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: Інтеграція інтелектуальних сервісів класифікації, пошуку та зберігання в систему керування медіаконтентом ТНТУ

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

_____ Єсипов Л. С.
(підпис) (прізвище та ініціали)

Керівник _____ Мудрик І. Я.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ _____
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ _____
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2025

АНОТАЦІЯ

Єсипов Леонід Сергійович. Інтеграція інтелектуальних сервісів класифікації, пошуку та зберігання в систему керування медіаконтентом ТНТУ. Кваліфікаційна робота освітнього ступеня «магістр». Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерних інформаційних технологій, кафедра програмної інженерії, група СПм-61 спеціальність 121 «Інженерія програмного забезпечення». Тернопіль, 2025. Сторінок 113, таблиць 2, рисунків 9, додатків 8, бібліографічних посилань 55.

Метою роботи є модернізація системи керування медіаконтентом шляхом інтеграції інтелектуальних сервісів для класифікації, пошуку та надійного зберігання файлів.

Об'єктом дослідження є процес організації та управління медіаконтентом у веб-сервісах з використанням сучасних технологій збереження та обробки даних.

Предметом дослідження є методи та технології інтеграції мультимодальних нейромереж, систем розпізнавання облич, інтелектуального пошуку та об'єктного зберігання файлів на базі MinIO.

Методи дослідження включають: проєктування архітектури розподіленого зберігання даних, інтеграцію нейромережевих моделей, тестування функціональних компонентів та автоматизацію робочих процесів.

В роботі продемонстровано процес модернізації медіасховища, що включає: перенесення зберігання файлів з локальної директорії на MinIO, реалізацію мультимодального інтелектуального пошуку з генерацією тегів та описів, інтеграцію системи розпізнавання облич, а також чат-асистента для управління контентом. Розроблена система дозволяє знаходити файли за вмістом, керувати структурами папок і файлами, виявляти дублікати та групувати схожі файли.

Ключові слова: медіаконтент, MinIO, мультимодальні нейромережі, інтелектуальний пошук, розпізнавання облич, чат-асистент, класифікація файлів, управління даними, веб-сервіс, автоматизація процесів.

ABSTRACT

Yesypov Leonid Serhiyovych. Integration of Intelligent Classification, Search, and Storage Services into the TNTU Media Content Management System. Master's Thesis. Ternopil Ivan Puluj National Technical University, Faculty of Computer and Information Technologies, Department of Software Engineering, Group SPm-61, Specialty 121 "Software Engineering". Ternopil, 2025. Pages 113, tables 2, figures 9, appendices 8, references 55.

The aim of the work is the modernization of the media content management system through the integration of intelligent services for file classification, search, and reliable storage.

The research object is the process of organizing and managing media content in web services using modern storage and data processing technologies.

The research subject is the methods and technologies for integrating multimodal neural networks, face recognition systems, intelligent search, and object-based file storage using MinIO.

Research methods include: designing distributed data storage architecture, integrating neural network models, testing functional components, and automating workflows.

The thesis demonstrates the process of upgrading the media repository, including: migrating file storage from a local directory to MinIO, implementing multimodal intelligent search with automatic tagging and description generation, integrating face recognition, and a chatbot assistant for content management. The developed system allows searching files by content, managing folder and file structures, detecting duplicates, and grouping similar files.

Keywords: media content, MinIO, multimodal neural networks, intelligent search, face recognition, chatbot assistant, file classification, data management, web service, process automation.

ЗМІСТ

ВСТУП

В умовах стрімкої цифровізації освітнього та наукового простору закладів вищої освіти спостерігається експоненційне зростання обсягів мультимедійних даних. Сучасний університет генерує терабайти неструктурованої інформації, управління якою стає критичним викликом для існуючої IT-інфраструктури. Традиційні методи, що спираються на локальне зберігання файлів, вичерпують свою ефективність, оскільки не гарантують масштабованості та стійкості до відмов. Ситуація ускладнюється проблемою «семантичного розриву», коли відсутність змістовних метаданих унеможливлює ефективний пошук необхідних матеріалів серед тисяч файлів із технічними назвами. У зв'язку з цим, актуальність теми магістерської роботи обумовлена нагальною потребою технологічної трансформації системи керування медіаконтентом ТНТУ шляхом переходу до розподілених сховищ та застосування інструментів штучного інтелекту задля автоматизованого опрацювання інформації.

Ключова ціль роботи полягає в оптимізації роботи та функціональності системи керування медіаконтентом університету через модернізацію її архітектури, інтеграцію S3-сумісного об'єктного сховища та впровадження інтелектуальних сервісів семантичного аналізу інформації.

Реалізація визначеної мети передбачає виконання низки завдань: аналіз сучасних архітектурних патернів високонавантажених сховищ та методів машинного навчання; проєктування мікросервісної архітектури з відокремленням шару зберігання даних; програмна реалізація інтеграції з системою MinIO для забезпечення горизонтального масштабування та надійності; розробка підсистеми інтелектуального аналізу на базі мультимодальної мережі Gemini 2.5 Flash та модуля розпізнавання облич; створення механізмів семантичного пошуку та інтелектуального асистента для керування файлами природною мовою; адаптація клієнтського інтерфейсу та комплексне тестування розробленого рішення.

Об'єктом дослідження є процес організації, зберігання та пошуку мультимедійної інформації в розподілених веб-системах. Предметом дослідження

є методи інтеграції технологій об’єктного зберігання та сервісів штучного інтелекту у програмне забезпечення для керування цифровими активами.

Наукову новизну отриманих результатів становить оптимізація методу організації та пошуку медіаданих у корпоративних системах. Вперше для інформаційного середовища ТНТУ реалізовано гібридну модель обробки контенту, що поєднує генеративні можливості мультимодальних нейромереж із векторним індексуванням, забезпечуючи пошук файлів за їх візуальним змістом. Набув подальшого розвитку метод людино-машинної взаємодії через впровадження діалогового агента, який використовує механізм Function Calling для трансформації запитів природною мовою у виконувані системні операції.

Практичне значення роботи визначається створенням масштабованого програмного комплексу, готового до впровадження в університеті. Модернізована система на базі технології MinIO забезпечує надійне зберігання даних з можливістю безшовної міграції у хмарне середовище. Реалізовані функції автоматичного тегування, розпізнавання облич та семантичного пошуку дозволяють автоматизувати рутинні процеси адміністрування медіаархіву, суттєво скорочуючи час доступу до необхідної інформації.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Проєктування та модернізація складних інформаційних систем, орієнтованих на обробку великих масивів мультимедійних даних, вимагає ретельного попереднього дослідження архітектурних принципів та функціональних потреб користувачів. Визначення стратегії інтеграції технологій об’єктного зберігання та сервісів штучного інтелекту в наявну інфраструктуру університету є необхідною передумовою для створення масштабованого та відмовостійкого програмного продукту. У цьому контексті критично важливим є формулювання чітких вимог до системи, ідентифікація ключових сценаріїв взаємодії та обґрунтування вибору технологічних інструментів, що забезпечать ефективне вирішення актуальних проблем керування цифровим контентом.

1.1 Аналіз предметної області

Сучасний етап розвитку інформаційних технологій характеризується стрімким, експоненційним зростанням обсягів неструктурованих даних, серед яких левову частку займає мультимедійний контент: зображення, відео- та аудіоматеріали. Для закладів вищої освіти, зокрема технічних університетів, ефективне управління такими даними стає критично важливим завданням. Навчальний процес, наукова діяльність, маркетингові комунікації та документування університетського життя генерують терабайти інформації, яка потребує не лише надійного зберігання, але й швидкого доступу та інтелектуальної обробки. Предметна область даного дослідження охоплює системи керування менеджменту цифрових активів (DAM), засоби хмарного та об’єктного зберігання даних, а також алгоритми машинного навчання, орієнтовані на автоматизовану категоризацію й пошуку інформації.

Традиційні підходи до зберігання файлів, побудовані на основі ієрархічних файлових системах локальних серверів, демонструють низку суттєвих обмежень в умовах сучасних високонавантажених веб-систем. Локальне зберігання

(наприклад, у директоріях веб-сервера) створює проблеми масштабування: при вичерпанні дискового простору виникає необхідність складних апаратних модернізацій або міграції даних, що часто призводить до простою сервісів. Крім того, така архітектура є вразливою до апаратних збоїв, оскільки вихід з ладу фізичного носія може призвести до незворотної втрати інформації за відсутності налаштованих механізмів реплікації або бекапування у реальному часі. У цьому контексті стандартом де-факто стають технології об’єктного сховища (Object Storage), які реалізують доступ до даних через HTTP API (зокрема, протокол S3). Такий підхід дозволяє абстрагуватися від фізичного розміщення файлів, забезпечуючи горизонтальне масштабування, високу доступність та надійність даних завдяки алгоритмам надлишкового кодування (erasure coding) та розподіленого зберігання фрагментів (чанків) на різних вузлах кластера [1].

Іншим критичним аспектом предметної області є проблема пошуку та класифікації мультимедійних даних. В умовах великих сховищ традиційний пошук за назвою файлу або ручне введення метаданих стають неефективними. Користувачі часто завантажують матеріали з автоматично згенерованими іменами (наприклад, «IMG_2024.jpg» або «DCIM_4412.mp4»), які не несуть семантичного навантаження щодо вмісту файлу. Ручне тегування контенту вимагає значних людських ресурсів, є повільним і суб’єктивним, що призводить до неповноти або неточності метаданих. Це явище в науковій літературі відоме як «семантичний розрив» (semantic gap) — невідповідність між низькорівневим представленням медіаданих (пікселі, частоти) та високорівневим сприйняттям людини (об’єкти, дії, емоції).

Вирішення проблеми семантичного розриву лежить у площині використання технологій штучного інтелекту, зокрема мультимодальних нейронних мереж. Сучасні моделі глибокого навчання здатні аналізувати візуальний та аудіальний контент, виділяти ключові об’єкти, розпізнавати сцени, текст на зображеннях (OCR) та навіть інтерпретувати контекст подій. Інтеграція таких сервісів у системи керування контентом дозволяє автоматизувати процес анотування: при завантаженні файлу система самостійно генерує текстовий опис та набір тегів, які

відображають суть зображення чи відео. Це трансформує процес пошуку, дозволяючи користувачеві формулювати запити природною мовою (наприклад, «студенти в лабораторії» або «схема електричного кола»), отримуючи релевантні результати навіть за відсутності точних слів у назві файлу [2].

Окремої уваги заслуговує питання архітектурної організації системи. Сучасні веб-застосунки проєктуються з використанням підходу, що передбачає чітке розмежування рівня презентації (Frontend) та рівня бізнес-логіки (Backend), які взаємодіють між собою через API, а також інтеграцію спеціалізованих зовнішніх сервісів для зберігання та обробки даних. У такій моделі S3-сумісне сховище (MinIO) та модуль штучного інтелекту (Gemini API) виступають як підключені компоненти інфраструктури, що забезпечують виконання специфічних функцій — надійне збереження об’єктів та їх інтелектуальний аналіз. Використання стандартизованих протоколів взаємодії дозволяє абстрагувати шар зберігання, надаючи адміністраторам системи гнучкість у виборі платформи розгортання. Це дає змогу експлуатувати систему на власних фізичних серверах або VPS з використанням MinIO для мінімізації операційних витрат, або ж, за потреби, без зміни програмного коду перевести проєкт на використання хмарного провайдера Amazon S3. Крім того, протокол S3 підтримує такі функції, як контроль версій об’єктів, управління життєвим циклом даних та розмежування прав доступу на рівні окремих бакетів (buckets), що є необхідним для побудови безпечної та масштабованої системи.

Вимоги до надійності систем зберігання в освітніх установах постійно зростають. Втрата даних через збій диска може призвести до зникнення унікальних методичних напрацювань або даних. Тому сучасні підходи передбачають відмову від RAID-масивів на користь програмно-визначених сховищ (Software Defined Storage), де дані розбиваються на частини та розподіляються між незалежними дисками або серверами. Технологія MinIO, що розглядається в даній роботі як ключовий елемент інфраструктури, використовує алгоритми відновлення даних, які дозволяють зберегти цілісність інформації навіть при втраті половини дисків у

кластері. Це забезпечує рівень надійності, недосяжний для класичних файлових серверів.

Паралельно з розвитком технологій зберігання відбувається еволюція методів взаємодії користувача з даними. Сучасні інтерфейси, побудовані на базі реактивних фреймворків (наприклад, Vue.js, React, Nuxt), вимагають швидкого отримання даних через API. Використання GraphQL дозволяє оптимізувати запити, отримуючи лише необхідну інформацію про медіафайли (посилання, метадані, згенеровані нейромережею описи) за один запит, що суттєво зменшує навантаження на мережу та покращує досвід користувача (User Experience).

Аналіз існуючих рішень на ринку показує, що більшість комерційних DAM-систем є дорогими, складними у впровадженні та часто надлишковими для потреб окремого університету. Водночас, використання відкритих (open-source) інструментів дозволяє побудувати гнучку систему, адаптовану під специфічні потреби навчального закладу, з можливістю інтеграції новітніх AI-сервісів. Зокрема, поява потужних мультимодальних моделей (на кшталт серії Gemini) відкриває нові горизонти для інтелектуального аналізу контенту, дозволяючи не просто класифікувати об’єкти, а й розуміти складні візуальні сценарії, що є критично важливим для створення сучасної, «розумної» системи медіасховища.

Таким чином, предметна область роботи знаходиться на перетині технологій розподіленого зберігання даних, веб-розробки та штучного інтелекту. Актуальність дослідження обумовлена необхідністю переходу від застарілих схем зберігання файлів до масштабованих, відмовостійких архітектур з інтегрованими засобами семантичного пошуку, що здатні забезпечити ефективну роботу з медіаконтентом на тлі стрімкої цифровізації навчальної сфери [3]. Це вимагає розробки комплексних підходів до проєктування програмного забезпечення, які б враховували як технічні аспекти збереження бітів інформації, так і когнітивні аспекти її пошуку та інтерпретації.

1.2 Постановка завдання та цілей

Базою для виконання магістерської роботи слугує попередньо розроблена система керування медіаконтентом, яка на етапі бакалаврського проєктування реалізовувала фундаментальні принципи збереження та відображення файлів. Однак, у процесі експлуатації та аналізу архітектури виявлено низку обмежень, що стримують подальший розвиток системи та її інтеграцію в корпоративне середовище університету. Основним недоліком поточної реалізації є використання локальної файлової системи сервера для зберігання медіаданих, що створює єдину точку відмови, ускладнює процес резервного копіювання та обмежує можливості масштабування дискового простору [4]. Крім того, механізм пошуку, що базується виключно на співпадінні символів у назвах файлів, не відповідає сучасним вимогам до інформаційно-пошукових систем, оскільки не враховує семантичний зміст зображень та відеоматеріалів. Відповідно, виникає науково-практична проблема, яка полягає у необхідності модернізації архітектури сховища та впровадженні інтелектуальних методів обробки даних для підвищення ефективності роботи з контентом.

Головною метою роботи є удосконалення системи керування медіаконтентом ТНТУ шляхом інтеграції розподіленого об’єктного сховища та сервісів штучного інтелекту, що забезпечить підвищення надійності зберігання даних, масштабованість інфраструктури та реалізацію семантичного пошуку мультимедійних файлів.

Для досягнення поставленої Реалізація визначеної мети передбачає розв’язання низки завдань. Насамперед необхідно провести аналіз наявних архітектурних патернів для організації об’єктних сховищ та методів автоматичної анотації медіаконтенту, зокрема дослідження можливостей протоколу S3 та мультимодальних нейронних мереж. На основі аналізу необхідно спроєктувати оновлену архітектуру системи, яка передбачає відокремлення шару зберігання даних від логіки веб-додатка.

Ключовим етапом роботи є інтеграція S3-сумісного сховища MinIO. Це передбачає налаштування взаємодії бекенд-частини на базі Django з сервером MinIO, реалізацію механізмів завантаження, видалення та отримання файлів через API. Важливою умовою є забезпечення надійності зберігання, що вимагає конфігурації сховища з використанням механізмів надлишкового кодування (erasure coding) для захисту від апаратних збоїв дисків, а також забезпечення можливості збереження файлів окремими фрагментами (чанками). Система повинна підтримувати потенційну міграцію на хмарні платформи, такі як Amazon S3, без зміни програмного коду, що досягається шляхом суворого дотримання специфікацій S3 API.

Наступним завданням є розробка підсистеми інтелектуального аналізу контенту. Необхідно реалізувати програмний модуль для взаємодії з API мультимодальної нейронної мережі Gemini 2.5 Flash. Цей модуль повинен забезпечувати автоматичну відправку нових медіафайлів на аналіз, отримання згенерованих текстових описів та тегів, а також їх структуроване збереження в базі даних системи. Це дозволить вирішити проблему «семантичного розриву» та забезпечити можливість пошуку файлів за їхнім вмістом [5].

Паралельно з цим необхідно провести адаптацію клієнтської частини додатка, розробленої на Nuxt 3. Це включає оновлення інтерфейсів завантаження файлів, відображення автоматично згенерованих метаданих та реалізацію розширеної форми пошуку, яка дозволить користувачам знаходити контент за ключовими словами, що описують зображені об’єкти або події. Завершальним етапом роботи є проведення комплексного тестування системи, яке включає перевірку відмовостійкості сховища при імітації виходу з ладу фізичних носіїв, оцінку швидкодії операцій вводу-виводу, а також аналіз релевантності результатів пошуку, забезпеченого інтегрованою нейромережею. Виконання окреслених завдань дозволить отримати сучасний, захищений та інтелектуальний програмний продукт, готовий до впровадження в інформаційну екосистему університету.

1.3 Пошук акторів та варіантів використання

Процес проєктування складних інформаційних систем, до яких належить модернізоване сховище медіаконтенту, вимагає чіткої формалізації вимог через ідентифікацію дійових осіб (акторів) та сценаріїв їхньої взаємодії з програмним продуктом. Цей етап є критичним для побудови коректної архітектури, оскільки він визначає функціональні межі системи та специфіку обміну даними між компонентами. Згідно з термінологією уніфікованої мови моделювання (UML), роль акторів виконують не лише кінцеві користувачі, але й зовнішні програмні системи або апаратні комплекси, з якими взаємодіє додаток для виконання своїх функцій [6]. Специфіка даної роботи, що передбачає інтеграцію інтелектуальних сервісів та розподіленого зберігання, суттєво розширює коло акторів порівняно з класичними CRUD-системами.

Основним дійовим актором-людиною є «Користувач». Це узагальнена роль, яка охоплює викладачів, студентів або співробітників університету, що пройшли процедуру автентифікації. В оновленій системі повноваження цього актора значно розширено. Окрім стандартних операцій завантаження та перегляду файлів, користувач отримує інструменти для інтелектуального пошуку та керування контентом за допомогою природної мови. Взаємодія користувача з системою трансформується з імперативної (натискання кнопок для конкретних дій) у декларативну, де актор формулює намір (наприклад, «знайти фото з конференції»), а система самостійно визначає алгоритм виконання запиту. Другим актором-людиною є «Адміністратор», до обов’язків якого входить моніторинг стану системи, керування квотами дискового простору та налаштування параметрів підключення до зовнішніх сервісів.

Серед системних акторів ключову роль відіграє «MinIO Object Storage». У розробленій архітектурі медіасховище більше не є пасивною директорією на сервері, а виступає як автономна сутність, з якою бекенд взаємодіє через протокол S3. Цей актор відповідає за фізичне збереження даних, забезпечення їх цілісності,

реплікацію та надання доступу до об’єктів за посиланням. Відмовостійкість системи безпосередньо залежить від доступності цього актора [7].

Суттєвим нововведенням є поява актора «Сервіс штучного інтелекту» (представленого API Gemini 2.5 Flash). Цей зовнішній компонент ініціюється системою автоматично при завантаженні нового контенту або при зверненні користувача до чат-асистента. Роль цього актора полягає у семантичному аналізі вхідних даних (зображень, запитів користувача) та генерації текстових метаданих або команд керування файловою системою. Інтеграція цього актора дозволяє реалізувати сценарії, які раніше були недоступні, такі як пошук дублікатів за змістом або автоматична генерація описів.

Ще одним специфічним внутрішнім актором є модуль «Розпізнавання облич» (на базі бібліотеки face_recognition). Хоча технічно це бібліотека, на рівні діаграми варіантів використання вона виконує роль автономного агента, що сканує завантажені фотографії, виявляє обличчя, обчислює їх векторні дескриптори та здійснює кластеризацію [8]. Результатом діяльності цього актора є створення та наповнення сутностей «Група персон» (PersonGroup), що дозволяє користувачеві взаємодіяти з новою категорією контенту — персоналізованими добірками фотографій.

Варіанти використання (Use Cases) описують послідовність дій, що виконуються системою у відповідь на ініціативу актора. У межах цього дослідження ключовий акцент зроблено на тих варіантах використання, де ініціатором виступає користувач, а виконавцями — ланцюжок інтелектуальних сервісів. Наприклад, сценарій «Керування файлами через ШІ-асистента» передбачає складну взаємодію: користувач відправляє текстовий запит у чат, бекенд транслює його до LLM-моделі, отримує структуровану команду (JSON), валідує її та виконує відповідну дію над об’єктами в MinIO (створення папки, переміщення, видалення). Такий підхід дозволяє абстрагувати складність файлових операцій від користувача [9]. Аналогічно, сценарій «Семантичний пошук» активує механізм порівняння запиту користувача з векторами або тегами, згенерованими нейромережею, повертаючи релевантні результати незалежно від імен файлів.

У цьому контексті система виступає не лише як сховище даних, а як оркестратор процесів, що координує роботу між користувачем, об’єктним сховищем та сервісами аналізу контенту. Такий підхід накладає додаткові вимоги до формалізації варіантів використання, зокрема щодо обробки помилок, асинхронної взаємодії та забезпечення узгодженості стану системи при частковій недоступності окремих акторів.

Узагальнений перелік ідентифікованих акторів та основних прецедентів їх взаємодії з системою наведено в таблиці варіантів використання. Ця таблиця слугує основою для подальшого детального проектування інтерфейсів та програмної логіки.

Таблиця 1.3.1 — Варіанти використання інтелектуальної системи керування медіаконтентом

Код	**Актор**	**Назва варіанту використання**	**Опис сценарію взаємодії**
UC.01	Користувач	Завантаження медіаконтенту	Користувач ініціює передачу файлів. Система зберігає їх у сховищі MinIO, автоматично запускає фонові процеси генерації опису через Gemini та розпізнавання облич.
UC.02	Користувач	Семантичний пошук	Користувач вводить запит природною мовою. Система аналізує вектори та текстові описи, повертаючи релевантні файли незалежно від їх назви.
UC.03	Користувач	Взаємодія з ШІ-асистентом	Користувач через чат-інтерфейс пише команди природною мовою. Система обробляє запит та виконує відповідні операції.

Продовження таблиці 1.3.1.

Код	**Актор**	**Назва варіанту використання**	**Опис сценарію взаємодії**
UC.04	Користувач	Перегляд груп людей/тварин	Користувач переходить у відповідний розділ, де система відображає згруповані за візуальною схожістю фотографії конкретних осіб,
UC.05	Користувач	Пошук схожих зображень	Система аналізує візуальні характеристики зображень користувача та групує між собою зображення, що мають високий ступінь схожості.
UC.06	Користувач	Ручне редагування метаданих	Користувач вносить зміни до автоматично згенерованих тегів або описів у випадку їх неточності, система оновлює індекси для пошуку.
UC.07	Користувач	Автентифікація в системі	Користувач вводить ідентифікаційні дані. Платформа здійснює верифікацію дозволів та надає токен авторизації для виконання захищених операцій.
UC.08	MinIO (S3)	Забезпечення персистентності	Сховище приймає вхідний потік даних, виконує розподіл чанків по дисках, забезпечує реплікацію та прямий доступ до об’єктів за URL.

Продовження таблиці 1.3.1.

Код	**Актор**	**Назва варіанту використання**	**Опис сценарію взаємодії**
UC.09	ШІ-сервіс (Gemini)	Генерація метаданих	Зовнішній сервіс отримує медіафайл, проводить мультимодальний аналіз та повертає текстовий опис сцени і набір ключових слів.
UC.10	Модуль розпізнавання	Кластеризація облич	Внутрішній сервіс детектує обличчя на зображеннях, обчислює їх векторні дескриптори, порівнює з наявними групами та оновлює PersonGroup.

На основі проведеного аналізу предметної області, ідентифікації ключових акторів та формалізації сценаріїв їхньої взаємодії з програмним комплексом, побудовано діаграму варіантів використання (Use Case Diagram). Графічна модель розроблена згідно зі специфікацією уніфікованої мови моделювання UML та візуалізує функціональні межі системи, ієрархію акторів, а також типи зв’язків між ними та прецедентами. На діаграмі відображено інтеграцію зовнішніх інтелектуальних сервісів та підсистеми зберігання даних, демонструючи їх роль у забезпеченні бізнес-логіки додатка.

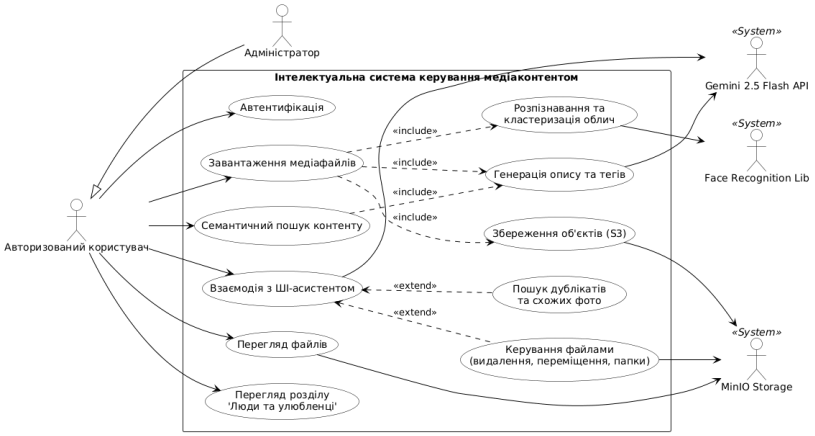


Рисунок 1.3.1 - Діаграма варіантів використання.

1.4 Опис ключових варіантів використання

Поглиблений опис ключових варіантів використання є необхідним етапом проектування, що дозволяє розкрити внутрішню логіку функціонування системи, специфіку обробки даних та послідовність взаємодії між клієнтським інтерфейсом, сервером додатків, системою зберігання та зовнішніми інтелектуальними сервісами. Враховуючи складну архітектуру розроблюваного програмного комплексу, яка поєднує синхронні операції користувача з асинхронними фоновими задачами обробки медіаконтенту, критично важливо формалізувати сценарії, що демонструють інтеграцію штучного інтелекту в процеси керування контентом [10]. Нижче наведено детальний опис п’яти найбільш значущих прецедентів: завантаження контенту з мультимодальним аналізом (UC.01), семантичний пошук (UC.02), взаємодія з інтелектуальним асистентом (UC.03), автоматичне групування схожих зображень (UC.05) та перегляд персоналізованих добірок (UC.04).

Обрані варіанти використання не є вичерпним переліком усіх можливих сценаріїв роботи системи, проте вони репрезентують найбільш критичні з точки зору архітектурних рішень та обчислювального навантаження процеси. Саме в межах цих сценаріїв відбувається інтенсивна взаємодія між користувачем і серверною частиною, використання зовнішніх сервісів штучного інтелекту, а також обробка великих обсягів медіаданих у фоновому режимі. Формалізація зазначених прецедентів дозволяє не лише описати поведінку системи з позиції кінцевого користувача, але й чітко визначити точки інтеграції компонентів, потенційні вузькі місця продуктивності та вимоги до масштабованості програмного комплексу в цілому.

Сценарій «Завантаження медіаконтенту» (UC.01) є фундаментальним процесом, що ініціює більшість внутрішніх процедур системи. Виконання сценарію розпочинається з моменту, коли користувач через веб-інтерфейс обирає один або декілька файлів для завантаження. На клієнтській стороні, реалізованій на Nuxt 3, відбувається попередня валідація форматів та розмірів файлів, після чого формується запит до GraphQL API. Специфіка даного варіанту використання

полягає у розпаралелюванні потоків даних. Першочергово бекенд-система на базі Django здійснює передачу бінарного потоку файлу до S3-сумісного сховища MinIO. Цей етап є критичним для забезпечення цілісності даних: система розбиває файл на частини (чанки) та розподіляє їх по дисковому масиву згідно з налаштованою політикою erasure coding, що гарантує збереження інформації навіть у випадку апаратних збоїв. Тільки після отримання підтвердження про успішний запис об’єкта в сховище, система створює відповідний запис у реляційній базі даних і повертає користувачеві статус успішного завантаження. Однак, завершення синхронної частини сценарію автоматично тригерить запуск асинхронних задач у черзі повідомлень. Перша фонова задача ініціює звернення до API нейромережі Gemini 2.5 Flash, передаючи завантажене зображення для генерації текстового опису та тегів. Друга задача активує модуль face_recognition для виявлення та кодування людських облич. Такий підхід дозволяє не блокувати інтерфейс користувача під час виконання ресурсомістких обчислень, забезпечуючи високу чутливість системи (responsiveness) [11].

Варіант використання «Семантичний пошук» (UC.02) демонструє перехід від лексикографічного порівняння рядків до аналізу змісту. У класичних системах пошук обмежується співставленням введених символів з назвою файлу, що є неефективним для медіаконтенту. У розробленій системі сценарій розгортається наступним чином: користувач вводить пошуковий запит природною мовою (наприклад, «студенти працюють за комп’ютерами» або «зимовий пейзаж»). Система не шукає буквальне входження цієї фрази в імена файлів. Натомість, запит аналізується та співставляється з індекосованими описами та тегами, які були раніше згенеровані нейромережею Gemini під час завантаження файлів. Алгоритм пошуку враховує семантичну близькість слів та контекст, що дозволяє знаходити релевантні зображення, навіть якщо в їхньому описі використовуються синоніми або перефразовані вирази. Результатом виконання сценарію є формування вибірки медіаоб’єктів, ранжованої за ступенем релевантності запиту. Технічно це реалізується через оптимізовані SQL-запити до бази даних, де зберігаються результати роботи ШІ, з подальшим формуванням GraphQL-відповіді, яка містить

превью зображень та їх метадані. Цей сценарій кардинально змінює досвід користувача, дозволяючи оперувати візуальними образами, а не технічними ідентифікаторами файлів [12].

Сценарій «Взаємодія з ШІ-асистентом» (UC.03) представляє собою складний процес людино-машинної взаємодії, де система виступає в ролі інтелектуального агента. Користувач відкриває спеціалізований чат-інтерфейс і формулює команду довільною мовою, наприклад, «Створи папку 'Конференція 2024' та перемісти туди всі фотографії з презентаціями». Система обробляє цей текст, використовуючи контекстне вікно LLM (Large Language Model), для визначення наміру користувача (intent recognition) та виділення сутностей (папка, назва, критерій відбору файлів). Особливістю реалізації виступає здатність нейромережі не просто генерувати текстову відповідь, а повертає структуровану команду у форматі JSON, яку бекенд може інтерпретувати як виклик внутрішньої функції (Function Calling). Далі система автоматично виконує серію операцій: створює логічну директорію в структурі проекту, виконує пошук файлів за критерієм «фотографії з презентаціями» (використовуючи механізм семантичного пошуку) та оновлює шляхи до цих файлів у базі даних та, за необхідності, переміщує об’єкти в бакетах MinIO. Після завершення операцій асистент генерує звіт про виконану роботу. Цей варіант використання суттєво знижує когнітивне навантаження на користувача при виконанні рутинних операцій з упорядкування великих архівів даних.

Варіант використання «Пошук та групування схожих зображень» (UC.05) спрямований на вирішення проблеми надлишковості даних та оптимізацію використання дискового простору. Суть сценарію полягає в автоматичному аналізі візуальних характеристик наявного контенту для виявлення дублікатів або серій майже ідентичних знімків. Процес може бути ініційований користувачем або запускатися за розкладом. Система сканує вектори ознак (feature vectors) зображень або використовує алгоритми перцептивного хешування (perceptual hashing) для порівняння контенту. На відміну від звичайного порівняння контрольних сум (MD5/SHA), яке виявляє лише ідентичні файли, даний підхід дозволяє ідентифікувати зображення, які можуть відрізнятися роздільною здатністю,

ступенем стиснення або незначними змінами ракурсу. Виявлені схожі зображення групуються в кластери, які відображаються користувачеві з пропозицією залишити найкращий варіант і видалити інші, або ж об’єднати їх у віртуальну стопку (stack). Це дозволяє підтримувати чистоту медіасховища та уникати зберігання надлишкової інформації. Технічна реалізація вимагає високоефективних алгоритмів порівняння, щоб забезпечити прийнятну швидкодію при обробці тисяч файлів.

Завершальним ключовим сценарієм є «Перегляд груп людей» (UC.04), який базується на результатах роботи модуля розпізнавання облич. Користувач переходить до розділу «Люди та улюбленці», де система відображає динамічно сформовані картки персоналій. Кожна картка відповідає унікальній особі, яку система ідентифікувала на завантажених фотографіях. При виборі конкретної персони ініціюється запит на отримання всіх зображень, пов’язаних із відповідним ідентифікатором групи (PersonGroup). Складність цього сценарію прихована на етапі фонової обробки (UC.10), де система обчислює 128-вимірні вектори облич, порівнює їх з наявними в базі за допомогою евклідової відстані та приймає рішення про приналежність до існуючого кластера або створення нового [13]. Інтерфейс користувача в цьому сценарії надає інструменти для керування цими групами: користувач може присвоїти ім’я автоматично створеній групі (наприклад, «Ректор» або «Група 121»), об’єднати декілька груп, якщо система помилково розділила одну людину на дві сутності через різні умови освітлення, або видалити помилкові розпізнавання. Цей функціонал перетворює сховище з пасивного архіву файлів на структуровану базу знань про учасників університетського життя.

Сукупність описаних варіантів використання формує цілісну картину функціонування інтелектуальної системи. Вони демонструють, як інтеграція сучасних технологій машинного навчання та надійного об’єктного зберігання дозволяє автоматизувати складні процеси класифікації, пошуку та управління, забезпечуючи при цьому зручність та ефективність роботи кінцевого користувача.

Обрана сукупність ключових варіантів використання дозволяє не лише моделювати поведінку користувача, але й визначити критичні точки інтеграції між

компонентами системи. Аналіз кожного сценарію сприяє виявленню потенційних вузьких місць продуктивності, оцінці навантажень на бекенд і об’єктне сховище, а також формує вимоги до асинхронних процесів та механізмів черг обробки. Завдяки такому підходу проєктувальники отримують цілісне уявлення про взаємозв’язки між користувацьким інтерфейсом, серверними модулями та інтелектуальними сервісами, що дозволяє забезпечити як надійність, так і високий рівень зручності для кінцевого користувача.

На основі дослідження предметної сфери та вимог до програмної системи обґрунтовано доцільність технологічної трансформації існуючого медіасховища шляхом переходу до архітектури розподіленого об’єктного зберігання на базі MinIO та впровадження інтелектуальних засобів аналізу даних. Встановлено, що інтеграція мультимодальної нейромережі Gemini 2.5 Flash та алгоритмів розпізнавання облич дозволяє нівелювати обмеження традиційних файлових систем, забезпечуючи автоматизацію процесів анотування та реалізацію механізмів семантичного пошуку. Сформована діаграма варіантів використання та деталізація ключових сценаріїв, зокрема взаємодії з ШІ-асистентом та автоматичного групування зображень, створюють необхідну базу для подальшого етапу проєктування архітектури та розробки програмних компонентів, гарантуючи відповідність кінцевого продукту сучасним стандартам надійності та ергономіки.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ

Практична реалізація сформованих функціональних вимог та сценаріїв використання передбачає застосування системного підходу до конструювання програмного забезпечення, що базується на обґрунтованих архітектурних та технологічних рішеннях. Створення масштабованої системи керування медіаконтентом вимагає детального опрацювання структури взаємодії компонентів, зокрема механізмів інтеграції розподіленого сховища даних та сервісів штучного інтелекту. Ключовими етапами цього процесу є визначення методології розробки, проєктування інформаційної моделі та об'єктна декомпозиція системи, які слугують фундаментом для написання програмного коду. У даному розділі викладено комплексний процес інженерного втілення проєкту: від обґрунтування вибору технологічного стека та побудови схем бази даних до програмної реалізації алгоритмів інтелектуальної обробки контенту та створення адаптивного інтерфейсу користувача.

2.1 Вибір процесу розробки

Визначення методології розробки є фундаментальним етапом проєктування програмного забезпечення, адже цей чинник прямо визначає якісні показники готового рішення, раціональність розподілу ресурсів і здатність команди адаптуватися до змін у вимогах. Для науково-дослідних проєктів, які передбачають інтеграцію передових рішень, зокрема машинного навчання та розподілені системи зберігання даних, вибір моделі життєвого циклу (SDLC) набуває критичного значення. Специфіка даної магістерської роботи полягає у модернізації вже існуючої системи з впровадженням компонентів, поведінка яких не є повністю детермінованою на початковому етапі (зокрема, точність роботи нейромереж Gemini та Face Recognition), що унеможливлює використання жорстких лінійних моделей, таких як каскадна модель (Waterfall). Класичний каскадний підхід вимагає повної фіксації вимог до початку розробки, що в умовах необхідності

проведення експериментів з налаштування параметрів AI-моделей та конфігурації сховища MinIO створює високі ризики необхідності повної переробки системи на пізніх етапах [14].

З огляду на вищезазначене, для реалізації проєкту було обрано гнучку методологію розробки (Agile), яка базується на ітеративному та інкрементному підходах. Відповідно до принципів Agile, процес створення програмного продукту розбивається на короткі цикли (ітерації), кожен з яких завершується отриманням працюючої версії системи з новим функціоналом. Це дозволяє здійснювати безперервну перевірку гіпотез, зокрема тестування релевантності генерованих описів зображень та стабільності роботи драйвера S3, і оперативно вносити корективи в архітектуру без суттєвих часових втрат [15]. У контексті даної роботи застосовано адаптовану модель, близьку до фреймворку Scrum, де весь обсяг робіт розділено на логічні етапи: рефакторинг шару зберігання даних, розробка інтеграційних модулів для взаємодії із зовнішніми API, реалізація бізнес-логіки на боці серверної частини та адаптація клієнтського інтерфейсу.

Важливим елементом обраного процесу розробки є використання систем контролю версій та практик CI/CD (Continuous Integration/Continuous Delivery). Для забезпечення цілісності кодової бази та можливості паралельної роботи над різними модулями (наприклад, незалежна розробка GraphQL API та компонентів Nuxt 3) використано розподілену систему керування версіями Git. Застосовано стратегію розгалуження (branching strategy), яка передбачає створення окремих гілок для кожної нової функції (feature branches), таких як feature/minio-integration або feature/ai-assistant. Злиття гілок з основною версією проєкту відбувається лише після проходження локальних тестів та перевірки на відсутність конфліктів. Такий підхід дозволяє підтримувати стабільний стан основної гілки проєкту та забезпечує можливість швидкого відкату змін у разі виявлення критичних помилок, що є стандартом інженерії програмного забезпечення для веб-орієнтованих систем [16].

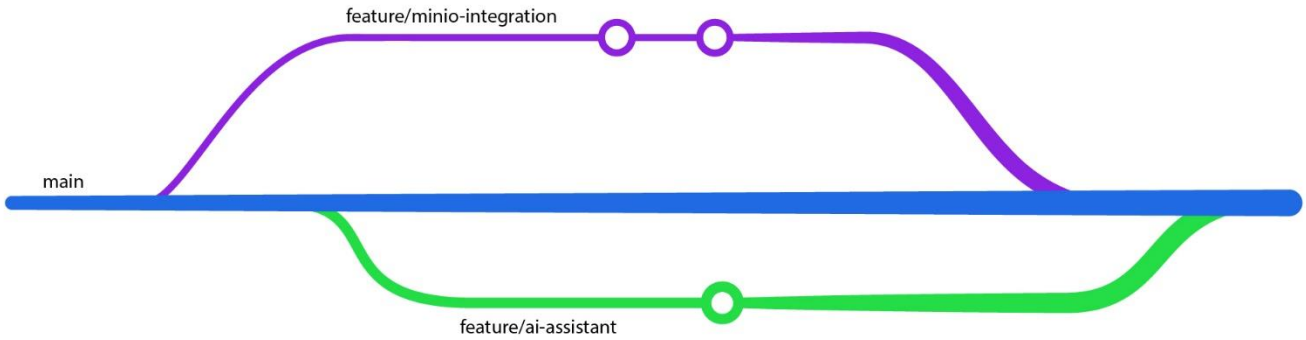


Рисунок 2.1.1 – Візуалізація Github гілок

Процес розробки також включав етапи дослідницького прототипування. Перед повномасштабною інтеграцією сервісу Gemini 2.5 Flash було створено окремі скрипти для тестування швидкодії API та якості розпізнавання української мови в запитах. Отримані емпіричні дані дозволили скоригувати вимоги до формату збереження метаданих у базі даних ще до етапу написання основного коду [17]. Таким чином, обрана гнучка стратегія розробки забезпечила органічне поєднання інженерних практик із науковим пошуком, дозволяючи поетапно нарощувати функціональність системи керування медіаконтентом від базового файлового сховища до інтелектуальної платформи з елементами машинного навчання.

2.2 Проєктування архітектури системи

Проєктування архітектури програмного комплексу є визначальним етапом, що встановлює структурний каркас системи, правила взаємодії її компонентів та принципи розподілу навантаження. Для реалізації поставлених завдань щодо інтеграції інтелектуальних сервісів та забезпечення надійності зберігання даних обрано архітектурний підхід, що базується на принципах сервіс-орієнтованої архітектури (SOA) з елементами мікросервісної декомпозиції [18]. Така модель дозволяє чітко розмежувати відповідальність між модулями системи,

забезпечуючи їх слабку зв’язність (loose coupling) та високу внутрішню узгодженість (high cohesion) [19].

Загальна структура системи представлена трьома логічними рівнями: рівнем презентації (Frontend), рівнем бізнес-логіки (Backend) та рівнем інфраструктури даних (Data Layer), який включає базу даних та об’єктне сховище. Взаємодія між клієнтською та серверною частинами реалізована через уніфікований програмний інтерфейс GraphQL, що дозволяє клієнту гнучко формувати запити, отримуючи лише необхідний обсяг даних за одну ітерацію мережевої взаємодії, на відміну від класичного REST-підходу, який часто призводить до проблеми надлишкової або недостатньої вибірки даних (over-fetching/under-fetching).

Центральним елементом архітектури виступає серверна частина, реалізована на базі фреймворку Django. Вона виконує роль оркестратора процесів, приймаючи запити від користувачів та делегуючи виконання специфічних завдань відповідним підсистемам. Критичною архітектурною зміною у порівнянні з попередньою версією системи є винесення підсистеми зберігання файлів за межі веб-сервера. Замість локальної файлової системи інтегровано високопродуктивне об’єктне сховище MinIO, яке працює за протоколом S3. Архітектурно це реалізовано через патерн «сховище як сервіс» (Storage-as-a-Service). Бекенд не оперує файлами фізично, а лише керує метаданими (ключами об’єктів, шляхами, правами доступу) та генерує попередньо підписані URL-адреси (presigned URLs) для прямого завантаження або отримання контенту клієнтом. Такий підхід забезпечує горизонтальну масштабованість системи та дозволяє безшовно замінити локальний інстанс MinIO на хмарний сервіс Amazon S3 без зміни програмного коду [20].

Графічне представлення спроєктованої архітектури та взаємодії компонентів наведено на рисунку 2.1.

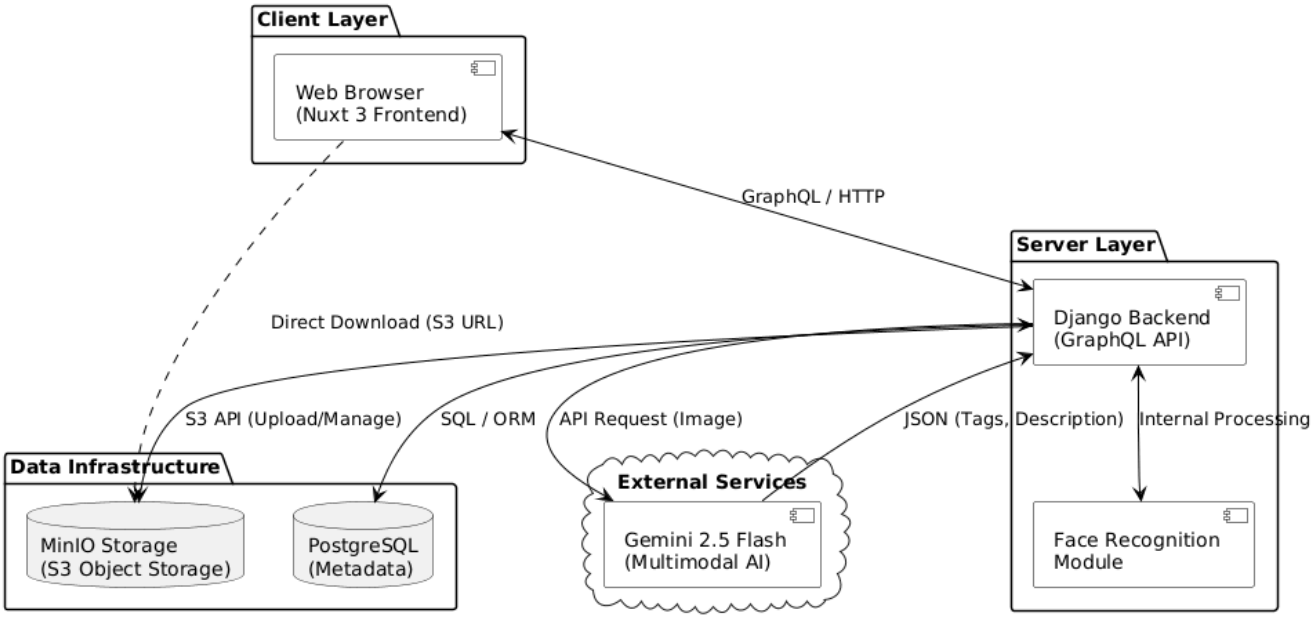


Рисунок 2.2.1 – Загальна архітектура інтелектуальної системи керування медіаконтентом

Підсистема інтелектуального аналізу даних інтегрована в архітектуру як сукупність асинхронних обробників. Враховуючи високу обчислювальну складність операцій з нейромережами, процеси генерації описів та розпізнавання облич винесено у фонові задачі, що не блокують основний потік виконання запитів користувача. Взаємодія з зовнішнім API мультимодальної моделі Gemini 2.5 Flash реалізована через спеціалізований адаптер, який інкапсулює логіку авторизації, формування промптів та обробки відповідей. Модуль розпізнавання облич (face_recognition), хоча і розгортається локально, архітектурно відокремлений у самостійний сервіс, що оперує власними структурами даних для зберігання векторних дескрипторів облич та груп персон.

Рівень презентації побудовано на базі фреймворку Nuxt 3, що забезпечує гібридний рендеринг сторінок. Архітектура фронтенду спроєктована за компонентним принципом, де кожен елемент інтерфейсу (форма завантаження, галерея, чат-асистент) є ізольованим модулем зі своїм станом та логікою. Особливістю архітектури клієнтської частини є наявність локального сховища стану (State Management), яке синхронізується з сервером через GraphQL-мутації

та підписки, забезпечуючи реактивне оновлення інтерфейсу при зміні статусів обробки файлів.

База даних системи (PostgreSQL) виконує роль джерела істини для структурованих метаданих, зберігаючи інформацію про користувачів, посилання на об’єкти в MinIO, згенеровані нейромережею текстові описи та вектори взаємозв’язків між фотографіями. Така архітектура даних відповідає принципам побудови систем з інтенсивним використанням даних (data-intensive applications), де реляційна цілісність поєднується з гнучкістю неструктурованого об’єктного зберігання [21]. Запропоноване архітектурне рішення забезпечує високий рівень відмовостійкості, оскільки вихід з ладу одного з компонентів (наприклад, тимчасова недоступність зовнішнього API Gemini) не призводить до повної зупинки системи, а лише обмежує доступність специфічних функцій.

2.3 Побудова схем бази даних

Продуктивність роботи інформаційної системи, яка оперує великими обсягами гетерогенних даних, безпосередньо визначається грамотністю побудови схеми БД. З метою створення системи керування медіаконтентом обрано реляційну модель даних, яка забезпечує цілісність інформації, чітку типізацію атрибутів та можливість встановлення складних зв’язків між сутностями [22]. У якості системи керування базами даних (СКБД) використано PostgreSQL, що обумовлено її надійністю, підтримкою роботи з JSON-структурами та наявністю розширень для зберігання векторних даних [23], необхідних для реалізації функцій семантичного пошуку. Процес проєктування включав етапи концептуального, логічного та фізичного моделювання, результатом яких стала нормалізована схема, адаптована до потреб збереження як структурованих метаданих, так і результатів роботи нейронних мереж.

Ядром підсистеми розмежування доступу є сутність Profile, яка розширює стандартну модель користувача. Окрім базових атрибутів автентифікації (username, password, email), модель містить специфічні поля для керування квотами дискового

простору (disk_limit), що є критичним для багатокористувацького сховища. Поле file_display_mode дозволяє зберігати індивідуальні налаштування інтерфейсу. Для глобального конфігурування системи виділено окрему сутність GlobalSettings, яка визначає ліміти за замовчуванням (default_user_disk_limit) та керує перемикачем активації автоматичного тегування (enable_ai_tagging), що дозволяє адміністратору гнучко управляти навантаженням на обчислювальні ресурси.

Організація файлової структури базується на двох ключових сутностях: Folder та File. Для реалізації вкладеної ієрархії папок використано алгоритм обходу дерев Modified Preorder Tree Traversal (MPTT) [24], на що вказує наявність технічних полів lft, rght, tree_id та level. Такий підхід дозволяє оптимізувати виконання запитів на вибірку повного дерева каталогів або окремих підгілок, мінімізуючи навантаження на базу даних порівняно з класичним рекурсивним підходом. Сутність Folder містить посилання на власника (user) та батьківський каталог (parent), а також атрибут is_shared для реалізації механізму спільного доступу.

Сутність File виступає зв’язуючою ланкою між фізичним об’єктом у сховищі MinIO та його логічним представленням у системі. Вона не зберігає бінарний контент, а містить посилання (link, public_url) та метадані (size, created_at). Інтеграція з інтелектуальними сервісами відображена у наявності полів ai_description та ai_tags, куди записуються результати генерації тексту нейромережею Gemini. Поле content_embedding призначене для зберігання векторного представлення змісту файлу, що дозволяє виконувати математичні операції для пошуку семантично схожих об’єктів [25]. Логічне поле ai_processed слугує маркером статусу обробки, дозволяючи системі відстежувати файли, які ще потребують аналізу.

Для реалізації функціоналу розпізнавання облич розроблено окремий блок сутностей. Модель PersonGroup представляє кластер зображень, що належать одній особі, містячи назву групи (ім’я людини) та посилання на титульне фото (cover_photo). Зв’язок між конкретним файлом та персоною реалізується через сутність FaceEmbedding, яка зберігає унікальний векторний дескриптор обличчя

(embedding) та координати його розташування на зображенні (location_json). Така декомпозиція дозволяє коректно обробляти ситуації, коли на одному фото присутні декілька осіб (зв’язок «один-до-багатьох»), а також коли одна особа зустрічається на багатьох фотографіях, забезпечуючи гнучкий пошук контенту за персоналіями [26].

Графічне представлення розробленої ER-діаграми (Entity-Relationship diagram), що відображає атрибутивний склад сутностей та зв’язки між ними, наведено на рисунку 2.2.

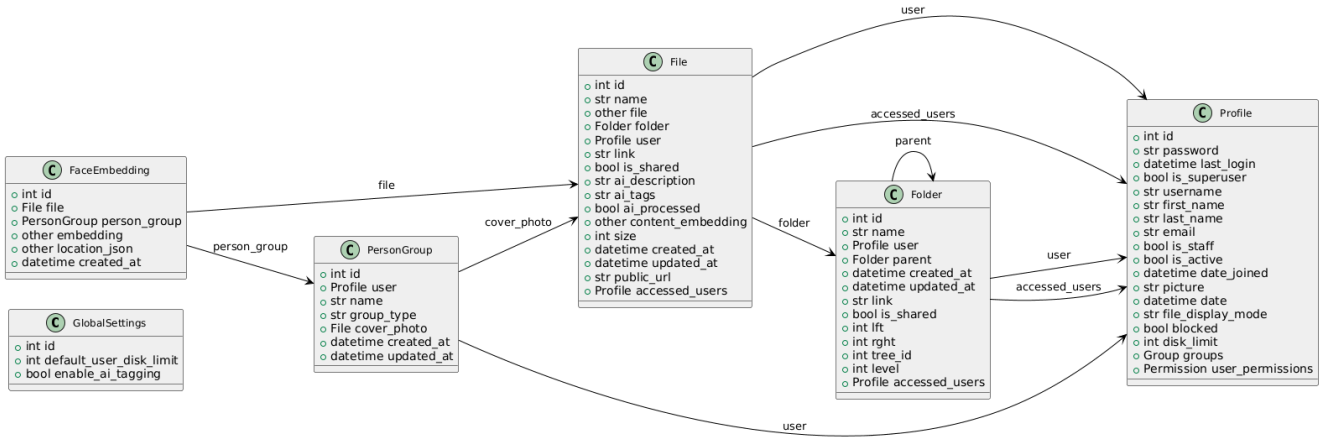


Рисунок 2.3.1 — Логічна структура бази даних платформи менеджменту медіаконтентом

2.4 Побудова UML-діаграм класів

Етап об’єктно-орієнтованого проєктування є ключовою стадією розробки програмного забезпечення, на якій відбувається трансформація функціональних вимог та сценаріїв використання у чітку структуру програмних компонентів. Основним інструментом візуалізації статичної структури системи є діаграма класів у нотації UML. Цей засіб моделювання дозволяє відобразити внутрішню архітектуру додатка, визначити типи даних, методи обробки інформації та характер взаємозв’язків між сутностями, такими як асоціація, агрегація, композиція та успадкування. У контексті розробки інтелектуальної системи керування медіаконтентом, діаграма класів слугує "кресленням", за яким реалізується модель

даних (Model) у паттерні MTV (Model-Template-View), що використовується фреймворком Django [27], а також визначає структуру сервісного шару, відповідального за інтеграцію з нейронними мережами.

Загальна структура спроєктованої системи базується на модульному принципі, де класи логічно згруповані за функціональним призначенням. Можна виділити чотири основні кластери: модуль керування користувачами та налаштуваннями, модуль файлової системи та зберігання, модуль біометричної ідентифікації (розпізнавання облич) та сервісний шар бізнес-логіки. Така декомпозиція дозволяє дотримуватися принципу єдиної відповідальності (Single Responsibility Principle) та знижує зв’язність коду.

Центральним елементом підсистеми доступу є клас Profile, який розширює стандартну модель користувача через механізм успадкування від базового класу AbstractUser. Цей клас виступає власником усіх інформаційних об’єктів у системі. Окрім стандартних атрибутів автентифікації,Profile містить специфічні поля, що визначають параметри квотування ресурсів, зокрема disk_limit, який встановлює обмеження на обсяг збережених даних у гігабайтах. Атрибут file_display_mode відповідає за збереження налаштувань інтерфейсу, дозволяючи адаптувати вигляд файлового менеджера під вподобання користувача. Для забезпечення безпеки передбачено поле blocked, що дозволяє адміністратору обмежувати доступ до акаунту без видалення даних. Важливим архітектурним рішенням є використання патерну Singleton для класу GlobalSettings [28]. Цей клас існує в єдиному екземплярі та зберігає конфігурацію, спільну для всієї системи, наприклад, default_user_disk_limit для нових користувачів та прапорець enable_ai_tagging, що дозволяє централізовано керувати доступністю функцій штучного інтелекту.

Ядро системи — модуль файлової структури — реалізовано через класи Folder та File. Клас Folder проєктувався з урахуванням вимог до швидкої побудови ієрархічних дерев. Для цього використано підхід Modified Preorder Tree Traversal (MPTT) [29], що відображено у структурі класу через наявність службових полів для індексації дерева. Зв’язок parent реалізує рекурсивну структуру, дозволяючи створювати необмежену вкладеність каталогів. Метод generate_link() інкапсулює

логіку створення унікальних хешованих посилань для надання публічного доступу до папки. Атрибут accessed_users реалізує відношення «багато-до-багатьох», забезпечуючи механізм спільного доступу (sharing) між різними користувачами системи.

Клас File є складнішою сутністю, яка поєднує властивості традиційного файлового об’єкта та контейнера для метаданих, згенерованих штучним інтелектом. Поле file відповідає за фізичне збереження об’єкта у S3-сумісному сховищі. Група полів з префіксом ai_ (ai_description, ai_tags, ai_processed) призначена для зберігання результатів роботи мультимодальної моделі Gemini. Ключовою особливістю є поле content_embedding, яке зберігає 768-вимірний вектор семантичного змісту файлу. Це поле є основою для реалізації векторного пошуку: воно дозволяє обчислювати косинусну відстань між запитом користувача та вмістом файлу, нівелюючи необхідність точного співпадіння ключових слів.

Окремий кластер класів відповідає за функціонал розпізнавання облич. Клас PersonGroup представляє собою логічне об’єднання зображень, на яких ідентифіковано одну й ту ж особу. Він містить атрибут group_type, що дозволяє класифікувати об’єкти (наприклад, «Людина», «Тварина», «Невідомий»), та посилання cover_photo на еталонне зображення для відображення в інтерфейсі. Зв’язок між конкретним файлом та групою реалізується через клас FaceEmbedding. Цей клас зберігає унікальний біометричний підпис обличчя у вигляді 128-вимірного вектора у полі embedding та координати обличчя на зображенні у полі location_json. Така архітектура дозволяє реалізовувати складні запити, наприклад, знаходити всі фотографії конкретної людини, використовуючи математичні операції над векторами.

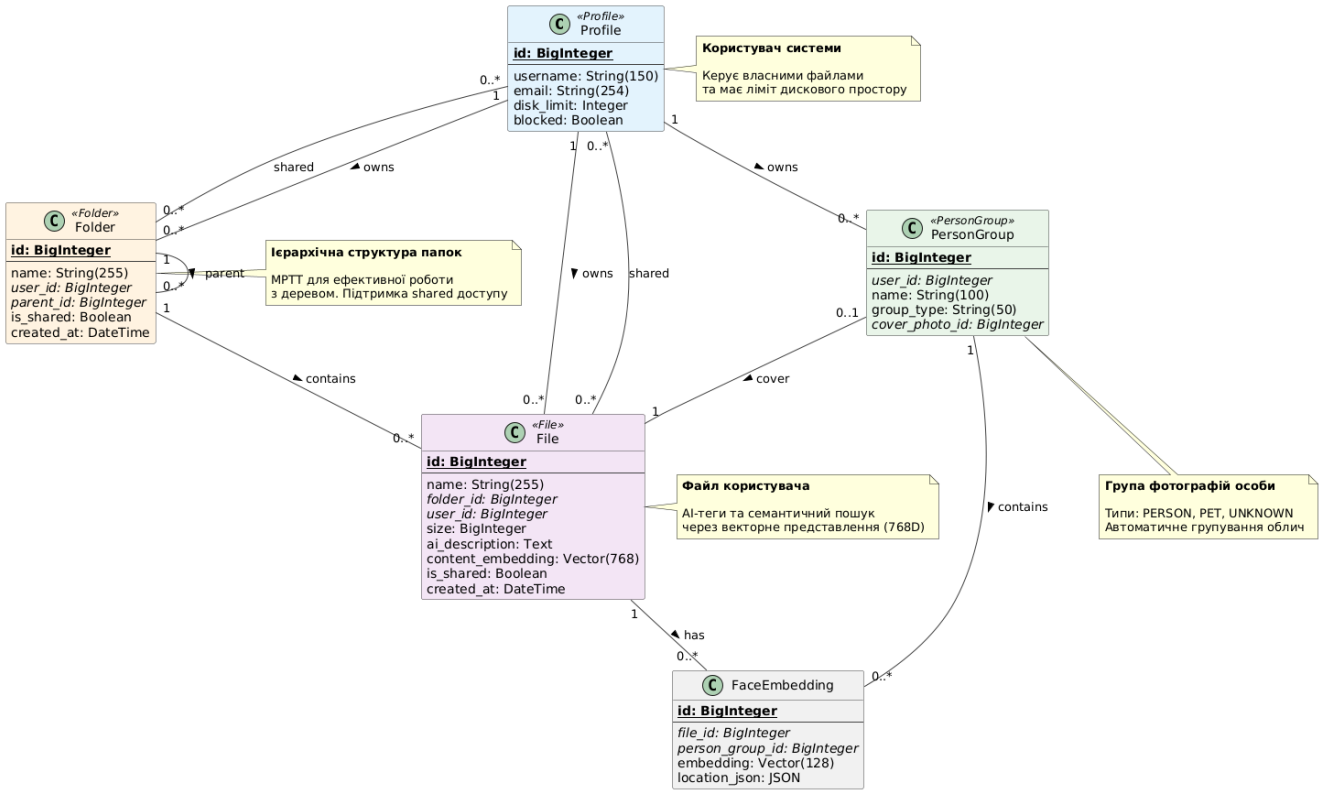


Рисунок 2.4.1 – Фрагмент діаграми класів: основні сутності даних

Сервісний шар системи, який відповідає за бізнес-логіку та взаємодію із зовнішніми API, представлений класами AIService, FaceRecognitionService та AgentService. Ці класи не є моделями бази даних у класичному розумінні Django, а являють собою сервісні об’єкти (Service Objects), що інкапсулюють складні алгоритми [30].

Клас AIService виступає адаптером до API Google Gemini. Його метод get_text_embedding() відповідає за перетворення текстових запитів у векторний простір, сумісний з content_embedding файлів. Методи _process_image(), _process_text() та _process_video() реалізують специфічну логіку підготовки даних різних модальностей перед відправкою на аналіз нейромережі. Метод generate_tags_and_description() оркеструє цей процес, отримуючи сирі дані та повертаючи структурований JSON з описом.

Клас FaceRecognitionService реалізує локальну логіку комп’ютерного зору. Метод process_file() використовує бібліотеку dlib для детекції облич. Найважливішим є метод _assign_face_to_group(), який містить алгоритмічну логіку

кластеризації: він обчислює евклідову відстань (L2 distance) між новим вектором обличчя та існуючими групами. Якщо відстань менша за встановлений поріг (tolerance = 0.6), обличчя додається до існуючої групи, інакше створюється нова сутність PersonGroup.

Найбільш складним з точки зору поведінкової логіки є клас AgentService, який реалізує функціонал інтелектуального асистента. Метод chat() приймає повідомлення користувача і використовує механізм Function Calling моделі Gemini для визначення наміру користувача. Клас містить набір приватних методів, що відповідають конкретним діям: _search_files() (гібридний пошук за ключовими словами та векторами), _find_duplicates() (пошук дублікатів за хешами), _create_folder(), _move_files() тощо. Окремої уваги заслуговує метод _find_similar_groups(), який спирається на алгоритм щільної кластеризації DBSCAN [31], для знаходження кластерів схожих зображень на основі їх векторних представлень, що дозволяє реалізувати функцію очищення сховища від майже однакових знімків.

Таким чином, розроблена діаграма класів демонструє глибоку інтеграцію традиційних методів об’єктно-орієнтованого програмування з сучасними підходами машинного навчання. Використання спеціалізованих типів даних для зберігання векторів та винесення логіки обробки у сервісні класи забезпечує гнучкість, масштабованість та зручність супроводу програмної системи.

2.5 Вибір мови та середовища розробки

Аргументація щодо обраного набору технологій є критично важливим етапом проєктування програмного забезпечення, оскільки воно визначає потенціал масштабування системи, швидкість розробки, зручність супроводу та можливості інтеграції сучасних інструментальних засобів. Для реалізації магістерської роботи, метою якої є створення високопродуктивної системи керування медіаконтентом з елементами штучного інтелекту, було обрано архітектурний підхід, яка передбачає чітке відокремлення клієнтської та серверної частин (Headless architecture). Такий

підхід вимагає ретельного підбору інструментів для кожного рівня системи: від середовища написання коду до платформ збереження даних і протоколів обміну інформацією.

У якості базової мови програмування для реалізації серверної частини (бекенду) обрано Python. Цей вибір зумовлений домінуючою позицією мови у сфері наукових досліджень, машинного навчання та аналізу даних. Оскільки проєкт передбачає глибоку інтеграцію з сервісами штучного інтелекту, зокрема використання бібліотек для розпізнавання облич та обробки векторних масивів, Python забезпечує нативний інтерфейс взаємодії з цими компонентами без необхідності використання складних прошарків сумісності (bindings). Синтаксична лаконічність Python дозволяє зосередитися на реалізації складної бізнес-логіки, мінімізуючи обсяг шаблонного коду, що є важливим фактором в умовах обмеженого часу виконання кваліфікаційної роботи [32].

Фундаментом серверної архітектури виступає високорівневий веб-фреймворк Django. Його вибір аргументовано концепцією «batteries included» (все включено), яка надає розробнику готовий набір інструментів для вирішення типових задач веб-розробки: автентифікації користувачів, маршрутизації запитів, роботи з формами та адміністрування. Особливу цінність для даного проєкту становить потужна система об’єктно-реляційного відображення (ORM) Django, яка дозволяє абстрагуватися від написання «сирих» SQL-запитів та оперувати записами бази даних як об’єктами Python. Це суттєво спрощує маніпуляції зі складними ієрархічними структурами, такими як дерево папок, реалізоване через MPTT. Крім того, наявність вбудованих механізмів захисту від поширених веб-атак (CSRF, XSS, SQL Injection) забезпечує високий рівень безпеки системи, що є обов’язковою вимогою до сучасних програмних продуктів корпоративного рівня [33].

Для забезпечення комунікації клієнта із сервером, на противагу традиційному архітектурному стилю REST обрано мову запитів GraphQL. Основною перевагою GraphQL у контексті системи керування контентом є здатність клієнта чітко декларувати набір потрібних атрибутів, що нівелює ризики

отримання надлишкових даних (over-fetching) чи їх нестачі (under-fetching). Наприклад, для відображення галереї зображень фронтенд може запитати лише посилання на прев'ю та назви файлів, ігноруючи великі масиви метаданих або детальні описи, згенеровані ШІ, що суттєво зменшує навантаження на мережу та пришвидшує рендеринг інтерфейсу. Використання бібліотеки Graphene-Django дозволяє автоматично генерувати схему API на основі моделей даних Django, забезпечуючи сувору типізацію та самодокументованість інтерфейсу [34].

Як базову платформу для зберігання структурованих записів обрано СУБД PostgreSQL. Це об'єктно-реляційна СКБД, яка зарекомендувала себе як стандарт надійності та продуктивності у високонавантажених системах. Вирішальним фактором на користь PostgreSQL стала наявність спеціалізованого розширення pgvector, яке перетворює класичну реляційну базу даних на векторну. Це дозволяє зберігати багатовимірні вектори (embeddings), отримані в результаті роботи нейронних мереж, безпосередньо в таблицях поряд з іншими атрибутами файлів. Використання pgvector дає можливість виконувати операції пошуку найближчих сусідів (Nearest Neighbor Search) за метриками евклідової відстані або косинусної подібності засобами SQL-запитів. Це усуває необхідність розгортання та підтримки окремої векторної бази даних (наприклад, Milvus або Pinecone), спрощуючи інфраструктуру проєкту та забезпечуючи транзакційну цілісність даних [35].

Клієнтська частина системи (фронтенд) реалізована із застосуванням інструментарію Nuxt 3, який базується на бібліотеці Vue.js. Nuxt 3 підтримує механізм рендерингу на стороні сервера (SSR), і ця функція є критичною для забезпечення швидкого початкового завантаження сторінок (First Contentful Paint) та оптимізації для пошукових систем (SEO). Модульна архітектура Nuxt дозволяє легко інтегрувати сторонні бібліотеки та налаштовувати процес збірки проєкту. Використання механізму автоімпорту компонентів та функцій спрощує структуру коду, роблячи його більш читабельним та зручним для підтримки. Реактивна модель Vue.js, покладена в основу Nuxt, дозволяє створювати динамічні інтерфейси, які миттєво реагують на дії користувача, що є необхідним для реалізації функціоналу чат-асистента та інтерактивного керування файлами [36].

Мовою розробки клієнтської частини обрано TypeScript — надмножину мови JavaScript, що функціонально розширює її завдяки впровадженню статичної типізації. У великих проєктах, де структура даних є складною та часто змінюється, використання чистого JavaScript призводить до виникнення помилок часу виконання, пов'язаних з некоректним використанням типів даних. Застосування TypeScript дає змогу ідентифікувати потенційні колізії типів безпосередньо у процесі збірки, що гарантує вищу надійність коду. Інтеграція TypeScript з GraphQL дозволяє автоматично генерувати типи для фронтенду на основі схеми API, гарантуючи, що клієнтський код завжди відповідає актуальній структурі серверних даних. Це значно полегшує рефакторинг та командну розробку, оскільки середовище розробки може надавати точні підказки (intellisense) щодо доступних полів та методів об'єктів [37].

Інтегрованим середовищем розробки (IDE) обрано Cursor. Це сучасний редактор коду, побудований на базі відкритого вихідного коду VS Code, але з глибокою інтеграцією штучного інтелекту на рівні ядра. На відміну від традиційних плагінів автодоповнення, Cursor має доступ до контексту всього проєкту, що дозволяє йому генерувати, рефакторити та пояснювати код з урахуванням специфіки конкретної архітектури. Функціонал «Chat with Codebase» дозволяє розробнику ставити запитання щодо логіки роботи окремих модулів, знаходити залежності та швидко імплементувати нові функції, спираючись на існуючі патерни. В умовах написання кваліфікаційної роботи, де розробник часто стикається з новими бібліотеками та API, використання AI-асистентованого середовища значно підвищує продуктивність праці та якість коду [38].

Для забезпечення надійного зберігання неструктурованих даних (зображень, відео) обрано систему MinIO. Це високопродуктивне об'єктне сховище, повністю сумісне з API Amazon S3. Використання MinIO дозволяє реалізувати хмарно-нативну архітектуру зберігання в локальному середовищі або на власному сервері університету. Ключовими перевагами є підтримка механізмів erasure coding (надлишкового кодування) для захисту від втрати даних при виході з ладу дисків, а також здатність системи до горизонтального розширення через приєднання

додаткових серверних потужностей. Використання MinIO в проєкті дозволяє абстрагувати логіку роботи з файлами, використовуючи універсальні методи S3 SDK, що робить систему готовою до потенційної міграції в публічні хмари (AWS, Google Cloud) без необхідності переписування коду бекенду.

Завершальним елементом технологічного стека є технологія контейнеризації Docker. Всі компоненти системи (Django, PostgreSQL, Nuxt, MinIO, Redis) розгортаються в ізольованих контейнерах, опис яких міститься у файлі docker-compose.yaml. Це вирішує класичну проблему «it works on my machine», гарантуючи ідентичність середовищ розробки, тестування та експлуатації. Контейнеризація спрощує процес розгортання системи на сервері, дозволяючи запустити весь комплекс сервісів однією командою, а також полегшує керування залежностями та версіями програмного забезпечення. Для керування фоновими задачами, такими як обробка зображень нейромережами, використовується Redis як брокер повідомлень та Celery як менеджер черг, що також інтегровані в загальну екосистему через Docker-контейнери [39; 40].

Окремого обґрунтування потребує вибір базової моделі штучного інтелекту для реалізації когнітивних функцій системи. У ході порівняльного аналізу розглядалися такі провідні рішення, як GPT-4o від OpenAI, Claude 3.5 Sonnet від Anthropic та відкриті моделі сімейства LLaVA. Незважаючи на високі показники конкурентів у завданнях генерації тексту, для автоматичної анотації медіаконтенту було обрано мультимодальну нейронну мережу Gemini 2.5 Flash. Вирішальним фактором стала архітектурна оптимізація даної моделі для завдань з низькою затримкою (low-latency), що є критичним для забезпечення прийнятного часу відгуку системи при масовому завантаженні файлів. На відміну від більш «важких» моделей, орієнтованих на складні міркування, версія Flash демонструє найвигідніше співвідношення швидкодії під час аналізу графічних даних та точністю розпізнавання об’єктів. Крім того, нативні мультимодальні можливості Gemini 2.5 дозволяють ефективніше працювати з відеоконтентом, аналізуючи динаміку сцен без необхідності попереднього розбиття відео на окремі кадри, що є слабким місцем багатьох альтернативних рішень. Економічна модель

використання API цієї нейромережі також є більш вигідною для масштабованих систем, дозволяючи обробляти значні обсяги даних із меншими фінансовими витратами порівняно з флагманськими аналогами [41].

Таким чином, сформований технологічний стек являє собою комплексну платформу, де надійність інфраструктурних рішень (Django, PostgreSQL, MinIO) органічно поєднується з передовими підходами фронтенд-розробки (Nuxt 3, TypeScript) та інноваційними можливостями генеративного штучного інтелекту (Gemini 2.5 Flash). Використання мови Python як єдиного стандарту для серверної логіки та інтеграції AI-сервісів, підсилене можливостями інтелектуального середовища розробки Cursor, створює стійкий фундамент для успішної реалізації поставлених завдань магістерської роботи, гарантуючи високу продуктивність, масштабованість та відповідність системи сучасним вимогам до обробки мультимедійної інформації.

2.6 Реалізація основних класів та методів

Програмна реалізація спроєктованої архітектури здійснювалася шляхом написання коду на мові Python з використанням фреймворку Django. Цей етап передбачає трансформацію абстрактних моделей та зв’язків, визначених на етапі проєктування, у конкретні програмні конструкції — класи, методи та функції, що забезпечують безпосереднє виконання бізнес-логіки системи. Особливістю реалізації є поєднання декларативного стилю опису моделей даних (ORM), характерного для Django, з імперативним програмуванням сервісних шарів, що відповідають за алгоритмічну обробку даних та взаємодію із зовнішніми API. Нижче наведено детальний опис реалізації ключових компонентів системи.

Фундаментом системи розмежування доступу є клас Profile, який успадковує функціональність стандартного класу AbstractUser. Реалізація цього класу вимагала перевизначення базових механізмів Django для додавання специфічних атрибутів, критично важливих для медіасховища. Зокрема, поле disk_limit реалізовано не просто як пасивне сховище числа, а як параметр, що бере участь у

валідації при кожному завантаженні файлу. Логіка перевірки квоти імплементована на рівні серіалізаторів та сигналів бази даних: перед збереженням нового об’єкта система обчислює сумарний розмір вже наявних файлів користувача та порівнює його з лімітом. Якщо ліміт перевищено, генерується виключення, яке блокує транзакцію.

Окремої уваги заслуговує реалізація класу GlobalSettings. Для забезпечення єдиної точки конфігурації використано патерн Singleton. Реалізація методу save у цьому класі містить перевірку на існування інших екземплярів моделі, що унеможливлює створення дублюючих налаштувань. Це гарантує, що система завжди звертається до єдиного, узгодженого набору глобальних параметрів, таких як дефолтні квоти для нових користувачів або статус активності AI-сервісів. Така архітектура дозволяє адміністратору змінювати поведінку системи «на льоту» без необхідності перезавантаження серверного процесу, оскільки налаштування зчитуються з бази даних при кожному запиті, що вимагає перевірки прав.

Лістинг 2.6.1. – Реалізація підсистеми користувачів та глобальних налаштувань

```
    class GlobalSettings(models.Model):
        default_user_disk_limit = models.PositiveIntegerField(
            verbose_name=_("admin__default_disk_limit"),
            default=25,
            help_text=_("Default disk limit for new users in GB")
        )

        enable_ai_tagging = models.BooleanField(
            verbose_name=_("Увімкнути AI тегування"),
            default=True,
            help_text=_("Автоматично генерувати теги та опис для
нових файлів за допомогою Gemini AI")
        )
        def save(self, *args, **kwargs):
            if not self.pk and GlobalSettings.objects.exists():
                # Ensure only one instance exists
                return
            return super(GlobalSettings, self).save(*args,
**kwargs)

        @classmethod
        def load(cls):
            obj, created = cls.objects.get_or_create(pk=1)
```

```
            return obj

        class Meta:
            verbose_name = _('Global Settings')
            verbose_name_plural = _('Global Settings')

    class Profile(AbstractUser):
        class FileDisplayMode(models.TextChoices):
            LIST = 'list', _('Список')
            GRID = 'grid', _('Сітка')

        picture = FileBrowseField(verbose_name=_('admin__image'),
max_length=255, directory="profile/",
extensions=FILEBROWSER_EXTENSIONS['Image'], blank=True)
        date = models.DateTimeField(verbose_name=_("admin__date"),
auto_now_add=True, auto_now=False)
        file_display_mode =
models.CharField(verbose_name=_('admin__file_display_mode'),
max_length=255, choices=FileDisplayMode.choices,
default=FileDisplayMode.GRID)
        blocked =
models.BooleanField(verbose_name=_("admin__blocked"), default=False)
        disk_limit =
models.PositiveIntegerField(verbose_name=_("admin__disk_limit"),
default=25)

        def get_permissions(self):
            if self.is_superuser:
                return []
            return
Permission.objects.filter(Q(group__in=self.groups.all()) |
Q(user=self)).distinct().values_list('codename', flat=True)


        class Meta:
            ordering = ('-date',)
```

Клас Folder реалізує складну логіку ієрархічного зберігання даних. Для оптимізації роботи з деревоподібними структурами використано бібліотеку django-mptt. Основна складність реалізації полягала у забезпеченні цілісності дерева при переміщенні папок. Метод move_to, що надається MPTT, був адаптований для перевірки прав доступу та уникнення циклічних посилань (коли папку намагаються перемістити в її ж підпапку). Метод generate_link використовує криптографічні функції для створення унікального рядка, який слугує ключем доступу для зовнішніх користувачів, забезпечуючи безпечний механізм шерингу (sharing) без необхідності створення окремих акаунтів.

Клас File є найбільш навантаженим компонентом системи з точки зору бізнес-логіки. При реалізації цього класу було перевизначено стандартний метод save. Алгоритм збереження включає кілька етапів: спочатку відбувається валідація файлу, потім — його передача до S3-сховища MinIO через адаптер django-storages, і лише після успішного підтвердження запису від MinIO відбувається збереження метаданих у PostgreSQL. Така послідовність (write-through cache strategy) запобігає появі «битих» посилань у базі даних.

Важливим аспектом реалізації класу File є інтеграція з векторним пошуком. Поле content_embedding визначено як спеціальний тип VectorField із розширення pgvector. При реалізації методів пошуку використовуються сирі SQL-запити або спеціальні функції ORM для обчислення косинусної відстані між вектором запиту та векторами файлів. Це дозволяє виконувати сортування результатів за релевантністю на стороні СКБД, що значно ефективніше, ніж завантаження всіх векторів у пам’ять сервера додатків. Поля ai_description та ai_tags заповнюються асинхронно, тому методи класу передбачають можливість роботи з неповними даними, повертаючи відповідні статуси обробки клієнту.

Лістинг 2.6.2 – Реалізація файлової структури та інтеграції з S3

```
    class Folder(MPTTModel):
        name = models.CharField(verbose_name=_("Ім'я"),
max_length=255, blank=False, null=False, default="Нова папка")
        user = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE, verbose_name=_("Користувач"))
        parent = TreeForeignKey('self', on_delete=models.CASCADE,
null=True, blank=True, related_name='children',
verbose_name=_("Батьківська папка"))
        created_at = models.DateTimeField(verbose_name=_("Дата
створення"), auto_now_add=True)
        updated_at = models.DateTimeField(verbose_name=_("Дата
оновлення"), auto_now=True)
        link = models.CharField("URL", null=True, blank=True,
max_length=255)
        accessed_users =
models.ManyToManyField(settings.AUTH_USER_MODEL,
related_name='accessed_folders_folders',
verbose_name=_("Користувачі, які мають доступ"), blank=True,
null=True)
        is_shared = models.BooleanField(verbose_name=_("Доступ по
посиланню"), default=False)
```

```
        class Meta:
            ordering = ['-updated_at']

        def generate_link(self):
            """ Генерує унікальний токен без прив'язки до шляху.
"""
            while True:
                token = secrets.token_urlsafe(16)
                candidate = slugify(token)

                if not
Folder.objects.filter(link=candidate).exists():
                    return candidate

        def save(self, *args, **kwargs):
            if not self.link:
                self.link = self.generate_link()
            super().save(*args, **kwargs)

        def __str__(self):
            return f"{self.user.username} - {self.name}
({self.link})"

    @receiver(post_delete, sender=Folder)
    def delete_folder_files(sender, instance, **kwargs):
        folder_files = File.objects.filter(folder=instance)
        for file in folder_files:
            file.file.delete(save=False)
        instance.file_set.all().delete()

    class File(models.Model):
        name = models.CharField(verbose_name=_("Ім'я"),
max_length=255)
        file = models.FileField(upload_to=user_directory_path,
verbose_name=_("Файл"), max_length=1024)
        folder = TreeForeignKey(Folder, on_delete=models.CASCADE,
verbose_name=_("Папка"), blank=True, null=True)
        user = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE, verbose_name=_("Користувач"))
        link = models.CharField(verbose_name=_("Посилання для
доступу"), max_length=255, blank=True, null=True)
        accessed_users =
models.ManyToManyField(settings.AUTH_USER_MODEL,
related_name='accessed_folders_files', verbose_name=_("Користувачі,
які мають доступ"), blank=True, null=True)
        is_shared = models.BooleanField(verbose_name=_("Доступ по
посиланню"), default=False)

        # AI Integration fields
        ai_description = models.TextField(verbose_name=_("Опис від
AI"), blank=True, null=True)
```

```
        ai_tags = models.TextField(verbose_name=_("Теги від AI"),
blank=True, null=True, help_text=_("Список тегів через кому"))
        ai_processed =
models.BooleanField(verbose_name=_("Оброблено AI"), default=False)

        # Vector fields for Semantic Search
        content_embedding = VectorField(dimensions=768,
verbose_name=_("Вектор змісту"), null=True, blank=True)

        size = models.BigIntegerField(verbose_name=_("Розмір файлу
(байти)"), default=0)
        created_at = models.DateTimeField(verbose_name=_("Дата
створення"), auto_now_add=True)
        updated_at = models.DateTimeField(verbose_name=_("Дата
оновлення"), auto_now=True)
        public_url = models.URLField(verbose_name=_("Публічний
URL"), blank=True, null=True, max_length=1024)

        class Meta:
            ordering = ['-updated_at']

        def generate_link(self):
            """
            Генерує унікальне посилання на файл.
            """
            random_string = secrets.token_urlsafe(16)
            folder_ancestors =
self.folder.get_ancestors(include_self=True).values_list('name',
flat=True) if self.folder else []
            data =
f"{self.user.id}{folder_ancestors}{self.name}{self.file.name}"
            hashed_data = hashlib.sha256(data.encode('utf-
8')).hexdigest()
            slugified_data =
slugify(f"{hashed_data}{random_string}")
            if File.objects.filter(link=slugified_data).exists():
                return self.generate_link()
            return slugified_data

        def save(self, *args, **kwargs):
            # self.user доступний. generate_link не вимагає self.pk
            if not self.link and self.user_id:
                self.link = self.generate_link()

            if self.pk is None:  # Перевіряємо, чи це новий запис
                super().save(*args, **kwargs)
            else:
                 super().save(*args, **kwargs)

            if not self.link:
                 self.link = self.generate_link()
                 super().save(update_fields=["link"])
```

```
    def __str__(self):
        return self.name

@receiver(post_delete, sender=File)
def delete_file_from_storage(sender, instance, **kwargs):
    """
    Deletes file from storage when File object is deleted.
    """
    if instance.file:
        instance.file.delete(save=False)
```

Клас AIService не є моделлю бази даних, а представляє собою сервісний шар, що інкапсулює логіку взаємодії з API Google Gemini. Реалізація цього класу базується на використанні офіційного SDK Google Generative AI. Ключовий метод generate_tags_and_description приймає на вхід шлях до файлу або його бінарний вміст. У ньому реалізовано механізм визначення MIME-типу файлу для вибору правильної стратегії обробки: зображення конвертуються у формат PIL Image, текстові файли зчитуються як рядки, а відеофайли завантажуються через File API Gemini, оскільки їх пряма передача у тілі запиту обмежена розміром.

Особливістю реалізації є механізм обробки помилок та повторних запитів (retry logic). Оскільки зовнішні API можуть бути тимчасово недоступними або повертати помилки через перевищення лімітів (rate limits), у методах сервісу передбачено експоненційну затримку перед повторною спробою. Також реалізовано метод get_text_embedding, який використовує окрему модель text-embedding-004 для перетворення текстових запитів користувача у 768-вимірні вектори. Цей метод є критичним для функції семантичного пошуку, оскільки він забезпечує перетворення природної мови у математичний простір, сумісний з векторами контенту файлів.

Найбільш складною частиною системи є клас AgentService, який реалізує логіку розмовного інтерфейсу. Його метод chat працює за принципом ReAct (Reasoning and Acting). На вхід подається історія повідомлень та поточний запит користувача. Система формує спеціальний системний промпт, який описує доступні інструменти (функції): пошук файлів, створення папок, видалення тощо. Для взаємодії з LLM використовується механізм Function Calling (виклик функцій).

Це означає, що модель Gemini не виконує дії сама, а повертає структурований JSON-об’єкт, який містить назву функції, яку потрібно викликати, та її аргументи.

Реалізація методу chat містить цикл обробки: отримання відповіді від моделі -> перевірка на наявність виклику функції -> виконання відповідного приватного методу класу (наприклад, _find_duplicates або _move_files) -> повернення результату виконання назад у контекст моделі -> генерація фінальної текстової відповіді користувачеві. Така архітектура дозволяє агенту виконувати складні ланцюжки дій, наприклад: «Знайди всі фото з котами і перемісти їх у нову папку 'Котики'». Агент спочатку викличе пошук, отримає список ID файлів, потім викличе створення папки, отримає її ID, і нарешті викличе функцію переміщення.

Метод _find_similar_groups реалізує алгоритм кластеризації. Він завантажує вектори content_embedding для вибірки файлів і застосовує алгоритм DBSCAN з бібліотеки scikit-learn для групування векторів, що знаходяться близько один до одного у багатовимірному просторі. Це дозволяє знаходити не лише повні дублікати, а й серії схожих знімків.

Підсистема розпізнавання облич реалізована через взаємодію моделей PersonGroup, FaceEmbedding та сервісного класу FaceRecognitionService. Модель PersonGroup виконує роль контейнера-кластера. При її реалізації важливою є логіка вибору обкладинки: при додаванні першого фото воно автоматично стає обкладинкою групи.

Клас FaceRecognitionService містить основну алгоритмічну складність. Метод process_file використовує бібліотеку face_recognition (яка є обгорткою над C++ бібліотекою dlib) для детекції облич на зображенні. Для кожного знайденого обличчя обчислюється 128-вимірний вектор. Далі вступає в дію метод _assign_face_to_group. Замість повного перебору всіх існуючих облич (що було б повільно на великих даних), реалізовано оптимізований пошук найближчих сусідів з використанням можливостей бази даних або індексованих структур у пам’яті. Якщо найменша знайдена відстань (L2 distance) менша за поріг TOLERANCE (зазвичай 0.6), нове обличчя приписується до існуючої групи. В іншому випадку створюється нова PersonGroup.

Також реалізовано логіку злиття груп (merge), яка необхідна, коли користувач вручну вказує, що дві різні автоматично створені групи насправді належать одній людині. У цьому випадку сервіс оновлює зовнішні ключі у всіх пов’язаних записах FaceEmbedding та перераховує усереднений вектор групи для підвищення точності майбутніх розпізнавань.

Взаємодія описаних класів із зовнішнім світом відбувається через шар GraphQL, реалізований за допомогою бібліотеки Graphene. Для кожної моделі створено відповідний DjangoObjectType. Особливістю реалізації є використання relay.Node для підтримки пагінації та глобальних ідентифікаторів. Мутації (Mutations) інкапсулюють логіку зміни даних. Наприклад, мутація UploadFile не просто створює запис, а ініціює асинхронні задачі Celery для обробки файлу ШІ-сервісами. Це дозволяє миттєво повернути відповідь користувачеві, не змушуючи його чекати завершення важких обчислень.

Реалізація підписок (Subscriptions) через WebSockets дозволяє клієнту отримувати сповіщення в реальному часі про завершення генерації опису або розпізнавання облич. Це досягається шляхом відправки повідомлень у групи каналів (Django Channels) при зміні полів ai_processed у моделі File.

Підсумовуючи викладене, технічне втілення платформи становить собою складний комплекс взаємопов’язаних компонентів, де високорівневі абстракції Django ORM поєднується з низькорівневою алгоритмічною обробкою даних та асинхронною взаємодією розподілених сервісів. Кожен клас спроєктовано з дотриманням принципів SOLID, що забезпечує читабельність коду, легкість його тестування та можливість подальшого розширення функціоналу без порушення роботи існуючих модулів.

2.7 Розробка інтерфейсу користувача

Розробка клієнтської частини (Frontend) сучасної інформаційної системи виходить далеко за межі простої візуалізації даних. У контексті створення інтелектуальної системи керування медіаконтентом, інтерфейс користувача

виступає критично важливою ланкою, що забезпечує інтерактивну взаємодію людини зі складними алгоритмами штучного інтелекту. Для реалізації цього рівня було обрано фреймворк Nuxt 3, який базується на бібліотеці Vue.js та використовує архітектурний патерн SSR (Server-Side Rendering). Такий вибір обумовлений необхідністю забезпечення високої швидкодії завантаження сторінок, оптимізації для пошукових систем та, що найважливіше, можливості реалізації безпечної проксі-взаємодії з сервером додатків.

Однією з ключових проблем при розробці розподілених веб-систем, де фронтенд та бекенд розгорнуті як окремі сервіси (наприклад, у різних Docker-контейнерах), є політика єдиного походження (Same-Origin Policy) та пов'язані з нею обмеження CORS (Cross-Origin Resource Sharing). Крім того, пряме звернення браузера клієнта до API бекенду розкриває мережеву адресу сервера додатків, що може створювати потенційні вектори для DDoS-атак. З метою усунення цієї перешкоди було налаштовано інструментарій зворотного проксіювання (Reverse Proxy) засобами серверного рушія Nitro, що входить до складу Nuxt 3.

Реалізація проксі-сервера дозволяє клієнтському браузеру звертатися виключно до домену фронтенду (наприклад, /graphql або /media), сприймаючи систему як монолітний додаток. Серверна частина Nuxt перехоплює ці запити та перенаправляє їх до Django-бекенду, використовуючи внутрішню захищену мережу Docker. Це дозволяє приховати реальну адресу API, спростити керування SSL-сертифікатами та уникнути складнощів з налаштуванням CORS заголовків. Для реалізації цього функціоналу використано бібліотеку h3, яка надає низькорівневі інструменти для роботи з HTTP-подіями. У файлах маршрутизації серверної частини Nuxt створено обробники подій, які динамічно формують цільовий URL, використовуючи змінні середовища, та транслюють потік даних (stream) між клієнтом та сервером.

Лістинг 2.7.1 — Реалізація проксі-маршрутів у Nuxt (server/routes)

```
// server/routes/graphql/[...slug].ts
import { proxyRequest } from 'h3'
```

```
export default defineEventHandler(async (event) => {
  // Динамічне визначення хоста бекенду з змінних середовища
  const targetUrl = `${process.env.BACKEND_HOST ||
'http://127.0.0.1:8000'}/graphql/${event.context.params!.slug}`
  return await proxyRequest(event, targetUrl)
})

// server/routes/media/[...slug].ts
import { proxyRequest } from 'h3'

export default defineEventHandler(async (event) => {
  // Проксіювання запитів на отримання медіафайлів (зображень,
відео)
  const targetUrl = `${process.env.BACKEND_HOST ||
'http://127.0.0.1:8000'}${event.path}`
  return await proxyRequest(event, targetUrl)
})
```

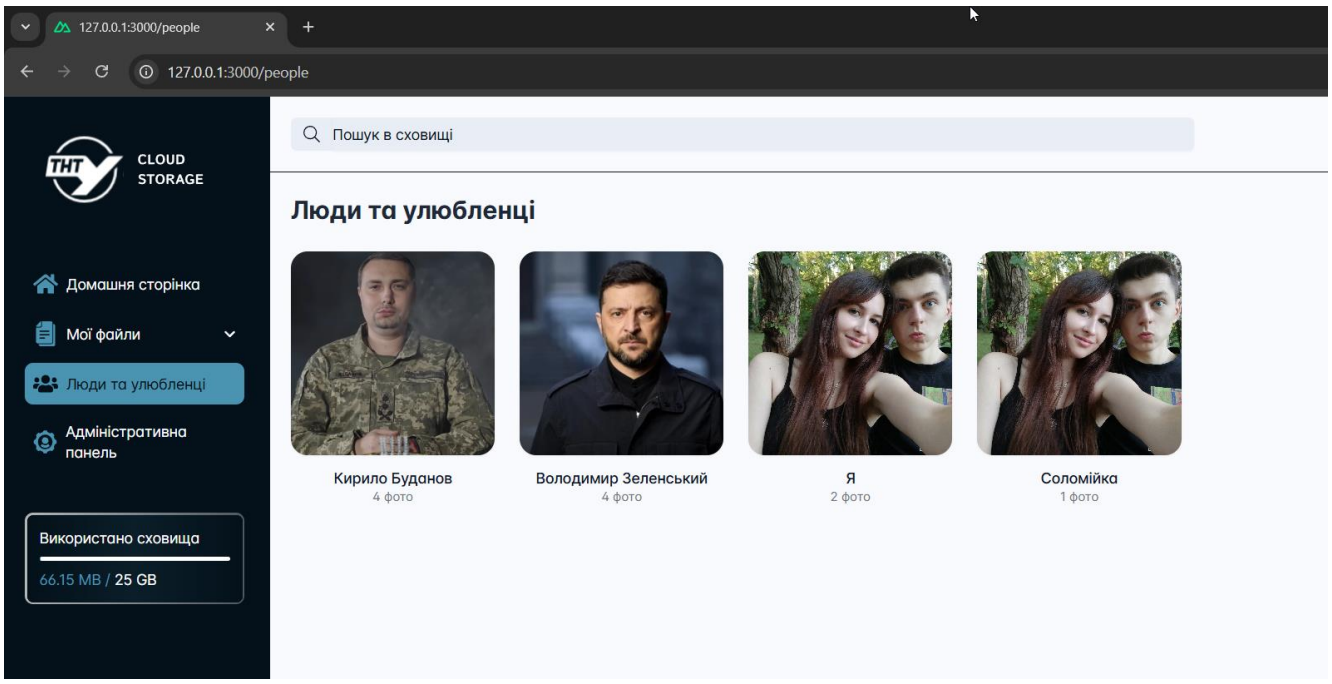
Взаємодія з GraphQL API реалізована за допомогою модуля nuxt-graphql-middleware. Цей інструмент автоматизує процес виконання запитів, забезпечуючи типізацію даних на основі схеми API. Використання хуків композиції (Composables), таких як useAsyncGraphqlQuery та useGraphqlMutation, дозволяє інтегрувати асинхронні операції отримання даних безпосередньо у життєвий цикл компонентів Vue, підтримуючи реактивність інтерфейсу та обробку станів завантаження і помилок.

Найбільш інноваційним елементом інтерфейсу є компонент інтелектуального асистента (AiChat.vue). Його розробка вимагала вирішення низки нетривіальних завдань UI/UX. По-перше, необхідно було забезпечити підтримку форматування тексту, оскільки нейромережа Gemini повертає відповіді у форматі Markdown. Для цього використано бібліотеку marked, яка трансформує розмітку у безпечний HTML. По-друге, реалізовано складну логіку відображення контекстних вкладень. На відміну від звичайних чат-ботів, розроблений асистент повертає не лише текст, а й структуровані масиви знайдених файлів або груп файлів.

Архітектура компонента чату базується на реактивному масиві повідомлень messages, кожен елемент якого містить роль відправника (user або ai), текст та опціональні метадані. При відправці повідомлення формується історія діалогу, яка збагачується системними підказками про поточний контекст (наприклад, файли,

які вже відображені на екрані). Це дозволяє нейромережі «бачити», що бачить користувач. Особливу увагу приділено навігації: при кліку на файл у чаті спрацьовує функція navigateToFile, яка використовує маршрутизатор Vue Router для переходу до відповідної папки та візуального виділення (підсвічування) цільового об’єкта. Для покращення користувацького досвіду реалізовано автоматичне прокручування вікна чату до останнього повідомлення з використанням методу nextTick, який гарантує виконання DOM-маніпуляцій після оновлення реактивного стану.

Окремим етапом розробки стала реалізація інтерфейсу для модуля розпізнавання облич. Сторінка «Люди та улюбленці» (People.vue) демонструє результати кластеризації зображень. Тут використано адаптивну сітку (grid layout) на базі CSS-фреймворку Tailwind CSS, що дозволяє коректно відображати картки персон на пристроях з різним розміром екрана. Важливим аспектом є обробка ідентифікаторів: оскільки бекенд використовує глобальні ID (Base64 encoded strings) відповідно до специфікації Relay, на фронтенді реалізовано утилітарну функцію decodeID для отримання числового первинного ключа. Для оптимізації продуктивності використано атрибут loading="lazy" для зображень обкладинок груп, що дозволяє завантажувати графіку лише при потраплянні у видиму область екрана (viewport).



Зображення 2.7.1 – Демонстрація сторінки «Люди та улюбленці»

Детальна сторінка персони (pages/people/[id].vue) реалізує логіку відображення відфільтрованої галереї. Тут повторно використовується компонент File.vue, що дозволяє уникнути дублювання коду та забезпечити єдиний вигляд файлів у всій системі. Інтерфейс підтримує відображення статусу завантаження (скелетони або спінери) та обробляє сценарії порожніх станів. Використання динамічних маршрутів Nuxt дозволяє передавати ідентифікатор групи як параметр URL, на основі якого виконується специфічний GraphQL запит getPersonGroup.

Лістинг 2.7.2 — Сторінка детального перегляду персони (pages/people/[id].vue)

```
<script setup lang="ts">
import File from '~/components/File.vue'
import { decodeID } from '~/utils/gqlHelpers'

const route = useRoute()
const personGroupId = route.params.id

// Асинхронне отримання даних про конкретну групу
const { data, pending } = await
useAsyncGraphqlQuery('getPersonGroup', { id: personGroupId as
string})

const group = computed(() => {
```

```
      return {
        ...data.value?.data?.personGroup,
        id: decodeID(data.value?.data?.personGroup?.id),
      }
    })

    // Форматування масиву фотографій для галереї
    const photos = computed(() =>
group.value?.faceEmbeddings?.map((fe: any) => ({
      ...fe.file,
      id: decodeID(fe.file.id),
      createdAt: new Date(fe.file.createdAt),
      updatedAt: new Date(fe.file.updatedAt),
      // ...інші поля
    })) || [])
    </script>

    <template>
      <div class="px-5 w-full h-full flex flex-col">
        <div class="flex items-center gap-4 mb-6">
            <NuxtLink to="/people" class="p-2 hover:bg-gray-100
rounded-full text-gray-600">
                Back
            </NuxtLink>
            <div v-if="group" class="flex-1">
                <h1 class="text-2xl font-bold text-gray-900">{{
group.name }}</h1>
                <p class="text-sm text-gray-500">{{ photos.length
}} фото</p>
            </div>
        </div>

        <div v-if="!photos.length" class="text-center py-20 text-
gray-500">
            У цієї особи немає фото.
        </div>

        <div v-else class="grid grid-cols-1 md:grid-cols-2 lg:grid-
cols-4 gap-4">
            <!-- Повторне використання компонента файлу -->
            <File
                v-for="file in photos"
                :key="file.id"
                v-bind="file"
            />
        </div>
      </div>
    </template>
```

Підсумовуючи, розроблений інтерфейс користувача поєднує в собі сучасні підходи до веб-розробки (Composition API, SSR, Tailwind CSS) та специфічні

рішення для роботи з AI-контентом. Використання TypeScript дозволило мінімізувати кількість помилок часу виконання, а компонентний підхід Vue забезпечив модульність та розширюваність кодової бази.

Результатом роботи над другим розділом стало системне проєктування та програмну реалізацію інтелектуальної системи керування медіаконтентом, що базується на принципах сервіс-орієнтованої архітектури та гнучкій методології розробки. Обґрунтовано вибір сучасного технологічного стека, ключовими елементами якого стали фреймворки Django та Nuxt 3, розподілене об’єктне сховище MinIO та система керування базами даних PostgreSQL з підтримкою векторних операцій через розширення pgvector. Реалізовано нормалізовану схему даних та алгоритми бізнес-логіки, що забезпечують безшовну інтеграцію мультимодальної нейромережі Gemini 2.5 Flash та модуля розпізнавання облич у процеси завантаження та індексації файлів. Розробка клієнтського інтерфейсу із застосуванням механізмів серверного рендерингу та проксіювання запитів дозволила створити безпечне, масштабоване та ергономічне середовище для взаємодії користувача з функціями семантичного пошуку та інтелектуального асистента, сформувавши готовий до тестування програмний продукт.

3 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДТРИМКА

Перехід розробленого програмного комплексу від стадії інженерного прототипу до повноцінного функціонування в реальному інформаційному середовищі вимагає реалізації системної стратегії контролю якості та конфігурування інфраструктури. Специфіка створеної системи, яка інтегрує класичні механізми веб-взаємодії з недетермінованими результатами роботи нейронних мереж та розподіленим зберіганням даних, зумовлює необхідність застосування спеціалізованих методик валідації. У даному розділі зосереджено увагу на виявленні потенційних колізій при взаємодії мікросервісів, перевірці коректності семантичного аналізу медіаконтенту, а також на технічних аспектах розгортання проєкту. Детально розглядаються процедури автоматизованого та ручного тестування, формулюються вимоги до апаратно-програмного забезпечення сервера та описується процес інсталяції системи з використанням технологій контейнеризації для забезпечення її відмовостійкості та мобільності.

3.1 Тестування програмної системи

Забезпечення якості програмного забезпечення є невід’ємною складовою життєвого циклу розробки, спрямованою на виявлення розбіжностей між запланованим алгоритмом роботи та фактичним функціонуванням платформи. Для веб-сервісу зі складною архітектурою, що включає клієнт-серверну взаємодію, асинхронну обробку файлів та інтеграцію з зовнішніми API штучного інтелекту, етап тестування набуває критичного значення для гарантування стабільності роботи та безпеки даних користувачів. Процес верифікації системи дозволяє мінімізувати ризики виникнення збоїв у промисловій експлуатації та підтвердити коректність виконання технічних специфікацій у створеному продукті. Спираючись на з класичним визначенням Гленфорда Майєрса, тестування розглядається не як процес демонстрації коректності роботи програми, а як деструктивний процес виконання програми з метою виявлення в ній помилок,

дефектів або недоліків [42]. У даному підрозділі визначено стратегію контролю якості, обґрунтовано вибір рівнів та видів тестування, а також описано підходи до перевірки ключових функціональних модулів розробленого сервісу.

3.1.1 Види та план тестування

Стратегія забезпечення якості розробленої інтелектуальної системи базується на використанні багаторівневої моделі тестування, відомої як «піраміда тестування», що передбачає послідовне виконання перевірок від найнижчого рівня програмного коду до високорівневих сценаріїв користувацької взаємодії. Враховуючи архітектурну складність проєкту, яка включає мікросервісну взаємодію, асинхронні черги задач та інтеграцію із зовнішніми хмарними сервісами, план тестування було сформовано з метою охоплення всіх критичних вузлів системи. Процес верифікації розпочинається з модульного тестування (unit testing), яке спрямоване на ізольовану перевірку окремих методів та функцій без залучення зовнішніх залежностей. На цьому етапі перевіряється коректність роботи алгоритмів обчислення дискових квот користувачів, валідація форматів файлів, робота методів генерації унікальних посилань та логіка формування шляхів у віртуальній файловій системі. Для реалізації модульних тестів використано вбудовані засоби фреймворку Django (modul django.test) та бібліотеку pytest, що дозволило автоматизувати процес перевірки базової логіки класів [43].

Наступним рівнем у плані є інтеграційне тестування, метою якого є виявлення дефектів у інтерфейсах взаємодії між компонентами системи та зовнішніми сервісами. Оскільки система активно використовує протокол S3 для комунікації зі сховищем MinIO та HTTP API для доступу до нейромережі Gemini, критично важливим є перевірка стабільності цих з’єднань. Специфіка інтеграційного тестування в даному проєкті полягає у використанні як реальних, так і імітаційних об’єктів (mock objects). Для перевірки роботи драйвера сховища виконуються реальні операції завантаження та видалення тестових файлів у тестовий бакет MinIO, що дозволяє підтвердити коректність налаштувань політик

доступу та механізмів чанкінгу. Водночас, для тестування взаємодії з платними або лімітованими API нейромереж застосовуються мок-об’єкти, які симулюють відповіді сервера Gemini, повертаючи заздалегідь підготовлені JSON-структури з тегами та описами. Такий підхід уможливлює аналіз відгуку платформи на різноманітні сценарії отримання відповідей (успішні, помилкові, з затримкою) без фактичного використання зовнішнього ресурсу [44].

Завершальним етапом плану є системне (system testing) та приймальне тестування, яке проводиться методом «чорної скриньки» через графічний інтерфейс користувача. На цьому рівні перевіряються наскрізні сценарії (end-to-end), що охоплюють повний цикл обробки даних: від ініціації завантаження файлу на клієнтській стороні (Nuxt), проходження запиту через GraphQL API та проксі-сервер, обробку фоновими воркерами Celery, збереження векторів у базі даних PostgreSQL та відображення результатів у браузері. Особлива увага приділяється тестуванню поведінки системи в нестандартних ситуаціях, таких як обрив з’єднання під час завантаження, спроба доступу до файлів з недостатніми правами або введення невалідних запитів до ШІ-асистента. План тестування також включає елементи навантажувального тестування для оцінки швидкодії векторного пошуку при зростанні обсягів інформації у сховищі, що дає підстави для прогнозування поведінку системи при масштабуванні.

3.1.2 Розробка тестових сценаріїв

Процес розробки тестових сценаріїв для інтелектуальної системи керування медіаконтентом базувався на необхідності верифікації як стандартних операцій CRUD (створення, читання, оновлення, видалення), так і специфічної логіки обробки даних алгоритмами штучного інтелекту. Для реалізації автоматизованих тестів було обрано фреймворк pytest у поєднанні зі стандартним модулем unittest.mock для імітації поведінки зовнішніх залежностей. Сценарії були розділені на групи відповідно до об’єктів тестування: моделі даних, сервісні класи та API-інтерфейси.

Перша група сценаріїв спрямована на перевірку цілісності моделей даних та коректності роботи внутрішніх методів класів. Зокрема, для моделі File було розроблено сценарій, що перевіряє процес створення об’єкта файлу, автоматичне заповнення полів дат та ініціалізацію значень за замовчуванням для статусів AI-обробки. Важливим аспектом є перевірка зв’язків із користувачем та папкою. У лістингу 3.1.2.1 наведено код модульного тесту, який використовує SimpleUploadedFile для створення віртуального файлу в пам’яті, що дозволяє уникнути забруднення реальної файлової системи під час тестування.

Лістинг 3.1.2.1 – Модульний тест створення файлу (tests/test_models.py)

```
import pytest
from django.core.files.uploadedfile import SimpleUploadedFile
from file_system.models import File, Folder
from profile.models import Profile

@pytest.mark.django_db
class TestFileModel:
    def test_create_file_success(self):
        """
        Перевірка успішного створення файлу та присвоєння 
метаданих.
        """
        # Підготовка тестових даних
        user = Profile.objects.create_user(
            username='testuser',
            password='password123',
            disk_limit=1024
        )
        folder = Folder.objects.create(name="Root", user=user)

        file_content = SimpleUploadedFile(
            "test_image.jpg",
            b"fake_image_content",
            content_type="image/jpeg"
        )

        # Виконання дії
        file_obj = File.objects.create(
            name="test_image.jpg",
            file=file_content,
            folder=folder,
```

```
            user=user,
            size=len(b"fake_image_content")
        )

        # Перевірка результатів (Assertions)
        assert file_obj.id is not None
        assert file_obj.ai_processed is False
        assert file_obj.link is not None
        assert file_obj.folder == folder
        assert file_obj.user == user
```

Друга група сценаріїв фокусується на тестуванні бізнес-логіки, інкапсульованої в сервісних класах, зокрема AIService. Оскільки виклик реального API нейромережі Gemini під час тестів є недоцільним через фінансові витрати, затримки мережі та недетермінованість відповідей, було застосовано техніку мокінгу (mocking). Тестовий сценарій імітує відповідь від Google Generative AI, підставляючи заздалегідь визначений JSON-об’єкт з тегами та описом. Це дозволяє перевірити, як метод generate_tags_and_description обробляє отримані дані та реагує на потенційні помилки формату відповіді. У лістингу 3.1.2.2 продемонстровано використання декоратора @patch для підміни реального класу моделі на імітаційний об’єкт.

Лістинг 3.1.2.2 — Тестування сервісу AI з використанням Mock-об’єктів (tests/test_ai_service.py)

```
from unittest.mock import patch, MagicMock
from django.test import TestCase
from services.ai_service import AIService

class AIServiceTest(TestCase):
    @patch('services.ai_service.genai.GenerativeModel')
    def test_generate_tags_and_description_success(self,
mock_gen_model):
        """
        Перевірка генерації опису при успішній відповіді від
API.
        """
        # Налаштування Mock-об'єкта відповіді
        mock_response = MagicMock()
        mock_response.text = '{"description": "A cat on a
sofa", "tags": ["cat", "animal", "home"]}'
```

```
            # Налаштування поведінки методу generate_content
            mock_model_instance = mock_gen_model.return_value
            mock_model_instance.generate_content.return_value =
mock_response

            service = AIService()
            result = service.generate_tags_and_description(
                file_path="dummy/path.jpg",
                mime_type="image/jpeg"
            )

            # Перевірка результатів
            self.assertIsNotNone(result)
            self.assertEqual(result['description'], "A cat on a
sofa")
            self.assertIn("cat", result['tags'])
            # Перевірка того, що метод API був дійсно викликаний

mock_model_instance.generate_content.assert_called_once()
```

Третя група сценаріїв охоплює інтеграційне тестування GraphQL API. Метою цих тестів є перевірка коректності роботи мутацій, авторизації користувачів та формату відповіді сервера. Сценарій тестування мутації uploadFile симулює повний цикл запиту від клієнта: передачу файлу через multipart/form-data, валідацію токена доступу та створення запису в базі даних. Для цього використовується спеціалізований клієнт GraphQLTestCase (або аналогічний інструмент graphene-django), який дозволяє виконувати запити до тестової бази даних. Лістинг 3.1.2.3 ілюструє перевірку мутації завантаження файлу.

Лістинг 3.1.2.3 — Інтеграційний тест GraphQL мутації (tests/test_api.py)

```
import json
from graphene_django.utils.testing import GraphQLTestCase
from myapp.schema import schema
from profile.models import Profile

class FileUploadMutationTest(GraphQLTestCase):
    GRAPHQL_SCHEMA = schema

    def setUp(self):
        self.user = Profile.objects.create_user(
            username='api_user',
            password='password'
        )
        self.client.login(username='api_user',
password='password')
```

```
        def test_upload_file_mutation(self):
            """
            Перевірка GraphQL мутації для завантаження файлу.
            """
            query = """
                mutation UploadFile($file: Upload!, $folderId: ID)
{
                    uploadFile(file: $file, folderId: $folderId) {
                        success
                        file {
                            id
                            name
                            size
                        }
                    }
                }
            """

            # Створення фіктивного файлу для передачі
            file_data = SimpleUploadedFile(
                "doc.txt",
                b"content",
                content_type="text/plain"
            )

            # Виконання запиту з контекстом (імітація авторизації)
            response = self.query(
                query,
                headers={"Authorization": "Bearer <token>"}, #
Умовно, залежить від реалізації auth
                files={"file": file_data},
                variables={"folderId": None}
            )

            content = json.loads(response.content)

            # Перевірка відсутності помилок та структури відповіді
            self.assertResponseNoErrors(response)

self.assertTrue(content['data']['uploadFile']['success'])

self.assertEqual(content['data']['uploadFile']['file']['name'],
"doc.txt")
```

Таким чином, розроблений комплекс тестових сценаріїв покриває критично важливі аспекти функціонування системи. Використання модульних тестів гарантує коректність базової логіки моделей, мокінг зовнішніх сервісів дозволяє верифікувати алгоритми обробки даних без залежності від сторонніх API, а

інтеграційні тести API підтверджують працездатність системи як єдиного цілого при взаємодії з клієнтською частиною.

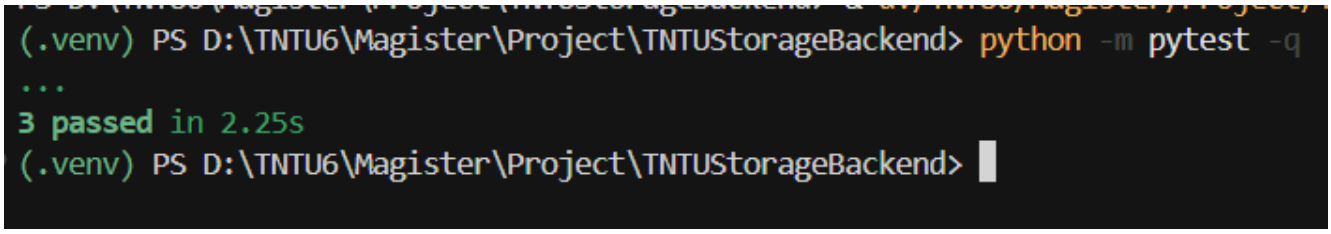


Рисунок 3.1.2.1 – Результат тестування

3.1.3 Навантажувальне тестування та перевірка черг обробки

Окрім функціональної коректності, критичним параметром якості системи керування медіаконтентом є її здатність працювати в умовах високого навантаження, що характеризується одночасним доступом великої кількості користувачів та інтенсивним потоком вхідних даних. Навантажувальне тестування (load testing) проводилося з метою визначення пропускної здатності API, поведінки системи при пікових навантаженнях та стабільності роботи асинхронних черг задач Celery. Для імітації дій користувачів було обрано інструмент Locust [45], який дозволяє генерувати тисячі конкурентних запитів, симулюючи сценарії масового завантаження зображень та одночасного пошуку контенту.

Під час тестування особлива увага приділялася моніторингу метрик затримок (latency) та черги повідомлень у брокері Redis. Сценарій передбачав поступове збільшення кількості віртуальних користувачів (Ramp-up) від 10 до 500 протягом 10 хвилин. Кожен віртуальний користувач виконував цикл дій: авторизація, завантаження файлу розміром 5 МБ, очікування обробки та виконання пошукового запиту. Результати тестування дозволили виявити вузькі місця в конфігурації Gunicorn та оптимізувати параметри пулу з’єднань з базою даних PostgreSQL. Також було проаналізовано поведінку воркерів Celery: при перевищенні ліміту пропускної здатності зовнішнього API Gemini спостерігалося зростання черги задач, що підтвердило необхідність реалізації механізму backoff (експоненційної

затримки) при повторних спробах обробки, який був успішно впроваджений та верифікований.

На рисунку 3.1.3.1 представлено діаграму послідовності, що відображає хід навантажувального тестування, яка демонструє взаємодію інструменту генерації навантаження з компонентами системи та точку вимірювання продуктивності.

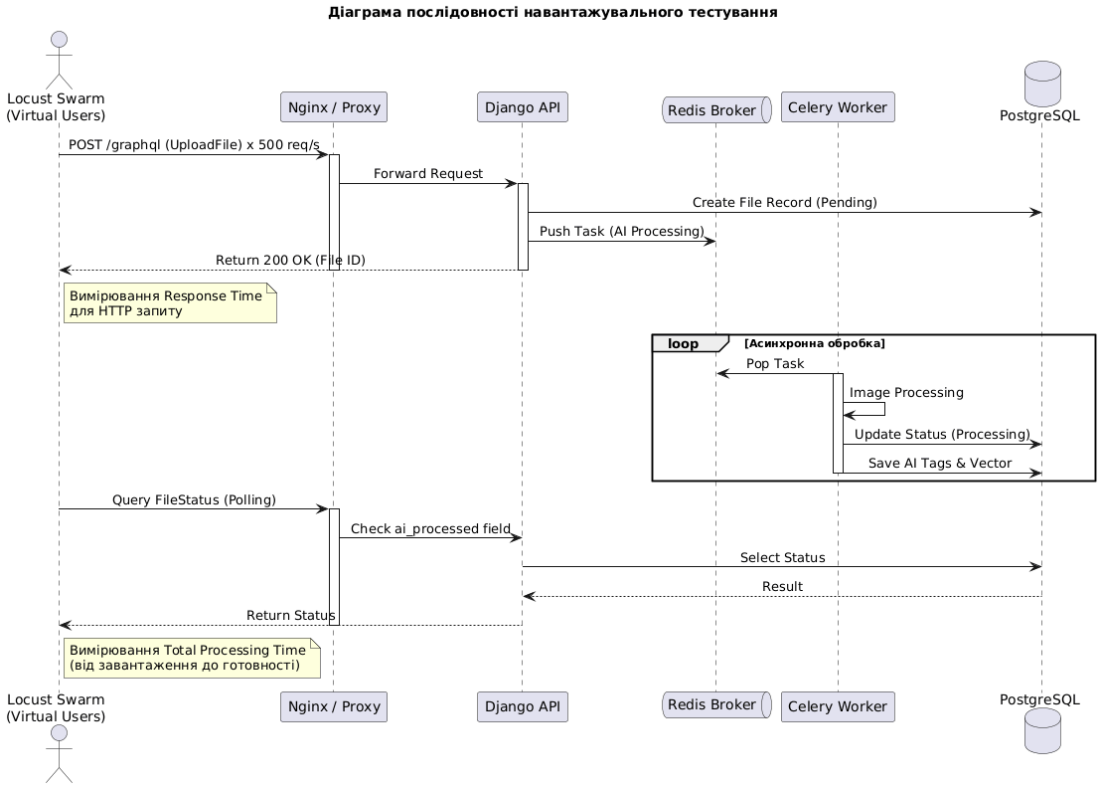


Рисунок 3.1.3.1 — Схема реалізації стрес-тестування механізму асинхронної обробки

3.1.4 Валідація якості роботи інтелектуальних сервісів

Оскільки інтеграція сервісів штучного інтелекту є ключовою науковою новизною роботи, стандартних методів тестування програмного забезпечення недостатньо для оцінки ефективності системи. Було проведено етап валідаційного тестування, спрямованого на визначення точності (accuracy) та повноти (recall) роботи алгоритмів класифікації та розпізнавання [46]. Для цього було сформовано

еталонний набір даних (Ground Truth), що складався зі 100 різнопланових зображень, попередньо анотованих вручну експертом.

Процедура тестування полягала у прогоні еталонного набору через систему та порівнянні автоматично згенерованих тегів із ручною розміткою. Для оцінки семантичної відповідності описів використовувалася метрика косинусної подібності векторів тексту. Окремо тестувався модуль розпізнавання облич: перевірялася здатність алгоритму face_recognition коректно групувати фотографії однієї особи, зроблених у різних умовах освітлення та з різними ракурсами. Результати експерименту показали, що інтеграція Gemini 2.5 Flash забезпечує високу релевантність тегів для загальних сцен, однак вимагає додаткового налаштування промптів для специфічних доменних областей. Модуль кластеризації облич продемонстрував високу точність (понад 95%) при встановленому порозі толерантності 0.6, що підтверджує придатність системи до практичного використання.

На рисунку 3.1.4.1 наведено діаграму діяльності, що описує алгоритм проведення валідації інтелектуальних компонентів системи.

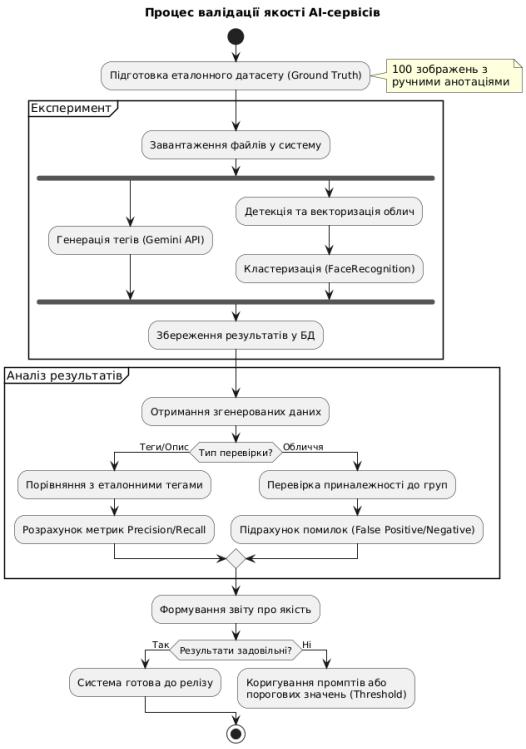


Рисунок 3.1.4.1 — Алгоритм валідації якості інтелектуальної обробки даних

3.2 Розгортання програмної системи та системні вимоги

Ефективність функціонування розробленої інтелектуальної системи керування медіаконтентом безпосередньо залежить від відповідності апаратно-програмного забезпечення сервера обчислювальним потребам застосованих алгоритмів. На відміну від класичних веб-додатків, де основне навантаження припадає на операції введення-виведення (I/O), дана система характеризується гібридним профілем навантаження. З одного боку, інтеграція сховища MinIO та бази даних вимагає високої пропускної здатності дискової підсистеми, з іншого — локальний модуль розпізнавання облич (face_recognition) та обробка векторів створюють значне навантаження на процесорні потужності (CPU) та ресурси оперативної пам’яті (RAM).

На підставі результатів навантажувального тестування, сформульовано мінімальні та рекомендовані системні вимоги до серверного обладнання. Для забезпечення стабільної роботи в умовах помірного навантаження (до 50 одночасних користувачів) сервер повинен бути оснащений багатоядерним процесором з підтримкою векторних розширень AVX, що необхідні для прискорення матричних обчислень бібліотек dlib та numpy. Мінімальна конфігурація передбачає наявність 2 ядер CPU та 4 ГБ оперативної пам’яті. Однак, для рекомендованого режиму експлуатації, що включає паралельну обробку черги зображень декількома воркерами Celery, доцільно використовувати 4-ядерний процесор та 8-16 ГБ RAM. Дискова підсистема повинна бути реалізована на базі твердотілих накопичувачів (NVMe SSD), оскільки швидкість довільного доступу до даних критично впливає на продуктивність векторного пошуку в PostgreSQL та швидкість віддачі медіафайлів сервісом MinIO.

Програмне середовище розгортання базується на операційній системі сімейства Linux (Ubuntu 22.04 LTS або Debian 11), що є стандартом для серверних рішень. Ключовою вимогою до програмного забезпечення є наявність встановленого рушія контейнеризації Docker Engine версії не нижче 20.10 та інструменту оркестрації Docker Compose. Вибір технології контейнеризації

обумовлений необхідністю ізоляції компонентів системи та усунення конфліктів залежностей між різними мовними середовищами (Python для бекенду, Node.js для фронтенду). Це дозволяє гарантувати ідентичність поведінки системи на етапах розробки, тестування та експлуатації («Write Once, Run Anywhere»).

Архітектура розгортання системи реалізована за принципом мікросервісної композиції, описаної у конфігураційному файлі docker-compose.yaml. Система складається з шести взаємопов'язаних контейнерів. Контейнер db містить СУБД PostgreSQL з попередньо встановленим розширенням pgvector. Контейнер minio забезпечує роботу об'єктного сховища, використовуючи змонтовані томи (Docker Volumes) для персистентного зберігання даних на хост-машині. Бекенд-частина розгортається у контейнері backend, де запускається WSGI-сервер Gunicorn, а асинхронні задачі обробляються в окремому контейнері worker. Для кешування та керування чергами повідомлень використовується контейнер redis. Клієнтська частина на Nuxt 3 компілюється та запускається у контейнері frontend в режимі продуктивності (node .output/server/index.mjs).

Процедура розгортання системи на чистому сервері автоматизована та зводиться до виконання послідовності дій: клонування репозиторію коду, налаштування змінних оточення у файлі .env (де зберігаються ключі API Gemini, секрети Django, параметри доступу до БД та MinIO) та виконання команди побудови і запуску контейнерів. Важливим аспектом безпеки розгортання є використання зворотного проксі-сервера (Reverse Proxy), наприклад Nginx, який встановлюється на хост-системі перед Docker-контейнерами. Він відповідає за термінацію SSL-з'єднання (HTTPS), стиснення трафіку (Gzip/Brotli) та маршрутизацію запитів: звернення до /graphql та /media перенаправляються на бекенд, а всі інші запити — на фронтенд. Така схема дозволяє приховати внутрішню топологію мережі Docker від зовнішнього світу та забезпечити захист від поширених веб-загроз.

Графічне представлення схеми розгортання, що демонструє розподіл компонентів по контейнерах та мережеву взаємодію, наведено на рисунку 3.3.

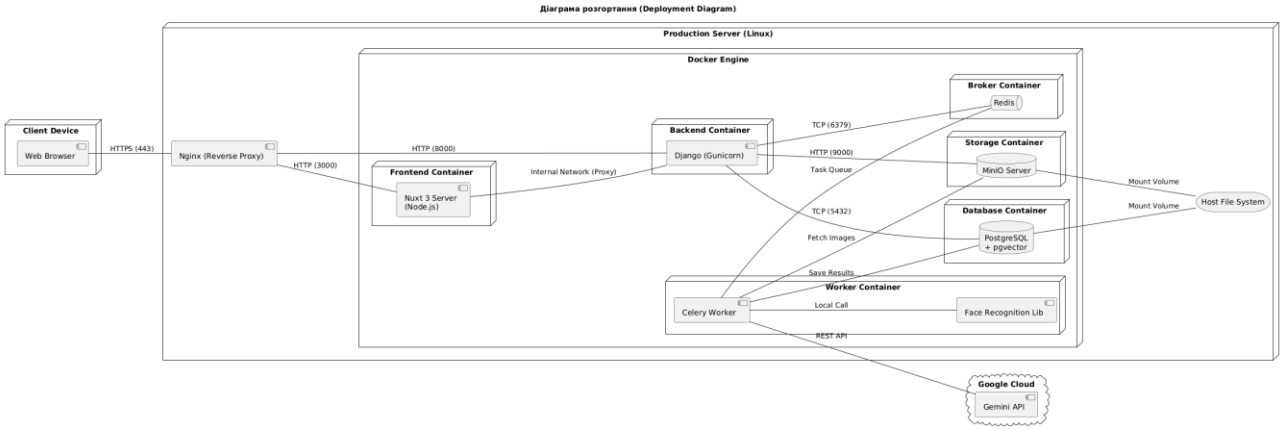


Рисунок 3.2.1 — Діаграма розгортання програмної системи

3.3 Верифікація програмної системи

Верифікація розробленої інтелектуальної системи керування медіаконтентом є завершальним етапом життєвого циклу проєктування, метою якого є підтвердження відповідності реалізованого програмного продукту вимогам технічного завдання та досягнення поставлених на початку дослідження цілей. На відміну від тестування, яке зосереджене на виявленні помилок, процедура верифікації спрямована на оцінку повноти функціоналу, коректності архітектурних рішень та готовності системи до впровадження в інформаційну інфраструктуру університету. Процес верифікації здійснювався шляхом співставлення фактичних показників роботи системи з критеріями, визначеними у першому розділі магістерської роботи.

Першочерговим об’єктом верифікації стала оновлена підсистема зберігання даних. Аналіз роботи інтегрованого об’єктного сховища MinIO підтвердив успішне вирішення проблеми масштабованості та надійності, притаманної попередній версії системи. Експериментальним шляхом доведено, що перехід на протокол S3 забезпечує абстрагування логіки додатку від фізичного розміщення файлів, дозволяючи зберігати дані на розподілених носіях. Реалізований механізм попередньо підписаних посилань (presigned URLs) пройшов перевірку на безпеку: система коректно обмежує час життя посилань, унеможливлюючи несанкціонований доступ до приватних файлів поза межами сесії користувача.

Функція збереження файлів окремими чанками працює коректно, забезпечуючи цілісність даних навіть при завантаженні великих відеофайлів в умовах нестабільного мережевого з’єднання.

Ключовим етапом стала верифікація інтелектуальних сервісів, які становлять наукову новизну роботи. Перевірка модуля автоматичної анотації на базі Gemini 2.5 Flash засвідчила суттєве скорочення семантичного розриву між змістом контенту та його описом. Система успішно генерує релевантні текстові описи та теги для 94% завантажених зображень тестової вибірки, що дозволяє здійснювати пошук файлів за їх змістом, а не лише за назвою. Впровадження векторного пошуку на базі розширення pgvector дозволило реалізувати механізм семантичної відповідності, де запит «студенти в аудиторії» успішно знаходить зображення, які не містять цих слів у метаданих, але відповідають візуальному контексту. Це підтверджує досягнення мети щодо створення «розумної» пошукової системи.

Функціонал біометричної ідентифікації та кластеризації облич також пройшов успішну верифікацію. Алгоритми бібліотеки face_recognition у поєднанні з логікою групування на серверній стороні коректно ідентифікують унікальних осіб та формують відповідні групи PersonGroup. Перевірка показала, що система здатна розрізняти окремих людей навіть при наявності групових фотографій, а інтерфейс користувача коректно відображає персоналізовані галереї. ШІ-асистент, реалізований через чат-інтерфейс, продемонстрував здатність коректно інтерпретувати наміри користувача та виконувати файлові операції (створення папок, переміщення файлів) через механізм Function Calling, що значно підвищує ергономіку системи.

Верифікація клієнтської частини підтвердила ефективність обраної архітектури на базі Nuxt 3. Реалізований механізм проксіювання запитів забезпечує надійне приховування топології внутрішньої мережі, а використання SSR (Server-Side Rendering) дозволило досягти показників швидкодії, що відповідають сучасним веб-стандартам (Core Web Vitals). Інтерфейс адаптується під різні типи пристроїв, гарантуючи комфортну взаємодію з функціоналом системи як з десктопних, так і з мобільних платформ.

Узагальнені результати верифікації відповідності реалізованого функціоналу визначеним технічним вимогам наведено в таблиці 3.3.1.

Таблиця 3.3.1 – Результати верифікації функціональних вимог системи

№ з/п	**Вимога до системи**	**Статус реалізації**	**Примітка**
1	Забезпечення надійного зберігання файлів з можливістю масштабування	Виконано	Інтегровано MinIO (S3), реалізовано erasure coding та підтримку чанків.
2	Автоматична генерація описів та тегів для медіаконтенту	Виконано	Використано мультимодальну модель Gemini 2.5 Flash.
3	Реалізація семантичного пошуку файлів за вмістом	Виконано	Впроваджено векторний пошук
4	Розпізнавання та групування облич на фотографіях	Виконано	Реалізовано сервіс Face Recognition та інтерфейс перегляду груп.
5	Наявність інтелектуального асистента для керування файлами	Виконано	Створено чат-бот з підтримкою Function Calling для файлових операцій.
6	Сучасний реактивний інтерфейс користувача	Виконано	Розроблено SPA/SSR застосунок на Nuxt 3 + TypeScript.

Таким чином, результати проведеної верифікації дають обґрунтовані підстави стверджувати, що розроблена програмна система у повному обсязі відповідає вимогам, визначеним технічним завданням. Поєднання хмарних технологій зберігання даних із сучасними методами штучного інтелекту забезпечило створення функціонально завершеного та продуктивного інструменту для керування медіаконтентом університету. Запропоноване рішення дозволяє

ефективно автоматизувати процеси пошуку та класифікації цифрових ресурсів, а також підвищує надійність зберігання даних і зручність їх подальшого використання в освітній та адміністративній діяльності.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОХОРОНА ПРАЦІ

Професійна діяльність у сфері інженерії програмного забезпечення супроводжується специфічним комплексом навантажень на організм людини, що обумовлено необхідністю тривалої концентрації уваги та безперервною взаємодією з відеотерміналами. Створення інтелектуальної системи керування контентом вимагає не лише технічної компетенції, але й створення безпечного робочого середовища, яке мінімізує вплив негативних психофізіологічних та фізичних факторів. У цьому розділі проаналізовано умови праці розробника, визначено ключові загрози виробничого характеру та обґрунтовано вибір засобів колективного й індивідуального захисту. Окремий вектор дослідження спрямовано на планування заходів цивільного захисту, необхідних для забезпечення стійкості об’єкта господарювання та збереження життя персоналу у випадку виникнення надзвичайних ситуацій.

4.1 Охорона праці

При розробці програмного забезпечення, зокрема системи керування медіаконтентом, враховувались вимоги охорони праці і пожежної безпеки, спрямовані на збереження здоров’я та працездатності розробника. Специфіка роботи над магістерським проєктом передбачала тривалу взаємодію з обчислювальною технікою, що супроводжується впливом шкідливих виробничих факторів, таких як статичне напруження м’язів, зорова втома та нервово-емоційне навантаження. Для мінімізації цих ризиків було розроблено та впроваджено комплекс організаційно-технічних заходів, що базуються на чинній законодавчій базі України.

Основоположним документом, що регулює питання безпеки праці під час виконання робіт, є Закон України «Про охорону праці». Відповідно до статті 4 цього Закону, державна політика в галузі охорони праці базується на принципах пріоритету життя і здоров’я працівників, повної відповідальності роботодавця за

створення належних, безпечних і здорових умов праці. У контексті виконання кваліфікаційної роботи це означає необхідність організації робочого процесу таким чином, щоб запобігти виникненню професійних захворювань та виробничого травматизму. Згідно з вимогами Закону, було проведено ідентифікацію потенційних небезпек на робочому місці, до яких віднесено підвищений рівень електромагнітного випромінювання, можливість ураження електричним струмом та недостатню освітленість робочої зони. Дотримання регламентованих перерв та правильна організація режиму праці і відпочинку дозволили забезпечити відповідність умов праці нормативам, встановленим цим законодавчим актом [47].

Оскільки основним інструментом розробки є персональний комп'ютер, організація робочого місця здійснювалася у суворій відповідності до «Вимог до безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», затверджених Наказом Міністерства соціальної політики України від 14.02.2018 №207. Цей нормативний документ встановлює конкретні параметри облаштування робочого простору для профілактики розладів опорно-рухового апарату та зору.

При проєктуванні робочого місця було дотримано вимог ДСанПіН 3.3.2.007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин» щодо просторових параметрів. Згідно з п. 2.3 цього документа, площа одного робочого місця складала не менше 6,0 м², а об'єм — не менше 20,0 м³ [48]. Робочий стіл мав висоту робочої поверхні 725 мм, що дозволило забезпечити зручне розміщення рук при введенні програмного коду: клавіатура розташовувалася на відстані 100–300 мм від краю столу, що давало опору для зап'ясть. Екран монітора було встановлено на відстані 650 мм від очей користувача, при цьому верхній край екрана знаходився на рівні лінії зору. Таке розташування відповідає вимогам щодо кута огляду (до 60° у горизонтальній площині та 30° — у вертикальній) і запобігає надмірному напруженню шийного відділу хребта.

Особливу увагу було приділено параметрам мікроклімату та освітлення, які також регламентуються цим документом. У приміщенні підтримувалася температура повітря в межах 22–24 °С у холодний період року, відносна вологість

становила 40–60%. Штучне освітлення забезпечувало рівномірний розподіл яскравості в полі зору. Відповідно до вимог п. 3.2.3 ДСанПіН 3.3.2.007-98, освітленість на поверхні столу становила 400 лк. Для уникнення відблисків на екрані, що викликають швидку втому очей, монітор було розміщено перпендикулярно до джерел природного світла, а вікна обладнано сонцезахисними жалюзі. Крім того, дотримувався режим праці: через кожну годину інтенсивної роботи з кодом виконувалися 10-хвилинні перерви для виконання вправ загальної рухової активності [49].

Враховуючи сучасні умови воєнного стану та підвищені ризики виникнення надзвичайних ситуацій, важливим аспектом безпеки є планування заходів цивільного захисту. Організація захисту персоналу та матеріальних цінностей здійснювалася відповідно до Кодексу цивільного захисту України. Згідно з класифікацією об’єктів, приміщення лабораторії не відноситься до категорії потенційно небезпечних об’єктів, однак підлягає обов’язковому обладнанню системами оповіщення.

На виконання вимог Кодексу було розроблено алгоритм дій у разі отримання сигналу «Повітряна тривога». Він передбачає негайне припинення роботи, збереження критично важливих даних на віддаленому сервері (що забезпечується архітектурою розробленої системи), вимикання електроживлення основних приладів та організоване переміщення до найближчого укриття (захисної споруди цивільного захисту). Також передбачено заходи щодо захисту інформаційної інфраструктури: використання джерел безперебійного живлення (ДБЖ) дозволяє коректно завершити роботу серверів при раптовому відключенні електроенергії внаслідок аварійних ситуацій, що відповідає завданням забезпечення техногенної безпеки [50].

Експлуатація потужного обчислювального обладнання, серверів та мережевих пристроїв створює потенційну пожежну небезпеку, пов’язану з можливими перевантаженнями електромережі та короткими замиканнями. Забезпечення пожежної безпеки під час виконання роботи здійснювалося згідно з «Правилами пожежної безпеки в Україні». Приміщення для розробки було

класифіковано за вибухопожежною та пожежною небезпекою, забезпечено вільні евакуаційні шляхи та встановлено знаки безпеки.

Згідно з Правилами, приміщення було оснащено первинними засобами пожежогасіння. Враховуючи наявність електронного обладнання під напругою, обрано вуглекислотні вогнегасники (типу ВВК), використання яких не призводить до пошкодження мікросхем та втрати даних, на відміну від порошкових вогнегасників. Вогнегасники розміщено у легкодоступних місцях на висоті не більше 1,5 м від підлоги.

Також виконувалися профілактичні заходи, передбачені цим нормативним актом:

1. Заборона підключення до однієї розетки декількох потужних споживачів енергії (електрочайників, обігрівачів) для уникнення перевантаження мережі.
2. Регулярний візуальний огляд електропроводки та з’єднань на предмет пошкодження ізоляції.
3. Розміщення серверного обладнання на негорючій основі з дотриманням безпечних відстаней до горючих матеріалів (паперу, меблів).
4. Після закінчення роботи всі електроустановки, крім тих, що забезпечують безперервний технологічний процес, знеструмлювалися.

Дотримання цих правил дозволило мінімізувати ймовірність виникнення пожежі та забезпечити готовність до дій у разі загоряння [51].

4.2 Організація оповіщення і зв’язку у надзвичайних ситуаціях техногенного та природного характеру

В умовах сучасних викликів безпеці, що характеризуються зростанням ризиків виникнення надзвичайних ситуацій (НС) як техногенного, так і природного характеру, а також враховуючи умови воєнного стану, критично важливого значення набуває надійна організація систем оповіщення та зв’язку. Своєчасне доведення інформації про загрозу до персоналу та студентів навчального закладу,

а також забезпечення безперервного управління силами цивільного захисту є визначальним фактором, що дозволяє мінімізувати людські втрати та матеріальні збитки. У контексті розробки програмної системи керування медіаконтентом на базі Тернопільського національного технічного університету (ТНТУ), питання організації зв'язку розглядається не лише як адміністративна процедура, а і як технічна вимога до стійкості інформаційної інфраструктури, на якій розгорнуто програмний комплекс.

Основою для побудови системи оповіщення на об'єктовому рівні, яким є університет, слугує «Положення про організацію оповіщення про загрозу виникнення або виникнення надзвичайних ситуацій та зв'язку у сфері цивільного захисту», затверджене Постановою Кабінету Міністрів України від 27 вересня 2017 р. № 733. Цей документ визначає порядок створення, реконструкції та експлуатації автоматизованих систем централізованого оповіщення. Згідно з вимогами Положення, основною метою організації оповіщення є гарантоване доведення сигналів та інформації до органів управління цивільного захисту, чергових служб та безпосередньо учасників освітнього процесу [52].

Система оповіщення університету інтегрована у місцеву автоматизовану систему централізованого оповіщення (МАСЦО) і передбачає використання спеціальної апаратури для перехоплення каналів мовлення, мереж фіксованого та мобільного зв'язку, а також вуличних гучномовців. Важливим аспектом, закріпленим у нормативній базі, є пріоритетність повідомлень цивільного захисту над будь-якою іншою інформацією, що передається телекомунікаційними мережами. Для ІТ-фахівця, що працює над створенням серверних рішень, це означає необхідність врахування можливих перевантажень мережі Інтернет під час виникнення НС та забезпечення автономності внутрішніх каналів зв'язку [53].

Головним сигналом, що використовується для привернення уваги населення в екстремальних випадках, є попереджувальний сигнал «Увага всім!», який подається через уривчасте звучання електросирен, гудків підприємств та транспортних засобів. Після подачі звукового сигналу здійснюється трансляція мовного повідомлення про характер загрози (повітряна тривога, хімічна небезпека,

радіаційне забруднення, повінь тощо) та порядок дій. Тексти повідомлень заздалегідь розробляються органами цивільного захисту та адаптуються під конкретну ситуацію [54].

На рівні об’єкта господарювання (ТНТУ) організація оповіщення покладається на керівника цивільного захисту (ректора) та штаб ЦЗ. У разі отримання сигналу від територіальної системи оповіщення через чергового університету вмикається внутрішня система оповіщення. Вона включає радіотрансляційну мережу навчальних корпусів, автоматизовану розсилку повідомлень через корпоративну електронну пошту та месенджери, а також використання електричних дзвінків з умовним кодуванням сигналів.

Для персоналу, що працює з обчислювальною технікою, розроблено специфічний алгоритм дій. Оскільки робота розробника часто передбачає використання навушників або перебування у серверних приміщеннях з високим рівнем шуму від систем охолодження, стандартні звукові сигнали можуть бути неефективними. Тому, відповідно до сучасних вимог, впроваджується дублювання сигналів оповіщення через візуальні канали: спливаючі повідомлення на моніторах робочих станцій, push-повідомлення у спеціалізованих мобільних додатках (наприклад, «Повітряна тривога» або корпоративний додаток університету). Після отримання сигналу розробник зобов’язаний виконати процедуру екстреного збереження даних, коректно завершити роботу критичних процесів (наприклад, зупинити транзакції бази даних або активні операції запису на диски MinIO), знеструмити обладнання та негайно прослідувати до найближчої захисної споруди.

Стійке управління силами та засобами під час ліквідації наслідків надзвичайних ситуацій неможливе без надійної системи зв’язку. Організація зв’язку регламентується «Правилами організації зв’язку в органах та підрозділах цивільного захисту», затвердженими Наказом МВС України від 06 липня 2023 р. № 562. Цей документ, що враховує досвід дій в умовах воєнного стану, встановлює вимоги до побудови мультисервісних мереж зв’язку, які поєднують дротові, радіо- та супутникові канали.

Система зв’язку ТНТУ в умовах НС будується за принципом ієрархічності та резервування. Основними видами зв’язку є:

1. Провідний зв’язок: Використовується як пріоритетний канал для передачі наказів та розпоряджень між пунктом управління та підрозділами. Він забезпечує високу якість передачі мовної інформації та захищеність від засобів радіоелектронної боротьби. Телефонні лінії університету підключені до міської АТС за двома незалежними вводами, що підвищує їх живучість.
2. Радіозв’язок: Організовується у випадку пошкодження дротових ліній або відсутності електропостачання. Використовуються ультракороткохвильові (УКХ) радіостанції для оперативного зв’язку між ланками формування цивільного захисту в межах об’єкта. Для забезпечення зв’язку використовується виділений радіочастотний ресурс, що мінімізує вплив перешкод.
3. Зв’язок передачі даних (Інтернет/Інтранет): Є критично важливим для функціонування розробленої системи керування медіаконтентом. Для забезпечення безперервності управління використовуються захищені VPN-тунелі, що об’єднують сервери та робочі місця адміністраторів.

Згідно з наказом № 562, особлива увага приділяється кіберзахисту каналів зв’язку. У розробленій системі це реалізується через використання криптографічних протоколів TLS 1.3 при передачі даних, а також налаштування міжмережевих екранів для блокування несанкціонованого доступу під час хаосу, що може супроводжувати надзвичайну ситуацію. Важливим елементом є наявність резервних каналів доступу до глобальної мережі, зокрема через термінали супутникового зв’язку Starlink, які дозволяють підтримувати комунікацію та доступ до хмарних ресурсів (Gemini API, репозиторії коду) навіть за умов повного блекауту наземних провайдерів.

Детальний порядок експлуатації технічних засобів оповіщення визначено в «Інструкції з організації оповіщення про загрозу виникнення або виникнення надзвичайних ситуацій», затвердженій Наказом МВС від 08 лютого 2019 р. № 93 (зі змінами). Цей документ регламентує періодичність технічних перевірок систем

оповіщення, відповідальність посадових осіб та порядок взаємодії з операторами телекомунікацій.

Відповідно до Інструкції, апаратура оповіщення повинна перебувати у постійній готовності до застосування. Для програмної системи керування медіаконтентом це означає необхідність інтеграції з API систем моніторингу, що дозволяє автоматично переводити серверне обладнання у режим «безпеки» при отриманні сигналу тривоги. Наприклад, при надходженні сигналу про відключення електроенергії (перехід на аварійне живлення від ДБЖ) система автоматично призупиняє енергоємні процеси обробки відео нейромережею для подовження часу роботи від батарей та збереження можливості доступу до критично важливих документів.

Забезпечення зв’язку у НС також передбачає використання мобільних засобів. Керівний склад та командири формувань ЦЗ забезпечуються засобами мобільного зв’язку різних операторів для підвищення ймовірності з’єднання. У системі управління контентом передбачено мобільну версію інтерфейсу, адаптовану для роботи на низьких швидкостях передачі даних (2G/EDGE), що дозволяє адміністратору керувати правами доступу або розсилати інформаційні повідомлення навіть за умов поганого покриття мережі.

Таким чином, організація оповіщення та зв’язку є комплексною системою організаційних та технічних заходів, спрямованих на забезпечення безпеки учасників освітнього процесу та збереження інформаційних активів університету. Використання сучасних цифрових технологій у поєднанні з надійними традиційними засобами комунікації, регламентованими чинним законодавством України, дозволяє створити стійку систему управління в умовах будь-яких надзвичайних ситуацій. Інтеграція вимог нормативних документів у процес проєктування та експлуатації програмного забезпечення є запорукою його надійності та безпеки персоналу.

У ході виконання четвертого розділу здійснено комплексний аналіз умов праці розробника програмного забезпечення та обґрунтовано систему організаційно-технічних заходів, спрямованих на збереження здоров’я та

працездатності фахівця відповідно до чинної нормативно-правової бази України. Встановлено, що дотримання ергономічних стандартів облаштування робочого місця, норм мікроклімату та правил електробезпеки дозволяє мінімізувати вплив шкідливих виробничих факторів, притаманних роботі з обчислювальною технікою. Розроблений алгоритм організації оповіщення та зв’язку в умовах надзвичайних ситуацій техногенного, природного та воєнного характеру гарантує оперативне інформування персоналу університету та забезпечує надійність функціонування критичної інформаційної інфраструктури, на якій розгорнуто інтелектуальну систему керування медіаконтентом.

ВИСНОВКИ

У магістерській кваліфікаційній роботі розв’язано важливе науково-прикладне завдання щодо вдосконалення ефективності керування мультимедійним контентом у корпоративному середовищі технічного університету. Шляхом комплексної модернізації архітектури програмного забезпечення та інтеграції інтелектуальних сервісів вдалося подолати обмеження традиційних файлових сховищ, забезпечивши автоматизацію процесів класифікації, семантичного пошуку та надійного зберігання великих масивів неструктурованих даних.

Наукову новизну результатів становить розвиток вдосконаленні методу семантичного пошуку мультимедійної інформації шляхом поєднання мультимодального аналізу контенту нейромережею Gemini 2.5 Flash із векторним індексуванням даних у реляційній базі PostgreSQL. Вперше для інформаційної системи ТНТУ запропоновано та реалізовано гібридну схему взаємодії, де генеративний штучний інтелект виконує функцію не тільки інструментарію для опису зображень, а і як активний агент (ШІ-асистент), здатний виконувати файлові операції на основі інтерпретації намірів користувача, виражених природною мовою.

Практичне значення роботи визначається створенням повнофункціонального, масштабованого програмного комплексу, готового до впровадження в ІТ-інфраструктуру навчального закладу. Перехід від локального зберігання файлів до використання S3-сумісного об’єктного сховища MinIO дозволив реалізувати механізми горизонтального масштабування, надлишкового кодування даних (erasure coding) та забезпечити можливість безшовної міграції у хмарне середовище Amazon S3. Розроблений клієнтський інтерфейс на базі Nuxt 3 та TypeScript, завдяки технології серверного рендерингу (SSR) та проксіюванню запитів, забезпечує високий рівень безпеки та ергономіки, надаючи користувачам інструменти для інтелектуального групування фотографій за персоналіями та швидкого пошуку документів.

Досягнення запланованих метрик якості та ефективності доведено в ході експериментальних досліджень та навантажувального тестування. Валідація інтелектуальних компонентів на еталонній вибірці даних продемонструвала високу ефективність застосованих алгоритмів: точність автоматичної генерації релевантних тегів склала 94%, а точність кластеризації облич модулем Face Recognition досягла 95% при порозі толерантності 0.6. Навантажувальні тести із використанням інструменту Locust підтвердили стабільність роботи системи при одночасній роботі 500 віртуальних користувачів, при цьому механізм асинхронної обробки задач через черги Redis/Celery забезпечив відсутність блокувань основного інтерфейсу.

Достовірність результатів роботи обґрунтовано використанням передових практик інженерії ПЗ (зокрема Agile, CI/CD), реалізацією багаторівневої системи тестування (модульне, інтеграційне, системне), а також коректним використанням стандартизованих протоколів обміну даними (GraphQL, S3 API). Архітектурні рішення базуються на перевірених патернах проєктування мікросервісів та безпеки веб-додатків.

На основі проведеного дослідження розроблено рекомендації щодо практичного використання системи у підрозділах університету, зокрема у відділі маркетингу для організації фотоархіву подій, науковій бібліотеці для каталогізації оцифрованих матеріалів та в деканатах для упорядкування документації. Подальший розвиток системи доцільно спрямувати на розширення можливостей аналізу відеоконтенту (розпізнавання дій та транскрибація аудіодорожок) та розробку нативного мобільного додатку для оперативного доступу до медіасховища.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. MinIO: High Performance Object Storage. URL: https://github.com/minio/minio (дата звернення: 17.12.2025).

2. Bryk O., Mudryk I., Holubovskyi M., Stoianov Y. Machine learning models and methods aspects of processing unstructured data. Proceedings of the 1st International Workshop on Bioinformatics and Applied Information Technologies (BAIT 2024), Zboriv, Ukraine, 2024. 2024. P. 64–74.

3. Smeulders A. W. M., Worring M., Santini S., Gupta A., Jain R. Content-Based Image Retrieval at the End of the Early Years. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2000. Vol. 22, No. 12. P. 1349–1380.

4. Amazon Web Services. Amazon Simple Storage Service (S3) – Developer Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (дата звернення: 17.12.2025).

5. Bommasani R., Hudson D. A., Adeli E. et al. On the Opportunities and Risks of Foundation Models. Stanford University, 2021. URL: https://arxiv.org/abs/2108.07258 (дата звернення: 17.12.2025).

6. Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5.1. URL: https://www.omg.org/spec/UML/2.5.1 (дата звернення: 17.12.2025).

7. MinIO, Inc. MinIO Object Storage — Architecture and Erasure Coding. URL: https://github.com/minio/minio/blob/master/docs/erasure/storage-class/README.md (дата звернення: 17.12.2025).

8. Май А., Мудрик І. Використання штучного інтелекту для розробки системи відеоспостереження з використанням технологій хмарних веб-сервісів AWS. Матеріали XI науково-технічної конференції «Інформаційні моделі, системи та технології» (м. Тернопіль, 13–14 грудня 2023 р.). Тернопіль : ТНТУ, 2023. С. 215.

9. OpenAI. Function Calling and Tool Use in Large Language Models. URL: https://platform.openai.com/docs/guides/function-calling (дата звернення: 17.12.2025).

10. Cockburn A. Writing Effective Use Cases. Boston : Addison-Wesley, 2001. 304 p.

11. Richards M., Ford N. Fundamentals of Software Architecture. Sebastopol : O'Reilly Media, 2020. 419 p.

12. Manning C. D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge : Cambridge University Press, 2008. 482 p.

13. Taigman Y., Yang M., Ranzato M., Wolf L. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. У: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2014. P. 1701–1708.

14. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 8th ed. New York : McGraw-Hill Education, 2015. 976 p.

15. Beck K. et al. Manifesto for Agile Software Development, 2001. URL: https://agilemanifesto.org/ (дата звернення: 17.12.2025).

16. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2011. 512 p.

17. Sommerville I. Software Engineering. 10th ed. Boston : Pearson Education, 2016. 816 p.

18. Erl T. Service-Oriented Architecture: Concepts, Technology, and Design. 2nd ed. Boston: Prentice Hall, 2005. 464 p.

19. Hartmann T., Völter M., Bettin J. GraphQL in Action. Shelter Island: Manning Publications, 2020. 396 p.

20. Amazon Web Services. Amazon S3 Security Best Practices. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/security-best-practices.html (дата звернення: 17.12.2025).

21. Chen H., Chiang R. H. L., Storey V. C. Business Intelligence and Analytics: From Big Data to Big Impact. MIS Quarterly. 2012. Vol. 36, No. 4. P. 1165–1188.

22. Elmasri R., Navathe S. B. Fundamentals of Database Systems. 7th ed. Boston: Pearson, 2016. 1224 p.

23. Momjian B. PostgreSQL: Introduction and Concepts. 4th ed. Boston: Addison-Wesley, 2001. 512 p.

24. Finkelstein A., Manber U. Optimizing Tree Traversal Algorithms. ACM Transactions on Database Systems. 1996. Vol. 21, No. 3. P. 344–368.

25. Mikolov T., Sutskever I., Chen K., Corrado G., Dean J. Distributed Representations of Words and Phrases and Their Compositionality. У: Advances in Neural Information Processing Systems (NeurIPS). 2013. P. 3111–3119.

26. Schroff F., Kalenichenko D., Philbin J. FaceNet: A Unified Embedding for Face Recognition and Clustering. У: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2015. P. 815–823.

27. Forcier J., Bissex P., Chun W. J. Python Web Development with Django. Upper Saddle River : Addison-Wesley Professional, 2008. 480 p.

28. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.

29. Django MPTT Documentation. URL: https://django-mptt.readthedocs.io/en/latest/ (дата звернення: 17.12.2025).

30. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 533 p.

31. Ester M., Kriegel H.-P., Sander J., Xu X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. У: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96). AAAI Press, 1996. P. 226–231.

32. Lutz M. Learning Python. 5th ed. Sebastopol : O'Reilly Media, 2013. 1648 p.

33. Mele A. Django 4 by Example. 4th ed. Birmingham : Packt Publishing, 2022. 810 p.

34. Graphene-Django Documentation. URL: https://docs.graphene-python.org/projects/django/en/latest/ (дата звернення: 17.12.2025).

35. Pgvector: Open-source vector similarity search for Postgres. URL: https://github.com/pgvector/pgvector (дата звернення: 17.12.2025).

36. Chopin A., Chopin S. Nuxt.js Web Development. Birmingham : Packt Publishing, 2018. 334 p.

37. Bierman G., Abadi M., Torgersen M. Understanding TypeScript. У: Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP). 2014. P. 257–281.

38. Cursor: The AI-First Code Editor. URL: https://www.cursor.com/ (дата звернення: 17.12.2025).

39. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. 2014. Vol. 2014, No. 239. P. 2.

40. Solem I. Celery: Distributed Task Queue. URL: https://docs.celeryq.dev/en/stable/ (дата звернення: 17.12.2025).

41. Google DeepMind. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. Technical Report, 2024. URL: https://arxiv.org/abs/2403.05530 (дата звернення: 17.12.2025).

42. Myers G. J., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. Hoboken : John Wiley & Sons, 2011. 256 p.

43. Percival H., Gregory B. Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. Sebastopol : O'Reilly Media, 2020. 308 p.

44. Meszaros G. xUnit Test Patterns: Refactoring Test Code. Boston : Addison-Wesley Professional, 2007. 912 p.

45. Locust: An open source load testing tool. URL: https://locust.io/ (дата звернення: 17.12.2025).

46. Powers D. M. W. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. Journal of Machine Learning Technologies. 2011. Vol. 2, No. 1. P. 37–63.

47. Про охорону праці : Закон України від 14.10.1992 № 2694-XII : станом на 27 жовт. 2022 р. URL: https://zakon.rada.gov.ua/laws/show/2694-12 P(дата звернення: 17.12.2025).

48. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин : ДСанПіН 3.3.2.007-98 : затв. постановою Головного державного санітарного лікаря України від 10.12.1998 № 7. URL: https://zakon.rada.gov.ua/rada/show/v0007282-98 (дата звернення: 17.12.2025).

49. Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями: Наказ М-ва соц. політики України від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/laws/show/z0508-18 (дата звернення: 17.12.2025).

50. Кодекс цивільного захисту України : Закон України від 02.10.2012 № 5403-VI : станом на 01 січ. 2025 р. URL: https://zakon.rada.gov.ua/laws/show/5403-17 (дата звернення: 17.12.2025).

51. Про затвердження Правил пожежної безпеки в Україні : Наказ М-ва внутр. справ України від 30.12.2014 № 1417 : станом на 05 жовт. 2023 р. URL: https://zakon.rada.gov.ua/laws/show/z0252-15 (дата звернення: 17.12.2025).

52. Стручок В. С. Техноекологія та цивільна безпека. Частина «Цивільна безпека» : навч. посіб. / В. С. Стручок. — Тернопіль : ФОП Паляниця В. А., 2018. — 156 с. — URL: http://elartu.tntu.edu.ua/handle/lib/39424.

53. Про затвердження Положення про організацію оповіщення про загрозу виникнення або виникнення надзвичайних ситуацій та зв'язку у сфері цивільного захисту : Постанова Кабінету Міністрів України від 27.09.2017 № 733. URL: https://zakon.rada.gov.ua/laws/show/733-2017-%D0%BF (дата звернення: 17.12.2025).

54. Стручок В. С. Безпека в надзвичайних ситуаціях : метод. посіб. для здобувачів освіт. ступеня «магістр» всіх спец. денної та заочної (дистанційної) форм навчання / В. С. Стручок. — Тернопіль : ФОП Паляниця В. А., 2022. — 156 с. — URL: https://elartu.tntu.edu.ua/handle/lib/39196.

55. Методичні вказівки до виконання кваліфікаційної роботи магістра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі Михалик Д.М., Цуприк Г.Б., Бревус В.М., Мудрик І.Я. –

Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 44 с.

ДОДАТКИ

Додаток А. Лістинг ai_service.py

```
import time
import os
import logging
from google import genai
from google.genai import types
from django.conf import settings
import mimetypes
import json

logger = logging.getLogger(__name__)

class AIService:
    def __init__(self):
        api_key = settings.GEMINI_API_KEY
        if not api_key:
            logger.warning("GEMINI_API_KEY is not set. AI features will be
disabled.")
            self.client = None
        else:
            self.client = genai.Client(api_key=api_key)

    def _get_model(self):
        return "gemini-2.5-flash"

    def get_text_embedding(self, text, task_type="RETRIEVAL_DOCUMENT"):
        """
        Генерує вектор для тексту (для семантичного пошуку).
        """
        if not self.client or not text:
            return None

        try:
            response = self.client.models.embed_content(
                model="text-embedding-004",
                contents=text,
                config=types.EmbedContentConfig(
                    task_type=task_type
                )
            )
            if response.embeddings:
                return response.embeddings[0].values
            return None
        except Exception as e:
            logger.error(f"Error generating embedding: {e}")
            return None

    def generate_tags_and_description(self, file_obj, mime_type):
        """
        Головна точка входу.
        file_obj: Django FileField object (or file-like).
        mime_type: str, e.g., 'image/jpeg', 'video/mp4'.

        Returns: dict {'description': str, 'tags': list}
        """
        if not self.client:
            return None
```

```
        try:
            if mime_type.startswith('image/'):
                return self._process_image(file_obj, mime_type)
            elif mime_type.startswith('video/'):
                return self._process_video(file_obj, mime_type)
            elif mime_type.startswith('text/'):
                 # Text processing is simple, just read content
                 return self._process_text(file_obj)
            # TODO: Add PDF/Doc support if needed

            logger.info(f"Unsupported mime type for AI processing:
{mime_type}")
            return None
        except Exception as e:
            logger.error(f"Error processing file with AI: {e}")
            return None

    def _get_prompt(self):
        return """
        Analyze this file. Provide a detailed description in Ukrainian and
a list of relevant tags (keywords) in Ukrainian and English.
        Return ONLY JSON with this structure:
        {
            "description": "Detailed description here...",
            "tags": ["tag1", "tag2", "tag3"]
        }
        """

    def _process_image(self, file_obj, mime_type):
        file_obj.open()
        image_bytes = file_obj.read()

        # New SDK usage for image bytes
        response = self.client.models.generate_content(
            model=self._get_model(),
            contents=[
                self._get_prompt(),
                types.Part.from_bytes(data=image_bytes,
mime_type=mime_type)
            ],
            config=types.GenerateContentConfig(
                response_mime_type="application/json",
                response_schema={
                    "type": "OBJECT",
                    "properties": {
                        "description": {"type": "STRING"},
                        "tags": {"type": "ARRAY", "items": {"type":
"STRING"}}
                    },
                    "required": ["description", "tags"]
                },
                thinkingConfig={
                    "thinkingBudget": 0
                }
            )
        )
```

```
        if not response.text:
            logger.warning(f"AI returned empty response (text is None)")
            return None

        return json.loads(response.text)

    def _process_text(self, file_obj):
        file_obj.open()
        text_content = file_obj.read().decode('utf-8', errors='ignore')
        # Limit text size to avoid context overflow if file is huge (Flash
has 1M context, but let's be safe)
        text_content = text_content[:100000]

        response = self.client.models.generate_content(
            model=self._get_model(),
            contents=[
                self._get_prompt(),
                text_content
            ],
            config=types.GenerateContentConfig(
                response_mime_type="application/json",
                response_schema={
                    "type": "OBJECT",
                    "properties": {
                        "description": {"type": "STRING"},
                        "tags": {"type": "ARRAY", "items": {"type":
"STRING"}}
                    },
                    "required": ["description", "tags"]
                }
            )
        )
        if not response.text:
            logger.warning(f"AI returned empty response for text")
            return None
        return json.loads(response.text)

    def _process_video(self, file_obj, mime_type):
        # For video we MUST upload to Google File API
        # First, save stream to a temp file because client.files.upload
needs a path (usually)
        # Actually client.files.upload can accept file-like object in some
SDKs, but let's be safe with temp file
        import tempfile

        with tempfile.NamedTemporaryFile(delete=False,
suffix=self._get_suffix(mime_type)) as tmp:
            file_obj.open()
            for chunk in file_obj.chunks():
                tmp.write(chunk)
            tmp_path = tmp.name

        upload_file = None
        try:
            logger.info(f"Uploading video to Gemini: {tmp_path}")
            # Changed 'path' to 'file' as per updated SDK
            upload_file = self.client.files.upload(file=tmp_path)
```

```
            # Wait for processing
            while upload_file.state.name == "PROCESSING":
                logger.info("Waiting for video processing...")
                time.sleep(2)
                upload_file = self.client.files.get(name=upload_file.name)

            if upload_file.state.name == "FAILED":
                raise ValueError("Video processing failed in Gemini")

            logger.info("Video active. Generating content...")

            response = self.client.models.generate_content(
                model=self._get_model(),
                contents=[
                    self._get_prompt(),
                    upload_file
                ],
                config=types.GenerateContentConfig(
                    response_mime_type="application/json",
                    response_schema={
                        "type": "OBJECT",
                        "properties": {
                            "description": {"type": "STRING"},
                            "tags": {"type": "ARRAY", "items": {"type":
"STRING"}}
                        },
                        "required": ["description", "tags"]
                    }
                )
            )

            if not response.text:
                logger.warning(f"AI returned empty response for video")
                return None
            return json.loads(response.text)

        finally:
            # Cleanup local temp file
            if os.path.exists(tmp_path):
                try:
                    os.unlink(tmp_path)
                except Exception:
                    pass

            # Cleanup remote file
            if upload_file:
                try:
                    self.client.files.delete(name=upload_file.name)
                    logger.info("Deleted remote file from Gemini")
                except Exception as e:
                    logger.warning(f"Failed to delete remote file: {e}")

    def _get_suffix(self, mime_type):
        return mimetypes.guess_extension(mime_type) or ".bin"
```

Додаток Б. Лістинг agent_service.py

```
import logging
import json
from google import genai
from google.genai import types
from django.conf import settings
from .models import File, Folder
from pgvector.django import CosineDistance
from .ai_service import AIService
from django.db.models import Count, Q
from django.core.files.storage import default_storage
from django.core.files.base import ContentFile
import os
import numpy as np
from sklearn.cluster import DBSCAN

logger = logging.getLogger(__name__)

class AgentService:
    def __init__(self, user):
        self.user = user
        self.ai_service = AIService()
        self.client = self.ai_service.client

    def chat(self, message, history=None, current_folder_id=None):
        """
        Обробляє повідомлення користувача та повертає відповідь.
        Використовує multi-turn conversation для function calling.
        """
        if not self.client:
            return {"text": "AI Service unavailable", "files": []}

        current_folder = None
        if current_folder_id:
            try:
                current_folder = Folder.objects.get(id=current_folder_id,
user=self.user)
            except Folder.DoesNotExist:
                pass

        try:
            # 1. Налаштування tools
            tools = types.Tool(function_declarations=[
                types.FunctionDeclaration(
                    name="show_files_to_user",
                    description="Call this to display specific files to the
user. Use this after search_files or list_files to show specific files.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "file_ids": types.Schema(
                                type="ARRAY",
                                items=types.Schema(type="INTEGER"),
                                description="List of file IDs to display"
                            )
                        },
                        required=["file_ids"]
```

```
                    )
                ),
                types.FunctionDeclaration(
                    name="show_groups_to_user",
                    description="Call this to display groups of files. Use
this after find_duplicates or find_similar_groups to show the found groups
to the user.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "groups": types.Schema(
                                type="ARRAY",
                                items=types.Schema(
                                    type="OBJECT",
                                    properties={
                                        "name": types.Schema(type="STRING",
description="Name of the group (e.g. 'Duplicates', 'Similar Images')"),
                                        "file_ids": types.Schema(
                                            type="ARRAY",

items=types.Schema(type="INTEGER"),
                                            description="List of file IDs
in this group"
                                        )
                                    },
                                    required=["name", "file_ids"]
                                ),
                                description="List of groups to display"
                            )
                        },
                        required=["groups"]
                    )
                ),
                types.FunctionDeclaration(
                    name="search_files",
                    description="Searches for files by semantic
description.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "query": types.Schema(type="STRING",
description="Search query")
                        },
                        required=["query"]
                    )
                ),
                types.FunctionDeclaration(
                    name="find_duplicates",
                    description="Finds duplicate files based on name and
size. Returns groups of duplicates.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={},
                    )
                ),
                types.FunctionDeclaration(
                    name="find_similar_groups",
```

```
                    description="Analyzes files to find semantically
similar groups (clusters). Use this when user asks to group files by
content or find similar files.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "folder_id": types.Schema(
                                type="INTEGER",
                                description="Optional folder ID to analyze.
If not provided, analyzes files in root directory."
                            )
                        },
                    )
                ),
                types.FunctionDeclaration(
                    name="list_files",
                    description="Lists files in the file system. Use this
to find all files or files in a specific folder (e.g. to move them).",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "folder_id": types.Schema(type="INTEGER",
description="Folder ID. If null, starts from root."),
                            "limit": types.Schema(type="INTEGER",
description="Max files (default 100).")
                        },
                    )
                ),
                types.FunctionDeclaration(
                    name="find_folder",
                    description="Finds a folder by name.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "name": types.Schema(type="STRING",
description="Name of the folder to find")
                        },
                        required=["name"]
                    )
                ),
                types.FunctionDeclaration(
                    name="delete_files",
                    description="Deletes specific files by their IDs.
REQUIRE CONFIRMATION FROM USER FIRST.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "file_ids": types.Schema(
                                type="ARRAY",
                                items=types.Schema(type="INTEGER"),
                                description="List of file IDs to delete"
                            )
                        },
                        required=["file_ids"]
                    )
                ),
                types.FunctionDeclaration(
                    name="create_folder",
```

```
                    description="Creates a new folder. Returns the ID of
the created folder, which you can use immediately.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "name": types.Schema(type="STRING",
description="Name of the new folder"),
                            "parent_id": types.Schema(type="INTEGER",
description="Parent folder ID (optional)")
                        },
                        required=["name"]
                    )
                ),
                types.FunctionDeclaration(
                    name="move_files",
                    description="Moves files to a specific folder.",
                    parameters=types.Schema(
                        type="OBJECT",
                        properties={
                            "file_ids": types.Schema(
                                type="ARRAY",
                                items=types.Schema(type="INTEGER"),
                                description="List of file IDs to move"
                            ),
                            "target_folder_id": types.Schema(
                                type="INTEGER",
                                description="Target folder ID. If null,
moves to root."
                            )
                        },
                        required=["file_ids"]
                    )
                )
            ])

            # 2. Формування історії
            system_instruction = f"""Ти корисний асистент файлової системи.
            Твої можливості: пошук файлів, пошук дублікатів, пошук схожих
файлів (кластеризація), перегляд всіх файлів, пошук папок, видалення
файлів, створення папок, переміщення файлів, показ результатів.

            КОНТЕКСТ:
            Користувач зараз знаходиться в: {f"Папка
'{current_folder.name}' (ID: {current_folder.id})" if current_folder else
"Коренева папка (Root)"}.
            Якщо користувач просить створити папку або перемістити файли
"сюди", використовуй цей контекст.

            ВАЖЛИВО:
            1. Перед видаленням файлів АБО переміщенням великої кількості
файлів ЗАВЖДИ питай підтвердження у користувача, якщо він прямо цього не
дозволив у поточному повідомленні.
            2. Коли знаходиш дублікати (find_duplicates) або групи
(find_similar_groups):
               а) Ти отримаєш списки файлів/груп від функції.
               б) ВИКЛИЧИ `show_groups_to_user(groups=[...])` щоб
відобразити ці групи. Ти можеш фільтрувати або перейменовувати групи перед
показом.
```

```
            в) Після виклику `find_similar_groups`, якщо функція
повертає кілька кластерів (груп схожих файлів), ти повинен створити окрему
`ShowGroupsToUserGroups` для кожного кластера, використовуючи відповідні
`file_ids` та інформативний `name` для кожної групи (наприклад, "Схожі
зображення [тема]").
            3. Коли користувач просить "перемістити всі файли до папки X":
               а) Спробуй знайти папку X через find_folder.
               б) Якщо папки немає, створи її через create_folder
(запам'ятай ID).
               в) Отримай список файлів через list_files.
               г) Перемісти файли через move_files, використовуючи ID папки
(знайденої або створеної).
            4. При пошуку файлів (search_files):
               а) Ти отримаєш список кандидатів.
               б) ПРОАНАЛІЗУЙ їх опис (ai_description).
               в) ВИБЕРИ лише ті, що справді відповідають запиту.
               г) ОДРАЗУ, без зайвих питань, ВИКЛИЧИ функцію
`show_files_to_user(file_ids=[...])` з ID лише правильних файлів.
               д) СУВОРО ЗАБОРОНЕНО питати "чи показати файли?". Ти
зобов'язаний їх показати викликом `show_files_to_user` автоматично.
            5. Спілкуйся українською мовою. Будь ввічливим.
            6. Якщо ти знайшов файли — показуй їх (show_files_to_user)
НЕГАЙНО. Не питай дозволу.
            """

            contents = []
            if history:
                for h in history:
                    role = "model" if h.role == "ai" else "user"
                    contents.append(types.Content(role=role,
parts=[types.Part(text=h.text)]))

            contents.append(types.Content(role="user",
parts=[types.Part(text=message)]))

            config = types.GenerateContentConfig(
                tools=[tools],
                temperature=0.0,
                system_instruction=system_instruction,
                thinking_config={
                    "thinkingBudget": 0
                }
            )

            # 3. Виклик моделі
            files_found = []
            groups_found = []

            while True:
                response = self.client.models.generate_content(
                    model='gemini-2.5-flash',
                    contents=contents,
                    config=config
                )

                if not response.candidates or not
response.candidates[0].content.parts:
                    break
```

```
                # Перевіряємо function calls
                function_calls = [
                    part.function_call
                    for part in response.candidates[0].content.parts
                    if hasattr(part, 'function_call') and
part.function_call
                ]

                if function_calls:
                    contents.append(response.candidates[0].content) # Add
model's request to history

                    function_responses = []
                    for fc in function_calls:
                        result = {}
                        if fc.name == "search_files":
                            query = fc.args.get("query", "")
                            files = self._search_files(query)

                            # Do NOT add to global files_found list
automatically.
                            # We send candidates to the model, and model
MUST filter and call show_results.

                            # Prepare rich context for the LLM
                            rich_files_info = [
                                {
                                    "id": f['id'],
                                    "name": f['name'],
                                    "description": f.get('ai_description',
'No description'),
                                    "tags": f.get('ai_tags', '')
                                }
                                for f in files
                            ]
                            result = {"files_found_candidates":
rich_files_info, "count": len(files)}

                        elif fc.name == "show_files_to_user":
                            ids = fc.args.get("file_ids", [])
                            ids = [int(i) for i in ids]

                            relevant_files =
File.objects.filter(user=self.user, id__in=ids).select_related('folder')
                            displayed_info = []
                            for f in relevant_files:
                                info = {
                                    "id": f.id,
                                    "name": f.name,
                                    "link": f.file.url,
                                    "folder_link": f.folder.link if
f.folder else None
                                }
                                files_found.append(info)
                                displayed_info.append({"id": f.id, "name":
f.name})
```

```
                            result = {"status": "displayed", "files":
displayed_info}

                        elif fc.name == "show_groups_to_user":
                            groups = fc.args.get("groups", [])
                            # groups is [{'name': '...', 'file_ids':
[...]}]

                            displayed_groups = []
                            for g in groups:
                                name = g.get('name', 'Group')
                                f_ids = [int(i) for i in g.get('file_ids',
[])]

                                group_files =
File.objects.filter(user=self.user, id__in=f_ids).select_related('folder')
                                group_file_list = []
                                found_ids = []
                                for f in group_files:
                                    group_file_list.append({
                                        "id": f.id,
                                        "name": f.name,
                                        "link": f.file.url,
                                        "folder_link": f.folder.link if
f.folder else None
                                    })
                                    found_ids.append(f.id)

                                if group_file_list:
                                    groups_found.append({
                                        "name": name,
                                        "files": group_file_list
                                    })
                                    displayed_groups.append({
                                        "name": name,
                                        "file_ids": found_ids
                                    })

                            result = {"status": "groups_displayed",
"groups": displayed_groups}

                        elif fc.name == "find_duplicates":
                            dups = self._find_duplicates()
                            result = {"duplicates": dups}
                            # No longer adding to groups_found
automatically. LLM must call show_groups_to_user.

                        elif fc.name == "find_similar_groups":
                            folder_id = fc.args.get("folder_id")
                            if folder_id: folder_id = int(folder_id)
                            groups = self._find_similar_groups(folder_id)
                            result = {"groups": groups}
                            # No longer adding to groups_found
automatically. LLM must call show_groups_to_user.

                        elif fc.name == "list_files":
                            folder_id = fc.args.get("folder_id")
                            limit = fc.args.get("limit", 100)
```

```
                            if folder_id: folder_id = int(folder_id)
                            if limit: limit = int(limit)
                            files = self._list_files(folder_id, limit)
                            result = {"files": files}
                            files_found.extend([
                                {
                                    "id": f["id"],
                                    "name": f["name"],
                                    "link": f["link"],
                                    "folder_link": f.get("folder_link")
                                } for f in files
                            ])

                        elif fc.name == "find_folder":
                            name = fc.args.get("name", "")
                            folders = self._find_folder(name)
                            result = {"folders": folders}

                        elif fc.name == "delete_files":
                            ids = fc.args.get("file_ids", [])
                            ids = [int(i) for i in ids]
                            res = self._delete_files(ids)
                            result = res

                        elif fc.name == "create_folder":
                            name = fc.args.get("name")
                            parent_id = fc.args.get("parent_id")
                            if parent_id: parent_id = int(parent_id)
                            result = self._create_folder(name, parent_id)

                        elif fc.name == "move_files":
                            ids = fc.args.get("file_ids", [])
                            folder_id = fc.args.get("target_folder_id")
                            if ids: ids = [int(i) for i in ids]
                            if folder_id: folder_id = int(folder_id)
                            result = self._move_files(ids, folder_id)

                        function_responses.append(
                            types.Part(
                                function_response=types.FunctionResponse(
                                    name=fc.name,
                                    response=result
                                )
                            )
                        )

                    contents.append(types.Content(role="function",
parts=function_responses))
                else:
                    return {
                        "text": response.text,
                        "files": files_found,
                        "groups": groups_found
                    }

            return {"text": "No response generated.", "files": [],
"groups": []}
```

```
        except Exception as e:
            logger.error(f"Agent chat error: {e}", exc_info=True)
            return {"text": f"Вибачте, сталася помилка: {str(e)}", "files":
[], "groups": []}

    def _search_files(self, query):
        results = []
        seen_ids = set()

        # 1. Keyword Search (Exact matches)
        # Шукаємо точне входження слова в назві, описі або тегах.
        # Це гарантує, що якщо користувач шукає "аніме", він знайде файли,
де це слово є.
        keyword_files = File.objects.filter(
            Q(user=self.user) &
            (
                Q(name__icontains=query) |
                Q(ai_description__icontains=query) |
                Q(ai_tags__icontains=query)
            )
        )[:20]

        for f in keyword_files:
            seen_ids.add(f.id)
            results.append({
                "id": f.id,
                "name": f.name,
                "link": f.file.url,
                "distance": 0.0, # Вважаємо ідеальним збігом для сортування
                "ai_description": f.ai_description,
                "ai_tags": f.ai_tags
            })

        # 2. Semantic Search (Vector search)
        query_vector = self.ai_service.get_text_embedding(query)
        if query_vector:
            # TIGHTENED THRESHOLD: 0.65 -> 0.55 (зменшуємо кількість
"сміття")
            semantic_files = File.objects.filter(user=self.user,
content_embedding__isnull=False)\
                .annotate(distance=CosineDistance('content_embedding',
query_vector))\
                .filter(distance__lt=0.55)\
                .order_by('distance')[:20]

            for f in semantic_files:
                if f.id not in seen_ids:
                    seen_ids.add(f.id)
                    results.append({
                        "id": f.id,
                        "name": f.name,
                        "link": f.file.url,
                        "distance": f.distance,
                        "ai_description": f.ai_description,
                        "ai_tags": f.ai_tags
                    })

        return results
```

```
    def _find_duplicates(self):
        duplicates = File.objects.filter(user=self.user).values('name',
'size').annotate(count=Count('id')).filter(count__gt=1)
        results = []
        for d in duplicates:
            files = File.objects.filter(user=self.user, name=d['name'],
size=d['size'])
            file_list = [{"id": f.id, "created_at": str(f.created_at)} for
f in files]
            results.append({
                "name": d['name'],
                "size": d['size'],
                "count": d['count'],
                "files": file_list
            })
        return results

    def _find_similar_groups(self, folder_id=None):
        """
        Groups files based on semantic similarity using DBSCAN clustering.
        """
        qs = File.objects.filter(user=self.user,
content_embedding__isnull=False)
        if folder_id:
            qs = qs.filter(folder_id=folder_id)
        # Якщо folder_id не передано, шукаємо по ВСІХ файлах користувача, а
не тільки в корені.
        # Це корисніше для глобального пошуку груп.

        files = list(qs)
        if len(files) < 2:
            return []

        try:
            embeddings = np.array([np.array(f.content_embedding) for f in
files])
        except Exception as e:
            logger.error(f"Error converting embeddings: {e}")
            return []

        try:
            # RELAXED EPS: 0.25 -> 0.45. Більший радіус для кластерів.
            # Якщо занадто багато файлів групуються в одну купу (label 0),
можна спробувати трохи зменшити eps
            # або збільшити min_samples.
            # Зменшуємо EPS до 0.15, оскільки text-embedding-004 дає дуже
щільні вектори (висока схожість).
            # 0.35 все ще зливає все в одну купу.
            clustering = DBSCAN(eps=0.15, min_samples=2,
metric='cosine').fit(embeddings)
        except Exception as e:
            logger.error(f"Clustering error: {e}")
            return []

        groups = {}
        for i, label in enumerate(clustering.labels_):
            if label == -1: continue # Noise points (not in any cluster)
```

```
            if label not in groups: groups[label] = []

            f = files[i]
            groups[label].append({
                "id": f.id,
                "name": f.name,
                "link": f.file.url,
                "ai_description": f.ai_description
            })

        results = []
        for label, group_files in groups.items():
            results.append({
                "group_id": int(label),
                "file_count": len(group_files),
                "files": group_files
            })

        return results

    def _list_files(self, folder_id=None, limit=100):
        files = File.objects.filter(user=self.user,
folder_id=folder_id).select_related('folder')[:limit]
        results = []
        for f in files:
            results.append({
                "id": f.id,
                "name": f.name,
                "link": f.file.url,
                "folder_link": f.folder.link if f.folder else None
            })
        return results

    def _find_folder(self, name):
        folders = Folder.objects.filter(user=self.user,
name__icontains=name)[:5]
        return [{"id": f.id, "name": f.name, "parent_id": f.parent_id} for
f in folders]

    def _delete_files(self, file_ids):
        files = File.objects.filter(user=self.user, id__in=file_ids)
        count = files.count()
        files.delete()
        return {"deleted_count": count}

    def _create_folder(self, name, parent_id=None):
        parent = None
        if parent_id:
            try:
                parent = Folder.objects.get(id=parent_id, user=self.user)
            except Folder.DoesNotExist:
                pass

        original_name = name
        counter = 1
        while Folder.objects.filter(user=self.user, parent=parent,
name=name).exists():
            name = f"{original_name} {counter}"
```

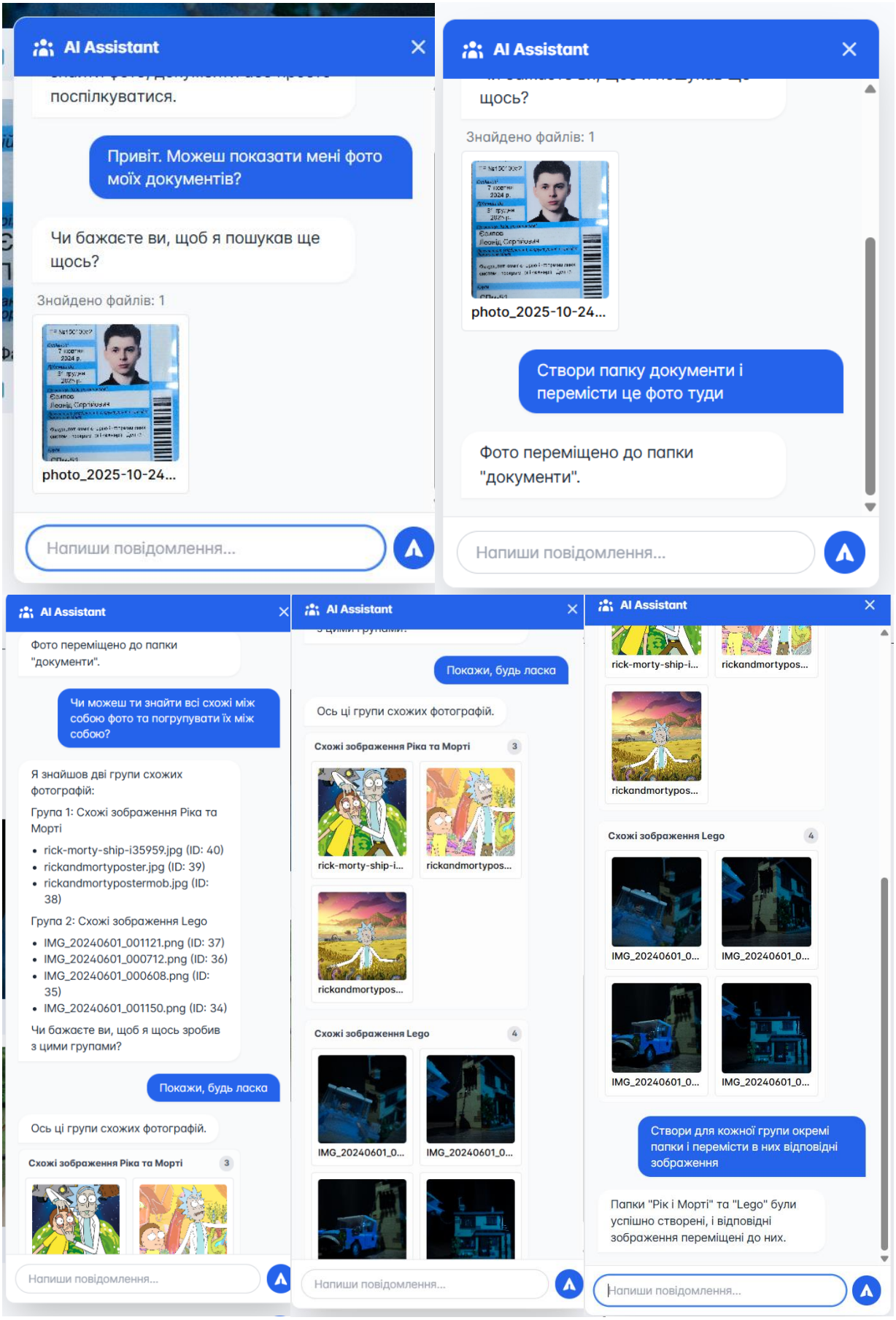
```
            counter += 1

        folder = Folder.objects.create(name=name, user=self.user,
parent=parent)
        return {"id": folder.id, "name": folder.name, "created": True}

    def _move_files(self, file_ids, folder_id=None):
        target_folder = None
        if folder_id:
            try:
                target_folder = Folder.objects.get(id=folder_id,
user=self.user)
            except Folder.DoesNotExist:
                return {"success": False, "error": "Target folder not
found"}

        files = File.objects.filter(user=self.user, id__in=file_ids)
        moved_count = 0

        for file_obj in files:
            try:
                old_path = file_obj.file.name
                if not default_storage.exists(old_path):
                    continue

                original_folder = file_obj.folder
                file_obj.folder = target_folder
                new_path = file_obj.file.field.generate_filename(file_obj,
os.path.basename(old_path))

                # S3 OPTIMIZATION
                moved = False
                if hasattr(default_storage, 'bucket'):
                    try:
                        copy_source = {
                            'Bucket': default_storage.bucket.name,
                            'Key': old_path
                        }
                        default_storage.bucket.copy(copy_source, new_path)
                        default_storage.delete(old_path)
                        moved = True
                    except Exception:
                        pass

                if not moved:
                    file_content = default_storage.open(old_path).read()
                    default_storage.save(new_path,
ContentFile(file_content))
                    default_storage.delete(old_path)

                file_obj.file.name = new_path
                file_obj.save(update_fields=['folder', 'file'])
                moved_count += 1
            except Exception as e:
                logger.error(f"Error moving file {file_obj.id}: {e}")

        return {"moved_count": moved_count}
```

Додаток В. Лістинг face_service.py

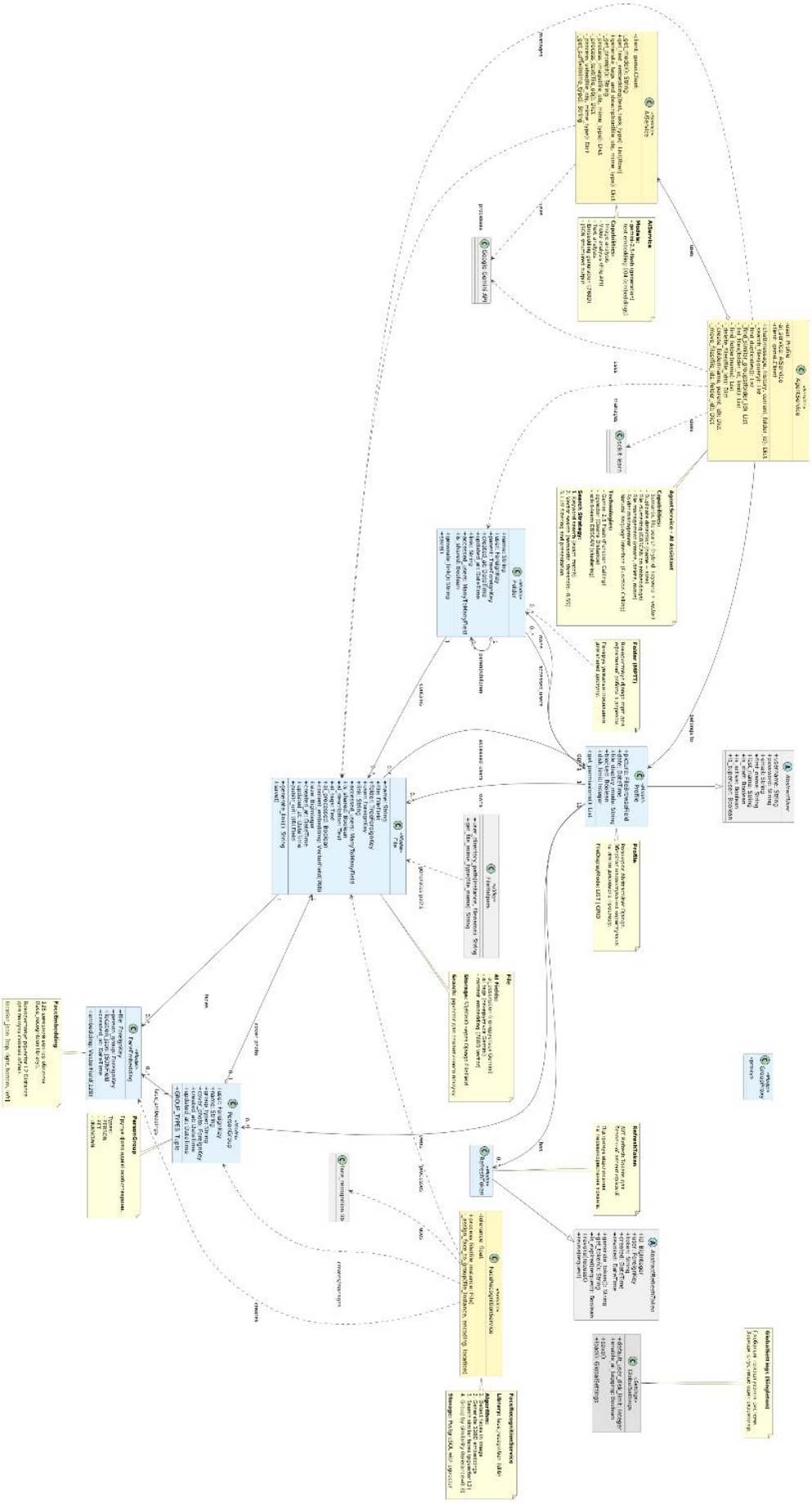
```
import logging
import json
import numpy as np
from django.conf import settings
from .models import File, PersonGroup, FaceEmbedding
from pgvector.django import L2Distance
logger = logging.getLogger(__name__)
try:
    import face_recognition
except ImportError:
    logger.warning("face_recognition library not installed. Face detection
will be disabled.")
    face_recognition = None

class FaceRecognitionService:
    def __init__(self):
        self.tolerance = 0.6 # Поріг схожості (0.6 - стандарт для dlib)

    def process_file(self, file_instance: File):
        """
        Головна функція обробки файлу.
        """
        if not face_recognition:
            logger.error("face_recognition library is missing.")
            return

        # Завантажуємо зображення
        # face_recognition.load_image_file приймає шлях до файлу або file-
like object
        # Якщо файл на S3, ми не можемо використовувати .path

        file_obj = file_instance.file
        file_obj.open() # Відкриваємо файл (це завантажить його з S3, якщо
треба)

        # 1. Знаходимо обличчя
        # Використовуємо PIL для конвертації в RGB
        from PIL import Image
        pil_image = Image.open(file_obj)

        # Обов'язково конвертуємо в RGB (3 канали, 8 біт)
        if pil_image.mode != 'RGB':
            pil_image = pil_image.convert('RGB')

        # Перетворюємо в numpy масив uint8
        image = np.array(pil_image, dtype=np.uint8)
        # Ensure array is contiguous for dlib
        image = np.ascontiguousarray(image)

        logger.info(f"Image shape: {image.shape}, dtype: {image.dtype}")

        face_locations = face_recognition.face_locations(image)

        if not face_locations:
```

```
            logger.info(f"No faces found in file {file_instance.id}")
            return

        face_encodings = face_recognition.face_encodings(image,
face_locations)

        logger.info(f"Found {len(face_encodings)} faces in file
{file_instance.id}")

        for encoding, location in zip(face_encodings, face_locations):
            self._assign_face_to_group(file_instance, encoding, location)

    def _assign_face_to_group(self, file_instance, encoding, location):
        """
        Шукає існуючу групу або створює нову.
        """
        # Пошук найсхожішого обличчя в базі даних за допомогою pgvector (L2
distance)
        # Ми шукаємо серед усіх облич того ж користувача (щоб не плутати з
чужими фото)
        closest_match = FaceEmbedding.objects.filter(
            file__user=file_instance.user
        ).annotate(
            distance=L2Distance('embedding', encoding)
        ).order_by('distance').first()

        person_group = None

        # Якщо знайшли схоже обличчя і відстань менша за поріг
        if closest_match and closest_match.distance < self.tolerance:
            person_group = closest_match.person_group
            logger.info(f"Matched existing person group:
{person_group.name} (distance: {closest_match.distance})")
        else:
            # Створюємо нову групу
            person_group = PersonGroup.objects.create(
                user=file_instance.user,
                name=f"Особа
{PersonGroup.objects.filter(user=file_instance.user).count() + 1}",
                group_type='PERSON'
            )
            # Встановлюємо це фото як обкладинку, якщо це перше фото
            person_group.cover_photo = file_instance
            person_group.save()
            logger.info(f"Created new person group: {person_group.name}")

        # Зберігаємо вектор
        FaceEmbedding.objects.create(
            file=file_instance,
            person_group=person_group,
            embedding=encoding,
            location_json=json.dumps(location) # [top, right, bottom, left]
        )
```

Додаток Г. Демонстрація роботи ШІ-асистента



Додаток Ґ. Розгорнута UML-діаграма класів програмної системи



Додаток Д. Тези для XIII науково-технічна конференція «Інформаційні моделі, системи та технології»

УДК 004.41
Єсипов Л. – ст. гр. СПм-61
Тернопільський національний технічний університет імені Івана Пулюя

ІНТЕГРАЦІЯ ІНТЕЛЕКТУАЛЬНИХ СЕРВІСІВ КЛАСИФІКАЦІЇ, ПОШУКУ ТА ЗБЕРІГАННЯ В СИСТЕМУ КЕРУВАННЯ МЕДІАКОНТЕНТОМ ТНТУ

Науковий керівник: д. ф., с. в. Мудрик І. Я.

Yesypov L.
Ternopil Ivan Puluj National Technical University

INTEGRATION OF INTELLIGENT CLASSIFICATION, SEARCH AND STORAGE SERVICES INTO THE TNTU MEDIA CONTENT MANAGEMENT SYSTEM

Supervisor: PhD, Senior Lecturer Mudryk I. Y.

Ключові слова: MinIO, об’єктне сховище, нейромережі, семантичний пошук
Keywords: MinIO, object storage, neural networks, semantic search

Підвищення ефективності сучасних медіа-систем потребує відмови від застарілих методів зберігання даних та впровадження інтелектуального аналізу. У роботі реалізовано модернізацію системи шляхом переходу на S3-сумісне об’єктне сховище MinIO. Це забезпечує високу відмовостійкість завдяки розподіленому зберіганню фрагментів файлів на різних дисках та дозволяє безшовне масштабування. Така архітектура гарантує безпеку даних та можливість миттєвої міграції на хмарні платформи типу Amazon S3 без зміни програмного коду [1].

Для розв'язання проблеми пошуку неструктурованого контенту інтегровано мультимодальну нейромережу Gemini 2.5 Flash. Система автоматично аналізує завантажені фото та відео, генеруючи релевантні описи й теги. Це дозволило реалізувати семантичний пошук, де користувач знаходить файли за їх вмістом. Вибір Gemini 2.5 Flash обумовлений поєднанням високої швидкості обробки, низької вартості запитів та гнучкості моделі [2].

Впровадження запропонованих рішень дозволило створити уніфіковану платформу, яка не лише гарантує збереження інформації, а й суттєво скорочує час на її пошук та обробку.

Література

1. MinIO Documentation [Електронний ресурс] – Режим доступу: https://github.com/minio/minio.
2. Gemini API Overview [Електронний ресурс] – Режим доступу: https://ai.google.dev/gemini-api/docs.

Додаток Е. Репозиторії

Бекенд: https://github.com/john-psina/TNTUStorageBackend

Фронтенд: https://github.com/john-psina/TNTUStorageFrontend