МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ

“До захисту допущений”
Зав. кафедри комп’ютерних наук та прикладної математики
д.т.н., професор Турбал Ю. В.
«___» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

«Програмна реалізація системи розпізнавання облич за допомогою алгоритмів машинного зору»

Виконав: Майба Павло Іванович
Студент навчально-наукового інституту кібернетики, інформаційних технологій та інженерії
група ІПЗ-41і-2

_____
(підпис)

Керівник: доц., к.т.н. Жуковський Віктор Володимирович

_____
(підпис)

Рівне – 2025

ЗМІСТ

РЕФЕРАТ

Кваліфікаційна робота: 63 сторінок, 26 рисунків, 14 наукових джерел.

Мета роботи: створення програмної системи для автоматизованого розпізнавання облич із використанням сучасних методів машинного зору

Інструменти реалізації: C#, ASP.NET Core, React, Redux, PostgreSQL, ONNX, FaceNet, SCRFD, EmguCV, Ant Design, TypeScript, REST API.

Актуальність теми обумовлена зростаючою потребою у безпечних та ефективних системах ідентифікації особи, що застосовуються в освітніх закладах, підприємствах та публічних установах. Використання алгоритмів комп'ютерного зору дає змогу автоматизувати процес контролю відвідуваності, покращити точність і мінімізувати людський фактор.

Ключові слова: розпізнавання облич, машинний зір, нейронні мережі, ONNX, комп'ютерне бачення, автоматизація, FaceNet, ASP.NET, React, PostgreSQL.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

СУБД – система управління базами даних (у рамках проєкту використовувалась PostgreSQL);

Frontend – клієнтська частина веб-застосунку, що реалізована з використанням React;

Backend – серверна частина програмного забезпечення, розроблена на базі ASP.NET Core;

FaceNet – глибинна нейронна мережа для побудови векторних представлень (ембеддінгів) облич;

SCRFD – високоточна модель виявлення облич, оптимізована для реального часу;

EmguCV – .NET-обгортка для бібліотеки OpenCV, що використовується для обробки зображень;

ONNX – відкритий формат представлення моделей машинного навчання для між-платформного використання;

API – інтерфейс прикладного програмування для обміну даними між клієнтом і сервером;

ВСТУП

Сучасна епоха цифрової трансформації супроводжується стрімким зростанням обсягів інформації, яку щодня генерують, передають та обробляють комп'ютерні системи. Дані стали не лише технічним активом, а й цінним стратегічним ресурсом, що впливає на прийняття рішень, безпеку, зручність користувача та ефективність роботи в багатьох галузях. На фоні цього зростає потреба в автоматизованих системах, здатних ефективно обробляти зображення, відео та інші візуальні потоки інформації — зокрема, системах машинного зору.

Однією з ключових задач комп'ютерного зору є розпізнавання облич — процес, у якому система виявляє, аналізує та ідентифікує людське обличчя на зображенні або у відеопотоці. Це завдання є актуальним у контексті як прикладних, так і теоретичних досліджень, оскільки воно поєднує в собі інтереси в сфері штучного інтелекту, безпеки, автоматизації та аналітики поведінки користувача. Системи розпізнавання облич використовуються в банках, державних установах, навчальних закладах, аеропортах, смартфонах та навіть у побутових пристроях — від камер відеоспостереження до систем "розумного дому".

Важливість таких рішень зростає в умовах потреби контролювати фізичну присутність людей у певному середовищі, автоматизувати облік відвідуваності або забезпечити швидкий і зручний доступ до цифрових сервісів без паролів. Саме тому все більше інституцій впроваджують системи ідентифікації на основі облич, як більш зручну та менш інвазивну альтернативу класичним методам, як-от RFID-картки, пін-коди чи ручні списки.

Істотний прогрес у цій галузі став можливим завдяки розвитку алгоритмів глибинного навчання, що дозволяють обробляти візуальні дані з високою точністю. Серед найефективніших підходів — використання попередньо натренованих моделей, зокрема FaceNet, яка генерує векторні представлення (ембеддінги) облич, що зберігають унікальні біометричні ознаки. Для виявлення облич на зображенні застосовуються сучасні детектори, зокрема SCRFD — модель, оптимізована для реального часу, що демонструє високу точність навіть у складних умовах зйомки.

У контексті цієї кваліфікаційної роботи було поставлено завдання розробити систему розпізнавання облич з нуля, реалізувавши повноцінний програмний комплекс, який об’єднує серверну логіку на основі ASP.NET Core, клієнтську частину з використанням React та вбудовані модулі для захоплення зображення, виявлення облич, обробки ознак та звірки з базою. Дані про користувачів та результати обробки зберігаються у СУБД PostgreSQL, що дозволяє вести облік, аналітику та візуалізацію відвідуваності. Особливу увагу приділено інтеграції з камерою, обробці потокового відео та можливості ручного чи автоматичного підтвердження особи.

Таким чином, запропонована система є актуальним і практично значущим рішенням, яке демонструє, як за допомогою сучасних інструментів штучного інтелекту можна реалізувати ефективну, масштабовану та зручну у використанні платформу для розпізнавання облич. Результати цієї роботи можуть бути застосовані як у навчальних закладах, так і в інших організаціях, де необхідно фіксувати присутність користувачів або контролювати доступ.

I. АНАЛІТИКА ТЕХНОЛОГІЙ ІДЕНТИФІКАЦІЇ ОСІБ НА ОСНОВІ ОБЛИЧЧЯ

1.1 Стан і перспективи розпізнавання облич у цифрових системах

Розпізнавання облич — одна з ключових задач комп'ютерного зору, яка вже протягом кількох десятиліть активно досліджується і впроваджується в різноманітні цифрові сервіси. На початкових етапах ця технологія базувалася на простих евристичних алгоритмах, які аналізували геометричні пропорції обличчя на статичному зображенні. Проте з розвитком обчислювальної техніки, зростанням доступності графічних процесорів і проривами в галузі глибинного навчання якість і точність таких систем зросли в рази.

Сьогодні технології розпізнавання облич впроваджуються у багатьох сферах: контроль доступу в будівлях, відеоспостереження в режимі реального часу, ідентифікація користувача в мобільних пристроях, онлайн-верифікація особи у фінансових операціях, облік відвідуваності в навчальних закладах тощо. Наприклад, в освітньому середовищі подібні системи дають змогу фіксувати присутність студентів без використання ручного запису чи карток, що значно спрощує адміністративну роботу.

З технічного погляду, сучасні системи розпізнавання облич складаються з декількох послідовних етапів: виявлення обличчя на зображенні або відео, нормалізація зображення, витяг біометричних ознак (ембеддінгів) і їх порівняння з раніше збереженими зразками в базі даних. Найбільш популярні моделі сьогодні — це нейронні мережі, зокрема Convolutional Neural Networks (CNN), серед яких виділяються такі підходи, як FaceNet, ArcFace, Dlib, VGGFace, InsightFace.

Прогрес у цій галузі стимулюється зростаючою потребою в безпечній, безконтактній та швидкій верифікації особи. При цьому розробники все частіше фокусуються не лише на точності розпізнавання, а й на оптимізації ресурсів, адаптації до умов поганого освітлення, часткових перекриттів обличчя, змін кута зйомки тощо. Особливу актуальність мають моделі, здатні працювати в реальному часі на звичайних пристроях — таких як веб-камери ноутбуків, Raspberry Pi або смартфони.

Попри успіхи, які спостерігаються в цій галузі, існує низка викликів, що стримують повсюдне впровадження систем розпізнавання облич. Одним із ключових обмежень є залежність точності моделей від якості вхідних зображень. Навіть найсучасніші системи можуть давати похибки в умовах низької роздільної здатності, занадто темного або пересвіченого фону, а також при наявності сторонніх об'єктів, які частково закривають обличчя. Іншим викликом є уніфікація форматів векторних ознак, коли системи з різною архітектурою та методами нормалізації не можуть напряму обмінюватись ембеддінгами.

Ще один вектор розвитку — це інтеграція розпізнавання облич у edge-системи (тобто пристрої, що обробляють дані локально, без надсилання в хмару). Такий підхід дозволяє зменшити затримки, зберегти конфіденційність та працювати навіть без стабільного інтернет-з'єднання. Розробка ефективних, легких моделей для edge-пристроїв — актуальний напрямок, який відкриває нові горизонти для застосування цієї технології в автономних системах безпеки, носимій електроніці та промислових рішеннях.

Важливою рушійною силою розвитку є поява великих відкритих датасетів. Корпорації та університети публікують набори на мільйони зображень (MS-Celeb-1M, VGGFace2, GFace), що дозволяє тренувати універсальні моделі із високою здатністю до узагальнення. Разом із тим, використання реальних фото викликає етичні питання щодо згоди та права на забуття, особливо в юрисдикціях, де діє GDPR. Як відповідь з'являються синтетичні датасети, згенеровані за допомогою генеративних нейромереж StyleGAN, які дають подібну різноманітність, але не містять справжніх персональних даних.

Не менш активно біометрія розширює присутність у транспортній галузі. У низці аеропортів ЄС уже працюють "біометричні коридори", де паспортний контроль замінений безконтактним сканером обличчя; тривалість проходження контролю зменшується на 35-40 %.

Паралельно формуються вузькоспеціалізовані рішення: у медицині — ідентифікація пацієнта в реанімації, у банківському секторі — підтвердження VIP-клієнта без введення ПІН-коду, у промисловості — контроль доступу до небезпечних ділянок. Кожна з цих галузей висуває власні метрики успішності: у

банку критична точність (FAR < 0,001 %), а у промисловості — час спрацювання (< 0,5 с). Це підштовхує виробників до створення адаптивних моделей, здатних балансувати між швидкістю та точністю залежно від сценарію.

Нарешті, суттєве значення має протидія spoofing-атакам. Дедалі частіше базові RGB-камери доповнюють інфрачервоним каналом або аналізом мікрорухів обличчя, щоби переконатися, що перед камерою жива людина, а не фото чи відео. Підтримка "живості" (liveness detection) вже фігурує у вимогах до державних проєктів е-ідентифікації, і невпровадження цього механізму розглядають як серйозну уразливість системи.

У перспективі очікується подальше вдосконалення алгоритмів і поява систем, що враховують додаткові фактори — емоційний стан, рух, міміку, а також поєднання технологій розпізнавання облич з іншими біометричними методами — відбитками пальців, голосом або райдужкою ока. Одночасно з цим, актуальною залишається проблема захисту приватності, оскільки розпізнавання облич несе ризики порушення конфіденційності та зловживання даними. Тому сучасні розробки також зосереджені на етичному використанні таких систем, захисті даних і прозорості алгоритмів.

Таким чином, розпізнавання облич — це не лише технологія сьогодення, а й потужний інструмент майбутнього, здатний трансформувати взаємодію людини з цифровими системами в усіх сферах — від освіти до національної безпеки.

1.2 Методи комп’ютерного зору в задачах ідентифікації особи

Комп’ютерний зір — це міждисциплінарна галузь, що об’єднує машинне навчання, обробку зображень, штучний інтелект та математику з метою надання комп’ютерам здатності "бачити" та інтерпретувати візуальну інформацію. В контексті задачі розпізнавання облич, комп’ютерний зір виступає як основа для побудови алгоритмів, що дозволяють виявити, сегментувати, аналізувати та класифікувати обличчя на зображеннях або у відеопотоці.

Розпізнавання облич умовно можна поділити на два основні етапи: виявлення облич (face detection) і подальша ідентифікація (face recognition). Перший етап відповідає за локалізацію обличчя на зображенні, а другий — за встановлення тотожності або верифікацію особи.

На ранніх етапах розвитку технології використовувались класичні алгоритми комп’ютерного зору, такі як:

- **Haar Cascade Classifier** — один із перших ефективних методів виявлення облич, реалізований у бібліотеці OpenCV. Його робота ґрунтується на використанні каскадних фільтрів та інтегрального зображення. Цей метод був швидким, але мав низьку стійкість до поворотів голови та змін освітлення.
- **Histogram of Oriented Gradients (HOG)** — метод, що виділяє ознаки форми об’єктів, і широко застосовувався в поєднанні з Support Vector Machines (SVM) для виявлення обличчя. Більш точний, ніж Haar, але поступається сучасним нейронним мережам.
- **LBP (Local Binary Patterns)** — використовувався для витягу текстурних ознак та також застосовувався в задачах верифікації облич.

Справжній прорив у точності й адаптивності відбувся із впровадженням глибинного навчання. Нейронні мережі, особливо згорткові (CNN), дозволили системам навчатися представлень ознак облич на величезних наборах даних, що значно покращило точність навіть в складних умовах. Найбільш відомі сучасні підходи включають:

- **FaceNet** — нейромережевий алгоритм, який переводить зображення обличчя у векторний простір фіксованої розмірності (ембеддінг). Порівняння відбувається шляхом обчислення евклідової відстані між векторами.

- **ArcFace** — подальший розвиток ідеї FaceNet із вдосконаленим функціоналом втрат, що дозволяє ще краще розрізняти схожі обличчя.
- **Dlib CNN face recognition** — популярна бібліотека з відкритим кодом, яка надає передтреновані моделі для виявлення та розпізнавання облич.
- **InsightFace / RetinaFace / SCRFD** — моделі, оптимізовані для швидкої та точної роботи у реальному часі, використовуються в мобільних та веб-додатках.

Паралельно з CNN-архітектурами набирають популярності трансформерні моделі, які використовують механізм самоуваги. Наприклад, серії ViT-Face або Swin-Face демонструють порівняну з CNN точність, але краще захоплюють довгострокові залежності у зображенні — це знижує кількість помилок при сильних поворотах голови. Крім того, дослідники комбінують CNN і Vision Transformer у гібридних мережах, що успішно проходять найлютіші бенчмарки, такі як IJB-C.

Щоб утримувати високу точність у реальних умовах, сучасні системи рідко покладаються на єдиний алгоритм. Зазвичай застосовується комбінація методів: детектор SCRFD для швидкого локалізування облич, ArcFace або FaceNet перетворюють вирівняне обличчя на ембеддінг, далі швидкі методи пошуку найближчих сусідів (Faiss, Annoy) визначають, чи належить ембеддінг до відомої особи. Така модульність дає змогу оновлювати окремі компоненти без повної перебудови системи.

Останні дослідження демонструють цінність мультимодальної біометрії: поєднання зображення обличчя з голосом або динамікою набору тексту підвищує загальну точність і стійкість до spoofing-атак. Наприклад, платформи для онлайн-іспитів уже використовують подвійний канал: камера + мікрофон; невідповідність обох ознак відразу сигналізує про потенційну підміну.

Таким чином, методи комп'ютерного зору в задачах ідентифікації особи постійно еволюціонують. Сьогодні головна тенденція — створення оптимізованих, точних, ресурсно-економних рішень, які здатні працювати в реальному часі на звичайному обладнанні, з урахуванням етичних та правових норм використання біометрії.

1.3 Огляд алгоритмів обробки та класифікації облич

У сучасних інформаційних системах розпізнавання облич стало важливим інструментом для ідентифікації та автентифікації користувачів. Цей процес включає декілька основних етапів: підготовку зображення, витяг біометричних ознак (ембеддінгів), порівняння з базою даних і класифікацію. Кожен з цих етапів реалізується за допомогою спеціалізованих алгоритмів комп'ютерного зору, які постійно вдосконалюються.

Першим кроком є обробка зображення обличчя, що включає нормалізацію кольору, масштабування до стандартного розміру, усунення шуму та орієнтацію по очах. Якість цих попередніх дій прямо впливає на точність подальшої класифікації.

Ключова роль належить ембеддінгам обличчя — числовим векторним представленням, що описують унікальні особливості кожної особи. Замість порівняння самих зображень система оперує векторами, які легко порівнювати математично.

Для побудови таких векторів найчастіше застосовуються глибокі нейронні мережі, зокрема:

- FaceNet — один із перших потужних алгоритмів, що формує вектор ознак із мінімальною відстанню між векторами однієї особи і максимальною — для різних.
- ArcFace — алгоритм з удосконаленою функцією втрат, що забезпечує кращу міжкласову роздільність.
- DeepFace, VGGFace — інші приклади глибоких моделей, здатних працювати з великою кількістю класів (осіб).

Після отримання векторного представлення, система виконує класифікацію або пошук збігів. Це може бути:

- Проста перевірка схожості — наприклад, обчислення косинусної або евклідової відстані між ембеддінгами.
- Пошук найближчого сусіда (KNN) — у випадках, коли потрібно знайти найбільш схожий запис у базі.

- Кластеризація — наприклад, для групування невідомих осіб за схожістю облич.

Додатково використовуються методи оптимізації пошуку, такі як індексація векторів (наприклад, Faiss), що дозволяє системам працювати ефективно навіть із великими наборами даних.

Слід відзначити, що точність класифікації суттєво залежить від таких факторів:

- якість вхідного зображення (освітлення, положення голови, роздільна здатність),
- наявність або відсутність "шуму" (окуляри, маски, волосся на обличчі),
- кількість та якість прикладів одного користувача в базі.

Таким чином, алгоритми обробки та класифікації облич є центральними елементами сучасних систем розпізнавання. Їх розвиток спрямований на підвищення точності, зменшення помилкових спрацювань, а також забезпечення роботи в реальному часі з мінімальними ресурсами.

1.4 Аналіз рішень і сервісів для розпізнавання особистості

З розвитком технологій комп'ютерного зору з'явилась велика кількість інструментів і сервісів, орієнтованих на реалізацію систем розпізнавання особистості на основі зображення обличчя. Частина з них орієнтована на хмарні обчислення та готові API, інші — на бібліотеки з відкритим кодом, які дозволяють повністю контролювати процес обробки зображень. Проте, попри зовнішню схожість, ці підходи кардинально відрізняються за рівнем відкритості, гнучкості, точності, витратності й технічних обмежень.

Більшість глобальних технологічних компаній пропонують хмарні сервіси для розпізнавання облич. Такі рішення мають високу точність і зручні в інтеграції, особливо при розробці мобільних застосунків або великих систем зі складною архітектурою. Разом з тим, використання сторонніх хмарних платформ у задачах, пов'язаних із біометричними даними, породжує низку ризиків — від потреби передавати зображення третім сторонам до юридичних обмежень щодо зберігання персональної інформації. Крім того, такі сервіси, зазвичай, працюють за моделлю оплати за кількість запитів, що робить їх економічно невигідними при великій кількості користувачів або високій частоті розпізнавання.

Комерційні SDK, що позиціонуються як готові інструменти для інтеграції біометрії в різні системи, також заслуговують на увагу. Вони зазвичай забезпечують високу якість результатів, підтримку апаратного прискорення та додаткову функціональність. Однак ці рішення є закритими, дорогими і, в багатьох випадках, складними для інтеграції з сучасним web-стеком. Крім того, обмеження ліцензування та залежність від конкретного виробника звужують гнучкість їхнього використання у власних проєктах.

Натомість бібліотеки з відкритим кодом, такі як FaceNet, ArcFace, InsightFace або SCRFD, надають більше свободи. Вони дозволяють розробнику повністю контролювати обчислювальний процес: від детекції обличчя до класифікації ембеддінгів. Такі інструменти працюють локально, не потребують підключення до зовнішніх API, можуть бути адаптовані до будь-якої специфіки й інтегровані у власну інфраструктуру. Це особливо важливо в сферах, де

обробка персональних даних має бути максимально захищеною та незалежною від сторонніх сервісів.

Розроблена в рамках цієї кваліфікаційної роботи система використовує саме open-source підхід. Вона побудована на основі моделі FaceNet для побудови ембеддінгів обличчя та детектора SCRFD, що дозволяє здійснювати розпізнавання локально, без доступу до Інтернету, з високою точністю і повним контролем над даними. Такий вибір обґрунтований прагненням створити ефективне, масштабоване та етично безпечне рішення, яке не залежить від ліцензійних обмежень або змін політики зовнішніх платформ. На відміну від готових комерційних або хмарних продуктів, ця система є відкритою до розширення, її архітектура допускає модифікацію під конкретні задачі, а вся обробка даних відбувається виключно в межах локального середовища.

Таким чином, порівняльний аналіз рішень у сфері розпізнавання облич вказує на доцільність обрання власного програмного рішення на базі відкритих технологій. Воно не лише забезпечує необхідний функціонал, але й дозволяє уникнути ризиків, пов’язаних із конфіденційністю, фінансовими витратами та обмеженнями сторонніх API.

1.5 Системне формулювання задачі проєкту та очікувані результати

У процесі розробки будь-якої інформаційної системи важливо не лише дослідити предметну область, а й здійснити чітке системне формулювання задачі. Це дозволяє структурувати вимоги до функціональності, обрати відповідну архітектуру, методи реалізації та визначити очікувані результати, що будуть досягнуті в результаті впровадження розробленого рішення.

У рамках даного проєкту предметною областю є автоматизоване розпізнавання облич із метою ідентифікації особи та реєстрації присутності в навчальному середовищі. Система має забезпечувати розпізнавання в реальному часі з використанням веб-камери, обробку фото, виявлення обличчя, витяг ознак, порівняння з базою зареєстрованих користувачів та фіксацію результату у базі даних. Проєкт реалізується як кросплатформенна веб-система, що включає клієнтську частину (інтерфейс), серверну частину (обробка логіки) та модуль комп'ютерного зору (FaceNet + SCRFD).

Загальна постановка задачі

Метою роботи є створення універсальної, надійної та розширюваної програмної системи, яка реалізує процес біометричного розпізнавання облич та подальшого обліку присутності особи в навчальному закладі або іншій організації. Система має забезпечити:

- безпечне розпізнавання користувачів на основі облич;
- роботу у реальному часі через камеру;
- можливість автоматичного або ручного підтвердження особи;
- інтеграцію з базою даних користувачів;
- гнучкий розподіл ролей;
- повну автономність без використання сторонніх хмарних API.

Основні технічні цілі:

- Забезпечити якісне виявлення облич незалежно від умов освітлення, позиції голови, наявності аксесуарів (окулярів, головних уборів).
- Реалізувати модуль ембеддінгів (FaceNet) для порівняння облич за векторними ознаками.

- Розробити веб-інтерфейс для керування користувачами, сесіями, групами та історією розпізнавань.
- Побудувати архітектуру, яка дозволить масштабувати систему при необхідності — з розподіленням навантаження між клієнтською та серверною частинами.

Основні функціональні компоненти системи

1. Модуль розпізнавання облич.

Центральним елементом системи є реалізація алгоритмів машинного зору. Обробка включає в себе:

- нормалізацію вхідного зображення;
- виявлення облич (SCRFD-модель у форматі ONNX);
- обрізання та масштабування області обличчя до потрібного формату;
- подачу обличчя у FaceNet-модель;
- отримання векторного представлення;
- порівняння з базою векторів користувачів та прийняття рішення про збіг.

2. База даних користувачів.

База містить інформацію про зареєстрованих осіб: ім’я, електронну адресу, роль, групу, ембеддінги облич, стан облікового запису, та історію розпізнавань.

3. Модуль керування сесіями.

Передбачає створення сесій викладачем (занять, лекцій, подій), реєстрацію присутності студентів на основі факту успішного розпізнавання, а також ручне підтвердження у разі проблем з обличчям (наприклад, якщо людина змінила свій образ).

4. Інтерфейс користувача.

Система має веб-інтерфейс, побудований на React, з окремими панелями для кожного типу користувачів:

- Адміністратор може створювати групи, сесії, додавати/блокувати користувачів, переглядати відвідуваність.
- Викладач має доступ до сесій своєї групи, може запускати процес розпізнавання та переглядати результати.

- Студент бачить свої відмітки, фото з розпізнавання та інформацію про поточні сесії.

Запит на розпізнавання обробляється сервером, який пересилає зображення на модуль розпізнавання (FaceNet + SCRFD), отримує векторні представлення обличчя і зіставляє їх із векторами з бази.

Обробка виняткових ситуацій

Система враховує можливість збоїв та винятків. Наприклад:

якщо не вдалося знайти обличчя на зображенні — повертається відповідне повідомлення користувачу;

якщо ембеддінг не схожий на жоден із зареєстрованих — фіксується відмова у присутності і відправляється заявка на ручну перевірку;

Вимоги до системи

- Працездатність у реальному часі.
- Можливість використання як вебкамери.
- Сумісність з різними браузерами та мобільними пристроями.
- Підтримка розділення прав доступу для користувачів різного типу.
- Безпека: захищене зберігання ембеддінгів, шифрування токенів автентифікації (JWT), перевірка сесій.
- Простота розгортання системи на локальному сервері або у хмарі.

Сценарії використання системи

Автоматичне відмічання: студент підходить до камери — система автоматично розпізнає обличчя — присутність фіксується.

Півручний режим: система не змогла ідентифікувати — викладач підтверджує присутність вручну.

Огляд статистики: адміністратор переглядає аналітику щодо відвідуваності тощо.

Очікувані результати

У результаті виконання роботи передбачається створення повноцінного прототипу системи, який буде виконувати такі функції:

- захоплення та обробка зображення з камери користувача;
- точне виявлення облич на зображенні та формування їх ембеддінгів;
- порівняння ембеддінгу з базою відомих користувачів;
- прийняття рішення про відповідність і реєстрація присутності;
- відображення результату в інтерфейсі адміністратора або викладача;
- зберігання статистики розпізнань.

Конкурентні переваги

Система має переваги перед комерційними продуктами:

- Дає повний контроль над даними.
- Легко модифікується під потреби.
- Безкоштовна у використанні — не потребує ліцензій.

Кінцевим очікуваним результатом є функціональний MVP (minimum viable product), що дозволяє проводити автоматизовану реєстрацію присутності студентів за допомогою розпізнавання облич із подальшим збереженням інформації в базі даних.

II. АРХІТЕКТУРА І ПРОЄКТУВАННЯ СИСТЕМИ РОЗПІЗНАВАННЯ ОБЛИЧ

2.1 Загальна архітектурна концепція системи

Система розпізнавання облич призначена для автоматичного визначення особи та фіксації факту її присутності у навчальному середовищі. Усі обчислення виконуються на сервері, без передавання біометричних даних у сторонні сервіси.

Контекст функціонування

Веб-клієнт (React-додаток), що працює на пристрої користувача, зчитує зображення з камери, яке передає на REST API, реалізоване у серверному застосунку ASP.NET Core. Після проходження етапів обробки та розпізнавання результат (якщо обличчя ідентифіковано) зберігається до бази PostgreSQL у вигляді запису про відвідування дивись Рис. (2.1).

Рисунок 2.1 Контекстна діаграма взаємодії компонентів системи

Режими роботи

Камерний (Lecture Mode). Викладач відкриває сесію, система безперервно обробляє потокове відео з аудиторної камери, автоматично відмічаючи всіх присутніх студентів.

Режим самопозначення (Self-Check Mode). Студент на власному пристрої запускає сторінку потрібної сесії, дозволяє доступ до веб-камери й надсилає одиночний кадр, по успішному розпізнанню запис відразу зберігається в журналі.

Узагальнені функціональні вимоги

Швидкодія – результат повертається у режимі, комфортному для користувача, та не блокує паралельну роботу обох режимів.

Журнал відвідувань – кожен успішний збіг створює запис у Attendance; якщо збіг відсутній — створюється запит на ручну перевірку.

Ролі доступу:

- Administrator — повний контроль над системою;
- Moderator — адміністрування без права видаляти адміністраторів чи модераторів;
- Lecturer — відкриття/закриття сесій, вибір режиму, перегляд відвідуваності власних груп;
- Student — запуск самопозначення та перегляд особистих відміток.

Модулі машинного зору – використовуються наперед натреновані ONNX-моделі SCRFD (детекція) та FaceNet (ембеддинг);

Просте розгортання – один контейнер ASP.NET Core і один контейнер PostgreSQL

Зовнішні обмеження та середовище експлуатації

Обладнання – у розпорядженні один сервер із GPU для інференсу та резервне CPU-ядро для служебних задач. У викладачів камери стандартної чіткості для розпізнавання з аудиторій, на особистих пристроях студентів використовуються вбудовані веб-камери.

Мережа – сервер розташований у внутрішньому сегменті; назовні відкрито тільки порт 443.

Нормативні вимоги – дані облич та їхні ембеддинги не передаються стороннім сервісам, що відповідає Закону України «Про захист персональних даних».

Людські ресурси – увесь життєвий цикл — аналіз, розробку, тестування й деплой виконує один студент.

Архітектура серверного застосунку

Для реалізації бекенду обрано підхід Clean Architecture дивись Рис. (2.2).

, що розділяє логіку на чотири незалежні шари:

1. Presentation — API-контролери ASP.NET Core.
2. Application — сервіси, які обробляють запити та координують модулі.
3. Domain — сутності та бізнес-логіка.

4. Infrastructure — реалізації доступу до БД.

Цей підхід дозволяє:

Ключові виклики зводяться до того, аби:

- забезпечити модульність і тестованість системи;
- у майбутньому винести ML-модулі в окремий мікросервіс без зміни бізнес-логіки;
- швидко масштабувати окремі компоненти (наприклад, систему розпізнавання).

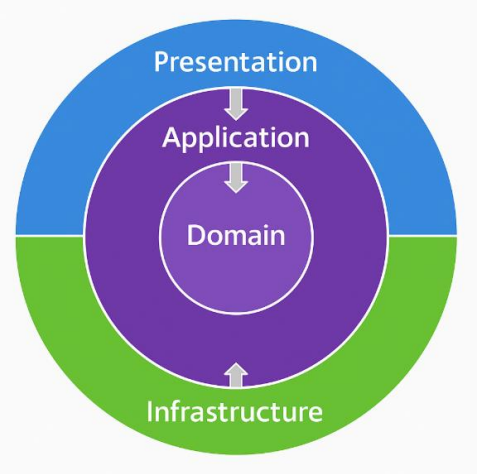

Рисунок 2.2 Схема взаємодії шарів архітектури програмного продукту

Алгоритм обробки зображення

Кожен кадр, що надсилається на сервер, проходить послідовність етапів (дивись діаграму послідовності Рис. (2.2)):

1. Виявлення обличчя (модель SCRFD);
2. Вирізання області обличчя;
3. Побудова вектора ембеддінгу (FaceNet);
4. Пошук найближчого ембеддінгу у БД;
5. Якщо знайдено — створення запису відвідування, інакше — створення запиту на модерацію.

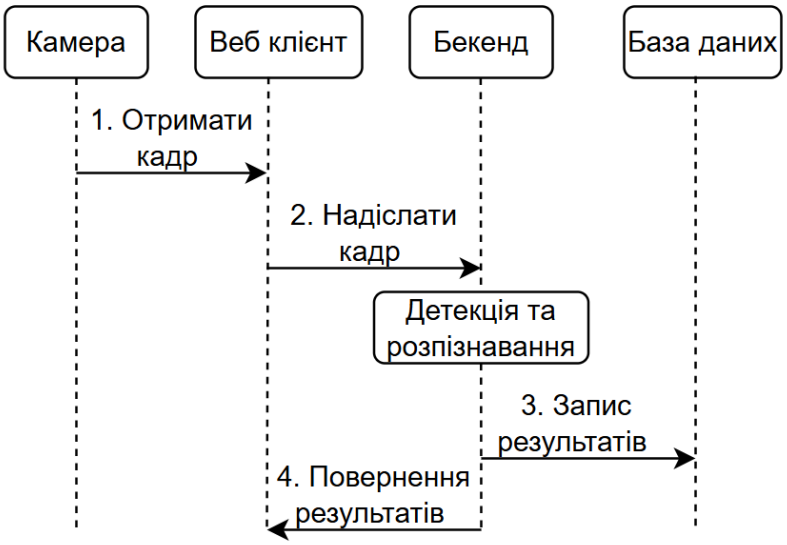

Рисунок 2.3 Діаграма послідовності обробки кадру в системі розпізнавання облич

Архітектура клієнтського застосунку (UI)

Інтерфейс реалізовано як SPA (Single Page Application) на React. Користувачі мають різні ролі, відповідно до яких генерується структура доступних сторінок та дозволених дій. Основні вимоги:

Адаптивність — коректне відображення на ноутбуках, планшетах, смартфонах.

Інтуїтивність — мінімум кліків до ключових дій (відмітка, створення сесії, перегляд журналу).

Безпечна взаємодія з камерою — дозвіл користувача, без збереження фото на клієнті.

2.2 Проектування структури бази даних

База даних є фундаментальним елементом системи розпізнавання облич, оскільки саме вона забезпечує зберігання критично важливої інформації — даних про користувачів, їх вектори облич, групи, сесії, результати розпізнавання, журнали активності, а також відмітки присутності. Для реалізації обрано реляційну модель з використанням PostgreSQL як основної системи керування базами даних, яка забезпечує надійність, транзакційність, масштабованість та хорошу підтримку індексів.

Підхід до проектування

У реалізації структури бази даних було використано підхід Code First, запропонований у рамках технології Entity Framework Core. Цей підхід передбачає створення таблиць та зв'язків безпосередньо на основі C#-класів, що репрезентують доменні сутності системи.

Основна перевага Code First — це уніфікація логіки: структура зберігається в єдиному джерелі — програмному коді, що виключає дублювання у вигляді SQL-скриптів. Зміни в бізнес-вимогах оперативно вносяться у модель, після чого база оновлюється через автоматичні міграції.

Завдяки підтримці механізму міграцій, усі зміни в моделі даних фіксуються як кроки, які зручно застосовувати по мірі розвитку функціоналу. Це дозволяє не лише уникнути розсинхронізації між кодом і базою, а й ефективно управляти історією змін. У поєднанні з чітким розмежуванням шарів архітектури (доступу до даних, логіки, контролерів), така структура сприяє підтримці чистоти проєкту та дозволяє масштабувати його відповідно до вимог Clean Architecture.

Також цей підхід спрощує розгортання системи: EF Core самостійно генерує схему, керує зв'язками між таблицями та підтримує Fluent API для тонкого налаштування структури — типів зв'язків, поведінки при видаленні тощо.

Враховуючи масштаб і динамічність проєкту, Code First є оптимальним рішенням, що дозволяє швидко адаптувати систему до змін і водночас підтримувати повну інтеграцію з програмною логікою.

ASP.NET Identity та AppUser

Система автентифікації та керування ролями реалізована за допомогою ASP.NET Identity, що забезпечує гнучку і розширювану модель. Базовий клас IdentityUser було успадковано в моделі AppUser, до якої додано додаткові властивості:

- FullName — ПІБ користувача;
- FaceVectors — колекція векторів облич;
- StudentPhotos — список завантажених фото;
- AgreedToImageProcessing — ознака згоди на обробку зображень;
- MainPhotoFileName — основне фото, що використовується для попереднього перегляду.

Основні сутності бази даних

В архітектурі системи виділено 12 основних сутностей (класи в каталозі Entities). Ключові з них:

- AppUser — користувач (студент, викладач, модератор, адміністратор);
- FaceVector — ембеддинг вектора обличчя;
- Session — навчальна сесія;
- SessionHistory — історія запусків сесій;
- Attendance — факти відвідуваності;
- StudentGroup — зв’язок студентів із групами;
- PlannedSession — заплановані майбутні сесії;
- SessionFaceVector — розпізнані обличчя за конкретну сесію;
- StudentPhoto — фото студентів;
- Group — навчальна група;
- RefreshToken — механізм оновлення токенів автентифікації.

Усі сутності зв’язані між собою через зовнішні ключі, що забезпечує цілісність даних.

На Рис. (2.4) подано схему структури бази даних, яка відображає основні сутності програмного забезпечення, їхні ключові атрибути та зв’язки. Схему

складено відповідно до логіки взаємодії компонентів системи, реалізованої у програмному коді.

Для побудови зв'язків між сутностями в системі було використано механізм конфігурації через Fluent API, що є частиною підходу Code First у технології Entity Framework Core. Зокрема, реалізовано кілька типів асоціацій: один-до-багатьох, як у випадку, коли об'єкт AppUser має декілька пов'язаних записів FaceVectors, StudentPhotos, Sessions і RefreshTokens; багато-до-багатьох, де проміжна сутність StudentGroup описує належність студентів до академічних груп; а також неявні зв'язки один-до-одного, як між PlannedSession та Session. Для забезпечення цілісності даних і зменшення ймовірності появи "висячих" записів, встановлено каскадне видалення пов'язаних сутностей, яке дозволяє автоматично очищати підлеглі об'єкти після видалення основного запису.

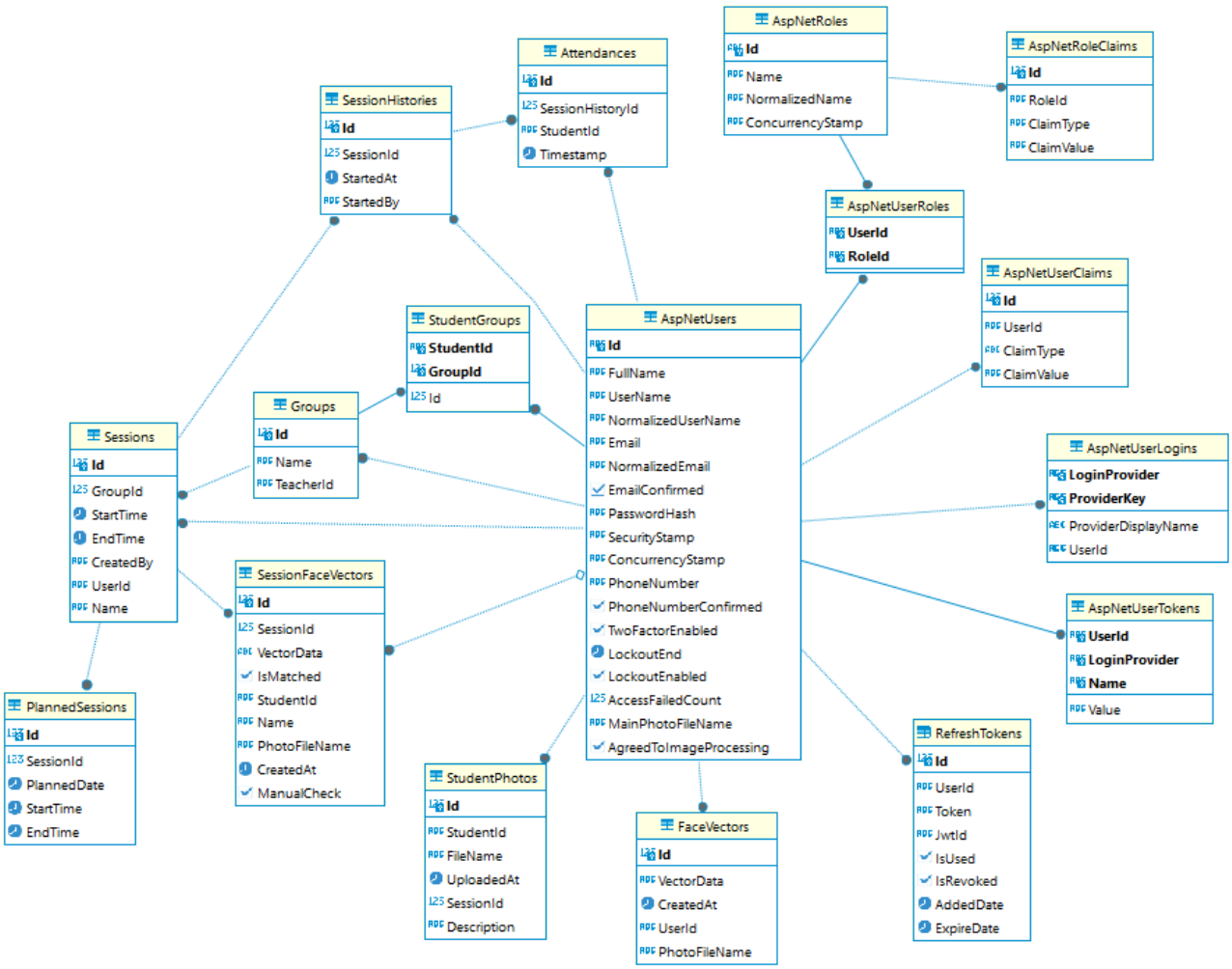

Рисунок 2.4 – Схема логічної структури бази даних системи розпізнавання облич

Серед причин вибору PostgreSQL як основної системи управління базами даних варто виокремити її підтримку складних структур даних, зокрема типів JSON і масивів, що створює резерв для потенційної роботи з неструктурованими або напівструктурованими даними. Крім того, PostgreSQL забезпечує високий рівень транзакційної цілісності, підтримує індексацію різних типів запитів, зокрема по часткових збігах і виразах, а також є перевіреним рішенням для масштабованих систем з перспективою розвитку.

Таким чином, спроєктована структура бази даних відповідає принципам логічної нормалізації, повністю узгоджується з концепцією об'єктно-реляційного відображення (ORM), забезпечує зручність роботи в межах архітектури Clean Architecture, дозволяє централізовано керувати автентифікацією, фіксацією присутності та розпізнаванням облич, а також закладає підґрунтя для подальшого розширення функціоналу системи.

2.3 Архітектурний стиль та реалізація програмної логіки

У процесі проєктування серверної частини системи було застосовано архітектурний стиль Clean Architecture у поєднанні з принципами об’єктно-орієнтованого програмування (ООП). Основна мета — забезпечити модульність, гнучкість, тестованість та незалежність логіки від інфраструктурних реалізацій. Проєктування передбачало створення чітких контрактів між компонентами, що полегшує підтримку та розширення системи в майбутньому.

Особливу увагу приділено побудові прикладної логіки, що реалізується у вигляді сервісів, які відповідають за обробку бізнес-сценаріїв. Ці сервіси взаємодіють із зовнішніми ресурсами або базою даних виключно через інтерфейси, що забезпечує інверсію залежностей. Внутрішні реалізації можуть змінюватися без впливу на логіку застосунку, що відповідає вимогам до проєктування розширюваних систем.

Для організації доступу до даних було застосовано патерн специфікацій (Specification Pattern), який дозволяє описувати запити як окремі об’єкти. Це забезпечує повторне використання умов фільтрації, сортування та включення навігаційних властивостей, а також знижує дублювання коду. Сервіси не оперують запитами напряму — вони делегують побудову запитів репозиторіям, які приймають специфікації як параметри.

Вхідні дані перевіряються на етапі обробки запиту за допомогою механізму валідації. Для кожного об’єкта передачі даних (DTO) створюється окремий валідатор, який визначає обов’язкові поля, обмеження формату, діапазони значень та інші правила. Завдяки цьому вдається гарантувати цілісність і правильність даних ще до передачі в бізнес-логіку. Такий підхід спрощує виявлення помилок, підвищує безпеку системи та дозволяє централізовано керувати перевірками.

Передача даних між клієнтською частиною та сервером здійснюється виключно через DTO. Це забезпечує чітке розмежування між внутрішніми структурами застосунку та зовнішнім API. Для формування відповіді сервер використовує обгортку, яка містить статус виконання, повідомлення та корисні

дані. Така структура відповіді уніфікує формат комунікації та полегшує обробку результатів на фронтенді.

RESTful-проєктування API забезпечує передбачувану структуру запитів та відповідей. Кожен ресурс має унікальний маршрут, дії розділені за HTTP-методами (GET, POST, PUT, DELETE), а клієнт одержує уніфіковану відповідь незалежно від типу запиту. Додатково було закладено підтримку фільтрації, пагінації, автозаповнення та асинхронної перевірки у процесі обробки запитів.

У проєктуванні закладено можливість масштабування — окремі компоненти можуть бути винесені в окремі мікросервіси без зміни логіки основної системи. Наприклад, модуль розпізнавання облич, модуль аналітики чи обробки фотографій. Завдяки низькій зв’язаності між компонентами, система зберігає стабільність при розширенні функціоналу та допускає заміну окремих реалізацій без впливу на інші частини.

Загалом, застосований підхід до проєктування дозволяє побудувати надійну й масштабовану систему, придатну до розвитку та інтеграції з іншими сервісами. Об'єктно-орієнтовані принципи, інтерфейсна взаємодія та стандартизований API — усе це забезпечує високу якість архітектури та придатність до промислового використання.

2.4 Інструментальне забезпечення розробки та середовище реалізації

У процесі створення програмного забезпечення важливо не лише коректно спроєктувати архітектуру й реалізувати логіку системи, але й забезпечити ефективну організацію робочого середовища. Це передбачає вибір оптимальних інструментів для написання, тестування, налагодження, контролю версій, а також розгортання застосунку. Для реалізації проєкту було використано низку сучасних інструментів, які дозволили забезпечити повний цикл розробки — від створення бази даних до деплою й віддаленого керування.

Середовище розробки

Основною IDE для розробки серверної частини проєкту обрано Visual Studio 2022. Цей інструмент забезпечує повну інтеграцію з платформою .NET, підтримує створення ASP.NET Core застосунків, автоматичне застосування міграцій бази даних, зручне налагодження, а також інтеграцію з Git для контролю версій. Завдяки вбудованому профілювальнику і системі запуску тестів, розробка сервісів і API відбувалась із високим рівнем зручності та контролю.

Для фронтенд-частини системи використовувалась Visual Studio Code — легкий, але надзвичайно потужний редактор з підтримкою розширень. У поєднанні з розширеннями ESLint, Prettier, GitLens, Docker та IntelliSense для JavaScript/TypeScript, VS Code дозволив зручно реалізовувати компоненти інтерфейсу, редагувати конфігураційні файли (наприклад, nginx.conf, docker-compose.yml, .env) та перевіряти правильність структури проєкту.

Для роботи з базою даних використовувався DBeaver — потужний багатоплатформовий інструмент для адміністрування СКБД, зокрема PostgreSQL. Його було використано для візуального перегляду структури таблиць, запуску складних SQL-запитів, тестування індексації, діагностики зв’язків між таблицями, а також імпорту та експорту даних. Візуальне представлення зовнішніх ключів значно спростило аналіз побудованої схеми та прискорило процес проєктування.

Серверна інфраструктура та інструменти адміністрування

Для розгортання серверної частини застосунку було використано віртуальний сервер Amazon EC2 (AWS) з ОС Ubuntu 22.04 LTS — стабільним та

підтримуваним середовищем, сумісним із .NET, PostgreSQL, Nginx, Certbot і Docker.

Деплой відбувався вручну через SSH за допомогою MobaXterm — клієнта з підтримкою SFTP, терміналу та візуального файлового менеджера. Він спростив передавання файлів, запуск скриптів, перегляд логів і керування доступом.

Для автозапуску бекенду було налаштовано systemd-сервіси. HTTPS-з'єднання забезпечено сертифікатами Let's Encrypt, отриманими через Certbot. Усі HTTP-запити обробляються через Nginx, який виконує функції реверс-проксі та маршрутизації.

Docker-контейнеризація

У межах проєктування було враховано можливість контейнеризації застосунку з метою забезпечення переносимості, стандартизації середовища розгортання та підтримки CI/CD. Було створено Dockerfile для серверної частини (ASP.NET Core backend), який дозволяє збирати образ застосунку та запускати його у ізольованому середовищі.

Також створено файл docker-compose.yml, який автоматизує розгортання backend-сервісу разом із супутніми компонентами, зокрема PostgreSQL (у ролі СКБД) та Nginx (як реверс-проксі та точка входу для HTTPS-з'єднань).

Фронтенд-частина (React) не контейнеризувалася. Замість цього було створено білд (npm run build), після чого статичні файли (index.html, bundle.js, assets) було вручну завантажено на сервер і розміщено в окремому каталозі, який обслуговується через Nginx як статичний сайт. Такий підхід дозволив уникнути зайвої складності та швидко налаштувати розгортання на продуктивному середовищі.

Контроль версій

Весь код зберігається у Git-репозиторії на платформі GitHub. Це дало змогу контролювати версії, фіксувати історію змін, використовувати гілки для розділення функціоналу, а також забезпечити співпрацю з іншими учасниками команди. У Visual Studio та VS Code інтегровано Git-плагіни для зручної роботи з комітами, переглядом змін та вирішенням конфліктів.

ІІІ. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Реалізація серверної частини (Backend)

Реалізація серверної частини системи здійснювалася на базі фреймворку ASP.NET Core із застосуванням архітектурного стилю Clean Architecture, що передбачає чітке розділення відповідальностей між рівнями Presentation, Application, Domain та Infrastructure. Такий підхід дозволив досягти високої модульності, гнучкості та масштабованості застосунку.

Загальна структура

Основна логіка взаємодії з користувачами, розпізнавання облич, управління сесіями та збереження даних реалізована у вигляді сервісів. Усі сервіси реалізують відповідні інтерфейси з Application-рівня, а їхня реалізація міститься в Infrastructure-рівні.

Дані зберігаються у базі PostgreSQL, доступ до якої здійснюється за допомогою Entity Framework Core з підходом Code First. Уся взаємодія з базою інкапсульована в репозиторіях, що дозволяє легко тестувати бізнес-логіку та уникати дублювання коду.

Основні сервіси системи

FaceNetDetectorService

Сервіс відповідає за виявлення облич на зображеннях, що надходять із камери користувача. Він є першим етапом у ланцюжку розпізнавання особи та забезпечує визначення координат облич для подальшої обробки.

Усередині сервісу використовується модель SCRFD, збережена у форматі ONNX-файлу det_2.5g.onnx. Це оптимізована модель виявлення облич, яка балансує між швидкістю та точністю, і підходить для використання в реальному часі. Модель приймає на вхід зображення розміром 640×640, і на виході повертає три рівні детекції (multi-scale detection), кожен з яких містить:

- карту впевненості (confidence map) — ймовірність наявності обличчя в кожній області;
- bounding boxes — координати прямокутників навколо ймовірних облич.

Використання ONNX-моделі забезпечується через Microsoft.ML.OnnxRuntime, що дозволяє виконувати інференс безпосередньо в .NET-середовищ

Принцип роботи сервісу

1. Підготовка зображення:
- зображення завантажується з байтового масиву;
- масштабується до 640×640;
- перетворюється у тензор float32 форми [1, 3, 640, 640] (формат CHW);
- значення пікселів нормалізуються до діапазону [0…1].
2. Інференс моделі:
- створюється сесія ONNX-моделі;
- передається підготовлений тензор;
- модель повертає три виходи — по кожному рівню детекції (stride 8, 16, 32);
- кожен вихід містить впевненість і координати bounding boxes.
3. Постобробка:
- для кожної детекції обчислюється косинусна впевненість (через Sigmoid);
- координати прямокутників декодуються в піксельні значення на основі stride;
- виконується non-maximum suppression (NMS), який усуває перекриття між детекціями на основі метрики IoU (Intersection over Union).

На Рис. 3.7 представлено фрагмент коду, де виконується обробка тензора та побудова bounding boxes на основі результатів SCRFD-моделі.

4. Результат: метод DetectFacesAsync() повертає список об’єктів FaceBoundingBox, кожен з яких містить: X, Y, Width, Height, Score.

```
const int anchorsPerPixel = 2;
float sx = (float)ow / S, sy = (float)oh / S;
var faces = new List<FaceBoundingBox>();

for (int l = 0; l < 3; l++)
{
    var scores = scoreL[l];
    var boxes = boxL[l];
    int anchors = scores.Length;                        // 12800 / 3200 / 800
    int sid = (int)Math.Sqrt(anchors / anchorsPerPixel);
    int st = _stride[l];

    for (int a = 0; a < anchors; a++)
    {
        float conf = Sigmoid(scores[a]);
        if (conf < 0.55f) continue;

        int g = a / anchorsPerPixel;
        int gy = g / sid;
        int gx = g % sid;

        float left = boxes[a * 4 + 0] * st;
        float top = boxes[a * 4 + 1] * st;
        float right = boxes[a * 4 + 2] * st;
        float bot = boxes[a * 4 + 3] * st;

        float cx = (gx + 0.5f) * st;
        float cy = (gy + 0.5f) * st;

        int x = (int)((cx - left) * sx);
        int y = (int)((cy - top) * sy);
        int xe = (int)((cx + right) * sx);
        int ye = (int)((cy + bot) * sy);

        int w = xe - x, h = ye - y;
        if (w <= 0 || h <= 0) continue;

        faces.Add(new() { X = x, Y = y, Width = w, Height = h, Score = conf });
    }
}
```

Рисунок 3.1 Побудова прямокутників облич на основі виходу моделі det_2.5g.onnx

Перевагами підходу з використанням моделі SCRFD є її висока точність, зокрема при розпізнаванні облич із частковими перекриттями або при поворотах голови. Завдяки оптимізованій архітектурі inference моделі займає менше 100 мс навіть на звичайному процесорі без використання GPU-прискорення, що робить її придатною для реального часу. Крім того, SCRFD демонструє стабільні результати незалежно від кольору шкіри, форми обличчя, освітлення та інших зовнішніх чинників, що забезпечує її універсальність у практичному застосуванні.

FaceNetEmbedderService

Сервіс FaceNetEmbedderService відповідає за побудову ембеддінгів — векторів ознак, що унікально описують обличчя. Для цього використовується модель FaceNet, конвертована у формат ONNX (facenet.onnx). Дана модель широко використовується в системах біометричної ідентифікації, оскільки дозволяє представляти кожне обличчя у вигляді вектора з фіксованою

розмірністю (512 чисел), який можна порівнювати з іншими за допомогою метрик відстані

Процес формування ембеддінгу включає кілька етапів:

1. Підготовка зображення – Кожне виявлене обличчя вирізається із зображення згідно координат FaceBoundingBox. Отриманий фрагмент масштабується до розміру 160×160 пікселів, що є вимогою вхідного шару моделі.
2. Нормалізація – кожен піксель обличчя нормалізується за формулою:

$$\text{значення} = \frac{RGB - 127.5}{128.0}$$

RGB — значення кольору кожного каналу (Red, Green, Blue) пікселя

127.5 — середнє значення діапазону [0, 255]

128.0 — нормалізаційний коефіцієнт, який масштабує значення.

Це дозволяє привести значення пікселів до діапазону [-1.0, 1.0], до формату якому була навчена модель.

3. Формування тензора – дані конвертуються у тензор форми [1, 160, 160, 3], де 1 — batch size, 3 — канали кольору (RGB).
4. Інференс через ONNX Runtime – після підготовки дані передаються в модель через Microsoft.ML.OnnxRuntime. На виході повертається масив довжини 512 — ембеддінг обличчя.
5. Результат – для кожного обличчя формується окремий вектор, який зберігається в базі даних або використовується для порівняння з іншими обличчями.

Реалізація основного методу класу представлена на Рис.3.2

```
public class FaceNetEmbedderService : IFaceNetEmbedderService
{
    private readonly InferenceSession _session;

    public FaceNetEmbedderService()
    {
        var modelPath = Path.Combine(AppContext.BaseDirectory, "Models", "faceNet.onnx");
        _session = new InferenceSession(modelPath);
    }

    private float[] GetEmbedding(byte[] faceBytes)
    {
        using var ms = new MemoryStream(faceBytes);
        using var img = Image.Load<Rgb24>(ms);
        img.Mutate(x => x.Resize(160, 160));

        float[] input = new float[160 * 160 * 3];
        int idx = 0;
        for (int y = 0; y < 160; y++)
        {
            for (int x = 0; x < 160; x++)
            {
                var pixel = img[x, y];
                input[idx++] = (pixel.R - 127.5f) / 128f;
                input[idx++] = (pixel.G - 127.5f) / 128f;
                input[idx++] = (pixel.B - 127.5f) / 128f;
            }
        }

        var inputTensor = new DenseTensor<float>(input, new[] { 1, 160, 160, 3 });
        var result = _session.Run(new List<NamedOnnxValue> { NamedOnnxValue.CreateFromTensor("image_input", inputTensor) });

        return result.First().AsEnumerable<float>().ToArray();
    }
```

Рисунок 3.2 Формування ембеддінгу обличчя в FaceNetEmbedderService

FaceNetProcessorService

Центральний сервіс, що реалізує повний цикл розпізнавання:

- Виявлення облич (через FaceNetDetectorService)
- Генерація ембеддінгів (через FaceNetEmbedderService)
- Порівняння з наявними векторами студентів (через FaceNetRecognizerService)
- Збереження результатів у базі (через SessionFaceVectorService, FaceVectorService, FilesService)

Сервіс також вирізає фрагменти облич, зберігає зображення та повертає результати у вигляді списку розпізнаних користувачів з іменем, фото та ID.

FaceNetRecognizerService

Виконує порівняння векторів ознак за допомогою косинусної схожості. Якщо відстань між векторами менша за заданий поріг, вважається, що це той самий користувач. Сервіс використовується для порівняння збережених векторів з новими.

Інші ключові сервіси

Окрім модулів розпізнавання, у системі реалізовано низку функціональних сервісів, що відповідають за управління користувачами, сесіями, групами, відвідуваністю та взаємодію з базою даних. Їхня логіка згрупована відповідно до

принципів архітектури — інтерфейси розташовані в Application-рівні, реалізації — в Infrastructure.

До ключових сервісів належать:

UserService — керує створенням, редагуванням та блокуванням користувачів, а також зміною ролей і отриманням студентів за групою.

GroupService — відповідає за CRUD-операції над навчальними групами, а також прив’язку студентів і викладачів.

SessionService та SessionHistoryService — реалізують логіку створення сесій, запуску та збереження історії.

AttendanceService — фіксує присутність студентів, обробляє як ручні, так і автоматичні відмітки.

FaceVectorService — зберігає ознакові вектори облич користувачів.

FilesService — відповідає за збереження зображень, що надсилаються з фронтенду.

JwtService — забезпечує генерацію access/refresh токенів для автентифікації.

Інші сервіси, такі як для роботи з фото студентів, запланованими сесіями, позначками відвідування тощо, також реалізовані в рамках проєкту. Повний перелік сервісів подано на Рис. 3.3.

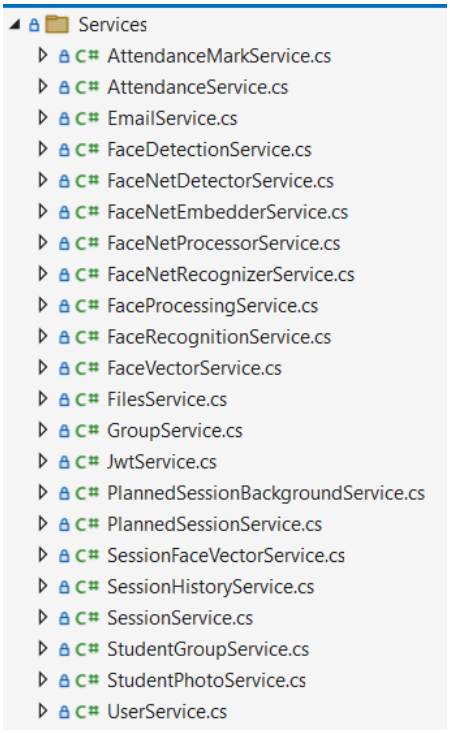

Рисунок 3.3 Структура сервісів у проєкті (знімок дерева файлів у Visual Studio Code)

Обробка запиту на розпізнавання

При надходженні зображення:

- Фото конвертується у байти
- Визначаються координати облич
- Кожне обличчя вирізається та подається до моделі FaceNet
- Результати звіряються з відомими векторами
- Збіги зберігаються, а нові обличчя додаються до бази

Клієнт отримує відповідь з інформацією про розпізнаних осіб

3.2 Реалізація клієнтської частини (Frontend)

Клієнтська частина розробленої системи реалізована за допомогою бібліотеки React — сучасного інструмента для створення односторінкових інтерфейсів (SPA). Вибір саме цього фреймворку обумовлений його поширеністю, активною підтримкою, широкою екосистемою та високою продуктивністю під час роботи з динамічними даними. У якості модуля для збірки застосунку використовується Vite — надзвичайно швидкий інструмент, який забезпечує миттєве перезбирання, гаряче оновлення модулів (HMR) та оптимізацію під продакшн.

Структура проєкту є модульною дивись Рис 3.6: усі компоненти згруповані у відповідні папки за ролями користувачів (наприклад, Admin, Student, Teacher), а загальні елементи винесені в окрему директорію components. Це забезпечує кращу масштабованість проєкту та спрощує навігацію в коді.

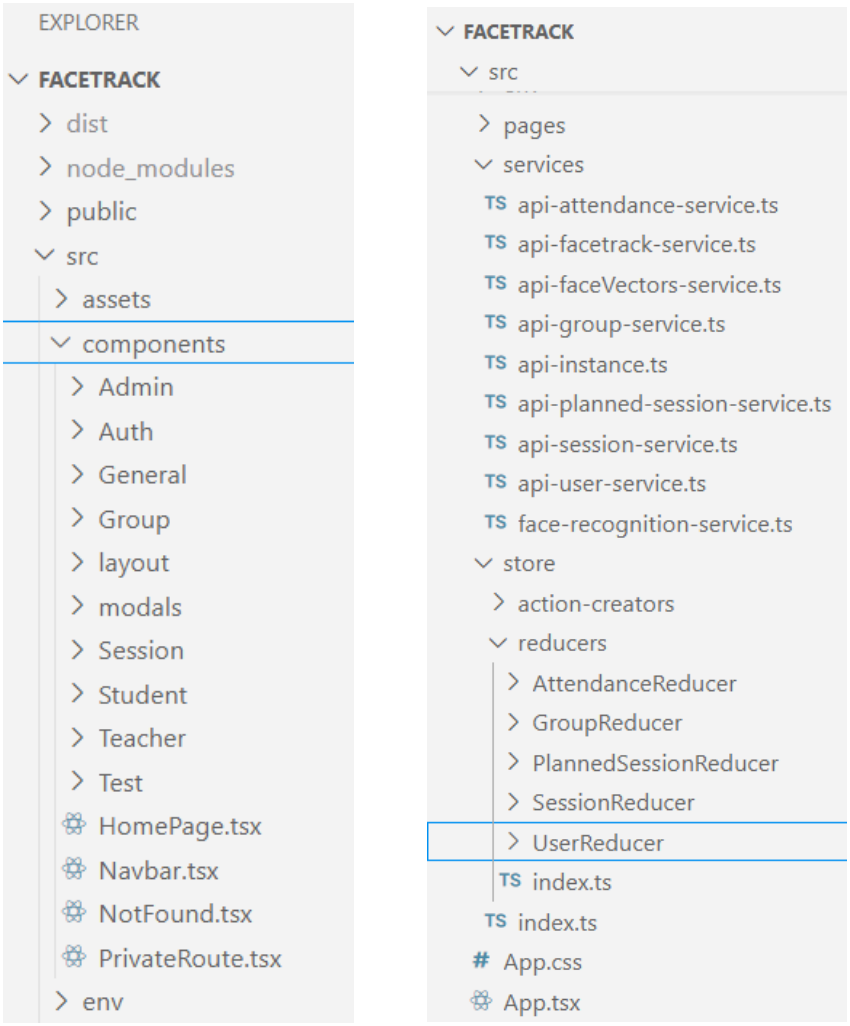

Рисунок 3.4 Структура директорій клієнтської частини проєкту

У проєкті активно застосовуються шаблони проєктування Redux для управління станом. Завдяки Redux-архітектурі вдалося централізувати обробку стану застосунку, зокрема зберігання інформації про користувача, список груп, сесій, історії відвідуваності тощо. Усі ред'юсери згруповано у папці store/reducers, а відповідні action creators — в store/action-creators. Наприклад, для роботи з користувачами реалізовано UserReducer, userActions та відповідні сервіси у api-user-service.ts дивись фрагмент коду Рис. 3.5.

```
import { UserState, UserActions, UserActionTypes } from "./types";

const initialState: UserState = {
    user: null,
    users: [],
    loggedInUser: null,
    token: null,
    refreshToken: null,
    role: null,
    loading: false,
    error: null,
    currentPage: 1,
    totalPages: 0,
    pageSize: 10,
    totalCount: 0,
};

Complexity is 31 Bloody hell...
const UserReducer = (state = initialState, action: UserActions): UserState => {
    switch (action.type) {
        case UserActionTypes.USER_START_REQUEST :
            return { ...state, loading: true, error: null };
        case UserActionTypes.LOGIN_USER_SUCCESS:
            return {
                ...state,
                user: action.payload.user || null,
                loggedInUser: action.payload.user || null,
                token: action.payload.token || null,
                refreshToken: action.payload.refreshToken || null,
                role: action.payload.user?.role || null,
                loading: false,
            };
        case UserActionTypes.FETCH_STUDENTS_SUCCESS:
            return {
                ...state,
                users: action.payload.users,
                currentPage: action.payload.currentPage,
                totalPages: action.payload.totalPages,
                pageSize: action.payload.pageSize,
                totalCount: action.payload.totalCount,
                loading: false,
            };
```

Рисунок 3.5 фрагмент коду UserReducer

З'єднання з сервером реалізоване через бібліотеку axios, яку інкапсульовано в окремому модулі api-instance.ts дивись Рис. 3.6. Цей модуль

містить логіку для підстановки JWT токена до кожного запиту, а також обробку ситуацій з протермінованими токенами: при виникненні помилки авторизації (401) відбувається автоматичне оновлення токена через спеціальний endpoint /refresh-token, без необхідності втручання користувача. Це забезпечує безперервну роботу сесії користувача, покращуючи UX.

```
import axios from "axios";
import { APP_ENV } from "../env";

const instance = axios.create({
    baseURL: APP_ENV.BASE_URL + "/api",
});

instance.interceptors.request.use(
    Complexity is 4 Everything is cool!
    (config: any) => {
        const token = getAccessToken();
        if (token) {
            config.headers["Authorization"] = "Bearer " + token;
        }
        return config;
    },
    (error) => Promise.reject(error)
);

instance.interceptors.response.use(
    (res) => res,
    Complexity is 11 You must be kidding
    async (err) => {
        const originalConfig = err.config;

        if (err.response?.status === 401 && !originalConfig._retry && getAccessToken()) {
            originalConfig._retry = true;
            try {
                const rs = await refreshAccessToken();
                if (!rs) throw new Error("Failed to refresh token");

                const { accessToken, refreshToken } = rs.data;
                setAccessToken(accessToken);
                setRefreshToken(refreshToken);

                originalConfig.headers["Authorization"] = "Bearer " + accessToken;
                return instance(originalConfig);
            } catch (_error) {
                console.error("Failed to refresh token, logging out user.");
                removeTokens();
                window.location.href = "/login";
                return Promise.reject(_error);
            }
        }

        return Promise.reject(err);
    }
);
```

Рисунок 3.6 Axios із підключенням токена та обробкою помилок

Уся взаємодія між фронтендом і серверною частиною побудована на базі REST API дивись Рис. 3.7. Наприклад, автентифікація реалізована через POST-запит /auth/login, що повертає JWT-токен. Усі інші дії (наприклад, керування

користувачами, створення сесій, зміна ролей, розпізнавання облич) також виконуються виключно через API-запити до backend-сервісу. Це дозволяє чітко дотримуватися принципу розділення відповідальностей (Separation of Concerns) та забезпечити масштабованість.

```
import instance, { getAccessToken } from "./api-instance";
import type { AxiosError } from "axios";

const responseBody: any = (response: any) => response.data;

const requests = {
    get: (url: string, params?: any) => instance.get(url, { params }).then(responseBody),
    post: (url: string, body?: any) => instance.post(url, body).then(responseBody),
    put: (url: string, body?: any) => instance.put(url, body).then(responseBody),
    delete: (url: string) => instance.delete(url).then(responseBody),
};

const User = {
    login: (userData: any) => requests.post("/auth/login", userData),
    refreshToken: (refreshToken: string) => requests.post("/RefreshToken", { refreshToken }),
    getStudentByGroupId: (groupId: number) => requests.get(`user/group/${groupId}`),
    addStudentToGroup: (email: string, groupId: number) => requests.post("/user/addStudent", { email,groupId }),
    auditStudent: (email: string) => requests.get("/user/auditStudent", { email }),
    registerUser: (data: { email: string; password: string; confirmPassword: string, groupId: number, fullName: string }) =>
      requests.post("/user/registerStudent", data),
    filterUsers: (filter: any) => requests.post("/user/filter", filter),
    updateUser: (updatedUser: { id: string; fullName: string; email: string; }) => requests.put("/User/update", updatedUser),
    deleteUser: (id: string) =>  requests.delete(`/User/delete/${id}`),
    changeUserRole: (userId: string, newRole: string) => requests.post('/User/changerole', { userId, newRole }),
```

Рисунок 3.7 API-запити

Інтерфейс побудований з урахуванням ролей користувачів. Кожен тип користувача (адміністратор, викладач, студент) після входу потрапляє на відповідну панель (Dashboard), яка містить лише релевантні функції. Наприклад, адмін бачить статистику користувачів, кнопки для швидкого створення груп чи сесій, доступ до сторінок керування дивись Рис. 3.8. Викладач бачить перелік своїх груп і сесій, студент — лише поточні доступні сесії для відмітки присутності через розпізнавання обличчя.

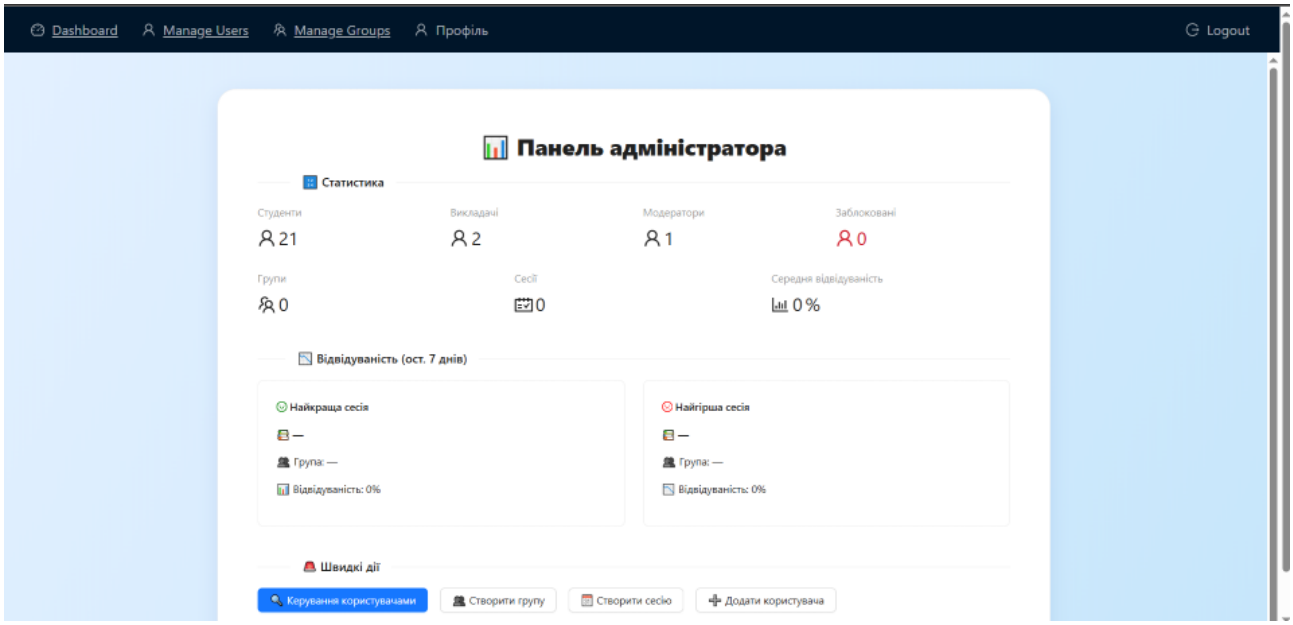

Рисунок 3.8 – Головна панель адміністратора з кнопками керування (Dashboard)

Інтерфейс адаптивний та зручний у використанні на різних пристроях. Компоненти інтерфейсу створені з використанням бібліотек antd та bootstrap, що дозволило поєднати кастомізовану стилізацію з перевіреними UI-компонентами. Наприклад, форма входу стилізована відповідно до брендованого дизайну системи, із використанням кольорового градієнту, а на панелях керування використовуються значки, діаграми та кнопки з інтуїтивним розташуванням.

Окрему увагу приділено функціоналу розпізнавання обличчя на клієнті. На сторінці відмітки відвідуваності компонент react-webcam захоплює зображення з камери користувача, після чого надсилає його на backend для розпізнавання дивись Рис. 3.9. У разі успішного визначення особи, інтерфейс виводить підтвердження про успішну дію. Додатково реалізовано попередження про згоду на обробку зображення, що враховує юридичний аспект.

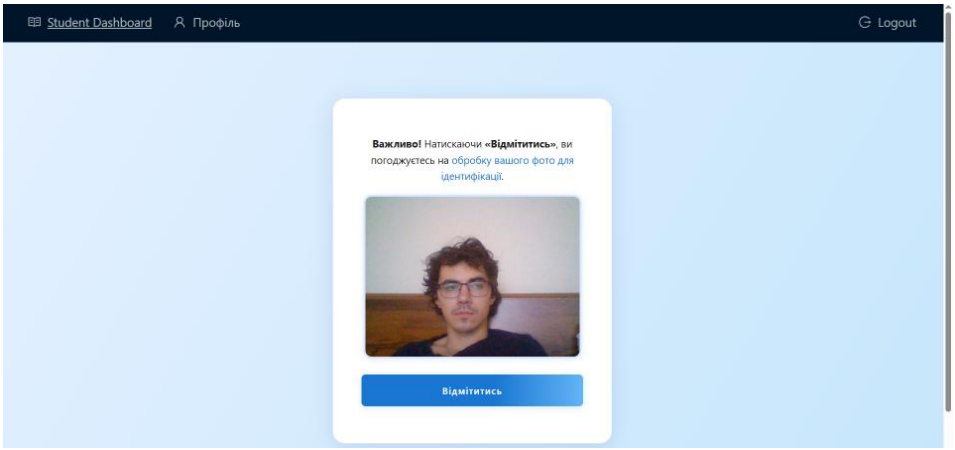

Рисунок 3.9 Сторінка розпізнавання облич для відмітки присутності (Webcam)

Для керування маршрутами застосовується react-router-dom, з підтримкою захищених маршрутів (PrivateRoute). Це дозволяє забезпечити доступ до окремих сторінок лише користувачам з відповідною роллю, з автоматичним перенаправленням у разі несанкціонованого доступу.

Варто зазначити, що інтерфейс підтримує базову обробку помилок. Наприклад, при неправильному введенні пароля користувач бачить повідомлення про помилку входу, при проблемах із мережею — виводиться попередження. Для реалізації сповіщень використовується компонент message із бібліотеки Ant Design.

Розгортання клієнтської частини здійснюється за допомогою стандартної команди npm run build, яка створює оптимізовану production-версію проєкту. Отриманий статичний вміст (HTML, CSS, JS) завантажується на сервер, де обслуговується через Nginx як frontend частина вебзастосунку. Важливо, що всі запити до API перенаправляються на backend-сервер через налаштовану proxy-конфігурацію, що дозволяє уникнути CORS-помилок.

Загалом, клієнтська частина системи реалізована у відповідності до сучасних стандартів веброзробки, з акцентом на зручність, безпеку та масштабованість. Вона дозволяє інтерактивно взаємодіяти з користувачем, обробляти складну логіку і забезпечує зручний інтерфейс для адміністраторів, викладачів і студентів у межах одного цілісного застосунку.

3.3 Розгортання програмного забезпечення

Для розгортання серверної частини системи було обрано хмарну інфраструктуру AWS Lightsail, яка забезпечує швидкий старт, базову керованість і підтримку всіх необхідних інструментів для Linux-інстансів. У процесі налаштування було створено віртуальний сервер на основі Ubuntu 22.04 LTS дивись Рис. 3.10, що забезпечує стабільність, актуальність пакетів та повну сумісність із Docker, Nginx і Certbot.

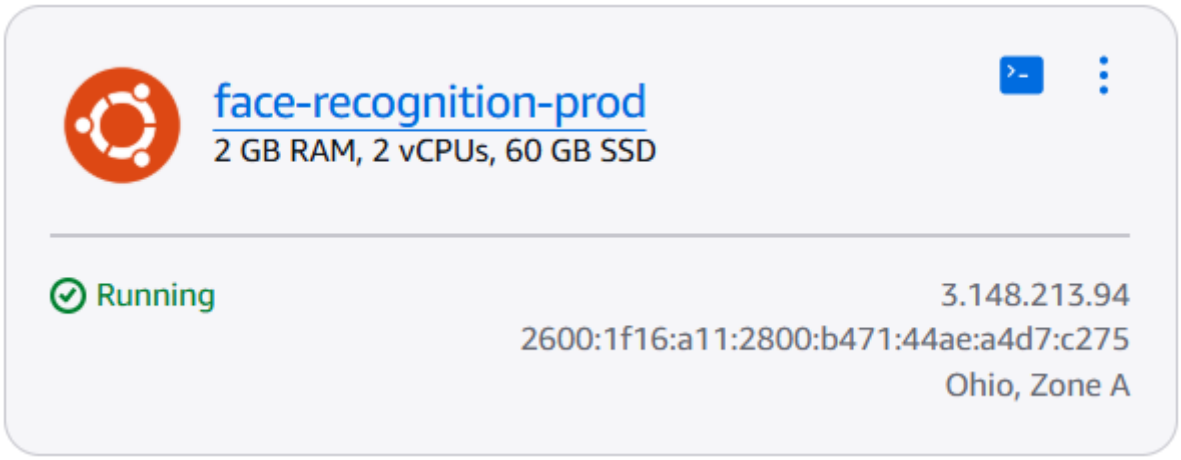

Рисунок 3.10 Віртуальний сервер на основі Ubuntu 22.04

Інстанс має 2 віртуальні процесорні ядра, 2 ГБ оперативної пам’яті та 60 ГБ SSD-сховища. Така конфігурація є мінімально необхідною для запуску всіх компонентів системи: backend API, бази даних PostgreSQL, веб-інтерфейсу, та забезпечення взаємодії через HTTPS. Водночас, цей сервер не є оптимальним для тривалого або масштабного використання: він більше підходить для тестування або демонстрації системи, оскільки не розрахований на середньостатистичне чи інтенсивне навантаження, яке характерне для продуктивного середовища. У випадку розгортання в реальному сценарії використання, доцільно розглянути сервери із вищими характеристиками або з підтримкою GPU

Після запуску інстансу через SSH було виконано оновлення системи та встановлення необхідних утиліт. Було налаштовано Docker як платформу контейнеризації. За допомогою стандартної команди docker build -t face-api . було зібрано образ backend-застосунку, а запуск здійснено з монтуванням директорії

для зображень (/images) та публікацією порту 8080 через -p 5081:8080. Також було створено окремий контейнер із PostgreSQL, налаштовано збереження даних та відкрито порт 5022 дивись Рис. 3.11

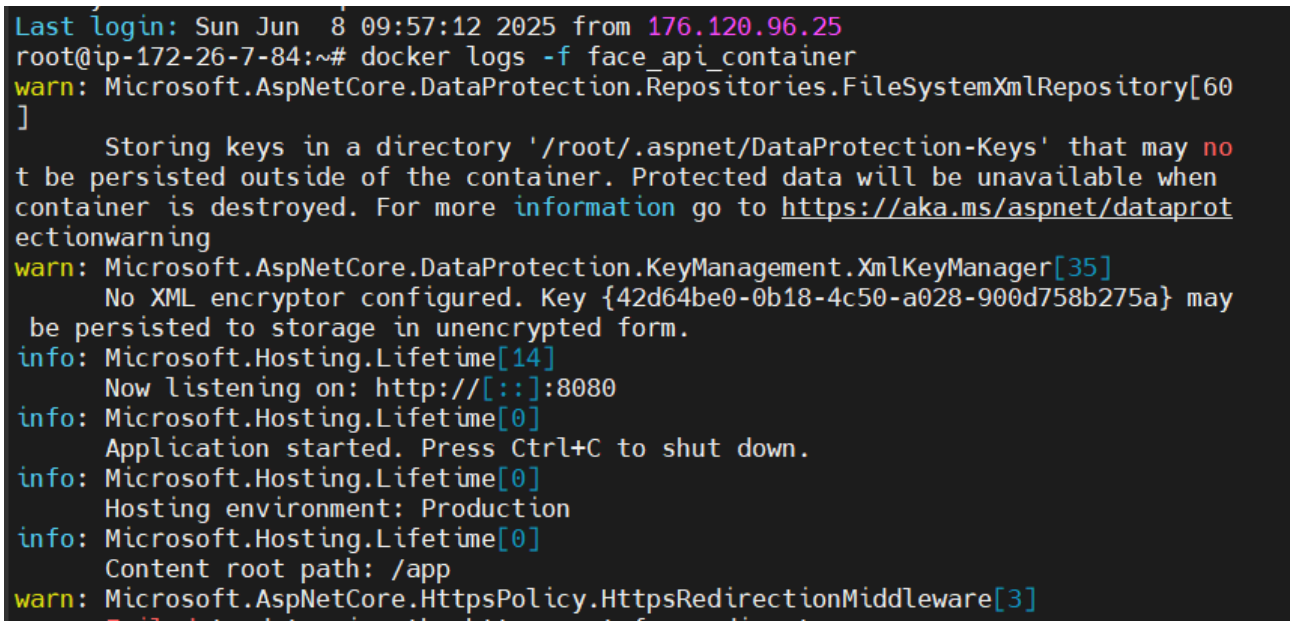

Рисунок 3.11 Лог запуску контейнера з API-інтерфейсом face-api.

Фронтенд-система була зібрана командою npm run build, після чого згенеровані файли (HTML, JS, CSS) розміщено у директорії /var/www/face-frontend. Для керування маршрутизацією між статичними файлами та API-запитами налаштовано Nginx як зворотній проксі. У конфігураційному файлі default створено дві основні секції: одна для обслуговування клієнтської частини, інша — для проксирування запитів /api/ до backend-контейнера дивись Рис. 3.12.

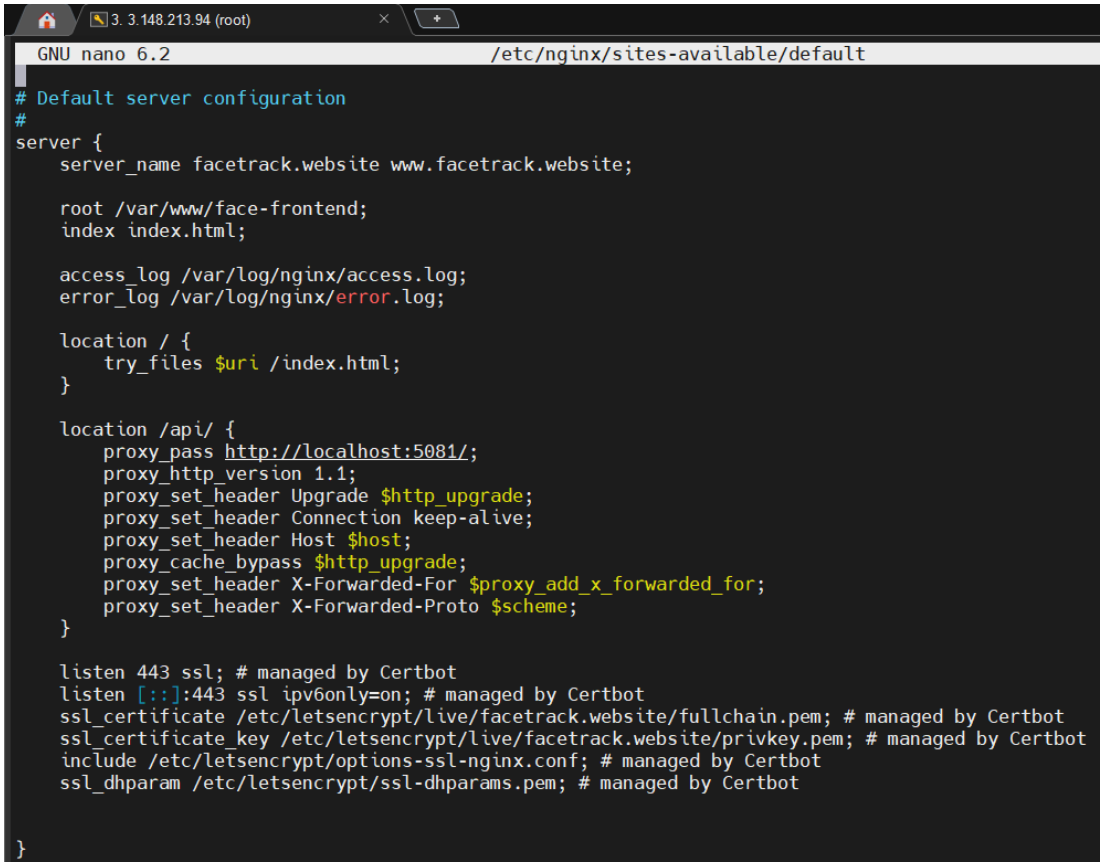

Рисунок 3.12 Конфігурація Nginx для роздачі frontend-файлів і проксирування запитів до backend API.

Для забезпечення захищеного з'єднання з сайтом використано утиліту Certbot, яка автоматично отримала та встановила SSL-сертифікат від Let's Encrypt. Доменне ім'я facetrack.website було закріплено за сервером, а HTTPS-з'єднання перевірено за допомогою команди curl -I https://facetrack.website. Це дозволяє безпечно передавати дані, зокрема фотографії та облікову інформацію.

Щоб сервер працював безперервно, усі основні служби налаштовано на автоматичний запуск: Docker-контейнери стартують із параметром --restart=always, а nginx запускається через systemd. Моніторинг здійснювався за допомогою аналізу логів командою docker logs -f face_api_container, а також через sudo tail -f /var/log/nginx/error.log у разі потреби.

У підсумку було реалізовано повноцінну систему розгортання, яка охоплює контейнери, реверс-проксі, HTTPS, фаєрвол і доменне ім'я. Такий підхід дозволив запустити функціональну версію системи в реальному мережевому середовищі, яка готова до демонстрації та подальшого масштабування за потреби.

IV.ТЕСТУВАННЯ ТА ДОСЛІДНА ЕКСПЛУАТАЦІЯ СИСТЕМИ

4.1 Загальні умови тестування

Загальне тестування розробленої системи проводилося з метою перевірки її цілісного функціонування в умовах, наближених до реального навчального процесу. Основну увагу було зосереджено не на ізольованих компонентах, а на повному циклі користувацької взаємодії: від моменту запуску інтерфейсу до збереження результату у базу даних. Такий підхід дозволив зафіксувати усі ключові аспекти поведінки системи в реальному часі.

Тестування охоплювало різні комбінації пристроїв та браузерів, а також моделювання ситуацій зі слабким інтернет-з'єднанням. Це дозволило виявити особливості рендерингу інтерфейсу, поведінку API при затримках та стійкість до помилок при нестабільних умовах. Кожен тест виконувався вручну з фіксацією результатів у логах, що забезпечувало об'єктивну оцінку реакції системи на зовнішні чинники.

Перед початком було розгорнуто повноцінне середовище, включаючи домен, HTTPS-доступ, фронтенд, API-сервіс і базу даних. Це дозволило перевірити не лише технічну працездатність, а й надійність комунікації між усіма рівнями архітектури. Важливим аспектом також було спостереження за часом реакції після натискання основної кнопки на клієнті, щоб оцінити комфорт взаємодії.

Крім базових сценаріїв, були змодельовані крайові ситуації: відправлення пустих знімків, передача низькоякісних фото, закриття вкладки під час активної сесії, повторне оновлення сторінки тощо. Метою було виявити, чи викликає така поведінка системні помилки або дестабілізацію взаємодії. У більшості випадків вдалося забезпечити адекватну реакцію системи — або через механізм обробки помилок, або через повторну ініціалізацію процесу без втрати попередніх даних.

У фокусі тестування також перебувала зручність для кінцевого користувача. Сценарії проходили за логікою «одна дія — один результат», без складних переходів або зайвих елементів. Усі тексти були локалізовані, а інтерфейс зведений до мінімуму, необхідного для розуміння. Оцінювалась не

лише функціональність, а й візуальна логіка — наприклад, відображення повідомлень, фідбек після дії, зміна стану кнопок.

Протягом усього тестування важливим показником була швидкодія у ключових точках — час завантаження сторінки, момент активації камери, передача зображення на бекенд, повернення результату. Це дозволило оцінити загальний рівень оптимізації та виявити вузькі місця, які впливають на швидкість реакції.

Окремим етапом стала перевірка стабільності роботи при повторному запуску дій — наприклад, подвійне натискання кнопки «відмітитись», оновлення сторінки під час обробки запиту або випадкова перерва зв’язку з сервером. Усі ці ситуації були змодельовані з метою перевірки, чи не призведуть вони до помилок або дублювання записів.

У підсумку тестування дало змогу зробити висновок про базову готовність системи до повноцінного використання. У процесі були виявлені кілька критичних деталей, які вдалося оперативно усунути, а також сформовано перелік побажань щодо подальшої оптимізації взаємодії. Саме загальне тестування створило основу для переходу до наступних етапів — вимірювання продуктивності та точності розпізнавання.

4.2 Функціональне тестування серверної частини

Для оцінювання продуктивності та стійкості серверної частини система була розгорнута у вигляді двох Docker-контейнерів: face-api, що містить ASP.NET Core-бекенд разом із моделями SCRFD + FaceNet, і postgres, у якому працює база даних. Обидва контейнери запускаються з параметром --restart=always, а статичні файли фронтенду та TLS-термінація обслуговуються Nginx-проксі.

Методика полягала у відтворенні двох послідовних сценаріїв навантаження. У першому — контрольному — 10 студентів протягом п’ятнадцяти хвилин реєструвалися, робили селфі та надсилали знімки на ручну модерацію, оскільки їхні ембеддінги ще не були збережені. У другому — піковому — студенти, для яких векторні представлення вже існували, майже синхронно натискали «Відмітитись». У цей момент бекенд мусив завантажити фотографію, згенерувати ембеддинг, порівняти його з базою, а за збігу — зафіксувати присутність у PostgreSQL.

На діаграмі середнього використання процесора Lightsail-інстансу за добу (дивись рис. 4.1) помітно, що під час першої хвилі навантаження обчислювальні ресурси миттєво зросли до ≈ 55 %, проте одразу повернулися у «стійку» зелено позначену область (< 20 %). Жодного запиту не було втрачено, а час відповіді залишився в межах трьох секунд. Натомість друга хвиля викликала два короткочасні сплески 54–60 % і перевела інстанс у так звану burstable zone (оранжева смуга графіка). Саме в ці кількахвилинні інтервали клієнти почали спостерігати затримки понад сім секунд, що частково призвело до тимчасового зростання черги запитів.

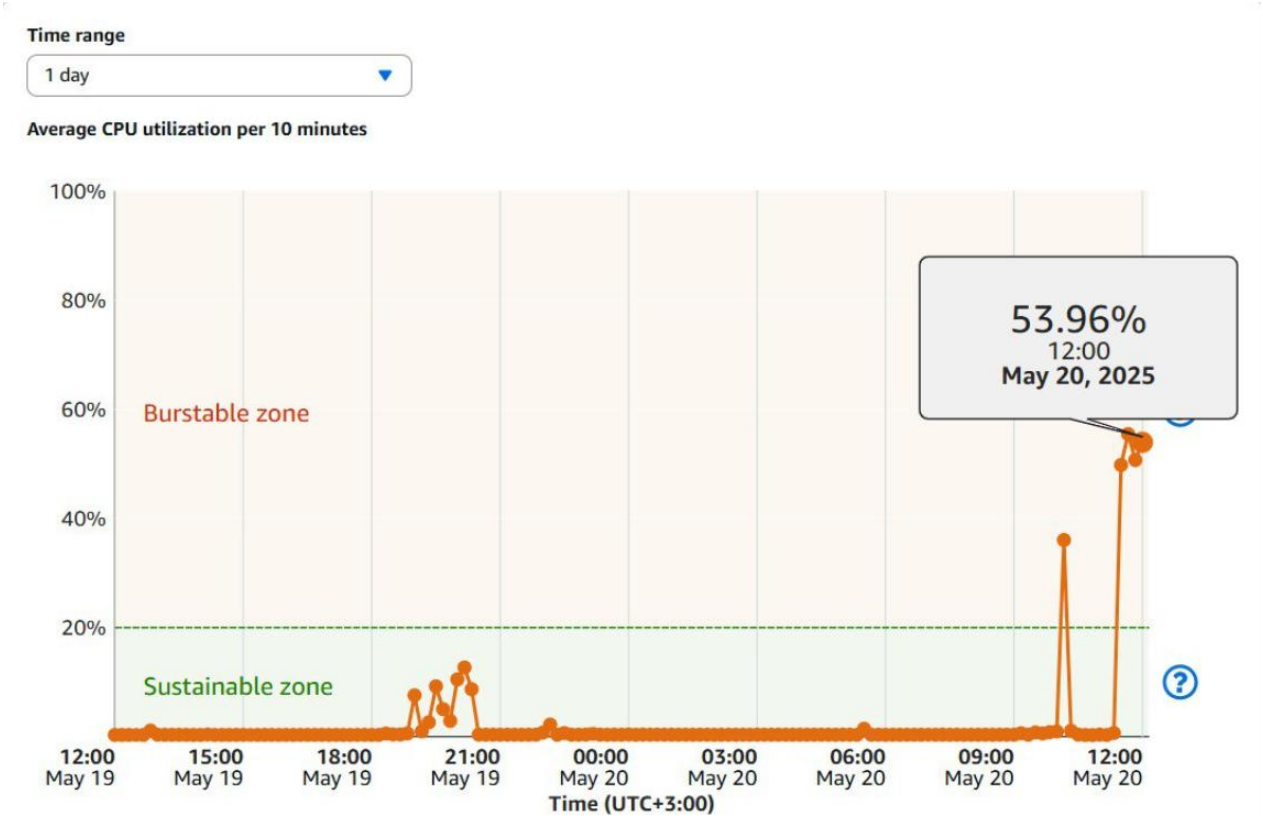

Рисунок 4.1 Середня завантаженість CPU Lightsail-інстансу за 24 години;

Проведені експерименти показали, що тестовий інстанс AWS Lightsail із базовою конфігурацією цілком справляється з помірним навантаженням (десяток паралельних запитів) і не допускає втрати даних чи збоїв; однак навіть короткий піковий сплеск миттєво виводить CPU у burstable-режим і збільшує час відповіді до відчутних пауз. Це свідчить, що для реального розгортання необхідно або оптимізувати обробку зображень (попереднє ресайз-масштабування, окремий диск під PostgreSQL), або переїхати на потужніший сервер із додатковими ядрами, більшим обсягом RAM і, за потреби, GPU — лише тоді система зможе стійко обслуговувати сотні одночасних запитів без затримок.

4.3 Функціональне тестування клієнтської частини

Клієнтський інтерфейс — односторінковий застосунок на React, який у реальному часі працює з камерою користувача, надсилає знімок до бекенду й отримує результат розпізнавання. Його тестування проводили у браузері Chrome, а також у Firefox і Safari на ноутбуках та мобільних пристроях. Мета — переконатися, що усі типові дії (вхід, вибір сесії, знімок, обробка відповіді, повідомлення про помилки і тому подібне) виконуються без збоїв і зрозумілі користувачу.

1) Ключовий сценарій «Студент → Відмітитись»

Авторизація. Форма логіну перевіряє коректність електронної пошти, довжину пароля та відображає підказки при помилках. У середньому від натискання кнопки Sign In до отримання JWT-токена минає – 0,32 с.

Отримання списку сесій. Після входу інтерфейс автоматично завантажує сесії і виводить лише актуальні записи. Час відповіді – 0,29 с

Надсилання знімка. На натисканні кнопки робиться JPEG-кадр, який додається до FormData і POST-запитом надсилається на бекенд.

Отримання результату. За звичайних умов підтвердження приходить за ≤ 3 с; користувач бачить повідомлення про успішне відмічання. У разі потреби модерації відправляється інша відповідь, а кнопка змінює стан на «Ваша заявка на ручну перевірку відправлена» дивись Рис. 4.2.

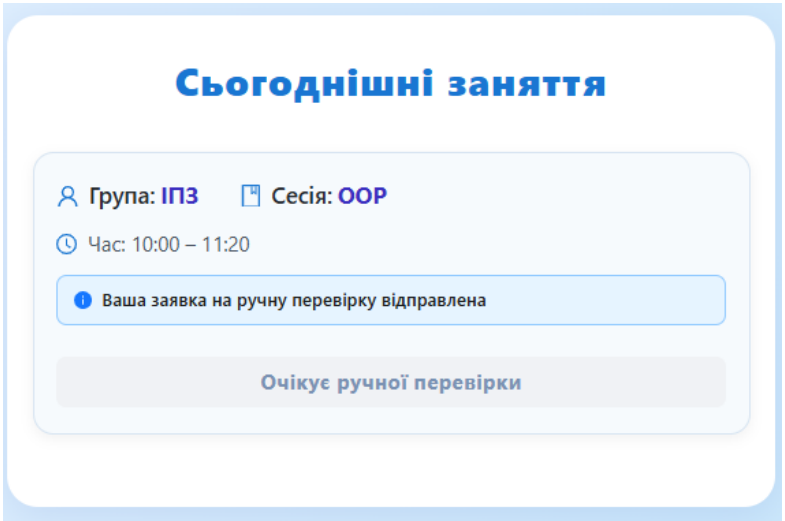

Рисунок 4.2 Інтерфейс у випадку не автоматичного розпізнавання

2) Тестові серії розпізнавання на фронтенді

Щоб оцінити загальну точність роботи системи, було проведено тестування на основі 100 зображень однієї публічної особи, зібраних із відкритих джерел. Зображення охоплювали різні умови — освітлення, ракурси, вирази обличчя, фонове середовище — без акценту на окремі сценарії, а з метою перевірити стійкість загального механізму ідентифікації.

Система була налаштована так, що перше фото використовувалось для створення ембеддингу (векторного представлення особи), після чого кожне наступне зображення оброблялось з метою перевірки — чи співпаде воно з базовим профілем якщо то фото знову додається в базу і настпні фото порівнюються зі всіма екземплярами (детальний опис наведений у додатку Б).

Фрагменти екранів із найтиповішими випадками наведено:

Базове розпізнавання з повідомленням Ви вже відмітились — Рис. 4.3;

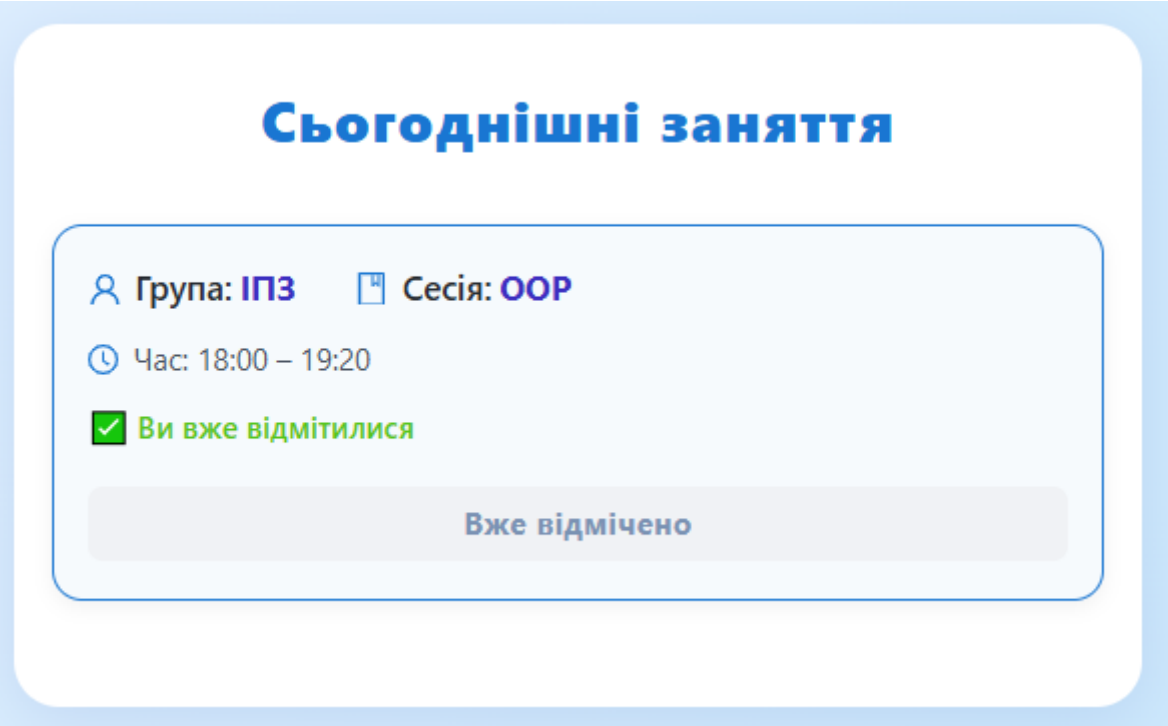

Рисунок 4.3 Інтерфейс у випадку автоматичного розпізнавання

Помилка «Обличчя не знайдено» при занадто темному кадрі — Рис. 4.4;

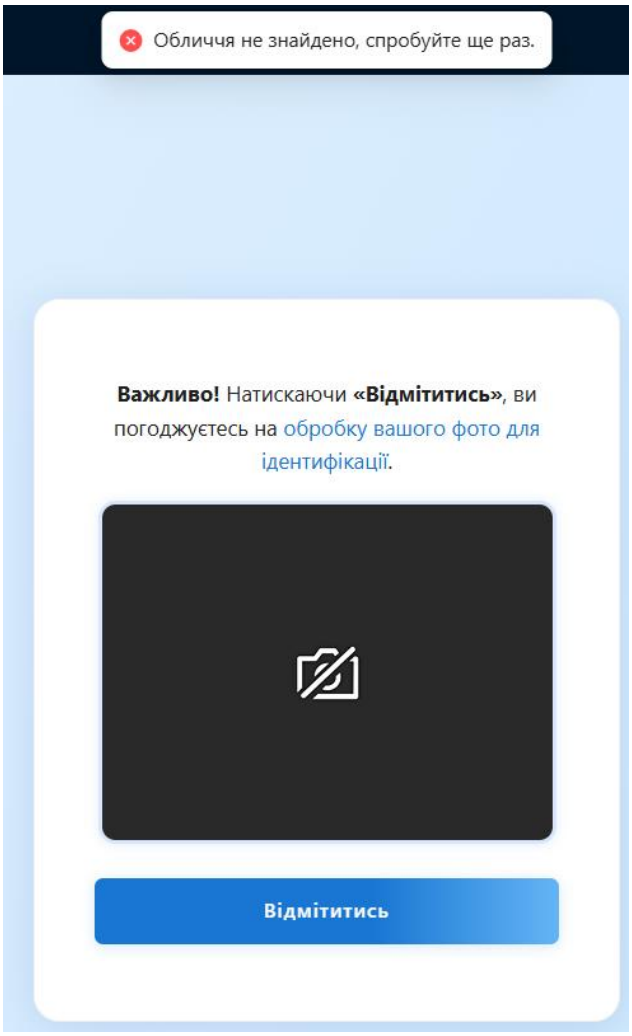

Рисунок 4.4 Інтерфейс у випадку коли не було знайдено лиця на кадрі

3) Результати

Інтерфейс лишається інтуїтивним для нефахівця: одна кнопка — одна дія.

Фронтенд коректно передає зображення й обробляє всі відповіді API, однак чутливий до якості фото.

Більшість збоїв спричинені зовнішніми факторами: погане світло або перекриття, а не помилками коду.

Подальший розвиток: запропоновано реалізувати клієнтську компресію зображення перед відправкою та інтегрувати анімацію-підказку правильного позиціонування обличчя.

Таким чином, функціональне тестування довело, що клієнтська частина здатна стабільно працювати у реальних умовах навчального процесу, а виявлені межі (освітлення, перекриття) уже враховані у плані подальшої оптимізації.

ВИСНОВОК

У процесі розробки програмного продукту було створено рішення для автоматизованого обліку відвідуваності на основі розпізнавання облич, яке охоплює повний цикл взаємодії користувача — від авторизації до фіксації присутності. Система орієнтована на реальні сценарії навчального процесу й враховує потреби як студентів, так і викладачів.

Розроблений функціонал забезпечує простий і зрозумілий інтерфейс, який дозволяє користувачеві з мінімальними зусиллями пройти ідентифікацію. Ключові модулі успішно реалізують основні задачі: реєстрація, вибір сесії, захоплення зображення, обробка й збереження результатів.

Особлива увага під час роботи приділялася обробці нестандартних ситуацій: неякісним фото, відсутності облич, повторному входу — для всіх цих випадків реалізовані відповідні механізми, що забезпечують стабільну роботу системи.

Система пройшла дослідну експлуатацію в умовах, максимально наближених до реального використання. За результатами тестування підтверджено її працездатність, зручність, а також відповідність заявленим вимогам. Незважаючи на виявлені окремі обмеження при пікових навантаженнях, загальна стабільність та коректність роботи системи дозволяє впевнено говорити про її ефективність.

Результати дипломної роботи свідчать про досягнення поставленої мети — створення інструменту, що автоматизує процес фіксації присутності, зменшує навантаження на персонал. Отримане рішення має потенціал для подальшого впровадження та вдосконалення в межах освітніх установ.

Система може бути розширена додатковими модулями, такими як інтеграція з внутрішніми платформами, повідомлення для викладачів та живу ідентифікацію. Це відкриває перспективи її масштабування та адаптації під ширший спектр задач у сфері освіти.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker Documentation [Електронний ресурс]. — https://docs.docker.com/get-started/
2. Nginx Official Documentation. Deploying reverse proxy with Nginx [Електронний ресурс]. — https://nginx.org/en/docs/
3. OpenCV Library. Open Source Computer Vision Library [Електронний ресурс]. — https://opencv.org/
4. ONNX Runtime Documentation. Inference with pre-trained models [Електронний ресурс]. — https://onnxruntime.ai/
5. PostgreSQL Global Development Group. PostgreSQL Documentation [Електронний ресурс]. — https://www.postgresql.org/docs/
6. AWS Lightsail Documentation. Getting started with Amazon Lightsail [Електронний ресурс]. — https://lightsail.aws.amazon.com/
7. Let’s Encrypt. Certbot for HTTPS setup [Електронний ресурс]. — https://certbot.eff.org/
8. Yakhyo Khamidov. Face Re-identification System [Електронний ресурс]. — https://github.com/yakhyo/face-reidentification
9. Yakhyo Khamidov. Facial Analysis Demo & Docs [Електронний ресурс]. — Режим доступу: https://yakhyo.github.io/facial-analysis/
10. Microsoft Docs. ASP.NET Core Documentation [Електронний ресурс]. — https://learn.microsoft.com/en-us/aspnet/core/
11. Microsoft Docs. Entity Framework Core Documentation [Електронний ресурс]. — https://learn.microsoft.com/en-us/ef/core/
12. React Documentation. React: A JavaScript library for building user interfaces — Режим доступу: https://react.dev/
13. Vite Documentation. Next Generation Frontend Tooling [Електронний ресурс]. — Режим доступу: https://vitejs.dev/
14. Redux Toolkit Documentation [Електронний ресурс]. — https://redux-toolkit.js.org/

ДОДАТКИ

Додаток А. Репозиторій з програмним кодом

Додаток Б. Тестування розпізнавання облич

Додаток А

Розміщення програмного забезпечення на репозиторії

https://github.com/Mayba04/FaceRecognitionAPI – Backend

https://github.com/Mayba04/FaceTrack – Frontend

Додаток Б

Тестування модуля розпізнавання обличчя

Для тестування системи ідентифікації особи було обрано публічну особу — Леся Нікітюк, українська телеведуча. Було зібрано 100 фотографій із відкритого доступу — переважно з її Instagram-сторінки, з акцентом на різноманітність умов (ракурси, освітлення, міміка, фон, одяг). Знімки охоплювали як портрети, так і фото в повний зріст, з різною якістю та рівнем деталізації.

Перше фото використовувалося для ініціалізації векторного представлення (ембеддингу) особи. Усі наступні зображення проходили через модуль розпізнавання для перевірки, чи співпадають із вже збереженим профілем. У випадку невпізнавання — створювався новий вектор. Таким чином перевірялась не лише точність першого зіставлення, а й стійкість системи до змін у зовнішності чи контексті.

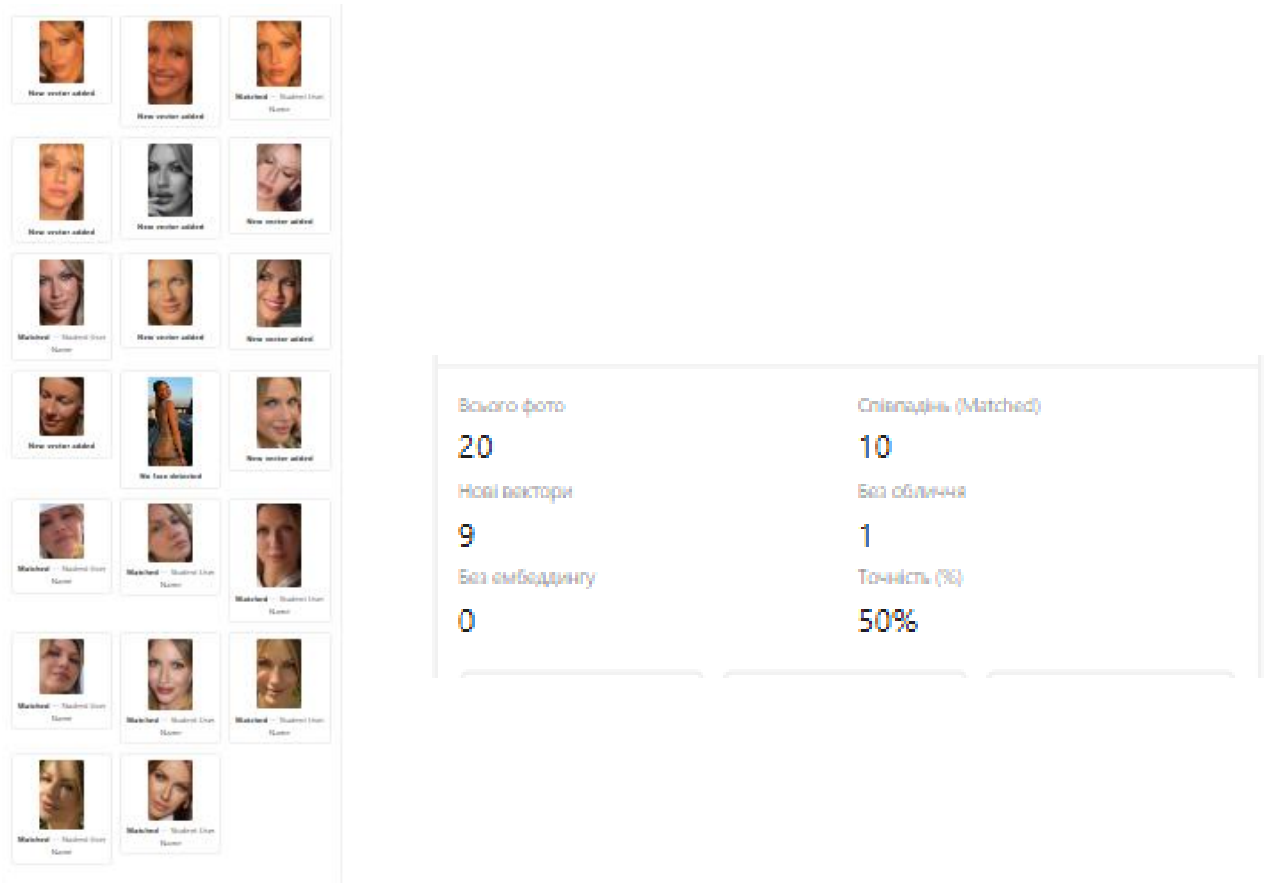

Рисунок Б.1 – Приклади кадрів серії 1

Фото № 01–10 У першій серії система розпізнала лише 10 з 20 фото, тобто точність склала 50 %. Один кадр узагалі не був детектований (не знайдено обличчя), ще 9 — класифіковані як нові особи дивись (Рис. Б.1)

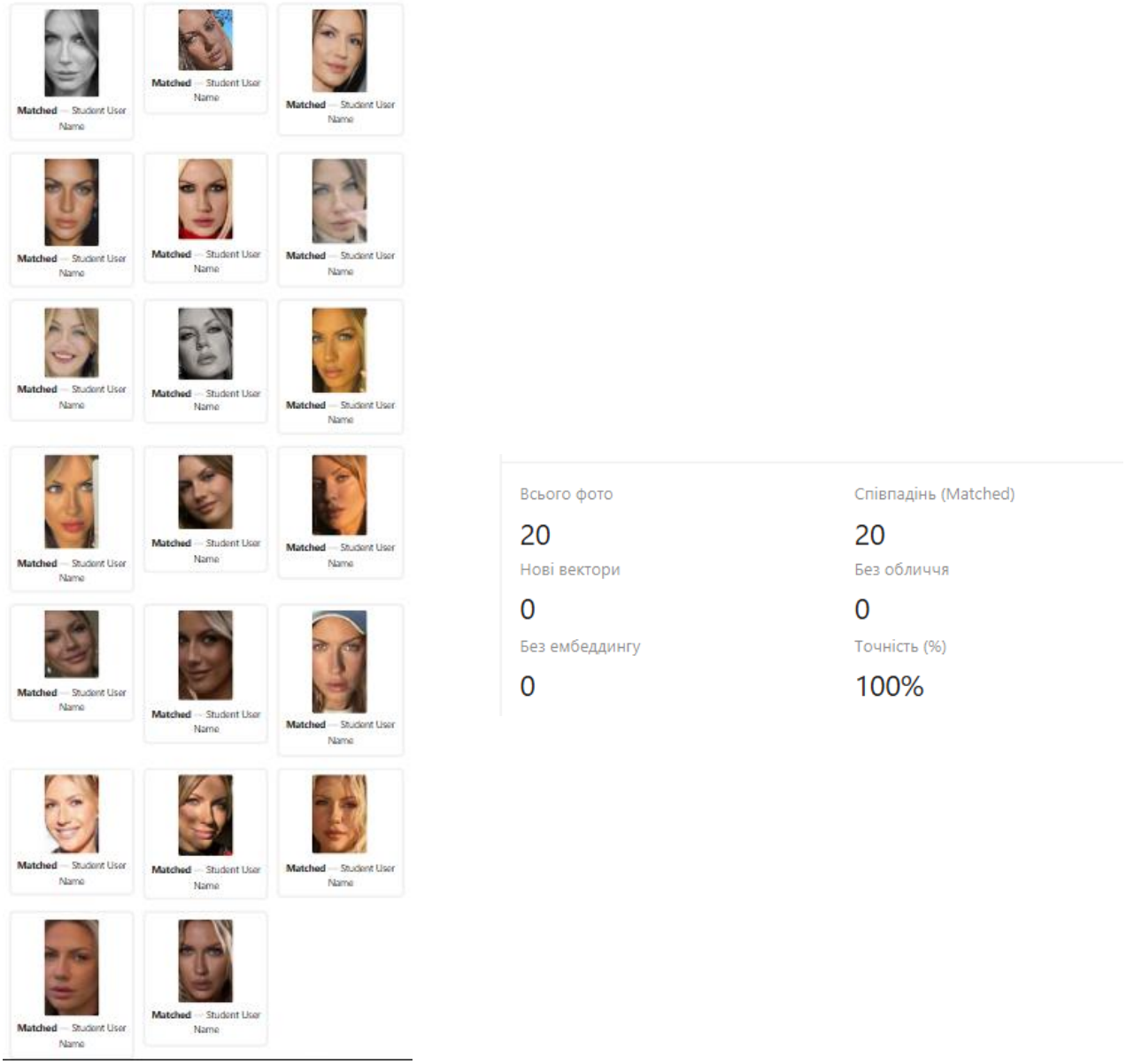

Рисунок Б.2 – Кадри серії 2

Фото № 21–40 На другому етапі усі 20 фотографій були правильно ідентифіковані без створення нових ембеддингів. Система продемонструвала повну відповідність нових зображень збереженому профілю дивись (Рис. Б.2).

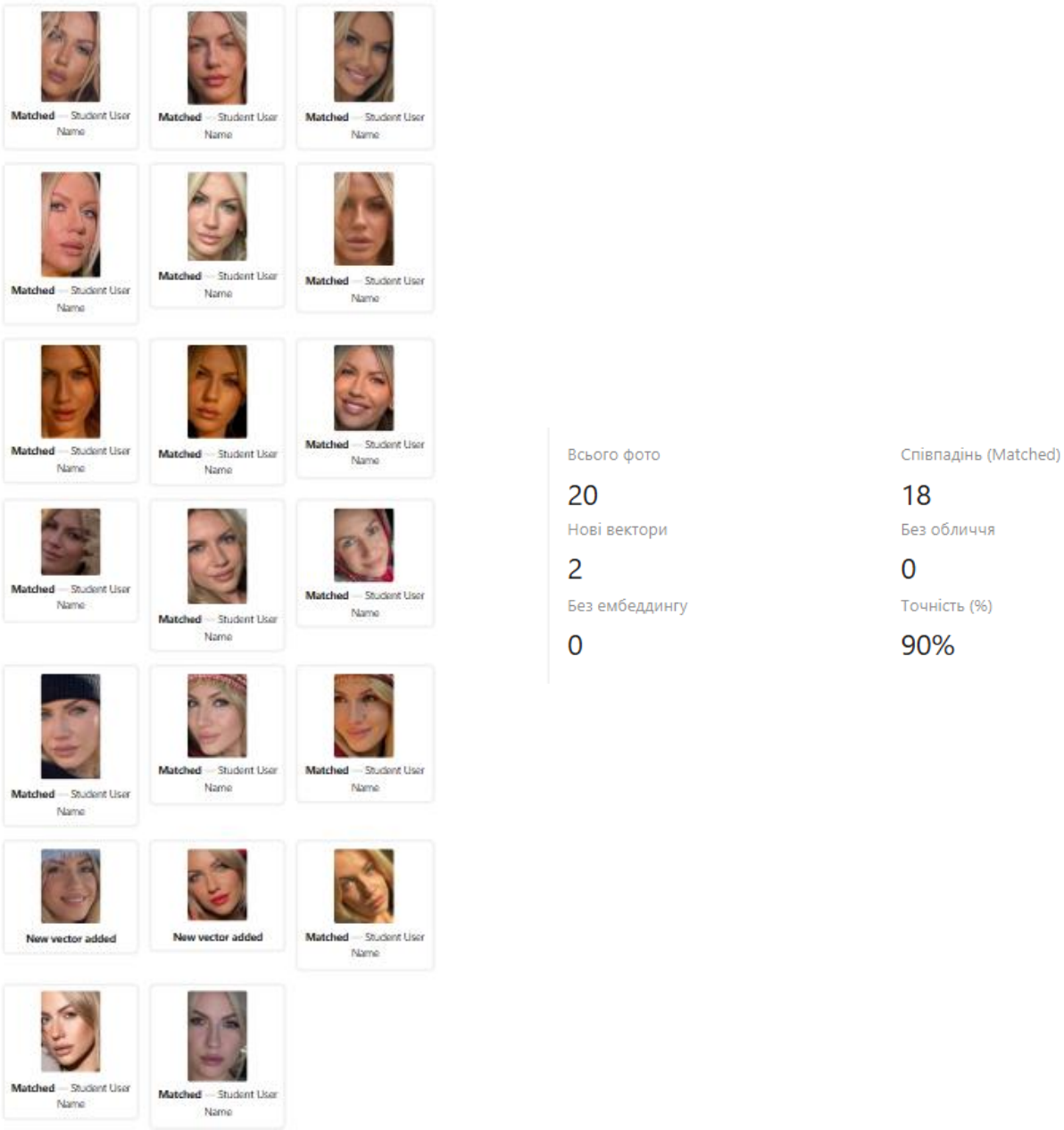

Рисунок Б.3 – Кадри серії 3

Фото № 41–60 У третій серії система показала високу точність — 18 з 20 кадрів успішно співпали, лише 2 були класифіковані як нові дивись (Рис. Б.3).

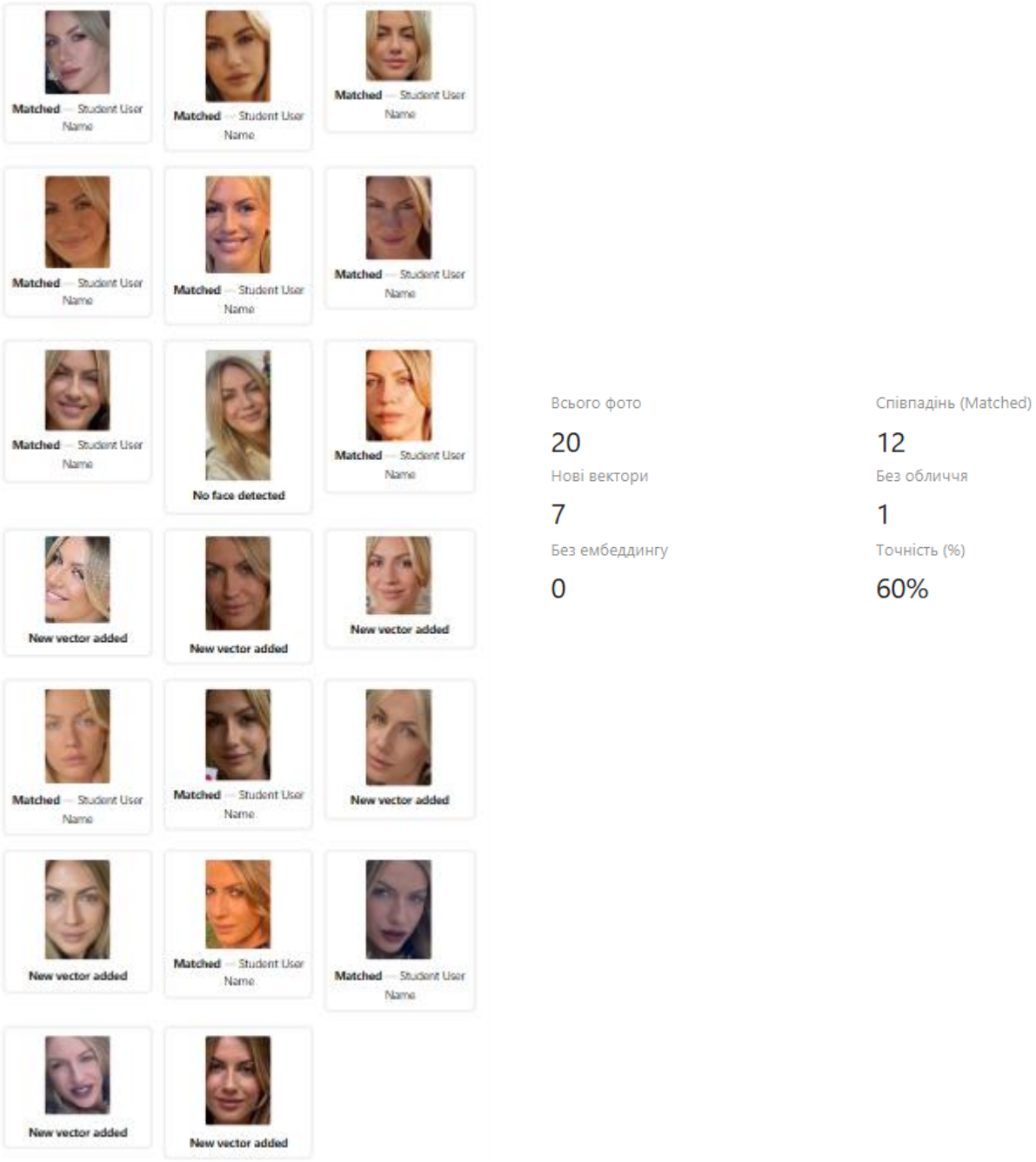

Рисунок Б.4 – Кадри серії 4

Фото № 61–80 У четвертому наборі точність суттєво знизилася: із 20 фото лише 12 були співставлені з існуючим профілем, 7 визнані новими, 1 не пройшов етап детекції дивись (Рис. Б.4).

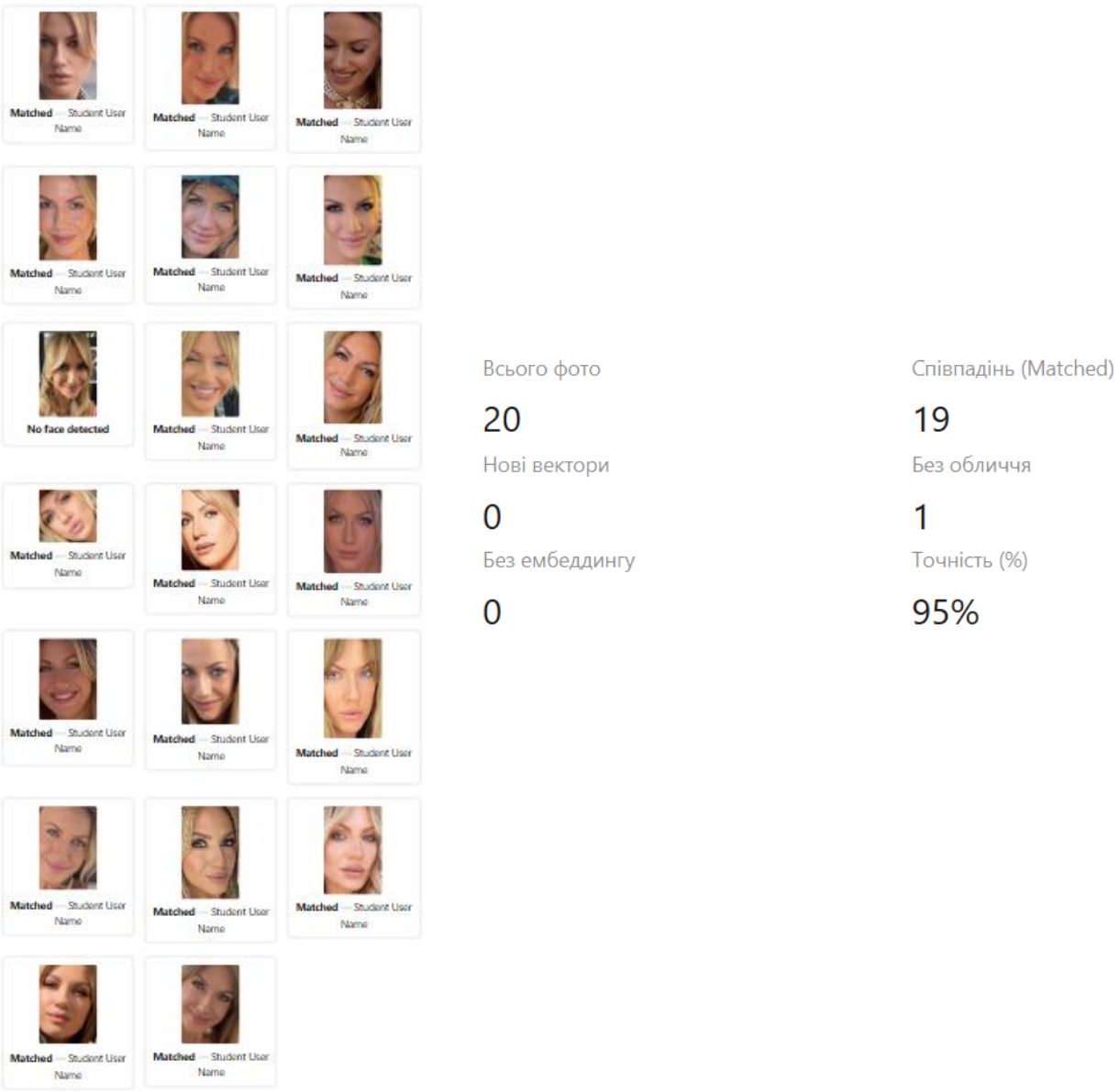

Рисунок Б.5 – Кадри серії 5

Фото № 81–100 П’ята серія показала 95 % точності — 19 зіставлень успішні, одне фото не пройшло детекцію дивись (Рис. Б.5)

Зведена таблиця результатів

Таблиця 1 Результати тестування модуля розпізнавання обличчя

Серія	Кількість фото	Співпадінь	Нові вектори	Без обличчя	Точність
1	20	10	9	1	50 %
2	20	20	0	0	100 %
3	20	18	2	0	90 %
4	20	12	7	1	60 %
5	20	19	0	1	95 %
Загалом	**100**	**79**	**18**	3	**79 %**

Загальна точність модуля розпізнавання облич склала 79 %. Найкращі результати отримано на фото з чіткими рисами та природним освітленням, тоді як погане освітлення, сильна міміка або зміна ракурсу можуть спричинити хибну класифікацію. Незважаючи на це, модель показала стабільну роботу в більшості умов, а автоматичне додавання ембеддингів поступово підвищує точність на наступних етапах.