ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтелектуальна система управління для публічних проєктів Github»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро ГРИГОРЧУК
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Дмитро ГРИГОРЧУК

Керівник: Віталій ЗАЛИВА
доктор філософії(PhD)

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ___ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Григорчуку Дмитру Івановичу

1. Тема кваліфікаційної роботи: «Інтелектуальна система управління для публічних проєктів Github»,
керівник кваліфікаційної роботи доктор філософії (PhD) Віталій ЗАЛИВА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про принципи управління репозиторіями програмного коду, аналіз сучасних підходів до інтеграції штучного інтелекту у DevOps-інструменти, технічна

документація щодо REST API провайдерів хостингу коду (GitHub, GitLab, Bitbucket), методи побудови повностекових web-застосунків.

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

1. аналіз предметної області управління репозиторіями та існуючих аналогів CRM-систем для розробників;
2. визначення вимог та проєктування архітектури повностекового web-застосунку;
3. реалізація серверної частини на Express 5, TypeScript, Prisma ORM та PostgreSQL;
4. реалізація клієнтської частини на React 19, TypeScript та Bootstrap 5;
5. інтеграція AI-аналітики з підтримкою мультипровайдерності (OpenAI, Gemini, DeepSeek);
6. тестування на трьох рівнях: модульне, компонентне (Storybook), наскрізне (Playwright);
7. розробка демонстраційних матеріалів та оформлення проєкту.

5. Перелік графічного матеріалу: презентація.

1. аналіз аналогів;
2. вимоги до програмного забезпечення;
3. програмні засоби реалізації;
4. архітектура web-застосунку;
5. діаграма варіантів використання;
6. схема бази даних;
7. екранні форми;
8. апробація результатів дослідження.

6. Дата видачі завдання «23» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02–28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03–07.03.2026	
3	Огляд методик реалізації повностекових web-застосунків з AI-інтеграцією	08.03–17.03.2026	
4	Проєктування архітектури та схеми бази даних	18.03–02.04.2026	
5	Програмна реалізація серверної та клієнтської частин	03.04–19.04.2026	
6	Тестування застосунку на трьох рівнях	20.04–28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04–11.05.2026	
8	Розробка демонстраційних матеріалів	12.05–18.05.2026	
9	Попередній захист роботи	11.05–22.05.2026	

Здобувач вищої освіти _____ Дмитро ГРИГОРЧУК

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА

РЕФЕРАТ

Текстова частина бакалаврської кваліфікаційної роботи має обсяг 65 сторінок, у ній наведено 4 таблиці, 25 рисунків та 30 використаних джерел.

Мета роботи - підвищення ефективності управління репозиторіями програмного коду шляхом створення єдиного інтерфейсу з інтеграцією засобів штучного інтелекту для аналізу гілок, задач, запитів на злиття та виявлення проблем у коді.

Об'єкт дослідження - процес агрегації, моніторингу та інтелектуального аналізу репозиторіїв програмного коду з різних платформ хостингу (GitHub, GitLab, Bitbucket).

Предмет дослідження - повностековий веб застосунок для управління репозиторіями з AI-аналітикою (PRISM AI), що базується на сучасному стеку технологій React, Express, TypeScript, PostgreSQL та Prisma ORM.

Короткий зміст роботи: У роботі проаналізовано предметну область управління репозиторіями програмного коду та інструменти DevOps-аналітики. Проведено огляд існуючих CRM-подібних систем для розробників, визначено їх недоліки та можливості для покращення. Обґрунтовано вибір сучасного технологічного стеку: React з TypeScript, Express, PostgreSQL з Prisma ORM, Bootstrap, Docker Compose. Спроєктовано та реалізовано повностековий веб застосунок PRISM AI, який агрегує метрики репозиторіїв з GitHub, GitLab та Bitbucket в єдиному інтерфейсі, забезпечує AI-аналіз гілок, задач, запитів на злиття та автоматичний код-рев'ю з підтримкою п'яти провайдерів ШІ (OpenAI, Google Gemini, DeepSeek, OpenRouter, Custom). Реалізовано систему автентифікації на JWT-токенах з bcrypt-хешуванням паролів, обмеження частоти запитів (rate-limiting), а також серверну валідацію через Zod. Застосунок контейнеризовано за допомогою Docker Compose з трьома сервісами (frontend, backend, PostgreSQL). Проведено тестування на трьох рівнях:

модульне (Jest для серверної частини, Vitest для клієнтської), компонентне (Storybook), наскрізне (Playwright). Підготовлено демонстраційні матеріали та оформлено результати дослідження.

Наукова новизна роботи виражається у виробленні уніфікованого механізму зведення репозиторіїв із різних хостингів через спільний шар абстракції провайдерів та у поєднанні ШІ-моделей із провайдерами, які налаштовує сам користувач, заради автоматичної обробки метаданих та вихідного коду.

Практична значущість виражається у створенні придатного до промислового застосування продукту, який знімає частину розумового навантаження з розробників і DevOps-інженерів під час супроводу великої кількості сховищ, дає змогу швидше знаходити дефекти у коді завдяки ШІ-аналізу та піддається адаптації для команд різного розміру.

Цей веб застосунок може застосовуватись у DevOps-аналітиці, керуванні проєктами розробки програмного забезпечення, інтеграції ШІ-засобів у процеси перевірки коду та для корпоративної звітності за репозиторіями.

КЛЮЧОВІ СЛОВА: ВЕБ ЗАСТОСУНОК, CRM, РЕПОЗИТОРІЙ, ШТУЧНИЙ ІНТЕЛЕКТ, GITHUB, GITLAB, BITBUCKET, REACT, EXPRESS, TYPESCRIPT, POSTGRESQL, PRISMA, REST API, DOCKER, JWT, DEVOPS, СТАТИЧНИЙ АНАЛІЗ КОДУ, КОД-РЕВ'Ю.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API, Application Programming Interface (програмний інтерфейс застосунку)

AI, Artificial Intelligence (штучний інтелект)

CI/CD, Continuous Integration та Continuous Deployment (безперервна інтеграція та постачання)

CRM, Customer Relationship Management (керування відносинами з клієнтами)

CSS, Cascading Style Sheets

DTO, Data Transfer Object (об'єкт передачі даних)

DevOps, Development Operations (об'єднання розробки та експлуатації)

DXA, одиниця вимірювання у форматі OOXML (1/1440 дюйма)

HTTP, HyperText Transfer Protocol

HTML, HyperText Markup Language

JS, JavaScript

JSON, JavaScript Object Notation

JWT, JSON Web Token

LLM, Large Language Model (велика мовна модель)

MVP, Minimum Viable Product (мінімально життєздатний продукт)

ORM, Object-Relational Mapping (об'єктно-реляційне відображення)

PR, Pull Request (запит на злиття)

REST, REpresentational State Transfer

SPA, Single Page Application (односторінковий застосунок)

SQL, Structured Query Language

TS, TypeScript

UI, User Interface (користувацький інтерфейс)

UX, User Experience (користувацький досвід)

VCS, Version Control System (система контролю версій)

XSS, Cross-Site Scripting

ВСТУП

Актуальність теми: Стрімкий поступ галузі створення програмного забезпечення разом із масовим переходом команд на DevOps-практики за останнє десятиліття спричинили багаторазове збільшення обсягу репозиторіїв із вихідним кодом, з якими розробникам доводиться мати справу щодня. Типова інженерна команда нині супроводжує десятки активних сховищ, які розосереджені одночасно по кільком сервісам хостингу, серед яких лідерами лишаються GitHub, GitLab і Bitbucket. Через цю розрізненість виникають помітні незручності: інженерам доводиться раз у раз переходити з одного інтерфейсу до іншого, причому кожен сервіс по-своєму подає інформацію, формує метрики та надсилає сповіщення. Як підсумок, страждає продуктивність, втрачається фокус, а на формування цілісного уявлення про стан проєкту витрачається помітно більше часу.

Окремий пласт труднощів пов'язаний з опрацюванням даних, що накопичуються у самих сховищах. Метадані щодо коммітів, запитів на злиття, гілок та задач несуть у собі вагомі сигнали щодо стану проєкту, темпу команди, ймовірних ризиків технічного боргу та слабких місць у кодовій базі. Проте подаються вони у вигляді необробленого тексту, тож для отримання висновків інженер змушений переглядати та інтерпретувати їх власноруч. На практиці це призводить до того, що навіть звичайне завдання, на кшталт огляду змін у певній гілці за останній тиждень, перетворюється на трудомістку операцію.

Прогрес у сфері великих мовних моделей (Large Language Models, LLM) та доступність публічних API від таких постачальників, як OpenAI, Google, DeepSeek та інші, відкривають принципово інший рівень автоматизації подібних задач. Теперішні LLM можуть стисло переказувати великі обсяги тексту, помічати закономірності у вихідному коді, давати поради щодо рефакторингу та розподіляти задачі за змістом. Якщо інтегрувати такі моделі у DevOps-інструменти, можна суттєво пришвидшити

рутинні аналітичні операції. Особливу цінність має підхід, за якого користувач сам обирає, послугами якого ШІ-провайдера скористатися та яку саме модель застосувати, балансуючи між точністю результату та витратами.

Поряд із потребою в аналітиці не втрачає актуальності й питання консолідації даних із різних хостингів у єдиному робочому просторі. Хоча існує чимало інформаційних панелей під окремі сервіси (зокрема GitHub Insights або GitLab Operations Dashboard), на ринку бракує легких, відкритих та крос-платформних рішень, які водночас зводили б репозиторії з трьох провідних провайдерів та містили ШІ-аналітику, налаштовану під обраного користувачем постачальника. Розробка подібного інструмента має прикладну вагу та становить науковий інтерес як зразок інтеграції LLM у виробничі DevOps-процеси.

У рамках цієї кваліфікаційної роботи реалізовано повностековий веб застосунок PRISM AI, що втілює описану ідею. Технологічна основа проєкту така: на клієнті працюють React і TypeScript, на сервері Express із TypeScript, для зберігання даних застосовано PostgreSQL у поєднанні з Prisma ORM, автентифікація побудована на JWT, а розгортання забезпечує Docker Compose. Серед архітектурних рішень варто виокремити шар абстракції над провайдерами сховищ коду, що зводить особливості трьох різних API до уніфікованих контрактів, а також ШІ-модуль, який підтримує одразу п'ять OpenAI-сумісних провайдерів.

Об'єктом дослідження є процедура зведення, нагляду та інтелектуальної обробки сховищ вихідного коду, які зберігаються на різних сервісах хостингу.

Предметом дослідження є архітектура і програмна реалізація повностекового веб застосунку PRISM AI, що забезпечує єдиний спосіб роботи зі сховищами GitHub, GitLab та Bitbucket, а також підключення ШІ-моделей для автоматичного опрацювання метаданих і вихідного коду.

Метою роботи є зростання продуктивності роботи з репозиторіями вихідного коду через створення спільного інтерфейсу з вбудованими

ШІ-засобами для автоматичного аналізу гілок, задач, запитів на злиття та виявлення дефектів у коді.

Для реалізації цієї мети у роботі виконано такі завдання:

- виконано огляд наявних рішень для роботи з репозиторіями та CRM-подібних інструментів для розробників, з'ясовано їхні переваги та недоліки;

- окреслено цільову аудиторію та сформовано функціональні і нефункціональні вимоги до програмного продукту;

- обґрунтовано підбір технологій для серверної і клієнтської складових застосунку;

- розроблено клієнт-серверну архітектуру з тришаровою організацією бекенду (controllers, services, utils);

- спроєктовано структуру реляційної бази даних з помірною нормалізацією, що пристосована до характерних сценаріїв вибірки;

- створено REST API, що складається з 19 ендпоінтів та має автоматичну OpenAPI-документацію у Swagger UI;

- реалізовано прошарок-абстракцію над провайдерами сховищ, який зводить дані GitHub, GitLab та Bitbucket до спільних типів;

- підключено модуль ШІ-аналітики, що працює з п'ятьма постачальниками LLM та виконує п'ять видів обробки;

- збудовано клієнт у вигляді односторінкового застосунку (SPA) з вісьмома функціональними блоками та підтримкою тем;

- захищеність системи реалізовано за рахунок JWT-токенів, bcrypt-хешування паролів, обмеження частоти запитів та валідації через Zod;

- виконано контейнеризацію застосунку засобами Docker Compose з трьома сервісами;

- проведено тестування на трьох рівнях, а саме модульне, компонентне у Storybook та наскрізне у Playwright;

- сформовано демонстраційні матеріали та задокументовано отримані результати.

Методи дослідження. У дослідженні застосовано прийоми системного аналізу для розгляду предметної області, принципи об'єктно-орієнтованого програмування для організації серверної частини, шаблони проєктування «Адаптер» (під абстракцію провайдерів) і «Сервісний шар» (для розміщення бізнес-логіки), методи реляційного моделювання при побудові схеми даних, а також модульне, компонентне і наскрізне тестування для підтвердження коректності розробленого рішення.

Наукова новизна роботи полягає у формуванні уніфікованого способу інтеграції трьох незалежних REST API сховищ коду через спільний шар абстракції з єдиним типовим контрактом, завдяки чому підтримка нових провайдерів додається без втручання у бізнес-логіку. Додатковим елементом новизни є архітектура ШІ-модуля з провайдерами, налаштовуваними самим користувачем, де клієнт обирає провайдера, модель та надає власний API-ключ, а сервіс лише пересилає запити, не зберігаючи спільного централізованого ключа.

Практична значущість роботи виражається у тому, що отримано придатний до використання у промислових умовах продукт, який знижує розумове навантаження на розробників під час супроводу великої кількості сховищ, прискорює пошук дефектів завдяки автоматизованому ШІ-рев'ю, формує стислі звіти за значними обсягами метаданих (комміти, задачі, запити на злиття) та запускається командою будь-якого складу однією командою docker-compose up. До того ж створений шар абстракції провайдерів придатний до повторного використання у будь-яких інших проєктах, де потрібна одночасна робота з кількома хостингами вихідного коду.

Структура роботи. Кваліфікаційна робота охоплює реферат, перелік умовних позначень, вступ, чотири основні розділи, висновки, перелік джерел та два додатки. У першому розділі розглянуто предметну область і проаналізовано наявні аналоги. Другий розділ містить обґрунтування технологічного стеку. У третьому розділі описано архітектуру та практичну реалізацію застосунку. Четвертий розділ зосереджено на тестуванні та демонстрації отриманих результатів. Сумарний обсяг роботи становить 65 сторінок.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ УПРАВЛІННЯ РЕПОЗИТОРІЯМИ ПРОГРАМНОГО КОДУ

1.1 Управління репозиторіями як складова DevOps-процесу

Робота зі сховищами програмного коду посідає центральне місце у сучасному циклі створення програмного забезпечення. У парадигмі DevOps репозиторій сприймається не лише як місце зберігання вихідного коду, а й як точка координації командної роботи: тут одночасно ведеться історія змін, обговорюються задачі, виконується контроль якості через запити на злиття та запускаються автоматичні процеси збирання й доставки. Через збільшення масштабу продуктів та різноманіття команд звичайна компанія тепер супроводжує десятки, а часом і сотні сховищ, які можуть бути одночасно розміщені на кількох хостингах.

Розробник сьогодні постійно перебуває в умовах, коли корпоративні сховища тримаються у GitLab, відкриті бібліотеки розміщено в GitHub, а застарілі сервіси й далі живуть у Bitbucket. Така мультиплатформна обстановка породжує цілу низку складнощів: кожна служба пропонує свою UI-парадигму, відмінну модель доступу, власний формат метаданих та свої ліміти API. Унаслідок цього супровід репозиторіїв перетворюється з технічної дрібниці на окремий пласт управлінської роботи, для якого потрібні цілеспрямовані інструменти.

Якщо подивитися на щоденну роботу типового full-stack або DevOps-фахівця, можна виокремити кілька регулярних видів взаємодії зі сховищами, а саме: спостереження за активністю (нові комміти, відкриті запити на злиття, свіжі issue), перегляд змін у гілках (особливо у feature-гілках перед їхнім вливанням), відповідь на сповіщення, рев'ю коду, контроль стану CI/CD, пошук фрагмента коду у великій кодовій базі та аудит

безпеки. Більшість цих операцій або вимагає опрацювання значних обсягів тексту, або тягне за собою стрибки між платформами, а нерідко й те, й інше.

Окремо варто підкреслити роль уніфікації даних. Кожен хостинг описує сутності по-своєму: те, що у GitHub називають «issue», у GitLab іменується так само, проте має інший набір полів та інші правила переходу між станами; в Bitbucket поняття «issue» теж присутнє, але зі своїми нюансами. Pull-запити з GitHub та Bitbucket і merge-запити з GitLab по суті виконують однакову функцію, проте у відповідях API мають геть різну структуру. Гілки в усіх трьох сервісах концептуально однакові, однак отримання верхнього комміту разом з його метаданими в GitHub передбачає додатковий виклик API, тоді як GitLab та Bitbucket повертають більше деталей одразу.

Ще одне проблемне поле це аналітика, побудована на даних зі сховищ. Лідам команд та технічним керівникам потрібна інформація про швидкість розробки, регулярність ревізій, активність контриб'юторів і характерні типи задач для ухвалення технічних та управлінських рішень. Хоча платформи дають базові показники на кшталт кількості коммітів, форків чи зірок, висновки рівня «що нового сталося у проєкті за тиждень» або «які типи задач переважають у беклозі» здебільшого формуються вручну. Саме цю прогалину може закрити інтеграція ШІ-моделей у DevOps-інструментарій.

Завдяки впровадженню великих мовних моделей вдається автоматизувати цілий ряд операцій: складати лаконічні підсумки переліку коммітів обраної гілки, класифікувати задачі за тематикою, помічати «нетипові» запити на злиття (занадто великі за обсягом, без опису чи без тестів), а також формулювати рекомендації щодо рефакторингу. Принципова перевага такого підходу полягає у гнучкості: користувач сам визначає бажане співвідношення якості і витрат, обираючи модель у діапазоні від найточніших і дорогих варіантів (GPT-4o) до економних альтернатив на кшталт Gemini Flash чи DeepSeek Chat.

Якщо узагальнити, супровід репозиторіїв у нинішньому DevOps-контексті це багатовимірна задача, що включає в себе зведення даних

з різних джерел, приведення метаданих до спільного вигляду, моніторинг активності, обробку показників та реакцію на події. Створення інструмента, який охоплював би всі ці аспекти в одному робочому просторі та водночас використовував можливості ШІ, виглядає одночасно практично запитуваним та доречним за тематикою для бакалаврської роботи у галузі програмної інженерії.

1.2 CRM-підхід до управління репозиторіями

Поняття CRM (Customer Relationship Management) виникло в середовищі маркетингу та продажів як підхід до впорядкування взаємодій із клієнтами. Класичні системи цього класу, такі як Salesforce, HubSpot чи Pipedrive, пропонують спільний робочий простір, у якому зосереджені контакти клієнта, історія комунікації, поточний статус угоди та запланована активність. Користь такого підходу полягає у меншому часі на пошук потрібних даних, можливості розставляти пріоритети, упорядковувати інформацію та автоматизувати щоденну рутину.

Подібну логіку можна перенести і на управління репозиторіями. Тут аналогом «клієнта» виступає «репозиторій», що має власну історію подій, набір метрик, поточний стан та рівень пріоритету. Замість угод фігурують запити на злиття, які послідовно проходять стадії «відкритий», «на рев'ю», «змерджений» або «відхилений». Роль задач переходить до issues з власними статусами та призначеними виконавцями. У такій моделі CRM-подібний інтерфейс для інженерів природно відображає саме ті сутності, з якими розробник стикається щодня.

Основні принципи CRM, які відображені у запропонованому застосунку, такі:

- централізація даних, коли всі репозиторії доступні з одного інтерфейсу, незалежно від того, на якому хостингу вони перебувають;

- контекстна аналітика, що передбачає швидкий доступ до метаданих та ШІ-обробки для кожного окремого репозиторію;

- пріоритизація завдяки механізму обраних (favorites), за допомогою якого користувач відмічає найважливіші для нього проєкти;

- пошук та фільтрація, що включає швидкий пошук за назвою та відбір репозиторіїв за платформою хостингу;

- персоналізація, за якої кожен користувач сам формує свій набір репозиторіїв та параметри ШІ-аналітики.

Головна відмінність від традиційних CRM зосереджена у природі джерела даних. У CRM таким джерелом є власна клієнтська база, тоді як у пропонованому застосунку дані надходять з зовнішніх API хостингів сховищ. Це породжує особливі архітектурні рішення, серед яких потреба у шарі абстракції над провайдерами, кешування метаданих у локальній базі та обмеження частоти звернень до зовнішніх API. На рівні UX вся ця складність прихована від користувача: розробник бачить єдиний дашборд та працює з ним так само, як і з типовим CRM, а технічні нюанси опрацьовуються бекендом.

Інша важлива розбіжність полягає в інтеграції штучного інтелекту. У звичайних CRM ШІ зазвичай прогнозує ймовірність укладання угоди або виконує сегментацію клієнтів, тоді як у даному продукті він орієнтований на роботу з текстовим контентом і вихідним кодом. Це пояснює специфіку обраних архітектурних рішень, як-от винесення логіки в окремий сервіс project-ai.service.ts, ізольовані промпти для кожного виду аналізу та окремий обмежувач частоти (rate limiter) для ШІ-ендпоінтів, що зумовлено високою ціною викликів LLM.

1.3 Цільова аудиторія

На етапі проєктування веб застосунку було виокремлено декілька ключових сегментів цільової аудиторії, кожен з яких має власні потреби та типові сценарії використання, а саме:

- full-stack та backend-розробники, для яких важливим є оперативний огляд особистих і командних сховищ, контекст запитів на злиття та аналіз останніх змін у feature-гілках; вирішальне значення для цієї аудиторії має швидкість роботи, мінімальна кількість дій для типових сценаріїв та тісна інтеграція з GitHub чи GitLab;

- DevOps-інженери та SRE-фахівці, які щодня працюють з десятками сховищ і потребують потужних засобів агрегації, спостереження за активністю, виявлення аномалій (наприклад, надто великих запитів на злиття) та автоматичного аналізу для зменшення обсягу рутини;

- технічні ліди та архітектори, що звертаються до застосунку для координації команди, спостереження за прогресом feature-розробки, швидкої оцінки якості коду й формування звітів для стейкхолдерів;

- менеджери розробки та проджект-менеджери, які бачать загальну картину активності у проєктах без потреби заглиблюватися в особливості кожної платформи; узагальнення задач і запитів на злиття засобами ШІ допомагає швидко готувати огляди для клієнтів;

- мейнтейнери open-source-проєктів, що паралельно супроводжують кілька публічних сховищ і повинні швидко реагувати на нові issues та запити на злиття від спільноти; ШІ-класифікація задач полегшує розстановку пріоритетів;

- студенти та початківці у розробці, які користуються застосунком як навчальним середовищем для розуміння подій у сховищах відомих open-source-проєктів; ШІ-аналіз коду та зведення запитів на злиття дозволяють швидше освоїти структуру кодової бази.

Спираючись на портрет цільової аудиторії, можна сформувати такі вимоги до інтерфейсу: мінімалістична та прагматична подача без зайвого декору, швидке завантаження сторінок, прозора навігація, наявність темної та світлої тем (звична потреба для розробників), адаптивність до різних розмірів екрана (хоч основний сценарій залишається desktop-орієнтованим) та доступність базових функцій без громіздких налаштувань.

1.4 Дослідження існуючих аналогів

Щоб закласти обґрунтовану основу для проєктування, виконано огляд наявних рішень у відповідному сегменті. Цей огляд допомагає виявити переваги та недоліки існуючих продуктів, з'ясувати, чого користувачі очікують від такого типу застосунків, та визначити позиціонування власної розробки. У межах огляду розглянуто чотири найвпливовіші продукти, а саме GitHub Dashboard, GitLab Operations Dashboard, Gitify та Sourcegraph.

1.4.1 GitHub Dashboard

GitHub Dashboard це стартова сторінка GitHub за замовчуванням, яка відкривається користувачу одразу після входу. Вона побудована з кількох основних блоків: переліку нещодавніх сховищ, агрегованої стрічки активності у тих репозиторіях, на які підписаний користувач, секції «For you» з рекомендованими проєктами та зведеного блоку запитів на злиття й задач, що потребують уваги.

До переваг GitHub Dashboard належать тісна інтеграція з рештою функціоналу платформи, високий рівень опрацювання UI та UX, миттєвий доступ до робочих сутностей і якісно налаштована персоналізація стрічки. Водночас цей продукт замкнено в межах однієї платформи: якщо команда веде репозиторії одночасно у GitHub та GitLab, то частина її роботи у дашборді просто не відображається. Окремо слід зазначити відсутність ШІ-засобів аналізу всередині дашборду, адже GitHub Copilot вбудовано в

редактор коду, але не у Dashboard. Базова аналітика (insights) обмежена показниками типу кількості коммітів та активних контриб'юторів.

Висновок такий: GitHub Dashboard добре підходить командам, повністю зосередженим у межах GitHub, але не покриває мультиплатформний сценарій та не пропонує можливостей штучного інтелекту. Інтерфейс цього інструмента наведено на рисунку 1.1.

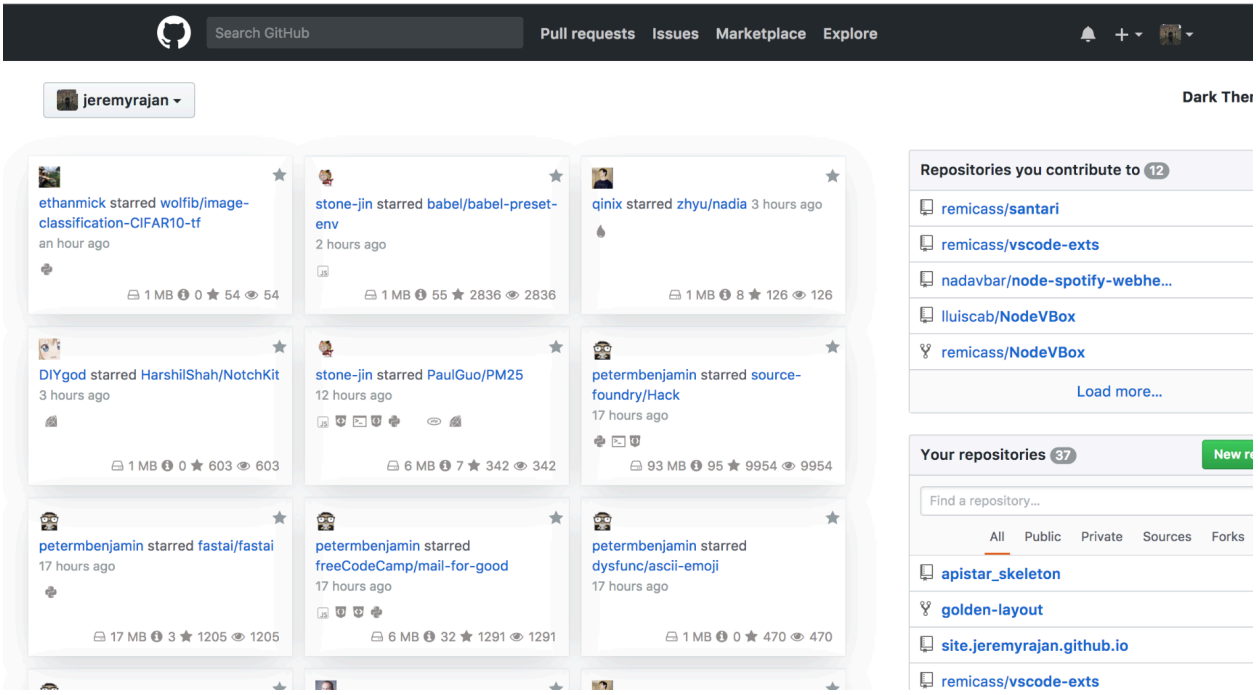


Рис. 1.1 Стартова сторінка GitHub Dashboard

1.4.2 GitLab Operations Dashboard

GitLab Operations Dashboard є спеціалізованим інтерфейсом усередині GitLab, призначеним для зведення стану CI/CD-пайплайнів та активності в обраних проєктах. Цільова аудиторія цього дашборду це переважно DevOps-фахівці, і він дозволяє переглядати поточний статус останніх пайплайнів, оцінку часу до досягнення зеленого білда та зведену статистику щодо стабільності проєктів.

Сильні сторони GitLab Operations Dashboard включають розгорнуту інформацію про стан CI/CD, продуману агрегацію для DevOps-команди, наявність метрики MTTR (Mean Time To Restore) та темну тему за

замовчуванням. Проте слабкі місця подібні до попереднього випадку, бо інструмент так само замкнено у межах GitLab, до нього неможливо додати сховища з GitHub чи Bitbucket, а ШІ-аналіз відсутній. Окремою перешкодою є те, що це enterprise-функція, доступна не у всіх тарифних планах GitLab.

Висновок: інструмент корисний для команд, що цілком сидять у GitLab і мають enterprise-ліцензію, проте він не закриває проблем мультиплатформності та інтегрованої ШІ-аналітики. Загальний вигляд інтерфейсу показано на рисунку 1.2.

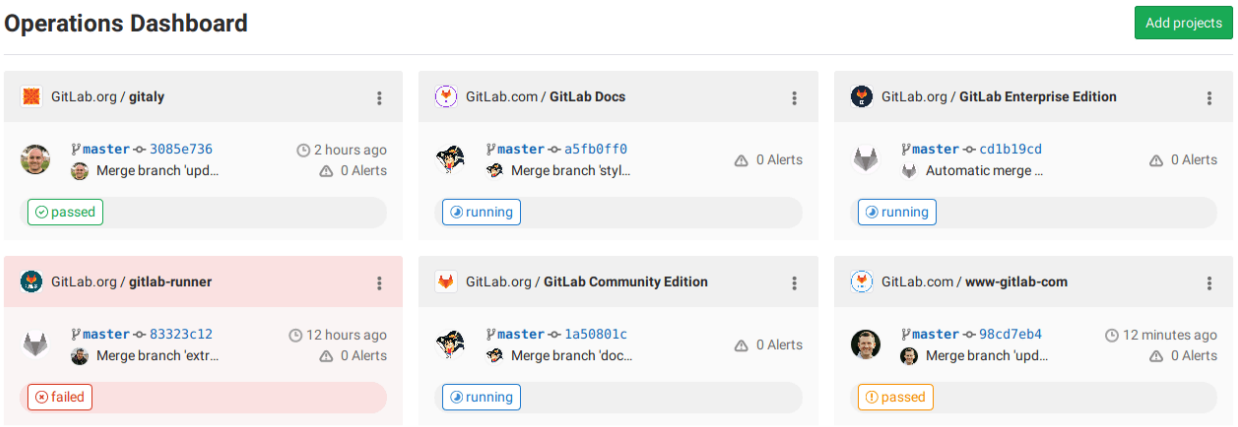


Рис. 1.2 GitLab Operations Dashboard

1.4.3 Gitify

Gitify являє собою open-source десктопний застосунок на базі Electron, що збирає сповіщення з GitHub і GitLab в одному вікні. Його основна функція полягає лише в перегляді нотифікацій (про новий запит на злиття, нову задачу або новий коментар), а повноцінного дашборду репозиторіїв він не пропонує.

До переваг Gitify належить підтримка одразу двох платформ, мінімалістичний інтерфейс, відкритий код, безкоштовність та нативна інтеграція із системними сповіщеннями macOS, Windows та Linux. У недоліках перебувають вузький функціонал, обмежений лише нотифікаціями, відсутність веб версії, брак ШІ-аналізу та неможливість керувати сховищами, оскільки доступний лише режим читання.

Висновок: Gitify добре ілюструє мінімалістичний підхід до мультиплатформності, проте обмежується лише сповіщеннями. Запропонований у роботі застосунок ширший за функціоналом і доступний як SaaS-рішення. Інтерфейс застосунку наведено на рисунку 1.3.

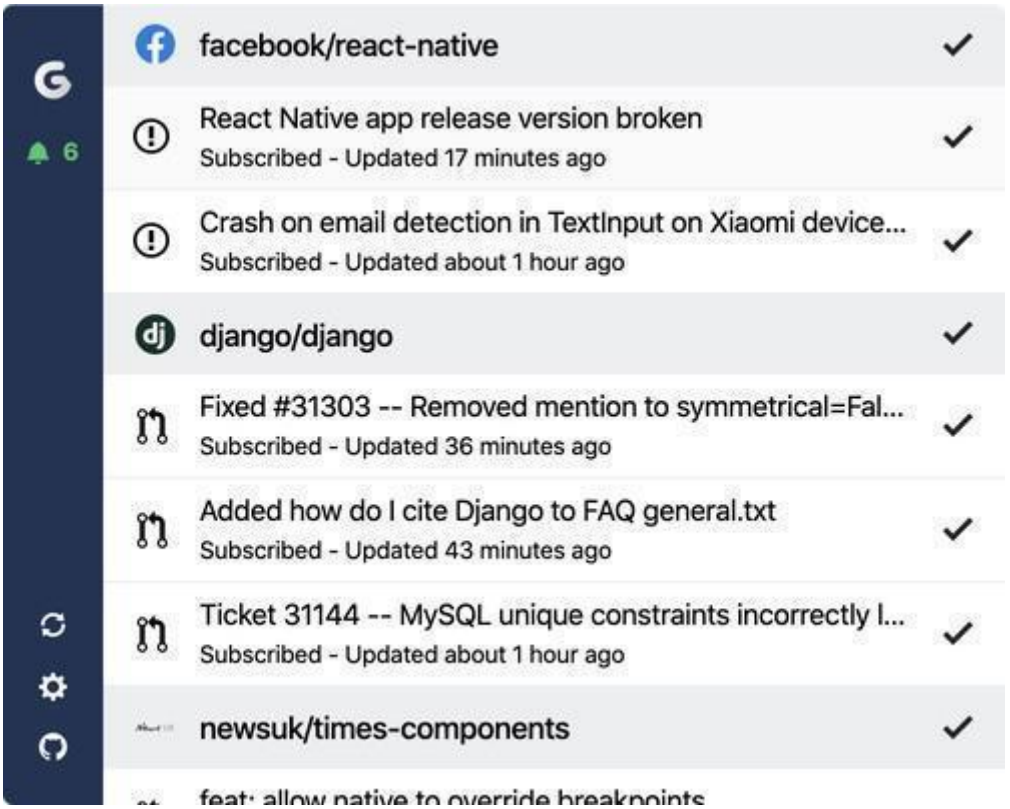


Рис. 1.3 Інтерфейс десктопного застосунку Gitify

1.4.4 Sourcegraph

Sourcegraph це комерційна платформа, націлена на пошук, навігацію та аналіз коду у великих кодових базах. Вона інтегрується з основними хостингами (GitHub, GitLab, Bitbucket, Azure DevOps), забезпечує потужний пошук за регулярними виразами та структурною подібністю, а також постачається з ШІ-асистентом під назвою Cody.

Серед переваг можна назвати всеосяжну підтримку платформ, високу глибину аналітики коду, інтеграцію зі ШІ (Cody) для відповідей на запити щодо кодової бази та корпоративний рівень якості. Серед недоліків

залишається висока вартість для невеликих команд (декілька сотень доларів на місяць у розрахунку на одного користувача), непросте розгортання self-hosted-версії та переважна спрямованість на дослідження коду, а не на CRM-стиль роботи з репозиторіями. Sourcegraph не позиціонує себе як «дашборд для щоденного огляду стану проєктів».

Висновок: Sourcegraph виступає потужним рішенням для глибокого дослідження коду у великих компаніях, проте надмірно складний та дорогий для звичайного сценарію огляду власних проєктів. Розроблений у цій роботі застосунок заповнює нішу між мінімалістичним Gitify та enterprise-орієнтованим Sourcegraph, пропонуючи доступний CRM-стиль із вбудованими ШІ-можливостями. Приклад інтерфейсу пошуку коду в Sourcegraph показано на рисунку 1.4.

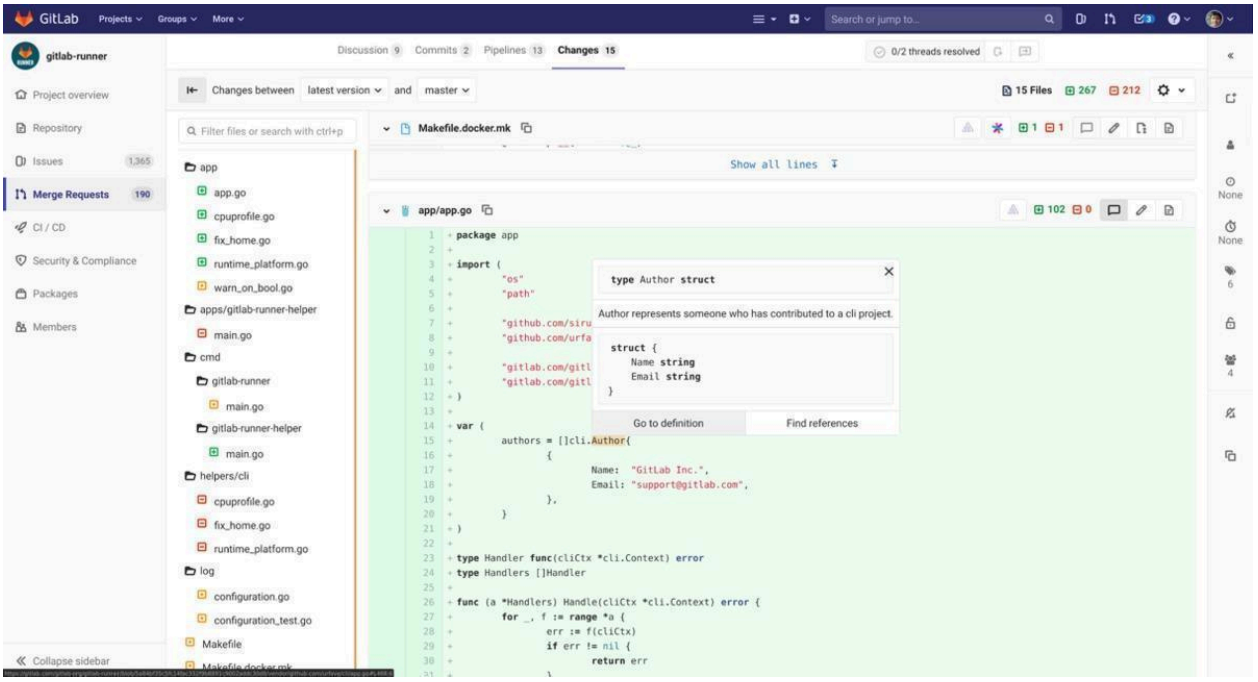


Рис. 1.4 Інтерфейс пошуку коду в Sourcegraph

1.4.5 Висновки щодо аналогів

Узагальнене зіставлення розглянутих аналогів за ключовими ознаками представлено у таблиці 1.1. Самі критерії дібрано так, щоб охопити функціональні можливості (підтримка платформ, ШІ-аналіз, агрегація

метрик), цінову доступність для звичайного користувача та технічну доступність (відкритий код, доступ через веб, складність установлення).

Таблиця 1.1

Порівняльна характеристика існуючих аналогів

Критерій	GitHub Dashboard	GitLab Ops	Gitify	Sourcegraph	PRISM AI
Підтримка GitHub	+	–	+	+	+
Підтримка GitLab	–	+	+	+	+
Підтримка Bitbucket	–	–	–	+	+
AI-аналіз даних	–	–	–	+	+
AI-рев'ю коду	–	–	–	-	+
Web-доступ	+	+	–	+	+
Open-source	–	–	+	+	+
Безкоштовно	+	-	+	–	+
Self-host	–	+	+	+	+
CRM-стиль інтерфейсу	–	–	–	–	+

З аналізу таблиці видно очевидну нішу для нашого продукту: на ринку поки що немає безкоштовного open-source-рішення, яке одночасно охоплювало б три провідні платформи хостингу, мало інтерфейс у стилі CRM та інтегрований ШІ-аналіз із налаштовуваними провайдерами. GitHub і GitLab замкнуті у межах власних екосистем, Gitify обмежується сповіщеннями, а Sourcegraph занадто складний і дорогий enterprise-варіант. PRISM AI закриває цю прогалину, орієнтуючись на типового full-stack-розробника та невеликі або середні команди.

Слід підкреслити, що жоден з розглянутих конкурентів не пропонує ШІ-аналізу з повним налаштуванням провайдера на боці користувача. Sourcegraph Cody спирається на внутрішній API, GitHub Copilot працює тільки з OpenAI, а GitLab Duo використовує власні моделі. У PRISM AI

користувач сам обирає один з п'яти провайдерів (OpenAI, Google Gemini, DeepSeek, OpenRouter або кастомний OpenAI-сумісний ендпоінт), визначає модель та підставляє свій API-ключ. У такий спосіб він отримує повний контроль над вартістю аналізу, рівнем конфіденційності даних (наприклад, можна під'єднати локально розгорнуту LLM через Ollama) та якістю отриманих відповідей.

1.5 Визначення вимог до застосунку

Спираючись на результати аналізу предметної області, портрета цільової аудиторії та конкурентного оточення, сформовано такий перелік функціональних вимог до системи:

- можливість реєстрації та входу користувачів через електронну пошту і пароль з безпечним зберіганням облікових даних;

- додавання сховищ з трьох платформ хостингу (GitHub, GitLab, Bitbucket) як за коротким посиланням, так і за повним URL;

- перегляд списку сховищ з пагінацією, пошуком за назвою, фільтрацією за платформою та позначкою «обраний»;

- сортування сховищ за різними параметрами, такими як дата створення, кількість зірок, форків і задач;

- перегляд деталізованої інформації про репозиторій, включно з гілками, задачами та запитами на злиття, з налаштуванням ліміту записів;

- ШІ-узагальнення останніх змін у вибраній гілці на основі історії коммітів;

– AI-узагальнення задач (issues) та pull-запитів;

- ШІ-рев'ю коду з виявленням ймовірних проблем та їхньою категоризацією за рівнем серйозності;

- ШІ-генерація варіантів виправлення для конкретного фрагмента коду;

- гнучке налаштування ШІ-провайдера, що включає вибір з п'яти варіантів, введення власного API-ключа, вказання моделі та базової адреси;

- позначення сховищ як обраних та можливість їхнього видалення із системи;

- оновлення метаданих репозиторію за командою користувача;

- стартовий онбординг для нових користувачів з можливістю його приховання.

Нефункціональні вимоги:

- час реакції інтерфейсу на звичайну операцію менший за 200 мс для дій, що не передбачають звернень до зовнішнього API;

- тривалість опрацювання ШІ-запиту до 30 секунд, з урахуванням обраної моделі;

- обмеження частоти запитів, а саме 5 спроб входу або реєстрації на хвилину, 20 ШІ-запитів на хвилину та 100 загальних запитів на хвилину з однієї IP-адреси;

- зберігання паролів виключно у вигляді bcrypt-хешу зі значенням salt rounds, що дорівнює 10;

– JWT-токени з терміном дії 24 години;

– Zod-валідація вхідних даних на рівні API;

- розгортання застосунку через docker-compose єдиною командою;

- покриття серверної частини модульними тестами на рівні не нижче 70 відсотків;

Виконання перелічених вимог дає змогу отримати завершений продукт промислового рівня, що відповідає сучасним очікуванням щодо якості та безпеки. У подальших розділах розкрито вибір технологічного стеку та представлено докладний опис програмної реалізації.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

Для побудови веб застосунку, що агрегує та піддає ШІ-аналізу репозиторії вихідного коду, попередньо було проведено огляд актуальних інструментів та сформовано стек, який забезпечує розумний баланс між швидкістю розробки, експлуатаційною надійністю, представленістю фахівців на ринку та відповідністю освітнім завданням кваліфікаційної роботи. У межах цього розділу обґрунтовано доцільність обрання кожного зі складників стеку.

2.1 Середовище розробки

Основним середовищем розробки обрано Visual Studio Code (VS Code), кросплатформний редактор від Microsoft, що став де-факто стандартом у веб розробці. Цей редактор є безкоштовним і має відкритий код, працює на macOS, Windows та Linux, підтримує понад 50 мов програмування через якісні мовні сервіси, побудовані на протоколі LSP (Language Server Protocol).

Серед розглянутих альтернатив були WebStorm від JetBrains та Sublime Text. WebStorm пропонує глибшу інтеграцію з JavaScript і TypeScript, потужні засоби рефакторингу та вбудовані інструменти аналізу, проте є комерційним продуктом з мінімальною ціною ліцензії приблизно 59 доларів на рік для індивідуальних користувачів. Sublime Text є швидким і легким редактором, але потребує значного налаштування, щоб отримати функціональність, що в VS Code присутня одразу після встановлення.

Ключові розширення VS Code, задіяні у цьому проєкті, такі:

- TypeScript Vue Plugin (Volar) як мовний сервіс TypeScript;
- ESLint для статичного аналізу коду JavaScript та TypeScript;
- Prettier для автоматичного форматування коду;

- Tailwind CSS IntelliSense для підказок CSS-класів (плагін залишився з ранніх стадій, хоча основним стилізатором у проєкті є Bootstrap);

- Prisma для підсвічування синтаксису та автодоповнення у prisma-схемах;

- Docker для керування контейнерами безпосередньо з редактора;

- GitLens, що додає розширені можливості Git, такі як blame, історія та порівняння;

- REST Client для тестування HTTP-запитів прямо з редактора;

- Error Lens для підкреслення помилок безпосередньо у тексті коду;

- Live Share для спільного редагування коду разом із керівником.

У VS Code присутній вбудований термінал, завдяки якому можна одночасно правити код і виконувати команди (npm, git, docker) без переходу між вікнами. Підтримка системи контролю версій Git реалізована безпосередньо в редакторі та не потребує встановлення додаткових утиліт.

2.2 Мова програмування TypeScript

Як основну мову програмування для всіх компонентів застосунку (клієнтська частина, сервер і тести) обрано TypeScript, статично типізовану надбудову над JavaScript, створену компанією Microsoft. TypeScript транспілюється у звичайний JavaScript, отже зберігає доступ до всієї JS-екосистеми бібліотек, водночас даючи переваги статичної типізації, серед яких виявлення помилок ще на етапі компіляції, повноцінне автодоповнення в IDE та безпечний рефакторинг.

Альтернативи, що розглядалися:

- JavaScript ES2022 як спрощений варіант без типів, проте при цьому втрачається типобезпека, що є критичним для великих проєктів;

- Python з FastAPI або Django, який добре підходить для бекенду, однак вимагає писати фронтенд іншою мовою, що ускладнює використання спільних DTO-типів;

- Go з фреймворками Gin або Fiber дає швидку серверну частину, але вимагає окремої роботи з шаблонами та клієнтським кодом;

- Rust з Axum або Actix забезпечує найвищу продуктивність, проте має значний поріг входу та надмірну складність для проєкту рівня бакалаврської роботи.

Вибір на користь TypeScript є оптимальним з ряду міркувань:

- єдина мова на клієнтській та серверній сторонах, що спрощує спільне використання типів, валідаційних схем і допоміжних утиліт;

- зрілий інструментарій, до якого належать компілятор tsc, лінтер ESLint з TypeScript-плагіном, форматер Prettier, а також тестові фреймворки Jest, Vitest і Playwright;

- багата екосистема типізованих бібліотек, оскільки фактично всі сучасні npm-пакети постачаються разом з типами (вбудованими або у вигляді пакетів @types/*);

- велика й активна спільнота разом з якісною документацією, що полегшує пошук готових рішень порівняно з вузькоспеціалізованими мовами;

- сприятлива ситуація на ринку праці, де позиції full-stack-розробників на TypeScript широко представлені, тому набуті в межах проєкту навички стають практично корисними для працевлаштування.

У межах проєкту застосовується TypeScript у строгому режимі (strict: true), який вмикає всі ключові опції типобезпеки, зокрема noImplicitAny, strictNullChecks, strictFunctionTypes та noImplicitThis. Клієнтську частину компілює Vite (під капотом використовується ESBuild), а серверну збирає офіційний tsc у директорію dist/.

2.3 Frontend: React 19 та Vite

Клієнтська частина продукту побудована на React, провідній бібліотеці для створення інтерфейсів від компанії Meta (Facebook). React реалізує

компонентну парадигму, у якій інтерфейс описується як композиція функціональних компонентів, що приймають props та повертають JSX-вузли.

Чому саме React, а не альтернативи:

- Vue.js, який легше освоїти, проте має скромнішу екосистему сторонніх компонентів та помітно меншу кількість вакансій на ринку;

- Angular, повноцінний фреймворк з вбудованим маршрутизатором, формами та HTTP-клієнтом, проте з істотно вищим порогом входу і зайвою складністю для проєкту подібного масштабу;

- Svelte з нестандартним підходом без віртуального DOM, проте менш зрілий і з обмеженою екосистемою;

- Solid.js з максимальною продуктивністю, але вузьким нішевим застосуванням і невеликою спільнотою.

Версія React, з якою працює проєкт, дає кілька важливих нововведень, серед яких використано наступні:

- удосконалена робота з механізмом Suspense та асинхронними компонентами;

- нові хуки useFormStatus та useFormState для зручної роботи з формами;

- вбудована підтримка концепції Actions, орієнтованої на серверні дії;

- оптимізована гідрація сторінок під час серверного рендерингу.

У ролі складальника застосовано Vite, сучасний інструмент авторства Evan You (творця Vue.js), що показує значну перевагу за швидкістю порівняно з традиційним Webpack. На етапі розробки Vite спирається на ESM (ES Modules), що дає миттєвий старт dev-сервера та hot reload із затримкою до 50 мс. Для production-збирання Vite використовує вбудований Rollup.

Маршрутизацію реалізовано на React Router DOM, який підтримує одночасно декларативний та імперативний стилі навігації. Для роботи з формами обрано React Hook Form, бібліотеку, що мінімізує кількість зайвих перерендерів завдяки uncontrolled-компонентам та інтегрується з валідацією на основі Zod-схем.

У ролі HTTP-клієнта використано Axios, обгортку над fetch API із зручним налаштуванням базового URL, заголовків і обробки помилок. У проєкті створено єдиний екземпляр api.ts з перехоплювачем (interceptor), який автоматично підставляє JWT-токен з localStorage до кожного запиту.

2.4 Backend: Express 5 та Node.js 22

Серверна частина реалізована на базі Express, мінімалістичного та гнучкого веб фреймворку для Node.js. Express лишається найпопулярнішим вибором для побудови REST API на Node.js завдяки своїй простоті, зрілості, величезній екосистемі middleware-модулів та активній спільноті. Останній мажорний реліз став першим за майже десятиліття та приніс вагомі зміни, серед яких вбудована підтримка async/await в обробниках запитів (тепер не потрібно вручну обгортати їх у try/catch), оновлений роутер та посилені налаштування безпеки за замовчуванням.

Альтернативи, що були розглянуті:

- Fastify, що швидший за Express та має вбудовану валідацію через JSON Schema, але поступається в розмірах спільноти та кількості готових middleware;

- Koa як наступник Express від тих самих авторів, повністю побудований на async/await, проте з меншою екосистемою та повільнішим розвитком;

- NestJS, повноцінний фреймворк з декораторами, dependency injection та модульною архітектурою, що виявляється надмірно складним для проєкту такого масштабу;

- Hono, новий ультрашвидкий фреймворк, що поки залишається занадто молодим і нішевим.

Вибір Express зумовлено ґрунтовною документацією, низьким порогом входу, гнучкістю у проєктуванні архітектури та можливістю підключати лише потрібні middleware. З освітнього погляду це також виграшний варіант, адже

практичний досвід з Express легко переноситься на більшість TypeScript-бекенд проєктів.

У ролі середовища виконання використано Node.js LTS, тобто поточну версію з довготривалою підтримкою, у якій присутні оптимізації V8, вбудований test-runner (хоча на серверній стороні проєкту обрано Jest), нативна обробка .env-файлів через прапорець --env-file та помітне зростання продуктивності фактично в усіх сценаріях.

У проєкті задіяно такий набір middleware:

- cors для налаштування політики CORS, що дозволяє приймати запити від клієнта, розміщеного на іншому домені;

- express.json для парсингу JSON-тіла запитів;

- pino-http для структурованого логування HTTP-запитів у форматі JSON;

- express-rate-limit для контролю частоти запитів від однієї IP-адреси;

- swagger-ui-express разом з swagger-jsdoc для автоматичної генерації OpenAPI-документації з JSDoc-коментарів та її подання у вебінтерфейсі.

2.5 ORM та база даних: Prisma 6 та PostgreSQL 15

Системою керування базами даних обрано PostgreSQL, реляційну СУБД з понад тридцятип'ятирічною історією, яка відома надійністю, високим рівнем відповідності стандарту SQL, підтримкою складних типів даних (JSON, JSONB, масивів та переліків) і безперервним розвитком. PostgreSQL використовується такими компаніями, як Apple, Spotify, Reddit та Instagram, і за рейтингом DB-Engines стабільно входить до трійки найпопулярніших СУБД у світі.

Альтернативи, що були розглянуті:

- MySQL, теж розповсюджена реляційна СУБД, проте PostgreSQL пропонує ширші можливості та кращу підтримку JSON;

- SQLite, що чудово підходить для прототипів, але не розрахована на продакшен-навантаження з кількома одночасними користувачами;

- MongoDB, документна NoSQL-база, що не є оптимальною для структурованих даних з вираженими зв'язками;

- Redis, призначений переважно для кешування і не для основного зберігання даних.

Як ORM використано Prisma, сучасний type-safe інструмент, що автоматично генерує TypeScript-клієнт на основі декларативної схеми. У порівнянні з традиційними ORM Prisma має кілька суттєвих переваг, а саме:

- стовідсоткова типобезпека, оскільки згенерований клієнт містить вичерпні типи для усіх моделей, зв'язків та запитів;

- декларативна схема, зосереджена в одному файлі, що значно полегшує її супровід та читання порівняно з розрізненими моделями;

- автоматичні міграції, коли команда prisma migrate створює SQL-міграції на основі різниці між версіями схеми;

- Prisma Studio як візуальний інспектор бази даних з можливістю безпосереднього редагування записів;

- відсутність класового ORM-підходу, завдяки чому процес тестування спрощено, бо клієнт легко замокати без створення складних об'єктів.

Серед альтернатив у сфері ORM розглядалися: TypeORM (складніший, з декораторами, нагадує Doctrine з PHP), Sequelize (старіший за віком та без типобезпеки за замовчуванням) і Drizzle (новий та дуже швидкий, але ще недостатньо зрілий).

Схема бази даних PRISM AI описує дві основні моделі: User (користувач) та Project (репозиторій), які пов'язані зв'язком «один-до-багатьох». Детальніше опис схеми наведено у розділі 3.2.

2.6 CSS-фреймворк Bootstrap 5

Для стилізації клієнтської частини обрано Bootstrap, найпопулярніший у світі CSS-фреймворк, що був створений командою Twitter та продовжує розвиватися завдяки відкритій спільноті. У його склад входить дванадцятиколоночна grid-система, побудована на CSS Flexbox, набір готових компонентів (кнопки, форми, навігація, модальні вікна, табуляції), утилітарні класи для відступів, кольорів та шрифтів, темна і світла теми, які перемикаються через атрибут data-bs-theme, а також JavaScript-плагіни для інтерактивних елементів.

Серед альтернатив можна виділити Tailwind CSS (підхід utility-first, дуже гнучкий, проте вимагає писати більше коду для типових компонентів), Material UI (готовий набір компонентів у стилі Material Design з суттєво більшим розміром бандла та жорстко закладеним стилем), Ant Design (поширений у сегменті enterprise UI, але важкий) та Chakra UI (з приємним DX, але менш зрілий).

На користь Bootstrap зважили такі чинники: висока швидкість розробки завдяки готовим компонентам, нативна підтримка темної теми (важлива для розробників), відсутність залежності від jQuery (на відміну від Bootstrap 4) та широка популярність серед розробників.

2.7 Інтеграція AI: OpenAI-сумісний протокол

Інтеграцію з ШІ-моделями побудовано на основі OpenAI-сумісного REST-протоколу, що став де-факто стандартом, який підтримують усі провідні постачальники LLM. Завдяки цьому користувач може обирати з кількох провайдерів без правок у коді, бо змінюється лише базова адреса (base URL) та API-ключ.

Підтримувані провайдери:

- OpenAI, оригінальний постачальник моделей серії GPT (gpt-4o, gpt-4o-mini, gpt-3.5-turbo), що забезпечує найвищу якість для більшості задач;

- Google Gemini, LLM від Google з конкурентоспроможною якістю та значно нижчою вартістю (наприклад, gemini-2.0-flash);

- DeepSeek, китайський провайдер, сфокусований на роботі з кодом і вирізняється дуже низькою вартістю (deepseek-chat, deepseek-coder);

- OpenRouter, агрегатор, що пропонує єдиний API до сотень моделей, включно з open-source-варіантами на кшталт Llama та Mistral;

- Custom, тобто будь-який OpenAI-сумісний ендпоінт, як-от локальна Ollama або корпоративне проксі.

Сам код інтеграції зі ШІ зосереджено у двох файлах. У utils/ai.ts розміщено низькорівневу функцію callLLM, яка надсилає HTTP-запит до /chat/completions та обробляє помилки 401, 403 і 429. У services/project-ai.service.ts знаходиться високорівнева логіка: тут формуються промпти, виконуються виклики LLM і відбувається парсинг відповідей. Тексти усіх промптів винесено в constants/project-ai.constants.ts, що спрощує їх подальше доопрацювання.

2.8 Контейнеризація: Docker Compose

Щоб гарантувати відтворюваність середовища розгортання, застосунок упаковано в контейнери засобами Docker та Docker Compose. Docker сьогодні є стандартом контейнеризації, що дає змогу пакувати застосунок разом із залежностями у переносний образ, а Docker Compose виступає інструментом для оркестрації багатоконтейнерних рішень через декларативний YAML-файл.

У проєкті визначено три сервіси:

- frontend на базі Nginx, що віддає статичні файли скомпільованого React-застосунку;

- backend на Node.js Alpine з зібраним TypeScript-кодом, який при старті виконує prisma migrate deploy і піднімає сервер;

- db у вигляді PostgreSQL, дані якого зберігаються в іменованому volume під назвою pgdata.

У Dockerfile фронтенду застосовано багатоступінчасте збирання: на першому етапі (node:18-alpine) Vite готує production-бандл, а на другому етапі (nginx:stable-alpine) до контейнера копіюються лише статичні файли у директорію Nginx. Завдяки такому розподілу підсумковий образ важить близько 25 МБ замість приблизно 400 МБ у разі прямолінійного підходу.

Як альтернативу можна було б розглянути Kubernetes, проте для проєкту бакалаврського рівня він виявляється надмірно складним. Docker Compose у цьому випадку є оптимальним рішенням, прийнятним як для локальної розробки, так і для малих чи середніх продакшен-середовищ.

2.9 Тестові інструменти

Тестування застосунку охоплює три рівні, причому для кожного з них дібрано спеціалізований інструмент.

Модульне тестування серверної частини: Jest є найбільш поширеним тестовим фреймворком у JS-екосистемі. До його сильних сторін належать вбудоване мокування, snapshot-тести, паралельний тест-ранер та зручні assertion-API. Інтеграційні тести API виконуються через Supertest, що піднімає Express-застосунок прямо в пам'яті без окремого HTTP-сервера. Prisma-клієнт мокається через jest.mock, щоб ізолювати тести від реальної бази даних.

Модульне тестування клієнтської частини: Vitest це швидкий тест-ранер на базі Vite з API, сумісним з Jest. У парі з Testing Library для React він дозволяє писати декларативні тести компонентів, орієнтовані на поведінку користувача. Vitest без додаткового налаштування працює з ESM-модулями та TypeScript.

Компонентне тестування: Storybook це інструмент для розробки UI-компонентів у ізольованому середовищі. Кожен компонент описується у файлі *.stories.tsx, де перелічуються різні його стани. Storybook test-runner

автоматично прогонить усі stories, перевіряючи відсутність помилок рендеру та виконуючи snapshot-тестування DOM. Завдяки цьому компоненти можна контролювати як візуально, так і автоматично, незалежно від того, до якої сторінки вони згодом будуть підключені.

Наскрізне тестування: Playwright виступає сучасною альтернативою Cypress та Selenium від компанії Microsoft. Він перевершує конкурентів за кількома параметрами, серед яких підтримка декількох браузерів (Chromium, Firefox, WebKit), вбудоване мокування мережі через page.route, покращена логіка очікувань (auto-waiting) та більша швидкість виконання. У проєкті наскрізні тести спираються на повне мокування відповідей API, що дозволяє перевіряти клієнтський сценарій без піднятого реального сервера.

Контроль якості коду автоматизовано засобами Husky та lint-staged. Husky відповідає за встановлення git-хуків.

Lint-staged запускає перевірки (ESLint, Prettier, типізація TypeScript) лише на тих файлах, які потрапили до staged-області, що суттєво пришвидшує виконання pre-commit хуків порівняно з прогоном по всьому проєкту. Така зв'язка гарантує, що до репозиторію не потрапляє код із синтаксичними помилками чи порушеннями стилю, а розробник отримує миттєвий зворотний зв'язок ще до створення комміту. Додатково на етапі pre-push можна налаштувати запуск повного набору модульних тестів, аби виключити потрапляння у віддалену гілку завідомо «зламаних» змін.

Окрім локальних перевірок, аналогічний набір лінтерів і тестів виконується у CI-середовищі (наприклад, GitHub Actions) під час відкриття pull request, що створює другий рівень захисту на випадок, якщо розробник свідомо чи випадково обійшов git-хуки за допомогою прапорця --no-verify.

Усі залежності фіксуються через package-lock.json, а версія Node.js закріплюється у файлі .nvmrc, що забезпечує відтворюваність середовища як на машинах розробників, так і на CI-раннерах.

3 РЕАЛІЗАЦІЯ WEB-ЗАСТОСУНКУ

3.1 Проєктування архітектури

Архітектуру веб застосунку PRISM AI побудовано за класичною тришаровою клієнт-серверною моделлю з чітким розмежуванням відповідальності між шарами та продуманими інтерфейсами обміну даними. Загальну схему архітектури представлено на рисунку 3.1.

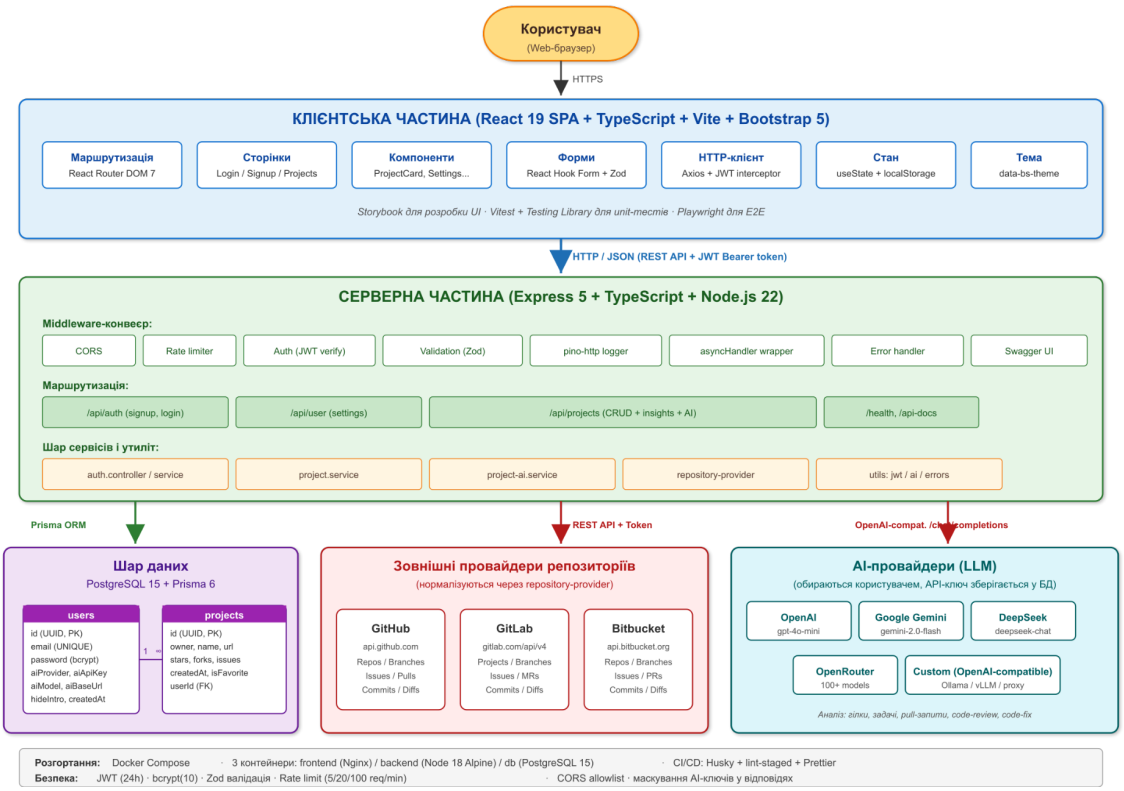


Рис. 3.1 Архітектура web-застосунку PRISM AI

Архітектура застосунку реалізована за класичною тришаровою моделлю, що історично довела свою ефективність для веб-систем і дозволяє чітко розмежувати зони відповідальності між компонентами. Такий поділ полегшує паралельну розробку, спрощує тестування кожного шару окремо та робить можливою заміну будь-якого з рівнів без впливу на інші, наприклад, у майбутньому фронтенд може бути переписаний на іншому фреймворку, а

серверна частина мігрована на інший runtime, без необхідності переробляти всю систему.

Перший шар, клієнтська частина (React SPA), реалізована як односторінковий застосунок із маршрутизацією на стороні браузера. Користувач спілкується із системою тільки через цей шар у вигляді веб інтерфейсу. На клієнті немає бізнес-логіки безпеки, бо він лише прикріплює JWT-токен до запитів та показує дані, отримані від сервера. Валідація у браузері існує переважно як зручність для користувача, а не як межа безпеки.

Другий шар це серверна частина у вигляді REST API на Express, де зосереджено всю бізнес-логіку, керування доступом до даних, валідацію та інтеграцію зі сторонніми сервісами. Сама серверна частина поділяється на чотири підшари:

- маршрутизація (routes/), де описуються HTTP-ендпоінти та middleware-стек для кожного з них;

- контролери (controllers/), що приймають запити, передають їх сервісам та формують відповіді клієнту;

- сервіси (services/), у яких міститься бізнес-логіка та виконується агрегація даних з кількох джерел;

- утиліти (utils/), куди винесено низькорівневі функції роботи з API провайдерів, JWT та обробку помилок.

Третій шар це рівень зберігання даних та зовнішніх інтеграцій. Він охоплює базу PostgreSQL, у якій постійно зберігаються користувачі та їхні проєкти, а також зовнішні REST API GitHub, GitLab і Bitbucket для отримання метаданих репозиторіїв та LLM-провайдерів для ШІ-аналізу.

Окремо варто розглянути шлях проходження запиту крізь middleware-стек. Типовий захищений запит послідовно проходить наступні етапи: CORS, JSON parser, HTTP logger, rate limiter, autentifikatsiya (перевірка JWT), валідація через Zod-схему, контролер у обгортці asyncHandler, виклик сервісу, утиліти та Prisma, формування відповіді та, у разі помилки, передача керування глобальному error handler.

Така організація дає кілька переваг. По-перше, чітке розмежування відповідальності полегшує тестування, бо кожен шар можна перевіряти ізольовано через моки. По-друге, з'являється можливість для масштабування, оскільки за потреби окремі сервіси легко винести у самостійні мікросервіси. По-третє, посилюється безпека, бо кожен запит проходить стандартизований перевірочний пайплайн. По-четверте, спрощується діагностика завдяки структурованим логам на кожному кроці.

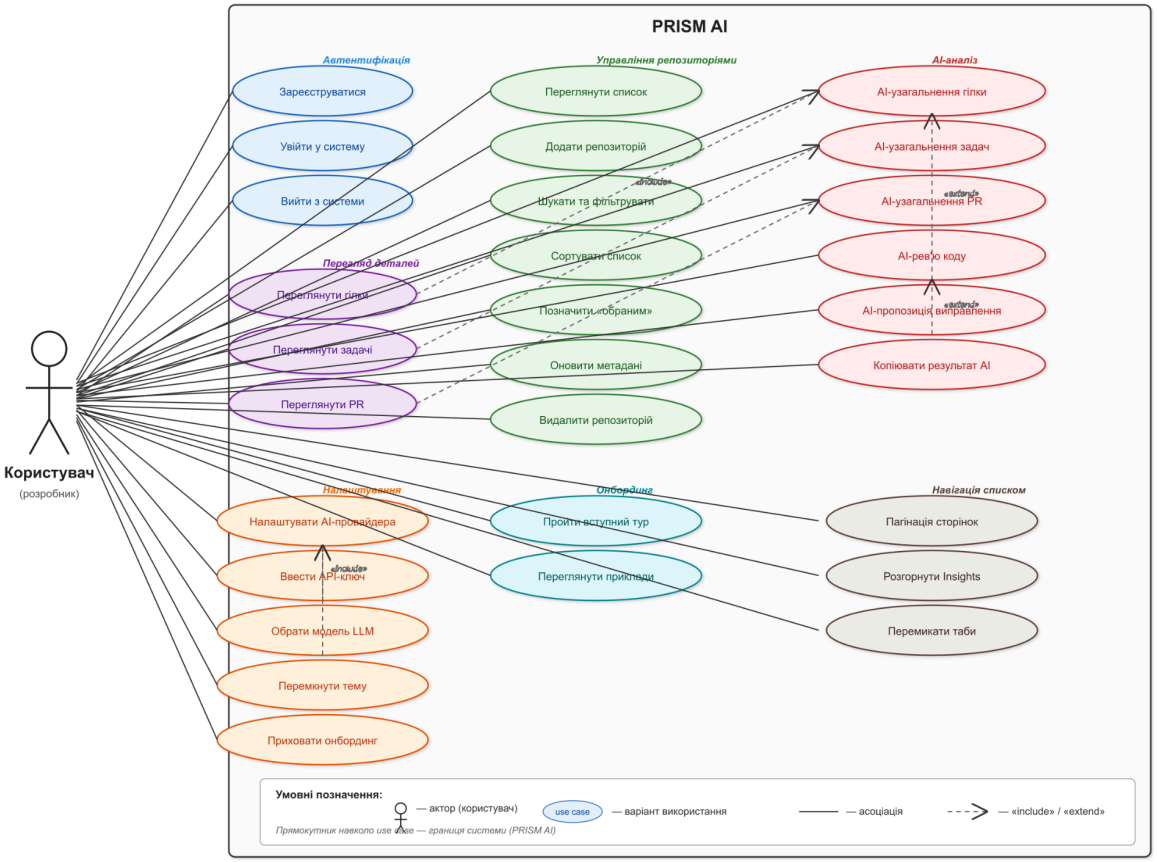


Рис. 3.2 Діаграма варіантів використання для користувача

На діаграмі варіантів використання (рисунок 3.2) представлено основні сценарії взаємодії користувача із системою, серед яких реєстрація та автентифікація, перегляд списку репозиторіїв, додавання нового сховища, перегляд деталей конкретного репозиторію (гілки, задачі, запити на злиття), запит на ШІ-аналіз різних типів, редагування параметрів ШІ-провайдера, керування обраними репозиторіями та їхнє видалення.

3.2 Схема бази даних

Базу даних застосунку спроєктовано з мінімальною нормалізацією, тобто містить лише ті таблиці, які справді потрібні для роботи системи. Метаінформація про сховища (списки коммітів, задач, запитів на злиття) не дублюється у локальному сховищі, а отримується «на льоту» з API провайдерів. Локально зберігаються лише ті дані, що безпосередньо належать користувачам, а саме облікові записи та довідник доданих ними репозиторіїв.

Опис схеми бази даних записано мовою Prisma Schema Language у файлі server/prisma/schema.prisma. У ньому визначено дві моделі (схему бази даних показано на рисунку 3.3).

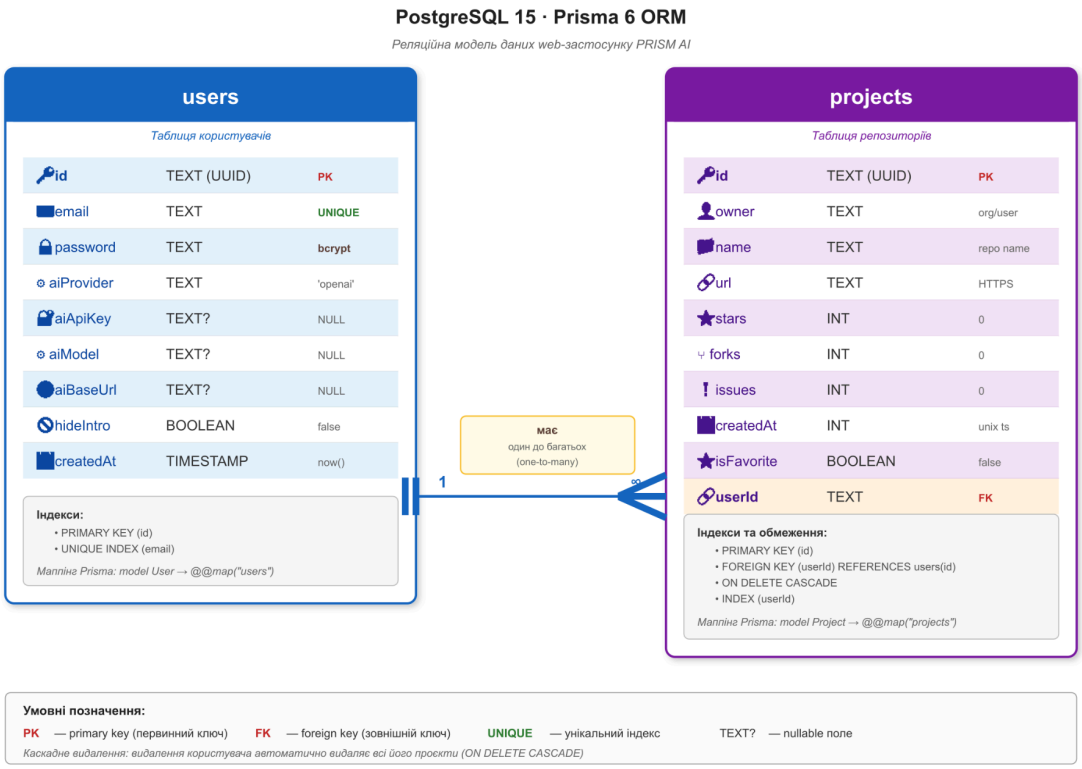


Рис. 3.3 Схема бази даних PRISM AI

Таблиця 3.1

Структура таблиці users

Поле	Тип	Опис
id	TEXT (UUID)	Унікальний ідентифікатор користувача
email	TEXT (UNIQUE)	Електронна пошта користувача
password	TEXT	Хеш паролю (bcrypt, 10 раундів)
aiProvider	TEXT	Обраний AI-провайдер (за замовч. openai)
aiApiKey	TEXT NULL	API-ключ AI-провайдера (опційно)
aiModel	TEXT NULL	Кастомна модель LLM (опційно)
aiBaseUrl	TEXT NULL	Кастомний base URL (опційно)
hideIntro	BOOLEAN	Чи приховано вступну анімацію
createdAt	TIMESTAMP	Дата реєстрації користувача

Таблиця 3.2

Структура таблиці projects

Поле	Тип	Опис
id	TEXT (UUID)	Унікальний ідентифікатор
owner	TEXT	Власник репозиторію (organization або user)
name	TEXT	Назва репозиторію
url	TEXT	Повний HTML-URL репозиторію
stars	INT	Кількість зірок (snapshot)
forks	INT	Кількість форків (snapshot)
issues	INT	Кількість відкритих задач (snapshot)
createdAt	INT	Дата створення репозиторію (Unix timestamp)
isFavorite	BOOLEAN	Маркер «обраний»
userId	TEXT (FK)	Зовнішній ключ на users.id

Зв'язок між User та Project має тип «один-до-багатьох», тобто один користувач може володіти багатьма проєктами, проте кожен проєкт належить одному користувачу. У базі це втілено через зовнішній ключ projects.userId з посиланням на users.id та опцією onDelete: Cascade. Завдяки цьому при видаленні користувача автоматично знищуються й усі його проєкти.

Поля aiApiKey, aiModel та aiBaseUrl наразі зберігаються у відкритому вигляді, що є свідомим компромісом між простотою реалізації та рівнем безпеки. У промисловому середовищі доцільно додати шифрування цих полів симетричним ключем AES-256, що зберігається в environment variables. При цьому у відповіді на GET /api/user/settings ключ повертається у замаскованому вигляді: відкриті лише останні чотири символи, а решта замінюється зірочками, що знижує ризик випадкового витоку через вкладку Network у DevTools.

Поле createdAt у проєктах зберігається у вигляді Unix timestamp (INT), що спрощує сортування на стороні бази без додаткових перетворень. Поля stars, forks та issues є snapshot-метриками, які оновлюються або при додаванні проєкту, або через ручне оновлення засобами PATCH /api/projects/:id/update.

3.3 REST API

REST API застосунку реалізовано через 19 ендпоінтів, які охоплюють усі функціональні вимоги. Усі вони, за винятком авторизаційних, вимагають передачі JWT-токена в заголовку Authorization: Bearer <token>. Архітектура API дотримується принципів REST: HTTP-методи відповідають CRUD-операціям, окремі ендпоінти ідентифікують ресурси, а статусні коди описують типи помилок.

Окремі ендпоінти ШІ-аналізу (шлях /ai/*) використовують метод POST, бо не є ідемпотентними та можуть споживати значні ресурси LLM-провайдера. Метод POST також дає можливість передавати параметри (наприклад, назву гілки) у тілі запиту, а не в URL.

Таблиця 3.3

Список усіх REST API ендпоінтів PRISM AI

Метод	Шлях	Призначення
POST	/api/auth/signup	Реєстрація нового користувача
POST	/api/auth/login	Автентифікація, повертає JWT-токен
GET	/api/user/settings	Отримати налаштування поточного користувача
PUT	/api/user/settings	Оновити налаштування
GET	/api/projects	Отримати список проєктів з фільтрами
POST	/api/projects	Додати новий проєкт
GET	/api/projects/:id/details	Деталі проєкту: гілки, задачі, PR
GET	/api/projects/:id/branches	Список гілок проєкту
GET	/api/projects/:id/issues	Список задач проєкту
GET	/api/projects/:id/pulls	Список pull-запитів проєкту
PATCH	/api/projects/:id/favorite	Перемкнути «обраний»
PATCH	/api/projects/:id/update	Оновити метадані проєкту
DELETE	/api/projects/:id	Видалити проєкт
POST	/api/projects/:id/ai/branch-summary	AI-узагальнення змін у гілці
POST	/api/projects/:id/ai/issues-summary	AI-узагальнення задач
POST	/api/projects/:id/ai/pulls-summary	AI-узагальнення pull-запитів
POST	/api/projects/:id/ai/code-review	AI-рев'ю коду гілки
POST	/api/projects/:id/ai/code-fix	AI-пропозиція виправлення фрагменту
GET	/health	Health-check (для моніторингу)

Усі ендпоінти задокументовані через JSDoc-коментарі з тегами @swagger. Бібліотека swagger-jsdoc автоматично обходить усі route-файли та формує специфікацію OpenAPI 3.0, доступну через Swagger UI за адресою http://localhost:5001/api-docs. Розробники та QA-інженери можуть інтерактивно перевіряти API без потреби у додаткових інструментах. Зовнішній вигляд Swagger UI з документацією REST API наведено на рисунку 3.4.

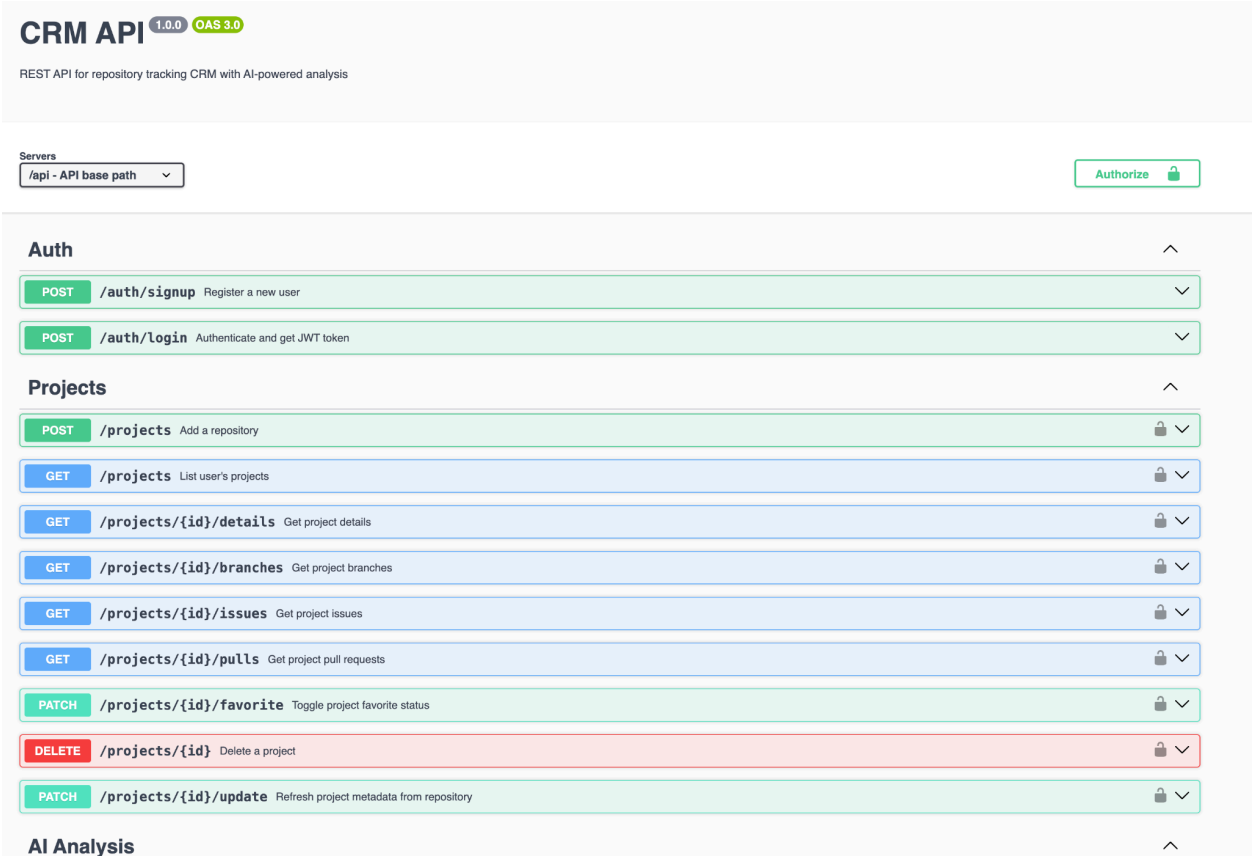


Рис. 3.4 Swagger UI з документацією REST API

3.4 Серверна частина

3.4.1 Шар маршрутизації та middleware

Маршрутизація на сервері побудована на express.Router з груповими префіксами, що дає змогу чітко розмежовувати зони відповідальності та підключати різні набори middleware до різних наборів ендпоінтів. У файлі app.ts маршрути підключаються так: app.use('/api/auth', authLimiter, authRoutes), app.use('/api/user', generalLimiter, userRoutes), app.use('/api/projects', generalLimiter, projectRoutes).

Кожна група маршрутів має власний rate-limiter, бо різні типи операцій мають різну вартість і різний характер потенційних зловживань. Для авторизаційних ендпоінтів (signup та login) встановлено найжорсткіший ліміт у п'ять запитів на хвилину, оскільки вони є основною ціллю brute-force атак.

ШІ-ендпоінти обмежені двадцятьма запитами на хвилину через дорогі виклики LLM. Загальні API можуть приймати до ста запитів на хвилину, що цілком достатньо звичайному користувачу, проте стримує автоматизовані сценарії використання.

Middleware автентифікації працює так: бере заголовок Authorization, перевіряє його відповідність формату «Bearer <token>», валідує JWT за допомогою jsonwebtoken.verify, дістає userId з payload та прикріплює його до req.userId. Якщо токена немає або він невалідний, middleware кидає AppError зі статусом 401, який перехоплює глобальний error-handler та перетворює на JSON-відповідь.

Middleware валідації приймає Zod-схему та автоматично перевіряє за нею req.params, req.query та req.body. Якщо валідація не пройшла, middleware готує розгорнуте повідомлення про помилки із зазначенням конкретного поля та причини й повертає статус 400 Bad Request.

Особливо корисною утилітою є asyncHandler, обгортка для асинхронних контролерів, яка автоматично передає помилки в error-handler. Без неї довелося б у кожному контролері писати власні try/catch блоки та вручну викликати next(err).

3.4.2 Абстракція провайдерів репозиторіїв

Найскладнішою складовою серверної частини є шар абстракції над провайдерами репозиторіїв, реалізований у файлі server/src/utils/repository-provider.ts. Його завдання сховати специфіку API трьох різних платформ хостингу за єдиним інтерфейсом, який бачить решта системи.

До переліку конкретних відмінностей, з якими має справу ця абстракція, належать:

- різні формати URL, а саме github.com, gitlab.com та bitbucket.org;

- різні підходи до автентифікації, коли публічні API GitHub доступні без ключа, а приватні репозиторії потребують окремих токенів;

- різні моделі сутностей, наприклад pull request у GitHub та Bitbucket проти merge request у GitLab;

- різні стилі пагінації, зокрема Link header у GitHub, параметри page та per_page у GitLab, а також start і limit у Bitbucket;

- неоднакова структура відповідей API, де поля коммітів, задач і гілок мають різні назви та формати представлення дат.

Усі ці відмінності зводяться до спільних TypeScript-інтерфейсів: UnifiedRepository, UnifiedBranch, UnifiedIssue, UnifiedPullRequest та UnifiedCommit. Решта системи оперує виключно цими типами та не залежить від конкретного провайдера.

Розбір вхідного рядка з посиланням на репозиторій виконує функція parseRepositoryInput, що підтримує кілька форматів: «owner/repo» (за замовчуванням GitHub), «gitlab:group/subgroup/repo», «bitbucket:workspace/repo» або повний URL за HTTPS. Завдяки цьому користувач має гнучкість при додаванні проєкту через AddProjectButton.

Окремий клієнт реалізовано для GitHub у файлі utils/github.ts. Особливість GitHub API полягає в тому, що список гілок повертає лише мінімум інформації, тобто назву та SHA верхнього коміту, без даних про автора та повідомлення коміту. Для збагачення цих даних потрібно робити окремий запит на кожну гілку, що в типовому репозиторії може означати 30-50 запитів. Щоб не виходити за межі rate-limit GitHub (60 запитів на годину для неавтентифікованих та 5000 для авторизованих), запити виконуються пакетами по вісім одночасно через Promise.all з обмеженням, після чого витримується пауза.

3.4.3 AI-сервіс

Сервіс ШІ-аналізу поділено на два рівні. На низькому рівні працює утилітна функція callLLM у файлі utils/ai.ts, що виконує HTTP POST до OpenAI-сумісного ендпоінту /chat/completions. Вона приймає baseUrl, apiKey, model та messages (масив повідомлень з ролями system та user) і повертає рядок із відповіддю LLM. Окрім того, функція обробляє типові помилки 401

(неавторизовано), 403 (заборонено) та 429 (rate limit), перетворюючи їх на повідомлення, зрозумілі для користувача.

На високому рівні працює сервіс project-ai.service.ts, у якому міститься п'ять функцій-аналізаторів. Кожна з них послідовно виконує такі кроки:

- зчитує налаштування ШІ-провайдера з таблиці users для поточного userId;

- визначає baseUrl, apiKey та model з урахуванням обраного провайдера та користувацьких перевизначень;

- звертається до відповідного repository-provider для отримання потрібних даних з API хостингу;

- формує текстовий промпт зі специфічним system-повідомленням та user-повідомленням, що містить отримані дані;

– викликає callLLM та отримує відповідь;

- для окремих типів аналізу (code-review, code-fix) парсить відповідь у JSON та перевіряє її структуру;

– повертає DTO з результатом аналізу.

Окремої уваги заслуговує підхід до парсингу JSON відповідей у функціях code review та code fix. Оскільки LLM-моделі схильні «обгортати» JSON у Markdown-блоки на кшталт json…, реалізовано двоетапну стратегію: спочатку здійснюється спроба прямого парсингу через JSON.parse, а у разі невдачі виконується видалення обгортки та повторна спроба. Якщо обидві спроби виявляються невдалими, сервіс повертає структуровану помилку із зазначенням, що модель не дотрималася формату, що дає змогу інтерфейсу запропонувати користувачу повторити запит або змінити модель.

Самі промпти зберігаються у constants/project-ai.constants.ts. Цей підхід має кілька плюсів: легко налаштовувати промпти без втручання в логіку, є можливість проводити A/B тестування та закладено базу для локалізації на українську мову у майбутньому. Усі промпти заздалегідь містять інструкцію відповідати українською, що відповідає мовним потребам цільової аудиторії застосунку. Послідовність викликів під час ШІ-аналізу гілки наведено на рисунку 3.5.

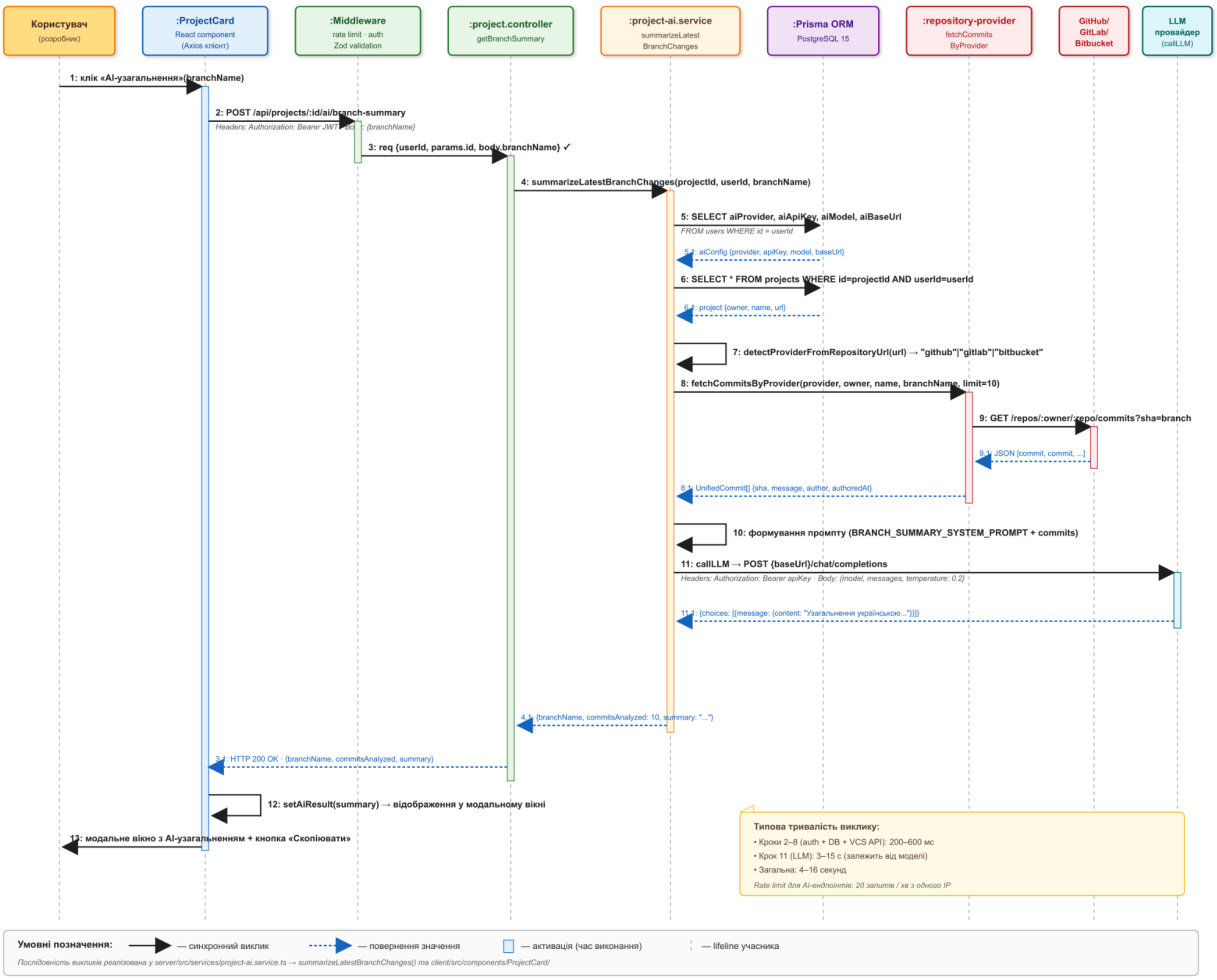


Рис. 3.5 Послідовність викликів під час ШІ-аналізу гілки

3.5 Клієнтська частина

3.5.1 Маршрутизація та автентифікація

Клієнтська частина побудована як SPA (Single Page Application) з маршрутизацією на боці браузера через React Router DOM. У файлі App.tsx оголошено чотири маршрути: «/» (редирект на «/projects» для авторизованого користувача та на «/login» у протилежному випадку), «/login», «/signup» і захищений маршрут «/projects». Catch-all-маршрут «*» перенаправляє на головну сторінку.

Стан автентифікації керується через React useState на рівні компонента App. Початкове значення зчитується з localStorage за ключем «token». Після

успішного входу токен зберігається у localStorage, а під час виходу видаляється з нього. Такий простий підхід виявляється цілком достатнім для застосунку подібного масштабу, тоді як введення глобального state-менеджера на кшталт Redux чи Zustand було б надмірним.

Перехоплювач Axios у файлі services/api.ts автоматично додає JWT-токен до кожного запиту, дістаючи його з localStorage. Якщо сервер повертає 401 (некоректний або застарілий токен), перехоплювач очищає токен у localStorage і виконує перенаправлення на /login. Це централізована обробка протермінованих сесій, що позбавляє від дублювання коду у компонентах.

3.5.2 Компонентна архітектура

Компоненти клієнтської частини організовано за патерном barrel exports: кожен компонент лежить в окремій директорії з файлом index.ts, який його ре-експортує. Завдяки цьому компоненти можна імпортувати зручним шляхом через @/components без зазначення конкретного файлу, а також спрощується їх перейменування або переміщення.

Список основних компонентів з їх відповідальністю:

- AddProjectButton, модальне вікно для додавання репозиторію, що підтримує усі формати посилань;
- AppErrorBoundary, React Error Boundary, що ловить помилки у дереві компонентів та показує UI відновлення;
- DeleteProjectButton, компонент підтвердження видалення проєкту;
- IntroGuide, багатокроковий онбординг для нових користувачів;
- LogoutButton, що очищає токен та виконує перенаправлення;
- ProjectCard, найскладніший компонент, що відображає метрики репозиторію та розкривну панель інсайтів;
- SettingsButton, модальне вікно для конфігурації ШІ-провайдера;
- UpdateProjectButton, що оновлює метадані проєкту з API хостингу.

Кожному компоненту відповідає окремий файл Storybook stories, у якому перелічені різні його стани (default, loading, error, mocked-data тощо). Це дозволяє розробникам та дизайнерам працювати з компонентом ізольовано та автоматично перевіряти весь набір візуальних станів.

3.5.3 Головна сторінка проєктів

Сторінка Projects.tsx є головним екраном застосунку після авторизації. Вона відображає список доданих репозиторіїв з такими можливостями: пошук за назвою з debounce у 300 мс для зменшення кількості зайвих API-запитів, фільтр за платформою (GitHub, GitLab, Bitbucket або All), сортування за датою, кількістю зірок, форків та задач в обох напрямках, окремий фільтр обраних та пагінація по три проєкти на сторінку.

Новим користувачам, у яких ще не додано жодного проєкту, виводиться IntroGuide, тобто покроковий wizard, що знайомить з функціоналом застосунку та допомагає додати перший репозиторій. IntroGuide можна вимкнути за допомогою чекбокса «Не показувати знову», стан якого зберігається у налаштуваннях через PUT /api/user/settings.

Сторінку реалізовано з акцентом на UX: для кожного стану (loading, empty, error, populated) передбачено окреме візуальне представлення, помилки API повідомляються через toast-нотифікації, а пагінація реалізована за допомогою елемента з номерами сторінок та стрілками «Вперед» і «Назад».

3.5.4 Панель інсайтів та AI-аналізу

Найбільшим за обсягом компонентом є ProjectCard, який налічує близько 1100 рядків коду. Окрім базової інформації про репозиторій, він містить розкривну панель «Insights» з чотирма вкладками: Branches (гілки), Issues (задачі), Pull Requests (запити на злиття) та Code Review (ШІ-рев'ю).

Вкладка Branches показує перелік гілок репозиторію, впорядкованих за датою останнього коміту. Для кожної гілки доступна кнопка «AI-узагальнення», яка відкриває модальне вікно з результатом запиту POST /api/projects/:id/ai/branch-summary. У тому ж вікні розміщено кнопку «Code Review», що запускає ШІ-рев'ю коду цієї гілки.

Вкладка Issues відображає перелік задач з кольоровими індикаторами статусу (open та closed). Над списком розміщено кнопку «AI-узагальнення задач», яка показує згенероване LLM-зведення про теми та пріоритети активних задач.

Вкладка Pull Requests виводить список запитів на злиття з аналогічним ШІ-узагальненням. Для кожного запиту відображено автора, дату відкриття, кількість коммітів та заголовок.

Вкладка Code Review подає результати останнього ШІ-рев'ю коду у вигляді переліку знахідок (findings). Кожна знахідка містить тип (bug, performance, security, style), рівень серйозності (high, medium, low), посилання на файл та фрагмент коду, опис самої проблеми та посилання «Запитати виправлення», яке запускає POST /api/projects/:id/ai/code-fix для генерації покращеного коду разом з поясненням.

Кожен ШІ-результат супроводжується кнопкою «Скопіювати в буфер», що задіює Navigator.clipboard API для копіювання тексту відповіді. Це зручно при перенесенні згенерованих текстів у Slack, коментарі GitHub або документацію. Загальний вигляд панелі інсайтів проєкту з AI-аналізом представлено на рисунку 3.6.

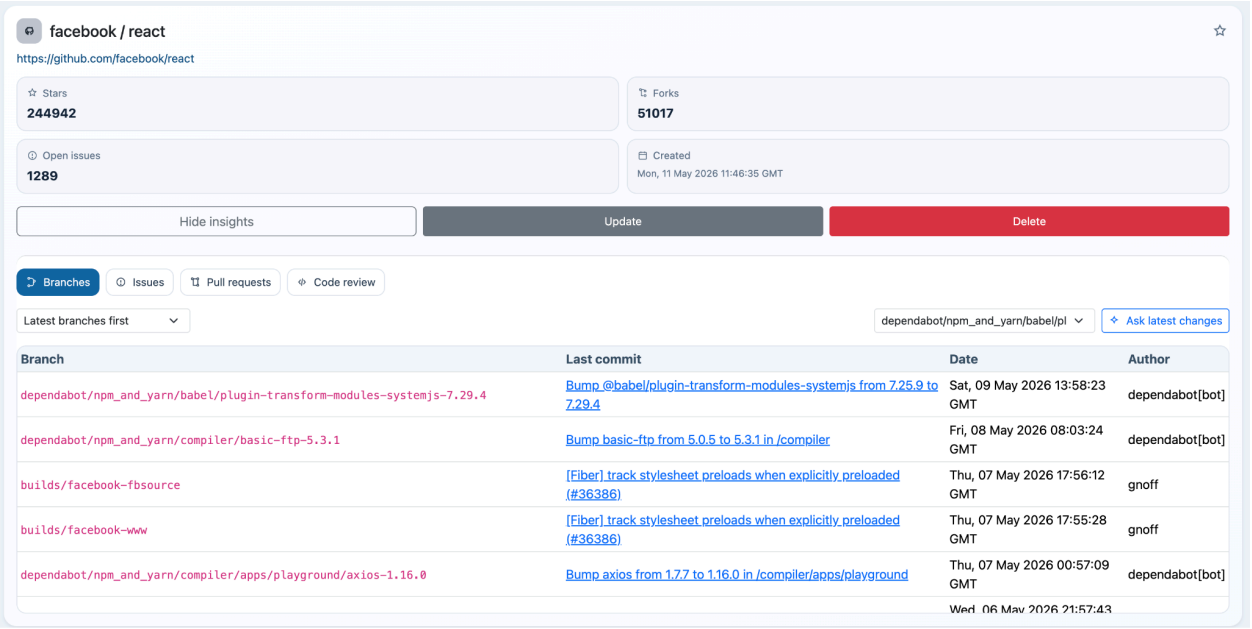


Рис. 3.6 Панель інсайтів проєкту з AI-аналізом

3.6 Контейнеризація та розгортання

Розгортання застосунку виконується через Docker Compose з трисервісною конфігурацією. У файлі docker-compose.yml описані сервіси frontend, backend та db з відповідними залежностями та портами.

Сервіс frontend збирається з Dockerfile, який реалізує багатоступінчасте збирання. На першому етапі (node:18-alpine) виконуються npm ci та npm run build, на другому етапі (nginx:stable-alpine) до контейнера копіюється лише статика з /dist у /usr/share/nginx/html. Налаштування Nginx у файлі client/nginx.conf забезпечує SPA-маршрутизацію через директиву try_files $uri $uri/ /index.html, що гарантує повернення index.html для будь-якого URL, а далі вже React Router розв'язує конкретний маршрут.

Сервіс backend збирається з server/Dockerfile, у якому лише одна стадія node:18-alpine. Команда entry-point спочатку запускає npx prisma migrate deploy, а потім node dist/index.js, тож схема бази завжди оновлюється до актуального стану перед стартом сервера. Це особливо важливо під час оновлення застосунку у продакшен-середовищі.

Сервіс db використовує офіційний образ postgres:15. Дані зберігаються в іменованому volume pgdata, що забезпечує їх збереження між перезапусками контейнерів. Підключення між backend і db виконується через внутрішню Docker-мережу за іменем сервісу (DATABASE_URL=postgresql://user:pass@db:5432/prism).

Усі чутливі параметри (JWT-секрет, паролі до бази даних, ШІ-ключі за замовчуванням) винесено у файл .env, шаблон якого .env.example версіонується разом з кодом. Сам файл .env додано до .gitignore та .dockerignore, щоб він гарантовано не потрапляв у репозиторій або у Docker-образ.

4 ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РЕЗУЛЬТАТІВ

Тестування веб застосунку PRISM AI виконано на трьох рівнях, що відповідають класичній піраміді тестування: модульний рівень становить найбільшу частку за кількістю тестів, компонентний посідає середню позицію, а наскрізний рівень містить найменшу кількість тестів, проте є найдорожчим з точки зору виконання. Такий розподіл дає оптимальний баланс між покриттям функціоналу та часом прогону всього тестового набору.

4.1 Модульне тестування серверної частини

Серверну частину покрито тринадцятьма тестовими файлами у директорії server/src/__tests__/, які впорядковано за типами тестованих сутностей: контролери, сервіси, middleware, утиліти та інтеграційні тести роутерів. У сумі тестовий пакет нараховує понад 200 тест-кейсів. Запуск здійснюється командою npm test.

Інтеграційні тести спираються на Supertest, що імітує HTTP-запити до Express-застосунку. Завдяки цьому можна перевірити повний ланцюжок обробки запиту, тобто middleware, контролер, сервіс і відповідь, без необхідності піднімати справжній HTTP-сервер. Prisma-клієнт мокається на рівні модуля через jest.mock('../prisma'), що повністю ізолює тести від реальної бази даних.

Окрему увагу приділено перевіркам, пов'язаним з безпекою: контролюється відсутність витоку чутливих даних у відповідях (наприклад, паролі ніколи не повертаються у відповіді на /api/auth/signup), коректність роботи JWT-валідації (протерміновані токени відхиляються зі статусом 401) та застосування bcrypt при збереженні паролів. Тести також підтверджують,

що навіть з валідним токеном користувач не отримає доступу до проєктів іншого користувача, оскільки сервіси завжди звіряють userId.

Таблиця 4.1

Приклади тест-кейсів серверної частини

№	Назва тесту	Опис	Очікуваний результат
1	Signup, успіх	POST /signup з валідним email і паролем	201 + {id, email, createdAt}
2	Signup, дублікат	POST /signup з email, що вже існує	409 Conflict
3	Signup, невалідний email	POST /signup з email без @	400 + поле email
4	Login, успіх	POST /login з валідними даними	200 + {token}
5	Login, невірний пароль	POST /login з неправильним паролем	401 Unauthorized
6	Get projects, без токена	GET /projects без Authorization	401 Unauthorized
7	Add project, github URL	POST /projects з github.com URL	201 + проєкт
8	Add project, gitlab	POST /projects з gitlab:org/repo	201 + проєкт
9	Add project, дублікат	POST /projects з вже доданим репо	409 Conflict
10	Delete, чужий проєкт	DELETE /projects/:id іншого користувача	404 Not Found
11	AI summary, без ключа	POST /ai/branch-summary без AI-ключа	400 + повідомлення
12	Rate limit, login	6+ запитів за хвилину на /login	429 Too Many Requests

Під час запуску всі тести проходять успішно. Покриття серверного коду модульними тестами сягає приблизно 80 відсотків, що відповідає вимогам бакалаврського проєкту. Звіт про покриття створюється командою npm run test:coverage та зберігається у директорії coverage/. Результат виконання модульних тестів серверної частини наведено на рисунку 4.1.

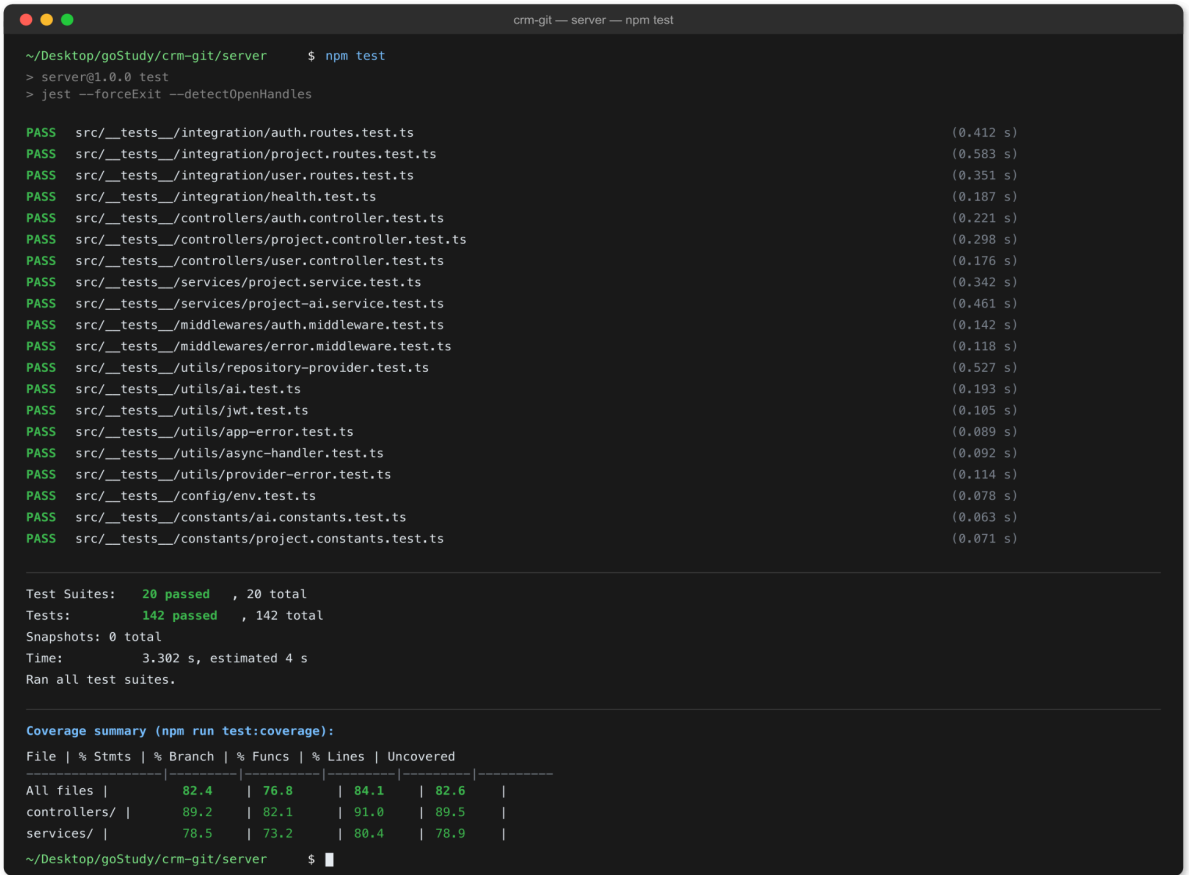


Рис. 4.1 Результат виконання модульних тестів серверної частини

4.2 Модульне тестування клієнтської частини

Клієнтську частину покрито тестами, що зберігаються у директорії client/src/__tests__/. Для запуску використовується Vitest у середовищі jsdom (емулятор DOM для Node.js) у поєднанні з React Testing Library, яка надає декларативні API для пошуку елементів у DOM та імітації взаємодії користувача.

Філософія Testing Library полягає в тестуванні компонентів у такий спосіб, у який з ними взаємодіє реальний користувач. Це означає, що елементи шукаються за роллю, текстом або лейблом, а не за CSS-селекторами чи id. Такий підхід робить тести стійкими до рефакторингу, бо якщо змінити реалізацію компонента (наприклад, замінити div на section), але зберегти UX, тести й далі будуть успішно проходити.

Покриття клієнтської частини включає такі сценарії: автентифікаційні форми Login і Signup (перевірка валідації, надсилання запиту та обробки помилок), AppErrorBoundary (відображення UI відновлення при викиданні помилки у дочірньому компоненті) і компонент App (перевірка маршрутизації та редиректів залежно від стану авторизації).

4.3 Компонентне тестування (Storybook)

Storybook у проєкті виконує подвійну роль: він є середовищем розробки UI-компонентів та водночас рівнем компонентного тестування. Кожен компонент описано у файлі *.stories.tsx з кількома історіями (stories), які відображають різні стани цього компонента.

Storybook test-runner у фоновому режимі відкриває кожну історію в headless Chromium та виконує дві групи перевірок. Перша це контроль відсутності помилок під час рендеру, серед яких помилки React та попередження про невказані props. Друга це snapshot-тестування, коли DOM-структура компонента звіряється з раніше збереженим snapshot. Будь-яка зміна автоматично призводить до невдачі тесту, тож розробник змушений свідомо оновити snapshot.

Завдяки цьому підходу регресії виявляються швидко: якщо випадково додати зайвий клас, помилково змінити структуру JSX чи забути про props, тести одразу провалюються. У багатьох сценаріях автоматичні snapshot-тести замінюють ручне візуальне тестування, що особливо цінно під час рефакторингу спільних компонентів, які використовуються у кількох десятках місць застосунку повне ручне перевикористання було б надто трудомістким, тоді як автоматизована перевірка займає лічені секунди. Інтерфейс Storybook з компонентами PRISM AI показано на рисунку 4.2.

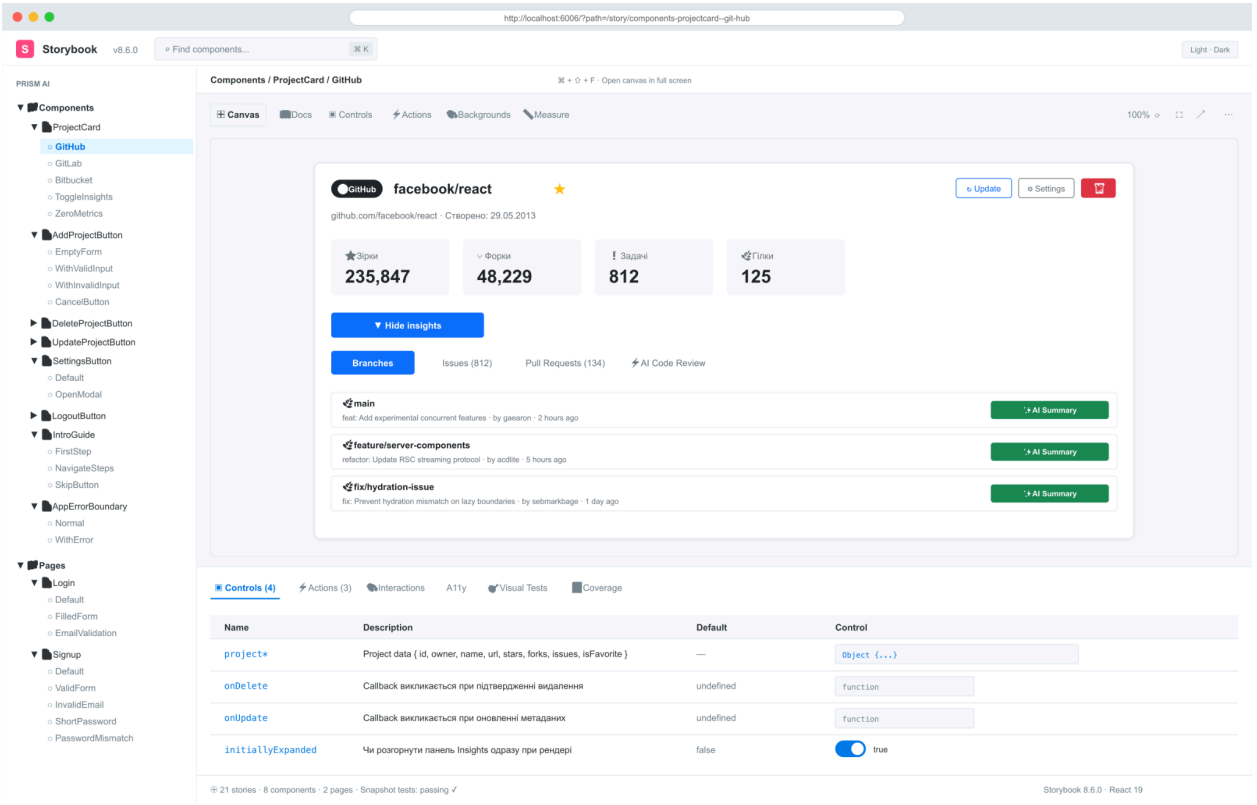


Рис. 4.2 Інтерфейс Storybook з компонентами PRISM AI

4.4 Наскрізне тестування (Playwright)

Наскрізне (E2E) тестування реалізовано через Playwright у директорії e2e/. Ці тести піднімають справжній браузер Chromium та повністю імітують дії користувача, включно з відкриттям сторінки, заповненням форм, кліками та перевіркою отриманих результатів. На відміну від модульних, E2E-тести працюють одночасно з усім стеком: клієнтом, сервером та базою даних.

Втім, для пришвидшення та незалежності від справжнього backend-сервера тести використовують мокання мережі через page.route. Допоміжний модуль e2e/helpers/mock-api.ts містить функції mockAuthAPI, mockProjectsAPI, mockUserAPI та mockAllAPIs, які налаштовують перехоплення відповідних API-запитів і повертають заздалегідь визначені відповіді. У такий спосіб можна перевіряти поведінку клієнта без необхідності піднімати реальний сервер чи базу даних.

Таблиця 4.2

Сценарії наскрізного тестування

№	Сценарій	Кроки
1	Реєстрація нового користувача	Відкриття /signup, введення email і пароля, submit, перехід на /projects
2	Невалідна реєстрація	Відкриття /signup, порожній email, submit, відображення помилки
3	Логін успішний	Відкриття /login, введення правильних даних, submit, перехід на /projects
4	Логін з невірним паролем	Відкриття /login, невірний пароль, submit, поява toast із повідомленням про помилку
5	Logout	Авторизований користувач, клік на LogoutButton, перехід на /login
6	Додавання проєкту	Сторінка /projects, клік на AddProject, введення URL, submit, поява проєкту в списку
7	Пошук проєкту	Сторінка /projects з кількома проєктами, введення тексту у поле пошуку та подальша фільтрація
8	Перемикач теми	Клік на ThemeSwitcher та перевірка зміни атрибута data-bs-theme
9	Перегляд деталей	Клік на проєкт, поява панелі Insights, перемикання між вкладками
10	AI-аналіз гілки	Панель Insights, вкладка Branches, вибір гілки, виклик ШІ-узагальнення та відображення результату
11	Налаштування AI	Сторінка Settings, зміна провайдера, введення ключа, save та перевірка збереженого стану
12	Видалення проєкту	Сторінка /projects, клік DeleteButton, підтвердження та зникнення проєкту зі списку

Запуск тестів виконується командою npm run test:e2e і повний прогон триває близько 45 секунд. У разі невдачі певного тесту Playwright автоматично робить скриншот та зберігає trace-файл, завдяки чому можна візуально відтворити кроки виконання тесту. Звіт Playwright з пройденими E2E-тестами наведено на рисунку 4.3.

Запуск тестів виконується командою npm run test:e2e і повний прогон триває близько 45 секунд. Такий показник є прийнятним для регулярного запуску у локальному середовищі розробника перед кожним комітом та не

створює суттєвого бар'єру для впровадження тестів у щоденний робочий процес. Швидкість досягається завдяки паралельному виконанню тестових файлів на кількох воркерах, що Playwright автоматично налаштовує відповідно до кількості доступних ядер процесора, а також завдяки повторному використанню браузерного контексту між тестами в межах одного файлу.

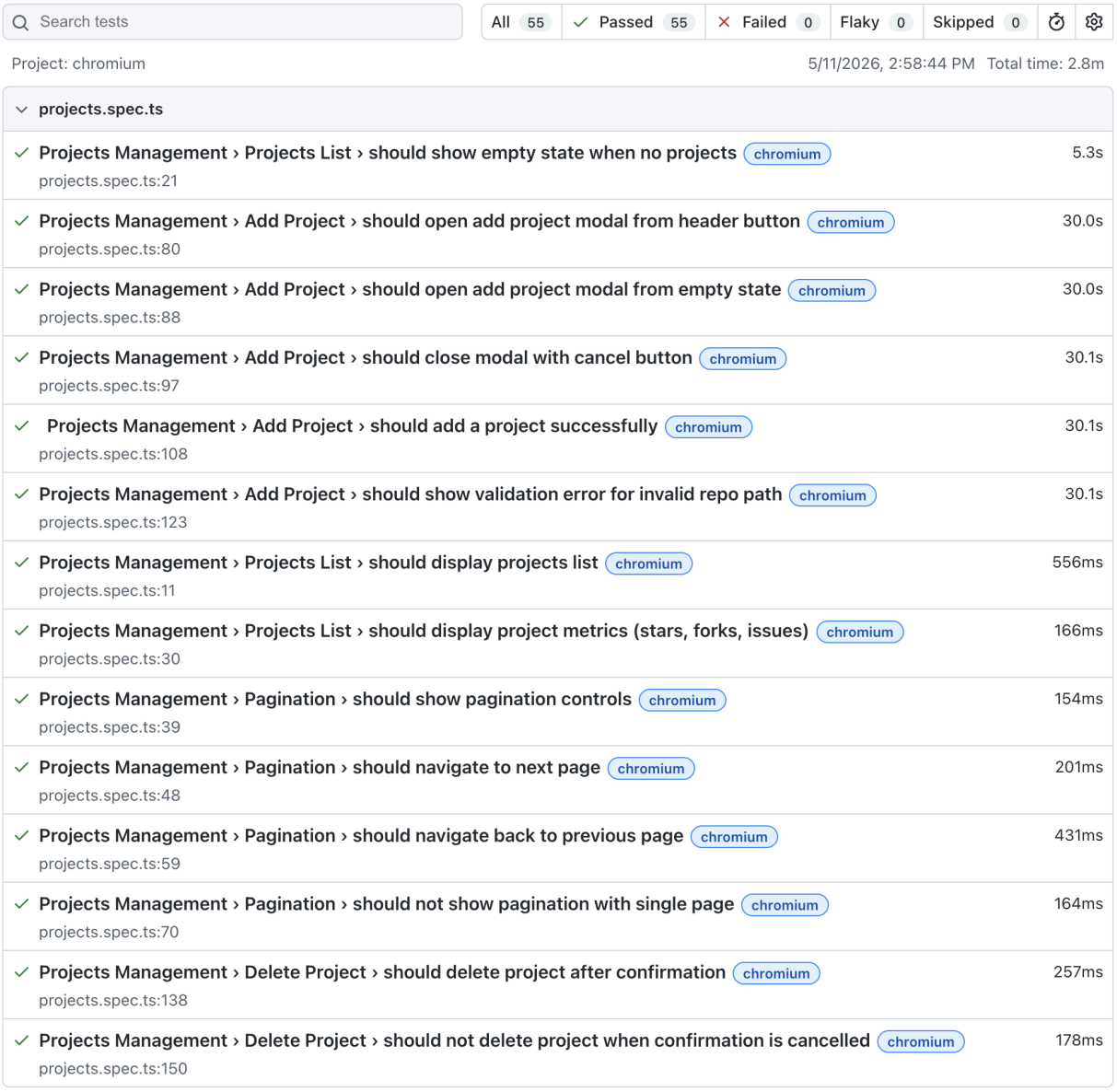


Рис. 4.3 Звіт Playwright з пройденими E2E-тестами

4.5 Демонстрація web-застосунку

У цьому підрозділі наведено скриншоти основних екранів застосунку, які демонструють реалізований функціонал.

Сторінка авторизації (рисунок 4.4) має дві вкладки під назвами «Sign In» і «Sign Up», між якими можна вільно перемикатися без перезавантаження сторінки. Форми оснащені клієнтською валідацією: email перевіряється на коректність формату, а пароль на мінімальну довжину у вісім символів. Якщо сервер повертає помилку, користувач бачить зрозуміле повідомлення.

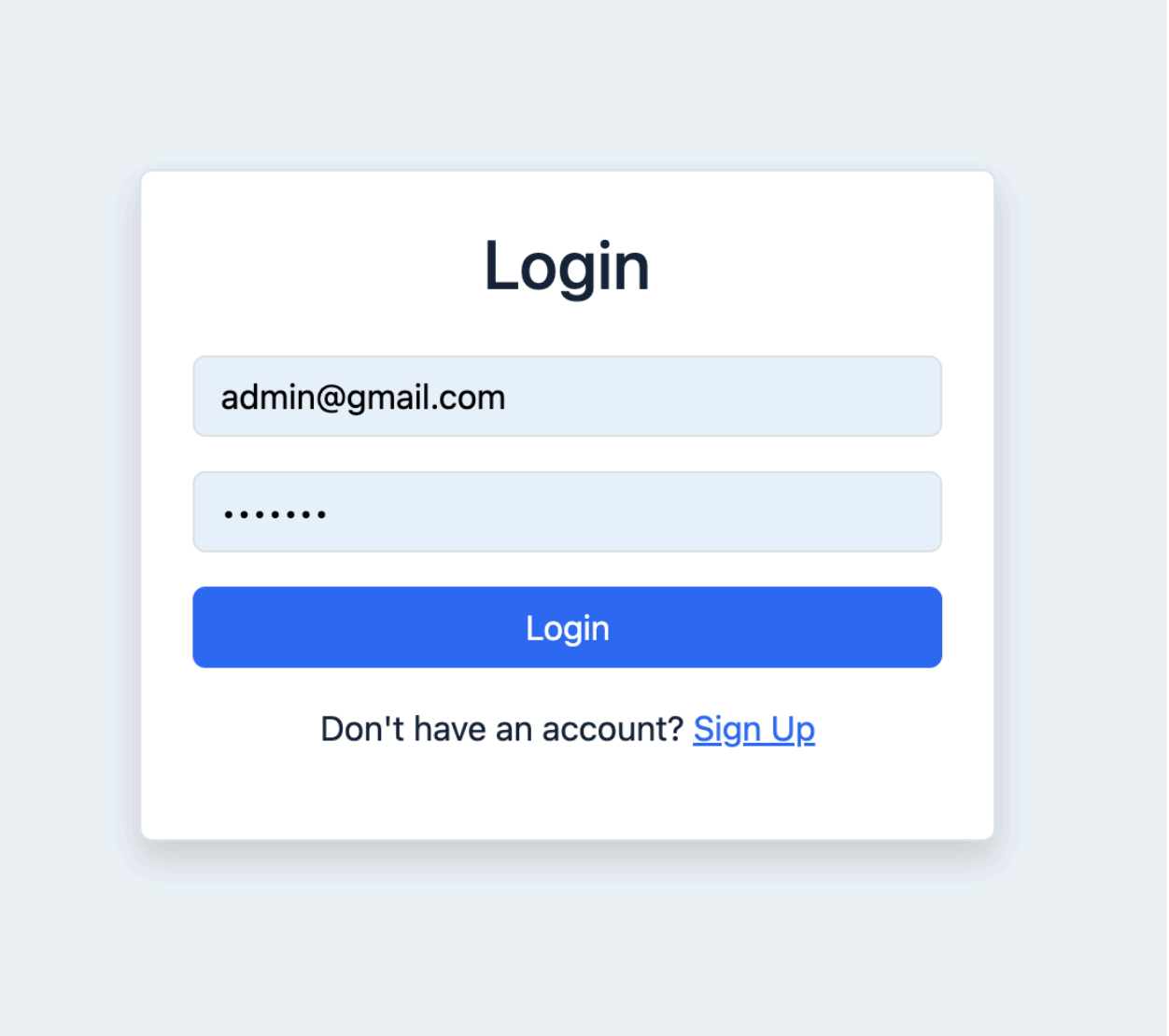


Рис. 4.4 Сторінка автентифікації PRISM AI

Головна сторінка проєктів (рисунок 4.5) показує перелік доданих репозиторіїв у вигляді карток. На кожній картці є назва репозиторію та його власник, кольоровий бейдж платформи (GitHub, GitLab або Bitbucket), ключові метрики (зірки, форки, відкриті задачі), дата створення та набір кнопок керування (favorites, update, delete, expand insights).

Кожну картку реалізовано як окремий React компонент ProjectCard, що приймає об'єкт проєкту через пропси та самостійно відповідає за відображення значень метрик. Метрики відмальовуються з відповідними піктограмами з набору Bootstrap Icons: зірка для stars, гілка для forks та коло із крапкою для відкритих issues. Поряд із картками реалізовано функцію виділення обраних проєктів натискання на іконку зірки у правому верхньому куті картки додає або видаляє проєкт зі списку улюблених. Стан «обрано» зберігається у базі даних та підтягується автоматично після перезавантаження сторінки.

Поряд із картками реалізовано функцію виділення обраних проєктів натискання на іконку зірки у правому верхньому куті картки додає або видаляє проєкт зі списку улюблених.

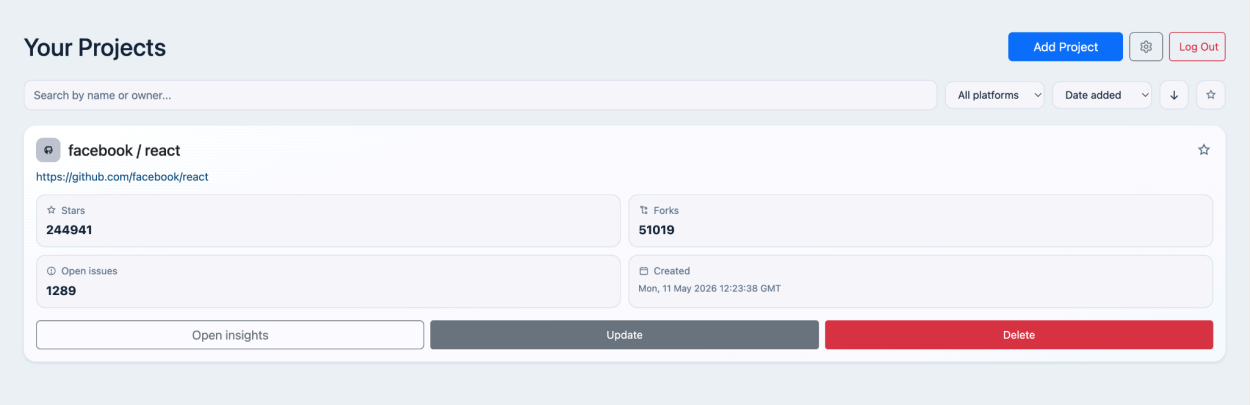


Рис. 4.5 Головна сторінка проєктів

Над списком проєктів розміщено панель керування з полем пошуку, фільтром платформи, вибором сортування та фільтром «обрані». Усі ці налаштування застосовуються одночасно та комбінуються в одному GET-запиті до /api/projects.

Після натискання кнопки розкривання панелі Insights (рисунок 4.6) під карткою з'являється панель з чотирма вкладками. Перемикання між ними відбувається без перезавантаження, а дані для кожної вкладки довантажуються при першому відкритті.

Така стратегія lazy loading дозволяє уникнути надлишкових мережевих запитів до тих розділів, які користувача не цікавлять у поточний момент, та суттєво скорочує час початкового відкриття панелі.

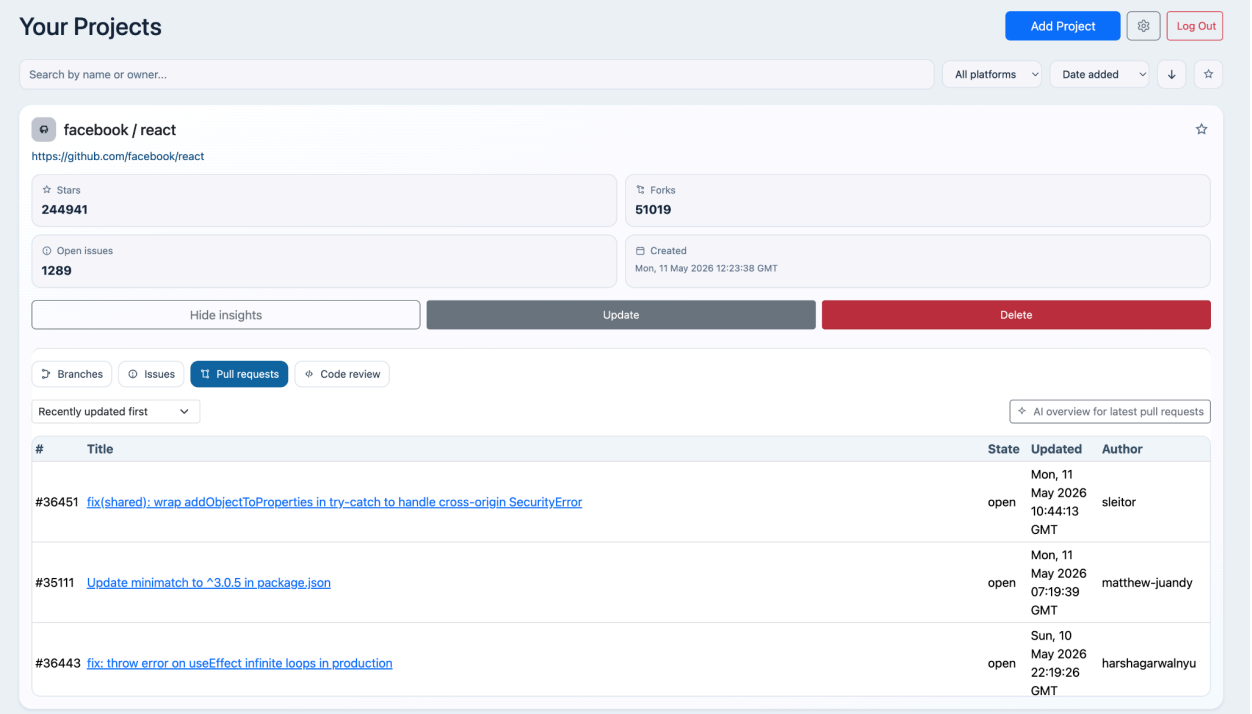


Рис. 4.6 Панель Insights з вкладками Branches, Issues, Pulls та Code Review

Модальне вікно ШІ-узагальнення (рисунок 4.7) відкривається після натискання відповідної кнопки. Під час очікування відповіді LLM (зазвичай від 5 до 15 секунд) у вікні відображається спіннер. Після отримання відповіді користувач бачить згенерований текст разом з кнопкою «Скопіювати у буфер». У разі помилки (наприклад, некоректний API-ключ) на екрані з'являється відповідне повідомлення.

Після отримання відповіді користувач бачить згенерований текст разом з кнопкою «Скопіювати у буфер». Текст відображається з підтримкою Markdown-форматування, що дозволяє моделі структурувати відповідь за допомогою заголовків, списків та виділення ключових фрагментів, роблячи її значно зручнішою для читання порівняно з суцільним абзацом. Натискання на кнопку копіювання використовує сучасний Clipboard API браузера та супроводжується короткочасною зміною підпису кнопки на «Скопійовано», що підтверджує успішність дії без потреби у додатковому сповіщенні.

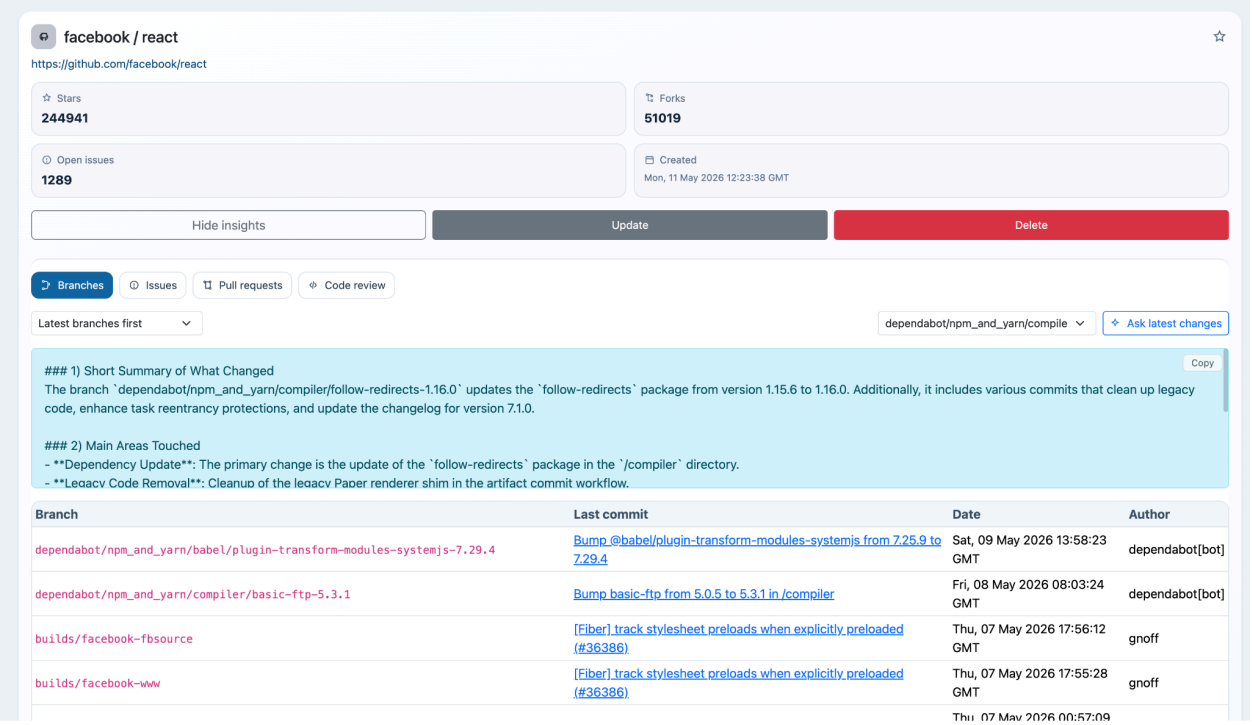


Рис. 4.7 Модальне вікно AI-узагальнення гілки

Сторінка налаштувань (рисунок 4.8) містить форму для конфігурації ШІ-провайдера. Користувач обирає провайдера зі списку (OpenAI, Gemini, DeepSeek, OpenRouter або Custom), вводить API-ключ та за бажанням змінює модель і base URL. Поточний ключ для безпеки відображається у замаскованому вигляді на кшталт ****abcd.

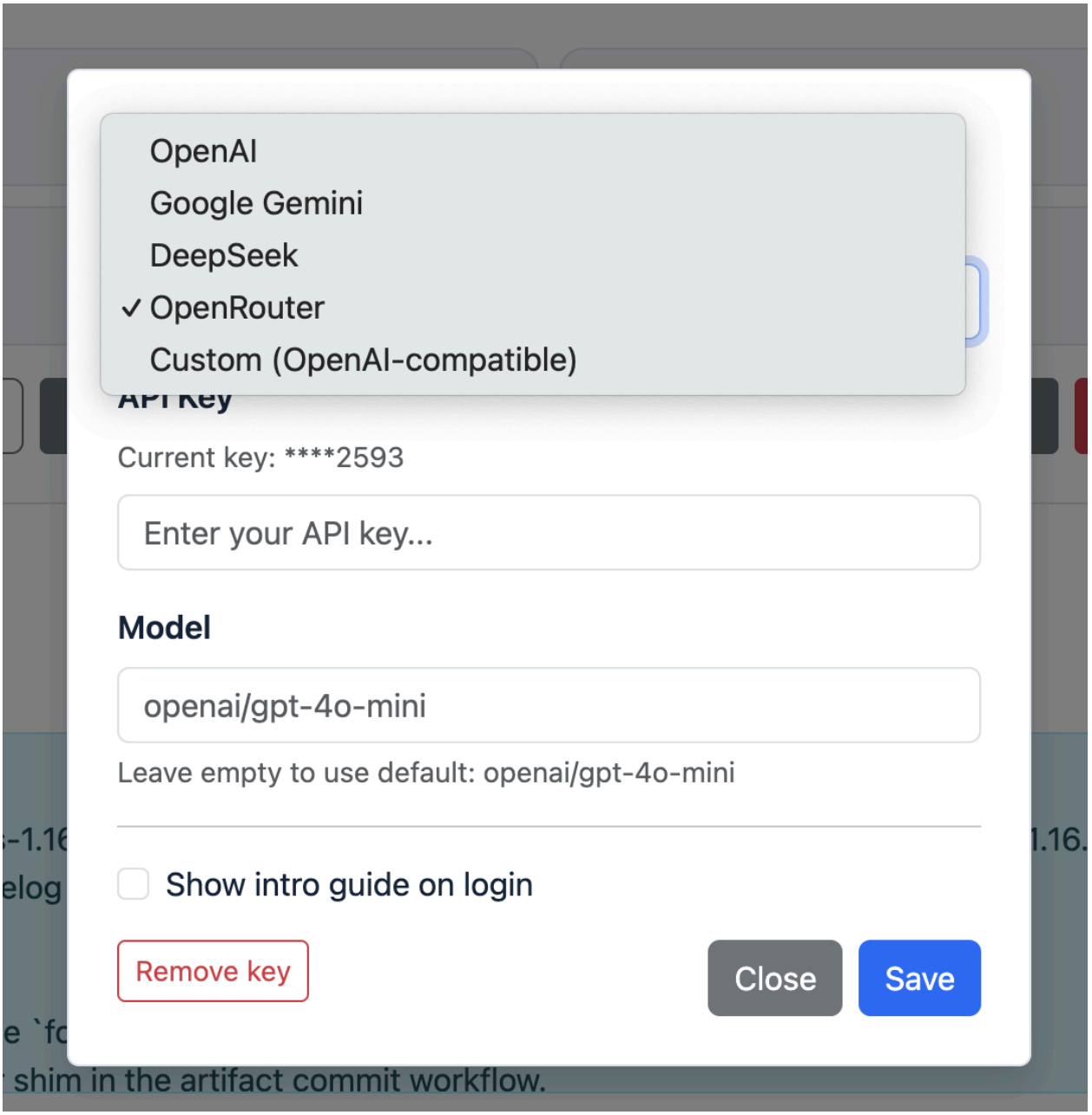


Рис. 4.8 Сторінка налаштувань AI-провайдера

Темна тема (рисунок 4.9) активується через перемикач у верхньому правому куті. Підтримуються три режими: «system» (наслідує системні налаштування), «light» та «dark». Перемикання виконується миттєво за рахунок зміни атрибута data-bs-theme на html-елементі. Усі компоненти Bootstrap нативно підтримують зміну тем без додаткових стилів.

Перемикання виконується миттєво за рахунок зміни атрибута data-bs-theme на html-елементі. Такий підхід використовує вбудований механізм тематизації Bootstrap, що базується на CSS перемінних і дозволяє змінювати кольорову схему без перезавантаження стилів чи перерахунку DOM дерева. Початкове значення атрибута встановлюється синхронно ще до відмалювання сторінки за допомогою невеликого inline скрипту в head, що запобігає неприємному ефекту «миготіння» (flash of unstyled content), коли користувач спочатку бачить світлу тему, а потім різко переключається на темну.

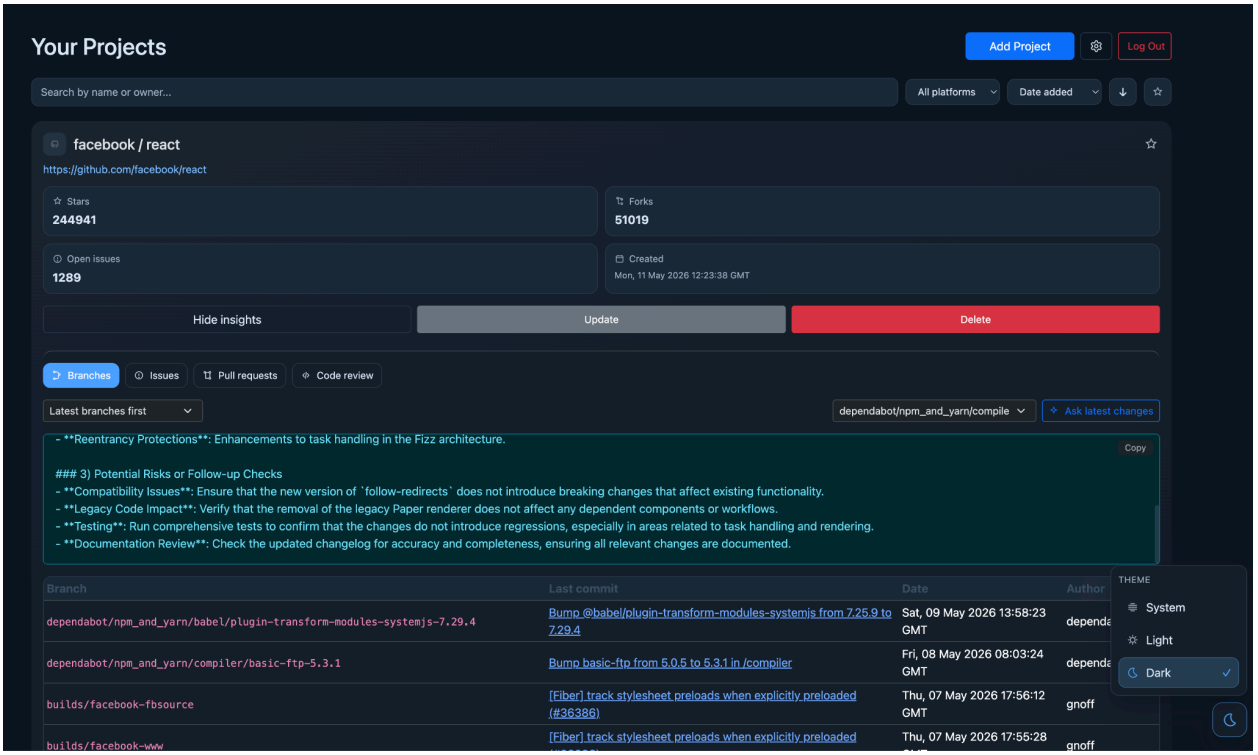


Рис. 4.9 PRISM AI у темній темі

ВИСНОВКИ

У ході виконання кваліфікаційної роботи пройдено повний цикл створення повностекового веб застосунку PRISM AI, призначеного для агрегації та ШІ-аналізу репозиторіїв програмного коду. На етапі аналізу предметної області вивчено сучасні практики супроводу репозиторіїв у DevOps, оглянуто чотири провідні аналоги (GitHub Dashboard, GitLab Operations Dashboard, Gitify, Sourcegraph) та виявлено вільну нішу, яку становить безкоштовне open-source-рішення з мультиплатформною агрегацією та інтегрованою ШІ-аналітикою з налаштовуваним провайдером.

Спираючись на результати аналізу предметної області та портрет цільової аудиторії, сформовано функціональні та нефункціональні вимоги до системи. Обґрунтовано вибір сучасного технологічного стеку, що включає React з TypeScript на клієнтській стороні, Express з TypeScript на сервері, PostgreSQL як СУБД у поєднанні з Prisma ORM, Bootstrap для стилізації, Vite як інструмент збирання, Docker Compose для контейнеризації та Jest, Vitest, Storybook і Playwright для трьох рівнів тестування.

Спроєктовано клієнт-серверну архітектуру з тришаровою організацією бекенду (controllers, services, utils), що забезпечує чітке розмежування відповідальності та зручність тестування. Створено реляційну схему бази даних з мінімальною нормалізацією, у якій лише дві таблиці users та projects пов'язані зв'язком «один-до-багатьох».

Реалізовано REST API з 19 ендпоінтами, які охоплюють усі функціональні вимоги, разом з автоматично згенерованою OpenAPI-документацією у Swagger UI. Усі ендпоінти захищено стандартизованим middleware-стеком, що включає CORS, JSON-парсинг, HTTP-логування, обмеження частоти запитів, JWT-автентифікацію та Zod-валідацію.

Однією з ключових архітектурних інновацій є шар абстракції над провайдерами репозиторіїв (utils/repository-provider.ts), що зводить особливості API GitHub, GitLab та Bitbucket до єдиних TypeScript-інтерфейсів. Завдяки цьому решта системи працює зі сховищами

однаково, незалежно від платформи хостингу. Для підтримки нового провайдера у майбутньому достатньо доповнити лише цей шар, не вносячи правок у бізнес-логіку.

Підключено модуль ШІ-аналітики з підтримкою п'яти OpenAI-сумісних провайдерів (OpenAI, Google Gemini, DeepSeek, OpenRouter та Custom) та п'яти видів аналізу: узагальнення гілок, узагальнення задач, узагальнення запитів на злиття, рев'ю коду з класифікацією знайдених проблем за категоріями (bug, security, performance, style) і рівнем серйозності, а також генерація варіантів виправлення для конкретних фрагментів. Принциповою особливістю реалізації є те, що користувач самостійно конфігурує провайдера та надає власний API-ключ, тоді як сам застосунок виступає лише посередником і не зберігає централізованого ключа.

Клієнтську частину реалізовано у вигляді SPA з вісьмома функціональними компонентами, що побудовані за патерном barrel exports. Кожен компонент має відповідний файл Storybook stories для ізольованої розробки та автоматизованого snapshot-тестування. Також реалізовано систему перемикання тем (system, light і dark) на базі нативних можливостей Bootstrap.

Безпеку забезпечено завдяки JWT-токенам з 24-годинним терміном дії, bcrypt-хешуванню паролів з десятьма раундами, обмеженню частоти запитів (5 на хвилину для входу, 20 на хвилину для ШІ та 100 на хвилину для загальних викликів), Zod-валідації всіх вхідних даних та маскуванню ШІ-ключів у відповідях API.

Виконано контейнеризацію застосунку через Docker Compose з трьома сервісами (frontend на Nginx, backend на Node.js та db на PostgreSQL) із багатоступінчастим збиранням, що дозволяє отримати кінцевий образ frontend розміром близько 25 МБ. Розгортання запускається однією командою docker-compose up.

Проведено комплексне тестування на трьох рівнях: модульне на сервері (Jest з понад 200 тест-кейсами та покриттям приблизно 80 відсотків), модульне на клієнті (Vitest у поєднанні з React Testing Library), компонентне (Storybook зі snapshot-тестами для всіх компонентів) та наскрізне (Playwright з дванадцятьма ключовими сценаріями і повним моканням мережі через page.route).

Підготовлено демонстраційні матеріали, до яких увійшли скриншоти всіх ключових екранів інтерфейсу, фрагменти коду основних модулів та архітектурні діаграми. Матеріали оформлено як презентацію для захисту роботи.

Розроблений веб застосунок PRISM AI є повноцінним продуктом, готовим до промислового використання та таким, що відповідає сучасним стандартам якості програмного забезпечення. Його можна застосовувати як інструмент для full-stack-розробників, DevOps-інженерів, технічних лідів та мейнтейнерів open-source-проєктів, які працюють з репозиторіями на різних платформах хостингу.

Можливі напрями подальшого розвитку системи такі: інтеграція з додатковими платформами (Azure DevOps, Gitea, Forgejo), розширення ШІ-функцій (автоматичні рев'ю запитів на злиття через webhook-інтеграцію, генерація CHANGELOG, прогнозування часу мерджу), підтримка командної роботи (спільні дашборди для команд, ролі та права доступу), інтеграція з системами управління проєктами (Jira, Linear) та побудова мобільних застосунків на React Native для оперативного моніторингу.

Робота пройшла апробацію, за результатами якої було підготовлено такі тези доповідей:

1. Григорчук Д.І., Залива В.В. Визначення вимог до системи управління проєктами (CRM) для публічних проєктів GitHub: Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 269-271.

2. Григорчук Д.І., Залива В.В. Автоматизація аналізу та моніторингу програмних репозиторіїв у CRM системі з використанням великих мовних моделей: Матеріали VII Науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 15 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 341-343.

ПЕРЕЛІК ПОСИЛАНЬ

1. Lwakatare L. E., Kuvaja P., Oivo M. Dimensions of DevOps. *Lecture Notes in Business Information Processing*. 2015. Vol. 212. P. 212-217.
2. Bass L., Weber I., Zhu L. DevOps: A Software Architect's Perspective. Boston: Addison-Wesley Professional, 2015. 352 p.
3. Kim G., Humble J., Debois P., Willis J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland: IT Revolution Press, 2021. 480 p.
4. Lwakatare L. E., Crnkovic I., Bosch J. DevOps for AI: Challenges in Development of AI-enabled Applications. *Software, Telecommunications and Computer Networks (SoftCOM)*. 2020. P. 1-6.
5. Brown T., Mann B., Ryder N. et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*. 2020. Vol. 33. P. 1877-1901.
6. Liu A., Feng B., Xue B. et al. DeepSeek-V3 Technical Report. *arXiv*. 2024. arXiv:2412.19437. 56 p.
7. Cherny B. Programming TypeScript: Making Your JavaScript Applications Scale. Sebastopol: O'Reilly Media, 2019. 314 p.
8. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol: O'Reilly Media, 2020. 310 p.
9. Cantelon M., Harter M., Holowaychuk T., Rajlich N. Node.js in Action. 2nd ed. Shelter Island: Manning Publications, 2017. 392 p.
10. Mardan A. Pro Express.js: Master Express.js — The Node.js Framework for Your Web Development. 2nd ed. New York: Apress, 2024. 248 p.
11. Richardson L., Amundsen M. RESTful Web APIs: Services for a Changing World. Sebastopol: O'Reilly Media, 2013. 408 p.
12. Obe R. O., Hsu L. S. PostgreSQL: Up and Running. 3rd ed. Sebastopol: O'Reilly Media, 2018. 314 p.
13. Krause J. M. Introducing Bootstrap 4. 2nd ed. Berkeley: Apress, 2020. 213 p.
14. Wong J. Pro Docker. New York: Apress, 2023. 380 p.

15. Osherove R. The Art of Unit Testing: with Examples in JavaScript. 3rd ed. Shelter Island: Manning Publications, 2024. 320 p.
16. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force, May 2015. 30 p.
17. Provos N., Mazières D. A Future-Adaptable Password Scheme. *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, 1999. P. 81-91.
18. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: doctoral dissertation. Irvine: University of California, 2000. 162 p.
19. GitHub REST API Documentation. URL: https://docs.github.com/en/rest (дата звернення: 12.03.2025).
20. GitLab REST API Documentation. URL: https://docs.gitlab.com/ee/api/ (дата звернення: 14.03.2025).
21. Bitbucket Cloud REST API Documentation. URL: https://developer.atlassian.com/cloud/bitbucket/rest/intro/ (дата звернення: 16.03.2025).
22. OpenAI API Reference. URL: https://platform.openai.com/docs/api-reference (дата звернення: 18.03.2025).
23. Google AI for Developers: Gemini API. URL: https://ai.google.dev/gemini-api/docs (дата звернення: 20.03.2025).
24. Vite Documentation. URL: https://vitejs.dev/guide/ (дата звернення: 22.03.2025).
25. Prisma ORM Documentation. URL: https://www.prisma.io/docs (дата звернення: 30.03.2025).
26. Zod TypeScript Schema Validation. URL: https://zod.dev (дата звернення: 10.04.2025).
27. Jest Testing Framework Documentation. URL: https://jestjs.io/docs/getting-started (дата звернення: 12.04.2025).
28. Vitest: A Vite-native Testing Framework. URL: https://vitest.dev (дата звернення: 14.04.2025).
29. Storybook 8 Documentation. URL: https://storybook.js.org/docs (дата звернення: 16.04.2025).
30. Playwright End-to-End Testing Documentation. URL: https://playwright.dev/docs/intro (дата звернення: 18.04.2025).

ДОДАТОК А ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Інтелектуальна система управління для публічних проєктів GitHub

Виконав студент 4 курсу
групи ПД-44
Григорчук Дмитро
Керівник роботи
доцент, доктор філософії (PhD) Залива Віталій

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення ефективності управління та аналізу публічних репозиторіїв GitHub за рахунок використання інтелектуального вебзастосунку з централізованим моніторингом і аналізом коду засобами штучного інтелекту.

Об’єкт дослідження: процес управління публічними програмними проєктами на платформі GitHub.

Предмет дослідження: методи, моделі та програмні засоби розробки інтелектуальної системи управління та аналізу публічних репозиторіїв з використанням мовних моделей.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної області та існуючих рішень для управління проєктами GitHub.
2. Визначити функціональні та нефункціональні вимоги до програмного забезпечення.
3. Розробити архітектуру клієнт-серверного вебзастосунку (React, Express, PostgreSQL).
4. Реалізувати серверну частину з REST API, JWT-автентифікацією та інтеграцією з GitHub, GitLab і Bitbucket API.
5. Реалізувати клієнтську частину з адаптивним інтерфейсом та інтегрованою аналітикою на базі моделей штучного інтелекту (OpenAI, Gemini, DeepSeek).
6. Виконати тестування програмного забезпечення на рівнях unit, integration та E2E.

АНАЛІЗ АНАЛОГІВ

Критерій	GitHub Projects	ZenHub	GitKraken	Linear	PRISM AI
Підтримка GitHub / GitLab / Bitbucket	Тільки GitHub	Тільки GitHub	Усі три	Тільки GitHub	GitHub / GitLab / Bitbucket
Аналіз коду штучним інтелектом	–	–	–	–	+
Огляд issues / PR штучним інтелектом	–	–	–	–	+
Перегляд метаданих (stars, forks, issues)	–	+	+	–	+
Доступність як вебзастосунок	+	+	–	+	+

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

- Реєстрація та автентифікація користувачів (JWT)
- Додавання репозиторіїв з GitHub, GitLab, Bitbucket за URL
- Перегляд метаданих репозиторіїв: stars, forks, issues
- Перегляд гілок, issues та pull requests із сортуванням
- Аналіз штучним інтелектом: summary гілок, огляд issues/PR, code review
- Налаштування провайдера штучного інтелекту (OpenAI, Gemini, DeepSeek)

Нефункціональні вимоги

- Адаптивний інтерфейс (підтримка розмірів екрана від 320 px до 4K)
- Rate limiting для захисту API-ендпоінтів
- Валідація вхідних даних
- Контейнеризація
- Документація API
- Покриття коду автоматичними тестами

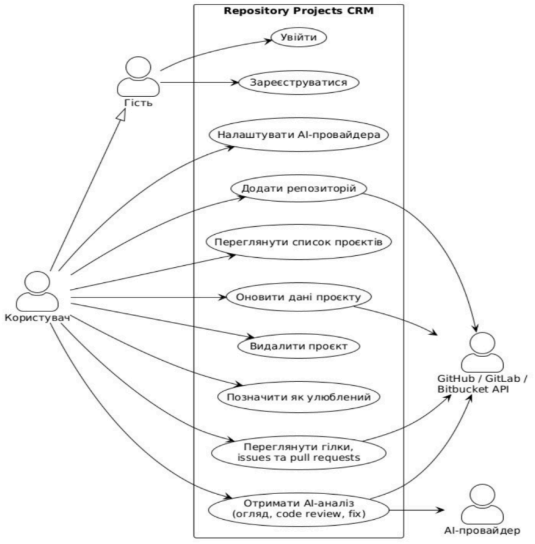
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Prisma

Storybook

Vite

Діаграма варіантів використання



АРХІТЕКТУРА СИСТЕМИ

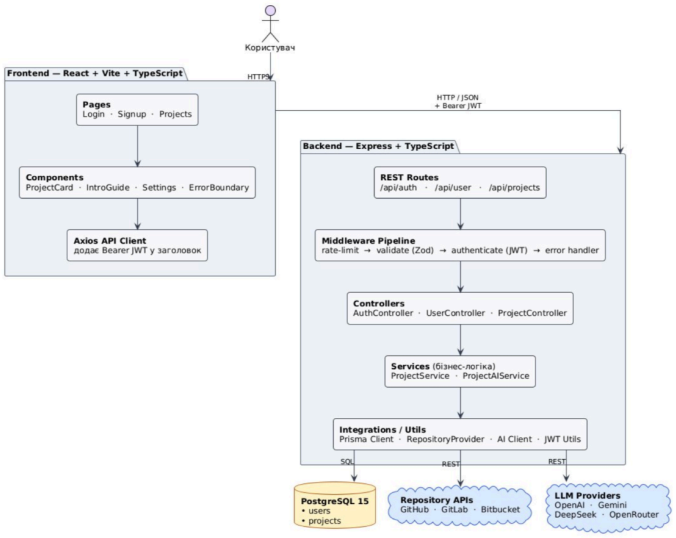
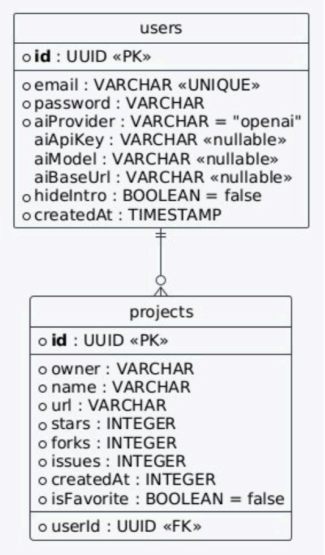


СХЕМА БАЗИ ДАНИХ



ЕКРАННІ ФОРМИ

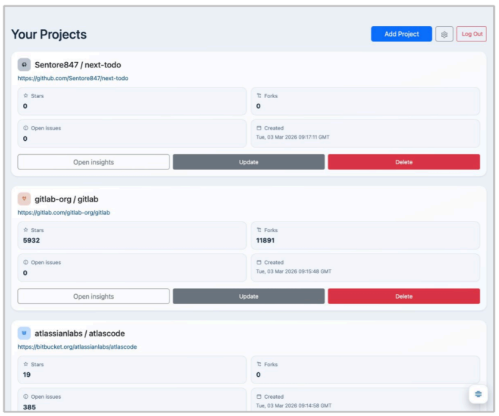


Рис. 1. Головна сторінка проєктів

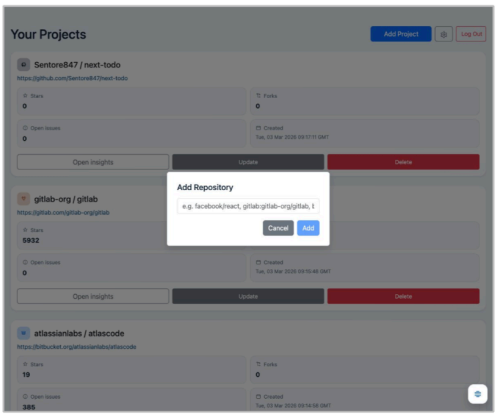


Рис. 2. Додавання репозиторію

ЕКРАННІ ФОРМИ

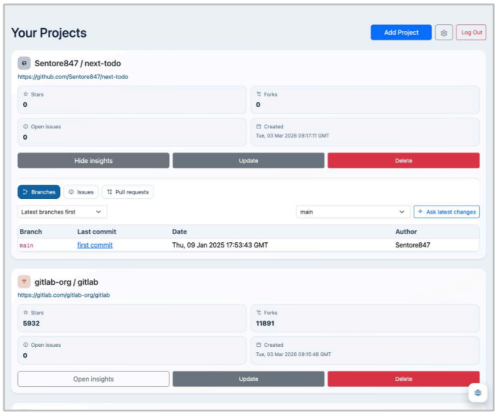


Рис. 3. Аналітика репозиторію

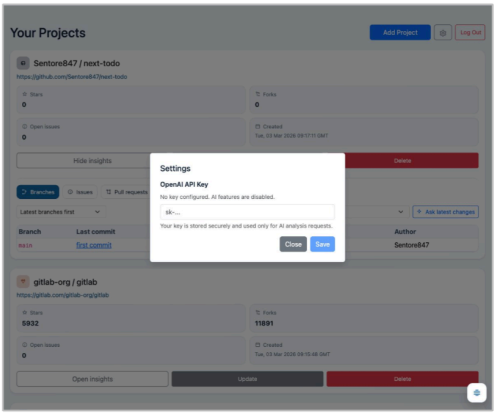


Рис. 4. Налаштування AI

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Григорчук Д.І., Залива В.В. Визначення вимог до системи управління проєктами (CRM) для публічних проєктів GitHub: Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 269-271.
2. Григорчук Д.І., Залива В.В. Автоматизація аналізу та моніторингу програмних репозиторіїв у CRM системі з використанням великих мовних моделей: Матеріали VII Науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 15 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 341-343.

ВИСНОВКИ

1. Проведено аналіз предметної області та існуючих рішень для управління проєктами GitHub, що дозволило визначити функціональні вимоги до системи, виявити обмеження наявних рішень (відсутність уніфікованого доступу до різних git-платформ та AI-аналітики в одному інтерфейсі) та обґрунтувати доцільність розробки власного застосунку.
2. Розроблено архітектуру клієнт-серверного вебзастосунку на базі React, Express та PostgreSQL, завдяки чому забезпечено чітке розділення відповідальності між шарами, можливість незалежного масштабування фронтенду й бекенду, а також гнучкість у подальшому розширенні функціоналу без зміни клієнтської частини.
3. Реалізовано серверну частину з REST API, JWT-автентифікацією та інтеграцією з GitHub, GitLab і Bitbucket API, що дало змогу побудувати єдиний уніфікований інтерфейс доступу до репозиторіїв трьох різних платформ через паттерн `RepositoryProvider`, забезпечити безпечну авторизацію без зберігання сесій на сервері та централізовано керувати правами доступу до ресурсів кожного користувача.
4. Реалізовано клієнтську частину з адаптивним інтерфейсом та інтерактивними компонентами, що забезпечило коректне відображення застосунку на пристроях різного розміру, покращило користувацький досвід завдяки інтерактивному онбордингу, обробці помилок та можливості перемикання тем (System / Light / Dark).
5. Інтегровано AI-аналітику для інтелектуального аналізу коду, code review та генерації підсумків, що дало користувачам змогу автоматично отримувати огляд активності в репозиторії, виявляти потенційні проблеми в коді без ручного аналізу та значно скоротити час, потрібний для оцінки стану проєкту. Підтримка декількох AI-провайдерів через єдиний OpenAI-сумісний інтерфейс забезпечила гнучкість вибору та незалежність від конкретного постачальника.
6. Виконано тестування програмного забезпечення на всіх рівнях: unit-тестування окремих модулів (Vitest на клієнті, Jest на сервері), інтеграційне тестування API-ендпоінтів (Supertest) та наскрізне E2E-тестування користувацьких сценаріїв (Playwright), що дозволило підтвердити коректність роботи бізнес-логіки, виявити та усунути помилки на ранніх етапах розробки, а також гарантувати стабільність системи при подальших змінах і запобігти регресіям.

ДОДАТОК Б ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

server/prisma/schema.prisma

```
generator client {
  provider      = "prisma-client-js"
  binaryTargets = ["native",
"linux-musl-openssl-3.0.x", "debian-openssl-1.1.x"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id          String    @id @default(uuid())
  email       String    @unique
  password    String
  aiProvider  String    @default("openai")
  aiApiKey    String?
  aiModel     String?
  aiBaseUrl   String?
  hideIntro   Boolean   @default(false)
  createdAt   DateTime  @default(now())
  projects    Project[]
  @@map("users")
}

model Project {
  id          String   @id @default(uuid())
  owner       String
  name        String
  url         String
  stars       Int      @default(0)
  forks       Int      @default(0)
  issues      Int      @default(0)
  createdAt   Int
  isFavorite  Boolean  @default(false)
  userId      String
  user        User     @relation(fields: [userId],
references: [id], onDelete: Cascade)
  @@map("projects")
}
```

server/src/middlewares/auth.middlewa re.ts

```
import { Request, Response, NextFunction } from
'express';
import jwt from 'jsonwebtoken';
import { env } from '../config/env';
import { AppError } from '../utils/app-error';

export interface AuthenticatedRequest extends
Request {
  userId?: string;
}

export function authenticate(
  req: AuthenticatedRequest,
  _res: Response,
  next: NextFunction,
): void {
  const auth = req.headers.authorization;
  if (!auth || !auth.startsWith('Bearer ')) {
    throw new AppError(401, 'Missing or invalid
authorization header');
  }
  const token = auth.substring(7);
  try {
    const payload = jwt.verify(token,
env.JWT_SECRET) as { userId: string };
    req.userId = payload.userId;
    next();
  } catch {
    throw new AppError(401, 'Invalid or expired
token');
  }
}
```

services/project-ai.service.ts

```
export async function
summarizeLatestBranchChanges(
  userId: string,
  projectId: string,
  branchName: string,
): Promise<BranchSummaryResponse> {
    const user = await prisma.user.findUnique({ where:
{ id: userId } });
  if (!user || !user.aiApiKey) {
    throw new AppError(400, 'AI provider not
configured');
  }
  const project = await prisma.project.findFirst({
    where: { id: projectId, userId },
  });
  if (!project) throw new AppError(404, 'Project not
found');

  const commits = await fetchRecentCommits(project,
branchName, 10);
  const commitsText = commits
    .map(c => `${c.sha.slice(0, 7)} - ${c.message} by
${c.author}`)
    .join('\n');

  const summary = await callLLM({
    provider: user.aiProvider,
    apiKey: user.aiApiKey,
    model: user.aiModel,
    baseUrl: user.aiBaseUrl,
    messages: [
      { role: 'system', content:
BRANCH_SUMMARY_SYSTEM_PROMPT },
      { role: 'user', content: `Branch:
${branchName}\nCommits:\n${commitsText}` },
```

```
    ],
    maxTokens: 450,
    temperature: 0.2,
  });

  return {
    branchName,
    commitsAnalyzed: commits.length,
    summary,
  };
}


export function ProjectCard({ project, onUpdate,
onDelete }: Props) {
  const [insightsOpen, setInsightsOpen] =
useState(false);
  const [activeTab, setActiveTab] =
useState<TabKey>('branches');
  const [aiResult, setAiResult] = useState<string |
null>(null);
  const [loading, setLoading] = useState(false);

  const handleSummary = async (branchName: string)
=> {
    setLoading(true);
    try {
          const { data } = await
api.post<BranchSummaryResponse>(
        `/projects/${project.id}/ai/branch-summary`,
        { branchName },
      );
      setAiResult(data.summary);
    } catch (err) {
      toast.error(getErrorMessage(err));
    } finally {
      setLoading(false);
    }
  };

  return (
    <Card className="mb-3">
      <Card.Body>
        <div className="d-flex
justify-content-between">
          <h5>{project.owner}/{project.name}</h5>
          <div className="d-flex gap-2">
            <FavoriteToggle project={project} />
            <UpdateProjectButton id={project.id}
onUpdate={onUpdate} />
            <DeleteProjectButton id={project.id}
onDelete={onDelete} />
          </div>
        </div>
        <div className="text-muted small mt-1">
          ★ {project.stars} • ⑂ {project.forks} •
Issues: {project.issues}
        </div>
        <Button
          variant="link"
          onClick={() =>
setInsightsOpen(!insightsOpen)}>
          {insightsOpen ? 'Hide' : 'Show'} insights
        </Button>
        {insightsOpen && (
          <InsightsPanel
            project={project}
            activeTab={activeTab}
            onTabChange={setActiveTab}
            onAiSummary={handleSummary}
            loading={loading}
            aiResult={aiResult}
          />
        )}
      </Card.Body>
    </Card>
  );
}
```

docker-compose.yml

```
version: "3.9"

services:
  frontend:
    build:
          context: ./client
      args:
        VITE_API_URL: ${VITE_API_URL}
    ports:
      - "${PORT_FRONTEND}:80"
    depends_on:
      - backend

  backend:
    build:
```

```
    context: ./server
   ports:
    - "${PORT_BACKEND}:5000"
   environment:
    DATABASE_URL: ${DATABASE_URL}
    JWT_SECRET: ${JWT_SECRET}
    ALLOWED_ORIGINS:
${ALLOWED_ORIGINS}
   depends_on:
    - db

  db:
   image: postgres:15
   environment:
    POSTGRES_USER: ${POSTGRES_USER}
    POSTGRES_PASSWORD:
${POSTGRES_PASSWORD}
    POSTGRES_DB: ${POSTGRES_DB}
   volumes:
    - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```