gemini для верифікування, обчислення оцінки готовності та масове збереження результатів у MongoDB. Сумарно процес може займати від десятків секунд до кількох хвилин, залежно від розміру репозиторію.

Якби сканування виконувалося синхронно в межах HTTP-запиту, система не змогла б обробляти кількох одночасних користувачів. У Node.js асинхронна операція блокує обробку інших запитів, і кожне сканування (десятки секунд до хвилин) паралізувало би весь сервер для інших клієнтів. По-друге, типові проміжні проксі та хмарні балансувальники мають ліміти часу очікування (зазвичай 30-120 секунд), які спрацювали б під час сканування великих репозиторіїв. По-третє, у разі помилки довелося б повторювати всю операцію з початку.

Натомість обраний підхід – переведення тривалих операцій у фонові обробники через чергу повідомлень - повністю усуває ці проблеми. HTTP-запит лише ставить завдання в чергу та повертає клієнтові підтвердження зі станом «в обробці». Обробник у фоновому режимі споживає завдання, виконує його та оновлює стан у базі даних. Клієнтський

застосунок дізнається про завершення через періодичне оновлення стану за допомогою запитів до API.

Як інструмент організації черг обрано BullMQ – сучасну бібліотеку для Node.js, що використовує Redis як проміжного сховища черг. Redis у даному контексті виступає не як кеш, а як надійне сховище структур даних черг: упорядкованих списків очікуваних завдань, наборів активних та завершених завдань, а також лічильників, які BullMQ використовує для координації обробки.

У системі визначено дві окремі черги: detection-queue (для завдань визначення набору технологій проєктів) та scan-queue (для завдань сканування). Окремість черг дозволяє: ізолювати обробку (навіть якщо одне з сканувань зависне, це не блокує інших операцій), налаштувати окремі параметри паралелізму для кожної черги, розгорнути обробники на окремих машинах з різними характеристиками (наприклад, виділити більш потужні машини для сканування). Архітектура цієї взаємодії показана на рисунку 2.5.

Архітектура з чергами надає системі важливу властивість горизонтального масштабування. У разі зростання навантаження достатньо запустити додаткові екземпляри обробників на нових машинах. BullMQ автоматично розподілить між ними завдання за рахунок Redis-семантики. Принципово важливо, що HTTP-шар і шар оброблення можуть масштабуватися незалежно: якщо система обмежена частотою користувацьких запитів, можна додати екземпляри HTTP-сервера; якщо тривалістю сканувань – більше обробників. Це є класична властивість архітектури «виробник-споживач», яка робить систему гнучкою до зміни шаблонів навантаження.

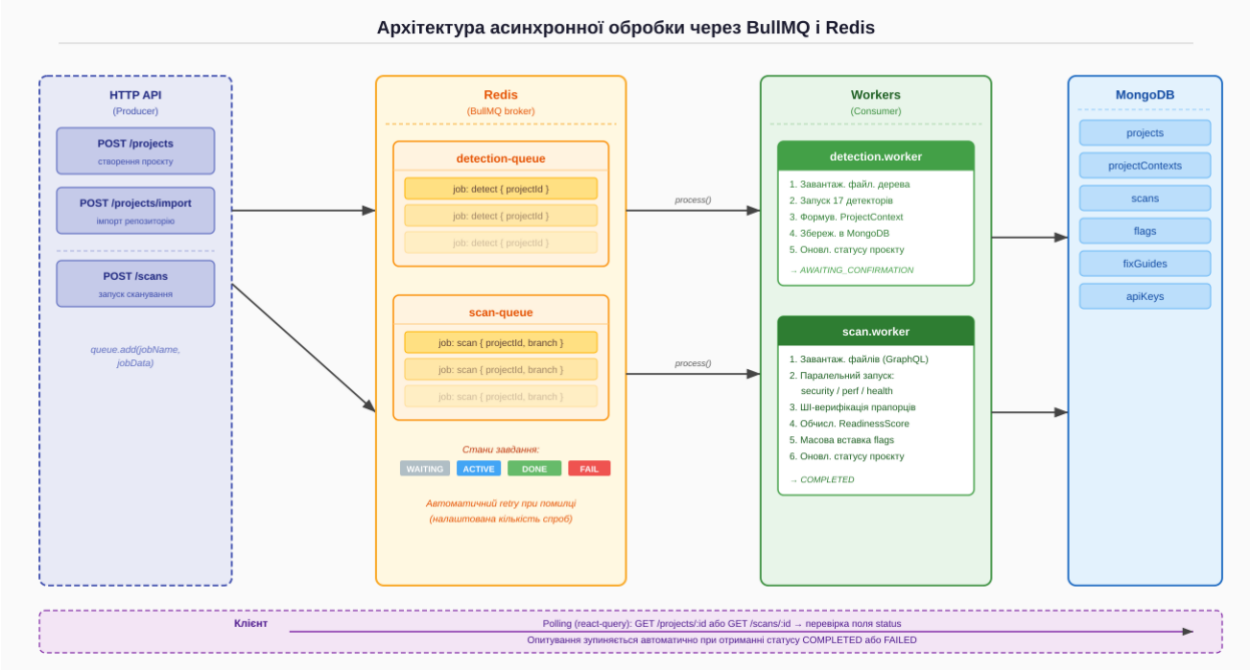

Рисунок 2.5 – Архітектура асинхронної обробки через BullMQ і Redis

2.7 Інтеграція з зовнішніми сервісами

Дві зовнішні служби, з якими інтегрується система, мають принципово різну природу. GitHub виступає одночасно як ідентифікаційний постачальник OAuth, постачальник переліку репозиторіїв користувача та джерело вихідного коду, тобто це багатоцільова інтеграція. Google Gemini, навпаки, є вузькоспеціалізованим постачальником: він використовується лише для верифікування виявлених дефектів та генерації документа з рекомендаціями.

Як ідентифікаційний постачальник обрано GitHub, оскільки це найбільш природний вибір для платформи, орієнтованої на роботу із Git-репозиторіями. Користувач один раз дозволяє доступ через OAuth, після чого система отримує можливість одночасно автентифікувати користувача, отримувати перелік його репозиторіїв та читати вміст файлів. Інтеграція виконується через офіційний SDK @octokit/rest для REST API та власні запити до GraphQL API GitHub.

Інтеграція з GitHub реалізована у двох вимірах: через REST API та через GraphQL API. REST API використовується для операцій, що добре лягають у CRUD-парадигму: отримання інформації про репозиторій, перелік гілок, перевірка існування репозиторію, отримання супровідних даних фіксації змін. Для цих операцій застосовується офіційний SDK @octokit/rest, який абстрагує деталі автентифікації, обробки HTTP-помилок та пагінації.

GraphQL API GitHub використовується для однієї конкретної, але критичної операції – пакетного завантаження вмісту багатьох файлів за один запит. Це принципово важливо, оскільки типовий репозиторій може містити сотні файлів, що підлягають скануванню, і виконання сотень окремих REST-запитів призвело б до значного збільшення часу та потенційного спрацьовування обмежень API GitHub на частоту звернень. Спеціалізована функція агрегує до п’ятдесяти файлів в один GraphQL-запит, що зменшує мережевий трафік. Це є однією з найважливіших архітектурних оптимізацій у системі і безпосередньо впливає на здатність ефективно сканувати великі проєкти.

OAuth-інтеграція з GitHub реалізована через better-auth з конфігурацією GitHub як одного з підтримуваних постачальників. Користувач натискає кнопку входу, перенаправляється на сторінку авторизації GitHub, дозволяє запитувані дозволи. Better-auth завершує обмін кодом авторизації на ключ доступу, створює або оновлює запис користувача та перенаправляє його на головну сторінку. Сам ключ доступу зберігається в колекції облікових записів і не відправляється клієнту в тілі відповідей API.

Як постачальник великої мовної моделі на поточному етапі використовується Google Gemini, конкретно – модель «gemini-2.5-flash». Цей вибір обґрунтовано двома основними факторами: великим контекстним вікном моделі (до одного мільйона токенів), яке дозволяє передавати повноцінний вміст файлів разом із виявленими кандидатами на дефекти, та можливістю константного режиму JSON-виводу. Останнє дозволяє моделі

повертати структуровані результати у вигляді JSON-масиву без додаткових кроків оброблення.

Принципово важливою архітектурною особливістю є те, що Gemini не є жорстко зашитою залежністю в код для сканування вмісту файлів– натомість він прихований за абстракцією «AiProvider», який визначає єдиний контракт. Фактична реалізація для Gemini створюється фабричною функцією «createGeminiProvider», яка приймає розшифрований ключ API та повертає функцію відповідного підпису. Цей підхід дозволяє додати підтримку інших постачальників у майбутньому без жодної зміни в основному коді сканування.

Ключ API користувача зберігається в базі даних у зашифрованому вигляді. Перед кожним викликом до Gemini сервіс піднімає активний ключ для поточного користувача, розшифровує його та використовує для створення постачальника. Це означає, що система ніколи не використовує одного спільного ключа на всіх користувачів.

2.8 Архітектура аналітичного рушія

Аналітичний рушій – це найоб’ємніша та найскладніша частина системи, у якій реалізовано всю логіку статичного аналізу коду, верифікації через ШІ та обчислення оцінки готовності. Він структурно поділений на три великі частини: модуль визначення набору технологій, модуль сканування та модуль оцінювання.

Процес аналізу одного репозиторію умовно складається з двох незалежних фаз: визначення набору технологій (виконується одноразово під час підключення проєкту або при його повторному імпорті) та сканування (виконується щоразу, коли користувач запитує перевірку конкретної гілки). Розділення цих фаз є важливим архітектурним рішенням, оскільки визначення набору технологій це дешева операція, результат якої довго

залишається актуальним, а сканування – дорога операція, яку слід запускати на вимогу. Результат визначення набору технологій зберігається у колекції projectContexts і використовується сканером для адаптивної активації правил: наприклад, правила, специфічні для Mongoose, активуються лише за умови, що в проєкту виявлено саме цю ORM-бібліотеку.

Конвеєр виявлення (detection pipeline) запускається у фоновому обробнику detection.worker і складається з п’яти послідовних кроків. На першому кроці завантажується файлове дерево репозиторію за обраною гілкою через REST-виклик до GitHub. На другому кроці виконується відбір файлів, важливих для виявлення використовуваних технологій. На третьому кроці завантажується вміст відібраних файлів через GraphQL запити, аналогічно до сканування.

На четвертому кроці запускаються сімнадцять окремих детекторів, кожен з яких аналізує отриманий набір файлів і повертає визначені факти про проєкт. Перелік детекторів та їхнє призначення наведено у таблиці 2.4.

На п’ятому кроці отриманий об’єкт ProjectContext, що містить значення detectedMetadata та projectStructure, зберігається в MongoDB як 1:1-сутність до проєкту, а проєкт переходить у стан AWAITING_CONFIRMATION (або в COMPLETED, якщо у нього вже є успішне попереднє сканування - це дозволяє оновити набір технологій без втрати попередніх результатів).

Таблиця 2.4 - Детектори конвеєра виявлення використаних технологій

Детектор	**Призначення**	**Приклади значень**
1	**2**	**3**
detectLanguage	Основна мова програмування	TypeScript, JavaScript
detectFrameworks	Використовувані фреймворки	Express, Next.js, Vue, Nuxt
detectAuthLibraries	Бібліотеки автентифікації	better-auth, passport, clerk

Продовження таблиці 2.4

1	2	3
detectDatabases	Використовувані бази даних	MongoDB, PostgreSQL, MySQL
detectORMLibraries	ORM / ODM бібліотеки	Mongoose, Prisma, Drizzle
detectValidationLibraries	Бібліотеки перевірки вхідних даних	Zod, Joi, Yup
detectLogging	Бібліотеки логування	Winston, Pino, Morgan
detectTesting	Фреймворки тестування	Jest, Vitest, Mocha
detectRateLimitLibraries	Бібліотеки обмеження частоти запитів	express-rate-limit, upstash
detectProjectStructure	Тип файлової структури проєкту	FRONTEND_ONLY, BACKEND_ONLY, FULLSTACK, MONOREPO, SPLIT_FOLDER

Конвеєр сканування, реалізований у функції runScan, є оркестратором усього процесу аналізу. Він починається з паралельного завантаження документів Project і ProjectContext. Далі здійснюється отримання останнього версії коду цільової гілки через REST API GitHub – це важливо для відтворюваності результатів: усі прапорці прив’язуються саме до цієї версії, і користувач у будь-який момент може повернутися до зафіксованого стану сканування.

Далі створюється документ Scan із полем «status» зі значенням «SCANNING» та порядковим номером. Одночасно змінюється поле «status» батьківського проєкту на «SCANNING». Після цього з колекції apiKeys піднімається активний ключ Gemini для поточного користувача (якщо такого немає, сканування одразу завершується помилкою). Розшифрований ключ передається у фабричну функцію «createGeminiProvider», яка повертає «AiProvider» для подальшого використання.

Наступний крок – однократне завантаження всього набору сканованих файлів через спеціалізовану функцію «fetchScanFiles». Вона виконує один REST-виклик для отримання повного дерева файлів, відфільтровує його за переліком сканованих розширень (.ts, .js, .tsx, .jsx, .yaml, .yml, .env та package.json), виключає шляхи, які заздалегідь визнано неактуальними (node_modules, dist, build, coverage, .next, .nuxt, тестові файли, .d.ts, публічні статичні ресурси, документація, e2e-тести). Залишені файли впорядковуються так, щоб ті, які з більшою ймовірністю містять критичні категорії проблем (файли з «auth», «middleware», «route», «config» у назві, а також файли, відомі як точки входу або маршрути за результатами виявлення), завантажувалися першими. Усе подальше завантаження вмісту виконується пакетно через GraphQL.

Після завантаження файлів стартують три незалежні конвеєри аналізу (runSecurityScan, runPerformanceScan, runCodeHealthScan) у паралельному режимі. Усі три отримують один і той самий набір файлів, значення «detectedMetadata», «projectStructure» та постачальника ШІ. Результати трьох конвеєрів об’єднуються, кожен прапорець відзначається відповідним типом (SECURITY, PERFORMANCE, CODE_HEALTH). Далі обчислюється оцінка готовності, виконується масова вставка прапорців і оновлюється документ Scan, а проєкт переходить у стан COMPLETED.

Для кожного з трьох напрямків аналізу визначено власний набір категорій правил. Узагальнений перелік категорій правил наведено у таблиці 2.5.

Таблиця 2.5 – Категорії правил статичного аналізу за напрямками

Напрямок аналізу	**Категорії правил**
Безпека (SECURITY)	Виявлення секретів у коді, вразливості до ін’єкцій, проблеми автентифікації, витік даних, небезпечна конфігурація, криптографічні помилки
Продуктивність (PERFORMANCE)	Антипатерни API, відсутнє чи неефективне кешування, запити N+1, тощо, фронтенд-патерни, специфіка Node.js - синхронний IO у обробниках, безмежні setInterval тощо
Якість коду (CODE_HEALTH)	загальна якість, порожні catch, проковтнуті помилки, відсутність структурованого логування та health-check, зловживання «any», ts-ignore, тощо

Окремо слід зазначити, що деякі правила реалізують не просто рядковий пошук, а перевірки на рівні файла та крос-файлові перевірки. Прикладом є правило MISSING_HEALTH_CHECK: воно проходить по всіх файлах точок входу та маршрутах сервера в пошуках обробника /health (або healthz, liveness, ping, status). Якщо такого обробника немає – прапор виставляється; якщо є, але всередині не виявлено ключових слів-індикаторів перевірки статусу підключення баз даних (readyState, isOpen, ping(), тощо), виставляється окремий прапор HEALTH_CHECK_NO_DB_STATUS. Такі складніші правила суттєво підвищують цінність аналізу.

Архітектурно конвеєри «runSecurityScan», «runPerformanceScan», «runCodeHealthScan» мають однакову сигнатуру (приймають набір файлів, «detectedMetadata», «projectStructure», «aiProvider» та повертають набір прапорців), однакову внутрішню послідовність кроків, однаковий контракт результату (масив структурованих прапорців). Відмінності полягають лише в наборі правил, що активуються, та в специфічних оптимізаціях (наприклад, для «security» виконується додаткове сканування «dependencyScan», для «code-health» – ні). Спільні компоненти конвеєрів винесено в підкаталог, який імпортується трьома конвеєрами. Цей підхід забезпечує, що додавання нового напрямку аналізу не вимагатиме переписування ні оркестратора «runScan», ні шару верифікації.

2.9 Верифікація через штучний інтелект як головна концепція

Серед усіх архітектурних рішень системи верифікація через велику мовну модель посідає особливе місце. Це не допоміжна можливість, а свідомо обраний центральний елемент архітектури, навколо якого побудована вся конвеєрна-частина системи.

Класичні інструменти статичного аналізу страждають від фундаментальної проблеми: вони базуються на регулярних виразах і синтаксичних патернах, тому неминуче виробляють велику кількість хибно-позитивних результатів. У результаті, доступний користувачеві набір прапорців містить значну частину «шумових» спрацьовувань, що поступово знижує довіру до інструмента – користувач починає ігнорувати попередження, серед яких часом губляться реально критичні дефекти.

Архітектура системи вирішує цю проблему принципово: статичний аналіз перестає бути джерелом істини щодо знайдених дефектів і стає лише генератором кандидатів. Остаточне рішення про те, чи є кандидат реальним дефектом, приймає велика мовна модель, маючи у своєму розпорядженні

повноцінний вміст файлу та контекстну інформацію про проєкт. Модель таким чином отримує здатність розрізняти справжні дефекти від хибних позитивних результатів, враховуючи семантичний контекст коду.

Наслідком цього вибору є те, що активний ключ API постачальника ШІ є обов'язковою передумовою для сканування. Якщо у користувача немає такого ключа, сканування одразу завершується помилкою, а не виконується «частково» зі лише статичними результатами. Це свідоме архітектурне рішення: побічна можливість «обійти» ШІ-верифікації створювала б два шляхи виконання сканування з різною якістю результату, що ускладнювало б інтерпретацію оцінки готовності і підривало б основну ідею системи.

Шар верифікації отримує на вхід набір «сирих» прапорців (кандидатів, виявлених статичним скануванням), набір завантажених файлів і постачальника ШІ. Спочатку прапорці групуються за файлами, до яких вони належать. Це критично, оскільки верифікація моделі вимагає показати повноцінний вміст файлу, а одне відсилання контексту разом із кількома прапорцями того ж файлу значно ефективніше за окремі запити.

Далі прапорці організуються в партії з урахуванням бюджету токенів, будуються системні та користувацькі повідомлення для моделі, і відправляються паралельні асинхронні запити до ШІ для кожної партії. Отримані результати об'єднуються та нормалізуються, після чого верифіковані прапорці повертаються у сервіс сканування.

Поведінка під час об'єднання результатів регламентується чіткими правилами. Якщо модель підтвердила прапорець, він зберігається, а його поля «severity», «title» і «description» перезаписуються збагаченими значеннями моделі. Якщо модель не підтвердила прапорець, його повністю відкидають як хибно-позитивний. Якщо модель опустила прапорець у відповіді, це не вважається відмовою. Прапорець зберігається з вихідним статичним описом, що захищає систему від мовчазних втрат у разі помилок форматування відповіді.

Великі мовні моделі мають жорстке обмеження на розмір контекстного вікна (для gemini-2.5-flash це близько одного мільйона токенів). Тому критичним кроком є побудова партій файлів так, щоб кожен виклик умістився у заданий бюджет токенів. Для цього у системі визначено константу «TOKEN_BUDGET = 200_000» (свідомо встановлену значно нижче за теоретичну межу моделі, щоб залишити запас для відповіді та уникнути специфічних випадків).

Розподіл прапорців за файлами враховує бюджет токенів. Для кожного файла оцінюється його розмір у токенах за допомогою наближення «один токен це приблизно чотири символи». Файли додаються до поточної групи послідовно, поки загальна оцінка не перевищує встановлений ліміт. При перевищенні ліміту починається нова група. Якщо окремий файл сам по собі більший за ліміт, він утворює власну групу – це нормально для великих файлів, оскільки модель все ще може їх обробити.

Архітектурно цей підхід забезпечує те, що типове сканування невеликого або середнього репозиторію вкладається в один-три виклики до моделі, а не у сотні. Це принципово впливає одночасно на час сканування та на вартість використання сервісу.

Однак не всі правила потребують верифікування моделлю. Деякі регулярні вирази настільки точні, що їхні спрацьовування практично не дають хибнопозитивних результатів. Прикладами є виявлення відомих форматів секретів, виявлення однозначно небезпечних викликів, окремі високоточні правила продуктивності та якості коду. Для таких правил визначено набір ідентифікаторів, які повністю обходять шар верифікації, прапорці з цими ідентифікаторами одразу зараховуються як підтверджені. Це є важливою архітектурною оптимізацією, що дозволяє знизити кількість викликів до моделі для типових випадків, де її судження свідомо не дасть більше інформації, ніж сама регулярна перевірка.

Якість роботи механізму верифікування безпосередньо залежить від якості запитів, що подаються моделі. У системі застосовано декомпозицію

запиту на два чітко розмежовані компоненти: системний запит (system prompt) і користувацьке повідомлення (user message). Системний запит формується одноразово на початку сканування з метаданих проєкту. У запиті чітко зафіксовано формат очікуваної відповіді (JSON-масив об’єктів з полями flagIndex, confirmed, severity, title, description) та правила, яких модель має дотримуватися (зокрема, не вигадувати прапорці, що не були передані, не змінювати ідентифікатор правила, не повертати додаткового тексту поза JSON).

Користувацьке повідомлення для кожної групи файлів організується структурованим форматом: спочатку вказується шлях файла, потім його повноцінний вміст, а далі нумерований список прапорців із основною інформацією (правило, рядок, фрагмент коду, опис) та додатковим контекстом, який статичний аналізатор міг прикріпити (оточуючі рядки, вміст функції тощо). Такий структурований формат дозволяє моделі зрозуміти, який фрагмент коду потребує верифікування та чому він позначений як підозрілий.

2.10 Алгоритм оцінювання готовності до промислової експлуатації

Окремою функціональною можливістю системи є обчислення оцінки готовності кожного завершеного сканування – цілого числа в діапазоні від 0 до 100, яке користувач бачить як єдиний показник «здоров’я» поточної гілки. Хоча конкретне числове значення оцінки залежить від результатів сканування, сама формула оцінювання, її параметри та властивості є архітектурним рішенням і розглядаються тут.

Формула побудована на основі експоненційного загасання сумарного «штрафу», що обчислюється з прапорців сканування за такою послідовністю:

- базовий внесок прапорця залежить від його серйозності: 40 одиниць за CRITICAL, 15 за HIGH, 5 за MEDIUM, 1 за LOW.

- кожен прапорець множиться на коефіцієнт типу: SECURITY множиться на 1,0, PERFORMANCE на 0,7, CODE_HEALTH на 0,4.
- у межах кожної серйозності (окрім CRITICAL) внесок обмежується «стелею»: для HIGH – 150 одиниць сумарно, для MEDIUM – 60, для LOW – 20.
- окремо встановлюється «стеля серйозності за типом»: будь-який прапорець SECURITY може мати серйозність до CRITICAL, тоді як PERFORMANCE і CODE_HEALTH ефективно обмежуються рівнем HIGH (навіть якщо правило позначило їх як CRITICAL, фактично вони вважаються HIGH).
- після обчислення сумарного штрафу застосовується експоненційне загасання з коефіцієнтом $k = 0{,}025$.

Верхня межа серйозності за типом не дає збити загальну оцінку великою кількістю малозначущих прапорців: проєкт із двома сотнями проблем рівня LOW не зазнає такого самого штрафу, як проєкт із кількома CRITICAL. Експоненційне загасання забезпечує плавну та інтуїтивну поведінку шкали: достатньо якісні проєкти отримують 60–80 балів, бездоганно чисті – 100, а серйозно несправні наближаються до нуля.

Отримана оцінка візуалізується в інтерфейсі за допомогою кольорового кодування: значення ≥ 70 відображаються зеленим («Оптимально»), значення в діапазоні 40-69 – жовтим («Потребує уваги»), значення < 40 – червоним («Критично»). Логіка кольорового кодування реалізована в окремому файлі /client/lib/score.ts, що дозволяє за потреби налаштовувати порогові значення без зміни обчислювальної формули.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНЮВАННЯ ЗАСТОСУНКУ

3.1 Розгортання застосунку

Застосунок розгортається у двох режимах: локальному середовищі розроблення та виробничому хмарному середовищі. Обидва режими використовують однакову кодову базу, відрізняючись лише конфігурацією зовнішніх сервісів та змінними середовища.

Для запуску застосунку локально необхідно мати встановлені Node.js версії 20 або вище та Docker. Бази даних MongoDB та Redis запускаються як контейнери Docker без необхідності їх локального встановлення. Достатньо виконати команди запуску контейнерів, після чого обидві служби стають доступними на стандартних мережевих портах (MongoDB на порту 27017 та Redis на порту 6379).

Після запуску баз даних встановлюються залежності клієнтської та серверної частин командою «npm install». Конфігурація здійснюється через файли змінних середовища у кожному з каталогів. Серверна частина запускається на мережевому порту 3000, клієнтська на 8080. Для реєстрації вебзастосунку в системі відкритої авторизації GitHub необхідно створити OAuth-застосунок та вказати адресу зворотного виклику «http://localhost:8080/api/auth/callback/github».

У виробничому режимі застосунок розгорнуто на платформі Railway, яка забезпечує автоматичне розгортання при кожному оновленні основної гілки репозиторію. Клієнтська та серверна частини розгорнуті як окремі служби Railway з незалежними адресами. Замість локальних контейнерів Docker використовуються хмарні керовані сервіси баз даних: MongoDB Atlas для бази даних MongoDB та Upstash для Redis. Обидва сервіси надають посилання для підключення, які підставляються у змінні середовища Railway без будь-яких змін у коді застосунку.

MongoDB розгорнуто в безкоштовній конфігурації M0, що забезпечує необхідну продуктивність для поточного навантаження. Upstash надає Redis-сумісний сервіс із моделлю оплати за кількість команд, що є економічно виправданим для черг завдань із нерівномірним навантаженням. Для виробничого розгортання OAuth-застосунок GitHub реєструється окремо з адресою зворотного виклику виробничого домену.

У таблиці 3.1 наведено порівняння конфігурацій двох середовищ розгортання.

Таблиця 3.1 – Конфігурація середовищ розгортання

Компонент	**Локальне середовище**	**Виробниче середовище**
MongoDB	Контейнер Docker (мережевий порт 27017)	MongoDB Atlas (хмарний сервіс)
Redis	Контейнер Docker (мережевий порт 6379)	Upstash Redis (хмарний сервіс)
Клієнтська частина	Локально (мережевий порт 8080)	Railway (окрема служба)
Серверна частина	Локально (мережевий порт 3000)	Railway (окрема служба)
Розгортання	Ручний запуск	Автоматично

3.2 Демонстрація роботи застосунку

Повний цикл взаємодії із застосунком складається з семи послідовних кроків: вхід до системи, реєстрація ключа доступу до постачальника великих

мовних моделей, перегляд панелі керування, підключення репозиторію, запуск сканування, перегляд результатів та отримання настанови з виправлення. Усі кроки виконуються у вебінтерфейсі без встановлення додаткових інструментів.

Головна сторінка застосунку (рис. 3.1) дає загальне уявлення про поточний стан усіх проєктів. У верхній частині розташовано чотири картки статистики. Перша – загальна кількість підключених проєктів. Друга – середня оцінка готовності до промислової експлуатації по всіх проєктах із завершеними скануваннями. Третя – загальна кількість критичних знахідок по всіх проєктах. Четверта – кількість проєктів «під загрозою», тобто тих, що мають оцінку нижче 40 або містять хоча б одну критичну позначку.

Нижче карток розташована таблиця проєктів. Вона відсортована так, що завершені проєкти з найнижчими оцінками виводяться першими, це одразу привертає увагу до найбільш проблемних репозиторіїв. Для кожного проєкту відображаються назва, знайдені вразливості, оцінка готовності та час останнього сканування. Натискання на рядок переходить до сторінки відповідного проєкту.

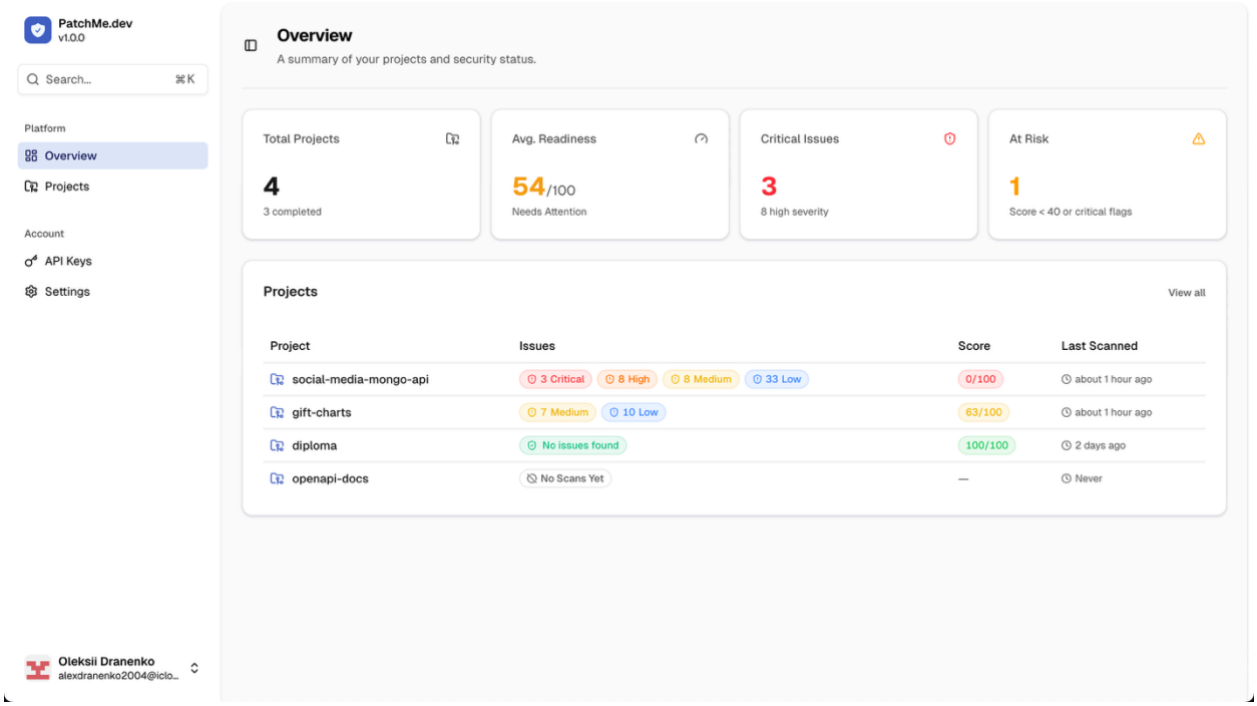

Рисунок 3.1 – Панель керування

Першою необхідною дією є реєстрація ключа доступу до постачальника великих мовних моделей. Сторінка «/api-keys» (рис. 3.2) містить форму для введення ключа Google Gemini. Після збереження ключ відображається у скороченому вигляді, повне значення у жодному елементі інтерфейсу не з'являється. За потреби ключ можна видалити та замінити іншим.

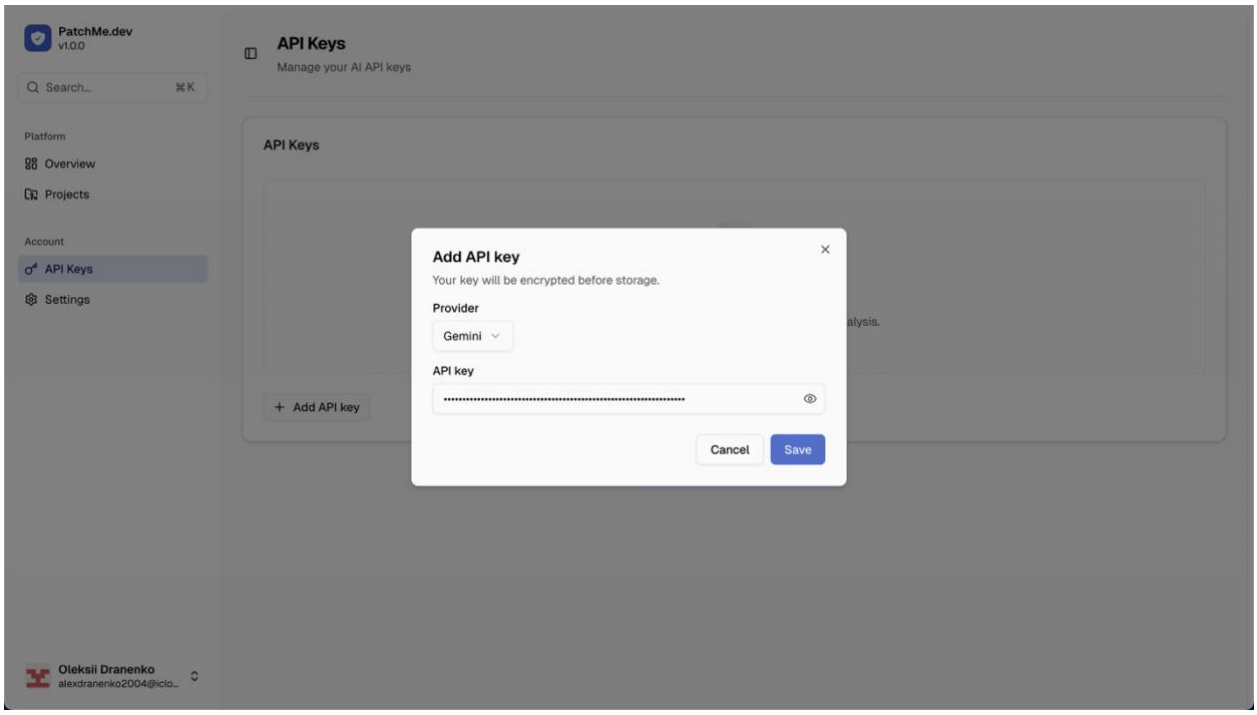

Рисунок 3.2 – Реєстрація ключа доступу до постачальника великих мовних моделей

На сторінці /projects (рис. 3.3) розташована кнопка «Add project», що відкриває модальне вікно підключення нового репозиторію. Вікно пропонує два варіанти залежно від того, де знаходиться репозиторій.

Перший варіант (рис. 3.4) це підключення власного репозиторію. Після вибору цієї вкладки відображається список усіх репозиторіїв облікового запису GitHub, під яким виконано вхід, включно з приватними. Репозиторії можна відфільтрувати за назвою за допомогою пошукового поля. Для підключення достатньо натиснути на потрібний репозиторій у списку.

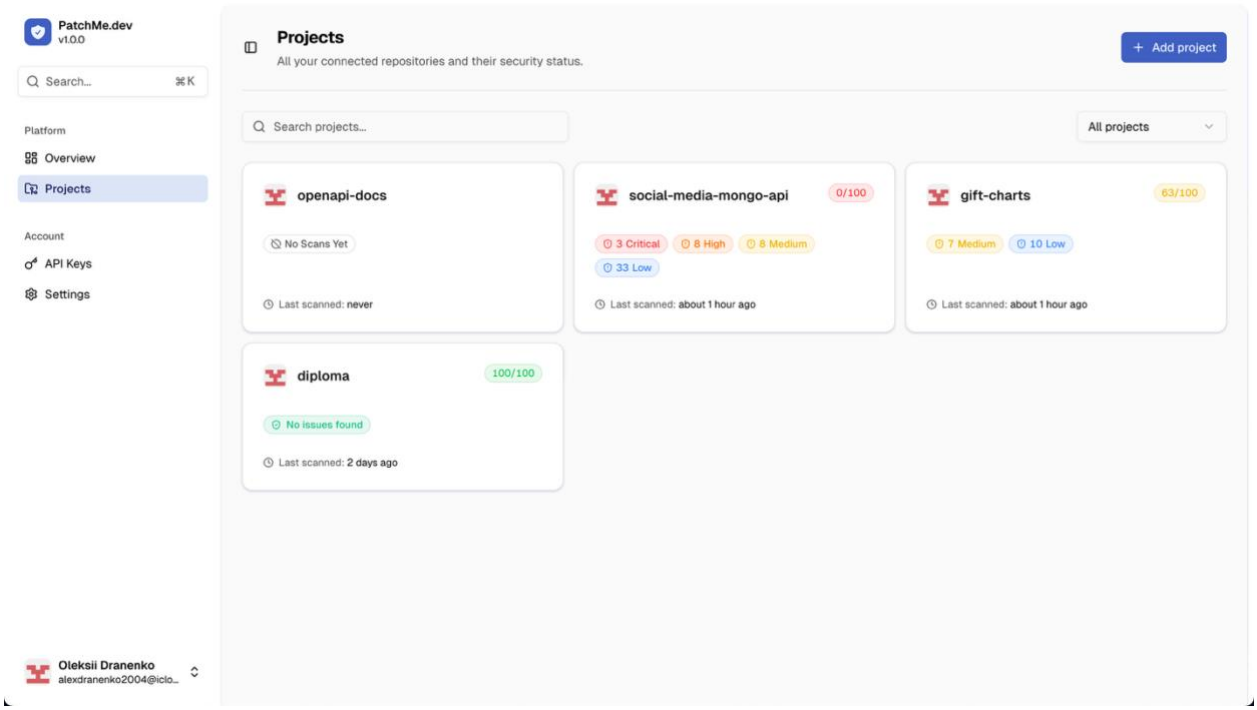

Рисунок 3.3 – Сторінка /projects

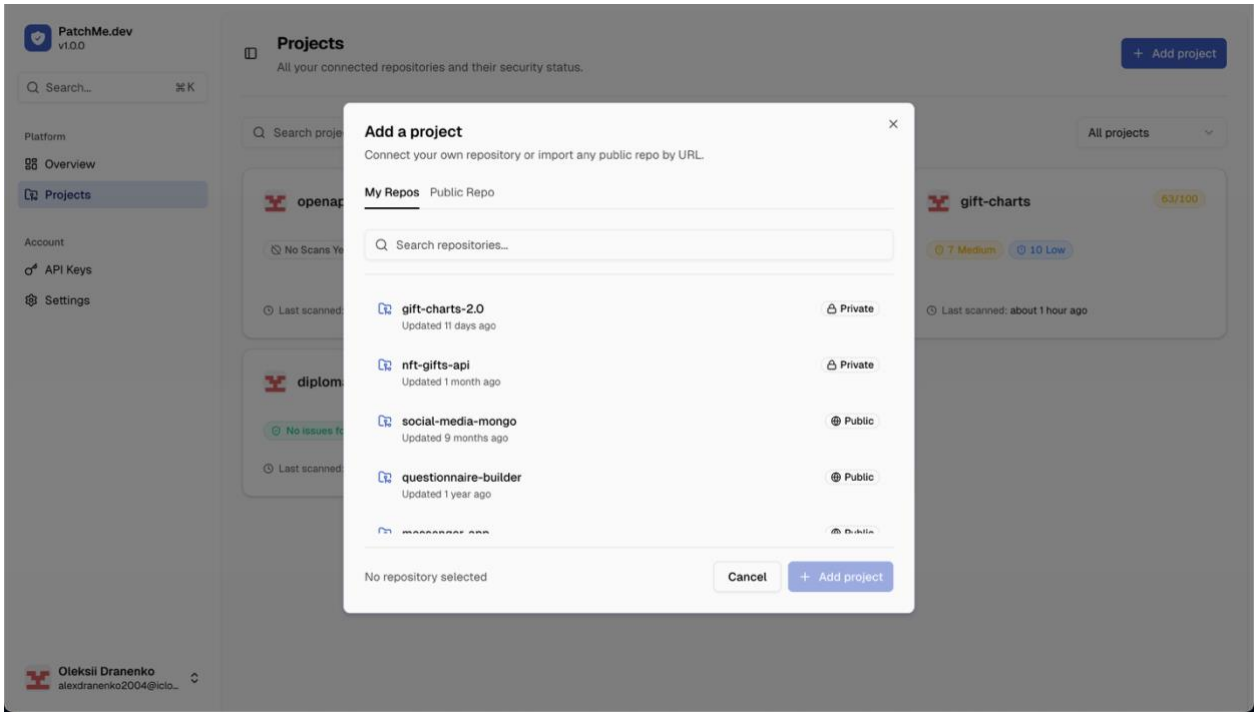

Рисунок 3.4 – Підключення власного репозиторію зі списку

Другий варіант (рис. 3.5) це імпорт публічного репозиторію за посиланням. Ця вкладка призначена для аналізу будь-якого публічного репозиторію GitHub незалежно від того, чи є він у списку власних.

Користувач вставляє повне посилання на репозиторій у текстове поле та підтверджує імпорт.

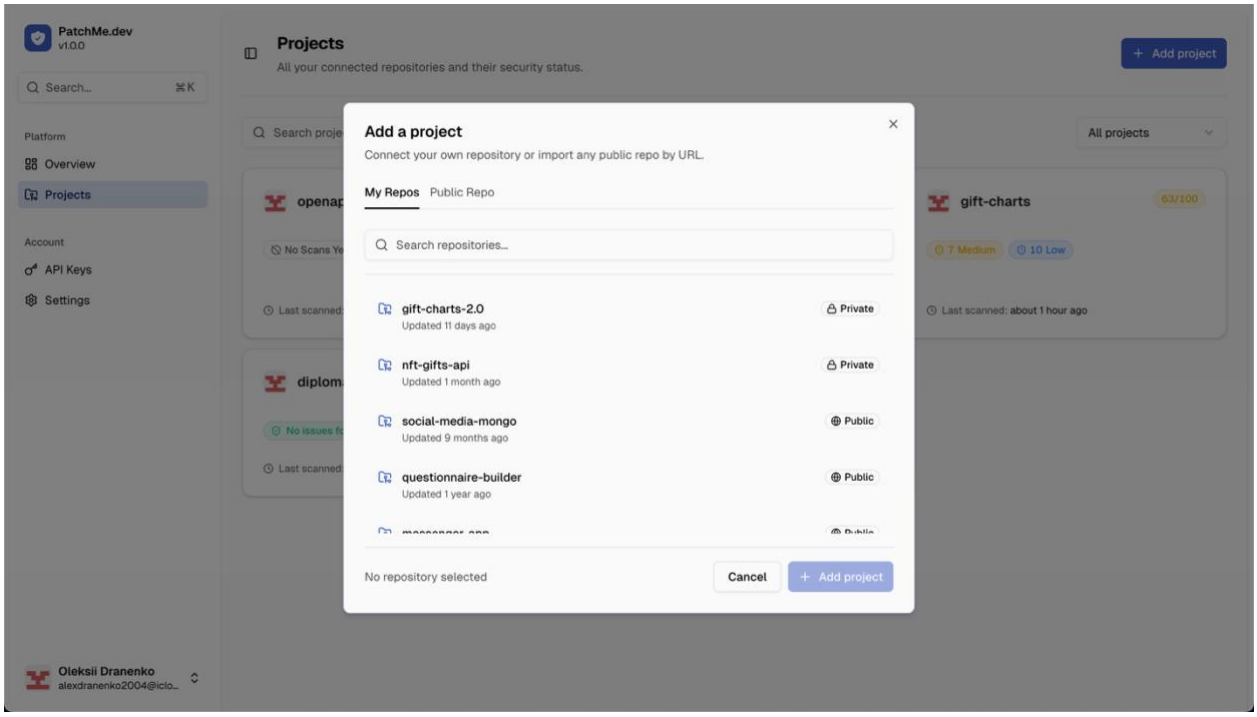

Рисунок 3.5 – Підключення публічного репозиторію за посиланням

Одразу після підключення система автоматично розпочинає визначення технологічного набору та файлової структури репозиторію без будь-яких додаткових дій з боку користувача. Результати відображаються на сторінці /projects/[id]/details, що складається з двох окремих блоків.

Перший блок (рис. 3.6) містить виявлені технології проєкту. Для кожної виявленої технології відображається її назва. Якщо певна категорія у проєкті відсутня, це також фіксується і враховується під час формування оцінки готовності.

Другий блок (рис. 3.7) відображає файлову структуру проєкту. Ці дані дозволяють системі під час сканування зосередити аналіз на найбільш критичних з погляду безпеки та якості файлах.

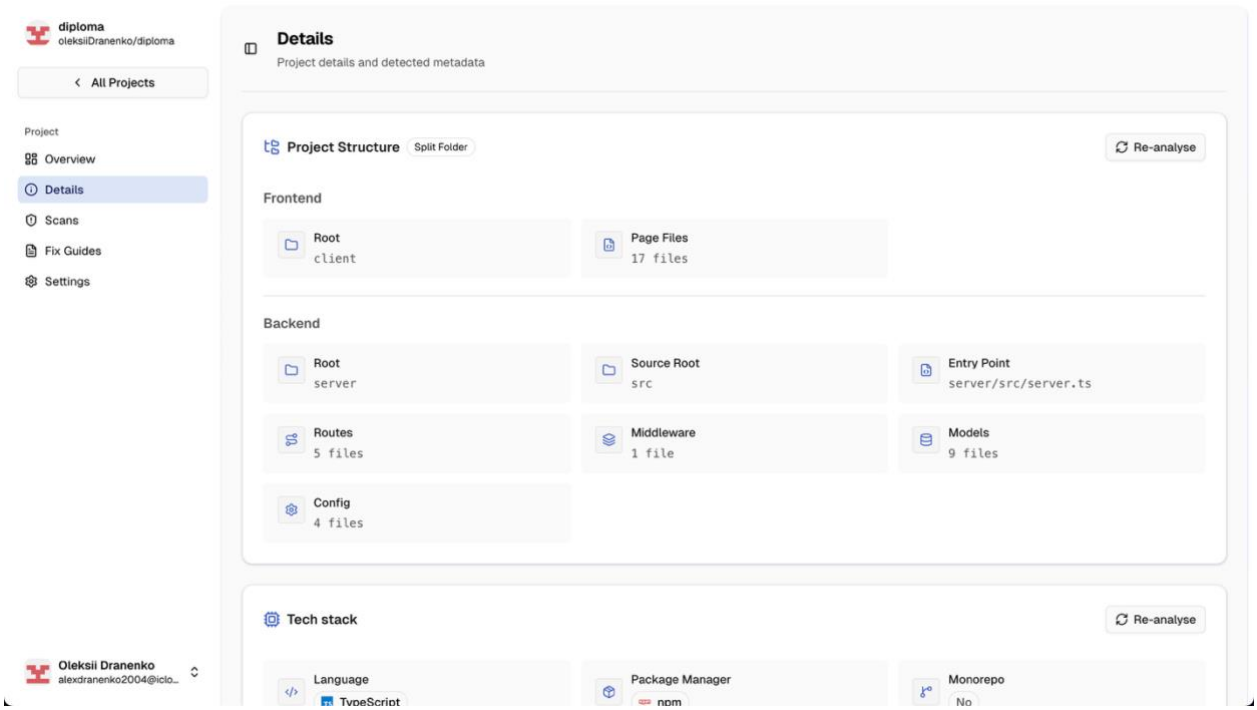

Рисунок 3.6 – Приклад файлової структури проєкту

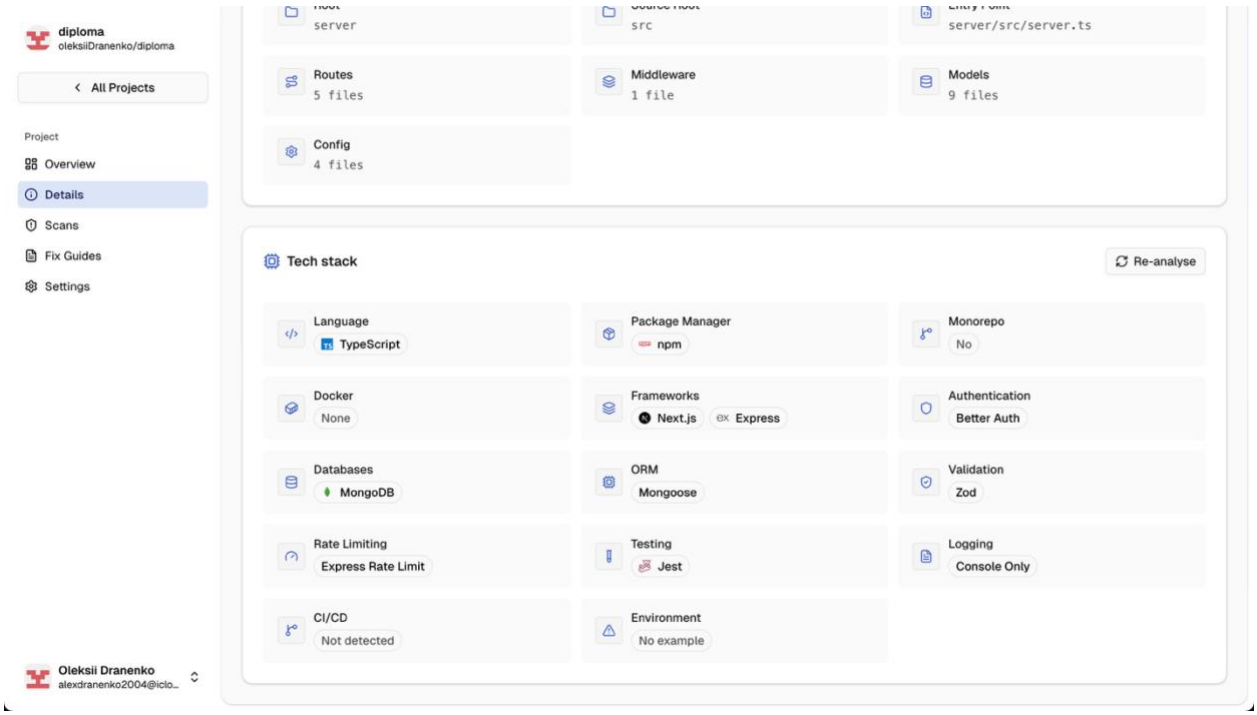

Рисунок 3.7 – Приклад технологічного набору проєкту

Сторінка /projects/[id]/scans (рис. 3.8) містить повну історію сканувань проєкту. Останнє сканування виділено окремо, решта згруповані у розгортуваний список із лічильником. Для запуску нового сканування є

кнопка «New scan», яка відкриває модальне вікно вибору гілки (рис. 3.9). Перелік гілок завантажується безпосередньо з GitHub, тому відображає актуальний стан репозиторію на момент запуску.

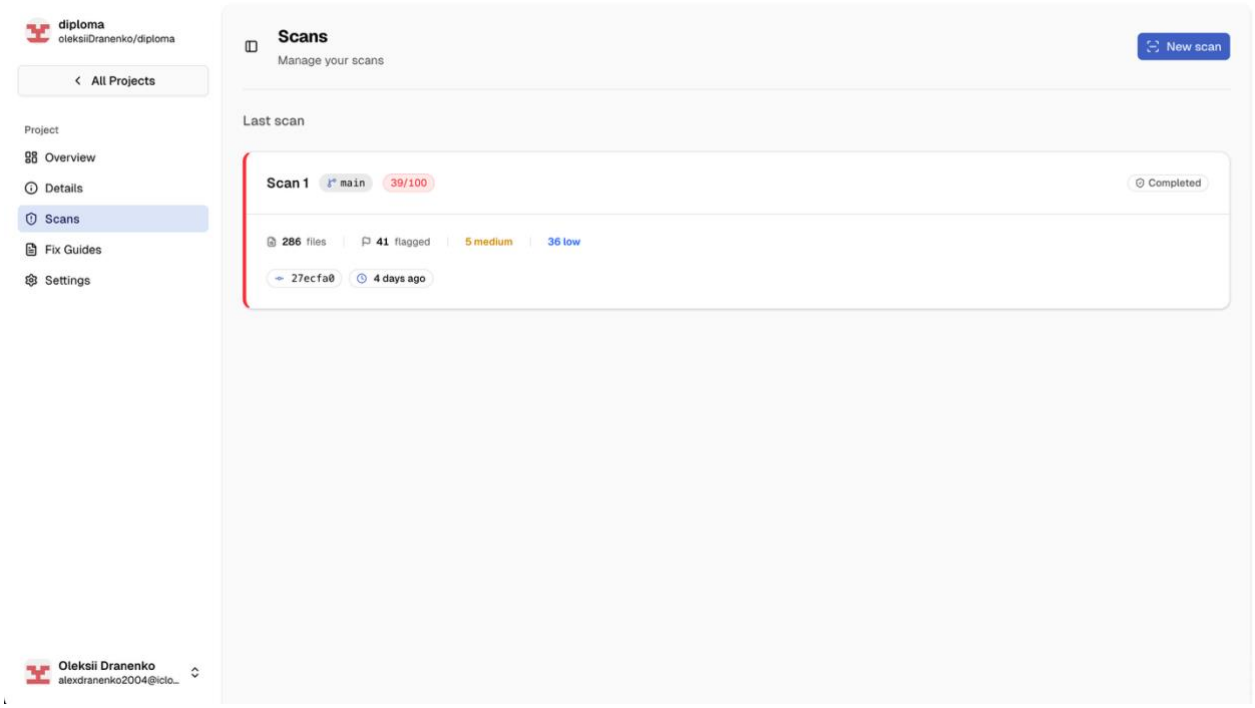

Рисунок 3.8 – Приклад історії сканувань проєкту

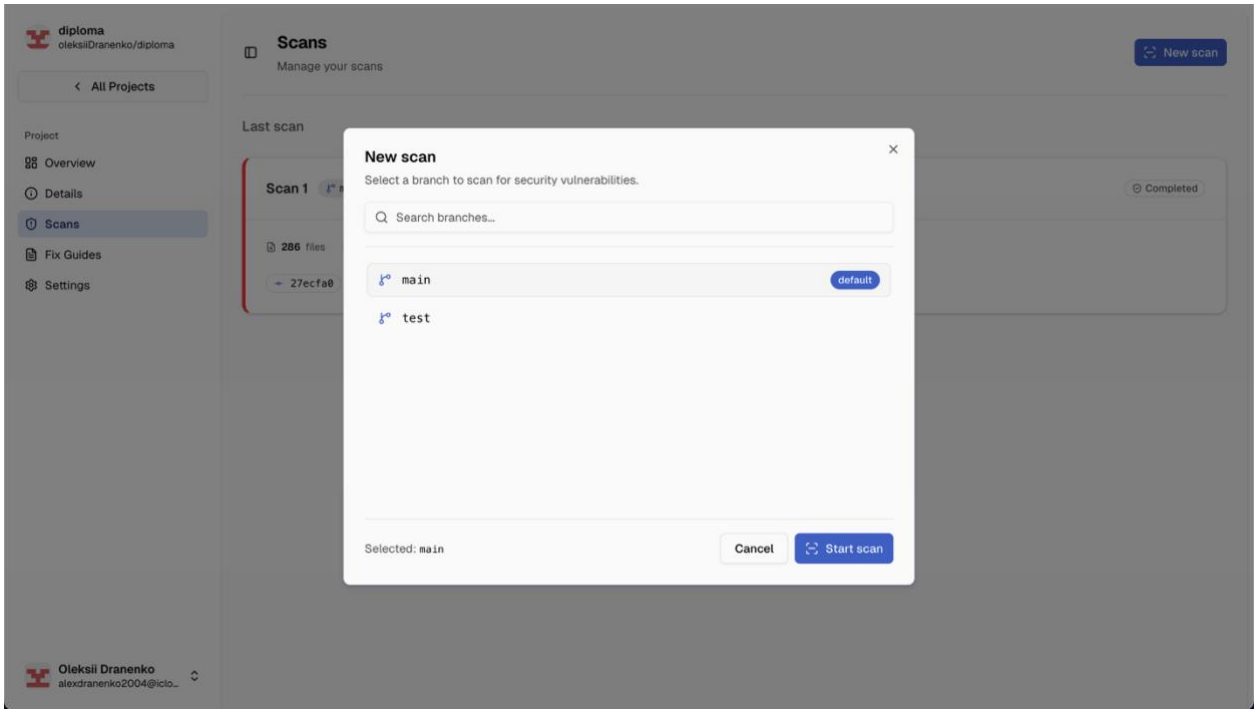

Рисунок 3.9 – Вибір гілки проєкту для запуску сканування

Детальна сторінка сканування /projects/[id]/scans/[scanId] (рис. 3.10) відображає повний перелік знайдених вразливостей. В заголовку вказано номер сканування, назву гілки та оцінку готовності. Список знайдених вразливостей підтримує одночасну фільтрацію за двома параметрами: напрямком аналізу та рівнем серйозності вразливості. Кожен елемент списку показує шлях до файлу, номер рядка, рівень серйозності з відповідним кольором та заголовок з коротким описом виявленої проблеми.

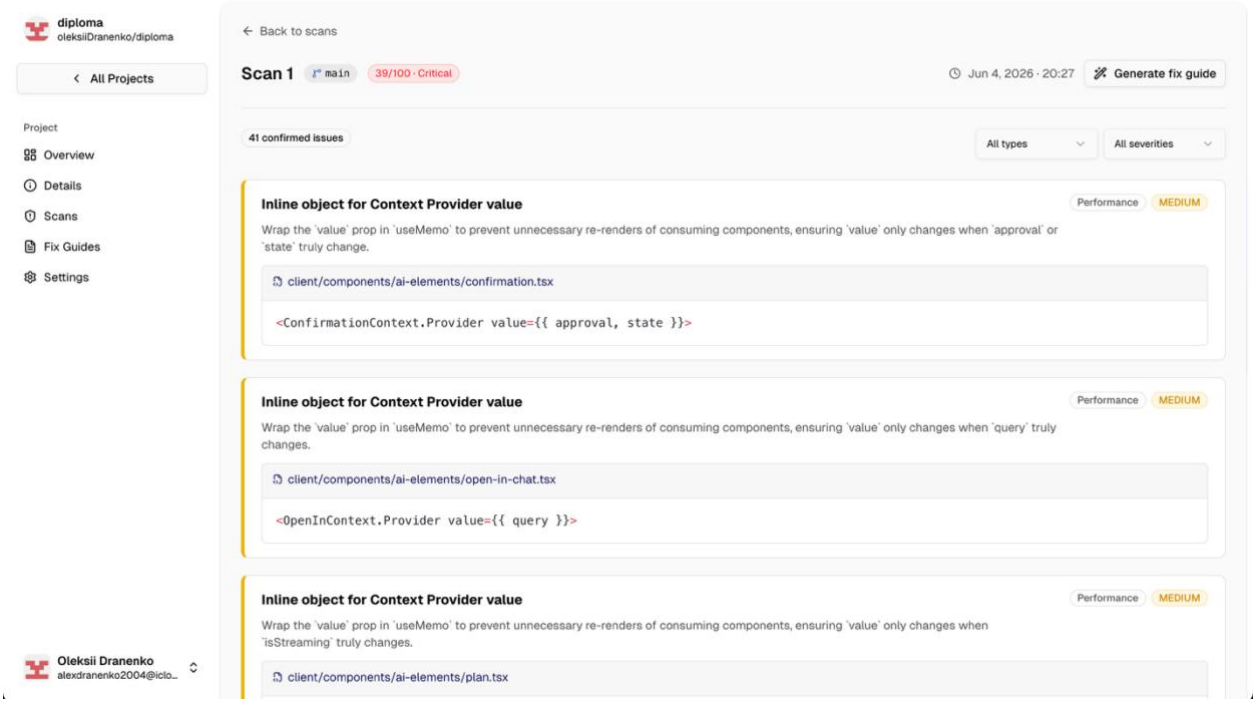

Рисунок 3.10 – Приклад списку вразливостей проєкту

Натискання на будь-яку вразливість відкриває сторінку /projects/[id]/scans/[scanId]/file (рис. 3.11) – перегляд відповідного вихідного файлу. Сторінка відображає повний вихідний код файлу із синтаксичним підсвіченням: проблемні рядки виділені кольором, при наведенні на які з’являється блок з назвою та рівнем серйозності конкретної вразливості. У заголовку сторінки наведено перелік проблем за рівнями серйозності, що дає змогу одразу оцінити загальний стан файлу.

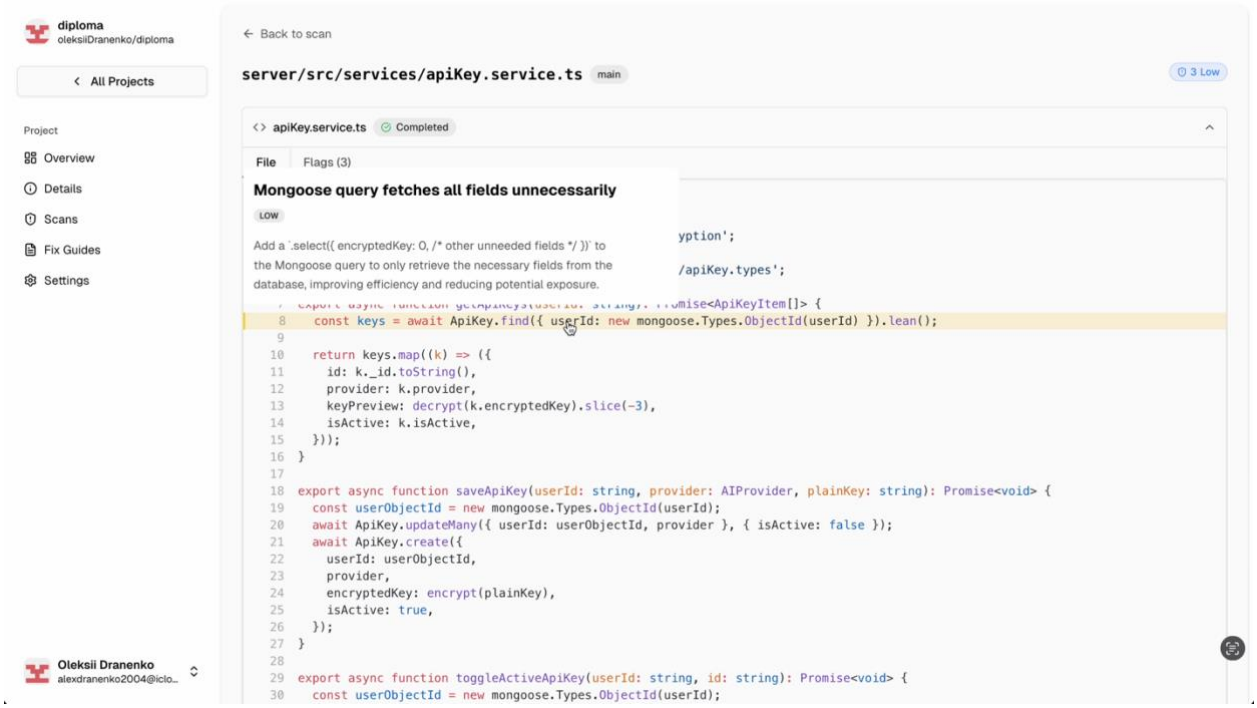

Рисунок 3.11 – Приклад вихідного коду файлу з вразливістю

На сторінці результатів сканування розташована кнопка «Generate fix guide», що відкриває модальне вікно (рис. 3.12). У ньому відображається повний список позначок поточного сканування, де користувач позначає ті, для яких необхідно отримати інструкції з виправлення. Передбачена можливість обрати всі позначки одночасно або лише окрему підмножину (наприклад, лише критичні вади безпеки). Після підтвердження вибору починається генерація документа у форматі Markdown.

Після завершення генерації користувач автоматично перенаправляється на сторінку /projects/[id]/fix-guides. Кожен запис у списку відображає дату створення та кількість вразливостей, що були включені до документа. Файл можна відкрити для перегляду безпосередньо в інтерфейсі (рис 3.13) або завантажити на свій пристрій.

Завантажений файл призначений для використання у подальшому робочому процесі виправлення: його можна передати як контекст будь-якому ШІ-асистенту (Cursor, Claude або ChatGPT). Оскільки документ містить конкретні шляхи до файлів, номери рядків, опис кожної вади та покрокові інструкції з виправлення, ШІ-асистент отримує достатньо інформації для

того, щоб одразу запропонувати готові зміни до коду без необхідності самостійно шукати проблемні місця у репозиторії.

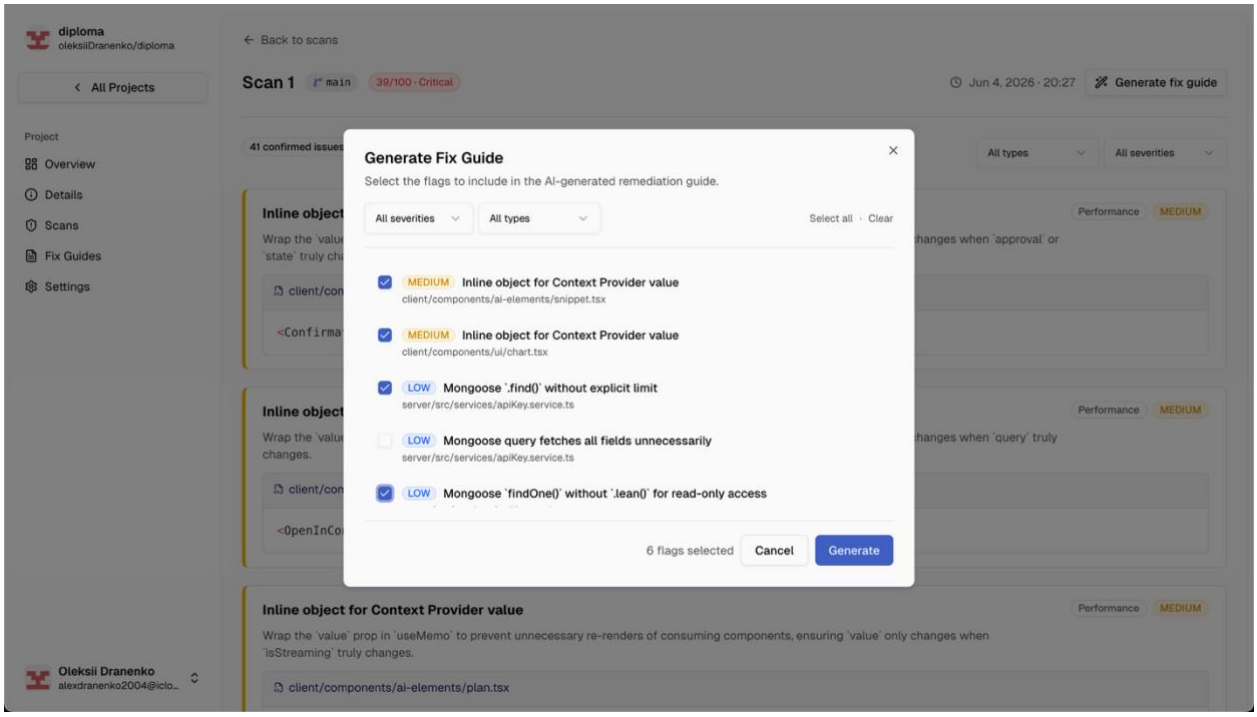

Рисунок 3.12 – Приклад вибору вразливостей для генерації інструкції по усуненню вразливостей

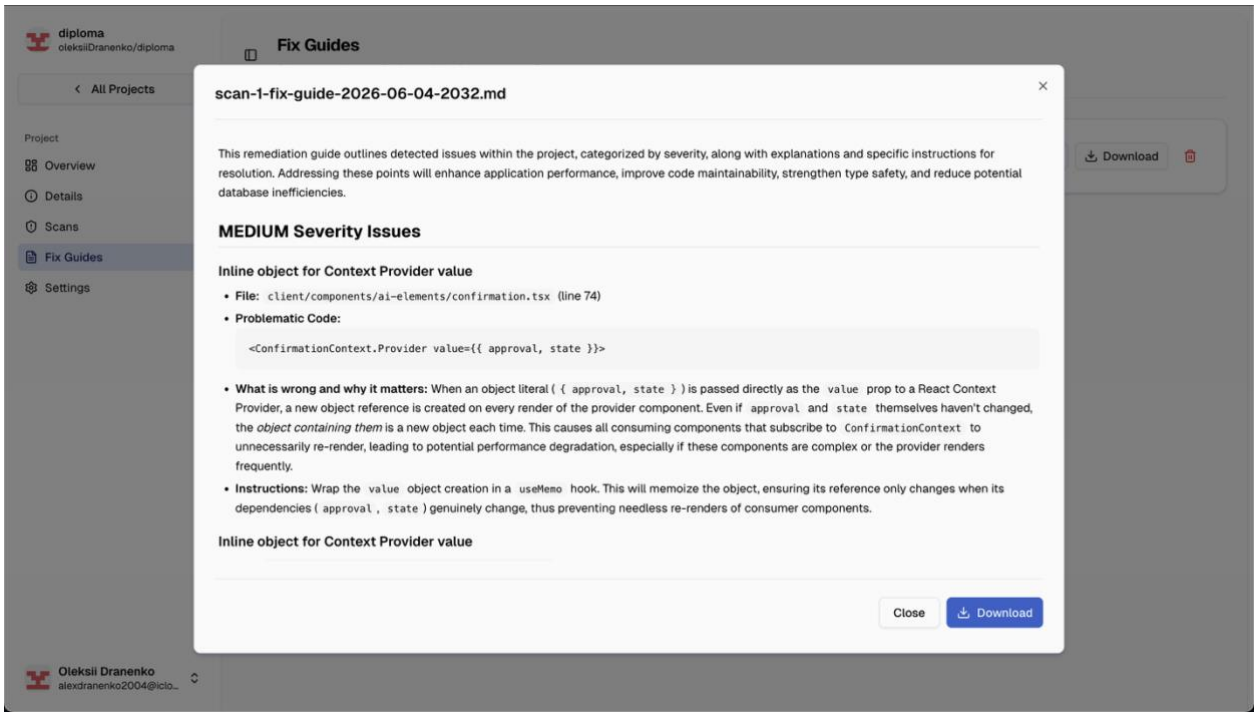

Рисунок 3.13 – Приклад згенерованої інструкції по усуненню вразливостей

Для наочної демонстрації повного циклу роботи застосунку виконано практичне застосування згенерованої інструкції з виправлення на одному з тестових репозиторіїв. За результатами першого сканування проєкт отримав оцінку готовності 39 балів зі 100 – виявлено 41 підтверджену проблему різних рівнів серйозності. Згенеровану інструкцію було передано до Claude Code у Visual Studio Code як контекст для внесення змін. Після послідовного усунення всіх 41 проблеми та повторного сканування тієї самої гілки оцінка готовності зросла до 100 балів із 100. На рисунку 3.14 наведено список сканувань цього проєкту, де обидва результати відображаються поруч.

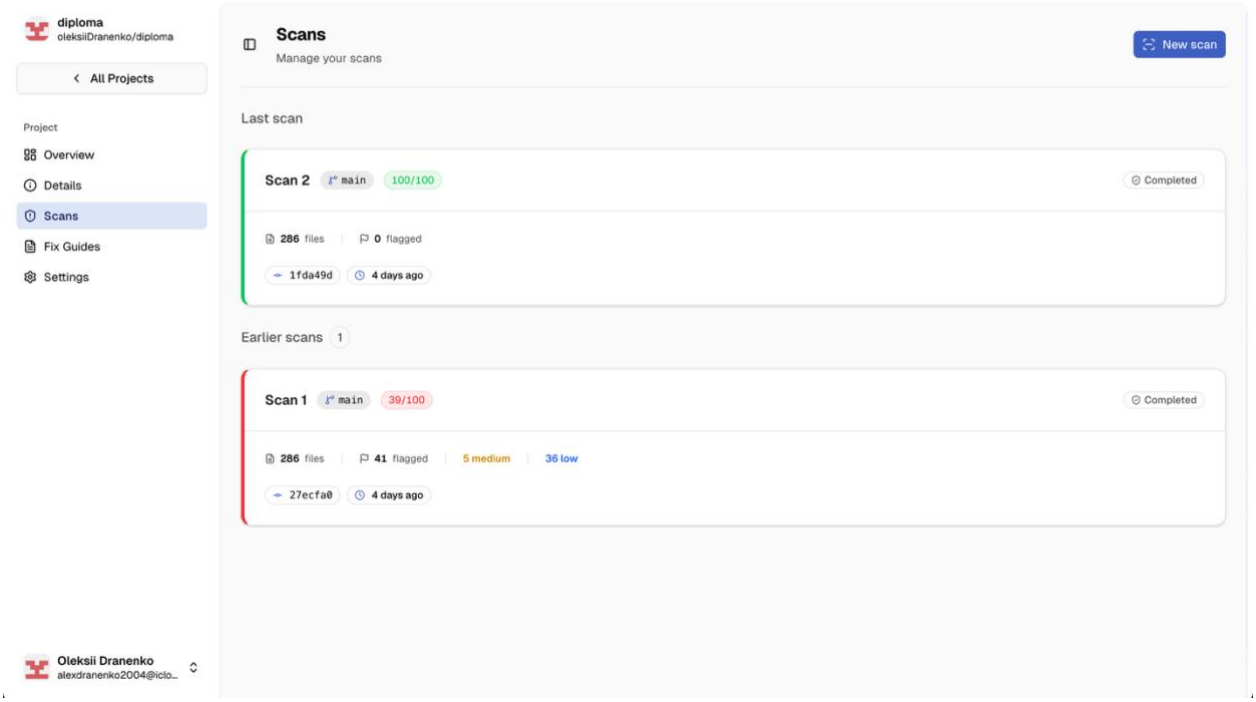

Рисунок 3.14 – Результати двох сканувань одного репозиторію до та після застосування інструкції з виправлення знайдених вразливостей

Наведений приклад демонструє, що застосунок забезпечує не лише виявлення вразливостей, а й повний замкнений цикл роботи з ними – від першого сканування до підтвердженого усунення всіх виявлених проблем.

3.3 Результати експериментального дослідження

У рамках першої групи експериментального дослідження проаналізовано 15 публічних репозиторіїв, у яких було знадено явні ознаки використання ШІ-інструментів для генерації коду. Критерієм відбору слугувала наявність у репозиторії щонайменше однієї з таких ознак: пряме посилання на платформи ШІ-генерації коду у файлі README, або присутність файлів конфігурації ШІ-асистентів, зокрема CLAUDE.md. Відібрані репозиторії є повноцінними вебзастосунками або серверними застосунками на базі Node.js та TypeScript різного рівня складності, від простих телеграм-ботів до повностекових застосунків з авторизацією та базами даних.

На рисунку 3.15 наведено знімок панелі керування застосунку після завершення всіх 15 сканувань. Список відсортовано за оцінкою готовності від найнижчої до найвищої, що одразу виявляє найбільш проблемні репозиторії.

Project	Issues	Score	Last Scanned
FinAi	4 Critical, 7 High, 15 Medium, 347 Low	1/100	about 17 hours ago
Task-Management-System-	2 Critical, 3 High, 5 Medium, 14 Low	2/100	about 17 hours ago
backend-twt	1 Critical, 5 High, 3 Medium, 2 Low	6/100	14 minutes ago
Vibes-Social	6 High, 10 Medium, 22 Low	7/100	39 minutes ago
api-personal-finance	5 High, 1 Medium, 4 Low	18/100	42 minutes ago
bot-english-backend	3 High, 3 Medium	25/100	22 minutes ago
quantumvest	3 High, 31 Low	29/100	about 17 hours ago
opnet-multisender	2 High, 1 Medium, 11 Low	36/100	13 minutes ago
minicode	2 High, 3 Medium, 1 Low	36/100	24 minutes ago
feedback-app	1 Critical, 2 High, 1 Medium, 2 Low	41/100	about 1 hour ago
kleinhacks-whisper	6 Medium, 11 Low	45/100	1 minute ago
Gov-Apis-Portal	3 Medium, 65 Low	47/100	about 17 hours ago
sunset	2 Medium, 231 Low	55/100	about 1 hour ago
telegram_bot_analytics	1 High, 1 Medium, 7 Low	60/100	41 minutes ago
betting	1 Medium, 22 Low	70/100	about 17 hours ago

Рисунок 3.15 – Результати сканування 15 публічних репозиторіїв на панелі керування

Детальні кількісні результати по кожному репозиторію наведено у таблиці 3.2.

Таблиця 3.2 – Результати сканування репозиторіїв, створених із використанням ШІ-інструментів

Репозиторій	**Критичний рівень**	**Високий рівень**	**Середій рівень**	**Низький рівень**	**Оцінка**
1	**2**	**3**	**4**	**5**	**6**
FinAi	4	7	15	347	1/100
Task-Management-System-	2	3	5	14	2/100
backend-twt	1	5	3	2	6/100
Vibes-Social	0	6	10	22	7/100
api-personal-finance	0	5	1	4	18/100
bot-english-backend	0	3	3	0	25/100
quantumvest	0	3	0	31	29/100
opnet-multisender	0	2	1	11	36/100
minicode	0	2	3	1	36/100
feedback-app	1	2	1	2	41/100
kleinhacks-whisper	0	0	6	11	45/100

Продовження таблиці 3.2

1	2	3	4	5	6
Gov-Apis-Portal	0	0	3	65	47/100
sunset	0	0	2	231	55/100
telegram_bot_analytics	0	1	1	7	60/100
betting	0	0	1	22	70/100

Результати демонструють стійку та однозначну закономірність. Жоден із 15 проаналізованих репозиторіїв не отримав оцінку готовності вище 70 балів, а 9 із 15 – нижче 40, що за прийнятою у застосунку шкалою відповідає статусу «критично». Середня оцінка по групі становить 32 бали зі 100. Чотири репозиторії отримали оцінку нижче 10 балів – FinAi (1/100), Task-Management-System- (2/100), backend-twt (6/100) та Vibes-Social (7/100), що свідчить не про поодинокі недоліки, а про системну відсутність базових практик безпеки у цих проєктах.

Характер виявлених вразливостей є показовим. У репозиторіях із найнижчими оцінками переважають проблеми в напрямку безпеки: відсутність обмеження частоти запитів на маршрутах автентифікації, некоректне або надмірно дозволювальне налаштування політики міжсайтових запитів, зберігання конфігураційних даних у відкритому вигляді та відсутність перевірки прав доступу на окремих маршрутах. Ці вразливості є саме тим класом проблем, що характерний для коду, згенерованого без глибокого розуміння вимог безпеки виробничого середовища: ШІ-інструменти генерують функціонально коректний код, але системні захисні механізми в ньому часто або відсутні, або реалізовані лише формально.

Окремої уваги заслуговують репозиторії з аномально великою кількістю проблем низького рівня серйозності. Репозиторій sunset містить

231 таку проблему, проте отримав оцінку 55 балів – вищу, ніж більшість репозиторіїв з меншою їх загальною кількістю. Це є безпосереднім відображенням алгоритму оцінювання: проблеми низького рівня мають обмежену сумарну вагу у формулі штрафу, тому навіть їх велика кількість не обвалює оцінку так, як кілька вразливостей критичного або високого рівня. Показовим контрастом є репозиторій bot-english-backend: він містить лише 6 позначок сумарно, проте отримав оцінку 25 балів – виключно через те, що всі 3 вразливості з них мають високий рівень серйозності. Репозиторій Gov-Apis-Portal з 65 низькими позначками та оцінкою 47 балів демонструє схожу закономірність: масштабний, але не критичний технічний борг впливає на оцінку значно м'якше, ніж поодинокі проблеми безпеки.

Критичні вразливості виявлено у 4 із 15 репозиторіїв (FinAi, Task-Management-System-, backend-twt та feedback-app). Проблеми високого рівня серйозності присутні у 10 із 15 репозиторіїв, тобто лише 4 репозиторії з нижньої частини таблиці (kleinhacks-whisper, Gov-Apis-Portal, sunset та betting) не містять жодної вразливості критичного або високого рівня, що пояснює їх відносно вищі оцінки. Вади середнього рівня серйозності виявлено у 13 із 15 репозиторіїв. Проблеми низького рівня присутні у 14 із 15 репозиторіїв (єдиний виняток становить bot-english-backend, що містить виключно вразливості середнього та високого рівнів). Загальна картина свідчить про те, що проблеми низького рівня є фоновою характеристикою практично будь-якого ШІ-згенерованого проєкту, тоді як критичні та високі вади визначають кінцеву оцінку готовності.

Отримані результати підтверджують першу гіпотезу дослідження: репозиторії, створені із залученням ШІ-інструментів генерації коду, демонструють низький рівень готовності до промислової експлуатації. Середня оцінка 32/100 по групі є значно нижчою від порогу «потребує уваги» (40 балів), а більше половини репозиторіїв потрапляють у критичну зону. Це узгоджується з висновками досліджень, наведених у першому розділі, де

зазначається, що ШІ-згенерований код містить вади безпеки у 2,74 рази частіше порівняно з кодом, написаним людиною.

3.4 Порівняльний аналіз з існуючим рішенням

Для проведення порівняльного аналізу з розробленим застосунком обрано інструмент CodeRabbit. На відміну від SonarQube та Snyk, що вимагають налаштування CI/CD-інфраструктури та розуміння процесу інтеграції інструментів статичного аналізу, CodeRabbit підключається до репозиторію GitHub за лічені хвилини через встановлення GitHub App без жодних конфігураційних файлів та без технічної підготовки. Саме ця простота входу робить його найбільш доречним інструментом для порівняння. Обидва рішення адресовані однаковій аудиторії та однаковому сценарію використання – швидка перевірка коду без розгортання власної інфраструктури.

Принципова відмінність між двома підходами полягає у точці застосування аналізу. CodeRabbit працює виключно в контексті запитів на злиття, він аналізує різницю між гілками та залишає коментарі до змін у коді. Розроблений застосунок натомість аналізує повну кодову базу у будь-який момент, незалежно від наявності активних запитів на злиття. Щоб забезпечити коректне порівняння в умовах, де обидва інструменти мають можливість перевірити один і той самий код, для цілей дослідження створено спеціальний тестовий репозиторій із двома гілками. Основна гілка main містить чистий базовий застосунок без вад. У другій гілці vulnerabilities зосереджено всі навмисно внесені вади, після чого з неї створено запит на злиття до main. Це дозволило CodeRabbit проаналізувати різницю між гілками у контексті запиту на злиття, а розробленому застосунку виконати повне сканування гілки vulnerabilities як цілісної кодової бази.

До тестового репозиторію навмисно внесено 35 вразливостей, що охоплюють два напрямки аналізу: безпеку та якість коду. Проблеми безпеки

включають відсутність глобальних захисних проміжних обробників, некоректну конфігурацію автентифікації та сесій, неправильне використання криптографічних функцій, вразливості типу ін'єкцій та витік чутливих даних у відповідях сервера. Проблеми якості коду включають порожні блоки обробки помилок, відсутність маршруту перевірки стану системи та використання небезпечних конструкцій TypeScript. Повний перелік внесених вад із зазначенням категорії та рівня серйозності наведено у таблиці 3.3.

Таблиця 3.3 – Перелік вразливостей, навмисно внесених до тестового репозиторію

№	Вразливість	Напрямок	Серйозність
1	**2**	**3**	**4**
1	Відсутній обробник обмеження частоти запитів у файлі точки входу	Безпека	Висока
2	Відсутній глобальний проміжний обробник автентифікації	Безпека	Висока
3	Відсутній імпорт та використання бібліотеки helmet	Безпека	Середня
4	jwt.sign()без параметра expiresIn	Безпека	Висока
5	jwt.sign() з алгоритмом none	Безпека	Критична
6	res.cookie() без прапорів httpOnly, secure та sameSite	Безпека	Висока
7	credentials: true у CORS-конфігурації разом з origin: "*"	Безпека	Критична
8	session() з секретом написаним в коді	Безпека	Висока

Продовження таблиці 3.3

1	2	3	4
9	Math.random().toString(36) як токен	Безпека	Висока
10	Math.random() присвоєно змінній з іменем secret, token або password	Безпека	Висока
11	randomBytes(4) – замала кількість байт для безпечного токена	Безпека	Середня
12	createHash("md5") або createHash("sha1")	Безпека	Середня
13	res.json({ password: user.password }) – чутливе поле у відповіді	Безпека	Висока
14	bcrypt.hash(password, 6) – кількість раундів нижче 10	Безпека	Висока
15	new RegExp(req.query.search) – ін'єкція регулярного виразу	Безпека	Висока
16	res.redirect(req.query.next) – відкрите перенаправлення	Безпека	Висока
17	res.setHeader() зі значенням з req.body – ін'єкція заголовка	Безпека	Середня
18	Object.assign({}, req.body) – забруднення прототипу	Безпека	Висока
19	fetch(req.body.url) – підробка запитів на стороні сервера (SSRF)	Безпека	Критична
20	mongoose.find({ $where: ... }) – NoSQL-ін'єкція	Безпека	Критична

Продовження таблиці 3.3

1	2	3	4
21	createCipher()– застарілий API	Безпека	Середня
22	res.status(500).json({ error: err.message }) — докладне повідомлення помилки	Безпека	Середня
23	console.log(req.body) — журналювання тіла запиту	Безпека	Низька
24	Різні повідомлення для "користувач не знайдений" та "невірний пароль"	Безпека	Середня
25	err.stack надсилається у тілі відповіді	Безпека	Середня
26	bodyParser.json() без параметра limit	Безпека	Низька
27	debug: true у конфігураційному об'єкті	Безпека	Низька
28	NODE_TLS_REJECT_UNAUTHORIZED=0 у файлі середовища	Безпека	Критична
29	Порожній блок catch — помилка поглинається мовчки	Якість коду	Середня
30	Блок catch повертає лише null або undefined	Якість коду	Низька
31	Блок catch викликає лише console.log — викликач не може виявити збій	Якість коду	Низька
32	.then() без .catch() на наступному рядку	Якість коду	Низька
33	Відсутній маршрут перевірки стану системи (/health, /ping)	Якість коду	Середня

Продовження таблиці 3.3

1	2	3	4
34	as unknown as SomeType — подвійне приведення типів в TypeScript	Якість коду	Низька
35	// @ts-nocheck на початку файлу	Якість коду	Низька

Сканування тестового репозиторію розробленим застосунком виконано на гілці vulnerabilities. За результатами сканування виявлено 48 підтверджених позначок. Оцінка готовності до промислової експлуатації склала 1 бал зі 100, що відповідає статусу «критично». На рисунку 3.16 наведено сторінку результатів сканування із загальними показниками.

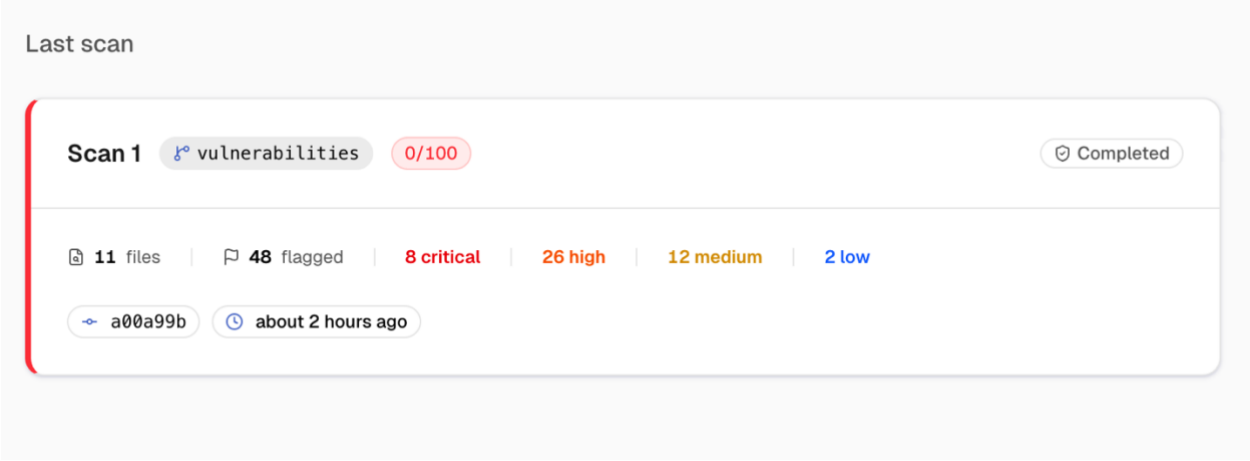

Рисунок 3.16 – Сканування тестового репозиторію розробленим застосунком

З 35 вразливостей, навмисно внесених до тестового репозиторію, застосунок виявив 32. Три вразливості не отримали окремих позначок: відсутність прапорів secure та sameSite на cookie (охоплено суміжною позначкою щодо httpOnly), вразливість щодо витоку err.stack у відповіді та різні повідомлення для «користувач не знайдений» і «невірний пароль». Водночас застосунок виявив 13 додаткових вразливостей, що виходять за

межі навмисно внесених, переважно у напрямках продуктивності та якості коду: відсутність заголовків Cache-Control на маршрутах, відсутність middleware стиснення відповідей та необроблені відхилення Promise. Загальний показник виявлення навмисно внесених вразливостец становить 32 з 35, тобто 91%.

CodeRabbit проаналізував запит на злиття та залишив коментарі до 14 файлових фрагментів. З 35 навмисно внесених вразливостей CodeRabbit виявив 14, що становить 40% від загальної кількості. Серед знайденого CodeRabbit коректно ідентифікував найбільш очевидні критичні проблеми. Більшість цих знахідок супроводжуються конкретними пропозиціями щодо виправлення з готовим кодом.

З 35 навмисно внесених вад CodeRabbit пропустив 60%. Пропуски стосуються переважно двох категорій.

По-перше, вразливості, що потребують семантичного розуміння контексту використання. Відсутність прапорів httpOnly, відсутність параметра expiresIn у jwt.sign(), використання randomBytes(4) з недостатньою кількістю байт – всі ці вразливості вимагають розуміння того, що конкретне значення або відсутній параметр є проблемою в контексті безпеки, а не просто стилістичним недоліком. CodeRabbit залишив коментар щодо Math.random(), але не помітив randomBytes(4) як недостатньо випадковий, і не відреагував на bcrypt з 6 раундами.

По-друге, вся категорія вад якості коду. CodeRabbit не виявив жодної з 7 проблем якості коду: порожніх блоків catch, блоків catch що повертають null, подвійного приведення типів «as unknown as T» та @ts-nocheck. Також пропущено вразливості витоку даних середнього рівня: докладне повідомлення помилки з err.message, вивід req.body у консоль та різні повідомлення для «користувач не знайдений» і «невірний пароль» що уможливлює перебір користувачів. Вразливості ін'єкції ненасиченого середнього рівня також залишилися без коментарів.

Підсумовуючи, розроблений застосунок виявив 91% навмисно внесених вразливостей, а CodeRabbit 40%. Оскільки основна гілка містить лише базове налаштування, а всі файли з вразливостями вносяться в одному запиті на злиття, CodeRabbit мав доступ до повного вмісту кожного файлу, тобто різниця у результатах не пов'язана з обмеженням контексту. Вона пояснюється підходом до аналізу: CodeRabbit покладається виключно на велику мовну модель, яка добре розпізнає очевидні та широко відомі патерни вразливостей, але пропускає ті, що не мають виразного синтаксичного прояву (відсутні елементи, помилки у значеннях параметрів, архітектурні прогалини та проблеми якості коду). Розроблений застосунок поєднує детермінований статичний аналіз із верифікацією через велику мовну модель: статичний аналізатор забезпечує систематичне охоплення за чітко визначеним каталогом правил, а модель підтверджує знахідки та відсіває хибно-позитивні спрацьовування. Такий гібридний підхід дозволяє виявляти вразливості, які чиста ШІ-перевірка стабільно пропускає незалежно від обсягу наданого контексту.

ВИСНОВКИ

У рамках кваліфікаційної роботи розроблено і реалізовано веборієнтовану систему для автоматизованого інкрементального аналізу та верифікації вихідного коду програмного забезпечення. Робота охоплює повний цикл створення програмного продукту – від аналізу наявних аналогічних рішень до практичної реалізації та експериментального оцінювання запропонованого хмарного інструментарію із використанням сучасних засобів розроблення, завдяки чому вирішено такі основні завдання:

- сформовано детальні технічні вимоги до системи, обґрунтовано критерії вибору технологічного стеку та спроєктовано адаптивний інтерфейс користувача з інтеграцією протоколу безпечної віддаленої авторизації (GitHub Open Authorization);
- досліджено методи аналізу абстрактних синтаксичних дерев для мови програмування JavaScript, на основі чого розроблено перший рівень архітектури системи – детермінований сканер для первинного виявлення потенційно небезпечних конструкцій у коді;
- розроблено другий рівень архітектури – інтелектуальний модуль верифікації на основі штучного інтелекту, що взаємодіє з інтерфейсом програмування моделі Gemini 2.5 Flash за допомогою оптимізованих бінарних запитів для контекстного аналізу фрагментів коду та відсікання хибно-позитивних результатів;
- реалізовано алгоритми автоматичного визначення технологічного стеку аналізованого проєкту (зокрема бібліотек Prisma, Mongoose, Express) для динамічної адаптації правил визначення під структуру коду;
- обґрунтовано та програмно впроваджено математичну модель розрахунку інтегрального показника відповідності промисловим стандартам за метриками безпеки, надійності й продуктивності програмного коду;

- спроєктовано схему збереження результатів сканування у базі даних MongoDB з підтримкою історії аналізів, а також розроблено генератор структурованих інструкцій з усунення виявлених дефектів коду.
- проведено експериментальну апробацію застосунку на вибірці з 15 публічних репозиторіїв, створених із використанням ШІ-інструментів генерації коду, за результатами якої середня оцінка готовності по групі склала 32 бали зі 100, що підтвердило практичну цінність розробленого інструменту для виявлення вразливостей у такому класі проєктів.

Розглянуто існуючі інструменти статичного аналізу зі схожим функціоналом та проаналізовано науково-технічні літературні джерела, що стосуються сучасних задач верифікації програмного коду. Вивчено наявні методи виявлення дефектів, особливості їх застосування на рівні окремих файлів та у контексті метаданих проєкту, а також проаналізовано напрями зниження рівня хибно-позитивних спрацьовувань за допомогою інтелектуальних систем. Для практичної реалізації модулів верифікації та контекстного аналізу використано можливості великої мовної моделі Gemini 2.5 Flash, а механізм розподілу фонових завдань побудовано на основі черги BullMQ. Вся серверна та клієнтська логіка системи реалізована за допомогою програмних платформ Next.js та Node.js.

Проведено комплексне тестування працездатності розробленого вебзастосунку, зокрема перевірку коректності підключення репозиторіїв, стабільності асинхронного оброблення завдань та точності обчислення зведеної оцінки готовності проєкту до промислової експлуатації. Для оцінки ефективності гібридного підходу виконано порівняльний аналіз із інструментом CodeRabbit на спеціально підготовленому тестовому репозиторії з 35 навмисно внесеними вразливостями: розроблений застосунок виявив 32 з них (91%), тоді як CodeRabbit – 14 (40%). Різниця пояснюється поєднанням детермінованого статичного аналізу з верифікацією через велику мовну модель, що дозволяє виявляти архітектурні прогалини та вади якості коду, які чиста ШІ-перевірка стабільно пропускає.

Розглянуто перспективи подальшого покращення та розвитку системи, які включають у себе розширення каталогу детермінованих правил аналізу для інших мов програмування, вдосконалення запитної інженерії для підвищення деталізації інструкцій з виправлення коду, а також інтеграцію розробленого інструментарію безпосередньо у процеси безперервної інтеграції та розгортання (CI/CD) програмних продуктів. Результати кваліфікаційної роботи можуть бути корисними у сфері автоматизації процесів контролю якості коду, а також при створенні допоміжних інтелектуальних систем для нетехнічних авторів програмного забезпечення.

Результати роботи апробовано у вигляді тез доповіді під час Міжнародного молодіжного форуму «Радіоелектроніка і молодь у XXI столітті» [15].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Collins Dictionary. (2025). *Collins word of the year 2025*. HarperCollins Publishers, 1-2.

2. GitHub. (2026). *GitHub Octoverse 2025: The state of open source and AI*. GitHub, Inc, 1-15.

3. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2785-2799.

4. CodeRabbit. Documentation. URL: https://docs.coderabbit.ai/ (дата звернення 01.06.2026).

5. Sandoval, G., Chaparro, O., Alsharif, S., Pasareanu, C. S., & Poshyvanyk, D. (2023). Empirical evaluation of GitHub Copilot's code suggestions. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1418-1430.

6. Snyk. (2025). *The state of open source security report 2025*. Snyk Ltd, 1-12.

7. Brito, T., Ferreira, M., Monteiro, M., Lopes, P., Barros, M., Santos, J. F., & Santos, N. (2023). *Study of JavaScript static analysis tools for vulnerability detection in Node.js packages* (arXiv Report No. 2301.05097), 1-14.

8. OWASP Foundation. (2021). *OWASP Top Ten 2021, 1-2*.

9. FIRST.org. (2019). *Common Vulnerability Scoring System v3.1: Specification document, 1-24*.

10. Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. *2013 IEEE/ACM 35th International Conference on Software Engineering (ICSE)*, 712-721.

11. Kula, R. G., De Roover, C., German, D. M., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, *23*(1), 384-417.

12. Thongtanunam, P., Kula, R. G., Cruz, A. M. R., & Yoshida, N. (2016). Investigating code review practices for web applications. *Proceedings of the International Workshop on Conducting Empirical Studies in Industry*, 27-32.

13. Bohdan, N., Tvoroshenko, I., Gorokhovatskyi, V., & Kobylin, O. (2025). Development of a hybrid method to enhance context memory for a chatbot application based on large language models. *International Journal of Academic Information Systems Research*, *9*(10), 7-18.

14. Suprun, A., Tvoroshenko, I., Gorokhovatskyi, V., & Yakovleva, O. (2025). Development and research of a method for the combined use of large language models for text generation. *International Journal of Academic and Applied Research*, *9*(10), 249-263.

15. Драненко, О. А. (2026). Інструмент автоматизованого аналізу production compliance Node.js застосунків. *Матеріали XXVI Міжнародної науково-практичної конференції «Topical Issues of Practice and Science»*, 50-52.

16. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *2022 IEEE Symposium on Security and Privacy (SP)*, 754-768.

17. Snyk Ltd. Documentation. URL: https://docs.snyk.io/getting-started-guides/getting-started (дата звернення 01.06.2026).

18. Node.js Security Working Group. (2025). *Node.js Security Guidelines and Best Practices, 1-2*.

19. Express.js Foundation. (2025). *Express Security Best Practices, 1-2*.

20. Pattabiraman, K., Nakamura, T., Artho, C., Tanaka, S., Kiezun, A., & Gmeiner, B. (2020). Rate limiting and throttling for web services. *ACM Transactions on the Web*, *14*(2), 1-26.

21. Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., & Esmeir, S. (2019). Browser security: Lessons from Google Chrome. *IEEE Security & Privacy*, *17*(5), 70-79.

22. Saxe, J., & Sanders, R. (2018). *Malware data science: Detecting, classifying, and analyzing malicious code*. No Starch Press, 23-35.

23. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). What makes a good bug report? *IEEE Transactions on Software Engineering*, *36*(5), 618-643.

24. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill, 519-557.

25. Yakovleva, O., Matúšová, S., Liubchenko, V., & Maksimov, H. (2025). Innovative solutions based on AI in education, on the example of generating multimodal lecture notes. *Scientific Journal of Bratislava University of Economics and Management «Public Administration and Regional Development, Economics, Management and Marketing»*, 21 (1), 140-163.

26. Творошенко І.С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. *Харків: ХНУРЕ*.