Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій та робототехніки
(повна назва факультету)
Кафедра комп'ютерних та інформаційних технологій і систем
(повна назва кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістра
(освітньо-кваліфікаційний рівень)

на тему

«Створення та впровадження хмарної платформи для інтелектуальної системи відслідковування й управління ризиковими явищами у програмних продуктах»

Виконав: студент 6 курсу, групи 601-ТН
спеціальності
122 Комп'ютерні науки
(шифр і назва напряму)
Кандзюба Ігор Ігорович
(прізвище та ініціали)
Керівник Скакаліна О.В.
(прізвище та ініціали)
Рецензент _____
(прізвище та ініціали)

Полтава – 2025 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

спеціальність 122 «Комп'ютерні науки»

на тему

«Створення та впровадження хмарної платформи для інтелектуальної системи відслідковування й управління ризиковими явищами у програмних продуктах»

Студента групи 601-ТН Кандзюби Ігора Ігоровича

Керівник роботи:

кандидат технічних наук,

доцент Скакаліна О.В.

Завідувач кафедри:

кандидат фізико-математичних наук,

доцент Двірна О. А

Полтава – 2025

РЕФЕРАТ

Кваліфікаційна робота магістра: 159 с., 40 малюнків, 8 таблиць, 4 додатки, джерела.

Об’єкт дослідження: процеси створення, впровадження та оптимізації хмарної платформи для забезпечення функціонування інтелектуальної системи управління ризиковими явищами з інтеграцією штучного інтелекту для аналізу звітів про ризикові явища у програмному забезпеченні.

Мета роботи: розробка хмарної платформи для забезпечення функціонування інтелектуальної системи управління ризиками, що включає вебзастосунок для відслідковування завдань та застосування алгоритмів штучного інтелекту для аналізу звітів про помилки у програмному забезпеченні, зокрема їх типів, серйозності та пріоритетності, з метою підвищення ефективності управління ризиковими явищами у програмному забезпеченні.

Методи: проектування та розробка хмарної платформи для розгортання вебзастосунків, створення бази даних для зберігання звітів про ризикові явища, реалізація алгоритмів штучного інтелекту для оцінки пріоритету та серйозності ризикових явищ, розробка інтерфейсу користувача на основі React, інтеграція серверної частини з використанням Node.js, впровадження та тестування рішення на Oracle Cloud.

Ключові слова: хмарна платформа, відслідковування ризикових явищ, штучний інтелект, React, Node.js, Express, Sequelize, PostgreSQL, Python, GitLab CI/CD, автоматизація впровадження, аналіз звітів про помилки у ПЗ.

Master’s thesis: 159 p., 40 figures, 8 tables, 4 appendices, 33 sources.

S

Q

L

ЗМІСТ

Д [illegible]

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

API – Application Programming Interface

JS – JavaScript

RPC – Remote Procedure Call

DOM – Document Object Model

JSON – JavaScript Open Notation

REST – REpresentational State Transfer

– Hypertext Transfer Protocol (Secure)

SQL – Structured Query Language

MVC – Model-View-Controller

XML – Extensible Markup Language

БД – База даних

ШІ – Штучний інтелект

СКБД – Система керування базою даних

ООП – Об’єктно-орієнтоване програмування

ПЗ – Програмне забезпечення

ER-діаграма – Entity Relationship діаграма

VPS – Virtual Private Server

CI/CD – Continuous Integration / Continuous Development

DNS – Domain Name System

SMOTE – Synthetic Minority Oversampling Technique

SPA – Single Page Application

VCN – Virtual Cloud Network

CLI – Command Line Interface

UI/UX – User Interface / User Experience

– Java Script XML

– Oracle Cloud Infrastructure

– Software Development Kit

– Unified Modeling Language

– Secure Socket Layer

MIME – Multipurpose Internet Mail Extensions

– Platform as a Service

– Term Frequency–Inverse Document Frequency

– Comma Separated Values

– Amazon Web Services

– Transport Layer Security

– Object Relational Mapping

SSH – Secure Shell

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій та цифрової трансформації все більше компаній звертають увагу на ефективність управління програмними продуктами. Програмне забезпечення стає все складнішим, а вимоги до його якості, надійності та безпеки – більш суворими. Однак, зростання складності також збільшує кількість потенційних ризикових явищ, що можуть виникати на всіх етапах розроблення. У зв'язку з цим надзвичайно важливим стає впровадження систем, які допомагають розробникам оперативно й ефективно відстежувати, аналізувати та керувати такими явищами.

Одним із підходів до вирішення цієї проблеми є використання хмарних технологій для впровадження інтелектуальних систем, які об'єднують інструменти відслідковування і управління ризиковими явищами безпосередньо у програмних продуктах. Хмарні платформи надають безліч переваг вебзастосункам, такі як масштабованість, доступність, гнучкість та здатність до швидкої адаптації до змін. Крім того, використання інтелектуальних алгоритмів у поєднанні з такими платформами дозволяє створювати системи, які допомагають не тільки відслідковувати та управляти ризиковими явищами, але й проводити інтелектуальний аналіз створених звітів.

Застосування штучного інтелекту в таких системах відкриває нові можливості для аналізу, адже алгоритми машинного навчання дозволяють автоматично визначати відповідність опису помилок їхнім пріоритетам і серйозності. Обраний підхід сприяє точнішому плануванню ресурсів і встановленню пріоритетів завдань команди розробників. Крім того, описані технології забезпечують автоматизацію аналізу звітів, зосереджуючись на ключових аспектах для покращення ефективності управління ризиками.

Метою даної роботи є розробка хмарної платформи для впровадження інтелектуальної системи відслідковування й управління ризиковими явищами у програмних продуктах. Досягнення цієї мети передбачає створення вебзастосунку для відстеження й управління помилками, реалізацію розподільчої

системи призначення завдань з урахуванням складності й досвіду команди розробників, аналіз створених звітів за допомогою штучного інтелекту, а також впровадження цього проекту на власноруч розроблену хмарну платформу. Такий підхід забезпечить усі переваги, пов’язані з масштабованістю, доступністю та гнучкістю системи, що були зазначені у попередніх абзацах, і дозволить оптимізувати процеси аналізу та управління ризиками в умовах реального використання.

Актуальність дослідження зумовлена необхідністю зниження витрат на усунення помилок у програмному забезпеченні та підвищення ефективності управління процесами розробки. Використання хмарної інфраструктури на базі Oracle Cloud забезпечить високий рівень доступності та надійності платформи, що дозволить значно прискорити процеси аналізу та вирішення проблем у програмних продуктах.

Отже, у дипломній роботі детально розглядатиметься процес створення хмарної платформи та впровадження інтелектуальної системи, що реалізовуватиме методи відстежування й управління ризиковими явищами у програмних продуктах.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД ХМАРНИХ ПЛАТФОРМ ТА ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ УПРАВЛІННЯ РИЗИКОВИМИ ЯВИЩАМИ

Опис предметної області

Хмарні платформи та інтелектуальні системи управління ризиковими явищами займають важливе місце у сучасних інформаційних технологіях. Вони відіграють ключову роль у забезпеченні високої якості програмних продуктів, оптимізації процесів розробки, а також підвищенні ефективності управління ресурсами. З розвитком цифрової трансформації та зростанням складності програмних рішень стає критично важливим використовувати інноваційні підходи для своєчасного виявлення і вирішення дефектів у програмному забезпеченні.

Хмарні платформи – це середовища, які забезпечують доступ до обчислювальних ресурсів і даних через інтернет. Основними характеристиками таких платформ є масштабованість, гнучкість, висока доступність і зменшення витрат на інфраструктуру [1]. Вони дають змогу компаніям зосередитися на розробці програмних продуктів, а не на управлінні інфраструктурою. Хмарна інфраструктура, як-от Oracle Cloud, надає можливості для розгортання відповідних платформ, зберігання даних, аналізу великих обсягів інформації та інтеграції з іншими сервісами [2].

Впровадження застосунків на хмарній платформі порівняно з традиційним сервером або хостингом має низку суттєвих відмінностей, що впливають на процеси розробки, розгортання та управління застосунками. Такі платформи надають автоматизоване управління інфраструктурою, що позбавляє розробників необхідності займатися налаштуванням і підтримкою серверів та інших компонентів.

У випадку з традиційними хостингом, весь тягар управління інфраструктурою лягає на плечі команди розробників або адміністраторів. Вони повинні забезпечувати належну конфігурацію та оптимізацію серверів, встановлювати необхідне програмне забезпечення, гарантувати безпеку системи, а також виконувати її оновлення та підтримку. Таким чином, традиційний хостинг потребує значних зусиль і часу, а також технічних знань і досвіду в управлінні серверними середовищами.

Хмарні платформи зазвичай мають інтегровані механізми для автоматичного масштабування застосунків залежно від змін у навантаженні. Ця особливість дозволяє легко адаптуватися до зростання користувацького попиту без необхідності вносити суттєві зміни до коду чи конфігурації середовища. Натомість використання традиційних серверів потребує ручного втручання для забезпечення масштабованості, що включає розширення інфраструктури або перенесення застосунку на більш потужні сервери [3].

Що стосується інтеграції з іншими сервісами, хмарні платформи зазвичай пропонують більш гнучкі та прості у використанні засоби для зв’язку з різними інструментами і сервісами, такими як бази даних, системи моніторингу та аналітики. У традиційному підході інтеграція з такими сервісами може бути значно складнішою, оскільки вимагає додаткових налаштувань та конфігурацій.

Крім того, у питаннях безпеки та оновлення хмарні платформи мають явну перевагу, забезпечуючи автоматичне застосування оновлень, що мінімізує ризики вразливостей. Натомість у традиційному середовищі безпека і підтримка завжди залишаються на відповідальності розробників і адміністраторів, що потребує додаткових зусиль для забезпечення надійного захисту даних і програм.

Зрештою, хмарні платформи надають більш зручне середовище для розгортання застосунків завдяки своїй автоматизації та інтеграції DevOps практик, які значно спрощують процеси розробки та впровадження. Традиційний серверний підхід, хоча і забезпечує повний контроль над усіма аспектами середовища, вимагає значно більше технічних знань, часу і ресурсів для підтримки та оптимізації.

Перш ніж розбирати системи, які націлені на роботу з ризиковими явищами, потрібно надати їм точне визначення.

Ризикові явища (дефекти, баги) – це помилки, недоліки або несправності у програмному забезпеченні, які призводять до неправильної роботи програми або її невідповідності очікуваним результатам [4]. Вони виникають через помилки у вихідному коді, проектуванні або в процесі розробки, і можуть впливати на функціональність, продуктивність, стабільність або безпеку програми. Головною метою команди розробників є їх виявлення та виправлення для забезпечення належної роботи програмного продукту.

Під час роботи з дефектами в різних застосунках важливо чітко розуміти їхній «життєвий шлях», тому далі розглянемо різні статуси помилок.

Усе починається з того, що тестувальник виявляє дефект, який у контексті контролю якості отримує статус «виявлено» (Submitted). Після успішної реєстрації в системі йому присвоюється статус «новий» (New).

Після цього, залежно від рішення менеджера проекту, з дефектом можуть відбутися наступні сценарії:

відхилено» (Declined) – дефект може бути не визнаний таким або вважатися неактуальним, що призводить до його відхилення;
відкладений» (Deferred) – виправлення цього дефекту не є пріоритетом на поточному етапі розробки, тому виправлення відкладається;
відкрито» (Open) – відповідальна особа підтверджує наявність помилки та визначає, що її потрібно виправити.

Якщо ризикове явище підтверджено, йому спочатку надається статус «призначено» (Assigned), що означає, що виправлення цього дефекту доручено конкретному розробнику. Коли відповідальний розробник успішно усуває помилку, їй надається статус «виправлено» (Fixed).

Після цього дефект проходить перевірку, і в залежності від результату йому призначаються наступні статуси:

перевірено» (Verified) – тестувальник перевірив, чи дійсно дефект був усунутий;

повторно відкрито» (Reopened) – якщо під час перевірки виявляється, що дефект все ще присутній, його знову відкривають для подальшого виправлення;
закрито» (Closed) – після успішного виправлення дефекту та кількох циклів перевірок, остаточно закривається, оскільки більше не потребує уваги команди.

На рисунку 1.1 представлена діаграма, яка ілюструє ключові етапи життєвого циклу ризикових явищ.

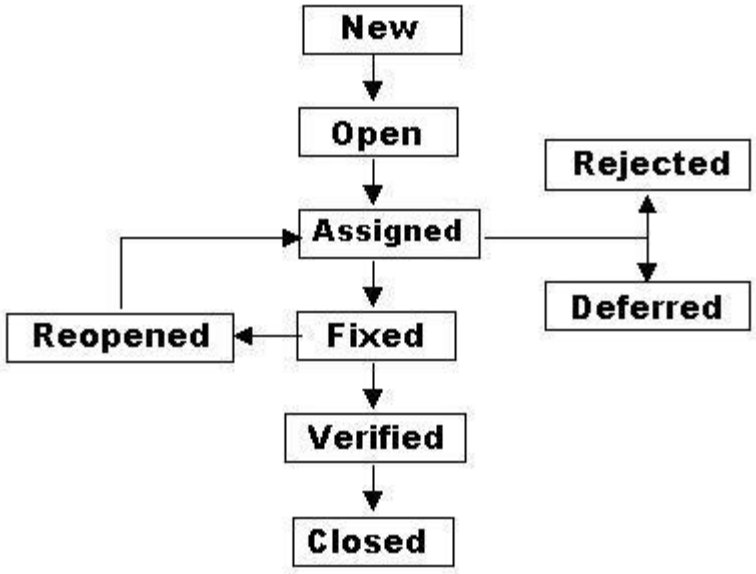

Рисунок 1.1 – Життєвий цикл ризикових явищ у ПЗ

Інтелектуальні системи управління ризиковими явищами у програмних продуктах спрямовані на пришвидшення процесів аналізу та усунення помилок, що виникають під час розробки та використання програмного забезпечення. Такі системи забезпечують функціонал для додавання звітів про помилки та відстеження проектів, із якими вони пов’язані. Завдяки цьому команди розробників, тестувальники та менеджери проектів можуть ефективніше взаємодіяти, координувати роботу й контролювати прогрес усунення помилок. Важливою складовою таких систем є використання розумних алгоритмів зі штучним інтелектом для автоматизації процесів аналізу, пріоритизації помилок і надання рекомендацій щодо їх виправлення. Таким чином, можна мінімізувати

вплив помилок на кінцевих користувачів, скоротити час на їх усунення та покращити загальну якість програмного продукту.

Об’єднання цих двох аспектів – хмарних платформ та інтелектуальних систем управління ризиками створює потужну інфраструктуру, яка забезпечує впровадження розроблюваної інтелектуальної системи для управління і відслідковування ризикових явищ. Об’єднання цих двох аспектів дає змогу значно підвищити ефективність управління життєвим циклом програмного забезпечення, прискорити випуск оновлень і мінімізувати витрати на технічне обслуговування.

Для створення вебзастосунку, який отримає найбільший приріст ефективності після впровадження на хмарну платформу, рекомендовано дотримуватися певних стандартів при розробці. Як приклад, можна виділити наступні стандарти вебзастосунків у контексті використання хмарних платформ:

сі компоненти вебзастосунку повинні бути організовані у вигляді модулів, які відповідають за конкретні функціональні блоки.

Монолітна архітектура передбачає об’єднання всіх функцій застосунку в єдиній серверній системі, зберігаючи чіткий поділ логіки через використання модулів. Вебзастосунок включає централізовану базу даних та API для комунікації з додатковими компонентами, такими як сервіс штучного інтелекту. Основна бізнес-логіка застосунку реалізована на одній платформі, як приклад можна обрати Node.js, тоді як сервіс штучного інтелекту забезпечує обробку специфічних завдань, наприклад аналіз звітів про дефекти. Взаємодія між центральним сервером і допоміжними сервісами здійснюється через чітко визначені REST або RPC-інтерфейси.

икористання контейнерів, що спрощує розгортання та управління застосунком.

Контейнеризація за допомогою технологій, таких як Docker, дозволяє упакувати застосунок разом з усіма його залежностями в ізольоване середовище, що забезпечує стабільність роботи застосунку в різних середовищах (локально, на тестових серверах, у хмарі тощо). Контейнери дозволяють швидко розгортати,

масштабувати і оновлювати впроваджені сервіси, що полегшує управління та знижує ймовірність конфліктів між компонентами системи.

изайн API повинен дотримуватися RESTful принципів або GraphQL (залежно від вимог), забезпечуючи стійкість, гнучкість і сумісність з іншими системами.

PI є основою для зв'язку між сервісами та іншими компонентами застосунку. RESTful API слідує встановленим правилам і конвенціям, що полегшують розуміння, розробку і використання API [5]. GraphQL є альтернативним підходом, що надає можливість запитувати лише ті дані, які необхідні, що знижує обсяг переданих даних та прискорює роботу застосунку. Вибір між REST і GraphQL залежить від конкретних вимог до продуктивності, гнучкості та типу запитів.

икористання стандартів OAuth 2.0 або JWT (JSON Web Token) для автентифікації користувачів і контролю доступу до ресурсів.

OAuth 2.0 і JWT є широко визнаними стандартами для автентифікації та авторизації. OAuth 2.0 використовується для надання обмеженого доступу до ресурсів стороннім застосункам без передачі паролів користувачів. JWT – це формат токенів, що дозволяє безпечно зберігати інформацію про сесію користувача і передавати її між клієнтом і сервером. Токени JWT зазвичай використовуються для автентифікації в сучасних вебзастосунках через їх легкість і можливість швидкого перевірки [6].

сі дані, що передаються між клієнтом та сервером, повинні бути зашифровані за допомогою протоколу HTTPS/TLS.

Шифрування даних є необхідною умовою для захисту інформації, що передається між клієнтом і сервером, від несанкціонованого доступу або прослуховування. HTTPS – це безпечна версія HTTP, яка використовує протокол TLS для шифрування даних [7]. Завдяки цьому забезпечуються конфіденційність і цілісність даних, що особливо важливо для захисту особистих даних і безпеки онлайн-транзакцій.

онфіденційна інформація повинна зберігатися в спеціалізованих сховищах і не повинна бути включена безпосередньо в код.

Зберігання чутливої інформації, такої як API–ключі, токени доступу, паролі тощо, повинно бути організоване у спосіб, який мінімізує ризик витоку даних. Замість того, щоб включати ці дані безпосередньо в код, слід використовувати спеціалізовані файли конфігурації, такі як .env файли, або системи управління секретами.

од повинен бути добре документований з використанням коментарів, щоб полегшити його розуміння і супровід.

Документація є критично важливою для підтримки й розвитку будь–якого застосунку. Коментарі в коді допомагають іншим розробникам швидко зрозуміти логіку роботи програмного забезпечення, що знижує час на навчання нових членів команди та спрощує процес внесення змін. Вони повинні бути короткими, але інформативними, пояснюючи складні алгоритми або важливі рішення прийняті при розробці.

икористання стилів кодування, що відповідають стандартам конкретної мови програмування (наприклад, PEP 8 для Python, Airbnb Style Guide для JavaScript/React тощо).
астосунок повинен підтримувати горизонтальне масштабування для обробки збільшеного навантаження без значних змін у коді.

Горизонтальне масштабування передбачає додавання нових серверів або екземплярів застосунків для розподілу навантаження замість збільшення потужності одного сервера. Таке рішення дає змогу легко адаптувати систему до зростання кількості користувачів або обсягу даних без суттєвих змін у структурі чи логіці коду. Обрана архітектура забезпечує високу доступність і стійкість застосунку.

Огляд існуючих програмних рішень

На сьогодні існує чимало рішень для управління ризиковими явищами в програмному забезпеченні. Проте більшість із них охоплює широкий спектр додаткових функцій, зокрема управління проектами чи впровадження різних методологій розробки програмного забезпечення. У подальшому буде проведено огляд основних платформ, їх ключового функціоналу та переваг. На основі цього аналізу буде зроблено висновки щодо базового набору функцій, характерних для сучасних систем управління ризиками.

або MantisBT) – безкоштовна система з відкритим кодом для відстеження дефектів і управління проектами. Вона розрахована на команди розробників, які потребують простого та ефективного інструменту для реєстрації та контролю ризикових явищ і завдань у процесі розробки програмного забезпечення. MantisBT має інтуїтивний інтерфейс, завдяки чому підходить як для команд початківців, так і для досвідчених розробників [8].

Mantis дозволяє користувачам створювати завдання, додавати детальний опис, призначати відповідальних і встановлювати пріоритети для кожного з них (рис. 1.2).

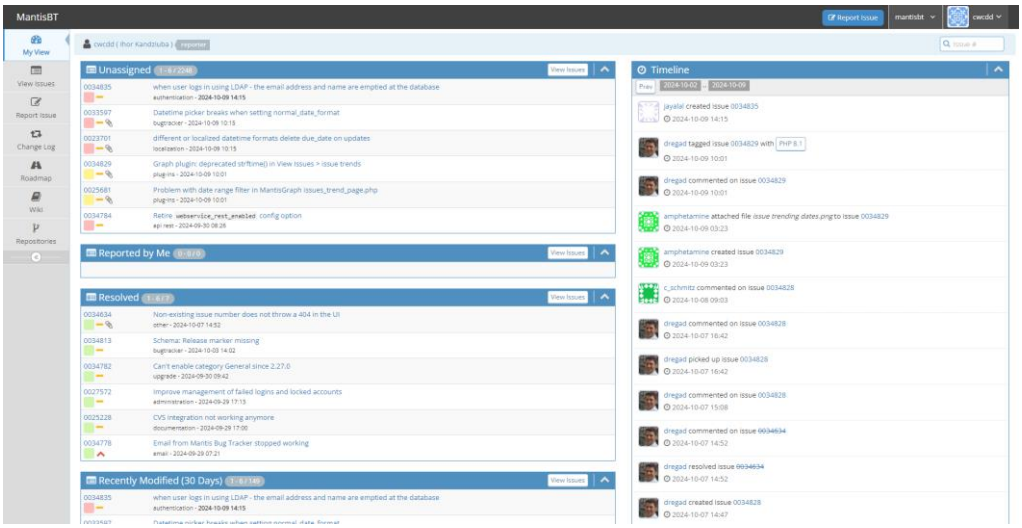

Рисунок 1.2 – Головна сторінка MantisBT

Також є можливість додати інформацію про статус помилки, категорію, серйозність, повторюваність і терміновість, що дозволяє організувати завдання більш структуровано (рис. 1.3).

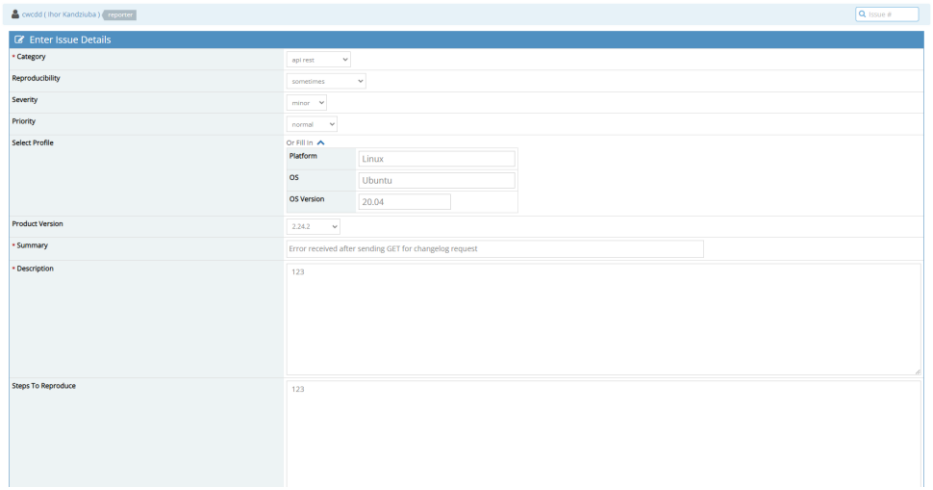

Рисунок 1.3 – Приклад створення звіту про помилку для проекту в Mantis

Застосунок підтримує ведення кількох проектів одночасно з можливістю створення підпроектів (рис. 1.4). Користувачі можуть налаштовувати дозволи і ролі для учасників проекту, визначаючи, які дії можуть виконувати різні члени команди.

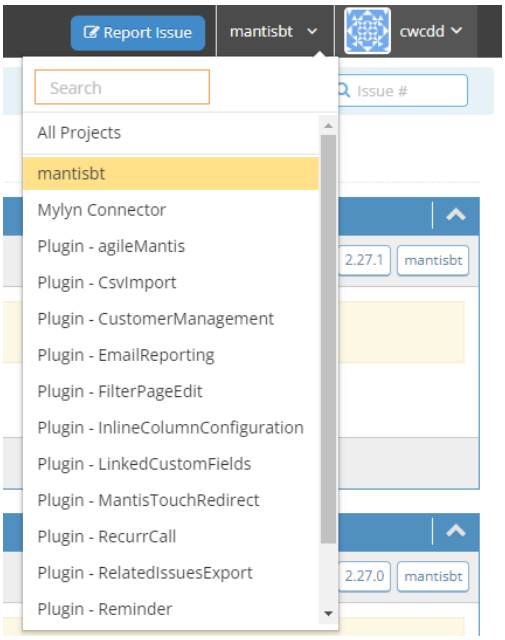

Рисунок 1.4 – Меню вибору проектів

Інструмент надає розширені можливості для фільтрації та пошуку задач, дозволяючи користувачам швидко знаходити потрібну інформацію. Є можливість створювати та зберігати власні фільтри для частого використання, що спрощує процес пошуку специфічних помилок.

MantisBT є веборієнтованим застосунком, що означає його доступність через браузер з будь–якої операційної системи, не вимагаючи додаткового програмного забезпечення на комп'ютері користувача. Водночас, завдяки відкритому коду, компанії можуть встановлювати його на власних серверах для забезпечення автономності та контролю над даними.

Jira – це популярний інструмент для управління проектами і відслідковування помилок у ПЗ, розроблений компанією Atlassian. Спочатку Jira була створена як платформа для відслідковування помилок у програмних застосунках, але з часом вона еволюціонувала в потужний інструмент для управління проектами, особливо для команд, які використовують методології

Компанія Atlassian з роками додала значну кількість функціоналу до Jira, але з найголовнішого можна виділити управління проектами, а саме створення,

призначення та пріоритизація помилок у ПЗ, завдань та інших робочих елементів, а також відстеження їхнього прогресу.

Головна сторінка Jira забезпечує користувача інструментами для швидкого доступу до важливої інформації та навігації між проектами і завданнями (рис. Тут розташовані розділи, які відображають останні проєкти, призначені завдання, переглянуті елементи та інші ключові аспекти роботи. У верхній частині розташоване меню для переходу між основними функціями системи, такими як проекти, фільтри та плани. Інтерфейс організований так, щоб спростити управління робочими процесами та надати зручний доступ до ключових дій.

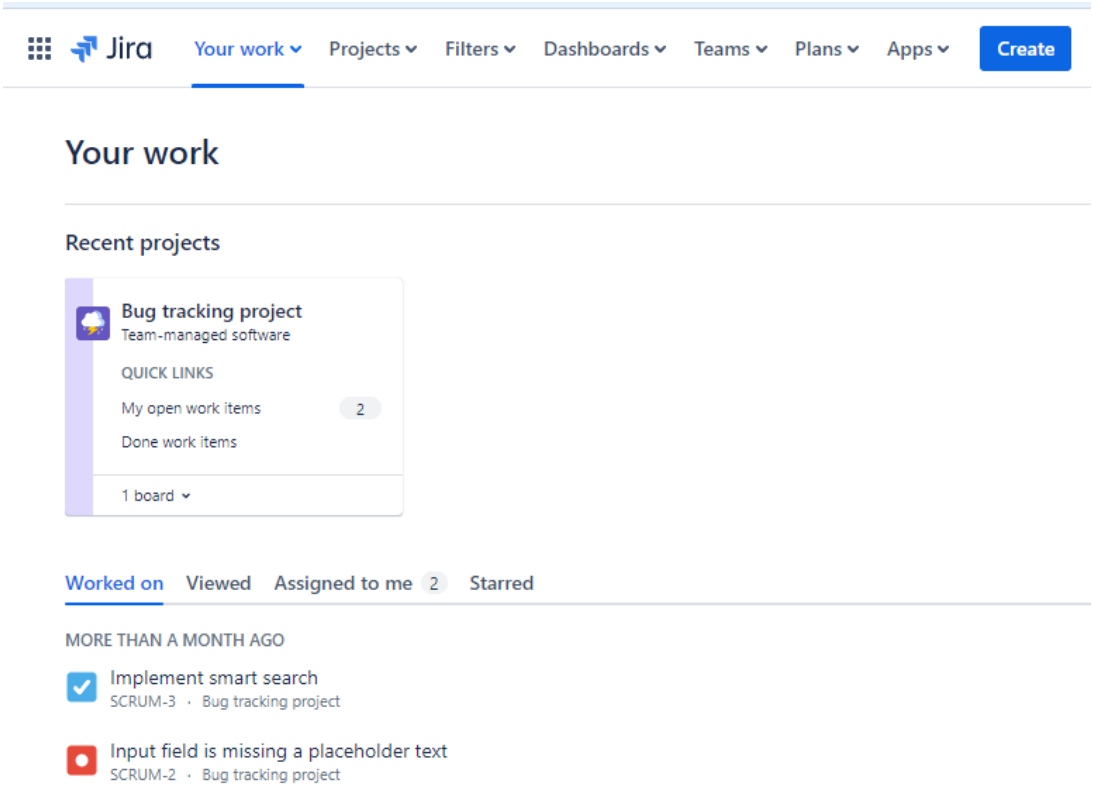

Рисунок 1.5 – Фрагмент головної сторінка Jira

Також при створенні завдання, можна призначити: команду розробників, виконавця, пов’язані з нею інші завдання тощо (рис. 1.6).

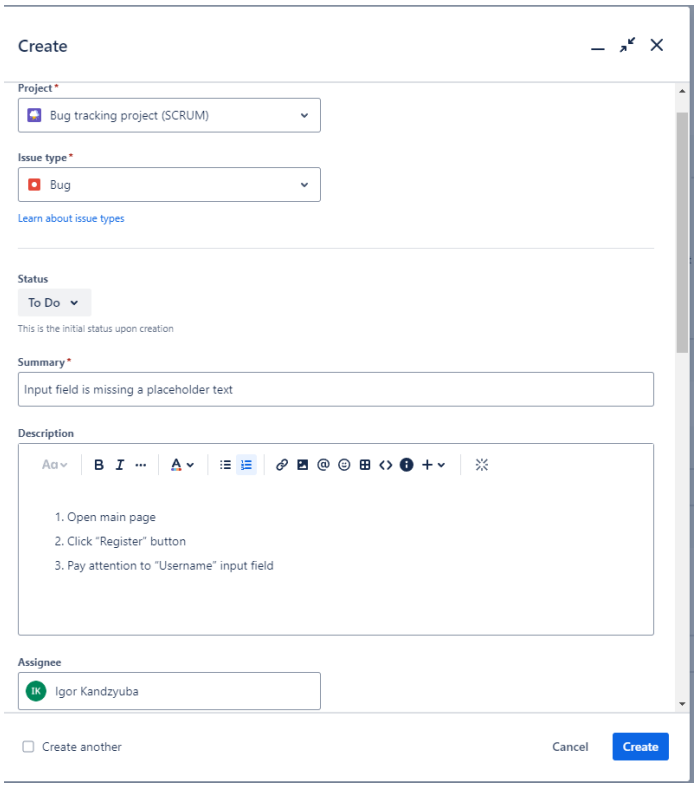

Рисунок 1.6 – Приклад створення звіту про помилку для проекту в Jira

Підтримка методологій управління проектами дає змогу командам ефективно планувати роботу, відстежувати прогрес і вчасно реагувати на зміни (рис. 1.7).

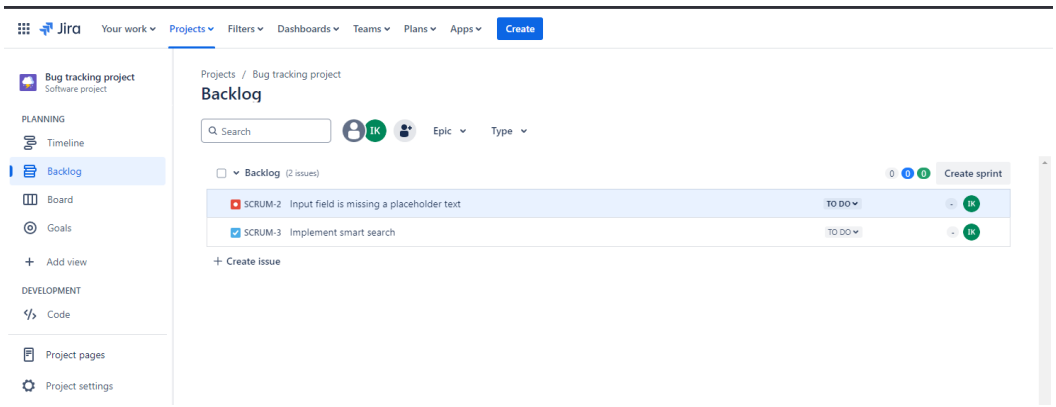

Рисунок 1.7 – Приклад сторінки проекту в Jira

Jira пропонує широкі можливості для аналітики і створення звітів, що допомагають керівникам команд і менеджерам проектів приймати обґрунтовані рішення.

GitHub Issues – це вбудована система відслідковування завдань і управління проектами, яка є частиною платформи GitHub. Вона дає змогу користувачам створювати, обговорювати й відстежувати помилки, а також визначати їхній пріоритет, планувати нові функціональні можливості та інші завдання в межах репозиторіїв GitHub.

Схожим чином до Jira користувачі можуть створювати завдання, які містять заголовок, опис, теги і можуть бути призначені конкретним членам команди (рис. 1.8). Завдання включають звіти про помилки, нові функції, запити на зміни або загальні задачі для проекту. Кожна задача може містити коментарі, які дозволяють членам команди обговорювати деталі, пропонувати рішення і обмінюватися ідеями.

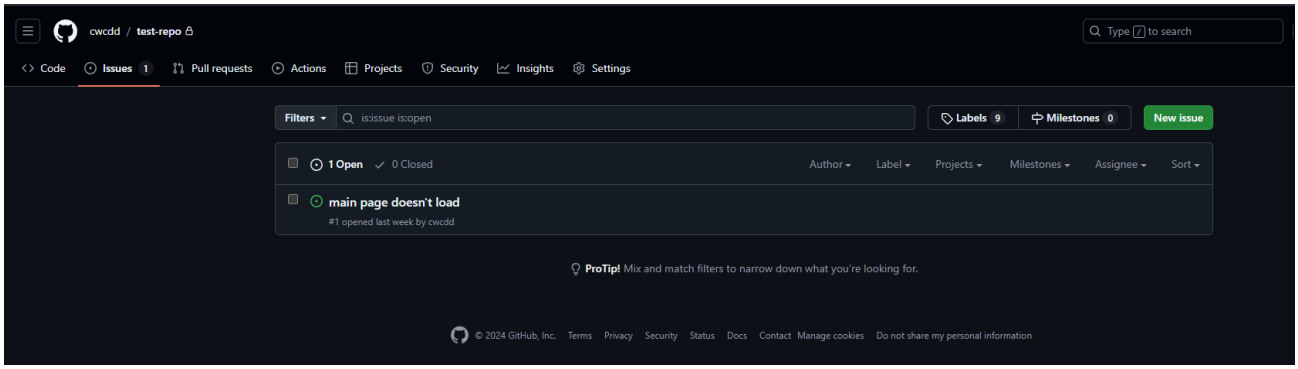

Рисунок 1.8 – Головна сторінка GitHub Issues зі створеним завданням

допомагають класифікувати й фільтрувати завдання за певними характеристиками, наприклад: bug, enhancement, help wanted тощо (рис. 1.9). конкретної мети чи релізу, що сприяє організації робочого процесу й відстеженню прогресу проекту.

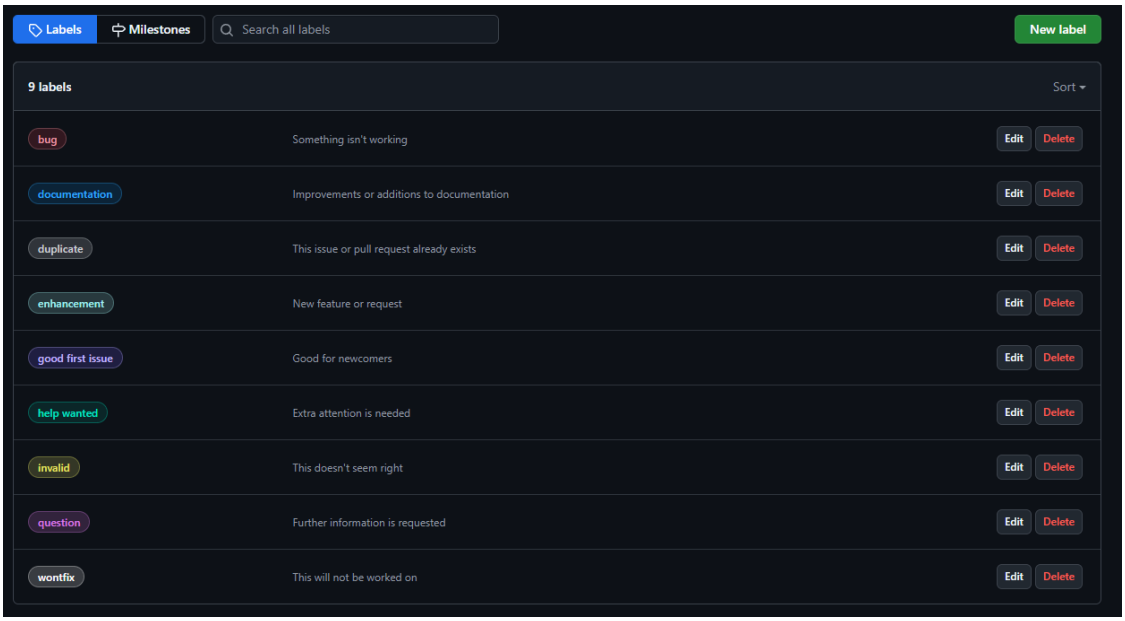

Рисунок 1.9 – Приклад меню Labels від GitHub Issues

Інструмент можна інтегрувати з проектними дошками для візуального управління завданнями у вигляді карток, що переміщуються між етапами роботи.

Створені помилки можуть бути безпосередньо пов’язані з Pull Requests, що дає змогу відстежувати, які зміни в коді вирішують конкретні завдання або помилки. Крім того, за допомогою GitHub Actions можна автоматизувати багато аспектів роботи з ними, наприклад, створення нових завдань або їх автоматичне закриття.

Як висновок, можна стверджувати, що всі застосунки, наведені у прикладах, мають набір функціоналу, який є сталим стандартом для систем відслідковування й управління ризиковими явищами. Можна виділити наступні можливості даних систем:

ідстеження помилок та завдань, з додаванням різноманітної інформації

татус та пріоритизація завдань;

оментарі та обговорення;

ільтрація та пошук;

віти та аналітика.

Отже, розроблюваний застосунок повинен реалізувати перелічений базовий функціонал, а також те, що зробить його унікальним проектом серед вже наявних систем. Наприклад, це може бути використання інтелектуальних алгоритмів для розподілу завдань між членами команди або машинного навчання для автоматичного визначення категорії, серйозності чи пріоритету помилок.

Далі проаналізуємо існуючі рішення хмарних платформ, щоб визначити стандарти для розробки власної системи.

– це популярна хмарна платформа як послуга (PaaS), що дозволяє розробникам легко впроваджувати, управляти та масштабувати застосунки без необхідності піклуватися про інфраструктуру. Heroku підтримує широкий спектр мов програмування, таких як Python, Ruby, Node.js, Java, PHP, Go тощо, і є дуже зручним для розгортання вебзастосунків та API [10].

Дана платформа працює на базі інфраструктури AWS. Іншими словами, Heroku використовує сервери, віртуальні машини, мережеві та інші обчислювальні ресурси, надані Amazon, для роботи власних сервісів.

Взагалі, Heroku відома можливістю розгортання застосунків з використанням системи контролю версій Git. Для розгортання достатньо виконати команду git push heroku main.

Також безперечним плюсом є підтримка популярних мов програмування та їх екосистеми, що дозволяє розробникам використовувати будь-яку з цих мов для розробки і розгортання застосунків.

Впроваджені проекти на Heroku можна легко масштабувати, додаючи нові , що є віртуальними контейнерами для запуску застосунків. Завдяки цьому ресурси можна оперативно збільшувати або зменшувати залежно від навантаження.

Heroku надає вбудовані інструменти для CI/CD, що дає змогу автоматично тестувати та розгортати нові версії застосунків.

– це хмарна платформа як послуга (PaaS) від AWS, яка дозволяє впроваджувати, управляти і масштабувати вебзастосунки та сервіси. Elastic Beanstalk підтримує різноманітні технології, такі як Java, .NET, PHP, Node.js, Python, Ruby, Go, Docker

тощо. Ця платформа автоматично управляє всіма аспектами інфраструктури, що дозволяє розробникам зосередитися на створенні та оптимізації застосунків [11].

Elastic Beanstalk також працює на основі інфраструктури AWS, оскільки це продукт самого Amazon. Він безпосередньо інтегрується з іншими їхніми сервісами, такими як Amazon EC2 (віртуальні машини), Amazon S3 (хмарне сховище), Amazon RDS (бази даних) тощо.

Платформа автоматично розгортає та керує інфраструктурою, включаючи сервери, мережі, балансування навантаження, масштабування та оновлення.

Вбудовані засоби моніторингу застосунків та інфраструктури дозволяють відстежувати їхню продуктивність, аналізувати лог-файли й отримувати сповіщення про події.

Проаналізувавши наявний функціонал у популярних PaaS рішеннях, можна вивести певний перелік функцій, які є необхідними для створення власної платформи.

По-перше, це автоматичне управління інфраструктурою. Платформа повинна автоматично розгортати, управляти і масштабувати інфраструктуру. Реалізація цього рішення дає змогу розробникам зосередитися на створенні функціоналу, мінімізуючи витрати часу на керування інфраструктурою.

По-друге, це масштабування за вимогами. Обидві платформи дозволяють масштабувати застосунки відповідно до навантаження, забезпечуючи можливість користувачам вручну регулювати кількість обчислювальних ресурсів для досягнення оптимальної продуктивності.

По-третє, підтримка контейнерів Docker. Elastic Beanstalk і Heroku мають вбудовану підтримку Docker-контейнерів, що дає змогу розробникам розгортати застосунки в контейнерах для забезпечення більшої гнучкості, узгодженості середовища та спрощення управління залежностями. Використання Docker дозволяє створити стандартизоване середовище для застосунку, яке легко перенести між різними платформами, мінімізуючи ризик конфліктів конфігурації.

Також важливо зазначити підтримку процесів автоматизації розробки й впровадження. Обидві платформи добре інтегруються з інструментами CI/CD,

що дозволяє автоматизувати процеси розгортання, тестування та оновлення застосунків.

Постановка задачі проектування платформи та інтелектуальної системи

Зважаючи на те, що проект складається з двох частин, а саме з платформи та вебзастосунку, то потрібно виділити функціонал для кожної з них окремо.

Основні вимоги до реалізації хмарної платформи:

нфраструктура – використання Oracle Cloud для створення віртуальної приватної хмари, яка буде основою для розробки вебзастосунку. Платформа повинна забезпечувати створення та управління основними компонентами інфраструктури, такими як віртуальні мережі (VCN), підмережі, групи безпеки та екземпляри віртуальних серверів (VPS);

онтейнеризація – використання Docker для створення, запуску та управління контейнерами на VPS, що забезпечує ізоляцію, портативність та ефективне розгортання застосунків;

інтеграція – платформа має підтримувати безперервну інтеграцію та розгортання, з можливістю роботи з сервісами, такими як GitHub або GitLab, для зберігання коду та автоматизації конвеєру розгортання

правління доступом, а саме механізми автентифікації та авторизації (OAuth, SSO), які забезпечують безпечний доступ до сервісів платформи;

асштабування за вимогами – підтримка можливості масштабування застосунків відповідно до поточного навантаження, що дозволяє користувачам самостійно адаптувати ресурси у разі потреби;

оніторинг та логування – впровадження інструментів для запису та аналізу журналів роботи платформи, що дозволяє відстежувати використання ресурсів та ідентифікувати потенційні проблеми.

Основні вимоги до реалізації вебзастосунку:

одавання та управління ризиковими явищами – можливість створення, редагування та пріоритизації помилок, а також їх прив’язка до конкретних членів команди та проектів;

нтелектуальна система розподілу завдань – автоматизоване призначення ризикових явищ на основі критеріїв, таких як складність, досвід розробника та поточна завантаженість;

налітичні дані – генерація звітів про продуктивність команди, статус помилок, час на виправлення тощо;

ласифікація ризикових явищ – застосування алгоритмів машинного навчання для автоматичної класифікації помилок на основі ключових слів, історичних даних та інших характеристик;

– використання технологій, як Node.js (Express) або Python (Django) для реалізації серверної логіки;

– React, Angular або Vue.js для створення інтуїтивного та динамічного SPA користувацького інтерфейсу;

береження даних – реляційні БД (PostgreSQL, MySQL) для зберігання даних про задачі, користувачів, звіти тощо;

ібридна архітектура – монолітний backend, та виділені сервіси, які потребують додаткового масштабування (наприклад, якщо навантаження на загальну серверну систему відносно невелике, окрім декількох точок доступу до API, то їх можна винести у окремі сервіси);

онтейнеризація сервісів – як вже було зазначено у описі хмарної платформи – використання Docker для ізоляції сервісів.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ХМАРНОЇ ПЛАТФОРМИ ТА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

Функціонал та структура хмарної платформи

Архітектура хмарної платформи базується на розподілі функціональності між кількома взаємопов'язаними модулями, які забезпечують ефективну роботу платформи, масштабованість і легкість у використанні. Кожен модуль відповідає за виконання певного завдання, що дозволяє дотримуватись принципів модульного програмування і спрощує тестування, розширення та підтримку системи.

При розробці хмарної платформи використання об'єктно-орієнтованого програмування стає основою для створення зрозумілої, модульної та легко керованої архітектури [12]. Завдяки принципу інкапсуляції кожен компонент платформи, наприклад, управління інфраструктурою, кластеризацією чи застосунками, реалізується як незалежна сутність із чітко визначеним інтерфейсом. Цей принцип дозволяє приховати складну внутрішню логіку роботи з OCI чи іншими зовнішніми системами, спрощуючи їх інтеграцію з іншими частинами платформи. Наприклад, взаємодія з мережами чи контейнерами може бути спрощена до використання публічних методів без знання специфіки API або деталей реалізації.

Поліморфізм у контексті платформи дозволяє уніфікувати роботу з різними ресурсами, такими як екземпляри віртуальних серверів, віртуальні мережі чи кластеризовані сервіси. Такий підхід уніфікує обробку ресурсів, що робить код більш гнучким і зручним для розширення. Наприклад, управління ресурсами здійснюється через загальні методи, які працюють із абстракціями, незалежно від конкретного типу об'єкта. Завдяки цьому можна легко додавати нові типи ресурсів, не змінюючи основну логіку роботи платформи.

Модульність, яку забезпечує ООП, сприяє організації платформи у вигляді окремих взаємопов’язаних компонентів, таких як менеджери для інфраструктури, кластерів чи застосунків. Обраний підхід спрощує тестування, рефакторинг і подальший розвиток системи, оскільки кожен модуль може працювати автономно. Наприклад, зміни в компоненті, що відповідає за балансувальники навантаження, не впливають на модулі, які займаються розгортанням застосунків чи управлінням мережами.

Проектування об’єктно орієнтованої архітектури виконано орієнтуючись на створену діаграму класів (рис. 2.1) у нотації UML [13].

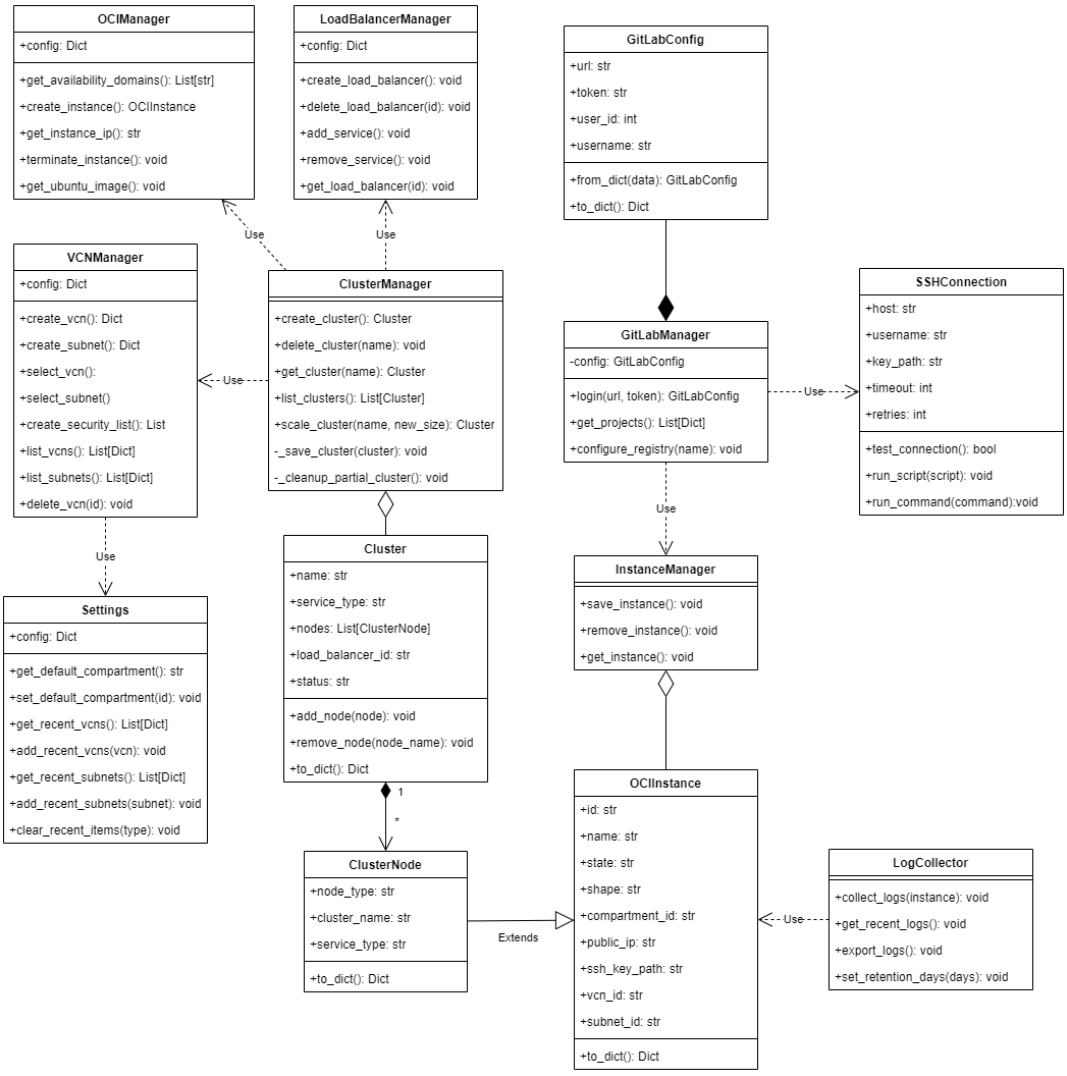

Рисунок 2.1 – Діаграма класів хмарної платформи

CLI-інтерфейс – основний інструмент взаємодії користувача з платформою, який забезпечує виконання основних команд. Він дозволяє уникнути складнощів роботи з вебінтерфейсом і зосередитись на виконанні команд через термінал. Запланований функціонал включає:

- інтерфейс повинен реалізовувати автентифікацію користувача за допомогою токену, який зберігається локально;
- управління інфраструктурою Oracle Cloud, а саме налаштування VCN, підмереж, створення та управління VPS, також налаштування балансувальників навантаження;
- розгортання застосунків, що включає завантаження вже створених Docker контейнерів з реєстру GitLab, їх налаштування та запуск;
- моніторинг стану застосунків: отримання інформації про статус контейнера, споживання ресурсів та помилки;
- управління застосунками: старт, зупинка, перезавантаження контейнерів і видалення проектів;
- оновлення контейнера, при завантаженні нового коду до GitLab.

Проектування архітектури хмарної платформи є ключовим етапом її розробки, оскільки від цього залежить її функціональність, стабільність, безпека та здатність адаптуватися до потреб користувачів. Особливо це важливо для платформи, яка поєднує інструменти для управління інфраструктурою, інтеграцію з DevOps-середовищем і розгортання застосунків. Ретельно продумана архітектура забезпечує стабільність, гнучкість і адаптивність розробки до змін.

Архітектура розроблена у вигляді трирівневої модульної структури, що включає ядро (Core), командний рівень (Command) і утиліти (Utilities). Завдяки такій архітектурі досягається чіткий розподіл функцій і можливість легкого масштабування чи додавання нових компонентів.

Модуль «core» виступає як ядро системи, надаючи абстракції для роботи з

через SSH. Модульний підхід забезпечує високу гнучкість у розширенні функціоналу, дозволяючи легко додавати нові компоненти або адаптувати існуючі до змінних вимог проекту. Крім того, централізованість критичної логіки в цій директорії сприяє підтримуваності системи.

Модуль «commands» відповідає за організацію обробників CLI-команд, кожен з яких інкапсулює бізнес-логіку для виконання певної функції. Винесення команд в окремий модуль забезпечує ізоляцію їхньої логіки від інших компонентів, що сприяє зручності тестування, підвищує зрозумілість коду та полегшує внесення змін.

Модуль «utils» містить допоміжні утиліти для виконання повторюваних методів у різних частинах системи. Організація коду в цій директорії підвищує його читабельність і структурованість, сприяє повторному використанню та спрощує рефакторинг і розширення функціоналу. Утиліти створені з акцентом на універсальність, що дає змогу застосовувати їх у різних модулях без потреби у модифікаціях.

Варто зазначити, що кожен модуль або клас виконує конкретну функцію. Наприклад, обробники команд CLI не займаються прямою взаємодією з OCI, а делегують ці завдання модулям із «core». Обраний підхід спрощує тестування та забезпечує зручність підтримки коду.

Використання архітектурних підходів, таких як абстракції над OCI SDK, дозволяє інтегрувати нові сервіси Oracle чи зовнішні API (наприклад, інших платформ CI/CD) без значного переписування існуючого коду. Модульна структура спрощує додавання нових CLI-команд чи функцій.

Інструмент використовує OCI SDK для створення та управління обчислювальними ресурсами, такими як екземпляри приватних серверів (VPS), мережі (VCN, підмережі, шлюзи) та SSH ключі [14]. Використання SDK дає можливість легко адаптувати інструмент до змін API або політик Oracle Cloud.

Для визначення можливостей користувача платформи спроектовано діаграму прецедентів у нотації UML (рис. 2.2).

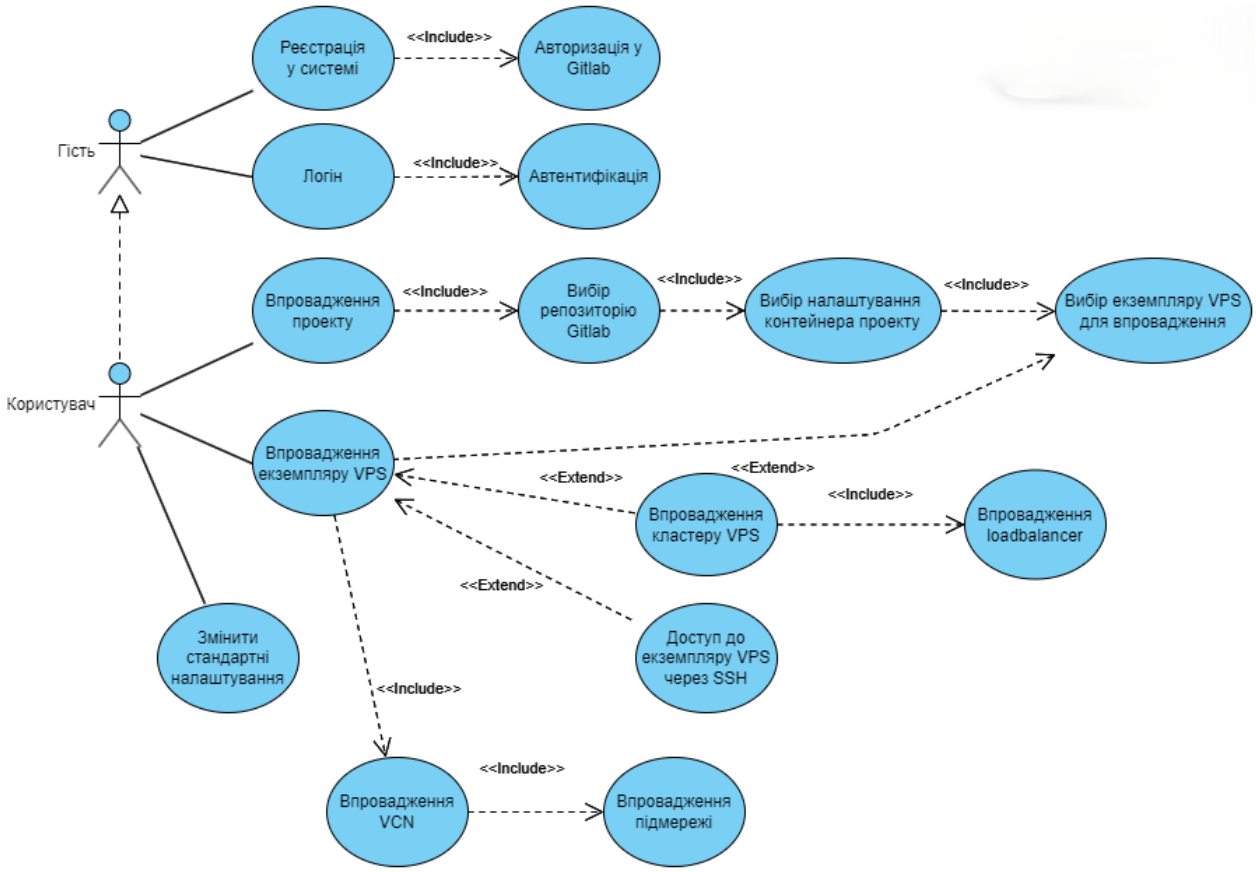

Рисунок 2.2 – Діаграма прецедентів для хмарної платформи

Діаграма відображає дві основні ролі: гість і користувач. Остання роль є розширеною та забезпечує доступ до всіх можливостей платформи.

Гостю доступний лише базовий функціонал, який включає:

- команди реєстрації та входу до системи;
- автентифікацію користувачів, інтегровану з OCI, що гарантує безпечне виконання операцій із хмарною інфраструктурою.

Авторизовані користувачі мають доступ до функціоналу управління інфраструктурою та застосунками, який включає наступні можливості:

- платформа дозволяє налаштовувати параметри OCI та локальної конфігурації інструменту CLI, наприклад облікові дані та регіони Oracle Cloud;

- розділ управління інфраструктурою охоплює функції створення, видалення та отримання інформації про ресурси OCI: обчислювальні екземпляри (VPS), мережі (VCN), шлюзи тощо;
- CLI також підтримує автоматизацію таких процесів, як генерація ключів для доступу до хмарних об’єктів і налаштування мережевого середовища;

- інтеграція з GitLab дозволяє налаштовувати з’єднання через API, перевіряти доступність реєстру контейнерів та автоматизувати завантаження Docker-образів із реєстру та підготовку до розгортання;
- команда «deploy» забезпечує управління обчислювальними екземплярами для розгортання застосунків. Процеси включають: встановлення Docker, завантаження образів із GitLab, запуск контейнерів та перевірку їх статусу;
- платформа веде журнали виконаних команд та обробляє помилки, що спрощує діагностику і забезпечує стабільність.

Хмарна платформа інтегрує функціонал через чітко визначені межі між шарами архітектури. Подібна структура дозволяє досягти високої гнучкості, масштабованості та простоти використання. Розподіл функцій між модулями сприяє ефективній роботі платформи, що орієнтована на потреби розробників і адміністраторів.

Проектування архітектури інтелектуальної системи

Сучасні вебзастосунки вимагають високого рівня гнучкості, продуктивності та здатності до масштабування, що накладає особливі вимоги до їх архітектурного проектування. У цьому контексті доцільним є використання парадигми багаторівневої архітектури N-tier, яка забезпечує чітке розмежування функціональності між рівнями, а також дозволяє адаптувати систему до зростаючих вимог користувачів [15]. У поєднанні з шаблоном проектування

MVC (Model-View-Controller), який структурує логіку всередині окремих рівнів, це створює основу для побудови ефективної, модульної та масштабованої системи.

архітектурна модель розділяє систему на декілька рівнів, кожен з яких виконує конкретні завдання (рис. 2.3). Основна мета такої структури – чіткий розподіл відповідальностей між частинами системи, що дозволяє легко масштабувати окремі компоненти, підтримувати їх незалежно і знизити ризик взаємного впливу.

Презентаційний рівень (Presentation tier) – відповідає за взаємодію з користувачем. Він приймає введені дані, відображає результати обробки і забезпечує зручний інтерфейс. У нашому випадку, цей рівень реалізований через вебінтерфейс, який надсилає запити до серверної частини системи.

Логічний рівень (Logic tier) – реалізує бізнес-логіку застосунку. Цей рівень обробляє запити від користувача, виконує перевірку введених даних, координує взаємодію між рівнями, а також взаємодіє зі спеціалізованим модулем для виконання аналітичних завдань. Логічний рівень також відповідає за прийняття рішень, такі як визначення пріоритетності завдань або їх критичності.

Рівень даних (Data tier) – відповідає за збереження інформації. У системі цей рівень реалізовано у вигляді баз даних, де зберігається вся інформація, пов’язана із задачами, їх атрибутами, статусами та результатами аналізу.

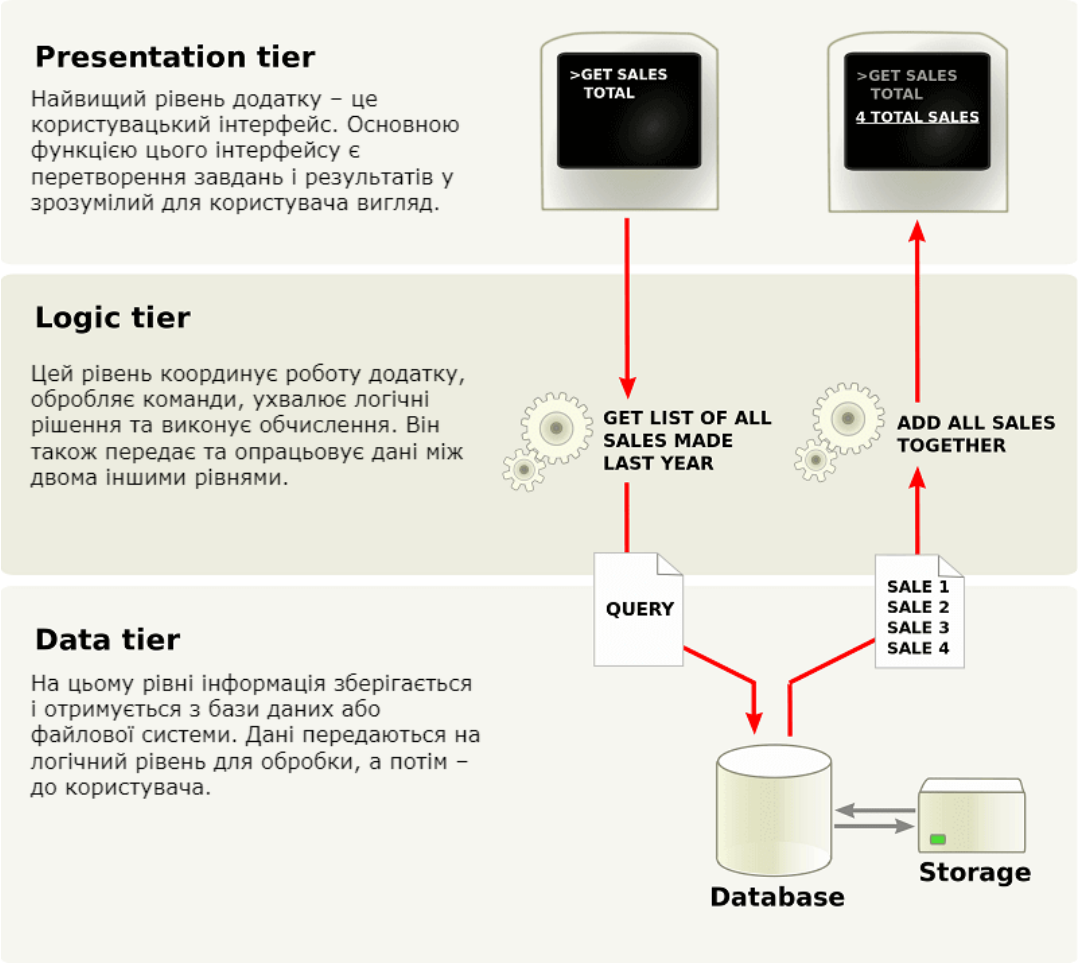

Рисунок 2.3 – N-tier архітектура на прикладі вебзастосунку

У складних системах може бути доданий інтеграційний або спеціалізований рівень для виконання вузькоспеціалізованих завдань (наприклад, обробка великих обсягів даних, штучний інтелект, інтеграція з іншими системами тощо).

Розподіл на рівні забезпечує кращу підтримуваність системи, оскільки зміни в одному рівні зазвичай не впливають на інші. Крім того, таке розмежування дозволяє використовувати різні технології для реалізації кожного рівня залежно від вимог до продуктивності, масштабованості чи інтеграції.

Далі буде розглянуто безпосередньо реалізацію MVC у контексті N-tier архітектури.

– це програмний архітектурний шаблон, який розділяє програму на три основні компоненти, кожен з яких відповідає за окремий аспект функціональності (рис.

Такий підхід спрощує процес розробки, тестування та масштабування програмного забезпечення.

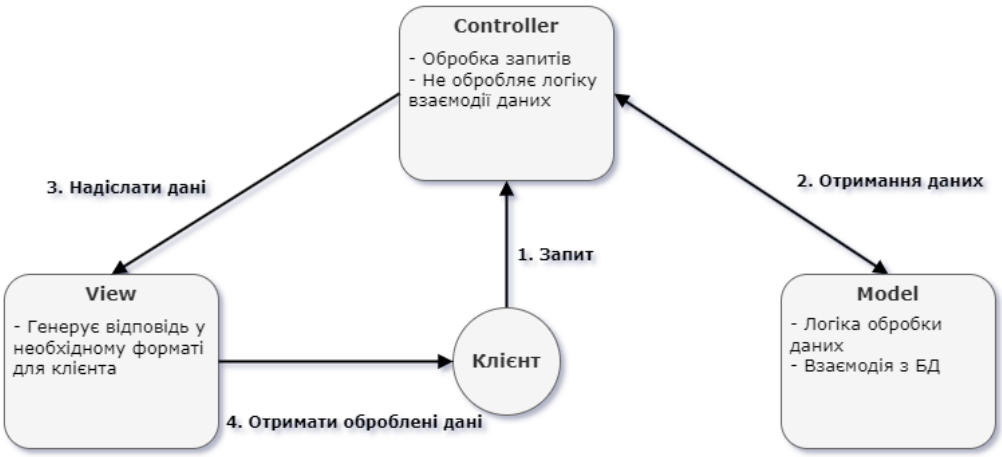

Рисунок 2.4 – Архітектурний шаблон MVC

є основним компонентом, який відповідає за управління даними, бізнес-логікою та правилами домену. Вона забезпечує абстракцію роботи з даними, включаючи їх отримання, збереження, модифікацію та перевірку. Модель визначає структуру даних (наприклад, їх атрибути і взаємозв'язки) та взаємодіє з рівнем даних (базами даних або іншими сховищами), виконуючи бізнес-логіку, необхідну для опрацювання інформації.

У контексті використання View на програмному рівні він може представляти форматований результат, який передається користувачеві у вигляді JSON, XML, HTML або іншого типу відповіді залежно від вимог клієнта. Його основна роль полягає у відокремленні процесу відображення інформації від бізнес-логіки. Цей компонент не виконує складних операцій, а лише отримує дані з моделі чи контролера та форматує їх у вигляді, зрозумілому клієнту (наприклад, вебзастосунку або іншій системі).

є сполучною ланкою між Model та View. Його основна функція полягає в обробці запитів від клієнта, виконанні відповідних бізнес-операцій через Model та формуванні відповіді через View. Контролер виконує логіку маршрутизації запитів, перевіряє вхідні дані, викликає необхідні методи моделі для виконання

операцій із даними і передає отримані результати для форматування у View. Він відповідає за координацію всіх компонентів, діючи як центральний елемент системи.

У даному випадку MVC реалізується на логічному рівні N-tier архітектури, допомагаючи структурувати API (рис. 2.5). Однак принципи даного шаблону можуть застосовуватися й на інших рівнях для організації логіки взаємодії між компонентами.

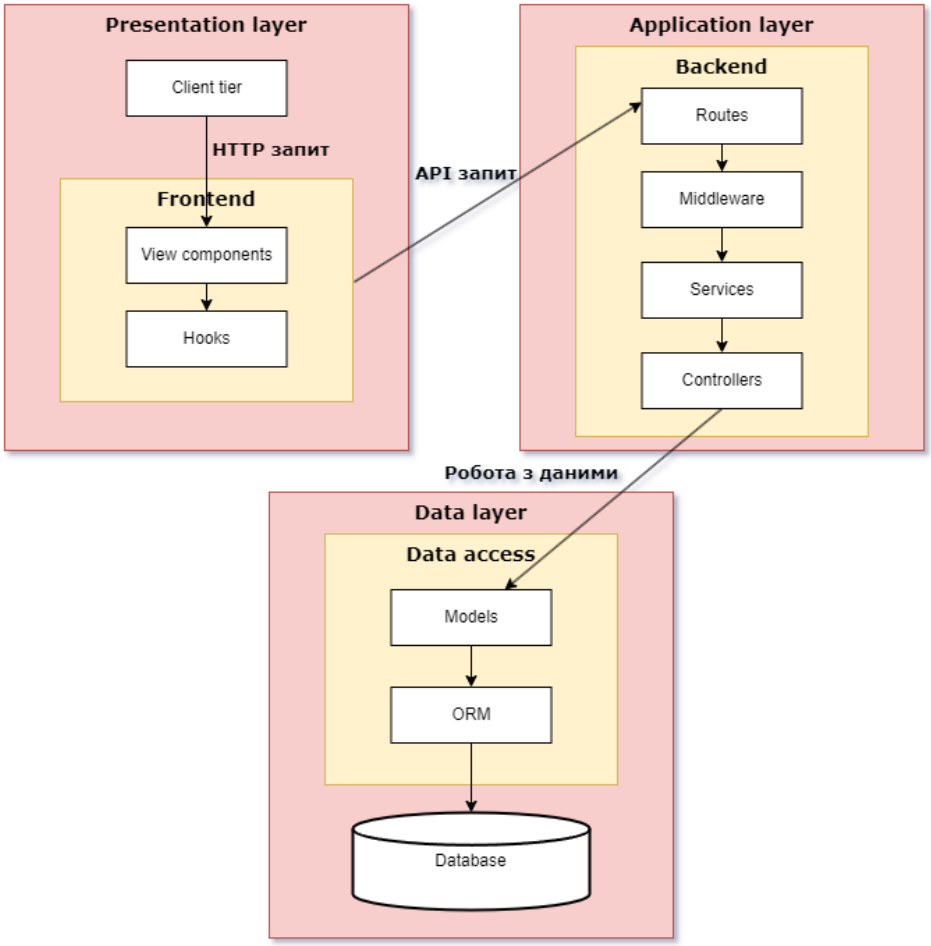

Рисунок 2.5 – Реалізація N-tier та MVC у контексті розроблюваної інтелектуальної системи

Обраний підхід забезпечує зручність у підтримці системи, дозволяє легко додавати нові функціональні можливості та тестувати окремі компоненти.

Далі детально розглянемо шаблон MVC безпосередньо у контексті розробки системи. По-перше, при розробці використовуючи MVC варто

приділити увагу моделі, адже саме вона є серцем архітектури і містить основну бізнес-логіку. Модель не тільки обробляє дані, а й відповідає за цілісність і консистентність даних, що особливо важливо при роботі з великою кількістю транзакцій або комплексними бізнес-процесами. Зазвичай, у складних системах модель містить кілька рівнів (наприклад, сервіси та репозиторії), що забезпечує додаткову гнучкість і зручність при тестуванні.

По-друге, контролер виконує роль посередника між користувачем і системою, приймаючи запити від користувача та формуючи відповіді, використовуючи модель і представлення. Важливо уникати надмірного об’єму коду, що може ускладнити підтримку. Доброю практикою є розподіл контролерів відповідно до конкретного функціоналу (наприклад, окремий контролер для управління проектами чи обробки звітів). Такий підхід спрощує навігацію в коді, полегшує тестування та забезпечує легке розширення застосунку.

По-третє, компонент представлення відповідає за відображення даних, отриманих від контролера. Важливо, щоб представлення не містило складної логіки, оскільки його основна мета – формувати зручний і зрозумілий інтерфейс для користувача. Більшість логіки має залишатися в моделі або контролері, щоб уникнути надмірної залежності інтерфейсу від змін у бізнес-логіці.

У контексті впровадження застосунку на основі архітектури MVC за допомогою хмарної платформи важливо враховувати її переваги та обмеження. MVC, як правило, є монолітною архітектурою, що забезпечує чітке розділення відповідальностей між компонентами, такими як модель, представлення та контролер. Однак масштабованість у такій структурі здебільшого реалізується вертикально, тобто за рахунок збільшення ресурсів сервера. Обране рішення ефективне для невеликих або середніх систем, але зі зростанням навантаження можуть виникати обмеження продуктивності через монолітну природу застосунку.

Використання хмарної платформи дозволяє частково вирішити ці обмеження за рахунок автоматизації управління ресурсами, зокрема масштабування та балансування навантаження. Хмарні сервіси, такі як

контейнеризація або оркестрація, можуть бути інтегровані для підвищення ефективності. Описаний підхід дозволяє не лише збільшувати продуктивність вертикально, а й забезпечувати горизонтальне масштабування, що є важливим для систем зі зростаючим навантаженням.

Для великих систем може знадобитися адаптація архітектури. Наприклад, можна застосувати гібридний підхід, де основний застосунок залишається монолітним і побудованим на основі MVC, але критично важливі або ресурсоємні функції виносяться в окремі сервіси. Такі сервіси можуть бути реалізовані як мікросервіси, що дозволяє їх незалежне масштабування та полегшує обробку високого навантаження.

Крім того, впровадження інтелектуальних алгоритмів для аналізу звітів про задачі вимагає значних обчислювальних ресурсів, які можна ефективно забезпечити за допомогою хмарних технологій. Наприклад, розміщення компонентів машинного навчання як окремих сервісів у хмарі дозволяє паралельно виконувати інтенсивні обчислення та інтегрувати результати з основною системою. Завдяки такому підходу можна поєднати простоту розробки та управління моноліту з гнучкістю та масштабованістю мікросервісів, що робить систему більш стійкою до змін і навантажень.

Для визначення основних можливостей користувача застосунку, була створена діаграма прецедентів у нотації UML (рис. 2.6).

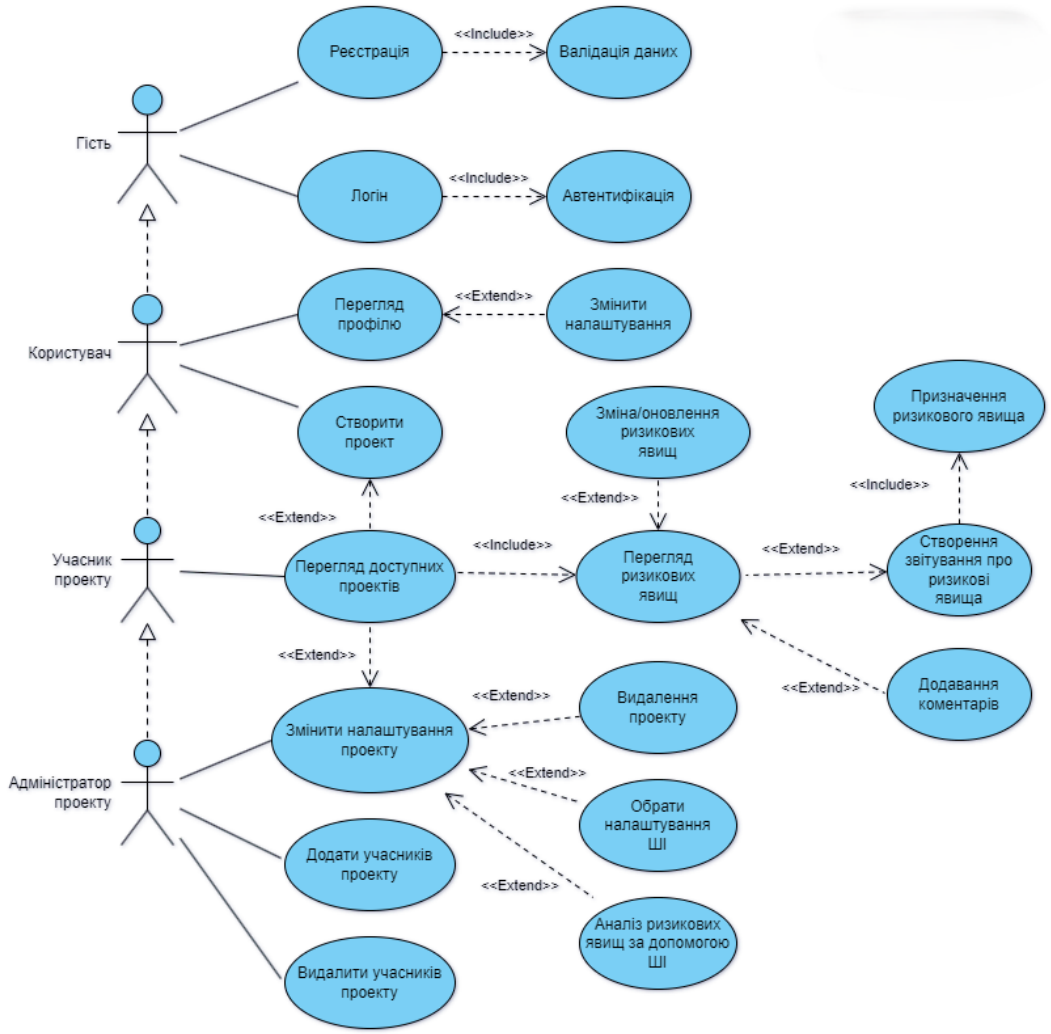

Рисунок 2.6 – Діаграма прецедентів інтелектуальної системи

Із наведеної діаграми видно, що реалізація передбачає чотири ролі: гість, користувач, учасник і адміністратор проекту. Кожна роль має власний набір можливостей, причому адміністратор проекту отримує найширший функціонал.

Гість має базові можливості, що включають:

- реєстрація – гість може зареєструватися на платформі, що включає валідацію даних під час реєстрації;
- логін – доступний функціонал авторизації, що включає автентифікацію.

Користувач натомість є більш розширеною роллю, яка наслідує функціонал гостя та має додаткові можливості:

- перегляд профілю – він може переглядати власний профіль і при необхідності змінювати налаштування та обрати бажані моделі ШІ;
- створення проекту – користувач може ініціювати створення нового проекту.

Учасник проекту є розширеною роллю користувача, яка надає доступ до певних функцій управління проектами та ризиковими явищами:

- переглядати доступні проекти – доступ до проектів, де користувач є учасником, також включає перегляд ризикових явищ з можливістю створення нових, або оновлення вже існуючих (де учасник є автором явищ).

Адміністратор проекту є найбільш привілейованою роллю, що наслідує можливості користувача та має розширений арсенал можливостей:

- налаштування проектів – адміністратор може змінювати параметри проектів, які включають автоматичне призначення задач та управління налаштуваннями ШІ;
- аналіз ризикових явищ за допомогою ШІ – адміністратор має можливість здійснювати аналіз ризикових явищ з використанням інтелектуальних моделей для отримання прогнозів чи оцінок ризику;
- видалення проекту – адміністратор може видаляти проекти за необхідності.

Проектування бази даних інтелектуальної системи

Проектування бази даних для інтелектуальної системи відслідковування й управління ризиковими явищами є ключовим етапом у забезпеченні ефективного зберігання, організації та швидкого доступу до даних. Розроблена архітектура бази даних реалізована за допомогою реляційної реляційні системи керування базами даних, тому розглянемо переваги використання даних систем саме у контексті впровадження у хмарному середовищі [16].

Реляційні системи керування базами даних є невід'ємною частиною у сучасних CI/CD процесах. Їхня роль полягає у забезпеченні стабільного зберігання даних, що є критичним для успішного виконання безперервної інтеграції та розгортання. Завдяки підтримці транзакцій реляційні бази даних гарантують цілісність інформації навіть у разі складних сценаріїв оновлення застосунків, що є типовим у CI/CD середовищах. Водночас вони пропонують інструменти для автоматизованого управління змінами, такі як механізми міграцій, які дозволяють підтримувати синхронність між схемою бази даних і застосунком у різних середовищах розгортання.

Основою роботи реляційних СКБД є ACID-властивості, які забезпечують атомарність, узгодженість, ізоляцію та довговічність транзакцій [17]. Завдяки цьому вдається уникнути втрати даних і зберегти їхню цілісність навіть при одночасному виконанні великої кількості операцій. Структура даних у таких базах організована у вигляді таблиць, між якими існують зв'язки через зовнішні ключі. Ця модель сприяє проектуванню складних і масштабованих систем, підтримуючи високий рівень нормалізації для уникнення надлишкових даних, а також забезпечує швидкий доступ до інформації завдяки індексації.

У сучасних хмарних середовищах часто виникає потреба в масштабуванні реляційних баз даних для обробки зростаючих обсягів даних і запитів. Масштабування може бути вертикальним, коли збільшуються апаратні ресурси одного сервера, або горизонтальним, що передбачає додавання нових серверів до системи. Горизонтальне масштабування реалізується через шардинг (sharding), реплікацію (replication) або кластеризацію (clustering). Реплікація, наприклад, дозволяє створювати копії бази даних на кількох серверах, забезпечуючи високу доступність даних та балансування навантаження. Шардинг, у свою чергу, розподіляє дані між різними серверами, що особливо ефективно при роботі з великими базами даних (рис. 2.7).

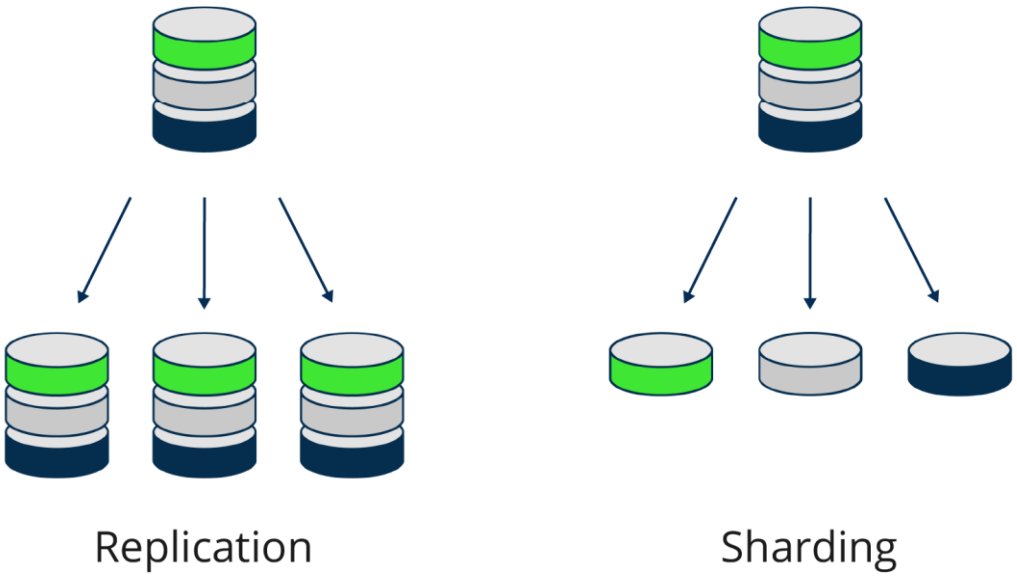

Рисунок 2.7 – Техніки масштабування баз даних

Окрім масштабування, важливу роль у підвищенні продуктивності відіграють такі техніки, як кешування та розділення на частини (partitioning). Кешування зменшує навантаження на базу даних, забезпечуючи швидкий доступ до часто використовуваних результатів запитів (рис. 2.8), тоді як розділення на частини структурує великі таблиці, полегшуючи їх обробку. Всі описані методи дозволяють реляційним базам даних залишатися ефективними навіть за умов постійного росту системи.

Більшість з наведених технік масштабування може реалізовувати хмарна платформа, за допомогою якої і планується впровадження Docker-контейнера бази даних.

Використання реляційних СКБД у середовищах постійної інтеграції та розробки супроводжується викликами, такими як конфлікти міграцій або проблеми продуктивності під високим навантаженням. Для їх вирішення застосовуються інструменти автоматизації міграцій, аналіз планів запитів та індексація. Завдяки цьому можна не лише уникнути збоїв у роботі бази даних, але й забезпечити швидке реагування на зміни в структурі застосунку.

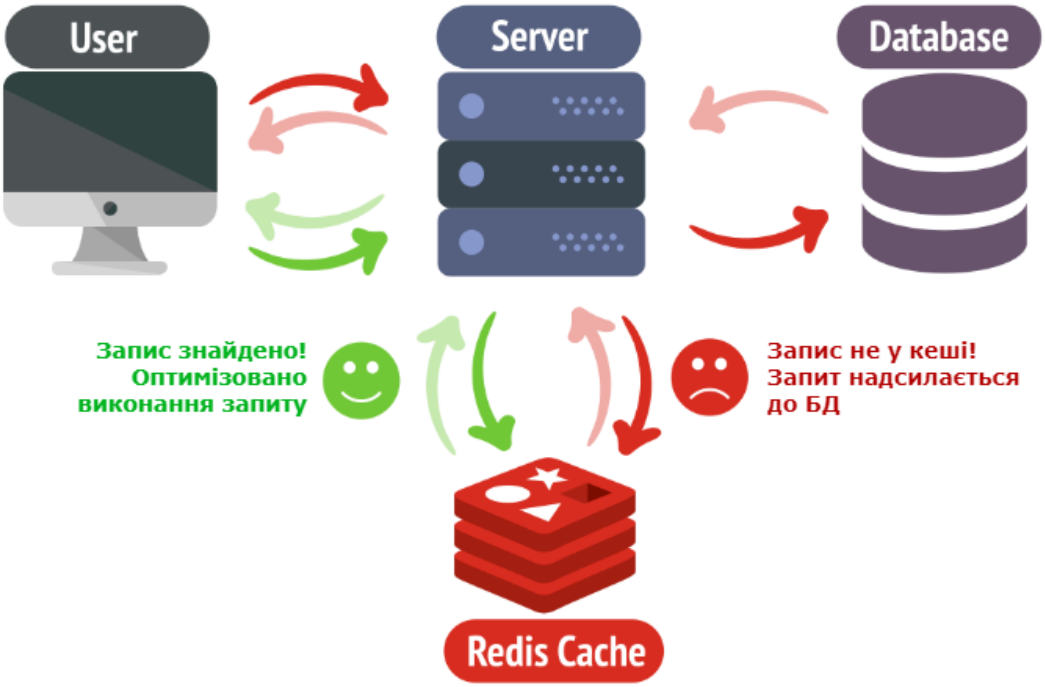

Рисунок 2.8 – Діаграма роботи кешування Redis

У результаті проектування потрібно визначати структуру таблиць та зв’язків між ними. Орієнтуючись на створену діаграму прецедентів для користувачів інтелектуальної системи у розділі 2.2, визначимо основі сутності майбутньої бази даних:

- моделювання користувачів та їх автентифікації – створення таблиці для зберігання інформації про користувачів, включаючи дані для автентифікації та верифікації;
- проекти та управління членами команди – структурування даних про проекти, учасників та їх ролі у системі;
- управління завданнями – створення таблиці для зберігання інформації про задачі, їх пріоритет, статус та зв’язок із користувачами;
- додаткові функції: коментарі та мультимедійні файли – забезпечення підтримки комунікації між користувачами через коментарі та управління вкладеннями для завдань.

Як один з найпопулярніших методів проектування, використано ER-діаграму (рис. 2.9) для визначення сутностей та їхніх зв’язків.

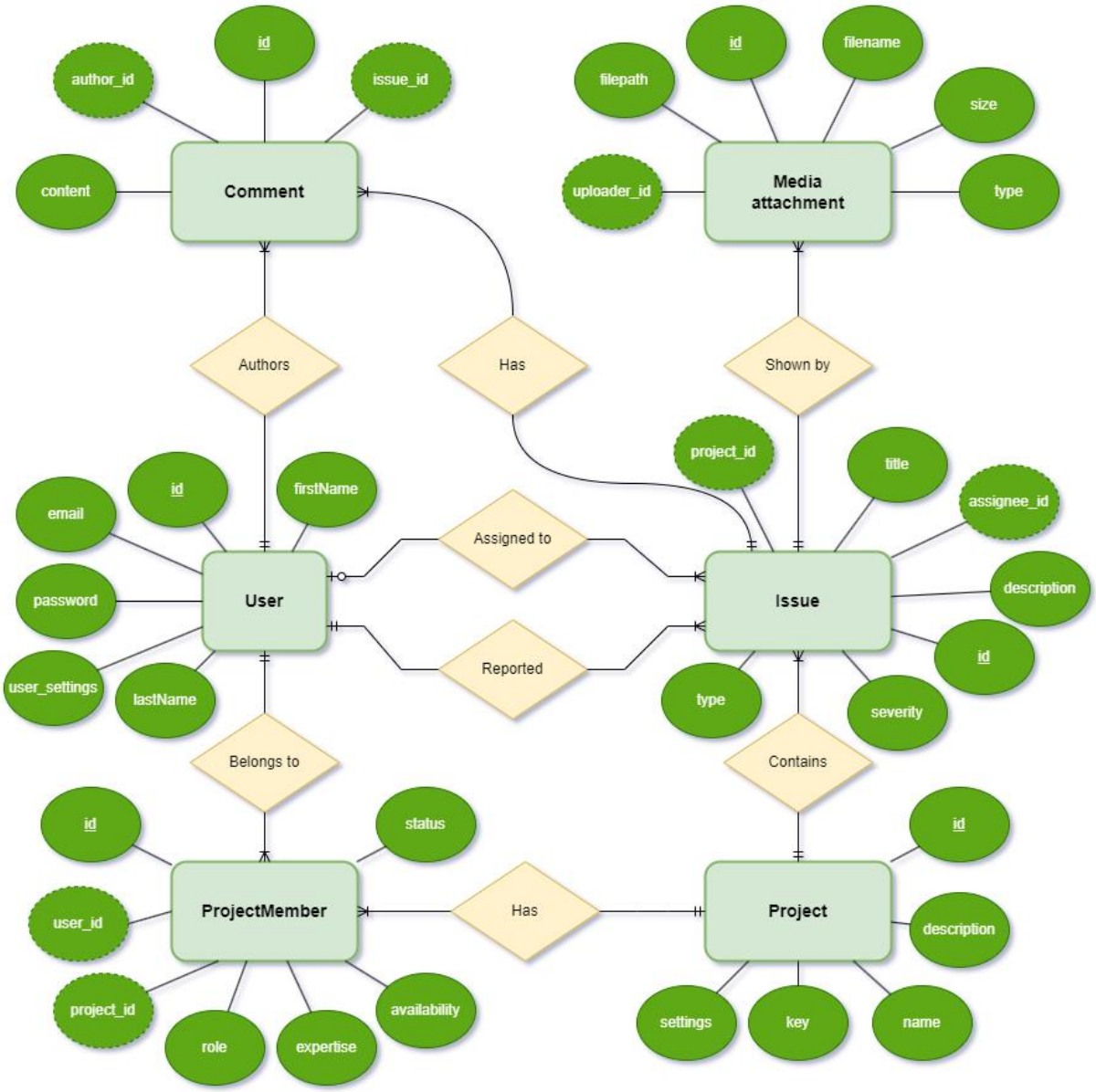

Рисунок 2.9 – ER-діаграма бази даних інтелектуальної системи

Проектування дизайну сторінок

Одним із ключових аспектів інтелектуальної системи відслідковування й управління ризиковими явищами є створення інтуїтивно зрозумілого та функціонального інтерфейсу користувача. Дизайн сторінок визначає не лише

естетичну складову вебзастосунку, а й значною мірою впливає на зручність використання та продуктивність кінцевих користувачів.

Процес проектування інтерфейсу базується на сучасних принципах UI/UX дизайну. Основна увага приділяється простоті використання та ефективності, що досягається завдяки використанню мінімалізму та лаконічності [18]. Усі елементи інтерфейсу є чіткими та зрозумілими, а надмірної деталізації або зайвих візуальних ефектів уникають відповідно до принципу KISS (Keep It

Усі сторінки системи спроектовані з урахуванням принципу узгодженості. Використовуються стандартизовані шаблони інтерфейсу та узгоджені кольорові схеми. Такий підхід допомагає користувачам легко орієнтуватися у системі, незалежно від того, на якій сторінці вони знаходяться.

Зміст сторінок організовано відповідно до принципу ієрархії, що допомагає користувачам швидко знаходити потрібну інформацію. Ключові елементи виділяються заголовками вищого рівня, менш важливі дані структуруються підзаголовками та списками [19]. Візуальні підказки, такі як кольорові акценти та маркери, додатково спрощують орієнтацію на сторінці.

Усі інтерактивні елементи, зокрема кнопки, випадаючі списки та індикатори прогресу, спроектовано відповідно до принципів Jakob Nielsen’s (рис. 2.10). Особлива увага приділяється миттєвому зворотному зв’язку для користувачів, наприклад повідомлення про успішне виконання дій або сповіщення про помилки. Обраний підхід зменшує ймовірність помилок і підвищує довіру до системи.

Головна сторінка включає панель моніторингу з основними метриками: кількістю активних завдань, середнім часом їх виконання та розподілом задач між розробниками. Елементи організовано за принципом пріоритизації інформації, де найбільш важлива інформація розташована зверху.

Рисунок 2.10 – Принципи дизайну Jakob Nielsen's Usability Heuristics

Сторінка проектів містить організований у вигляді карток або таблиці, які відображають базову інформацію (назва, статус, відповідальний). Для зручності доступу передбачено можливість пошуку та фільтрації.

Сторінка завдань передбачає список звітів про ризикові явища або задачі із можливістю фільтрації за статусом, пріоритетом та серйозністю. Інформація відображається у форматі табличного представлення даних, що забезпечує структурованість і зручність використання.

Сторінка створення задач містить зрозумілу форму з вертикальним розташуванням полів та чіткими підписами. Обов'язкові поля виділяються спеціальними позначеннями, а у випадку помилки система надасть відповідні повідомлення.

Сторінка аналітики містить інтерактивні діаграми та графіки, створені за допомогою бібліотеки Chart.js [20]. Графіки оснащені чіткими підписами, інформативними легендами та можливістю налаштовувати параметри відображення.

Адаптивний дизайн потрібно врахувати для всіх сторінок вебзастосунку. Обраний підхід забезпечує коректну роботу інтерфейсу на пристроях із різними розмірами екранів. Наприклад, на широкоформатних екранах елементи аналітичних панелей розташовуються в кілька колонок, тоді як на мобільних

пристроях вони відображаються у вигляді послідовного списку. Також передбачено приховування неважливих елементів на малих екранах для збереження зручності та концентрації користувачів на ключових задачах.

Для підвищення ефективності та комфортності використання платформи передбачено розширений зворотний зв’язок. Наприклад, під час виконання критичних дій, таких як створення задачі або видалення проекту, користувач отримує як текстове підтвердження, так і візуальні анімації. Це дозволить уникнути непорозумінь та помилок. Крім того, для критичних подій, як-от завершення тривалих операцій, передбачено оповіщення у вигляді звукових сигналів. Такий підхід підвищує довіру до системи та сприяє покращенню користувацького досвіду.

Проектування дизайну сторінок з урахуванням згаданих стандартів забезпечує інтуїтивну взаємодію користувачів із платформою, зменшує криву навчання та підвищує продуктивність роботи.

РОЗДІЛ 3
РОЗРОБКА ХМАРНОЇ ПЛАТФОРМИ ТА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

3.1 Вибір платформи та технологій хмарної платформи

Розробка хмарної платформи базується на принципах автоматизації, гнучкого управління ресурсами та інтеграції сучасних інструментів DevOps. Вибір технологій здійснювався з урахуванням функціональних вимог проєкту, серед яких автоматизоване управління контейнерами, інтеграція з GitLab для оптимізації CI/CD-процесів, моніторинг і логування, а також автоматична конфігурація мережевої інфраструктури та підтримка SSL-з'єднань.

У якості хмарного провайдера обрано Oracle Cloud через низку вагомих переваг. Основними факторами стали наявність безкоштовного тарифного плану, який забезпечує достатні обчислювальні та мережеві ресурси для ефективної розробки та тестування, а також розвинений API, що дозволяє інтегрувати автоматизацію через SDK. Крім того, OCI пропонує підтримку контейнеризації, що є ключовим аспектом сучасної розробки, та інтеграцію з інструментами оркестрації застосунків. Значною перевагою також є комплексний набір сервісів для управління мережею, моніторингу та забезпечення безпеки, які відіграють критичну роль у створенні хмарної платформи, що відповідає сучасним вимогам стабільності та захисту даних.

Задля автоматизації розгортання обрано GitLab CI/CD завдяки його потужним можливостям створення конвеєрів (GitLab pipelines), які значно спрощують і прискорюють процес впровадження оновлень. Обрана платформа забезпечує інтегроване середовище, що поєднує зберігання коду та автоматизацію збірки застосунків. Обраний підхід дозволяє мінімізувати складність інфраструктури, зменшуючи залежність від сторонніх інструментів, та сприяє оптимізації процесів розробки й тестування.

Основою розробки став консольний застосунок, створений мовою Python, обраною за її гнучкість і широкий набір бібліотек, які суттєво спрощують автоматизацію завдань. Використання бібліотек Click, OCI SDK та PyYAML дозволило реалізувати зручний командний інтерфейс, інтеграцію з хмарною інфраструктурою Oracle та гнучке управління конфігураціями. Click виступає основним інструментом для створення інтерактивного консольного інтерфейсу, забезпечуючи ефективну взаємодію користувачів із платформою [21].

Контейнеризація реалізована на основі Docker, який гарантує ізоляцію застосунків, спрощує їх розгортання та масштабування [22]. Docker Daemon налаштовано на екземплярах Oracle VPS, що дозволяє автоматизувати розгортання та управління контейнерами. Для обробки вхідних запитів, маршрутизації, балансування навантаження та SSL-термінації використовується Nginx, який забезпечує стабільну роботу сервісного контейнера та його захищеність.

Управління інфраструктурою Oracle здійснюється за допомогою бібліотеки OCI SDK, яка є ключовим інструментом у розробці хмарної платформи. Вона надає змогу автоматизувати створення й управління ресурсами, такими як екземпляри VPS, мережі та групи безпеки, забезпечуючи інтеграцію програмної логіки платформи з апаратними можливостями Oracle Cloud. Також використання OCI SDK дозволяє реалізувати динамічне масштабування інфраструктури, що забезпечує автоматичне додавання або видалення обчислювальних ресурсів залежно від навантаження.

Зберігання й отримання контейнерів реалізовано за допомогою інтеграції платформи з реєстром контейнерів GitLab, що підтримує автоматизацію процесів CI/CD [23]. Завдяки цьому забезпечується безперервність роботи системи та можливість автоматичного розгортання оновлень контейнерів у обраному середовищі.

Для моніторингу та аналізу логів системи використовується комплекс інструментів, який включає Elasticsearch та Filebeat. Filebeat відповідає за

надійний збір і передачу логів з усіх компонентів системи до Elasticsearch, де вони централізовано зберігаються та індексуються для швидкого пошуку.

Завдяки інтеграції цих інструментів платформа забезпечує комплексний підхід до роботи з логами, дозволяючи ефективно відстежувати стан системи, виявляти й діагностувати проблеми, а також аналізувати тренди в роботі застосунків і всієї інфраструктури.

З метою підвищення рівня безпеки платформи використовується Let's Encrypt, що забезпечує автоматизоване отримання й оновлення SSL-сертифікатів [24]. Це не лише спрощує процес керування сертифікатами, а й гарантує відповідність сучасним стандартам захисту даних.

Обрані технології для хмарної платформи забезпечують автоматизацію, гнучкість і стабільність її роботи, відповідаючи сучасним вимогам DevOps. Інтеграція з Oracle Cloud, Docker, GitLab CI/CD та інструментами моніторингу дозволяє ефективно керувати інфраструктурою, забезпечуючи масштабованість і високий рівень безпеки.

Вибір платформи та технологій інтелектуальної системи

Ефективна реалізація інтелектуальної системи потребує ретельного вибору платформ і технологій, які забезпечують високу продуктивність, надійність та масштабованість. У цьому підрозділі буде розглянуто процес прийняття рішень щодо вибору технологій для розробки клієнтської частини, серверної частини, а також інтеграції інтелектуальних функцій.

Технології клієнтської частини. Розробка інтелектуальної системи завжди починається з побудови основи, яка забезпечує зручність та ефективність взаємодії з користувачем. У цьому контексті клієнтська частина відіграє ключову роль, адже саме вона формує перше враження про застосунок і забезпечує інтерактивність. Для її реалізації в проекті обрано React – популярну JavaScript-бібліотеку, яка надає потужний інструментарій для створення динамічних,

інтерактивних і масштабованих вебзастосунків. Завдяки компонентно-орієнтованому підходу, React спрощує розробку модульних і багаторазових блоків інтерфейсу, що полегшує процес тестування та підтримки. Бібліотека дозволяє ефективно реалізовувати навіть складну логіку, забезпечуючи високу продуктивність і зручність роботи з застосунком [25].

React дозволяє створювати повторно використовувані компоненти, які інкапсулюють логіку, розмітку й стиль. Такий підхід сприяє модульності коду, полегшує його масштабування та забезпечує зручність повторного використання компонентів у різних частинах проекту.

Однією з ключових технологій даної бібліотеки є віртуальний DOM, який забезпечує високу продуктивність застосунків. На відміну від традиційного DOM, де кожна зміна вноситься безпосередньо у структуру сторінки, React працює з віртуальним представленням DOM у пам’яті. Зміни спочатку застосовуються до віртуального DOM, після чого бібліотека обчислює мінімальні зміни, необхідні для синхронізації з реальним (рис. 3.1). Обраний підхід значно скорочує ресурсоємні маніпуляції, покращуючи швидкодію та забезпечуючи плавну роботу інтерфейсу користувача.

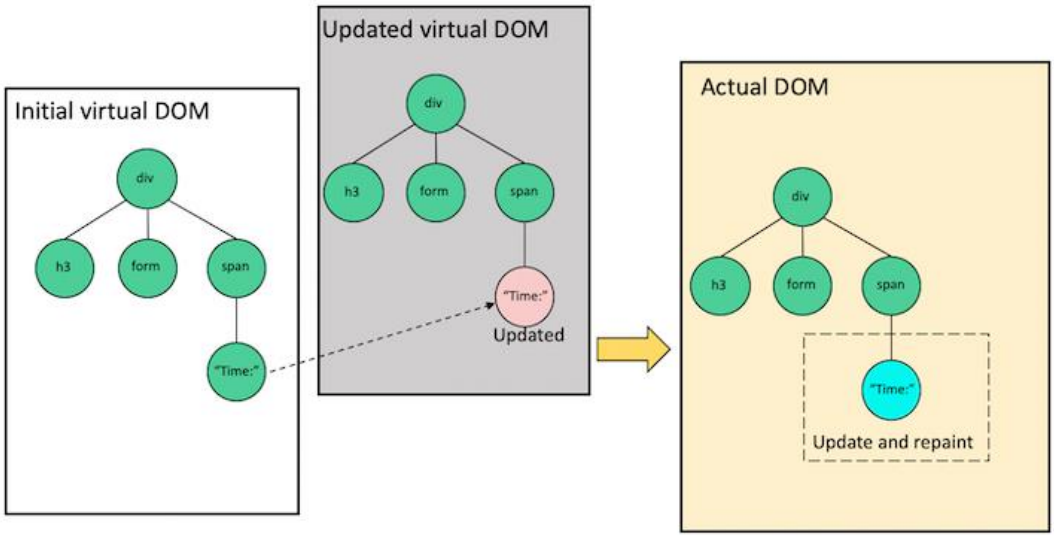

Рисунок 3.1 – Принцип роботи Virtual DOM

Дана бібліотека використовує односпрямований потік даних, що означає передачу інформації від батьківського компонента до дочірніх. Такий підхід гарантує передбачувану поведінку застосунку, що сприяє спрощенню його налагодження та підтримки.

React легко інтегрується з такими бібліотеками та інструментами, як Redux для керування станом і React Router для маршрутизації, що дає змогу розширювати функціональність застосунку відповідно до вимог.

Замість створення окремих файлів для розмітки, логіки та стилів, React використовує JSX (JavaScript XML), який дає змогу писати HTML-подібний код у JavaScript. Такий підхід підвищує зручність роботи над інтерфейсом, спрощуючи поєднання логіки та структури елементів.

Використання Tailwind CSS у поєднанні з React спрощує розробку сучасного інтерфейсу користувача. Tailwind – це CSS-фреймворк, заснований на утилітарних класах, що дозволяє стилізувати компоненти без написання власного Його використання дозволяє швидко створювати прототипи та забезпечувати високий рівень налаштування стилів, зберігаючи узгодженість дизайну завдяки попередньо визначеним класам.

Tailwind також чудово поєднується з компонентним підходом React, дозволяючи створювати стильові рішення безпосередньо у JSX-коді компонентів. Наприклад, компоненти з певними стилями можна легко повторно використовувати, змінюючи лише параметри для їхнього налаштування.

Важливо зазначити, що React активно підтримується спільнотою та компанією Meta, що гарантує регулярні оновлення, виправлення помилок і стабільний розвиток. Завдяки своїй популярності, дана бібліотека має величезну базу навчальних матеріалів, плагінів та бібліотек, що значно спрощує процес розробки. У поєднанні з Tailwind, React дозволяє створювати сучасний, продуктивний і адаптивний інтерфейс користувача, задовольняючи потреби найвибагливіших застосунків.

Технології серверної частини. Розробка логічної частини є ключовим етапом

створення будь-якого сучасного вебзастосунку, адже саме вона відповідає за обробку запитів, управління даними та взаємодію між клієнтською і серверною частинами. У даному проекті backend реалізовано за допомогою технологій Node.js, Express та Sequelize. Такий підхід забезпечує високу продуктивність, спрощує побудову RESTful API та дозволяє ефективно працювати з реляційною базою даних. Обрані технології сприяють створенню гнучкої, масштабованої та надійної серверної архітектури.

Node.js є ключовою технологією для серверної частини проекту завдяки своїй високій продуктивності, побудованій на основі неблокуючої моделі введення-виведення та асинхронної архітектури [27]. Ці характеристики дозволяють ефективно обробляти численні одночасні запити без необхідності створення великої кількості потоків. Особливістю використання Node.js є робота з однопотоковою моделлю, яка керується подіями (event-driven architecture), що забезпечує швидкий відгук системи навіть за умов високого навантаження (рис. Водночас цей підхід вимагає ретельного управління асинхронними операціями для уникнення проблем, таких як «callback hell». Для вирішення цієї проблеми у проекті активно використовуються promise і async/await.

Express.js, як мінімалістичний фреймворк для Node.js, спрощує створення серверної логіки. Він забезпечує швидку й гнучку реалізацію маршрутизації, що лежить в основі RESTful API. Express.js підтримує «middleware», що дає змогу обробляти запити на різних етапах, наприклад, для перевірки автентифікації, логування або валідації даних. У межах проекту ці можливості використовуються для забезпечення безпеки (додаткові обробники для автентифікації та авторизації) та підтримки модульності коду.

Рисунок 3.2 – Event-driven архітектура Node.js

Взаємодія з реляційною базою даних PostgreSQL реалізована за допомогою Sequelize, яка виступає як ORM [28]. Замість написання складних SQL-запитів розробник працює з об’єктами, що представляють таблиці бази даних, що підвищує швидкість розробки, а також покращує читабельність і підтримуваність коду. Нюансами роботи з Sequelize є налаштування моделей даних, які включають визначення полів, їх типів та зв’язків між таблицями. Для забезпечення цілісності даних використовуються вбудовані механізми валідації та обмежень на рівні моделей. Крім того, Sequelize підтримує міграції, що дозволяє керувати змінами структури бази даних централізовано та впорядковано.

RESTful API є ключовим механізмом взаємодії між клієнтською та серверною частинами. У проекті реалізовано ключові точки, що відповідають CRUD-операціям: створення (Create), читання (Read), оновлення (Update), видалення (Delete).

Нюанси реалізації RESTful API включають правильне використання HTTP-методів (GET, POST, PUT, DELETE тощо) і статус-кодів (наприклад, 200 «OK»,

узгодженості API використовуються JSON як формат обміну даними та стандартизовані відповіді сервера, що містять ключові поля: статус, повідомлення, дані.

Особливу увагу приділено обробці помилок. Наприклад, у разі некоректного запиту сервер повертає детальне повідомлення про помилку з поясненням причини відхилення, що значно скорочує час, потрібний для тестування та інтеграції, і покращує користувацьку взаємодію з застосунком.

Як підсумок, Node.js, Express.js та Sequelize, у поєднанні з RESTful API, забезпечують ефективну, модульну та масштабовану серверну архітектуру, яка відповідає вимогам сучасних вебзастосунків і дозволяє інтегрувати аналітичні модулі, наприклад на основі штучного інтелекту, без значних ускладнень.

Технології сервісу штучного інтелекту. Python обрано мовою для створення сервісу штучного інтелекту завдяки своїй простоті, універсальності та широкій екосистемі бібліотек. У контексті розробки сервісу штучного інтелекту вона забезпечує зручні інструменти для роботи з даними, розробки моделей машинного навчання та їхньої інтеграції у вебзастосунок.

Однією з ключових переваг мови Python є її багатий набір бібліотек, серед яких важливу роль відіграють scikit-learn, imbalanced-learn та FastAPI. Бібліотека scikit-learn дозволяє швидко створювати, налаштовувати і навчати моделі машинного навчання, надаючи широкий вибір алгоритмів і зручних інструментів для їхньої оптимізації. Бібліотека imbalanced-learn незамінна для роботи з незбалансованими наборами даних, оскільки надає ефективні методи, такі як SMOTE, підвибірки (undersampling) та інші підходи для збалансування класів, що покращують якість моделі й допомагають уникнути перенавчання. Водночас FastAPI пропонує потужний інструментарій для створення високопродуктивних і масштабованих REST API із вбудованою підтримкою асинхронності, автоматичною генерацією документації та валідацією даних на основі анотацій типів.

Використання Python у поєднанні з FastAPI дозволяє ефективно створювати вебсервіси, здатні обробляти вхідні дані, інтегрувати передбачення моделей машинного навчання та забезпечувати ключові функціональні можливості, наприклад такі як валідація вхідних даних, логування операцій і обробка помилок.

У проекті Python використовується як основна мова для побудови сервісу, що включає завантаження навчених моделей, виконання прогнозів і аналіз даних ризикових явищ. Використання FastAPI забезпечує швидку інтеграцію моделей машинного навчання у вебзастосунок, дозволяючи створювати високопродуктивний сервіс із гнучкими можливостями масштабування та інтерактивною документацією API. Таким чином, Python виступає універсальним інструментом, який поєднує можливості розробки штучного інтелекту та вебтехнологій.

При вирішенні задачі класифікації характеристик дефектів програмного забезпечення, які включають визначення пріоритету, типу та критичності, вибір алгоритму Random Forest є виправданим завдяки його універсальності та продуктивності. Обраний алгоритм особливо ефективний у багатокласових задачах, оскільки підтримує роботу з гетерогенними даними (як текстовими, так і числовими) та враховує дисбаланс класів [29]. Алгоритм добре масштабується на великих наборах даних завдяки паралельній обробці, а також стійкий до шуму, що є важливим при роботі з реальними даними, які можуть бути неповними або містити помилки.

У порівнянні з іншими алгоритмами (табл. 3.1), наприклад Gradient забезпечує швидше навчання, що є важливим для великих обсягів даних, хоча Gradient Boosting може досягати вищої точності на складних задачах. На відміну від ExtraTrees, який працює швидше, Random Forest має кращу точність і є більш гнучким у налаштуванні.

Методи на кшталт SVM показують високу точність на малих наборах даних, але погано масштабується для великих задач, які характерні для цього проекту.

KNN, хоча й простий, є неефективним для великих наборів через повільність класифікації.

Завдяки своїй ансамблевій структурі, здатності працювати з багатокласовими задачами, інтегрувати різні типи даних (текстові та числові), а також уникати перенавчання, Random Forest ідеально підходить для обробки текстових даних і прогнозування характеристик дефектів у запропонованій системі.

Таблиця 3.1 – Порівняння алгоритмів класифікації

Алгоритм	**Переваги**	**Недоліки**
	Баланс точності, швидкості та інтерпретації	Великі набори дерев мають значні вимоги до пам’яті
	Вища точність на складних даних	Повільніше навчання
	Швидкість навчання	Трохи нижча точність
	Висока точність на невеликих наборах даних	Не масштабується для великих даних
	Простота	Повільна класифікація на великих даних

Принцип роботи алгоритму базується на концепції ансамблевого навчання, яка передбачає використання кількох моделей для вирішення однієї задачі. У випадку Random Forest цією базовою моделлю є дерево ухвалення рішень. Алгоритм створює великий набір дерев рішень, які навчаються на різних підмножинах даних, і об’єднує їхні результати для підвищення точності та стійкості моделі.

На етапі навчання алгоритм генерує випадкові підмножини даних із початкового набору (метод бутстрепінгу), де кожна підмножина формується із заміною. Це означає, що деякі приклади можуть повторюватися, а інші –

залишатися поза підмножиною. Далі для кожного дерева обирається випадкова підмножина ознак, яка використовується для розбиття вузлів дерева. Такий підхід, відомий як випадкове відбирання ознак, сприяє зменшенню кореляції між деревами, що підвищує загальну продуктивність ансамблю.

Кожне дерево приймає рішення самостійно, класифікуючи приклад до певного класу. У процесі передбачення алгоритм об'єднує результати окремих дерев через механізм голосування: клас, який отримав найбільшу кількість голосів, визначається як кінцевий результат (рис. 3.3).

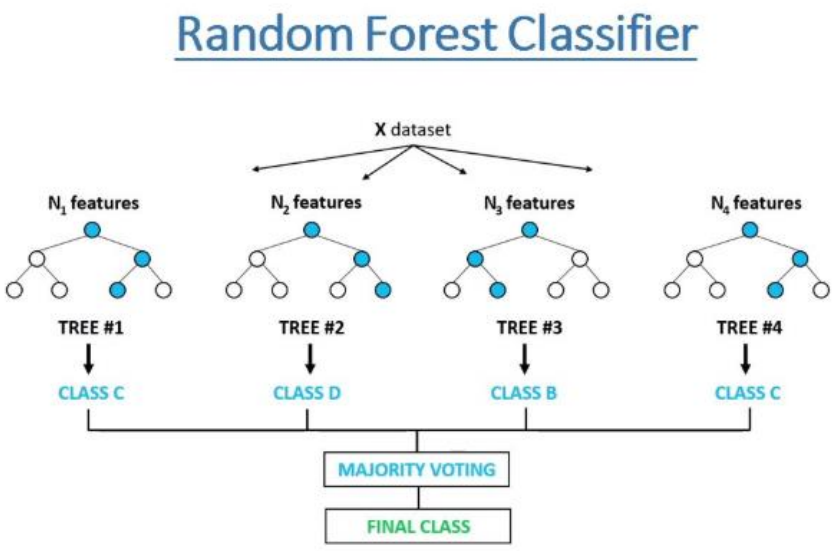

Рисунок 3.3 – Архітектура Random Forest Classifier

Рішення у вузлі дерева приймається на основі певного критерію розбиття даних, який визначає, як найкраще поділити набір даних для підвищення якості моделі. Найчастіше для цього використовуються критерії інформаційного приросту (Information Gain) або нечистота Джині (Gini Impurity).

Інормаційний приріст показує різницю між невизначеністю (ентропією) батьківського вузла та зваженою сумою невизначеності дочірніх вузлів після розбиття (3.1). Якщо розбиття даних суттєво зменшує невизначеність, то приріст інформації буде великим.

$$IG(S, A) = H(S) - \sum_{i=1}^{m} \frac{|S_i|}{|S|} H(S_i),$$

де, $H(S)$ – ентропія батьківського вузла, S – набір даних у батьківському вузлі, S_i – ентропія i-го дочірнього вузла, S_i/S – пропорція кількості зразків у піднаборі розбиття.

Ентропія використовується для оцінки чистоти вузлів дерева рішень. Для набору даних з k класами (3.2) H вимірює ступінь невизначеності у наборі даних (3.2).

$$H(S) = -\sum_{i=1}^{k} p_i \log_2 p_i ,$$

де p_i – зразки у вузлі, належать до класу i у вузлі. Нечистота Джині використовується для оцінки чистоти вузлів дерева рішень. Для набору даних з k класів (3.3) G вимірює ступінь невизначеності у наборі даних (3.2).

$$G(S) = 1 - \sum_{i=1}^{k} p_i^2,$$

де p_i – зразки у вузлі, належать до класу i у вузлі. Показник нечистоти Джині зменшується, якщо вузол стає чистішим, тобто зразки у вузлі належать до одного класу. У крайніх випадках, коли вузол містить лише зразки одного класу, нечистота Джині дорівнює нулю, що свідчить про ідеальну чистоту вузла. Навпаки, якщо зразки у вузлі рівномірно розподілені між усіма класами, індекс Джині наближається до свого максимального значення, що вказує на високу невизначеність і неоднорідність.

Ансамблеве голосування – це метод прийняття рішень у моделях ансамблю, де кожне дерево прогнозує свій клас y_k, а кінцевий результат обирається як найбільш популярний клас (3.4).

$$\hat{y} = \arg\max_{k} \sum_{i=1}^{T} I(h_i(X) = y_k),$$

де T – загальна кількість дерев, $h_i(X)$ – передбачення i-го дерева для зразка X, I – характеристична функція, яка дорівнює 1, якщо умова виконується, і 0 в іншому випадку.

Випадкове відбирання ознак – це процес, у якому на кожному вузлі дерева вибирається випадкова підмножина ознак F із загального числа ознак M (3.5). Такий підхід запобігає перенавчанню дерев і зменшує кореляцію між ними.

$$\mathrm{F} = \sqrt{M},$$

де M – загальна кількість ознак у наборі даних.

Вибір через бутстрепінг (Bootstrap Aggregation) – це метод, у якому для кожного дерева створюється вибірка S_i шляхом випадкового відбору із заміною з початкового набору даних S. Кількість зразків у S_i дорівнює розміру S, але кожен зразок може з'явитися декілька разів. Завдяки цьому методу створюється різноманітність серед дерев, а також покращується стійкість ансамблю.

Важливість ознак (Feature Importance) – один із ключових аспектів алгоритму Random Forest, який дозволяє оцінити внесок кожної ознаки у передбачення. Важливість ознаки A обчислюється як середнє зменшення критерію розбиття (наприклад, Джині) на всіх вузлах, де ця ознака використовується (3.6).

$$FI(A) = \frac{1}{T} \sum_{j=1}^{n} \sum_{t \in T_j(A)} \Delta G_t,$$

де

Математична основа Random Forest Classifier робить його потужним інструментом для багатокласових задач завдяки використанню дерев рішень, ансамблевого голосування, бутстрепінгу та випадкового відбору ознак.

Для ефективного аналізу текстових даних у цьому проекті використовується техніка TF-IDF, яка перетворює текст у числові вектори, враховуючи важливість слів у документі відносно всього корпусу текстів. Вона складається з двох компонентів: TF (частота терміну) та IDF (зворотна частота документа).

Частота терміну (TF) вимірює, наскільки часто термін t зустрічається в документі d (3.7).

$$TF(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

де $f_{t,d}$ – кількість появ терміну t у документі d, а $\sum_{t' \in d} f_{t',d}$ – загальна кількість термінів у цьому документі.

Зворотна частота документа (IDF) оцінює, наскільки рідко термін t зустрічається у всьому наборі текстів D (3.8).

$$IDF(t,D) = \log \frac{N}{1 + |\{d \in D : t \in d\}|},$$

де N – загальна кількість документів у наборі, а $|\{d \in D : t \in d\}|$ – кількість документів, у яких з'являється термін t. Додавання 1 у знаменнику запобігає діленню на нуль.

Остаточний числовий вектор TF-IDF обчислюється як:

$$TF - IDF(t,d,D) = TF(t,d) * IDF(t,D)$$

Цей показник надає високий бал словам, які часто з’являються у конкретному документі, але рідко зустрічаються в інших документах, що робить TF-IDF корисним для виявлення ключових слів.

– це метод синтетичного збільшення вибірки, що використовується для балансування класів у нерівномірних даних шляхом створення нових зразків для менш представленого класу.

SMOTE генерує нові точки у просторі ознак між існуючими зразками меншості. Для кожного зразка меншості x_i, алгоритм:

- визначає k-найближчих сусідів $\{x_1, x_2 \dots x_k\}$ у просторі ознак серед зразків цього ж класу;
- в
- и
- п
- в
- и
- м
- о
- н
- и
- й
- о
- м
- р
- о
- б
- и
- р
- a

$$x_{new} = x_i + \lambda * (x_j - x_i),$$

де $\lambda \in [0; 1]$ – випадкове число, що визначає де між x_i та x_j буде створено нову точку.

Обраний метод інтерполює нові точки вздовж відрізка між x_i та x_j, що дозволяє ефективно заповнювати простір меншості, зберігаючи структуру даних.

Таким чином, TF-IDF є ефективним інструментом для перетворення тексту в числовий формат із врахуванням важливості слів, тоді як SMOTE виступає потужним методом для вирівнювання класового дисбалансу, зберігаючи при цьому структуру даних.

З переваг даного методу можна виділити відсутність простого дублювання зразків меншості, створюючи нові точки, що знижує ризик перенавчання. Також генерація синтетичних точок розширює інформаційне представлення меншості, підвищуючи ефективність класифікації.

xnew (3.10).

Ключовою перевагою алгоритму є здатність оцінювати важливість ознак шляхом аналізу їхнього внеску у зменшення похибки класифікації на вузлах дерев. Це дозволяє визначити найбільш впливові ознаки, що підвищує інтерпретованість моделі.

Як висновок можна зазначити, що алгоритм Random Forest Classifier є оптимальним вибором для вирішення задачі класифікації завдяки його гнучкості та стійкості до перенавчання. У задачах з дисбалансом класів він підтримує автоматичне балансування за допомогою спеціальних параметрів. Крім того, обраний алгоритм працює з різноманітними типами даних, включаючи числові та текстові, а у поєднанні з TF-IDF ефективно аналізує текстові дані.

3.3 Розробка хмарної платформи

Основою архітектури є використання принципів об’єктно-орієнтованого програмування, що забезпечує чітке структурування компонентів, спрощує розширення функціоналу та підвищує зручність підтримки. Розроблена архітектура хмарної платформи базується на модульному дизайні, який представлено діаграмою класів у розділі 2.1.

Функціонал розгортання базується на використанні бібліотеки OCI для керування інфраструктурою Oracle, Docker для контейнеризації застосунків, та як користувацький інтерфейс. Платформа повинна забезпечувати гнучке, безпечне та масштабоване розгортання сервісів інтелектуальної системи та бази даних відповідно до заданих користувачем параметрів. Наприклад, впровадження всіх сервісів виключно в межах конкретної віртуальної хмарної мережі.

истема автентифікації використовує модуль «auth» (додаток А.1), що забезпечує реєстрацію, вхід та управління сесіями користувачів. Важливим елементом цього етапу стала інтеграція з GitLab для доступу до контейнерного реєстру та Oracle Cloud для налаштування управління інфраструктурою.

(додаток А.2) виконує роль абстрактного класу для моделювання екземпляру

віртуального сервера. Він включає атрибути, такі як унікальний ідентифікатор (у сервісі Oracle), назву, поточний стан, обраний процесор (наприклад, у даному випадку всі екземпляри створені на Ampere A1), публічну IP-адресу, ідентифікатори віртуальної хмарної мережі і підмереж. Методи, такі як to_dict(), дозволяють зберігати його стан у зручному форматі для подальшої обробки.

Управління віртуальними хмарними мережами та пов'язаними з ними мережевими ресурсами в OCI забезпечується за допомогою VCNManager (додаток А.3). Основний функціонал класу включає створення та видалення VCN, підмереж, інтернет-шлюзів, списків безпеки та налаштування таблиць маршрутизації. Методи create_vcn() і create_subnet() виконують перевірку параметрів, таких як формат CIDR-блоків і унікальність імен, перед створенням відповідних мережевих ресурсів.

Менеджер OCI (додаток А.4) забезпечує управління ресурсами, які охоплюють екземпляри віртуальних серверів, мережі та їхні компоненти. Методи включають створення віртуальних хмарних мереж за допомогою VCNManager, підмереж, інтернет-шлюзів, а також екземплярів віртуальних серверів за допомогою OCIInstance. Також, він підтримує моніторинг IP-адрес, оновлення таблиць маршрутизації та видалення ресурсів. Використання клієнтів і композиційних операцій OCI дозволяє забезпечити надійне виконання операцій у різних станах життєвого циклу ресурсів.

(додаток А.5) забезпечує управління локальною конфігурацією екземплярів віртуальних серверів. Основними завданнями даного класу є збереження, оновлення, видалення та отримання даних про екземпляри, які представлені об'єктами OCIInstance. Він також підтримує автоматизоване завантаження конфігурацій, централізоване збереження змін і видалення даних екземплярів. Крім того, реалізовано можливість отримання інформації про конкретні екземпляри та загального списку створених конфігурацій. Розроблена реалізація спрощує процес адміністрування та інтеграції екземплярів у локальну екосистему.

(додаток А.6), що успадковується від OCIInstance, розширює функціональність

для роботи з кластерними вузлами. Він додає атрибути для опису вузлів, такі як тип, асоціація з кластером і пов’язаний сервіс впровадження. Цей сервіс є важливим доповненням, що дозволяє створювати балансувальник навантаження між двома або більше однаковими сервісами. Функціональність об’єкта дає змогу як зберігати дані вузлів у зручному форматі, так і обробляти їх динамічно.

Логічна структура кластера представлена класом Cluster (додаток А.7), що включає вузли, тип сервісу, статус і можливий ідентифікатор балансувальника навантаження. Методи даного класу забезпечують динамічне управління вузлами в кластері. Статичний метод обробки даних дозволяє ініціалізувати об’єкт кластера з даних у вигляді словника, а також виконує зворотну операцію для збереження конфігурацій.

(додаток А.8) виконує ключову роль в управлінні життєвим циклом кластерів, забезпечуючи їх створення, масштабування, видалення та збереження конфігурацій. Він інтегрується з іншими компонентами системи, такими як InstanceManager, LoadBalancerManager та OCIManager, для оркестрації вузлів, налаштування мережевої інфраструктури та балансування навантаження. Основна функціональність передбачає координацію всіх ресурсів, необхідних для повноцінної роботи кластерів, включаючи забезпечення їх стабільності та відповідності вимогам системи.

Описаний клас також включає механізми для управління конфігураціями кластерів, що дозволяє централізовано зберігати та відновлювати дані про стан кластерів. Вбудовані механізми очищення забезпечують безпечне видалення або відновлення ресурсів у разі збоїв під час створення чи масштабування кластерів.

Окрім цього, підтримується управління SSH ключами та можливість виконання віддалених команд на серверах. Ця функціональність дозволяє динамічно взаємодіяти з вузлами кластеру для виконання адміністративних завдань, таких як запуск скриптів або оновлення конфігурацій, що робить клас ClusterManager універсальним інструментом для ефективного управління кластерними середовищами.

Система збору та управління логами представлена класом LogCollector (додаток А.9). Основний принцип роботи полягає в тому, що клас підключається до віртуальних машин через SSH, збирає різні типи логів (системні логи, метрики системи, логи контейнерів) та зберігає їх у JSON-форматі в локальній директорії. Система підтримує збір логів як з окремих екземплярів VPS, так і з цілих кластерів. Також розроблено функціонал управління життєвим циклом логів, включаючи можливість експорту в зовнішню директорію та налаштування періоду зберігання з автоматичним видаленням застарілих записів.

Робота користувача з платформою охоплює всі етапи взаємодії: від реєстрації до розгортання застосунку в контейнері. На першому етапі користувач реєструється на платформі та проходить автентифікацію в GitLab і Oracle Cloud Infrastructure (OCI). Цей процес включає створення ключів API та їх додавання до облікового запису Oracle Cloud для забезпечення необхідної інтеграції.

Наступним кроком є створення інфраструктури. Платформа автоматизує запуск віртуального сервера, включно з налаштуванням віртуальної хмарної мережі (VCN), підмережі, генерацією SSH ключів, встановленням Docker та налаштуванням доступу до реєстру GitLab. Користувач отримує інформацію про IP-адресу інстанції, SSH ключі та мережеву конфігурацію для подальшого використання.

Перед розгортанням застосунку користувач налаштовує середовище за допомогою інтерактивних команд, додаючи конфігурацію сервісу, обираючи контейнерний образ із реєстру GitLab, а також задаючи змінні середовища, порти й інші параметри. Розгортання може виконуватися як на основі попередньо підготовленої конфігурації сервісу, так і вручну шляхом вказання параметрів контейнера.

Візуалізацію описаних процесів взаємодії користувача з платформою наведено на рисунку 3.4.

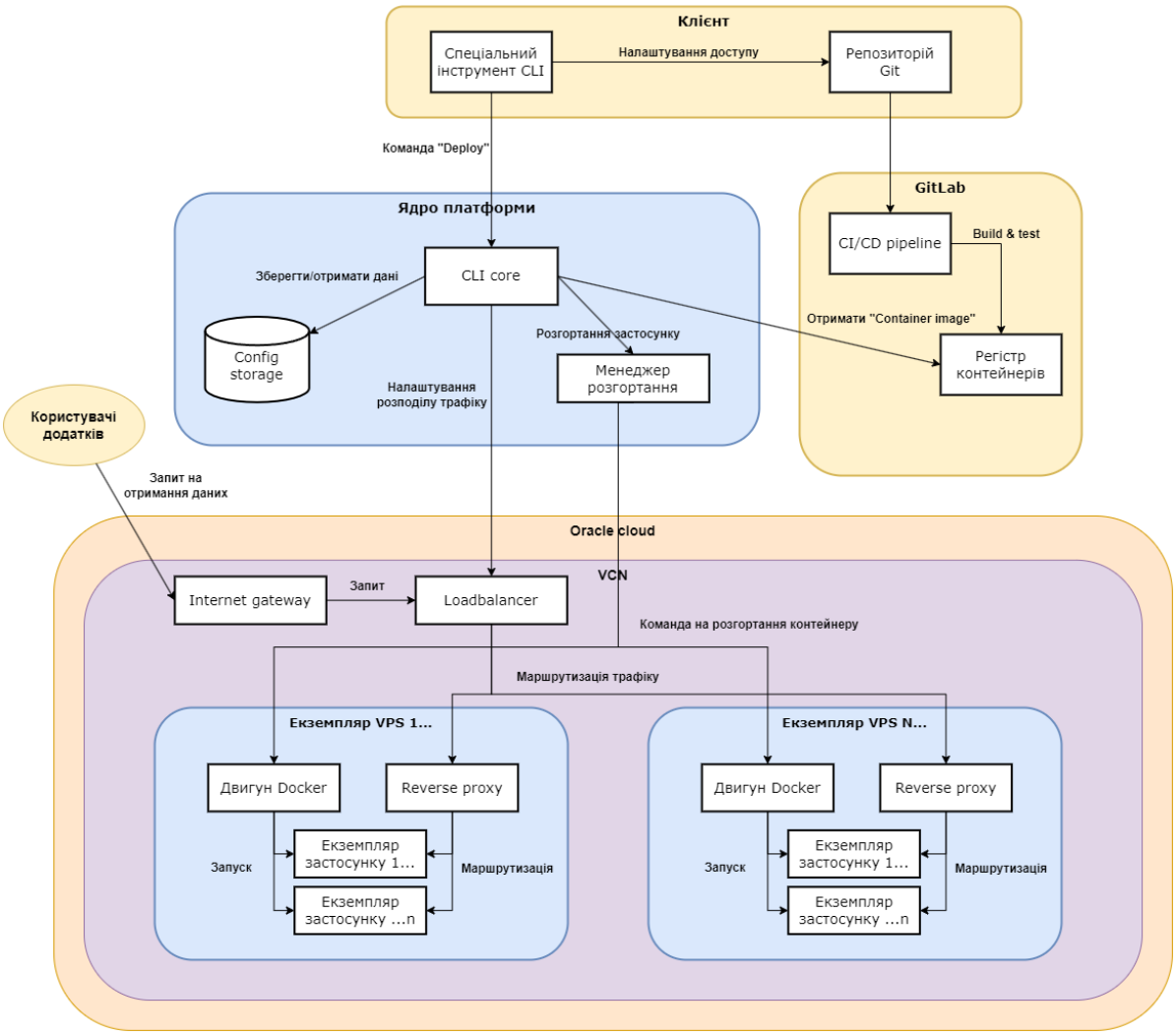

Рисунок 3.4 – Діаграма принципу розгортання екземплярів застосунків з використанням хмарної платформи

Після завершення розгортання користувач має змогу перевірити стан застосунку та його доступність у мережі за допомогою спеціальних команд. За потреби виконуються додаткові налаштування правил безпеки, зокрема для компонентів користувацького інтерфейсу. Упродовж усього процесу платформа надає інтерактивні підказки, забезпечує зворотний зв’язок і пропонує скасування процесу у разі помилок.

3.4 Розробка інтерфейсу інтелектуальної системи

Розробка інтерфейсу клієнтської частини застосунку орієнтована на основні принципи дизайну UI/UX, описані в розділі 2.2.

Першими сторінками, доступними для користувача, є сторінки реєстрації та автентифікації. На сторінці реєстрації (рис. 3.5) розміщена форма, яка містить поля для введення імені, прізвища, електронної адреси та пароля.

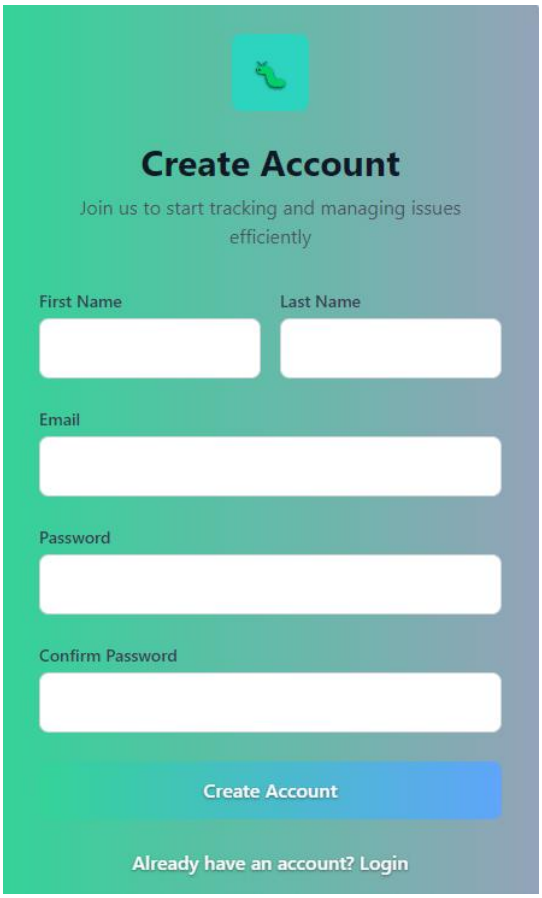

Рисунок 3.5 – Форма реєстрації користувача

Після підтвердження електронної пошти, користувачу представляється сторінка автентифікації (рис 3.6), яка забезпечує вхід до системи. Після успішної автентифікації стає доступним основний функціонал інтелектуальної системи.

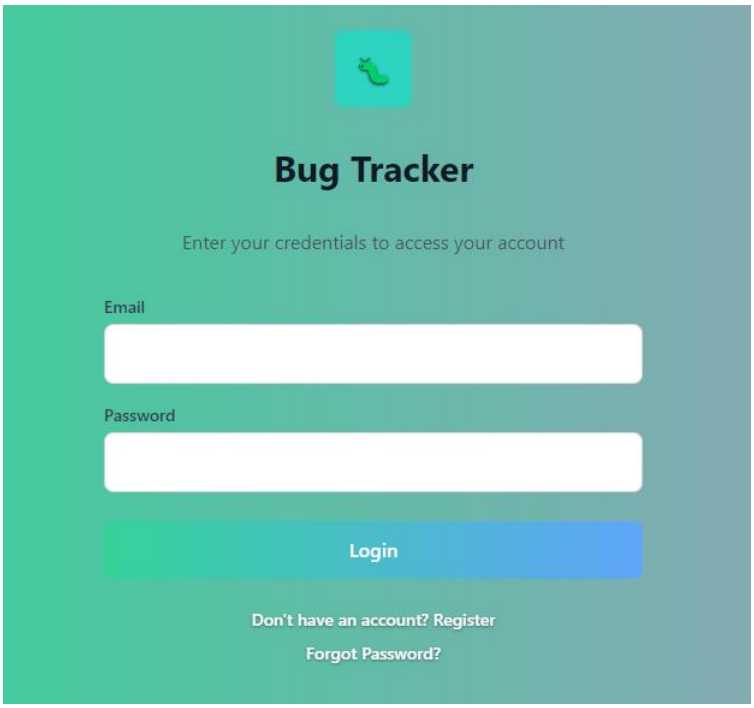

Рисунок 3.6 – Форма автентифікації користувача

Головною сторінкою сайту є Dashboard (рис. 3.7), яка представляє користувачу огляд останньої активності. На ній відображаються створені задачі, їх розподіл за статусом виконання, а також прогрес у межах призначених проектів.

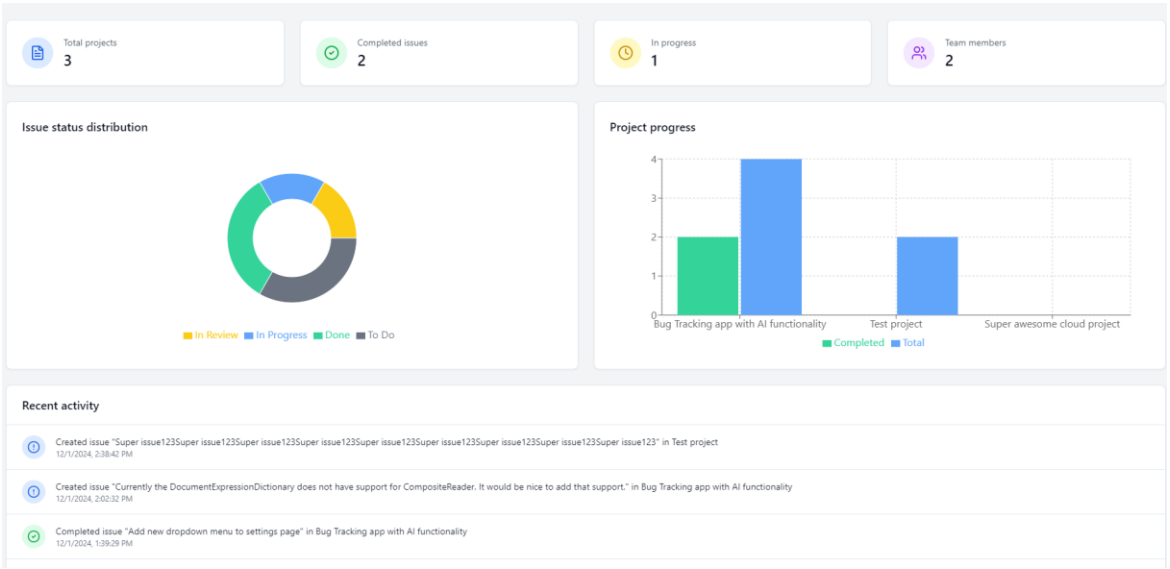

Рисунок 3.7 – Головна сторінка вебзастосунку

Наступна важлива сторінка – це сторінка доступних задач (ризикових явищ). На ній конкретніше відображаються характеристики повідомлень про ризикові явища, а саме їх назва, тип, пріоритет, статус виконання та особа, якій призначено вирішення вказаної задачі (рис. 3.8).

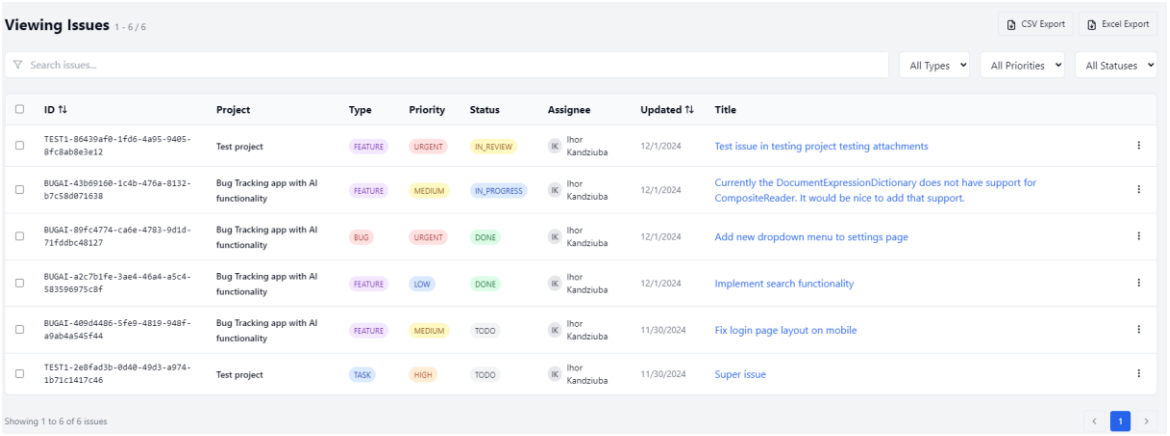

Рисунок 3.8 – Сторінка доступних задач

Для детального ознайомлення з повідомленням про поставлену задачу, користувач може скористатися сторінкою, яка відображає всю інформацію, пов’язану з конкретною проблемою (рис. 3.9). Тут надається можливість вивчити її опис, статус та переглянути прикріплені мультимедійні файли. Якщо користувач є автором задачі, він може її редагувати, змінюючи назву, пріоритет, прикріплені файли та інші параметри. Водночас для інших користувачів передбачено функціонал коментування, а також, за умови призначення їм цього завдання, зміну статусу виконання.

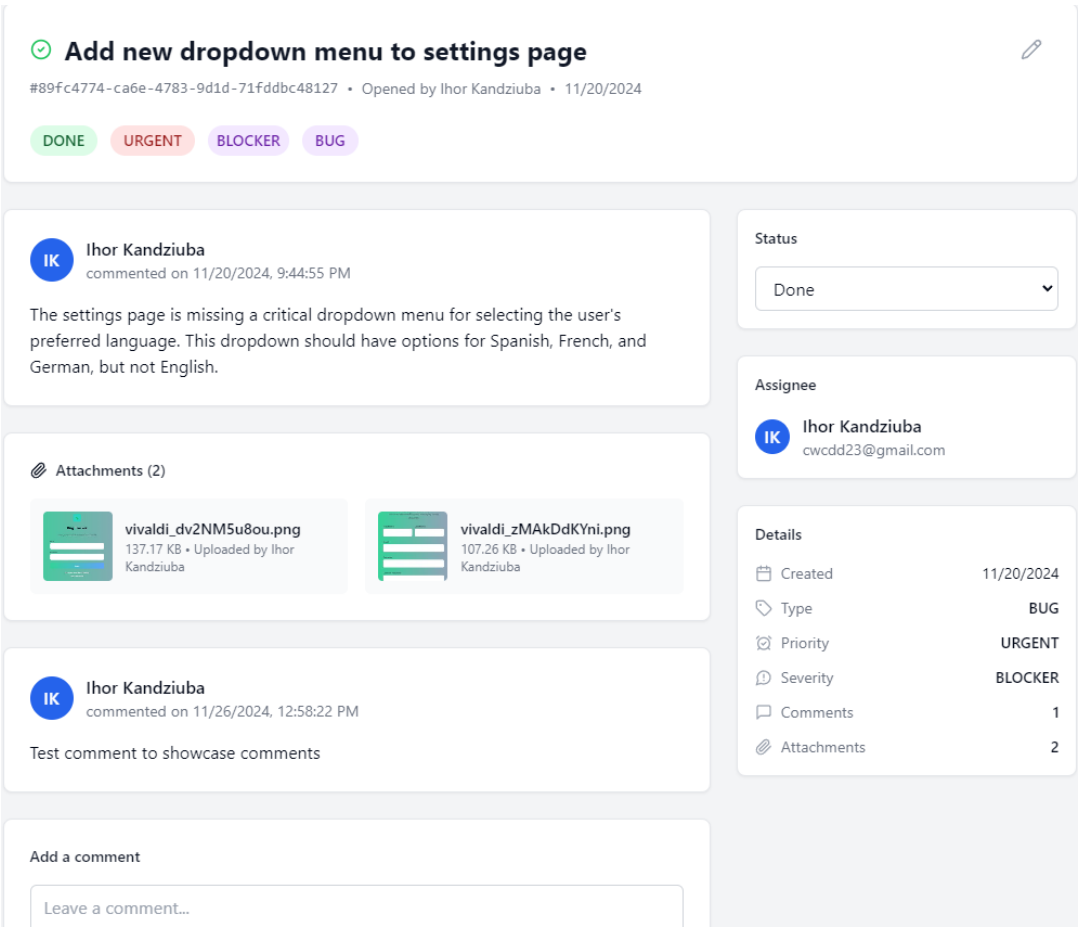

Рисунок 3.9 – Сторінка конкретної задачі

Сторінка Projects відображає всі доступні користувачу проекти у вигляді окремих карток. Кожна картка містить назву, унікальний код для ідентифікації, а також короткий опис, якщо він наявний (рис. 3.10). Крім того, для кожного проекту зазначено кількість пов’язаних задач із розподілом за статусами, такими як «To Do», «In Progress» тощо. Також відображається кількість учасників команди, які працюють над проектом.

Детальний огляд профілю користувача надає інформацію про його активність та участь у проектах (рис. 3.11). На сторінці відображається кількість завершених завдань, пов’язаних із ризиковими явищами, а також тих, що перебувають у процесі виконання. Крім того, представлено список проектів, учасником яких є користувач, із зазначенням їхньої назви, статусу та кількості завдань.

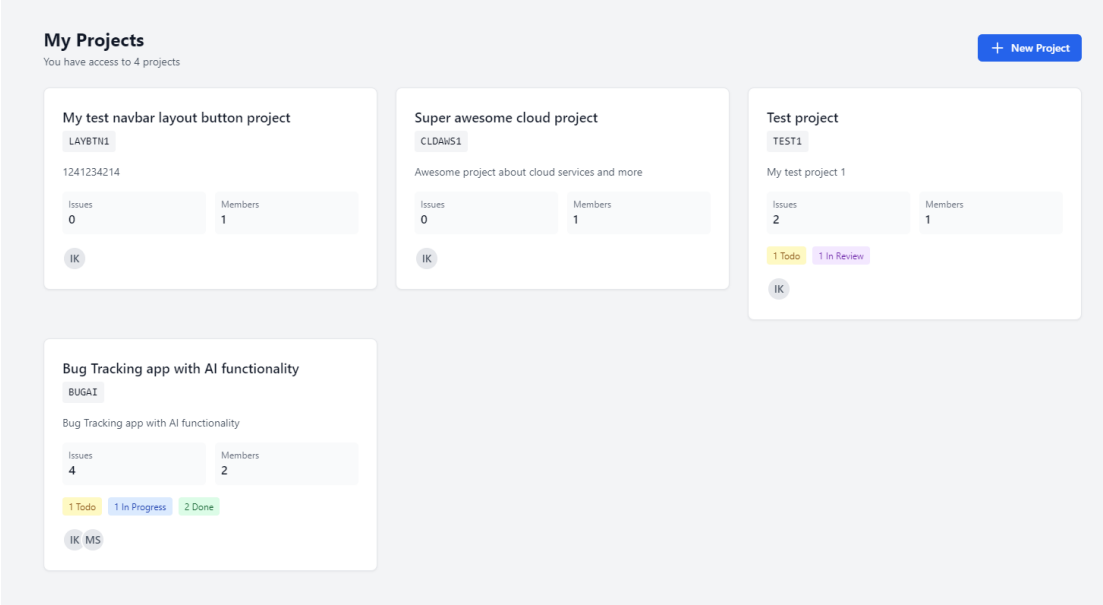

Рисунок 3.10 – Сторінка перегляду доступних проектів для користувача

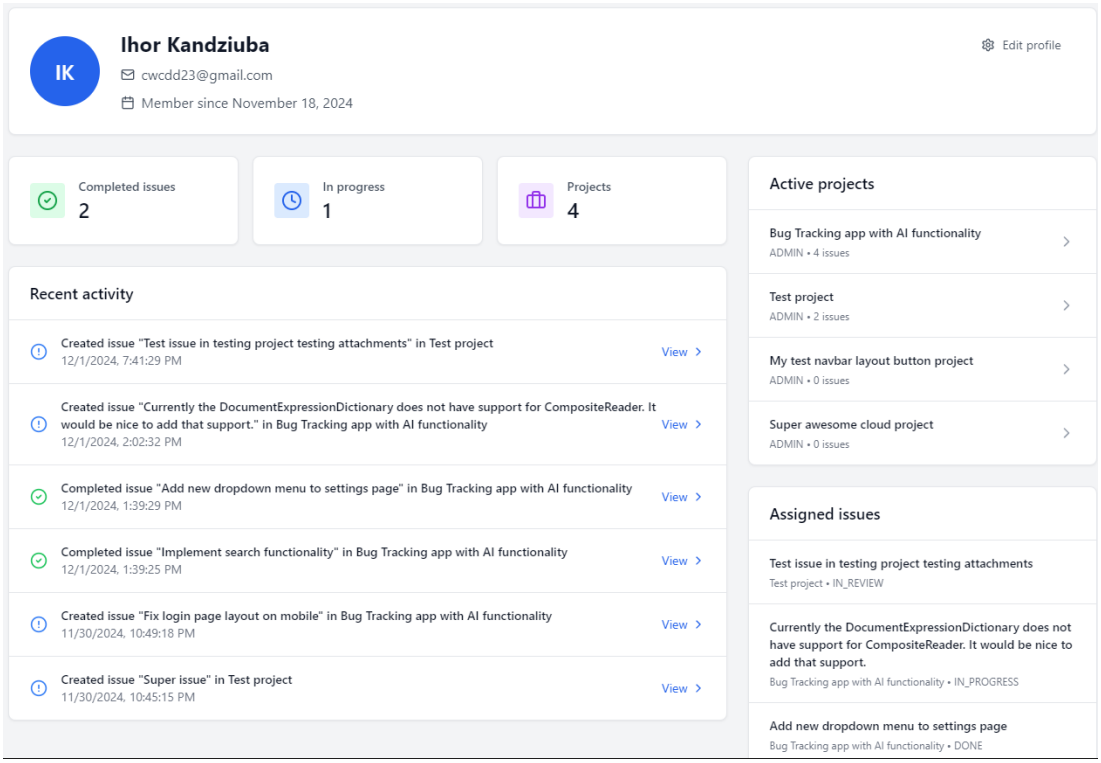

Рисунок 3.11 – Сторінка відображення профілю користувача

При необхідності зміни будь-якої персональної інформації, або ж зовнішнього вигляду сайту користувачу доступна сторінка налаштувань (рис.

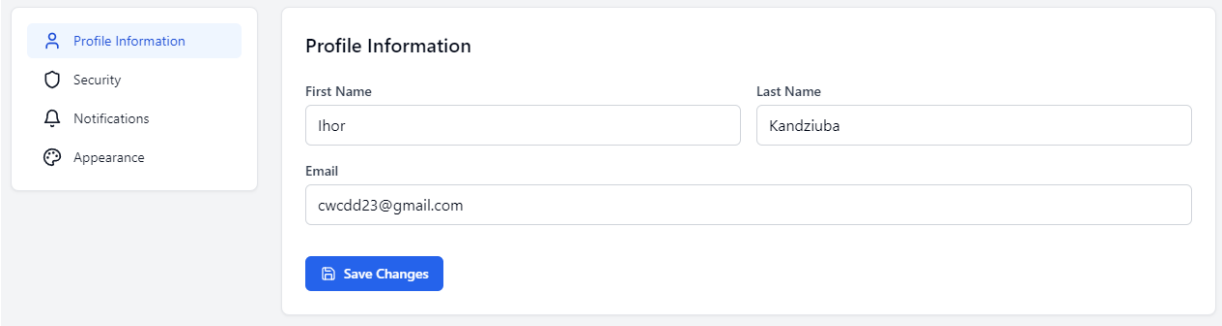

Рисунок 3.12 – Сторінка налаштувань вебзастосунку

Сторінка звітів та аналітичних діаграм, де можна обрати фільтри та завантажити дані звітів у потрібному форматі (рис. 3.13).

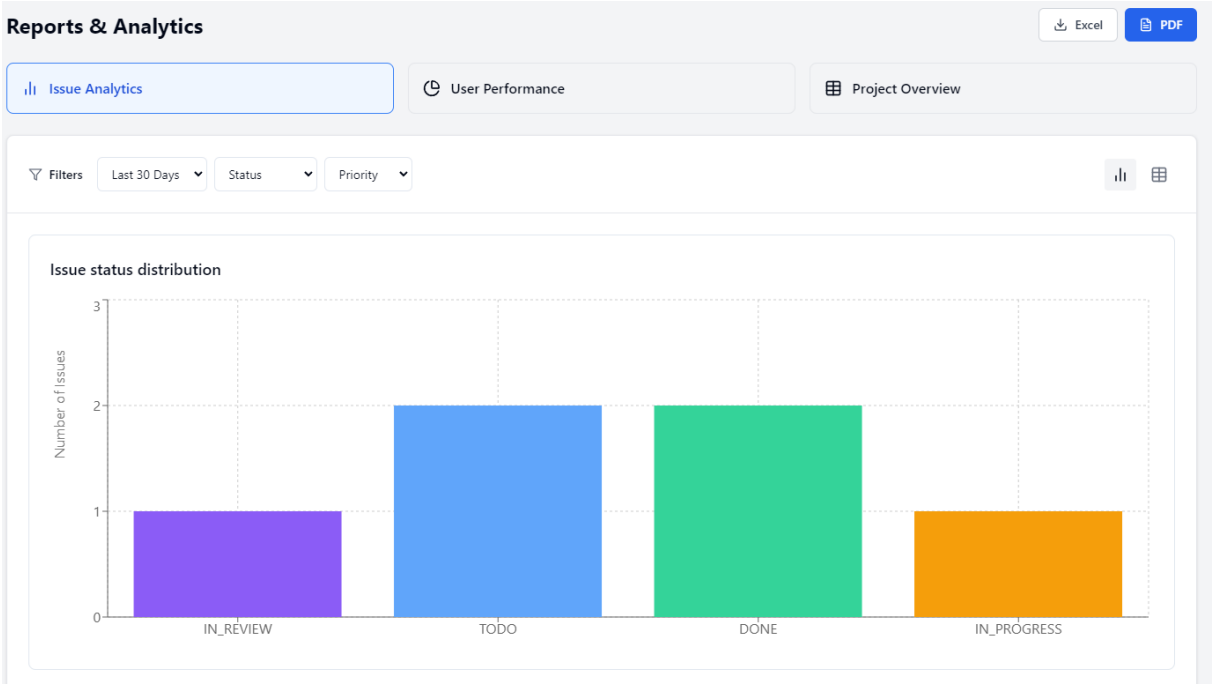

Рисунок 3.13 – Сторінка створення звітів та перегляду аналітичних даних

Tailwind CSS надає набір утилітарних класів для швидкого налаштування стилів, зокрема адаптивності. Завдяки медіа-класам, як-от sm:, md: або lg:, стилі автоматично змінюються залежно від розміру екрана. Обране рішення надає змогу створювати гнучкі сітки й адаптивні макети, забезпечуючи коректне відображення на будь-яких пристроях (рис. 3.14, 3.15). Процес передбачає

тестування дизайну на різних роздільних здатностях екрана, щоб переконатися в його коректній роботі та відповідності вимогам.

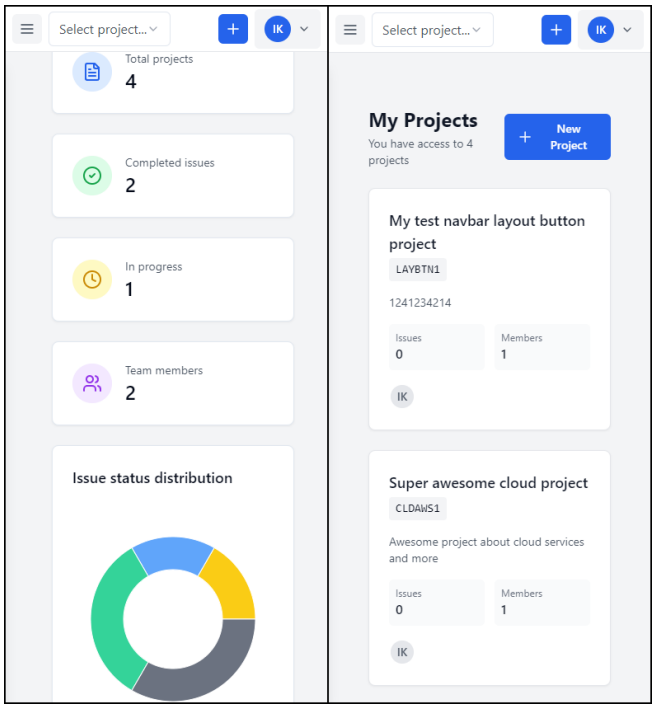

Рисунок 3.14 – Приклад адаптивного дизайну сторінок Dashboard та Projects

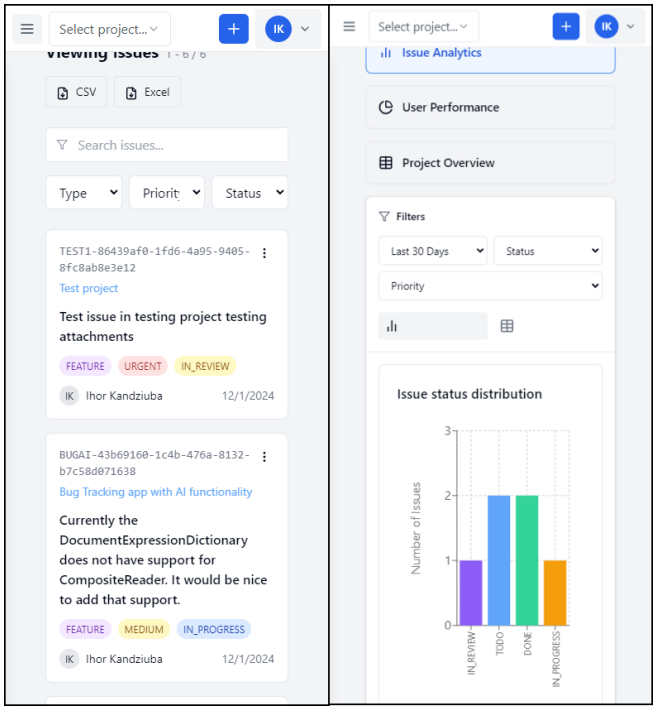

Рисунок 3.15 – Приклад адаптивного дизайну сторінок Bug reports та Analytics

Розробка функціональної частини інтелектуальної системи

Розробка функціональної частини є ключовим етапом створення інтелектуальної системи, що об'єднує серверну частину та сервіс штучного інтелекту. У цьому розділі описано реалізацію основних компонентів, які забезпечують взаємодію користувачів із системою, обробку даних і застосування алгоритмів штучного інтелекту для аналізу ризикових явищ.

3.5.1 Розробка серверної частини. Розробка починається зі створення безпосередньо серверної частини, а саме шаблону MVC (рис. 3.16).

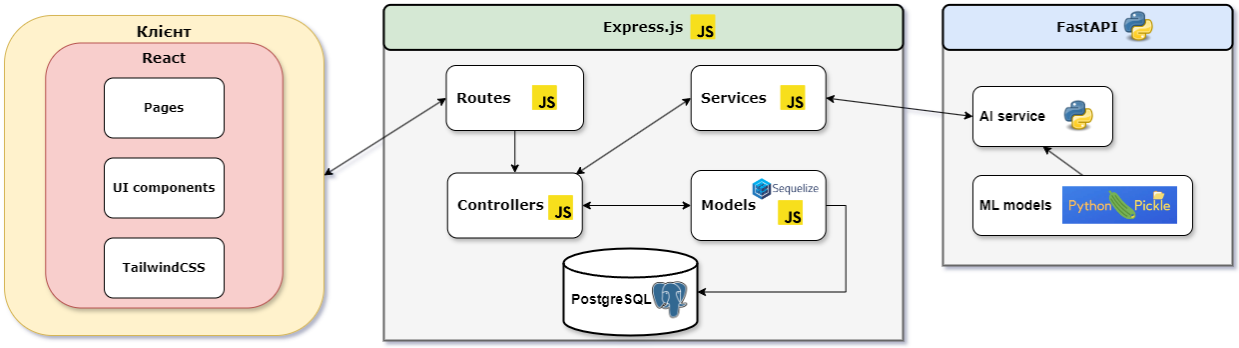

Рисунок 3.16 – Практична реалізація функціональної частини

Моделі в системі реалізовані за допомогою ORM-бібліотеки Sequelize, що забезпечує ефективну взаємодію з базою даних (додаток Б.1). Вони представляють логічну структуру таблиць, зокрема визначення атрибутів, зв'язків між моделями та бізнес-правил, що регулюють взаємодію з даними.

Контролери відповідають за обробку HTTP-запитів і викликають відповідні методи сервісів або моделей. Наприклад, під час аналізу звітів проекту контролер перевіряє отримані дані, зберігає інформацію в базі даних і надсилає її для аналізу Python-сервісу (додаток Б.2).

У реалізації відсутній традиційний компонент представлення у контексті шаблону MVC, який відповідає за серверний рендеринг HTML. Натомість функціональність маршрутизації реалізована через серверні маршрути, які

забезпечують зв’язок між клієнтською частиною та логікою серверу (додаток Б.3). Реалізований підхід спрямований на створення сучасного вебзастосунку формату SPA, де рендеринг і динамічна взаємодія виконуються на стороні клієнта, забезпечуючи швидкий та інтерактивний користувацький досвід.

Розроблено систему автоматичного розподілу завдань, яка працює за простою логікою: коли створюється нове завдання без конкретного виконавця, система аналізує всіх активних учасників проекту за трьома основними критеріями: досвід, навантаження та продуктивність (додаток Б.3). Ці критерії мають різну вагу: досвід є найважливішим і дорівнює 40%, а навантаження та продуктивність мають по 30%. Варто зазначити, що обрані значення не є фіксованими, тому адміністратор проекту може будь-коли змінити ці коефіцієнти, щоб адаптувати формулу до потреб конкретного проекту.

Щоб отримати рівень досвіду, система оцінює рівень знань учасника у вирішенні певного типу завдань (наприклад, виправлення багів або реалізація нових функцій). Цей рівень вимірюється від 1 до 5, де 5 – максимальний рівень. Наприклад, якщо учасник виконує успішно завдання будь-якого його досвід підвищується на 0,1. Щоб привести оцінку до 100-бальної шкали, рівень досвіду множиться на 20.

Для отримання навантаження система враховує кількість активних завдань у учасника порівняно з максимально дозволеною кількістю (5 завдань). Додатково враховується відсоток доступності учасника. Наприклад, якщо у члена команди три активні завдання з п’яти можливих, і доступність становить 80%, бал навантаження обчислюється шляхом віднімання поточного завантаження (60%) від доступності (80%), що дає 20 балів.

Продуктивність обчислюється шляхом аналізу того, як швидко і якісно учасник завершував подібні завдання протягом останніх 30 днів. Ідеальний час завершення – 24 години, що приносить 100 балів. Якщо середній час виконання завдання становить 48 годин, бал продуктивності буде дорівнювати 50.

Після обчислення значень за всіма трьома критеріями система комбінує їх за формулою оцінки загального балу (3.11).

$$Score = (E * 0{,}4) + (W * 0{,}3) + (P * 0{,}3),$$

де E – досвід учасника, W – наватаженість учасника, P – продуктивність учасника.

Після підрахунку загального бала всіх активних учасників проекту завдання отримує учасник з найвищим загальним балом (за умови, що у нього менше п’яти поточних завдань).

3.5.2 Розробка сервісу штучного інтелекту. Створення сервісу на основі штучного інтелекту є складним процесом, що поєднує інноваційні технології та багатоступеневий підхід до роботи з даними. Центральною ідеєю такого сервісу є здатність виконувати автоматизований аналіз звітів про помилки, який допомагає приймати обґрунтовані рішення щодо ризикових явищ. Цей процес включає побудову алгоритмів машинного навчання, які інтегруються у вебзастосунок для забезпечення взаємодії з користувачами через інтуїтивно зрозумілий інтерфейс.

Основними етапами розробки є попередня обробка даних, навчання моделей і реалізація REST API для взаємодії між клієнтською частиною і сервером. Початковий етап, що стосується роботи з даними, є найважливішим, оскільки якість даних прямо впливає на ефективність і точність створеної моделі. Він складається з процесів збору, очищення та підготовки даних, які є критично важливими для подальшого моделювання.

По-перше, слід почати з обробки та очищення даних. Даний етап передбачає завантаження, очищення та трансформацію даних для подальшого використання у моделюванні (Додаток В.1).

Спочатку відбувається завантаження датасету у форматі CSV з ресурсу

Очищення даних включає:

- видалення рядків з пропущеними ключовими значеннями;
- заповнення порожніх текстових полів значеннями за замовчуванням;
- обробка тексту: нормалізація пробілів, видалення спецсимволів.

Трансформація даних включає:

- призначення категорій (наприклад, пріоритет на «URGENT», тощо);
- розрахунок часу вирішення дефекту;
- динамічне обчислення критичності на основі пріоритету та часу вирішення;
- збереження даних у форматі JSON (запаковані у .gz для збереження місця на диску).

Навчання моделей націлене на побудову класифікаційних алгоритмів, які здатні аналізувати характеристики ризикових явищ. Першим етапом цього процесу є попередня обробка даних. Вона включає розподіл даних на навчальну та тестову вибірки зі збереженням пропорцій кожного класу для уникнення дисбалансу (стратифікація). Текстові дані векторизуються за допомогою алгоритму TF-IDF, після чого параметри оптимізуються для покращення представлення текстових характеристик.

На етапі навчання моделі створюються класифікатори, кожен з яких спеціалізується на окремій метриці: визначення пріоритету, типу завдання та його критичності. Для цього використовується алгоритм Random Forest Classifier, а для усунення нерівності класів застосовується SMOTE. У процесі оцінювання результативності моделі розраховуються метрики класифікації, зокрема macro F1 core, які дають змогу комплексно оцінити якість передбачень. Завершальним етапом є збереження навчених моделей у форматі .pkl для подальшої інтеграції (додаток В.2).

Інтеграція моделей штучного інтелекту у вебзастосунок здійснюється через REST API. Для цього реалізовано механізм завантаження моделей з хмарного сховища Amazon S3, де вони зберігаються, щоб забезпечити доступ для

різних екземплярів контейнерів сервісу штучного інтелекту. Процес завантаження контролюється через спеціальний менеджер моделей (клас ModelManager), який забезпечує логування прогресу та перевірку цілісності файлів (додаток В.3).

Однією із ключових точок доступу до API є аналіз проектів, яка забезпечує автоматичний аналіз характеристик завдань. Описаний функціонал включає прогнозування типу, пріоритету та критичності ризикових явищ, а також впевненості моделей у своїх передбаченнях. Додатково реалізовано механізми агрегації метрик на рівні проекту, такі як визначення середнього рівня складності дефектів.

Для забезпечення коректної роботи API використовується валідація вхідних даних за допомогою Pydantic, який гарантує правильну типізацію даних та ефективну обробку потенційних помилок. Ці механізми інтегровані через функціонал FastAPI, що забезпечує високу швидкодію та гнучкість у взаємодії з клієнтським застосунком (додаток В.4).

Розробка бази даних

Реалізація бази даних для системи відслідковування ризикових явищ у застосунку ґрунтується на використанні PostgreSQL, відомої своїми можливостями забезпечення надійності, масштабованості й підтримкою сучасних типів даних. Нижче наведено опис основних таблиць, їх взаємозв’язків і методів оптимізації.

Таблиця «Users» призначена для зберігання даних про користувачів системи:

- основні поля – електронна пошта, зашифрований пароль, ім’я, прізвище;
- система верифікації реалізована полями isVerified, verificationToken та визначенням його терміну дії;

- додано унікальність для електронної пошти для уникнення дублювання облікових записів;
- автоматична генерація унікального ідентифікатора через

- встановлення часових міток createdAt та updatedAt.

Таблиця «Projects» забезпечує зберігання інформації про проекти:

- поля назви, унікального ключа (короткий ідентифікатор), опису та налаштувань (обрано формат JSONB для гнучкості);
- зв’язок із учасниками та завданнями через інші таблиці.

Таблиця «ProjectMembers» реалізує зв’язок багато-до-багатьох між користувачами та проектами:

- поля ролі та статусу забезпечують рольову модель (наприклад, адміністратор, учасник) і поточний статус користувача в проекті;
- враховується продуктивність кожного члена через поля issuesCompleted, averageCompletionTime, а також деталізовані метрики продуктивності у форматі JSONB;
- додано унікальний індекс для зв’язків userId і projectId.

Таблиця «Issues» зберігає інформацію про завдання проекту:

- типи завдань, статуси, пріоритети та серйозність задаються через обмеження CHECK, що забезпечує коректність даних;
- зв’язок із проектами, авторами reporterId та виконавцями assigneeId реалізовано через зовнішні ключі;
- поля для текстового опису та часових міток;
- індекси для швидкого пошуку за проектами, авторами та виконавцями.

Таблиця «Comments» моделює коментарі до завдань:

- має зв’язки з таблицями Issues і Users;

- поле content зберігає текст коментаря.

Таблиця «Attachments» призначена для зберігання файлів, прикріплених до ризикових явищ:

- поля для назви файлу, оригінальної назви, MIME-типу, розміру та шляхів збереження;
- зв’язок з таблицями Issues і Users.

Щодо реляційної моделі, то вона побудована на основі спроектованої ER-діаграмі у розділі 2.3 (рис 3.17). Схема побудована на основі зв’язків один-до-багатьох і багато-до-багатьох, забезпечуючи чітку ієрархію:

- Users ↔ Projects через ProjectMembers;
- Projects ↔ Issues через зовнішній ключ projectId;
- Issues ↔ Comments/Attachments через відповідні зв’язки.

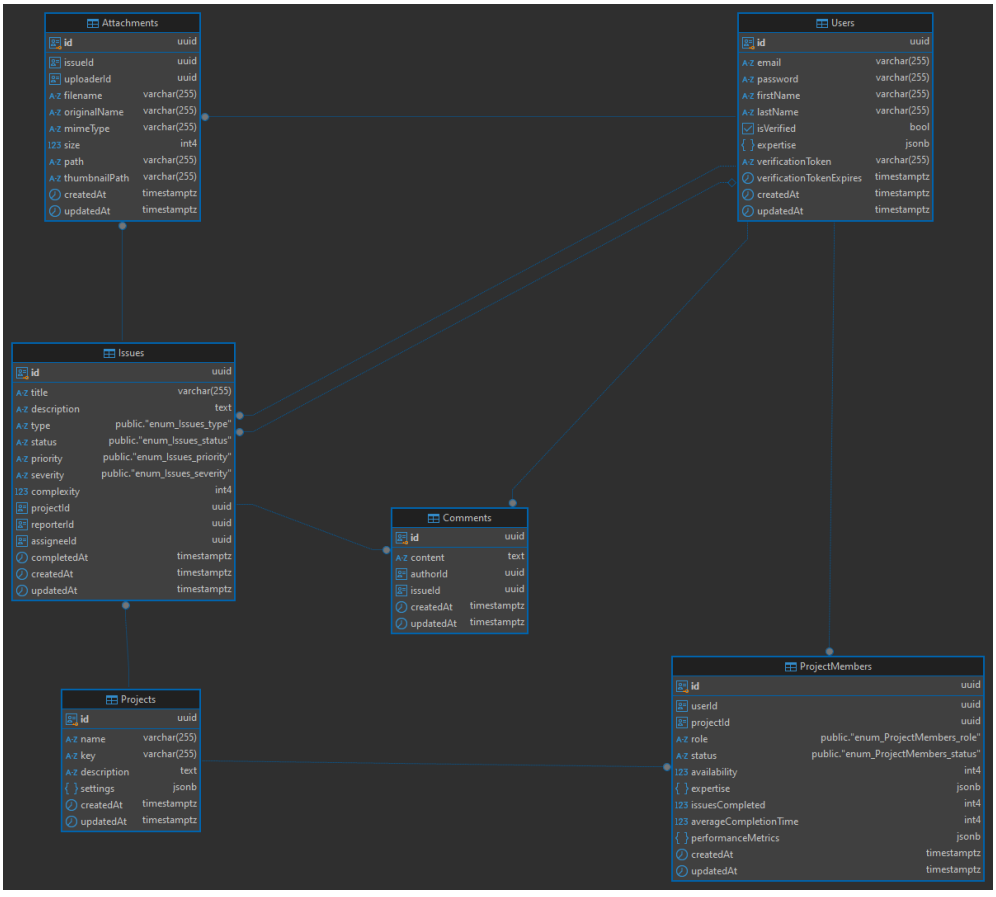

Рисунок 3.17 – Структура бази даних проекту

Отже, розроблена база даних побудована на основі PostgreSQL, що забезпечує надійність, гнучкість і масштабованість системи. Використання сучасних рішень, таких як JSONB для зберігання гнучких налаштувань, індекси для оптимізації запитів та обмеження для перевірки коректності даних, дозволяє ефективно працювати з інформацією про користувачів, проекти та ризикові явища. Проектування реляційної моделі з урахуванням зв’язків між ключовими таблицями гарантує логічність структури та спрощує управління даними.

РОЗДІЛ 4
ТЕСТУВАННЯ ХМАРНОЇ ПЛАТФОРМИ ТА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

Огляд використаних видів тестування

Тестування – це процес оцінювання роботи програмного продукту з метою виявлення дефектів, перевірки його відповідності заданим вимогам та забезпечення якості [31]. Воно дозволяє підтвердити відповідність програмного забезпечення функціональним і нефункціональним вимогам, сприяючи гарантуванню його стабільності, ефективності та відповідності потребам користувачів.

Функціональне тестування – один із ключових видів перевірки програмного забезпечення, що забезпечує відповідність його функціональних характеристик технічним вимогам. Головна мета цього процесу – підтвердження реалізації всіх функцій, передбачених технічним завданням. Тестування базується на аналізі специфікацій функціональності компонентів чи системи загалом, перевіряючи визначену поведінку.

На практиці воно може охоплювати перевірку роботи аналізу штучного інтелекту, правильність збереження інформації у базі даних, а також коректність агрегації й відображення даних у користувацькому інтерфейсі.

Інтеграційне тестування перевіряє взаємодію між різними компонентами програмного забезпечення. На цьому етапі аналізується, наскільки узгоджено й коректно дані передаються між модулями, такими як сервіс штучного інтелекту, бази даних, API чи інші механізми комунікації. Наприклад, тестування може включати оцінку точності передачі даних від сервісу штучного інтелекту до серверної частини, правильність вибірки даних із бази та їхнього відображення на сайті, а також перевірку стабільності роботи API.

Тестування продуктивності – ключовий етап оцінки швидкодії, стійкості та масштабованості програмного забезпечення. Його мета – визначити, наскільки

швидко й ефективно система виконує функції за різних навантажень. Наприклад, це тестування може охоплювати перевірку швидкості отримання даних серверною частиною від сервісу ШІ, часу запису й вибірки даних із бази, а також оцінку затримок у відображенні інформації на сайті.

Обраний підхід до тестування дозволяє забезпечити високу якість роботи системи, відповідність її функціоналу очікуванням користувачів та ефективність управління ризиковими явищами у програмному забезпеченні.

4.2 Тест-дизайн складових хмарної платформи

Чекліст тестування консольного інструмента як складової хмарної платформи включає:

- перевірка функціоналу реєстрації;
- перевірка процесу створення інфраструктури Oracle Cloud;
- перевірка функціоналу команди впровадження сервісу.

Тестування консольного інструмента передбачає створення тест-кейсів (табл. 4.1-4.3).

Таблиця 4.1 – Тест-кейс 1 для перевірки консольного інструмента

№	Крок	Дія	Очікуваний результат
	Передумови	Користувач не зареєстрований у системі	
	Опис	Перевірка функціоналу реєстрації	
		Ввести команду для проведення реєстрації у платформі.	Консоль відображає поля для введення реєстраційних даних.
		Ввести коректні дані для реєстрації у поля «Name» та	Повідомлення про успішну реєстрацію користувача.

Таблиця 4.2 – Тест-кейс 2 для перевірки консольного інструмента

№	Крок	Дія	Очікуваний результат
	Передумови	Користувач зареєстрований у системі	
	Опис	Перевірка створення екземпляру інфраструктури Oracle	
		Ввести команду «create infrastructure --name (назва)» з бажаним параметром назви.	У інтерактивному меню консоль відображає доступні VCN та підмережі.
		Обрати «Create new VCN».	Повідомлення про успішне створення віртуальної мережі.
		Обрати «Create new subnet».	Повідомлення про успішне створення підмережі та публічну IP-адресу екземпляру.

Таблиця 4.3 – Тест-кейс 3 для перевірки консольного інструмента

№	Крок	Дія	Очікуваний результат
	Передумови	ористувач автентифікований у системі ільовий екземпляр VPS налаштований під GitLab	
	Опис	Перевірка команди впровадження сервісу	
		Ввести команду «deploy» для впровадження сервісу.	Консоль відображає доступні сервіси для впровадження.
		Обрати сервіс «AI-service» з наданого меню.	Консоль відображає доступні екземпляри VPS для впровадження обраного сервісу.
		Обрати бажаний екземпляр VPS з наданого меню.	Повідомлення про успішне впровадження сервісу з інформацією про публічну IP-адресу та використаний порт.

Тест-дизайн складових інтелектуальної системи

Чекліст тестування вебзастосунку як складової інтелектуальної системи включає:

- перевірку процесів взаємодії користувачів із системою, зокрема реєстрації, авторизації та відображення персональної інформації;
- тестування створення та управління проектами, включаючи введення даних, збереження та відображення у списку проектів;
- перевірку створення завдань та роботи з ними, зокрема введення атрибутів задач, їх збереження та відображення на відповідних сторінках;
- тестування фільтрації та пошуку інформації за різними параметрами, такими як тип, статус чи текстові запити.

Проведення тестування вебзастосунку передбачає створення тест-кейсів (табл. 4.4-4.7).

Таблиця 4.4 – Тест-кейс 1 для перевірки вебзастосунку

№	Крок	Дія	Очікуваний результат
	Передумови	Відкрита сторінка «Login»	
	Опис	Перевірка функціоналу авторизації	
		Заповнити поля форми коректними даними.	Візуальне відображення введених користувачем даних у полях форми.
		Натиснути лівою кнопкою миші на кнопку «Login».	Після успішної авторизації виконано перенаправлення на «Dashboard».
		Знайти у стрічці навігаційного меню дані поточного користувача.	У стрічці навігаційного меню відображаються коректні ім’я та електронна пошта користувача.

Таблиця 4.5 – Тест-кейс 2 для перевірки вебзастосунку

№	Крок	Дія	Очікуваний результат
	Передумови	Користувач авторизований та знаходиться на сторінці	
	Опис	Перевірка створення нового проекту	
		Натиснути лівою кнопкою миші на кнопку «New	Відкрито модальне вікно створення проекту.
		Ввести вибрані назву, ключ та опис проекту.	Візуальне відображення введених користувачем даних у полях форми.
		Натиснути лівою кнопкою миші на кнопку «Create	Проект створено, і його відображено у списку.

Таблиця 4.6 – Тест-кейс 3 для перевірки вебзастосунку

№	Крок	Дія	Очікуваний результат
	Передумови	Відкрита сторінка конкретного проекту	
	Опис	Перевірка створення та відображення нового завдання	
		Натиснути лівою кнопкою миші на кнопку «New	Відкрито модальне вікно створення задачі.
		Ввести коректний заголовок, опис, тип , серйозність, пріоритет, та додати медіа-додатки.	Візуальне відображення введених користувачем даних у полях форми.
		Натиснути лівою кнопкою миші на «Create Issue».	Задача створена та додана до списку.
		У списку обрати створену задачу та перейти за посиланням.	На сторінці задачі коректно відображено тип, пріоритет, серйозність та мультимедійні вкладення.

Таблиця 4.7 – Тест-кейс 4 для перевірки вебзастосунку

№	Крок	Дія	Очікуваний результат
	Передумови	Відкрита сторінка «Issues»	
	Опис	Перевірка роботи фільтрів і пошуку завдань	
		Вибрати тип «Bug» у контекстному меню фільтрів.	Відображаються лише задачі з типом «Bug».
		Вибрати статус «In progress» у контекстному меню фільтрів	Відображаються задачі зі статусом «In progress».
		Ввести текст «Login page у поле пошуку.	Відображаються задачі, які містять пошуковий текст –

4.4 Тестування за допомогою Lint та Jest

Lint та Jest є популярними інструментами, які використовуються для забезпечення якості коду, хоча їхні функції суттєво відрізняються. Вони виконують різні цілі у процесі розробки програмного забезпечення, але можуть ефективно доповнювати одне одного в одному проекті.

Lint – це інструмент статичного аналізу коду, який перевіряє відповідність коду встановленим стилістичним та синтаксичним правилам. Основною метою є виявлення помилок ще до виконання коду, що дозволяє запобігти багатьом потенційним проблемам. Наприклад, ESLint є поширеним «лінтером» для коду на JavaScript, який допомагає знаходити помилки, такі як неправильне використання змінних, відсутність крапок із комами або недотримання стилю кодування [32]. Використання Lint значно покращує читабельність коду та сприяє дотриманню єдиних стандартів при розробці.

Jest, у свою чергу, є фреймворком для тестування, створеним для забезпечення автоматизованого тестування JavaScript-застосунків. Його основна мета – перевірка коректності роботи функціональності коду через написання

тестів, що можуть охоплювати модульне, інтеграційне та snapshot-тестування пропонує функції для зручного налагодження, як-от вбудована перевірка очікуваних результатів і можливість запуску тестів у паралельному режимі. Завдяки цим особливостям Jest здобув популярність серед розробників React-застосунків.

Чеклісти, наведені у розділі 4.3, стали основою для ретельного тестування вебзастосунку на всіх етапах розробки. Вони дозволяють не лише автоматизувати перевірки завдяки Jest, але й забезпечити структурований підхід до виявлення потенційних проблем у функціональності. Кожен тест-кейс охоплює конкретну задачу, що робить процес перевірки ефективним та цілеспрямованим (додаток Д.1).

Таким чином, у проекті Lint забезпечив статичний аналіз коду, допомагаючи виявляти помилки та дотримуватися стандартів написання. Для автоматизованого тестування функціональності використовувався Jest, який перевіряв коректність роботи застосунку та знижував ризик виникнення помилок, що сприяло стабільності та високій якості кінцевого продукту.

Впровадження інтелектуальної системи

Завершивши розробку тестів, інтелектуальна система може бути впроваджена за допомогою створеної хмарної платформи. Процес впровадження починається із завантаження коду проекту на платформу GitLab. Для організації роботи над проектом використовуються окремі репозиторії для клієнтської частини, серверної частини та сервісу штучного інтелекту. Це дозволяє чітко структурувати процес розробки та впровадження.

Для створення готового Docker-контейнера необхідно налаштувати два основних конфігураційних файли: .gitlab-ci.yml для автоматизації роботи з GitLab CI/CD та Dockerfile для побудови контейнерів.

Файл .gitlab-ci.yml відповідає за налаштування GitLab Pipeline. Він дозволяє автоматизувати різні етапи життєвого циклу проекту, такі як тестування, збірка тощо. Цей файл є інструкцією, яка визначає, які дії потрібно виконувати, у якій послідовності та в яких умовах. Наприклад, можна налаштувати автоматичний запуск тестів, які перевіряють функціональність застосунку, а після їх успішного завершення – автоматичну збірку Docker-образу. У .gitlab-.yml також зазначається середовище, в якому виконуються команди (наприклад, Node.js або Python), та вказуються залежності, які потрібно встановити перед запуском певних завдань.

Dockerfile слугує інструкцією для створення Docker-образу. У ньому визначається платформа (наприклад, Node.js для вебзастосунків або Python для сервісів штучного інтелекту), яка буде використана як базова. Потім у файлі зазначається послідовність дій, необхідних для створення готового образу. Це може включати встановлення залежностей, копіювання вихідного коду застосунку, виконання підготовчих команд, таких як тестування чи збірка проекту, а також вказівку основної команди, яка запускає застосунок усередині контейнера. Додатково у Dockerfile можна визначити порти, які будуть відкриті для доступу до сервісу, та інші специфічні параметри для контейнера.

У конкретному випадку збірки Docker-контейнера для поточної інтелектуальної системи є певні нюанси. Важливо зазначити, що треба врахувати кінцеву платформу розгортання, тобто якщо інтелектуальна система буде впроваджуватись на Oracle Cloud VPS, то налаштовувати Dockerfile відповідним чином. Важливо врахувати, що ті екземпляри віртуальних серверів, які будуть використані для впровадження, працюють на платформі ARM64, а не стандартній x86.

Після встановлення екземпляру консолі потрібно виконати процес реєстрації та прив'язки ключів доступу до репозиторію GitLab (рис. 4.1).

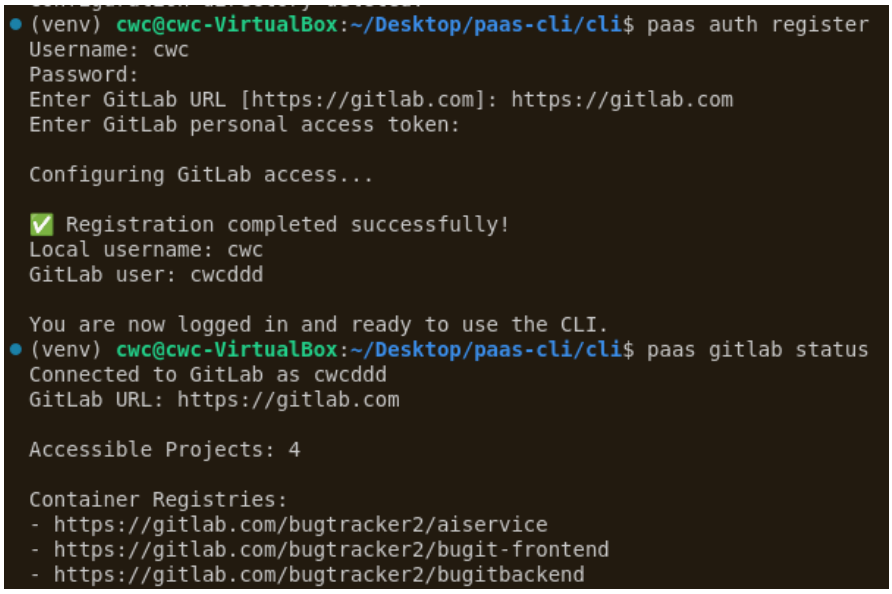

Рисунок 4.1 – Виконання команди реєстрації та перевірка статусу репозиторіїв GitLab

На наступному етапі створюється VPS, який слугуватиме платформою для розгортання необхідних сервісів (рис. 4.2).

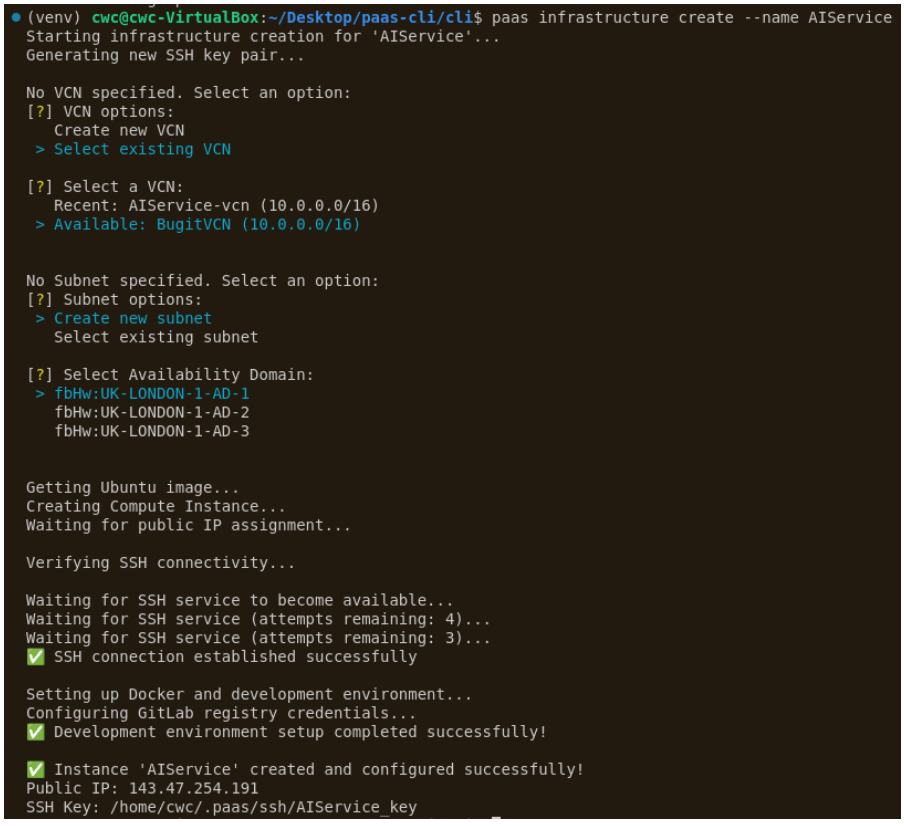

Рисунок 4.2 – Створення екземпляру VPS для сервісу штучного інтелекту

З наведеного рисунку можна зробити висновок, що команда є інтерактивною та простою у виконанні. Також консоль надає достатньо інформації, щоб зробити висновок про успішність створення та налаштування екземпляру VPS.

Щоб впровадити бажаний сервіс на щойно створений VPS потрібно спочатку виконати певні налаштування. Спочатку обирається репозиторій, потім з GitLab реєстру контейнерів обирається Docker контейнер та його тег. Також потрібно виконати налаштування безпосередньо самого контейнера, тобто як він буде працювати на обраному екземплярі: відповідність порта VPS до порта безпосередньо контейнера застосунку, призначення томів (Volume mapping) для збереження даних за межами життєвого циклу контейнера та файл середовища, де вказані секрети, які потрібні для розгортання (рис. 4.3).

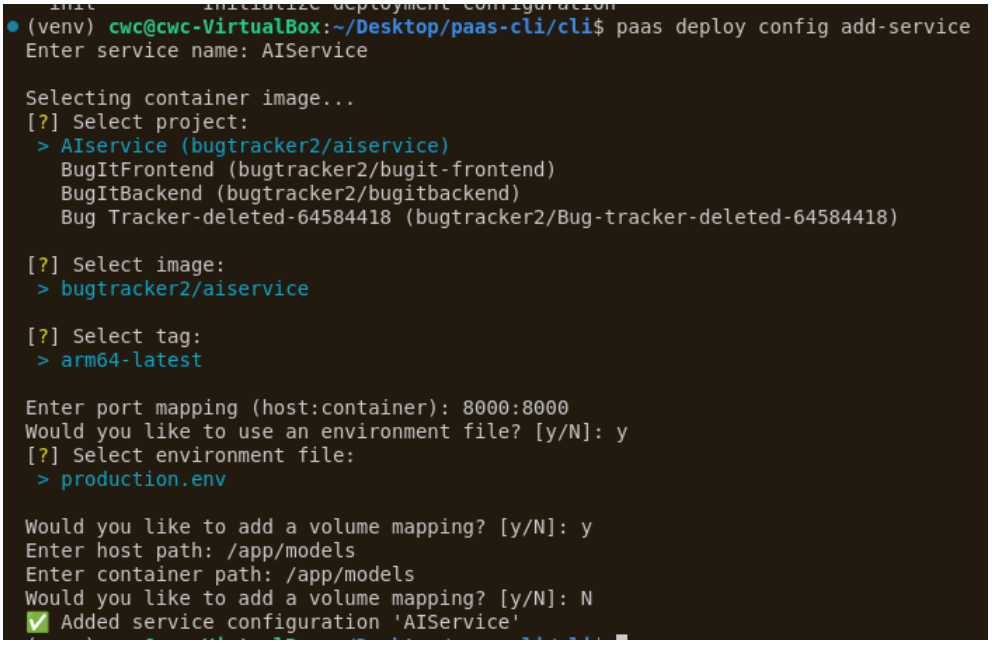

Рисунок 4.3 – Конфігурація сервісу для подальшого впровадження

По закінченню налаштувань сервісу, він зберігається до конфігураційних файлів користувача, тобто його можна буде впроваджувати по обраному шаблону безліч разів.

На наступному етапі впровадження спочатку обирається бажаний сервіс, після чого з переліку доступних екземплярів вибирається VPS, і розпочинається запуск контейнера (рис. 4.4).

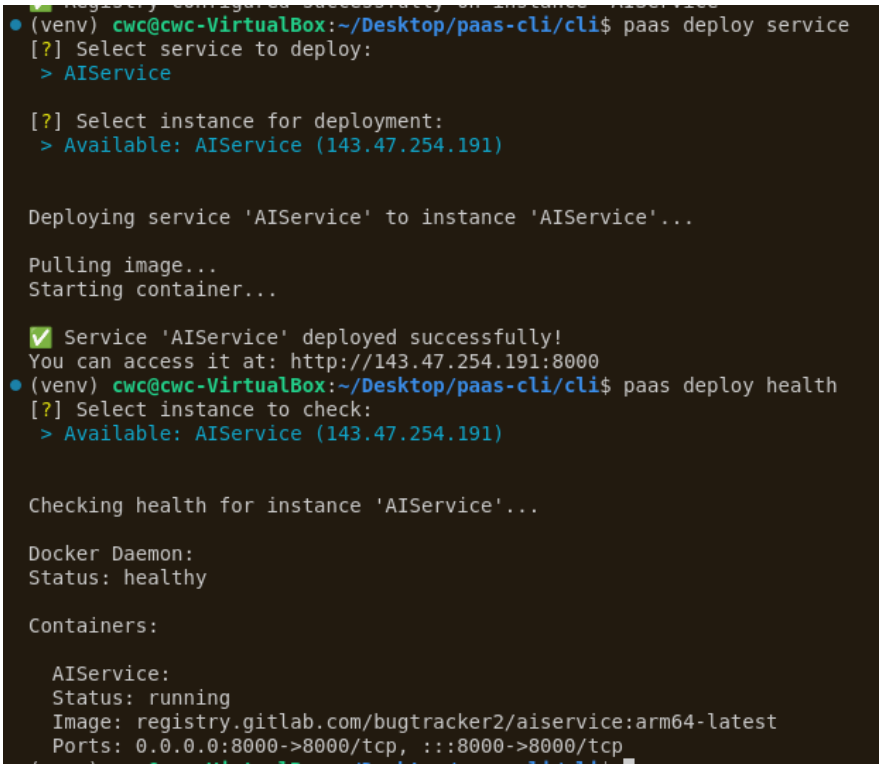

Рисунок 4.4 – Впровадження сервісу штучного інтелекту

З наведеного рисунку можна зробити висновок, що розроблений сервіс штучного інтелекту, який потрібен для логічної частини інтелектуальної системи, впроваджено успішно і він працює на локальній адресі машини за портом 8000.

Схожим чином виконано впровадження і інших сервісів, потрібних для роботи інтелектуальної системи: клієнтська частина (Frontend), серверна частина (Backend) та база даних.

У результаті впровадження користувачам стане доступним вебзастосунок для відстежування та управління ризиковими явищами, який матиме функції управління проектами, аналізу створених завдань за допомогою алгоритмів штучного інтелекту, створення аналітичних звітів у різних файлових форматах тощо.

ВИСНОВКИ

Мета дипломної роботи полягала в розробці та впровадженні хмарної платформи для інтелектуальної системи відслідковування й управління ризиковими явищами у програмних продуктах. На основі аналізу сучасних програмних рішень було сформульовано вимоги до системи та її функціональних компонентів.

Функціонал розробленої системи надає користувачам зручний інструмент для аналізу та управління ризиковими явищами. Інтелектуальна система дозволяє автоматично оцінювати відповідність опису ризикових явищ їх пріоритетам і серйозності, що сприяє оптимізації процесу їх обробки та класифікації. Інтеграція з хмарною платформою спрощує розгортання та використання вебзастосунку.

Створена хмарна платформа, що базується на Oracle Cloud, слугує спрощеним аналогом популярних PaaS і забезпечує надійне середовище для роботи системи. Вона дозволяє розгортати застосунок у хмарі з мінімальними технічними вимогами до користувача, що значно підвищує її доступність та зручність у використанні.

Хмарна платформа дозволяє знизити витрати на інфраструктуру завдяки автоматизації розгортання через GitLab CI/CD та раціональному використанню обчислювальних ресурсів. Такий підхід додатково сприяє зменшенню екологічного впливу завдяки оптимізації енергоспоживання.

Для реалізації вебзастосунку використано сучасний набір технологій: React для клієнтської частини, Node.js і Express для серверної, Sequelize для ORM і для управління реляційною базою даних. Інтелектуальні функції реалізовано за допомогою Python і моделей, навчених на спеціально відібраному наборі даних. Така комбінація технологій забезпечує зручність користувацького інтерфейсу та ефективність аналізу даних, створюючи високопродуктивну систему.

Використана архітектура вебзастосунку є масштабованою, що відкриває перспективи розширення функціоналу. Зокрема, це передбачає підтримку нових алгоритмів машинного навчання, інтеграцію з іншими хмарними сервісами для оптимізації процесів, а також масштабування бази даних для ефективної обробки великих обсягів звітів про ризикові явища.

Під час тестування впровадження було підтверджено стабільність роботи розробленої системи, що свідчить про надійність обраних методів проектування та реалізації. Автоматизація процесів розгортання завдяки GitLab CI/CD дозволяє мінімізувати людський фактор та підвищує продуктивність розробників.

Розроблений застосунок може бути інтегрований із популярними інструментами управління проектами, такими як Jira, GitHub, Trello тощо. Реалізація описаної інтеграції підвищить універсальність і зручність використання системи для широкої аудиторії, одночасно спрощуючи її впровадження в існуючі робочі процеси.

Розробка хмарної платформи та вебзастосунку продемонструвала можливість інтеграції сучасних технологій для створення ефективної системи управління ризиковими явищами, що відповідає потребам індустрії програмного забезпечення.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

loud computing: Concepts, technology, & architecture / ред.: P. R. 1974-, M. Zaigham. – [Б. м.]: Pearson, 2013. – 489 с.

elcome to Oracle Cloud Infrastructure [Електронний ресурс] // Oracle Cloud D

loud Native Infrastructure: Patterns for Scalable Infrastructure and Applications. – [Б. м.]: O'Reilly Media, 2017. – 160 с.

oftware testing: A craftsman's approach. – Boca Raton, [Florida] : CRC Press, Taylor & Francis Group, 2014. – 464 с.

ESTful API Design. – [Б. м.]: CreateSpace Independent Publishing Platform, 2016. – 294 с.

adden N. API Security in Action / Neil Madden. – [Б. м.] : Manning Publications Company, 2020. – 570 с.

tI. Bulletproof TLS and PKI, Second Edition: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications / Ivan Ristic. – [Б. м.] : Feisty Duck,

antis Documentation [Електронний ресурс] // Mantis Bug Tracker. – Режим доступу: екрана.

ocumentation Heroku Dev Center [Електронний ресурс] // Heroku Dev Center. – Режим доступу: https://devcenter.heroku.com/categories/reference (дата звернення: 16.10.2024). – Назва з екрана.

WS elastic beanstalk documentation [Електронний ресурс] // Docs Amazon Web Services. – Режим доступу: https://docs.aws.amazon.com/elastic-beanstalk/ (дата звернення: 16.10.2024). – Назва з екрана.

Reading, USA : Addison-Wesley, 1995. – 395 с.

ML @ Classroom: An Introduction to Object-Oriented Modeling. – [Б. м.] : Springer, 2015. – 220 с.

DK for Python [Електронний ресурс] // Oracle Cloud Infrastructure Documentation. – Режим доступу: https://docs.oracle.com/en-
з екрана.

architect by implementing effective architecture concepts. – [Б. м.] : Packt Publishing, 2018. – 594 с.
tephens R. Beginning Database Design Solutions: Understanding and Implementing Database Design Concepts for the Cloud / Rod Stephens. – [Б. м.] : Wiley & Sons,

arrington J. L. Relational database design and implementation: Clearly explained / Jan L. Harrington. – 3-тє вид. – Amsterdam : Morgan Kaufmann/Elsevier, 2009. – 420 с.
oach to Web Usability. – [Б. м.] : Pearson Education, 2014. – 216 с.
bout Face: The Essentials of Interaction Design / David Cronin [та ін.]. – [Б. м.] :

hart.js [Електронний ресурс] // Open source HTML5 Charts. – Режим доступу:

lick Documentation (8.1.x) [Електронний ресурс] // Welcome to Click. – Режим доступу: https://click.palletsprojects.com/en/stable/ (дата звернення: 03.11.2024). – Назва з екрана.
ane S. Docker – Up & Running: Shipping Reliable Containers in Production / Sean Kane. – 3-тє вид. – [Б. м.]: O'Reilly Media, 2023. – 330 с.
se GitLab [Електронний ресурс] // GitLab Documentation. – Режим доступу:

ocumentation [Електронний ресурс] // Let's Encrypt. – Режим доступу:

earning React: Modern Patterns for Developing React Apps. – [Б. м.] : O'Reilly Media, 2020. – 310 с.
appin N. Modern CSS with Tailwind: Flexible Styling without the Fuss / Noel Rappin. – [Б. м.] : Pragmatic Bookshelf, 2021. – 92 с.

ahn E. Express in Action: Writing, building, and testing Node.js applications / Evan Hahn. – [Б. м.] : Manning Publications, 2016. – 256 с.

equelize v6 [Електронний ресурс] // Feature-rich ORM for modern JavaScript. – Режим доступу: https://sequelize.org/docs/v6/ (дата звернення: 03.11.2024). – Назва з екрана.

ntroduction to Machine Learning with Python: A Guide for Data Scientists. – [Б. м.] : O'Reilly Media, 2016. – 376 с.

ira Issue Reports v1 [Електронний ресурс] // Kaggle: Your Machine Learning and D

yers G. J. The art of software testing / Glenford J. Myers. – 3-тє вид. – Hoboken, N.J

tting Started with Jest [Електронний ресурс] // Jest. – Режим доступу: (дата звернення: 20.11.2024). – Назва з екрана.

ДОДАТОК А
ВИХІДНИЙ КОД ХМАРНОЇ ПЛАТФОРМИ

одуль auth

```
import click
import hashlib
from ..core.config import load_config, save_config
from ..core.gitlab import GitLabManager, GitLabConfig
from ..utils.exceptions import PaasError
@click.group()
def auth():
    """Authentication commands"""
    pass
@auth.command(name="register")
@click.option("--username", prompt=True, help="Choose a username")
@click.option("--password", prompt=True, hide_input=True, help="Choose a password")
@click.option("--gitlab-url", prompt="Enter GitLab URL", default="https://gitlab.com",
              help="GitLab instance URL")
@click.option("--gitlab-token", prompt="Enter GitLab personal access token", hide_input=True,
              help="GitLab personal access token")
def register(username, password, gitlab_url, gitlab_token):
    """Register a new user and configure GitLab access"""
    try:
        config = load_config()

        if config.get("password hash"):
            raise PaasError("A user is already registered. You can only register once.")

        # register local user
        password_hash = hashlib.sha256(password.encode()).hexdigest()
        config["username"] = username
        config["password_hash"] = password_hash

        # configure GitLab access
        click.echo("\nConfiguring GitLab access...")
        gitlab_manager = GitLabManager()
        gitlab_config = gitlab_manager.login(gitlab_url, gitlab_token)

        # explicitly set logged_in status
        config["logged in"] = True

        # ensure GitLab configuration is properly saved
        config["gitlab"] = {
            "url": gitlab_url,
            "token": gitlab_token,
            "user_id": gitlab_config.user_id,
            "username": gitlab_config.username
```

```
        }

        # save the complete configuration
        save_config(config)

        click.echo("\n✅ Registration completed successfully!")
        click.echo(f"Local username: {username}")
        click.echo(f"GitLab user: {gitlab config.username}")
        click.echo("\nYou are now logged in and ready to use the CLI.")

    except Exception as e:
        raise PaasError(f"Registration failed: {str(e)}")

@auth.command(name="login")
@click.option("--username", prompt=True, help="Your username")
@click.option("--password", prompt=True, hide_input=True, help="Your password")
def login(username, password):
    """Log in to the CLI"""
    config = load_config()
    stored_hash = config.get("password_hash")

    if not stored_hash:
        raise PaasError("No user is registered. Please register first.")

    entered_hash = hashlib.sha256(password.encode()).hexdigest()
    if entered_hash == stored_hash:
        config["logged_in"] = True
        save_config(config)
        click.echo("✅ Login successful!")
    else:
        raise PaasError("Invalid username or password.")

@auth.command(name="logout")
def logout():
    """Log out from the CLI"""
    config = load_config()
    if config.get("logged_in"):
        config["logged_in"] = False
        save_config(config)
        click.echo("✅ Logged out successfully.")
    else:
        click.echo("You are not logged in.")

@auth.command(name="status")
def status():
    """Show current authentication status"""
    config = load_config()

    click.echo("\nAuthentication Status:")
    click.echo(f"Local user: {config.get('username', 'Not configured')}")
    click.echo(f"Logged in: {'Yes' if config.get('logged_in') else 'No'}")
```

```
    if 'gitlab' in config:
        click.echo("\nGitLab Configuration:")
        click.echo(f"URL: {config['gitlab'].get('url', 'Not configured')}")
        click.echo(f"Username: {config['gitlab'].get('username', 'Not configured')}")
    else:
        click.echo("\nGitLab: Not configured")
```

лас OCIInstance

```
class OCIInstance:
    def __init__(self, data: Dict):
        self.id = data.get('id')
        self.name = data.get('display_name')
        self.state = data.get('lifecycle_state')
        self.shape = data.get('shape')
        self.compartment_id = data.get('compartment_id')
        self.public_ip = data.get('public_ip')
        self.ssh_key_path = data.get('ssh_key_path')
        self.vcn_id = data.get('vcn_id')
        self.subnet_id = data.get('subnet_id')

    def to_dict(self) -> Dict:
        return {
            'id': self.id,
            'display_name': self.name,
            'lifecycle state': self.state,
            'shape': self.shape,
            'compartment id': self.compartment id,
            'public_ip': self.public_ip,
            'ssh_key_path': self.ssh_key_path,
            'vcn_id': self.vcn_id,
            'subnet_id': self.subnet_id
        }
```

лас VCN

```
from typing import Dict, List, Optional
import oci
import ipaddress
from ..utils.exceptions import PaasError
from ..core.settings import Settings

class VCNManager:
    def __init__(self, config: Dict):
        self.config = config
        self.network_client = oci.core.VirtualNetworkClient(self.config)
        self.compute_client = oci.core.ComputeClient(self.config)
```

```
        self.network_composite =
oci.core.VirtualNetworkClientCompositeOperations(self.network_client)
        self.settings = Settings()

    def create_vcn_with_options(self, compartment_id: str, name: str,
                                cidr_block: str = "10.0.0.0/16",
                                dns_label: Optional[str] = None,
                                create igw: bool = True) -> Dict:
        """Create VCN with additional options"""
        try:
            vcn_details = oci.core.models.CreateVcnDetails(
                compartment_id=compartment_id,
                display_name=name,
                cidr_block=cidr_block,
                dns label=dns label or name.replace("-", "")[:15]
            )
            response = self.network composite.create vcn and wait for state(
                vcn_details,
                wait_for_states=[oci.core.models.Vcn.LIFECYCLE_STATE_AVAILABLE]
            )
            # create internet gateway
            if create igw:
                igw = self.create_internet_gateway(
                    compartment id,
                    response.data.id,
                    f"{name}-igw"
                )
                self.update_route_table(response.data.default_route_table_id, igw.id)
            vcn data = {
                'id': response.data.id,
                'name': response.data.display_name,
                'cidr_block': response.data.cidr_block,
                'state': response.data.lifecycle_state,
                'dns_label': response.data.dns_label
            }

            self.settings.add_recent_vcn(vcn_data)
            return vcn_data
        except Exception as e:
            raise PaasError(f"Failed to create VCN: {str(e)}")

    def list_vcns(self, compartment_id: str) -> List[Dict]:
        """List all VCNs in compartment"""
        try:
            vcns = self.network_client.list_vcns(compartment_id).data
            return [
                {
                    'id': vcn.id,
                    'name': vcn.display_name,
                    'cidr_block': vcn.cidr_block,
                    'state': vcn.lifecycle_state
```

```
            }
            for vcn in vcns
        ]
    except Exception as e:
        raise PaasError(f"Failed to list VCNs: {str(e)}")

def list_subnets(self, compartment_id: str, vcn_id: str) -> List[Dict]:
    """List all subnets in VCN"""
    try:
        subnets = self.network client.list subnets(
            compartment_id,
            vcn_id=vcn_id
        ).data
        return [
            {
                'id': subnet.id,
                'name': subnet.display name,
                'cidr_block': subnet.cidr_block,
                'state': subnet.lifecycle_state
            }
            for subnet in subnets
        ]
    except Exception as e:
        raise PaasError(f"Failed to list subnets: {str(e)}")

def delete_vcn(self, vcn_id: str, force: bool = False) -> None:
    """Delete VCN and optionally its dependent resources"""
    try:
        # check VCN exists and get its details
        vcn = self.network_client.get_vcn(vcn_id).data
        if force:
            # delete dependent resources
            # delete subnets
            subnets = self.list_subnets(vcn.compartment_id, vcn_id)
            for subnet in subnets:
                self.network client.delete subnet(subnet['id'])
            # delete internet gateways
            igws = self.network_client.list_internet_gateways(
                vcn.compartment_id,
                vcn_id
            ).data
            for igw in igws:
                self.network_composite.delete_internet_gateway_and_wait_for_state(
                    igw.id,
                    wait_for_states=[oci.core.models.InternetGateway.LIFECYCLE_STATE_TERMINATED]
                )
            # delete security lists (except default)
            security_lists = self.network_client.list_security_lists(
                vcn.compartment_id,
                vcn_id
            ).data
```

```
            for sl in security_lists:
                if not sl.display_name.startswith('Default'):
                    self.network_client.delete_security_list(sl.id)
        # delete the VCN
        self.network_composite.delete_vcn_and_wait_for_state(
            vcn_id,
            wait_for_states=[oci.core.models.Vcn.LIFECYCLE_STATE_TERMINATED]
        )
    except Exception as e:
        raise PaasError(f"Failed to delete VCN: {str(e)}")

def create_subnet_with_options(self, compartment_id: str, vcn_id: str, name: str,
                               cidr_block: str, availability_domain: str,
                               prohibit_public_ip_on_vnic: bool = False) -> Dict:
    """Create subnet with additional options"""
    try:
        subnet details = oci.core.models.CreateSubnetDetails(
            compartment_id=compartment_id,
            vcn_id=vcn_id,
            display_name=name,
            cidr_block=cidr_block,
            availability_domain=availability_domain,
            dns_label=name.replace("-", "")[:15],
            prohibit_public_ip_on_vnic=prohibit_public_ip_on_vnic
        )
        response = self.network_composite.create_subnet_and_wait_for_state(
            subnet_details,
            wait_for_states=[oci.core.models.Subnet.LIFECYCLE_STATE_AVAILABLE]
        )
        subnet_data = {
            'id': response.data.id,
            'name': response.data.display_name,
            'cidr_block': response.data.cidr_block,
            'state': response.data.lifecycle_state,
            'prohibit_public_ip': response.data.prohibit_public_ip_on_vnic
        }

        self.settings.add_recent_subnet(subnet_data)
        return subnet_data
    except Exception as e:
        raise PaasError(f"Failed to create subnet: {str(e)}")

def create_security_list(self, compartment_id: str, vcn_id: str, name: str,
                         ingress_rules: List[Dict], egress_rules: List[Dict]) -> Dict:
    """Create security list with specified rules"""
    try:
        ingress = [
            oci.core.models.IngressSecurityRule(
                protocol=rule['protocol'],
                source=rule['source'],
                source_type=rule.get('source_type', 'CIDR_BLOCK'),
```

```
                    tcp_options=oci.core.models.TcpOptions(
                        destination_port_range=oci.core.models.PortRange(
                            min=rule['tcp_options']['destination_port_range']['min'],
                            max=rule['tcp_options']['destination_port_range']['max']
                        )
                    ) if rule.get('tcp_options') else None,
                    is_stateless=rule.get('is_stateless', False),
                    description=rule.get('description', '')
                )
                for rule in ingress_rules
            ]
            egress = [
                oci.core.models.EgressSecurityRule(
                    protocol=rule['protocol'],
                    destination=rule['destination'],
                    destination_type=rule.get('destination_type', 'CIDR_BLOCK'),
                    tcp_options=None,
                    udp_options=None,
                    icmp_options=None,
                    is_stateless=rule.get('is_stateless', False),
                    description=rule.get('description', '')
                )
                for rule in egress_rules
            ]
            security_list_details = oci.core.models.CreateSecurityListDetails(
                compartment_id=compartment_id,
                vcn_id=vcn_id,
                display_name=name,
                ingress_security_rules=ingress,
                egress_security_rules=egress
            )
            response = self.network_client.create_security_list(security_list_details)
            return {
                'id': response.data.id,
                'name': response.data.display_name,
                'state': response.data.lifecycle_state
            }
        except Exception as e:
            raise PaasError(f"Failed to create security list: {str(e)}")

    @staticmethod
    def _validate_cidr(cidr: str) -> bool:
        """Validate CIDR block format"""
        try:
            ipaddress.ip_network(cidr)
            return True
        except ValueError:
            return False

    @staticmethod
    def _is_subnet_of(subnet_cidr: str, vcn_cidr: str) -> bool:
```

```
        """Check if subnet CIDR is within VCN CIDR"""
        try:
            return ipaddress.ip_network(subnet_cidr).subnet_of(
                ipaddress.ip_network(vcn_cidr)
            )
        except ValueError:
            return False

    def create_internet_gateway(self, compartment_id: str, vcn_id: str, name: str):
        """Create internet gateway"""
        igw_details = oci.core.models.CreateInternetGatewayDetails(
            compartment_id=compartment_id,
            vcn_id=vcn_id,
            display_name=name,
            is_enabled=True
        )
        response = self.network_composite.create_internet_gateway_and_wait_for_state(
            igw_details,
            wait_for_states=[oci.core.models.InternetGateway.LIFECYCLE_STATE_AVAILABLE]
        )
        return response.data

    def update_route_table(self, rt_id: str, igw_id: str):
        """Update route table with internet gateway route"""
        route_rules = [
            oci.core.models.RouteRule(
                destination='0.0.0.0/0',
                destination_type='CIDR_BLOCK',
                network_entity_id=igw_id
            )
        ]
        update_details = oci.core.models.UpdateRouteTableDetails(route_rules=route_rules)
        return self.network_client.update_route_table(rt_id, update_details).data
```

лас OCIManager

```
class OCIManager:
    def __init__(self, config: Dict):
        self.config = config
        self.compute_client = oci.core.ComputeClient(self.config)
        self.network_client = oci.core.VirtualNetworkClient(self.config)
        self.identity_client = oci.identity.IdentityClient(self.config)
        self.compute_composite = oci.core.ComputeClientCompositeOperations(self.compute_client)
        self.network_composite =
oci.core.VirtualNetworkClientCompositeOperations(self.network_client)

    def get_availability_domains(self, compartment_id: str) -> List[str]:
        """Get list of availability domains"""
```

```
        return [ad.name for ad in
self.identity_client.list_availability_domains(compartment_id).data]

    def create_vcn(self, compartment_id: str, name: str):
        """Create Virtual Cloud Network"""
        vcn_details = oci.core.models.CreateVcnDetails(
            compartment_id=compartment_id,
            display name=f"{name}-vcn",
            cidr_block="10.0.0.0/16",
            dns label=name.replace("-", "")[:15]
        )

        response = self.network_composite.create_vcn_and_wait_for_state(
            vcn_details,
            wait for states=[oci.core.models.Vcn.LIFECYCLE STATE AVAILABLE]
        )
        return response.data

    def create_subnet(self, compartment_id: str, vcn_id: str, ad_name: str, name: str):
        """Create subnet in VCN"""
        subnet_details = oci.core.models.CreateSubnetDetails(
            compartment id=compartment id,
            vcn_id=vcn_id,
            availability domain=ad name,
            display_name=f"{name}-subnet",
            cidr_block="10.0.1.0/24",
            dns label=name.replace("-", "")[:15]
        )

        response = self.network_composite.create_subnet_and_wait_for_state(
            subnet_details,
            wait_for_states=[oci.core.models.Subnet.LIFECYCLE_STATE_AVAILABLE]
        )
        return response.data

    def create internet gateway(self, compartment id: str, vcn id: str, name: str):
        """Create internet gateway"""
        igw_details = oci.core.models.CreateInternetGatewayDetails(
            compartment_id=compartment_id,
            vcn_id=vcn_id,
            display_name=f"{name}-igw",
            is_enabled=True
        )

        response = self.network_composite.create_internet_gateway_and_wait_for_state(
            igw_details,
            wait_for_states=[oci.core.models.InternetGateway.LIFECYCLE_STATE_AVAILABLE]
        )
        return response.data

    def update_route_table(self, rt_id: str, igw_id: str):
```

```
        """Update route table with internet gateway route"""
        route_rules = [
            oci.core.models.RouteRule(
                destination='0.0.0.0/0',
                destination_type='CIDR_BLOCK',
                network_entity_id=igw_id
            )
        ]

        update_details = oci.core.models.UpdateRouteTableDetails(route_rules=route_rules)
        return self.network_client.update_route_table(rt_id, update_details).data

    def get_ubuntu_image(self, compartment_id: str):
        """Get Ubuntu 20.04 image"""
        # only 20.04 LTS is supported by OCI library, no clue why
        images = self.compute_client.list_images(
            compartment_id,
            operating_system="Canonical Ubuntu",
            operating_system_version="20.04"
        ).data
        return next((img for img in images if "20.04" in img.display_name), None)

    def create_instance(self, compartment_id: str, subnet_id: str, ad_name: str,
                        name: str, image_id: str, ssh_key: str):
        """Create compute instance"""
        # selecting A1 Ampere by default, because it's in a free tier
        instance_details = oci.core.models.LaunchInstanceDetails(
            compartment_id=compartment_id,
            availability_domain=ad_name,
            display_name=f"{name}-instance",
            shape="VM.Standard.A1.Flex",
            shape_config=oci.core.models.LaunchInstanceShapeConfigDetails(
                ocpus=1,
                memory_in_gbs=6
            ),
            subnet_id=subnet_id,
            source_details=oci.core.models.InstanceSourceViaImageDetails(
                source_type="image",
                image_id=image_id
            ),
            metadata={
                "ssh_authorized_keys": ssh_key
            }
        )

        response = self.compute_composite.launch_instance_and_wait_for_state(
            instance_details,
            wait_for_states=[oci.core.models.Instance.LIFECYCLE_STATE_RUNNING]
        )
        return response.data
```

```
def get_instance_ip(self, instance_id: str, max_attempts: int = 10) -> str:
    """Get instance public IP with retries"""
    for attempt in range(max_attempts):
        try:
            vnic_attachments = self.compute_client.list_vnic_attachments(
                compartment_id=self.config['tenancy'],
                instance_id=instance_id
            ).data

            if vnic attachments:
                vnic = self.network_client.get_vnic(vnic_attachments[0].vnic_id).data
                if vnic.public_ip:
                    return vnic.public_ip

            if attempt < max attempts - 1:
                time.sleep(5)

        except Exception as e:
            if attempt < max_attempts - 1:
                time.sleep(5)
            else:
                raise e

    return "Unknown"

def terminate_instance(self, instance_id: str):
    """Terminate compute instance"""
    return self.compute_composite.terminate_instance_and_wait_for_state(
        instance id,
        wait_for_states=[oci.core.models.Instance.LIFECYCLE_STATE_TERMINATED]
    )

def delete_subnet(self, subnet_id: str):
    """Delete subnet"""
    return self.network_composite.delete_subnet_and_wait_for_state(
        subnet id,
        wait_for_states=[oci.core.models.Subnet.LIFECYCLE_STATE_TERMINATED]
    )

def delete_internet_gateway(self, igw_id: str):
    """Delete internet gateway"""
    return self.network_composite.delete_internet_gateway_and_wait_for_state(
        igw_id,
        wait_for_states=[oci.core.models.InternetGateway.LIFECYCLE_STATE_TERMINATED]
    )

def delete_vcn(self, vcn_id: str):
    """Delete VCN"""
    return self.network_composite.delete_vcn_and_wait_for_state(
        vcn_id,
        wait_for_states=[oci.core.models.Vcn.LIFECYCLE_STATE_TERMINATED]
```

```
        )
```

лас InstanceManager

```
from pathlib import Path
from typing import Dict, List, Optional
from ..core.config import load_config, save_config
from .oci import OCIInstance

class InstanceManager:
    def __init__(self):
        self.instances = {}
        self._load_instances()

    def _load_instances(self):
        config = load_config()
        self.instances = {}
        for name, data in config.get('instances', {}).items():
            try:
                self.instances[str(name)] = OCIInstance(data)
            except Exception as e:
                print(f"Warning: Failed to load instance {name}: {e}")

    def save_instance(self, name: str, instance: OCIInstance):
        config = load config()
        if 'instances' not in config:
            config['instances'] = {}
        config['instances'][str(name)] = instance.to_dict()
        save_config(config)
        self. load instances()

    def remove_instance(self, name: str):
        config = load_config()

        # get instance data before removing
        instance = self.get_instance(name)

        if 'instances' in config and str(name) in config['instances']:
            # remove SSH keys if they exist
            if instance and instance.ssh_key_path:
                try:
                    private_key = Path(instance.ssh_key_path)
                    public_key = Path(f"{instance.ssh_key_path}.pub")
                    if private_key.exists():
                        private_key.unlink()
                    if public key.exists():
                        public_key.unlink()
                except Exception as e:
                    print(f"Warning: Could not remove SSH keys: {e}")
```

```
            del config['instances'][str(name)]
            save_config(config)
            self._load_instances()

    def get_instance(self, name: str) -> Optional[OCIInstance]:
        return self.instances.get(str(name))

    def list_instances(self) -> List[OCIInstance]:
        return list(self.instances.values())
```

лас ClusterNode

```
from typing import Dict, List, Optional, Set
from pathlib import Path
from .instances import InstanceManager
from .loadbalancer import LoadBalancerManager
from .oci import OCIInstance, OCIManager
from ..utils.exceptions import PaasError
from ..core.config import load_config, save_config
from ..core.sshmanager import SSHKeyManager

class ClusterNode(OCIInstance):
    def __init__(self, data: Dict):
        super().__init__(data)
        self.node_type = data.get('node_type', 'worker')
        self.cluster_name = data.get('cluster_name')
        self.service_type = data.get('service_type')

    def to_dict(self) -> Dict:
        data = super().to_dict()
        data.update({
            'node_type': self.node_type,
            'cluster_name': self.cluster_name,
            'service_type': self.service_type
        })
        return data
```

лас Cluster

```
class Cluster:
    def __init__(self, name: str, service_type: str):
        self.name = name
        self.service_type = service_type
        self.nodes: List[ClusterNode] = []
        self.load_balancer_id: Optional[str] = None
        self.status = 'initializing'

    def to_dict(self) -> Dict:
        return {
```

```
            'name': self.name,
            'service_type': self.service_type,
            'nodes': [node.to_dict() for node in self.nodes],
            'load_balancer_id': self.load_balancer_id,
            'status': self.status
        }

    @classmethod
    def from_dict(cls, data: Dict) -> 'Cluster':
        cluster = cls(data['name'], data['service type'])
        cluster.nodes = [ClusterNode(node_data) for node_data in data['nodes']]
        cluster.load_balancer_id = data.get('load_balancer_id')
        cluster.status = data.get('status', 'unknown')
        return cluster

    def add_node(self, node: ClusterNode):
        """Add a node to the cluster"""
        self.nodes.append(node)

    def remove_node(self, node_name: str):
        """Remove a node from the cluster by name"""
        self.nodes = [node for node in self.nodes if node.name != node_name]
```

лас ClusterManager

```
class ClusterManager:
    def __init__(self, config: Dict):
        self.config = config
        self.instance_manager = InstanceManager()
        self.oci_manager = OCIManager(config)
        self.lb_manager = LoadBalancerManager(config)
        self.ssh_key_manager = SSHKeyManager()

    def _save_cluster(self, cluster: Cluster):
        """Save cluster configuration"""
        config = load_config()  # use the imported load_config
        if 'clusters' not in config:
            config['clusters'] = {}
        config['clusters'][cluster.name] = cluster.to_dict()
        save_config(config)  # use the imported save_config

    def _generate_ssh_key(self, name: str) -> str:
            """Generate SSH key for node and return public key content"""
            try:
                # force no passphrase and non-interactive key generation
                private_key, public_key = self.ssh_key_manager.generate_key(
                    name=name,
                    encrypted=False  # this ensures no passphrase
                )
```

```
            # read and return the public key content
            with public_key.open() as f:
                return f.read().strip()

        except Exception as e:
            raise PaasError(f"Failed to generate SSH key for {name}: {str(e)}")

def  cleanup partial cluster(self, node names: List[str], lb id: Optional[str]):
    """Clean up partially created cluster resources"""
    cleanup errors = []

    # delete load balancer if it was created
    if lb_id:
        try:
            print("Deleting load balancer...")
            self.lb_manager.delete_load_balancer(lb_id)
        except Exception as e:
            cleanup_errors.append(f"Failed to delete load balancer: {str(e)}")

    # delete created nodes
    for node_name in node_names:
        try:
            print(f"Deleting node {node_name}...")
            self.instance manager.remove instance(node name)
        except Exception as e:
            cleanup_errors.append(f"Failed to delete node {node_name}: {str(e)}")

    if cleanup_errors:
        print("\nWarning: Some cleanup operations failed:")
        for error in cleanup_errors:
            print(f"- {error}")

def _load_cluster_config(self, name: str) -> Optional[Dict]:
    """Load cluster configuration"""
    config = load_config()  # use the imported load_config
    return config.get('clusters', {}).get(name)

def _load_all_clusters(self) -> List[Dict]:
    """Load all cluster configurations"""
    config = load_config()  # use the imported load_config
    return list(config.get('clusters', {}).values())

def _delete_cluster_config(self, name: str):
    """Delete cluster configuration"""
    config = load_config()  # use the imported load_config
    if 'clusters' in config and name in config['clusters']:
        del config['clusters'][name]
        save_config(config)  # use the imported save_config

def create_cluster(self, compartment_id: str, name: str, service_type: str,
                   node_count: int, vcn_id: str, subnet_id: str) -> Cluster:
```

```
"""Create a new cluster with specified number of nodes"""
created_nodes: List[str] = []  # track created node names
created_lb_id: Optional[str] = None  # track created load balancer ID

try:
    if self.get_cluster(name):
        raise PaasError(f"Cluster {name} already exists")

    if node_count < 1:
        raise PaasError("Node count must be at least 1")

    print(f"\nCreating cluster '{name}' with {node_count} nodes...")
    cluster = Cluster(name, service_type)

    # create nodes
    for i in range(node_count):
        node name = f"{name}-node-{i+1}"
        print(f"\nCreating node {i+1} of {node_count}: {node_name}")

        # generate SSH key for the node
        print(f"Generating SSH key for {node_name}...")
        ssh key = self. generate ssh key(node name)

        print(f"Creating compute instance...")
        instance = self.oci_manager.create_instance(
            compartment_id=compartment_id,
            subnet id=subnet id,
            ad_name=self.oci_manager.get_availability_domains(compartment_id)[0],
            name=node name,
            image_id=self.oci_manager.get_ubuntu_image(compartment_id).id,
            ssh_key=ssh_key
        )

        # update instance data with SSH key path and cluster info
        node_data = instance.to_dict()
        node data.update({
            'node_type': 'worker',
            'cluster_name': name,
            'service_type': service_type,
            'ssh_key_path': str(Path.home() / '.paas' / 'ssh' / f'{node_name}_key')
        })

        node = ClusterNode(node_data)
        cluster.add node(node)
        self.instance_manager.save_instance(node_name, node)
        created_nodes.append(node_name)  # Track the created node
        print(f"Node {node_name} created successfully")

    # create load balancer if needed
    if service_type in ['backend']:
        print("\nCreating load balancer...")
```

```
                lb = self.lb_manager.create_load_balancer(
                    compartment_id=compartment_id,
                    subnet_id=subnet_id,
                    display_name=f"{name}-lb",
                    backend_nodes=cluster.nodes
                )
                cluster.load_balancer_id = lb['id']
                created_lb_id = lb['id']  # track the created load balancer
                print("Load balancer created successfully")

            cluster.status = 'running'
            self._save_cluster(cluster)

            print("\nCluster created successfully!")
            return cluster

        except Exception as e:
            # cleanup on failure
            print("\nError occurred, cleaning up...")
            self._cleanup_partial_cluster(created_nodes, created_lb_id)
            raise PaasError(f"Failed to create cluster: {str(e)}")

    def delete_cluster(self, name: str):
        """Delete cluster and all its resources"""
        try:
            cluster = self.get_cluster(name)
            if not cluster:
                raise PaasError(f"Cluster {name} not found")

            print(f"\nDeleting cluster '{name}'...")

            # delete load balancer if exists
            if cluster.load_balancer_id:
                print("Deleting load balancer...")
                self.lb_manager.delete_load_balancer(cluster.load_balancer_id)

            # delete all nodes
            for node in cluster.nodes:
                print(f"Deleting node {node.name}...")
                self.instance_manager.remove_instance(node.name)

            self._delete_cluster_config(name)
            print("Cluster deleted successfully")

        except Exception as e:
            raise PaasError(f"Failed to delete cluster: {str(e)}")

    def get_cluster(self, name: str) -> Optional[Cluster]:
        """Get cluster details"""
        try:
            cluster_data = self._load_cluster_config(name)
```

```
            return Cluster.from_dict(cluster_data) if cluster_data else None
        except Exception as e:
            raise PaasError(f"Failed to get cluster: {str(e)}")

    def list_clusters(self) -> List[Cluster]:
        """List all clusters"""
        try:
            clusters_data = self._load_all_clusters()
            return [Cluster.from_dict(data) for data in clusters_data]
        except Exception as e:
            raise PaasError(f"Failed to list clusters: {str(e)}")

    def scale_cluster(self, name: str, new_size: int) -> Cluster:
        """Scale cluster to specified number of nodes"""
        try:
            cluster = self.get_cluster(name)
            if not cluster:
                raise PaasError(f"Cluster {name} not found")

            if new_size < 1:
                raise PaasError("Node count must be at least 1")

            current_size = len(cluster.nodes)
            if new_size == current_size:
                return cluster

            if new_size > current_size:
                # add nodes
                for i in range(current_size, new_size):
                    node_name = f"{name}-node-{i+1}"
                    instance = self.oci_manager.create_instance(
                        compartment_id=cluster.nodes[0].compartment_id,
                        subnet_id=cluster.nodes[0].subnet_id,
                        ad_name=self.oci_manager.get_availability_domains(
                            cluster.nodes[0].compartment_id
                        )[0],
                        name=node_name,
                        image_id=self.oci_manager.get_ubuntu_image(
                            cluster.nodes[0].compartment_id
                        ).id,
                        ssh_key=self._generate_ssh_key(node_name)
                    )

                    node_data = instance.to_dict()
                    node_data.update({
                        'node_type': 'worker',
                        'cluster_name': name,
                        'service_type': cluster.service_type
                    })

                    node = ClusterNode(node_data)
```

```
                    cluster.add_node(node)
                    self.instance_manager.save_instance(node_name, node)

                    # add to load balancer if exists
                    if cluster.load_balancer_id:
                        self.lb_manager.add_backend(
                            cluster.load_balancer_id,
                            'primary-backend-set',
                            node.public_ip
                        )
            else:
                # remove nodes
                nodes_to_remove = cluster.nodes[new_size:]
                for node in nodes_to_remove:
                    # remove from load balancer if exists
                    if cluster.load_balancer_id:
                        self.lb manager.remove backend(
                            cluster.load_balancer_id,
                            'primary-backend-set',
                            node.public ip
                        )
                    self.instance manager.remove instance(node.name)
                    cluster.remove_node(node.name)

            self._save_cluster(cluster)
            return cluster

        except Exception as e:
            raise PaasError(f"Failed to scale cluster: {str(e)}")
```

лас LogCollector

```
from typing import Dict, List, Optional, Union
from pathlib import Path
import json
import datetime
from .sshmanager import SSHConnection
from .instances import InstanceManager
from .cluster import Cluster, ClusterNode
from ..utils.exceptions import PaasError

class LogCollector:
    def __init__(self, log_dir: Optional[str] = None):
        """Initialize LogCollector with optional custom log directory"""
        self.log_dir = Path(log_dir) if log_dir else Path.home() / '.paas' / 'logs'
        self.log_dir.mkdir(parents=True, exist_ok=True)
        self.instance_manager = InstanceManager()

    def _get_ssh_connection(self, host: str, username: str, key_path: str) -> SSHConnection:
```

```
    """Create and verify SSH connection"""
    try:
        ssh = SSHConnection(
            host=host,
            username=username,
            key_path=key_path
        )
        if not ssh.test_connection():
            raise PaasError(f"Failed to establish SSH connection to {host}")
        return ssh
    except Exception as e:
        raise PaasError(f"SSH connection error: {str(e)}")

def _collect_system_logs(self, ssh: SSHConnection) -> Dict:
    """Collect system-level logs"""
    logs = {}

    # collect system logs
    for log_file in ['/var/log/syslog', '/var/log/dmesg']:
        try:
            result = ssh.run_command(f'tail -n 1000 {log_file}')
            logs[log_file] = result.stdout
        except Exception as e:
            logs[log_file] = f"Error collecting log: {str(e)}"

    # collect system metrics
    try:
        metrics = {}
        # CPU info
        cpu = ssh.run_command("top -bn1 | grep 'Cpu(s)'")
        metrics['cpu'] = cpu.stdout

        # memory info
        mem = ssh.run_command("free -m")
        metrics['memory'] = mem.stdout

        # disk usage
        disk = ssh.run_command("df -h")
        metrics['disk'] = disk.stdout

        logs['system_metrics'] = metrics
    except Exception as e:
        logs['system_metrics'] = f"Error collecting metrics: {str(e)}"

    return logs

def _collect_docker_logs(self, ssh: SSHConnection) -> Dict:
    """Collect Docker-related logs"""
```

```
        logs = {}

        try:
            # current containers
            containers = ssh.run_command(
                "docker ps --format '{{.Names}}'"
            ).stdout.splitlines()

            # logs for each container
            for container in containers:
                container = container.strip()
                if container:
                    logs[container] = {
                        'logs': ssh.run_command(
                            f"docker logs --tail 1000 {container}"
                        ).stdout,
                        'status': ssh.run_command(
                            f"docker inspect {container}"
                        ).stdout
                    }

        except Exception as e:
            logs['error'] = f"Failed to collect Docker logs: {str(e)}"

        return logs

    def collect_instance_logs(self, instance_name: str) -> Dict:
        """Collect logs from a specific instance"""
        try:
            instance = self.instance_manager.get_instance(instance_name)
            if not instance:
                raise PaasError(f"Instance '{instance_name}' not found")

            ssh = self._get_ssh_connection(
                host=instance.public_ip,
                username='ubuntu',
                key_path=instance.ssh_key_path
            )

            logs = {
                'timestamp': datetime.datetime.now().isoformat(),
                'instance': instance_name,
                'system': self._collect_system_logs(ssh),
                'docker': self._collect_docker_logs(ssh)
            }

            # save logs
```

```
            log_file = self.log_dir /
f"{instance_name}_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
            with open(log_file, 'w') as f:
                json.dump(logs, f, indent=2)

            return logs

        except Exception as e:
            raise PaasError(f"Failed to collect logs for instance {instance_name}: {str(e)}")

    def collect_cluster_logs(self, cluster: Cluster) -> Dict:
        """Collect logs from all nodes in a cluster"""
        logs = {
            'timestamp': datetime.datetime.now().isoformat(),
            'cluster_name': cluster.name,
            'nodes': {}
        }

        for node in cluster.nodes:
            try:
                logs['nodes'][node.name] = self.collect_instance_logs(node.name)
            except Exception as e:
                logs['nodes'][node.name] = f"Failed to collect logs: {str(e)}"

        # save cluster logs
        log_file = self.log_dir /
f"cluster_{cluster.name}_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
        with open(log_file, 'w') as f:
            json.dump(logs, f, indent=2)

        return logs

    def get_recent_logs(self, instance_name: Optional[str] = None, hours: int = 24) -> List[Dict]:
        """Get recent logs for an instance or all instances"""
        try:
            cutoff_time = datetime.datetime.now() - datetime.timedelta(hours=hours)
            logs = []

            for log_file in self.log_dir.glob('*.json'):
                # check if file is recent enough
                if datetime.datetime.fromtimestamp(log_file.stat().st_mtime) < cutoff_time:
                    continue

                # check if it matches instance name if specified
                if instance_name and not log_file.name.startswith(f"{instance_name}_"):
                    continue

                try:
```

```
                with open(log_file) as f:
                    logs.append(json.load(f))
            except Exception as e:
                logs.append({
                    'file': str(log_file),
                    'error': f"Failed to read log file: {str(e)}"
                })

        return logs

    except Exception as e:
        raise PaasError(f"Failed to get recent logs: {str(e)}")

def export_logs(self, output_dir: str, instance_name: Optional[str] = None) -> str:
    """Export logs to a specific directory"""
    try:
        output_path = Path(output_dir)
        output_path.mkdir(parents=True, exist_ok=True)

        # get logs to export
        logs = self.get_recent_logs(instance_name)

        # create export file
        timestamp = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
        export_file = output_path / f"logs_export_{timestamp}.json"

        with open(export_file, 'w') as f:
            json.dump(logs, f, indent=2)

        return str(export_file)

    except Exception as e:
        raise PaasError(f"Failed to export logs: {str(e)}")

def set_retention_days(self, days: int):
    """Set log retention period and clean up old logs"""
    try:
        cutoff_time = datetime.datetime.now() - datetime.timedelta(days=days)

        for log_file in self.log_dir.glob('*.json'):
            if datetime.datetime.fromtimestamp(log_file.stat().st_mtime) < cutoff_time:
                log_file.unlink()

    except Exception as e:
        raise PaasError(f"Failed to set retention period: {str(e)}")
```

ДОДАТОК Б
ВИХІДНИЙ КОД ЛОГІЧНОЇ ЧАСТИНИ

```
const { Model } = require("sequelize");

const { logger } = require("@utils/logger");

module.exports = (sequelize, DataTypes) => {
  class Project extends Model {
    static associate(models) {
      Project.hasMany(models.Issue, {
        foreignKey: "projectId",
        as: "issues",
        onDelete: "CASCADE",
      });
      Project.belongsToMany(models.User, {
        through: models.ProjectMember,
        foreignKey: "projectId",
        as: "members",
        onDelete: "CASCADE",
      });
    }
  }

  Project.init(
    {
      id: {
        type: DataTypes.UUID,
        defaultValue: DataTypes.UUIDV4,
        primaryKey: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      key: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
        validate: {
          len: [2, 10],
          isUppercase: true,
        },
      },
```

```
    description: {
      type: DataTypes.TEXT,
      allowNull: true,
    },
    settings: {
      type: DataTypes.JSONB,
      defaultValue: {
        autoAssignment: {
          enabled: true,
          maxIssuesPerUser: 5,
          considerExpertise: true,
          weightFactors: {
            expertise: 0.4,
            workload: 0.3,
            performance: 0.3,
          },
        },
        ai: {
          enabled: false,
          provider: "local",
          apiKey: null,
          features: {
            type: true,
            severity: true,
            priority: true,
          },
        },
        notifications: {
          issues: true,
          comments: true,
          members: true,
        },
      },
      allowNull: false,
    },
  },
  {
    sequelize,
    modelName: "Project",
    timestamps: true,
  }
);

// instance methods for better settings management
Project.prototype.getSetting = function (path) {
  logger.debug("Getting setting for path:", path);
  logger.debug("Current settings:", this.settings);
```

```
  const parts = path.split(".");
  let value = this.settings;

  for (const part of parts) {
    if (!value || typeof value !== "object") {
      logger.warn("Value not found for path:", path);
      return null;
    }
    value = value[part];
  }

  logger.debug("Found value:", value);
  return value;
};

Project.prototype.updateSetting = async function (path, value) {
  logger.debug("Updating setting:", { path, value });
  logger.debug(
    "Current settings before update:",
    JSON.stringify(this.settings, null, 2)
  );

  try {
    const parts = path.split(".");
    const lastPart = parts.pop();

    // create a deep copy of current settings
    const newSettings = JSON.parse(JSON.stringify(this.settings));
    let pointer = newSettings;

    // navigate to the correct nesting level
    for (const part of parts) {
      if (!pointer[part]) {
        pointer[part] = {};
      }
      pointer = pointer[part];
    }

    // update the value
    pointer[lastPart] = value;

    logger.debug(
      "New settings object:",
      JSON.stringify(newSettings, null, 2)
    );

    // update in database
    const _result = await this.update(
```

```
        {
          settings: newSettings,
        },
        {
          returning: true,
          plain: true,
        }
      );

      // update the instance itself
      this.settings = newSettings;

      logger.debug(
        "Settings after update:",
        JSON.stringify(this.settings, null, 2)
      );
      return this.settings;
    } catch (error) {
      console.error("Error updating settings:", error);
      throw error;
    }
  };

  Project.prototype.isFeatureEnabled = function (feature) {
    return !!this.getSetting(`ai.features.${feature}`);
  };

  Project.prototype.toggleFeature = async function (feature, enabled) {
    return await this.updateSetting(`ai.features.${feature}`, enabled);
  };

  Project.prototype.getNotificationSettings = function () {
    return this.settings.notifications;
  };

  Project.prototype.getAutoAssignmentConfig = function () {
    return this.settings.autoAssignment;
  };

  return Project;
};



const db = require("@models");
const { logger } = require("@utils/logger");
const Project = db.Project;
const ProjectMember = db.ProjectMember;
```

```
const Issue = db.Issue;
const User = db.User;
const AIService = require("@services/NLP-service");

const updateProjectAiSettings = async (req, res, next) => {
  try {
    const { projectId } = req.params;
    const userId = req.user.userId;
    const { enabled, provider, apiKey, features } = req.body;

    logger.debug("Received AI settings update:", {
      enabled,
      provider,
      features,
    });

    const projectMember = await ProjectMember.findOne({
      where: {
        projectId,
        userId,
        role: "ADMIN",
      },
    });

    if (!projectMember) {
      return res.status(403).json({ message: "Unauthorized" });
    }

    const project = await Project.findByPk(projectId);
    if (!project) {
      return res.status(404).json({ message: "Project not found" });
    }

    // update entire AI settings object at once to maintain consistency
    const newAiSettings = {
      enabled,
      provider,
      apiKey: provider === "openai" ? apiKey : null,
      features,
    };

    logger.debug("Updating AI settings to:", newAiSettings);

    // update settings in a single operation
    await project.updateSetting("ai", newAiSettings);

    // fetch fresh project data to verify update
    const updatedProject = await Project.findByPk(projectId);
```

```
    logger.debug("Updated project AI settings:", updatedProject.settings.ai);

    // return sanitized settings
    const responseSettings = {
      enabled: updatedProject.getSetting("ai.enabled"),
      provider: updatedProject.getSetting("ai.provider"),
      features: updatedProject.getSetting("ai.features"),
    };

    res.json({
      message: "AI settings updated successfully",
      settings: responseSettings,
    });
  } catch (error) {
    console.error("Error updating AI settings:", error);
    next(error);
  }
};

const getProjectAiSettings = async (req, res, next) => {
  try {
    const { projectId } = req.params;
    const userId = req.user.userId;

    const projectMember = await ProjectMember.findOne({
      where: { projectId, userId },
    });

    if (!projectMember) {
      return res.status(403).json({ message: "Unauthorized" });
    }

    const project = await Project.findByPk(projectId);
    if (!project) {
      return res.status(404).json({ message: "Project not found" });
    }

    // get all AI settings at once
    const settings = project.getSetting("ai");

    // return sanitized response
    const responseSettings = {
      enabled: settings?.enabled ?? false,
      provider: settings?.provider ?? "local",
      features: settings?.features ?? {
        severity: true,
        complexity: true,
      },
```

```
    };

    res.json(responseSettings);
  } catch (error) {
    console.error("Error getting AI settings:", error);
    next(error);
  }
};

const applyProjectAiSettings = async (req, res, next) => {
  try {
    const projectId = req.params.projectId || req.body.projectId;
    if (!projectId) {
      return next();
    }
    const project = await Project.findByPk(projectId);
    if (!project) {
      return next();
    }
    const aiEnabled = project.getSetting("ai.enabled");
    if (!aiEnabled) {
      return next();
    }
    // add AI settings to request object
    req.aiSettings = project.getSetting("ai");
    next();
  } catch (error) {
    console.error("Error in AI settings middleware:", error);
    next(error);
  }
};
const analyzeProject = async (req, res, next) => {
  try {
    const { projectId } = req.params;
    const userId = req.user.userId;
    const project = await Project.findByPk(projectId);
    if (!project) {
      return res.status(404).json({ message: "Project not found" });
    }
    const aiEnabled = project.getSetting("ai.enabled");
    if (!aiEnabled) {
      return res
        .status(400)
        .json({ message: "AI analysis is not enabled for this project" });
    }
    // check permissions
    const projectMember = await ProjectMember.findOne({
      where: { projectId, userId },
```

```
});

if (!projectMember) {
  return res.status(403).json({ message: "Unauthorized" });
}

// get project with issues
const projectWithIssues = await Project.findOne({
  where: { id: projectId },
  include: [
    {
      model: Issue,
      as: "issues",
      include: [
        {
          model: User,
          as: "assignee",
          attributes: ["id", "firstName", "lastName"],
        },
      ],
    },
  ],
});
if (!projectWithIssues?.issues?.length) {
  return res.status(404).json({ message: "No issues found for analysis" });
}
// get analysis options from settings
const analysisOptions = {
  provider: project.getSetting("ai.provider"),
  features: project.getSetting("ai.features"),
};
logger.debug("Running analysis with options:", analysisOptions);
const analysis = await AIService.analyzeProject(
  projectWithIssues.issues,
  analysisOptions
);
if (!analysis?.issueAnalysis) {
  return res
    .status(500)
    .json({ message: "Analysis failed to return results" });
}
// transform response to match frontend expectations
res.json({
  issueAnalysis: analysis.issueAnalysis.map((issue) => ({
    id: issue.id,
    original: {
      title: issue.original.title,
      description: issue.original.description,
```

```
          type: issue.original.type,
          priority: issue.original.priority,
          severity: issue.original.severity,
          status: issue.original.status,
        },
        suggested: {
          title: issue.suggested.title,
          description: issue.suggested.description,
          type: issue.suggested.type,
          priority: issue.suggested.priority,
          severity: issue.suggested.severity,
          status: issue.suggested.status,
        },
        improvements: {
          title: issue.improvements.title,
          description: issue.improvements.description,
          severity: {
            original: issue.improvements.severity.original,
            suggested: issue.improvements.severity.suggested,
            reason: issue.improvements.severity.reason,
          },
        },
        analysis: {
          severity: issue.analysis.severity,
          type: issue.analysis.type,
          priority: issue.analysis.priority,
        },
      })),
      summary: {
        averageComplexity: analysis.summary.averageComplexity,
        consistencyMetrics: {
          highConsistency: analysis.summary.consistencyMetrics.highConsistency,
          totalIssues: analysis.summary.consistencyMetrics.totalIssues,
        },
      },
      recommendations: analysis.recommendations,
    });
  } catch (error) {
    console.error("Error in analyzeProject:", error);
    next(error);
  }
};

module.exports = {
  updateProjectAiSettings,
  getProjectAiSettings,
  applyProjectAiSettings,
  analyzeProject,
```

```
};



const db = require("@models");
const { Op } = require("sequelize");
const { logger } = require("@utils/logger");

class AutoAssignmentService {
  constructor() {
    this.User = db.User;
    this.Project = db.Project;
    this.Issue = db.Issue;
    this.ProjectMember = db.ProjectMember;
  }

  async findBestAssignee(issue) {
    try {
      // get project settings
      const project = await this.Project.findByPk(issue.projectId);
      if (!project) {
        logger.warn("Project not found for auto-assignment");
        return null;
      }

      // get weight factors from project settings or use defaults
      const weightFactors = project.getSetting(
        "autoAssignment.weightFactors"
      ) || {
        expertise: 0.4,
        workload: 0.3,
        performance: 0.3,
      };

      // validate that weights sum to 1
      const totalWeight = Object.values(weightFactors).reduce(
        (sum, weight) => sum + weight,
        0
      );
      if (Math.abs(totalWeight - 1) > 0.01) {
        logger.warn("Weight factors don't sum to 1, using default weights");
        weightFactors.expertise = 0.4;
        weightFactors.workload = 0.3;
        weightFactors.performance = 0.3;
      }

      // get all active project members
      const projectMembers = await this.ProjectMember.findAll({
```

```
  where: {
    projectId: issue.projectId,
    status: "ACTIVE",
    availability: { [Op.gt]: 0 },
  },
  include: [
    {
      model: this.User,
      attributes: ["id", "firstName", "lastName"],
      include: [
        {
          model: this.Issue,
          as: "assignedIssues",
          where: {
            status: { [Op.in]: ["TODO", "IN_PROGRESS", "IN_REVIEW"] },
          },
          required: false,
        },
      ],
    },
  ],
});
if (!projectMembers?.length) {
  logger.warn("No available project members found");
  return null;
}
// calculate scores for each member using project-specific weights
const memberScores = await Promise.all(
  projectMembers.map(async (member) => {
    if (!member.User) return null;
    const expertiseScore = this.calculateExpertiseScore(member, issue);
    const workloadScore = this.calculateWorkloadScore(
      member.User.assignedIssues?.length || 0,
      member.availability
    );
    const performanceScore = await this.calculatePerformanceScore(
      member,
      issue
    );
    const totalScore =
      expertiseScore * weightFactors.expertise +
      workloadScore * weightFactors.workload +
      performanceScore * weightFactors.performance;

    logger.debug("Member score calculation:", {
      memberId: member.userId,
      expertiseScore,
      workloadScore,
```

```
          performanceScore,
          weights: weightFactors,
          totalScore,
        });

        return {
          memberId: member.userId,
          totalScore,
          currentWorkload: member.User.assignedIssues?.length || 0,
        };
      })
    );
    // get max issues per user from project settings
    const maxIssuesPerUser =
      project.getSetting("autoAssignment.maxIssuesPerUser") || 5;
    // filter valid scores and find best match
    const validScores = memberScores.filter(
      (score) => score !== null && score.currentWorkload < maxIssuesPerUser
    );
    if (!validScores.length) {
      logger.warn("No eligible members found for assignment");
      return null;
    }
    const bestMatch = validScores.reduce((best, current) =>
      current.totalScore > best.totalScore ? current : best
    );
    logger.info("Best match found:", {
      memberId: bestMatch.memberId,
      score: bestMatch.totalScore,
      weights: weightFactors,
    });
    return bestMatch.memberId;
  } catch (error) {
    logger.error("Error in findBestAssignee:", error);
    return null;
  }
}
calculateExpertiseScore(member, issue) {
  const expertise = member.expertise || {};
  return (expertise[issue.type] || 1) * 20; // scale 1-5 expertise to 20-100
}
calculateWorkloadScore(currentIssues, availability) {
  const maxIssues = 5;
  const workloadPercentage = (currentIssues / maxIssues) * 100;
  return Math.max(0, Math.min(100, availability - workloadPercentage));
}
async calculatePerformanceScore(member, issue) {
  try {
```

```
      const thirtyDaysAgo = new Date();
      thirtyDaysAgo.setDate(thirtyDaysAgo.getDate() - 30);
      const completedIssues = await this.Issue.findAll({
        where: {
          assigneeId: member.userId,
          type: issue.type,
          status: "DONE",
          completedAt: {
            [Op.gte]: thirtyDaysAgo,
          },
        },
      });
      if (!completedIssues.length) return 50; // default score for no history
      // calculate average completion time
      const avgCompletionTime =
        completedIssues.reduce((acc, issue) => {
          return (
            acc + (new Date(issue.completedAt) - new Date(issue.createdAt))
          );
        }, 0) / completedIssues.length;
      // convert to base 100 score
      const hoursToComplete = avgCompletionTime / (1000 * 60 * 60);
      const timeScore = Math.min(100, (24 / hoursToComplete) * 100);
      return timeScore;
    } catch (error) {
      logger.error("Error calculating performance score:", error);
      return 50; // default score on error
    }
  }
  async handleIssueAssignment(issue) {
    try {
      const assigneeId = await this.findBestAssignee(issue);
      if (assigneeId) {
        logger.info("Issue auto-assigned successfully:", {
          issueId: issue.id,
          assigneeId,
        });
      }
      return assigneeId;
    } catch (error) {
      logger.error("Error in handleIssueAssignment:", error);
      return null;
    }
  }
}
module.exports = new AutoAssignmentService();
```

ДОДАТОК В
ВИХІДНИЙ КОД СЕРВІСУ
ШТУЧНОГО ІНТЕЛЕКТУ

```
import pandas as pd
import os
import json
import gzip
import logging
import re

# basic logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger()
# constants for severity calculation
CRITICAL_THRESHOLD = 604800  # 7 days in seconds
MAJOR_THRESHOLD = 2592000   # 30 days in seconds
# tag remapping
PRIORITY_MAP = {
    'Blocker': 'URGENT',
    'Critical': 'HIGH',
    'Major': 'MEDIUM',
    'Minor': 'LOW',
    'Trivial': 'LOW',
    'Optional': 'LOW'
}
TYPE_MAP = {
    'Bug': 'BUG',
    'New Feature': 'FEATURE',
    'Feature Request': 'FEATURE',
    'Improvement': 'FEATURE',
    'Enhancement': 'FEATURE',
    'Task': 'TASK',
    'Sub-task': 'TASK',
    'Technical task': 'TASK',
    'Other': 'TASK'
}
# clean dataset columns
FINAL_COLUMNS = ['title', 'priority', 'type', 'severity', 'description', 'created', 'resolved',
'resolution_time', 'text']

def preprocess_text(text):
    """Preprocess text by normalizing whitespace and removing special characters."""
    if not isinstance(text, str):
        return 'No text provided'
```

```
    # remove special characters and normalize whitespace
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text.lower()

def calculate_severity(row):
    """Calculate severity based on priority and resolution time."""
    resolution_time = row['resolution_time']
    priority = row['priority']
    # calculate severitry based on resolution time and priority
    if priority in ['URGENT', 'HIGH']:
        return 'CRITICAL'
    if priority == 'MEDIUM' and resolution_time > CRITICAL_THRESHOLD:
        return 'MAJOR'
    if resolution_time > MAJOR_THRESHOLD:
        return 'MAJOR'
    return 'MINOR'

def map_and_clean_data(df):
    """Clean, preprocess, and calculate severity for the dataset."""
    # combine and preprocess text fields
    df['description'] = df['description'].fillna('No description provided')
    df['title'] = df['title'].fillna('No title provided')
    df['text'] = df['title'] + ' ' + df['description']
    df['text'] = df['text'].apply(preprocess_text)
    # map priority and type
    df['priority'] = df['priority'].map(PRIORITY_MAP).fillna('MEDIUM')
    df['type'] = df['type'].map(TYPE_MAP).fillna('TASK')

    # handle dates and calculate resolution time
    df['created'] = pd.to_datetime(df['created'], errors='coerce')
    df['resolved'] = pd.to_datetime(df['resolved'], errors='coerce')
    df['resolved'] = df['resolved'].fillna(pd.Timestamp.now())
    df['resolution_time'] = (df['resolved'] - df['created']).dt.total_seconds()
    df = df.dropna(subset=['created', 'resolved', 'resolution_time'])
    df = df[df['resolution_time'] >= 0]
    # calculate severity
    df['severity'] = df.apply(calculate_severity, axis=1)
    # only save columns that are required for ml training
    df = df[FINAL_COLUMNS]

    logger.info(f"Rows remaining after cleaning: {len(df)}")
    logger.info(f"Columns in cleaned dataset: {df.columns.tolist()}")
    return df

def save_cleaned_data(df, output_path):
    """Save cleaned data to a compressed JSON file."""
    df['created'] = df['created'].dt.strftime('%Y-%m-%dT%H:%M:%S')
```

```
    df['resolved'] = df['resolved'].dt.strftime('%Y-%m-%dT%H:%M:%S')
    # compressing into .gz to save space
    with gzip.open(output_path + '.gz', 'wt', encoding='utf-8') as f:
        json.dump(df.to_dict(orient='records'), f, indent=2, ensure_ascii=False)
    logger.info(f"Data successfully saved to {output_path}.gz")

def main(input_path, output_path='processed_issues', sample_size=None):
    logger.info("Loading dataset...")
    files = os.listdir(input_path)
    csv_files = [f for f in files if f.endswith('.csv')]
    if not csv_files:
        raise FileNotFoundError("No CSV files found in the specified directory.")
    file_path = os.path.join(input_path, csv_files[0])
    df = pd.read_csv(file_path)
    logger.info(f"Dataset loaded with {len(df)} rows.")

    if sample_size:
        logger.info(f"Sampling {sample_size} rows...")
        df = df.sample(n=min(sample_size, len(df)), random_state=42)

    logger.info("Cleaning and preprocessing data...")
    df = map_and_clean_data(df)

    logger.info("Saving processed data...")
    save_cleaned_data(df, output_path)

if __name__ == "__main__":
    # dataset path
    input_dir = "C:\\Users\\cwc\\.cache\\kagglehub\\datasets\\antonyjr\\jira-issue-reports-v1\\versions\\1"
    output_file = "processed_issues"
    # only 500k samples out of 700k for now
    main(input_path=input_dir, output_path=output_file, sample_size=500000)
```

```
from sklearn.utils.class_weight import compute_class_weight
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import LabelEncoder
import nltk
from sklearn.model_selection import train_test_split
from nltk.corpus import stopwords
import re
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import classification_report, f1_score, ConfusionMatrixDisplay
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE
import logging
import matplotlib.pyplot as plt
import gzip
import pickle

nltk.download('stopwords')
STOPWORDS = set(stopwords.words('english'))

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger()
NUMBER_OF_JOBS = 6

# text preprocessing
class TextCleaner(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.pattern = re.compile(r'[^\w\s]')

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return X.apply(self._clean_text)

    def _clean_text(self, text):
        text = text.lower()
        text = re.sub(self.pattern, '', text)
        text = ' '.join(word for word in text.split() if word not in STOPWORDS)
        return text

# calculate class weights
def calculate_class_weights(y):
    """Calculate class weights to handle imbalance."""
    classes = np.unique(y)
    weights = compute_class_weight('balanced', classes=classes, y=y)
    return dict(zip(classes, weights))

# pipeline automation
def build_pipeline(model_name, X_train_text, X_test_text, y_train, y_test, label_encoder):
    """Build and execute a pipeline with preprocessing, augmentation, and modeling."""
    logger.info(f"Building pipeline for {model_name}...")
    try:
        # encode labels
        y_train_encoded = label_encoder.fit_transform(y_train)
        y_test_encoded = label_encoder.transform(y_test)
```

```
        # get class weights
        class_weights = calculate_class_weights(y_train_encoded)

        # configure SMOTE with class-specific strategy
        smote_strategy = {
            label: int(max(1.5 * y_train_encoded.tolist().count(label), 5000))
            for label in np.unique(y_train_encoded)
        }

        # pipeline with preprocessing, augmentation, and classification
        pipeline = imbalanced_make_pipeline(
            TextCleaner(),
            TfidfVectorizer(max_features=30000, ngram_range=(1, 3), stop_words='english'),
            SMOTE(sampling_strategy=smote_strategy, random_state=42),
            RandomForestClassifier(
                random_state=42,
                class_weight=class_weights,
                n_estimators=200,
                max_depth=30,
                n_jobs=NUMBER_OF_JOBS
            )
        )

        # fit pipeline
        logger.info(f"Training {model_name} pipeline...")
        pipeline.fit(X_train_text, y_train_encoded)

        # predict and evaluate
        y_pred = pipeline.predict(X_test_text)
        report = classification_report(y_test_encoded, y_pred, target_names=label_encoder.classes_,
zero_division=0)
        macro_f1 = f1_score(y_test_encoded, y_pred, average='macro')
        logger.info(f"{model_name.upper()} Classification Report:\n{report}")
        logger.info(f"{model_name.upper()} Macro F1-Score: {macro_f1:.4f}")

        # confusion matrix
        ConfusionMatrixDisplay.from_predictions(y_test_encoded, y_pred,
display_labels=label_encoder.classes_)
        plt.title(f"{model_name.upper()} Confusion Matrix")
        plt.show()

        # save pipeline
        pipeline_filename = f'{model_name}_pipeline.pkl'
        with open(pipeline_filename, 'wb') as f:
            pickle.dump(pipeline, f)
        logger.info(f"Pipeline saved to {pipeline_filename}")

    except Exception as e:
```

```
        logger.error(f"Error in {model_name} pipeline: {e}")

# main training logic
def train_models(data_path='processed_issues.gz'):
    """Train models with enhanced preprocessing, augmentation, and pipeline automation."""
    try:
        logger.info("Loading dataset...")
        with gzip.open(data_path, 'rt', encoding='utf-8') as f:
            df = pd.read_json(f)
        logger.info(f"Dataset loaded successfully. Shape: {df.shape}")
    except Exception as e:
        logger.error(f"Failed to load data: {e}")
        return

    # get and validate only required columns
    required_columns = ['text', 'priority', 'type', 'severity']
    missing_columns = [col for col in required_columns if col not in df.columns]
    if missing_columns:
        logger.error(f"Missing required columns in data: {missing_columns}")
        return

    # convert target labels to strings for consistency
    for col in ['priority', 'type', 'severity']:
        df[col] = df[col].astype(str)

    # feature preparation (use only text)
    X_text = df['text']
    y_labels = {
        'priority': df['priority'],
        'type': df['type'],
        'severity': df['severity']
    }

    # train-test splits
    splits = {
        label_name: train_test_split(
            X_text, y, test_size=0.2, stratify=y, random_state=42
        )
        for label_name, y in y_labels.items()
    }

    # train pipelines for each label
    for model_name, (X_train_text, X_test_text, y_train, y_test) in splits.items():
        label_encoder = LabelEncoder()
        build_pipeline(model_name, X_train_text, X_test_text, y_train, y_test, label_encoder)

if __name__ == "__main__":
    train_models()
```

```
import os
import boto3
import stat
from pathlib import Path
from botocore.exceptions import ClientError, NoCredentialsError
from logging import getLogger
from dotenv import load_dotenv
import time

# configure logging
logger = getLogger(__name__)

current_dir = Path(__file__).parent
load_dotenv()

MODEL_CACHE_DIR = os.getenv('MODEL_CACHE_DIR', './models')

class ProgressTracker:
    def __init__(self, total_size, interval=30):
        self.total_size = total_size
        self.current_size = 0
        self.last_log_time = time.time()
        self.interval = interval  # log interval in seconds
        self.start_time = time.time()

    def __call__(self, bytes_transferred):
        self.current_size += bytes_transferred
        current_time = time.time()

        # only log if interval has passed
        if current_time - self.last_log_time >= self.interval:
            elapsed_time = current_time - self.start_time
            progress = (self.current_size / self.total_size) * 100
            speed = self.current_size / (elapsed_time + 0.01) / (1024 * 1024)  # MB/s

            logger.info(
                f"Progress: {progress:.1f}% ({self.current_size:,}/{self.total_size:,} bytes) "
                f"Speed: {speed:.2f} MB/s"
            )
            self.last_log_time = current_time

class ModelManager:
    """Simple manager for handling ML model files in S3."""
```

```
def __init__(self):
    # initialize S3 client
    self.s3 = boto3.client('s3',
        aws_access_key_id=os.getenv('AWS_ACCESS_KEY_ID'),
        aws_secret_access_key=os.getenv('AWS_SECRET_ACCESS_KEY'),
        region_name=os.getenv('AWS_REGION')
    )
    self.bucket = os.getenv('AWS_BUCKET_NAME')

    # create local models directory with proper permissions
    self.model_dir = Path(current_dir) / 'models'  # using Path objects
    self._ensure_directory_exists()

def _ensure_directory_exists(self):
    """Create models directory with proper permissions if it doesn't exist."""
    try:
        if not isinstance(self.model_dir, Path):
            self.model_dir = Path(self.model_dir)

        if not self.model_dir.exists():
            logger.info(f"Creating models directory at {self.model_dir}")
            self.model_dir.mkdir(parents=True, exist_ok=True)

            # set directory permissions (read/write for current user)
            os.chmod(self.model_dir,
                    stat.S_IRUSR | stat.S_IWUSR | stat.S_IXUSR)  # User RWX

        # test write permissions by creating a test file
        test_file = self.model_dir / '.test'
        try:
            test_file.touch()
            test_file.unlink()  # remove test file if successful
            logger.info("Directory permissions verified")
        except Exception as e:
            logger.error(f"Directory permission test failed: {str(e)}")
            raise

    except Exception as e:
        logger.error(f"Failed to create or verify models directory: {str(e)}")
        raise

def upload_model(self, model_name: str, local_path: Path) -> bool:
    """Upload a model from local storage to S3."""
    if not self.bucket:
        logger.warning("S3 bucket not configured")
        return False

    # ensure we're working with Path objects
```

```
    local_path = Path(local_path) if isinstance(local_path, str) else local_path

    try:
        # validate file
        if not local_path.exists():
            logger.error(f"File not found: {local_path}")
            return False

        if not local_path.is_file():
            logger.error(f"Not a file: {local_path}")
            return False

        # get file size
        file_size = local_path.stat().st_size
        logger.info(f"Starting upload of {model_name} (size: {file_size:,} bytes)")

        # create progress tracker
        progress = ProgressTracker(file_size)

        # upload directly to S3
        s3_key = f"models/{model_name}"
        with open(local_path, 'rb') as file_obj:
            self.s3.upload_fileobj(
                file_obj,
                self.bucket,
                s3_key,
                Callback=progress
            )

        logger.info(f"Successfully uploaded model {model_name}")
        return True

    except Exception as e:
        logger.error(f"Failed to upload model {model_name}: {str(e)}")
        return False

def download_model(self, model_name: str) -> Path:
    """Download a model from S3 to local storage."""
    if not self.bucket:
        logger.warning("S3 bucket not configured")
        return None

    local_path = self.model_dir / model_name
    s3_key = f"models/{model_name}"

    try:
        # get file size from S3
        response = self.s3.head_object(Bucket=self.bucket, Key=s3_key)
```

```
            file_size = response['ContentLength']

            logger.info(f"Starting download of {model_name} (size: {file_size:,} bytes)")

            # create progress tracker
            progress = ProgressTracker(file_size)

            # download the file
            with open(local_path, 'wb') as file_obj:
                self.s3.download_fileobj(
                    self.bucket,
                    s3_key,
                    file_obj,
                    Callback=progress
                )

            logger.info(f"Successfully downloaded model to {local_path}")
            return local_path

        except ClientError as e:
            logger.error(f"Failed to download model {model_name}: {str(e)}")
            if local_path.exists():
                logger.info(f"Using existing local model: {model_name}")
                return local_path
            raise
        except Exception as e:
            logger.error(f"Unexpected error downloading model: {str(e)}")
            raise

    def get_model_path(self, model_name: str) -> Path:
        """Get the path to a model, downloading it if necessary."""
        local_path = self.model_dir / model_name

        if not local_path.exists():
            return self.download_model(model_name)

        return local_path

    def _ensure_directory_exists(self):
        """Create models directory with proper permissions if it doesn't exist."""
        try:
            if not self.model_dir.exists():
                logger.info(f"Creating models directory at {self.model_dir}")
                self.model_dir.mkdir(parents=True, exist_ok=True)

                # set directory permissions (read/write for current user)
                os.chmod(self.model_dir,
                         stat.S_IRUSR | stat.S_IWUSR | stat.S_IXUSR)  # user RWX
```

```
            # test write permissions by creating a test file
            test_file = self.model_dir / '.test'
            try:
                test_file.touch()
                test_file.unlink()  # remove test file if successful
                logger.info("Directory permissions verified")
            except Exception as e:
                logger.error(f"Directory permission test failed: {str(e)}")
                raise

        except Exception as e:
            logger.error(f"Failed to create or verify models directory: {str(e)}")
            raise


    def test_connection(self) -> dict:
        """
        Test S3 connection and bucket accessibility.

        Returns:
            dict: Connection test results including status and details
        """
        results = {
            'connection': False,
            'bucket_exists': False,
            'bucket_accessible': False,
            'message': '',
            'error': None
        }

        try:
            # test 1: Check AWS credentials and connection
            try:
                self.s3.list_buckets()
                results['connection'] = True
                logger.info("Successfully connected to AWS S3")
            except NoCredentialsError:
                results['message'] = "AWS credentials not found or invalid"
                logger.error(results['message'])
                return results
            except Exception as e:
                results['message'] = "Failed to connect to AWS S3"
                results['error'] = str(e)
                logger.error(f"{results['message']}: {e}")
                return results

            # test 2: Check if bucket exists
```

```
            if not self.bucket:
                results['message'] = "No bucket name configured"
                logger.warning(results['message'])
                return results

            try:
                self.s3.head_bucket(Bucket=self.bucket)
                results['bucket_exists'] = True
                logger.info(f"Bucket '{self.bucket}' exists")
            except ClientError as e:
                error_code = e.response.get('Error', {}).get('Code')
                if error_code == '404':
                    results['message'] = f"Bucket '{self.bucket}' does not exist"
                elif error_code == '403':
                    results['message'] = f"Bucket '{self.bucket}' exists but is not accessible"
                else:
                    results['message'] = f"Error checking bucket: {str(e)}"
                logger.error(results['message'])
                return results

            # test 3: Check permissions by attempting to list objects
            try:
                self.s3.list_objects_v2(Bucket=self.bucket, MaxKeys=1)
                results['bucket_accessible'] = True
                results['message'] = "Successfully connected to S3 and accessed bucket"
                logger.info(results['message'])
            except ClientError as e:
                results['message'] = f"Cannot list objects in bucket: {str(e)}"
                logger.error(results['message'])
                return results

        except Exception as e:
            results['message'] = f"Unexpected error during connection test: {str(e)}"
            results['error'] = str(e)
            logger.error(results['message'])

        return results
```

```
from fastapi import FastAPI, HTTPException, Request
from pydantic import BaseModel
from typing import List, Optional
from enum import Enum
import numpy as np
```

```python
import logging
import sys
import pickle
from model_manager import ModelManager
import time
from dotenv import load_dotenv
import os
from contextlib import asynccontextmanager

load_dotenv()

# configure logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('python_service.log'),
        logging.StreamHandler(sys.stdout)
    ]
)

logger = logging.getLogger(__name__)

HOST = os.getenv('HOST', '0.0.0.0')
PORT = int(os.getenv('PORT', 8000))
MODEL_CACHE_DIR = os.getenv('MODEL_CACHE_DIR', './models')
ENVIRONMENT = os.getenv('ENVIRONMENT', 'development')

# define enums to match the Sequelize model
class IssueType(str, Enum):
    BUG = "BUG"
    FEATURE = "FEATURE"
    TASK = "TASK"

class IssueStatus(str, Enum):
    TODO = "TODO"
    IN_PROGRESS = "IN_PROGRESS"
    IN_REVIEW = "IN_REVIEW"
    DONE = "DONE"

class IssuePriority(str, Enum):
    LOW = "LOW"
    MEDIUM = "MEDIUM"
    HIGH = "HIGH"
    URGENT = "URGENT"

class IssueSeverity(str, Enum):
    MINOR = "MINOR"
```

```
    MAJOR = "MAJOR"
    CRITICAL = "CRITICAL"
    BLOCKER = "BLOCKER"

class Issue(BaseModel):
    id: str
    title: str
    description: Optional[str]
    type: IssueType
    status: IssueStatus
    priority: IssuePriority
    severity: IssueSeverity
    projectId: str
    reporterId: Optional[str]
    assigneeId: Optional[str]
    completedAt: Optional[str]

class ProjectData(BaseModel):
    issues: List[Issue]

class MLPredictor:
    MODEL_TYPES = ['type', 'severity', 'priority']

    def __init__(self):
        self.models = {}
        self.model_manager = ModelManager()
        self.initialized = False

    def is_initialized(self) -> bool:
        return self.initialized

    def initialize(self, retries=3):
        """Initialize models with retry logic"""
        for attempt in range(retries):
            try:
                self.load_models()
                self.initialized = True
                return True
            except Exception as e:
                if attempt == retries - 1:
                    logger.error(f"Failed to load models after {retries} attempts: {e}")
                    return False
                logger.warning(f"Attempt {attempt + 1} failed, retrying in 5 seconds...")
                time.sleep(5)

    def load_models(self):
        """Load ML models using the model manager."""
        try:
```

```
            logger.info("Loading ML models...")

            for model_type in self.MODEL_TYPES:
                model_name = f'{model_type}_model.pkl'

                try:
                    # get model path from manager (downloads if needed)
                    model_path = self.model_manager.get_model_path(model_name)

                    if not model_path or not model_path.exists():
                        raise FileNotFoundError(f"Model {model_type} not found")

                    # load the model
                    with open(model_path, 'rb') as f:
                        self.models[model_type] = pickle.load(f)

                    logger.info(f"Successfully loaded {model_type} model")

                except Exception as e:
                    error_msg = f"Error loading {model_type} model: {str(e)}"
                    logger.error(error_msg)
                    raise RuntimeError(error_msg)

            # validate models
            for model_type, model in self.models.items():
                if not hasattr(model, 'predict'):
                    raise AttributeError(f"Model {model_type} doesn't have predict method")
                logger.info(f"Validated {model_type} model")

        except Exception as e:
            error_msg = f"Error loading ML models: {str(e)}"
            logger.error(error_msg)
            raise RuntimeError(error_msg)

    def predict(self, text: str) -> dict:
        """Make predictions using loaded models."""
        if not self.initialized:
            logger.error("Attempting to predict with uninitialized models")
            return None

        predictions = {}
        try:
            for model_type, model in self.models.items():
                prediction = model.predict([text])[0]
                confidence = float(np.max(model.predict_proba([text])))
                predictions[model_type] = {
                    "prediction": prediction,
                    "confidence": confidence
```

```
                }
        except Exception as e:
            logger.error(f"Prediction error: {str(e)}")
            return None
        return predictions

ml_predictor = MLPredictor()

@asynccontextmanager
async def lifespan(app: FastAPI):
    # startup
    if not ml_predictor.initialize():
        logger.warning("Starting application with uninitialized ML models")
    yield

app = FastAPI(lifespan=lifespan)

@app.middleware("http")
async def log_requests(request: Request, call_next):
    logger.info(f"Request: {request.method} {request.url}")
    body = await request.body()
    logger.debug(f"Body: {body.decode()}")
    response = await call_next(request)
    logger.info(f"Response status: {response.status_code}")
    return response

# health check for docker build
@app.get("/health")
async def health_check():
    return {
        "status": "up",
        "environment": ENVIRONMENT,
        "ml_models_loaded": ml_predictor.is_initialized(),
        "host": HOST,
        "port": PORT
    }

def analyze_issue_ml(issue: Issue) -> dict:
    if not ml_predictor.is_initialized():
        raise HTTPException(
            status_code=503,
            detail="ML service not fully initialized"
        )

    text = f"{issue.title} {issue.description if issue.description else ''}"

    # get ML predictions
    ml_predictions = ml_predictor.predict(text)
```

```
    if not ml_predictions:
        raise HTTPException(status_code=500, detail="ML prediction failed")

    # calculate complexity score based on text length and prediction confidences
    text_complexity = len(text.split()) / 100  # normalize by 100 words
    confidence_avg = sum(pred['confidence'] for pred in ml_predictions.values()) / 3
    complexity_score = min(5, max(1, round((text_complexity + (1 - confidence_avg)) * 2.5)))
    return {
        "id": issue.id,
        "original": issue.dict(),
        "suggested": {
            "title": issue.title,
            "description": issue.description,
            "type": ml_predictions['type']['prediction'],
            "status": issue.status,
            "severity": ml_predictions['severity']['prediction'],
            "priority": ml_predictions['priority']['prediction']
        },
        "improvements": {
            "severity": {
                "original": issue.severity,
                "suggested": ml_predictions['severity']['prediction'],
                "confidence": ml_predictions['severity']['confidence'],
                "reason": f"ML model prediction with {ml_predictions['severity']['confidence']:.2%}
confidence"
            },
            "priority": {
                "original": issue.priority,
                "suggested": ml_predictions['priority']['prediction'],
                "confidence": ml_predictions['priority']['confidence'],
                "reason": f"ML model prediction with {ml_predictions['priority']['confidence']:.2%}
confidence"
            },
            "type": {
                "original": issue.type,
                "suggested": ml_predictions['type']['prediction'],
                "confidence": ml_predictions['type']['confidence'],
                "reason": f"ML model prediction with {ml_predictions['type']['confidence']:.2%} confidence"
            }
        },
        "analysis": {
            "severity": {
                "level": ml_predictions['severity']['prediction'],
                "confidence": ml_predictions['severity']['confidence']
            },
            "complexity": {
                "score": complexity_score,
                "factors": {
```

```
                    "text_length": text_complexity,
                    "prediction_confidence": confidence_avg
                }
            }
        }
    }
@app.post("/analyze/project")
async def analyze_project(project: ProjectData):
    if not project.issues:
        raise HTTPException(status_code=400, detail="No issues provided")

    try:
        logger.debug("Analyzing issues...")

        # analyze each issue using ML models
        detailed_analyses = [analyze_issue_ml(issue) for issue in project.issues]

        # calculate project-wide metrics
        total_issues = len(detailed_analyses)
        avg_complexity = np.mean([
            analysis["analysis"]["complexity"]["score"]
            for analysis in detailed_analyses
        ])

        # calculate confidence-weighted severity distribution
        severity_distribution = {
            severity: sum(1 for analysis in detailed_analyses
                          if analysis["analysis"]["severity"]["level"] == severity)
            for severity in IssueSeverity.__members__
        }

        high_severity_count = sum(
            1 for analysis in detailed_analyses
            if analysis["analysis"]["severity"]["level"] in ["CRITICAL", "BLOCKER"]
            and analysis["analysis"]["severity"]["confidence"] > 0.7
        )
        # generate recommendations based on ML predictions
        recommendations = []

        # check for high-confidence severity mismatches
        severity_mismatches = sum(
            1 for analysis in detailed_analyses
            if (analysis["original"]["severity"] != analysis["suggested"]["severity"]
                and analysis["improvements"]["severity"]["confidence"] > 0.8)
        )
        if severity_mismatches / total_issues > 0.3:
            recommendations.append({
                "type": "severity_review",
```

```
                "title": "High number of potential severity misclassifications",
                "description": "Consider reviewing issue severity classifications based on ML model
suggestions"
            })
        # check for high complexity issues
        high_complexity_count = sum(
            1 for analysis in detailed_analyses
            if analysis["analysis"]["complexity"]["score"] >= 4
        )

        if high_complexity_count / total_issues > 0.4:
            recommendations.append({
                "type": "complexity_management",
                "title": "High proportion of complex issues",
                "description": "Consider breaking down complex issues into smaller tasks"
            })
        response_data = {
            "issueAnalysis": detailed_analyses,
            "summary": {
                "averageComplexity": float(avg_complexity),
                "severityDistribution": {
                    "distribution": severity_distribution,
                    "highSeverity": high_severity_count,
                    "totalIssues": total_issues
                }
            },
            "recommendations": recommendations
        }

        logger.debug(f"Analysis complete. Generated {len(detailed_analyses)} analyses")
        return response_data

    except Exception as e:
        logger.error(f"Analysis error: {str(e)}", exc_info=True)
        raise HTTPException(status_code=500, detail=str(e))
if __name__ == "__main__":
    import uvicorn
    logger.info(f"Starting server on {HOST}:{PORT}")
    uvicorn.run(
        app,
        host=HOST,
        port=PORT,
        log_level=os.getenv('LOG_LEVEL', 'info').lower()
    )
```

ДОДАТОК Д

ВИХІДНИЙ КОД ТЕСТ-КЕЙСІВ ВЕБЗАСТОСУНКУ

ест-кейси Jest

```
import { render, screen, fireEvent, waitFor } from "@testing-library/react";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import { BrowserRouter } from "react-router-dom";
import userEvent from "@testing-library/user-event";
import axios from "axios";
import Login from "../pages/Login/Login";
import Projects from "../pages/Projects/Projects";
import ProjectDetails from "../pages/ProjectDetails/ProjectDetails";
import Issues from "../pages/Issues/Issues";

// mock axios
jest.mock("axios");
// mock react-router-dom hooks
const mockNavigate = jest.fn();
jest.mock("react-router-dom", () => ({
  ...jest.requireActual("react-router-dom"),
  useNavigate: () => mockNavigate,
  useParams: () => ({ id: "1" }),
}));
// test setup helper
const setupTest = (Component) => {
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        retry: false,
      },
    },
  });
  return render(
    <QueryClientProvider client={queryClient}>
      <BrowserRouter>
        <Component />
      </BrowserRouter>
    </QueryClientProvider>
  );
};
// case 1: login functionality
describe("Login Page", () => {
  beforeEach(() => {
    localStorage.clear();
    jest.clearAllMocks();
```

```
  });
  test("should successfully login and redirect to dashboard", async () => {
    const mockUser = {
      firstName: "John",
      lastName: "Doe",
      email: "john@example.com",
    };
    axios.post.mockResolvedValueOnce({
      data: { user: mockUser },
    });
    setupTest(Login);
    // fill in the form
    await userEvent.type(screen.getByLabelText(/email/i), "john@example.com");
    await userEvent.type(screen.getByLabelText(/password/i), "password123");
    // submit form
    fireEvent.click(screen.getByRole("button", { name: /login/i }));
    await waitFor(() => {
      // check if navigation to dashboard occurred
      expect(mockNavigate).toHaveBeenCalledWith("/dashboard");
    });
  });
});
// case 2: project creation
describe("Projects Page", () => {
  beforeEach(() => {
    localStorage.clear();
    jest.clearAllMocks();
  });
  test("should create a new project successfully", async () => {
    // mock initial projects query
    axios.get.mockResolvedValueOnce({ data: [] });
    // mock project creation
    const mockNewProject = {
      id: 1,
      name: "Test Project",
      key: "TEST",
      description: "Test Description",
    };
    axios.post.mockResolvedValueOnce({
      data: { project: mockNewProject },
    });
    setupTest(Projects);
    // open modal
    fireEvent.click(screen.getByText(/new project/i));
    // fill in project details
    await userEvent.type(
      screen.getByLabelText(/project name/i),
      "Test Project"
```

```
    );
    await userEvent.type(screen.getByLabelText(/project key/i), "TEST");
    await userEvent.type(
      screen.getByLabelText(/description/i),
      "Test Description"
    );
    // submit form
    fireEvent.click(screen.getByRole("button", { name: /create project/i }));
    await waitFor(() => {
      expect(mockNavigate).toHaveBeenCalledWith("/projects/1");
    });
  });
});
// case 3: issue creation
describe("Project Details Page", () => {
  beforeEach(() => {
    localStorage.clear();
    jest.clearAllMocks();
  });
  test("should create a new issue with attachments", async () => {
    // mock project data
    axios.get.mockResolvedValueOnce({
      data: {
        id: 1,
        name: "Test Project",
        issues: [],
      },
    });
    // mock issue creation
    const mockNewIssue = {
      id: 1,
      title: "Test Issue",
      description: "Test Description",
      type: "BUG",
      priority: "HIGH",
      severity: "MAJOR",
    };
    axios.post.mockResolvedValueOnce({
      data: { issue: mockNewIssue },
    });

    setupTest(ProjectDetails);

    // open new issue modal
    fireEvent.click(screen.getByText(/new issue/i));

    // fill in issue details
    await userEvent.type(screen.getByLabelText(/title/i), "Test Issue");
```

```
    await userEvent.type(
      screen.getByLabelText(/description/i),
      "Test Description"
    );
    // select type, priority, and severity
    await userEvent.selectOptions(screen.getByLabelText(/type/i), "BUG");
    await userEvent.selectOptions(screen.getByLabelText(/priority/i), "HIGH");
    await userEvent.selectOptions(screen.getByLabelText(/severity/i), "MAJOR");
    // mock file upload
    const file = new File(["test"], "test.png", { type: "image/png" });
    const fileInput = screen.getByLabelText(/attachments/i);
    await userEvent.upload(fileInput, file);
    // submit form
    fireEvent.click(screen.getByRole("button", { name: /create issue/i }));
    await waitFor(() => {
      expect(axios.post).toHaveBeenCalledWith(
        expect.any(String),
        expect.objectContaining({
          title: "Test Issue",
          type: "BUG",
          priority: "HIGH",
          severity: "MAJOR",
        })
      );
    });
  });
});
// case 4: issue filtering
describe("Issues Page", () => {
  beforeEach(() => {
    localStorage.clear();
    jest.clearAllMocks();
  });
  test("should filter issues correctly", async () => {
    // mock issues data
    const mockIssues = [
      { id: 1, title: "Login page issue", type: "BUG", status: "IN_PROGRESS" },
      { id: 2, title: "Feature request", type: "FEATURE", status: "TODO" },
    ];
    axios.get.mockResolvedValue({
      data: {
        issues: mockIssues,
        total: 2,
        currentPage: 1,
        totalPages: 1,
      },
    });
    setupTest(Issues);
```

```
    // test type filter
    await userEvent.selectOptions(
      screen.getByRole("combobox", { name: /type/i }),
      "BUG"
    );
    await waitFor(() => {
      expect(axios.get).toHaveBeenCalledWith(
        expect.any(String),
        expect.objectContaining({
          params: expect.objectContaining({ type: "BUG" }),
        })
      );
    });
    // test status filter
    await userEvent.selectOptions(
      screen.getByRole("combobox", { name: /status/i }),
      "IN_PROGRESS"
    );
    await waitFor(() => {
      expect(axios.get).toHaveBeenCalledWith(
        expect.any(String),
        expect.objectContaining({
          params: expect.objectContaining({ status: "IN_PROGRESS" }),
        })
      );
    });
    // test search
    const searchInput = screen.getByPlaceholderText(/search issues/i);
    await userEvent.type(searchInput, "Login page issue");
    await waitFor(() => {
      expect(axios.get).toHaveBeenCalledWith(
        expect.any(String),
        expect.objectContaining({
          params: expect.objectContaining({ search: "Login page issue" }),
        })
      );
    });
  });
});
```