НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет/(ННІ) _____**інформаційних технологій**_____

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
__інформаційних технологій__	_____комп'ютерних наук_____
(назва факультету (ННІ))	(назва кафедри)
_____ ___Ігор БОЛБОТ____	_____ __Белла ГОЛУБ_____
(підпис) (ім'я ПРІЗВИЩЕ)	(підпис) (ім'я ПРІЗВИЩЕ)
"___" _____ 2025 р.	"___" _____ 2025 р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему __Інтелектуальна система перевірки знань з програмування_____

_____

Спеціальність _____121 «Інженерія програмного забезпечення»_____
(код і найменування)

Освітня програма _____Програмне забезпечення інформаційних систем_____
(назва)

Орієнтація освітньої програми_____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

__к.ф-м.н. доцент_____ _____ _____Віктор КИРИЧЕНКО_
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Керівник магістерської кваліфікаційної роботи

____к. т. н., доцент_____ _____ ____Тарас ЛЕНДЄЛ____
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Виконав _____ Максим ТИМОШЕНКО
(підпис) (ім'я ПРІЗВИЩЕ здобувача)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) _____**інформаційних технологій**_____

ЗАТВЕРДЖУЮ
Завідувач кафедри
__комп'ютерних наук_____
_к.т.н., доцент_____ _____ _Белла ГОЛУБ
(науковий ступінь, вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)
"_01_" _листопада__2024 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

_____Тимошенко Максим Петрович_____
(прізвище, ім'я, по батькові)

Спеціальність _____121 «Інженерія програмного забезпечення»_____
(код і найменування)

Освітня програма _____Програмне забезпечення інформаційних систем_____
(назва)

Орієнтація освітньої програми_____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи Інтелектуальна система перевірки знань з програмування _____
_____

затверджена наказом від "_01_" _листопада_____2024р. № 1963 «С»_____

Термін подання завершеної роботи на кафедру ____2025.11.29_____
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи _актуальні веб-джерела та методичні рекомендації щодо UI/UX-дизайну, принципів гейміфікації та адаптивного навчання, а також офіційна технічна документація та програмні інтерфейси обраного стеку технологій, зокрема React, NestJS, PostgreSQL, KeyDB та сервісу виконання коду PistonAPI_____

Перелік питань, що підлягають дослідженню:

1. Дослідження особливостей побудови графічного інтерфейсу для взаємодії з навчальним контентом
2. Аналіз методів побудови рейтингових таблиць та синтезу методу навчання
3. Дослідження принципів адаптації архітектури вебплатформи для навчання з урахуванням різних пристроїв користувачів та інтеграції сервісів
4. Аналіз методів для обробки даних для збереження та управління навчальною вибіркою

Перелік графічного матеріалу (за потреби) _____
_____

Дата видачі завдання "___10___" _вересня____2025 р.

Керівник магістерської кваліфікаційної роботи _____ _Тарас ЛЕНДЄЛ____
(підпис) (ім'я ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ ___Максим ТИМОШЕНКО_____
(ім'я ПРІЗВИЩЕ)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Скорочення/Позначення	Розшифрування та пояснення
AES	Advanced Encryption Standard (Стандарт розширеного шифрування даних)
API	Application Programming Interface (Інтерфейс прикладного програмування)
Auth	Authentication (Автентифікація – процес перевірки ідентичності користувача)
Backend	Серверна частина системи (Рівень прикладної логіки та даних)
CI/CD	Continuous Integration/Continuous Delivery (Безперервна інтеграція та доставка)
CORS	Cross-Origin Resource Sharing (Поділ ресурсів між доменами)
CSS	Cascading Style Sheets (Каскадні таблиці стилів)
DB	Database (База даних)
ELO	Elo rating system (Система рейтингу Ело)
ERD	Entity-Relationship Diagram (Діаграма сутностей та зв'язків)
ETL	Extract, Transform, Load (Видобування, перетворення та завантаження даних)
Flexbox	Flexible Box Module (Модель гнучких блоків у CSS)
Frontend	Клієнтська частина системи (Рівень представлення)
HTTPS	HyperText Transfer Protocol Secure (Захищений протокол передачі гіпертексту)
IDE	Integrated Development Environment (Інтегроване середовище розробки)

IT	Information Technology (Інформаційні технології)
JSONB	JSON Binary (Бінарний формат JSON у PostgreSQL)
JWT	JSON Web Token (Токен веб-токен JSON)
MVC	Model-View-Controller (Архітектурний патерн Модель-Вигляд-Контролер)
ORM	Object-Relational Mapping (Об'єктно-реляційне відображення)
Prisma	ORM-інструмент для Node.js/TypeScript
React	Бібліотека JavaScript для створення користувацьких інтерфейсів
REST	Representational State Transfer (Передача представницького стану)
Sandbox	Ізольоване середовище виконання
SQL	Structured Query Language (Мова структурованих запитів)
TLS	Transport Layer Security (Захист транспортного рівня)
UI	User Interface (Інтерфейс користувача)
UML	Unified Modeling Language (Уніфікована мова моделювання)
Unit	Модульне тестування
UX	User Experience (Досвід користувача)

ВСТУП

Актуальність теми зумовлена об'єктивною потребою ІТ-галузі у фахівцях, які володіють високим рівнем практичних навичок програмування. Існуючі освітні підходи, що базуються на ручній перевірці коду, характеризуються суттєвою затримкою зворотного зв'язку та обмеженою масштабованістю, що негативно впливає на ефективність навчання. З огляду на це, критично важливим є створення інтелектуальної навчальної вебплатформи, яка здатна забезпечити миттєву, об'єктивну оцінку коду та мотивувати користувачів через механізми гейміфікації. Об'єктом дослідження виступає процес проектування та розробки такої вебплатформи з адаптивним модульним підходом, тоді як предметом дослідження є методи та принципи побудови архітектури програмного забезпечення, графічного інтерфейсу, системи обробки даних та алгоритмів автоматичної перевірки коду. Метою магістерської роботи є проектування та розробка масштабованої інтелектуальної платформи, що поєднує структурований навчальний контент із системою автоматичної перевірки коду, рейтинговими таблицями та адаптивним інтерфейсом.

Для досягнення поставленої мети було послідовно вирішено низку дослідницьких та проектних завдань. Спочатку було досліджено особливості побудови графічного інтерфейсу для мінімізації когнітивного навантаження та забезпечення крос-платформної адаптації. Паралельно було проаналізовано методи побудови рейтингових таблиць та синтезовано ефективний метод навчання, що комбінує модульність, адаптивність та механізми гейміфікації. Ключовим етапом стало обґрунтування вибору інструментальних засобів (стек React, NestJS, PostgreSQL) та розробка адаптивної архітектури, включаючи аналіз методів обробки даних, використання кешування (KeyDB) для оптимізації доступу до рейтингової інформації та інтеграцію зовнішніх сервісів. Завершальні завдання включали розробку програмного забезпечення платформи

з реалізацією REST API та алгоритму взаємодії із сервісом виконання коду PistonAPI, а також проведення всебічного тестування.

В процесі роботи використовувалися методи системного аналізу для дослідження предметної області, об'єктно-орієнтоване моделювання (UML) для візуалізації архітектури та сутностей, а також компонентний підхід у поєднанні з тестуванням для верифікації коректності роботи модулів. Наукова новизна отриманих результатів полягає у синтезі методу побудови навчального процесу, який об'єднує модульне навчання, гейміфікаційні елементи на основі рейтингу та адаптивний підхід до вибору складності завдань. Особливо вагомою є розробка та обґрунтування спрощеної високопродуктивної архітектури, де внутрішній Judge System, що потребує складної інфраструктури (Docker/Kubernetes Sandbox), замінено на синхронну інтеграцію із зовнішнім Code Execution Service (PistonAPI). Таке рішення підвищує стабільність системи та знижує експлуатаційні витрати, зберігаючи при цьому ключову вимогу ізоляції виконання коду. Розроблені та реалізовані структури REST API оптимізовані для забезпечення безшовної взаємодії між адаптивним фронтендом (React) та бекендом (NestJS).

Основні положення та результати магістерської роботи були апробовані в рамках XVI Міжнародної науково-практичної конференції молодих вчених «Інформаційні технології: Економіка, Техніка, Освіта» (2025 р.) у секції «Наука про дані: технології OLTP і OLAP, машинне навчання, методи штучного інтелекту» (доповідь «ІНТЕЛЕКТУАЛЬНА СИСТЕМА ОЦІНКИ ЗНАНЬ З ПРОГРАМУВАННЯ»). Структурно магістерська робота складається зі вступу, трьох основних розділів, висновків, переліку використаних джерел та додатків, охоплюючи системний аналіз, моделювання та безпосередню розробку системи.

РОЗДІЛ 1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Загальна характеристика предметної області

У сучасних умовах цифровізації освіти особливої актуальності набувають вебплатформи для навчання програмуванню. Їхня поява зумовлена потребою у створенні інтерактивного, доступного й ефективного середовища для формування практичних навичок написання коду. Розвиток інформаційних технологій зумовив суттєві зміни в методах навчання: дедалі більше навчальних процесів переходить у цифровий формат, що відкриває нові можливості для викладачів і студентів.

Традиційні засоби тестування знань, засновані лише на виборі варіантів відповіді, не дають змоги оцінити реальний рівень розуміння та практичних умінь розробника. Такі підходи виявляються недостатніми в умовах, коли головною метою навчання є формування навичок створення програмних рішень, а не запам’ятовування фактів. У результаті значна частина процесу оцінювання потребує автоматизації — насамперед там, де від користувача очікується написання коду, перевірка результатів виконання й отримання об’єктивної оцінки.

Тому на перший план виходять інтелектуальні навчальні системи, що забезпечують можливість взаємодії користувача з програмним середовищем у режимі реального часу. Такі платформи дозволяють студентам отримувати практичні завдання, виконувати їх безпосередньо в інтерфейсі вебдодатка, а потім одразу бачити результати своєї роботи. Це робить процес навчання більш наочним і підвищує залученість користувачів.

Інтелектуальна система оцінки знань з програмування поєднує кілька ключових функціональних напрямів:

- подання навчального контенту у вигляді лекцій, тестів і практичних завдань, що забезпечує структурований процес засвоєння матеріалу;

- запуск користувацького коду на сервері та перевірка його результатів, що створює основу для об’єктивного оцінювання рівня виконання завдання;
- формування рейтингу користувачів на основі результатів виконання завдань, що стимулює зацікавленість і створює елемент змагальності;
- збирання статистики виконання завдань для подальшого аналізу.

У межах предметної області поєднуються принципи веброзробки, програмного тестування, структурування навчального контенту й елементів гейміфікації. Це поєднання дозволяє створити цілісне інтерактивне середовище, у якому навчальний процес стає не лише ефективним, а й більш мотивуючим для користувача. Такий підхід відповідає сучасним тенденціям розвитку освіти, коли основну роль відіграють не лише обсяг отриманих знань, а й практичні навички їх застосування у контексті реальних завдань програмування.

1.2.Дослідження особливостей побудови графічного інтерфейсу для взаємодії з навчальним контентом

Інтерфейс навчальної платформи повинен бути зручним і знайомим користувачу: навігація має бути спрощена, а елементи розташовані за логічними патернами. Це особливо важливо, якщо користувачі мають різний рівень технічної підготовки – наприклад, новачки потребують більш очевидних ярликів і зрозумілого меню[1]. Чітка ієрархія вмісту (групування матеріалів за темами чи модулями) допомагає організувати навчальний матеріал природно та зменшити когнітивне навантаження на учня.

Модульність контенту: навчальні матеріали (лекції, тести, практичні завдання з кодом тощо) варто розділяти на автономні блоки чи модулі. Кожен модуль може містити відео, текст та перевірочні запитання, що дозволяє користувачу засвоювати інформацію поетапно без перевантаження[2]. Такий підхід спрощує структуру курсу та розбиває великі обсяги інформації на керовані частини.

Візуальна ієрархія і навігація: важливий контент слід виділяти за допомогою типографіки, кольору та розташування. Наприклад, заголовки та

ключові блоки виконання вправ роблять інтерфейс більш зрозумілим. Необхідно чітко відокремлювати головне поле контенту (де відображаються лекції і завдання) від навігаційних панелей і меню, щоб користувач не губився в інтерфейсі. Правильна контрастність і достатні відступи гарантують, що тексти та кнопки легко читаються та сприймаються.

Реактивність і мінімалістичний дизайн: інтерфейс має швидко реагувати на дії користувача (клацання, введення даних тощо) без довгих затримок, а при цьому не бути перевантаженим візуальними ефектами чи зайвою графікою. Мінімалістичний підхід передбачає видалення всього непотрібного – залишають лише ті елементи, які необхідні для навчального процесу. Наприклад, часто використовують багато «білого простору» та спрощені іконки, щоб уникнути відволікань. Такий чистий та лаконічний дизайн дозволяє користувачу сфокусуватися на навчанні і знижує навантаження на увагу.

Гейміфікація: впровадження ігрових елементів (прогрес-барів, бейджів, таблиць лідерів, очок тощо) підвищує мотивацію й активність користувачів. Наприклад, прогрес-бар демонструє ступінь проходження курсу, а бейджі видаються за виконання певних завдань. Заохочення у вигляді очок і рейтингу створюють відчуття досягнень і наповнюють навчання елементами гри. Дослідження показують, що таке стимулювання утримує увагу студентів і заохочує їх долати мотиваційні труднощі.

1.3.Аналіз методів побудови рейтингових таблиць та синтезу методу навчання

Система рейтингу є потужним мотиваційним інструментом у сучасних навчальних платформах. Гейміфікаційні елементи (наприклад, бали, рівні, рейтингові таблиці) підвищують мотивацію учнів та залученість у навчальний процес[3]. Зокрема, використання таблиць лідерів (leaderboards) у навчанні може сприяти змаганню між учасниками та покращенню їх результативності.

Найпоширеніші підходи до побудови рейтингу:

- Бальна система: учень отримує бали за кожну виконану вправу або урок. Нарахування балів забезпечує відчутний зворотний зв’язок і стимулює виконувати додаткові завдання[4].
- Рівнева система (level system): учасники підвищують рівні, що відкриває їм нові модулі чи привілеї. Підвищення рівня є типовим прийомом гейміфікації, який мотивує учнів до поступового освоєння матеріалу.
- ELO-рейтинги: система, запозичена із шахів, динамічно оновлює оцінку учасників на основі їх успішності. Кожна «зустріч» учня із завданням розглядається як змагання двох суперників, тож рейтинг змінюється пропорційно до несподіваності результату: чим складніше завдання та сильніший суперник, тим більший вплив на рейтинг.

1.4.Дослідження принципів адаптації архітектури вебплатформи для різних пристроїв та інтеграції сервісів

Сучасна освітня веб-платформа повинна забезпечувати стабільну і передбачувану роботу як на десктопах, так і на мобільних пристроях; це досягається поєднанням адаптивної верстки на боці фронтенду, компонентної розробки інтерфейсу, чітко визначеного API-шару та модульної архітектури бекенду. Нижче наведено ключові принципи й рекомендовані практики.

Для забезпечення коректного відображення інтерфейсу на різних ширинах екранів використовується адаптивна верстка з застосуванням CSS Grid та Flexbox – вони дозволяють створювати як двовимірні макети сторінок, так і гнучкі рядкові/стовпчикові компонування, які плавно підлаштовуються під розмір вікна та орієнтацію пристрою [5]. Використання медіа-запитів і принципів «mobile-first» гарантує, що ключовий контент залишається доступним і зручним на смартфонах, планшетах та десктопах.

Інтерфейс слід проєктувати компонентно: кожен UI-елемент – самостійний, перевикористовуваний блок з чіткими вхідними даними (props/state) і обмеженою відповідальністю [6]. Така структура полегшує тестування, розробку та рефакторинг, а також дозволяє підтримувати однакову

логіку й вигляд у веб- та мобільних клієнтах (через спільні бібліотеки компонентів або код-генерацію). Компонентний підхід також спрощує інтеграцію редакторів коду, панелей результатів і модулів навігації.

Фронтенд комунікує з бекендом через чітко задокументований RESTful (або схожий) API: визначені ендпоінти для аутентифікації, отримання/оновлення контенту модулів, передачі результатів виконання завдань і моніторингу прогресу. Визначення контрактів (формат запитів/відповідей, коди помилок, схем автентифікації) робить інтеграцію детерміністичною і дозволяє паралельну розробку клієнтської та серверної частин.

Для бекенду рекомендовано застосувати модульну (modular monolith) архітектуру: логічно відділені модулі (завдання, рейтинг, аналітика, автентифікація) розміщуються в одному кодовому базисі, але із чіткими межами відповідальності та інтерфейсами між ними. Така організація спрощує локальне масштабування, тестування, розгортання й подальший перехід до мікросервісів, якщо це стане необхідним. NestJS як фреймворк надає засоби для організації модулів, CLI-workspace та шаблони проєктування, що відповідають цим вимогам [7].

Використання зовнішнього сервісу (наприклад, Piston) для ізольованого і безпечного виконання фрагментів коду дозволяє знизити ризики і навантаження на власні ресурси. Зовнішній двигун виконує код у контейнеризованому середовищі [8], повертає результат виконання та логи, а платформа передає до нього лише структуровані запити на виконання.

Для єдиної, масштабованої і безпечної автентифікації рекомендується використовувати перевірені сервіси як-от Firebase Authentication – вони забезпечують підтримку email/password, OAuth-провайдерів (Google, Apple) і телефонної автентифікації, а також механізми виявлення та відновлення облікових записів [9]. Інтеграція з основною SQL-базою (PostgreSQL) здійснюється через мапінг зовнішніх і внутрішніх ідентифікаторів користувачів.

Безпеку взаємодії між компонентами забезпечують:

- Транспортний шар: усі зовнішні та внутрішні виклики мають відбуватися через HTTPS/TLS; це забезпечує конфіденційність і цілісність трафіку.
- CORS: на фронтенд-серверах і в API слід жорстко сконфігурувати політику CORS, дозволяючи лише довірені домени й методи запитів та уникати «вільних» масок.
- Аутентифікація та токени: для передачі прав доступу використовуються стандартизовані токени (JWT) або сеансові механізми з короткоживучими токенами й можливістю їх відкликання; критичні дані у сховищі шифруються (наприклад, AES для полів, що зберігаються), а секрети – у спеціалізованих сховищах (secret manager).

1.5. Стратегії зберігання навчального контенту та управління даними

Обробка та надійне зберігання даних є критичною основою для стабільної роботи інтелектуальної платформи. Дані системи поділяються на дві основні категорії: навчальний контент (лекційні матеріали) та операційні/аналітичні дані (прогрес користувачів).

Навчальні матеріали мають бути доступними та легко оновлюваними. Розглядаються такі формати:

- PDF/Word – зручно для друку та традиційного перегляду, але складно інтегрувати в динамічний веб-інтерфейс.
- Markdown: Найкращий формат для зберігання лекційного контенту. Markdown забезпечує легкість написання, версіонування, швидке парсинг та відображення в інтерфейсі (HTML) з мінімальними накладними витратами [10]. Він також дозволяє інтегрувати в контент приклади коду та інші інтерактивні елементи.

Управління операційними даними включає інформацію про користувачів, їхній прогрес, структуру модулів. Основними методами обробки виступають:

- ORM-технології (TypeORM) – для надійної взаємодії з реляційною базою даних PostgreSQL.

- Кешування (KeyDB) – для швидкого доступу до часто запитуваної статистики та рейтингових таблиць.

В контексті розгортання та довгострокової стійкості, застосовується гібридна стратегія зберігання даних, що є стандартною практикою для більшості сучасних IT-компаній.

Цей підхід дозволяє оптимізувати витрати та мінімізувати ризики, поєднуючи переваги різних типів інфраструктури. Основні компоненти та критичні операційні дані платформи спроєктовані для розміщення у хмарному середовищі (Cloud Storage), яке гарантує високу гнучкість масштабування та доступність (High Availability).

Однак, архітектура, побудована на контейнеризації (Docker), підтримує можливість гібридного розгортання і використання локальних (On-Premise) або виділених ресурсів для певних чутливих або архівних даних, забезпечуючи незалежність від конкретного хмарного провайдера та гарантуючи адаптацію платформи до різних вимог.

1.6. Вибір та обґрунтування інструментальних засобів

Ефективність та надійність програмної системи безпосередньо залежать від обґрунтованого вибору інструментальних засобів. Для реалізації інтелектуальної навчальної вебплатформи було обрано сучасний, масштабований стек технологій, оптимізований для розробки асинхронних, високопродуктивних застосунків.

1.6.1. Засоби реалізації клієнтської частини (Frontend)

У межах дослідження клієнтської частини вебплатформи розглянуто можливості використання бібліотеки React у поєднанні з мовою програмування TypeScript як базових інструментів для побудови сучасного, надійного та масштабованого інтерфейсу користувача. Проведений аналіз показав, що така комбінація технологій є однією з найпоширеніших у сучасній веброзробці завдяки поєднанню продуктивності, зручності у супроводі та високої

адаптивності до змін вимог [11]. React забезпечує компонентний підхід, який дозволяє структурувати інтерфейс у вигляді окремих функціональних блоків, що легко модифікуються та повторно використовуються. Такий підхід особливо ефективний у контексті навчальних платформ, де передбачено постійне оновлення контенту, взаємодію користувача з численними елементами та необхідність динамічного відображення результатів.

TypeScript, у свою чергу, розглядається як засіб підвищення надійності програмного коду через упровадження статичної типізації. Це дає змогу виявляти помилки ще на етапі розробки та забезпечує узгодженість типів даних під час обміну між клієнтською та серверною частинами системи [12]. Застосування TypeScript також полегшує підтримку великих проєктів, у яких важливо дотримуватися суворої структури коду, що є актуальним для освітніх платформ, орієнтованих на тривалу експлуатацію й постійне оновлення. Проведений аналіз літературних джерел і практичних кейсів засвідчив, що поєднання React і TypeScript дає змогу досягти високої стабільності користувацьких інтерфейсів, а також створює передумови для безпечної інтеграції з зовнішніми сервісами через REST API.

Окрему увагу в межах дослідження приділено структурі інтерфейсу. Виділено два основні напрями: користувацьку частину, орієнтовану на взаємодію з навчальним контентом, та адміністративну панель, що забезпечує управління даними. Користувацький інтерфейс доцільно реалізовувати на базі React із використанням принципів адаптивного дизайну (Responsive Design), що гарантує коректне відображення контенту на пристроях із різними розмірами екрана. Такий підхід відповідає сучасним вимогам до зручності користування та забезпечує доступність навчального матеріалу незалежно від типу пристрою. Згідно з проведеним аналізом, важливим аспектом є також ергономічність подання навчального контенту: чітка візуальна ієрархія, помірна кількість інтерактивних елементів та узгоджені кольорові схеми сприяють кращому сприйняттю інформації.

Для дослідження можливостей створення адміністративного інтерфейсу розглянуто використання фреймворку react-admin як інструменту для побудови зручних панелей управління контентом. Його особливістю є наявність готових компонентів для виконання типових операцій із даними, таких як створення, редагування, видалення чи фільтрація записів. Завдяки цьому react-admin може суттєво зменшити витрати часу на розробку інтерфейсів адміністрування та підвищити стандартизацію їхньої структури [13]. Дослідження показало, що такий підхід особливо ефективний для навчальних систем, де адміністратор або викладач часто взаємодіє з великим обсягом даних і потребує інтуїтивного середовища для управління навчальним процесом.

Узагальнюючи результати, можна зробити висновок, що поєднання React, TypeScript і react-admin є доцільним технологічним рішенням для побудови клієнтської частини сучасної навчальної вебплатформи. Цей підхід забезпечує гнучкість, стійкість і масштабованість системи, водночас підтримуючи високий рівень зручності користування та надійності. З погляду дослідження, зазначені технології демонструють оптимальне співвідношення між складністю впровадження та якістю кінцевого результату, що підтверджує їхню ефективність у контексті створення інтерактивних навчальних середовищ нового покоління.

1.6.2. Засоби реалізації серверної частини (Backend)

У процесі дослідження серверної частини вебплатформи проаналізовано доцільність використання фреймворку NestJS, побудованого на основі Node.js та мови TypeScript. NestJS було обрано як об’єкт дослідження завдяки його сучасній модульній архітектурі, орієнтованій на підтримку принципів структурності, масштабованості та чистоти коду [14]. Фреймворк наслідує концепції, характерні для Angular, і поєднує об’єктно-орієнтований, функціональний та реактивний підходи до побудови серверної логіки. Завдяки цьому NestJS розглядається як ефективне рішення для побудови навчальних

систем, які потребують обробки великої кількості асинхронних операцій і взаємодії з різними джерелами даних.

У ході дослідження визначено, що ключовою перевагою NestJS є його модульна структура, яка ґрунтується на трьох основних складових – модулях, контролерах і провайдерах. Така організація забезпечує високу структурованість застосунку, полегшує підтримку, а також сприяє дотриманню принципів SOLID. Кожен модуль відповідає за окремий аспект функціональності системи, що дозволяє локалізувати зміни й мінімізувати ризик впливу однієї частини на іншу. Це особливо важливо для навчальних платформ, які з часом можуть розширюватися новими модулями, такими як управління курсами, аналітика результатів чи інтеграція з зовнішніми сервісами.

З технічного погляду, використання Node.js як середовища виконання є обґрунтованим вибором для вебплатформ, що інтенсивно працюють із асинхронними I/O-операціями, такими як взаємодія з базою даних або зовнішніми API. Під час дослідження встановлено, що неблокуюча модель обробки запитів у Node.js забезпечує високу швидкість відгуку REST API, що має вирішальне значення для навчальних платформ, де користувачі очікують миттєвого зворотного зв’язку після надсилання результатів або виконання коду. У порівнянні з традиційними синхронними підходами, Node.js демонструє значно кращі показники масштабованості при великій кількості одночасних з’єднань, що робить його доцільним вибором для інтерактивних освітніх систем.

Окремо досліджено вплив використання TypeScript на якість і стабільність серверної частини. Статична типізація дозволяє виявляти помилки на етапі компіляції, що підвищує надійність та полегшує командну розробку. Завдяки суворій структуризації коду з використанням інтерфейсів, декораторів та сервісів, TypeScript у межах NestJS сприяє створенню більш передбачуваної логіки, полегшує супровід і сприяє збереженню єдиного стилю кодування.

Загалом проведене дослідження показало, що поєднання NestJS, Node.js і TypeScript формує ефективну архітектурну основу для побудови серверної частини навчальної вебплатформи. Такий підхід забезпечує не лише технічну ефективність і масштабованість, але й створює гнучку методологічну базу для подальшого розширення системи. Таким чином, використання зазначених технологій є виправданим з погляду наукового та прикладного аналізу, оскільки вони відповідають сучасним тенденціям у сфері розроблення інтерактивних освітніх рішень і сприяють формуванню стабільних інфраструктур для довготривалих навчальних процесів.

1.6.3. Вибір та обґрунтування сховища даних

У межах дослідження систем збереження даних для навчальних вебплатформ проаналізовано доцільність використання реляційної системи управління базами даних PostgreSQL у поєднанні з KeyDB як інструментом кешування та швидкого доступу до оперативної інформації.

PostgreSQL розглядається як оптимальний вибір для систем, що потребують високої надійності, цілісності даних і підтримки складних транзакцій. Вона забезпечує стабільну роботу з великими обсягами структурованої інформації [15], що особливо актуально для навчальних платформ, у яких зберігаються профілі користувачів, результати тестів і статистичні дані.

KeyDB, у свою чергу, досліджувався як інструмент для підвищення швидкодії системи за рахунок використання підходу “in-memory data store” [16]. Його структура типу “ключ–значення” забезпечує надзвичайно швидкий доступ до даних, що особливо важливо у випадках частого оновлення або зчитування інформації. У контексті навчальної платформи KeyDB доцільно застосовувати для обробки даних, пов’язаних із рейтингами користувачів, які постійно змінюються в режимі реального часу. Використання KeyDB у ролі кешуючого шару дозволяє суттєво знизити навантаження на основну базу даних

PostgreSQL, скоротити час відгуку серверної частини та забезпечити стабільність системи навіть за великої кількості одночасних запитів.

Таким чином, проведене дослідження підтверджує ефективність комбінованого використання PostgreSQL і KeyDB у межах архітектури навчальної вебплатформи. Такий підхід забезпечує баланс між стабільністю зберігання структурованих даних і високою швидкодією доступу до оперативної інформації, що є критично важливим для систем, орієнтованих на інтерактивність та динамічну взаємодію з користувачем.

1.6.4. Обґрунтування інтеграції зовнішнього сервісу PistonAPI

У рамках дослідження механізмів автоматичної перевірки програмного коду проаналізовано доцільність використання зовнішнього сервісу PistonAPI як основного засобу виконання користувацьких рішень у безпечному середовищі. Такий підхід є сучасною тенденцією у побудові навчальних платформ, орієнтованих на роботу з кодом, оскільки дозволяє уникнути складнощів, пов’язаних із налаштуванням, моніторингом і підтримкою власної інфраструктури виконання. Зокрема, відмова від розгортання черг повідомлень типу RabbitMQ чи Kafka та локальних execution workers суттєво спрощує архітектуру системи й зменшує кількість потенційних точок відмови.

Дослідження показало, що передача функцій ізоляції виконання коду зовнішньому провайдеру, такому як PistonAPI, підвищує рівень безпеки та стабільності всієї платформи [17]. Сервіс забезпечує повне розмежування середовищ виконання, що унеможливлює несанкціонований доступ до мережевих ресурсів, файлової системи або даних інших користувачів. Такий рівень ізоляції є критично важливим для навчальних систем, у яких користувачі мають можливість надсилати власний код на виконання. Крім того, застосування спеціалізованого зовнішнього рішення зменшує потребу в адмініструванні складних контейнерних або віртуалізованих інфраструктур, що зазвичай вимагають значних ресурсів і технічної експертизи.

З економічного погляду, використання PistonAPI дозволяє скоротити експлуатаційні витрати, пов’язані з підтримкою серверних ресурсів, масштабуванням середовищ виконання та оновленням компіляторів або інтерпретаторів. Це, у свою чергу, звільняє ресурси для зосередження на розробленні навчального контенту, оптимізації логіки оцінювання й удосконаленні інтерфейсу користувача. У процесі аналізу також виявлено, що інтеграція з PistonAPI відбувається через стандартизований REST-інтерфейс, що забезпечує гнучкість у розширенні функціональності системи та можливість швидкої заміни або оновлення сервісу без суттєвих змін у внутрішній архітектурі.

Таким чином, проведене дослідження підтвердило, що використання PistonAPI як зовнішнього сервісу для виконання коду є доцільним і технологічно обґрунтованим рішенням. Воно забезпечує баланс між безпекою, надійністю та ефективністю, знижуючи технічну складність системи й дозволяючи зосередитися на ключових аспектах навчального процесу.

1.6.5.Інші інструментальні засоби

Firebase Authentication розглядається як ефективний інструмент для організації процесу автентифікації без необхідності створення та супроводу власного сервісу авторизації. Його перевага полягає у високому рівні безпеки, підтвердженому масштабним використанням технології в комерційних і освітніх проєктах, а також у підтримці багатьох методів входу – зокрема, через сторонніх провайдерів, таких як Google або Apple. Це дозволяє забезпечити зручність і швидкість входу для користувачів, зменшуючи бар’єр початку роботи з платформою. У межах дослідження встановлено, що застосування готових сервісів автентифікації знижує ризики, пов’язані з неправильним зберіганням облікових даних або вразливістю до атак типу «brute force» та «SQL injection». Firebase забезпечує централізоване управління доступом і надійне шифрування токенів, що робить його доцільним вибором для систем, орієнтованих на безпечну роботу з персональними даними [18].

Таким чином, дослідження показало, що Firebase Authentication створює надійну інфраструктурну основу для управління користувачами. Сервіс дозволяє значно скоротити обсяг технічних робіт, підвищити безпеку системи та забезпечити стабільну роботу платформи навіть за великої кількості користувачів. Застосування цієї технології підтверджує ефективність використання хмарних сервісів у контексті навчальних вебплатформ, де особливе значення мають надійність, масштабованість та захист персональних даних.

РОЗДІЛ 2. МОДЕЛЮВАННЯ СИСТЕМИ

2.1. Підхід до моделювання

Для проєктування інтелектуальної платформи оцінки знань з програмування застосовано поєднання функціонального та об’єктно-орієнтованого підходів. Такий підхід забезпечує логічну послідовність етапів побудови системи та дозволяє чітко визначити її основні компоненти, їх взаємозв’язки та межі відповідальності. Використання декількох рівнів моделювання створює цілісне уявлення про структуру платформи, що важливо для підтримки узгодженості між бізнес-логікою, даними та користувацькою взаємодією.

Моделювання виконано на трьох рівнях, кожен з яких відображає різні аспекти роботи системи та забезпечує її цілісність:

- Функціональний рівень (Use Cases) – охоплює визначення основних ролей користувачів та опис ключових сценаріїв їхньої взаємодії з платформою. На цьому етапі визначаються основні дії, які можуть виконувати користувачі, та способи їх реалізації у межах системи.
- Статичний рівень (ERD / класи) – відображає модель даних, структуру сутностей та зв’язки між ними. Для реалізації використано класи та структури, визначені за допомогою інструментів TypeORM у середовищі TypeScript, що забезпечує узгодженість типів і гнучкість при подальшому розширенні структури бази даних.
- Динамічний рівень (послідовності) – описує сценарії виконання основних операцій, таких як реєстрація користувача, запуск коду або оновлення рейтингу. Цей рівень відображає логіку послідовної взаємодії компонентів і дозволяє краще зрозуміти порядок обміну даними між клієнтською та серверною частинами.

Моделювання орієнтовано на масштабованість і розділення відповідальностей (Separation of Concerns), що є необхідною умовою для забезпечення стабільності та подальшого розвитку системи. Кожен компонент платформи виконує чітко визначену роль, що полегшує підтримку та оновлення коду без впливу на інші модулі.

Архітектурно система реалізована як трирівневий застосунок, який складається з клієнтської частини (React UI), серверної логіки (NestJS REST API) та бази даних. Такий підхід є стандартним для сучасних вебсистем, оскільки він забезпечує відокремлення шарів представлення, логіки та збереження даних, а також полегшує масштабування та налагодження.

Для виконання програмного коду система інтегрується із зовнішнім сервісом PistonAPI, який відповідає за безпечне виконання користувацьких програм у віддаленому середовищі. Це рішення дозволяє зменшити навантаження на власну інфраструктуру та спрощує процес розгортання системи.

Для обробки бізнес-логіки використовується REST API, який забезпечує чіткий поділ між клієнтом і сервером, а також зручну взаємодію між компонентами через HTTP-запити. PostgreSQL використовується для збереження даних, що гарантує надійність, транзакційність і підтримку складних зв’язків між сутностями. Для кешування проміжних результатів застосовується KeyDB, який забезпечує високу швидкодію при обробці повторних запитів і знижує навантаження на основну базу даних.

2.2. Діаграма прецедентів

Діаграма прецедентів (Use Case Diagram) є першим етапом моделювання системи згідно з методологією UML і використовується для графічного відображення функціональних вимог до системи з точки зору різних груп користувачів (акторів) [19]. Вона визначає, які дії (прецеденти) можуть бути виконані акторами в межах системи.

Для інтелектуальної навчальної платформи визначено три основні внутрішні актори: Користувач (Студент), який взаємодіє з навчальним контентом; Викладач/Автор завдань, відповідальний за створення та управління контентом; та Системний адміністратор, що виконує налаштування та моніторинг. Крім того, діаграма враховує взаємодію з ключовими зовнішніми системами, зокрема Сервісом виконання коду (PistonAPI), який є критично важливим для реалізації автоматичної перевірки.

На рисунку 2.2.1 представлена діаграма прецедентів інтелектуальної навчальної платформи, що деталізує взаємодію акторів із ключовим функціоналом.

Рис. 2.1. Діаграма прецедентів інтелектуальної навчальної платформи

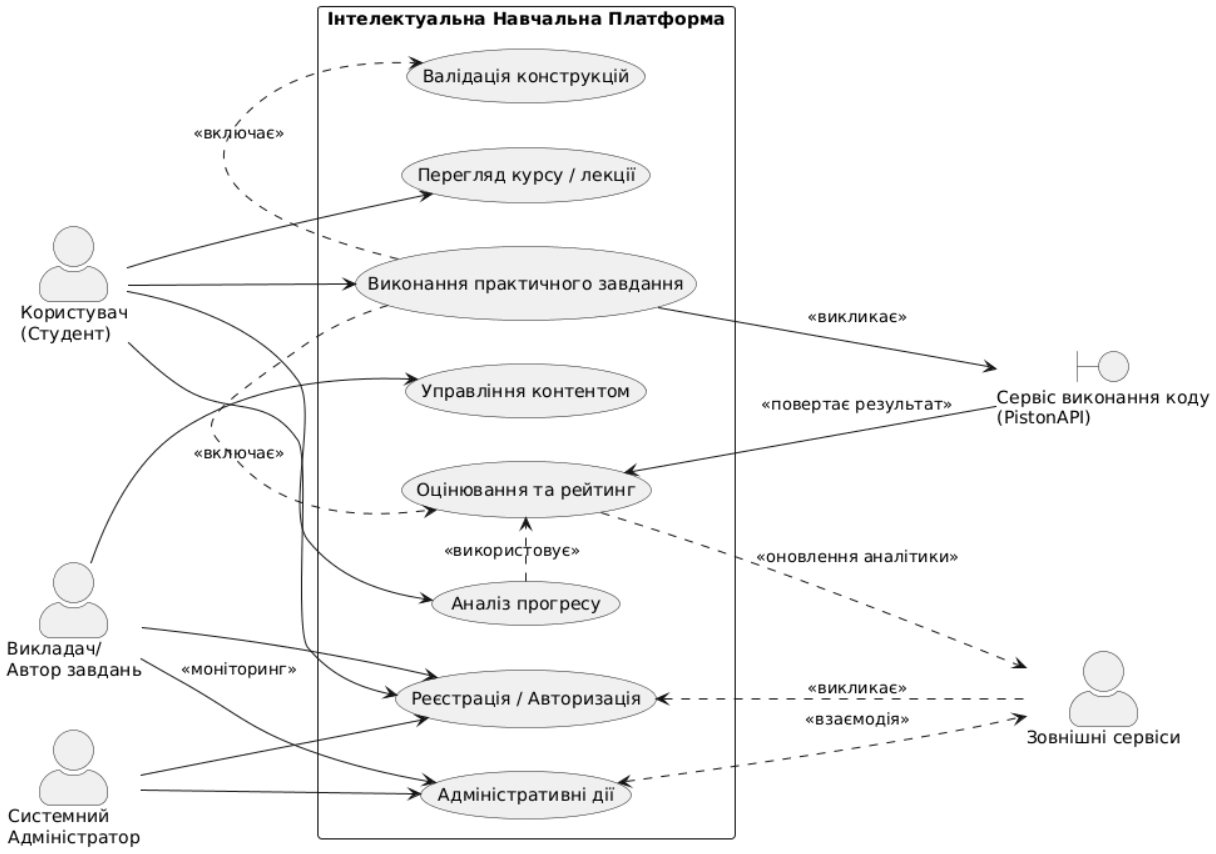

Рис. 2.2.1 Діаграма прецедентів

Як видно з діаграми, центральним функціональним блоком є Виконання практичного завдання. Цей прецедент включає допоміжні прецеденти – Валідація використання конструкцій та Оцінювання та рейтинг. Ключовим

елементом моделі є пряма взаємодія прецеденту Виконання практичного завдання із зовнішнім Сервісом виконання коду (PistonAPI). Це підкреслює реалізовану синхронну архітектуру перевірки коду, де Backend API виступає посередником, викликаючи PistonAPI та передаючи отриманий результат далі для Оцінювання та рейтингу.

Інші ключові взаємодії включають управління контентом, доступне лише Викладачу, та адміністративні дії, які забезпечують моніторинг та управління інфраструктурою. Таким чином, діаграма прецедентів слугує основою для подальшого проектування статичних (діаграма класів) та динамічних моделей (діаграма послідовності), деталізуючи ключові потоки даних у системі.

2.3. Статична модель: сутності та ERD

2.3.1. Опис статичної моделі та ключових сутностей

Статична модель системи описує структуру даних (сутність) та їхні взаємозв'язки, що є основою для проектування бази даних (Entity-Relationship Diagram – ERD) [20]. Усі сутності реалізовані як класи TypeScript із використанням TypeORM для взаємодії з PostgreSQL, що забезпечує цілісність даних та спрощує розробку.

Основні сутності, що формують ядро навчальної платформи:

- User (Користувач): Центральна сутність, що фіксує профільні дані, ролі (student, teacher, admin) та історію активності. Має зв'язки «Один-до-багатьох» із Progress та Submission.
- Lecture (Лекція): Зберігає теоретичний контент у форматі Markdown. Має зв'язок «Один-до-одного» з Module, забезпечуючи безпосередню інтеграцію теорії у навчальний модуль.
- Module (Навчальний модуль): Структурна одиниця курсу, що об'єднує лекцію та пов'язані завдання.
- CodeTask (Практичне завдання): Містить параметри автоматичної перевірки коду, включаючи опис, набір тестів (test_case) та критичні

обмеження виконання (time_limit_ms, memory_limit_kb), які передаються до PistonAPI.

- TestTask (Тестове завдання): Сутність для звичайних тестів із варіантами відповідей.
- Progress (Прогрес): сутність, яка фіксує належність користувача до певного модуля навчальної платформи. Вона відображає факт того, що користувач закінчив відповідний модуль.

2.3.2. Взаємозв'язки між сутностями

Модель даних побудована за реляційним підходом, що забезпечує структуроване зберігання та логічну узгодженість інформації.

Один-до-багатьох (One-to-Many) – один User може мати декілька записів Progress, що дозволяє фіксувати завершення різних навчальних модулів одним користувачем.

Один-до-одного (One-to-One) – кожен Module має рівно одну пов’язану Lecture. Це забезпечує чітке поєднання теоретичного матеріалу з відповідним навчальним контекстом без дублювання або можливості невідповідності.

Багато-до-одного (Many-to-One) – кожен CodeTask та TestTask належать певному Module, що забезпечує структурне групування практичних і тестових завдань у межах відповідних навчальних блоків.

Таке проєктування забезпечує цілісність даних та оптимізує механізми доступу через ORM (TypeORM), що спрощує побудову запитів та навігацію між сутностями.

2.3.3. Діаграма класів (UML)

На рисунку 2.3.3.1 представлена діаграма класів, що візуалізує статичну структуру даних системи, включаючи ключові атрибути та взаємозв'язки.

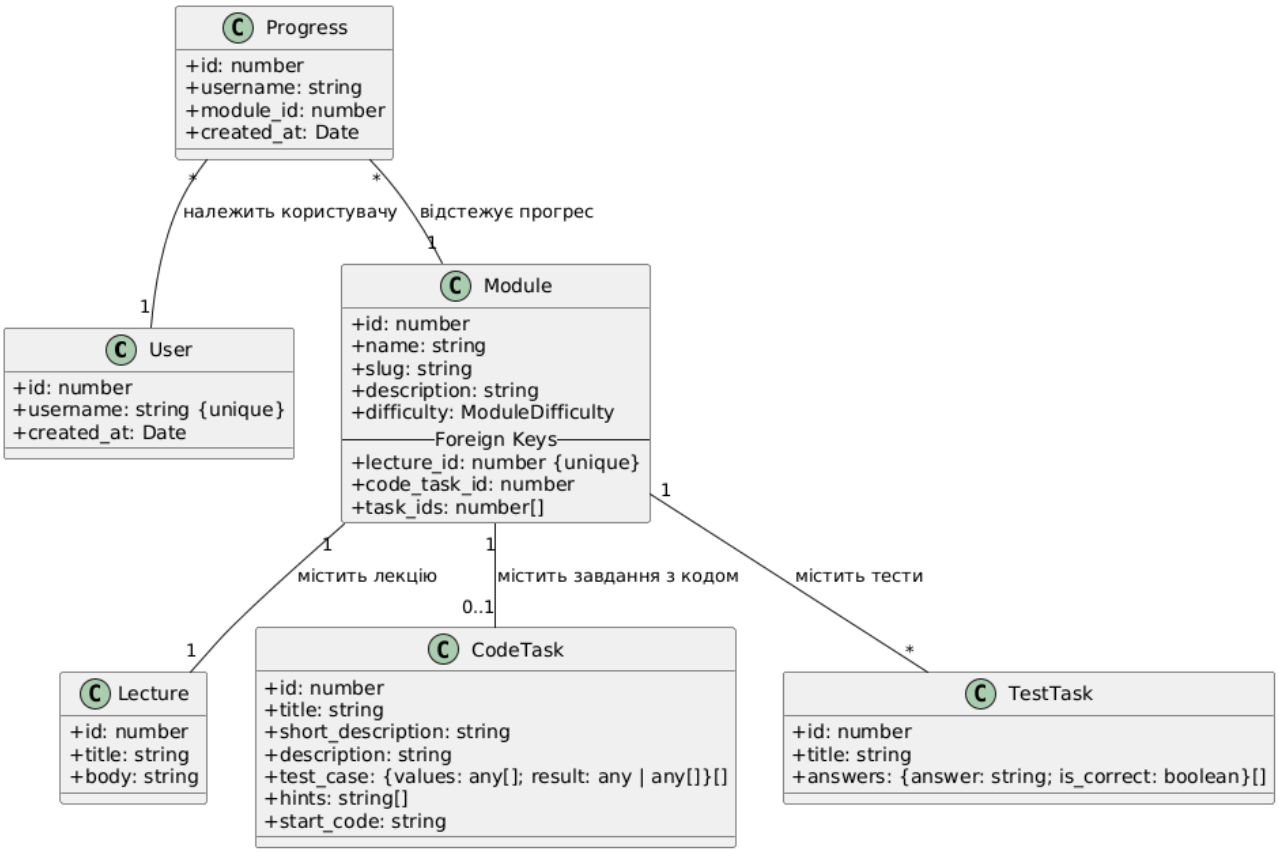

Рисунок 2.3.3.1 Діаграма класів

2.4.Динамічна модель

2.4.1.Сценарій: Виконання завдання

Цей сценарій описує синхронний потік управління для ключового прецеденту «Виконання завдання з кодом» і демонструє взаємодію між клієнтською частиною, Backend API та зовнішнім PistonAPI відповідно до нової логіки (див. рис. 2.4.1.1).

1. Ініціалізація (GET Data): Компонент монтується. Фронтенд робить синхронний GET /code-task/:id для отримання даних про завдання (включаючи test_case та start_code). Тестові кейси нормалізуються та зберігаються у стані компонента.
2. Підготовка Коду (Frontend Logic): Користувач у UI натискає "Надіслати код" (handleSubmit). Фронтенд генерує єдиний тестовий скрипт, який включає: код користувача, функцію для ітерації по всіх test_case,

виконання функції користувача з цими тестами та вивід фінальних результатів у спеціальному форматі (TEST_RESULTS:).

3. Зовнішній виклик (PistonAPI): Фронтенд виконує синхронний HTTP-запит до Piston API (https://emkc.org/api/v2/piston/execute), передаючи згенерований тестовий скрипт як файл (main.js).
4. Виконання: Piston API виконує весь згенерований скрипт в ізольованому середовищі (Sandbox). Скрипт послідовно тестує код користувача і друкує фінальний JSON-результат у stdout.
5. Обробка (Frontend Logic): Фронтенд очікує на відповідь, парсить data.run.output від Piston API, шукаючи рядок TEST_RESULTS:. Отриманий JSON-рядок десеріалізується, і Фронтенд визначає фінальний статус: чи дорівнює passedCount (кількість пройдених тестів) totalTests (загальній кількості).
6. Фіксація (POST Progress - Успіх): Тільки якщо всі тести пройдено (passedCount === totalTests), Фронтенд робить синхронний POST /progress, передаючи username (з localStorage) та module_id.
7. Збереження (Backend): Backend отримує запит на /progress, створює запис Progress у базі даних (фіксуючи час проходження та зв'язок з користувачем і модулем).
8. Фідбек: Фронтенд відображає детальний форматний вивід тестів (formatTestResults) та оновлює стан isTaskCompleted для зміни кнопки "Надіслати код" на "Назад до модулів".

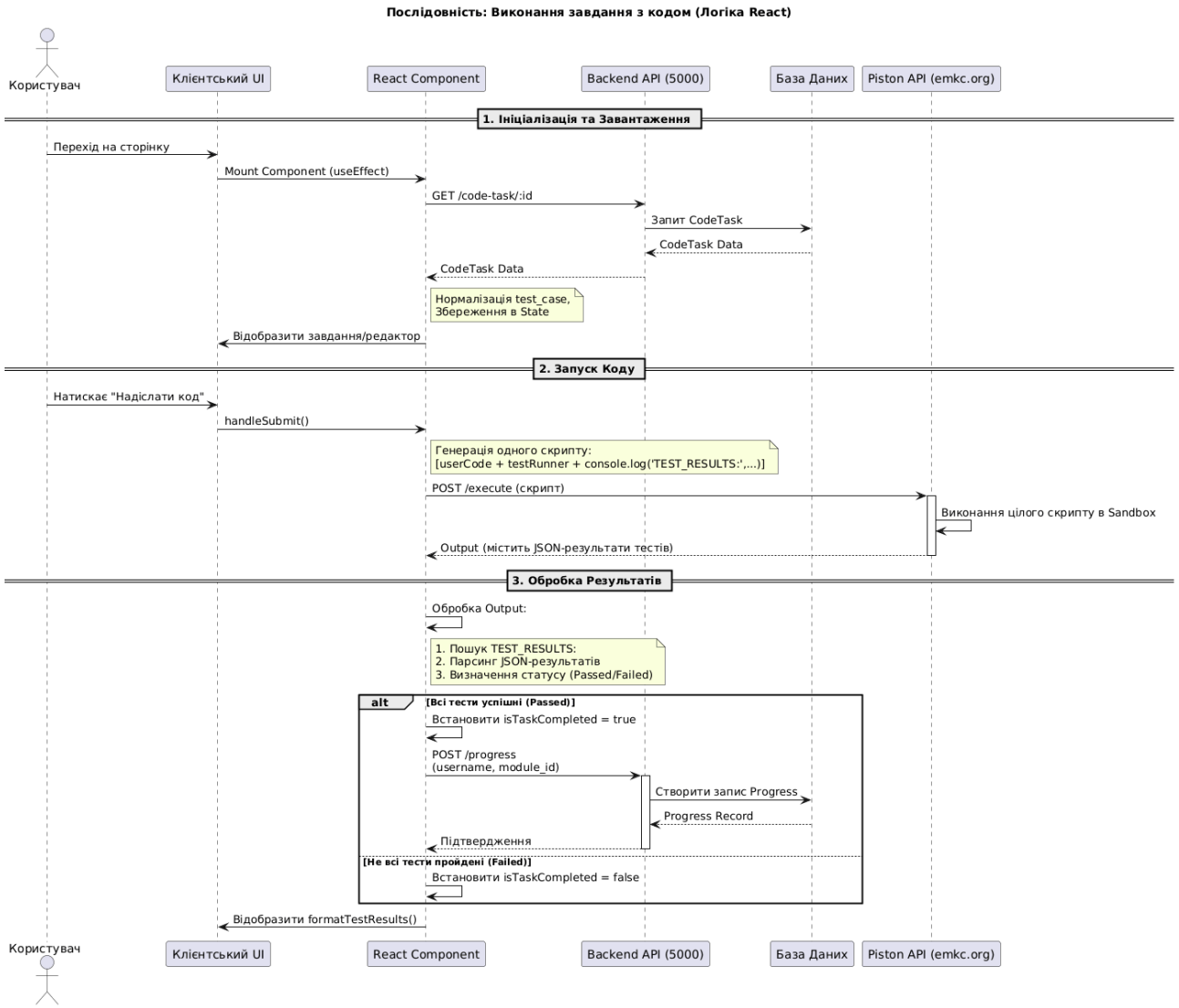

Рис. 2.4.1.1 Діаграма послідовності виконання завдання

2.5.Компонентна та розгортальна архітектура

Проєктована інтелектуальна навчальна платформа реалізована на принципах багаторівневої архітектури, що є стандартним та перевіреним підходом у розробці сучасних, високонавантажених вебзастосунків. Цей вибір був зумовлений прагненням забезпечити не лише поточну функціональність, але й довгострокову стійкість, масштабованість та гнучкість системи.

В основі архітектури лежить принцип чіткого функціонального розділення (Separation of Concerns), який дозволяє мінімізувати залежності між компонентами, забезпечуючи зниження рівня зв’язності (decoupling). Таке розділення підвищує прозорість внутрішніх механізмів, спрощує локалізацію та

виправлення помилок, а також дозволяє команді розробників автономно працювати над окремими рівнями без ризику впливу на інші частини системи.

Система логічно структурована у вигляді трьох основних рівнів:

- Рівень представлення (Frontend): Відповідає виключно за взаємодію з користувачем та візуалізацію даних. Реалізований за допомогою бібліотеки React, що дозволяє створити динамічний, швидкий та адаптивний клієнтський інтерфейс.
- Прикладний рівень (Backend / Business Logic Layer): Серце системи, що концентрує всю бізнес-логіку. Створений на базі фреймворку NestJS, він функціонує як уніфікований REST API, керуючи потоками даних, перевіркою валідності запитів, процесами авторизації, а також складними алгоритмами, як-от обчисленням рейтингу.
- Рівень даних (Data Layer): Призначений для надійного збереження та управління інформацією. Для цього використовується реляційна база даних PostgreSQL, яка гарантує цілісність та стійкість даних, доповнена шаром кешування на основі KeyDB для оптимізації доступу до часто запитуваних статистичних даних та таблиць лідерів.

Крім того, розгортальна архітектура передбачає інтеграцію з ізольованими зовнішніми сервісами для виконання критично важливих, але не профільних завдань. Зокрема, для безпечного виконання коду користувача застосовується Code Execution Service (PistonAPI), а автентифікація користувачів делегована надійному зовнішньому провайдеру Firebase Authentication. Така стратегія дозволяє зняти з основної платформи додаткове навантаження та мінімізувати вектори атак, що значно підвищує загальну безпеку та ефективність рішення.

2.5.1. Компоненти системи та їхня функціональна відповідальність

Кожен компонент системи виконує строго визначену роль, мінімізуючи зв'язність між модулями.

Рівень представлення забезпечується компонентом Frontend (React). Його єдине завдання – це забезпечення швидкого та адаптивного інтерфейсу

користувача (UI) і досвіду (UX), відповідаючи за візуалізацію навчального контенту, інтерактивні елементи та керування користувацьким сеансом. Всі запити до бізнес-логіки відбуваються через REST API.

Прикладний рівень являє собою Backend API на фреймворку NestJS, який є «мозком» системи. Він концентрує всю бізнес-логіку платформи: управління станом даних, валідацію запитів, авторизацію доступу до ресурсів та реалізацію алгоритму рейтингування. Серверна частина виступає єдиним посередником між клієнтом та сховищами даних, а також ініціює перевірку програмного коду.

Ключовим функціональним елементом є Code Execution Service (PistonAPI), який виступає як зовнішній ізольований сервіс для виконання коду, що надіслав студент. Це архітектурне рішення є свідомим вибором для спрощення інфраструктури, оскільки воно дозволяє відмовитися від складної внутрішньої Judge System (яка потребувала б налаштування Docker чи Kubernetes Sandbox). Таким чином, замість черг повідомлень (як-от RabbitMQ), використовується синхронний виклик PistonAPI, де Backend API передає код, очікує на результат і обробляє його, що значно підвищує стабільність системи.

Рівень даних складається з двох основних сховищ. PostgreSQL обрано як основну реляційну базу даних для персистентного зберігання профілів, завдань та структурованих результатів, гарантуючи цілісність та надійність інформації. Паралельно використовується Cache (KeyDB) як in-memory data store. Це сховище призначене для оперативної інформації, як-от сесії користувачів та динамічний рейтинг, що дозволяє мінімізувати навантаження на PostgreSQL та радикально підвищити швидкість відгуку системи, особливо під час пікових навантажень.

Система автентифікації (Auth Service) делегована зовнішньому сервісу Firebase Auth, що дозволяє забезпечити високий рівень безпеки та надійне шифрування токенів без необхідності супроводу власного сервісу авторизації.

2.5.2. Безпека виконання коду

Питання безпеки є центральним та пріоритетним для інтелектуальної навчальної платформи, оскільки система передбачає виконання довільного програмного коду, наданого користувачем. Такий функціонал, хоч і є критично важливим для освітнього процесу, несе у собі значні ризики безпеки. Несанкціоноване виконання коду може призвести до низки загроз, включаючи:

- Атаки типу "Відмова в обслуговуванні" (DoS): Запуск коду з нескінченними циклами або надмірним споживанням системних ресурсів (CPU, пам'яті).
- Витік даних (Data Leakage): Спроби отримати доступ до файлової системи або мережевих ресурсів, що належать серверу.
- Масштабування привілеїв: Використання вразливостей для отримання доступу до інших частин системи.

Для усунення цих загроз критично необхідним є забезпечення повної ізоляції середовища виконання (Sandbox). Пісочниця створює жорстко обмежене віртуальне середовище, у якому користувацький код не має можливості взаємодіяти з основною операційною системою чи базою даних.

Ця ізоляція досягається завдяки використанню зовнішнього спеціалізованого провайдера – PistonAPI. PistonAPI виступає як надійний ізольований сервіс (Code Execution Service), який:

- Приймає код від Backend API платформи.
- Запускає його в ізольованому контейнері, де доступ до зовнішніх ресурсів, окрім стандартного вводу/виводу, суворо обмежений.
- Застосовує ліміти на час виконання (timeout) та споживання пам'яті.
- Повертає результат (успіх, помилка компіляції чи виконання) назад на прикладний рівень.

Такий підхід до архітектури виконання коду забезпечує двояку перевагу: він мінімізує ризики безпеки та знімає з основної платформи необхідність підтримувати складну інфраструктуру віртуалізації або контейнеризації,

дозволяючи Backend-серверу зосередитися виключно на бізнес-логіці. Це є ключовим елементом надійності та стійкості системи.

2.5.3. Розгортальна архітектура

Система проєктувалася з урахуванням сучасних індустріальних стандартів та принципів контейнеризації. Для розгортання кожен логічний компонент платформи – Frontend (React), Backend (NestJS API), Cache (KeyDB) та База даних (PostgreSQL) – пакується в окремий Docker-контейнер. Цей підхід є фундаментальним для досягнення повної портативності та узгодженості середовищ між розробкою, тестуванням та продуктивним розгортанням.

Використання Docker надає переваги в ізоляції ресурсів, запобігаючи конфліктам залежностей, та забезпечує декларативне управління через файли Dockerfile і docker-compose.yml, що гарантує просте відтворення середовища та ефективне використання обчислювальних ресурсів.

Розгортальна архітектура є високогнучкою: мінімально необхідна конфігурація може бути розгорнута на єдиній віртуальній машині за допомогою Docker Compose, що ідеально підходить для середовищ розробки. Проте, архітектурно система повністю готова до оркестрації у хмарних середовищах (Kubernetes або Docker Swarm), де контейнери можуть бути автоматично масштабовані (горизонтально) та розподілені між кількома машинами.

Це забезпечує високу доступність (High Availability) та дозволяє проводити нарощування функціональності та оновлення системи без зупинки обслуговування (Zero-Downtime Deployment).

Таким чином, розгортальна архітектура системи закладає міцний фундамент для довгострокового зростання та операційної стійкості платформи. Діаграму розгортання наведено на рисунку 2.5.3.1.

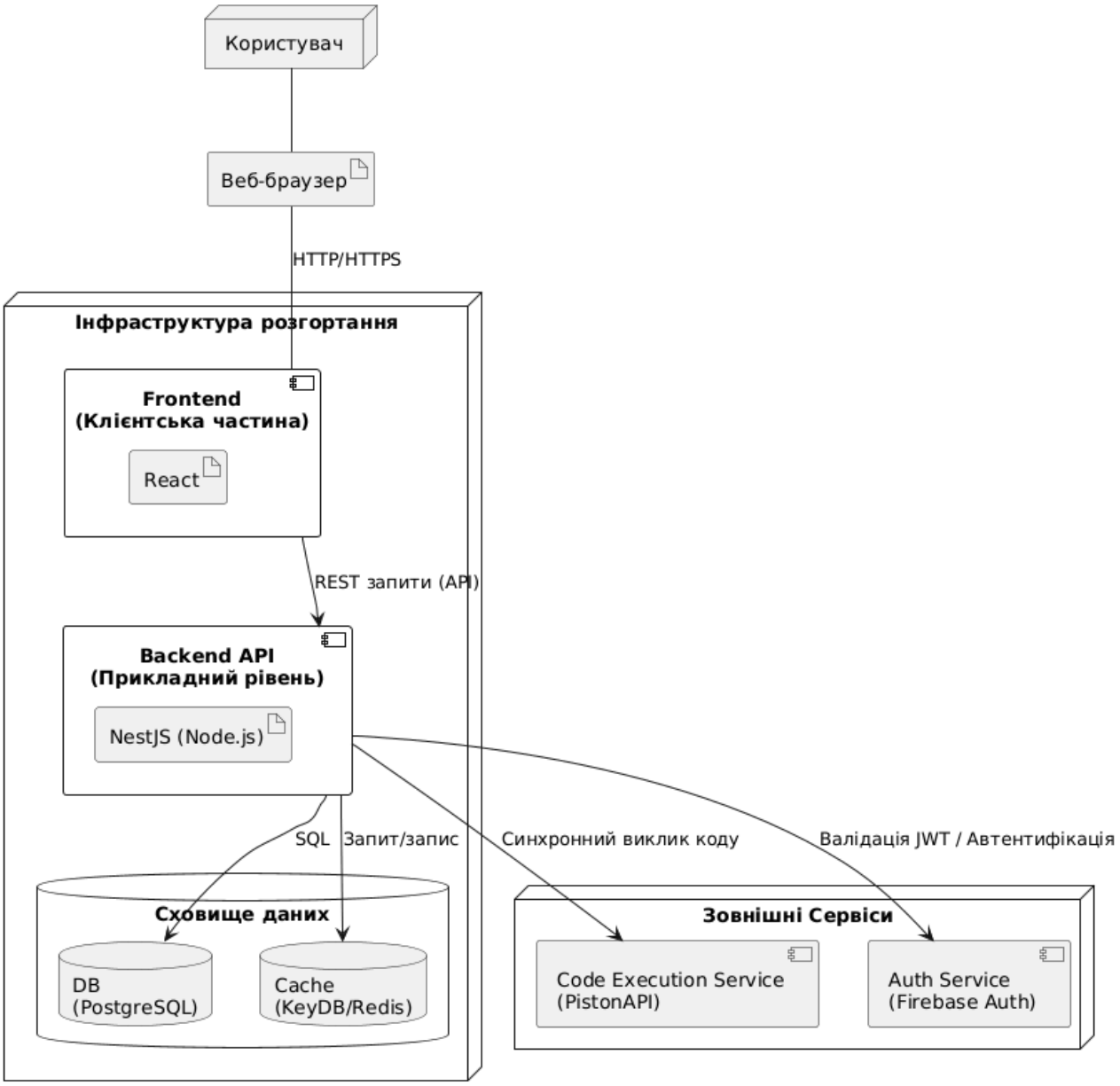

Рис. 2.5.3.1 Діаграма розгортання

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

3.1. Загальна архітектура системи

Розроблена інтелектуальна навчальна система базується на принципах багаторівневої архітектури, що передбачає чітке розділення функціональних елементів системи відповідно до їх призначення. Такий підхід дозволяє не лише забезпечити структурованість і прозорість внутрішніх механізмів взаємодії, а й закладає основу для подальшого масштабування без необхідності суттєвих змін у загальній структурі системи. Кожен структурний елемент платформи виконує строго визначену роль, що сприяє зниженню рівня зв’язності між компонентами та підвищує автономність розробки та оновлення окремих підсистем.

У рамках запропонованої архітектури можна виділити три ключові рівні:

1. Рівень представлення (Frontend) – відповідає за взаємодію з користувачем, відображення контенту, обробку введених даних та забезпечення інтуїтивного інтерфейсу.
2. Прикладний рівень (Backend / Business Logic Layer) – реалізує усі алгоритмічні процеси, керує логікою перевірки коду, визначає правила доступу, забезпечує контроль цілісності даних.
3. Рівень зберігання даних (Data Layer) – включає основну реляційну базу даних та шар кешування, що використовується для прискорення доступу до найбільш запитуваних даних, зокрема рейтингової інформації.

Взаємодія між рівнями здійснюється через універсальний протокол HTTP із використанням REST API, що забезпечує стандартизовану, передбачувану і незалежну від клієнтської платформи модель доступу до ресурсів.

Особливої уваги заслуговує те, що процес виконання програмного коду користувача не реалізується всередині серверної частини. Навпаки, система передає код до спеціалізованого зовнішнього сервісу – PistonAPI, який забезпечує запуск програм у ізольованому середовищі (sandbox). Таке рішення

мінімізує ризики, пов’язані з можливими зловмисними діями користувача, та водночас знімає з платформи необхідність підтримувати складну інфраструктуру віртуалізації або контейнеризації.

Таким чином, архітектура системи побудована таким чином, щоб:

- забезпечити гнучкість та адаптивність у випадку зміни вимог та підключення нових сервісів;
- створити умови для поетапного нарощування функціональності без зупинки системи;
- гарантувати стабільність і безпеку обробки користувацького коду;
- підтримувати розподіленість навантаження та оптимізовану обробку запитів.

Це дає можливість у перспективі розширити платформу як за кількістю користувачів, так і за різноманітністю навчального контенту, не переосмислюючи фундаментальні архітектурні рішення.

3.1.1. Компонентна архітектура

Серверна частина платформи реалізована на основі фреймворку NestJS, який підтримує модульну модель побудови застосунків, що дозволяє групувати взаємопов’язані компоненти у логічні блоки. Такий підхід значно спрощує розширення системи та її супровід, оскільки модуль може бути розроблений, модифікований або замінений незалежно від інших частин системи.

Нижче наведено основні модулі платформи та їх функціональне призначення.

1. Frontend-рівень (React/TypeScript):
   - Призначення: Відповідає за інтерфейс користувача (UI) та інтерактивні елементи.
   - Реалізація: Побудований на React та TypeScript, забезпечуючи компонентний підхід та високу надійність. Для адміністрування використано react-admin, що прискорює розробку панелі керування.
2. Backend API (NestJS):

- Призначення: Ядро системи. Реалізує бізнес-логіку, валідацію, управління даними та авторизацію.
- Реалізація: NestJS-сервер із REST ендпоінтами, що використовує архітектурний шаблон MVC (Model-View-Controller) та принципи об'єктно-орієнтованого програмування.

3. Execution Service (PistonAPI):
   - Призначення: Зовнішній API для безпечного виконання користувацького коду в ізольованому середовищі (Sandbox).
   - Обґрунтування: Замінює внутрішній Execution Sandbox, що спрощує інфраструктуру та перекладає відповідальність за ізоляцію на спеціалізований сервіс.
4. База даних (PostgreSQL):
   - Призначення: Центральне сховище структурованих даних (користувачів, завдань, результатів, статистики).
   - Особливості: Обрано за її надійність, підтримку складних реляційних зв'язків та функціонал JSONB для зберігання неструктурованих даних (тести, метадані профілів).
5. Cache (KeyDB):
   - Призначення: Використовується для високошвидкісного доступу до даних.
   - Використання: Кешування сесій, проміжних результатів та зберігання динамічного leaderboard (рейтингу), що мінімізує навантаження на PostgreSQL.
6. Auth-сервіс (Firebase Auth):
   - Призначення: Єдина, надійна система автентифікації, що забезпечує вхід через сторонні провайдери.

3.2. Взаємодія компонентів та організація внутрішніх процесів системи

Взаємодія складових частин розробленої вебплатформи організована таким чином, щоб забезпечити цілісність інформаційних потоків, узгодженість виконання навчальних сценаріїв та стабільність роботи системи навіть за умов підвищеного навантаження. Важливим аспектом функціонування системи є не лише наявність окремих функціональних модулів, а й спосіб їх координації та узгодження дій, оскільки саме це визначає якість користувацького досвіду та надійність роботи застосунку.

У процесі взаємодії клієнтської частини з серверною ініціюються послідовні запити, на які сервер формує відповіді, що базуються на стані поточних даних, результатах попередньої діяльності користувача та визначених правилах логічного переходу між етапами навчання. Таким чином, система реалізує станоорієнтовану модель навчальної взаємодії, де кожен крок користувача має узгоджене продовження.

Узагальнений механізм обробки запиту відбувається за такою послідовністю:

1. Формування запиту користувачем у клієнтському інтерфейсі.
2. Передача запиту на серверну частину через REST API із зазначенням ідентифікаційного маркера автентифікації.
3. Валідація автентичності та коректності запиту, що виключає можливість несанкціонованого доступу.
4. Звернення до бази даних або кешу залежно від характеру запитаних даних.
5. Формування структурованої відповіді із урахуванням поточного навчального статусу користувача.
6. Повернення відповіді на клієнтську частину та її відображення у зручній для сприйняття формі.

Особливої уваги потребує аспект послідовності та ієрархії доступу. Платформа не дозволяє користувачеві переходити до наступного розділу або

складнішого завдання за відсутності підтвердженого успішного виконання попереднього. Така модель забезпечує поступальне формування знань, що відповідає дидактичним принципам навчання та сприяє усвідомленому засвоєнню матеріалу.

З метою оптимізації навантаження на сервер частина даних, які часто запитуються багатьма користувачами (наприклад, рейтингові таблиці), зберігається в шарі кешування. Це дає змогу значно скоротити час відповіді та підвищити масштабованість платформи при збільшенні кількості користувачів.

Таким чином, внутрішня логіка взаємодії компонентів системи побудована за принципами:

- розділення відповідальностей;
- мінімізації зв’язності компонентів;
- пріоритету контролю доступу та цілісності даних;
- оптимізації інформаційних потоків за частотою використання даних.

Це забезпечує надійність, керованість та розширюваність системи у процесі подальшої експлуатації.

3.3. Механізм автоматичної перевірки програмного коду

Механізм автоматизованої перевірки розв’язань у створеній навчальній платформі побудований таким чином, щоб забезпечити об’єктивність оцінювання результатів, незалежність процесу контролю від людського чинника та можливість багаторазового виконання тестів у просторі, ізольованому від основного серверного середовища. Такий підхід дозволяє реалізувати самостійну роботу користувача з можливістю негайного отримання розгорнутого зворотного зв’язку щодо правильності та повноти виконаного програмного рішення.

Основна логіка перевірки реалізована на клієнтській стороні додатку у компоненті CodeTaskPage. Саме на цьому рівні оброблюється структура тестових даних, формується спеціальний виконуваний код для надіслання на зовнішній сервіс та здійснюється аналіз результатів виконання. Серверна

частина у даному процесі виступає насамперед джерелом навчального завдання та тестових прикладів.

3.3.1. Етап 1. Отримання та нормалізація тестових даних

Після завантаження сторінки компонент здійснює запит до серверної частини з метою отримання структури завдання, яка включає:

1. текстове формулювання задачі;
2. набір тестових прикладів;
3. початковий шаблон коду;
4. перелік підказок.

Оскільки тестові приклади можуть бути збережені на сервері в різних форматах (рядки, масиви або об’єкти), на клієнтському рівні здійснюється процедура нормалізації, що переводить вхідні параметри та очікуваний результат у єдину уніфіковану модель. Така уніфікація забезпечує можливість подальшої автоматизованої обробки незалежно від формату, у якому викладач сформував тест.

3.3.2. Етап 2. Побудова середовища виконання

Безпосередньо перед перевіркою рішення система формує генерований фрагмент коду, у якому:

1. розміщується вихідна функція користувача без змін;
2. додається масив тестових прикладів;
3. для кожного тесту здійснюється виклик функції з набором параметрів;
4. результат виконання порівнюється із очікуваним значенням.

Усі результати тестування накопичуються у структурі results, яка в кінці виконання виводиться у стандартний потік логів у вигляді JSON-даних із маркером TEST_RESULTS:. Саме цей маркер є ключем для подальшого аналізу.

3.3.3. Етап 3. Віддалене виконання програмного коду

Сформований код надсилається через HTTP-запит до сервісу PistonAPI, який забезпечує виконання програм у ізольованому контейнері. Ізоляція гарантує:

- неможливість несанкціонованого доступу до системних ресурсів;
- захист платформи від шкідливих або рекурсивно-блокуючих програм;
- обмеження часу виконання, що запобігає зависанням.

Таким чином, система може безпечно обробляти довільний код, введений користувачем.

3.3.4. Етап 4. Обробка результатів та формування зворотного зв'язку

Після отримання відповіді від сервісу виконання, клієнтська частина аналізує вивід і за допомогою регулярного виразу виділяє блок TEST_RESULTS:[...]. Отримані дані парсяться у структуру JavaScript-об'єктів, де для кожного тесту міститься:

- набір вхідних параметрів;
- очікуваний результат;
- фактичний результат роботи програми;
- оцінка правильності виконання (passed: true/false);
- текстове повідомлення про помилку (за наявності).

На основі цієї інформації формується розгорнуте текстове повідомлення, яке надається користувачеві у тому ж інтерфейсі, де редагується код. Такий підхід дозволяє не лише оцінити правильність, але й виявити джерело помилки шляхом аналізу розбіжностей між очікуваним та фактичним результатами.

3.3.5. Етап 5. Фіксація результату виконання завдання

У випадку, коли всі тестові приклади виконані успішно, система:

- позначає завдання як виконане на рівні інтерфейсу;
- здійснює запит на серверну частину з метою фіксації прогресу користувача;

- надає можливість перейти до наступного навчального модуля.

Таким чином, автоматична перевірка поєднується з механізмом контролю послідовності навчального процесу.

3.4. Розробка клієнтської частини та організація користувацької взаємодії

Клієнтська частина вебплатформи є ключовим елементом, що визначає зручність роботи користувача, якість взаємодії з навчальним контентом і загальну ефективність засвоєння матеріалу. В основі реалізації інтерфейсу лежить концепція інтерактивної навчальної взаємодії, коли студент не лише спостерігає за поданими матеріалами, але й активно взаємодіє з ними під час виконання практичних завдань.

Фронтенд-частина реалізована із застосуванням бібліотеки React, що дає змогу будувати інтерфейс за компонентною моделлю. Компонентний підхід сприяє модульності та повторному використанню елементів, а також забезпечує чітке розділення логіки відображення та поведінки. Завдяки цьому окремі частини інтерфейсу (опис завдання, редактор коду, підказки, тестові приклади тощо) можуть оновлюватися незалежно одна від одної, не порушуючи цілісності роботи сторінки.

3.4.1. Компонованість і структура інтерфейсу

У межах сторінки розв’язання практичного завдання (CodeTaskPage) інтерфейс поділений на декілька функціональних областей:

- область опису завдання, яка містить розгорнутий текстовий матеріал із постановкою проблеми;
- редактор програмного коду, що дозволяє студентові вводити й редагувати рішення;
- область демонстрації тестових прикладів, яка допомагає зорієнтуватися у структурі вхідних та вихідних даних;

- секція підказок, що формує підтримку навчальної діяльності без розкриття готового розв’язку;
- блок виведення результатів, який демонструє підсумок автоматичної перевірки.

Таке композиційне групування забезпечує змістову завершеність кожного етапу взаємодії та підтримує принцип поступовості навчання.

3.4.2. Використання інтерактивного редактора програмного коду

Центральним елементом клієнтського інтерфейсу є редактор коду, реалізований на основі Monaco Editor – того ж компонента, який використовується в середовищі Visual Studio Code. Завдяки цьому редактор підтримує:

- підсвічування синтаксису;
- підказки під час введення коду;
- форматування та вирівнювання;
- автоматичне закриття дужок і структурних блоків.

Застосування такого інструменту суттєво підвищує зручність роботи та забезпечує поступове занурення у професійне середовище, що є особливо важливим при підготовці студентів до роботи з реальними інструментами індустрії.

3.4.3. Адаптивність та доступність

Інтерфейс платформи побудований з урахуванням ключових принципів адаптивного дизайну (Responsive Web Design). Це гарантує, що всі навчальні матеріали, інтерактивні елементи та функціональні компоненти коректно відображаються та повністю функціонують незалежно від типу пристрою, який використовує студент: ноутбук, планшет чи мобільний телефон. Адаптивність інтерфейсу забезпечує неперервність навчального процесу в різних умовах використання, дозволяючи студентам ефективно працювати поза стаціонарним робочим місцем. Застосування гнучких сіток, адаптивних зображень та

медіа-запитів (media queries) дозволяє системі динамічно перебудовувати макет, оптимально використовуючи доступний простір екрана. Таким чином, забезпечується рівний доступ до навчального контенту та функціоналу, що є важливою передумовою для інклюзивного та ефективного онлайн-навчання.

3.4.4. Принципи організації зворотного зв'язку

Особлива увага приділена прозорості, деталізації та структурованості результатів перевірки завдань. Зворотний зв'язок (feedback) реалізований як ключовий інструмент усвідомленого навчання. Після завершення виконання завдань, які вимагають перевірки коду, студент отримує розгорнутий звіт, що включає такі критично важливі елементи:

- Перелік виконаних тестів для перевірки рішення.
- Вхідні дані, які були подані на вхід програми студента.
- Очікуваний результат, який мала повернути коректно написана програма.
- Отриманий результат, який повернула програма студента.

Такий підхід формує усвідомлене навчання та стимулює студента не просто знайти випадкове правильне рішення, а глибоко зрозуміти логіку виконання програми, виявити та виправити помилку, аналізуючи розбіжності між очікуваним і фактичним виводом.

3.4.5. Поведінкова логіка інтерфейсу

Поведінкова логіка інтерфейсу розроблена для забезпечення плавного, інтуїтивно зрозумілого та неперервного навчального процесу. Успішне завершення завдання автоматично ініціює зміну стану інтерфейсу:

- Динамічне оновлення: У випадку, коли всі тестові приклади виконано успішно і рішення верифіковано, кнопка надсилання рішення автоматично замінюється на елемент, що пропонує перехід до наступних навчальних модулів або завдань.
- Візуальний маркер: Саме завдання позначається як виконане (наприклад, зеленою позначкою).

Це забезпечує систематичне продовження навчання, усуває необхідність ручного контролю прогресу з боку користувача та створює позитивне підкріплення, стимулюючи студента до подальшої взаємодії з платформою. Логіка мінімізує когнітивне навантаження та підтримує високий рівень залученості (engagement).

3.5. Розробка серверної частини та забезпечення функціональної цілісності системи

Розробка серверної частини (Backend) є фундаментальною складовою архітектури програмного забезпечення, оскільки саме вона забезпечує надійну, безпечну та ефективну обробку даних, керування бізнес-логікою і координацію взаємодії між усіма компонентами системи. На відміну від клієнтської частини, яка орієнтована на візуальне представлення інформації та безпосередню взаємодію з користувачем, серверна частина є прихованим ядром, що відповідає за виконання найважливіших процесів – від автентифікації до збереження даних і валідації запитів. Саме від якості її реалізації залежить стабільність і цілісність усієї інформаційної системи.

3.5.1. Архітектурні принципи побудови Backend

Під час розроблення серверної частини головним завданням є забезпечення її масштабованості, безпеки та відповідності сучасним стандартам веброзробки. Для цього серверна логіка реалізована з урахуванням принципів інверсії залежностей (Dependency Injection). Такий підхід дозволяє ізолювати бізнес-логіку від технічних деталей реалізації, що суттєво спрощує розширення системи та тестування окремих модулів.

Ключовим аспектом при розробці серверної частини стало впровадження модульної архітектури на базі NestJS, побудованої поверх середовища Node.js з використанням TypeScript. Така технологічна комбінація обґрунтована необхідністю створення гнучкого, асинхронного й надійного середовища, здатного обробляти велику кількість запитів у режимі реального часу. Модульна

структура NestJS забезпечує чітке логічне розділення функціональних блоків, зокрема модулів автентифікації, управління користувачами, взаємодії з базою даних, обробки навчального контенту та механізмів перевірки коду.

3.5.2. Реалізація бізнес-логіки та функціональності

Ядро серверної частини зосереджено на реалізації бізнес-логіки системи, яка визначає її поведінку у відповідь на дії користувачів. Сюди входять такі ключові компоненти:

- Управління життєвим циклом користувача (Authentication & Authorization) реалізовано на основі токенів JWT (JSON Web Token), що гарантують безпечний обмін даними між клієнтом і сервером. Це забезпечує можливість гнучкого контролю доступу до ресурсів платформи. Для підтримки надійної автентифікації інтегровано механізми взаємодії з Firebase Authentication, який підтримує багатофакторний вхід (через email, Google або телефон).
- Сервер виконує найбільш ресурсоємні завдання, які вимагають складних обчислень або доступу до бази даних. Зокрема, це формування результатів тестування, фіксація прогресу користувача, перерахунок рейтингу, валідація вхідних даних і виконання запитів до зовнішніх сервісів. Завдяки застосуванню асинхронної моделі Node.js, такі операції виконуються паралельно без блокування основного потоку, що забезпечує високу швидкодію системи навіть при значному навантаженні.
- Для зберігання даних використовується реляційна база PostgreSQL, яка відзначається стабільністю, підтримкою транзакцій та гнучкістю у побудові складних запитів. Зв’язок із базою реалізовано через ORM-бібліотеку TypeORM. Це зменшує кількість помилок під час роботи з даними та прискорює розробку. Для підвищення продуктивності доступу до часто запитуваних даних, таких як таблиці рейтингу чи статистика виконання завдань, застосовується шар кешування на основі KeyDB.

- На рівні контролерів реалізовано багаторівневу валідацію даних із використанням клас-декораторів NestJS. Це дозволяє відхиляти потенційно шкідливі або некоректні запити ще до передачі їх у бізнес-логіку.

3.5.3. Розробка API та забезпечення взаємодії

Комунікація між клієнтською частиною та сервером реалізована через стандартизований RESTful API, який забезпечує незалежність клієнтських застосунків від серверної реалізації. API має чітку структуру, де кожен ресурс системи має свій ендпоінт і підтримує набір HTTP-методів (GET, POST, PUT, DELETE) відповідно до стандартів REST.

Особливу увагу приділено принципу Statelessness – сервер не зберігає стан клієнта між запитами, що дозволяє легко масштабувати систему горизонтально, додаючи нові інстанси без ризику втрати даних. Усі дані автентифікації передаються у вигляді токенів, що зберігаються на клієнті.

Крім того, реалізовано централізовану систему обробки помилок, яка повертає стандартизовані відповіді у форматі JSON. Це дозволяє клієнтській частині передбачувано реагувати на будь-які виняткові ситуації, а розробникам – швидко ідентифікувати джерело проблеми.

3.5.4. Забезпечення функціональної цілісності системи

Під функціональною цілісністю розуміється стабільна взаємодія між усіма компонентами системи – сервером, клієнтом, базою даних і зовнішніми сервісами – без порушення логічних зв’язків або втрати даних. Для її досягнення впроваджено такі механізми:

Використання єдиної моделі даних – усі сутності системи визначені у вигляді типізованих класів, що забезпечує узгодженість між клієнтським та серверним кодом.

Логічна цілісність транзакцій. Операції, які змінюють кілька пов’язаних сутностей (наприклад, створення користувача та запис прогресу), виконуються у межах транзакцій PostgreSQL.

Обробка одночасних запитів. NestJS ефективно керує асинхронністю, що дозволяє системі підтримувати десятки одночасних клієнтів без конфліктів у доступі до даних.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ, ТЕСТУВАННЯ ТА ОБҐРУНТУВАННЯ АРХІТЕКТУРНИХ РІШЕНЬ

4.1. Програмні та апаратні вимоги

4.1.1. Програмні та апаратні вимоги до розгорнутої системи

Перехід від етапу проєктування до реальної імплементації вимагає детального визначення системних вимог, що забезпечують не лише функціональну коректність, але й стабільну експлуатацію розробленої вебплатформи. Для гарантування надійної працездатності системи на всіх етапах її життєвого циклу, від внутрішнього тестування до демонстрації, був сформований наступний вичерпний перелік мінімальних апаратних та програмних вимог.

Визначення цих вимог ґрунтувалося на комплексному аналізі необхідної продуктивності для ефективної обробки складних запитів (особливо тих, що стосуються алгоритмів рейтингування та великих обсягів навчального контенту) та обов'язкової економічної доцільності. Мінімізація витрат при збереженні високої ефективності є ключовим показником раціонального інженерного підходу.

На основі цього аналізу, системні вимоги доцільно класифікувати за трьома основними напрямками: серверна частина, клієнтська частина та середовище розробки, що відображено у зведеній таблиці 4.1.1.1.

Таблиця 4.1.1.1

Апаратні вимоги до розгорнутої системи

Тип вимоги	Компонент	Мінімальні апаратні вимоги	Необхідне програмне забезпечення
Серверна частина	Сервер застосунку та БД	2 vCPU, 4 GB RAM, 50 GB SSD	Операційна система (Linux, наприклад, Ubuntu Server v20+), Runtime Environment (Node.js v18+), Сервер БД (PostgreSQL, KeyDB).
Клієнтська частина	Комп'ютер кінцевого користувача	2-ядерний процесор, 4 GB RAM	Web-браузер (Google Chrome v100+ або Firefox v90+), Стабільне підключення до мережі Інтернет.
Середовище розробки	Локальна машина розробника	4-ядерний процесор, 8 GB RAM	Операційна система (Windows/macOS/Linux), IDE (VS Code), Система контролю версій (Git), Docker.

4.1.2. Обґрунтування мінімальної конфігурації

Мінімальна апаратна конфігурація системи визначається специфікою архітектури вебплатформи, що включає серверну частину на Node.js, реляційну базу даних PostgreSQL, кешуючий механізм на основі KeyDB та інтеграцію з зовнішнім сервісом віддаленого виконання програмного коду (PistonAPI). Для коректної роботи всіх компонентів необхідно забезпечити ресурсний баланс між процесорною потужністю, обсягом оперативної пам’яті та стабільністю файлової системи.

Вибір конфігурації 2 vCPU та 4 GB RAM обумовлений тим, що серверна логіка реалізована на платформі Node.js, яка активно використовує асинхронність і подієву модель, тому не потребує великої кількості процесорних потоків, але вимагає стабільного CPU-часу для обробки паралельних запитів та асинхронних викликів. На практиці цього достатньо для одночасного запуску бекенду та СУБД PostgreSQL на одній віртуальній машині

без значних затримок у відповідях навіть під навантаженням у вигляді десятків паралельних клієнтських запитів. Збереження мінімального рівня RAM (4 GB) виправдане необхідністю підтримувати постійно активний пул з'єднань з базою даних, кешувати найчастіші операції у KeyDB та забезпечувати стабільне виконання Node.js-процесів.

Додатковим аргументом на користь вибраної конфігурації є використання контейнеризації (Docker), яка потребує окремих ресурсів для запуску ізольованих контейнерів, але при цьому дозволяє оптимізувати розподіл процесорного часу та пам'яті. Конфігурація у 4 GB RAM є мінімальною межею, за якої можливо одночасно розгорнути контейнер серверної частини, контейнер з PostgreSQL та допоміжні сервіси, не створюючи надмірного свопінгу та деградації продуктивності.

Файловий простір обсягом 50 GB SSD обрано з урахуванням потреб зберігання навчального контенту, транзакційних даних, структурованих таблиць прогресу користувачів, а також логів API та службових файлів системи. Використання SSD забезпечує низьку латентність при роботі з індексами PostgreSQL, що важливо для швидких вибірок даних у модулях рейтингування та трекінгу прогресу.

Окремо варто підкреслити роль вимог до програмного середовища. Використання Node.js v18+ пояснюється потребою підтримки сучасних стандартів JavaScript та коректної роботи асинхронних процесів. СУБД PostgreSQL забезпечує транзакційну цілісність даних, що є критично важливим для операцій фіксації результатів виконання завдань і ведення рейтингових таблиць. KeyDB, як in-memory механізм, дозволяє компенсувати невисоку апаратну конфігурацію за рахунок зменшення навантаження на основну базу даних через кешування повторюваних запитів.

Таким чином, мінімальна конфігурація 2 vCPU / 4 GB RAM / 50 GB SSD є обґрунтованим компромісом між достатнім рівнем продуктивності, економічною доцільністю та вимогами до надійної експлуатації системи. Вона забезпечує стабільну роботу всіх ключових компонентів вебплатформи,

підтримує можливість подальшого масштабування та повністю відповідає цільовому сценарію використання в рамках кваліфікаційної роботи.

4.2. Економічне обґрунтування впровадження системи

4.2.1. Обґрунтування масштабу розробки та трудомісткість

Перехід від успішної демонстрації функціональності системи до її економічного обґрунтування є критично важливим етапом, оскільки він дозволяє оцінити масштаб виконаних робіт, визначити їхню внутрішню вартість та підтвердити раціональність інвестицій часу і ресурсів. В контексті кваліфікаційної роботи, економічне обґрунтування фокусується не стільки на фінансових показниках, скільки на оцінці трудомісткості та підтвердженні значущості розробленого рішення.

Для визначення внутрішнього масштабу проєкту було застосовано метод експертної оцінки трудомісткості, при якому загальний обсяг робіт поділяється на ключові етапи життєвого циклу розробки програмного забезпечення (див. табл. 4.2.1.1), що дозволяє прозоро і системно відобразити, які саме фази проєкту були найбільш ресурсоємними.

Таблиця 4.2.1.1

Етапи життєвого циклу розробки системи

Етап життєвого циклу	Опис основних робіт	Оцінка в людино-годинах (люд.-год)	Відсоток від загального обсягу
I. Аналіз та Проєктування	Системний аналіз предметної області (Розділ 1), розробка моделей (UML, Діаграма класів), синтез алгоритмів.	180	34.3%

II. Розробка програмного забезпечення	Кодування Front-end та Back-end функціоналу, інтеграція СУБД, налагодження API.	250	47.6%
III. Тестування та Верифікація	Функціональна демонстрація, усунення помилок, оформлення результатів та документації (Розділ 4).	95	18.1%
Загальна трудомісткість	–	525	100%

Як видно з представленої таблиці, загальний обсяг робіт, витрачений на виконання кваліфікаційної роботи, оцінюється у 525 людино-годин. Ця цифра чітко підтверджує значний масштаб та складність проєкту, причому левова частка зусиль (47.6%) була спрямована безпосередньо на Розробку програмного забезпечення. Це свідчить про високий ступінь практичної реалізації та значну інженерну роботу.

Зі стратегічної точки зору, створення власного рішення забезпечує повну інтелектуальну власність та виняткову гнучкість у подальшій адаптації та масштабуванні. На відміну від придбання комерційних ліцензій, яке часто супроводжується прив'язкою до сторонніх розробників та обмеженнями функціоналу, власне рішення дозволяє негайно реагувати на зміни вимог та інтегрувати будь-які специфічні функції, що є критично важливим для освітньої чи вузькопрофільної платформи.

4.2.2. Обговорення отриманих результатів та перспективи розвитку системи

Підбиваючи підсумки всієї роботи, можна констатувати, що мета кваліфікаційної роботи, сформульована у Вступі, була повністю досягнута та підтверджена результатами демонстрації у Розділі 4.2.

Незважаючи на успішну реалізацію, для забезпечення наукової об'єктивності необхідно визначити і поточні обмеження реалізованої системи. Серед них слід виділити необхідність подальшої оптимізації запитів до бази даних для роботи з надзвичайно великими обсягами даних (наприклад, понад 100 тис. користувачів) та відсутність розширених інструментів аналітики та моніторингу навантаження у поточному інтерфейсі адміністратора.

Ці обмеження безпосередньо визначають перспективи подальшого розвитку проєкту, що демонструє його потенціал і життєздатність за межами академічної роботи. Пріоритетними напрямками для майбутнього вдосконалення є:

1. Масштабування та відмовостійкість: Перехід на кластерну архітектуру БД або впровадження мікросервісів для гарантування високої доступності та стійкості до відмов при експоненційному зростанні кількості користувачів.
2. Розширена аналітика: Інтеграція спеціалізованих інструментів бізнес-аналізу (BI) та створення модуля детальних звітів, які дозволять адміністраторам отримувати глибоке розуміння ефективності навчальних матеріалів та динаміки прогресу користувачів.
3. Вдосконалення користувацького досвіду (UX): Розширення функціоналу елементами гейміфікації та розробка нативних мобільних застосунків, що сприятиме підвищенню залученості та спростить доступ до платформи.

Таким чином, розроблена система є повноцінним, функціональним та економічно обґрунтованим рішенням, яке не лише відповідає поставленим цілям, але й має чіткий вектор для подальшого комерційного чи наукового розвитку.

ВИСНОВКИ

У магістерській роботі проведено комплексне дослідження принципів побудови інтелектуальної вебплатформи для навчання програмуванню, спрямоване на вдосконалення процесу взаємодії користувача з навчальним контентом, обробки даних та організації рейтингової оцінки результатів навчання.

У ході дослідження:

Проаналізовано особливості побудови графічного інтерфейсу та обґрунтовано доцільність використання компонентного підходу (React). Визначено, що ключовими факторами ефективності є зниження когнітивного навантаження, забезпечення інтуїтивності та адаптивності інтерфейсу для різних пристроїв (Responsive Web Design).

Досліджено методи побудови рейтингових таблиць та обґрунтовано архітектурні засоби для підтримки ефективного методу навчання, який поєднує модульність, динамічний зворотний зв'язок і гейміфікаційні механіки. Показано, що рейтингова система, реалізована з використанням кешування (KeyDB), може виконувати не лише мотиваційну, а й високопродуктивну аналітичну функцію.

Розглянуто принципи адаптації архітектури навчальної вебплатформи до масштабування. Встановлено, що застосування трирівневої REST-архітектури у поєднанні з контейнеризацією (Docker) та готовністю до оркестрації забезпечує масштабованість, стабільність і кросплатформність системи. Обґрунтовано роль зовнішніх сервісів автентифікації та виконання коду (PistonAPI) як факторів, що підвищують безпеку та знижують інфраструктурну складність.

Виконано аналіз методів обробки даних для збереження та управління навчальною вибіркою. Обґрунтовано ефективність застосування ORM-технологій (TypeORM) для роботи з реляційною базою даних PostgreSQL, а також використання механізмів валідації даних на прикладному рівні.

Наголошено на важливості контролю цілісності даних та застосуванні кешування (KeyDB) для забезпечення високої швидкості доступу до аналітичних результатів.

У результаті проведених досліджень сформовано узагальнену архітектурно-методичну модель побудови сучасних навчальних вебплатформ, орієнтованих на інтерактивність, персоналізацію та об’єктивну оцінку знань користувача.

Отримані результати мають прикладне значення для подальшого розвитку систем електронного навчання, зокрема у частині створення інтелектуальних освітніх середовищ із миттєвим зворотним зв’язком і гейміфікованими елементами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. VarenyaZ. The ultimate guide to UI/UX design best practices for education. *VarenyaZ*. URL: https://varenyaz.com/the-ultimate-guide-to-ui-ux-design-best-practices-for-education/ (дата звернення: 05.11.2025).
2. Design D. UI/UX design for education: enhancing learning experiences online. *Medium*. URL: https://devoq.medium.com/ui-ux-design-for-education-enhancing-learning-experiences-online-8facc531bbc8 (date of access: 05.11.2025).
3. Leaderboard design principles to enhance learning and motivation in a gamified educational environment: development study - pubmed. *PubMed*. URL: https://pubmed.ncbi.nlm.nih.gov/33877049 (дата звернення: 05.11.2025).
4. Garcia-Iruela M., Hijón-Neira R., Connolly C. Analysis of three methodological approaches in the use of gamification in vocational training. *MDPI*. 2021. Vol. 12, no. 8. Article 300. URL: https://www.mdpi.com/2078-2489/12/8/300 (дата звернення: 05.11.2025).
5. Responsive web design - Learn web development | MDN. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/CSS_layout/Responsive_Design (дата звернення: 06.11.2025).
6. Components and props – react. *React – A JavaScript library for building user interfaces*. URL: https://legacy.reactjs.org/docs/components-and-props.html (date of access: 06.11.2025).
7. Documentation | NestJS - A progressive Node.js framework. *Documentation | NestJS - A progressive Node.js framework*. URL: https://docs.nestjs.com/ (дата звернення: 06.11.2025).

8. Engineer-man/piston: a high performance general purpose code execution engine. *GitHub*. URL: https://github.com/engineer-man/piston (дата звернення: 06.11.2025).
9. Firebase authentication. *Firebase*. URL: https://firebase.google.com/docs/auth (дата звернення: 06.11.2025).
10. Henro. Markdown heeft mijn manier van schrijven veranderd. *Data-Pro BV*. URL: https://www.data-pro.nu/markdown-heeft-mijn-manier-van-schrijven-veranderd/ (дата звернення: 08.11.2025).
11. TypeScript: Переваги та Використання в Україні – UA – Galaktica. *UA – Galaktica*. URL: https://galaktica.io/blog/typescript-tse/ (дата звернення: 08.11.2025).
12. Stainless - Why Is TypeScript Good? Benefits, Features and Use Cases. *Stainless*. URL: https://www.stainless.com/sdk-api-best-practices/why-is-typescript-good-benefits-features-and-use-cases (date of access: 08.11.2025).
13. React-admin - Key Concepts. *Marmelab - Digital innovation studio*. URL: https://marmelab.com/react-admin/Architecture.html (date of access: 08.11.2025).
14. Nest js: основні характеристики та переваги фреймворку. *FoxmindEd*. URL: https://foxminded.ua/nest-js/ (дата звернення: 08.11.2025).
15. PostgreSQL: About. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/about/ (date of access: 08.11.2025).
16. KeyDB: High-Performance, Scalable, and Open-Source Solution. *OctaByte Blog*. URL: https://blog.octabyte.io/posts/databases/keydb/keydb-high-performance-scalable-and-open-source-solution/ (date of access: 08.11.2025).

17. Makhene T. Benefits of sandboxing. *HIPAA compliant email - Paubox*. URL: https://www.paubox.com/blog/benefits-of-sandboxing (date of access: 08.11.2025).
18. Jon Welling. What is Firebase Authentication?. *CBT Nuggets*. URL: http://www.cbtnuggets.com/blog/technology/programming/what-is-firebase-authentication (date of access: 08.11.2025).
19. Rational Software Architect. *IBM*. URL: https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-use-case (date of access: 09.11.2025).
20. DBMS - ER Model Basic Concepts. *Free Tutorials on Technical and Non Technical Subjects*. URL: https://www.tutorialspoint.com/dbms/er_model_basic_concepts.htm (date of access: 09.11.2025).