Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Впровадження векторного пошуку у системі документообігу

Виконав(ла): студент(ка) 6 курсу, групи СПм-61
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

		Ямко В. О.
	(підпис)	(прізвище та ініціали)
Керівник		Петрик М. Р.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		Стоянов Ю. М.
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Петрик М. Р.
	(підпис)	(прізвище та ініціали)
Рецензент		
	(підпис)	(прізвище та ініціали)

Тернопіль
2024

АНОТАЦІЯ

Кваліфікаційна робота магістра на тему «Впровадження векторного пошуку у системі документообігу» написана Ямко Владиславом Олексійовичем, студентом Тернопільського національного технічного університету імені Івана Пулюя, Факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СПм-61.

Відомості про обсяг: сторінок – 62, рисунків – 11, таблиць – 0, частин – 4, додатків – 3, посилань – 18, формул – 0.

Метою наукової роботи є створення програмного комплексу для автоматизованого управління документами з використанням векторного пошуку, чанкування та ембедінгу. Рішення дозволяє оптимізувати процеси обробки та зберігання великих обсягів даних, автоматизуючи керування документами, зокрема їх класифікацію, пошук та фільтрацію.

Програма використовує інноваційні підходи до обробки текстових даних, такі як векторизація, що забезпечує ефективний пошук за змістом. Чанкування, в свою чергу, дає змогу розділяти великі документи на менші частини, що полегшує обробку та покращує швидкість пошукових запитів. Водночас ембедінг допомагає відображати зміст документів у вигляді векторів, що дозволяє забезпечити точні результати пошуку за допомогою математичних моделей.

Апаратно-програмний комплекс працює з такими даними, як інформація про клієнтів, документи та мітки. Використовуються сучасні технології, такі як NestJS для серверної логіки, TypeORM для роботи з базою даних та MongoDB для зберігання даних.

Цей комплекс дозволяє не лише ефективно зберігати та управляти даними, але й здійснювати швидкий та точний пошук по документах, що містять важливу інформацію для користувачів. Рішення також дозволяє автоматично оновлювати та організовувати дані, забезпечуючи зручний доступ до необхідної інформації в будь-який час.

ABSTRACT

The master`s thesis titled "Implementation of Vector Search in Document Management System" was written by Vladyslav Yamko, a student of the Ternopil Ivan Puluj National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, group SPm-61.

Details on volume: pages – 62, figures – 11, tables – 0, sections – 4, appendices – 3, references – 18, formulas – 0.

The aim of this scientific work is to create a software complex for automated document management using vector search, chunking, and embedding. The solution optimizes data processing and storage processes by automating document management, including classification, search, and filtering.

The program utilizes innovative approaches to text data processing, such as vectorization, which enables efficient content-based search. Chunking, in turn, allows for splitting large documents into smaller parts, facilitating processing and improving the speed of search queries. Embedding, at the same time, helps to represent the content of documents as vectors, ensuring accurate search results through mathematical models.

The hardware-software complex works with data such as client information, documents, and labels. Modern technologies are used, such as NestJS for server-side logic, TypeORM for database operations, and MongoDB for data storage.

This complex not only allows for efficient data storage and management but also provides fast and precise search across documents containing important information for users. The solution also enables automatic updating and organization of data, ensuring convenient access to necessary information at any time.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ООП (Object-Oriented Programming) – Об’єктно-орієнтоване програмування. Парадигма програмування, яка базується на поняттях об’єктів, класів, інкапсуляції, успадкування та поліморфізму.

UML (Unified Modeling Language) – Уніфікована мова моделювання. Стандарт мови візуального моделювання програмних систем.

API (Application Programming Interface) – Програмний інтерфейс додатка. Набір правил і протоколів для взаємодії між різними програмними компонентами.

JSON (JavaScript Object Notation) – Текстовий формат обміну даними, що базується на синтаксисі JavaScript, але використовується незалежно від нього.

JWT (JSON Web Token) – Формат токена доступу, що містить зашифровану інформацію про користувача чи сесію, використовується для аутентифікації та авторизації.

Node.js – Виконавче середовище для JavaScript поза браузером, базується на рушії V8. Дозволяє створювати серверні застосунки на JavaScript.

Nest.js – Прогресивний фреймворк для Node.js, побудований на TypeScript, що забезпечує модульну архітектуру та використання ООП-принципів у серверній розробці.

React – Бібліотека JavaScript для побудови інтерфейсів. Використовується для створення динамічних клієнтських застосунків.

Next.js – Фреймворк на базі React, що полегшує серверний рендеринг та статичну генерацію сторінок, підвищуючи продуктивність та SEO.

MongoDB – Документо-орієнтована NoSQL база даних. Зберігає дані у BSON-форматі, надаючи гнучкість у зберіганні та масштабуванні.

TypeScript – Надмножина JavaScript з системою статичної типізації, що підвищує надійність та зрозумілість коду.

OCR (Optical Character Recognition) – Оптичне розпізнавання символів. Технологія перетворення зображень тексту в машинозчитний формат.

REST (Representational State Transfer) – Стиль архітектури програмних інтерфейсів, який використовує стандартні HTTP-методи для взаємодії з ресурсами.

HTTPS (HyperText Transfer Protocol Secure) – Безпечний протокол передачі гіпертексту. Забезпечує шифрування даних і автентифікацію сторін.

CI/CD (Continuous Integration / Continuous Deployment) – Безперервна інтеграція та безперервне розгортання. Автоматизовані процеси тестування, збірки та доставки коду.

PDF (Portable Document Format) – Портативний формат документів для текстово-графічних матеріалів.

Code Style Tools (ESLint, Prettier) – Інструменти для підтримання єдиного стилю та якості коду.

ШІ (Штучний інтелект / AI) – Галузь інформатики, що займається моделюванням розумної поведінки машин та алгоритмів.

Чанкування (Chunking) – Процес розбиття великого тексту на менші, логічно зв’язані фрагменти (чанки). Покращує ефективність аналізу та пошуку.

Ембедінг (Embedding) – Векторне представлення тексту (слова, речення, документа) у числовому багатовимірному просторі. Використовується для семантичного пошуку та аналізу схожості.

Векторний пошук (Vector Search) – Метод пошуку інформації за векторними репрезентаціями даних. Замінює пошук за ключовими словами пошуком за семантичною близькістю між ембедінгами.

Модель машинного навчання / Модель ШІ – Комп’ютерна модель, навчена на великих наборах даних для виконання задач класифікації, пошуку, генерації тексту чи інших інтелектуальних функцій.

Кешування ембедінгів – Збереження раніше отриманих векторних репрезентацій для повторного використання, щоб зменшити кількість звернень до зовнішніх сервісів та пришвидшити роботу системи.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасні інформаційні технології відіграють надзвичайно важливу роль у підвищенні ефективності та точності процесів управління великими обсягами даних, зокрема документів. Одним із перспективних підходів до покращення організації та пошуку інформації є застосування методів векторного пошуку, чанкування (розподілу великих текстів на менші фрагменти) та ембедінгу (векторизації змісту). Такий підхід дозволяє не тільки оптимізувати процеси зберігання та класифікації документів, але й забезпечити значно точніший, швидший та змістовно орієнтований пошук. Це особливо актуально в умовах зростання обсягів даних у різних сферах — від корпоративних архівів до наукових, медичних та освітніх матеріалів. Впровадження таких технологій сприяє оперативному доступу до потрібної інформації, що прискорює прийняття рішень та підвищує загальну продуктивність.

Мета роботи полягає у створенні апаратно-програмного комплексу для автоматизованого управління документами з використанням векторного пошуку, чанкування та ембедінгу. Такий комплекс повинен забезпечити ефективне зберігання, обробку та пошук великих обсягів текстових даних за змістом, спростити класифікацію та фільтрацію документів, а також надати можливість швидкого доступу до релевантної інформації.

Завдання роботи передбачають:

Розробити систему векторного пошуку, що дозволить класифікувати та знаходити необхідні документи на основі змісту, а не лише за ключовими словами.

Реалізувати механізм чанкування для розділення великих документів на менші фрагменти, що покращить ефективність обробки та пришвидшить пошук.

Застосувати технології ембедінгу для представлення тексту документів у вигляді векторів, що дозволить здійснювати точний семантичний пошук.

Створити серверну частину системи на основі NestJS та MongoDB, забезпечивши ефективне зберігання, обробку та доступ до даних.

Розробити інтерактивний інтерфейс користувача для керування документами, здійснення пошуку та отримання результатів обробки.

Об’єктом дослідження виступає процес проектування інформаційної системи для автоматизованого управління документами з використанням сучасних методів аналізу текстових даних. Предметом дослідження є створення комплексної системи, яка реалізує векторизацію тексту, класифікацію, семантичний пошук, фільтрацію документів та інтерактивний інтерфейс для користувача, що спрощує процес доступу до релевантної інформації.

Отже, дана робота спрямована на поєднання новітніх методів обробки текстових даних, таких як векторний пошук, чанкування і ембедінг, із сучасними технологіями розробки серверних і клієнтських застосунків. Це дозволить створити ефективну, гнучку та зручну у використанні систему управління документами.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОСЛІДЖЕННЯ ТА ТЕХНІЧНЕ ЗАВДАННЯ

1.1. Аналіз предметної області

У сучасному світі, що характеризується стрімким зростанням інформаційних потоків та постійним ускладненням структури даних, потреба в ефективних системах управління документами стає дедалі актуальнішою. Організації різного профілю — державні установи, комерційні компанії, науково-дослідні центри, освітні заклади — щодня опрацьовують тисячі, а іноді й мільйони документів. Ці документи можуть бути різного формату: текстові файли, PDF, зображення, аудіо- та відеоматеріали, структуровані та неструктуровані дані. Усі вони потребують належної організації, зберігання, швидкого пошуку та гарантійної доступності в потрібний момент.

Проблема полягає в тому, що традиційні методи збереження та обробки документів, засновані на простому пошуку за ключовими словами або каталогізації за жорстко визначеними категоріями, вже не задовольняють сучасні вимоги. По-перше, суто ключове слово не завжди дає можливість знайти потрібну інформацію, особливо коли документ може містити синоніми, перефразування чи складні контекстуальні зв’язки. По-друге, неминучим є питання масштабування: чим більше документів накопичується, тим складніше й довше стає процес їхнього опрацювання. По-третє, зростання обсягів даних призводить до того, що користувач, який намагається знайти певний документ чи фрагмент інформації, змушений витрачати дедалі більше часу на пошук і фільтрування результатів.

Саме тому в останні роки все більшої уваги набувають підходи до обробки документів, орієнтовані на семантику та зміст. Розвиток машинного навчання, обробки природної мови (NLP) та штучного інтелекту (ШІ) відкрив двері для нових способів представлення текстової інформації [1]. Однією з ключових технологій стало перетворення тексту у векторні представлення — так звані ембедінги.

Ембедінги дають змогу відобразити лексичне та семантичне значення слів у багатовимірному просторі, що дозволяє алгоритмам визначати схожість між різними фрагментами тексту не за формальними ознаками, а за змістом.

Застосування ембедінгів у системах управління документами суттєво покращує ефективність пошуку. Якщо раніше пошуковий механізм намагався знайти документи за наявністю певного слова чи фрази, то тепер він може оцінювати семантичну близькість між запитом користувача та масивом доступних текстів. Це означає, що навіть якщо користувач використовує інше формулювання, синоніми або описує проблему складнішою мовою, система здатна запропонувати релевантні результати.

Ще один важливий елемент сучасних систем — це чанкування документів. Враховуючи, що великі документи важко індексувати та опрацьовувати цілісно, їх розбивають на логічні фрагменти — чанки (англ. chunk). Кожен чанк репрезентує певний завершений фрагмент інформації. Таке розбиття дозволяє працювати з меншими за обсягом одиницями, що спрощує обчислювальні завдання. Користувач не обов’язково має переглядати весь документ; досить звернутися лише до релевантного чанку, тим самим суттєво економлячи час.

Чанкування також покращує роботу алгоритмів машинного навчання, оскільки менші текстові фрагменти легше обробляти. Це сприяє точнішому формуванню ембедінгів та кращій релевантності результатів пошуку. По суті, чанкування створює умови для багатостороннього аналізу: можна виділяти не лише конкретні слова, але й цілі смислові одиниці, фрагменти тексту, контекст яких є більш зрозумілим для моделі.

Векторний пошук — ще один стовп сучасних систем управління документами. Він полягає у тому, що кожний чанк тексту, кожен документ або кожна сутність у документі зберігається не у вигляді звичайного рядка символів, а як вектор чисел. Коли користувач формує запит (також перетворений у вектор), система може обчислити косинусну подібність або іншу метрику між цим вектором-запитом та векторами документів. Результатом є список документів,

впорядкований за схожістю змісту. Таким чином, пошук за змістом стає більш гнучким і точним.

Інтеграція таких підходів до предметної області управління документами передбачає розв’язання низки практичних задач. По-перше, має бути забезпечена можливість роботи з різноманітними форматами документів. У реальних умовах це не лише текстові файли, а й скановані PDF-документи, де текст може бути зображенням. У такому випадку слід застосовувати технології оптичного розпізнавання символів (OCR), щоб перетворити графічний текст у машинозчитуваний формат. По-друге, документи можуть містити специфічну термінологію, вузькопрофільну лексику, а іноді й декілька мов в одному тексті. Це висуває вимоги до моделі ембедінгів, яка повинна коректно працювати з багатомовними наборами даних та специфічними словниками.

Ще одним аспектом аналізу предметної області є потреба в безпеці та конфіденційності. Документом може бути комерційна таємниця, юридичний документ або персональні дані. Отже, система має забезпечувати доступ до документів згідно з політикою безпеки: неавторизовані користувачі не повинні мати можливості переглядати або копіювати інформацію. Це висуває вимоги до механізмів аутентифікації, авторизації, шифрування з’єднання між клієнтом та сервером, а також до управління ролями і правами доступу.

Інтеграція з існуючими інструментами документообігу — це ще одна складова предметної області. Багато організацій уже мають системи зберігання файлів, файлообмінники, системи планування та управління проектами, CRM або ERP-системи. Для досягнення максимальної ефективності новий продукт має легко інтегруватися з цими системами: через API, веб-хуки, конекторні модулі тощо. Це дозволить побудувати цілісну екосистему управління даними, в якій пошуковий інструмент відіграватиме роль «розумного» помічника, що знаходить потрібну інформацію у зручній для користувача формі.

Щодо інтерфейсу користувача, то предметна область диктує вимогу простоти і зрозумілості. Навіть якщо під капотом працюють складні алгоритми машинного навчання, кінцевий користувач хоче простий інтерфейс: поле для введення запиту, можливість вибору фільтрів (дата, автор, тип документу), а також зрозуміле сортування та групування результатів. Корисним може бути функціонал перегляду фрагментів документів, щоб одразу побачити контекст пошукової відповідності, без потреби відкривати весь документ.

Важливу роль відіграє масштабованість. Предметна область управління документами — це динамічна сфера, де обсяги даних постійно зростають. Система, яка добре працює з кількома сотнями документів, повинна адекватно реагувати й на ситуації, коли йдеться про сотні тисяч або навіть мільйони файлів. Масштабованість означає здатність системи гнучко розподіляти навантаження між кількома серверами, ефективно кешувати часто використовувані запити, забезпечувати високу відмовостійкість та швидку індексацію нових документів. Саме тому при виборі технічних рішень треба звертати увагу на використання розподілених баз даних, механізмів шардінгу, хмарних обчислень та індексаційних служб.

Ще один аспект — дотримання стандартів і протоколів. У багатьох галузях (юриспруденція, фінанси, медицина, освіта) існують спеціальні вимоги до збереження і використання документів: терміни зберігання, необхідність створення резервних копій, відповідність нормам захисту даних (наприклад, GDPR у Європейському Союзі). Система управління документами, яка претендує на широке застосування, повинна врахувати ці нормативні та регуляторні аспекти, надаючи інструменти для аудиту, логування, відновлення даних та контролю версій документів.

Не можна оминати й питань продуктивності. Швидкий відгук системи на запит користувача суттєво впливає на загальну якість роботи. Якщо пошук триває надто довго, користувачі будуть незадоволені. Використання ембедінгів, індексів та ефективних алгоритмів пошуку відіграє ключову роль у досягненні прийнятної

швидкодії. Оптимізація може включати передобчислення ембедінгів, створення спеціалізованих індексів для векторного пошуку, застосування кешування результатів запитів та інші техніки.

Загалом, аналіз предметної області демонструє, що сучасна система управління документами, здатна здійснювати семантичний пошук, повина поєднувати низку технологій та підходів. Векторні представлення тексту, чанкування, інтеграція з базами даних та зовнішніми сервісами, облік безпеки, масштабованість та продуктивність — усе це формує складний, багаторівневий простір завдань. Водночас, вирішення цих завдань відкриває шлях до створення системи, яка кардинально спрощує роботу з документами, робить пошук інформації дійсно «розумним» та суттєво підвищує ефективність робочих процесів.

Таким чином, предметна область сучасних систем управління документами — це не просто цифрове сховище файлів. Це динамічний, інтелектуальний простір, де документи структуруються, індексуються та аналізуються таким чином, щоб користувач отримав максимум корисної інформації з мінімальними витратами часу і зусиль. Застосування ембедінгів, векторного пошуку, чанкування та суміжних технологій дозволяє здійснити перехід від простих пошукових інструментів до повноцінних інтелектуальних систем, що розуміють зміст документів і можуть надавати користувачеві контекстуально релевантні відповіді. Це не просто еволюція, а справжня революція у сфері управління інформацією.

2. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА СТРУКТУРИ ПРОГРАМНОГО ПРОДУКТУ

2.1. Розробка архітектури програмного продукту

При створенні програмного продукту, що оперує великими обсягами даних, надзвичайно важливо приділити увагу правильному проектуванню архітектури. Вибір архітектурного підходу та пов’язаних із ним технологій суттєво впливають на продуктивність, масштабованість, надійність та зручність використання системи [2, 3]. У даному випадку завдання полягає в ефективній обробці текстових документів, їхньому розбитті на фрагменти, отриманні векторних уявлень, реалізації семантичного пошуку за змістом, а також у забезпеченні оперативної взаємодії з користувачем через веб-інтерфейс.

Архітектура має бути розроблена з урахуванням можливості швидкого масштабування для обробки дедалі більших обсягів даних, а також мати чітко визначені зони відповідальності між компонентами. Це дозволить гнучко розвивати продукт у майбутньому, інтегрувати нові функціональні можливості, реагувати на зміну вимог та забезпечувати стабільну роботу за різних навантажень.

Основні концепції та підходи до побудови архітектури

При проектуванні архітектури програмного продукту важливим є вибір парадигми розробки. Серед популярних рішень для складних веб-застосунків виділяють мікросервісну та монолітну архітектури [2, 4, 5]. Монолітна архітектура характеризується об’єднанням усього функціоналу в один цілісний застосунок, що полегшує початкову розробку, але ускладнює масштабування, оновлення та підтримку. Натомість мікросервісна архітектура пропонує розділення застосунку на незалежні компоненти (сервіси), кожен із яких реалізує окрему бізнес-логіку або функцію. На рисунку нижче наведено приклад (рис. 2.1).

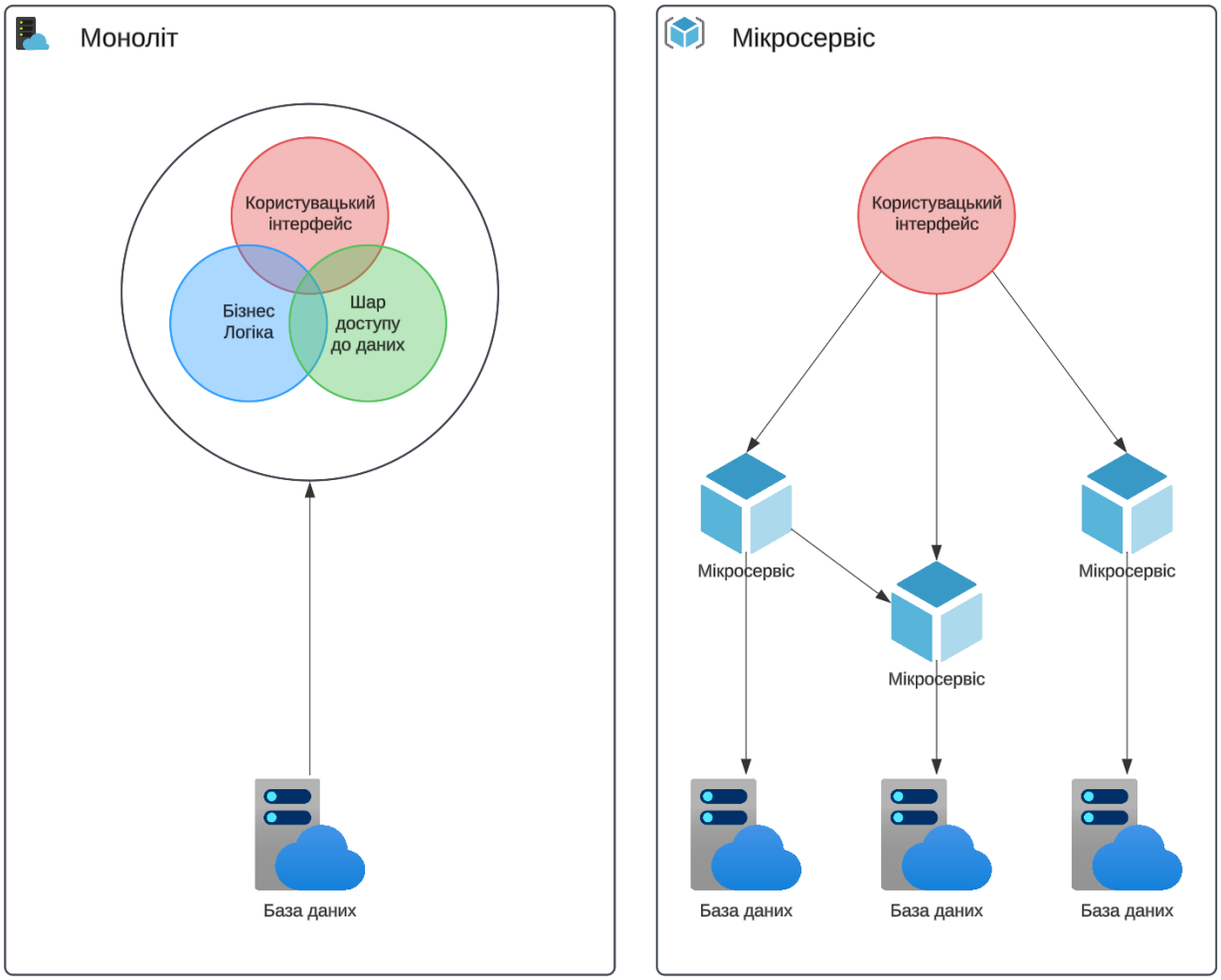

Рисунок 2.1 – Архітектури моноліту та мікро сервісу

Для даного проєкту мікросервісний підхід є більш придатним, оскільки обробка великих обсягів даних, робота з векторними репрезентаціями, інтеграція з зовнішніми API та забезпечення масштабованості є ключовими вимогами [6, 7]. Мікросервіси дозволять виділити окремі функціональні блоки, такі як:

Сервіс завантаження та попередньої обробки документів. Він прийматиме файли від користувача, здійснюватиме первинне читання, перевірку формату, за потреби – перетворення зображень в текст (OCR) та попередню нормалізацію вмісту.

Сервіс чанкування та ембедінгу. Даний компонент буде відповідальний за розбиття тексту на фрагменти (чанки), виклик зовнішнього сервісу чи моделі для отримання векторних уявлень та подальше збереження результатів.

Сервіс пошуку та індексації. Він реалізовуватиме логіку семантичного пошуку. Зберігатиме векторні індекси та метадані, забезпечуватиме швидкий пошук релевантних фрагментів документів, ранжування результатів і повернення їх на запити користувача.

Сервіс користувацького інтерфейсу (Frontend). Це рівень взаємодії з кінцевим користувачем, який відображає результати пошуку, надає форми для завантаження файлів та параметрів для фільтрації чи уточнення запитів.

Сервіс керування доступом і безпекою. Відповідає за аутентифікацію та авторизацію користувачів, керування ролями та правами доступу, а також за шифрування трафіку між компонентами системи.

Кожен із зазначених сервісів може бути розгорнутий на окремому сервері чи контейнері, що дозволяє гнучко масштабувати окремі ділянки системи за необхідності.

Використання Node.js, Nest.js та пов’язаних технологій

У серверній частині доцільно застосувати Node.js — асинхронне середовище виконання JavaScript, яке добре підходить для високонавантажених, мережево-орієнтованих застосунків [8]. Завдяки подієво-орієнтованій моделі та неблокуючому вводу-виводу, Node.js забезпечує високу продуктивність та стабільну роботу за великої кількості одночасних підключень.

Один із перевірених фреймворків для побудови надійних серверних застосунків на Node.js — Nest.js. Його ключові переваги — модульна архітектура, вбудоване впровадження залежностей, підтримка TypeScript, зрозуміла структура проекту і високий ступінь повторного використання коду. Nest.js особливо добре підходить для розробки мікросервісів завдяки наявності інструментів для інтеграції з транспортними протоколами (HTTP, gRPC), взаємодії з брокерами повідомлень (RabbitMQ, Kafka) та іншими сервісами.

У межах Nest.js можна створити окремі модулі для різних аспектів роботи системи: модуль для завантаження файлів, модуль пошуку, модуль ембедінгу,

модуль взаємодії з базою даних тощо. Завдяки цьому забезпечується чітка логічна сегментація, а також полегшується тестування та супровід.

Взаємодія з базою даних

Збереження даних є одним із ключових аспектів архітектури. В нашому випадку, коли йдеться про великі обсяги документів, які можуть мати гнучку структуру, використання нереляційних (NoSQL) баз даних, таких як MongoDB, стає оптимальним вибором [9]. MongoDB зберігає дані у форматі BSON, що нагадує JSON, а це полегшує роботу з напівструктурованими та неструктурованими даними, якими є текстові документи та пов’язані з ними метадані.

MongoDB може бути розподілена на декількох вузлах, що забезпечує горизонтальне масштабування, реплікацію даних і високий рівень відмовостійкості. Це особливо важливо в контексті безперервного зростання обсягу інформації. Крім того, вона пропонує можливості для зберігання та швидкого пошуку векторів через інтеграцію спеціалізованих індексів або зовнішніх інструментів.

Слід відзначити, що для індексації векторних даних (ембедінгів) можуть застосовуватися спеціалізовані механізми [10]. Наприклад, використання вбудованих функцій MongoDB для векторних індексів або інтеграція з зовнішніми інструментами. Це дозволить здійснювати швидке семантичне порівняння, пошук близьких векторів та ефективно обробляти запити користувачів на семантичний пошук.

Зовнішні сервіси для ембедінгу та аналітики

Сучасні сервіси обробки природної мови, такі як OpenAI API, надають можливість виконувати складні операції над текстом без необхідності розгортати власні моделі глибинного навчання. Застосування зовнішніх API дозволяє зосередитись на бізнес-логіці, не витрачаючи ресурси на підготовку та хостинг мовних моделей.

Однак така залежність від зовнішнього сервісу потребує стабільного інтернет-з’єднання, продуманого кешування результатів для зменшення вартості та часу відгуку, а також врахування обмежень та політик використання API. Архітектура повинна включати шар абстракції над сервісом ембедінгу, щоб у разі потреби можна було змінити постачальника або модуль обчислення ембедінгів з мінімальними модифікаціями коду.

Клієнтський інтерфейс (Frontend)

Для клієнтської частини логічно обрати фреймворки типу React, Next.js або Vue.js [11, 12]. Ці інструменти дозволяють будувати динамічні, інтерактивні та швидкі інтерфейси, які будуть реагувати на дії користувачів у реальному часі. На клієнтському боці можна організувати форми завантаження файлів, наочне відображення результатів пошуку, а також панелі управління параметрами.

Фронтенд взаємодіє з бекендом через REST API або GraphQL API. Наприклад, REST endpoints можуть призначатися для завантаження файлу, запуску процесу ембедінгу, пошуку за семантикою, отримання списку документів, видалення чи оновлення метаданих. Важливо забезпечити безпечний обмін даними, застосовуючи HTTPS протоколи та, при потребі, JWT-токени для аутентифікації.

Безпека та керування доступом

Обробка документів часто пов’язана з конфіденційною інформацією. Тому архітектура має включати шар аутентифікації та авторизації користувачів. Це можуть бути модулі Nest.js для обробки JWT-токенів, OAuth 2.0 чи інтеграції з зовнішньою системою управління особистими даними. Користувач повинен мати права лише на ті дані, до яких йому надано доступ. Також важливо забезпечити логування операцій, аудит змін та при потребі реалізувати механізми шифрування даних у спокої (at-rest) і в русі (in-transit).

Масштабування, резервне копіювання та відмовостійкість

У разі збільшення навантаження, мікросервісна архітектура дозволяє запускати додаткові екземпляри сервісів пошуку, обробки ембедінгів чи

завантаження документів [15]. Балансувальник навантаження розподілить вхідні запити між усіма доступними інстансами, забезпечуючи високу доступність системи.

Для захисту від збоїв слід налаштувати реплікацію бази даних, резервне копіювання та моніторинг роботи усіх компонентів. Інструменти типу Prometheus та Grafana допоможуть відслідковувати метрики продуктивності та стабільності, а системи алертингу — оперативно повідомляти про проблеми.

Інтеграція з іншими системами

Розроблений програмний продукт може знадобитися інтегрувати з іншими системами: системами управління проектами, CRM, ERP, сервісами збирання аналітики чи хмарними сховищами. Для цього можна передбачити спеціальні модулі або шлюзи, які реалізують API для імпорту та експорту даних, а також обмін повідомленнями через брокери (RabbitMQ, Kafka).

Архітектура має бути достатньо гнучкою для легкого додавання нових інтерфейсів взаємодії. Важливо документувати кожний API-ендпоінт, забезпечувати стабільність та зворотну сумісність, якщо можливо, та надавати розробникам інструменти для тестування інтеграцій.

Переваги мікросервісної та модульної архітектури

Обраний підхід до архітектури забезпечує низку переваг:

Масштабованість: Кожний мікросервіс можна масштабувати окремо, додаючи нові екземпляри у відповідь на зростання навантаження.

Гнучкість: Легше впроваджувати нові функціональні можливості, змінювати постачальника ембедінгів чи базу даних, не ламаючи решту системи.

Надійність: Збої у одному сервісі не паралізують всю систему. Решта компонентів можуть продовжити роботу, поки несправність не буде усунена.

Продуктивність: Використання векторних індексів і семантичного пошуку забезпечує швидку та точну видачу результатів, зменшуючи час, необхідний користувачеві на пошук потрібної інформації.

Простота супроводу: Логічне розділення компонентів за функціональністю полегшує тестування, оновлення, усунення помилок та розгортання.

Проектування архітектури програмного продукту для обробки великих обсягів документів та здійснення семантичного пошуку — складний, проте вирішальний етап. Використання мікросервісного підходу, Node.js з Nest.js, MongoDB для зберігання даних та інтеграція з зовнішнім API для ембедінгу тексту формують стійкий фундамент. Така архітектура є гнучкою, масштабованою, безпечною та зручною для розвитку в майбутньому.

В результаті кінцевий користувач отримує зручний інтерфейс, швидкий доступ до потрібної інформації, можливість семантичного пошуку, а інженери — простий у підтримці та модифікації програмний продукт. Завдяки цьому підходу можна швидко реагувати на зростання обсягів даних, підвищувати якість пошуку, інтегрувати додаткові сервіси та адаптувати систему під конкретні вимоги.

2.2. Аналіз інструментальних засобів розробки

При виборі інструментальних засобів для розробки програмного забезпечення важливо враховувати не лише мову програмування та фреймворки, а й середовище розробки (IDE або редактор коду), доступні плагіни, інструменти для тестування, контролю версій та розгортання. Комплексний підхід до цього питання забезпечує ефективність процесу створення застосунку, зрозумілість архітектури, зручність налагодження, а також гнучкість подальшого розвитку проєкту.

Мова програмування та пов’язані інструменти:

У нашому випадку для розробки було обрано мову програмування JavaScript (та її надмножину TypeScript, де це доцільно), оскільки вона забезпечує кросплатформеність, підтримується у великій кількості середовищ, має широкий набір готових бібліотек та фреймворків, а також має потужну спільноту

розробників. JavaScript є невіддільною частиною веб-розробки та застосовується як на фронтенді (React, Next.js, Vue.js), так і на бекенді (Node.js, Nest.js). Така універсальність дозволяє уніфікувати технологічний стек застосунку, використовувати одну й ту саму мову на всіх рівнях архітектури та скоротити час на навчання додаткових мов.

У рамках екосистеми Node.js можна ефективно керувати залежностями за допомогою пакетних менеджерів (npm, yarn), встановлювати, оновлювати та видаляти сторонні бібліотеки й фреймворки. Також JavaScript/TypeScript добре інтегрується з інструментами для статичного аналізу коду (ESLint, Prettier), що покращують якість коду та забезпечують дотримання обраних стандартів оформлення. TypeScript, у свою чергу, надає систему типів, що підвищує надійність та зрозумілість коду, знижує кількість помилок під час розробки та полегшує налагодження.

Середовища розробки та редактори коду:

Серед безлічі інструментальних середовищ найбільш популярними для JavaScript/TypeScript є Visual Studio Code (VS Code), JetBrains WebStorm, IntelliJ IDEA та інші. Кожне з них має свої переваги.

VS Code — легкий та швидкий редактор коду, що надає широкі можливості розширення за допомогою плагінів. Він безкоштовний, кросплатформений, має інтуїтивний інтерфейс, підтримує інтеграцію з Git, терміналом та інструментами для налагодження. Також VS Code пропонує IntelliSense — систему автодоповнення та підказок, яка значно прискорює роботу розробника. Встановлення розширень для підтримки TypeScript, ESLint, Prettier, а також специфічних фреймворків (Nest.js, React) робить процес розробки комфортним та прозорим.

WebStorm — комерційне IDE, спеціалізоване на JavaScript і TypeScript. Воно надає широкий набір вбудованих інструментів для рефакторингу, налагодження, тестування, інтеграції зі системами контролю версій. WebStorm має потужний інтегрований інтелектуальний аналіз коду, що дозволяє виявляти помилки та

неефективні конструкції ще до запуску програми. Перевагою є глибока інтеграція з фреймворками, такими як React, Vue, Angular, Node.js та інші. Однак WebStorm є платним продуктом, що може стати перешкодою для деяких команд, особливо на етапі старту проєкту.

IntelliJ IDEA — це багатофункціональне IDE від JetBrains, більш відоме для мов Java та Kotlin, але також придатне й для JavaScript/TypeScript. Воно може бути надмірним для чисто JavaScript-орієнтованого проєкту, оскільки має дуже широкі функціональні можливості, які не завжди потрібні. Тим не менше, якщо в проєкті планується використання кількох мов або інтеграція зі складними бекенд-системами, IntelliJ IDEA може виявитись вдалим вибором.

Зважаючи на умови проєкту, його масштаб, стек технологій (Node.js, Nest.js, React, MongoDB), а також потребу в максимально швидкому старті розробки, було обрано Visual Studio Code. Причини такого вибору полягають у простоті встановлення, гнучкості, великій кількості доступних розширень, високій продуктивності та активній спільноті користувачів.

Інструменти контролю версій та організації робочого процесу:

Для спільної роботи над проєктом необхідно використовувати системи контролю версій, такі як Git. Git дозволяє відстежувати історію змін, створювати гілки для нових функцій, виправлень помилок і тестування, а також здійснювати безболісне злиття коду з різних гілок. Зазвичай Git використовують разом із GitHub, GitLab чи Bitbucket як віддаленими репозиторіями та платформами для командної співпраці, рев'ю коду, трекінгу задач та інтеграції з CI/CD.

Поєднання Git з інструментами безперервної інтеграції та доставки (CI/CD), такими як GitHub Actions, GitLab CI або CircleCI, дозволяє автоматизувати тестування, збірку та деплой застосунку. Це підвищує якість та надійність продукту, знижує ризик появи помилок у промисловому середовищі та пришвидшує цикл розробки.

Інструменти для тестування та налагодження:

Важливим аспектом розробки є забезпечення якості коду та коректної роботи функціоналу. Для цього використовують інструменти тестування, наприклад Jest або Mocha, які дозволяють створювати модульні, інтеграційні та end-to-end тести. Jest інтегрується з React, Node.js, Nest.js та TypeScript, забезпечуючи зрозумілий синтаксис для опису тестів та корисні звіти про покриття коду тестами.

Для налагодження коду можна скористатися вбудованими можливостями VS Code, поставити брейкпойнти, переглядати змінні під час виконання, крокувати по коду та швидко знаходити джерело проблем. Це значно спрощує виправлення помилок та підвищує продуктивність розробника.

Інструменти для документування:

Документування коду та API є важливим етапом у розробці будь-якого проєкту. На рівні коду можна використовувати JSDoc або TypeDoc (для TypeScript), які генерують документацію з коментарів у коді. Для опису REST API чи GraphQL API можна застосувати Swagger/OpenAPI чи GraphQL Playground, що допомагають команді та стороннім розробникам швидко розібратися у можливостях API, форматі запитів та відповідей. Це впливає на зручність розвитку проєкту, його підтримку та інші аспекти життєвого циклу програмного забезпечення.

Інструменти для аналітики та моніторингу:

Після розгортання застосунку у промисловому середовищі необхідним стає моніторинг продуктивності та аналіз логу для виявлення помилок і вузьких місць у роботі системи. Для цього можна використовувати інструменти на зразок Prometheus, Grafana для візуалізації метрик, Elastic Stack (ELK) для аналізу логів або Sentry для відстеження винятків. Хоча це не завжди стосується початкового етапу розробки, передбачити можливість швидкої інтеграції з подібними інструментами — хороша практика.

Висновок щодо вибору інструментів:

Обрання JavaScript/TypeScript у поєднанні з Node.js, Nest.js, React та MongoDB — оптимальний вибір для створення сучасної, гнучкої та масштабованої системи. Використання VS Code як редактора коду, Git для контролю версій, Jest

для тестування, ESLint і Prettier для дотримання кодстайлу та інструментів документування типу Swagger забезпечує зручність і ефективність у розробці. Такий набір інструментальних засобів дозволяє команді швидко розпочати роботу, легко адаптуватися до змін вимог та технологій, а також підтримувати високий рівень якості й зрозумілості коду.

Таким чином, належний аналіз та підбір інструментальних засобів розвитку програмного продукту є важливим етапом, який суттєво впливає на продуктивність розробників, якість кінцевого результату та спроможність проєкту зберігати актуальність і гнучкість у довгостроковій перспективі.

2.3 Пошук акторів та варіантів використання

У процесі проектування програмного продукту важливим етапом є визначення користувацьких ролей (акторів) та варіантів їх взаємодії із системою. Чітке розмежування прав доступу та функціональних можливостей кожного типу користувачів допомагає забезпечити безпеку, масштабованість та простоту використання програмного продукту. У підсумку, це дозволяє створити рішення, яке задовольняє потреби різних категорій користувачів: від звичайних відвідувачів, які бажають отримати доступ до загальної інформації, до адміністраторів, відповідальних за технічну підтримку та управління системою. На рисунку нижче наведено приклад (рис. 2.2).

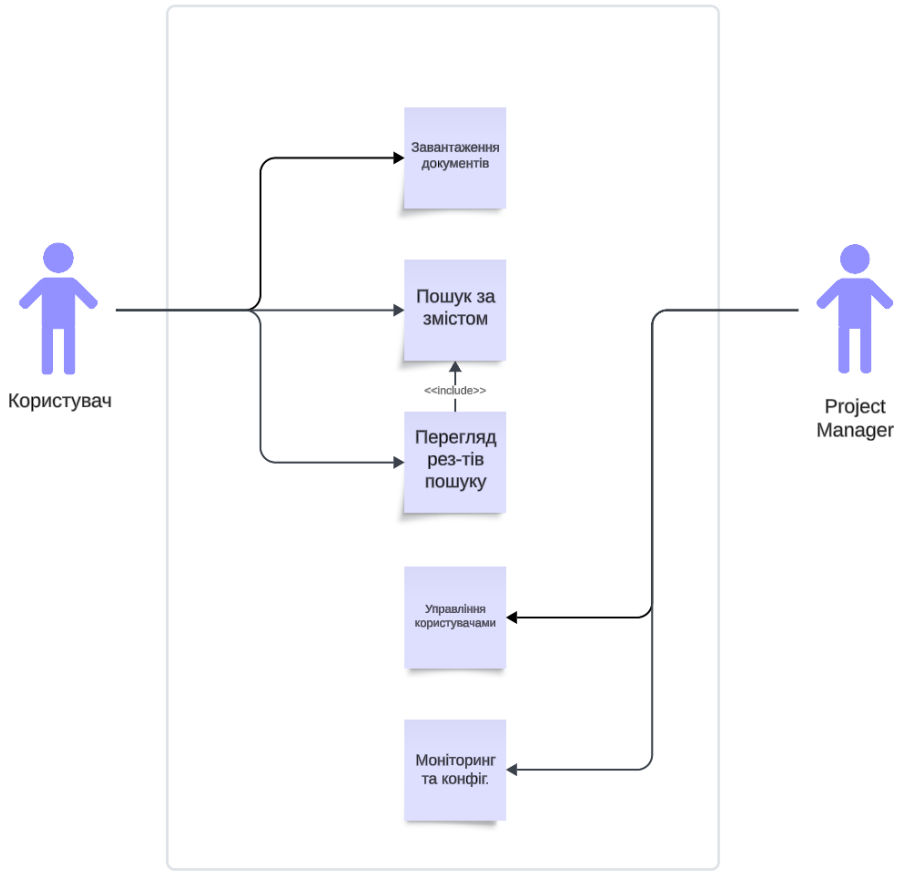

Рисунок 2.2 – Діаграма варіантів використання

Визначення ролей (акторів):

Загальний підхід до розмежування доступу у системі полягає у тому, що користувачі отримують розширені права користування. На вершині ієрархії знаходиться адміністратор, котрий контролює роботу системи, забезпечує безпеку і стабільність.

Користувач

Як тільки відвідувач успішно пройде процедуру авторизації (надавши коректний логін та пароль), він перетворюється на повноцінного «учасника» системи. Тепер він має значно ширші можливості:

- Може завантажувати власні документи. Система автоматично обробляє завантажений файл: здійснює розбиття на логічні фрагменти (чанкування), ембедінг для представлення змісту у векторному форматі, і зберігає результати у базі даних.

- Може виконувати семантичний пошук за змістом, знаходячи релевантні фрагменти документів.
- Може переглядати результати пошуку, уточнювати запити, застосовувати фільтри, обирати потрібні фрагменти для детального перегляду.
- Має змогу налаштовувати власний профіль: змінювати пароль, оновлювати контактну інформацію та інші персональні параметри.

Так, авторизований користувач отримує ключ до основного функціоналу системи. Він тепер не просто гість, а активний учасник процесу, який може вільно користуватись наданими інструментами для опрацювання документів та отримання корисної інформації.

Адміністратор

Нарешті, є актор із найвищим ступенем впливу — адміністратор. Це не пересічний користувач, а фахівець, який відповідає за роботу системи:

- Він може керувати обліковими записами: активувати нових користувачів, блокувати порушників, змінювати ролі.
- Має доступ до статистики, логів, моніторингу продуктивності, може контролювати завантаження системи, регулювати доступ до зовнішніх сервісів.
- Здатен змінювати конфігурації: від параметрів кешування до оновлення ключів для ембедінг-сервісу.
- Забезпечує безпеку, надійність, проводить бекап даних і дбає про те, щоб система залишалася стабільною навіть за підвищених навантажень.

Адміністратор — це гарант стабільності та ефективності. Він має достатньо повноважень, аби адаптувати систему під мінливі вимоги, зберігаючи водночас безпеку і цілісність даних.

Таким чином, у системі чітко виділяються дві основні ролі, кожна з яких має свій набір прав і можливостей. Користувач отримує ключ до повного функціоналу:

завантаження, пошук, перегляд результатів, налаштування профілю. Адміністратор же контролює весь процес, забезпечуючи правильну роботу та стабільний розвиток системи.

2.4 Опис варіантів використання

Після визначення акторів та їх ролей у системі, наступним кроком є детальний опис варіантів використання (Use Cases). Це забезпечує чітке розуміння того, як саме користувачі взаємодіють із системою, які кроки здійснюються для досягнення конкретних цілей і які результати вони отримують. Такий підхід допомагає формалізувати вимоги до функціональності, спростити розробку та полегшити створення тестових сценаріїв.

Нижче наведено описи ключових варіантів використання, визначених під час аналізу предметної області та користувацьких ролей. Кожен варіант містить інформацію про актора, цілі, основний сценарій взаємодії, передумови, постумови та можливі альтернативні чи помилкові шляхи.

1. Завантаження документів (Upload Documents)

Актор: Авторизований користувач

Ціль: Завантажити текстовий документ у систему для подальшого пошуку за змістом.

Передумови:

Користувач авторизований та має активну сесію.

Система готова до збереження нових документів і взаємодії з інструментами обробки даних.

Основний сценарій:

Користувач відкриває сторінку «Завантаження документів» у своєму особистому кабінеті.

Користувач натискає кнопку «Обрати файл» та вибирає необхідний документ (PDF або інший текстовий формат).

Користувач підтверджує завантаження.

Система приймає файл, перевіряє формат, за потреби виконує OCR (якщо документ — зображення), розбиває текст на чанки, отримує ембедінги через зовнішнє API.

Система зберігає результати в базу даних (текстові фрагменти, ембедінги, метадані).

Система повідомляє користувача про успішне завантаження документа.

Альтернативні сценарії:

Якщо документ некоректного формату або перевищує максимальний розмір, система повідомляє про помилку.

Якщо виникли проблеми зі з’єднанням до зовнішнього API ембедінгів, система може зберегти документ у чергу для повторної обробки пізніше.

Постумови:

Документ доступний для семантичного пошуку авторизованого користувача.

2. Пошук за змістом (Semantic Search)

Актор: Авторизований користувач

Ціль: Знайти фрагменти документів, які семантично відповідають запиту користувача.

Передумови:

Користувач авторизований.

У системі є завантажені та оброблені документи з ембедінгами.

Основний сценарій:

Користувач відкриває сторінку пошуку у власному кабінеті.

Користувач вводить пошуковий запит у текстове поле.

Система перетворює запит у вектор за допомогою ембедінгів (звертаючись до зовнішнього API або використовуючи локальний кеш).

Система порівнює вектор запиту з векторами фрагментів документів у базі, виконує пошук найближчих сусідів.

Система формує список найбільш релевантних результатів і відображає їх користувачу.

Альтернативні сценарії:

За відсутності релевантних результатів система повідомляє користувача про відсутність збігів.

Якщо зовнішнє API ембедінгів недоступне, система може використовувати попередньо кешовані результати або повернути відповідне повідомлення про помилку.

Постумови:

Користувач бачить релевантні фрагменти документів, має змогу переглянути деталі або спробувати інший запит.

3. Перегляд результатів пошуку (View Search Results)

Актор: Авторизований користувач

Ціль: Ознайомитися з контекстом знайдених документів та за потреби переглянути оригінал.

Передумови:

Користувач щойно виконав пошук.

Є знайдені релевантні фрагменти.

Основний сценарій:

Користувач бачить список результатів: заголовок фрагменту, короткий опис чи частину тексту.

Користувач вибирає конкретний результат.

Система відображає детальніший фрагмент, контекст у документі або надає посилання на завантаження оригіналу.

Альтернативні сценарії:

Якщо фрагмент більше не доступний (наприклад, документ було видалено), система повідомляє користувача.

Постумови:

Користувач краще розуміє зміст знайденого фрагмента та може використовувати його у власних цілях.

4. Управління користувачами (Manage Users)

Актор: Адміністратор

Ціль: Забезпечити безперебійну роботу системи, контролюючи доступ користувачів.

Передумови:

Адміністратор авторизований із правами на керування користувачами.

Основний сценарій:

Адміністратор відкриває панель управління користувачами.

Отримує список зареєстрованих користувачів, може фільтрувати, сортувати за станом.

Адміністратор вибирає конкретного користувача для активації, деактивації або блокування.

Система оновлює дані у базі, змінюючи статус користувача.

Альтернативні сценарії:

Якщо адміністратор намагається заблокувати неіснуючий або вже видалений обліковий запис, система виводить помилку.

Якщо у системі налаштовані складні ролі та права, адміністратор може призначати користувачу певні ролі (наприклад, «модератор»).

Постумови:

Система відображає оновлені дані про користувачів, дозволяючи підтримувати порядок, безпеку та контроль за ресурсами.

2.5 Шаблон розробки застосунку

Сучасні технології значно вплинули на те, як люди взаємодіють з програмним забезпеченням. Веб-застосунки стали невід'ємною частиною повсякденного життя, завдяки їх доступності, гнучкості та зручності використання. В основі веб-застосунків лежить клієнт-серверна архітектура, яка забезпечує ефективний обмін даними між користувачем (клієнтом) і сервером. У цій архітектурі клієнтом виступає веб-браузер, який відправляє запити до сервера, а сервер обробляє їх та повертає результати. Основна схема роботи веб-застосунку представлена на рисунку (рис. 2.3).

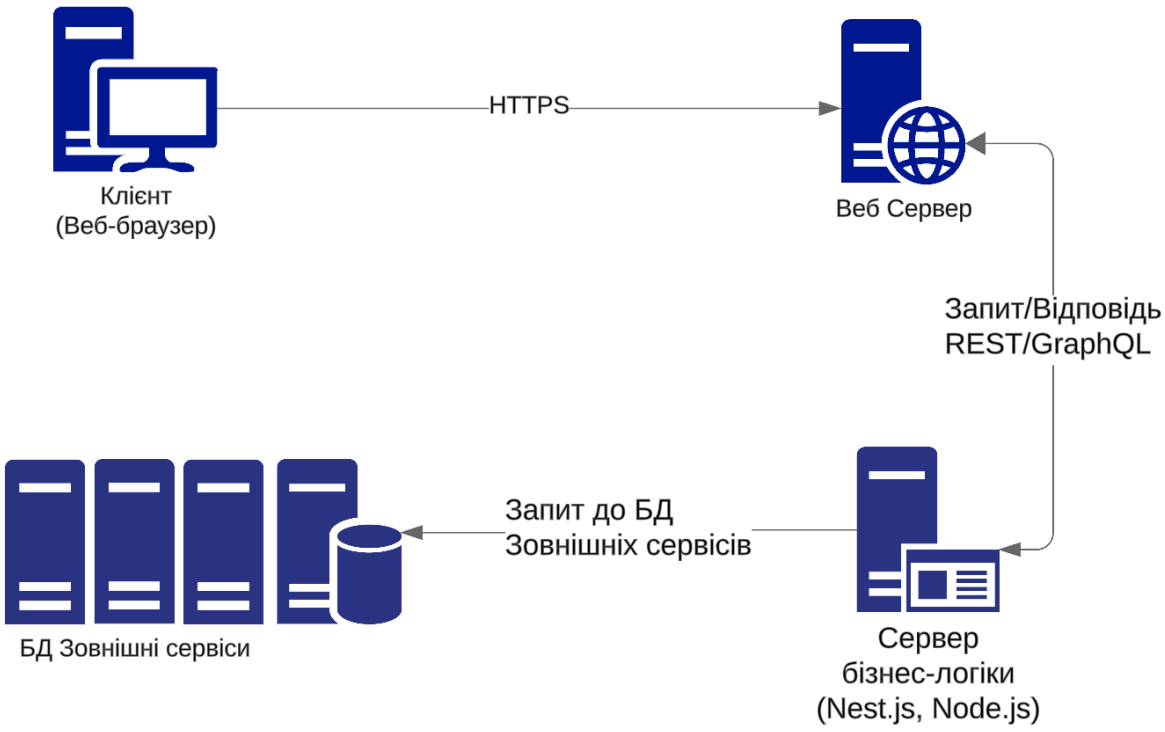

Рисунок 2.3 – Схема роботи веб-застосунку

Під час вибору архітектурного підходу для розробки даного програмного продукту було враховано кілька ключових переваг веб-браузера як клієнта:

Універсальність

Веб-браузери доступні на різноманітних пристроях, включаючи комп’ютери, смартфони, планшети та навіть побутову техніку. Це дозволяє уникнути необхідності створення окремих програм для кожної операційної системи чи платформи.

Комплексність

Сучасні браузери підтримують додаткові технології, такі як JavaScript, WebAssembly, та модулі, що дозволяють виконувати складні обчислення і взаємодії, недоступні для класичних програм.

Безпека

Веб-браузери забезпечують безпечну передачу даних між клієнтом і сервером за допомогою таких технологій, як HTTPS, а також підтримують механізми захисту, наприклад, політику CORS та обмеження доступу до локальних ресурсів.

Багаторівневість

Більшість сучасних веб-застосунків мають багаторівневу архітектуру. Така структура складається з кількох рівнів:

- Клієнтський рівень (веб-браузер) відповідає за відображення інтерфейсу та взаємодію з користувачем.
- Веб-сервер приймає та обробляє запити клієнта.
- Сервер бізнес-логіки реалізує обробку даних, виконання алгоритмів та інтеграцію з іншими сервісами.
- Сервер бази даних забезпечує зберігання та обробку структурованої інформації.

Кожен із цих рівнів може виступати як клієнт або сервер залежно від характеру взаємодії. Наприклад, сервер бізнес-логіки діє як клієнт, надсилаючи запити до бази даних.

Архітектурний підхід

Для реалізації програмного продукту було обрано архітектуру багаторівневого веб-застосунку, яка забезпечує:

Масштабованість — система легко адаптується до збільшення кількості користувачів чи розширення функціональності.

Гнучкість — можливість додавання нових компонентів або зміни існуючих без шкоди для інших частин системи.

Високу продуктивність — розподіл функцій між рівнями дозволяє уникнути перевантаження окремих компонентів.

Зважаючи на переваги описаного підходу, багаторівнева клієнт-серверна архітектура була обрана як основа для розробки даного веб-застосунку.

2.6 Абстрактний рівень системи

Під час проєктування програмного продукту, що реалізований за допомогою об’єктно-орієнтованого підходу (ООП), ми отримуємо чітку та логічну структуру коду. ООП дає змогу розділити програму на окремі сутності (класи), які відповідають певним концепціям предметної області. В наданому коді такі сутності представлені класами-ентітями, які описують зберігання та обробку документів, файлів, клієнтів та лейблів. На рисунку нижче наведено приклад (рис. 2.4, 2.5).

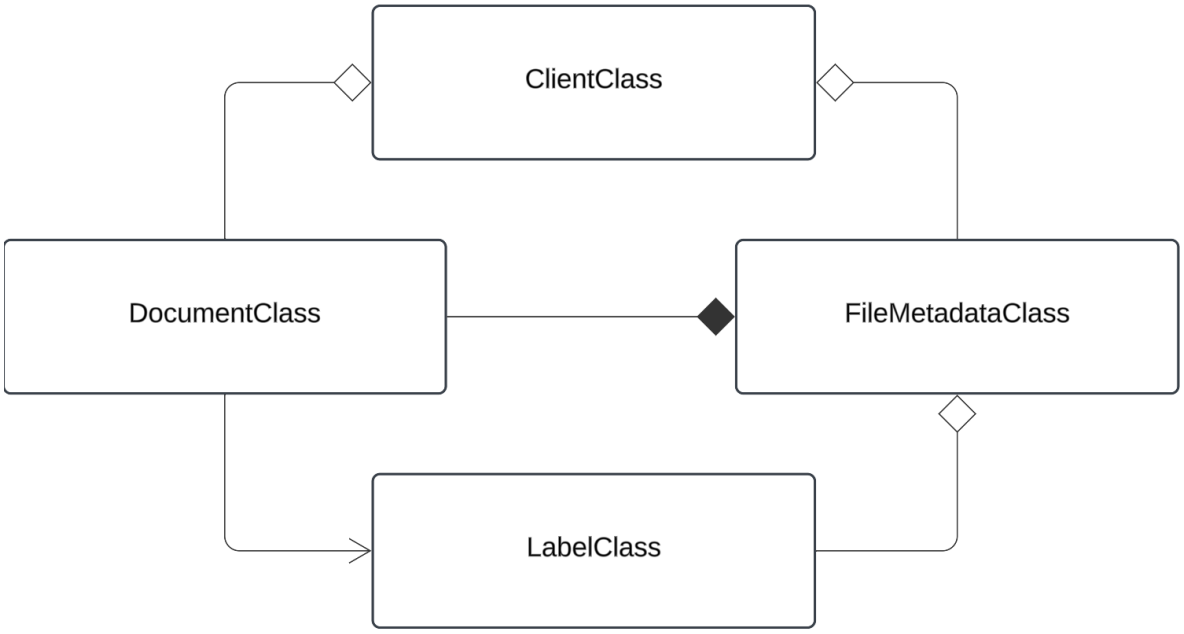

Рисунок 2.4 – UML діаграма класів

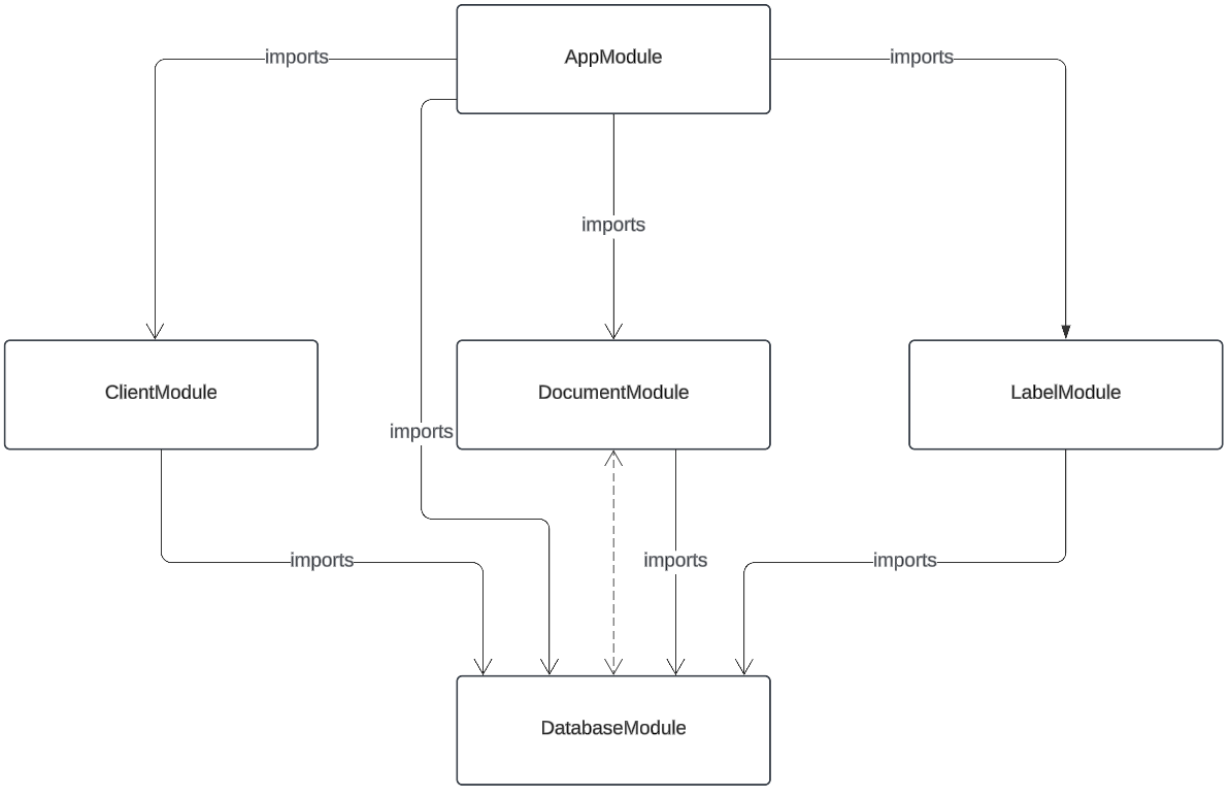

Рисунок 2.5 – UML діаграма модулів

Основні класи в системі — це:

- Client: описує клієнта системи з його основними даними (ім’я, email, телефон, статус).
- Document: представлення документа, що включає інформацію про клієнта (client_id), назву, ім’я файлу, текст та векторне представлення (embedding), а також пов’язані лейбли.
- FileMetadata: містить метадані про завантажені файли (наприклад, document_name, дати завантаження та оновлення, статус активності та прив’язані лейбли).
- Label: описує лейбли (мітки), які можуть бути призначені документам чи файлам. Лейбл має назву та векторне представлення (embedding).

Використання ООП-концепцій дозволяє:

- Інкапсуляція: Сховати внутрішню логіку обробки сутностей за методами класів. Наприклад, доступ до даних клієнтів або документів здійснюється через поля та методи відповідних класів, без необхідності знати внутрішні деталі реалізації.
- Успадкування (за потреби): За необхідності можна створювати ієрархії класів для спільного функціоналу, але в даному коді переважно наведені окремі сутності без складної ієрархії.
- Поліморфізм та абстракція: Можливі у випадку, якщо будуть введені інтерфейси чи базові класи. Потенційно можна визначити базові інтерфейси для роботи з даними та їх конкретні реалізації.

Таким чином, абстрактний рівень системи полягає у відображенні предметної області у вигляді взаємопов’язаних класів-ентітей. Це сприяє чіткості архітектури, спрощує підтримку коду та полегшує майбутні доповнення чи модифікації системи.

2.7 Архітектура на рівні класів і підсистем

Застосунок, що оперує документами, клієнтами, файлами та лейблами, побудовано на основі модульної та багато-рівневої архітектури. В основі лежить клієнт-серверна модель, де серверна частина складається з кількох логічних підсистем (модулів):

- DatabaseModule: Надає доступ до бази даних (MongoDB).
- ClientModule: Відповідає за управління клієнтами, їх основними даними та статусами.
- DocumentModule: Реалізує логіку завантаження, збереження та пошуку документів.
- LabelModule: Керує лейблами, що асоціюються з документами і файлами.
- FileModule: Відповідає за управління метаданими завантажених файлів.

Ці модулі взаємодіють через чітко визначені інтерфейси, що підвищує гнучкість і стійкість архітектури щодо змін вимог. Кожен модуль має власні сервіси та контролери, а також сутності (класи), що відображають структуру даних у базі.

Архітектура системи передбачає декілька логічних шарів:

1. Шар даних (Data Layer):
    - Класи-ентіті (Client, DocumentEntity, FileEntity, LabelEntity) описують структуру даних.
    - DocumentRepository або аналогічні інтерфейси можуть бути запроваджені для абстракції доступу до даних.
2. Шар бізнес-логіки (Service Layer):
    - Сервіси (ClientService, DocumentService, LabelService) інкапсулюють логіку роботи із сутностями, забезпечують правила обробки даних, валідацію та виклики до репозиторіїв.

3. Шар презентації (Presentation Layer):
    - Контролери (ClientController, DocumentController, LabelController) приймають HTTP-запити, викликають методи сервісів та формують HTTP-відповіді.
    - На цьому рівні відбувається взаємодія з клієнтським інтерфейсом.
4. Інтеграційний шар (Integration Layer):
    - Може включати EmbeddingService (інтерфейс) та реалізації для отримання ембедінгів, інтеграцію з зовнішніми API.

Окремі підсистеми (наприклад, робота з лейблами чи документами) можна масштабувати або замінювати без впливу на інші частини системи завдяки чіткому розділенню відповідальностей.

DocumentService

Цей сервіс інкапсулює логіку роботи з документами: створення, пошук, оновлення, асоціацію з лейблами. Він використовує репозиторії (через TypeORM) для взаємодії з базою, а також може звертатися до зовнішнього EmbeddingService для семантичного пошуку.

ClientService

ClientService керує операціями над сутностями клієнтів, забезпечуючи логіку реєстрації, оновлення статусу тощо.

LabelService

LabelService дозволяє керувати лейблами і прив’язувати їх до документів та файлів, формуючи семантичні зв’язки.

Таке проєктування архітектури на рівні класів і підсистем забезпечує модульність, гнучкість та простоту підтримки системи. Діаграми ієрархії класів унаочнюють загальну структуру, не перевантажуючи деталями, а детальний опис ключових класів з атрибутами та методами дає чітке уявлення про відповідальності кожної сутності. Завдяки цьому, система залишається стійкою до змін вимог та

легко адаптується до нових викликів, зберігаючи якість коду та логіку обробки даних.

2.8 Проектування бази даних

Під час реалізації системи використано документо-орієнтований підхід до зберігання даних з допомогою MongoDB та ORM TypeORM. Кожна сутність представлена окремим класом, що відображає структуру та властивості об’єкта у базі даних.

Основні сутності (ентітеті) системи, визначені у наданому коді:

1. Client

   Зберігає інформацію про клієнтів системи.

   Основні поля:

   - `_id`: Унікальний ідентифікатор (ObjectId).
   - `name`, `email`, `phone`: Дані клієнта.
   - `status`: Статус клієнта (наприклад, active/inactive).

2. DocumentEntity

   Представляє документ, пов’язаний із певним клієнтом. Вміщує текст, метадані та векторне представлення для семантичного пошуку.

   Основні поля:

   - `_id`: Унікальний ідентифікатор (ObjectId).
   - `client_id`: Ідентифікатор клієнта (string).
   - `title`, `document_name`, `text`: Атрибути документа.
   - `embedding`: Масив чисел для векторної репрезентації змісту.
   - `labels`: Масив ObjectId, що вказує на пов’язані лейбли.

3. FileEntity

   Зберігає метадані про завантажені файли, пов’язані з клієнтом. Основні поля:

   - `_id`: Унікальний ідентифікатор (ObjectId).
   - `client_id`: Ідентифікатор клієнта (ObjectId).
   - `title`, `document_name`: Інформація для ідентифікації файлу.
   - `upload_date`, `update_date`: Дати завантаження та оновлення.
   - `active`: Прапорець активності файлу.
   - `labels`: Масив ObjectId лейблів, пов’язаних із файлом.

4. LabelEntity

   Містить дані про лейбли (мітки), які можуть бути призначені документам та файлам. Основні поля:

   - `_id`: Унікальний ідентифікатор (ObjectId).
   - `name`: Назва лейбла.
   - `embedding`: Масив чисел для векторної репрезентації лейбла.

Зв’язки між сутностями:

Один клієнт може мати багато документів та файлів (через поля `client_id` у DocumentEntity та FileEntity).

Документи та файли можуть бути пов’язані з багатьма лейблами (через масиви `labels`), а один лейбл може належати різним документам та файлам, утворюючи відношення «багато-до-багатьох».

Оскільки MongoDB не підтримує реальні зовнішні ключі на рівні СУБД, зв’язки є логічними. Відповідальність за підтримання цілісності даних покладається на бізнес-логіку системи.

Така структура бази даних забезпечує гнучкість, масштабованість та ефективну роботу з неструктурованими та напівструктурованими даними, дозволяючи швидко здійснювати пошук, класифікацію та керування документами, файлами, лейблами та інформацією про клієнтів.

2.9 Діаграма розгортання застосунку

Аби краще візуалізувати процес розгортання застосунку надано діаграму на котрій зображено даний процес в деталях (рис. 2.5):

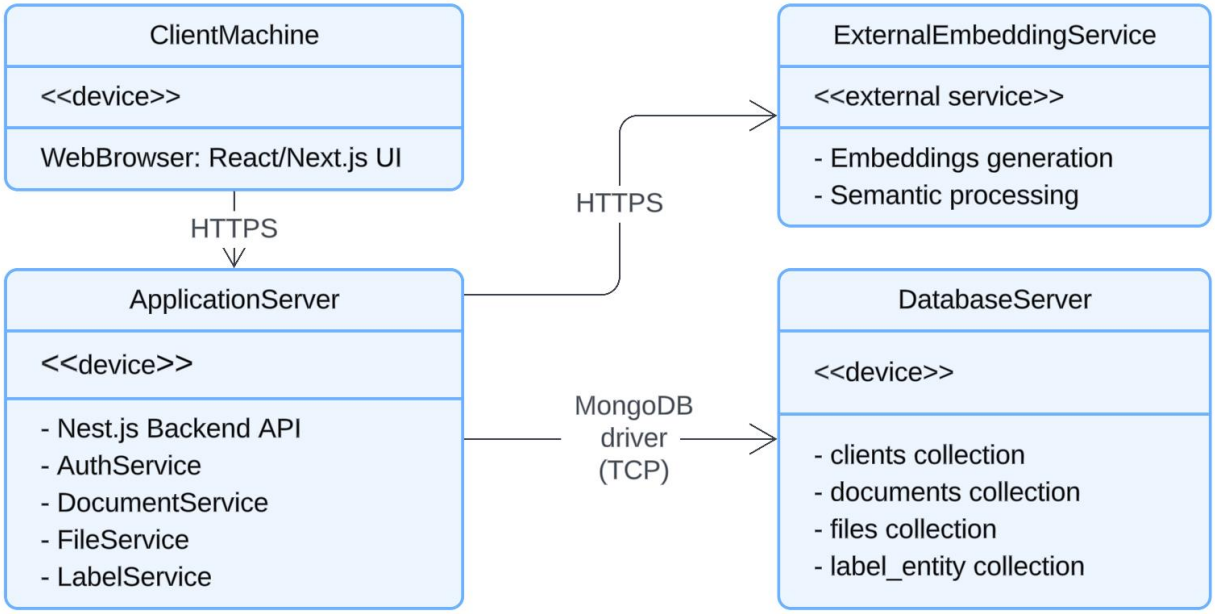

Рисунок 2.6 – Діаграма розгортання застосунку

Client Machine (Web Browser)

Це клієнтський пристрій користувача (ПК, ноутбук, планшет чи смартфон), на якому працює веб-браузер. Інтерфейс користувача реалізовано за допомогою технологій React/Next.js. Користувач взаємодіє з системою через веб-інтерфейс. Запити до серверу здійснюються по протоколу HTTPS.

Application Server (Node.js/Nest.js)

Серверна частина застосунку, що відповідає за виконання бізнес-логіки, авторизацію, обробку документів, взаємодію з базою даних та зовнішніми сервісами. Реалізований на Node.js із використанням фреймворку Nest.js. Виставляє REST, з яким взаємодіє клієнтська частина. Забезпечує доступ до колекцій у MongoDB через MongoDB драйвер (TCP-з’єднання). Звертається до

зовнішнього сервісу ембедінгів (OpenAI API) по HTTPS для отримання векторних репрезентацій тексту.

Database Server (MongoDB)

Сервер бази даних, на якому зберігаються колекції. Дані зберігаються у документно-орієнтованому форматі BSON. Application Server підключається до MongoDB по TCP, використовуючи офіційний драйвер для Node.js. Забезпечує можливість масштабування, реплікації та шардінгу при зростанні обсягів даних.

External Embedding Service (OpenAI API)

Зовнішній сервіс, що надає функціонал отримання ембедінгів для тексту, використовуючи модельні можливості OpenAI. Application Server звертається до нього по HTTPS-запитах, передає текстові фрагменти та отримує назад вектори ембедінгів. Це дозволяє здійснювати семантичний пошук, класифікацію та аналіз текстів на більш високому рівні.

Основні принципи взаємодії:

Користувач через веб-браузер взаємодіє з клієнтським інтерфейсом (React/Next.js), відправляючи запити до Application Server.

Application Server обробляє запити, виконує авторизацію та аутентифікацію, звертається до бази даних, отримує або оновлює дані про документи, файли, лейбли, клієнтів. При необхідності отримати ембедінги, Application Server звертається до зовнішнього сервісу (OpenAI API) та кешує результати для підвищення продуктивності. Application Server повертає результати клієнтському інтерфейсу у вигляді JSON-відповідей, які інтерпретує та відображає React/Next.js UI.

MongoDB забезпечує надійне та гнучке зберігання інформації, тоді як OpenAI API надає інтелектуальні обчислювальні ресурси для аналізу тексту.

Таким чином, діаграма розгортання дозволяє уявити фізичні вузли та їх оточення, визначити протоколи, за якими вони спілкуються, та зрозуміти, як логічна архітектура застосунку інкапсульована у фізичну інфраструктуру.

3. РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір мови програмування та технологій розробки

Під час проектування та розробки веб-застосунку було проаналізовано вимоги до функціональності, продуктивності, масштабованості та зручності подальшого супроводу. На основі проведеного аналізу прийнято рішення застосувати стек технологій, який забезпечує ефективну взаємодію між клієнтською та серверною частинами, а також надійну роботу з базою даних. Основним критерієм відбору було збереження єдиної мови програмування на різних рівнях системи, підтримка об’єктно-орієнтованого підходу, гнучкість у проектуванні архітектури та висока продуктивність.

На стороні клієнта для розробки інтерфейсу користувача було обрано Next.js, що є розширенням фреймворку React. Next.js надає механізми серверного рендерингу та статичної генерації сторінок, покращуючи продуктивність і SEO. Це дозволяє створювати ефективні одно- та багатосторінкові веб-застосунки з динамічною взаємодією між компонентами, швидкою реакцією інтерфейсу без потреби в повному перезавантаженні сторінки. Для реалізації візуальної складової інтерфейсу було використано MUI (Material UI), що прискорює розробку завдяки готовим адаптивним компонентам та уніфікованому стилю, базованому на концепціях Material Design.

Серверну частину реалізовано на платформі Node.js з використанням фреймворку Nest.js, який забезпечує модульну структуру коду, впровадження залежностей, об’єктно-орієнтовану архітектуру та інтеграцію з різноманітними сервісами. Nest.js, натхненний ідеями Angular, дозволяє логічно організувати проєкт у вигляді модулів, контролерів та сервісів, що підвищує підтримуваність і зрозумілість коду. Завдяки цьому вибору вдається зберегти єдність технологічного стеку, оскільки одна й та сама мова (JavaScript/TypeScript) використовується як на стороні клієнта, так і на стороні сервера.

Для збереження та управління даними було обрано MongoDB із хмарною платформою MongoDB Atlas. Такий документо-орієнтований підхід до моделювання даних забезпечує гнучкість у внесенні змін до схеми, просте горизонтальне масштабування та швидкий доступ до інформації. Atlas надає безпечний доступ через HTTPS, а також дозволяє взаємодіяти з даними з різних платформ. Це розв’язує питання безпеки, продуктивності й управління даними, полегшуючи інтеграцію з іншими сервісами.

Окремо слід відзначити використання OpenAI для розширення функціональних можливостей застосунку за рахунок інтелектуальної обробки даних. Інтеграція з моделями штучного інтелекту дає можливість виконувати складні текстові операції, отримувати змістовні відповіді та узагальнення даних безпосередньо в рамках існуючого застосунку. Це дозволяє підвищити рівень взаємодії з користувачем, автоматизувати деякі операції та покращити загальну якість сервісу.

У процесі розробки використовувалося інтегроване середовище Visual Studio Code. Цей редактор коду підтримує численні мови програмування, має розширені можливості автодоповнення (IntelliSense) та розширюється за допомогою плагінів, що дозволяють гнучко налаштовувати робочий простір під специфіку проєкту. Дебагінг-користування, можливість встановлення брейкпойнтів та покрокового виконання коду покращують процес тестування та налагодження застосунку.

Таким чином, вибір мови програмування, інструментарію та технологій розробки був зумовлений прагненням досягти максимальної продуктивності, гнучкості, безпеки та зручності у створенні комплексного веб-застосунку, що включає як клієнтську, так і серверну логіку, а також роботу з даними та елементи штучного інтелекту.

3.2. Розробка функцій об’єктів проектованого програмного забезпечення

У розробленому програмному продукті передбачено комплекс взаємопов’язаних функцій, мета яких – забезпечити ефективну обробку даних від початкового отримання файлів до їхньої повної інтеграції у внутрішню структуру бази даних із подальшою можливістю семантичного пошуку та аналізу. Усі методи згруповані за логікою виконуваних ними завдань і послідовно трансформують вхідні дані у формат, придатний для швидкого, точного та масштабованого використання.

Першим ключовим етапом є зчитування початкових даних, які надходять у формі PDF-файлів. Впроваджений метод отримує шлях до файлу, здійснює верифікацію його доступності та зчитує вміст у двійковому вигляді. Далі за допомогою сторонньої бібліотеки PDF-текст екстрагується з документу. При вдалому виконанні цього кроку програма переходить до формування текстового представлення даних.

Отримавши текст, програма зіштовхується з необхідністю логічної сегментації отриманого контенту. Текст часто є надто об’ємним і нерівномірно структурованим, тому реалізовано методи для розбиття його на менші фрагменти або «чанки». Такий підхід дозволяє керувати обчислювальними ресурсами більш раціонально, а також полегшує подальший пошук. Кожен фрагмент відповідає певному логічному блоку інформації, що сприяє кращому семантичному аналізу.

Наступним кроком є контекстуалізація цих фрагментів. Для підвищення ефективності пошуку та точності відповіді на складні запити, до кожного чанку додається стисле контекстне пояснення. Цей процес реалізовано через звернення до зовнішнього сервісу, який за допомогою моделі генерації тексту може витягнути основний зміст та оформити його у вигляді короткого резюме фрагменту. Як результат, система надалі може ранжувати фрагменти не тільки за ключовими словами, а й за змістовною близькістю.

Отримані фрагменти з контекстом необхідно підготувати до семантичних порівнянь. Для цього застосовується метод отримання векторних репрезентацій (ембедінгів). Кожен фрагмент тексту разом із контекстом передається моделі, яка повертає набір числових значень. Цей вектор відображає семантичний зміст фрагменту в багатовимірному просторі. Використовуючи ембедінги, надалі можна порівнювати ступінь схожості між різними фрагментами або визначати релевантні результати пошуку з урахуванням сенсу, а не просто ключових слів.

Після успішного отримання ембедінгів постає завдання збереження даних для подальшого використання у запитах і пошукових операціях. Тут задіяно методи інтеграції з базою даних, яка, у випадку даного проєкту, є документо-орієнтованою (MongoDB). Для кожного фрагменту з ембедінгом та контекстом створюється відповідний документ, що записується у колекцію. Такий підхід полегшує зберігання й індексацію, а також спрощує масштабування у майбутньому.

Окрім процесу початкового імпорту та розкладу файлів на фрагменти, у програмі реалізовано функції оновлення та видалення даних. Наприклад, при оновленні користувач може замінити або доповнити файл новою версією, після чого система автоматично переобчислює ембедінги та оновлює колекції в базі даних. При видаленні документа забезпечується коректне видалення файла з локальної файлової системи та відповідних документів із бази.

Ключовим аспектом архітектури є модульність та розширюваність. Кожен метод відповідає за конкретну операцію, починаючи від первинного завантаження даних і закінчуючи пошуковими запитами. Такий підхід спрощує тестування та налагодження, оскільки кожну функцію можна перевірити окремо, визначаючи її коректну роботу без впливу інших частин системи. Надалі, якщо виникне потреба у розширенні функціоналу, це можна буде зробити, додаючи нові методи або розширюючи існуючі, зберігаючи чітку структуру та логіку роботи.

Також важливим є те, що завдяки чіткому розмежуванню обов’язків кожної функції, у разі необхідності заміни зовнішніх сервісів або переходу на іншу СУБД, не потрібно переробляти всю систему з нуля. Достатньо внести корективи у

відповідних методах, що взаємодіють із цими сервісами. Це робить проєкт більш стійким до змін технологічного стеку, а також збільшує термін його актуальності.

Отже, сукупність методів проєктованого програмного забезпечення забезпечує повний цикл обробки даних: від читання файлу та його аналізу, через формування контекстних фрагментів і перетворення їх у векторні представлення, до надійного збереження отриманих результатів. Така продумана логічна послідовність дій дозволяє системі ефективно розв’язувати поставлені завдання та гарантує масштабованість, гнучкість і надійність роботи у майбутньому.

3.3 Ілюстрація роботи створеного програмного забезпечення

Для наочної демонстрації функціонування розробленого програмного забезпечення розглянемо типовий сценарій взаємодії користувача із системою. Цей приклад допоможе зрозуміти послідовність дій та результат кожного етапу.

Крок 1: Завантаження документа

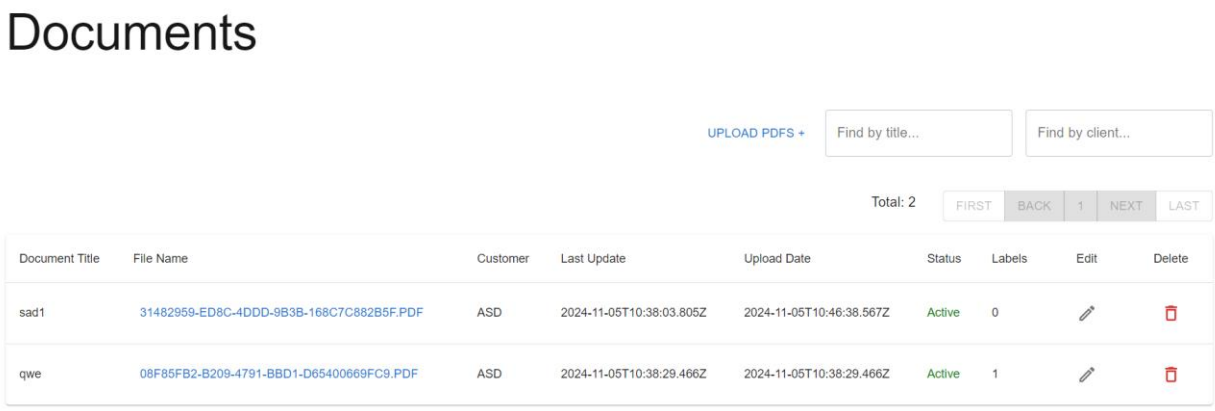

Рисунок 3.1 – Вікно керування документами

Авторизований користувач переходить до сторінки завантаження документів. Наприклад, він обирає PDF-файл зі свого локального сховища. Після надсилання файлу до сервера, система створює запис у колекції files (FileEntity),

фіксуючи метадані — назву, час завантаження, стан активності, та асоціює файл з конкретним клієнтом через client_id. Цей крок забезпечує чітку прив’язку завантажених документів до конкретного користувача.

Крок 2: Семантичний пошук

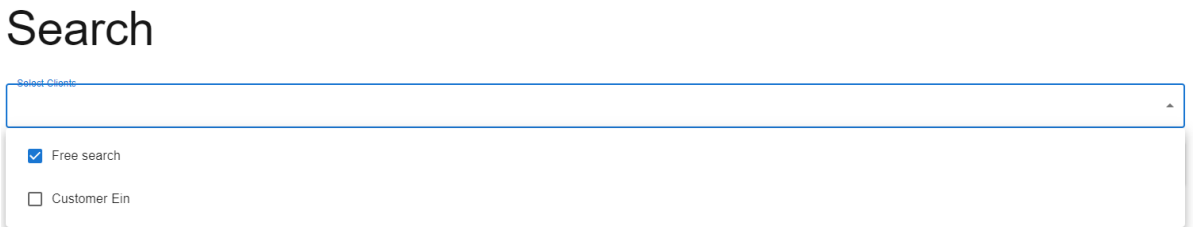

Рисунок 3.2 – Вікно семантичного пошуку по документам

Після завантаження й обробки документів користувач може перейти до сторінки пошуку. Він вводить запит у текстове поле. Система перетворює цей запит на embedding та порівнює його з embedding’ами збережених документів та їхніх фрагментів. Завдяки векторному пошуку користувач отримує результати, які найбільш змістовно відповідають поставленому запиту, а не просто збігаються за ключовими словами. Це дозволяє знайти релевантну інформацію навіть при використанні синонімів чи перефразування.

Крок 3: Перегляд результатів та робота з лейблами

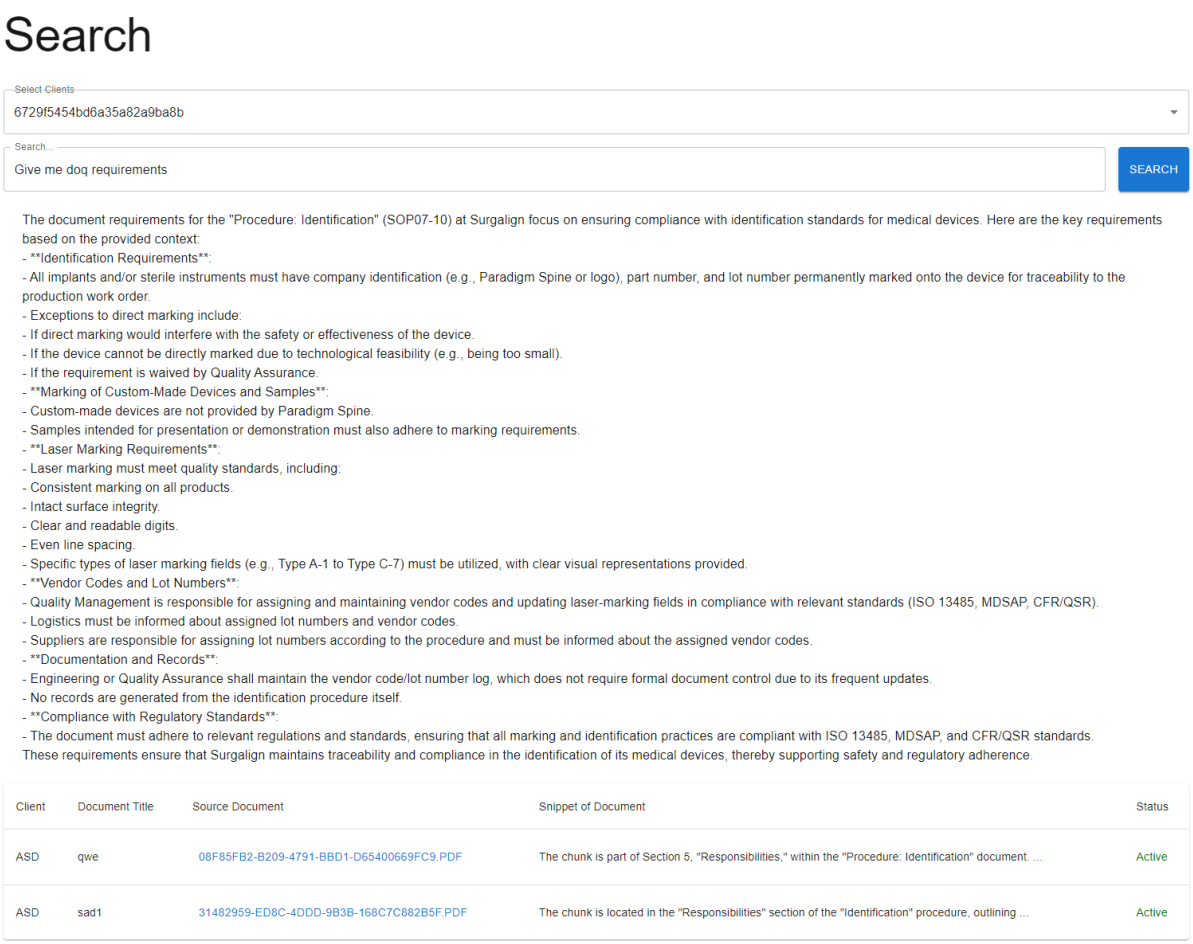

Рисунок 3.3 – Вікно результатів пошуку по документам

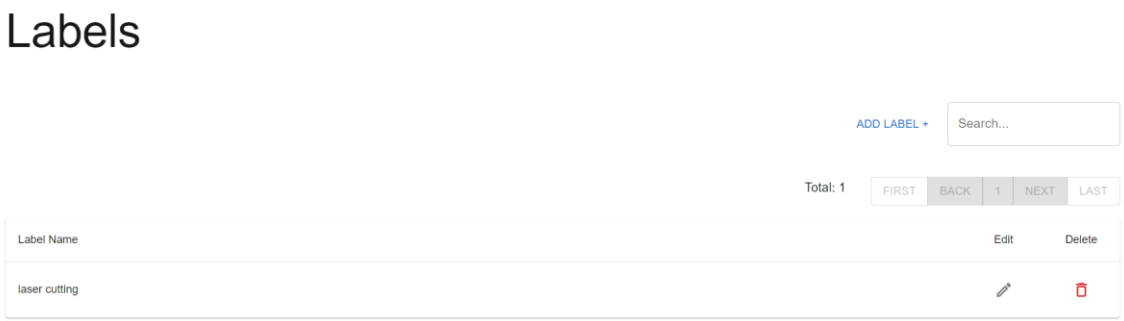

Рисунок 3.4 – Вікно керування лейблами

Система повертає користувачеві список фрагментів документів, упорядкованих за релевантністю. Користувач може переглядати відібрані частини, за потреби переходити до повного тексту документа. Також він може відмічати документи певними лейблами, які зберігаються у колекції label_entity (LabelEntity).

Лейбли допомагають систематизувати інформацію та спрощують подальші пошуки.

Крок 4: Оновлення та керування даними

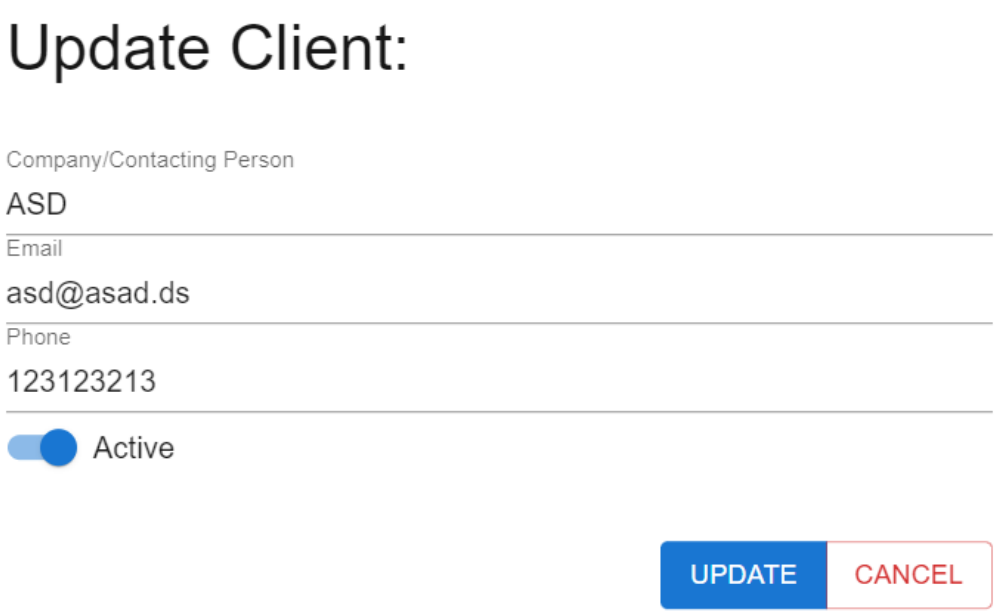

Рисунок 3.5 – Вікно керування даними клієнта

Користувач у будь-який момент може оновити інформацію про документ, змінити його лейбли, деактивувати або видалити файл з бази. Зміни будуть відображені у відповідних колекціях (documents, files, label_entity) та позначені у полях, наприклад, update_date у FileEntity або видалення об’єкта з колекції.

Приклад взаємодії:

Користувач авторизується та бачить свій список файлів.

Завантажує новий файл: система фіксує FileEntity зі станом active = true, встановлює upload_date, пов’язує з client_id. Система автоматично обробляє документ, додає запис до DocumentEntity, генерує embedding, пов’язує потрібні лейбли. Користувач задає семантичний запит, отримує упорядкований список релевантних фрагментів тексту. Обирає найбільш підходящі результати, переглядає документ, додає лейбли через LabelEntity для кращої класифікації.

Ця послідовність дій демонструє, як створене програмне забезпечення інтегрує механізми авторизації, завантаження та обробки документів, семантичного пошуку та керування лейблами, забезпечуючи інтуїтивний та продуктивний досвід користувача.

3.4 Тестування програмного забезпечення та оцінка якості

Тестування є важливим етапом розробки програмного забезпечення, що спрямований на виявлення помилок, перевірку працездатності та оцінку відповідності системи встановленим вимогам. У процесі тестування розроблений веб-застосунок проходить різні види перевірок, які забезпечують упевненість у його стабільності, безпечності та коректній роботі з великими обсягами даних.

Основні види тестування:

Модульне тестування

На початкових етапах кожен модуль або функціональний блок тестується окремо. Це дозволяє швидко виявляти помилки в логіці та локалізувати їх у конкретному компоненті. Наприклад, модуль завантаження документів чи обробки ембедінгів можна протестувати окремо, перевіряючи, чи коректно викликаються методи, чи правильно обробляються вхідні дані.

Інтеграційне тестування

Після успішних модульних перевірок компоненти об’єднуються у більші підсистеми, і тестується їх взаємодія. Наприклад, взаємодія DocumentEntity з FileEntity та LabelEntity, чи коректно DocumentService викликає методи DocumentRepository, чи правильно відповідає система на запити завантаження та пошуку документів. Інтеграційні тести дають змогу виявити проблеми, що виникають на стиках між окремими блоками.

Системне тестування

На цьому етапі перевіряється повна реалізація програмного продукту в реальних умовах експлуатації. Тестування охоплює клієнтський інтерфейс, серверну логіку, роботу з базою даних. Перевіряється працездатність сценаріїв: авторизація, завантаження документів, семантичний пошук, перегляд результатів, керування лейблами, оновлення метаданих.

Регресійне тестування

Після внесення змін у код чи додавання нового функціоналу повторно виконується регресійне тестування. Воно гарантує, що нові модифікації не вплинули негативно на вже реалізовані, раніше протестовані можливості.

Тестування продуктивності та навантаження

Перевіряється, як система поводиться за значної кількості одночасних користувачів чи обробки великого обсягу документів. Аналізується час відгуку, здатність системи витримувати високі навантаження без помітного зниження продуктивності чи відмов.

Інструменти та підходи до тестування:

- Юніт-тестування: Використання фреймворків (наприклад, Jest для Node.js) для автоматизації тестів логіки окремих класів та методів.
- Інтеграційні та системні тести: Застосування інструментів для автоматизації тестування API (Postman, Newman) або енд-ту-енд тестів інтерфейсу (Cypress, Playwright).
- Моніторинг та логування: Використання інструментів логування та аналітики для оцінки стабільності системи під час тестів, що особливо важливо при тестуванні продуктивності.

Оцінка якості:

Якість програмного забезпечення оцінюється за такими критеріями:

- Функціональність: Наскільки система відповідає вимогам і виконує заявлені операції (завантаження, пошук, керування лейблами, робота з клієнтами).

- Надійність: Відсутність критичних помилок, стабільна робота при тривалому використанні та при високому навантаженні.
- Продуктивність: Час відгуку на запити користувачів, можливість обробляти великі обсяги даних без суттєвого погіршення швидкості.
- Зручність використання: Інтуїтивний інтерфейс, наявність підказок та зрозумілої навігації.
- Масштабованість: Можливість легко додавати новий функціонал, розширювати обсяг оброблюваних даних або кількість користувачів без істотних змін архітектури.

Таким чином, комплексний підхід до тестування та ретельна оцінка якості забезпечують упевненість у працездатності системи, її готовності до експлуатації та здатності задовольнити потреби кінцевих користувачів.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Охорона праці

На етапі розробки та впровадження програмного забезпечення для автоматизованого управління документами, що обробляє великі обсяги інформації, надзвичайно важливо забезпечити такі умови праці, за яких користувачі системи матимуть безпечне, зручне та продуктивне робоче середовище. У випадку даного кваліфікаційного проєкту йдеться про створення системи, яка дозволяє здійснювати пошук, класифікацію та обробку документів, а також взаємодію між персоналом і комп'ютерними ресурсами. Тому, від самого початку розробки потрібно враховувати нормативно-правові вимоги у сфері охорони праці, аби підсумкове рішення відповідало діючим державним стандартам та забезпечувало захист здоров'я працівників.

Зокрема, в умовах експлуатації системи документного обігу оператори, інженери, адміністратори й інші фахівці повинні мати можливість ефективно виконувати свої завдання без загроз для їхнього здоров'я. Це включає не лише дотримання ергономічних вимог до розміщення робочих місць і обладнання, але й забезпечення належного освітлення, оптимальних мікрокліматичних умов, низького рівня шуму та інших факторів, що впливають на самопочуття людини. Крім того, врахування протипожежних та електробезпекових норм є критично важливим для запобігання нещасним випадкам, пов'язаним із використанням електрообладнання, систем живлення та інформаційно-комунікаційної інфраструктури.

При проєктуванні та впровадженні програмного забезпечення обов'язково слід дотримуватись нормативних вимог із охорони праці, техніки безпеки та протипожежної безпеки. Це забезпечує створення безпечних і комфортних умов праці для користувачів системи, а також знижує ризики професійних захворювань,

травматизму та негативного впливу виробничого середовища на здоров’я працівників.

Основу законодавчого регулювання в цій сфері складають Кодекс законів про працю України, зокрема частина 2 ст. 2 та частина 1 ст. 21, що закріплюють обов’язки роботодавця зі створення належних умов праці. Крім того, вимоги щодо безпечних умов праці встановлено в Законі України «Про охорону праці» (ст. 13), який визначає права працівників на безпечне та здорове робоче середовище, а також обов’язки роботодавця в цій сфері.

При створенні та використанні програмного продукту враховуються чинні державні санітарні норми та правила, зокрема ДСанПіН 3.3.2.007-98, які регулюють умови праці з використанням персональних ЕОМ, освітлення, мікроклімат, рівні шуму тощо [14]. Дотримання цих норм сприяє збереженню зору, зниженню втомлюваності та підвищенню продуктивності працівників.

Згідно з ДСТУ 7234:2011, конструкція робочого місця повинна забезпечувати можливість регулювання його параметрів відповідно до антропометричних характеристик працівника [15]. Елементи робочого місця мають бути розташовані так, щоб працівник міг виконувати завдання з мінімальним фізичним навантаженням. Відповідно до п. 4.3 ДСанПіН 3.3.2.007-98, робоче місце слід розміщувати таким чином, аби природне світло падало зліва, що покращує умови освітлення та знижує напруження зору [14].

При проєктуванні приміщень, де працюватимуть користувачі системи, слід дотримуватись норм освітлення, забезпечуючи рівномірне чи комбіноване освітлення робочих зон [14]. Розташування світильників згідно з вимогами ДСанПіН допомагає уникнути відблисків та прямого попадання світла в очі. Шумові характеристики робочого середовища повинні відповідати діючим стандартам, що допоможе підтримувати високу продуктивність праці та мінімізує негативний вплив на здоров’я працівників.

Положення про пожежну безпеку, зокрема НАПБ А.01.001-2014, визначають загальні вимоги до протипожежного захисту [16]. У приміщеннях, де

розміщено комп’ютерне обладнання та сервери, необхідно дотримуватися правил пожежної безпеки: забезпечувати наявність первинних засобів пожежогасіння (вогнегасників), позначати шляхи евакуації та сигнальні системи оповіщення. Електрообладнання повинно експлуатуватися згідно з нормами електробезпеки та заземлення.

Всі електроприлади та комп’ютерна техніка повинні бути перевірені на предмет справності та безпечності. Забороняється використання пошкоджених кабелів, перевантаження електромережі та експлуатація невідповідних за потужністю подовжувачів чи розгалужувачів. Робочі місця операторів повинні бути організовані з урахуванням ергономічних принципів, що покращує самопочуття працівників та знижує ймовірність професійних захворювань опорно-рухового апарату.

Окрім інженерно-технічних рішень, важливою складовою є інструктажі та навчання працівників з питань охорони праці, електробезпеки, пожежної безпеки та правил поведінки в аварійних і надзвичайних ситуаціях. Роботодавець має організовувати вступні та періодичні інструктажі, перевірки знань, а також забезпечити доступність інструкцій та плакатів з безпеки.

Таким чином, запропоновані в кваліфікаційній роботі рішення зі створення та впровадження програмного забезпечення відповідають законодавчим та нормативно-правовим вимогам в галузі охорони праці, техніки безпеки та протипожежної безпеки. Виконання рекомендацій сприяє створенню безпечного та комфортного робочого середовища для користувачів, знижує ризики професійних захворювань та виробничого травматизму, а також забезпечує стабільну та ефективну роботу системи.

4.2 Безпека життєдіяльності

У сучасних умовах надзвичайних ситуацій воєнного часу надзвичайно важливим є забезпечення безпеки користувачів при роботі з інформаційними системами, зокрема системами документообігу. Це стосується як фізичного захисту людей та обладнання, так і інформаційної безпеки. Специфіка воєнного часу вимагає особливої уваги до планування дій, зменшення ризиків та наслідків від можливих атак на інфраструктуру (енергетичну, телекомунікаційну), забезпечення безперервності роботи та швидкого відновлення в разі порушення функціонування системи.

Як зазначено у методичних посібниках з безпеки в надзвичайних ситуаціях, важливою складовою є розробка планів евакуації, організації колективних і індивідуальних засобів захисту, а також формування навичок поведінки під час загроз безпеці [17]. Для користувачів системи документообігу це може означати, що вони повинні знати, як діяти в разі припинення електропостачання, збою інтернет-зв’язку, кібернетичних атак або фізичного пошкодження серверного обладнання. Необхідно передбачити резервні канали зв’язку, генератори живлення, можливість швидкого переходу на альтернативні сервери чи дата-центри, а також засоби захисту даних — шифрування, резервне копіювання та регулярне оновлення програмного забезпечення.

Під час воєнних загроз важливе значення має захист місць розташування серверного обладнання та забезпечення фізичного доступу до нього лише уповноваженому персоналу. За рекомендаціями експертів із цивільної безпеки, слід здійснювати періодичний аудит захищеності приміщень та дотримуватись вимог фортифікації (наприклад, зміцнення будівель, наявність укриттів, систем відеоспостереження) [18]. Додатково потрібен комплексний моніторинг стану мережевих ресурсів і системи документообігу, щоб оперативно виявляти ознаки несанкціонованого доступу або кібератак.

Іншим аспектом безпеки є підготовка користувачів. Вони повинні знати інструкції щодо дій під час сигналу тривоги, розуміти план евакуації, володіти базовими знаннями надання першої домедичної допомоги у разі поранень.

Організація навчальних тренувань, інструктажів і поширення інформаційних матеріалів сприяє зниженню ризиків та мінімізації втрат (як людських, так і матеріальних).

Таким чином, забезпечення безпеки в надзвичайних ситуаціях воєнного часу для користувачів системи документообігу передбачає комплекс заходів, включаючи захист інфраструктури, інформаційних ресурсів, навчання персоналу та підготовку до оперативних дій у разі критичних інцидентів. Ці заходи підвищують живучість системи, знижують ризик втрати даних та пошкодження обладнання, а також сприяють збереженню життя і здоров’я людей.

ВИСНОВКИ

У результаті виконаної роботи вдалося побудувати комплексний клієнт-серверний застосунок, що забезпечує повний цикл обробки даних — від початкового отримання документів у форматі PDF до їхнього семантичного аналізу та ефективного пошуку. Застосовані технології (Next.js, Nest.js, MongoDB, OpenAI, MUI) та фреймворки дозволили поєднати прозору архітектуру клієнтської частини з продуктивною та гнучкою серверною логікою. Завдяки цьому система стала більш масштабованою та здатною до подальшого розширення функціоналу.

Реалізовані методи дозволяють послідовно зчитувати файли, перетворювати їх у текст, логічно розділяти на фрагменти, отримувати векторні репрезентації (ембедінги) та зберігати ці дані в документо-орієнтованій базі даних. Такий підхід забезпечив можливість семантичного пошуку, який не обмежується лише ключовими словами, а орієнтується на змістовну близькість фрагментів. Це поліпшує точність і корисність результатів для кінцевого користувача.

Використання модульної архітектури, дотримання принципів об’єктно-орієнтованого підходу та розумне розмежування обов’язків між методами спростили налагодження, тестування та майбутню підтримку коду. За потреби можна легко замінювати зовнішні сервіси, масштабувати інфраструктуру або оновлювати логіку без ризику зламати інші частини системи.

Таким чином, побудований програмний продукт продемонстрував свою життєздатність, забезпечивши ефективну обробку та індексацію великих обсягів даних, надавши зручні засоби їхнього пошуку та аналізу. Отриманий результат дає підґрунтя для подальших експериментів, вдосконалень та інтеграцій з іншими сервісами, задовольняючи як сучасні потреби користувачів, так і майбутні виклики розвитку інформаційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Stefanyshyn, I. , Pastukh, O., Stefanyshyn, V. , Baran, I. , Boyko, I. Robustness of AI algorithms for neurocomputer interfaces based on software and hardware technologies CEUR Workshop Proceedings, 2024, 3742
2. M. Fowler, Patterns of enterprise application architecture. Boston, Mass. ; Munich: Addison-Wesley, 2015.
3. R. S. Pressman, Software Engineering: A Practitioner’s Approach. McGraw-Hill Science, Engineering & Mathematics, 2010.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Pearson Education, 1994.
5. IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE, 2000.
6. Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, 2004.
7. NestJS Official Documentation: https://docs.nestjs.com/
8. Node.js Documentation: https://nodejs.org/en/docs/
9. MongoDB Documentation: https://www.mongodb.com/docs/
10. OpenAI API Documentation: https://platform.openai.com/docs/introduction
11. React Official Documentation: https://reactjs.org/docs/getting-started.html
12. Next.js Official Documentation: https://nextjs.org/docs
13. Dean, J., Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters.” Commun. ACM 51, 1 (January 2008), 107–113
14. “Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин,” Офіційний вебпортал парламенту України. https://zakon.rada.gov.ua/rada/show/v0007282-98#Text[

15. “ДСТУ 7234:2011 Дизайн і ергономіка. Обладнання виробниче. Загальні вимоги дизайну та ергономіки,” Budstandart.com, 2022. https://online.budstandart.com/ua/catalog/doc-page?id_doc=59893
16. “Про затвердження Правил пожежної безпеки в Україні,” Офіційний вебпортал парламенту України, 2014. https://zakon.rada.gov.ua/laws/show/z0252-15#Text
17. Стручок В.С. Безпека в надзвичайних ситуаціях: методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання. Тернопіль: ФОП Паляниця В. А., 156 с. Отримано з https://elartu.tntu.edu.ua/handle/lib/39196.
18. Стручок В.С. Техноекологія та цивільна безпека. Частина «Цивільна безпека»: навчальний посібник. Тернопіль: ФОП Паляниця В. А., 156 с. Отримано з http://elartu.tntu.edu.ua/handle/lib/39424.

ДОДАТКИ

ДОДАТОК А – Тези

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
імені ІВАНА ПУЛЮЯ
НАУКОВЕ ТОВАРИСТВО ім. ШЕВЧЕНКА

М А Т Е Р І А Л И

XII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

18–19 грудня 2024 року

УДК 004.41
В. Ямко; М. Петрик, д.ф.-м.н., професор
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ВПРОВАДЖЕННЯ ВЕКТОРНОГО ПОШУКУ У СИСТЕМІ ДОКУМЕНТООБІГУ

UDC 004.41
V. Yamko; M. Petryk, Doctor of Physical and Mathematical Sciences, Professor

IMPLEMENTATION OF VECTOR SEARCH IN A DOCUMENT MANAGEMENT SYSTEM

Системи управління документами пройшли довгий шлях від фізичних архівів до складних цифрових рішень. Сьогодні вони активно використовуються у багатьох сферах для зберігання, пошуку та аналізу даних. Однією з важливих сучасних тенденцій у цій галузі є інтеграція штучного інтелекту, зокрема методів роботи з векторними просторами та контекстного аналізу тексту.

Звичайний пошук за ключовими словами у текстових полях, таких як назва документа чи ім'я клієнта, має суттєві обмеження. Він часто не враховує семантичну близькість слів. Наприклад, запити «договір оренди» та «контракт на оренду» можуть бути релевантними, але традиційні методи пошуку цього не розуміють.

Для розв'язання цієї проблеми використовуються текстові ембедінги – числові векторні представлення тексту, які зберігають його семантичний зміст. Завдяки алгоритмам, як-от Word2Vec, GloVe, або сучасним моделям типу BERT і GPT, кожен текстовий фрагмент можна представити у вигляді багатовимірного вектора [1]. Векторні пошуки дозволяють знаходити документи за їх схожістю у векторному просторі, навіть якщо слова у запиті не повністю збігаються зі словами у документі.

Наприклад, запит може бути перетворений на вектор, який порівнюється із векторами, що представляють документи у базі. Це відкриває нові можливості для гнучкого пошуку:

- Розуміння синонімів та контексту.
- Пошук релевантних документів, навіть якщо формулювання у запиті відрізняється.
- Аналіз зв’язків між документами для рекомендації пов’язаних матеріалів.

У випадках, коли документ містить великий обсяг тексту, наприклад, десятки сторінок, пряме використання ембедінгів для всього тексту може бути неефективним. Виникає ризик втрати важливої інформації або збільшення обчислювальної складності.

Контекстне чанкування є підходом, який дозволяє розбивати великий текст на менші логічні частини (чанки), кожна з яких аналізується окремо [2]. Наприклад, документ можна поділити за абзацами, розділами або навіть реченнями, залежно від поставленої задачі.

Чанки обробляються за допомогою моделей, які створюють ембедінги для кожного фрагмента тексту. Це дозволяє:

- Проводити більш точний пошук, фокусуючись лише на релевантних частинах документа.
- Підвищувати ефективність обчислень за рахунок паралельної обробки фрагментів.
- Зберігати контекстні зв’язки між частинами документа, що важливо для відновлення повної картини під час пошуку.

Для зберігання та пошуку векторів у реальному часі використовуються спеціалізовані рішення, такі як Pinecone, Weaviate або MongoDB Atlas Search [3]. Вони підтримують:

- Зберігання багатовимірних векторів у базах даних.
- Пошук найближчих сусідів (nearest neighbor search) для швидкого знаходження релевантних записів.
- Інтеграцію з традиційними SQL або NoSQL системами для гібридного підходу до пошуку.

Такі системи можуть стати незамінними у великих організаціях, де пошук і аналіз документів є ключовою задачею. Завдяки ембедінгам і контекстному чанкуванню досягається новий рівень точності та швидкості обробки інформації.

Література

1. Hugging Face Transformers [Електронний ресурс] – Режим доступу до ресурсу: https://huggingface.co/docs/transformers/index
2. LangChain Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://python.langchain.com/docs/introduction/
3. MongoDB Atlas Search Documentation [Електронний ресурс] – Режим доступу: https://www.mongodb.com/docs/atlas/search/.

ДОДАТОК Б – Лістинг коду інформаційної системи

Лістинг коду - main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';  202.7k (gzipped: 42.7k)
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.useGlobalPipes(new ValidationPipe({ transform: true }));
  app.setGlobalPrefix('api');
  app.enableCors();

  // Swagger configuration
  const config = new DocumentBuilder()
    .setTitle('Document API')
    .setDescription('API for managing documents')
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api/docs', app, document);

  await app.listen(process.env.PORT);
}
bootstrap();
```

Лістинг коду – app.module.ts

```
import { Module } from '@nestjs/common';  202.7k (gzipped: 42.7k)
import { DocumentModule } from './document/document.module';
import { DatabaseModule } from './database/database.module';
import { ConfigModule } from '@nestjs/config';  32.2k (gzipped: 9.8k)
import { ClientModule } from './client/client.module';
import { LabelModule } from './label/label.module';

You, 2 months ago | 1 author (You)
@Module({
  imports: [
    ConfigModule.forRoot({ isGlobal: true }),
    DocumentModule,
    DatabaseModule,
    ClientModule,
    LabelModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

Лістинг коду – label.module.ts

```
import { Module } from '@nestjs/common';  202.7k (gzipped: 42.7k)
import { LabelService } from './label.service';
import { LabelController } from './label.controller';
import { TypeOrmModule } from '@nestjs/typeorm';  129.6k (gzipped: 28.2k)
import { LabelEntity } from 'src/database/entities/label.entity';
import { DocumentEntity } from 'src/database/entities/document.entity';

You, 2 months ago | 1 author (You)
@Module({
  imports: [TypeOrmModule.forFeature([LabelEntity, DocumentEntity])],
  controllers: [LabelController],
  providers: [LabelService],
  exports: [LabelService],
})
export class LabelModule {}
```

Лістинг коду - main.ts

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  Patch,
  Post,
  Query,
} from '@nestjs/common';
import {
  ApiOperation,
  ApiParam,
  ApiQuery,
  ApiResponse,
  ApiTags,
} from '@nestjs/swagger';
import { LabelService } from './label.service';
import { UpdateLabelDto } from './dto/update-label.dto';
import { CreateLabelDto } from './dto/create-label.dto';
import { PaginationResultDto } from 'src/common/dto/pagination-result.dto';
import { GetLabelsDto, LabelDto } from './dto/get-label.dto';

...
@ApiTags('label')
@Controller('label')
export class LabelController {
  constructor(private readonly labelService: LabelService) {}

  /**
   * Creates a new label.
   * @param createLabelDto Data transfer object for creating a label.
   * @returns The created label.
   */
  @Post()
  @ApiOperation({ summary: 'Create a new label' })
  @ApiResponse({ status: 201, description: 'Label created successfully' })
  create(@Body() createLabelDto: CreateLabelDto) {
    return this.labelService.create(createLabelDto);
  }
```

```
/**
 * Retrieves all labels with pagination support, optionally filtered by label name.
 * @param page The page number.
 * @param limit The number of items per page.
 * @param name The label name to search for.
 * @returns A paginated result containing the list of labels.
 */
@Get()
@ApiOperation({
  summary: 'Get all labels with pagination and search by name',
})
@ApiQuery({
  name: 'page',
  required: false,
  description: 'Page number',
  type: Number,
})
@ApiQuery({
  name: 'limit',
  required: false,
  description: 'Number of items per page',
  type: Number,
})
@ApiQuery({
  name: 'name',
  required: false,
  description: 'Label name to search for',
  type: String,
})
@ApiResponse({ status: 200, description: 'List of labels' })
async findAll(
  @Query() query: GetLabelsDto,
): Promise<PaginationResultDto<LabelDto>> {
  const { page, limit, name } = query;
  const data = await this.labelService.findAll(page, limit, name);
  return new PaginationResultDto<LabelDto>(data);
}
```

```
  /**
   * Retrieves a label by its ID.
   * @param id The ID of the label.
   * @returns The found label.
   */
  @Get(':id')
  @ApiOperation({ summary: 'Get a label by id' })
  @ApiParam({ name: 'id', description: 'Label id' })
  @ApiResponse({ status: 200, description: 'Label retrieved successfully' })
  findOne(@Param('id') id: string) {
    return this.labelService.findOne(id);
  }

  /**
   * Updates a label.
   * @param id The ID of the label to update.
   * @param updateLabelDto Data transfer object for updating a label.
   * @returns The updated label.
   */
  @Patch(':id')
  @ApiOperation({ summary: 'Update a label' })
  @ApiParam({ name: 'id', description: 'Label id' })
  @ApiResponse({ status: 200, description: 'Label updated successfully' })
  update(@Param('id') id: string, @Body() updateLabelDto: UpdateLabelDto) {
    return this.labelService.update(id, updateLabelDto);
  }

  /**
   * Deletes a label if it's not associated with any document.
   * @param id The ID of the label to delete.
   */
  @Delete(':id')
  @ApiOperation({ summary: 'Delete a label' })
  @ApiParam({ name: 'id', description: 'Label id' })
  @ApiResponse({ status: 200, description: 'Label deleted successfully' })
  remove(@Param('id') id: string) {
    return this.labelService.remove(id);
  }
}
```

Лістинг коду –label.service.ts

```
import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { ObjectId } from 'mongodb';
import { DocumentEntity } from 'src/database/entities/document.entity';
import { LabelEntity } from 'src/database/entities/label.entity';
import { MongoRepository } from 'typeorm';
import { CreateLabelDto } from './dto/create-label.dto';
import { UpdateLabelDto } from './dto/update-label.dto';
import { ConfigService } from '@nestjs/config';
import axios from 'axios';
import { LabelDto } from './dto/get-label.dto';

You, 2 months ago | 1 author (You) | Complexity is 6 It's time to do something...
@Injectable()
export class LabelService {
  constructor(
    @InjectRepository(LabelEntity)
    private readonly labelRepository: MongoRepository<LabelEntity>,
    @InjectRepository(DocumentEntity)
    private readonly documentRepository: MongoRepository<DocumentEntity
    protected readonly configService: ConfigService,
  ) {}

  /**
   * Creates a new label.
   * @param createLabelDto Data transfer object for creating a label.
   * @returns The created label.
   */
  async create(createLabelDto: CreateLabelDto): Promise<LabelEntity> {
    const label = new LabelEntity();
    label.name = createLabelDto.name;
    label.embedding = await this.generateEmbedding(label.name);
    return this.labelRepository.save(label);
  }
```

```
/**
 * Retrieves all labels with pagination and optional search by name.
 * @param page The page number.
 * @param limit The number of items per page.
 * @param name The label name to search for.
 * @returns A paginated result containing the list of labels.
 */
Complexity is 5 Everything is cool!
async findAll(
  page: number,
  limit: number,
  name?: string,
): Promise<{
  data: LabelDto[];
  total: number;
  page: number;
  lastPage: number;
}> {
  const skip = (page - 1) * limit;

  const query: any = {};

  if (name) {
    query.name = { $regex: name, $options: 'i' };
  }

  const [labels, total] = await this.labelRepository.findAndCount({
    where: query,
    skip,
    take: limit,
    order: { name: 1 }, // Optional: sort by name
    select: ['_id', 'name'], // Exclude 'embedding' from the selection
  });

  const lastPage = Math.ceil(total / limit);

  // Transform labels to DTOs
  const data = labels.map((label) => label);

  return { data, total, page, lastPage };
}
```

```
/**
 * Retrieves a label by its ID.
 * @param id The ID of the label.
 * @returns The found label.
 * @throws NotFoundException if the label is not found.
 */
```

Complexity is 4 Everything is cool!

```
async findOne(id: string): Promise<LabelEntity> {
  const label = await this.labelRepository.findOneBy({
    _id: new ObjectId(id),
  });
  if (!label) {
    throw new NotFoundException(`Label with id ${id} not found`);
  }
  return label;
}

/**
 * Updates a label.
 * @param id The ID of the label to update.
 * @param updateLabelDto Data transfer object for updating a label.
 * @returns The updated label.
 */
```

Complexity is 4 Everything is cool!

```
async update(
  id: string,
  updateLabelDto: UpdateLabelDto,
): Promise<LabelEntity> {
  const label = await this.findOne(id);
  if (updateLabelDto.name && updateLabelDto.name !== label.name) {
    label.name = updateLabelDto.name;
    label.embedding = await this.generateEmbedding(label.name);
  }
  return this.labelRepository.save(label);
}
```

```
/**
 * Deletes a label if it's not associated with any document.
 * @param id The ID of the label to delete.
 * @throws BadRequestException if the label is associated with any document.
 */
```

Complexity is 3 Everything is cool!

```
async remove(id: string): Promise<void> {
  const label = await this.findOne(id);

  // Check if any DocumentEntity has this label
  const count = await this.documentRepository.countBy({
    labels: new ObjectId(id),
  });
  console.log({ count });

  if (count > 0) {
    throw new BadRequestException(
      'Cannot delete label because it is associated with documents',
    );
  }

  await this.labelRepository.delete(label._id);
}
```

```
  /**
   * Generates an embedding for the given text using the OpenAI API.
   * @param text The text to generate an embedding for.
   * @returns The embedding vector.
   */
  Complexity is 6 It's time to do something...
  private async generateEmbedding(text: string): Promise<number[]> {
    try {
      const response = await axios.post(
        this.configService.getOrThrow('EMBEDDING_URL'),
        {
          input: text,
          model: this.configService.getOrThrow('EMBEDDING_MODEL'),
        },
        {
          headers: {
            'Content-Type': 'application/json',
            Authorization: `Bearer ${this.configService.getOrThrow(
              'OPENAI_API_KEY',
            )}`,
          },
        },
      );
      return response.data.data[0].embedding;
    } catch (error) {
      console.error(
        'Error generating embedding:',
        error.response?.data?.error || error.message,
      );
      return [];
    }
  }
}
```

Лістинг коду – document.module.ts

```ts
import { Module } from '@nestjs/common';  202.7k (gzipped: 42.7k)
import { DocumentController } from './document.controller';
import { DocumentService } from './document.service';
import { TypeOrmModule } from '@nestjs/typeorm';  129.6k (gzipped: 28.2k)
import { DocumentEntity } from 'src/database/entities/document.entity';
import { ClientModule } from 'src/client/client.module';
import { FileEntity } from 'src/database/entities/file.entity';
import { DatabaseModule } from 'src/database/database.module';
import { LabelEntity } from 'src/database/entities/label.entity';

/**
 * The `DocumentModule` is responsible for handling document-related functionalities.
 * It imports necessary modules, registers controllers, and provides services required for document operations.
 */
You, 2 months ago | 1 author (You)
@Module({
  imports: [
    TypeOrmModule.forFeature([DocumentEntity, FileEntity, LabelEntity]),
    ClientModule,
    DatabaseModule,
  ],
  controllers: [DocumentController],
  providers: [DocumentService],
})
export class DocumentModule {}
```

Лістинг коду – document.controller.ts

```
import {
  BadRequestException,
  Body,
  Controller,
  Delete,
  Get,
  Header,
  Param,
  Post,
  Query,
  Res,
  StreamableFile,
  UploadedFile,
  UploadedFiles,
  UseInterceptors,
} from '@nestjs/common';
import { DocumentService } from './document.service';
import { FileInterceptor, FilesInterceptor } from '@nestjs/platform-express';
import { multerOptions } from './config/multer.config';
import { Response } from 'express';
import { GetFilesDto, PaginationResultDto } from './dto/get-files.dto';
import { UploadDocumentDto } from './dto/upload-document.dto';
import { UpdateDocumentDto } from './dto/update-document.dto';
import {
  ApiBody,
  ApiConsumes,
  ApiOperation,
  ApiParam,
  ApiQuery,
  ApiResponse,
  ApiTags,
} from '@nestjs/swagger';

/**
 * Controller responsible for handling document-related HTTP requests,
 * such as uploading, downloading, updating, and deleting PDF documents.
 */
```

```typescript
@ApiTags('document')
@Controller('document')
export class DocumentController {
  /**
   * Constructs a new instance of the DocumentController.
   * @param documentService Service for handling document operations.
   */
  constructor(protected readonly documentService: DocumentService) {}

  /**
   * Uploads multiple PDF files for a specific client.
   * @param clientId The ID of the client.
   * @param uploadDocumentDto Data transfer object containing document titles.
   * @param files Array of uploaded PDF files.
   * @returns A message indicating successful upload and processing.
   * @throws BadRequestException if no files are uploaded.
   */
  Complexity is 7 It's time to do something...
  @Post('upload/:clientId')
  @UseInterceptors(FilesInterceptor('files', 10, multerOptions))
  @ApiOperation({ summary: 'Upload multiple PDFs for a client' })
  @ApiParam({ name: 'clientId', description: 'Client ID' })
  @ApiBody({ type: UploadDocumentDto })
  @ApiConsumes('multipart/form-data')
  @ApiResponse({ status: 201, description: 'PDFs uploaded successfully' })
  @ApiResponse({ status: 400, description: 'No files uploaded' })
  async uploadPDFs(
    @Param('clientId') clientId: string,
    @Body() uploadDocumentDto: UploadDocumentDto,
    @UploadedFiles() files: Express.Multer.File[],
  ): Promise<Record<string, any>> {
    if (!files || files.length === 0) {
      throw new BadRequestException('No files uploaded');
    }

    files.forEach(async (file, index) => {
      const pdfPath = file.path;
      await this.documentService.processPDF(
        pdfPath,
        clientId,
        file.originalname,
```

```
      uploadDocumentDto.titles[index].trim(),
    );
  });
  return { message: 'PDFs uploaded and processed successfully' };
}

/**
 * Generates a summary for a client based on the provided input string.
 * @param clientId The ID of the client.
 * @param inputString The input string to generate the summary.
 * @returns The summary output.
 */
@Post('summary')
@ApiOperation({
  summary:
    'Get summary or answer any requirement-specific question for one user, multiple users, or across all users',
})
@ApiBody({
  schema: {
    type: 'object',
    properties: {
      clientIds: { type: 'array', items: { type: 'string' }, nullable: true },
      inputString: { type: 'string' },
    },
  },
})
@ApiResponse({ status: 200, description: 'Answer generated' })
async getSummary(
  @Body('clientIds') clientIds: string[] | null,
  @Body('inputString') inputString: string,
) {
  const output = await this.documentService.createFinalOutput(
    clientIds,
    inputString,
  );
  return output;
}
```

```
/**
 * Downloads a PDF document for a specific client.
 * @param clientId The ID of the client.
 * @param documentName The name of the document to download.
 * @param res The HTTP response object.
 * @returns A streamable file containing the PDF.
 */
@Get('download/:clientId/:documentName')
@Header('Content-Type', 'application/pdf')
@ApiOperation({ summary: 'Download a PDF document' })
@ApiParam({ name: 'clientId', description: 'Client ID' })
@ApiParam({ name: 'documentName', description: 'Document name' })
@ApiResponse({ status: 200, description: 'PDF downloaded' })
async downloadPDF(
  @Param('clientId') clientId: string,
  @Param('documentName') documentName: string,
  @Res({ passthrough: true }) res: Response,
) {
  const fileStream = await this.documentService.getOriginalPDF(
    clientId,
    documentName,
  );

  res.set({
    'Content-Disposition': `attachment; filename="${documentName}"`,
  });

  return new StreamableFile(fileStream);
}
```

```
/**
 * Retrieves all files with pagination support, optionally filtered by client name.
 * @param query The query parameters for pagination and filtering.
 * @returns A paginated result containing the list of files.
 */
@Get('')
@ApiOperation({ summary: 'Get all files with pagination' })
@ApiQuery({
  name: 'page',
  required: false,
  description: 'Page number',
  type: Number,
})
@ApiQuery({
  name: 'limit',
  required: false,
  description: 'Number of items per page',
  type: Number,
})
@ApiQuery({
  name: 'clientName',
  required: false,
  description: 'Client name',
  type: String,
})
@ApiQuery({
  name: 'documentTitle',
  required: false,
  description: 'Document title to search for',
  type: String,
})
@ApiResponse({ status: 200, description: 'List of files' })
async getAllFiles(@Query() query: GetFilesDto): Promise<PaginationResultDto> {
  const { page = 1, limit = 10, clientName, documentTitle } = query;
  return await this.documentService.getAllFiles(
    page,
    limit,
    clientName,
    documentTitle,
  );
}
```

```
/**
 * Updates an existing PDF for a client.
 * @param clientId The ID of the client.
 * @param documentName The name of the document to update.
 * @param updateDocumentDto Data transfer object containing the new title.
 * @param file The updated PDF file.
 * @returns A message indicating successful update and processing.
 * @throws BadRequestException if no file is uploaded.
 */
```

Complexity is 4 Everything is cool!

```
@Post('update/:clientId/:documentName')
@UseInterceptors(FileInterceptor('file', multerOptions))
@ApiOperation({ summary: 'Update an existing PDF' })
@ApiParam({ name: 'clientId', description: 'Client ID' })
@ApiParam({ name: 'documentName', description: 'Document name' })
@ApiBody({ type: UpdateDocumentDto })
@ApiConsumes('multipart/form-data')
@ApiResponse({ status: 200, description: 'PDF updated successfully' })
async updatePDF(
  @Param('clientId') clientId: string,
  @Param('documentName') documentName: string,
  @Body() updateDocumentDto: UpdateDocumentDto,
  @UploadedFile() file?: Express.Multer.File,
): Promise<Record<string, any>> {
  const documentTitle = updateDocumentDto.title.trim();

  if (file) {
    const pdfPath = file.path;
    await this.documentService.updateFile(
      clientId,
      documentName,
      documentTitle,
      pdfPath,
    );
  } else {
    await this.documentService.updateFile(
      clientId,
      documentName,
      documentTitle,
    );
```

```
    }
    return { message: 'PDF updated and processed successfully' };
  }

  /**
   * Deletes a PDF document for a specific client.
   * @param clientId The ID of the client.
   * @param documentName The name of the document to delete.
   * @returns A message indicating successful deletion.
   */
  Complexity is 3 Everything is cool!
  @Delete('delete/:clientId/:documentName')
  @ApiOperation({ summary: 'Delete a PDF' })
  @ApiParam({ name: 'clientId', description: 'Client ID' })
  @ApiParam({ name: 'documentName', description: 'Document name' })
  @ApiResponse({ status: 200, description: 'PDF deleted successfully' })
  async deletePDF(
    @Param('clientId') clientId: string,
    @Param('documentName') documentName: string,
  ): Promise<Record<string, any>> {
    await this.documentService.deleteFile(clientId, documentName);
    return { message: 'PDF deleted successfully' };
  }
}
```

ДОДАТОК Б – Диск із кваліфікаційною роботою бакалавра