МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп'ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ НАВЧАННЯМ (LMS)

Виконав: студент групи 1П-22
Спеціальності
121 Інженерія програмного забезпечення
Ілля ТЕТЬОРА
Керівник:
Валентин ДМИТРЮК

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО

«_____» _____ 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Тетьорі Іллі Сергійовичу

1. Тема кваліфікаційної роботи Проєктування та реалізація системи управління навчанням (LMS)

Керівник роботи Дмитрюк Валентин Віталійович, викладач

затверджені наказом закладу фахової передвищої освіти

від «06» жовтня 2025 року № 84/1-у.

2. Строк подання студентом кваліфікаційної роботи 02.06.2026

3. Вихідні дані до кваліфікаційної роботи: мова програмування Ruby та фреймворк Ruby on Rails, об'єктно-реляційна СКБД PostgreSQL, бібліотека побудови інтерфейсів React.js, технологія двоспрямованого зв'язку реального часу ActionCable (WebSocket), система асинхронної обробки фонових завдань Sidekiq із сховищем Redis, екосистема автентифікації Devise та OmniAuth, фреймворк автоматизованого тестування RSpec, мова опису діаграм UML (діаграми прецедентів, класів та діяльності).

4. Зміст кваліфікаційної роботи: аналіз предметної області та огляд наявних рішень (роль і місце систем управління навчанням, порівняльний аналіз наявних платформ, аналіз технологій розроблення серверної частини, постановка задачі на розроблення програмного продукту); проєктування та програмна реалізація системи (формування вимог до програмного

забезпечення, проєктування архітектури та бази даних, програмна реалізація серверної логіки й автентифікації, реалізація взаємодії в реальному часі та асинхронної обробки); тестування, контроль якості та розгортання (стратегії й середовища тестування, оцінювання якості та зручності використання, розгортання застосунку та налаштування CI/CD).

5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Аналіз предметної області та огляд наявних рішень	09.12.2025	
3	Розділ 2. Проєктування та програмна реалізація системи	10.03.2026	
4	Розділ 3. Тестування, контроль якості та розгортання	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студент _____ Ілля ТЕТЬОРА

Керівник роботи _____ Валентин ДМИТРЮК

АНОТАЦІЯ

Кваліфікаційну роботу присвячено актуальній проблемі автоматизації процесів дистанційного та змішаного навчання шляхом проєктування й розроблення сучасної системи управління навчанням (LMS). Головним результатом роботи є створення повнофункціонального full-stack вебзастосунку, що охоплює повний навчальний цикл: реєстрацію користувачів із розподілом ролей, створення та наповнення курсів, проходження уроків і тестів, автоматичне оцінювання та обмін повідомленнями в реальному часі.

У роботі детально проаналізовано та програмно реалізовано серверну архітектуру на базі фреймворку Ruby on Rails із застосуванням нормалізованої реляційної бази даних PostgreSQL. Суть запропонованого рішення полягає у винесенні бізнес-логіки в окремий сервісний шар на основі патерну «Інтерактор», що забезпечує тонкі контролери, високу тестованість та керованість складними сценаріями (наприклад, записом студента на курс). Спроєктовано систему обміну даними в реальному часі на основі технології ActionCable (WebSocket) та механізм асинхронної обробки ресурсомістких операцій за допомогою зв’язки Sidekiq і Redis.

Функціональні можливості розробленого застосунку включають гнучку автентифікацію через Devise та OmniAuth (зокрема через провайдери соціальних мереж), розмежування прав доступу між викладачем і студентом, систему запрошень на приватні курси за унікальним токеном, інтерактивний чат і миттєві сповіщення без перезавантаження сторінки.

Програмний продукт реалізовано із застосуванням сучасних стандартів екосистеми Ruby on Rails та точкової інтеграції бібліотеки React для інтерактивних модулів. Проведено автоматизоване тестування за допомогою фреймворку RSpec, яке підтвердило коректність ключової бізнес-логіки, надійність системи розмежування доступу та стабільність роботи фонових процесів.

Ключові слова: СИСТЕМА УПРАВЛІННЯ НАВЧАННЯМ, RUBY ON RAILS, POSTGRESQL, REACT, WEBSOCKET, SIDEKIQ, REST API, АВТЕНТИФІКАЦІЯ, UML.

ABSTRACT

The qualification work is dedicated to the relevant problem of automating distance and blended learning processes through the design and development of a modern Learning Management System (LMS). The main result of the work is a fully functional full-stack web application that covers the complete learning cycle: user registration with role separation, creation and population of courses, completion of lessons and quizzes, automatic grading, and real-time messaging.

The work analyzes in detail and programmatically implements a server-side architecture based on the Ruby on Rails framework using a normalized relational PostgreSQL database. The essence of the proposed solution is the extraction of business logic into a separate service layer based on the Interactor pattern, which ensures thin controllers, high testability, and controllability of complex scenarios (such as enrolling a student in a course). A real-time data exchange system based on ActionCable (WebSocket) technology and a mechanism for asynchronous processing of resource-intensive operations using the Sidekiq and Redis stack have been designed.

The functionality of the developed application includes flexible authentication via Devise and OmniAuth (including social network providers), separation of access rights between teacher and student, a system of invitations to private courses using a unique token, an interactive chat, and instant notifications without reloading the page.

The software product is implemented using modern standards of the Ruby on Rails ecosystem and point integration of the React library for interactive modules. Automated testing was carried out using the RSpec framework, which confirmed the correctness of the key business logic, the reliability of the access control system, and the stability of background processes.

Keywords: LEARNING MANAGEMENT SYSTEM, RUBY ON RAILS, POSTGRESQL, REACT, WEBSOCKET, SIDEKIQ, REST API, AUTHENTICATION, UML.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

API Програмний інтерфейс застосунку (Application Programming Interface).

CI/CD Безперервна інтеграція та безперервне розгортання (Continuous Integration / Continuous Deployment).

CRUD Базові операції над даними: створення, читання, оновлення, видалення (Create, Read, Update, Delete).

CSRF Міжсайтова підробка запиту (Cross-Site Request Forgery).

DOM Об’єктна модель документа (Document Object Model).

HTTP Протокол передачі гіпертексту (HyperText Transfer Protocol).

JSON Текстовий формат обміну даними (JavaScript Object Notation).

LMS Система управління навчанням (Learning Management System).

MVC Архітектурний шаблон «Модель–Подання–Контролер» (Model–View–Controller).

OAuth Відкритий протокол авторизації через зовнішнього провайдера (Open Authorization).

ORM Об’єктно-реляційне відображення (Object-Relational Mapping).

REST Передача репрезентативного стану — стиль побудови вебсервісів (Representational State Transfer).

SPA Односторінковий вебзастосунок (Single Page Application).

SQL Мова структурованих запитів (Structured Query Language).

SSL/TLS Криптографічні протоколи захищеної передачі даних (Secure Sockets Layer / Transport Layer Security).

UI Інтерфейс користувача (User Interface).

UML Уніфікована мова моделювання (Unified Modeling Language).

URL Уніфікований локатор ресурсу (Uniform Resource Locator).

UX Досвід взаємодії користувача з інтерфейсом (User Experience).

WebSocket Протокол двоспрямованого зв’язку між клієнтом і сервером у реальному часі.

XSS Міжсайтове виконання сценаріїв (Cross-Site Scripting).

ВСТУП

Сучасна система освіти переживає етап глибокої цифрової трансформації, у межах якої традиційні форми передачі знань поступово доповнюються або повністю замінюються електронними платформами. Пандемічні обмеження останніх років, поширення гнучких графіків навчання та глобалізація освітнього ринку остаточно закріпили дистанційний і змішаний формати як повноцінні складові навчального процесу. Згідно з аналітичними звітами у сфері освітніх технологій (EdTech), світовий ринок цифрових освітніх платформ оцінюється у сотні мільярдів доларів США та демонструє стійкий середньорічний темп приросту на рівні 15–18 %. Центральним елементом цієї екосистеми виступає система управління навчанням (Learning Management System, LMS) — програмний комплекс, що забезпечує організацію, доставку, супровід та контроль навчального процесу.

Попри значну кількість наявних рішень, більшість із них або є надмірно складними та перевантаженими функціоналом промислового масштабу, або, навпаки, не забезпечують ключових для якісного навчання можливостей: миттєвого зворотного зв’язку між учасниками, гнучкого розмежування ролей, зручної системи запрошень та автоматизації рутинних операцій. Окремою проблемою залишається продуктивність: під час зростання кількості користувачів синхронне виконання ресурсомістких операцій (масові розсилки, генерація нагадувань) блокує веб-сервер і погіршує досвід взаємодії. Це створює потребу в розробленні збалансованого програмного продукту, який поєднує повноту навчального циклу, інтерактивність реального часу та масштабовану серверну архітектуру.

Поява зрілих серверних фреймворків, зокрема Ruby on Rails, разом із технологіями двоспрямованого зв’язку (WebSocket) та системами асинхронної обробки завдань відкрила нові можливості для побудови надійних і водночас швидких у розробленні освітніх платформ. Саме на перетині цих технологій і лежить тема цієї кваліфікаційної роботи.

Об’єктом дослідження виступають бізнес-процеси організації дистанційного та змішаного навчання у середовищі сучасних вебзастосунків.

Предметом дослідження є методи проєктування, архітектурні рішення та технології розроблення серверної частини платформи управління навчанням.

Метою роботи є підвищення ефективності організації навчального процесу шляхом проєктування та реалізації сучасної масштабованої LMS-платформи з використанням актуальних технологій веброзроблення. Для досягнення мети необхідно вирішити такі завдання:

– проаналізувати предметну область, роль і вимоги до сучасних освітніх платформ, а також здійснити порівняльний огляд наявних LMS-рішень;

– обґрунтувати вибір технологічного стека та архітектури серверної частини програмного продукту;

– спроєктувати нормалізовану базу даних і об’єктну модель предметної області (користувачі, курси, теми, уроки, тести, оцінки, записи на курси);

– реалізувати серверну частину з розподілом ролей (викладач, студент) та винесенням бізнес-логіки в окремий сервісний шар;

– налаштувати REST API-ендпоінти та з’єднання реального часу на основі WebSocket;

– реалізувати асинхронну обробку фонових завдань (сповіщення, нагадування, архівування курсів);

– покрити розроблений функціонал автоматизованими тестами та перевірити коректність роботи системи.

Методологічну основу роботи становлять методи системного аналізу (для дослідження предметної області та вимог), методи об’єктно-орієнтованого проєктування та моделювання мовою UML (для побудови структури системи), методи нормалізації реляційних баз даних (для проєктування схеми збереження даних), а також методи автоматизованого та функціонального тестування (для перевірки коректності реалізації).

Поєднання цих методів забезпечує науково обґрунтований і водночас практично орієнтований підхід до розв’язання поставленої задачі.

Наукова та практична новизна роботи полягає у комплексному поєднанні в межах єдиного програмного продукту трьох архітектурних підходів, що зазвичай розглядаються окремо: винесення бізнес-логіки в сервісний шар інтерактoрів для забезпечення тестованості, організації двоспрямованого зв’язку реального часу для миттєвого зворотного зв’язку та асинхронної обробки фонових завдань для масштабованості. Запропонована інтеграція цих підходів на прикладі предметної області систем управління навчанням демонструє практичний шлях побудови сучасних освітніх платформ і може слугувати методичним орієнтиром для подібних розробок.

Структурно кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаної літератури та додатків. У першому розділі проаналізовано предметну область, наявні рішення та технології розроблення. Другий розділ присвячено проєктуванню та програмній реалізації системи. У третьому розділі описано тестування, оцінювання якості та розгортання програмного продукту. Загальний обсяг роботи, ілюстрований рисунками та таблицями, відображає повний життєвий цикл розроблення — від аналізу вимог до введення системи в експлуатацію.

Практичне значення роботи полягає в тому, що розроблений програмний продукт є завершеним і працездатним рішенням, придатним для організації навчального процесу в межах навчального закладу, освітнього центру або окремого викладача. Запропоновані архітектурні рішення — винесення бізнес-логіки в інтерактори, асинхронна обробка завдань та обмін даними в реальному часі — можуть бути використані як референсний приклад побудови масштабованих вебзастосунків на базі Ruby on Rails.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД НАЯВНИХ РІШЕНЬ

1.1 Сфери застосування та значення систем управління навчанням

Система управління навчанням (LMS) — це програмний застосунок або вебплатформа, призначена для адміністрування, документування, відстеження, звітування, автоматизації та доставки освітніх курсів, навчальних програм і матеріалів. На відміну від простого сховища файлів чи відеохостингу, LMS реалізує цілісний навчальний цикл, у якому кожен учасник має чітко визначену роль, а взаємодія між учасниками регламентована бізнес-правилами. Саме комплексність і керованість цього циклу відрізняють повноцінну систему управління навчанням від набору розрізнених інструментів.

Історично концепція LMS виникла в корпоративному секторі для організації внутрішнього навчання персоналу, проте згодом поширилася на всі рівні освіти — від загальноосвітніх закладів до університетів і масових відкритих онлайн-курсів. Сьогодні системи управління навчанням застосовуються у кількох ключових сферах, кожна з яких висуває власні вимоги до функціональності та архітектури програмного продукту.

Перша сфера — формальна освіта (школи, коледжі, університети). Тут LMS виступає цифровим середовищем, що супроводжує очне або змішане навчання: викладач публікує матеріали, видає завдання, перевіряє роботи та виставляє оцінки, а студенти отримують доступ до структурованого контенту й відстежують власну успішність. Критичними для цієї сфери є розмежування ролей, контроль дедлайнів та прозора система оцінювання.

Друга сфера — корпоративне навчання та підвищення кваліфікації. Компанії використовують LMS для адаптації нових співробітників, сертифікації та регулярного підвищення кваліфікації. Тут на перший план виходять масштабованість, автоматизація розсилок і звітність про проходження курсів.

Третя сфера — комерційні освітні платформи та індивідуальне репетиторство. Окремі викладачі та невеликі освітні центри потребують легкого у розгортанні рішення, що дозволяє швидко створити курс, запросити обмежене коло учасників за персональним посиланням і організувати оперативну комунікацію. Саме на потреби цієї сфери орієнтований розроблений у межах роботи програмний продукт.

Окремо варто розглянути еволюцію самого поняття системи управління навчанням. Перше покоління таких систем, що з'явилося наприкінці 1990-х років, зводилося переважно до електронних каталогів матеріалів та простих засобів контролю проходження курсів. Друге покоління, сформоване у 2000-х роках, додало інтерактивні елементи — тести, форуми, журнали оцінок — та закріпило модель розмежування ролей. Сучасне, третє покоління систем характеризується глибокою інтеграцією технологій реального часу, мобільною доступністю, персоналізацією навчальних траєкторій та широким застосуванням автоматизації. Розроблюваний у межах роботи продукт належить саме до третього покоління, оскільки поєднує структурований контент із миттєвим зворотним зв'язком та автоматизованими фоновими процесами.

Важливим критерієм класифікації LMS є також спосіб їх розгортання. Розрізняють хмарні рішення за моделлю «програмне забезпечення як послуга» (SaaS), коли постачальник бере на себе всю інфраструктуру, та рішення для самостійного розгортання (self-hosted), що встановлюються на власних серверах організації. Перші вирізняються простотою початку роботи, проте обмежують контроль над даними та можливості кастомізації; другі надають повний контроль, але потребують технічної підтримки. Розроблена система належить до категорії self-hosted, що є оптимальним для навчальних закладів, які прагнуть зберігати контроль над персональними даними студентів та гнучко налаштовувати функціональність під власні потреби.

Незалежно від сфери застосування, ефективна система управління навчанням повинна забезпечувати реалізацію кількох взаємопов’язаних бізнес-процесів. До них належать: реєстрація та автентифікація користувачів із подальшим вибором або призначенням ролі (студент чи викладач); управління курсом, що охоплює його створення, наповнення темами та уроками, а також запис студентів; власне навчальний процес — перегляд уроків різних типів, проходження тестування й автоматичне отримання оцінки; комунікація в реальному часі через сповіщення та обмін повідомленнями. Окремо слід виділити асинхронні бізнес-процеси: масова розсилка сповіщень чи нагадувань про дедлайни не повинна блокувати основний потік веб-сервера, а отже, потребує винесення у фонову чергу обробки.

Слід також враховувати вимоги до якості програмного забезпечення, визначені міжнародним стандартом ISO/IEC 25010, що описує модель якості продукту через вісім характеристик: функціональну придатність, продуктивність, сумісність, зручність використання, надійність, безпеку, супроводжуваність та переносність. Для системи управління навчанням найбільш значущими є функціональна придатність (повнота реалізації навчального циклу), зручність використання (низький поріг входження для викладачів і студентів), надійність (цілісність навчальних даних та оцінок) і безпека (захист персональних даних та розмежування доступу). Ці характеристики стали орієнтиром під час проєктування та оцінювання розробленого продукту.

Узагальнюючи, можна стверджувати, що значення систем управління навчанням полягає у переведенні неформалізованої та трудомісткої взаємодії «викладач — студент» у керовану, автоматизовану та масштабовану програмну форму. Ключовими вимогами до такої системи є автоматизація рутинних дій (контроль дедлайнів, перевірка тестів, розсилки), інтерактивність (зворотний зв’язок у реальному часі) та безпека (надійне розмежування прав доступу між ролями). Саме ці три вимоги стали

визначальними під час формування технічного завдання на розроблення програмного продукту.

Доцільно окремо наголосити на педагогічному вимірі систем управління навчанням. Сучасна дидактика розрізняє синхронне навчання (одночасна взаємодія учасників у реальному часі) та асинхронне навчання (самостійне опрацювання матеріалів у зручний час). Повноцінна LMS має підтримувати обидва режими: асинхронний — через структуровані уроки, тести й матеріали, доступні у будь-який момент, та синхронний — через чат, сповіщення й оперативний зворотний зв’язок. Саме поєднання цих режимів забезпечує так зване змішане навчання (blended learning), яке за результатами численних досліджень демонструє вищу ефективність порівняно з суто очним або суто дистанційним форматом.

Ще одним важливим аспектом є мотивація та залученість студентів. Дослідження освітніх технологій послідовно доводять, що пасивне споживання матеріалів дає слабкий навчальний результат, тоді як активна взаємодія — проходження тестів із негайним зворотним зв’язком, отримання своєчасних сповіщень про оцінки та можливість поставити запитання в реальному часі — істотно підвищує засвоєння матеріалу. Отже, базовою вимогою до розроблюваної системи має бути не лише доставка контенту, а й забезпечення безперервної інтерактивної взаємодії між учасниками навчального процесу. Ця вимога безпосередньо вплинула на вибір технологій реального часу та асинхронної обробки, розглянутих у наступних підрозділах.

1.2 Огляд та порівняльний аналіз наявних рішень

Перед початком проєктування власного програмного продукту доцільно проаналізувати наявні на ринку системи управління навчанням. Це дозволяє визначити усталені галузеві практики, виявити сильні та слабкі сторони існуючих рішень і обґрунтувати доцільність розроблення нової системи. Розглянемо найпоширеніші платформи, що репрезентують різні підходи до організації навчання.

Moodle — найвідоміша система управління навчанням з відкритим вихідним кодом, написана мовою PHP. Її головними перевагами є безкоштовність, надзвичайна гнучкість та величезна екосистема плагінів. Водночас Moodle часто критикують за перевантажений і застарілий інтерфейс, високий поріг входження для викладачів та складність адміністрування й розгортання, що потребує окремого досвідченого спеціаліста.

Google Classroom — хмарне рішення від компанії Google, тісно інтегроване з пакетом сервісів Google Workspace. Воно вирізняється простотою та мінімалістичним інтерфейсом, однак значною мірою залежить від екосистеми Google, має обмежені можливості гнучкого налаштування навчального процесу та не призначене для побудови складних структурованих курсів із власною логікою оцінювання.

Coursera та подібні масові відкриті платформи (MOOC) орієнтовані передусім на доставку готового відеоконтенту широкій аудиторії. Вони демонструють високу якість матеріалів та продуману методику, проте є закритими комерційними продуктами, не призначеними для самостійного розгортання й організації приватного навчання невеликою групою.

Canvas LMS — сучасна хмарна платформа, що набула значного поширення в університетах США та Європи. Вона вирізняється продуманим інтерфейсом, відкритим API для інтеграцій та модульною структурою. Водночас Canvas орієнтована на великі освітні установи, має складну модель ліцензування й надлишкову для невеликого колективу функціональність, а її самостійне розгортання з відкритого коду потребує значних інфраструктурних ресурсів.

Окрему категорію становлять авторські інструменти створення курсів, такі як iSpring або Articulate, що зосереджені на розробленні інтерактивного контенту, проте не є повноцінними системами управління навчанням і потребують окремої LMS для доставки створених матеріалів. Аналіз цих рішень засвідчує загальну тенденцію: ринок чітко поділений на важкі

корпоративні платформи та вузькоспеціалізовані інструменти, тоді як ніша простих, відкритих та інтерактивних систем для невеликих освітніх колективів залишається недостатньо заповненою.

Для систематизації результатів огляду доцільно навести порівняльну характеристику розглянутих рішень за ключовими критеріями, що мають значення для цільової сфери застосування (табл. 1.1).

Таблиця 1.1 – Порівняльний аналіз наявних систем управління навчанням

Критерій	**Moodle**	**Google Classroom**	**Coursera**	**Розроблена система**
Відкритий код / самостійне розгортання	Так	Ні	Ні	Так
Простота інтерфейсу	Низька	Висока	Висока	Висока
Гнучка структура курсу (теми/уроки/тести)	Так	Частково	Так	Так
Зворотний зв’язок у реальному часі	Частково	Ні	Ні	Так
Запрошення на приватні курси за токеном	Частково	Так	Ні	Так
Автоматизація фонових процесів	Так	Частково	Так	Так
Поріг входження для викладача	Високий	Низький	Низький	Низький

Як видно з наведеного порівняння, наявні рішення утворюють два полюси: важкі та гнучкі, але складні у впровадженні системи (Moodle), і прості, але закриті або обмежені у налаштуванні платформи (Google Classroom, Coursera). Жодне з них повною мірою не задовольняє потреби невеликого освітнього колективу чи окремого викладача, який прагне отримати водночас просту у використанні, відкриту та інтерактивну систему з повноцінним зворотним зв’язком у реальному часі. Це підтверджує

доцільність розроблення власного програмного продукту, який поєднує переваги обох підходів: повноту навчального циклу та гнучкість, притаманні професійним LMS, із простотою інтерфейсу й легкістю розгортання, характерними для хмарних рішень.

1.3 Аналіз технологій розроблення серверної частини вебзастосунків

Оскільки система управління навчанням є передусім складним серверним застосунком з інтенсивною роботою з даними, розмежуванням доступу та бізнес-логікою, ключовим етапом аналізу є вибір технологій для побудови саме серверної частини. Розглянемо основні архітектурні підходи та технологічні складові, релевантні для поставленої задачі.

За способом організації коду сучасні вебзастосунки будуються переважно за двома підходами — монолітним та мікросервісним. Мікросервісна архітектура передбачає поділ системи на множину незалежних служб, що взаємодіють мережею. Вона виправдана для великих систем з окремими командами розробників, проте суттєво ускладнює розгортання, налагодження та супровід на ранніх етапах. Для проєкту масштабу кваліфікаційної роботи, що розробляється одним виконавцем, оптимальним є монолітний підхід: єдина кодова база, спільна транзакційна цілісність даних, простота локального запуску та розгортання. Саме тому як основу обрано монолітний фреймворк Ruby on Rails.

Вибір серверної мови програмування та фреймворку є визначальним рішенням, що впливає на швидкість розроблення, продуктивність та супроводжуваність системи. Серед найпоширеніших альтернатив доцільно розглянути три екосистеми. Платформа Node.js (мова JavaScript) забезпечує високу пропускну здатність завдяки неблокувальній моделі введення-виведення, проте її екосистема фрагментована, а відсутність єдиного «канонічного» фреймворку покладає на розробника значний обсяг архітектурних рішень. Фреймворк Django (мова Python) є зрілим і добре документованим рішенням із вбудованою адміністративною панеллю, однак

менш гнучкий у частині метапрограмування. Фреймворк Ruby on Rails (мова Ruby) вирізняється найвищою швидкістю розроблення інформаційних систем із насиченою предметною логікою, цілісною та узгодженою екосистемою, а також виразним синтаксисом, що підвищує читабельність коду. Порівняння цих екосистем за ключовими критеріями наведено у таблиці 1.2.

Таблиця 1.2 – Порівняння серверних технологій розроблення

Критерій	**Node.js**	**Django (Python)**	**Ruby on Rails**
Швидкість розроблення	Середня	Висока	Дуже висока
Цілісність екосистеми	Низька	Висока	Дуже висока
Вбудовані засоби (ORM, тести)	Частково	Так	Так
Підтримка реального часу	Нативна	Через розширення	Нативна (ActionCable)
Зрілість конвенцій	Низька	Висока	Дуже висока

З урахуванням характеру задачі — система з насиченою предметною логікою, складними зв’язками між сутностями та потребою у швидкому розробленні одним виконавцем — оптимальним вибором визнано фреймворк Ruby on Rails.

Ruby on Rails — це повностековий вебфреймворк, побудований на архітектурному шаблоні «Модель–Подання–Контролер» (MVC) та філософії «угода важливіша за конфігурацію» (convention over configuration). Він надає готові механізми для маршрутизації запитів, роботи з базою даних, валідації, безпеки та тестування, що дозволяє зосередитися на бізнес-логіці, а не на інфраструктурному коді. Завдяки високій швидкості розроблення та зрілій екосистемі бібліотек (гемів) Rails є одним із найпопулярніших рішень для створення інформаційних систем із насиченою предметною логікою, до яких належить і LMS.

Для збереження даних застосовують реляційні або документоорієнтовані СКБД. Предметна область системи управління навчанням має чітко виражену реляційну природу: користувачі, курси, теми, уроки, питання та оцінки пов’язані між собою численними зв’язками «один

до багатьох» і «багато до багатьох», а цілісність цих зв’язків є критичною. Тому обґрунтованим є вибір реляційної СКБД PostgreSQL — надійної, продуктивної та багатофункціональної системи з підтримкою складних запитів, транзакцій та обмежень цілісності. Взаємодія з базою даних здійснюється через вбудований у Rails механізм об’єктно-реляційного відображення (ORM) Active Record, що подає рядки таблиць у вигляді об’єктів предметної області.

Шаблон «Модель–Подання–Контролер», покладений в основу фреймворку, заслуговує на детальніший розгляд, оскільки визначає загальну організацію коду. Модель інкапсулює дані та бізнес-правила, не залежить від способу їх відображення та відповідає за взаємодію з базою даних. Подання відповідає виключно за представлення даних користувачеві й не містить бізнес-логіки. Контролер виступає посередником: приймає запит, звертається до моделей та обирає відповідне подання для формування відповіді. Чіткий розподіл цих відповідальностей знижує зв’язність компонентів, спрощує тестування й супровід та дозволяє змінювати один шар, не зачіпаючи інші. Водночас практика показує, що класичний MVC недостатній для складних бізнес-сценаріїв, тому його доповнюють сервісним шаром, про що йтиметься у другому розділі.

Механізм об’єктно-реляційного відображення Active Record заслуговує окремої уваги як ключовий елемент роботи з даними. Він реалізує однойменний архітектурний патерн, за яким кожен клас-модель відповідає таблиці бази даних, екземпляр класу — рядку таблиці, а атрибути об’єкта — стовпцям. Active Record бере на себе генерацію SQL-запитів, керування зв’язками між таблицями, валідацію даних та міграції схеми. Це звільняє розробника від написання рутинного SQL-коду й дозволяє оперувати поняттями предметної області, водночас зберігаючи можливість виконання оптимізованих запитів за потреби. Механізм міграцій забезпечує версіонування схеми бази даних, що є критичним для командної розробки та безпечного оновлення робочих систем.

Окремої уваги потребує технологія забезпечення зворотного зв’язку в реальному часі. Класична модель «запит — відповідь» протоколу HTTP не дозволяє серверу самостійно ініціювати передачу даних клієнту, що унеможливлює миттєву доставку повідомлень чи сповіщень без постійного опитування сервера. Для подолання цього обмеження застосовують протокол WebSocket, який встановлює постійне двоспрямоване з’єднання між клієнтом і сервером. У фреймворку Ruby on Rails цю технологію реалізує підсистема ActionCable, що інтегрує WebSocket із рештою застосунку через механізм каналів та трансляцій.

Окремо слід зупинитися на питанні безпеки серверної частини. Фреймворк Ruby on Rails надає вбудовані механізми захисту від найпоширеніших вебзагроз: автоматичне екранування виведення для запобігання міжсайтовому виконанню сценаріїв (XSS), токени захисту від міжсайтової підробки запитів (CSRF), параметризовані запити до бази даних, що унеможливлюють SQL-ін’єкції, а також механізм «сильних параметрів» для контролю масового присвоєння атрибутів. Ці засоби, доповнені продуманим розмежуванням доступу та безпечним зберіганням паролів засобами бібліотеки Devise, утворюють багаторівневу систему захисту даних користувачів.

Нарешті, важливою складовою серверної архітектури є система асинхронної обробки фонових завдань. Виконання ресурсомістких або тривалих операцій безпосередньо у межах HTTP-запиту неприпустиме, оскільки призводить до затримок відповіді та погіршення досвіду користувача. Для винесення таких операцій у фонові черги використано зв’язку технологій Sidekiq та Redis: Sidekiq виконує завдання у окремих потоках, а високопродуктивне сховище Redis виступає брокером черг. Детальніше обґрунтування та реалізацію цього механізму наведено у другому розділі.

Підсумовуючи аналіз технологій, варто наголосити, що обраний технологічний стек є не набором випадкових інструментів, а цілісною та

взаємоузгодженою екосистемою. Фреймворк Ruby on Rails, СКБД PostgreSQL, підсистема реального часу ActionCable, система фонової обробки Sidekiq із Redis, бібліотеки автентифікації Devise та OmniAuth, фреймворк тестування RSpec — усі ці складові природно інтегруються між собою, мають зрілу документацію та широку спільноту. Це знижує ризики розроблення, прискорює реалізацію та забезпечує довготривалу підтримуваність продукту. Точкова інтеграція бібліотеки React доповнює стек на боці клієнта там, де потрібна підвищена інтерактивність, не порушуючи загальної цілісності рішення.

Таким чином, проведений технологічний аналіз дозволяє стверджувати, що поставлена задача може бути ефективно розв’язана за допомогою монолітної архітектури на базі Ruby on Rails із сервісним шаром бізнес-логіки. Цей висновок є відправною точкою для формальної постановки задачі, наведеної у наступному підрозділі.

1.4 Постановка задачі на розроблення програмного продукту

На основі проведеного аналізу предметної області, огляду наявних рішень та доступних технологій можна сформулювати конкретну постановку задачі. Узагальнення виявлених недоліків існуючих систем дозволяє виокремити чотири ключові проблеми, на розв’язання яких спрямовано розроблення програмного продукту.

1. Відсутність зворотного зв’язку в реальному часі — у багатьох наявних рішеннях викладачі та студенти не отримують миттєвих сповіщень про нові завдання, відповіді чи оцінки, що знижує залученість і оперативність взаємодії.

2. Складність контролю успішності — бракує зручного інструменту для відстеження прогресу студентів, виставлення оцінок та перегляду результатів тестування у межах єдиного середовища.

3. Проблема автентифікації та доступу — відсутність гнучкої системи реєстрації (зокрема через провайдери соціальних мереж) та надійного

розмежування ролей (викладач, студент) створює ризики безпеки й незручність для користувачів.

4. Масштабованість і продуктивність — за зростання кількості користувачів синхронне виконання процесів сповільнює роботу системи, тому фонова обробка завдань стає необхідністю, а не опцією.

Розв’язання кожної з окреслених проблем потребує конкретних технологічних та архітектурних рішень. Проблему зворотного зв’язку в реальному часі вирішує впровадження технології WebSocket через підсистему ActionCable. Проблему контролю успішності — проєктування цілісної моделі оцінювання, що пов’язує оцінки з конкретними роботами студентів. Проблему автентифікації та доступу — застосування перевірених бібліотек Devise і OmniAuth разом із послідовним розмежуванням ролей на рівні бізнес-логіки. Проблему масштабованості — винесення ресурсомістких операцій у фонові черги Sidekiq. Таким чином, кожна виявлена проблема отримує обґрунтоване технологічне розв’язання, що в сукупності формує цілісну концепцію системи.

Виходячи з окреслених проблем, метою розроблення є створення повнофункціональної системи управління навчанням, що реалізує повний навчальний цикл та усуває зазначені недоліки. Функціональні вимоги до програмного продукту передбачають реалізацію таких можливостей:

– реєстрація та автентифікація користувачів, у тому числі через зовнішніх провайдерів за протоколом OAuth, із розмежуванням ролей «студент» та «викладач»;

– створення викладачем публічних і приватних курсів, наповнення їх темами та уроками трьох типів (текстовий, відео, тест);

– запис студентів на курси, зокрема запрошення на приватні курси за унікальним токеном-посиланням;

– проходження тестів із автоматичною перевіркою відповідей та виставленням оцінок;

– обмін повідомленнями та доставка сповіщень у реальному часі без перезавантаження сторінки;

– асинхронне виконання фонових завдань — розсилки, нагадування про дедлайни, архівування завершених курсів.

Нефункціональні вимоги охоплюють надійність (цілісність даних та коректність розмежування доступу), продуктивність (відсутність блокування веб-сервера під час масових операцій), безпеку (захист від типових вебзагроз, безпечне зберігання паролів) та тестованість (покриття ключової бізнес-логіки автоматизованими тестами). Деталізацію вимог, проєктування архітектури та програмну реалізацію наведено у наступному розділі.

Висновки до розділу. У першому розділі проаналізовано предметну область систем управління навчанням, визначено їхню роль, сфери застосування та ключові бізнес-процеси. Здійснено порівняльний огляд наявних рішень, який підтвердив доцільність розроблення власного програмного продукту, що поєднує гнучкість професійних LMS із простотою хмарних сервісів. Обґрунтовано вибір технологічного стека на базі фреймворку Ruby on Rails, СКБД PostgreSQL, технології реального часу ActionCable та системи фонової обробки Sidekiq. Сформульовано постановку задачі, функціональні та нефункціональні вимоги, що стали основою для проєктування системи.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

2.1 Формування вимог до програмного забезпечення

Проєктування програмної системи розпочинається з деталізації вимог, сформульованих у постановці задачі, та визначення кола її користувачів. У розробленій системі управління навчанням виокремлено двох основних дійових осіб (акторів) — викладача та студента, ролі яких визначають доступний набір дій. Окремою службовою «дійовою особою» виступає сама система, що автономно виконує фонові операції (нагадування, архівування, розсилки) без безпосередньої участі користувача.

Викладач є власником навчального контенту. Він створює курси, наповнює їх темами та уроками, формує тести з питаннями й варіантами відповідей, переглядає роботи студентів, виставляє оцінки та запрошує учасників на приватні курси. Студент є споживачем контенту: він переглядає доступні курси, записується на них, проходить уроки й тести, отримує оцінки, зберігає курси в закладки та спілкується з іншими учасниками. Розмежування цих можливостей є критичною вимогою безпеки і реалізується на рівні бізнес-логіки сервера.

На основі ролей сформовано перелік основних прецедентів використання системи, який у стислому вигляді подано у таблиці 2.1. Прецеденти згруповано за бізнес-процесами, що дозволяє простежити повноту покриття функціональних вимог.

Таблиця 2.1 – Основні прецеденти використання системи

Бізнес-процес	Прецедент	Актор
Автентифікація	Реєстрація та вхід (email або OAuth)	Студент, Викладач
Управління курсом	Створення та редагування курсу	Викладач
Управління курсом	Наповнення курсу темами та уроками	Викладач
Запис на курс	Запис на публічний курс	Студент
Запис на курс	Запрошення на приватний курс за токеном	Викладач → Студент

Навчання	Перегляд уроку (текст / відео / тест)	Студент
Навчання	Проходження тесту та автооцінювання	Студент
Оцінювання	Виставлення оцінки за роботу	Викладач
Комунікація	Обмін повідомленнями в реальному часі	Студент, Викладач
Комунікація	Отримання сповіщень	Студент, Викладач

Для повного розуміння взаємодії користувачів із системою доцільно простежити основні потоки даних. Під час реєстрації дані користувача проходять валідацію, пароль хешується, і створюється обліковий запис із роллю за замовчуванням. Під час створення курсу викладач формує його структуру, яка зберігається у вигляді ієрархії пов’язаних записів. Під час проходження тесту відповіді студента зіставляються з правильними варіантами, на основі чого автоматично обчислюється оцінка. Кожен із цих потоків супроводжується відповідними сповіщеннями, що доставляються зацікавленим користувачам у реальному часі або асинхронно.

Для ілюстрації взаємодії акторів із системою розглянемо типовий наскрізний сценарій. Студент реєструється у системі або входить через обліковий запис Google, після чого переглядає каталог публічних курсів. Обравши курс, він записується на нього, що активує ланцюжок перевірок і створює відповідний запис, а викладач курсу отримує сповіщення про нового учасника. Далі студент послідовно проходить уроки обраного курсу: читає текстові матеріали, переглядає відео та виконує тести, відповіді на які автоматично оцінюються. Виставлені оцінки одразу стають доступними студентові, а за потреби він може поставити запитання викладачеві через чат у реальному часі. Цей сценарій охоплює більшість прецедентів системи й демонструє цілісність навчального циклу.

Деталізуючи нефункціональні вимоги, слід наголосити на чотирьох аспектах. Надійність забезпечується обмеженнями цілісності на рівні бази даних та валідаціями на рівні моделей предметної області. Безпека досягається шифруванням паролів, захистом від міжсайтової підробки

запитів (CSRF) та послідовним розмежуванням доступу. Продуктивність гарантується винесенням тривалих операцій у фонові черги та індексуванням ключових стовпців бази даних. Тестованість закладається в архітектуру шляхом відокремлення бізнес-логіки від контролерів, що робить її доступною для модульного тестування.

2.2 Проєктування архітектури програмної системи

Архітектуру системи побудовано на основі класичного для фреймворку Ruby on Rails шаблону «Модель–Подання–Контролер» (MVC), доповненого додатковим сервісним шаром бізнес-логіки. Такий підхід забезпечує чіткий розподіл відповідальностей між компонентами та високу супроводжуваність коду.

Доцільно навести типову структуру каталогів застосунку Ruby on Rails, що відображає архітектурний поділ. Каталог моделей (app/models) містить класи предметної області; каталог контролерів (app/controllers) — обробники запитів; каталог подань (app/views) — шаблони сторінок; каталог інтеракторів (app/interactors) — класи бізнес-логіки; каталог каналів (app/channels) — обробники з’єднань реального часу; каталог фонових завдань (app/jobs) — асинхронні операції; каталог поштових модулів (app/mailers) — формування листів. Така стандартизована структура є частиною філософії «угода важливіша за конфігурацію» і дозволяє будь-якому розробникові, знайомому з фреймворком, швидко орієнтуватися в проєкті.

Шар моделей відповідає за подання сутностей предметної області та їхні зв’язки, валідацію даних і взаємодію з базою даних через ORM Active Record. Шар контролерів приймає HTTP-запити, керує автентифікацією й авторизацією та координує виклик бізнес-логіки. Шар подань формує HTML-сторінки та JSON-відповіді для клієнта. Для частин інтерфейсу, що потребують підвищеної інтерактивності (чат, лічильник сповіщень), застосовано точкову інтеграцію бібліотеки React, тоді як решта інтерфейсу будується засобами серверного рендерингу та технології Hotwire/Turbo.

Принциповою архітектурною особливістю системи є винесення бізнес-логіки в окремий сервісний шар на основі патерну «Інтерактор». Потреба в цьому виникає тоді, коли контролер стає «товстим» і важко тестованим, а модель починає знати надто багато — водночас зберігає дані, надсилає листи, обчислює бали та викликає зовнішні служби. Щоб уникнути такого змішування відповідальностей, кожну бізнес-операцію інкапсульовано в окремий клас-інтерактор, що виконує рівно одну дію та повертає контекст із результатом або помилкою. Якщо на будь-якому кроці виникає помилкова ситуація, викликається метод переривання, і весь ланцюжок операцій зупиняється. Конкретну реалізацію цього підходу детально розглянуто у підрозділі 2.4.

Об’єктну модель предметної області реалізовано у вигляді набору класів-моделей Active Record, що відображають основні сутності системи. Структуру каталогу моделей наведено на рисунку 2.1.

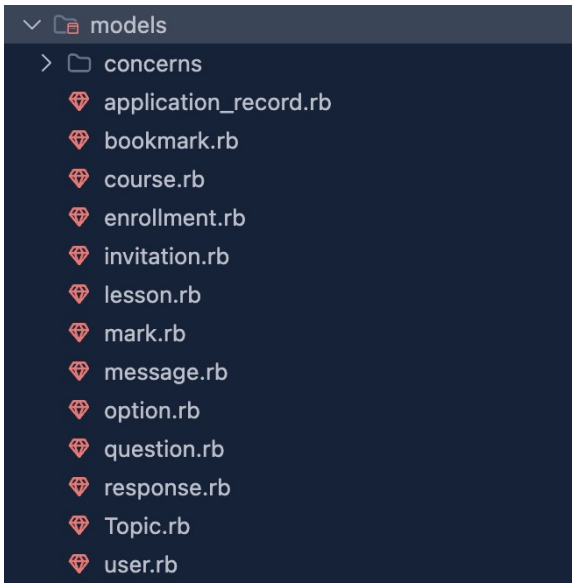

Рисунок 2.1 – Структура каталогу моделей предметної області

Як видно з рисунка, кожній сутності предметної області відповідає окремий клас-модель: користувач (user), курс (course), тема (topic), урок (lesson), питання (question), варіант відповіді (option), відповідь студента (response), оцінка (mark), запис на курс (enrollment), запрошення (invitation), закладка (bookmark) та повідомлення (message). Такий поділ забезпечує

єдину відповідальність кожної моделі та узгоджується з принципами чистої архітектури.

Зв’язки між сутностями описуються декларативно засобами Active Record. Як приклад наведемо фрагмент моделі курсу, що оголошує його зв’язки з викладачем, темами, записами на курс та студентами, а також валідації обов’язкових полів:

```
class Course < ApplicationRecord
  belongs_to :instructor, class_name: "User"
  has_many :topics, dependent: :destroy
  has_many :enrollments, dependent: :destroy
  has_many :students, through: :enrollments, source: :user
  has_many :invitations, dependent: :destroy

  validates :title, presence: true
  scope :active, -> { where(is_archived: false) }
end
```

Такий декларативний опис робить код самодокументованим: зв’язки «один до багатьох» (курс — теми) та «багато до багатьох» через проміжну таблицю (курс — студенти через записи) виражаються одним рядком, а директива dependent: :destroy гарантує каскадне видалення пов’язаних записів і цілісність даних. Іменовані вибірки (scope) інкапсулюють типові запити до бази даних, наприклад вибірку лише активних курсів.

Розглянемо також життєвий цикл типового запиту в архітектурі системи. Запит від браузера спершу потрапляє до маршрутизатора, який зіставляє URL та HTTP-метод із відповідним контролером і дією. Контролер перевіряє автентифікацію й авторизацію користувача, після чого делегує виконання бізнес-логіки відповідному інтерактору. Інтерактор взаємодіє з моделями, які зчитують або змінюють дані в базі через ORM. За потреби тривалі операції ставляться у фонову чергу, а зміни, що мають бути доставлені іншим користувачам, транслюються через ActionCable. Нарешті, контролер формує відповідь — HTML-сторінку через шар подань або JSON для клієнтських компонентів. Такий чіткий потік керування спрощує розуміння та супровід системи.

2.3 Проєктування бази даних

Для збереження та обробки інформації розробленої системи управління навчанням спроєктовано нормалізовану реляційну базу даних під управлінням СКБД PostgreSQL. Архітектура схеми логічно розділена на кілька взаємопов’язаних підсистем: модуль управління навчальним контентом (курси, теми, уроки), модуль авторизації та контролю доступу (користувачі, записи на курси, запрошення) і підсистему тестування знань (питання, варіанти, відповіді, оцінки). Нормалізація схеми до третьої нормальної форми усуває надлишковість даних та аномалії оновлення.

Центральною сутністю навчального контенту є курс, що належить викладачу й поділяється на теми, а теми, своєю чергою, — на уроки. Урок може мати один із трьох типів: текстовий, відео або тест. Уроки-тести містять питання з варіантами відповідей, кожен із яких має прапорець правильності. Відповіді студентів пов’язуються з уроками або питаннями, а виставлені оцінки — з уроком, студентом і конкретною відповіддю. Ієрархію навчального контенту та зв’язки між сутностями наочно подано на рисунку 2.2.

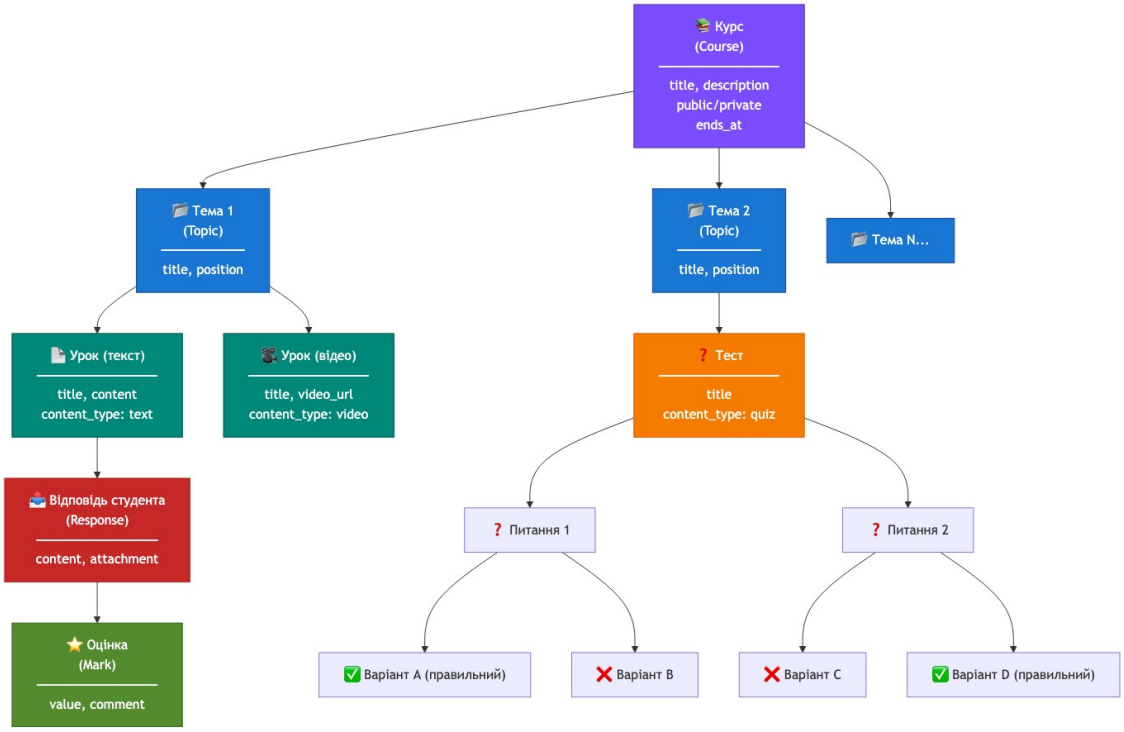

Рисунок 2.2 – Структура навчального контенту та зв’язки між сутностями

Узагальнений перелік основних сутностей бази даних, їхнього призначення та ключових зв’язків наведено у таблиці 2.2. Це дозволяє цілісно охопити структуру предметної області перед детальним розглядом окремих таблиць.

Таблиця 2.2 – Основні сутності бази даних та їхні зв’язки

Сутність	Призначення	Основні зв’язки
users	Користувачі системи (студент / викладач)	має багато courses, enrollments
courses	Навчальні курси	належить instructor, має topics
topics	Теми в межах курсу	належить course, має lessons
lessons	Уроки (текст / відео / тест)	належить topic, має questions
questions	Питання тесту	належить lesson, має options
options	Варіанти відповіді	належить question
responses	Відповіді студентів (поліморфні)	належить user, lesson / question
marks	Оцінки (0–100)	належить user, lesson, response
enrollments	Записи студентів на курси	належить user, course
invitations	Запрошення за токеном	належить course
bookmarks	Закладки курсів	належить user, course
messages	Повідомлення чату	належить user

Опишемо детальніше призначення та зв’язки ключових сутностей системи:

– User (користувач) — має роль student або teacher. Викладач створює курси, студент на них записується. Підтримується вхід через провайдер OAuth2 або за допомогою пари «email — пароль».

– Course (курс) — належить викладачу, може бути публічним або приватним. Має дату завершення, після настання якої автоматично запускаються фонові завдання архівування та нагадування.

– Topic / Lesson (тема / урок) — курс ділиться на теми, а теми — на уроки. Урок має один із трьох типів: текстовий, відео або тест; після дедлайну урок автоматично закривається.

– Question / Option (питання / варіант) — уроки типу «тест» містять питання з варіантами відповіді, кожен із яких має прапорець правильності is_correct.

– Response (відповідь) — відповідь студента. Поліморфна: може стосуватися як уроку, так і окремого питання тесту; до відповіді можна прикріпити файл.

– Mark (оцінка) — оцінка від 0 до 100, прив’язана до уроку, студента та відповіді. Під час виставлення оцінки викладач зобов’язаний вказати відповідну роботу студента.

– Enrollment (запис на курс) — унікальний запис студента на курс, що створюється через ланцюжок перевірок доступності курсу та ролі користувача.

– Invitation (запрошення) — запрошення на приватний курс за унікальним токеном; перехід за посиланням переводить запрошення у статус «прийнято» та автоматично записує студента на курс.

– Bookmark (закладка) — унікальний зв’язок «користувач — курс», що дозволяє студенту зберігати курси для швидкого доступу.

– Message (повідомлення) — після збереження одразу транслюється всім учасникам через ActionCable без додаткового HTTP-запиту.

Поліморфний зв’язок сутності відповідей заслуговує на окреме пояснення, оскільки є нетривіальним проєктним рішенням. Студент може давати відповіді як на урок загалом (наприклад, надсилаючи виконане завдання у вигляді файлу), так і на окреме питання тесту. Замість створення двох окремих таблиць застосовано поліморфну асоціацію: таблиця відповідей містить пару стовпців — ідентифікатор пов’язаного запису та назву його типу, — що дозволяє одній сутності посилатися на різні типи батьківських записів. Це зменшує дублювання структури й спрощує подальше розширення системи новими типами відповідей.

Дотримання принципів нормалізації забезпечує усунення трьох класів аномалій. Аномалії вставлення усуваються тим, що кожна сутність

зберігається у власній таблиці й не потребує наявності пов’язаних даних для створення. Аномалії оновлення усуваються відсутністю дублювання: зміна, наприклад, імені викладача відбувається в єдиному місці. Аномалії видалення контролюються директивами каскадного видалення, що узгоджено прибирають залежні записи. Водночас для часто використовуваних запитів передбачено індекси, які компенсують можливе зниження швидкості з’єднань таблиць, властиве сильно нормалізованим схемам.

Фізичну реалізацію схеми бази даних описано у файлі схеми (schema.rb), що генерується автоматично на основі міграцій. Як приклад розглянемо структуру таблиці користувачів, наведену на рисунку 2.3.

```
create_table "users", force: :cascade do |t|
  t.string "first_name", null: false
  t.string "last_name", null: false
  t.string "role", default: "student", null: false
  t.string "email", default: "", null: false
  t.string "encrypted_password", default: "", null: false
  t.string "reset_password_token"
  t.datetime "reset_password_sent_at"
  t.datetime "remember_created_at"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.string "avatar"
  t.string "uid"
  t.string "provider"
  t.index ["email"], name: "index_users_on_email", unique: true
  t.index ["reset_password_token"], name: "index_users_on_reset_password_token", unique: true
end
```

Рисунок 2.3 – Структура таблиці users

Таблиця користувачів містить персональні дані (ім’я, прізвище), роль зі значенням за замовчуванням student, поля автентифікації Devise (зашифрований пароль, токен скидання пароля) та поля для входу через зовнішніх провайдерів (provider, uid, avatar). Для забезпечення цілісності та продуктивності створено унікальні індекси за стовпцями електронної пошти та токена скидання пароля. Аналогічно спроєктовано таблицю курсів (рис. 2.4).

```
create_table "courses", force: :cascade do |t|
  t.string "title"
  t.text "description"
  t.bigint "instructor_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.boolean "public", default: true, null: false
  t.datetime "ends_at"
  t.boolean "is_archived", default: false, null: false
  t.index ["instructor_id"], name: "index_courses_on_instructor_id"
  t.index ["is_archived"], name: "index_courses_on_is_archived"
end
```

Рисунок 2.4 – Структура таблиці courses

Стратегію індексування бази даних розроблено з урахуванням типових запитів системи. Індекси створено для всіх зовнішніх ключів, оскільки за ними найчастіше виконуються з'єднання таблиць та фільтрація (наприклад, вибірка всіх тем певного курсу або всіх записів певного студента). Унікальні індекси, окрім прискорення пошуку, виконують роль обмежень цілісності — зокрема, унеможливлюють дублювання записів студента на курс чи повторне додавання курсу до закладок. Для стовпців, за якими часто здійснюється фільтрація за станом (наприклад, ознака архівації курсу), також передбачено окремі індекси. Водночас надмірне індексування свідомо уникається, оскільки кожен індекс уповільнює операції запису та збільшує обсяг бази даних.

Таблиця курсів містить назву, опис, зовнішній ключ на викладача (instructor_id), ознаку публічності (public), дату завершення (ends_at) та ознаку архівації (is_archived). Індекси за стовпцями instructor_id та is_archived прискорюють вибірку курсів конкретного викладача та фільтрацію активних курсів. Подібний підхід — явне визначення зовнішніх ключів, значень за замовчуванням, обмежень NOT NULL та індексів — застосовано до всіх таблиць бази даних, що гарантує її цілісність і продуктивність.

2.4 Програмна реалізація серверної логіки та автентифікації

Реалізацію бізнес-логіки системи побудовано на основі патерну «Інтерактор», згаданого у підрозділі 2.2. Розглянемо його застосування на

прикладі одного з найскладніших сценаріїв системи — запису студента на курс. Ця операція не зводиться до простого створення запису: вона потребує перевірки ролі користувача, перевірки доступності курсу та власне збереження запису з подальшим сповіщенням викладача. Замість того щоб розмістити всю цю логіку в контролері чи моделі, її розбито на три окремі інтерактори, об'єднані організатором EnrollStudent (рис. 2.5).

```
module Enrollments
  class EnrollStudent
    include Interactor::Organizer

    organize CheckStudentRole, CheckCourseAvailability, SaveEnrollment
  end
end
```

Рисунок 2.5 – Організатор EnrollStudent, що поєднує три інтерактори

Організатор оголошує послідовність виконання інтеракторів за допомогою директиви organize. Кожен інтерактор у ланцюжку отримує спільний контекст, виконує свою перевірку або дію та передає керування наступному. Якщо будь-який крок завершується невдало, виконується відкат і ланцюжок переривається. Розглянемо кожен крок окремо.

Перший крок — інтерактор CheckStudentRole — перевіряє, чи має користувач роль студента. Лише студент може записатися на курс; за спроби запису з боку викладача контекст переривається з відповідним повідомленням про помилку. Логіку цього кроку зображено на рисунку 2.6.

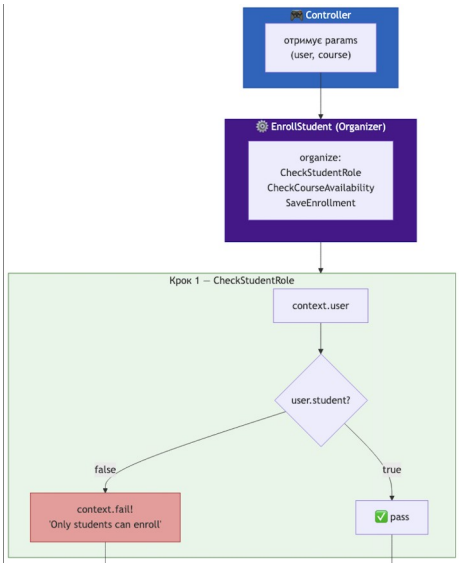

Рисунок 2.6 – Крок 1: перевірка ролі користувача (CheckStudentRole)

Другий крок — інтерактор CheckCourseAvailability — перевіряє доступність курсу: курс має бути публічним та не архівованим. У разі недоступності курсу виконання переривається з повідомленням «Course not available»; інакше керування передається далі (рис. 2.7).

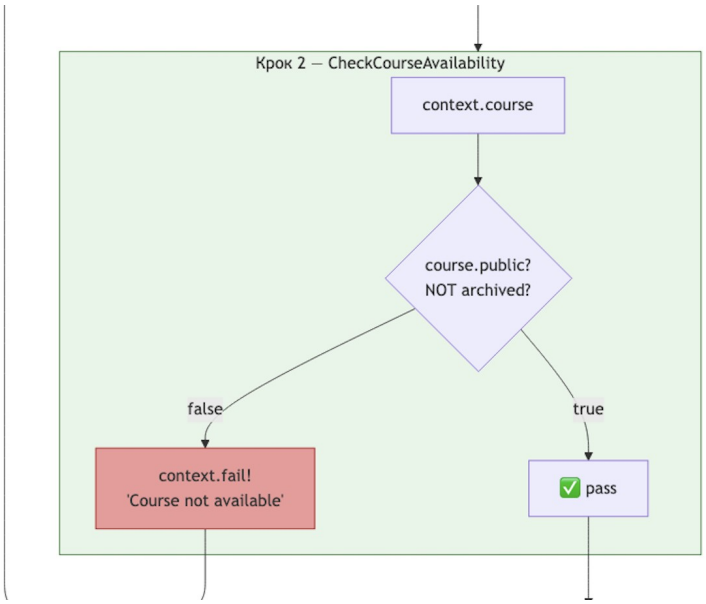

Рисунок 2.7 – Крок 2: перевірка доступності курсу (CheckCourseAvailability)

Третій крок — інтерактор SaveEnrollment — знаходить або створює запис про зарахування. Якщо студент уже записаний на курс, повертається відповідне повідомлення; в іншому разі запис зберігається, а викладачу надсилається сповіщення через механізм відкладеної доставки (deliver_later), що передає завдання у фонову чергу. Завершується сценарій перенаправленням користувача з повідомленням про успіх (рис. 2.8).

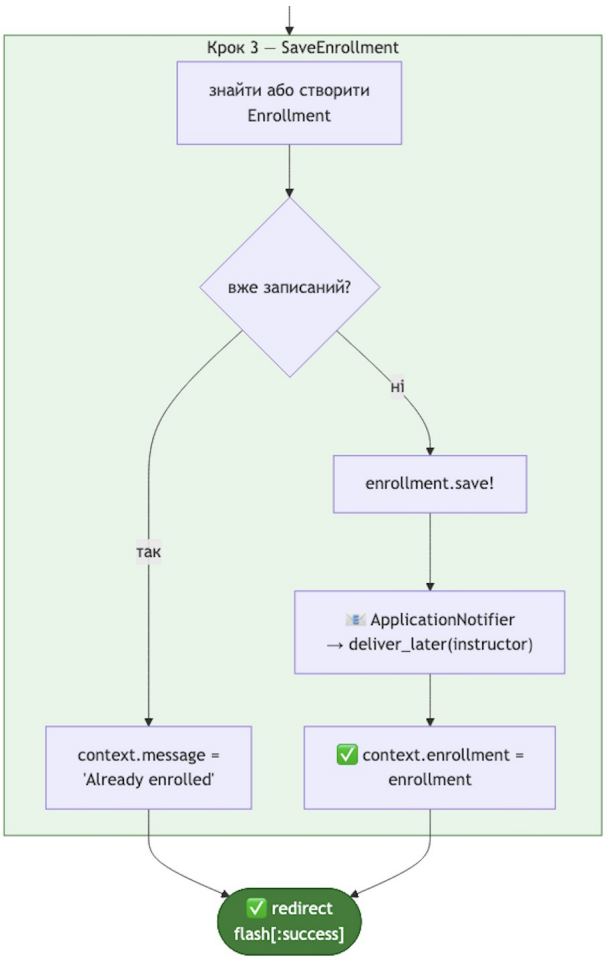

Рисунок 2.8 – Крок 3: збереження запису на курс (SaveEnrollment)

Варто наголосити на ще одній перевазі ланцюжкової організації інтеракторів — транзакційності та можливості відкату. Якщо третій крок завершується невдало вже після успішного виконання перших двох, організатор автоматично викликає метод відкату (rollback) для кожного раніше виконаного інтерактора у зворотному порядку. Це гарантує, що система ніколи не залишиться у неузгодженому проміжному стані: або сценарій виконується повністю, або не залишає жодних слідів. Така поведінка особливо важлива для операцій, що змінюють кілька пов’язаних записів.

Перевагами такого підходу є висока тестованість (кожен інтерактор перевіряється окремим модульним тестом), повторне використання (інтерактори можна комбінувати в інших сценаріях) та прозорість потоку

керування. Контролер при цьому залишається тонким: він лише викликає організатор і реагує на результат його виконання.

Маршрутизація запитів описана декларативно у конфігурації застосунку за принципами REST. Ресурсний підхід автоматично створює стандартний набір маршрутів для операцій над сутностями, а вкладені ресурси відображають ієрархію предметної області (теми та уроки в межах курсу). Фрагмент опису маршрутів має такий вигляд:

```
Rails.application.routes.draw do
  devise_for :users, controllers: {
    omniauth_callbacks: "users/omniauth_callbacks" }

  resources :courses do
    resources :topics do
      resources :lessons
    end
    resources :enrollments, only: [:create, :destroy]
    resources :invitations, only: [:create]
  end
  resources :messages, only: [:index, :create]
  resources :notifications, only: [:index]
end
```

Контролер запису на курс при цьому залишається мінімальним: він викликає організатор EnrollStudent із параметрами поточного користувача й курсу та формує відповідь залежно від успішності операції. Уся складність перевірок і збереження прихована в інтерактorах, що наочно демонструє перевагу обраної архітектури.

Окремою важливою підсистемою є автентифікація та контроль доступу. Технічну реалізацію високого рівня безпеки виконано за допомогою екосистеми бібліотек Devise та OmniAuth. Devise відповідає за класичну реєстрацію й вхід за допомогою пари «email — пароль», безпечне зберігання паролів у зашифрованому вигляді, відновлення доступу та керування сесіями. OmniAuth додає можливість входу через зовнішніх провайдерів.

Налаштування входу через зовнішнього провайдера здійснюється в конфігурації Devise, де указуються ідентифікатор та секретний ключ застосунку (що зберігаються у змінних середовища, а не в коді), запитувані дозволи та адреса зворотного виклику (рис. 2.9).

```ruby
Devise.setup do |config|
  config.omniauth :google_oauth2, ENV["GOOGLE_CLIENT_ID"], ENV["GOOGLE_CLIENT_SECRET"], {
    access_type: "offline",
    prompt: "select_account",
    scope: "email profile",
    redirect_uri: "http://localhost:3000/users/auth/google_oauth2/callback"
}
```

Рисунок 2.9 – Налаштування автентифікації через провайдер Google (OmniAuth)

Зберігання конфіденційних ключів у змінних середовища, а не у вихідному коді, є важливою практикою безпеки: вона унеможливлює випадкове розкриття секретів через систему контролю версій. Запитувані дозволи (scope) обмежено мінімально необхідним набором — електронною поштою та базовими даними профілю, що відповідає принципу найменших привілеїв.

Devise генерує також повний набір подань для сценаріїв автентифікації — реєстрації, входу, відновлення пароля та підтвердження облікового запису, — які за потреби можна кастомізувати під дизайн застосунку (рис. 2.10).

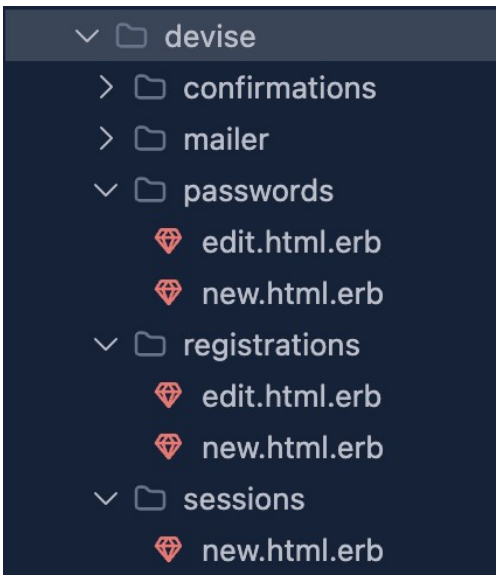

Рисунок 2.10 – Структура подань автентифікації, згенерованих Devise

Процес автентифікації через зовнішнього провайдера відбувається за таким алгоритмом: під час спроби входу клієнт перенаправляється на захищену сторінку провайдера (наприклад, Google, Facebook або GitHub), де підтверджує доступ до свого профілю. Після успішної перевірки провайдер

повертає користувача на спеціальний callback-URL застосунку разом із даними профілю. Система знаходить наявного користувача за полями provider та uid або створює нового, після чого встановлює авторизовану сесію. Такий підхід усуває потребу зберігати й перевіряти паролі сторонніх сервісів та підвищує зручність реєстрації.

Безпека сесій та захист від типових атак забезпечуються вбудованими механізмами фреймворку. Кожна форма, що змінює дані, містить токен захисту від міжсайтової підробки запитів (CSRF), який перевіряється сервером перед виконанням дії. Дані сесії зберігаються у підписаних та зашифрованих cookie, що унеможливлює їх підробку на боці клієнта. Паролі користувачів ніколи не зберігаються у відкритому вигляді — застосовується стійка криптографічна функція хешування з індивідуальною «сіллю» для кожного запису, що робить марними атаки за наперед обчисленими таблицями.

Розмежування прав доступу реалізовано на рівні контролерів та бізнес-логіки: перед виконанням дій, доступних лише певній ролі, система перевіряє роль поточного користувача й відмовляє у доступі за відсутності відповідних прав. Результат успішного запису студента на курс із боку інтерфейсу показано на рисунку 2.11.

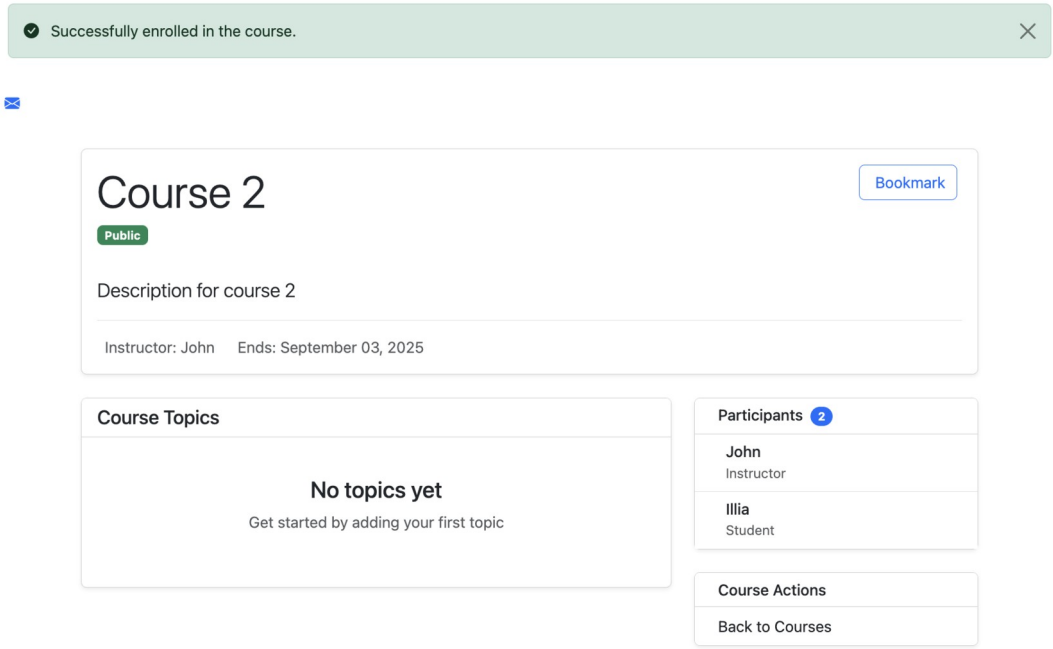

Рисунок 2.11 – Сторінка курсу після успішного запису студента

На сторінці курсу відображається його назва, ознака публічності, опис, відомості про викладача та дату завершення, а також список тем, перелік учасників із зазначенням їхніх ролей та панель доступних дій. Кнопка додавання до закладок дозволяє студенту зберегти курс для швидкого доступу.

2.5 Реалізація взаємодії в реальному часі та асинхронної обробки

Однією з ключових вимог до системи є забезпечення зворотного зв’язку в реальному часі. Цю функціональність реалізовано на основі технології ActionCable, що інтегрує протокол WebSocket із рештою застосунку. Взаємодія організована через канали: клієнт підписується на канал, а сервер транслює повідомлення всім підписникам.

Найпоказовішим прикладом є підсистема обміну повідомленнями. Компонент чату, реалізований засобами React, під час завантаження сторінки підключається до ActionCable через WebSocket, одразу виконує HTTP-запит на отримання історії повідомлень, після чого підписується на канал чату й слухає нові повідомлення в реальному часі. Інтерфейс чату наведено на рисунку 2.12.

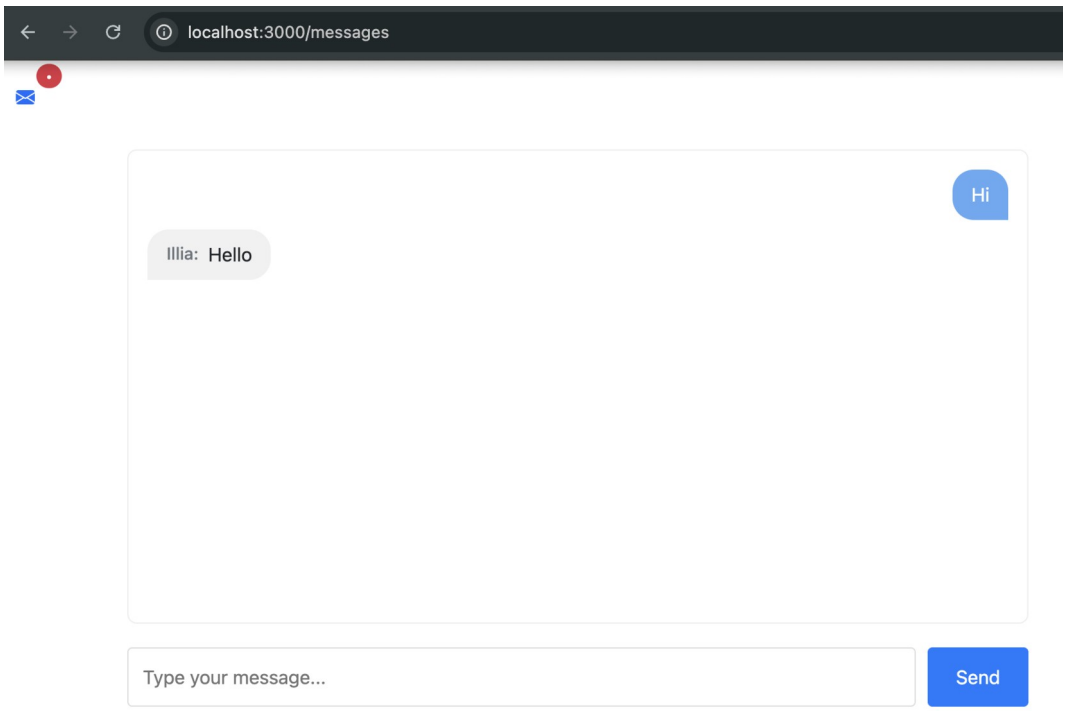

Рисунок 2.12 – Інтерфейс обміну повідомленнями в реальному часі

На серверному боці взаємодію в реальному часі реалізує канал ActionCable. Під час підписки клієнта канал визначає потік трансляції, а метод створення повідомлення транслює його всім підписникам. Спрощено логіку каналу та трансляції можна подати так:

```
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_room"
  end
end

# у моделі Message після збереження:
after_create_commit do
  ActionCable.server.broadcast("chat_room",
    { id: id, user: user.first_name, content: content })
end
```

Завдяки зворотному виклику after_create_commit трансляція відбувається лише після успішного завершення транзакції збереження, що гарантує узгодженість даних: клієнти отримують лише ті повідомлення, які справді збережені в базі.

Під час відправлення повідомлення компонент спочатку відображає його локально як тимчасове (так зване оптимістичне оновлення, optimistic update) — це створює відчуття миттєвої реакції без очікування відповіді

сервера. Коли сервер підтверджує збереження через трансляцію, тимчасовий запис замінюється реальним. На рівні моделі це забезпечується тим, що повідомлення після збереження одразу транслюється всім учасникам через ActionCable без додаткового HTTP-запиту. Структуру таблиці повідомлень наведено на рисунку 2.13.

```
create_table "messages", force: :cascade do |t|
  t.string "content"
  t.bigint "user_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index ["user_id"], name: "index_messages_on_user_id"
end
```

Рисунок 2.13 – Структура таблиці messages

Технологія ActionCable у своїй основі використовує механізм публікації та підписки (publish/subscribe) поверх сховища Redis. Коли сервер транслює повідомлення у певний потік, воно публікується в каналі Redis, на який підписані всі вебпроцеси застосунку. Завдяки цьому повідомлення доставляється навіть тим клієнтам, чиє WebSocket-з’єднання обслуговується іншим екземпляром сервера. Така архітектура є ключовою для горизонтального масштабування: вона дозволяє запускати кілька екземплярів застосунку, зберігаючи єдиний простір трансляцій реального часу.

Аналогічно реалізовано підсистему сповіщень: окремий клієнтський модуль підписується на канал сповіщень і за кожного нового сповіщення оновлює лічильник непрочитаних у шапці сайту без перезавантаження сторінки. Це забезпечує користувачеві миттєву поінформованість про нові події — оцінки, повідомлення чи запрошення.

Сповіщення в системі є самостійною сутністю, що дозволяє зберігати історію подій та відрізняти прочитані повідомлення від непрочитаних. Коли в системі відбувається значуща подія — виставлення оцінки, запрошення на курс, новий запис студента, — створюється відповідне сповіщення для зацікавленого користувача, яке одразу транслюється йому через канал реального часу та паралельно зберігається в базі даних. Завдяки цьому

користувач бачить нові сповіщення миттєво, а також може переглянути повний перелік минулих подій під час наступного входу. Запит на отримання кількості непрочитаних сповіщень повертає дані у форматі JSON, що дозволяє клієнтському модулю оновлювати лічильник без перезавантаження сторінки.

Окремо слід пояснити вибір гібридного підходу до побудови інтерфейсу. Більшість сторінок застосунку рендериться на сервері та оновлюється за допомогою технології Hotwire/Turbo, яка дозволяє підвантажувати фрагменти HTML без повного перезавантаження сторінки, зберігаючи при цьому простоту серверного підходу. Однак для двох сценаріїв — чату та лічильника сповіщень — потрібна складніша клієнтська логіка зі станом, оптимістичними оновленнями та постійним з'єднанням. Саме для них застосовано бібліотеку React. Такий збалансований підхід дозволяє уникнути ускладнення всього застосунку важким клієнтським кодом там, де він не потрібен, і водночас забезпечити багатий інтерактивний досвід там, де він критичний. Це рішення відповідає сучасній тенденції до помірного, точкового застосування фронтенд-фреймворків замість суцільної побудови односторінкових застосунків.

Другою складовою забезпечення продуктивності є механізм асинхронної обробки фонових завдань на основі зв'язки Sidekiq та Redis. У класичній синхронній веб-архітектурі виконання ресурсомістких операцій — наприклад, масової розсилки сповіщень або генерації звітів — блокує основний потік веб-сервера. Це призводить до критичних затримок у відповіді (timeout) та погіршення досвіду користувача. Винесення таких операцій у фонову чергу усуває цю проблему: контролер лише ставить завдання в чергу й одразу повертає відповідь, а саме завдання виконується окремим процесом Sidekiq.

Прикладом фонового завдання є нагадування студентам про наближення дедлайну курсу (рис. 2.14). Завдання отримує ідентифікатор курсу, знаходить курс і перебирає всіх його студентів, надсилаючи кожному

відповідний лист. Оскільки таких студентів можуть бути сотні, виконання операції у фоновому режимі є обов’язковим.

```ruby
class CourseReminderJob < ApplicationJob
  queue_as :default

  def perform(course_id)
    course = Course.find_by(id: course_id)
    @students = course.students
      You, 8 months ago • feat: Add reminer job to mail users about soon …
    @students.find_each do |student|
      CourseReminderMailer.course_reminder(course, student).deliver_now
    end
  end
end

```

Рисунок 2.14 – Фонове завдання нагадування про курс (CourseReminderJob)

Технічно фонове завдання оголошується як клас, що успадковує базовий клас завдань та визначає чергу виконання й метод perform із корисним навантаженням. Постановка завдання в чергу виконується одним викликом методу perform_later, після чого керування негайно повертається до контролера. Sidekiq, що працює окремим процесом, забирає завдання з черги у сховищі Redis та виконує його у одному з робочих потоків. Така модель забезпечує високу пропускну здатність: десятки завдань можуть оброблятися паралельно, не впливаючи на час відповіді основного вебсервера. Для відмовостійкості Sidekiq автоматично повторює невдалі завдання за наростаючим інтервалом, а стан черг можна відстежувати через вебінтерфейс моніторингу.

Фонові завдання застосовано й у сценарії запрошення студентів на приватні курси. Під час створення запрошення система генерує унікальний токен та у фоновому режимі надсилає студенту лист із персональним посиланням (рис. 2.15). Перехід за посиланням переводить запрошення у статус «прийнято» та автоматично записує студента на курс, повторно використовуючи розглянутий вище ланцюжок інтеракторів.

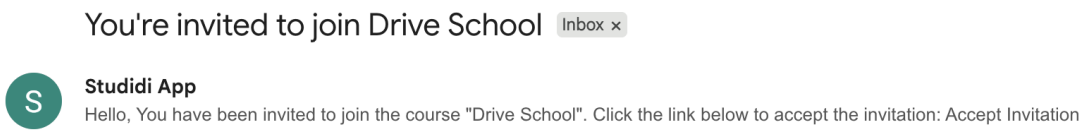

Рисунок 2.15 – Лист-запрошення на приватний курс

2.6 Обробка помилок, валідація та забезпечення цілісності даних

Надійність системи управління навчанням значною мірою визначається тим, наскільки послідовно вона забезпечує цілісність та коректність даних. У розробленому продукті застосовано багаторівневий підхід до валідації та обробки помилок, що поєднує захист на рівні бази даних, моделей та бізнес-логіки.

Перший рівень — рівень бази даних. Обмеження NOT NULL, унікальні індекси та зовнішні ключі гарантують узгодженість даних навіть у разі помилок у прикладному коді. Наприклад, унікальний складений індекс за парою «студент — курс» унеможливлює повторний запис студента на той самий курс на найнижчому рівні, незалежно від логіки застосунку.

Другий рівень — рівень моделей. Валідації Active Record перевіряють коректність даних перед збереженням: обов’язковість полів, формат електронної пошти, належність ролі до допустимого переліку значень, межі числових значень оцінок. У разі порушення валідації запис не зберігається, а об’єкт містить детальний перелік помилок, який передається користувачеві у зрозумілій формі.

Третій рівень — рівень бізнес-логіки. Інтерактори перевіряють складніші правила, що залежать від стану системи: доступність курсу, відповідність ролі дії, дотримання дедлайнів. У разі порушення такого правила інтерактор перериває виконання й повертає осмислене повідомлення про помилку, не вносячи жодних змін до бази даних завдяки транзакційності операцій.

Важливим аспектом обробки помилок є інформування користувача. Система чітко розрізняє помилки введення, спричинені діями користувача (некоректні дані, порушення бізнес-правил), та внутрішні помилки системи.

У першому випадку користувачеві показується зрозуміле повідомлення з поясненням причини та способу її усунення, а введені дані зберігаються, щоб не змушувати заповнювати форму повторно. У другому випадку користувач отримує загальне повідомлення, тоді як технічні подробиці фіксуються в журналі для подальшого аналізу розробником. Такий підхід поєднує зручність використання із захистом від розкриття внутрішньої будови системи, що могло б бути використане зловмисниками.

Окрему увагу приділено обробці помилок під час виконання фонових завдань та взаємодії з зовнішніми службами (поштовий сервер, провайдери автентифікації). Оскільки такі операції можуть завершуватися невдало з незалежних від системи причин, передбачено механізм повторних спроб та логування помилок, що дозволяє діагностувати й усувати збої без втрати даних. Сукупність описаних рівнів захисту забезпечує високу стійкість системи до некоректних даних і непередбачених ситуацій.

Висновки до розділу. У другому розділі деталізовано вимоги до програмного забезпечення, визначено акторів та прецеденти використання системи. Спроєктовано архітектуру на базі шаблону MVC із додатковим сервісним шаром інтеракторів, що забезпечує тонкі контролери та високу тестованість. Розроблено нормалізовану реляційну базу даних PostgreSQL та описано основні сутності предметної області. Програмно реалізовано ключову бізнес-логіку (запис на курс через ланцюжок інтеракторів), систему автентифікації на базі Devise та OmniAuth, взаємодію в реальному часі через ActionCable та асинхронну обробку фонових завдань за допомогою Sidekiq і Redis. Отримана архітектура повністю відповідає сформульованим функціональним і нефункціональним вимогам.

РОЗДІЛ 3
ТЕСТУВАННЯ, КОНТРОЛЬ ЯКОСТІ ТА РОЗГОРТАННЯ

3.1 Стратегії та середовища тестування

Для забезпечення високої надійності та стабільності роботи розробленої системи проведено комплексне тестування програмного продукту за допомогою фреймворку RSpec — найпоширенішого інструмента автоматизованого тестування в екосистемі Ruby. Автоматизованими тестами покрито більшу частину кодової бази, що включає перевірку ключової бізнес-логіки, коректність роботи моделей і контролерів, а також роботу фонових завдань.

Вибір методології тестування зумовлений як вимогами надійності, так і особливостями екосистеми. Застосовано підхід розроблення на основі поведінки (Behaviour-Driven Development, BDD), за якого тести формулюються як опис очікуваної поведінки системи природною мовою. Це робить тести не лише засобом перевірки, а й живою документацією: читаючи специфікації, можна зрозуміти, що саме система має робити у кожному сценарії. Фреймворк RSpec, обраний для реалізації цього підходу, надає виразний предметно-орієнтований синтаксис для опису прикладів поведінки та очікуваних результатів.

Стратегію тестування побудовано за принципом піраміди тестування, згідно з яким основу покриття становлять швидкі та численні модульні тести, а вершину — нечисленні інтеграційні та системні перевірки. Відповідно до цього в системі застосовано кілька рівнів тестів.

Модульні тести моделей перевіряють коректність валідацій, зв’язків між сутностями та бізнес-методів. Зокрема, тестується унікальність запису студента на курс, обов’язковість указання відповіді під час виставлення оцінки, межі допустимих значень оцінки (від 0 до 100) та коректність поліморфних зв’язків відповідей. Тести інтеракторів перевіряють кожен крок бізнес-сценаріїв окремо: наприклад, що інтерактор CheckStudentRole

перериває виконання за спроби запису з боку викладача, а CheckCourseAvailability — за недоступності курсу. Саме винесення логіки в інтерактори робить її зручною для ізольованого тестування.

Тести контролерів та запитів перевіряють коректність маршрутизації, кодів відповідей HTTP та, що особливо важливо, роботу системи розмежування прав доступу. Окрему увагу приділено перевірці того, що студент не може виконувати дії викладача (створювати курси, виставляти оцінки), а викладач — дії, призначені для студента. Тести фонових завдань переконуються, що відповідні завдання правильно ставляться в чергу та виконують очікувані дії — наприклад, надсилання нагадувань усім студентам курсу.

Розглянемо приклад модульного тесту для інтерактора перевірки ролі користувача. Тест описує очікувану поведінку у форматі, наближеному до природної мови, що характерно для методології розроблення на основі поведінки (BDD):

```
RSpec.describe Enrollments::CheckStudentRole do
  it "перериває контекст, якщо користувач не студент" do
    teacher = create(:user, role: "teacher")
    result = described_class.call(user: teacher)
    expect(result).to be_a_failure
    expect(result.message).to eq("Only students can enroll")
  end

  it "пропускає далі, якщо користувач студент" do
    student = create(:user, role: "student")
    result = described_class.call(user: student)
    expect(result).to be_a_success
  end
end
```

Наведений тест перевіряє обидві гілки логіки інтерактора: успішну та помилкову. Використання фабрик (create) дозволяє швидко готувати тестові дані, а виразні твердження (expect ... to) роблять намір тесту очевидним. Сукупність подібних тестів утворює надійну захисну сітку: будь-яка зміна, що порушує очікувану поведінку, негайно виявляється під час запуску набору тестів.

Кількісну оцінку покриття коду тестами отримано за допомогою інструмента SimpleCov, який автоматично формує звіт про частку виконаних під час тестування рядків коду. За результатами прогону повного набору тестів загальне покриття склало 79,52 %: із 1206 значущих рядків коду тестами охоплено 959, що для системи такого масштабу є високим показником. Деталізований звіт покриття за категоріями компонентів наведено на рисунку 3.1.

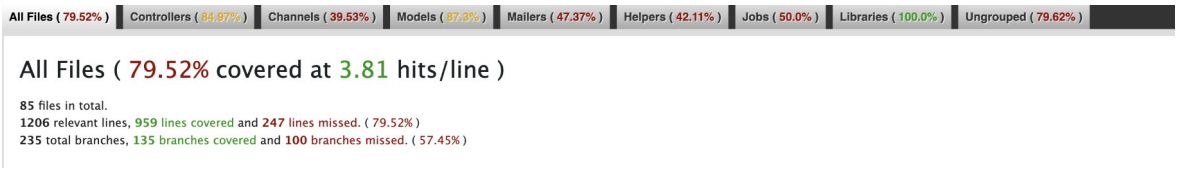

Рисунок 3.1 – Звіт про покриття коду автоматизованими тестами (SimpleCov)

Як видно зі звіту, найвищий рівень покриття мають найкритичніші для коректності складові: моделі предметної області (87,3 %) та контролери разом із системою розмежування доступу (84,97 %). Допоміжні бібліотеки покрито повністю (100 %). Нижчий рівень покриття каналів реального часу та поштових модулів пояснюється складністю автоматизованого тестування асинхронної взаємодії та зовнішніх інтеграцій; ці складові додатково перевірено вручну. Узагальнені показники покриття за категоріями подано у таблиці 3.1.

Таблиця 3.1 – Покриття компонентів системи автоматизованими тестами

Категорія компонентів	**Покриття, %**	**Тип тестів**
Моделі предметної області	87,30	Модульні
Контролери та доступ	84,97	Інтеграційні
Допоміжні бібліотеки	100,00	Модульні
Фонові завдання (Jobs)	50,00	Модульні
Поштові модулі (Mailers)	47,37	Модульні
Канали (ActionCable)	39,53	Інтеграційні
Загалом по системі	79,52	—

Тестування виконується в ізольованому тестовому середовищі з окремою базою даних, що очищається між тестами, та з підставними (mock) об’єктами для зовнішніх служб — поштового сервера й провайдерів

автентифікації. Це забезпечує детермінованість і повторюваність результатів незалежно від зовнішніх чинників.

3.2 Оцінювання якості та зручності використання

Окрім автоматизованого тестування, проведено ручне функціональне тестування та оцінювання зручності використання (юзабіліті) системи з позиції кінцевого користувача. Метою цього етапу була перевірка цілісності ключових сценаріїв роботи та відповідності інтерфейсу очікуванням користувачів.

У межах функціонального тестування перевірено наскрізні сценарії: повний шлях викладача від реєстрації до створення курсу, наповнення його уроками й тестами та виставлення оцінок; повний шлях студента від реєстрації до запису на курс, проходження тесту й отримання оцінки; а також сценарій запрошення на приватний курс і обміну повідомленнями в реальному часі. Усі основні сценарії відпрацювали коректно та узгоджено з очікуваною поведінкою системи.

Для кількісного оцінювання якості використано три показники: результативність виконання типових завдань, часову ефективність та суб’єктивну задоволеність користувачів. Результативність визначалася як частка успішно завершених сценаріїв, часова ефективність — як відношення часу виконання завдання користувачем до еталонного, а задоволеність вимірювалася за п’ятибальною шкалою. Узагальнені результати оцінювання наведено у таблиці 3.2.

Таблиця 3.2 – Результати оцінювання якості та зручності використання

Показник якості	Значення	Інтерпретація
Результативність виконання завдань	93,3 %	Високий рівень досягнення цілей
Часова відносна ефективність	80,7 %	Прийнятний рівень для нового користувача
Середній бал задоволеності	4,7 з 5	Позитивне сприйняття інтерфейсу
Частка коректно опрацьованих сценаріїв	100 %	Повна функціональна цілісність

Методику оцінювання побудовано на засадах міжнародного стандарту ISO 9241-11, що визначає зручність використання як міру, з якою продукт може бути використаний визначеними користувачами для досягнення визначених цілей із результативністю, ефективністю та задоволеністю у визначеному контексті використання. До тестування було залучено групу користувачів, які виконували типовий набір завдань — реєстрацію, запис на курс, проходження тесту та обмін повідомленнями — без попереднього навчання, що дозволило оцінити інтуїтивність інтерфейсу.

За результатами виконання практичних завдань загальна результативність становила 93,3 %, що свідчить про високий рівень досягнення користувачами поставлених цілей. Часова відносна ефективність склала 80,7 %, що є прийнятним показником, оскільки частина часу витрачалася на ознайомлення з новим інтерфейсом. Середній бал задоволеності користувачів становив 4,7 з 5, що підтверджує зручність, зрозумілість і позитивне сприйняття інтерфейсу. Виявлені під час тестування незначні зауваження було усунено, що дозволило підвищити загальну ергономічність системи.

3.3 Розгортання застосунку та налаштування CI/CD

Фінальним етапом життєвого циклу розроблення стало розгортання застосунку в робочому середовищі та налаштування процесу безперервної інтеграції й розгортання (CI/CD). На відміну від суто клієнтських застосунків, повностекова система на базі Ruby on Rails потребує розгортання кількох взаємопов’язаних компонентів: вебсервера застосунку, СКБД PostgreSQL, сховища Redis та процесу обробки фонових завдань Sidekiq.

Конфігурація різних середовищ виконання — розроблення, тестування та експлуатації — відокремлена від коду й керується через змінні середовища. Конфіденційні дані (паролі бази даних, ключі провайдерів автентифікації, секрет для підпису сесій) ніколи не потрапляють у систему контролю версій, а передаються застосунку у захищений спосіб під час

розгортання. Такий підхід відповідає методології «дванадцятифакторного застосунку» (Twelve-Factor App) і є галузевим стандартом побудови хмарних сервісів. У робочому середовищі додатково вмикається примусове використання захищеного протоколу HTTPS, кешування статичних ресурсів та компіляція клієнтських активів, що підвищує продуктивність і безпеку.

Для забезпечення відтворюваності середовища та спрощення розгортання застосунок контейнеризовано за допомогою технології Docker. Кожен компонент системи запускається у власному контейнері, а їхня узгоджена робота описується засобами оркестрації контейнерів. Такий підхід гарантує ідентичність середовищ розроблення та експлуатації й усуває класову проблему «на моїй машині працює».

Для автоматизації процесу оновлення налаштовано конвеєр CI/CD на базі сервісу GitHub Actions. Загальна послідовність процесу така: розробник вносить зміни до коду та відправляє їх у централізований репозиторій на платформі GitHub. Кожне відправлення до основної гілки автоматично запускає конвеєр, який послідовно виконує кілька етапів: встановлення залежностей, статичний аналіз коду (лінтинг), запуск повного набору автоматизованих тестів RSpec та, у разі успішного проходження всіх перевірок, складання образу й розгортання нової версії застосунку.

Структуру конвеєра описано декларативно у конфігураційному файлі робочого процесу. Він визначає тригер запуску (відправлення коду до репозиторію), середовище виконання та послідовність кроків. Спрощено опис основних етапів складання та тестування має такий вигляд:

```
name: CI
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    services:
      postgres: { image: postgres:16 }
      redis: { image: redis:7 }
    steps:
      - uses: actions/checkout@v4
      - uses: ruby/setup-ruby@v1
      - run: bundle install
```

```
      - run: bundle exec rubocop
      - run: bundle exec rspec
```

Як видно з конфігурації, конвеєр піднімає допоміжні служби PostgreSQL та Redis, встановлює залежності, виконує статичний аналіз коду інструментом RuboCop та запускає повний набір тестів RSpec. Лише за умови успішного проходження всіх кроків виконується подальше розгортання.

Такий конвеєр повністю нівелює ризики людського фактора під час розгортання: жодна зміна, що не пройшла автоматизованих тестів, не потрапляє в робоче середовище. Це гарантує стабільність робочої версії та дозволяє безпечно й часто доставляти оновлення користувачам. Поєднання контейнеризації та автоматизованого конвеєра забезпечує масштабованість системи: за потреби кількість екземплярів вебсервера або обробників фонових завдань можна збільшити без зміни коду застосунку.

3.4 Моніторинг, логування та перспективи масштабування

Введення системи в експлуатацію не завершує життєвого циклу розроблення, а лише переводить його у фазу супроводу. Для забезпечення безперебійної роботи передбачено засоби моніторингу та логування, що дозволяють своєчасно виявляти й усувати проблеми у робочому середовищі.

Логування реалізовано на кількох рівнях: вебсервер фіксує всі вхідні запити та коди відповідей, застосунок записує ключові бізнес-події та помилки, а процес Sidekiq веде журнал виконання фонових завдань. Структуровані журнали дозволяють відстежувати поведінку системи в часі та проводити ретроспективний аналіз інцидентів. Стан черг фонових завдань та статистику їх виконання можна спостерігати через вебінтерфейс моніторингу Sidekiq, що відображає кількість оброблених, запланованих та невдалих завдань.

Окремо контролюються показники продуктивності: час відповіді сервера, навантаження на базу даних та використання пам’яті. Раннє

виявлення відхилень цих показників дозволяє запобігти деградації якості сервісу до того, як вона стане помітною для користувачів.

Щодо перспектив масштабування, обрана архітектура закладає кілька рівнів розвитку. Горизонтальне масштабування вебсерверів досягається запуском додаткових екземплярів застосунку за балансувальником навантаження, оскільки застосунок спроєктовано як переважно безстановий. Масштабування обробки фонових завдань забезпечується збільшенням кількості процесів та потоків Sidekiq. Масштабування бази даних можливе через реплікацію для розподілу навантаження на читання. Таким чином, система здатна зростати разом зі збільшенням кількості користувачів без принципової зміни архітектури, що підтверджує правильність прийнятих проєктних рішень.

Висновки до розділу. У третьому розділі описано стратегію та реалізацію тестування системи за допомогою фреймворку RSpec на кількох рівнях — від модульних тестів моделей та інтеракторів до перевірки розмежування доступу й фонових завдань. Проведено функціональне тестування та оцінювання зручності використання, які підтвердили високу результативність (93,3 %) та задоволеність користувачів (4,7 з 5). Налаштовано розгортання застосунку з використанням контейнеризації Docker та конвеєра CI/CD на базі GitHub Actions, що забезпечує надійну та автоматизовану доставку оновлень. Результати розділу підтверджують, що розроблена система є функціонально завершеною, надійною та придатною до промислової експлуатації.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено повнофункціональну систему управління навчанням (LMS) — сучасний full-stack вебзастосунок, що реалізує повний навчальний цикл та усуває ключові недоліки наявних рішень. Проведений аналіз предметної області підтвердив актуальність створення такого інструменту для організації дистанційного та змішаного навчання. Під час роботи було успішно вирішено всі поставлені завдання.

1. Проаналізовано предметну область систем управління навчанням, визначено їхню роль, сфери застосування та ключові бізнес-процеси. Здійснено порівняльний огляд наявних рішень (Moodle, Google Classroom, Coursera), який обґрунтував доцільність розроблення власного продукту, що поєднує гнучкість професійних LMS із простотою хмарних сервісів.

2. Обґрунтовано вибір технологічного стека: монолітна архітектура на базі фреймворку Ruby on Rails, реляційна СКБД PostgreSQL, технологія реального часу ActionCable, система фонової обробки Sidekiq із Redis та екосистема автентифікації Devise і OmniAuth.

3. Спроєктовано нормалізовану реляційну базу даних та об’єктну модель предметної області, що охоплює користувачів, курси, теми, уроки, тести, відповіді, оцінки, записи на курси, запрошення та повідомлення з коректними зв’язками й обмеженнями цілісності.

4. Реалізовано серверну частину з розмежуванням ролей викладача та студента й винесенням бізнес-логіки в окремий сервісний шар на основі патерну «Інтерактор», що забезпечило тонкі контролери та високу тестованість коду.

5. Налаштовано REST API та взаємодію в реальному часі на основі WebSocket (ActionCable), завдяки чому реалізовано інтерактивний чат і миттєві сповіщення без перезавантаження сторінки, а також гнучку автентифікацію через зовнішніх провайдерів.

6. Реалізовано асинхронну обробку фонових завдань (розсилки, нагадування про дедлайни, архівування курсів) за допомогою Sidekiq і Redis, що усунуло блокування веб-сервера під час масових операцій та підвищило продуктивність системи.

7. Розроблений функціонал покрито автоматизованими тестами RSpec, проведено функціональне тестування та оцінювання зручності використання, які підтвердили надійність, коректність розмежування доступу та високу задоволеність користувачів. Налаштовано конвеєр CI/CD для автоматизованого розгортання застосунку.

Окремо слід відзначити теоретичну та практичну цінність прийнятих архітектурних рішень. Застосування патерну «Інтерактор» для організації бізнес-логіки, використання технології реального часу на основі WebSocket та винесення ресурсомістких операцій у фонові черги утворюють референсний набір практик, придатний для повторного використання у широкому класі вебзастосунків із насиченою предметною логікою. Розроблена система демонструє, що поєднання цих практик дозволяє досягти водночас високої швидкості розроблення, надійності та масштабованості — властивостей, які традиційно вважаються такими, що конкурують між собою.

З практичного погляду розроблений продукт є самодостатнім інструментом, готовим до використання реальними навчальними колективами. Він покриває повний навчальний цикл, забезпечує безпеку персональних даних, надає сучасний інтерактивний інтерфейс та здатний обслуговувати зростальну кількість користувачів без зміни архітектури. Це підтверджує, що мету кваліфікаційної роботи досягнуто в повному обсязі.

Отримані результати підтверджують, що розроблений програмний продукт повністю відповідає поставленим вимогам і може ефективно використовуватися для організації навчального процесу. Проведена робота також дозволила окреслити напрями подальшого вдосконалення: розширення аналітики навчального прогресу, додавання відеоконференцій, інтеграцію систем перевірки робіт на плагіат та розроблення мобільного клієнта.

Водночас ці напрями не є критичними для базової працездатності системи та належать до перспектив подальшого розвитку програмного продукту.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Hartl M. Ruby on Rails Tutorial: Learn Web Development with Rails. 7th ed. Boston : Addison-Wesley Professional, 2022. 1056 p.

2. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.

3. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.

4. Ruby on Rails Guides. URL: https://guides.rubyonrails.org/ (дата звернення: 03.06.2026).

5. Active Record Basics. Ruby on Rails Guides. URL: https://guides.rubyonrails.org/active_record_basics.html (дата звернення: 03.06.2026).

6. Action Cable Overview. Ruby on Rails Guides. URL: https://guides.rubyonrails.org/action_cable_overview.html (дата звернення: 03.06.2026).

7. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/ (дата звернення: 03.06.2026).

8. Date C. J. An Introduction to Database Systems. 8th ed. Boston : Pearson, 2003. 1024 p.

9. Sidekiq Documentation. URL: https://github.com/sidekiq/sidekiq/wiki (дата звернення: 03.06.2026).

10. Redis Documentation. URL: https://redis.io/docs/ (дата звернення: 03.06.2026).

11. Devise. Flexible authentication solution for Rails. GitHub repository. URL: https://github.com/heartcombo/devise (дата звернення: 03.06.2026).

12. OmniAuth. Standardized multi-provider authentication. URL: https://github.com/omniauth/omniauth (дата звернення: 03.06.2026).

13. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force, 2012. DOI: 10.17487/RFC6749.

14. Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. Internet Engineering Task Force, 2011. DOI: 10.17487/RFC6455.

15. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Ph.D. dissertation. University of California, Irvine, 2000. 162 p.

16. Interactor. Simple, single-purpose objects for Ruby. GitHub repository. URL: https://github.com/collectiveidea/interactor (дата звернення: 03.06.2026).

17. RSpec Documentation. URL: https://rspec.info/documentation/ (дата звернення: 03.06.2026).

18. Chelimsky D., Astels D., Helmkamp B. The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends. Raleigh : Pragmatic Bookshelf, 2010. 448 p.

19. React Documentation. URL: https://react.dev/ (дата звернення: 03.06.2026).

20. Hotwire. HTML over the wire. URL: https://hotwired.dev/ (дата звернення: 03.06.2026).

21. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol : O’Reilly Media, 2020. 706 p.

22. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. 2014. Vol. 2014, Issue 239. Article 2.

23. GitHub Actions Documentation. URL: https://docs.github.com/en/actions (дата звернення: 03.06.2026).

24. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2010. 512 p.

25. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Geneva : International Organization for Standardization, 2023.

26. ISO 9241-11:2018. Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. Geneva : International Organization for Standardization, 2018. 31 p.

27. Nielsen J. Usability Engineering. San Francisco : Morgan Kaufmann, 1993. 362 p.

28. OWASP Top 10:2021. The Ten Most Critical Web Application Security Risks. OWASP Foundation. URL: https://owasp.org/Top10/ (дата звернення: 03.06.2026).

29. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide. 2nd ed. Boston : Addison-Wesley, 2005. 496 p.

30. Kohan D. Modern Learning Management Systems: Design Principles and Pedagogical Foundations. International Journal of Educational Technology. 2021. Vol. 18, No. 2. P. 45–63.

ДОДАТКИ

Додаток А – Посилання на репозиторій та розгорнутий програмний продукт

Програмний код розробленого проєкту розміщено на платформі GitHub за адресою https://github.com/teterailiya/lms-platform. Розгорнутий вебсайт системи управління навчанням доступний за відповідним посиланням, наданим у супровідних матеріалах до кваліфікаційної роботи.

Додаток Б – Лістинг ключових програмних модулів

У цьому додатку наведено вихідний код ключових програмних модулів системи, що ілюструють застосовані архітектурні рішення.

Б.1 Організатор та інтерактори запису студента на курс

```
module Enrollments
  class EnrollStudent
    include Interactor::Organizer
    organize CheckStudentRole, CheckCourseAvailability,
SaveEnrollment
  end

  class CheckStudentRole
    include Interactor
    def call
      unless context.user.student?
        context.fail!(message: "Only students can enroll")
      end
    end
  end

  class CheckCourseAvailability
    include Interactor
    def call
      course = context.course
      unless course.public? && !course.is_archived?
        context.fail!(message: "Course not available")
      end
    end
  end

  class SaveEnrollment
    include Interactor
    def call
      enrollment = Enrollment.find_or_initialize_by(
        user: context.user, course: context.course)
      if enrollment.persisted?
        context.message = "Already enrolled"
      else
        enrollment.save!
        ApplicationNotifier.with(course: context.course)
          .deliver_later(context.course.instructor)
        context.enrollment = enrollment
      end
    end
  end
end
```

Б.2 Модель повідомлення з трансляцією в реальному часі

```
class Message < ApplicationRecord
```

```
  belongs_to :user
  validates :content, presence: true

  after_create_commit do
    ActionCable.server.broadcast(
      "chat_room",
      { id: id, user: user.first_name, content: content })
  end
end
```

Б.3 Фонове завдання нагадування про курс

```
class CourseReminderJob < ApplicationJob
  queue_as :default

  def perform(course_id)
    course = Course.find_by(id: course_id)
    return unless course
    course.students.find_each do |student|
      CourseReminderMailer
        .course_reminder(course, student).deliver_now
    end
  end
end
```

Б.4 Модель користувача з підтримкою входу через зовнішнього провайдера

```
class User < ApplicationRecord
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable,
         :omniauthable, omniauth_providers: [:google_oauth2]

  has_many :courses, foreign_key: :instructor_id
  has_many :enrollments, dependent: :destroy
  has_many :enrolled_courses, through: :enrollments,
           source: :course

  enum role: { student: "student", teacher: "teacher" }

  def self.from_omniauth(auth)
    where(provider: auth.provider, uid: auth.uid)
      .first_or_create do |user|
      user.email = auth.info.email
      user.password = Devise.friendly_token[0, 20]
      user.first_name = auth.info.first_name
      user.last_name = auth.info.last_name
      user.avatar = auth.info.image
    end
  end
end
```

Б.5 Обробник зворотного виклику автентифікації через зовнішнього провайдера

```
class Users::OmniauthCallbacksController <
    Devise::OmniauthCallbacksController
  def google_oauth2
    @user = User.from_omniauth(request.env["omniauth.auth"])
    if @user.persisted?
      sign_in_and_redirect @user, event: :authentication
    else
      redirect_to new_user_registration_url
    end
  end
end
```

Б.6 Міграція створення таблиці записів на курс із обмеженням унікальності

```
class CreateEnrollments < ActiveRecord::Migration[7.1]
  def change
    create_table :enrollments do |t|
      t.references :user, null: false, foreign_key: true
      t.references :course, null: false, foreign_key: true
      t.timestamps
    end
    add_index :enrollments, [:user_id, :course_id],
              unique: true
  end
end
```