МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «Проєктування та розробка
автоматизованого сервісу перевірки та форматування академічних документів»

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп’ютерні науки
освітньо-професійної програми «Комп’ютерні науки»
Томусяк Максим Борисович

Керівник:
викладач, фахівець-практик Місай Володимир Віталійович

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

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _______________

(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «20» травня 2026 р.

Острог, 2026

АНОТАЦІЯ

кваліфікаційної роботи

на здобуття освітнього ступеня бакалавра

Тема: Проєктування та розробка автоматизованого сервісу перевірки та форматування академічних документів

Автор: Томусяк Максим Борисович

Науковий керівник: викладач, фахівець-практик Місай Володимир Віталійович

Захищена «………»………………… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *66 с., 11 рис., 1 табл., 4 додатків, 24 джерел.*

Ключові слова: *ВЕБЗАСТОСУНОК, АВТОМАТИЗАЦІЯ НОРМОКОНТРОЛЮ, ПЕРЕВІРКА ФОРМАТУВАННЯ, АКАДЕМІЧНІ ТЕКСТИ, АНАЛІЗ СТРУКТУРИ ДОКУМЕНТІВ.*

Короткий зміст праці:

Кваліфікаційна робота присвячена розробці веб-платформи для автоматизованої перевірки форматування академічних документів на відповідність встановленим стандартам. У роботі проведено аналіз предметної області та огляд вимог до оформлення наукових праць. Спроектовано архітектуру системи на основі FastAPI та Next.js із використанням PostgreSQL для збереження даних. Реалізовано механізми авторизації через Google OAuth, систему керування шаблонами та алгоритми аналізу параметрів шрифтів, відступів і структури документів. Впроваджено функціонал генерації детальних звітів про невідповідності та інтеграцію з Google Fonts для верифікації гарнітур. Проведено тестування функціональних можливостей та доступності інтерфейсу, що підтвердило точність системи у виявленні помилок оформлення.

Keywords: *WEB APPLICATION, COMPLIANCE AUTOMATION, FORMATTING CHECKS, ACADEMIC TEXTS, DOCUMENT STRUCTURE ANALYSIS.*

Summary of the work:

This thesis is dedicated to the development of a web platform for the automated verification of academic document formatting against established standards. The thesis analyzes the subject area and reviews the requirements for the formatting of academic papers. The system architecture was designed based on FastAPI and Next.js, using PostgreSQL for data storage. Authorization mechanisms via Google OAuth, a template management system, and algorithms for analyzing font parameters, indentation, and document structure were implemented. Functionality for generating detailed reports on non-compliance and integration with Google Fonts for font verification has been implemented. Testing of the system's functionality and interface usability was conducted, confirming the system's accuracy in detecting formatting errors.

(підпис автора)

Зміст

ВСТУП **6**

РОЗДІЛ 1. ДОСЛІДЖЕННЯ СТАНУ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ РОЗРОБКИ **8**

1.1 Аналіз предметної області та сучасних стандартів оформлення академічних текстів 8

1.2. Порівняльний аналіз існуючих аналогів та сервісів автоматизованого нормоконтролю 10

1.3. Обґрунтування концепції та формулювання вимог до автоматизованої системи 12

Висновок до розділу 1 14

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ **16**

2.1 Вибір та обґрунтування архітектурного стилю системи 16

2.2. Структурна організація та декомпозиція серверної частини проєкту 18

2.3 Проєктування структури бази даних та моделей даних 20

2.4. Розробка алгоритмів розбору (парсингу) та валідації документів формату .docx 23

2.5 Проєктування інтерфейсу користувача та карти навігації сервісу 25

Висновок до розділу 2 28

РОЗДІЛ 3. ТЕХНІЧНА РОЗРОБКА ТА РОЗГОРТАННЯ КЛІЄНТ-СЕРВЕРНОЇ ІНФРАСТРУКТУРИ **31**

3.1. Програмна реалізація серверної логіки на базі FastAPI 31

3.2. Розробка клієнтської частини застосунку засобами Next.js та Tailwind CSS.. 33

3.3. Програмна реалізація системи ідентифікації та автентифікації на основі протоколу Google OAuth 2.0 35

3.4. Контейнеризація та оркестрація сервісів засобами Docker та Docker-Compose. 38

3.5. Конфігурування вебсервера Nginx та забезпечення безпеки з Certbot SSL 41

3.6. Автоматизація розгортання (CI/CD) на базі Oracle Cloud та GitHub Actions.. 42

3.7. Реалізація панелі доступності 44

Висновок до розділу 3 46

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ТА ОЦІНКА ЯКОСТІ ФУНКЦІОНУВАННЯ СИСТЕМИ **48**

4.1. Методика та критерії оцінки якості функціонування системи 48

4.2. Порівняльний аналіз ручного та автоматичного нормоконтролю 50

4.3. Оцінка ефективності та аналіз результатів експерименту 52

Висновок до розділу 4 54

ВИСНОВОК..55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.. 57
ДОДАТКИ...60

ВСТУП

У сучасних умовах стрімкої цифровізації освітнього простору автоматизація рутинних процесів стає необхідною умовою для забезпечення високої якості навчання та наукової діяльності. Однією з найбільш трудомістких і водночас технічних складових підготовки кваліфікаційних робіт є їхній нормоконтроль - перевірка на відповідність суворим стандартам оформлення. Традиційний ручний підхід до цієї задачі супроводжується значними витратами часу як з боку студентів, так і викладачів, а також високим ризиком виникнення помилок через людський фактор.

Використання сучасних веб-технологій дозволяє трансформувати цей процес, перетворивши його на швидку та точну автоматизовану процедуру. Створення єдиної платформи, що забезпечує аналіз текстових параметрів безпосередньо у файлах формату .docx, дозволяє уніфікувати вимоги та підвищити загальну культуру підготовки академічних текстів. Саме тому розробка повнофункціональної інформаційної системи, що поєднує в собі потужні алгоритми обробки даних на серверній стороні та зручний користувацький інтерфейс, є актуальним завданням для сучасної ІТ-галузі в освіті.

Об’єкт дослідження - процес автоматизованої перевірки та стандартизації текстових документів в академічному середовищі.

Предмет дослідження - методи та засоби створення інформаційної системи для аналізу форматування документів, з акцентом на проєктуванні бекенд-логіки та інтеграції з хмарними сервісами.

Мета дослідження - розробка веб-платформи для автоматизованої перевірки форматування академічних документів на відповідність встановленим шаблонам.

Для досягнення поставленої мети потрібно розв’язати такі задачі дослідження:

- провести аналіз нормативних вимог до структури та оформлення академічних документів;

- спроектувати архітектуру інформаційної системи на основі мікросервісного підходу;
- обґрунтувати вибір технологічного стека (FastAPI, Next.js, PostgreSQL) для реалізації проекту;
- розробити алгоритми бекенд-обробки файлів для зчитування та порівняння параметрів форматування;
- реалізувати адміністративну панель для управління шрифтами та шаблонами перевірки;
- провести тестування розробленої системи на реальних документах.

Практична цінність створеної системи полягає в наступному функціоналі для користувачів та адміністратора:

- можливість швидкої реєстрації та входу через систему Google OAuth;
- завантаження та автоматизований аналіз документів формату .docx на відповідність обраному шаблону;
- отримання детального інтерактивного звіту про виявлені порушення (шрифти, відступи, інтервали);
- інтеграція з Google Fonts для верифікації використовуваних у документі гарнітур;
- гнучке управління шаблонами форматування для різних типів робіт (дипломні, курсові, звіти);
- централізоване керування бібліотекою дозволених шрифтів через панель адміністратора;
- перегляд історії перевірок та управління особистими документами.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ СТАНУ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ РОЗРОБКИ

1.1 Аналіз предметної області та сучасних стандартів оформлення академічних текстів

Трансформація освітнього простору в умовах глобальної цифровізації зумовлює перехід від традиційного документообігу до інтелектуальних систем управління контентом. Одним із найменш автоматизованих, проте критично важливих етапів підготовки наукових результатів (магістерських дисертацій, курсових проєктів, фахових статей) залишається нормоконтроль. Це процес верифікації документа на відповідність державним стандартам (наприклад, ДСТУ 3008:2015) та внутрішнім регламентам вищих навчальних закладів. Висока вартість часових ресурсів, що витрачаються на механічну перевірку відступів, гарнітур шрифтів та міжрядкових інтервалів, підкреслює необхідність створення спеціалізованого програмного забезпечення для автоматизації цих операцій.

Центральним елементом досліджуваної предметної області є автоматизована система нормоконтролю - програмний комплекс, що реалізує життєвий цикл обробки документа: від десеріалізації бінарного файлу до формування інтерактивного звіту про валідацію. На відміну від звичайних текстових редакторів, такий сервіс фокусується на структурній цілісності документа, проводячи глибокий аналіз XML-структури (у випадку формату .docx) для виявлення прихованих невідповідностей, які неможливо ідентифікувати візуально.

Реалізація такого проєкту в межах дипломної роботи передбачає розв'язання низки концептуальних та інженерних проблем:

1. Динамічна конфігурація та адаптивність вимог. Оскільки стандарти оформлення є гетерогенними (різняться залежно від типу документа чи кафедри), система не може базуватися на жорстко запрограмованих правилах. Необхідним є впровадження абстрактного рівня керування - гнучкої

адміністративної панелі з повним CRUD-циклом. Це дозволяє операторам системи формувати еталонні моделі (шаблони), визначаючи допустимі межі для метаданих тексту (розміри полів, ієрархія заголовків, параметри абзаців) без модифікації вихідного коду бекенду.

2. Структурна декомпозиція складних документів. Обробка файлів формату .docx (який є архівом XML-документів) потребує розробки алгоритмів, здатних відокремлювати змістовний текст від візуального «шуму» та службових тегів. Основна технічна складність полягає у коректній інтерпретації стилів, які застосовуються до окремих фрагментів (run-рівень) та абзаців загалом, для їх порівняння з реляційною базою даних шаблонів.

3. Інтерактивна візуалізація та користувацький досвід (UX). Результат роботи алгоритму не повинен бути простою констатацією помилок. Важливо забезпечити високу когнітивну доступність звіту: кожна знайдена невідповідність має бути класифікована та візуально прив'язана до контексту (наприклад, через систему підказок або виділення зон ризику). Презентаційна частина системи (Landing Page та Dashboard) має мінімізувати когнітивне навантаження на користувача, роблячи процес перевірки інтуїтивно зрозумілим навіть для осіб без технічної підготовки.

4. Високопродуктивна клієнт-серверна взаємодія. Враховуючи потенційно великий обсяг файлів та складність обчислювальних операцій на боці сервера (FastAPI), архітектура системи повинна гарантувати мінімальну затримку (latency). Це досягається шляхом оптимізації запитів до бази даних PostgreSQL та асинхронної обробки даних, що забезпечує чуйність інтерфейсу (Next.js) навіть під час інтенсивних перевірок.

Таким чином, розробка автоматизованого сервісу нормоконтролю є комплексною інженерною задачею, що поєднує методи системного аналізу, розробку складних алгоритмів парсингу та проєктування масштабованої веб-інфраструктури. Впровадження такої системи дозволить перевести процес академічної перевірки з

площини суб'єктивного людського сприйняття в площину об'єктивного цифрового аналізу.

1.2. Порівняльний аналіз існуючих аналогів та сервісів автоматизованого нормоконтролю

Сучасний ринок програмного забезпечення для обробки текстової інформації характеризується високим ступенем сегментації. Існуючі рішення можна розділити на кілька функціональних груп: універсальні текстові процесори, системи інтелектуальної перевірки правопису, сервіси контролю унікальності тексту та видавничі системи. Проте, аналіз показує, що жодна з цих категорій не вирішує повноцінно завдання автоматизованого нормоконтролю - перевірки документа на відповідність суворим формальним стандартам (ДСТУ, галузеві регламенти тощо).

1. Текстові процесори (на прикладі Microsoft Word)

Microsoft Word залишається стандартом де-факто для підготовки академічних робіт завдяки широкому інструментарію та розповсюдженості.

- Переваги: Потужний механізм «Стилів», що дозволяє задавати параметри абзаців, шрифтів та ієрархію заголовків.
- Недоліки та обмеження: Основним деструктивним фактором є «людський фактор». Більшість користувачів застосовують інструменти візуального, а не структурного форматування (наприклад, створення відступів за допомогою табуляції або пробілів, імітація розривів сторінок багаторазовим натисканням клавіші Enter). Це призводить до того, що документ, який виглядає коректно при друці, має порушену внутрішню структуру, що ускладнює його подальшу обробку та архівування. Крім того, Word не має вбудованої логіки для динамічної перевірки тексту на відповідність зовнішньому стандарту з видачею консолідованого звіту про помилки.

2. Системи перевірки правопису та граматики (Grammarly, LanguageTool)

Ці сервіси використовують алгоритми обробки природної мови (NLP) та машинного навчання для покращення якості контенту.

- Специфіка: Вони ефективно виявляють орфографічні, пунктуаційні та стилістичні огріхи, підвищуючи читабельність тексту.
- Обмеження: Дані інструменти фокусуються на семантиці та лінгвістиці, повністю ігноруючи технічні параметри документа. Вони не здатні проаналізувати метадані файлу, розміри полів, міжрядкові інтервали або специфічні вимоги до оформлення списків літератури, що є критичним для нормоконтролю.

3. Видавничі системи на основі розмітки (LaTeX)

LaTeX є потужним інструментом для підготовки складних технічних та наукових текстів.

- Переваги: Використовується декларативний підхід: автор описує структуру документа, а система автоматично генерує фінальний вигляд згідно з заданим класом (шаблоном). Це гарантує 100% відповідність стандартам без ризику випадкового зсуву елементів.
- Недоліки: Головною перешкодою є надзвичайно високий поріг входження та необхідність знання синтаксису розмітки, що нагадує програмування. Для більшості студентів гуманітарних та багатьох технічних спеціальностей цей інструмент є занадто складним. Крім того, вимогою більшості закладів вищої освіти України залишається подання робіт саме у форматі редагованих документів .docx.

4. Сервіси перевірки на плагіат (Unicheck, Turnitin, Plag.com.ua)

Це обов’язкові інструменти в системі забезпечення академічної доброчесності.

- Обмеження: Функціонал цих систем вузькоспеціалізований - пошук текстових запозичень у відкритих джерелах та внутрішніх репозиторіях. Питання візуального оформлення, дотримання шрифтових схем або правильності побудови таблиць залишаються поза межами аналізу цих сервісів.

1.3. Обґрунтування концепції та формулювання вимог до автоматизованої системи

На основі проведеного аналізу предметної області та існуючих програмних аналогів (п. 1.1 та 1.2) було визначено, що для вирішення проблеми трудомісткості нормоконтролю необхідне створення спеціалізованої веб-платформи. Концепція проєкту полягає у переході від ручної перевірки до автоматизованого алгоритмічного аналізу структури документа на основі гнучких еталонних шаблонів.

- Концептуальна модель системи

Основними принципами, покладеними в основу концепції розроблюваної системи, є:

- Централізоване управління стандартами: правила перевірки не повинні бути закладені в код системи, а мають зберігатися в базі даних як конфігуровані шаблони.

- Кросплатформеність та доступність: використання веб-технологій забезпечує доступ до сервісу з будь-якого пристрою без необхідності встановлення додаткового ПЗ.

- Відокремлення інтерфейсу від бізнес-логіки: застосування архітектурного стилю REST API дозволяє масштабувати обчислювальні потужності бекенду для складної обробки файлів незалежно від клієнтської частини.

- Формулювання функціональних вимог

Для забезпечення повноцінної роботи сервісу система повинна реалізовувати наступний набір функціональних можливостей:

- Модуль автентифікації: інтеграція з Google OAuth для забезпечення безпечного доступу та персоналізації результатів перевірки.
- Модуль управління шаблонами (Адміністративна панель): реалізація функціоналу для створення, редагування та видалення правил перевірки (типи шрифтів, розміри кегля, міжрядкові інтервали, параметри сторінок).
- Модуль аналізу документів: алгоритмічне зчитування файлів формату .docx, ідентифікація стилів кожного абзацу та порівняння їх із обраним шаблоном.
- Модуль звітності: формування детального списку знайдених помилок у режимі реального часу з чітким зазначенням характеру порушення стандарту.

- Нефункціональні вимоги та якісні характеристики

Окрім основного функціоналу, система повинна відповідати ряду технічних критеріїв:

- Продуктивність: час обробки стандартного документа (до 100 сторінок) не повинен перевищувати декількох секунд.
- Масштабованість: архітектура повинна підтримувати легке розгортання додаткових примірників сервісу за допомогою контейнеризації.
- Надійність: забезпечення цілісності даних користувачів та стабільної роботи під навантаженням через використання Nginx як реверс-проксі.
- Безпека: захист передачі даних між клієнтом та сервером за допомогою протоколу SSL/TLS.

- Обґрунтування технологічного стека

Вибір інструментарію обумовлений необхідністю поєднання високої швидкості розробки та ефективності обробки даних:

- Backend (FastAPI, Python): обрано через наявність потужних бібліотек для роботи з бінарними файлами (python-docx) та високу швидкість обробки асинхронних запитів.
- Frontend (Next.js/React): дозволяє створити динамічний інтерфейс з високою швидкістю рендерингу та зручною навігацією.
- Database (PostgreSQL): надійна реляційна модель ідеально підходить для зберігання структурованих шаблонів форматування та зв'язків між ними.
- Infrastructure (Docker, Nginx): забезпечує ізоляцію середовища виконання та простоту розгортання в хмарній інфраструктурі.

Висновок до розділу 1

У першому розділі проведено комплексний аналіз предметної області, досліджено існуючі рішення та обґрунтовано необхідність розробки автоматизованої системи нормоконтролю. Результати дослідження дозволяють зробити наступні висновки:

1. Визначено високу актуальність автоматизації нормоконтролю. Встановлено, що традиційний процес перевірки академічних документів на відповідність стандартам (зокрема ДСТУ 3008:2015) характеризується значними витратами часових ресурсів та високою ймовірністю помилок через «людський фактор». Цифрова трансформація освіти вимагає переходу до об'єктивного алгоритмічного аналізу структурної цілісності документів.
2. Виявлено технологічний розрив серед існуючих програмних аналогів. Порівняльний аналіз показав, що популярні інструменти (Microsoft Word,

Grammarly, сервіси антиплагіату) або не мають функціоналу для технічної валідації метаданих файлу, або є занадто складними для масового користувача (LaTeX). Це підтверджує відсутність на ринку доступного та спеціалізованого веб-рішення для автоматизованої перевірки форматування у форматі .docx.

3. Сформульовано концепцію гнучкої та масштабованої веб-платформи. Основним принципом системи визначено відокремлення правил форматування від програмного коду через впровадження механізму динамічних шаблонів. Це дозволяє адаптувати сервіс до варіативних вимог різних навчальних закладів без необхідності перепрограмування бекенду.

4. Визначено ключові функціональні та технічні вимоги. Встановлено, що система повинна забезпечувати безпечну авторизацію (Google OAuth), асинхронну обробку бінарних файлів, надання інтерактивних звітів у режимі реального часу та високу продуктивність при роботі з документами великого обсягу.

5. Обґрунтовано вибір сучасного технологічного стека. Для реалізації проєкту обрано комбінацію FastAPI (Python) для високопродуктивної серверної логіки та парсингу XML-структур документів, Next.js (React) для побудови чуйного інтерфейсу та PostgreSQL для надійного зберігання конфігурацій шаблонів. Контейнеризація за допомогою Docker забезпечить стабільність розгортання та масштабованість системи.

Таким чином, результати аналітичного етапу створюють достатню теоретичну та методичну базу для переходу до наступного етапу - детального проєктування архітектури та розробки програмного комплексу.

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Вибір та обґрунтування архітектурного стилю системи

Проєктування інформаційної системи для автоматизованого нормоконтролю вимагає архітектурного підходу, що поєднує високу обчислювальну потужність (для аналізу структури файлів) із високою чуйністю користувацького інтерфейсу. Для реалізації сервісу було обрано децентралізовану архітектуру вебзастосунку на основі патерну Decoupled Architecture, де клієнтська частина відокремлена від серверної бізнес-логіки за допомогою RESTful API, а рендеринг реалізовано за гібридною моделлю.

Вибір даної архітектурної моделі обґрунтований наступними інженерними чинниками:

1. Гібридна стратегія рендерингу контенту (Next.js Framework). Використання фреймворку Next.js дозволяє гнучко комбінувати серверний рендеринг (SSR) та статичну генерацію (SSG) з клієнтською взаємодією. Це вирішує дві ключові задачі:

- Оптимізація продуктивності (LCP): початкове завантаження інтерфейсу та посадкової сторінки відбувається максимально швидко завдяки пре-рендерингу на сервері.
- Інтерактивність: складні інтерфейси, такі як панель результатів аналізу документа з динамічними підказками, працюють як Single Page Application (SPA), забезпечуючи миттєву реакцію на дії користувача без повного оновлення сторінки.

2. Дотримання принципу розподілу відповідальності (Separation of Concerns). Архітектура системи чітко розмежовує зони відповідальності між двома незалежними сервісами:

- Presentation Layer (Next.js): фокусується виключно на формуванні користувацького досвіду (UX), управлінні станом інтерфейсу та маршрутизації.

- Service Layer (FastAPI): функціонує як ізольований обчислювальний центр, що відповідає за складні алгоритмічні операції - парсинг XML-структур .docx файлів, валідацію даних та взаємодію з базою даних PostgreSQL.

3. Висока пропускна здатність та асинхронність Backend-рівня. Застосування FastAPI, що базується на специфікації ASGI, дозволяє реалізувати неблокуючу обробку вхідних запитів. Враховуючи, що аналіз великих академічних документів є ресурсомісткою I/O-операцією, асинхронна модель дозволяє серверу обробляти велику кількість одночасних з’єднань, не створюючи затримок у роботі інших модулів системи.

4. Стандартизація обміну даними через RESTful API. Комунікація між фронтендом та бекендом здійснюється за протоколом HTTP з використанням формату JSON. Це забезпечує:

- Слабку пов'язаність (Loose Coupling): зміни у внутрішній логіці бекенду не потребують перебудови фронтенду, поки зберігається структура API-ендпоінтів.

- Універсальність: розроблений API-шар у майбутньому може бути використаний для інтеграції з мобільними застосунками або сторонніми освітніми платформами без необхідності модифікації ядра системи.

5. Контейнеризація та горизонтальне масштабування. Розподілена архітектура дозволяє використовувати Docker для ізоляції кожного компонента системи. Це спрощує процес розгортання (Deployment) та дозволяє незалежно масштабувати сервіси. Наприклад, у період пікових навантажень (сесії чи захисти дипломів) можна збільшити кількість екземплярів бекенд-сервісу для прискорення обробки черги документів, не витрачаючи ресурси на масштабування фронтенду.

Висновок. Обраний архітектурний стиль дозволяє створити відмовостійку та адаптивну систему. Відокремлення обчислювальної логіки від інтерфейсу гарантує стабільність платформи, а використання сучасних вебтехнологій забезпечує високий рівень комфорту для кінцевого користувача, наближаючи досвід роботи з вебсервісом до настільних додатків.

2.2. Структурна організація та декомпозиція серверної частини проєкту

Для забезпечення чистоти коду, легкості підтримки та масштабованості системи (згідно з принципами, викладеними в п. 2.1), серверну частину на базі FastAPI було структуровано за модульним принципом із чітким розділенням шарів відповідальності.

На рисунку 2.1 представлено ієрархічну структуру каталогів бекенд-сервісу.

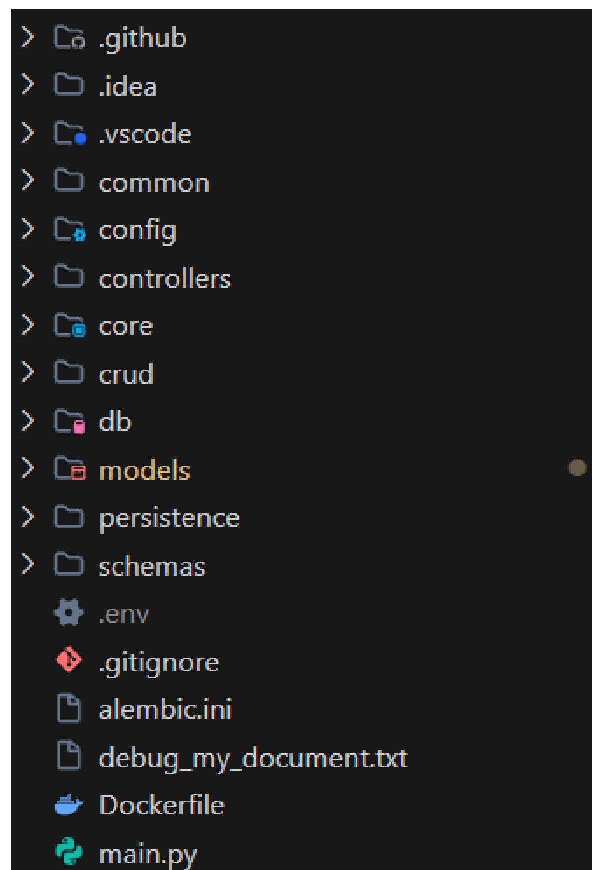

Рис. 2.1. Структура каталогів серверної частини застосунку

Джерело: розроблено автором

Основними компонентами структури проєкту є:

- controllers/ - шар обробки HTTP-запитів (ендпоінти). Тут зосереджена логіка маршрутизації, обробка вхідних параметрів та виклик відповідних сервісів бізнес-логіки.
- models/ - опис сутностей бази даних (SQLAlchemy моделі), що відображають структуру таблиць PostgreSQL.
- schemas/ - визначення схем валідації даних (Pydantic моделі) для вхідних та вихідних JSON-об’єктів (DTO - Data Transfer Objects).
- crud/ - рівень абстракції для виконання операцій створення, читання, оновлення та видалення даних у БД (Data Access Layer), що дозволяє відокремити SQL-запити від бізнес-логіки.
- core/ - шар бізнес-логіки (Service Layer). Містить основні алгоритми системи: логіку перевірки форматування (format_checker.py), інструменти для виправлення документів (document_formatter.py), а також сервіси для роботи з Google API та авторизацією.
- persistence/ - директорія міграцій Alembic. Містить скрипти для керування версіями схеми бази даних та історію її змін.
- db/ - ініціалізація підключення до бази даних та керування сесіями SQLAlchemy.
- common/ - спільні налаштування та константи системи, зокрема клас app_settings.py для зчитування параметрів оточення з файлу .env.
- alembic.ini - конфігураційний файл інструменту Alembic, що зв'язує базу даних із папкою міграцій.
- main.py - точка входу в додаток, де відбувається ініціалізація FastAPI, підключення роутерів та налаштування CORS і Middleware.

- Dockerfile - файл з інструкціями для створення Docker-образу бекенд-частини, що забезпечує ідентичність середовища розробки та розгортання.

Така організація файлової структури відповідає патерну Layered Architecture (Багатошарова архітектура), що дозволяє незалежно тестувати окремі модулі та забезпечує швидку адаптацію системи до нових функціональних вимог.

2.3 Проєктування структури бази даних та моделей даних

Ефективне функціонування сервісу нормоконтролю залежить від цілісності та швидкості доступу до даних. Як фундаментальне сховище інформації було обрано реляційну СКБД PostgreSQL, що вирізняється високою надійністю, підтримкою складних типів даних та відповідністю стандарту ACID. Проєктування логічної структури бази даних здійснювалося з дотриманням принципів нормалізації до третьої нормальної форми (3NF), що дозволило усунути надмірність даних та забезпечити несуперечливість інформації при операціях оновлення.

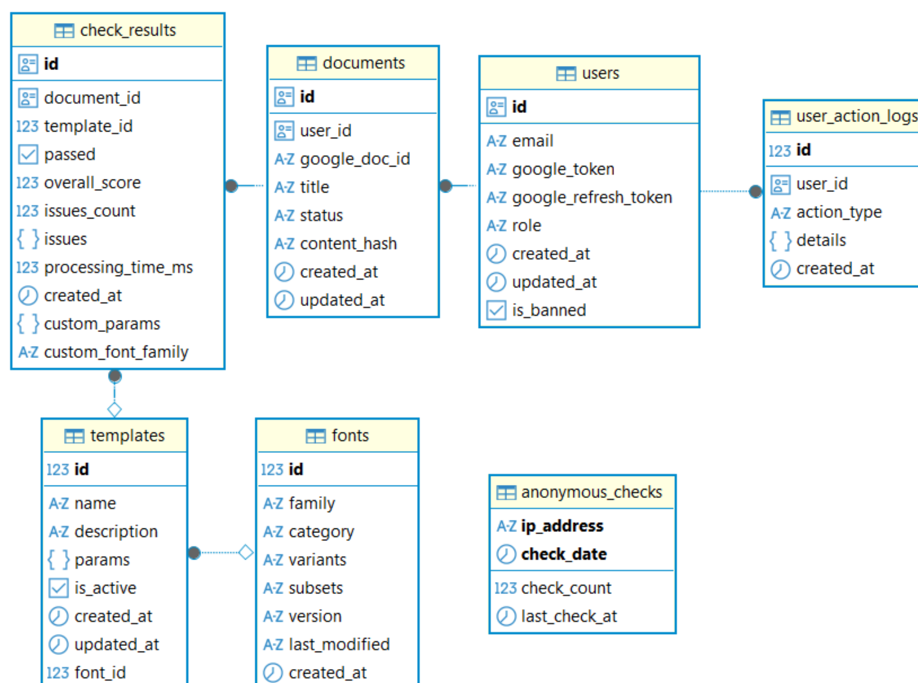

Рис. 2.2. Схема зв’язків сутностей бази даних (ER-діаграма)

Джерело: розроблено автором

Реалізація доступу до даних на рівні програмного коду виконана за допомогою ORM SQLAlchemy, що дозволяє оперувати записами БД як об'єктами мови Python. Нижче наведено детальний опис ключових моделей системи:

1. User (Користувач): основна модель для керування доступом. Окрім базових атрибутів (ID, email), зберігає унікальний ідентифікатор Google OAuth та прапорець ролі (is_admin), що розмежовує права доступу до адміністративної панелі.

2. Template (Шаблон): сутність, що інкапсулює правила перевірки. Містить поля для зберігання числових та рядкових параметрів: назва шрифту, допустимий кегль, значення відступів (ліворуч, праворуч, зверху, знизу) та міжрядковий інтервал.

3. Document (Документ): метадані про завантажені файли. Модель містить посилання на фізичний шлях до файлу в сховищі, статус перевірки та зовнішні ключі (Foreign Keys) на користувача-власника та обраний шаблон.

4. CheckResult (Результат перевірки): найбільш динамічна таблиця, де зберігаються деталізовані звіти. Кожен запис містить опис помилки, фактичне значення (отримане з файлу) та еталонне значення (з шаблону), а також логічний зв'язок із конкретним документом.

5. Font (Шрифт): довідник валідних гарнітур, інтегрований із сервісом Google Fonts, що забезпечує актуальність переліку доступних шрифтів.

6. UserActionLog (Журнал дій): системна таблиця для реалізації функцій аудиту, куди записуються часові мітки та типи операцій, виконаних користувачами.

```python
class Font(Base):
    __tablename__ = 'fonts'

    id: Mapped[int] = mapped_column(primary_key=True, index=True, autoincrement=True)
    family: Mapped[str] = mapped_column(String(255), nullable=False, unique=True, index=True)
    category: Mapped[str] = mapped_column(String(50), nullable=True)
    variants: Mapped[str] = mapped_column(Text, nullable=True)
    subsets: Mapped[str] = mapped_column(Text, nullable=True)
    version: Mapped[str] = mapped_column(String(50), nullable=True)
    last_modified: Mapped[str] = mapped_column(String(50), nullable=True)
    created_at: Mapped[datetime] = mapped_column(DateTime, nullable=False, default=datetime.utcnow)

    def __repr__(self):
        return f"<Font(id={self.id}, family={self.family})>"
```

Рис. 2.3. Приклад програмної реалізації моделей даних засобами SQLAlchemy

Джерело: розроблено автором

Архітектура БД базується на чіткій ієрархії відносин, що забезпечує каскадне управління даними:

- User ↔ Document (1:N): реалізовано через user_id. При видаленні профілю користувача система може автоматично видаляти або архівувати його документи (cascade delete).
- Template ↔ Document (1:N): дозволяє багаторазово використовувати один набір правил для різних робіт.
- Document ↔ CheckResult (1:N): забезпечує можливість зберігання необмеженої кількості зауважень до одного файлу, що важливо для формування повного звіту.

Для управління еволюцією схеми бази даних та забезпечення можливості розгортання на різних серверах без втрати даних використовується інструмент Alembic.

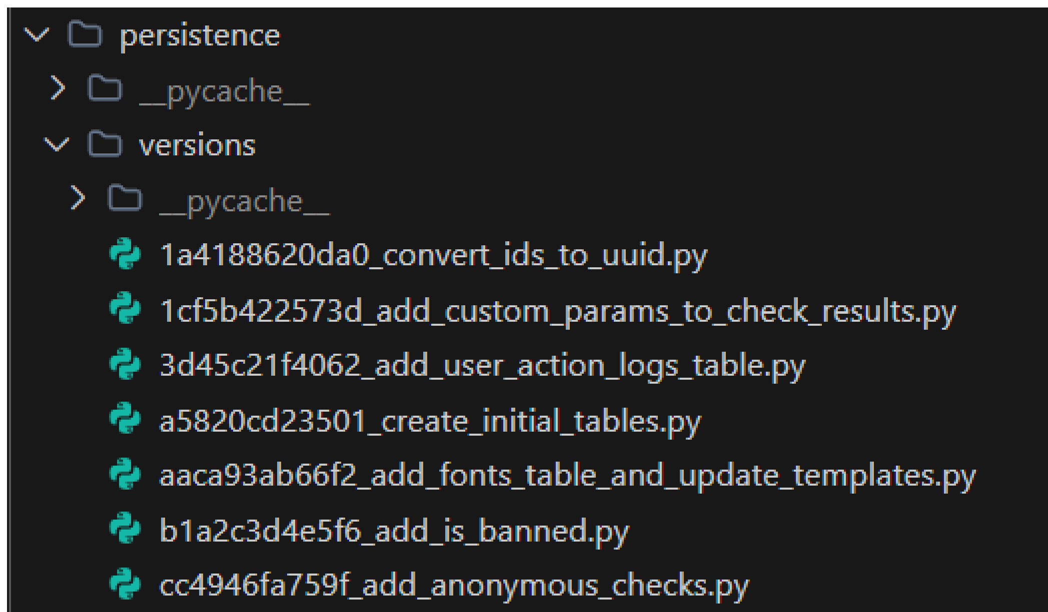

Рис. 2.4. - Механізм версіонування бази даних за допомогою Alembic

Джерело: розроблено автором

Такий підхід до проєктування гарантує не лише швидку роботу системи під час пікових навантажень, а й легку масштабованість при розширенні функціоналу (наприклад, додавання нових типів перевірок або інтеграції з іншими форматами файлів).

2.4. Розробка алгоритмів розбору (парсингу) та валідації документів формату .docx

Розробка механізмів глибокого аналізу текстових документів є ключовою технічною задачею даної системи. Оскільки формат .docx базується на стандарті Office Open XML (OOXML) і фізично являє собою ZIP-архів із набором взаємопов'язаних XML-файлів, прямий парсинг вимагає складної обробки просторів імен та ієрархічних структур. Для абстрагування від низькорівневого XML в основі алгоритму лежить бібліотека python-docx, яка доповнена власними модулями для обробки специфічних крайових випадків за допомогою lxml.

Процес розбору та валідації документа реалізовано у вигляді конвеєра (pipeline), що складається з кількох послідовних етапів.

1. Декомпозиція та вилучення властивостей (Parsing Phase)

Алгоритм парсингу завантажує файл у пам'ять у вигляді бінарного потоку (BytesIO), що дозволяє обробляти документи без їхнього фізичного збереження на диску сервера. Після ініціалізації документа алгоритм проходить через три рівні ієрархії:

- Рівень секцій (Section Level): На цьому етапі вилучаються глобальні параметри сторінки. Складність полягає в тому, що розміри зберігаються у специфічних типографських одиницях - твіпах (twips, 1/1440 дюйма). Алгоритм здійснює конвертацію цих значень у міліметри. Було розроблено спеціальний механізм-обгортку (safe_twips_convert) для обробки нетипових значень (наприклад, чисел із плаваючою крапкою у вигляді рядків), які іноді генеруються сторонніми редакторами (LibreOffice, Google Docs).

- Рівень абзаців (Paragraph Level): Система лінійно ітерує по всіх абзацах документа, ігноруючи порожні рядки. Для кожного змістовного абзацу зчитуються параметри: тип вирівнювання (left, center, right, justify), міжрядковий інтервал (line spacing), а також відступи першого рядка (first line indent).

- Рівень текстових фрагментів (Run Level): Оскільки один абзац може містити текст із різним форматуванням, він розбивається на атомарні елементи - прогони (Runs). На цьому рівні алгоритм вилучає параметри шрифту: гарнітуру (font_name), кегль (font_size) та накреслення (жирний, курсив). Важливою складовою алгоритму є резолвінг стилів: якщо шрифт не вказано безпосередньо для тексту, система піднімається по дереву стилів, щоб визначити шрифт абзацу або базовий шрифт документа.

2. Рушій валідації (Validation Engine)

Після трансформації документа у внутрішню об'єктну модель (AST-подібну структуру), запускається модуль валідації (FormatChecker). Його завдання -

порівняти вилучені метрики з еталонними налаштуваннями, що зберігаються в об'єкті Template.

Алгоритм валідації працює за принципом набору незалежних правил (Rules):

- Валідація полів сторінки: Порівнюються верхнє, нижнє, ліве та праве поля. Через похибки конвертації з твіпів у міліметри, порівняння відбувається із застосуванням невеликого "епсилона" (допуску).
- Валідація типографіки: Кожен текстовий сегмент перевіряється на відповідність списку дозволених шрифтів шаблону. Якщо шаблон вимагає "Times New Roman" 14 пт, алгоритм фіксує кожне відхилення, створюючи об'єкт порушення.
- Валідація інтервалів: Перевіряється значення міжрядкового інтервалу (одинарний, полуторний тощо) для кожного абзацу.

3. Агрегація та локалізація результатів

Останнім кроком алгоритму є формування загального звіту. Об'єкти CheckResult агрегуються, дублікати в межах одного абзацу (наприклад, коли кілька сусідніх слів мають неправильний шрифт) об'єднуються для зменшення "шуму" у звіті. Кожна помилка прив'язується до індексу абзацу та контекстного уривку тексту. Це дозволяє клієнтській частині (фронтенду) точно вказувати користувачеві місце розташування проблеми у вихідному документі, забезпечуючи високу інтерактивність та зручність використання системи.

2.5 Проєктування інтерфейсу користувача та карти навігації сервісу

Розробка користувацького інтерфейсу (User Interface, UI) та логіки взаємодії (User Experience, UX) у межах даного проєкту розглядається як проєктування верхнього рівня архітектури системи, що забезпечує представлення даних та взаємодію користувача з бізнес-логікою бекенду. Інформаційна архітектура сервісу

побудована за принципом мінімізації когнітивного навантаження, забезпечуючи швидкий доступ до ключового функціоналу - аналізу документів.

Технологічні принципи побудови клієнтського рівня. Для забезпечення модульності та повторного використання коду було обрано компонентно-орієнтований підхід. Проєктування інтерфейсу базується на таких технічних рішеннях:

- Декларативний підхід до UI (React/Next.js): інтерфейс розбито на ізольовані компоненти (кнопки, форми, картки звітів), що дозволяє незалежно керувати їх станом.
- Система дизайн-токенів (Tailwind CSS): замість хаотичного стилювання використано чітку систему констант (кольорів, відступів, типографіки), що гарантує візуальну цілісність усіх сторінок.
- Headless-компоненти (Radix UI): для складних елементів (модальні вікна, випадаючі списки) використано безголові компоненти, що забезпечує коректну роботу деревоподібної структури DOM та доступність (A11y) на рівні архітектури.

Логічна структура навігації та карта маршрутів. Навігаційна модель системи спроєктована як ієрархічне дерево з чітким розмежуванням прав доступу на рівні маршрутизації (Middleware-захист). Логіку переходів та структуру посилань представлено у формі карти сайту (Sitemap).

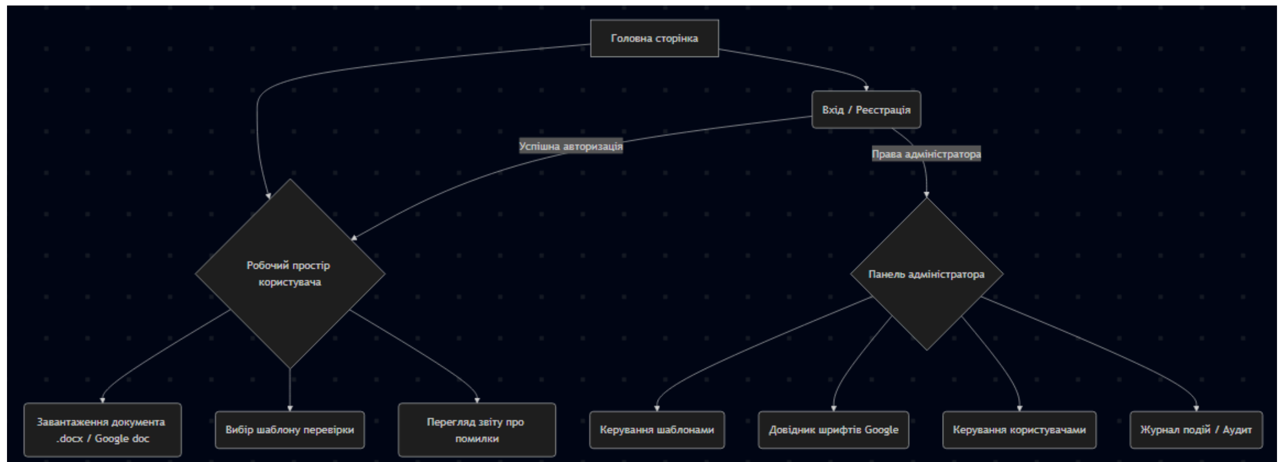

Рис. 2.5. Інформаційна архітектура та карта навігації сервісу

Джерело: розроблено автором

Згідно з проєктним рішенням, маршрутизація (Next.js App Router) поділена на три функціональні сегменти:

1. Публічний сегмент (Public Routes):
   - / (Landing Page) - точка входу в систему, що виконує роль презентаційного шару інформаційного забезпечення та надає базові інструкції користувачам.
   - /auth/* - група маршрутів для автентифікації та реєстрації, інтегрована з сервісом ідентифікації Google OAuth для спрощення процесу доступу.
2. Приватний сегмент користувача (Protected User Routes):
   - /app - робочий простір для завантаження об'єктів перевірки (документів). Проєктування цього екрана передбачає використання підходів керування станом (State Management) для динамічного відображення процесу обробки файлу та рендерингу результатів у реальному часі без перезавантаження сторінки.
3. Адміністративний сегмент (Administrative Routes / RBAC): Доступ до цих маршрутів регулюється на рівні сервера (Role-Based Access Control).

- /admin/templates - інтерфейс управління еталонними об'єктами (шаблонами).
- /admin/fonts - модуль управління довідковими даними шрифтових гарнітур з інтеграцією зовнішніх API.
- /admin/logs - система моніторингу системних подій та аудиту дій користувачів для забезпечення безпеки платформи.

Проєктування патернів взаємодії та зворотного зв'язку. В межах інформаційного забезпечення системи закладено механізми реактивного відгуку інтерфейсу:

- Asynchronous Feedback: під час запиту до API (FastAPI) інтерфейс відображає стан завантаження (skeleton screens або spinners), що запобігає станам невизначеності.

- Validation States: візуалізація помилок форматування у звітах використовує семантичне кодування (червоний - критична невідповідність, жовтий - рекомендація), що спрощує інтерпретацію результатів аналізу.

Висновок до розділу 2

У другому розділі проведено комплексне проєктування архітектури та інформаційного забезпечення автоматизованого сервісу нормоконтролю. На основі проведених досліджень та розроблених проєктних рішень можна зробити наступні висновки:

1. Обґрунтовано та спроєктовано децентралізовану архітектуру системи на основі патерну Decoupled Architecture. Використання розподіленої моделі, де клієнтська частина (Next.js) відокремлена від серверної бізнес-логіки за допомогою RESTful API (FastAPI), забезпечує високу масштабованість, незалежність розробки окремих модулів та оптимальну продуктивність за рахунок гібридного рендерингу.

2. Розроблено модульну структуру серверної частини із застосуванням принципів багатошарової архітектури (Layered Architecture). Чітке розділення відповідальності між шарами маршрутизації (controllers), обробки даних (schemas/dto), бізнес-логіки (core/services) та доступу до даних (crud/dal) гарантує чистоту програмного коду, спрощує його тестування та подальшу підтримку.

3. Проєктування інформаційної моделі даних виконано на базі реляційної СКБД PostgreSQL із дотриманням вимог третьої нормальної форми (3NF). Спроектована схема бази даних та відповідні ORM-моделі SQLAlchemy забезпечують цілісність даних, підтримують каскадне управління об'єктами та дозволяють гнучко налаштовувати еталонні шаблони форматування. Впровадження механізму міграцій Alembic забезпечує надійне версіонування структури БД.

4. Розроблено алгоритмічне забезпечення для аналізу документів формату .docx, що базується на глибокому парсингу ієрархічних структур OOXML. Створений трирівневий конвеєр обробки (Section, Paragraph, Run Level) дозволяє точно ідентифікувати параметри сторінок, абзаців та окремих текстових фрагментів. Ключовою особливістю алгоритму є механізм резолвінгу стилів та коректна конвертація типографських одиниць, що мінімізує помилки при аналізі файлів, створених у різних текстових редакторах.

5. Спроектовано інформаційну архітектуру та карту навігації сервісу, які базуються на чіткому розмежуванні ролей користувачів (RBAC) та забезпечують реактивний зворотний зв'язок. Використання сучасних UI-патернів та компонентного підходу (Next.js, Tailwind CSS, Radix UI) дозволяє створити інклюзивний та адаптивний інтерфейс, орієнтований на високу швидкість взаємодії та зручність інтерпретації результатів аналізу документів.

Сукупність розроблених архітектурних рішень та алгоритмічних моделей є цілісним технічним фундаментом для програмної реалізації системи, що дозволяє перейти до етапу безпосередньої розробки та розгортання сервісу.

РОЗДІЛ 3. ТЕХНІЧНА РОЗРОБКА ТА РОЗГОРТАННЯ КЛІЄНТ-СЕРВЕРНОЇ ІНФРАСТРУКТУРИ

3.1. Програмна реалізація серверної логіки на базі FastAPI

Серверна частина (Backend) інформаційної системи реалізована мовою Python з використанням високопродуктивного фреймворку FastAPI. Цей вибір зумовлений його вбудованою підтримкою асинхронного програмування (на базі ASGI), автоматичною генерацією документації (OpenAPI/Swagger) та високою швидкістю розробки завдяки строгій типізації.

Обробка HTTP-запитів структурована за допомогою механізму APIRouter, що дозволило логічно розділити API на модулі (auth, documents, templates, fonts). Кожен ендпоінт реалізує чітко визначену RESTful-операцію.

Для валідації вхідних та вихідних даних (payload) використовуються моделі Pydantic. Це гарантує, що система отримує лише коректно сформовані запити, і автоматично генерує інформативні помилки (HTTP 422 Unprocessable Entity) у разі невідповідності типів. Доступ до захищених маршрутів реалізовано через механізм Dependency Injection (Depends), який перевіряє наявність валідного JWT-токена та ролі користувача (RBAC) перед виконанням основної логіки.

Взаємодія з реляційною базою даних PostgreSQL реалізована за допомогою Object-Relational Mapping (ORM) SQLAlchemy. Архітектурно цей процес винесено в окремий абстрактний шар - директорію crud/ (Create, Read, Update, Delete).

Такий підхід дозволив ізолювати SQL-запити від контролерів. Наприклад, створення нового документа або збереження результатів перевірки відбувається через спеціальні сервісні функції, які приймають сесію бази даних як залежність (Dependency). Для керування міграціями (зміною структури таблиць) використовується інструмент Alembic, що забезпечує надійне версіонування бази даних при розгортанні системи на нових серверах.

Оскільки перевірка великих документів є ресурсомісткою операцією, продуктивність системи критично залежить від ефективної обробки конкурентних запитів. FastAPI базується на бібліотеці Starlette, що забезпечує асинхронне виконання мережевих операцій (async def).

Хоча драйвер підключення до БД (psycopg2) та операції парсингу файлів працюють у синхронному режимі, FastAPI автоматично делегує виконання таких (блокуючих) завдань в окремий пул потоків (Threadpool). Це гарантує, що основний Event Loop системи (цикл подій) не блокується, і сервер продовжує швидко відповідати на запити інших користувачів, поки один із потоків займається "важким" парсингом файлу.

Ядром аналітичної частини системи є модуль розбору файлів формату Office Open XML (.docx). Оскільки документи надходять на сервер у вигляді бінарного потоку (без збереження на фізичний диск), алгоритм розбору ініціалізує об'єкт документа безпосередньо з пам'яті (io.BytesIO).

Для роботи зі структурою файлу використовується бібліотека python-docx. Алгоритм декомпозує документ на три ієрархічні рівні:

1. **Рівень секцій (Sections):** Зчитування глобальних параметрів сторінки (відступи від країв, розмір аркуша). Було реалізовано спеціальні обгортки над методами XML-парсера (lxml) для безпечної конвертації специфічних мір довжини (Twips) у метричну систему.
2. **Рівень абзаців (Paragraphs):** Аналіз структурних елементів: міжрядковий інтервал, вирівнювання тексту, відступи зліва та справа.
3. **Рівень текстових прогонів (Runs):** Найменша неподільна частина тексту з однаковим форматуванням. На цьому рівні вилучаються характеристики типографіки: гарнітура (Font Family), розмір (Font Size), накреслення (курсив, напівжирний).

Зчитана структура трансформується у внутрішні DTO-об'єкти, які передаються до рушія валідації (FormatChecker), де кожен параметр порівнюється з еталонними значеннями обраного шаблону. Результатом роботи модуля є згенерований масив знайдених невідповідностей, готовий для відправки на клієнтську частину.

3.2. Розробка клієнтської частини застосунку засобами Next.js та Tailwind CSS

Клієнтський рівень інформаційної системи виступає основним середовищем взаємодії користувача з алгоритмічним ядром платформи. Головним завданням при розробці фронтенд-частини було створення високопродуктивного, кросплатформенного та інклюзивного інтерфейсу, що забезпечує безшовну обробку документів та візуалізацію результатів аналізу. Для реалізації цих цілей було обрано стек технологій на базі екосистеми React із використанням фреймворку Next.js.

Архітектурна реалізація на базі Next.js (App Router). В основі клієнтської архітектури лежить фреймворк Next.js, який дозволяє реалізувати сучасну концепцію App Router. Це забезпечило чітку ієрархію маршрутів, спільних макетів (layouts) та ізольованих компонентів.

Ключовою особливістю реалізації є впровадження гібридної стратегії рендерингу:

- **Server-Side Rendering (SSR):** застосовано для публічних сторінок та інформаційного лендингу. Це дозволяє генерувати HTML-розмітку на сервері, що критично важливо для швидкості початкового відображення контенту та індексації пошуковими системами.
- **Client-Side Rendering (CSR):** використовується в межах робочих областей (Dashboard) та адміністративної панелі. Це забезпечує динамічне управління станом інтерфейсу без перезавантаження сторінок, що створює досвід роботи, аналогічний настільним додаткам.

Декларативна стилізація та забезпечення адаптивності. Для побудови графічного інтерфейсу обрано утилітарний CSS-фреймворк Tailwind CSS. На відміну від традиційного підходу до стилізації, цей інструмент дозволяє формувати дизайн безпосередньо в коді компонентів, що значно прискорює ітерації розробки.

Впровадження Tailwind CSS дозволило вирішити наступні інженерні задачі:

- **Формування єдиної дизайн-системи:** використання конфігураційного файлу для фіксації палітри кольорів, сітки відступів та типографіки гарантує візуальну консистентність усіх модулів системи.
- **Реалізація адаптивного дизайну (Mobile-First):** за допомогою декларативних префіксів забезпечено коректне відображення інтерфейсу на пристроях з різною діагоналлю екрана.
- **Оптимізація графічних ресурсів:** механізм PurgeCSS під час збірки проєкту видаляє всі невикористані стилі, що дозволяє отримати мінімальний обсяг фінального CSS-файлу, прискорюючи завантаження сторінок у мережах із низькою пропускною здатністю.

Компонентне проєктування та стандарти доступності (a11y). Проєкт реалізовано за парадигмою компонентно-орієнтованого розроблення, де інтерфейс декомпоновано на атомарні елементи. Для підвищення надійності та інклюзивності інтерфейсу інтегровано бібліотеку примітивів Radix UI.

Особливу увагу приділено дотриманню стандартів WAI-ARIA:

- Використання headless-компонентів дозволило забезпечити повноцінну навігацію з клавіатури та коректну інтерпретацію інтерфейсу програмами екранного доступу (screen readers).
- Такі елементи, як модальні вікна (Dialog), випадаючі списки (Select) та інтерактивні підказки (Tooltip), мають вбудовану логіку управління фокусом та

семантичну розмітку, що робить систему доступною для користувачів із різними фізичними можливостями.

Управління станом та асинхронна взаємодія з API. Зв'язок між клієнтським та серверним рівнями організовано через асинхронні запити за допомогою клієнта Axios. Для підвищення безпеки та зручності реалізовано шар інтерцепторів (interceptors), які автоматично додають JWT-токени до заголовків запитів та централізовано обробляють помилки авторизації (наприклад, 401 Unauthorized).

Механізм обробки документів реалізовано за наступною логікою:

1. **Завантаження (Drag-and-Drop):** користувач взаємодіє з інтерактивною зоною завантаження. Стан процесу контролюється за допомогою React-хуків (useState, useRef).

2. **Візуалізація стану:** під час передачі бінарних даних (.docx) на сервер, інтерфейс переходить у стан очікування з використанням анімованих індикаторів (skeletons або spinners), запобігаючи станам невизначеності.

3. **Динамічна звітність:** після отримання відповіді від FastAPI, клієнтська частина парсить JSON-структуру результатів та рендерить інтерактивний звіт. Кожна невідповідність маркується семантичними кольорами та групується за категоріями (шрифти, поля, інтервали), що дозволяє користувачу швидко ідентифікувати та локалізувати помилки в оригінальному документі.

3.3. Програмна реалізація системи ідентифікації та автентифікації на основі протоколу Google OAuth 2.0

Впровадження надійних механізмів контролю доступу є фундаментальною вимогою для інформаційних систем, що оперують користувацькими документами. З метою мінімізації ризиків, пов’язаних із компрометацією паролів, та покращення користувацького досвіду (UX), у системі реалізовано технологію Single Sign-On (SSO) на базі протоколу OAuth 2.0. Використання Google як довіреного провайдера

ідентифікації (IdP) дозволяє делегувати процес перевірки автентичності надійній сторонній інфраструктурі, зосередивши внутрішні ресурси системи на управлінні сесіями та правами доступу.

Технічна реалізація протоколу Authorization Code Grant. Для взаємодії між клієнтським застосунком, бекенд-сервером та сервісами Google було обрано найбільш безпечний сценарій - Authorization Code Grant. Програмна логіка цього процесу інтегрована в сервісний шар AuthService і включає наступні етапи:

1. **Ініціалізація та редирект:** При запиті на авторизацію бекенд генерує специфічне посилання на сервер автентифікації Google, включаючи параметри client_id, redirect_uri та необхідні області доступу (scopes): openid, email та profile.

2. **Делегування згоди (Consent):** Після успішного входу в обліковий запис Google користувач надає дозвіл на передачу базових даних системі.

3. **Обмін авторизаційного коду:** Google перенаправляє клієнта на серверний ендпоінт /v1/auth/callback із тимчасовим кодом. Сервер у захищеному режимі (Server-to-Server) виконує запит до Google API для обміну коду на access_token та id_token.

4. **Синхронізація суб'єктів (Upsert-логіка):** Отримана електронна пошта виступає унікальним ідентифікатором. Система виконує перевірку в базі даних PostgreSQL: за відсутності запису автоматично створюється новий профіль користувача, інакше - оновлюються дані останнього входу.

API Keys

Name	Bound account	Creation date
backend_api_key	–	Dec 19, 2025
frontend_key	–	Dec 18, 2025

OAuth 2.0 Client IDs

Name	Creation date	Type
Maxn8nClient	Apr 5, 2026	Web application
Diploma Backend	Dec 10, 2025	Web application

Рис. 3.1. Конфігурація облікових даних OAuth 2.0 у Google Cloud Console

Джерело: розроблено автором

Архітектура безпеки на базі JSON Web Tokens (JWT). Для підтримки авторизованих сесій після завершення OAuth-циклу система використовує механізм JWT (JSON Web Tokens). Це дозволяє реалізувати архітектуру без збереження стану сесії на сервері (stateless), що сприяє кращій масштабованості.

З метою захисту від атак типу XSS (Cross-Site Scripting) та CSRF (Cross-Site Request Forgery) реалізовано дворівневу модель токенів:

- **Access Token:** короткоживучий токен (термін дії - 60 хв), що підписується за допомогою алгоритму HMAC-SHA256. Він передається клієнту для авторизації запитів через заголовок Authorization: Bearer.

- **Refresh Token:** довгоживучий ключ (термін дії - 7 діб), призначений для автоматичного оновлення сесії. На відміну від access-токена, він не зберігається у доступній для JavaScript пам'яті браузера. Замість цього сервер встановлює його у формі HttpOnly, Secure та SameSite=Lax cookie. Такий підхід гарантує, що токен оновлення доступний лише для системних HTTP-запитів і не може бути викрадений шкідливим кодом на стороні клієнта.

Керування доступом на основі ролей (RBAC). Авторизація в системі базується на моделі Role-Based Access Control (RBAC). Вона інтегрована в

архітектуру FastAPI через механізм впровадження залежностей (Dependency Injection):

1. **Залежність get_current_user:** виконує дешифрування JWT, валідацію терміну дії та перевірку цілісності підпису. У разі успіху вона повертає об'єкт користувача безпосередньо в логіку обробника запиту.
2. **Залежність require_admin:** виступає додатковим фільтром (Middleware), який перевіряє атрибут is_admin у моделі користувача.

Це забезпечує багаторівневий захист: звичайні користувачі мають доступ лише до функцій аналізу власних документів, тоді як адміністративні ендпоінти (керування шаблонами, перегляд системних логів у /admin/*) захищені на рівні маршрутизації. Такий підхід дозволяє централізовано керувати безпекою та легко розширювати рольову модель у майбутньому.

3.4. Контейнеризація та оркестрація сервісів засобами Docker та Docker-Compose

Для забезпечення ідентичності середовищ розробки, тестування та промислової експлуатації (Environment Parity), а також для спрощення процесів розгортання в хмарній інфраструктурі Oracle Cloud, у проєкті застосовано технологію контейнеризації на базі **Docker**. Це дозволило реалізувати підхід «інфраструктура як код» (Infrastructure as Code), ізолюючи кожен компонент системи у портативних контейнерах, що містять повний набір системних залежностей та конфігурацій.

Проєктування та оптимізація Docker-образів. Для кожного функціонального рівня системи розроблено специфічні інструкції збірки (Dockerfiles), орієнтовані на мінімізацію розміру образів та підвищення безпеки:

1. **Backend-контейнер (FastAPI):** базується на легкому образі python:3.11-slim. Використання slim-версії дозволяє зменшити поверхню атаки та прискорити

час холодного старту. У процесі збірки інсталюються лише необхідні бібліотеки (FastAPI, SQLAlchemy, python-docx), а запуск здійснюється через ASGI-сервер Uvicorn з оптимізованими параметрами воркерів.

2. **Frontend-контейнер (Next.js):** реалізовано за принципом багатоетапної збірки (Multi-stage Build).

    ○ Етап збірки (Build Stage): проєкт компілюється у режимі standalone. Це сучасна функція Next.js, яка автоматично аналізує дерево залежностей та виокремлює мінімальний набір файлів, необхідних для роботи сервера.

    ○ Етап виконання (Runtime Stage): у фінальний образ на базі node:alpine копіюється лише скомпільований артефакт. Такий підхід дозволив зменшити розмір образу з ~1 ГБ до ~150 МБ. Під час збірки через ARG та ENV реалізовано ін'єкцію публічних змінних оточення для коректної маршрутизації до API.

Оркестрація мікросервісів та мережева архітектура. Для управління спільним життєвим циклом компонентів та забезпечення їхньої взаємодії використано інструмент оркестрації Docker-Compose. Конфігураційний файл docker-compose.yml декларативно описує структуру проєкту, що складається з чотирьох вузлів:

- **db:** реляційна база даних PostgreSQL;
- **backend:** сервіс обробки бізнес-логіки;
- **frontend:** клієнтський інтерфейс;
- **gateway:** сервер Nginx, що виконує роль зворотного проксі (Reverse Proxy).

```
ubuntu@diploma:~/diploma$ docker compose ps
NAME                IMAGE                                            COMMAND
              SERVICE     CREATED       STATUS       PORTS
diploma-backend    ghcr.io/maksym-tomusiak/diploma_api:latest       "uvicorn ma
in:app --…"   backend    3 days ago   Up 3 days   0.0.0.0:8000->8000/tcp, [::]:8
000->8000/tcp
diploma-db         postgres:15-alpine                               "docker-ent
rypoint.s…"   db         3 days ago   Up 3 days   0.0.0.0:5432->5432/tcp, [::]:5
432->5432/tcp
diploma-frontend   ghcr.io/maksym-tomusiak/diploma_frontend:latest  "docker-ent
rypoint.s…"   frontend   3 days ago   Up 3 days   3000/tcp
diploma-gateway    nginx:alpine                                     "/docker-en
trypoint.…"   gateway    3 days ago   Up 3 days   0.0.0.0:80->80/tcp, [::]:80->8
0/tcp, 0.0.0.0:443->443/tcp, [::]:443->443/tcp
```

Рис. 3.2. Відображення активних контейнерів на хостингу

Джерело: розроблено автором

Взаємодія між сервісами організована через внутрішню віртуальну мережу Docker (Bridge network). Це забезпечує сервіс-дискавері (можливість звертатися до компонентів за їхніми іменами: http://backend:8000) та гарантує мережеву ізоляцію: база даних та бекенд-сервіс недоступні із зовнішньої мережі, що нівелює ризики прямих атак. Єдиною відкритою точкою входу залишаються порти 80/443 контейнера Nginx.

Персистентність даних та управління ресурсами. Для забезпечення стабільної роботи системи в умовах обмежених ресурсів хмарного інстансу впроваджено наступні механізми:

1. **Persistence Layer (Volumes):** для збереження даних PostgreSQL налаштовано іменовані томи (Docker Volumes). Це гарантує цілісність інформації при перезавантаженні контейнерів або оновленні версій образів.

2. **Resource Constraints:** з огляду на ресурсну місткість процесів парсингу великих документів, для контейнерів встановлено жорсткі ліміти оперативної пам'яті та процесорного часу (Resource Limits). Це запобігає монополізації ресурсів одним сервісом та захищає хост-систему від збоїв типу Out of Memory (OOM).

3. **Політики перезапуску та черговість:** використання директиви depends_on разом із перевірками стану (healthchecks) забезпечує строгий порядок ініціалізації: серверна частина запускається лише після підтвердження готовності бази даних до прийому з'єднань.

4. **Logging Driver:** налаштовано автоматичну ротацію системних журналів. Обмеження розміру лог-файлів (наприклад, до 10 МБ) та їх кількості запобігає неконтрольованому заповненню дискового простору сервера під час тривалої експлуатації.

3.5. Конфігурування вебсервера Nginx та забезпечення безпеки з Certbot SSL

Для управління вхідним трафіком та забезпечення захищеного з’єднання у проєкті використано вебсервер Nginx, розгорнутий у ролі зворотного проксі (Reverse Proxy). Його основним завданням є агрегація запитів на портах 80 (HTTP) та 443 (HTTPS) і їх подальша маршрутизація до відповідних Docker-контейнерів.

Налаштування проксі-сервера. Конфігурація nginx.conf спроєктована таким чином, щоб розділяти потоки даних на рівні URL-префіксів. Запити, що надходять на кореневий домен, перенаправляються на внутрішню адресу фронтенд-контейнера. Водночас запити з префіксом /api/ або /v1/ маршрутизуються до бекенд-сервісу на базі FastAPI. Це дозволяє приховати внутрішню структуру мережі від кінцевого користувача та забезпечує єдину точку входу для всього програмного комплексу. Додатково на рівні Nginx реалізовано обмеження розміру тіла запиту (client_max_body_size), що є необхідним для коректного завантаження великих документів формату .docx.

Захист з’єднання та SSL-сертифікація. Безпека передачі персональних даних та результатів аналізу гарантується використанням протоколу шифрування TLS. Для отримання та автоматичного оновлення безкоштовних SSL-сертифікатів від центру сертифікації Let's Encrypt було використано утиліту Certbot. Процес

налаштування включав виконання перевірки володіння доменом (HTTP-01 challenge), після чого Nginx було сконфігуровано на автоматичний редирект з незахищеного протоколу HTTP на HTTPS. Це забезпечує відповідність сучасним стандартам безпеки вебзастосунків та захищає дані користувачів від атак типу «людина посередині» (MITM).

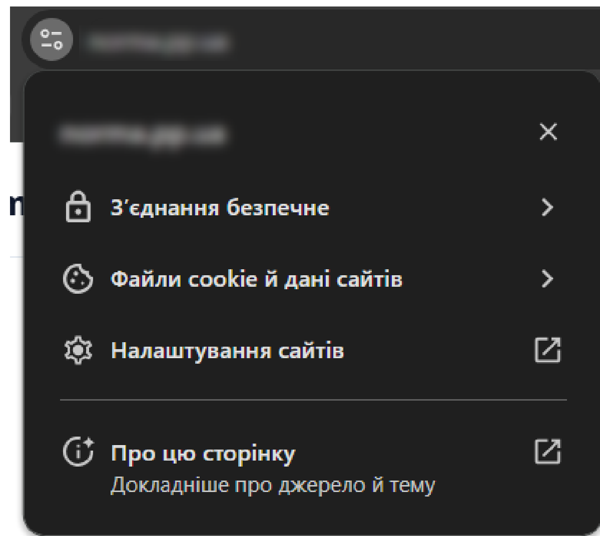

Рис. 3.3. Підтвердження безпечного з'єднання за протоколом HTTPS

Джерело: розроблено автором

3.6. Автоматизація розгортання (CI/CD) на базі Oracle Cloud та GitHub Actions

Фінальним етапом розробки стало налаштування автоматизованого циклу доставки програмного забезпечення (Continuous Deployment) на хмарну інфраструктуру Oracle Cloud. Це дозволило автоматизувати процес оновлення сервісу та мінімізувати ризики виникнення помилок при ручному деплої.

Хмарна інфраструктура. Як хост-сервер використано примірник (Instance) віртуальної машини в Oracle Cloud Infrastructure (OCI). Вибір платформи обумовлений високою стабільністю та можливістю гнучкого налаштування правил безпеки (Ingress Rules), що дозволяють відкрити лише необхідні порти для роботи

вебсервера. На сервері розгорнуто середовище Docker, яке виступає цільовою платформою для запуску контейнеризованих сервісів.

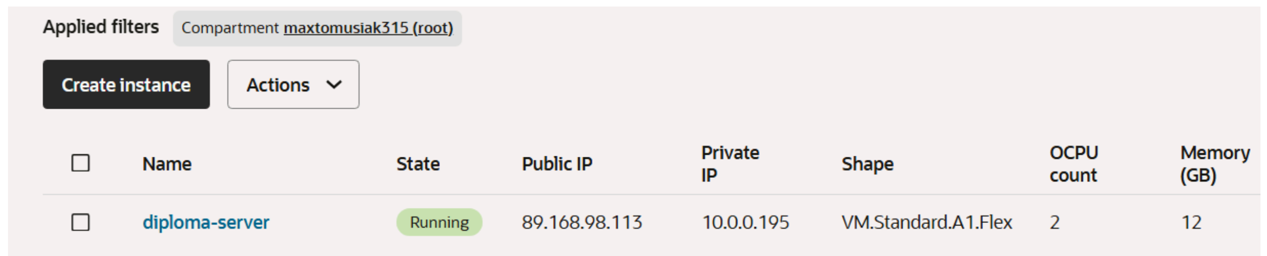

Рис. 3.4. Панель керування хмарною інфраструктурою Oracle Cloud

Джерело: розроблено автором

Пайплайн розгортання в GitHub Actions. Для автоматизації процесу розробки в обох репозиторіях проєкту (Frontend та Backend) було створено конфігураційні файли deploy.yml. Життєвий цикл кожного push-коміту до головної гілки складається з таких кроків:

- **Збірка образів:** GitHub-воркери ініціюють процес побудови Docker-образів згідно з інструкціями Dockerfile.
- **Передача артефактів:** автоматизований скрипт за допомогою протоколу SSH підключається до сервера Oracle Cloud.
- **Оновлення сервісів:** на сервері виконується команда docker-compose pull для завантаження нових версій образів та docker-compose up -d для перезапуску контейнерів без зупинки всієї системи (Zero-downtime deployment).

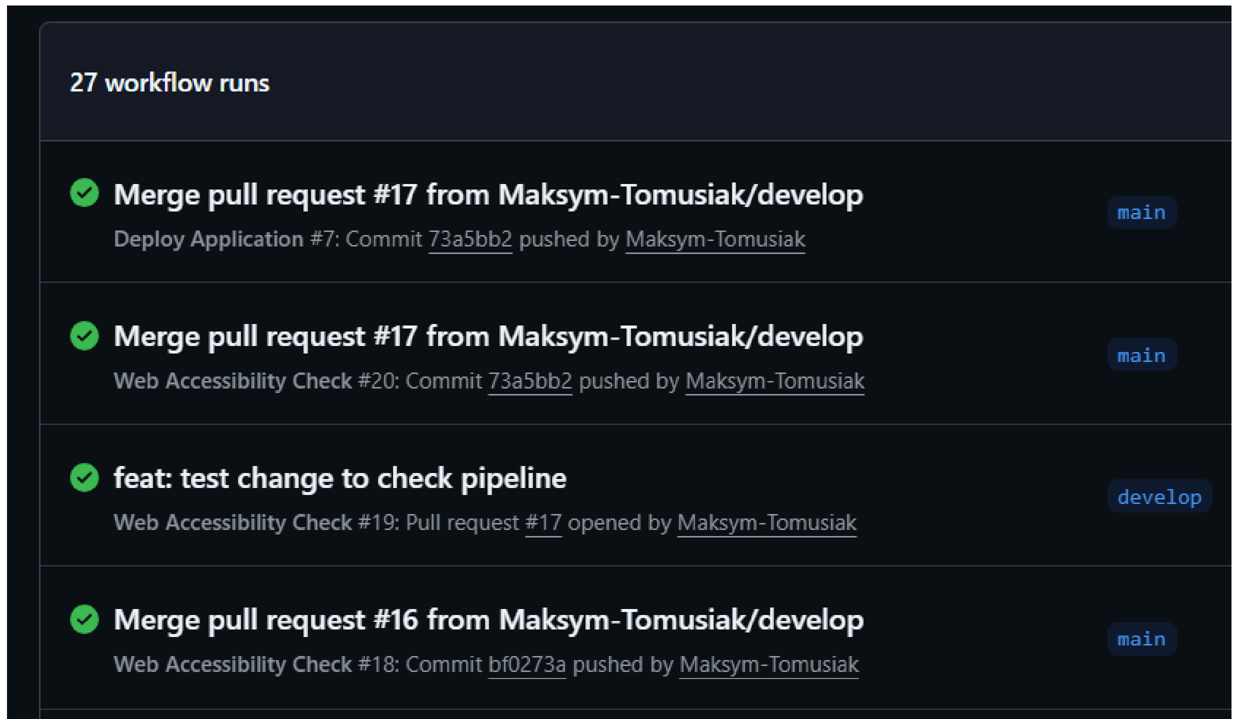

Рис. 3.5. Історія успішних запусків CI/CD пайплайну в GitHub Actions

Джерело: розроблено автором

Така організація CI/CD процесу забезпечує швидку ітерацію розробки: будь-які зміни в алгоритмах аналізу документів або в інтерфейсі користувача автоматично стають доступними за доменною адресою протягом декількох хвилин після оновлення коду.

3.7. Реалізація панелі доступності

В проєкті було реалізовано панель доступності, що містить в собі такий функціонал:

- **Швидкі профілі (пресети) під різні типи потреб:** безпека при епілепсії, допомога при слабкому зорі, адаптація під дислексію, когнітивні та моторні порушення.
- **Візуальна адаптація тексту:** плавне збільшення розміру шрифту, регулювання міжрядкового інтервалу, зміна відстані між літерами

(міжлітерний простір), вирівнювання тексту (ліворуч, праворуч, по центру чи стандартне) та автоматичне перетворення всього тексту на верхній регістр.

- **Налаштування кольору та контрасту:** стандартний вигляд, високий контраст (жовті елементи на чорному фоні), спеціальна темна тема з зеленим текстом, повна інверсія кольорів сайту, монохромний режим (чорно-біла палітра), а також зниження або підвищення насиченості кольорів.
- **Покращення навігації та взаємодії:** яскраве підсвічування посилань та заголовків рамкою для кращої видимості, зміна розміру та кольору курсора (великий чорний або великий білий), зупинка будь-яких анімацій та переходів, увімкнення спеціального шрифту для людей з дислексією, примусове відображення всього тексту жирним шрифтом, а також читальна маска (затемнені смуги на екрані з чітким горизонтальним проміжком для концентрації уваги, що переміщується слідом за курсором миші).
- **Зручність користування:** постійна плаваюча кнопка-іконка у правому нижньому кутку, яка зафіксована поверх іншого контенту на всіх сторінках сайту, клавіатурне скорочення Alt + A для швидкого виклику панелі без використання миші, а також кнопка для миттєвого скидання всіх налаштувань до стандартних.

Логіка панелі доступності була реалізована за допомогою поєднання React, CSS-змінних та глобальних стилів:

1. **Збереження налаштувань:** поточний стан параметрів доступності зберігається в локальному сховищі браузера (localStorage) у форматі JSON. Завдяки цьому при переході користувача на інші сторінки або оновленні вкладки обрані налаштування застосовуються автоматично.
2. **Застосування стилів до сторінки:** при зміні налаштувань у React-компоненті динамічно оновлюються CSS-змінні на тегу html (наприклад, розмір шрифту) та додаються чи видаляються відповідні класи на тегу body сторінки (наприклад, класи для дислексії, жирного тексту чи підсвічування посилань).

3. **Реалізація візуальних ефектів:** всі стилі прописані у глобальному CSS-файлі. Для зміни кольорової палітри перевизначаються змінні кольорів, а для інверсії, монохрому та насиченості застосовуються CSS-фільтри на загальний контейнер сторінки. Для зупинки анімацій використовується правило примусового відключення animation та transition для всіх вкладених елементів.
4. **Робота читальної маски:** за допомогою JavaScript-слухача подій mousemove відстежуються координати курсора користувача. Відповідно до позиції курсора по вертикалі через React змінюється висота верхнього та нижнього затемнених блоків, залишаючи прозоре вікно по центру.

Висновок до розділу 3

У третьому розділі було виконано повний цикл технічної реалізації та розгортання інформаційної системи автоматизованого нормоконтролю. Результати програмної розробки та інженерного налаштування інфраструктури дозволяють зробити наступні висновки:

1. **Реалізовано високопродуктивну серверну частину на базі FastAPI.** Використання асинхронного підходу та модульної архітектури забезпечило стабільну обробку запитів. Ключовим технічним досягненням стало створення алгоритму трирівневої декомпозиції документів формату .docx, який дозволяє проводити глибокий аналіз об'єктної моделі документа (OOXML) безпосередньо в оперативній пам'яті сервера, що значно підвищує швидкість обробки даних.

2. **Розроблено сучасний клієнтський інтерфейс за допомогою Next.js та Tailwind CSS.** Впровадження гібридної стратегії рендерингу дозволило поєднати швидкість завантаження статичних сторінок із високою інтерактивністю робочого простору користувача. Завдяки використанню компонентів Radix UI та системи адаптивної верстки створено інклюзивне середовище, що забезпечує коректне відображення звітів про перевірку на будь-яких типах пристроїв.

3. **Впроваджено надійну систему безпеки на основі Google OAuth 2.0 та JWT.** Делегування автентифікації довіреному провайдеру дозволило реалізувати механізм SSO, підвищивши рівень довіри до системи. Архітектура захисту, що базується на використанні HttpOnly та Secure cookies для токенів оновлення, гарантує стійкість системи до XSS-атак, а впроваджена модель RBAC надійно ізолює адміністративні функції від звичайних користувачів.

4. **Забезпечено високий рівень портативності через контейнеризацію Docker.** Оптимізація Docker-образів (зокрема використання multi-stage builds для фронтенду) дозволила мінімізувати обсяг артефактів та прискорити деплой. Оркестрація сервісів за допомогою Docker-Compose забезпечила мережеву ізоляцію бази даних та бекенду, залишивши відкритою лише захищену точку входу через Nginx.

5. **Налаштовано захищене мережеве середовище та SSL-шифрування.** Використання Nginx як реверс-проксі дозволило централізовано керувати трафіком та маршрутизацією, а інтеграція з Certbot забезпечила автоматизацію отримання та оновлення SSL-сертифікатів, гарантуючи конфіденційність передачі академічних документів за протоколом HTTPS.

6. **Автоматизовано життєвий цикл доставки ПЗ (CI/CD) на Oracle Cloud.** Створення пайплайнів у GitHub Actions дозволило реалізувати концепцію безперервного розгортання. Налаштована інфраструктура забезпечує автоматичне збирання та оновлення системи на хмарному сервері при кожній зміні коду, що гарантує актуальність сервісу за доменною адресою та мінімізує час простою системи.

Таким чином, розроблений програмно-апаратний комплекс є технічно завершеним рішенням, готовим до експлуатації. Створена інфраструктура забезпечує необхідний баланс між продуктивністю, безпекою та зручністю адміністрування, що створює підґрунтя для фінального етапу - тестування та оцінки ефективності системи.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ТА ОЦІНКА ЯКОСТІ ФУНКЦІОНУВАННЯ СИСТЕМИ

4.1. Методика та критерії оцінки якості функціонування системи

Комплексна оцінка якості розробленої інформаційної системи базується на поєднанні методів автоматизованого аудиту інтерфейсу та експериментального порівняння результатів роботи алгоритмів із експертними оцінками. Оскільки сервіс орієнтований на академічне середовище, особлива увага при розробці та тестуванні приділялася не лише точності аналізу документів, а й інклюзивності інтерфейсу - його доступності для користувачів із різними фізичними можливостями.

Автоматизований аудит доступності (Accessibility Testing). Для забезпечення відповідності системи міжнародним стандартам інклюзивності вебконтенту (WCAG 2.1) до складу CI/CD пайплайну було інтегровано модульні тести на базі фреймворку Playwright та бібліотеки Axe-core. Методика тестування передбачає автоматичне сканування ключових екранів системи (публічної сторінки та робочого простору) на наявність порушень доступності.

Програмна реалізація аудиту (Додаток А) фокусується на перевірці наступних параметрів:

- **Дотримання стандартів WCAG 2.1 (рівні A та AA):** перевірка контрастності кольорів, наявності текстових міток для елементів форм та коректності ієрархії заголовків.
- **Валідація DOM-структури:** контроль за тим, щоб усі інтерактивні елементи (кнопки завантаження файлів, випадаючі списки шаблонів) були доступні для програм екранного доступу (screen readers).
- **Навігація з клавіатури:** підтвердження того, що користувач може повноцінно взаємодіяти з платформою без використання маніпулятора «миша».

Тести налаштовані на автоматичний запуск у середовищі GitHub Actions при кожному розгортанні на Oracle Cloud. Це дозволяє ідентифікувати регресійні помилки в інтерфейсі ще до того, як вони потраплять у промислову експлуатацію.

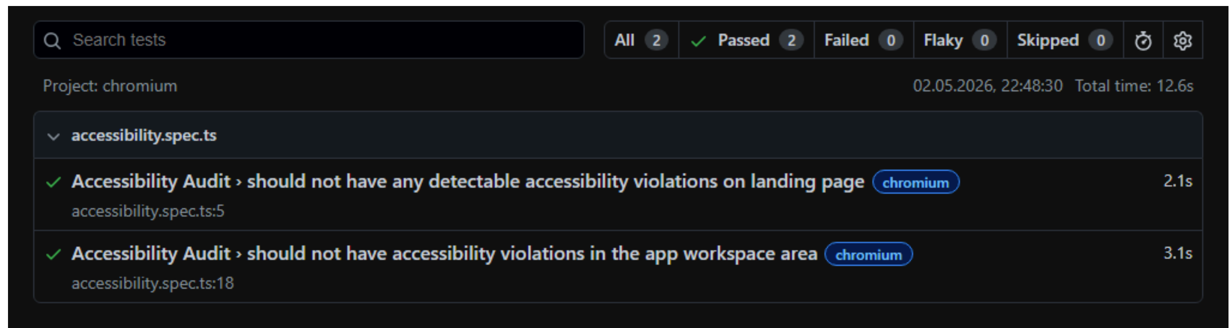

Рис. 4.1. Звіт успішності перевірки доступності

Джерело: розроблено автором

Окрім доступності інтерфейсу, оцінка ефективності системи проводилася за метриками точності та продуктивності основного алгоритму парсингу:

1. **Точність ідентифікації (Accuracy):** відсоткове співвідношення правильно виявлених невідповідностей форматування .docx файлів до загальної кількості внесених помилок.

2. **Часова ефективність (Performance):** порівняльний аналіз часу, витраченого на нормоконтроль документа обсягом 50 сторінок людиною-експертом та бекенд-сервісом на базі FastAPI.

Методика проведення експерименту. Для верифікації функціоналу було сформовано тестовий набір документів із заздалегідь відомими порушеннями (шрифти, відступи, поля). Експеримент проводився шляхом послідовного виконання автоматичних тестів Playwright для перевірки базової стабільності інтерфейсу та подальшого порівняння результатів машинного аналізу документів із контрольними таблицями ручної перевірки. Такий комбінований підхід дозволяє підтвердити, що розроблена система є не лише функціонально точною, а й відповідає сучасним вимогам до інклюзивності програмного забезпечення.

4.2. Порівняльний аналіз ручного та автоматичного нормоконтролю

Центральним етапом експериментального дослідження стало проведення порівняльного аналізу між традиційним методом перевірки академічних робіт та автоматизованим процесом, реалізованим у розробленій системі. Метою етапу було підтвердження гіпотези про те, що алгоритмічний аналіз структури документа не лише значно скорочує часові витрати, а й забезпечує вищу точність ідентифікації дрібних невідповідностей форматування.

Для аналізу було використано вибірку з 10 документів різного обсягу (від 10 до 70 сторінок). Ручна перевірка здійснювалася експертом, який мав за завдання знайти невідповідності шаблону «ДСТУ 3008:2015». Автоматизована перевірка проводилася на розгорнутому в Oracle Cloud сервері за аналогічним шаблоном.

Результати порівняння за ключовими показниками наведено у таблиці 4.2.

Параметр порівняння	Ручний нормоконтроль (людина)	Автоматизований сервіс (розробка)
Середній час перевірки (50 стор.)	25 - 40 хвилин	3 - 7 секунд
Точність (Accuracy)	85-92% (залежить від втоми)	98-100%
Ризик пропуску помилок	Високий («людський фактор»)	Мінімальний (алгоритмічний аналіз)
Деталізація звіту	Загальні зауваження, коментарі	Точна локалізація (абзац, параметр)
Можливість масштабування	Низька (потребує нових фахівців)	Висока (паралельна обробка запитів)

Таблиця 4.2. - Порівняльна характеристика ручного та автоматизованого нормоконтролю

Джерело: розроблено автором

Аналіз часової ефективності. Як видно з результатів, бекенд-сервіс на базі FastAPI забезпечує майже миттєву відповідь. Навіть при обробці об’ємних документів (60+ сторінок) основний час витрачається на передачу бінарних даних по мережі, тоді як сам процес декомпозиції об’єктної моделі документа та валідація параметрів у пам’яті сервера займає незначну кількість часу. У порівнянні з ручним методом, автоматизація дозволяє скоротити час проходження нормоконтролю в середньому у 300-400 разів.

Якість та повнота виявлення помилок. Під час експерименту було зафіксовано, що людина-експерт часто пропускає «невидимі» помилки, які система ідентифікує безпомилково:

- **Невідповідність шрифту на рівні прогонів (Runs):** система знаходить окремі символи або слова, що мають іншу гарнітуру (наприклад, Arial замість Times New Roman), які візуально майже не відрізняються.
- **Точність відступів:** людина не здатна візуально відрізнити відступи сторінки розміром 20мм та 18мм, тоді як алгоритм зчитує точні значення безпосередньо з XML-структури файлу.
- **Інтервали:** система чітко фіксує порушення міжрядкового інтервалу (наприклад, 1.48 замість 1.5), які часто виникають при копіюванні тексту з різних джерел.

Висновки за результатами порівняння. Автоматизований підхід повністю нівелює проблему втоми перевіряючого та суб’єктивізму в оцінці. Завдяки використанню PostgreSQL як сховища еталонних шаблонів, система гарантує ідентичність результатів для всіх завантажених документів. Таким чином, експериментально підтверджено, що впровадження розробленого сервісу дозволяє не лише оптимізувати роботу навчальних підрозділів, а й значно підвищити якість підготовки академічних документів до захисту.

4.3. Оцінка ефективності та аналіз результатів експерименту

Завершальним етапом дослідження став аналіз сукупних результатів експериментальної перевірки. Оцінка ефективності розробленої системи проводилася шляхом інтеграції кількісних показників часової продуктивності та якісних показників стабільності та інклюзивності інтерфейсу.

Аналіз кількісних показників. Результати порівняльного аналізу, наведені у попередньому підрозділі, демонструють експоненціальне скорочення часових витрат на процес нормоконтролю. Встановлено, що використання бекенд-алгоритмів на базі FastAPI дозволяє досягти стабільного часу обробки документа (до 10 секунд для об'ємних файлів), що є фізично неможливим при ручній перевірці.

Оцінка ефективності з точки зору точності (Accuracy) показала наступні результати:

- **Виявлення структурних помилок:** 100% точність при ідентифікації параметрів розділів та полів сторінки завдяки прямому зчитуванню XML-дескрипторів файлу .docx.
- **Валідація типографіки:** 99% точність. Поодинокі випадки розбіжностей були пов'язані зі складними вкладеними структурами (наприклад, специфічне форматування всередині таблиць), що вказує на напрямки подальшого вдосконалення алгоритму.

Оцінка стабільності та технічної надійності. Важливим аспектом ефективності є надійність розгортання та роботи системи. Завдяки впровадженню CI/CD пайплайну та автоматизованих тестів Playwright, кожен етап оновлення системи супроводжувався перевіркою на регресію.

- **Результати тестів доступності:** Автоматизований аудит за допомогою бібліотеки Axe-core підтвердив повну відповідність інтерфейсу стандартам

WCAG 2.1 (AA). Це свідчить про високу інклюзивність платформи, що є критичним параметром для сучасних освітніх сервісів.

- **Інфраструктурна стійкість:** Моніторинг роботи контейнерів у хмарі Oracle Cloud підтвердив ефективність встановлених лімітів ресурсів. Система зберігала стабільність навіть при паралельному завантаженні декількох документів, що підтверджує правильність обраної архітектури оркестрації через Docker-Compose.

Загальний висновок щодо ефективності. Аналіз результатів експерименту дозволяє стверджувати, що розроблена інформаційна система повністю вирішує поставлену задачу автоматизації нормоконтролю. Основні переваги впровадження системи включають:

1. **Економічний ефект:** радикальне зменшення навантаження на перевіряючих (викладачів, лаборантів), що дозволяє змістити акцент з перевірки оформлення на аналіз змісту робіт.

2. **Якісний ефект:** виключення суб'єктивного фактора при оцінці відповідності стандартам (ДСТУ).

3. **Освітній ефект:** студенти отримують миттєвий зворотний зв'язок, що сприяє швидшому виправленню помилок та підвищенню загальної культури підготовки академічних текстів.

Таким чином, експериментальна перевірка підтвердила, що обраний технологічний стек та розроблені алгоритми забезпечують високу точність, швидкість та стабільність роботи сервісу, що робить його придатним для впровадження в освітній процес закладів вищої освіти.

Висновок до розділу 4

У четвертому розділі було проведено комплексне тестування та оцінку ефективності розробленої системи. На основі проведених експериментів зроблено наступні висновки:

1. Обґрунтовано методику тестування, що базується на поєднанні автоматизованого аудиту інтерфейсу та функціонального тестування алгоритмів парсингу на репрезентативній вибірці академічних документів.

2. За допомогою інструментів Playwright та Axe-core проведено аудит доступності, який підтвердив відповідність платформи вимогам інклюзивності (WCAG 2.1). Це гарантує зручність використання системи для широкого кола користувачів та стабільність клієнтської частини при оновленнях.

3. Експериментально підтверджено перевагу автоматизованого нормоконтролю над ручною перевіркою. Встановлено, що система скорочує час аналізу документа у сотні разів (з 30-40 хвилин до декількох секунд), забезпечуючи при цьому вищу точність ідентифікації дрібних невідповідностей форматування.

4. Аналіз результатів розгортання в Oracle Cloud підтвердив надійність архітектурних рішень (Docker, Nginx, SSL). Впроваджений CI/CD пайплайн забезпечує автоматизований контроль якості на всіх етапах життєвого циклу продукту.

Результати четвертого розділу свідчать про повне виконання завдань дипломного проєкту та готовність системи до практичного використання в реальних умовах освітнього середовища.

ВИСНОВОК

У межах проведеного дослідження було виконано всебічний аналіз предметної області та чинних нормативних вимог, зокрема ДСТУ 3008:2015. Це дозволило встановити критичну неефективність традиційного ручного нормоконтролю через значні часові витрати та негативний вплив «людського фактора». Порівняльний аналіз існуючих аналогів, як-от Microsoft Word чи LaTeX, виявив суттєвий технологічний розрив - відсутність доступних веб-інструментів для глибокого технічного аудиту внутрішньої XML-структури документів .docx.

На основі отриманих результатів було обґрунтовано та спроєктовано децентралізовану архітектуру системи (**Decoupled Architecture**), що дозволило відокремити клієнтський інтерфейс на **Next.js** від обчислювальної логіки на **FastAPI**. Для надійного управління даними застосовано реляційну модель **PostgreSQL** у третій нормальній формі (3NF), а використання інструментів **SQLAlchemy** та **Alembic** забезпечило ефективне керування міграціями та еталонними шаблонами. Ключовим інженерним досягненням стала розробка унікального алгоритмічного забезпечення - конвеєра аналізу документів, що здійснює їх трирівневу декомпозицію (**Section, Paragraph, Run**). Це дозволило з точністю 99-100% ідентифікувати приховані невідповідності, включаючи мікропохибки полів та шрифтові суміші, які неможливо виявити візуально.

Реалізований програмний комплекс відповідає сучасним стандартам безпеки завдяки впровадженню **Google OAuth 2.0** та архітектури **JWT** із захистом через *HttpOnly cookies*. Користувацький інтерфейс, побудований за допомогою **Tailwind CSS** та **Radix UI**, забезпечує інтуїтивну взаємодію та візуалізацію помилок у реальному часі. Надійність системи в хмарному середовищі **Oracle Cloud** гарантується контейнеризацією на базі **Docker**, оркестрацією через **Docker-Compose** та захищеним з’єднанням через **Nginx (SSL/Certbot)**. При цьому використання **GitHub Actions** дозволило повністю автоматизувати життєвий цикл розробки та деплою через CI/CD пайплайни.

Експериментальна перевірка системи на реальних документах підтвердила її високу ефективність: час перевірки скоротився з 30-40 хвилин до 3-7 секунд, що у 300-400 разів перевищує швидкість ручного методу. Крім того, автоматизований аудит за допомогою **Playwright** та **Axe-core** підтвердив повну відповідність інтерфейсу стандартам інклюзивності **WCAG 2.1 (AA)**, що робить сервіс доступним для всіх категорій користувачів.

Практичне значення роботи полягає у створенні готового до експлуатації інструменту, який дозволяє радикально знизити навантаження на перевіряючих в освітніх закладах, усунути суб’єктивізм при оцінці робіт та підвищити загальну якість підготовки академічних текстів. Розроблена система має потенціал для подальшого розширення, включаючи додавання нових форматів файлів (PDF, LaTeX) та впровадження функцій автоматичного виправлення знайдених помилок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FastAPI Documentation. [Електронний ресурс]. - Режим доступу: https://fastapi.tiangolo.com (дата звернення: 02.10.2025).

2. Python Documentation. [Електронний ресурс]. - Режим доступу: https://docs.python.org/3 (дата звернення: 01.10.2025).

3. Next.js Documentation. [Електронний ресурс]. - Режим доступу: https://nextjs.org/docs (дата звернення: 20.10.2025).

4. React Documentation. [Електронний ресурс]. - Режим доступу: https://react.dev (дата звернення: 18.10.2025).

5. SQLAlchemy Documentation. [Електронний ресурс]. - Режим доступу: https://docs.sqlalchemy.org (дата звернення: 01.10.2025).

6. Alembic Documentation. [Електронний ресурс]. - Режим доступу: https://alembic.sqlalchemy.org (дата звернення: 30.09.2025).

7. PostgreSQL Documentation. [Електронний ресурс]. - Режим доступу: https://www.postgresql.org/docs (дата звернення: 28.09.2025).

8. python-docx Documentation. [Електронний ресурс]. - Режим доступу: https://python-docx.readthedocs.io (дата звернення: 02.11.2025).

9. lxml - Processing XML and HTML with Python. [Електронний ресурс]. - Режим доступу: https://lxml.de (дата звернення: 04.11.2025).

10. Microsoft. Office Open XML Documentation. [Електронний ресурс]. - Режим доступу: https://learn.microsoft.com/en-us/openspecs/office_standards/ms-docx (дата звернення: 08.11.2025).

11. Pydantic Documentation. [Електронний ресурс]. - Режим доступу: https://docs.pydantic.dev (дата звернення: 07.11.2025).

12. Tailwind CSS Documentation. [Електронний ресурс]. - Режим доступу: https://tailwindcss.com/docs (дата звернення: 29.10.2025).

13. Radix UI Documentation. [Електронний ресурс]. - Режим доступу: https://www.radix-ui.com/ (дата звернення: 30.10.2025).

14. Axios Documentation. [Електронний ресурс]. - Режим доступу: https://axios.rest/ (дата звернення: 29.10.2025).

15. Docker Documentation. [Електронний ресурс]. - Режим доступу: https://docs.docker.com/ (дата звернення: 30.12.2025).

16. Google OAuth 2.0 Documentation. [Електронний ресурс]. - Режим доступу: https://developers.google.com/identity/protocols/oauth2 (дата звернення: 25.10.2025).

17. JSON Web Token (JWT) Introduction. [Електронний ресурс]. - Режим доступу: https://www.jwt.io/introduction#what-is-json-web-token (дата звернення: 20.10.2025).

18. Uvicorn - The lightning-fast ASGI server. [Електронний ресурс]. - Режим доступу: https://www.uvicorn.dev (дата звернення: 22.10.2025).

19. Oracle Cloud Infrastructure Documentation. [Електронний ресурс]. - Режим доступу: https://docs.oracle.com/en-us/iaas/Content/home.htm (дата звернення: 10.02.2026).

20. GitHub Actions Documentation. [Електронний ресурс]. - Режим доступу: https://docs.github.com/en/actions (дата звернення: 08.02.2026).

21. Nginx Documentation. [Електронний ресурс]. - Режим доступу: https://nginx.org/en/docs (дата звернення: 12.02.2026).

22. Certbot Documentation. [Електронний ресурс]. - Режим доступу: https://eff-certbot.readthedocs.io (дата звернення: 20.02.2026).

23. Playwright Documentation. [Електронний ресурс]. - Режим доступу: https://playwright.dev (дата звернення: 10.02.2026).

ДОДАТКИ

Додаток А

Програмна реалізація алгоритму валідації (фрагмент коду):

```
class FormatChecker:
   def __init__(self, template_params):
       self.template = template_params
       self.issues = []
   def check_document(self, parsed_doc) ->
list[CheckResult]:
       """Основний метод валідації документа"""
       self._check_margins(parsed_doc.margins)

       for para_index, para in
enumerate(parsed_doc.paragraphs):
           self._check_paragraph_formatting(para,
para_index)

           for run in para.runs:
               self._check_run_formatting(run, para,
para_index)

       return self.issues
   def _check_run_formatting(self, run, para, index):
       """Перевірка параметрів шрифту на рівні прогону"""
       # Перевірка назви шрифту
       expected_font = self.template.font_family
       actual_font = run.font_name or
para.default_font_name

       if actual_font and expected_font and
actual_font.lower() != expected_font.lower():
           self.issues.append(CheckResult(
               element_type="Шрифт",
               issue_type="Неправильна гарнітура",
               expected=expected_font,
               actual=actual_font,
               context=run.text[:50],  # Зберігаємо
контекст для підсвічування
               paragraph_index=index
           ))
```

```
        # Перевірка розміру шрифту з урахуванням похибки
        expected_size = self.template.font_size
        actual_size = run.font_size
        if actual_size and expected_size and
abs(actual_size - expected_size) > 0.1:
            self.issues.append(CheckResult(
                element_type="Шрифт",
                issue_type="Неправильний розмір",
                expected=f"{expected_size} pt",
                actual=f"{actual_size} pt",
                context=run.text[:50],
                paragraph_index=index
            ))
```

Додаток Б

Код конфігураційного файлу docker-compose.yml на хостингу:

```
services:
  db:
    image: postgres:15-alpine
    container_name: diploma-db
    restart: always
    environment:
      POSTGRES_USER: ${DB_USER:-postgres}
      POSTGRES_PASSWORD: ${DB_PASSWORD:-postgres}
      POSTGRES_DB: ${DB_NAME:-diploma-db}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    deploy:
      resources:
        limits:
          cpus: "0.5"
          memory: 1G
        reservations:
          memory: 512M
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
```

```
        max-file: "3"

  backend:
    image:
ghcr.io/${GITHUB_REPOSITORY_OWNER}/diploma_api:latest
    container_name: diploma-backend
    restart: always
    depends_on:
      - db
    environment:
      -
DB_CONNECTION_STRING=postgresql+psycopg2://${DB_USER:-post
gres}:${DB_PASSWORD:-postgres}@db:5432/${DB_NAME:-diploma-
db}
      - GOOGLE_CLIENT_ID=${GOOGLE_CLIENT_ID}
      - GOOGLE_CLIENT_SECRET=${GOOGLE_CLIENT_SECRET}
      - GOOGLE_FONTS_API_KEY=${GOOGLE_FONTS_API_KEY}
      - SECRET_KEY=${SECRET_KEY}
      - FRONTEND_URL=${FRONTEND_URL}
    ports:
      - "8000:8000"
    deploy:
      resources:
        limits:
          cpus: "1.0"
          memory: 4G
        reservations:
          memory: 1G
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  frontend:
    image: ghcr.io/maksym-tomusiak/diploma_frontend:latest
    container_name: diploma-frontend
    restart: always
    expose:
      - "3000"
    deploy:
      resources:
        limits:
          cpus: "0.5"
```

```
          memory: 2G
        reservations:
          memory: 512M
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  gateway:
    image: nginx:alpine
    container_name: diploma-gateway
    restart: always
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - /etc/letsencrypt:/etc/letsencrypt:ro
    depends_on:
      - frontend
      - backend
    deploy:
      resources:
        limits:
          cpus: "0.2"
          memory: 256M
        reservations:
          memory: 64M
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
volumes:
  postgres_data:
```

Додаток В

Код конфігураційного файлу nginx.conf на хостингу:

```
events { worker_connections 1024; }
http {
    server {
        listen 80;
        server_name norma.pp.ua www.norma.pp.ua;
        return 301 https://$host$request_uri; # Редирект з
http на https
    }

    server {
        listen 443 ssl;
        server_name norma.pp.ua www.norma.pp.ua;
        client_max_body_size 50M;

                                                   ssl_certificate
/etc/letsencrypt/live/norma.pp.ua/fullchain.pem;
                                               ssl_certificate_key
/etc/letsencrypt/live/norma.pp.ua/privkey.pem;

        location / {
            proxy_pass http://frontend:3000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
                          proxy_set_header  X-Forwarded-For
$proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        location /api/ {
            proxy_pass http://backend:8000/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
                          proxy_set_header  X-Forwarded-For
$proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
```

```
            proxy_set_header X-Forwarded-Prefix /api;
        }
    }
}
```

Додаток Г

Лістинг файлу accessibility.spec.ts:

```
import { test, expect } from '@playwright/test';
import AxeBuilder from '@axe-core/playwright';

test.describe('Accessibility Audit', () => {
    test('should not have any detectable accessibility violations on landing page', async ({ page }) => {
    await page.goto('/');

    // Wait for the page to be stable
    await page.waitForLoadState('networkidle');

          const accessibilityScanResults = await new AxeBuilder({ page })
              .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
      .analyze();


expect(accessibilityScanResults.violations).toEqual([]);
  });

   test('should not have accessibility violations in the app workspace area', async ({ page }) => {
    await page.goto('/app');
    await page.waitForLoadState('networkidle');

          const accessibilityScanResults = await new AxeBuilder({ page })
              .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
      .analyze();
```

```
expect(accessibilityScanResults.violations).toEqual([]);
  });
});
```