Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

«Затверджую»
В.о. зав. кафедри комп'ютерних систем
та робототехніки
_____ к.ф.-м.н., доц. М.М. Хруслов
«___» _____ 2025 р.

Пояснювальна записка

до кваліфікаційної роботи
бакалавра

на тему: **«КОМП'ЮТЕРНА МОДЕЛЬ СИСТЕМИ АВТОМАТИЧНОГО ОЦІНЮВАННЯ КОДУ ТА ПЕРЕВІРКА СТУДЕНТСЬКИХ РОБІТ НА ПЛАГІАТ»**

Спеціальність 123 – Комп'ютерна інженерія
Галузь знань 12 – Інформаційні технології
Освітня програма «Комп'ютерна інженерія»

Захищено на засіданні
Екзаменаційної комісії № 44
протокол № __ від __.06.2025 р.
Оцінка _____ / _____
Голова Екзаменаційної комісії
_____ЧУГАЙ А.М.

Виконав:
студент 4 курсу, групи КІ–41
групи КІ–41 **КОНОНЕНКО Михайло Олексійович**_____

Керівник: к.е.н, доцент
закладу вищої освіти кафедри
комп'ютерних систем та робототехніки
ЧУБ Ольга Ігорівна_____

Рецензент: професор кафедри
теоретичної та прикладної інформатики
Харківського національного університету
імені В.Н. Каразіна, д.т.н., професор
ФРОЛОВ В'ячеслав Вікторович_____

Харків – 2025

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 82 сторінки, із яких 49 сторінок основної частини з 26 рисунками, 5 таблицями, 41 найменуваннями списку використаних джерел та 4 додатками.

Об'єкт дослідження – розробка комп'ютерної моделі системи автоматичного оцінювання програмного коду та перевірки студентських робіт на плагіат із використанням технологій веб-розробки та алгоритмів аналізу програмного коду.

Об'єкт дослідження – процес автоматизованого оцінювання студентських програмних рішень та виявлення плагіату у вихідному коді.

Предмет дослідження – методи та механізми автоматизованого аналізу програмного коду, а також алгоритми виявлення плагіату.

Проблема, що вирішується у роботі, полягає в підвищенні об'єктивності процесів перевірки та оцінювання студентських програмних робіт шляхом автоматизації аналізу коду і виявлення плагіату.

Область застосування – комп'ютерна інженерія сфера вищої освіти, зокрема заклади, що здійснюють підготовку ІТ-фахівців. Розроблена система може бути інтегрована в навчальні платформи, онлайн-курси, системи дистанційного навчання та використовуватися для автоматизованого контролю знань у дисциплінах, пов'язаних із програмуванням.

Ключові слова: автоматичне оцінювання коду, перевірка плагіату, модульна архітектура, тестові сценарії, ізольоване виконання коду, виявлення схожості.

ABSTRACT

An explanatory note to the master's attestation work is created in the introduction, three sections, conclusions, a list of sources used and four additional substances. The total volume of work is 80 pages, of which 49 pages of the main part with 26 figures, 5 table, 41 names of the list of used sources and two additions.

The aim of the qualification work is to develop a computer model of a system for automatic code evaluation and plagiarism detection in student assignments, using web development technologies and code analysis algorithms.

Object of the research is the process of automated assessment of student programming solutions and detection of plagiarism in source code.

Subject of the research is methods and mechanisms of automated code analysis, as well as algorithms for detecting code similarity and plagiarism.

The problem addressed in this work lies in improving the objectivity of the assessment and verification processes for student programming assignments through the automation of code analysis and plagiarism detection.

Scope is computer engineering, higher education, particularly institutions training IT specialists. The developed system can be integrated into learning platforms, online courses, distance learning systems, and used for automated knowledge control in programming-related courses.

The study explores the issue of ensuring objective evaluation and maintaining academic integrity in remote and blended learning environments. The work analyzes key stages of the learning process: assignment creation by instructors, code submission by students, automated testing, and comprehensive plagiarism detection.

Keywords: AUTOMATIC CODE EVALUATION, PLAGIARISM DETECTION, MODULAR ARCHITECTURE, TEST SCENARIOS, ISOLATED CODE EXECUTION, CODE SIMILARITY ANALYSIS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

Скорочення	Визначення
АП	Автотестування програм
АС	Автоматизована система
ПП	Перевірка на плагіат
DB	База даних
FS	Файлове сховище
UI	Інтерфейс користувача
API	Інтерфейс програмування застосунків
AST	Абстрактне синтаксичне дерево
RCE	Ізольоване виконання коду (Remote Code Execution)
Wn	Алгоритм Winnowing
NG	Аналіз n-грам
Tk	Токенізація коду
Dif	Метод символ-до-символу Difflib
MVC	«Модель-Подання-Контролер»
LMS	Система управління навчанням Learning Management System
Docker	Контейнеризація середовища розробки
JSON	Нотація об’єктів JavaScript
CLI	Інтерфейс командного рядка
HTTP	Протокол передачі гіпертексту
HTTPS	Захищений протокол передачі гіпертексту
PLATO	Programmed Logic for Automated Teaching Operations
CIA	Computer-Assisted Instruction
ORM	Object-Relational Mapping
COC	Convention over configuration

ВСТУП

Актуальність теми дослідження зумовлена поширенням дистанційного та змішаного навчання, в умовах якого істотно зростає кількість програмних завдань, що підлягають перевірці. За відсутності автоматизованих інструментів ця перевірка потребує значних часових затрат з боку викладачів і часто супроводжується ризиком суб'єктивного оцінювання. Це актуалізує потребу у створенні технологічних рішень, здатних забезпечити стабільну й масштабовану підтримку процесів оцінювання програмних робіт.

Значною проблемою в освітньому процесі залишається виявлення неавторських рішень, зокрема копіювання програмного коду, що ускладнюється зростанням обсягів поданих робіт. Наявні засоби автотестування здебільшого не дозволяють фіксувати схожість між текстами коду, зосереджуючись лише на перевірці правильності виконання. Це створює потребу в інтеграції алгоритмів виявлення схожості коду, здатних аналізувати не лише синтаксичні, а й структурні та семантичні характеристики рішень.

Автоматизована система, що поєднує автотестування з механізмами виявлення плагіату, здатна оптимізувати освітній процес, зменшити навантаження на викладачів і сприяти дотриманню принципів академічної доброчесності. Розробка такої платформи потребує обґрунтованого вибору архітектурних рішень, алгоритмів аналізу коду та методів безпечного виконання програм.

Мета кваліфікаційної роботи – розробити комп'ютерну модель системи автоматичного оцінювання програмного коду та перевірки студентських робіт на плагіат із використанням технологій веб-розробки та алгоритмів аналізу програмного коду.

Для досягнення поставленої мети в роботі було визначено такі **основні завдання**:

1) проаналізувати сучасні методи виявлення плагіату в програмному коді та механізми його автоматичного оцінювання;

2) дослідити питання захисту даних у веб-системах та обрати відповідні криптографічні технології для забезпечення безпеки;
3) розробити модель захисту даних користувачів із використанням сучасних засобів шифрування та врахуванням можливих вразливостей;
4) спроектувати архітектуру системи, що включає фронтенд, бекенд, базу даних, механізми тестування коду та перевірки на плагіат;
5) розробити функціональний інтерфейс користувача для подання, перегляду та перевірки кодових завдань;
6) реалізувати серверну частину системи з можливістю обробки й збереження робіт, автоматичного оцінювання та виявлення плагіату;
7) інтегрувати механізм виконання коду в реальному часі з підтримкою перевірки коректності розв’язків;
8) розробити API для взаємодії з іншими сервісами та забезпечення масштабованості системи;
9) провести тестування роботи системи, оцінити точність виявлення плагіату та її стабільність під навантаженням;
10) створити документацію до системи та сформулювати рекомендації щодо її впровадження та подальшого розвитку.

Запропоноване рішення дозволяє зменшити навантаження на викладачів, підвищити швидкість перевірки, мінімізувати людський фактор у виставленні оцінок та забезпечити дотримання принципів академічної доброчесності. Інструментарій, створений у межах дослідження, може бути адаптований для різних освітніх платформ і дисциплін, пов’язаних із програмуванням, що підкреслює універсальність і практичну придатність результатів роботи.

РОЗДІЛ 1.
АНАЛІЗ ДОСЛІДЖЕНЬ У СФЕРІ АВТОМАТИЧНОГО ОЦІНЮВАННЯ КОДУ ТА АНТИПЛАГІАТНИХ СИСТЕМ

Нові вимоги до якості викладання та об’єктивності оцінювання програмних завдань диктують необхідність створення гнучкої й масштабованої платформи.

1.1 Роль автоматизованих систем перевірки знань у сучасній освіті

Автоматизовані системи перевірки знань поєднали в собі інформаційні технології та педагогічні практики задля підвищення ефективності оцінювання та зворотного зв’язку.

Перші кроки датуються 1960-1970-ми роками, коли на університетських дослідницьких проектах, таких як PLATO [1], почали використовувати термінали для тестування студентів у формі простих тестів із вибором відповіді.

Перехід до мережевого середовища в 1990-х відкрив нові можливості: з’явилися платформи дистанційного навчання e-learning [5], де студенти могли завантажувати відповіді й автоматично отримувати простий зворотний зв’язок.

Однією з перших масових систем стала Blackboard [2], яка поєднала управління курсами з модулем тестування. У 2000-х роках поширилися відкриті LMS-платформи – Moodle[3], Sakai[4] та інші – із розширеними можливостями конфігурації автоматичних тестів, гнучкими налаштуваннями оцінювання та інтеграцією з зовнішніми серверами тестів.

Сучасні автоматизовані системи перевірки знань підтримують не лише закриті й відкриті питання, а й складні сценарії тестування практичних завдань. Сучасні автоматизовані системи перевірки знань дозволяють описати критерії оцінювання у вигляді скриптів, виконувати їх проти робіт студентів, аналізувати результати й формувати детальні звіти.

1.2 Проблеми оцінювання студентських програмних завдань вручну

Ручна перевірка вихідного коду студентських робіт залишається поширеним підходом, проте вона супроводжується низкою труднощів. Перевірка програмних завдань у групах понад 50 студентів стає вкрай неефективною через значні часові та кадрові витрати [12]. Викладач чи асистент щотижня витрачає від 10 до 20 годин виключно на рутинну перевірку вихідного коду та складання коментарів [13].

У таблиці 1.1 наведено порівняльну таблицю оцінки витрат часу та ресурсів при ручному і автоматизованому підходах для різного розміру групи. При зростанні кількості студентів автоматизована система забезпечує багаторазове зменшення трудовитрат та прискорює комунікацію результатів [12].

Дані в таблиці отримано на основі внутрішнього опитування викладачів факультету комп'ютерних наук ХНУ ім. В. Н. Каразіна та аналізу часових витрат на перевірку домашніх завдань під час двох семестрів 2021 року.

Для ручного підходу враховані середні показники зафіксовані під час моніторингу часу роботи викладачів і асистентів, а для автоматизованої перевірки – результати тестування прототипу платформи на реальному наборі студентських рішень.

Таблиця 1.1

Оцінка витрат часу та ресурсів при ручному і автоматизованому підходах до перевірки

Розмір групи студенів	Ручна перевірка: час на роботу (хв/завдання)	Ручна перевірка: загальний час (год/тиждень)	Автоматизована перевірка: час на роботу (хв/завдання)	Автоматизована перевірка: загальний час (год/тиждень)
30	15	7.5	2	1
100	15	25	2	3.3
300	15	75	2	10

1.3 Класифікація методів на виявлення запозичень

Розглянемо підходи на основі рядкового порівняння, аналізу токенів і синтаксичних дерев, а також семантичні і гібридні методи.

1.3.1 Методи на основі порівняння рядків

Методи, які працюють на рівні буквального порівняння послідовностей символів, залишаються одними з найшвидших засобів для попереднього відсіву явного копіювання в текстах і програмних кодах [14].

Застосування таких алгоритмів на рис. 1.1 [15] дозволяє виявляти ділянки однакового вмісту без необхідності розбирати структуру коду або враховувати його логічне призначення. За допомогою цих прийомів виявляють роботи, які були скопійовані без змін у відступах чи коментарях.

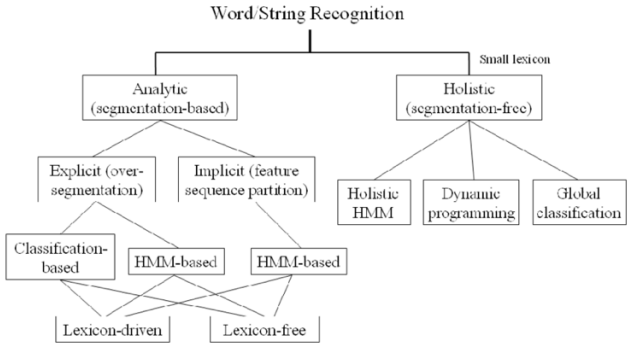

Рисунок 1.1 – Огляд методів на основі порівняння рядків

На початковому етапі весь вихідний файл перетворюють у безперервний рядок символів. Видаляють зайві пробіли, символи нового рядка та позначки коментарів.

Найпростіший спосіб пошуку однакових підрядків базується на переборі усіх можливих вікон заданої довжини k у головному рядку.

Для кожного такого вікна відбувається пряме порівняння з усіма вікнами у документі-оригіналі. Хоча в теорії цей підхід може вимагати багато часу при довгих текстах, на практиці він підходить для невеликих файлів або служить етапом початкового фільтрування підозрілих документів.

На рис. 1.2 описаний метод пошуку однакових підрядків у майбутній системі.

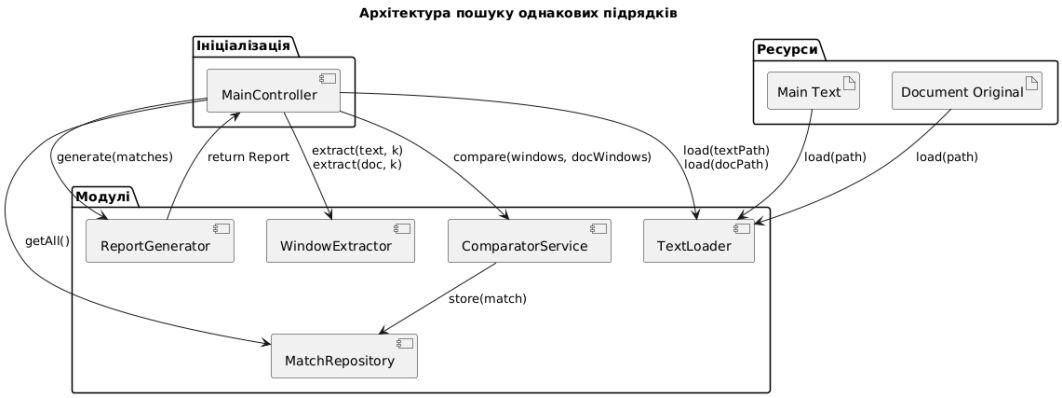

Рисунок 1.2 – Архітектура програми для пошуку однакових підрядків у коді

Попри свою швидкість, порівняння рядків має обмеження. Навіть незначні вставки або видалення символів можуть порушити суцільність підрядка і приховати справжній збіг.

1.3.2 Токен-орієнтовані підходи

Токен-орієнтований аналіз починається з лексичного розбору коду або тексту на окремі елементи – ключові слова, ідентифікатори та оператори на рис. 1.3 [16].

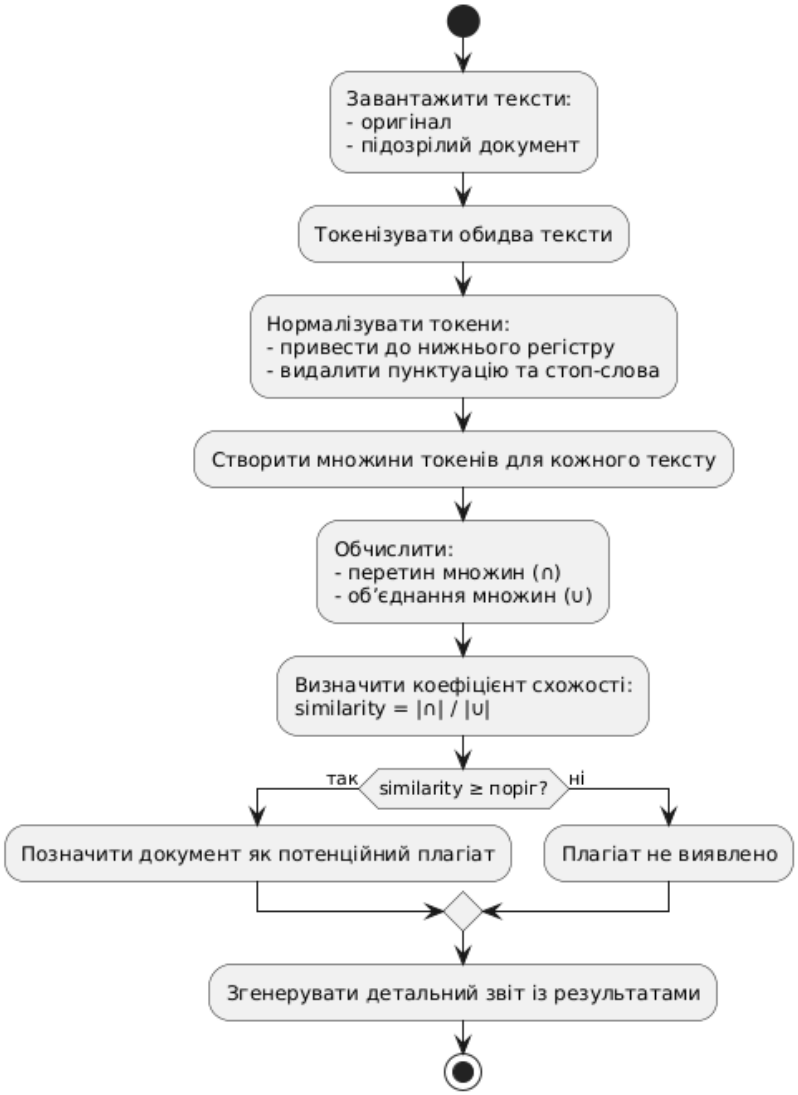

Рисунок 1.3 – Порівняння тексту за допомогою токенів

Після цього на рис. 1.3 утворюється послідовність токенів яка точно відображає логіку програми без зайвих форматувальних символів і коментарів. При цьому будь-які знаки пробілу або переносу рядка видаляються щоб не впливати на результати подальшого порівняння [17].

На наступному етапі на рис. 1.3 для порівняння двох послідовностей токенів застосовують алгоритм пошуку найдовшої спільної підпослідовності. Він формує таблицю в якій показано співпадіння префіксів двох токен-строк і забезпечує виявлення максимальних збігів навіть якщо частина коду була змінена або доповнена новими рядками.

Такий підхід виявляє споріднені ділянки коду незважаючи на зміну імен змінних або перестановку операторів. Він менш чутливий до косметичних правок ніж просте порівняння рядків і дозволяє точно фіксувати випадки копіювання алгоритмічних блоків при порівняно невеликих накладних витратах часу [18].

1.3.3 AST-аналіз

Підхід, який спирається на абстрактне синтаксичне дерево, спочатку обробляє вихідний код парсером, перетворюючи його в графічне представлення, де кожний вузол відображає конкретний синтаксичний елемент: оператор присвоєння, умовний вираз, виклик функції тощо. У цьому дереві дочірні елементи репрезентують внутрішню структуру конструкцій – наприклад, тіло циклів або аргументи функцій. Завдяки такому представленню будь-які косметичні правки або зміни форматування зовсім не впливають на саму структуру дерева [18].

Процес порівняння двох AST починається з обходу обох дерев у пошуках максимальної кількості збігів піддерев. Використовують алгоритм Tree Edit Distance, який обчислюють мінімальну кількість операцій вставки, видалення та заміни вузлів, необхідну для перетворення одного дерева в інше. Якщо кількість таких операцій менша за заздалегідь встановлений поріг, вважається, що два фрагменти коду мають спільне походження [19].

Паралельно для прискорення можна застосувати хешування структур піддерев – кожному вузлу присвоюється унікальний відбиток, що залежить від типу вузла і відбитків його нащадків.

1.3.4 Семантичні техніки

Семантичний аналіз виходить за межі поверхневих збігів у синтаксисі та ретельно досліджує, як саме фрагменти коду функціонують під час виконання (рис. 1.4). Спочатку будується контрольний потік програми, який визначає

порядок викликів функцій, умовні переходи та циклічні конструкції. Паралельно здійснюється аналіз потоку даних: відстежуються значення змінних від моменту їхнього оголошення до використання, а також обчислюються побічні ефекти кожного оператора [21].

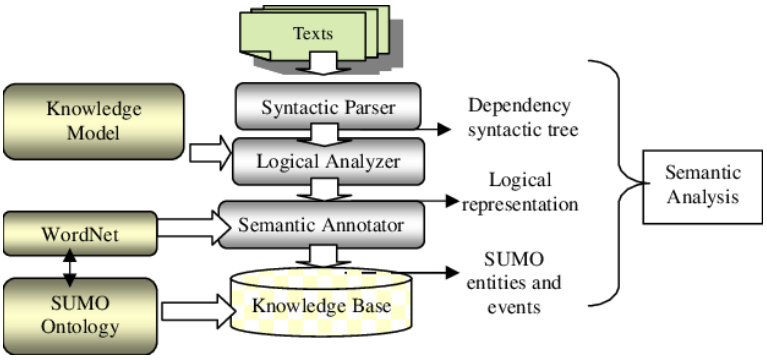

Рисунок 1.4 – Схема роботи семантичного аналізу

У процесі на рисунку 1.4 [20] формуються внутрішні моделі поведінки функцій. Кожна процедура описується не лише набором інструкцій, але й очікуваним входом і виходом. Порівняння таких моделей дозволяє виявити алгоритми, які, попри суттєві відмінності синтаксису або стилю коду, виконують однакову послідовність обчислень на даних. Наприклад, дві реалізації сортування можуть використовувати різні назви змінних і різний синтаксис циклів, але семантичний аналіз виявить однакові взаємозв’язки порівняння і обміну елементів, а також однакову кількість ітерацій у глибині алгоритму.

Аналіз побічних ефектів – наприклад, зміни глобальних змінних або взаємодія з файлами – допомагає відрізнити справжні алгоритмічні відповідності від випадкових подібностей.

1.4 Алгоритми порівняння програмного коду

Лінійні методи порівняння програмного коду оперують безпосередньо з послідовністю символів або токенів, перетворюючи весь файл у одну

безперервну стрічку даних. У випадку коду зазвичай застосовують попередню токенізацію, яка видаляє коментарі та пробіли, після чого отримані лексеми об’єднують у лінійну послідовність.

Порівняльна матриця розміром n×m заповнюється індексами збігів, а значення в останній клітинці вказує на довжину найдовшого спільного підрядка. У випадку великих обсягів коду часова складність O (n·m) стає критичною, проте алгоритм гарантує виявлення усіх буквальних копій, навіть якщо вони розкидані по файлу [23]. Підрядки фіксованої довжини k перетворюються на хеші за допомогою обчислення зваженої суми кодових значень символів. При переході до наступного вікна старий символ виключається з хешу, а новий додається, що дозволяє оновлювати значення за O (1). Якщо хеші двох підрядків збігаються, відбувається уточнююча побайтна перевірка. Детальний опис наданий у таблиці 1.2 [24].

У сукупності ці методи створюють ієрархію фільтрації: спочатку проводиться швидкий попередній аналіз з фінгерпринтингом для відсіву однозначних копій, далі застосовується ковзне хешування для точнішої ідентифікації підрядків, і лише після цього запускається LCS для остаточного підтвердження довгих збігів.

Таблиця 1.2

Параметри лінійних методів порівнянь

Параметр	Найдовший спільний підрядок	Ковзне хешування	Фінгерпринтинг
Точність виявлення	Висока	Середня	Середня-висока
Швидкість аналізу	Низька	Висока	Дуже висока
Чутливість до маскування	Дуже висока	Середня	Низька
Потреба в пам’яті	Висока	Низька	Низька

Структурні підходи підносять аналіз програми над рівень символів і токенів, фокусуючись на її архітектурі та взаємозв’язках компонентів [26].

Центральний метод побудови абстрактного синтаксичного дерева перетворює текст коду у вкладену ієрархію вузлів, де кожний вузол відповідає конкретній конструкції – функції, умовному оператору чи циклу.

Порівняння двох таких дерев здійснюється через зіставлення піддерев.

Алгоритми на основі відстані редагувань по дереву обчислюють мінімальну кількість операцій вставки, видалення чи заміни вузлів, необхідну для перетворення однієї структури на іншу, що дозволяє виявити навіть перебудовані версії одного й того ж алгоритму [25].

Паралельно з цим аналіз графів залежностей створює модель взаємодії між модулями, класами та інтерфейсами.

У такому графі вершини позначають файли або класи, а ребра – виклики методів або імпортування пакетів. Спільні шаблони зв’язків, наприклад, два класи, що завжди викликають однаковий набір сервісів, свідчать про запозичення архітектурного рішення.

1.5 Класифікація видів плагіату в програмному коді

Проведемо детальний аналіз основних форм плагіату в студентських програмних рішеннях. Для кожного виду плагіату опишемо характерні ознаки та відповідні технічні методи виявлення.

1.5.1 Пряме копіювання

Однією з найпростіших і водночас найпоширеніших форм плагіату в студентських проектах є пряме копіювання – точне або майже точне перенесення чужого коду без внесення суттєвих змін. В такому випадку алгоритм, коментарі та структура файлу залишаються незмінними, а студенти ховають факт списування за косметичними правками як відступи, імена змінних [52, 54].

За таких обставин антиплагіатна система повинна швидко ідентифікувати ідентичні фрагменти коду [27].

В таблиці 1.3 наведені основні підходи до виявлення прямого копіювання.

Таблиця 1.3

Порівняння методів у прямому копіюванні

Метод	Ігнорує форматування	Виявляє зміну імен змінних	Швидкість обробки
Байтове порівняння	Ні	Ні	Дуже висока
Токенізація та LCS	Так	Так	Висока
AST–порівняння	Так	Так	Середня
winnow	Так	Так	Висока

Байтове порівняння використовує контрольні суми – найшвидший метод, але чутливий до будь-яких змін у файлі. Токенізація випереджає байтове порівняння, оскільки ігнорує коментарі і пробіли, але потребує нормалізації імен. AST–порівняння аналізує синтаксичну структуру і виявляє ідентичні вузли дерева, стійкий до перейменування змінних, але повільніший.

1.5.2 Парофразування

Парофразування полягає в незначних змінах у вихідному коді, які не впливають на логіку алгоритму, але створюють візуальне враження унікальності. До таких змін належать перейменування змінних, рефакторинг відступів, перетворення циклів, перестановка незалежних блоків коду та переміщення коментарів [28]. Попри перейменування та інші косметичні зміни, логіка залишається ідентичною. Антиплагіатні модулі повинні долати цей рівень маскування, використовуючи поєднання кількох підходів у таблиці 1.4 [29].

Таблиця 1.4

Порівняння методів у мінімальних синтаксичних модифікаціях

Метод	Ігнорує зміну імен змінних	Ігнорує зміну структури циклів	Виявляє тотожність логіки
Байтове порівняння	Частково	Ні	Так
Токенізація та LCS	Так	Так	Так
AST–порівняння	Так	Ні	Частково
winnow	Так	Так	Так

Токенізація + LCS ігнорує пробіли й коментарі, але може сприймати різні структури циклів як відмінні. AST–порівняння аналізує синтаксичні вузли ігноруючи імена та форму циклів. N-gram–відбитки чутливі до порядку операторів, тому застосовні разом із іншими методами [30].

1.5.3 Структурний плагіат

Структурний плагіат передбачає копіювання загальної архітектури програми або основних алгоритмічних схем без прямого наслідування конкретного коду. Студент може змінити імена файлів, переставити класи або функції, але сутність побудови модулів і взаємодії між ними залишиться такою ж, як у вихідному проекті.

У таблиця 1.5 наведено порівняння ознак структурного плагіату [31].

Таблиця 1.5

Порівняння ознак структурного плагіату

Ознака	Оригінальний проект	Репліка студентів
Ієрархія директорій	Так	Так
Спільні модулі	Так	Так
Розподіл відповідальностей	Так	Так
Логіка конфігурації порогів	Так	Так

Діагностика структурного плагіату потребує аналізу метаданих проекту, синтаксичних дерев та графів залежностей модулів.

1.5.4 Мозаїчний плагіат як комбінування фрагментів чужого коду з власними доповненнями

Мозаїчний плагіат відрізняється тим, що студент не копіює код цілком, а «збирає» його з різноманітних джерел, поєднуючи шматки чужих рішень із власними фрагментами. Мозаїчний підхід ускладнює виявлення запозичень, оскільки структура програми стає гібридною: окремі методи або логічні блоки можуть належати різним авторам [33]. В результаті традиційні алгоритми порівняння одного файлу з іншим часто пропускають подібності, адже весь код не є ідентичним жодному з джерел.

Припустимо, студент має завдання об’єднати дві відсортовані послідовності.

У цьому прикладі конструкцію з merge_sorted_lists змінили: перейменували змінні (a→x, b→y, i→p, j→q), переписали кінець циклу через extend, а також додали власний метод виведення print_list.

Метод фінгерпринтингу полягає у розбитті вихідного коду на короткі послідовності з фіксованою довжиною і перетворенні кожної такої підпослідовності на компактне числове представлення «відбиток» [32]. Кожний файл або модуль коду отримує набір подібних відбитків, які легко зіставляти між різними роботами. Якщо два набори відбитків мають значну кількість співпадінь, це є ознакою можливого запозичення, навіть якщо студенти намагалися приховати схожість косметичними змінами.

1.6 Огляд наявних рішень

Світ освітніх платформ пропонує низку зрілих інструментів, що автоматизують перевірку коду та виявлення запозичень.

Кожне рішення володіє власними алгоритмами порівняння, інтерфейсом та можливістю інтеграції з LMS, відрізняючись за швидкістю й глибиною аналізу.

Детальний розбір CodeGrade, JPlag, MOSS, PlagScan і Domjudge дозволить відшукати оптимальні практики для формування ефективної та гнучкої системи оцінювання студентських робіт:

1) CodeGrade

CodeGrade надає повністю автоматизовану платформу для тестування коду: студент завантажує рішення, а система одразу виконує юніт-, інтеграційні й стрес-тести за налаштованими шаблонами. Викладач працює в єдиному веб-інтерфейсі, де поруч відображаються вихідний код, вхідні дані, результати тестів і коментарі. Вбудований антиплагіат ігнорує загальні бібліотеки та фреймворки, порівнюючи лише оригінальні алгоритмічні фрагменти на рівні токенів і AST, що зменшує хибні спрацьовування. Підписка на CodeGrade може бути дорогою для невеликих кафедр, оскільки вартість залежить від кількості користувачів і обсягів перевірок [34].

2) Jplag

Jplag використовує поєднання токен-аналізу та AST-зіставлення для виявлення схожості коду навіть після косметичних змін чи перейменувань. Інструмент підтримує Java, C/C++, Python, JavaScript та інші мови, а результати представляє в інтерактивних HTML-звітів із підсвічуванням повторів. Безкоштовний та відкритий, Jplag легко інтегрується в CI/CD-процеси через командний рядок. Основне обмеження — відсутність глибокого семантичного аналізу, що може пропустити логічно модифіковані алгоритми[35].

3) MOSS

MOSS використовує n-грами та алгоритм winnowing для виділення характерних “відбитків” коду: система розбиває файли на фрагменти

фіксованої довжини, обчислює їхні хеші та вибирає мінімальні значення в ковзному вікні. Підтримуються C, C++, Java, Python, Scheme та інші мови. Запуск аналізу здійснюється через CLI або веб-запит, а результати доступні у вигляді HTML-звіту за унікальним посиланням. Завдяки високій продуктивності MOSS здатен обробляти тисячі файлів за кілька хвилин, однак фіксований розмір n-грам може ускладнювати виявлення дрібних або розсіяних змін [36].

4) PlagScan

PlagScan призначений перш за все для академічних текстів і має доступ до великого масиву джерел – наукових журналів, веб-сторінок тощо. Код перевіряється як звичайний текст: рядки шингліться, токенізуються та порівнюються з індексом інтернет-ресурсів і внутрішньою базою. Інтеграція можлива через хмарний REST API або LTI в LMS, що робить налаштування швидким. Після аналізу система генерує детальний звіт із графіками збігів, списком джерел і підсвіченими фрагментами. Завдяки широкому охопленню текстів PlagScan чудово виявляє цитування та перефразування в документах, проте в аналізі програмного коду його точність менша, ніж у спеціалізованих інструментів. Незважаючи на це, єдиний інтерфейс для всіх типів матеріалів робить його зручним доповненням для забезпечення академічної доброчесності [37].

1.7 Вибір серверного фреймворку для реалізації платформи

Під час вибору серверного фреймворку для реалізації платформи автоматизованого оцінювання коду та перевірки на антиплагіат здійснено ґрунтовний аналіз кількох підходів і технологічних стеків.

По-перше, легковагові мікрофреймворки на кшталт Flask або Bottle забезпечують максимальну свободу в побудові архітектури, проте вимагають власноручної інтеграції ORM, модулів аутентифікації та адміністративної панелі [38].

Другим варіантом розглядалися асинхронні рішення, зокрема FastAPI та Sanic, які відзначаються високою продуктивністю при обробці одночасних запитів [39]. Хоча вбудована валідація даних через Pydantic у FastAPI значно спрощує перевірку вхідних даних, асинхронна модель вводить додаткову складність під час налагодження та потребує адаптації синхронних бібліотек.

Однак у контексті автоматичного тестування Python-коду та антиплагіату обґрунтованою виявилася сильна інтеграція у єдиному Python-стеці.

Усі наведені альтернативи довели свою цінність у певних випадках, але саме Django виявився найбільш збалансованим рішенням для поставленої задачі. Його архітектура Model–Template–View надає чіткий розподіл відповідальностей, вбудований ORM значно спрощує роботу з базою даних, а механізми аутентифікації та система адміністрування «з коробки» дозволяють одразу розгорнути повнофункціональний інтерфейс для викладачів і адміністраторів.

1.8 Архітектура на основі мікросервісів

Мікросервісний підхід дозволяє розбити великий веб-додаток на низку незалежних сервісів, кожен із яких зосереджується на чітко визначеній функції – від аутентифікації користувачів до запуску тестів чи перевірки коду на плагіат.

Мікросервісна архітектура базується на поділі системи на незалежні сервіси, кожен із яких реалізує окремий набір бізнес-функцій та має власний життєвий цикл розгортання [40]. На відміну від монолітного підходу, де всі компоненти інтегровані в одну велику програму з єдиною базою коду, мікросервіси існують як автономні одиниці, що спілкуються через чітко визначені інтерфейси.

Поділ дозволяє розробникам працювати над окремими сервісами без ризику порушити суміжні модулі, а інфраструктурі – розгортати та масштабувати кожен сервіс за потребою, не зачіпаючи решту системи.

Мікросервіси прийматимуть запити від користувачів, керуватиме сесіями та координуватиме взаємодію зі спеціалізованими мікросервісами для тестування коду, перевірки на плагіат або зберігання файлів.

Висновки до розділу 1

У результаті ґрунтовного аналізу наявних підходів до автоматичного оцінювання коду та виявлення плагіату було виокремлено ключові переваги й обмеження кожного з методів.

Швидкість обробки великих обсягів коду, масштабованість та об’єктивність оцінювання, позбавленого людського фактору, стали базовими вимогами для будь-якої навчальної системи.

Розгляд проблем ручного оцінювання коду виявив суттєві недоліки: надмірне навантаження викладачів, затримки у зворотному зв’язку для студентів, ризик упередженого виставлення оцінок та значні витрати часу й ресурсів у великих курсах.

Був зроблений вибір п’яти взаємодоповнювальних підходів до виявлення схожості програмного коду, які будуть реалізовані в модулі антиплагіату майбутньої системи:

- використання алгоритму difflib.SequenceMatcher для швидкого виявлення буквальних збігів на рівні символів;
- застосування токен-орієнтованого аналізу з лексичною нормалізацією та пошуком найдовших спільних підпослідовностей для ігнорування змін форматування та перейменувань;
- побудова та порівняння абстрактних синтаксичних дерев для виявлення структурних запозичень;
- генерування n-грам і порівняння їх множин для стійкого виявлення спільних фрагментів коду за допомогою метрики Жаккара;
- алгоритм Winnowing, який створює компактні «відбитки» через хешування n-грам і локальні мінімуми, забезпечуючи ефективне порівняння навіть великих файлів.

РОЗДІЛ 2.
ПРОЕКТУВАННЯ СИСТЕМИ ПЕРЕВІРКИ КОДУ ТА ЗАПОЗИЧЕНЬ

У процесі проектування слід урахувати одночасну підтримку кількох мов програмування, інтеграцію з існуючими навчальними системами та надійні механізми захисту від зловживань. Прагнення до модульності й незалежності компонентів забезпечить можливість еволюції окремих сервісів без відключення всієї системи.

Після теоретичного обґрунтування підходів до автоматичного тестування та виявлення плагіату настає час перейти до практичного втілення обраної архітектури.

2.1 Вимоги до системи

Для розробки системи необхідно описати функціональні та нефункціональні вимоги до системи.

2.1.1 Функціональні вимоги до системи

На рівні функціональних вимог платформа повинна забезпечити повний життєвий цикл роботи з програмними завданнями – від створення та публікації вправ викладачем до завантаження рішення та отримання результатів студентом.

Далі необхідним є інструментарій управління курсами: створення груп, предметів, розклад завдань із параметрами – максимальною кількістю спроб, ваговими коефіцієнтами для тестів, опціональним набором підзадач.

Для кожного завдання викладач повинен мати змогу завантажити або відредагувати тестові скрипти та задати умови автоматичного оцінювання – критерії успішного проходження окремих кейсів.

Після подання рішення система протягом лічених секунд запускає сервіс автотестування, ізолюючи виконання в безпечному контейнері чи обмеженому процесі, фіксує результати в базі та відображає студенту

детальний звіт із переліком успішних і невдалих перевірок. Паралельно з тестовим раннером виконується перевірка плагіату: модуль антиплагіату порівнює нову роботу з усіма попередніми поданнями та, за запитом викладача, видає ранжований список схожості за кожним із п’яти інтегрованих алгоритмів – рядковий аналіз, токенізація, AST-зіставлення, n-грами та фінгерпринтинг.

Звіти по плагіату мають бути представлені у вигляді таблиць із гіперпосиланнями на пари файлів та відсотками збігу.

На рис. 2.1 наданий повний перелік усіх функціональних вимог до системи.

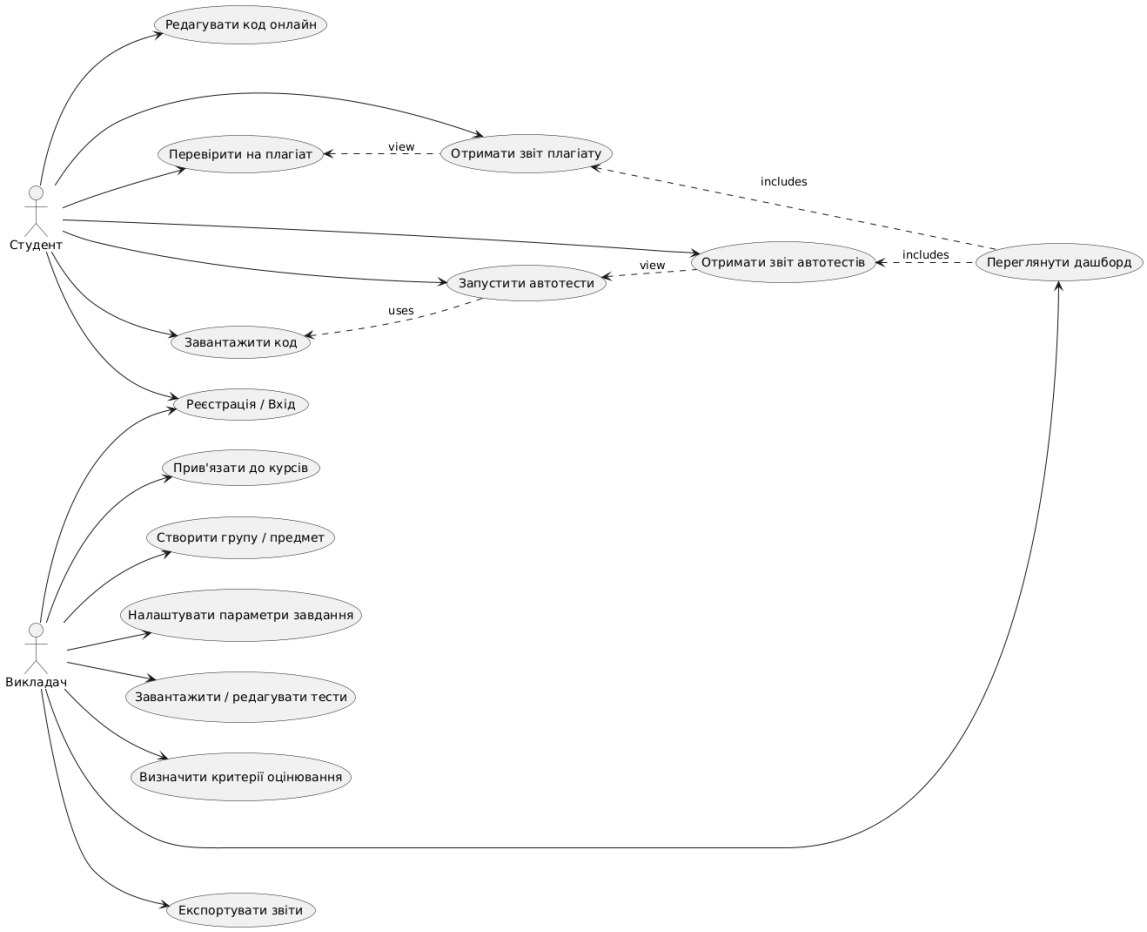

Рисунок 2.1 – Перелік функціональних вимог до системи

2.1.2 Нефункціональні вимоги до системи

До нефункціональних характеристик на рис. 2.2 відноситься забезпечення високої доступності та продуктивності під навантаженням одночасного подання сотень робіт.

Відгук платформи на дії користувача – завантаження інтерфейсу, запуск тестів, генерація звіту має відбуватися менше ніж за кілька секунд при нормальній завантаженості, а автотестування та антиплагіатні обчислення – завершуватися в межах одного-двох хвилин навіть за паралельного виконання.

Архітектурні рішення на рис. 2.2 повинні гарантувати горизонтальне масштабування: додавання нових екземплярів сервісів автотестування чи антиплагіату та розширення кластеру бази даних без суттєвої модифікації коду.

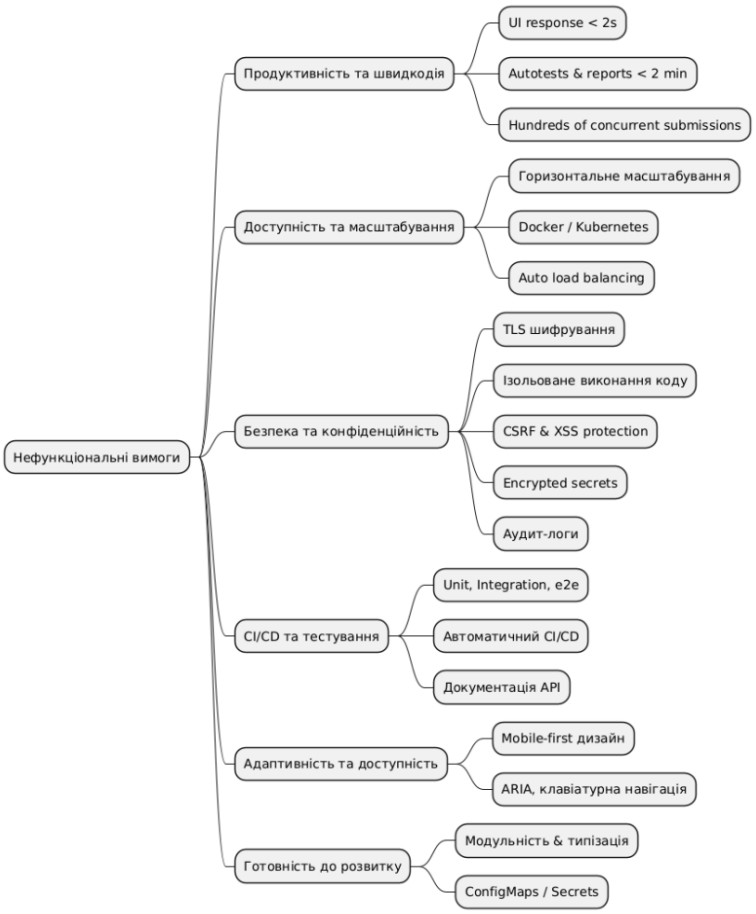

Рисунок 2.2 – Перелік нефункціональних вимог до системи

2.2 Мікросервісна архітектура

Наведені нижче п’ять контейнерів на рис. 2.3 утворять основу розгортання платформи, визначеної в розділі 1.8.

Кожен із них виконує чітко окреслену функцію та взаємодіє з іншими через REST-інтерфейси й спільні сховища даних.

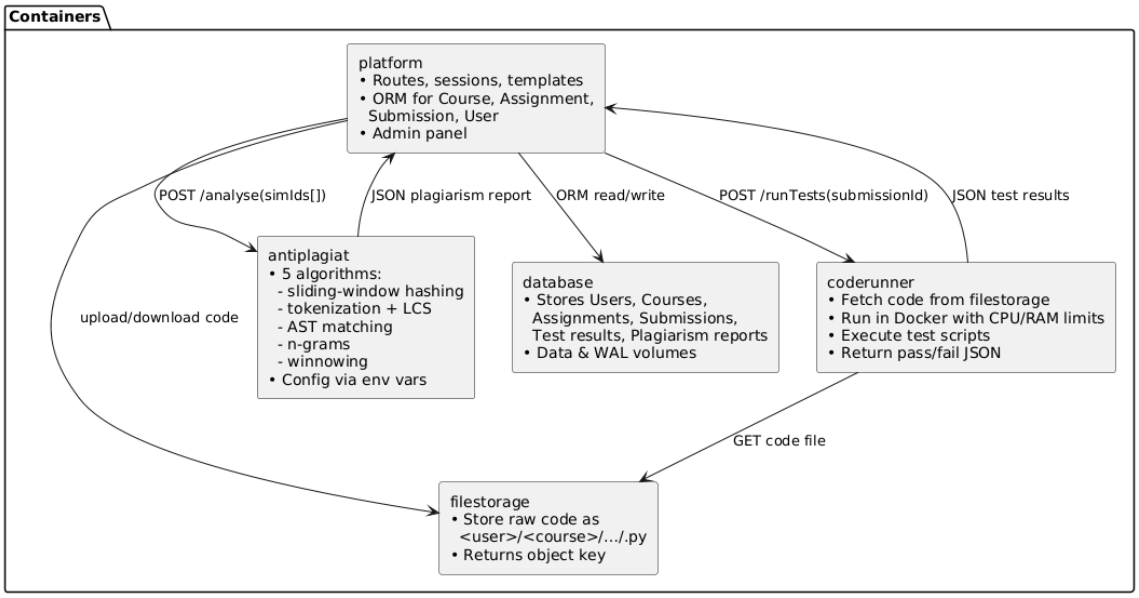

Рисунок 2.3 – Діаграма компонентів

2.2.1 Контейнер platform

Контейнер на рис. 2.3 на базі Django відповідальний за веб-інтерфейс та основну бізнес-логіку.

Він обробляє запити користувачів, відповідає за маршрутизацію, сесії, рендеринг шаблонів і виклик REST-ендпоінтів інших сервісів.

Усередині нього працює ORM для читання й запису сутностей «Курс», «Завдання», «Подання» та «Користувач», а також включено вбудовану адміністративну панель для управління всією системою.

2.2.2 Контейнер antiplagiat

Antiplagiat реалізує п’ять алгоритмів аналізу схожості – порівняння рядків із ковзним хешуванням, токенізацію + LCS, AST-порівняння, n-грами та winnowing.

На вході він приймає список ідентифікаторів подань завдань і повертає єдиний звіт у форматі JSON із відсотком подібності для кожної пари. Весь код виконується в ізольованому середовищі, а конфігурація алгоритмів задається через змінні середовища.

2.2.3 Контейнер filestorage

Об’єктне сховище на базі MinIO відповідає за збереження сирих файлів із рішеннями студентів.

Під час подання коду API цього контейнера приймає файл, зберігає його за схемою <student>/<course>/<assignment>/<submission>.py і повертає platform унікальний ключ доступу.

Інші сервіси звертаються до filestorage для читання або копіювання файлів перед аналізом чи тестуванням.

2.2.4 Контейнер coderunner

Сервіс для виконання тестів у бекграунді. Після отримання REST-запиту з ідентифікатором подання, coderunner завантажує відповідний файл із filestorage, запускає його всередині Docker-контейнера з обмеженнями CPU та пам’яті, передає на вхід скрипти тестів і збирає результати.

Підсумковий набір успішних і провалених тестів надсилається назад до platform у вигляді структурованого JSON.

2.2.5 Контейнер database

Контейнер із PostgreSQL забезпечує постійне зберігання всіх сутностей: облікових записів, курсів, завдань, подань, результатів тестів та звітів антиплагіату.

Йому належать дві томи – один для даних БД, інший для журналів транзакцій.

ORM у platform налаштовано на автоматичне створення міграцій і зв’язків між таблицями, що полегшує розвиток моделі даних без простоїв.

2.3 База даних

Проектування схеми зберігання даних є фундаментом будь-якої системи автоматичного оцінювання: саме від правильної моделі залежить швидкість пошуку записів, надійність збереження результатів і гнучкість подальшого розширення платформи.

2.3.1 Сутність «User»

User виступає центральним елементом моделі даних, адже всі операції в системі починаються з ідентифікації та визначення рівня доступу.

Запис у таблиці на рисунку 2.4 містить поле роль зі значеннями student, teacher або administrator. Student має обмежені можливості: перегляд курсів, завантаження програмних рішень, отримання результатів тестів.

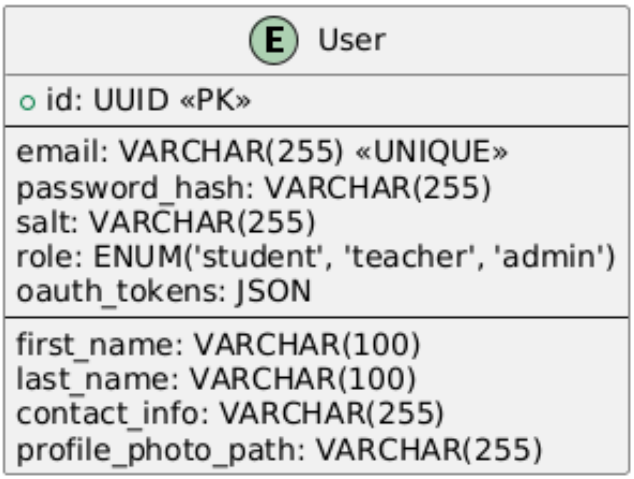

Рисунок 2.4 – Таблиця сутності «User»

Teacher здатний створювати завдання, налаштовувати автоматичні тести, переглядати подання student та ініціювати перевірку на плагіат.

Administrator володіє повними правами: керування обліковими записами, модифікація налаштувань платформи, доступ до логів аудиту.

Сегрегація ролей на рис. 2.5 гарантує розмежування відповідальності та зменшує загрозу несанкціонованих змін у критичних модулях.

Рисунок 2.5 – Сегрегація сутності «User»

Для верифікації особи під час входу і збереження сесій на рис. 2.4 кожен User має атрибути аутентифікації.

Поле email виконує роль унікального логіна, паролі зберігаються у вигляді захищених гешів із використанням сучасних алгоритмів та солі. Поля name та surname використовуються для персоналізації повідомлень і звітів. Контактна інформація може містити номер телефону або посилання на внутрішній месенджер для оперативного зв’язку.

2.3.2 Сутність «Task»

Сутність Task на рис. 2.6 відповідає за зберігання інформації про кожну вправу в межах певного курсу та зв’язки з її автором.

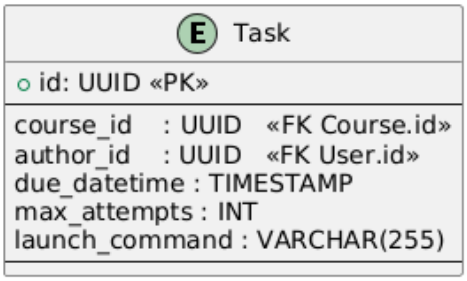

Рисунок 2.6 – Таблиця сутності «Task»

У результаті Task надає модулю TestRunner увесь набір параметрів – від контексту курсу й автора до команд запуску й очікуваних виходів – без необхідності додаткових звернень до файлової системи.

2.3.3 Сутність «Submission»

Сутність Submission на рис. 2.7 відповідає за фіксацію кожного надсилання коду студентом.

Мітка часу надсилання рішення на рис. 2.7 потрапляє в поле submitted_at, фіксуючи точний момент, коли студент відправив свій код. Одночасно поле test_status відображає поточний стан виконання автотестів – від очікування на черзі й активного запуску до успішного проходження всіх тестів або виявлення помилок.

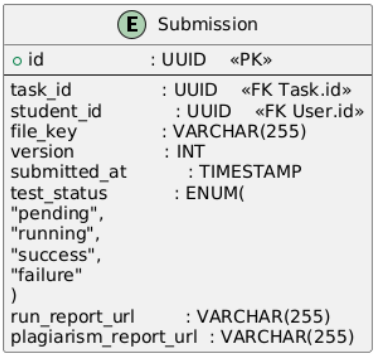

Рисунок 2.7 – Таблиця сутності “Submission”

Для швидкого доступу до результатів служби автотестування у полі run_report_url зберігається посилання на детальний звіт, де вказано, які тести були пройдені, а які – провалені. Аналогічно, plagiarism_report_url містить посилання на звіт модуля антиплагіату, де наведено підозрілі збіги та процентні показники схожості з іншими поданнями.

2.3.4 Сутність «Evaluation»

Сутність Evaluation на рис. 2.8 відповідає за збереження результатів перевірки коду студента.

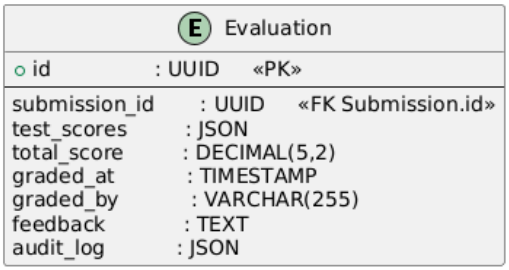

Рисунок 2.8 – Таблиця сутності «Evaluation»

Поле test_scores містить масив об’єктів із балами за кожен виконавчий тест, а total_score обчислює сумарний результат із урахуванням ваги кожного тестового кейсу.

Відмітка часу graded_at фіксує момент завершення оцінювання та підрахунку total_score, а graded_by позначає джерело виставлення балів – system для автоматичного підрахунку через модуль coderunner або ідентифікатор викладача в разі ручного коригування.

Поле feedback прагне надати студенту пояснення до оцінки та рекомендації. У цьому полі зберігаються коментарі викладача щодо виявлених помилок, поради з оптимізації коду або загальні зауваження до рішення.

2.3.5 Зв’язки між таблицями

Модель даних будується на взаємозв’язках, які відображають логіку освітнього процесу.

Зв’язок один-до-багатьох на рис. 2.9 між Course і Task забезпечує можливість для кожного курсу містити довільну кількість вправ.

Для Course створюється первинний ключ id, а у Task зберігається зовнішній ключ course_id, який встановлює приналежність завдання до певного курсу. Між Task і Submission існує аналогічна залежність один-до-багатьох.

Кожне завдання може отримати декілька подань від різних студентів чи в різні моменти часу, тому у Submission реалізовано зовнішній ключ task_id. Оцінка прив’язана до конкретного Submission

Залежність між Submission і Evaluation може бути або один-до-одного, коли кожну подачу оброблюють лише один раз і видають єдину остаточну оцінку, або один-до-багатьох, якщо передбачено кілька етапів оцінювання.

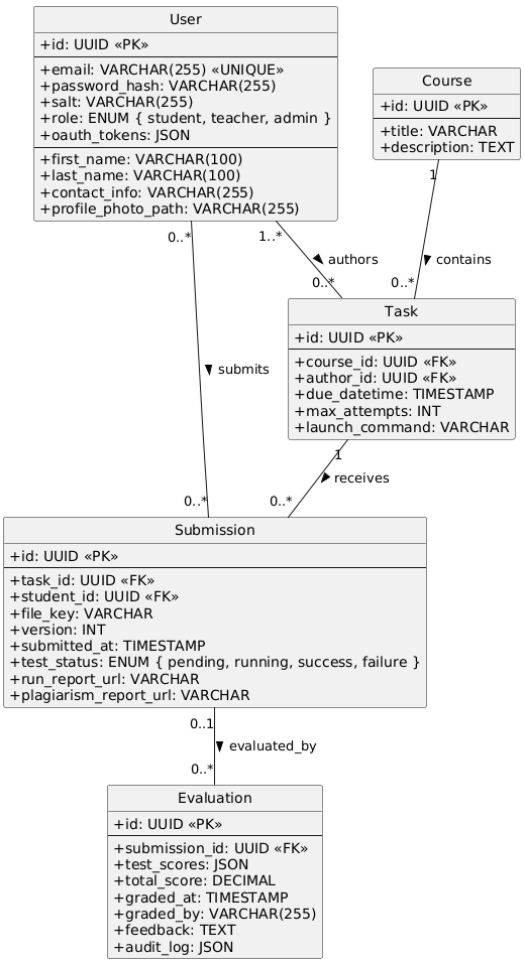

Рисунок 2.9 – Зв’язки між сутностями

У Evaluation використовується зовнішній ключ submission_id, який однозначно вказує на запис в таблиці Submission, до якого належать результати тестів, сума балів та коментарі. Така структура гарантує цілісність даних і дає змогу реконструювати повну історію оцінювання кожного рішення.

2.4 Формування вимог до модуля автотестування

Ізольоване середовище для виконання студентського коду було організовано на основі Docker-контейнерів, що гарантує відокремленість кожного запуску від хост-системи та інших сабмітів. Кожний контейнер має власну файлову систему, обмежений обсяг пам’яті й процесорного часу, завдяки чому будь-який некоректний або зловмисний код не впливає на роботу платформи загалом.

Водночас за необхідності можна швидко переключитися на інший базовий образ Docker, щоб протестувати рішення у новому середовищі або додати підтримку додаткових мов.

Ці шаблони описують команду старту python3 -m pytest tests/assignment1 --json-report та змінні середовища, потрібні для коректної роботи.

Викладач або адміністратор може створити новий шаблон із власними аргументами, додатковими параметрами таймауту та шляхи до тестових файлів, не змінюючи код модуля автотестування.

Усі вхідні параметри проходили сувору валідацію: перш ніж сформувати команду запуску, перевірялися тип і розмір переданих даних, допустимі розширення файлів, а також коректність шляхів до тестових сценаріїв.

Структура відповіді містила масив об’єктів із полями test_name, status у вигляді success чи failure, score та message із детальним описом помилки або очікуваного результату.

Базова реалізація дозволяла запускати Python-скрипти через unittest, але згодом був доданий модуль динамічної конфігурації, який підвантажує відповідні рантайми й утиліти залежно від сутності завдання.

Кожен Task-об’єкт зберігає в полі language значення типу «python», «java», «cpp» або «javascript», а для кожного із цих варіантів передбачено власний контейнер із необхідними інтерпретаторами, компіляторами та тестовими фреймворками.

Висновки до розділу 2

У другому розділі виконано систематичну постановку завдань і детальне проектування майбутньої платформи, що лягло в основу архітектури рішення.

Вибір мікросервісної архітектури було обґрунтовано потребою в незалежному розгортанні різних частин системи, гнучкій горизонтальній масштабованості й ізоляції ресурсномістких операцій.

П’ять контейнерів – platform, antiplagiat, filestorage, coderunner і database – відповідають за веб-інтерфейс і маршрутизацію, аналіз схожості коду, зберігання файлів студентських сабмітів, ізольоване виконання тестів та управління даними в БД відповідно.

Проектування моделі даних узгоджено з принципами нормалізації й цілісності: сутності User, Task, Submission і Evaluation охоплюють усі бізнес-атрибути та зв’язки між курсами, завданнями, поданнями та оцінками.

РОЗДІЛ 3.
РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Перехід від концептуальної моделі до конкретного коду відкриває можливість побачити, як усі раніше описані сервіси та алгоритми оживають у працездатному середовищі.

Нижче відтворено повний процес реалізації платформи: від налаштування контейнерів і зв’язку мікросервісів через REST-інтерфейси до безпосередньої інтеграції п’яти ключових методів виявлення плагіату та механізму автоматичного тестування.

Кожен крок супроводжується фрагментами коду, поясненнями конфігурацій і рекомендаціями щодо оптимального розгортання, що забезпечує плавний перехід від проектної документації до реального продукту.

Зі встановленими сервісами platform, antiplagiat, filestorage, coderunner та database платформа готова до прийому перших студентських рішень і негайної генерації як оцінок, так і звітів про ймовірний плагіат.

3.1 Технологічний стек і середовище розробки

У ході розробки застосовано Python 3.10 як головну мову програмування, оскільки вона забезпечувала широку підтримку бібліотек і дозволяла уніфікувати код декількох мікросервісів.

Основну платформу було реалізовано на Django 4.2, що дозволило скористатися вбудованим ORM-шаром, системою аутентифікації та адміністративною панеллю з коробки.

Для побудови легковагових REST-інтерфейсів у службах antiplagiat та coderunner було використано Flask 2.1, а для управління контейнерами безпосередньо з коду – Docker SDK for Python.

У продакшн-середовищі обрали PostgreSQL 14 завдяки її стабільності та потужним можливостям індексації.

Для зручності локальної розробки також підтримувався MySQL 8. Моделі даних у platform і filestorage були описані через Django ORM, що гарантувало єдину точку правди й автоматичну міграцію схем.

У coderunner та antiplagiat застосували SQLAlchemy, оскільки знадобилася гнучкість у роботі з різними СУБД і можливість складного динамічного формування запитів.

Керування версіями здійснювалося за допомогою Git із розміщенням вихідного коду на GitHub.

Розробка нових функцій і виправлення багів проводилася через pull-request-и, що дозволяло централізовано виконувати код-рев'ю й підтримувати високий рівень якості.

3.2 Реалізація backend-частини системи

У додатку platform було реалізовано основну функціональність веб-інтерфейсу на рис. 3.1.

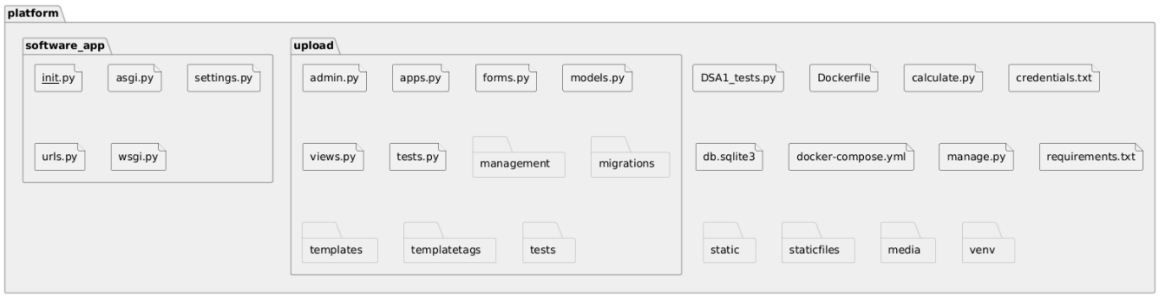

Рисунок 3.1 – Структуру мікросервісу platform

У файлі models.py з'явилися класи User із набором ролей і Course із полями author та title, що забезпечило зберігання інформації про викладачів і курси.

Між моделями виставлено зв'язок багато-до-одного, через який кожен курс асоціювався з конкретним викладачем. Було налаштовано власну модель користувача за допомогою AUTH_USER_MODEL у settings.py, завдяки чому

вбудована адміністративна панель підтримала роботу з ролями та правами без додаткової розробки.

У файлі urls.py було прописано маршрути для реєстрації, входу, списку курсів, перегляду деталей курсу і профілю користувача.

Шлях ' /, 'accounts/' – на модуль входу та реєстрації, 'courses/int:pk/' – на деталі конкретного курсу, а 'profile/' – на персональну сторінку користувача на рис. 3.2.

Кожен маршрут був прив’язаний до відповідної функції або класу у views.py через import із модуля platform.views, що забезпечувало чітку навігацію та можливість легко додавати нові ендпоїнти.

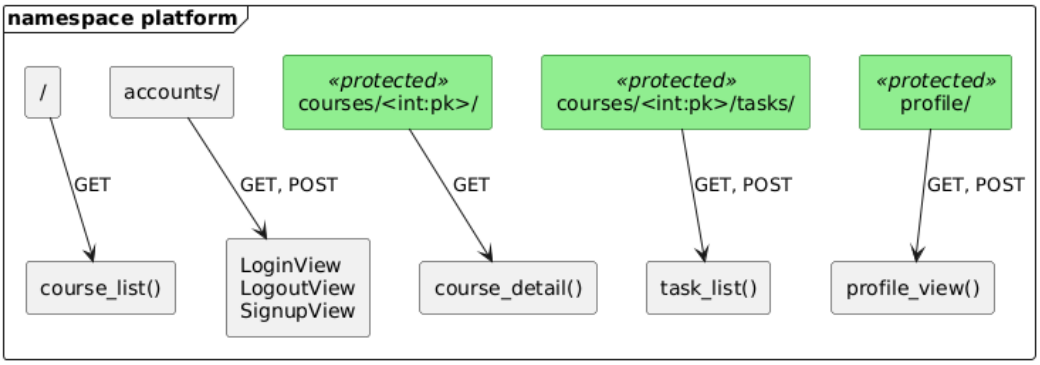

Рисунок 3.2 – Структуру urls.py

Файл views.py містив набір функцій-обробників: course_list для вибірки й відображення всіх курсів з пагінацією, course_detail для показу предмету разом із переліком завдань, profile_view для виведення інформації про поточного користувача й його ролі.

Функція create_course приймала POST-запити з формою CourseForm, виконувала валідацію та створення запису в базі, після чого перенаправляла на список курсів.

Шаблони були розміщені у каталозі templates/platform.

У файлах course_list.html, course_detail.html і profile.html за допомогою синтаксису Django Template Language відображалися поля моделей, формувались списки курсів і панелі керування для викладачів. Файл settings.py

містив реєстрацію platform у INSTALLED_APPS, налаштування підключення до бази даних і конфігурацію статичних файлів.

Вбудована адмін-панель надавала можливість керувати записами User і Course через інтерфейс /admin, що пришвидшило початкове завантаження тестових даних і налаштування ролей.

На ри. 3.3 зображено послідовність обробки кожного HTTP-запиту у backend-частині платформи.

Першим етапом проходження запиту є SecurityMiddleware, що забезпечує примусове перенаправлення на HTTPS, встановлення заголовків HSTS, додавання базових заголовків безпеки (X-Frame-Options, X-Content-Type-Options) і захист від clickjacking. Далі запит надходить у SessionMiddleware, який відповідає за завантаження й збереження даних сесії, а також за управління cookie. Після цього CommonMiddleware виконує нормалізацію URL (прохід без слешу/з додаванням), обробку умовних GET-запитів із підтримкою ETag і встановлення заголовка Content-Length.

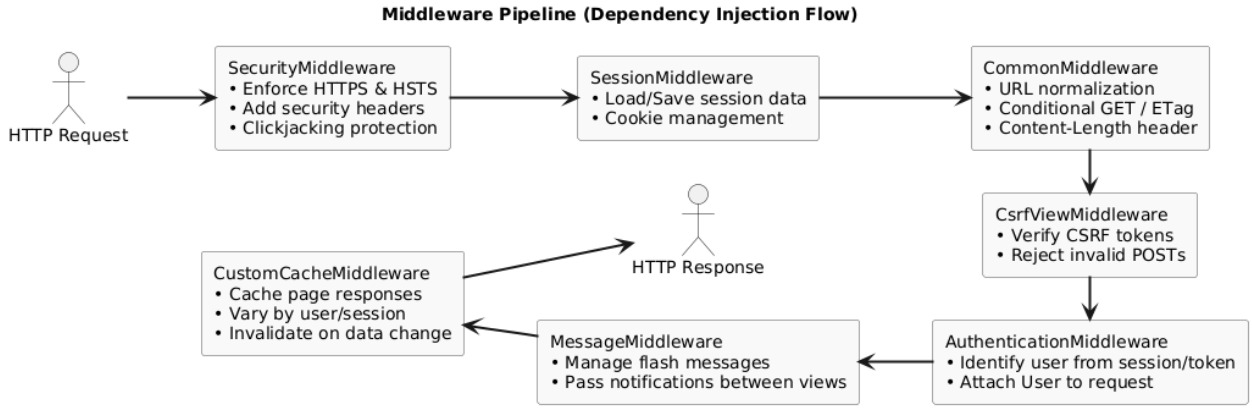

Рисунок 3.3 – Middleware Pipeline

Конфігурацію проекту було зосереджено в файлі settings.py, де було визначено ключові параметри середовища: шлях до бази даних PostgreSQL, схему авторизації через AUTH_USER_MODEL із власною моделлю User і підключення додатків platform, antiplagiat, filestorage, coderunner та rest_framework.

Налаштування INSTALLED_APPS містили модулі для роботи з REST-інтерфейсом і автоматичною документацією, а секція MIDDLEWARE забезпечувала обробку сесій, CSRF-захист і логіку кешування. Параметри STATIC_URL і STATICFILES_DIRS вказали шляхи до статичних ресурсів а шаблонна директивка TEMPLATES з блоком DIRS задала директорію templates для пошуку HTML-файлів.

Для обробки форм було створено forms.py, де CourseForm і UserRegistrationForm успадковувалися від forms.ModelForm.

Поля name, description і due_date отримали власні валідатори, а віджет DateTimeInput із параметром attrs={'type': 'datetime-local'} забезпечив коректний вибір дати й часу в браузері. Завдяки цьому у views.py не доводилося писати окремий код для перевірки формату дати.

Файл admin.py було доповнено класами CourseAdmin і UserAdmin із вказівкою list_display і search_fields, що дозволило в адмін-інтерфейсі швидко шукати курси за назвою та фільтрувати користувачів за email або роллю.

Реєстрація моделей через admin.site.register прискорила роботу з тестовими даними та налаштуванням ролей перед початком інтеграційного тестування.

Логіку підключення статики й медіа-файлів на рис. 3.4 було описано в urls.py і settings.py через settings.MEDIA_URL і settings.MEDIA_ROOT, що надало можливість зберігати аватарки користувачів і прикріплені документи.

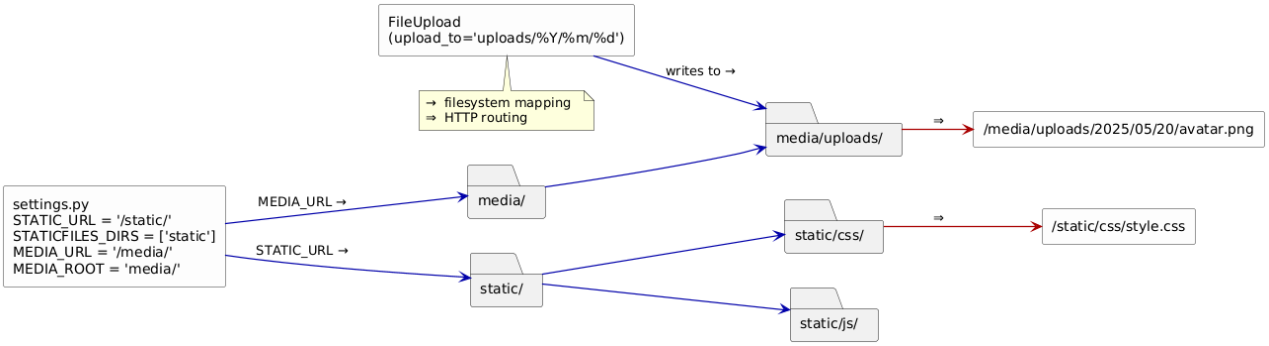

Рисунок 3.4 – Діаграма потоків файлів статики

Застосунок filestorage опанував завантаження файлів через модель FileUpload із полем FileField(upload_to='uploads/%Y/%m/%d'), а зміни в шаблоні profile.html дозволили показати фото профілю користувача за допомогою {{ user.profile.image.url }}.

3.3 Реалізація frontend-частини системи

Файл base.html на рис. 3.5 задає каркас усіх сторінок: у секції <head> підключено стилі Bootstrap і власний CSS із каталогу static, а внизу перед закриттям <body> імпортуються скрипти jQuery, Popper.js та Bootstrap JS. Навігаційна панель реалізована через <nav> із пунктами «Предмети», «Мої оцінки» та кнопками входу/реєстрації, причому активний пункт підсвічується за допомогою контекстної змінної {% if active_page == "subjects" %}.

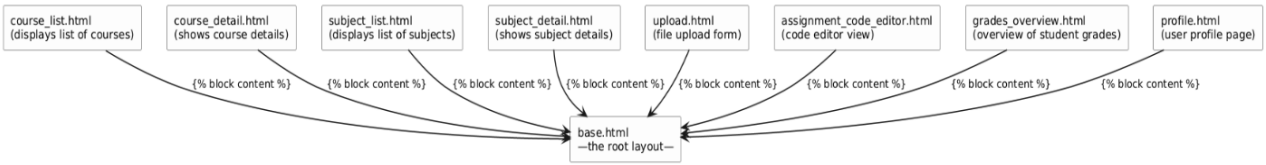

Рисунок 3.5 – Діаграма спадкування шаблонів

У base.html на рис. 3.5 визначено блок {% block content %}{% endblock %}, куди укладаються специфічні шаблони інших сторінок.

Assignment_detail.html на рисунку 3.5 показує заголовок вправи, опис завдання й два таби – «Код» та «Тести».

У табі «Код» приведено кнопку «Завантажити файл» (upload.html) і пряме посилання на онлайн-редактор (assignment_code_editor.html), де вбудовано Ace Editor.

Сторінка subject_list.html розміщує перелік предметів у вигляді карток Bootstrap Card з назвою, описом і кнопкою «Перейти», що веде на деталі курсу.

Пошук і сортування курсів винесено в окремий фрагмент над списком, а пагінація реалізована через {% if page_obj.has_previous %} і has_next із посиланнями на попередню й наступну сторінки.

У каталозі templates/platform було організовано ієрархію шаблонів: base.html служив головним макетом із визначеними блоками title, navbar, content і scripts.

У ньому підключалися загальні стилі Bootstrap, власні CSS-файли та контейнер для динамічного контенту.

В інших файлах – course_list.html, course_detail.html, profile.html – відбувалося розширення base.html через {% extends "platform/base.html" %} і заповнення блоку content специфічним вмістом: таблицею курсів із кнопками створення й редагування, деталями курсу з формами сабмітів та віджетами для відображення ролей користувачів.

3.4 Реалізація модуля автоматичного оцінювання

Для забезпечення надійної ізоляції виконання студентського коду було обрано механізм контейнеризації за допомогою Docker. Кожний запуск автоматичних тестів у додатку Ґ відбувався всередині окремого контейнера, який містив усі необхідні залежності та інтерпретатор Python.

Всі тимчасові файли, мережеві виклики й файлові операції залишалися в межах свого контейнера, а після завершення тестування образ одразу видалявся, гарантуючи чистоту середовища для наступного запиту. У разі необхідності підтримки легшого підходу могла використовуватися віртуальна система на базі venv або virtualenv, проте кінцевий вибір зупинився на Docker через його здатність ефективно ізолювати процеси на рівні ядра операційної системи та вбудовані механізми обмеження ресурсів.

3.5 Реалізація модуля перевірки на плагіат

Компонент antiplagiat було спроектовано за принципом чіткого розподілу обов’язків і залежностей.

У його складі виділено три основні підсистеми: контролер, сервіс порівнянь і модуль відображення результатів.

Контролер приймає HTTP-запити від клієнта, витягує з них ідентифікатори подань і вибрані методи порівняння, проводить початкову валідацію та формує завдання для асинхронної черги.

Сервіс поміж тим виконує власне порівняння коду: він доставляє файли з сховища, запускає потрібні алгоритми у ізольованому процесі або контейнері, об’єднує отримані метрики схожості в єдину структуру та зберігає проміжні й фінальні звіти в базі даних.

Для проведення тестування було створено 8 різних акаунтів, у яких були додані різні рішення для 4 домашнього завдання.

Порівняння результатів надано на рис. 3.6.

Token-based аналіз дав трохи вищі показники для деяких студентів: максимуми сягнули 91 % у andrii.kolesnyk4 та 93 % у olena.parkhomenko1.

AST-порівняння на рис. 3.7 базувалося на аналізі внутрішньої структури програм: двічі пропускали вихідний текст через ast.parse, отримуючи абстрактні синтаксичні дерева.

Далі обидва дерева перетворювалися в лінійні dump-рядки, де кожен вузол і його атрибути кодувалися у вигляді тексту.

Рисунок 3.6 – Результат перевірки методом токенізації

Рисунок 3.7 – Результат перевірки методом AST

Для алгоритму Winnowing на рис. 3.7 перш за все виконувалося розбиття сирцевого коду на підрядки довжини k, зазвичай k обирали рівним п’яти символам або токенам.

Кожна k-грама перетворювалася в числовий хеш за допомогою швидкого невироджувального хеш-функції (наприклад, Rabin-Karp), після чого утворювався масив хешів у тому самому порядку, в якому з’являлися k-грами в тексті.

Далі цей масив оброблявся у фіксованих «вікнах» розміру t: у кожному вікні з t хешів вибирався мінімальний — саме він ставав відбитком цього вікна.

В результаті на рис. 3.8 виходила набір відбитків, що розподілений по всьому коду, але досить компактний для порівняння.

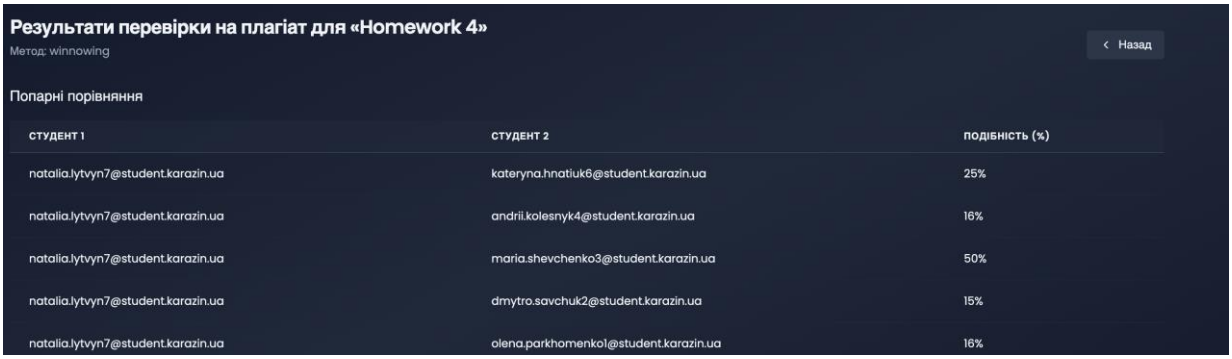

Рисунок 3.8 – Результат перевірки методом Winnowing

Метод n-грам на рис. 3.9 підходив більш прямо: із сирцевого тексту теж генерували всі можливі підрядки довжини n, але вже без хешування чи віконного мінімуму, а потім порівнювали дві множини n-грам між собою.

Для символних n-грам така множина включає послідовності сирцевих символів, а для токенних – послідовності токенів, отриманих після лексичного аналізу. Результати порівняння надані на рис. 3.9.

Рисунок 3.9 – Результат перевірки методом n-грам

У якості головної перевірки для порівняння ефективності усіх цих методів за основу використався метод difflib (рис. 3.10).

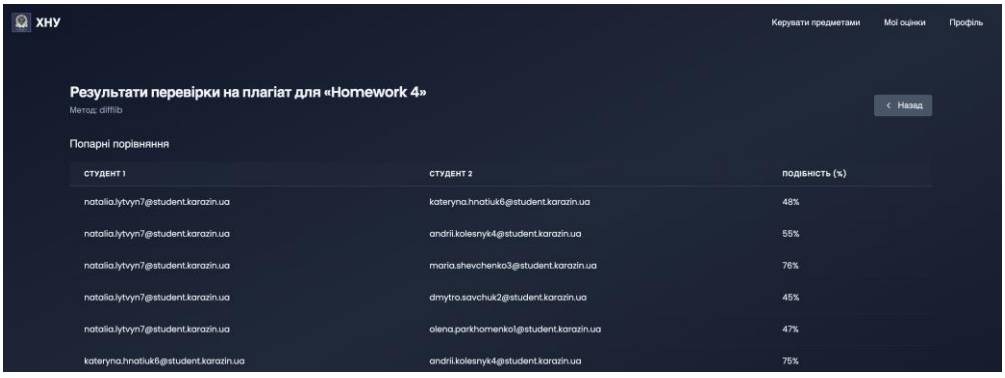

Рисунок 3.10 – Результат перевірки методом difflib

Символьний підхід показав помірний рівень схожості: більшість пар демонструють близько 50–80 % збігів, а лідером виявився dmytro.savchuk2 із 89 % максимальної подібності. Це свідчить про те, що difflib відчуває всі

текстові збіги, але легко завищує результати при загальних структурних шаблонах.

Token-based метод ігнорує нерелевантні пропуски та відступи, водночас підкреслює подібність логіки коду, яка явно повторюється у кількох роботах.

AST-порівняння виявило найнижчі загальні значення серед трьох «глибоких» методів: максимуми укладаються в 70–77 %. Структурні розбіжності – різні назви функцій чи порядок блоків – суттєво знижують показник, навіть якщо алгоритмічна суть залишається схожою.

Методи n-grams і Winnowing продемонстрували найжорсткішу селекцію.

При n-grams переважно спостерігаються цифри в діапазоні 15-62 %, із максимумом у 62 % для kateryna.hnatiuk6.

Winnowing дав ще нижчі результати – 12-58 % – адже відразу відсіює дрібні збіги та звертає увагу лише на найстійкіші «відбитки» коду.

Отже, по-перше, усі методи дають різні «дзеркала» схожості: від дуже чутливих до тексту до виважених, що витримують навіть зміни найменувань і стилю.

По-друге, верифікація одночасно швидкими та глибокими підходами допомагає розділити випадкові та навмисні позичання.

І, по-третє, за допомогою комбінованого аналізу можна чітко виокремити студентів із реальним ризиком копіювання – коли і негнучкі, і гнучкі алгоритми сходяться на високих відсотках збігу.

3.6 Інтерфейс користувача

Інтуїтивно зрозуміла навігація є основою будь-якої веб-платформи, оскільки вона знижує поріг входу нових користувачів та підвищує загальну ефективність роботи. У системі вона складається із трьох ключових елементів: головного меню, «хлібних крихт» і шапки сторінки (рис. 3.11).

Кожен із них виконує свою роль у допомозі швидко орієнтуватися, знаходити потрібні розділи та стежити за важливими повідомленнями.

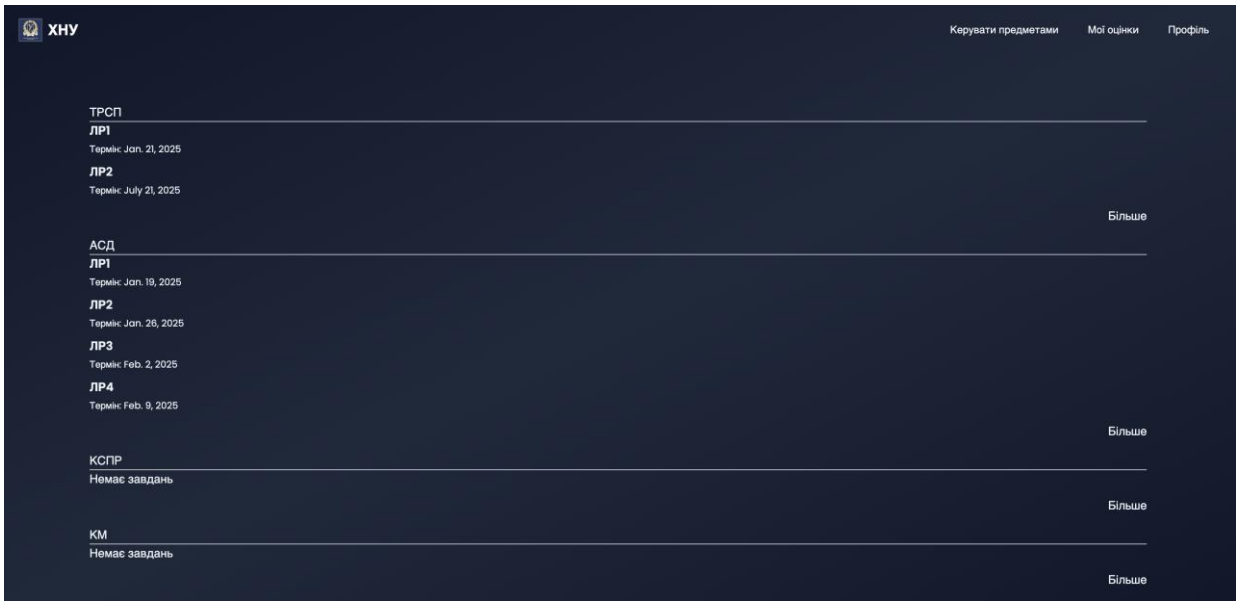

Рисунок 3.11 – Головна сторінка

Головне меню на рис. 3.11 розміщене у верхній частині екрану й містить чотири базові пункти: «Курси», «Профіль», «Налаштування» та «Вихід».

Натискання на «Курси» веде до списку всіх курсів, до яких має доступ користувач. Розділ «Профіль» відкриває персональні дані й історію дій, «Налаштування» дає змогу змінити контактну інформацію, пароль чи методи входу, а «Вихід» завершує сесію та повертає на сторінку аутентифікації.

Під головним меню розташована компактна горизонтальна лінія, що відображає шлях від головної сторінки до поточного розділу.

Після входу викладач потрапляє на персоналізований дашборд, що об’єднує ключову інформацію по всіх його курсах і дозволяє переходити до будь-яких дій за два кліки. Центральною частиною інтерфейсу є перелік активних курсів з попереднім оглядом, статистикою та кнопками для оперативного керування.

Угорі відображається список курсів на рисунку 3.12.

Рисунок 3.12 – Сторінка вчителя

Для кожного курсу на рис. 3.12 виводяться назва та короткий опис, а поряд розташовані кнопки «Перейти до курсу» та «Створити новий курс». Перша миттєво відкриває деталі обраного курсу, тоді як друга дозволяє створити новий курс.

Після натискання на конкретне завдання студент потрапляє на детальну сторінку на рис. 3.12 зі всією необхідною інформацією та інструментами для виконання. Вгорі розташовано блок із заголовком завдання та коротким описом, далі наведено дедлайн і поточний ліміт спроб.

На рис. 3.13 дедлайн відображається чіткою часовою міткою, а поруч виводиться індикатор кількості використаних і доступних спроб, що допомагає планувати роботу без ризику автоматичної блокування.

Безпосередньо під описом знаходиться зона для завантаження рішення: студенти можуть або перетягнути файл у спеціальний прямокутник, або натиснути кнопку «Оберіть файл».

Після вибору файлу одразу активується інтегрований редактор коду з підсвічуванням синтаксису, автодоповненням та лінійними номерами.

Рисунок 3.13 – Сторінка завдання по предмету

Внизу сторінки відображається історія подань на рисунку 3.13. Кожен запис містить часову мітку, результат автоматизованого тестування.

Під таблицею тестів розташовано блок «Коментар викладача», де відображаються текстові зауваження й рекомендації. Якщо викладач залишив фідбек, студент бачить його в окремому полі, що можна прокручувати.

Висновки до розділу 3

У третьому розділі дипломної роботи була впроваджена система, яка задовольняє усі визначені вимоги та демонструє узгоджену роботу п’яти мікросервісів у контейнеризованому середовищі. Django-додаток platform централізовано обробляє маршрути, керує моделями даних і надає REST-інтерфейси, тоді як antiplagiat, filestorage, coderunner і database виконують свої функціональні завдання автономно.

Модуль перевірки на плагіат реалізує п’ять методів порівняння: символний, токен-базований, AST-аналіз, n-грами й алгоритм Winnowing. Кожен алгоритм повертає рівень схожості й перелік ділянок з високою кореляцією, що дозволяє викладачу приймати аргументовані рішення. Інтерфейс об’єднує доступ до курсів, завдань, подань та звітів у єдиній консолі.

ВИСНОВКИ

У кваліфікаційній роботі виконане дослідження об’єднало огляд існуючих методів автоматизованого оцінювання коду та виявлення плагіату зі створенням інтегрованої мікросервісної платформи для академічного середовища. Аналіз історії розвитку і сучасних рішень у першому розділі показав багатоваріантність підходів від простих побайтових співставлень до глибокої семантичної обробки, а також обґрунтував вибір Python і фреймворку Django як основи для надійного й масштабованого серверного рішення.

Постановка завдання і проектування системи на другому етапі гарантували чітку реалізацію як функціональних, так і нефункціональних вимог.

Детальний опис об’єктів моделі даних – користувачів, курсів, подань та оцінок – а також вибір мікросервісної архітектури сприяли розділенню відповідальності між компонентами і полегшили майбутнє розширення платформи.

Реалізація платформи включила організацію backend-частини в рамках Django-додатку з REST-інтерфейсом, що обробляє маршрути та моделі, а також окремі сервіси для зберігання файлів, запуску автотестів і перевірки плагіату. Запуск студентського коду у Docker-контейнерах із захистом ресурсів забезпечив надійність і відтворюваність, результати тестування й плагіат-скори були уніфіковані у JSON й збережені в базі даних. Frontend-інтерфейс із шаблонами на Django-Template відтворив усі необхідні графічні форми для студентів і викладачів, включно з інтегрованим редактором коду.

Проведені експерименти продемонстрували ефективність п’яти різних алгоритмів порівняння коду: від рядкових і токен-базованих методів до AST-аналітики й методів на основі n-грам і Winnowing.

Результати їхнього застосування в реальних групах студентів підтвердили здатність системи виявляти як явні копії, так і структурні й семантичні відповідності з високою точністю та гнучкістю налаштувань.

Розроблена модель показала готовність до інтеграції з зовнішніми навчальними платформами, мобільними клієнтами та аналітичними інструментами.

Створена архітектура закладає основу для подальшого розширення функціональності: додавання нових мов програмування, вдосконалення семантичних алгоритмів і розвиток адаптивних сервісів сповіщень.

Завершуючи, ця дипломна робота внесла вагомий внесок у створення сучасного рішення для автоматизованого оцінювання й контролю академічної доброчесності. Отримані результати можуть бути використані для впровадження платформи у вищих навчальних закладах і подальшого дослідження у напрямках машинного навчання для аналізу програмних артефактів.

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАНЬ

1. PLATO – перша система комп’ютерного тестування студентів, розроблена в Університеті Іллінойсу в 1960-х роках. Режим доступу: https://en.wikipedia.org/wiki/PLATO_(computer_system)
2. Blackboard Learn – одна з перших масових LMS-платформ із вбудованим модулем автоматичного тестування, запущена в 1997 році. Режим доступу: https://en.wikipedia.org/wiki/Blackboard_Learn
3. Moodle – відкрита платформа дистанційного навчання, випущена 2002 року, що надала гнучкі інструменти для автоматизованого оцінювання. Режим доступу: https://en.wikipedia.org/wiki/Moodle
4. Sakai – відкрите програмне забезпечення для e-learning та дослідницьких проектів, випущене 2004 року, з розширеними можливостями автоматичних тестів. Режим доступу: https://en.wikipedia.org/wiki/Sakai_Project
5. E-learning – історичний огляд розвитку дистанційного навчання в мережі, який описує перехід від CAI до сучасних систем тестування. Режим доступу: https://en.wikipedia.org/wiki/E-learning
6. Automated Grading – огляд концепції автоматичного оцінювання програмних робіт із аналізом переваг у швидкості, масштабованості та об’єктивності. Режим доступу: https://en.wikipedia.org/wiki/Automated_grading_system
7. Edinburgh University CS Education Group. Automated assessment of programming assignments: speed and accuracy benefits. Режим доступу: https://www.inf.ed.ac.uk/teaching/cseducation/AutoAssessment.pdf
8. Coupling scalability and performance in cloud-based autograding systems for MOOCs / Liang, S., et al. – 2020. Режим доступу: https://dl.acm.org/doi/10.1145/3372885

9. Bias in Automated Essay Grading: Examining Objectivity in Machine-Based Assessment / Williamson, D. – 2018. Режим доступу: https://onlinelibrary.wiley.com/doi/full/10.1111/aeq.12123
10. Massive Open Online Courses and Automated Grading: Enhancing Speed and Fairness in Assessment / Hodges, C. – 2019. Режим доступу: https://www.sciencedirect.com/science/article/pii/S0360131518302346
11. Gronlund, N. E., & Wertenberger, E. M. (2005). Measurement and Assessment in Teaching. 10th ed. Pearson. Розділ про типи тестових завдань та їх реалізацію. Режим доступу: https://www.pearson.com/gronlund-assessment
12. Falk, J. C., & Balling, R. J. (1980). “Research in Computer-Based Testing at the University of Illinois.” Educational Communication and Technology Journal, 28(2), 55–62. Аналіз ресурсних витрат на створення і підтримку CAI-систем. Режим доступу: https://link.springer.com/article/10.1007/BF02765156
13. Romero, C., & Ventura, S. (2005). “Data Mining in Course Management Systems: Moodle Case Study and Tutorial.” Computers & Education, 51(1), 368–384. Оцінка масштабованості автоматичного тестування в e-learning. Режим доступу: https://www.sciencedirect.com/science/article/pii/S0360131507000831
14. Rabin, M. O., & Karp, R. M. (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31(2), 249–260. Режим доступу: https://doi.org/10.1147/rd.312.0249
15. Overview of Word String Recognition Methods. ResearchGate. Режим доступу: https://www.researchgate.net/figure/Overview-of-word-string-recognition-methods_fig1_319590347
16. Baker, B. S. (1995). A program for identifying duplicated code. Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 183–194. Режим доступу: https://doi.org/10.1145/199448.199462
17. Baxter, I. D., Yahin, A., Moura, L., Sant’Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. Proceedings of the International Conference

on Software Maintenance, 368–377. Режим доступу: https://doi.org/10.1109/ICSM.1998.744030

18. Zhang, K. & Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing, 18(6), 1245–1262. Режим доступу: https://doi.org/10.1137/0218071
19. Ahmed Sultan, Mahmoud Salim, Amina Gaber, Islam El Hosary "WESSA at SemEval-2020 Task 9: Code-Mixed Sentiment Analysis using Transformers" ResearchGate. Режим доступу: https://www.researchgate.net/figure/TF-IDF-vectorization-process_fig1_344335518
20. Krinke, J. (2001). Identifying similar code with program dependence graphs. Proceedings of the Eighth Working Conference on Reverse Engineering, 301–309. Режим доступу: https://doi.org/10.1109/WCRE.2001.957212
21. Baxter, I., Pidgeon, C., & Mehlich, M. (2004). DTL diff: A refinement-based approach to code clone detection and transformation. Science of Computer Programming, 53(2), 197–210. Режим доступу: https://doi.org/10.1016/j.scico.2004.06.002
22. Allamanis, M., Peng, H., & Sutton, C. (2016). A survey of machine learning for big code and naturalness. *ACM* Computing Surveys, 51(4), 1–37. Режим доступу: https://doi.org/10.1145/2934664
23. Rabin, M. O., & Karp, R. M. (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31(2), 249–260. Режим доступу: https://doi.org/10.1147/rd.312.0249
24. Manber, U. (1989). Finding similar files in a large file system. Proceedings of the USENIX Winter 1994 Technical Conference, 1–10. Режим доступу: https://www.usenix.org/legacy/publications/library/proceedings/winter94/full_papers/manber.pdf
25. Zhang, Y., Huo, S., & Yang, J. (2010). A fast tree,Äêto,Äêtree correction distance algorithm. Information Processing Letters, 110(24), 1118-1121. Режим доступу:: https://doi.org/10.1016/j.ipl.2010.09.010

26. Aiken, A. (1994). MOSS: A System for Detecting Software Plagiarism. University of California, Berkeley. Режим доступу: https://theory.stanford.edu/~aiken/moss.html
27. Juergens, E., Deissenboeck, F., & Hummel, B. (2010). JPlag: Finding Plagiarisms among a Set of Programs. Proceedings of the 16th Working Conference on Reverse Engineering, 125–134. Режим доступу: https://doi.org/10.1109/WCRE.2009.46
28. Baker, B. S. (1995). A Program for Identifying Duplicated Code. Proceedings of the 1995 IEEE Symposium on Software Tools and Engineering Practice, 49-58. Режим доступу: https://doi.org/10.1109/STEPS.1995.483865
29. Komondoor, R., & Horwitz, S. (2001). Using Slicing to Identify Duplication in Source Code. Proceedings of the 8th International Symposium on Static Analysis, 40-56. Режим доступу: https://doi.org/10.1007/3-540-47857-5_3
30. Baxter, I. D., Yahin, A., Moura, L. M., Sant'Anna, M., & Bier, L. (1998). Clone Detection Using Abstract Syntax Trees. Proceedings of the International Conference on Software Maintenance, 368-377. Режим доступу: https://doi.org/10.1109/ICSM.1998.738438
31. Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: A Multi-Language Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering, 28(7), 654–670. Режим доступу: https://doi.org/10.1109/TSE.2002.1019476
32. Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting. Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 76–85. Режим доступу: https://doi.org/10.1145/872757.872770
33. Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming, 74(7), 470–495. Режим доступу: https://doi.org/10.1016/j.scico.2008.09.009

34. CodeGrade. (2021). CodeGrade – Automated Feedback and Assessment Platform. Режим доступу: https://codegrade.com
35. JPlag. (2021). JPlag – Software Similarity Detection. Режим доступу: https://jplag.ipd.kit.edu
36. Aiken, A. (1994). MOSS – Measure of Software Similarity. Stanford University Technical Report.
37. PlagScan. (2022). PlagScan – Advanced Plagiarism Detection. Режим доступу: https://www.plagscan.com
38. Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python. O'Reilly Media.
39. Ramírez, S. (2021). FastAPI – The Good, the Bad and the Ugly. Real Python. Режим доступу: https://realpython.com/fastapi/
40. Newman, S. (2015). Building Microservices. O'Reilly Media.
41. Vilk, Y. (2019). Decomposing the Monolith: A Practical Guide. ThoughtWorks Technology Radar. Режим доступу: https://www.thoughtworks.com/radar/techniques#decompose

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **Бакалавр**
Галузь знань: 12 – Інформаційні технології
Спеціальність: 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри комп'ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«02» жовтня 2024року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Кононенко Михайло Олексійович
(прізвище, ім'я, по батькові студента)

1. Тема роботи: **«КОМП'ЮТЕРНА МОДЕЛЬ СИСТЕМИ АВТОМАТИЧНОГО ОЦІНЮВАННЯ КОДУ ТА ПЕРЕВІРКА СТУДЕНТСЬКИХ РОБІТ НА ПЛАГІАТ»**
керівник роботи **Чуб Ольга Ігорівна, кандидат економічних наук**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету ***16 квітня 2025 року № 4101-5/962***
2. Строк подання студентом роботи ***30 травня 2025 року***
3. Перелік питань, які потрібно розробити:
   1) проаналізувати сучасні методи виявлення плагіату в програмному коді та механізми його автоматичного оцінювання;
   2) дослідити питання захисту даних у веб-системах та обрати відповідні криптографічні технології для забезпечення безпеки;
   3) розробити модель захисту даних користувачів із використанням сучасних засобів шифрування та врахуванням можливих вразливостей;
   4) спроєктувати архітектуру системи, що включає фронтенд, бекенд, базу даних, механізми тестування коду та перевірки на плагіат;
   5) розробити функціональний інтерфейс користувача для подання, перегляду та перевірки кодових завдань;
   6) реалізувати серверну частину системи з можливістю обробки й збереження робіт, автоматичного оцінювання та виявлення плагіату;
   7) інтегрувати механізм виконання коду в реальному часі з підтримкою перевірки коректності розв'язків;
   8) розробити API для взаємодії з іншими сервісами та забезпечення масштабованості системи;
   9) провести тестування роботи системи, оцінити точність виявлення плагіату та її стабільність під навантаженням;
   10) створити документацію до системи та сформулювати рекомендації щодо її впровадження та подальшого розвитку.

План роботи:

№ з/п	Назви етапів роботи	Термін виконання
1.	Формулювання мети, завдань, об'єкта, предмета дослідження та актуальності теми	01.10.2024 – 15.11.2024
2.	Проведення огляду наукових і галузевих досліджень, пов'язаних з автоматизованим оцінюванням коду та виявленням плагіату	02.10.2024 – 30.11.2024
3.	Аналіз наявних підходів до перевірки коду на плагіат, включно з текстовими, структурними та семантичними методами	01.12.2024 – 15.12.2024
4.	Обґрунтування вибору інструментів, мов програмування, фреймворків і бібліотек для реалізації системи	16.12.2024 – 31.12.2024
5.	Проєктування загальної архітектури програмної системи з урахуванням вимог до безпеки, масштабованості та модульності	01.01.2025 – 15.01.2025
6.	Розробка серверної логіки (бекенду) для приймання, перевірки, зберігання та оцінювання програмних рішень	16.01.2025 – 30.01.2025
7.	Проєктування та реалізація користувацького інтерфейсу (frontend) для взаємодії студентів і викладачів із системою	01.02.2025 – 15.02.2025
8.	Інтеграція механізмів виконання коду з автоматизованим тестуванням та генерацією зворотного зв'язку	16.02.2025 – 28.02.2025
9.	Розробка та впровадження алгоритмів виявлення плагіату з використанням обраних методів аналізу коду	01.03.2025 – 15.03.2025
10.	Створення структури бази даних для зберігання інформації про роботи, результати оцінювання та звіти про подібність	16.03.2025 – 31.03.2025
11.	Налаштування серверного середовища та конфігурацій для забезпечення стабільної роботи системи	01.04.2025 – 15.04.2025
12.	Проведення комплексного тестування функціональності та стійкості системи до збоїв	16.04.2025 – 30.04.2025
13.	Аналіз продуктивності та масштабованості системи під навантаженням	01.05.2025 – 15.05.2025
14.	Підготовка технічної документації: опис архітектури, коду, API та інструкцій для користувачів	01.05.2025 – 15.05.2025
15.	Оформлення пояснювальної записки згідно з вимогами та підготовка до захисту	16.05.2025 – 30.05.2025

5. Дата видачі завдання ***02 жовтня 2024 року.***

Студент **Кононенко М.О.** _____

Керівник роботи **Чуб О. І.** _____

Додаток Б

Затверджую

_____

«_____»_____2025 р.

Технічне завдання
на розробку програмного виробу
«Комп'ютерна модель системи автоматичного оцінювання коду та перевірки студентських робіт на плагіат»

1.	Введення	1.1. Назва: Комп'ютерна модель системи автоматичного оцінювання коду та перевірки студентських робіт на плагіат 1.2. Галузь застосування: Освітні технології та автоматизація процесів оцінювання вищої школи з акцентом на систему підтримки академічної доброчесності
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 123 – Комп'ютерна інженерія 2.2. Завдання на кваліфікаційну роботу бакалавра № № 4101-5/962 від 16.04.2025 (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета: розробити комп’ютерну модель системи автоматичного оцінювання програмного коду та перевірки студентських робіт на плагіат із використанням технологій веб-розробки та алгоритмів аналізу програмного коду. 3.2. Призначення розробки створення єдиної веб-системи, що поєднає функції управління курсами й завданнями, автоматичного запуску тестових сценаріїв, аналізу вихідного коду на наявність запозичень та зручного перегляду результатів. Основні завдання – надати викладачеві інструменти для швидкого формування вправ і тестів, студенту – середовище для подання та коригування коду, а адміністрації – засоби моніторингу успішності та доброчесності навчального процесу. 3.3. Вхідні та вихідні дані

		До вихідних даних належать опис курсу й завдань, скрипти тестів та алгоритмічні шаблони перевірки, параметри порогів схожості для модуля антиплагіату. Вхідні дані формуються студентськими сабмітами у вигляді файлів з вихідним кодом, а також історією попередніх версій рішень, що дозволяє порівняти декілька ітерацій і відстежити динаміку змін та успішність проходження тестів.
4.	Технічні вимоги до програмного виробу	4.1. Функціональні характеристики Інтерфейс повинен бути інтуїтивно зрозумілим як для студентів, так і для викладачів, забезпечувати швидке створення завдань і завантаження рішень, а також негайно відображати результати автоматизованого тестування та звіти про плагіат. Модуль порівняння коду має обробляти як прямі, так і замасковані копіювання, зберігати точність виявлення навіть за великої кількості сабмітів і забезпечувати легке налаштування порогів подібності. Система повинна підтримувати одночасну роботу сотень користувачів без затримок і гарантувати стабільний розподіл навантаження між сервісами. 4.2. Вимоги до надійності: усі дані про користувачів, завдання, сабміти та їхні оцінки мають зберігатися з використанням щоденного резервного копіювання бази даних і файлового сховища. Передбачено валідацію кожного поданого файлу (перевірка розширення, розміру та коректності коду) та обробку помилок під час запуску автотестів і модуля антиплагіату, щоб уникнути втрати інформації та некоректних результатів. 4.3. Вимоги до умов експлуатації: система повинна коректно працювати в локальній мережі університету. 4.4. Вимоги до складу і параметрів технічних засобів: для розгортання достатньо серверів із процесорами мінімум Intel Xeon E3 або еквівалентами AMD, 8 ГБ оперативної пам’яті на вузол та сучасним SSD-диском. Docker та Kubernetes-кластери можуть бути розміщені на будь-яких ОС із підтримкою контейнеризації: Linux (різні дистрибутиви) або Windows Server

		4.5. Вимоги до інформаційної та програмної сумісності: підтримка різних СУБД (PostgreSQL, MySQL) забезпечує гнучкість вибору серед існуючих корпоративних рішень. Клієнтська частина повинна стабільно працювати в провідних браузерах (Chrome, Firefox, Edge) та на мобільних пристроях із адаптивним дизайном. 4.6. Вимоги до маркування та упаковки: дистрибутиви поставляються як набір Docker-образів із чіткими тегами версій у приватному реєстрі. Кожен образ містить усі залежності, а версіонування виконується за семантичним стандартом (Major.Minor.Patch) для прозорого оновлення. 4.7. Транспортування і зберігання: контейнери зберігаються в захищеному Docker-реєстрі з доступом через VPN. Файли образів можна завантажувати та передавати на інші хости без обмежень, забезпечуючи швидке розгортання на нових серверах. 4.8. Спеціальні вимоги: не передбачено додаткових умов для експлуатації чи безпеки, окрім стандартних практик контейнеризації та налаштування мережевих політик. Безпека аутентифікації базується на Django-механізмах з підтримкою OAuth і SSL/TLS шифруванням каналів зв’язку.
5.	Вимоги до програмної документації	Вимоги до програмної документації До складу програмної документації веб-системи автоматичного оцінювання коду та перевірки плагіату включені: 1) Повний опис архітектури та компонентів проєкту (модулі platform, testrunner, antiplagiat, filestorage, database) з поясненням їхньої взаємодії та API-контрактів, розділ 3 пояснювальної записки. 2) Технічне завдання на розробку системи, де визначені функціональні та нефункціональні вимоги, наведено як Додаток Б до пояснювальної записки. 3) Опис алгоритмів порівняння коду – difflib, tokenize + LCS, AST, n-grams і winnowing – з деталями реалізації в модулі antiplagiat, включений у додаток з псевдокодами та поясненнями.

<table>
<tr><td>6.</td><td>Вимоги до техніко-економічних показників</td><td colspan="2">Вимоги до техніко-економічних показників
Для обґрунтування продуктивності та вартості впровадження системи використовуються:
1) Технічне завдання (Додаток Б) із чітко визначеними метриками часу обробки одного сабміту та максимальної одночасної кількості запитів.
2) Розділ 3 пояснювальної записки з оцінкою навантаження на сервери при різних сценаріях (кількість студентів, частота запуску автотестів і плагіату) та вимогами до апаратного забезпечення.
3) Перелік джерел даних та наукових публікацій, що були використані для оцінки ефективності алгоритмів, статистики студентських навантажень і розрахунку економії часу викладачів у порівнянні з ручною перевіркою.</td></tr>
<tr><td rowspan="6">7.</td><td rowspan="6">Стадії і етапи розробки</td><td>Дата</td><td>Назва етапу</td></tr>
<tr><td>01.11.2024 – 15.11.2024</td><td>Формулювання мети, завдань, об’єкта, предмета дослідження та актуальності теми</td></tr>
<tr><td>16.11.2024 – 30.11.2024</td><td>Проведення огляду наукових і галузевих досліджень, пов’язаних з автоматизованим оцінюванням коду та виявленням плагіату</td></tr>
<tr><td>01.12.2024 – 15.12.2024</td><td>Аналіз наявних підходів до перевірки коду на плагіат, включно з текстовими, структурними та семантичними методами</td></tr>
<tr><td>16.12.2024 – 31.12.2024</td><td>Обґрунтування вибору інструментів, мов програмування, фреймворків і бібліотек для реалізації системи</td></tr>
</table>

		01.01.2025 – 15.01.2025	Проєктування загальної архітектури програмної системи з урахуванням вимог до безпеки, масштабованості та модульності
		16.01.2025 – 30.01.2025	Розробка серверної логіки (бекенду) для приймання, перевірки, зберігання та оцінювання програмних рішень
		01.02.2025 – 15.02.2025	Проєктування та реалізація користувацького інтерфейсу (frontend) для взаємодії студентів і викладачів із системою
		16.02.2025 – 28.02.2025	Інтеграція механізмів виконання коду з автоматизованим тестуванням та генерацією зворотного зв’язку
		01.03.2025 – 15.03.2025	Розробка та впровадження алгоритмів виявлення плагіату з використанням обраних методів аналізу коду
		16.03.2025 – 31.03.2025	Створення структури бази даних для зберігання інформації про роботи, результати оцінювання та звіти про подібність
		01.04.2025 – 15.04.2025	Налаштування серверного середовища та конфігурацій для забезпечення стабільної роботи системи

		16.04.2025 – 30.04.2025	Проведення комплексного тестування функціональності та стійкості системи до збоїв
		01.05.2025 – 15.05.2025	Аналіз продуктивності та масштабованості системи під навантаженням
		01.05.2025 – 15.05.2025	Підготовка технічної документації: опис архітектури, коду, API та інструкцій для користувачів
8.	Порядок контролю і приймання програмного продукту (моделі)	1. Перевірку ходу розробки програми виконувати раз в 3 тижні. 2. Захист розробленої моделі провести на засіданні Атестаційної комісії. 3. Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді в 1 примірнику на CD-R компакт-диску.	

Виконавець:
студент групи КІ-41
КОНОНЕНКО М.О.

Замовник:
к.е.н., доцент кафедри комп’ютерних систем та робототехніки
ЧУБ О.І.

Додаток В

Програма та методика випробувань програмного виробу «Комп'ютерна модель системи автоматичного оцінювання коду та перевірки студентських робіт на плагіат»

1. Об'єкт випробувань

1.1. Назва програмного виробу: Комп'ютерна модель системи автоматичного оцінювання коду та перевірки студентських робіт на плагіат.

1.2. Галузь застосування: Інформаційні технології. Освітні процеси в ІТ-навчанні та дистанційна оцінка якості коду.

1.3. Відомості про об’єкт випробувань запозичуються з розділу технічного завдання (Додаток Б до пояснювальної записки).

2. Мета випробувань

Підтвердження відповідності функціональних можливостей веб-платформи вимогам, викладеним у технічному завданні:

- автоматичне тестування студентських рішень,
- розрахунок та збереження оцінок,
- детекція плагіату за п’ятьма алгоритмами,
- коректне відображення результатів у веб-інтерфейсі.

3. Загальні положення

1. Підстави для проведення випробувань

Наказ керівника кафедри про затвердження складу атестаційної комісії для оцінки готовності програмного виробу до експлуатації.

2. Місце і тривалість випробувань

Приймальні випробування відбуваються у комп’ютерному класі кафедри протягом засідань атестаційної комісії.

3. Обсяг випробувань

Перевірка всіх заявлених функцій:

- створення та редагування завдань;
- завантаження та автотестування коду;
- побудова звітів про результати і плагіат;
- робота користувацьких ролей (студент, викладач, адміністратор).

Організації, які беруть участь у випробуваннях

Приймальні випробування організовує атестаційна комісія кафедри, до складу якої входять представники замовника, розробники платформи та запрошені фахівці. За потреби залучаються викладачі та системні адміністратори для перевірки розгортання й інтеграції.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

4.1 Функціональні характеристики

Програмний продукт має забезпечувати коректне створення завдань усіма ролями, зручний інтерфейс для завантаження коду та виведення результатів автотестів. Автоматична оцінка повинна бути об’єктивною, налаштовуватися під різні типи завдань і реагувати на зміни без втрати точності. Модуль перевірки плагіату має своєчасно видавати звіти з детальними метриками та підсвічуванням повторюваних фрагментів.

4.2 Надійність

Збереження даних про рішення, результати тестів та звіти антиплагіату гарантується за рахунок транзакцій бази даних і резервного копіювання. Помилка читання чи запису файлів не повинна призводити до втрати інформації, а система відновлення після збоїв забезпечить цілісність даних.

4.3 Умови експлуатації

Програмний продукт розрахований на роботу в локальній та мережевій інфраструктурі університету без постійного доступу до Інтернету. Відсутні обмеження щодо зовнішніх підключень, окрім стандартних HTTP/HTTPS.

4.4 Параметри технічних засобів

Запуск платформи можливий на будь-якому сервері під керуванням Windows Server, Linux або macOS. Для коректного виконання автотестів рекомендується процесор не нижче Intel Xeon E3 або аналогічний AMD, 8 ГБ оперативної пам’яті й SSD для зберігання кодів та звітів.

4.5 Інформаційна та програмна сумісність

Платформа сумісна з останніми версіями браузерів Chrome, Firefox і Edge. Серверна частина підтримує PostgreSQL або MySQL, а інтеграція з LMS здійснюється через LTI-інтерфейси.

4.6 Маркування та упаковка

Програмний продукт розповсюджується у вигляді Docker-образів з чіткими тегами версій. Кожний образ містить опис залежностей та інструкцію розгортання.

4.7 Транспортування та зберігання

Образи зберігаються в приватному реєстрі Docker. Для локальних бекапів архіви баз даних та медіафайлів копіюються на файловий сервер із дотриманням політик безпеки.

4.8 Спеціальні вимоги

Не передбачені додаткові вимоги щодо використання спеціального обладнання чи програмних компонентів.

5. Вимоги до програмної документації

До складу документації платформи автоматизованого оцінювання коду та перевірки на плагіат входять:

– Опис середовища розробки, включно з версіями мови Python, використаними бібліотеками та інструкціями з налаштування проекту;
– Програма та методика тестування сервісів автотестування і антиплагіату (представити як Додаток В до пояснювальної записки);

– Детальний опис архітектури системи, модулів і їх взаємодії, викладений у розділі 3 пояснювальної записки;
– Перелік використовуваних джерел: статей, технічних документацій і специфікацій зовнішніх API.

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Випробування виконуються на серверах і робочих станціях із встановленими Docker і Docker Compose. Для розробки та відлагодження використовуються IDE PyCharm і Postman, а дані зберігаються в СУБД PostgreSQL.

Випробування платформи автоматизованого оцінювання коду та перевірки на плагіат складаються з двох послідовних етапів: ознайомчого та основного функціонального тестування.

На ознайомчому етапі перевіряється повнота та якість програмної документації. Спочатку зіставляють наявні специфікації, інструкції зі встановлення та описи API з вимогами технічного завдання, щоб упевнитися в тому, що всі розділи присутні і відповідають заведеним стандартам оформлення. Потім оцінюють уніфікацію стилю та структурованість документації згідно з внутрішніми нормами.

Другий етап присвячений безпосередній перевірці функціональності сервісів. Насамперед розгортають усі мікросервіси через Docker Compose або Kubernetes і виконують базову перевірку працездатності: впевнитися, що веб-інтерфейс коректно запускається, REST-консоль доступна, а підключення до бази даних стабільне. Далі перевіряють кожен модуль окремо: запускають автотестування через сервіс TestRunner, переконуються, що команда запуску коректно обробляє шаблони тестів та формує звіт, потім викликають API модуля Antiplagiat із різними методами («difflib», «tokenize», «ast», «ngrams», «winnowing») і фіксують результати зіставлення.

Для оцінки коректності побудови звітів по закінченні кожної перевірки проводять аналіз повернених JSON-об’єктів, перевіряючи наявність полів із відсотком схожості, URL-посиланнями на докладні журнали та параметрами виконання тестів. У разі успішного проходження всіх тестів на обох етапах робота платформи вважається узгодженою з технічним завданням і готовою до впровадження в навчальному середовищі.

Для формалізації процесу випробувань пропонуються три номіновані сценарії:

- Тест підключення та автентифікації сервісів,
- Тест запуску функціональних автотестів із різними шаблонами,
- Тест комплексної перевірки плагіату з використанням усіх п’яти методів порівняння.

Тест 1 – Автентифікація

Переходимо до платформи

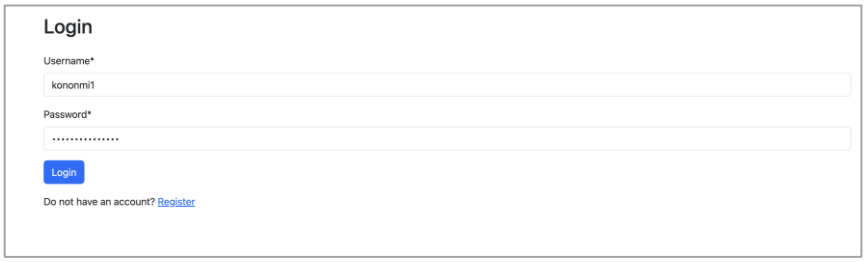

Рисунок В.1 – Тест 1

Вводимо дані:

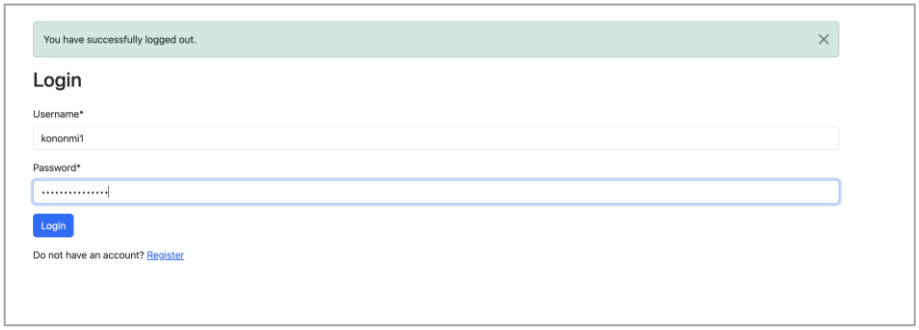

Рисунок В.1 – Тест 2

Головна сторінка (рис. В.3):

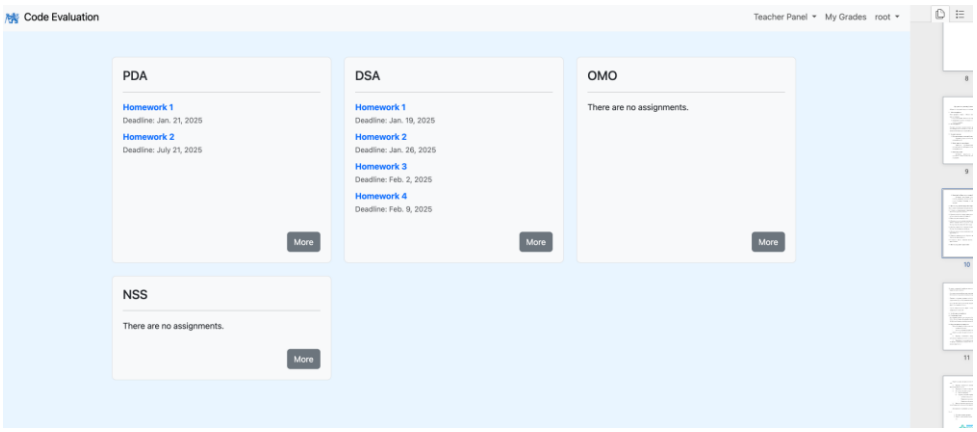

Рисунок В.3 – Тест 3

Тест 2 – Завантаження домашнього завдання

Переходимо в домашнє завдання 1 (рис. В.4):

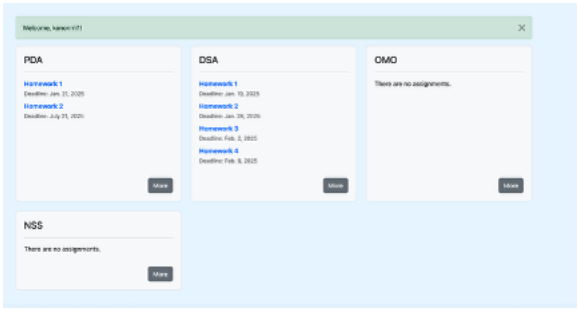

Рисунок В.4 – Тест 1

Відкриваємо домашнє завдання (рис. В.5):

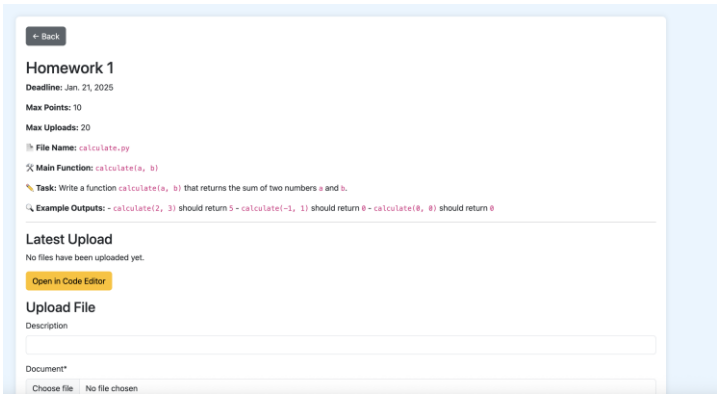

Рисунок В.5 – Тест 2

Додаємо рішення (рис. В.6):

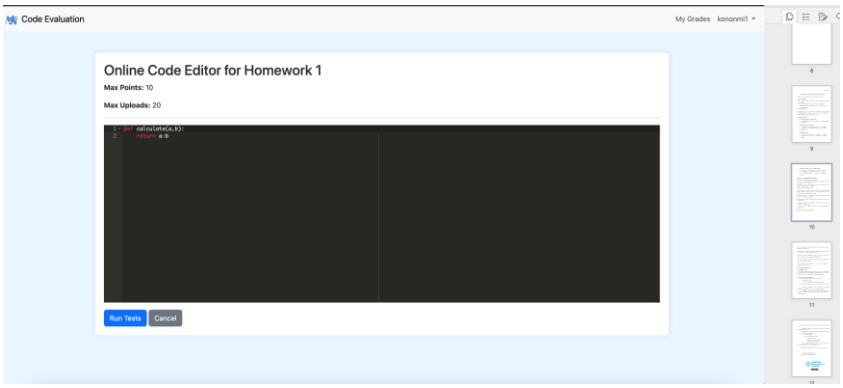

Рисунок В.6 – Тест 3

Відправляємо на перевірку (рис. В.7):

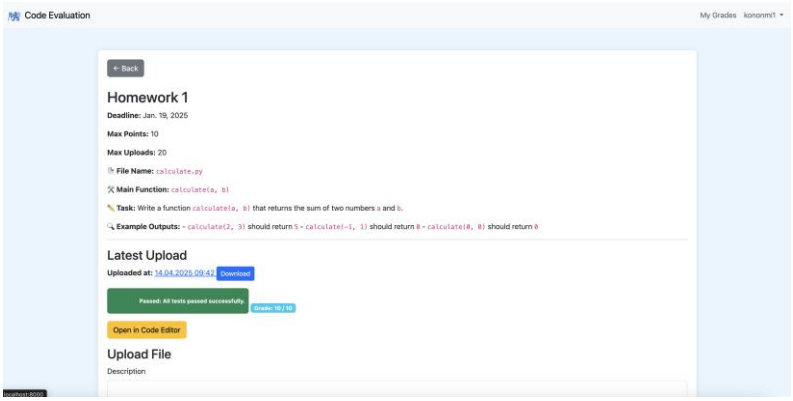

Рисунок В.7 – Тест 4

Тест 3 – Методи антиплагіату

Заходимо як вчитель (рис. В.8):

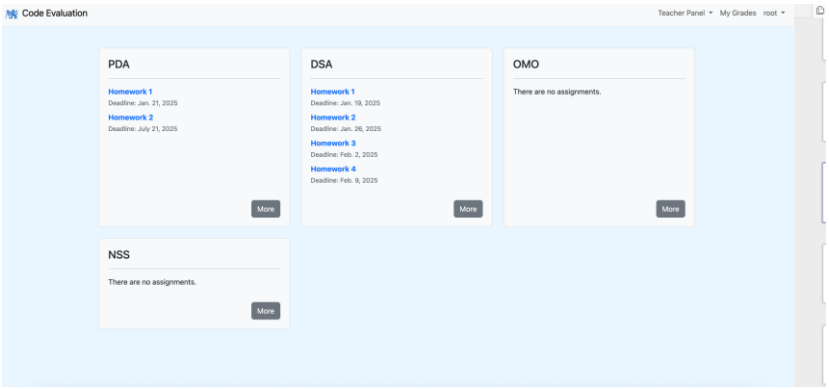

Рисунок 8 – В.3 Тест 1

Переходимо у панель предметів (рис. В.9):

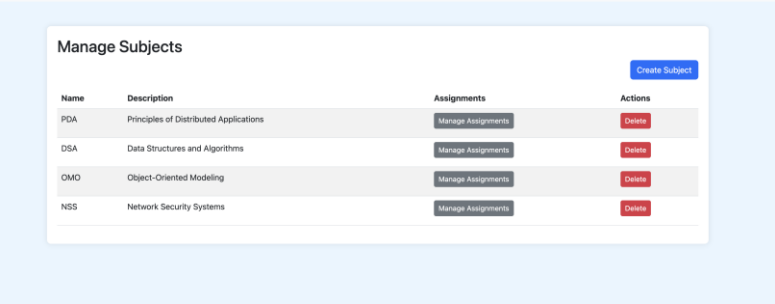

Рисунок В.9 – Тест 2

Обираємо предмет (рис. В.10):

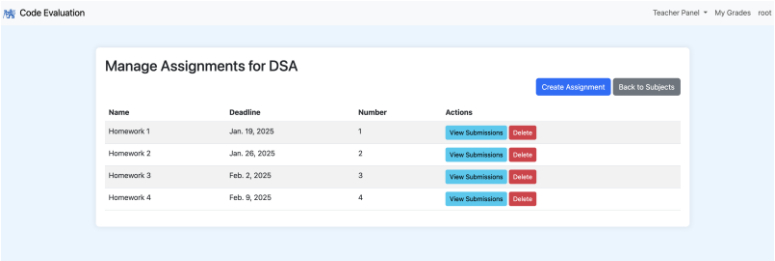

Рисунок В.10 – Тест 3

Тест вважається пройденим, якщо відбуваються вказані операції і їх відображення у прикладній програмі.

Висновки: тест 1 успішно пройшов випробування, тест 2 успішно пройшов випробування і тест 3 успішно пройшов випробування. Випробування пройшло успішно.

Виконавець:
студент групи КІ-41
КОНОНЕНКО М.О.

ДОДАТОК Г

ЛИСТИНГ ПРОГРАМНОГО КОДУ

1) Реалізація алгоритмів антиплагіату

```
import difflib
import io
import json
import argparse
import pathlib
import tokenize
import ast
import hashlib
import re
from typing import List, Set, Dict
def sequence_similarity(code1: str, code2: str) -> float:
    return difflib.SequenceMatcher(None, code1, code2).ratio()
def _normalize_tokens(code: str) -> List[str]:
    tokens: List[str] = []
    gen = tokenize.generate_tokens(io.StringIO(code).readline)
    ignorable = {
        tokenize.NEWLINE,
        tokenize.NL,
        tokenize.COMMENT,
        tokenize.INDENT,
        tokenize.DEDENT,
    }
    for toknum, tokval, *_ in gen:
        if toknum in ignorable:
            continue
        if toknum == tokenize.NAME:
```

```
                tokens.append("ID")
            elif toknum == tokenize.NUMBER:
                tokens.append("NUM")
            elif toknum == tokenize.STRING:
                tokens.append("STR")
            else:
                tokens.append(tokval)
        return tokens
    def token_lcs_similarity(code1: str, code2: str) -> float:
        t1 = _normalize_tokens(code1)
        t2 = _normalize_tokens(code2)
        return difflib.SequenceMatcher(None, t1, t2).ratio()
    def _ast_signature(code: str) -> List[str]:
        try:
            tree = ast.parse(code)
        except SyntaxError:
            return []
        return [type(node).__name__ for node in ast.walk(tree)]
    def ast_similarity(code1: str, code2: str) -> float:
        return difflib.SequenceMatcher(None, _ast_signature(code1),
_ast_signature(code2)).ratio()
    def _ngrams(s: str, n: int) -> Set[str]:
        return {s[i : i + n] for i in range(len(s) - n + 1)}
    def ngram_jaccard_similarity(code1: str, code2: str, n: int = 5) -> float:
        set1 = _ngrams(code1, n)
        set2 = _ngrams(code2, n)
        if not set1 and not set2:
            return 1.0
        return len(set1 & set2) / len(set1 | set2)
    def _fingerprints(text: str, k: int, w: int) -> Set[int]:
        clean = re.sub(r"\s+", " ", text).strip()
        hashes: List[tuple[int, int]] = []
```

```
    for i in range(len(clean) - k + 1):
        gram = clean[i : i + k]
        h = int(hashlib.md5(gram.encode()).hexdigest(), 16)
        hashes.append((h, i))
    if not hashes:
        return set()
    fps: Set[int] = set()
    for i in range(len(hashes) - w + 1):
        window = hashes[i : i + w]
        min_hash = min(window, key=lambda t: t[0])[0]
        fps.add(min_hash)
    return fps
def winnowing_similarity(code1: str, code2: str, k: int = 5, w: int = 4) -> float:
    fp1 = _fingerprints(code1, k, w)
    fp2 = _fingerprints(code2, k, w)
    if not fp1 and not fp2:
        return 1.0
    return len(fp1 & fp2) / len(fp1 | fp2)
def compare_all(code1: str, code2: str) -> Dict[str, float]:
    return {
        "sequence": sequence_similarity(code1, code2),
        "token_lcs": token_lcs_similarity(code1, code2),
        "ast": ast_similarity(code1, code2),
        "ngram_jaccard": ngram_jaccard_similarity(code1, code2),
        "winnowing": winnowing_similarity(code1, code2),
    }
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Порівняння двох файлів програмного коду п’ятьма методами."
    )
    parser.add_argument("file1", help="шлях до першого файлу")
```

```
    parser.add_argument("file2", help="шлях до другого файлу")
    args = parser.parse_args()
    code_a = pathlib.Path(args.file1).read_text(encoding="utf-8", errors="ignore")
    code_b = pathlib.Path(args.file2).read_text(encoding="utf-8", errors="ignore")
    result = compare_all(code_a, code_b)
    print(json.dumps(result, indent=2, ensure_ascii=False))
```

2) Реалізація функції перевірки коду

```
import sys, importlib.util, pathlib, glob, json, subprocess, types, math
def load_candidate(path):
    spec = importlib.util.spec_from_file_location("candidate", path)
    mod = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(mod)
    fn = getattr(mod, "dijkstra", None) or getattr(mod, "solve", None)
    if not callable(fn):
        raise AttributeError("expected a function named dijkstra or solve")
    return fn
def read_graph(fp):
    n, m = map(int, fp.readline().split())
    edges = [tuple(map(int, fp.readline().split())) for _ in range(m)]
    s = int(fp.readline())
    return n, edges, s
def parse_expected(text):
    return list(map(lambda x: float("inf") if x=="INF" else int(x), text.strip().split()))
def run_tests(candidate_fn, tests_dir):
    inputs = sorted(glob.glob(f"{tests_dir}/*.in"))
    total, passed = len(inputs), 0
    for inp_path in inputs:
        out_path = inp_path[:-3]+".out"
        if not pathlib.Path(out_path).exists():
            print(f"{inp_path}: missing .out file"); continue
```

```
            with open(inp_path) as f: n, edges, s = read_graph(f)
            with open(out_path) as f: expected = parse_expected(f.read())
            try:
                result = candidate_fn(n, edges, s)
            except Exception as e:
                print(f"{inp_path}: runtime error {e}"); continue
            ok = len(result)==len(expected) and all((a==b or (math.isinf(a) and math.isinf(b)))
for a,b in zip(result, expected))
            if ok: passed+=1; status="OK"
            else: status="FAIL"
            print(f"{pathlib.Path(inp_path).stem}: {status}")
        print(f"Passed {passed}/{total}")
    def main():
        if len(sys.argv)<3:
            print("usage: tester.py candidate.py tests_dir"); sys.exit(1)
        candidate_fn = load_candidate(sys.argv[1])
        run_tests(candidate_fn, sys.argv[2])
    if __name__=="__main__":
        main().
```