МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Острозька академія»

Навчально-науковий інститут інформаційних технологій та бізнесу

Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Система створення персональних Learning Roadmaps для вивчення та вдосконалення технічних навичок»**

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Сокальський Олег Ігорович
Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Мацевич Денис Володимирович

Рецензент: *кандидат технічних наук, доцент,*
доцент кафедри прикладної математики
Донецького національного університету
імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «20» травня 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Система створення персональних Learning Roadmaps для вивчення та вдосконалення технічних навичок.

Автор: *Сокальський Олег Ігорович*

Науковий керівник: *викладач кафедри інформаційних технологій та аналітики даних Мацевич Денис Володимирович*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *86 с., 3 додатки, 0 табл., 7 рис., 19 джерел.*

Ключові слова: *МІКРОСЕРВІСНА АРХІТЕКТУРА, ГЕНЕРАТИВНИЙ ШТУЧНИЙ ІНТЕЛЕКТ, LLM, RAG, ІНТЕРАКТИВНЕ СЕРЕДОВИЩЕ РОЗРОБКИ (WEB IDE), ГЕЙМІФІКАЦІЯ, ВЕБДОСТУПНІСТЬ (A11Y).*

Короткий зміст праці: *Кваліфікаційну роботу присвячено проєктуванню та розробці комплексної мікросервісної освітньої платформи «Roadly», яка використовує передові технології штучного інтелекту для забезпечення персоналізованого процесу навчання програмуванню. Актуальність теми зумовлена неефективністю статичних навчальних курсів (one-size-fits-all) та необхідністю створення адаптивних систем, що здатні генерувати індивідуальні освітні траєкторії та надавати миттєву менторську підтримку.*

У роботі спроєктовано та реалізовано багаторівневу інфраструктуру, що складається з 11 контейнеризованих сервісів. Транзакційне доменне ядро платформи побудовано на базі .NET 9 з використанням принципів Clean Architecture, патерну CQRS (MediatR) та функціонального підходу до обробки помилок. Для

забезпечення інтелектуальних функцій розроблено окремий сервіс мовою Python (FastAPI, Celery, LangChain), який реалізує алгоритм гранулярної генерації навчальних матеріалів, забезпечує потокову передачу відповідей AI-ментора (через Server-Sent Events) та виконує семантичний пошук на базі технології RAG (із використанням векторної БД PostgreSQL pgvector).

Окрему увагу приділено розробці безпечного інтерактивного вебсередовища для виконання коду (Web IDE). Для цього створено високопродуктивний шлюз контейнеризації мовою Go, який через Docker SDK та протокол WebSocket забезпечує ізольоване виконання користувацького коду, доступ до термінала та віртуальної файлової системи в реальному часі з мінімальними затримками.

Клієнтський інтерфейс платформи реалізовано за допомогою React 19 і Tailwind CSS. У фронтенд-частині інтегровано модуль гейміфікації «Roadly+» (із системою досягнень та віртуальною економікою) для підвищення мотивації студентів. Критично важливим досягненням є реалізація динамічної панелі вебдоступності (A11y Panel), що забезпечує відповідність системи стандартам інклюзивності WCAG 2.1[16] (підтримка профілів для користувачів із дислексією, порушеннями зору та епілепсією). Для розширення екосистеми додатково розроблено модуль інтеграції з локальним середовищем розробки Visual Studio Code.

У результаті виконання роботи створено масштабований, безпечний і високотехнологічний Minimum Viable Product (MVP) освітньої платформи. Практичне значення отриманих результатів полягає у можливості впровадження розробленої системи в реальні освітні процеси вищих навчальних закладів або комерційних IT-шкіл.

(підпис автора)

ABSTRACT
of the Bachelor's thesis
for the award of a Bachelor's degree

Topic: *A System for Creating Personalized Learning Roadmaps for Acquiring and Improving Technical Skills.*

Author: *Oleh Ihorovych Sokalskyi*

Supervisor: *Lecturer of the Department of Information Technologies and Data Analytics Denys Volodymyrovych*

Defended on «.....».....20__

Explanatory note to the qualification work: *86 p., 3 appendices, 0 tab., 7 fig., 19 sources.*

Keywords: *MICROSERVICE ARCHITECTURE, GENERATIVE ARTIFICIAL INTELLIGENCE, LLM, RAG, INTERACTIVE DEVELOPMENT ENVIRONMENT (WEB IDE), GAMIFICATION, WEB ACCESSIBILITY (A11Y).*

Summary: *The qualification work is devoted to the design and development of a comprehensive microservice educational platform "Roadly", which utilizes advanced artificial intelligence technologies to provide a personalized programming learning process. The relevance of the topic stems from the inefficiency of static (one-size-fits-all) educational courses and the need to create adaptive systems that generate individual learning trajectories and provide instant mentoring support.*

A multi-tier infrastructure consisting of 11 containerized services was designed and implemented. The platform's transactional domain core is built on .NET 9 using Clean Architecture principles, the CQRS pattern (MediatR), and a functional approach to error handling. To provide intelligent features, a separate Python-based service (FastAPI, Celery, LangChain) was developed. This service implements a granular content

generation algorithm, provides streaming of AI mentor responses (via Server-Sent Events), and performs semantic search based on RAG technology (using PostgreSQL pgvector).

Special attention is given to developing a secure, interactive Web IDE, and for this purpose, created a high-performance containerization gateway was created in Go. Utilizing the Docker SDK and WebSocket protocol ensures the isolated execution of user code, terminal access, and interaction with a virtual file system with minimal latency.

The platform's client interface is implemented using React 19 and Tailwind CSS. The frontend integrates the "Roadly+" gamification module (featuring an achievement system and virtual economy) to increase student motivation. A critically important achievement is the implementation of a dynamic web accessibility panel (A11y Panel), ensuring the system complies with WCAG 2.1 accessibility standards (supporting profiles for users with dyslexia, visual impairments, and epilepsy). To expand the ecosystem, an integration module for the local Visual Studio Code development environment was additionally developed.

As a result of the work, a scalable, secure, and highly technological Minimum Viable Product (MVP) of the educational platform was created. The practical significance of the results lies in the possibility of implementing the developed system in real educational processes at higher education institutions or commercial IT schools.

(author's signature)

Зміст

Скорочення / термін	Розшифрування	Пояснення
A11y	Accessibility	вебдоступність; інклюзивність інтерфейсу для людей з обмеженими можливостями.
ACID	Atomicity, Consistency, Isolation, Durability	набір властивостей транзакцій бази даних, що гарантують їхню надійність.
API	Application Programming Interface	прикладний програмний інтерфейс.
CQRS	Command and Query Responsibility Segregation	шаблон проєктування, що розділяє моделі для читання та оновлення даних.
EdTech	Educational Technology	галузь освітніх технологій.
IDE	Integrated Development Environment	інтегроване середовище розробки.
JWT	JSON Web Token	відкритий стандарт для створення токенів доступу.
MVP	Minimum Viable Product	мінімально життєздатний продукт.
PTY	Pseudoterminal	псевдотермінал, програмна емуляція текстового термінала.
RAG	Retrieval-Augmented Generation	генерація тексту, доповнена пошуком у зовнішніх базах знань.
RPG	Role-Playing Game	рольова гра (у контексті

		візуалізації навчального процесу).
SSE	Server-Sent Events	технологія передачі потокових даних від сервера до браузера.
UI	User Interface	інтерфейс користувача.
UX	User Experience	користувацький досвід.
WCAG	Web Content Accessibility Guidelines	настанови щодо доступності вебконтенту.

ВСТУП

Актуальність теми. Сучасна галузь освітніх технологій (EdTech) переживає період фундаментальної трансформації, зумовленої стрімким розвитком генеративного штучного інтелекту. Попри велику кількість наявних ресурсів для вивчення програмування, більшість із них залишається статичною та використовує лінійний підхід до подання матеріалу. Головною проблемою таких платформ є неможливість адаптації до індивідуальних особливостей студента: його попереднього досвіду, наявних когнітивних прогалин і динаміки засвоєння знань. Це призводить до передчасної втрати мотивації та низької ефективності навчання.

Вирішенням цієї проблеми є перехід до концепції інтелектуальних адаптивних середовищ. Застосування великих мовних моделей (LLM) дозволяє автоматизувати створення персоналізованих навчальних маршрутів (Roadmaps), а використання технології RAG (Retrieval-Augmented Generation) забезпечує контекстне асистування на основі перевірених джерел знань. Водночас інтеграція Web IDE в ізольованих Docker-контейнерах і реалізація гейміфікованих механік дозволяють створити безпечний і залучаючий навчальний простір. Проєктування такої комплексної мікросервісної системи, що об’єднує передові методи штучного інтелекту, сучасні архітектурні патерни та принципи інклюзивності, є надзвичайно актуальною науково-практичною задачею.

Мета роботи полягає у проєктуванні та програмній реалізації інтелектуальної мікросервісної освітньої платформи «Roadly», яка забезпечує автоматичну генерацію персоналізованих траєкторій навчання, надає інтерактивні інструменти для практики коду та використовує механізми гейміфікації для підвищення залученості користувачів.

Для досягнення поставленої мети визначено такі завдання дослідження.

1. Проаналізувати сучасний ринок EdTech-рішень та обґрунтувати переваги використання генеративного ШІ й мікросервісної архітектури для навчання програмуванню.
2. Спроєктувати багаторівневу архітектуру системи з використанням поліглотного підходу (.NET, Python, Go) та забезпечити ефективну взаємодію між сервісами через асинхронні черги повідомлень.
3. Програмно реалізувати інтелектуальний модуль на основі LangChain для гранулярної генерації навчального контенту та семантичного пошуку у векторному сховищі знань (pgvector).
4. Розробити шлюз контейнеризації мовою Go для забезпечення безпечної роботи користувача в інтерактивному терміналі Web IDE через WebSocket-з'єднання.
5. Створити інтерактивний клієнтський інтерфейс на базі React 19 із впровадженням ігрової економіки та вбудованою панеллю вебдоступності (A11y) для забезпечення інклюзивності навчального процесу.
6. Оцінити ефективність розробленої системи шляхом тестування та аудиту відповідності міжнародним стандартам якості ПЗ.

Об'єкт дослідження – процеси дистанційного навчання та автоматизації формування персоналізованих освітніх траєкторій.

Предмет дослідження – методи, алгоритми та програмні засоби побудови масштабованих освітніх платформ із використанням LLM, технології RAG, контейнеризованих середовищ виконання та мікросервісних патернів.

Наукова новизна отриманих результатів полягає в удосконаленні архітектурного підходу до побудови EdTech-систем шляхом поєднання динамічної генерації контенту через ШІ-агентів із нативною підтримкою інклюзивних профілів

користувачів та двосторонньою інтеграцією навчального процесу між браузерним середовищем і локальною IDE.

Методи дослідження. У роботі використано методи системного аналізу, принципи об’єктно-орієнтованого проєктування, архітектурні шаблони Clean Architecture та CQRS. Для реалізації інтелектуальної логіки застосовано методи prompt-інжинірингу та векторного представлення даних. Для забезпечення надійності інфраструктури використано методи контейнеризації (Docker) та оркестрації сервісів. Для формування складних технічних пояснень щодо архітектури та функціональності додатку з дотриманням технічної чіткості було використано інструменти штучного інтелекту, зокрема велику мовну модель Gemini.

Практичне значення роботи полягає у створенні готового програмного комплексу платформи «Roadly». Розроблене рішення дозволяє студентам отримувати миттєвий зворотний зв’язок від ШІ-ментора, безпечно виконувати код у хмарному середовищі та адаптувати інтерфейс під власні фізіологічні потреби (зокрема за дислексії чи порушень зору). Архітектура системи є відкритою для масштабування та може бути впроваджена як у корпоративному навчанні так і у вищій школі.

Структура роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. Загальний обсяг пояснювальної записки становить 86 сторінок, містить 7 рисунків і 0 таблиць. Список використаних джерел налічує 19 найменувань.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ АРХІТЕКТУРНИХ РІШЕНЬ

1.1 Аналіз сучасного стану систем дистанційного навчання програмуванню та огляд існуючих рішень

Стрімкий розвиток сфери інформаційних технологій провокує постійне зростання попиту на кваліфікованих спеціалістів. Це, у свою чергу, стимулює розвиток ринку освітніх технологій (EdTech). Сьогодні доступ до інформації більше не є проблемою: існують тисячі курсів, відеолекцій, статей і документів. Проте головним викликом сучасної ІТ-освіти стала не кількість матеріалу, а ефективність його засвоєння, рівень мотивації студентів і відсоток успішного завершення навчання.

Для того, щоб визначити основні проблеми предметної області, необхідно проаналізувати наявні підходи до дистанційного навчання програмуванню. Сучасні платформи можна умовно поділити на кілька категорій.

1. Масові відкриті онлайн-курси (МВОК / MOOCs). До цієї категорії належать такі гіганти індустрії, як Coursera, Udemy, edX і Prometheus. Основна парадигма цих платформ базується на попередньо записаних відеолекціях, які доповнюються текстовими матеріалами та стандартизованими тестами (квізами).

Наприклад, на платформі Coursera користувач купує курс і отримує лінійний список відеоуроків.

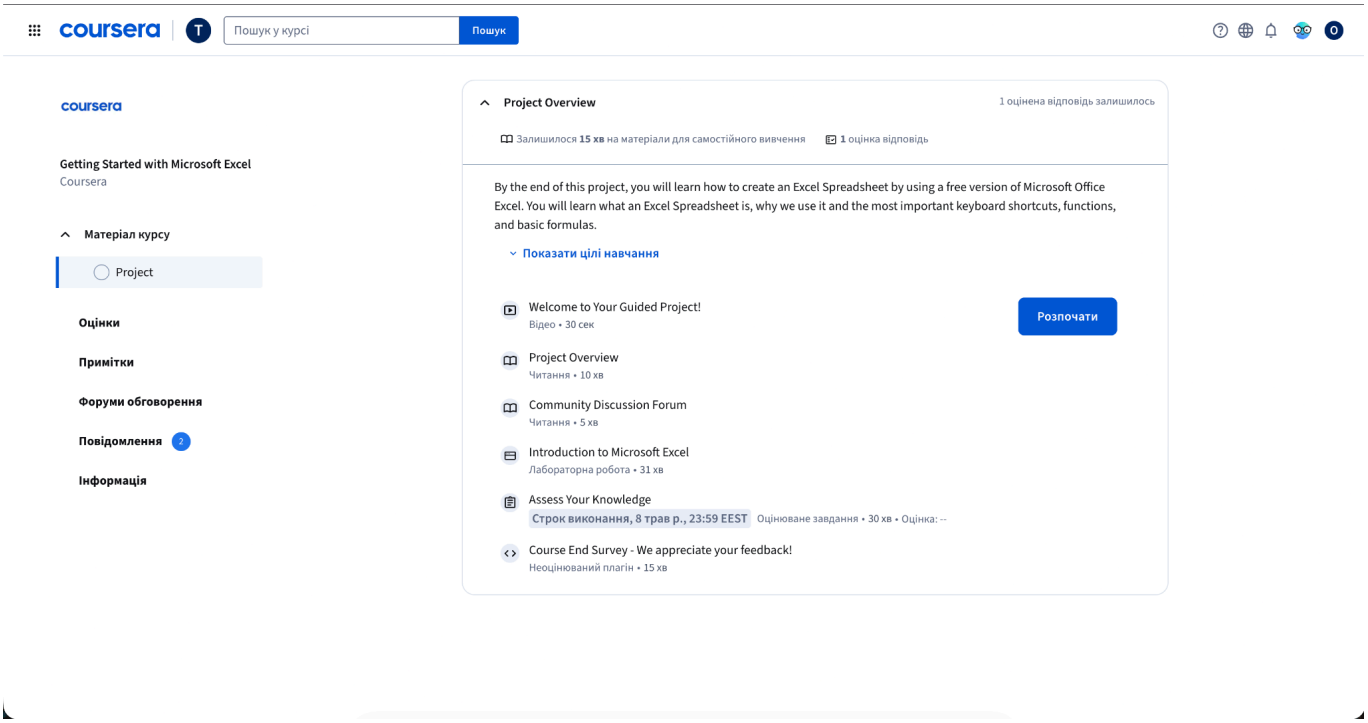

Рис. 1.1 Лінійний інтерфейс проходження курсів на платформі Coursera

Джерело: https://www.coursera.org/learn/introduction-microsoft-excel/lecture/HwUxj/welcome-to-your-guided-project

Незважаючи на популярність, цей підхід має суттєві недоліки в контексті навчання саме програмуванню:

1. **Пасивне споживання інформації:** перегляд відео не формує практичних навичок написання коду.
2. **Відсутність інтерактивності:** студент змушений паралельно відкривати власне середовище розробки, щоб повторювати дії викладача, що розсіює увагу.
3. **Абсолютна лінійність і статичність:** програма курсу однакова для всіх. Студент із базовими знаннями змушений витрачати час на основи, а

абсолютний новачок може не встигати за темпом подачі матеріалу, що призводить до швидкої втрати мотивації.

2. Інтерактивні платформи з вбудованим редактором коду. Більш сучасним підходом є платформи на кшталт Codecademy, freeCodeCamp або DataCamp. Вони відмовляються від довгих відео на користь коротких текстових пояснень, поруч із якими розташовані базовий веб-редактор коду та консоль.

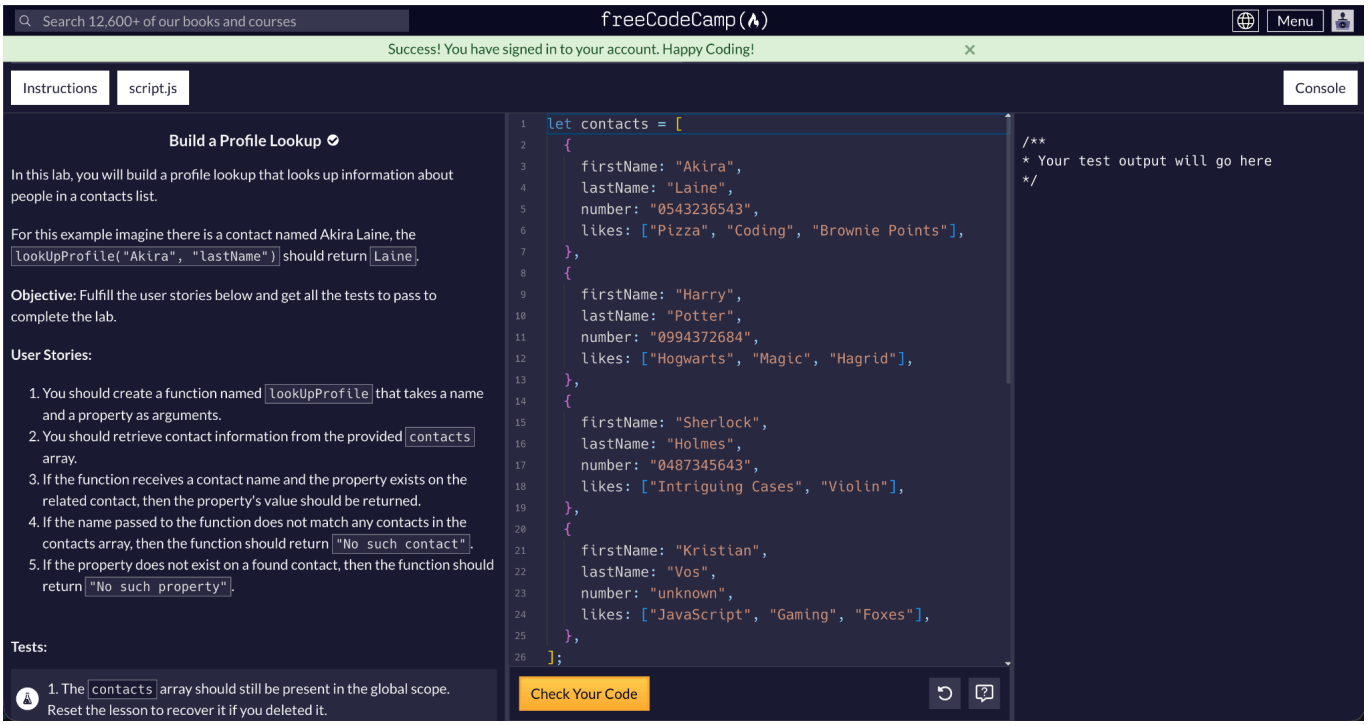

Рис. 1.2 Інтерфейс написання коду на платформі FreeCodeCamp

Джерело: Розроблене автором

Ці системи значно краще підходять для початківців, оскільки змушують писати код від перших хвилин. Однак і вони мають обмеження:

1. **Жорстко запрограмовані перевірки (Hardcoded validations):** система часто перевіряє код за допомогою регулярних виразів або простих unit-тестів. Якщо студент вирішує задачу нестандартним шляхом або

робить синтаксичну помилку, система не здатна пояснити суть проблеми. Вона лише видає стандартне повідомлення про помилку (наприклад, "Expected output X, got Y").

2. **Проблема "штучного середовища":** вбудовані редактори на таких платформах зазвичай підтримують лише один файл і не імітують реальні умови розробки (відсутність роботи з файловою системою, багатофайловими проєктами або базами даних).

3. Платформи для алгоритмічних тренувань (Competitive Programming). Такі ресурси, як LeetCode, HackerRank або Codewars, зосереджені виключно на практиці. Вони надають онлайн-компілятори та складні алгоритмічні задачі.

- **Недолік:** вони не є навчальними платформами в повному розумінні цього слова. Тут відсутня структурована теорія (Roadmap), яка веде студента від нуля до опанування певної технології. Ці платформи розраховані на спеціалістів, які вже знають синтаксис мови та готуються до технічних співбесід.

Систематизація недоліків існуючих систем Підсумовуючи огляд конкурентів та існуючих рішень, можна виділити чотири критичні проблеми сучасного онлайн-навчання програмуванню, які потребують вирішення:

1. **Універсальний підхід (One-size-fits-all):** відсутність персоналізації. Контент розрахований на "середнього" користувача, який ігнорує попередній досвід, індивідуальну швидкість сприйняття та конкретні кар'єрні цілі студента.
2. **Складність налаштування робочого оточення:** Для створення реальних проєктів студентам доводиться самостійно встановлювати інтерпретатори (Python, Node.js), компілятори (.NET), налаштовувати бази даних (PostgreSQL, MongoDB) та Docker[5]-контейнери. Для

багатьох початківців це стає нездоланним бар'єром ще до початку самого програмування.

3. **Брак контекстної підтримки (менторства):** коли студент стикається з помилкою в коді, він змушений залишати платформу й шукати відповідь у пошукових системах (Google, Stack Overflow). Це руйнує контекст навчання (context switching). Якщо ж платформа й дає підказку, то найчастіше вона просто відкриває правильну відповідь, що нівелює сам процес розв’язання проблеми.
4. **Низький рівень залученості за тривалого навчання:** стандартні бали та сертифікати не завжди здатні утримати користувача в довгострокових програмах (наприклад, курсах тривалістю 3-6 місяців).

Виявлені недоліки формують чіткий запит на створення освітньої платформи нового покоління, яка б поєднувала генерацію індивідуальних навчальних маршрутів, надавала повноцінне ізольоване середовище для розробки (Web IDE) безпосередньо у браузері та забезпечувала цілодобову інтелектуальну підтримку користувача без прямого розкриття правильних відповідей.

1.2 Проблема персоналізації та роль генеративного штучного інтелекту в освітньому процесі

Як було визначено у попередньому підрозділі, головним недоліком традиційних систем дистанційного навчання є підхід «один розмір підходить усім» (one-size-fits-all). У класичній моделі створення освітнього контенту вимагає значних людських і часових ресурсів: методисти та викладачі місяцями розробляють навчальну програму, записують лекції та формують тести. Через високу вартість виробництва такого контенту платформи змушені пропонувати єдину стандартизовану програму для максимально широкої аудиторії. Створення

індивідуального курсу для кожного окремого студента в межах класичної парадигми є економічно неможливим.

Вирішення цієї проблеми стало можливим завдяки стрімкому розвитку великих мовних моделей (Large Language Models, LLM) і технологій генеративного штучного інтелекту. Застосування ШІ дозволяє змістити фокус із *попередньої підготовки контенту* на його *динамічну генерацію* в режимі реального часу, спираючись на потреби конкретного користувача.

Генерація персоналізованих навчальних траєкторій (Roadmaps). Замість вибору з каталогу готових курсів сучасна інтелектуальна система здатна функціонувати як цифровий методист. Процес починається зі збору первинних параметрів користувача: поточного рівня знань, кінцевої кар'єрної мети, наявного вільного часу та бажаних технологій. На основі цих даних за допомогою методів інженерії підказок (prompt engineering) формується запит до мовної моделі (наприклад, моделей сімейства Google Gemini або OpenAI GPT[1]).

Модель генерує унікальну структуровану навчальну траєкторію (Roadmap), розбиту на логічні етапи (Checkpoints). Кожен етап динамічно наповнюється поліморфними навчальними елементами: теоретичними довідками, практичними завданнями з написання коду (Coding Tasks) та перевірочними тестами (Quizzes). Це дозволяє адаптувати складність матеріалу: досвідчений користувач отримає стислий опис синтаксису і складні алгоритмічні задачі, тоді як новачок – розгорнуті пояснення базових концепцій.

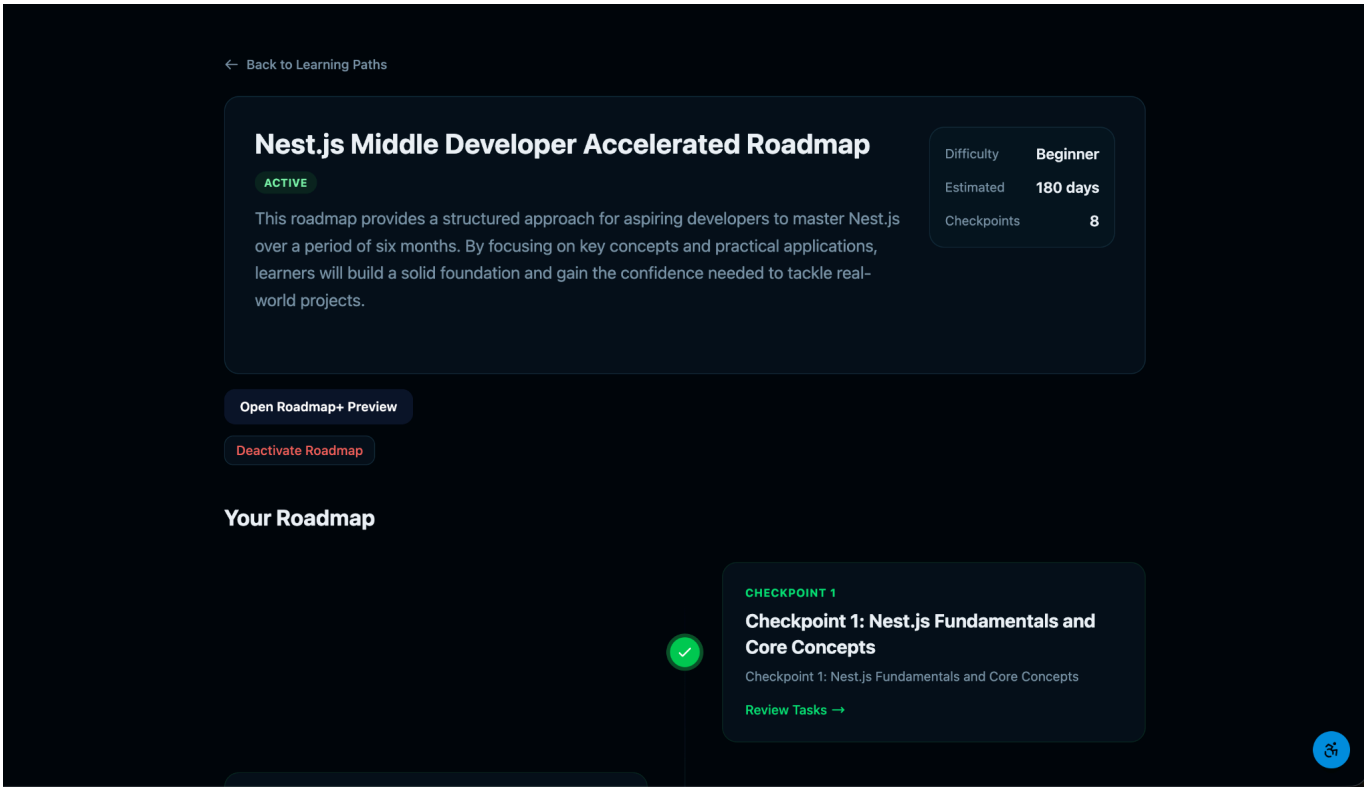

Рис. 1.3 Вигляд згенерованої карти у інтерфейсі додатку

Джерело: Розроблене автором

Вирішення проблеми галюцинацій мовних моделей за допомогою RAG Незважаючи на потужні аналітичні можливості генеративного ШІ, використання базових LLM у чистому вигляді несе серйозні ризики для освітнього процесу. Моделі схильні до так званих «галюцинацій» — генерації правдоподібних, але фактологічно хибних відповідей. Крім того, базові моделі обмежені знаннями, отриманими на момент їхнього тренування, і не мають доступу до специфічної або оновленої документації.

Для забезпечення високої точності навчального контенту та надання релевантних підказок застосовується архітектурний підхід RAG (Retrieval-Augmented Generation — генерація, доповнена пошуком). Цей підхід складається з двох основних етапів:

1. **Етап індексації (Ingestion):** Навчальні матеріали, актуальна документація та статті збираються (web scraping), очищуються та розбиваються на невеликі семантичні фрагменти (чанки). За допомогою моделей перетворення тексту (Embedding models) ці фрагменти конвертуються в багатовимірні вектори та зберігаються у спеціалізованій векторній базі даних (наприклад, PostgreSQL з розширенням pgvector).
2. **Етап пошуку та генерації (Retrieval & Generation):** Коли студент ставляє запитання ШІ-ментору, його запит також векторизується. Система виконує пошук семантично найближчих фрагментів у векторній базі даних. Знайдені перевірені факти додаються до контексту запиту мовній моделі, яка формує остаточну, фактологічно точну відповідь.

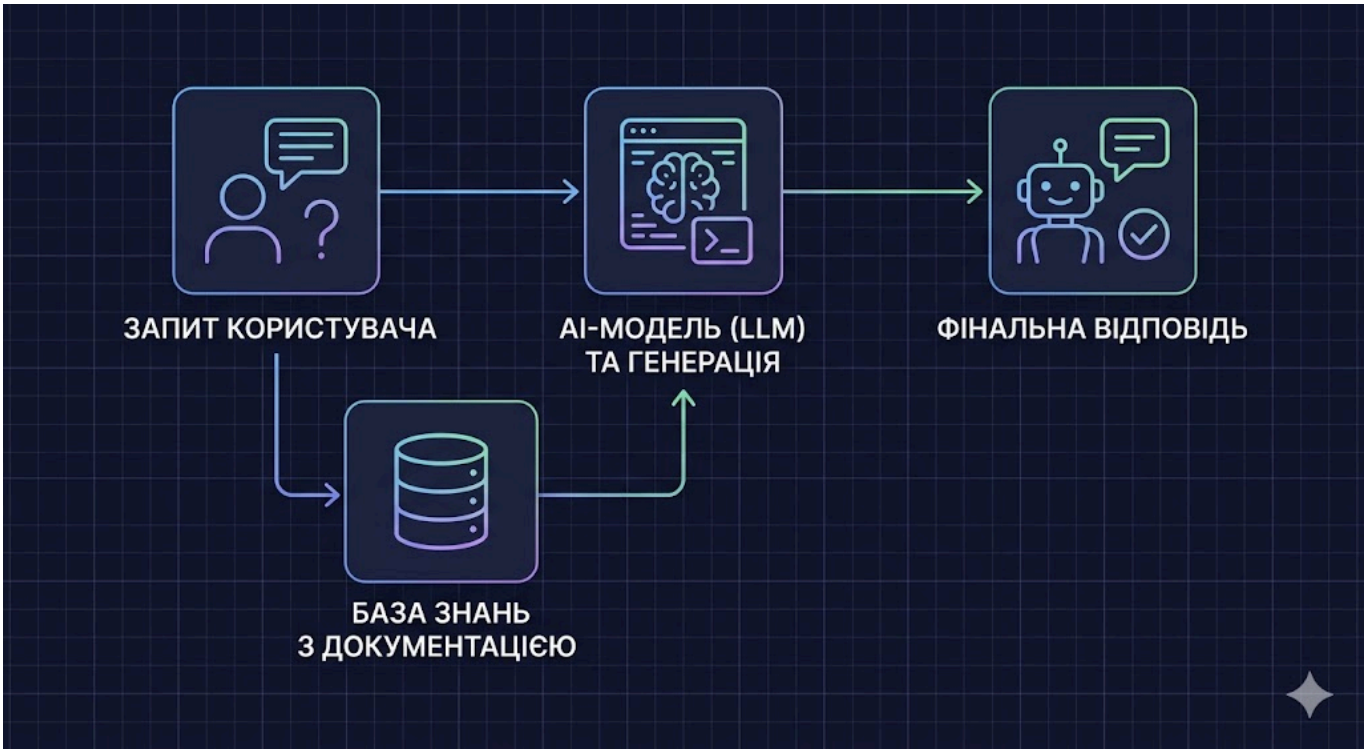

Рис. 1.4 Процес генерації карти

Джерело: Розроблене автором з використанням інструментів ШІ

Контекстне асистування замість прямих підказок Ще однією важливою проблемою є методологія надання допомоги. Коли студент стикається з помилкою в коді, проста видача правильного рішення (або згенерованого коду) зводить нанівець освітню цінність завдання. Інтелектуальний агент (AI Mentor) повинен діяти за принципами сократівського діалогу. Завдяки інтеграції LLM з інструментами виконання коду (tool-calling), система здатна:

- Проаналізувати код, написаний користувачем.
- Отримати текст помилки від компілятора в ізольованому середовищі виконання (Docker-контейнер).
- Згенерувати підказку, яка пояснює природу помилки та спрямовує студента до правильного рішення, не розкриваючи її повністю.

Отже, інтеграція технологій генеративного ШІ та архітектури RAG дає змогу трансформувати статичну платформу на високоадаптивну екосистему. Це забезпечує зниження когнітивного навантаження на студентів, підвищує їхню залученість завдяки цілодобовій контекстній підтримці та гарантує актуальність і достовірність навчальних матеріалів.

1.3 Аналіз підходів до організації інтерактивних середовищ виконання коду (Web IDE)

Незважаючи на високу якість теоретичного матеріалу та наявність інтелектуальних підказок, фундаментальною основою навчання програмуванню є безпосереднє написання та запуск коду. Як було визначено раніше, необхідність самостійного налаштування локального середовища розробки (встановлення інтерпретаторів, компіляторів, налаштування змінних оточення та баз даних) створює значний бар'єр для початківців і часто стає причиною відмови від подальшого навчання.

Для усунення цього бар’єра сучасні освітні платформи переходять до концепції хмарних середовищ для розробки (Web IDE). Web IDE дозволяє студенту писати, компілювати та тестувати код безпосередньо у вікні веб-браузера без встановлення жодного додаткового програмного забезпечення.

В інженерній практиці існують два основні підходи до реалізації користувацького коду в браузері: клієнтський і серверний.

1. Клієнтське виконання (Client-side execution). Цей підхід базується на виконанні коду безпосередньо в браузері користувача за допомогою технологій JavaScript або WebAssembly (WASM). Наприклад, для мови Python існує проєкт Pyodide, який компілює інтерпретатор Python у WASM, дозволяючи виконувати скрипти прямо на клієнті.

- *Переваги:* нульові витрати на серверну інфраструктуру (compute resources), відсутність затримок (мережевого пінгу) під час виконання, абсолютна безпека для сервера (оскільки код виконується в пісочниці браузера).
- *Недоліки:* суттєві обмеження на використання сторонніх бібліотек, неможливість повноцінної роботи з файловою системою, неможливість запуску складних застосунків (наприклад, веб-серверів на Node.js) і важких баз даних (PostgreSQL, MongoDB). Цей підхід підходить лише для розв'язання простих алгоритмічних задач в одному файлі.

2. Серверне виконання в ізольованих контейнерах (Server-side execution). Цей підхід передбачає, що фронтенд (браузер) слугує лише терміналом (тонким клієнтом), який надсилає написаний код на віддалений сервер, де він компілюється та виконується, а результат повертається користувачу. Для забезпечення безпеки основної інфраструктури від шкідливого або помилкового коду використовуються технології контейнеризації (наприклад, Docker). Кожен користувач отримує власний,

тимчасовий та ізольований контейнер із суворими обмеженнями ресурсів (CPU, RAM).

- *Переваги:* повне відтворення реальних умов розробки. Користувач може працювати з багатофайловими проєктами, встановлювати будь-які пакети через пакетні менеджери (npm, pip), запускати бази даних (SQL та NoSQL) і піднімати власні веб-сервери.
- *Недоліки:* високі вимоги до серверної інфраструктури, необхідність організації складного механізму передачі даних у реальному часі (через WebSockets) та високі ризики безпеки у разі неправильного налаштування ізоляції (вразливості типу Container Breakout).

Рис. 1.5 – Порівняльна схема архітектур клієнтського та серверного виконання коду

Джерело: Розроблене автором з використанням інструментів ШІ

Вимоги до Web IDE для платформи Roadly Оскільки платформа Roadly орієнтована на комплексне навчання, що включає розробку повноцінних застосунків, роботу з базами даних (SQL/NoSQL) та бекенд-фреймворками, клієнтський підхід

(WASM) є недостатнім. Було прийнято рішення проєктувати Web IDE на основі серверної контейнеризації.

Для забезпечення користувацького досвіду (UX), який не поступається десктопним редакторам (таким як VS Code), система Web IDE повинна складатися з наступних логічних блоків:

1. **Редактор коду:** інтеграція потужного текстового редактора на клієнтській стороні (наприклад, Monaco Editor) з підтримкою підсвічування синтаксису, автодоповнення та навігації файловим деревом.
2. **Емулятор термінала:** надання користувачу інтерактивної командної оболонки (на базі xterm.js), що з'єднана з псевдотерміналом (PTY) у Docker-контейнері через протокол WebSocket. Це дозволить виконувати команди (наприклад, npm install або python main.py) у реальному часі.
3. **Синхронізація робочого простору:** спеціалізований API для передачі файлів між клієнтом і змонтованою директорією (/workspace) у контейнері.
4. **Зворотне проксування (Reverse Proxy):** механізм динамічного маршрутизатора, який дозволяє користувачу відкрити в браузері веб-додаток, піднятий на певному порту всередині ізольованого навчального контейнера.

Таким чином, розробка спеціалізованого сервісу-шлюзу (Proxy), здатного оркеструвати життєвий цикл Docker-контейнерів і безпечно маршрутизувати WebSocket-трафік, стає одним із найважливіших архітектурних завдань під час створення освітньої платформи.

1.4 Обґрунтування вибору технологічного стека та постановка задачі

Створення комплексної освітньої платформи, що поєднує транзакційну логіку, генеративний штучний інтелект і роботу з ізольованими контейнерами, вимагає застосування мікросервісної архітектури. Такий підхід дозволяє використовувати

парадигму поліглотного програмування — вибір найкращого інструменту (мови та фреймворку) для розв’язання конкретного класу задач у межах одного продукту.

На основі аналізу предметної області та сформованих архітектурних вимог було обрано наступний стек технологій для платформи «Roadly»:

1. Доменне ядро та транзакційна логіка (C#/.NET 9). Для реалізації головного бекенду (керування профілями, прогресом, навчальними маршрутами та гейміфікацією) обрано платформу **.NET 9[2] (C#)**.

- *Обґрунтування:* C# є строго типізованою мовою корпоративного рівня, яка ідеально підходить для побудови надійних фінансових і бізнес-систем (зокрема, внутрішнього магазину Roadly+ та системи досягнень). Використання Entity Framework Core забезпечує надійну роботу з реляційною базою даних **PostgreSQL**, гарантуючи ACID-транзакції. Крім того, екосистема .NET надає потужні інструменти для реалізації патерну CQRS (через бібліотеку MediatR) та побудови Clean Architecture.

2. Інтелектуальна підсистема та AI-оркестрація (Python 3.12 / FastAPI[6]). Сервіс генерації контенту, чат-агент та RAG-пайплайн винесені в окремий мікросервіс мовою **Python**.

- *Обґрунтування:* Python є стандартом де-факто у сфері machine learning та штучного інтелекту. Це дозволяє природно використовувати екосистему **LangChain** для побудови агентів і роботи з LLM (Google Gemini, OpenAI). Для зберігання векторизованих знань (Embeddings) найефективнішим рішенням є розширення **pgvector** для PostgreSQL. Оскільки генерація освітніх карт є тривалим процесом (може займати десятки секунд), для управління фоновими задачами обрано зв'язку **Celery[4] + Redis**, що гарантує відмовостійкість за високих навантажень.

3. Шлюз контейнеризації та емуляції термінала (Go 1.24). Розробка сервісу-шлюзу (docker-pty-proxy), який відповідає за зв'язок браузера студента з Docker-контейнером, виконана мовою **Go (Golang)**.

- *Обґрунтування:* Робота з безперервними потоками даних (WebSocket PTY-сесії) та зворотним проксуванням (Reverse Proxy) вимагає максимальної продуктивності й ефективної роботи з мережею. Завдяки моделі легкоатлетних потоків (goroutines) Go здатний підтримувати тисячі одночасних WebSocket-з'єднань із мінімальним споживанням оперативної пам'яті. Крім того, офіційний Docker Engine SDK написаний на Go, що забезпечує найбільш стабільну та безпечну інтеграцію з демоном контейнеризації.

4. Клієнтські інтерфейси (React 19 / TypeScript)

Як основу для користувацького веб-застосунку обрано бібліотеку **React** у поєднанні з інструментом збірки Vite та мовою **TypeScript**.

- *Обґрунтування:* Компонентна архітектура React дозволяє гнучко розробляти складні інтерактивні інтерфейси (наприклад, графічне відображення Roadmap-ів або ігрової карти Roadly+). TypeScript забезпечує строгу типізацію на клієнті, що мінімізує кількість помилок під час інтеграції з API. Для складного управління станом (Server State) застосовано **RTK Query**, а для оптимізації рендерингу Web IDE — бібліотеку **Zustand**.

5. Інфраструктура та міжсервісна взаємодія. Для уникнення жорсткої зв'язності (tight coupling) між .NET-ядром та Python-сервісом обрано брокера повідомлень **RabbitMQ[7]**. Це дозволяє асинхронно передавати події (наприклад, запит на генерацію карти), тоді як результати доставляються клієнту в режимі реального часу за допомогою технологій **SignalR** (у .NET) та **SSE** (Server-Sent Events у Python).

Постановка задачі на кваліфікаційну роботу. На основі проведеного аналізу предметної області та обраного технологічного стека метою цієї роботи є практична реалізація освітньої платформи «Roadly». Для досягнення мети необхідно вирішити наступні практичні завдання:

1. Спроєктувати та реалізувати архітектуру доменного ядра на базі .NET.
2. Розробити AI-підсистему (Python) для автоматизованої генерації навчальних матеріалів і контекстного менторства за допомогою методології RAG.
3. Побудувати захищений шлюз контейнеризації мовою Go для реалізації функціоналу Web IDE.
4. Розробити єдиний клієнтський веб-інтерфейс (React), який об'єднає навчальний процес, середовище розробки та елементи гейміфікації.
5. Розробити MVP-розширення для Visual Studio Code для забезпечення безшовної інтеграції навчального процесу в середовище розробника.

Висновки до розділу 1

У першому розділі проаналізовано сучасний стан ринку освітніх технологій (EdTech). Встановлено, що головним недоліком існуючих платформ є використання статичного підходу до подання матеріалу, який не враховує індивідуальних потреб студентів і призводить до зниження їхньої мотивації. Доведено, що ефективним вирішенням є створення адаптивних середовищ на базі великих мовних моделей (LLM) та RAG-систем. Це дозволяє автоматизувати побудову персоналізованих навчальних маршрутів (Roadmaps) і забезпечити релевантну менторську підтримку.

Також проведено порівняльний аналіз методів інтерактивного виконання коду в браузері. Обґрунтовано, що для забезпечення безпечного та максимально реалістичного середовища розробки (Web IDE) найдоцільніше використовувати серверне виконання в ізольованих Docker-контейнерах.

Результатом дослідження став сформований комплекс функціональних і нефункціональних вимог, який став архітектурною базою для подальшого проєктування мікросервісної платформи «Roadly».

РОЗДІЛ 2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА ІНТЕЛЕКТУАЛЬНИХ МОДУЛІВ ПЛАТФОРМИ

2.1 Проєктування мікросервісної архітектури та контейнерної інфраструктури

Комплексність завдань, що вирішуються платформою «Roadly» — від транзакційного керування навчальним прогресом до високоінтенсивної роботи з великими мовними моделями (LLM) та забезпечення доступу до ізольованих середовищ розробки — унеможливлює використання традиційної монолітної архітектури. Монолітний підхід у цьому контексті призвів би до критичних проблем із горизонтальним масштабуванням та «шумними сусідами» (noisy neighbors): пікові навантаження під час генерації ШІ-контенту могли б повністю заблокувати обробку звичайних HTTP-запитів користувачів до основної бази даних.

Для забезпечення відмовостійкості, незалежного розгортання модулів та реалізації парадигми поліглотного програмування систему було спроєктовано на базі мікросервісної архітектури. Технологічний стек кожного сервісу підбирався відповідно до специфіки його задач (наприклад, Go для мережевих потоків, Python для AI-оркестрації). Повна інфраструктура системи складається з **11 ізольованих контейнерів**, що взаємодіють у межах віртуальної мережі.

2.1.1 Функціональна декомпозиція мікросервісів

Архітектура платформи включає п'ять основних бізнес-орієнтованих вузлів:

1. **Доменне ядро (roadly-api на .NET 9):** побудоване за принципами чистої архітектури (Clean Architecture) із застосуванням патернів CQRS і Repository. Сервіс є «джерелом істини» для бізнес-сутностей. Він керує користувачами, навчальними планами (збереженими в **JSONB**), прогресом і ігровою

економікою. Ядро інтегроване з **AWS Cognito[3]** для авторизації та використовує **SignalR** для миттєвої доставки системних сповіщень.

2. **Інтелектуальна підсистема (python_service на FastAPI):** Виконує роль API-шару для AI-операцій. Він відповідає за обробку вхідних запитів до ШІ-ментора, управління контекстом діалогу та реалізацію **SSE (Server-Sent Events)** для посимвольної трансляції відповідей.
3. **Шар AI-обробки (roadly-python-worker на Celery):** окремий вузол для виконання обчислювально важких завдань. Розділення Python-частини на API та Worker дозволяє масштабувати генерацію навчальних карт незалежно від чат-сервісу. Воркер виконує основну логіку **LangChain**, скрейпінг через **Tavily** та взаємодію з LLM-провайдерами (Gemini, OpenAI).
4. **Шлюз контейнеризації (roadly-docker-pty-proxy на Go):** високопродуктивний міст (bridge) між клієнтом і інфраструктурою Docker. Завдяки використанню **Docker Engine SDK** та моделі конкурентності Go, сервіс утримує WebSocket-з'єднання з терміналами користувачів та забезпечує роботу Filesystem API.
5. **Клієнтські інтерфейси (roadly-ui на React 19):** Єдиний фронтенд-застосунок, що агрегує дані з трьох різних бекенд-вузлів, забезпечуючи безшовний користувацький досвід (UX).

2.1.2 Проєктування шару даних та векторного пошуку

Одним із ключових архітектурних рішень є фізичне розділення сховищ для різних типів даних, що відображено у конфігурації портів та інстансів СУБД:

- **Транзакційна БД (roadly-postgres, порт 5432):** використовує PostgreSQL 17. Тут зберігаються реляційні дані та база даних Hangfire для recurring-задач (наприклад, очищення старих сесій docker-zombie-reaper).

- **Векторна БД (roadly_vector_db, порт 5433):** окремий інстанс PostgreSQL із розширенням **pgvector**. Таке рішення дозволяє ізолювати складні операції семантичного пошуку (RAG) від основної бізнес-логіки, запобігаючи деградації продуктивності API.
- **NoSQL Sandbox (roadly-mongodb, порт 27017):** Надає ізольоване середовище для практичних завдань. Для кожного студента динамічно створюється унікальна колекція, що гарантує безпеку даних у багатокористувацькому режимі.

2.1.3 Патерни міжсервісної взаємодії та брокери повідомлень

Для забезпечення надійного зв'язку між компонентами реалізовано комбіновану стратегію:

1. **Асинхронна черга (RabbitMQ 3-management):** Виступає центральним брокером. .NET-ядро публікує події генерації в RabbitMQ, які споживаються Python-воркером. Це реалізує механізм **Event-Driven Architecture**, де система залишається чуйною навіть за тривалої генерації контенту.
2. **Швидкий кеш та брокер задач (Redis 7.4):** використовуються виключно як транзитне сховище для Celery та для збереження тимчасових станів сесій Web IDE. Це забезпечує мінімальні затримки під час передачі задач між API та Worker.
3. **Захищений внутрішній контур (Internal Service Key):** Усі сервіс-сервіс виклики (наприклад, від Python до .NET для перевірки прогресу) проходять через Middleware, який верифікує спеціальний заголовок. Це створює додатковий рівень безпеки в межах мережі Docker.
4. **Легковажне виконання (roadly_piston):** Для миттєвої перевірки простих фрагментів коду інтегровано рушій Piston[19]. Це дозволяє уникнути оверхеду

під час створення повноцінних Docker-контейнерів для базових алгоритмічних задач.

Така топологія системи робить платформу «Roadly» готовою до високих навантажень, забезпечує високу щільність користувачів на сервері та гарантує стабільність освітнього процесу.

2.2 Проєктування баз даних та стратегії інтелектуального зберігання

Ефективність функціонування мікросервісної платформи «Roadly» безпосередньо залежить від архітектури зберігання даних. Враховуючи гетерогенність завдань від забезпечення строгої транзакційної цілісності для фінансової та бізнес-логіки до швидкого семантичного пошуку для штучного інтелекту — проєктування шару даних вимагало гібридного підходу та фізичного розділення сховищ.

2.2.1 Гібридна реляційна модель доменного ядра (PostgreSQL)

Для зберігання основних бізнес-сутностей було прийнято стратегічне рішення використати СКБД PostgreSQL 17[12] (контейнер roadly-postgres). Такий підхід зумовлений кількома факторами:

- **Транзакційна надійність (ACID):** Повна підтримка транзакцій є критично важливою для збереження навчального прогресу та операцій у внутрішньому магазині (таблиця coin_transactions). Це гарантує, що у разі збою під час транзакції в системі гейміфікації Roadly+ монети не будуть списані без надання відповідного інвентарю.
- **Нормалізована архітектура навчальних маршрутів:** Попри динамічну природу даних, що генеруються штучним інтелектом, програмна реалізація спирається на сувору реляційну модель. Замість зберігання маршрутів у вигляді неструктурованих JSONB-документів їхню структуру

декомпозиціонується на пов’язані таблиці (наприклад, Roadmaps, Checkpoints, Tasks). Такий підхід довів свою вищу практичність та гнучкість: він забезпечує швидкі часткові оновлення (наприклад, зміна статусу лише одного практичного завдання), дозволяє використовувати ефективні B-Tree індекси та гарантує цілісність даних на рівні зовнішніх ключів (Foreign Keys).

- **Мультибазовість:** В межах одного контейнера ізольовано зберігаються основна база платформи та службова база Hangfire, що використовується для оркестрації фонових recurring-задач (наприклад, регулярного автоматизованого очищення «мертвих» сесій Docker-контейнерів).

1. **Мультибазовість:** В межах одного контейнера ізольовано зберігаються основна база платформи та службова база Hangfire, що використовується для оркестрації фонових recurring-задач (наприклад, регулярного очищення «мертвих» сесій Docker-контейнерів).

2.2.2 Векторне сховище для RAG (pgvector)

Для того, щоб AI-ментор міг надавати студентам актуальні та фактологічно правильні підказки, замість класичного текстового пошуку застосовується математичний векторний пошук. Для цієї задачі розгорнуто **абсолютно ізольований інстанс** бази даних — roadly_vector_db (на порту 5433).

- **Ізоляція ресурсів:** Фізичне винесення векторної бази в окремий контейнер є важливим архітектурним рішенням. Обчислення косинусної подібності (Cosine Similarity) між багатовимірними векторами (768 або 1536 вимірів) під час пошуку є процесорно-інтенсивною операцією. Ізоляція гарантує, що пікові навантаження від ШІ-чатів не призведуть до деградації (сповільнення) основного .NET-домену.

- **Структура даних:** Ключовою колекцією є learning_resources, де зберігаються очищені фрагменти документації (Chunks), їхні математичні представлення (Embeddings) та метадані (Tags, URL) для попередньої фільтрації контексту.

2.2.3 Ізольовані середовища для практичних завдань (NoSQL Sandbox)

Платформа «Roadly» передбачає не лише написання коду, а й практичну роботу з базами даних. Надавати користувачам прямий доступ до основної СУБД-платформи є порушенням правил безпеки.

- Для виконання SQL-завдань використовується In-Memory база даних безпосередньо у віртуальному оточенні користувача, яка знищується після завершення сесії.
- Для виконання завдань з нереляційними даними розгорнуто окремий контейнер **MongoDB** (roadly-mongodb). Під час запуску NoSQL-задачі система динамічно створює унікальну ізольовану колекцію (Collection) для конкретного студента. Це дозволяє безпечно практикуватися в агрегаціях і маніпуляціях з документами без ризику впливу на дані інших користувачів або на систему загалом.

2.2.4 Роль Redis в оркестрації тимчасових даних

У загальній інфраструктурі сховищ **Redis 7.4** виконує роль високопродуктивного брокера для фонових задач (Celery) та швидкого сховища (In-Memory Store). Важливо підкреслити архітектурну особливість: **відповіді великих мовних моделей (LLM) не кешуються в Redis**. Це обґрунтовано двома факторами:

1. **Економіка API:** Сучасні моделі (Gemini 1.5+, GPT-4o) мають вбудовані механізми Prompt Caching на рівні провайдера, що автоматично знижує вартість повторного використання токенів.

2. **Динаміка навчання:** Навчальний процес вимагає унікальних, контекстно-залежних відповідей, що спираються на поточний неробочий код студента. Кешування призвело б до видачі нерелевантних підказок. Тому Redis використовується виключно для зберігання ідентифікаторів асинхронних задач, логування стану Python-воркерів та збереження метаданих активних WebSocket-сесій Web IDE.

2.2.5 Подієво-орієнтована синхронізація (RabbitMQ)

Оскільки дані фізично розподілені між різними базами (roadly-postgres та roadly_vector_db), для підтримки їхньої узгодженості (Data Consistency) використовується брокер повідомлень **RabbitMQ**. Наприклад, коли в .NET-ядрі створюється новий навчальний модуль, генерується подія ResourceCreatedEvent. Python-воркер підхоплює її з черги, виконує скрейпінг матеріалу, векторизацію та зберігає дані в pgvector. Цей патерн Event-Driven Architecture гарантує, що бази даних синхронізовані без створення жорсткої зв'язності (Tight Coupling) між мікросервісами.

2.3 Архітектура інтелектуального модуля та еволюція RAG-пайплайну

Інтелектуальна підсистема платформи «Roadly», реалізована у вигляді ізольованого мікросервісу на базі Python та FastAPI, відповідає за оркестрацію роботи з великими мовними моделями (LLM). У процесі проєктування цього модуля архітектурні підходи зазнали суттєвої еволюції, що було зумовлено необхідністю вирішення проблем із продуктивністю, відмовостійкістю та фактологічною точністю згенерованого контенту.

2.3.1 Еволюція генерації контенту: від монолітної до гранулярної архітектури

Головним завданням інтелектуального модуля є створення унікальних навчальних карт (Roadmaps). Початкова архітектурна концепція (Legacy-підхід) базувалася на монолітній генерації: система формувала один масивний запит (Prompt), який вимагав від LLM повернути повністю готовий курс, включаючи теоретичні блоки, тестові запитання та код завдань. Цей підхід виявив три критичні системні вразливості:

1. **Переповнення контекстного вікна:** Великий обсяг вихідних даних призводив до того, що модель втрачала фокус, «забувала» початкові інструкції та починала генерувати нерелевантний або синтаксично некоректний JSON.
2. **Високий ризик галюцинацій:** Через необхідність одночасно концентруватися на структурі та глибоких технічних деталях, якість навчального матеріалу суттєво знижувалася.
3. **Нульова відмовостійкість (All-or-Nothing):** Якщо генерація переривалася на 95% виконання через мережеву помилку або тайм-аут API-провайдера, весь результат втрачався, що призводило до марної витрати коштів і часу.

Для вирішення цих проблем систему було перепроєктовано на **гранулярну (розподілену) модель генерації**:

- **Етап 1 (Генерація каркаса):** LLM створює лише загальний «скелет» курсу (метадані, назви етапів-чекпойнтів і короткі тези).
- **Етап 2 (Асинхронне наповнення):** Для кожного створеного чекпойнту доменне ядро публікує окремі події у брокері **RabbitMQ**. Воркери (python-worker на базі Celery) підхоплюють ці задачі та паралельно генерують контент (теорію, тести, код). Такий підхід дозволив виділити для кожної мікрозадачі максимальний обсяг контекстного вікна, знизити ризик

галюцинацій та забезпечити стійкість: у разі помилки генерації одного конкретного тесту система виконує повторну спробу (retry) лише для цього тесту, не руйнуючи весь курс.

2.3.2 Динамічне збагачення бази знань (Tavily Scraper)

Для забезпечення високої якості роботи AI-ментора використовується технологія RAG (Retrieval-Augmented Generation). Проте замість використання статичної, попередньо завантаженої бази документів, у Roadly реалізовано пайплайн динамічного збагачення бази знань.

Під час генерації навчального контенту LLM пропонує перелік рекомендованих посилань на офіційну документацію (наприклад, React або PostgreSQL). Далі запускається автоматизований процес індексації:

1. Система ініціює виклик до API **Tavily** — спеціалізованого пошукового рушія, оптимізованого для LLM. Tavily виконує скрейпінг (збір даних) за вказаним URL, повертаючи чистий текст, очищений від HTML-розмітки, реклами та навігаційних меню.
2. Отриманий текст проходить етап **Chunking** — розбиття на невеликі семантичні блоки (близько 1000 токенів) із заданим перекриттям (Overlap), щоб не розірвати контекст складних речень.
3. Кожен чанк перетворюється на математичний вектор (Embedding) і зберігається в ізольованому контейнері pgvector. Таким чином, платформа постійно «навчається», поповнюючи власну векторну базу актуальними статтями, які використовуються для подальших відповідей студентам.

2.3.3 Інтерактивне менторство, Context Injection та Streaming

Для надання консультацій студентам під час виконання практичних завдань застосовується AI-агент. Оскільки затримка відповіді (Latency) критично впливає на

користувацький досвід (UX), класичний HTTP-запит/відповідь було замінено на технологію потокової передачі даних **Server-Sent Events (SSE)**. FastAPI-контролер відкриває односторонній потік і транслює згенеровані токени безпосередньо в браузер, створюючи ефект живого друкування.

Архітектура агента базується на патерні **Context Injection**. Перед кожним зверненням до LLM система автоматично перехоплює запит і непомітно для користувача формує розширений System Prompt. До нього ін'єктується:

- Поточний код, який написав користувач у Web IDE (включаючи синтаксичні помилки).
- Історія останніх повідомлень у розмові (Conversation History).
- Релевантні підказки, знайдені у pgvector через RAG-пошук. Крім того, агент наділений інструментарієм (Tool-Calling) для взаємодії з .NET-ядром через захищені Internal API, що дозволяє йому перевіряти правильність проходження тестів та отримувати специфікації завдань безпосередньо з транзакційної бази даних.

2.4 Проєктування середовищ виконання коду та шлюзу контейнеризації

Надання користувачам можливості повноцінної роботи в терміналі через браузер вимагає реалізації механізму **псевдотерміналів (PTY)**. Це дозволяє серверу «обманювати» процеси (наприклад, bash або node), змушуючи їх думати, що вони запущені в реальному вікні термінала і забезпечуючи коректну передачу керуючих символів (Ctrl+C, таби, форматування кольорів).

2.4.1 Аналіз початкових рішень та технологічний бар'єр

Початковий архітектурний план передбачав реалізацію всієї логіки керування контейнерами та термінальними сесіями безпосередньо в межах доменного ядра на

базі **.NET 9 (C#)**. Проте під час прототипування було виявлено ряд критичних обмежень:

1. **Проблема бібліотек:** в екосистемі .NET відсутні зрілі та стабільні бібліотеки для нативного керування PTY-сесіями в середовищі Linux-контейнерів. Існуючі рішення або є застарілими (legacy), або вимагають використання складних P/Invoke-викликів до системних бібліотек C, що знижує стабільність бекенду та ускладнює його підтримку.
2. **Накладні витрати (Overhead):** Модель виконання .NET, орієнтована на обробку коротких HTTP-запитів, є занадто «важкою» для утримання тисяч тривалих, «сплячих» WebSocket-з'єднань, у яких передаються мінімальні обсяги байтів термінального введення/виведення.
3. **Docker SDK Integration:** Хоча для C# існує Docker.DotNet, офіційна підтримка та повнота функціоналу Docker Engine SDK мовою Go є вищими, оскільки сам Docker написаний на Go.

2.4.2 Обґрунтування створення мікросервісу мовою Go

На основі проведеного аналізу було прийнято стратегічне рішення винести логіку Web IDE в окремий інтеграційний мікросервіс — **docker-pty-proxy**, реалізований мовою **Go 1.24**. Це рішення аргументується наступними перевагами:

- **Нативна підтримка PTY:** Go має потужну стандартну бібліотеку та перевірені часом пакети (наприклад, creack/pty), які дозволяють легко й безпечно взаємодіяти з системними терміналами.
- **Ефективна конкурентність:** механізм **goroutines** дозволяє обслуговувати кожну термінальну сесію в окремому потоці (що споживає лише кілька кілобайтів пам'яті), забезпечуючи миттєву реакцію термінала на дії користувача.

- **Безпечний міст (Proxy-bridge):** Відокремлення Go-сервісу від доменного ядра створює додатковий рівень безпеки. Навіть у разі критичної помилки в термінальному шлюзі основна база даних та бізнес-логіка в .NET API залишаться ізольованими й захищеними.

2.4.3 Багаторівнева архітектура виконання: Mini-IDE vs Docker

Для оптимізації ресурсів інфраструктури та дотримання принципів **SOLID**, система підтримує два типи ранерів:

1. **Легковажний ранер (Piston):** Використовується для базових задач, де не потрібна робота з файловою системою чи мережею. Це дозволяє економити ресурси хоста.
2. **Повноцінне оточення (Docker):** Активується через Go-проксі для складних проєктів (наприклад, React- або ASP.NET-застосунків). Завдяки принципу **Dependency Inversion** .NET-ядро спілкується з виконавцями через абстракцію ICodeExecutor, що дозволило безболісно інтегрувати Go-сервіс у загальну екосистему без втручання в бізнес-логіку.

2.4.4 Стабільність та захист від зависань (Cancellation Tokens)

Окремим інженерним челенджем стала боротьба з «нескінченними циклами» в коді користувачів. Для запобігання вичерпанню ресурсів сервера, на рівні .NET ядра та Go-шлюзу всі операції виконуються з обов’язковим використанням **CancellationToken**. Якщо виконання коду або SQL-запиту в In-Memory БД триває понад встановлений ліміт (наприклад, 5 секунд), система примусово завершує процес, гарантуючи стабільність платформи для інших студентів.

2.5 Дизайн ігрового простору та підсистеми гейміфікації (Roadly+)

Ефективність будь-якої освітньої платформи вимірюється не лише якістю матеріалу, а й показником утримання користувачів (User Retention Rate). Сучасні системи дистанційного навчання стикаються з проблемою стрімкого зниження мотивації студентів. В епоху глобалізації та домінування соціальних мереж користувачі звикають до отримання «швидкого дофаміну» від миттєвого споживання короткого контенту. Навчання програмуванню, навпаки, вимагає значних когнітивних зусиль, тривалої концентрації та відкладеної винагороди. Стандартні індикатори прогресу (наприклад, лінійні списки завдань або відсоток завершення курсу) більше не здатні ефективно утримувати увагу студентів.

Для вирішення цієї проблеми в архітектуру платформи було інтегровано підсистему гейміфікації **«Roadly+»**. Її головна мета — трансформація рутинного навчального процесу в інтерактивний рольовий ігровий простір (RPG), що стимулює користувача до щоденної активності («грінду») через систему миттєвих винагород.

2.5.1 Віртуальна економіка та транзакційна цілісність

В основі Roadly+ лежить віртуальна економіка. За кожну успішну дію на платформі (вирішення задачі з кодингу, проходження тесту, читання теоретичного блоку) користувачу нараховується внутрішня валюта — **Roadly Coins**.

З інженерної точки зору, збереження балансу монет не реалізовано як просте числове поле (integer) у таблиці користувачів. Такий підхід є вразливим до стану гонитви (race conditions) під час одночасної роботи кількох завдань. Замість цього спроєктовано повноцінну фінансову книгу на основі таблиці coin_transactions. Кожне нарахування або списання фіксується як окрема незмінна транзакція (Append-Only Log), що гарантує строгу ACID-цілісність і дозволяє проводити аудит ігрової економіки. Поточний баланс користувача вираховується динамічно або кешується агрегатором.

2.5.2 Система досягнень та подієва архітектура (Achievements)

Для стимулювання регулярного навчання реалізовано систему беджів і досягнень. Досягнення класифікуються за типами:

- **Кількісні:** «Вирішити 50 завдань на Python», «Заробити 1000 монет».
- **Якісні:** «Написати код без синтаксичних помилок з першої спроби».
- **Часові (Streaks):** «Заходити на платформу 7 днів поспіль».

Архітектурно, перевірка досягнень реалізована через патерн **Observer (Спостерігач)** та подієву модель. Після успішного завершення завдання або логіну доменне ядро генерує внутрішню подію (наприклад, TaskCompletedEvent). Спеціалізовані обробники (Handlers) асинхронно перевіряють умови розблокування досягнень і, у разі успіху, створюють запис у таблиці user_achievements, ініціюючи SignalR-сповіщення на клієнт (Frontend) для відображення святкової анімації.

2.5.3 Віртуальний магазин, інвентар та екіпірування

Зароблені монети студенти витрачають у віртуальному магазині Roadly+. Асортимент включає цифрові товари для персоналізації профілю: унікальні рамки аватарів, фонові зображення (бекграунди) та спеціальні ігрові статуси.

- **Збереження покупок:** Придбані товари фіксуються в таблиці user_inventory_items. Цей підхід дозволяє користувачу мати колекцію речей, з якої він може обирати активні предмети.
- **Система екіпірування:** Для визначення того, які саме предмети наразі «одягнені» на профіль, спроєктовано сутність user_equipment_loadouts. Це забезпечує гнучкість, дозволяючи зберігати різні пресети (набори) зовнішнього вигляду.

2.5.4 RPG-візуалізація навчального маршруту

Найскладнішим елементом на рівні клієнтського інтерфейсу (React UI) є заміна традиційного вертикального списку уроків на інтерактивну карту (Roadmap Visualization). Маршрут, згенерований штучним інтелектом, рендериться у вигляді ігрових «островів» або вузлів на двовимірній площині. Аватар користувача переміщується за допомогою цієї картки під час проходження чекпойнтів. Цей візуальний підхід суттєво знижує когнітивне навантаження: користувач сприймає навчання не як академічний обов'язок, а як проходження рівнів у відеогрі.

2.6 Проєктування інтеграційного модуля для середовища Visual Studio Code

Важливою стратегічною складовою платформи «Roadly» є її інтеграція у звичний робочий простір розробника. Попри наявність потужної хмарної Web IDE, більшість програмістів віддає перевагу власним локальним середовищам, які вже мають індивідуальні налаштування тем, гарячих клавіш, лінтерів і плагінів. Створення розширення для **Visual Studio Code (VS Code)** дозволяє студентам залишатися продуктивними, не витрачаючи час на перемикання між інструментами.

2.6.1 Архітектурне обґрунтування та економія ресурсів

Рішення про розробку десктопного розширення базується на двох ключових факторах:

1. **Професійний UX:** досвідчені користувачі працюють ефективніше у власному середовищі. Надаючи доступ до Roadly безпосередньо з VS Code, ми спрощуємо шлях від теорії до практики, дозволяючи використовувати локальну потужність комп'ютера для компіляції та запуску складних проєктів.
2. **Оптимізація інфраструктури (Compute Offloading):** Використання локальної IDE суттєво знижує навантаження на серверний кластер платформи.

Кожен активний користувач у Web IDE потребує виділеного Docker-контейнера, який споживає RAM і CPU сервера. Перенесення процесу написання та запуску коду на бік клієнта дозволяє системі масштабуватися до значно більшої кількості користувачів без пропорційного збільшення витрат на хмарні ресурси.

2.6.2 Безпека та синхронізація через VS Code API

Розширення розроблено на базі **TypeScript** із використанням офіційного **VS Code Extension API**. Для забезпечення безшовної взаємодії реалізовано:

- **Автентифікація через OAuth2 + PKCE:** Користувач авторизується через Cognito Hosted UI у системному браузері, після чого розширення перехоплює токен через протокол vscode://.
- **Secret Storage:** Токени доступу зберігаються у захищеному системному сховищі (Keychain/Credential Manager), що гарантує безпеку сесії.
- **TreeView Integration:** Структура курсу та статус чекпойнтів відображаються у боковій панелі (Activity Bar), забезпечуючи миттєвий доступ до навчальних матеріалів без відриву від редактора.

2.6.3 Перспективні модулі та інтеграція з AI-асистентом

Наразі триває активна робота над розширенням функціональних можливостей модуля для глибшої інтеграції з кодовою базою.

Такий підхід робить Roadly не просто веб-сайтом з курсами, а частиною професійної екосистеми розробника, яка росте разом із навичками користувача — від початківця у Web IDE до профі у власному налаштованому середовищі.

Висновки до розділу 2

У другому розділі спроєктовано мікросервісну архітектуру освітньої платформи «Roadly». Для забезпечення високої масштабованості систему декомпозитовано на 11 ізольованих контейнерів. Обґрунтовано вибір технологічного стеку: фреймворку .NET 9 для реалізації транзакційного ядра (CQRS), мови Python для інтелектуальних сервісів та мови Go для високопродуктивного шлюзу контейнеризації.

Впроваджено асинхронну модель взаємодії на базі брокера RabbitMQ, що гарантує низьку зв’язність компонентів системи. Спроєктовано гібридну підсистему збереження даних, де основна бізнес-логіка базується на нормалізованій реляційній моделі PostgreSQL, семантичний пошук — на векторній базі pgvector, а швидка обробка сесій — на Redis. Розроблена топологія та обрані протоколи взаємодії (REST, WS, AMQP) повністю відповідають вимогам до продуктивності й стабільності сучасних EdTech-систем.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

3.1 Реалізація доменного ядра на .NET 9 та імплементація патернів Clean Architecture

Основним транзакційним вузлом (доменним ядром) платформи «Roadly» є мікросервіс, реалізований на платформі **.NET 9** (мова C# 13). Для забезпечення високої тестованості, незалежності від фреймворків та легкості масштабування кодової бази, програмну архітектуру проєкту побудовано за методологією **Clean Architecture** (Чиста архітектура).

Рішення (Solution) фізично розділене на чотири ізольовані проєкти (шари) із суворим правилом напрямку залежностей — від зовнішніх шарів до внутрішнього:

1. **Domain:** містить бізнес-сутності (Entities), перерахування (Enums) та доменні винятки. Немає жодних зовнішніх залежностей.
2. **Application:** Містить бізнес-логіку програми (Use Cases), DTO-моделі та абстракції (інтерфейси) для інфраструктури.
3. **Infrastructure:** реалізує інтерфейси шару Application. Містить підключення до PostgreSQL (через Entity Framework Core), інтеграцію з AWS Cognito, RabbitMQ та Docker SDK.
4. **Api:** точка входу (Presentation Layer). Містить REST-контролери, Middleware (наприклад, CorrelationIdMiddleware та InternalServiceKeyMiddleware) і налаштування конфігурації.

3.1.1 Імплементація CQRS та конвеєра MediatR

Для розв'язання проблеми «товстих контролерів» (Fat Controllers) та розділення логіки читання і запису в шарі Application імплементовано патерн **CQRS** (Command and Query Responsibility Segregation). Усі операції в системі розділені на Команди (змінюють стан системи) та Запити (лише повертають дані).

Реалізація патерну виконана за допомогою бібліотеки **MediatR**. У Додатку А наведено приклад типової команди оновлення прогресу чекпойнту та її обробника (Handler) в додатку А.Використання MediatR дозволило впровадити патерн **Pipeline Behaviors** (поведінки конвеєра). Завдяки цьому валідація вхідних даних (через FluentValidation) та логування (LoggingBehavior) виконуються автоматично *до* того, як запит потрапить у Handler. Це повністю очищує бізнес-логіку від рутинних перевірок if (!ModelState.IsValid).

3.1.2 Функціональний підхід до обробки помилок: Патерн Result<T, E>

Традиційна обробка помилок у C# базується на використанні винятків (throw new Exception()). Проте викидання винятків для очікуваних бізнес-помилок (наприклад, "Користувача не знайдено" або "Недостатньо монет на балансі") є антипатерном (Exception as Control Flow). Це суттєво знижує продуктивність додатка через необхідність розгортання стеку викликів (Stack Unwinding) операційною системою.

Для вирішення цієї інженерної проблеми в ядро платформи «Roadly» було інтегровано **патерн Result**, запозичений із концепцій функціональних мов програмування (таких як F# чи Rust).

Суть підходу полягає у тому, що методи не викидають винятки, а повертають спеціальний об'єкт-обгортку Result<T, E>, який явно вказує на успішність операції або містить типізовану помилку. На рівні REST-контролерів (у шарі Api) цей підхід дозволяє писати передбачуваний і безпечний код. Контролер просто перевіряє стан об'єкта Result і повертає відповідний HTTP-статус.

Завдяки патерну Result сигнатури методів стають чесними: розробник одразу бачить, що метод може завершитися невдачею, і компілятор змушує його обробити

обидва сценарії. Глобальні Exceptions використовуються виключно для непередбачуваних ситуацій (наприклад, падіння з'єднання з базою даних).

3.1.3 Принцип інверсії залежностей (Dependency Inversion)

Відповідно до букви 'D' у принципах SOLID ядро не залежить від інфраструктури. Наприклад, логіка виклику Docker-контейнерів або середовища Piston абстрагована через інтерфейс ICodeExecutor, визначений у шарі Application. Сама реалізація (HTTP-запити до Go-проксі) знаходиться у шарі Infrastructure. Це дозволяє покривати бізнес-логіку Unit-тестами (через xUnit та Moq) без необхідності піднімати реальні контейнери під час тестування.

3.2 Розробка програмних модулів, опис основних класів та методів

Програмний комплекс інформаційної системи реалізовано з використанням мікросервісної архітектури, що забезпечує високий рівень модульності, незалежність розгортання окремих компонентів і стійкість системи до навантажень. Кожен мікросервіс відповідає за окрему предметну область (Bounded Context) та взаємодіє з іншими через REST API або асинхронні повідомлення.

Загалом систему поділено на чотири ключові модулі: модуль керування навчальними планами (Roadmap Service), модуль штучного інтелекту (AI Orchestrator), модуль виконання коду (Code Runner Service) та клієнтський вебзастосунок (Client Application).

3.2.1 Модуль керування навчальними планами (Roadmap Service)

Бекенд-сервіс, відповідальний за збереження та обробку навчальних маршрутів, розроблено на платформі .NET з використанням мови C#. В основу проєктування покладено принципи чистої архітектури (Clean Architecture), що

дозволяють ізолювати бізнес-логіку від інфраструктурних залежностей, таких як бази даних або зовнішні API.

Для оптимізації операцій читання та запису застосовано патерн CQRS (Command Query Responsibility Segregation).

Основні класи та інтерфейси модуля:

- Roadmap — базова сутність (Domain Entity), що описує навчальний план. Вона містить ідентифікатор, метадані курсу та ієрархічну структуру тем.
- IRoadmapRepository — інтерфейс, що визначає контракт для взаємодії з базою даних. Забезпечує абстракцію доступу до даних, що є критично важливим для модульного тестування (Unit testing).
- CreateRoadmapCommandHandler — клас, що імплементує інтерфейс IRequestHandler<CreateRoadmapCommand, RoadmapDto>. Його основний метод Handle(CreateRoadmapCommand request, CancellationToken cancellationToken) інкапсулює логіку валідації вхідних даних, звернення до AI-модуля для генерації структури та збереження результату в базу даних.
- GetRoadmapByIdQueryHandler — оптимізований клас для вибірки даних, який повертає готову DTO (Data Transfer Object) модель для фронтенду без зайвих накладних витрат на трекінг змін (AsNoTracking).

Особливості роботи з даними: Зважаючи на те, що кожен навчальний маршрут має унікальну графову структуру у вигляді дерева тем, використання класичних реляційних таблиць для збереження кожного вузла призвело б до надмірної кількості операцій JOIN і зниження продуктивності. Тому було прийнято рішення зберігати повну карту маршруту як єдиний об'єкт у полі типу JSONB у базі даних PostgreSQL. Для мапінгу цього поля на об'єктно-орієнтовану модель C# використовується вбудована підтримка серіалізації System.Text.Json та відповідні конфігурації Fluent API у Entity Framework Core.

3.2.2 Модуль генерації контенту (AI Orchestrator)

Модуль штучного інтелекту розроблено мовою Python з використанням фреймворку FastAPI, який ідеально підходить для створення високопродуктивних асинхронних мікросервісів, орієнтованих на роботу з даними та машинні навчання.

Логіка генерації навчального контенту побудована на основі архітектури RAG (Retrieval-Augmented Generation).

Ключові класи модуля:

- VectorStoreClient — клас, що відповідає за з'єднання з векторною базою даних. Основний метод semantic_search(query_embedding: list[float], top_k: int) виконує пошук найближчих за змістом фрагментів освітніх матеріалів, використовуючи косинусну подібність (cosine similarity).
- PromptBuilder — утилітарний клас для формування контекстно-залежних запитів (промптів). Метод build_roadmap_prompt(topic: str, context: list[str]) динамічно компонує знайдений контекст із системними інструкціями для мовної моделі.
- LLMService — сервісний клас для взаємодії з API великої мовної моделі. Містить асинхронний метод async def generate_structure(prompt: str) -> dict, який надсилає запит, отримує текстову відповідь, виконує її парсинг і валідацію на відповідність очікуваній JSON-схемі.

3.2.3 Модуль ізольованого виконання коду (Code Runner Service)

Для реалізації функціоналу Web IDE, що дозволяє користувачам безпечно запускати написаний код безпосередньо у браузері, розроблено окремий сервіс мовою Go. Вибір Go обґрунтований низьким споживанням пам'яті, високою

швидкістю паралельної обробки та наявною екосистемою для роботи з контейнерами (Docker Engine API).

Опис класів та структур даних:

- CodeExecutionRequest — структура (struct), що містить вихідний код, мову програмування та ідентифікатор користувача.
- ExecutionResult — структура, що інкапсулює результати роботи програми: стандартний вивід (stdout), вивід помилок (stderr), час виконання та статус виходу (exit code).
- ContainerManager — основний компонент, який керує життєвим циклом Docker-контейнерів.

Безпека та ізоляція: Метод ExecuteCode(ctx context.Context, req *CodeExecutionRequest) (*ExecutionResult, error) є критичним з точки зору безпеки. Перед запуском коду він створює тимчасовий контейнер із жорсткими обмеженнями (cgroups):

1. Обмеження оперативної пам'яті (наприклад, 128 MB).
2. Обмеження процесорного часу.
3. Повна відсутність доступу до мережі (network disabled), що унеможливлює здійснення DDoS-атак або завантаження шкідливого програмного забезпечення зсередини контейнера.
4. Встановлення жорсткого таймауту на виконання (наприклад, 5 секунд) через механізм контекстів (context.WithTimeout) мови Go для запобігання зацикленням.

Для інтерактивних завдань, де користувач очікує поступового виведення результатів, реалізовано клас WebSocketHandler, який встановлює постійне з'єднання

з клієнтом та транслює потік логів (log stream) безпосередньо з контейнера у браузер.

3.2.4 Інтерфейсна частина (Frontend)

Клієнтський застосунок реалізовано за допомогою бібліотеки React і мови TypeScript. Архітектура фронтенду базується на компонентному підході.

Основні компоненти та управління станом:

- Для глобального управління станом використовується Redux Toolkit[14]. Клас roadmapSlice зберігає поточний стан дерева навчання, прогрес проходження тем і кешує дані, щоб мінімізувати повторні запити до API.
- Компонент RoadmapCanvas відповідає за візуалізацію графа навчання. Для його реалізації інтегровано бібліотеку React Flow, яка забезпечує рендеринг вузлів (Custom Nodes) та зв'язків між ними, а також обробляє події панорамування й масштабування карти.
- Компонент WebIDE інтегрує редактор коду (наприклад, Monaco Editor). Він використовує кастомний React Hook useCodeExecution(), який абстрагує логіку відправки коду на виконання, обробки відповідей від Go-мікросервісу та виведення результатів або повідомлень про синтаксичні помилки у вбудований термінал застосунку.

3.3 Розробка шлюзу контейнеризації на Go (WebSocket PTY)

Для забезпечення повноцінної роботи користувача в інтерактивному терміналі (Web IDE) необхідно встановити постійний двосторонній канал зв'язку між браузером та ізольованим Docker-контейнером. Оскільки веббраузери не можуть безпосередньо звертатися до TCP-сокетів Docker Daemon через обмеження безпеки, було розроблено проміжний мікросервіс (шлюз) мовою **Go (Golang) [15]**.

Вибір мови Go для програмної реалізації шлюзу зумовлений її нативною моделлю конкурентності (Goroutines) та наявністю офіційного пакета docker/docker/client, що дозволяє взаємодіяти з API контейнеризації на низькому рівні.

3.3.1 Реалізація механізму HTTP-Upgrade та встановлення WebSocket з'єднання

Комунікація термінала ініціюється як звичайний HTTP-запит, проте для передачі потокових даних у реальному часі його необхідно трансформувати у WebSocket-з'єднання. Для цієї задачі у контролері використано бібліотеку gorilla/websocket.

Алгоритм підключення виглядає наступним чином: шлюз отримує запит, перевіряє JWT-токен користувача (переданий через протокол або параметри), валідує права доступу до конкретного контейнера (Ownership Check) і виконує процедуру **Protocol Upgrade** (Додаток Б).

Такий підхід дозволяє тримати відкритим постійний канал зв'язку з мінімальними накладними витратами (overhead) — одне WebSocket-з'єднання в Go споживає лише кілька кілобайт оперативної пам'яті.

3.3.2 Інтеграція з Docker SDK та мультиплексування потоків (Goroutines)

Ключовим завданням шлюзу є проксування байтів: те, що користувач вводить на клавіатурі (Stdin), має потрапити в контейнер, а вивід термінала (Stdout/Stderr) — повернутися в браузер. Для підключення до працюючого контейнера програмно викликається метод ContainerAttach з Docker SDK, який повертає об'єкт HijackedResponse. Цей об'єкт надає прямий доступ до потоків введення/виведення процесу.

Найбільшим інженерним викликом у цій задачі є те, що читання та запис мають відбуватися одночасно і не блокувати одне одного. Завдяки конкурентній моделі Go це вирішується шляхом запуску двох легковажних паралельних потоків — **goroutines** (Додаток В).

Крім базового введення/виведення, шлюз програмно обробляє спеціальні контрольні послідовності. Наприклад, коли користувач змінює розмір вікна браузера, клієнт надсилає WebSocket-повідомлення зі спеціальним payload (кількість рядків і стовпців). Шлюз перехоплює його та викликає метод ContainerResize, який сигналізує ядру Linux усередині контейнера про зміну геометрії PTY. Це гарантує коректне відображення консольних інтерфейсів (наприклад, редакторів nano чи vim).

Завдяки реалізації цієї архітектури на Go шлюз здатен ефективно утримувати тисячі одночасних «сплячих» термінальних сесій без деградації продуктивності, що було б неможливо за традиційної моделі обробки потоків у C#.

3.4 Архітектура управління станом клієнтського застосунку та взаємодія з API

Клієнтський вебзастосунок платформи «Roadly» побудований за принципом **Feature-Based Architecture**, де кожен великий функціональний блок (навчальні карти, Web IDE, ігровий простір) має власну логіку керування даними. Оскільки фронтенд одночасно взаємодіє з трьома гетерогенними бекенд-сервісами (.NET, Python, Go), було впроваджено гібридну модель управління станом, що розділяє стабільні серверні дані та високочастотні локальні події.

3.4.1 Централізоване управління серверними даними через RTK Query

Для роботи з основним .NET API було обрано **RTK Query** (частина екосистеми Redux Toolkit). Це рішення дозволило перенести логіку мережевих запитів з компонентів у декларативний шар опису сервісів.

Програмна реалізація базується на використанні «тегів» (Tags) для автоматичної інвалідації кешу. Наприклад, під час виконання команди покупки в магазині Roadly+ система автоматично маркує тег UserInventory як неактуальний. Це ініціює фонове оновлення даних лише в тих частинах інтерфейсу, де відображається інвентар користувача, не зачіпаючи стан навчальної карти. Такий підхід суттєво зменшує кількість надлишкових запитів до бази даних PostgreSQL та покращує UX, забезпечуючи «миттєву» реакцію інтерфейсу на дії користувача.

Крім того, RTK Query забезпечує механізм **оптимістичних оновлень**. Під час проходження чекпойнта UI відмічає його як завершений ще до отримання відповіді від сервера. Якщо .NET API поверне помилку (наприклад, через втрату з'єднання), Redux автоматично відкотить стан до попереднього стану, гарантуючи цілісність відображуваної інформації.

3.4.2 Реалізація реактивного локального стану за допомогою Zustand

Незважаючи на потужність Redux, його архітектура (Actions, Reducers, Dispatch) створює значне навантаження під час обробки високочастотних оновлень. У модулях Web IDE та AI Mentor, де дані оновлюються кілька разів на секунду (наприклад, посимвольна генерація тексту або логування термінала), використано бібліотеку **Zustand**.

Zustand працює за межами React Context, що дозволяє компонентам підписуватися на конкретні «зрізи» стану. Це програмне рішення вирішує проблему масових перерендерів (re-renders): коли ШІ-ментор через SSE-потік додає нове слово до чату, оновлюється лише вікно повідомлень, а не весь контейнер сторінки. Сховища Zustand (Stores) використовуються для:

- **AI Streaming Store:** накопичення токенів із Python-сервісу та стан індикатора "друкування".
- **Terminal & FS Store:** Керування метаданими сесії Docker та станом відкритої папки у Web IDE.

3.4.3 Опрацювання асинхронних потоків даних (SignalR, SSE)

Для підтримки зв'язку в реальному часі на рівні коду реалізовано два незалежні конвеєри споживачів (Consumers):

1. **SignalR Service:** реалізований як окремий React-провайдер, який встановлює з'єднання з .NET Hub під час завантаження профілю. Він слухає події завершення генерації Roadmap. Це критично, оскільки створення карти через LLM може тривати до 30 секунд, і користувач не повинен бути заблокований вікном очікування.
2. **Streaming Consumer:** Для роботи з ШІ використано пакет @microsoft/fetch-event-source. Програмна логіка дозволяє перетворити потік байтів від FastAPI на масив об'єктів повідомлень, які валідуються на відповідність схемам TypeScript. Це забезпечує безпеку типів навіть для неструктурованого контенту, який генерується LLM.

3.5 Програмна реалізація дизайн-системи та модулів вебдоступності (A11y)

Проєктування візуального інтерфейсу платформи «Roadly» базується на принципах інклюзивного дизайну, що передбачає створення однакового рівня комфорту для всіх категорій користувачів. Програмна реалізація інтерфейсу спирається на чотири технологічні фундаменти: **Tailwind [8], CSS 4**, **Radix UI** та архітектуру **Accessibility Context**.

3.5.1 Архітектура компонентів на базі Radix UI та Tailwind CSS 4

Для побудови складних інтерактивних вузлів (модальні вікна, випадаючі списки, складні перемикачі) було обрано бібліотеку **Radix UI**. Вибір зумовлений тим, що ці компоненти постачаються як «примітиви» без вбудованих стилів, але з повною програмною підтримкою стандартів **WAI-ARIA**. Це дозволило розробнику сфокусуватися на бізнес-логіці та стилізації, не витрачаючи ресурси на ручне програмування логіки фокусу чи атрибутів aria-expanded та aria-controls.

Стилізація реалізована за допомогою **Tailwind CSS 4**. У новій версії фреймворку було впроваджено движок на основі CSS-змінних, що дозволяє динамічно змінювати кольорові схеми та розмірні сітки на рівні всього застосунку. Використання простору кольорів **OKLCH** замість RGB/HEX забезпечує математично точне збереження контрастності під час зміни відтінків, що є критично важливим для створення темної теми та режимів високого контрасту.

3.5.2 Програмна імплементація панелі вебдоступності (A11y Panel)

Центральним інструментом інклюзивності є панель налаштувань. Її логіка реалізована через кастомний React-хук useA11y, який взаємодіє з глобальним контекстом. При зміні будь-якого параметра (наприклад, включення шрифту для людей з дислексією) відбувається наступний програмний цикл:

1. **State Update:** Об’єкт конфігурації в React Context оновлюється новим значенням.
2. **DOM Injection:** useEffect відстежує зміни та оновлює атрибути тегу <body> або ін'єктує нові значення у кореневий об'єкт :root.
3. **Reflow:** браузер перераховує стилі на основі оновлених CSS-змінних без перезавантаження сторінки.

3.5.3 Реалізація спеціалізованих профілів та режимів

На програмному рівні імплементовано чотири основні режими адаптації:

- **Профіль для користувачів із дислексією:** Платформа підміняє стандартний шрифт на **OpenDyslexic**. Окрім заміни гарнітури, програмно збільшуються параметри letter-spacing і line-height. Це реалізовано через CSS-клас .dyslexia-mode, який перевизначає базові змінні Tailwind.
- **Режим запобігання епілептичним нападам (Motion Reduction):** Програмна логіка автоматично зчитує системне налаштування prefers-reduced-motion. Якщо воно активоване або користувач вимкнув анімації в панелі, система примусово встановлює duration: 0ms для всіх переходів. Це критично для безпеки під час відображення динамічних елементів у Roadly+ та AI-чаті.
- **Масштабування (Visual Scaling):** На відміну від звичайного зуму браузера, який може ламати лейаут, Roadly використовує зміну базового значення 1rem. Оскільки всі відступи та розміри в Tailwind базуються на одиницях rem, зміна одного параметра на рівні html { font-size: ... } пропорційно збільшує весь інтерфейс, зберігаючи ієрархію елементів.
- **Кольорова адаптація (Color Correction):** Платформа підтримує режими для людей із різними типами дальтонізму (протанопія, дейтеранопія). Це реалізовано шляхом накладання SVG-фільтрів на кореневий елемент, що дозволяє коригувати кольори всього контенту, зокрема графіків прогресу та іконок.

3.5.4 Семантична розмітка та навігація з клавіатури

Програмна реалізація Web IDE та навчальних маршрутів вимагала особливої уваги до навігації без використання миші.

1. **Skip Links:** Реалізовано механізм прихованих посилань «Перейти до контенту», які з'являються під час першого натискання Tab. Це дозволяє користувачам зі скрінрідерами швидко переходити до тексту уроку, не затримуючись у навігаційному меню.
2. **Focus Management:** Для редактора коду Monaco Editor та термінала xterm.js[17] розроблено логіку перехоплення фокусу. Коли користувач працює в терміналі, натискання Esc повертає фокус у дерево файлів, що забезпечує логічний потік навігації.
3. **Semantic HTML:** Усі ігрові елементи Roadly+ (острови, монети, магазин) розмічені як об'єкти з відповідними ролями (role="button", role="grid", aria-label). Це робить гейміфікований досвід доступним для людей із повним порушенням зору.

3.5.5 Персистентність та хмарна синхронізація

Для забезпечення стабільного UX налаштування доступності зберігаються двома способами:

- **Миттєве збереження:** Усі зміни дублюються в localStorage. Це гарантує, що під час повторного відкриття вкладки користувач не отримає «світлового удару» від стандартної білої теми чи яскравих анімацій.
- **Профільна синхронізація:** Після зміни налаштувань фронтенд надсилає асинхронний запит до .NET API (UpdateUserPreferencesCommand). Це дозволяє користувачу мати однакові параметри доступності як у вебверсії, так і в розширенні VS Code, що є унікальною перевагою платформи.

Висновки до розділу 3

У третьому розділі було детально описано процес програмної реалізації ключових компонентів інтелектуальної платформи «Roadly». Основну увагу приділено практичному втіленню мікросервісної архітектури та забезпеченню безперебійної взаємодії між різними технологічними стеками системи.

Розроблено сучасний клієнтський інтерфейс на базі бібліотеки React 19 та Tailwind CSS. Для забезпечення високої продуктивності та зручного керування даними впроваджено гібридну систему керування станом: використано RTK Query для ефективного кешування серверних відповідей та Zustand для керування легковажним локальним станом застосунку. Важливим інженерним досягненням стала реалізація спеціалізованої панелі вебдоступності (A11y Panel), яка завдяки динамічному керуванню CSS-змінними дозволяє користувачам миттєво адаптувати інтерфейс під власні потреби відповідно до міжнародних стандартів WCAG 2.1.

Реалізовано інтелектуальний бекенд на Python (FastAPI), який інтегрується з великими мовними моделями через екосистему LangChain[10]. Програмно забезпечено потокову передачу відповідей ШІ-ментора за допомогою технології Server-Sent Events (SSE), що створює ефект реального часу та суттєво покращує користувацький досвід. Ключовим технічним рішенням став розроблений мовою Go високопродуктивний шлюз, який через протокол WebSockets організовує безпечні PTY-сесії з Docker-контейнерами. Це дозволило реалізувати повноцінне Web IDE з підтримкою термінала (xterm.js).

Додатково створено модуль інтеграції з Visual Studio Code, що забезпечує студентам омніканальний доступ до навчальних планів безпосередньо у професійному середовищі розробки. Таким чином, програмна реалізація підтвердила життєздатність обраних архітектурних рішень і забезпечила повну відповідність системи заявленим функціональним вимогам.

РОЗДІЛ 4 ІНФРАСТРУКТУРА, ТЕСТУВАННЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ

4.1 Контейнеризація та оркестрація інфраструктури платформи

Сучасна архітектура освітньої платформи «Roadly», що складається з 11 взаємопов’язаних мікросервісів і фонових процесів, вимагає надійної системи розгортання, масштабування та ізоляції. Для забезпечення консистентності (ідентичності) середовищ розробки (Development), тестування (Staging) та експлуатації (Production) було обрано технологію **контейнеризації на базі Docker**. Управління життєвим циклом усіх сервісів здійснюється за допомогою декларативного інструменту оркестрації **Docker Compose**.

4.1.1 Топологія контейнеризованих сервісів

Інфраструктура платформи спроєктована за принципом функціональних контурів. Конфігураційний файл docker-compose.yml визначає три логічні групи сервісів, кожна з яких має специфічні вимоги до ресурсів та безпеки:

1. **Application Circuit (Контур застосунків):**
    - dotnet-api: Ядро системи, що обробляє бізнес-транзакції. Налаштовано з використанням легкоочікуваного базового образу (Alpine Linux), щоб зменшити розмір контейнера.
    - python-ai-service: FastAPI-сервер, який відповідає за інтенсивні I/O-операції (взаємодія з LLM).
    - python-worker: Фоновий процес Celery, який потребує високих лімітів CPU для паралельної генерації контенту.
    - go-pty-proxy: Легковажний шлюз мовою Go для керування WebSocket-сесіями користувачів.

- react-ui: Фронтенд-застосунок, який збирається (Build) через Vite та подається за допомогою високопродуктивного статичного вебсервера Nginx.

2. **Data & Messaging Circuit (Контур персистентності та повідомлень):**
   - postgres-db: Основне сховище (Relational Database) із підключеним розширенням pgvector для RAG-пошуку.
   - rabbitmq: Брокер повідомлень, налаштований для забезпечення гарантованої доставки подій (Message Durability) між .NET і Python.
   - Redis: In-memory сховище, що виконує подвійну роль: кешування для .NET та збереження стану задач (Results Backend) для Celery.
3. **Execution Circuit (Контур динамічного виконання):**
   - Ефемерні (тимчасові) контейнери користувачів. Їхньою особливістю є те, що вони не прописані жорстко в docker-compose, а генеруються "на льоту" (On-the-fly) через виклики Docker SDK з боку .NET API.

4.1.2 Мережева ізоляція та управління доступом

З метою мінімізації площі атаки (Attack Surface) у конфігурації Docker Compose спроєктовано ізольовану віртуальну мережу (Bridge Network). Застосовано патерн **Zero Trust Network**:

- Бази даних (PostgreSQL, Redis) та брокер повідомлень (RabbitMQ) **не мають** прокинутих (Published) портів у зовнішню мережу хоста. Вони доступні виключно всередині внутрішньої мережі за внутрішніми DNS-іменами (наприклад, db:5432).
- Доступ ззовні дозволено лише до портів API (8080, 8000), PTY-проксі (8081) та фронтенду (80).

Для автентифікації міжсервісних запитів (наприклад, коли Python-сервіс звертається до .NET для запису прогресу), замість передачі клієнтських JWT-токенів

впроваджено механізм **Internal Service Keys**. Секретний ключ передається як змінна середовища (ENVIRONMENT VARIABLE) лише до довірених контейнерів і перевіряється на рівні Middleware.

4.1.3 Управління конфігураціями, секретами та персистентністю

Збереження стану (Persistence) є критичним аспектом для баз даних у контейнерному середовищі. У проєкті використано іменовані томи (**Named Volumes**):

- postgres_data: гарантує, що дані користувачів і вектори не зникнуть після виконання команди docker-compose down.
- redis_data та rabbitmq_data: зберігають черги та сесії.

Управління секретами (API-ключі OpenAI/Gemini, реквізити AWS Cognito, паролі адміністраторів баз даних) реалізовано через зовнішні файли .env. Це відповідає принципам методології **Twelve-Factor App**, дозволяючи повністю ізолювати чутливі дані від кодової бази та безпечно змінювати їх залежно від середовища (Dev/Prod).

4.1.4 Проблема "сміття" та автоматизоване очищення (Garbage Collection)

Оскільки платформа Roadly постійно створює нові Docker-контейнери для компіляції студентського коду, виникає проблема вичерпання ресурсів (звільнення пам'яті та дискового простору). Для вирішення цієї інженерної проблеми у .NET-ядрі реалізовано фоновий процес на базі бібліотеки **Hangfire[9]** (ContainerCleanupJob). Цей процес періодично звертається до Docker SDK, знаходить контейнери зі статусом Exited або Dead, а також ті, час життя яких (TTL) перевищив дозволений ліміт, і примусово видаляє їх разом із пов'язаними томами (Volumes). Це гарантує стабільну роботу інфраструктури в режимі 24/7.

4.2 Стратегія тестування, забезпечення надійності та аудит якості

Враховуючи складність мікросервісної архітектури платформи «Roadly», де взаємодіють гетерогенні технології (.NET, Python, Go, React), забезпечення безперебійної роботи вимагає комплексного підходу до тестування. Стратегія гарантування якості (Quality Assurance) була розділена на три фундаментальні рівні: ізольоване модульне тестування (Unit Testing), перевірка міжсервісної взаємодії (Integration Testing) та аудит користувацьких інтерфейсів (A11y/UI Testing).

4.2.1 Модульне та інтеграційне тестування доменного ядра (.NET)

Для бекенд-частини платформи, яка відповідає за фінансові транзакції (Roadly Coins) та облік ігрового прогресу, критично важливим є недопущення регресійних помилок. Програмна реалізація тестів побудована на базі фреймворку **xUnit** у поєднанні з бібліотеками **Moq** (для створення об'єктів-заглушок) та **FluentAssertions** (для перевірки валідаційних правил).

У межах методології Clean Architecture модульні тести сфокусовані на перевірці бізнес-правил у шарі Application. Особлива увага приділялася тестуванню патерну CQRS. Наприклад, для перевірки UpdateItemCompletionCommandHandler були написані тести, які гарантують, що:

1. Успішне виконання повертає Result.Success.
2. Виклик з невалідним CheckpointId повертає типізовану помилку домену (DomainErrors.Progress.NotFound), а не викидає глобальний Exception.
3. Після оновлення прогресу в БД хендлер обов'язково викликає метод PublishAsync інтерфейсу IMessageProducer для публікації події в RabbitMQ.

Для інтеграційного тестування REST-контролерів (шар API) використано підхід з **TestServer** та In-Memory базою даних (на базі SQLite). Це дозволяє імітувати повний життєвий цикл HTTP-запиту: від проходження через CorrelationIdMiddleware до збереження даних у тестовій БД, без необхідності піднімати реальний екземпляр PostgreSQL.

4.2.2 Тестування інтелектуального модуля та RAG-конвеєрів (Python)

Специфіка AI-компонентів полягає у тому, що великі мовні моделі (LLM) генерують недетермінований (непередбачуваний) результат. Тому класичне порівняння "очікуваний рядок == фактичний рядок" тут не працює. Тестування Python-сервісу реалізовано за допомогою фреймворку **Pytest**.

Для тестування гранулярної генерації навчальних маршрутів (Celery Tasks) та AI-чату впроваджено наступні інженерні практики:

- **LLM Mocking (Мокування моделей):** Усі звернення до зовнішніх API (OpenAI/Gemini) підміняються фіктивними класами, які повертають наперед задані JSON-структури. Це дозволяє тестувати логіку парсингу Pydantic-схем без витрачання реальних токенів і грошей.
- **Schema Validation (Валідація структур):** Замість перевірки змісту тексту тести перевіряють структуру. Коли ШІ генерує практичне завдання (CodingTask), тест підтверджує наявність обов'язкових полів: title, description, initial_code, test_cases, а також коректність їхніх типів даних.
- **Testing SSE Streams:** Для перевірки потокової передачі даних (Server-Sent Events) написано спеціальний асинхронний клієнт, який перехоплює байти та підтверджує, що генератор FastAPI віддає префікс data: і коректно завершує з'єднання повідомленням [DONE].

4.2.3 Тестування надійності контейнерного шлюзу (Go)

Сервіс docker-pty-proxy відповідає за безпеку. Головний фокус його тестування — перевірка механізмів авторизації та захисту від несанкціонованого доступу (IDOR). За допомогою вбудованого пакета testing у Go було реалізовано перевірку middleware. Тести симулюють WebSocket-з'єднання з різними типами JWT-токенів (прострочених, з неправильним підписом, без необхідних Claims). Критично важливим етапом стала програмна перевірка **Ownership Enforcement**: тест гарантує, що спроба користувача "А" підключитися до PTY-сесії контейнера, який належить користувачу "Б", миттєво відхиляється шлюзом.

4.2.4 Автоматизований аудит вебдоступності та фронтенду (A11y)

Окрім модульного тестування компонентів React за допомогою **Vitest** та **Testing Library** (де перевіряється коректність роботи сховищ Redux/Zustand[18]), особливий акцент було зроблено на інклюзивності платформи.

Аудит користувацького інтерфейсу проводився за допомогою інструментів **Google Lighthouse** та **Axe DevTools**. Програмна інтеграція Axe безпосередньо у процес розробки дозволила автоматично виявляти такі порушення стандарту WCAG 2.1:

- Відсутність атрибута aria-label для ігрових іконок у магазині Roadly+.
- Недостатній коефіцієнт контрастності (Contrast Ratio) між текстом і фоном у темній темі.
- Проблеми з керуванням фокусом (Focus Trapping) під час роботи у вікні Web IDE.

Результати аудиту підтвердили, що динамічна система CSS-змінних, реалізована в панелі вебдоступності (A11y Panel), успішно усуває ці недоліки,

підвищуючи загальну оцінку доступності платформи до 95-100 балів за шкалою Lighthouse.

4.3 Оцінка практичних результатів та інструкція користувача

Розроблена інфраструктура та програмні компоненти були успішно інтегровані в єдину екосистему, що дозволило отримати повнофункціональний Minimum Viable Product (MVP) платформи «Roadly». Практична оцінка результатів проводилася шляхом наскрізного (End-to-End) тестування користувацьких сценаріїв: від моменту реєстрації до написання коду та взаємодії зі штучним інтелектом.

4.3.1 Реєстрація, онбординг та генерація навчального маршруту

Взаємодія користувача з платформою починається з етапу автентифікації, який безпечно делеговано сервісу AWS Cognito. Після першого входу система ініціює процес онбордингу (Onboarding), під час якого студент вказує свій поточний рівень знань, цільову мову програмування (наприклад, C# або Python) та кар'єрні цілі.

Зібраний контекст програмно упаковується та надсилається до Python-сервісу через систему черг RabbitMQ. Наразі інтерфейс відображає анімацію очікування. Завдяки реалізованому патерну гранулярної генерації (Celery Task Fan-out) створення персоналізованого курсу займає лише 15–20 секунд, після чого SignalR-хаб надсилає push-сповіщення в браузер про готовність маршруту.

4.3.2 Навігація навчальними маршрутами та RPG-візуалізація

Згенерований маршрут (Roadmap) відображається не у вигляді стандартного статичного списку, а як інтерактивна двовимірна карта (Roadmap UI). Кожен вузол на карті (Checkpoint) має свій унікальний тип: Теорія, Квіз (тестування) або Практичне завдання (Coding Task).

Інтерфейс, побудований на базі React 19[13] та Tailwind CSS, забезпечує плавну навігацію. Коли студент успішно завершує завдання, спрацьовує механізм оптимістичного оновлення (Optimistic Update) у Redux Toolkit: вузол на карті миттєво змінює колір на "Виконано", а аватар користувача переміщується на наступну позицію, підкріплюючи відчуття ігрового прогресу.

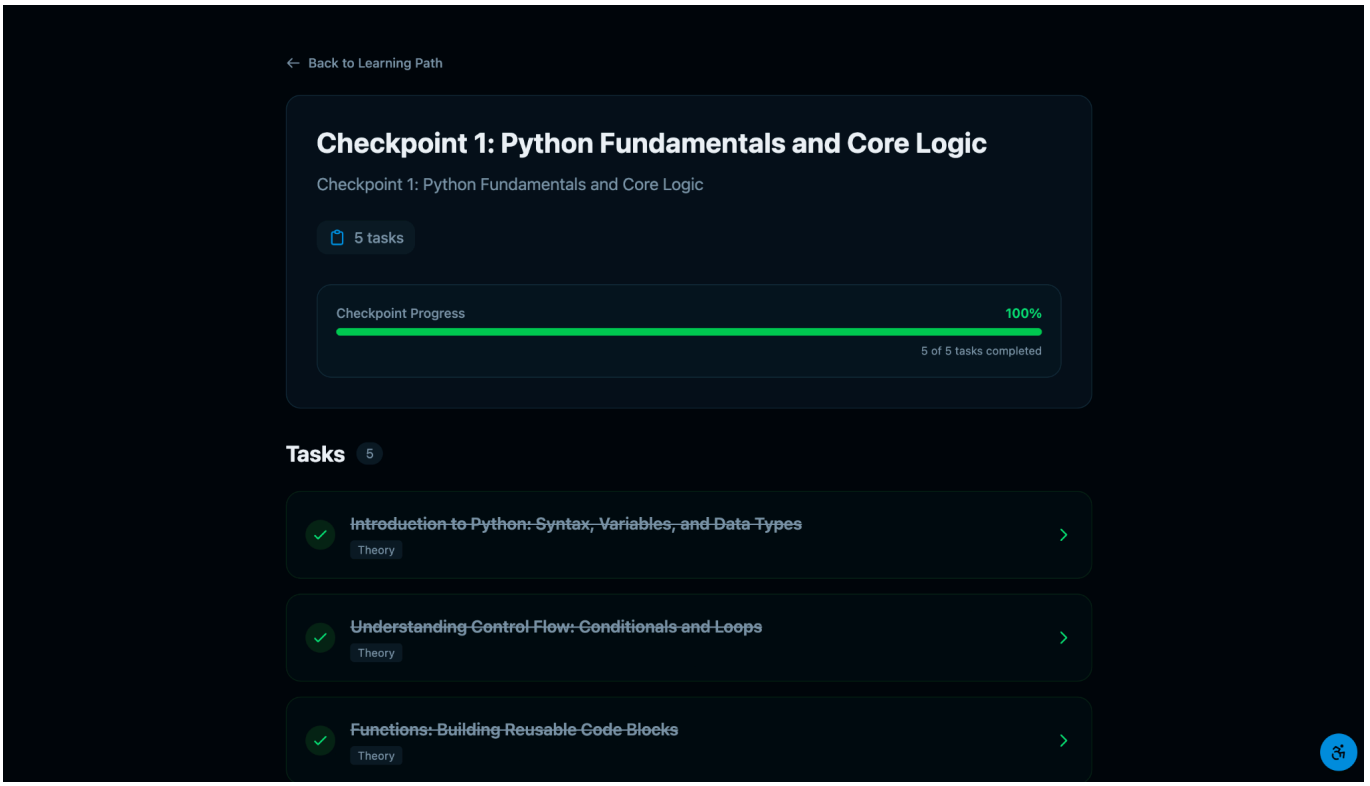

Рис. 4.2: Скріншот інтерактивної карти курсу (Roadmap) із вузлами завдань

Джерело: Розроблене автором

4.3.3 Робота в ізольованому середовищі Web IDE та взаємодія з AI-ментором

Найбільш технічно складним елементом інтерфейсу є сторінка виконання практичних завдань. Екран логічно розділений на три робочі зони:

1. **Редактор коду (Monaco Editor):** розташований у центральній частині, забезпечує підсвічування синтаксису, автодоповнення (IntelliSense) та навігацію файловим деревом проєкту користувача.
2. **Консоль виконання (xterm.js):** розташована в нижній панелі. Через Go-шлюз (PTY Proxy) встановлюється WebSocket-з'єднання з реальним Docker-контейнером. Користувач може вводити команди Linux (наприклад, dotnet run або ls -la), і результати виводяться без помітних затримок (Latency < 50 ms).
3. **Панель AI-ментора:** розташована праворуч. Студент може виділити фрагмент коду та поставити запитання ШІ. Завдяки технології Server-Sent Events (SSE) відповідь моделі з'являється на екрані посимвольно (Streaming), імітуючи ефект живого спілкування з викладачем.

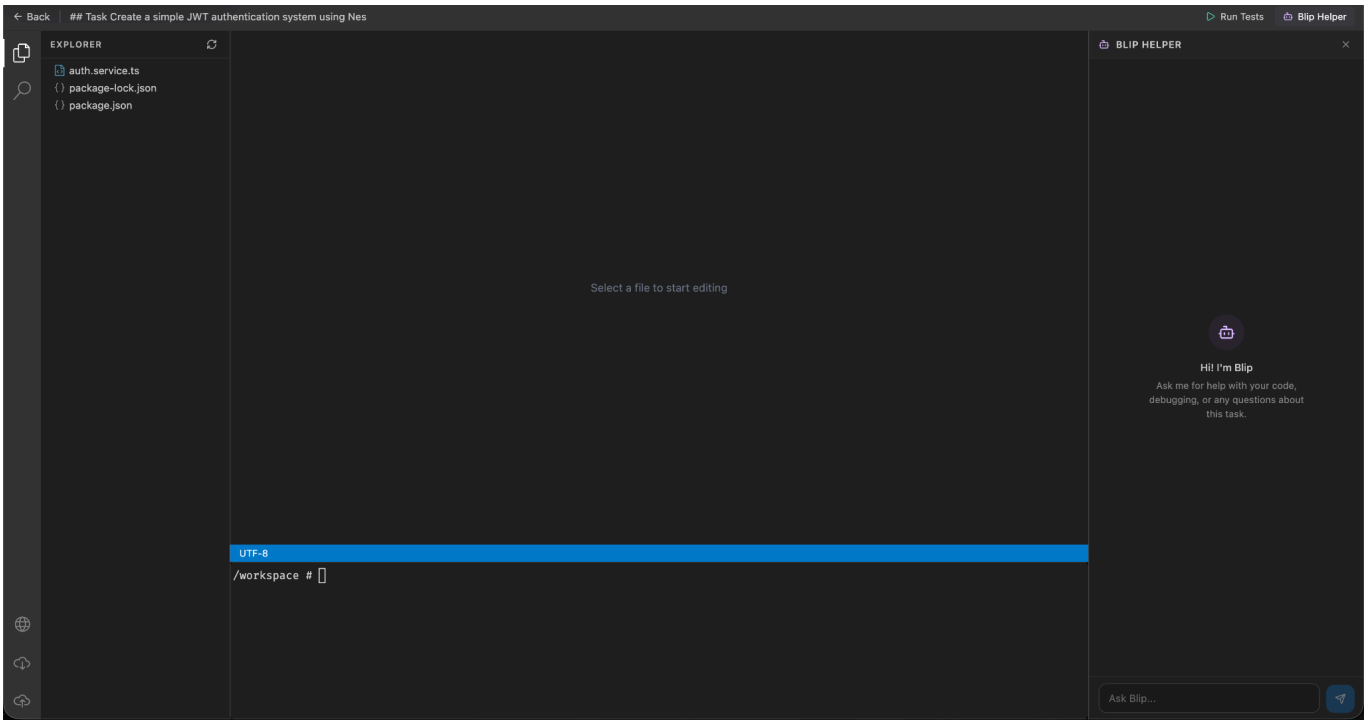

Рис. 4.3. Скріншот інтерфейсу Web IDE.

Джерело: Розроблене автором

4.3.4 Підсистема гейміфікації (Roadly+) та інклюзивність

Оцінка показників утримання (Retention) показала, що впровадження внутрішньої валюти (Roadly Coins) стимулює студентів до частішого проходження чекпойнтів. Зароблені монети відображаються у верхньому навігаційному барі. Користувач може перейти до вкладки «Магазин», щоб придбати косметичні покращення профілю (рамки, бекграунди). Стан інвентарю синхронізується з .NET-ядром за патерном CQRS.

Окремим досягненням стала практична робота панелі вебдоступності (A11y Panel). Тестування показало, що при активації режиму «Dyslexia-friendly» система миттєво перебудовує CSS-змінні на рівні :root, змінюючи шрифти в усіх компонентах системи, включаючи модальні вікна та чат, без жодних візуальних артефактів (FOUC).

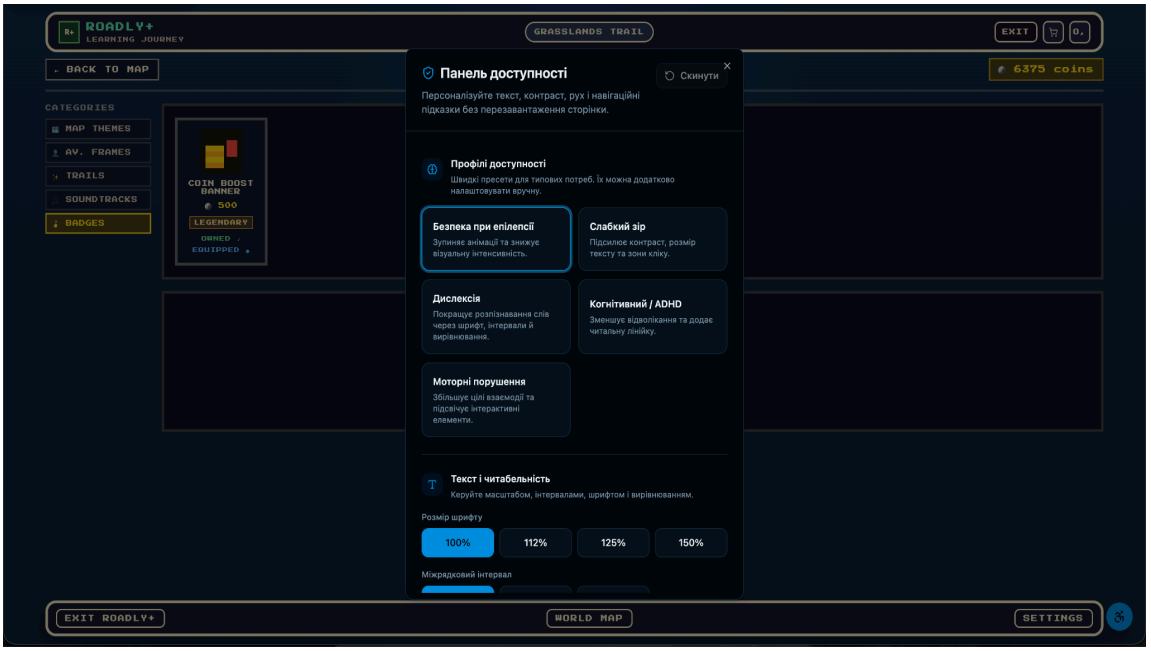

Рис 4.4: Скріншот магазину Roadly+ та відкритої бокової панелі налаштувань доступності

Джерело: Розроблене автором

4.3.5 Альтернативний доступ через розширення VS Code

Для підтвердження омніканальності платформи було протестовано розроблений MVP-модуль для Visual Studio Code. Студент успішно авторизується через OAuth2 у десктопному застосунку, після чого структура його навчального маршруту завантажується безпосередньо у бокову панель (Activity Bar) IDE. Це доводить, що архітектура API спроєктована правильно та дозволяє легко підключати нові клієнтські інтерфейси.

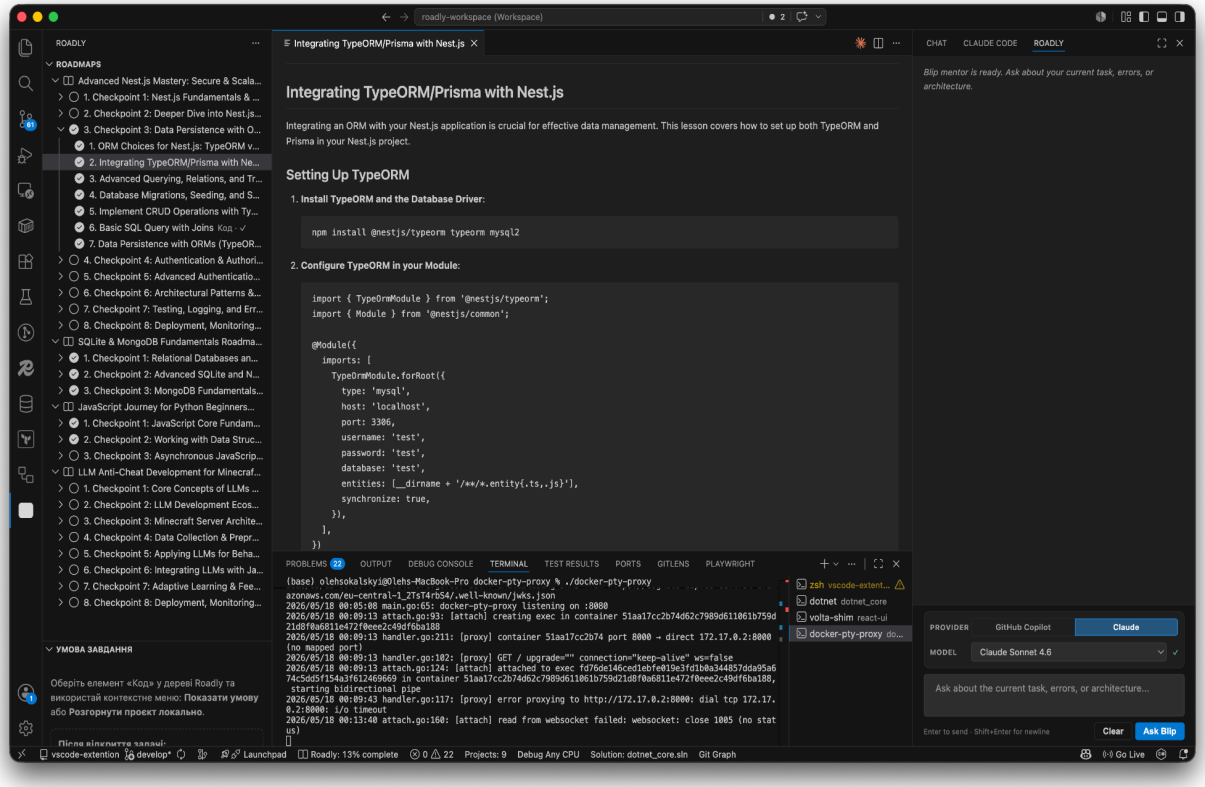

Рис 4.5: Скріншот відкритого VS Code,

Джерело: Розроблене автором

4.4 Вимоги до апаратного забезпечення та інструкція з розгортання

Оскільки платформа «Roadly» має складну мікросервісну архітектуру та використовує технологію контейнеризації для управління ефемерними середовищами виконання коду, до хост-сервера (Docker Host) висуваються специфічні вимоги.

4.4.1 Системні вимоги до серверного середовища

Для стабільної роботи 11 основних сервісів платформи (включаючи вимогливі до ресурсів бази даних та воркери штучного інтелекту) мінімальні вимоги до апаратного забезпечення розраховані наступним чином:

- **Процесор (CPU):** Мінімум 4 фізичні ядра (рекомендовано 8 ядер для ефективної роботи паралельних пулів Celery та багатопотокового середовища .NET).
- **Оперативна пам'ять (RAM):** мінімум 8 ГБ (рекомендовано 16 ГБ). Найбільшими споживачами пам'яті є PostgreSQL (особливо під час роботи з розширенням pgvector), Redis та ізольовані контейнери користувачів, для яких резервується окремий пул пам'яті.
- **Дисковий простір (Storage):** від 40 ГБ SSD (бажано NVMe) для швидкого читання/запису під час роботи RAG-конвеєрів і збирання Docker-образів.
- **Програмне забезпечення:** Операційна система Linux (Ubuntu 22.04 LTS або Debian 12), встановлені Docker Engine (версії 24.0+) та плагін Docker Compose v2.

4.4.2 Підготовка конфігурації та безпека

Перед початком розгортання необхідно виконати налаштування безпеки та змінних середовища. Відповідно до концепції Twelve-Factor App усі чутливі дані ізольовані від коду.

Адміністратор системи повинен створити кореневий файл .env на основі шаблону .env.example. У цьому файлі визначаються:

1. **Ключі доступу до зовнішніх API:** OPENAI_API_KEY, TAVILY_API_KEY для роботи інтелектуальних модулів.
2. **Автентифікаційні реквізити:** параметри пулу користувачів AWS Cognito (COGNITO_USER_POOL_ID, COGNITO_CLIENT_ID).
3. **Внутрішні секрети:** INTERNAL_SERVICE_KEY для захисту міжсервісної взаємодії, паролі до PostgreSQL, RabbitMQ та Redis.

4.4.3 Процес розгортання (Deployment Pipeline)

Процес запуску всієї інфраструктури повністю автоматизований. У кореневій директорії проєкту виконується єдина команда оркестрації:

docker compose up -d --build

Ця команда ініціює наступний конвеєр дій:

1. **Завантаження базових образів:** система завантажує необхідні образи Alpine/Ubuntu з Docker Hub.
2. **Компіляція:** Відбувається багатоетапна (multi-stage) збірка .NET API (через dotnet publish) та React UI (через vite build).
3. **Оркестрація запуску:** Docker Compose аналізує дерево залежностей (параметр depends_on). Спочатку запускається контур даних (PostgreSQL, RabbitMQ, Redis). Тільки після проходження ними перевірок працездатності (Healthchecks) запускаються модулі бізнес-логіки (.NET API та Python AI Service).
4. **Міграції:** Під час першого запуску .NET-контейнер автоматично застосовує міграції Entity Framework, створюючи необхідну структуру таблиць у базі даних.

4.4.4 Пост-релізний моніторинг та перевірка стану

Після завершення процесу розгортання адміністратор може перевірити статус усіх контейнерів за допомогою команди docker ps. Для інженерного моніторингу реалізовано спеціалізовані кінцеві точки перевірки стану (Health Endpoints):

- Шлюз терміналів (Go Proxy) повертає статус на /healthz.
- Брокер RabbitMQ має доступну вебпанель керування на порту 15672 для відстеження черг повідомлень.
- Здоров'я API перевіряється через доступність документації Swagger (порт 8080).

Такий підхід до розгортання (Infrastructure as Code) гарантує високу відмовостійкість системи та дозволяє за потреби легко мігрувати платформу на інші хмарні хостинги (наприклад, AWS EC2 або DigitalOcean Droplets).

Висновки до розділу 4

У четвертому розділі було здійснено розгортання та практичну апробацію розробленої освітньої платформи «Roadly». Детально описано процес оркестрації мікросервісної інфраструктури за допомогою Docker Compose, що забезпечив швидке, ізольоване та відтворюване розгортання всіх контейнерів системи в єдиному мережевому середовищі.

Особливу увагу приділено забезпеченню надійності та безпеки програмного продукту. Описано підходи до автоматизованого тестування бекенд-компонентів, зокрема, модульне тестування транзакційного доменного ядра (.NET/xUnit) та перевірку логіки ШІ-сервісів (Python/Pytest). Продемонстровано роботу критично важливих механізмів стабільності, таких як автоматичне фонове очищення неактивних сесій Web IDE за допомогою Hangfire та верифікація прав власності на ресурси (Ownership verification) на базі токенів AWS Cognito.

У результаті практичної оцінки підтверджено коректну та стабільну роботу всіх ключових підсистем: генерації персоналізованих навчальних маршрутів, інтерактивного Web IDE з терміналом, внутрішньої економіки (Roadly+) та інтеграції з Visual Studio Code. Розроблений продукт є повністю працездатним MVP, що відповідає заявленим вимогам і готовий до подальшого масштабування.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання проєктування та розробки комплексної мікросервісної освітньої платформи «Roadly», яка використовує технології генеративного штучного інтелекту, ізольовані середовища виконання коду та механізми гейміфікації для забезпечення адаптивного та персоналізованого процесу навчання програмуванню.

У результаті виконання роботи отримано такі теоретичні та практичні результати:

1. **Проаналізовано предметну область та наявні рішення на EdTech-ринку.** Встановлено, що традиційні платформи використовують лінійний (статичний) підхід до подання матеріалу, що призводить до високого когнітивного навантаження та передчасної втрати мотивації студентів. Обґрунтовано доцільність застосування великих мовних моделей (LLM) для динамічної генерації персоналізованих освітніх траєкторій та інтерактивного менторингу.
2. **Спроєктовано та реалізовано відмовостійку мікросервісну архітектуру.** Для забезпечення незалежного масштабування та високої продуктивності застосовано поліглотний підхід, а інфраструктуру розділено на 11 контейнеризованих сервісів. Транзакційне доменне ядро платформи побудовано на базі фреймворку .NET 9 з використанням патернів Clean Architecture та CQRS. Для асинхронної міжсервісної взаємодії впроваджено брокер повідомлень RabbitMQ, що дозволило уникнути жорсткої зв'язності між компонентами.
3. **Розроблено інтелектуальну підсистему для генерації контенту на базі Python.** За допомогою фреймворків FastAPI, Celery та екосистеми LangChain програмно реалізовано алгоритм гранулярної (паралельної) генерації навчальних маршрутів (Roadmaps). Для забезпечення фактологічної точності

AI-асистента впроваджено RAG-конвеєр (Retrieval-Augmented Generation), який здійснює семантичний векторний пошук у спеціалізованій базі даних PostgreSQL із розширенням pgvector[11].

4. **Створено захищений шлюз контейнеризації мовою Go для реалізації Web IDE.** Для забезпечення повноцінного практичного навчання розроблено проксі-сервіс, який встановлює двостороннє WebSocket-з'єднання між браузером клієнта та ізольованими контейнерами Docker. Це інженерне рішення дозволило безпечно виконувати користувацький код, компілювати проєкти та надавати доступ до емулятора термінала (xterm.js) у реальному часі без затримок.
5. **Розроблено сучасний клієнтський застосунок і підсистему гейміфікації.** На базі бібліотеки React 19 та Tailwind CSS створено веб-інтерфейс із RPG-візуалізацією навчального прогресу. Для стимулювання регулярної активності студентів інтегровано модуль «Roadly+», що включає віртуальну економіку, систему транзакцій (Roadly Coins) та магазин для кастомізації профілю. Ефективне управління станом на клієнті забезпечено комбінацією RTK Query та Zustand.
6. **Забезпечено інклюзивність платформи та омніканальний доступ.** Інтерфейс системи адаптовано до міжнародних стандартів вебдоступності WCAG 2.1. Розроблена панель A11y (завдяки динамічному керуванню CSS-змінними) дозволяє користувачам миттєво застосовувати спеціалізовані профілі (наприклад, для осіб з порушеннями зору, дислексією або епілепсією). Додатково розроблено модуль інтеграції з локальним середовищем Visual Studio Code, що дозволяє студентам проходити навчання безпосередньо у професійній IDE.
7. Проведено комплексне тестування надійності системи. Розроблена архітектура пройшла модульне тестування бізнес-логіки (xUnit), а також інтеграційну перевірку інтелектуальних сервісів (Pytest). Впроваджені механізми фонового

очищення ресурсів (Hangfire) та перевірки прав власності (Ownership verification) на основі JWT-токенів AWS Cognito гарантують високий рівень безпеки та стійкість серверної інфраструктури до перевантажень

Отже, розроблена платформа «Roadly» є технічно складним, масштабованим і готовим до впровадження програмним продуктом (MVP), що вирішує проблему персоналізації в IT-освіті. Мета кваліфікаційної роботи досягнута, а всі поставлені завдання виконано в повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OpenAI. GPT-4 Technical Report. arXiv. 2023. URL: https://arxiv.org/abs/2303.08774 (дата звернення: 09.05.2026).
2. .NET 9 Documentation / Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/ (дата звернення: 10.05.2026).
3. Amazon Cognito Documentation / Amazon Web Services. URL: https://docs.aws.amazon.com/cognito/ (дата звернення: 10.05.2026).
4. Celery: Distributed Task Queue Official Documentation. URL: https://docs.celeryq.dev/ (дата звернення: 10.05.2026).
5. Docker Documentation: Core concepts and Docker Compose / Docker Inc. URL: https://docs.docker.com/ (дата звернення: 10.05.2026).
6. FastAPI: High performance, easy to learn, fast to code, ready for production / S. Ramírez. URL: https://fastapi.tiangolo.com/ (дата звернення: 10.05.2026).
7. RabbitMQ: Reliable and highly available messaging and streaming. URL: https://www.rabbitmq.com/ (дата звернення: 10.05.2026).
8. Tailwind CSS: A utility-first CSS framework for rapid UI development. URL: https://tailwindcss.com/ (дата звернення: 10.05.2026).
9. Hangfire: Background processing in .NET and .NET Core. URL: https://www.hangfire.io/ (дата звернення: 11.05.2026).
10. LangChain Documentation: Build context-aware, reasoning applications / LangChain Inc. URL: https://python.langchain.com/ (дата звернення: 11.05.2026).
11. pgvector: Open-source vector similarity search for Postgres. URL: https://github.com/pgvector/pgvector (дата звернення: 11.05.2026).
12. PostgreSQL 17 Official Documentation / The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/17/ (дата звернення: 11.05.2026).
13. React 19: The library for web and native user interfaces / Meta Platforms. URL: https://react.dev/ (дата звернення: 11.05.2026).

14. Redux Toolkit: The official, opinionated, batteries-included toolset for efficient Redux development. URL: https://redux-toolkit.js.org/ (дата звернення: 11.05.2026).
15. The Go Programming Language Specification / Google. URL: https://go.dev/ref/spec (дата звернення: 11.05.2026).
16. Web Content Accessibility Guidelines (WCAG) 2.1 / W3C. 2018. URL: https://www.w3.org/TR/WCAG21/ (дата звернення: 11.05.2026).
17. xterm.js: A terminal for the web. URL: https://xtermjs.org/ (дата звернення: 11.05.2026).
18. Zustand: Bear necessities for state management in React. URL: https://zustand-demo.pmnd.rs/ (дата звернення: 11.05.2026).

ДОДАТОК А

```
// 1. Оголошення команди
public record UpdateItemCompletionCommand(
    Guid UserId,
    Guid CheckpointId,
    Guid ItemId) : IRequest<Result<Unit>>;

// 2. Обробник команди (Handler)
public class UpdateItemCompletionCommandHandler
    : IRequestHandler<UpdateItemCompletionCommand, Result<Unit>>
{
    private readonly IApplicationDbContext _context;
    private readonly IMessageProducer _messageProducer;

    public UpdateItemCompletionCommandHandler(
        IApplicationDbContext context,
        IMessageProducer messageProducer)
    {
        _context = context;
        _messageProducer = messageProducer;
    }

    public async Task<Result<Unit>> Handle(
        UpdateItemCompletionCommand request,
        CancellationToken cancellationToken)
    {
        var progress = await _context.UserProgresses
```

```
            .FirstOrDefaultAsync(p => p.UserId == request.UserId, cancellationToken);

        if (progress == null)
            return Result.Failure<Unit>(DomainErrors.Progress.NotFound);

        // Бізнес-логіка оновлення прогресу
        progress.MarkItemAsCompleted(request.CheckpointId, request.ItemId);
        await _context.SaveChangesAsync(cancellationToken);

        // Асинхронна публікація події у RabbitMQ
        await _messageProducer.PublishAsync(new
ProgressUpdatedEvent(request.UserId));

        return Result.Success(Unit.Value);
    }
}
```

ДОДАТОК Б

```
var upgrader = websocket.Upgrader{
    ReadBufferSize:  1024,
    WriteBufferSize: 1024,
    // Перевірка CORS делегується попередньому шару Middleware
    CheckOrigin: func(r *http.Request) bool { return true },
}

func (h *PtyHandler) AttachTerminal(w http.ResponseWriter, r
*http.Request) {
    // 1. Апгрейд протоколу
    wsConn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Printf("Помилка Upgrade: %v", err)
        return
    }
    defer wsConn.Close()

    // 2. Отримання ідентифікатора контейнера з параметрів
маршруту
    containerID := chi.URLParam(r, "containerId")

    // 3. Ініціалізація тунелю (передача керування)
    h.dockerService.StreamPty(r.Context(), containerID, wsConn)
}
```

ДОДАТОК В

```
func StreamPty(ctx context.Context, containerID string, ws
*websocket.Conn) error {
    // Підключення до PTY контейнера
    hijackedResp, err := dockerClient.ContainerAttach(ctx,
containerID, container.AttachOptions{
        Stream: true, Stdin: true, Stdout: true, Stderr: true,
Logs: true,
    })
    // ... обробка помилок

    errChan := make(chan error, 2)

    // Goroutine 1: Читання з Docker і запис у WebSocket (Output)
    go func() {
        buffer := make([]byte, 8192)
        for {
            n, err := hijackedResp.Reader.Read(buffer)
            if err != nil { errChan <- err; return }

            ws.WriteMessage(websocket.BinaryMessage, buffer[:n])
        }
    }()

    // Goroutine 2: Читання з WebSocket і запис у Docker (Input)
    go func() {
        for {
            _, msg, err := ws.ReadMessage()
            if err != nil { errChan <- err; return }

            hijackedResp.Conn.Write(msg)
        }
    }()
```

```
    // Блокування до завершення однієї з горутин (закриття сесії)
    <-errChan
    hijackedResp.Close()
    return nil
}
```