МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

СТРУТОВСЬКИЙ МАКСИМ ІГОРОВИЧ

Допускається до захисту:
завідувач кафедри
інформаційних технологій,
д.т.н., доцент
_____Т.В. Нескородєва
«_____» _____ 20___ р.

ДОСЛІДЖЕННЯ КЛІЄНТ-СЕРВЕРНИХ СИСТЕМ ДЕТЕКТУВАННЯ МЕДИЧНИХ МАСОК НА ЛЮДЯХ ЗА ДОПОМОГОЮ ШТУЧНИХ НЕЙРОННИХ МЕРЕЖ

Спеціальність 122 Комп'ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Федоров Є.Є., професор кафедри
інформаційних технологій,
д.т.н., професор

Оцінка: ___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2022

АНОТАЦІЯ

Струтовський М.І. Дослідження клієнт-серверних систем детектування медичних масок на людях за допомогою штучних нейронних мереж. Спеціальність 122 «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2022.

У кваліфікаційній (бакалаврській) роботі досліджено методи детектування медичних масок на людях та використання клієнт-серверної архітектури для застосунку. Встановлено, що для великої точності моделі розпізнавання потрібні потужні ресурси для обчислень. Показана реалізація сервісів, що поєднуються в єдиний застосунок для розпізнавання медичних масок.

Ключові слова: клієнт-серверна архітектура, нейронна мережа, виявлення, REST-API, Python, Deepstream SDK.

55 с., 29 рис., 42 джерел.

ABSTRACT

Strutovskyi M. Medical masks detection research of client-server systems using neural networks. Specialty 122 "Computer science", Vasyl' Stus Donetsk National University, Vinnytsia, 2022.

In the qualification (bachelor's) work have been researched detecting methods of medical masks on people and been developed application based on client-server architecture. Have been found, that for high model accuracy of the recognition masks is required powerful resources for calculations. Have been shown the implementation of services that are combined into a single application for the recognition of medical masks.

Keywords: client-server architecture, neural network, detection, REST-API, Python, Deepstream SDK.

55 pages, 29 drawings, 42 sources.

ЗМІСТ

ВСТУП

Пандемія коронавірусної хвороби спричинила багато економічних збитків різним країнам світу, та на жаль, забрала багато життів. І хоча вакцини проти цього вірусу вже винайдені та активно використовуються людством, це не гарантує стовідсоткову захищеність. Аби зупинити поширення коронавірусу, багато країн на законодавчому рівні ввели обмеження на кількість осіб, які можуть перебувати в закладах, обов'язкове носіння медичних масок тощо. І хоча, безпека та здоров'я завжди на першому місці, але від адміністративної відповідальності за порушення масочного режиму ніхто не звільняється.

В епоху інформаційних технологій та штучного інтелекту є доцільним створити сервіс, який буде допомагати автоматично відслідковувати порушників масочного режиму та застосовувати превентивні дії, що полегшить роботу правоохоронних органів та забезпечить більшу захищеність людей.

Об'єктом дослідження є процес детекції медичних масок на людях.

Предметом дослідження є нейромережеві методи детекції медичних масок на людях.

Мета роботи: дослідити нейромережеві методи детекції медичних масок на людях, що інтегровані в клієнт-серверну архітектуру.

Для досягнення цієї мети, поставлені і вирішені наступні завдання:

- провести аналіз існуючих систем детекції медичних масок на людях;
- провести аналіз існуючих нейромережевих методів детекції медичних масок на людях;
- провести аналіз існуючих фреймворків для реалізації нейромережевих методів детекції медичних масок на людях;
- провести аналіз клієнт-серверної архітектури, в яку інтегровані методи детекції медичних масок на людях.

Практична значимість роботи полягає в реалізації сервісу, який буде автоматично рахувати кількість людей в приміщенні, та застосовувати

превентивні дії до порушників карантинних заходів, такі як попередження та штраф.

Структура кваліфікаційної (бакалаврської) роботи складається з двох розділів – дослідження та реалізації задачі. Перший розділ поділяється на 7 підрозділів та використовує 14 рисунків. Другий розділ поділяється на 7 підрозділів та використовує 15 рисунків.

РОЗДІЛ 1

ДОСЛІДЖЕННЯ МЕТОДІВ ДЕТЕКЦІЇ

1.1. Детекція об’єктів

Детекція (виявлення) об’єктів – це техніка комп’ютерного зору для визначення місця розташування об’єктів на зображеннях або відео. За допомогою штучного інтелекту та машинного навчання можна розпізнати будь-які об’єкти – автомобілі, людей, тварин тощо. Дана технологія застосовується для різних рішень задач: розпізнавання машин, дорожніх знаків для систем автопілоту автомобілей (рис. 1.1); розпізнавання обличчя людини для розблокування телефону (FaceID).

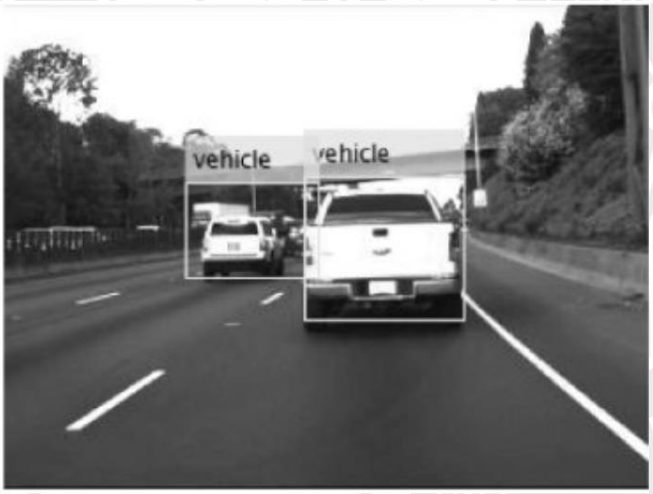

Рисунок 1.1 – Розпізнавання автомобілей

Для виявлення об’єктів використовуються різноманітні методи, серед яких згорткові нейронні мережі (CNN), такі як R-CNN і YOLOv5, є найпопулярніші. Вони здатні автоматично навчатись виявляти об’єкти в зображеннях.

Згорткові нейронні мережі – це клас глибинних штучних нейронних мереж прямого поширення, який успішно застосовується для аналізу візуальних зображень.

Варто зазначити, що для визначення найкращого підходу для виявлення об’єктів потрібно проаналізувати задачі програми та проблеми, які потрібно вирішити. Вибираючи між машинним навчанням та глибинним навчанням (deep-learning), слід пам’ятати про те, що глибинне навчання займе більше часу та ресурсів, але і буде більш ефективне за машинне навчання.

Виявлення об'єктів широко використовується в області спостереження, безпеки, криміналістики, автоматизованих систем транспортних засобів. У зв’язку з цим типом чутливих випадків, використовуючи детекцію об’єктів, надзвичайно важливо, щоб детектори працювали швидко і водночас забезпечували дуже хорошу точність. Існують деякі проблеми, під час реалізації детекції:

- Виявлення в різних масштабах . Однією з найпоширеніших проблем, з якою стикаються, є об’єкт, який виявляється в одному масштабі, може бути виявлений або не виявлений в іншому, меншому чи більшому масштабі. Для екстрактора ознак стає важливим узагальнити ознаки, які можна використовувати для будь-якого масштабу. Для цього використовуються пірамідальні мережі функцій, які допомагають витягувати функції в будь-якому масштабі (малому, середньому та великому). Цей тип екстракторів особливостей широко використовується в більшості детекторів об’єктів.

- Навчання для різних розмірів зображень. Ще один момент для загального виявлення об’єктів – це тренування на кожному зображенні вхідного розміру. Більшість регресорів та класифікаторів є обмежені у вхідних розмірах зображення. Тому змінити розмір під час виконання неможливо. Мережа, навчена на одній роздільній здатності, може не дати хороших результатів на іншій. І хоча повністю згорткові мережі є частковим рішенням цієї проблеми, обмеження на вхідний розмір зображень існує завжди.

- Швидкість/точність . Однією з найбільших проблем, з якою стикаються багато людей у промисловості, є фактор швидкості. Розгортання детекторів важких об’єктів на дешевому вбудованому пристрої викликає серйозне занепокоєння, яке не може збалансувати швидкість, так і точність. Отже, потрібно обирати між швидкістю і точністю роботи, а також лімітом на ресурси, що використовуються.

- Дисбаланс класів. Дисбаланс класів робить мережу упередженою до вивчення додаткової інформації та це суттєво впливає на точність. Щоб подолати цю проблему, деякі потрібно виконати дублювання зображень тих класів, яких недостатньо для використання, щоб створити рівне співвідношення позитивних (об’єктів) і негативних (фонових) вибірок.

Основною проблемою для реалізації задачі розпізнавання масок на обличчях, можна вважати швидкість та точність виконання.

1.2. Детекція облич з масками

Детекція (виявлення) обличчя – це комп’ютерна технологія, що визначає розташування і розміри людських облич на довільних (цифрових) зображеннях. Вона виявляє риси обличчя й ігнорує все інше, таке як будівлі, дерева та інші частини тіла. Багато алгоритмів виконують завдання виявлення облич як задачу бінарної класифікації. Тобто, з певної частини зображення виділяються характерні риси на основі яких навчений на тестових зображеннях класифікатор приймає рішення про те чи дана частина є обличчям, чи ні.

Насправді, існують багато методів розпізнавання обличчя. Першим методом є розпізнавання на основі знань. Цей метод спирається на набір правил, розроблених людьми відповідно до наших знань. Обличчя повинно мати ніс, очі та рот на певній відстані та положенні один від одного, ці знаня можна використати для навчання моделі. Проблема цього методу полягає в

тому, що потрібно створити відповідний набір правил. Якщо правила занадто загальні або надто детальні, система отримує багато помилкових спрацьовувань. Однак він працює не для всіх кольорів шкіри і залежить від умов освітлення, які можуть змінити точний відтінок шкіри людини на зображені.

Другий метод – метод відповідності шаблону. Метод відповідності шаблону використовує попередньо визначені або параметризовані шаблони облич, щоб знайти або виявити грані за кореляцією між попередньо визначеними або деформованими шаблонами та вхідними зображеннями. Модель обличчя може бути побудована за ребрами за допомогою методу виявлення країв. Різновидом цього підходу є контрольована фонова техніка. Якщо існує зображення обличчя у фронтальному положенні та звичайний однокольоровий фон, можна видалити фон, залишивши межі обличчя. Для цього підходу програмне забезпечення має кілька класифікаторів для виявлення різних типів облич, а також деякі для профільних облич, таких як детектори очей, носа, рота, а в деяких випадках навіть всього тіла. Хоча цей підхід простий у застосуванні, він зазвичай недостатній для виявлення обличчя.

Третій метод є розпізнавання обличчя за ознаками. Метод на основі ознак виділяє структурні особливості обличчя. Він навчається як класифікатор, а потім використовується для диференціації лицьових та нелицьових областей. Одним із прикладів цього методу є розпізнавання обличчя на основі кольору, яке сканує кольорові зображення або відео на предмет ділянок із типовим кольором шкіри, а потім шукає сегменти обличчя.

Оскільки задання роботи полягає у виявленні масок на обличчях, то можемо сказати, що тренування та використання моделі буде ідентичне до тренування моделей детекції облич, лише з відмінністю у датасеті. На рисунку 1.2 зображено приклад детекції маски на обличчі.

Рисунок 1.2 – Приклад розпізнавання маски на обличчі

Датасет – це набір даних, який обробляється комп'ютером як єдиний блок та може вміщувати в собі зображення, відео, числові дані, аудіозаписи тощо. Не варто недооцінювати роль датасету в тренуванні моделі, досить важливо є правильно обрати набір даних із правильною кількістю елементів та розмірами зображення.

Проаналізувавши усі можливі безкоштовні датасети, з відкритою ліцензією, автор виділив чотири набори даних:

1. Kaggle Medical Mask Dataset
2. MAFA – Masked Faces
3. FDDB Dataset
4. WiderFace Dataset

Перші два датасети містять в собі зображення типу JPG із масками на обличчях (рис. 1.3). Два останніх – обличчя без масок. Тобто знайшовши зображення облич з наявністю медичних масок та зображення облич без масок, нейронна мережа буде вчитись розрізняти два класи – «з маскою» та «без маски», тобто бінарна класифіцікація, як зазначено вище.

Рисунок 1.3 – Картинка із датасету MAFA

Також варто зазначити, що дані датасети можна завантажити з готовими мітками, якими позначають координати обличчя на зображені, щоб полегшити нейронній мережі знаходження розташування лиця.

Пропорція зображень з масками та без має бути однаковою, щоб уникнути перетренування моделі. Також потрібно знайти найкращу пропорцію для розбиття зображень на тренувальні та тестові. На тренувальних зображеннях нейронна мережа буде навчатись виявляти обличчя з масками та без масок, а на тестових зображеннях – «перевіряти» себе, тим самим покращуючи точність.

1.3. Вибір нейронної мережі

Як було зазначено в попередніх підрозділах, для детекції об’єктів використовуються згорткові нейронні мережі (CNN). Аби не конфігурувати нейронну мережу кожен раз, можна використати готові «хребти» нейронних мереж, тобто мережі, які вже мають налаштовану кількість шарів та інших налаштувань.

Серед популярних хребтів нейронних мереж можна виділити такі:

1. YOLOv5
2. Resnet18

3. DetectNet_v2

Ultralytics YOLOv5 – один із найвідоміших алгоритмів виявлення об’єктів завдяки своїй швидкості та точності розпізнавання. Існує п’ять версії YOLO, але остання найоптимальніша для використання та підходить для роботи з фреймворком PyTorch. Випуск YOLOv5 включає чотири різних моделей: YOLOv5s (найменша), YOLOv5m, YOLOv5l, YOLOv5x (найбільша). Вони відрізняються за кількістю шарів в нейронній мережі та вагою моделей. YOLOv5l та YOLOv5x є найточніші у розпізнанні, але повільні. Натомість YOLOv5m та YOLOv5s швидкі, завдяки пониженню точності розпізнання (рис. 1.4).

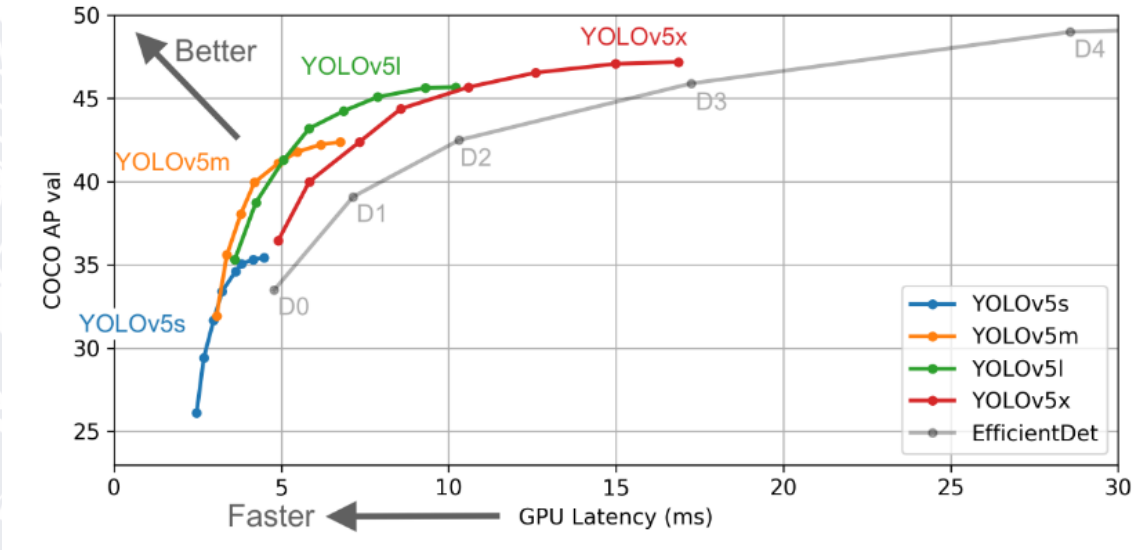

Рисунок 1.4 – Порівняння випусків YOLOv5

Resnet18 – алгоритм виявлення об’єктів, який містить в собі 18 шарів. Мережа має розмір вхідного зображення 256 на 256 пікселей. Попередньо навчена мережа може класифікувати зображення на 1000 категорій об’єктів. Дана нейронна мережа зайняла перше місце за детекцією об’єктів та сегментацією зображень в датасеті COCO. Структурна схема моделі resnet18 зображена на рисунку 1.5.

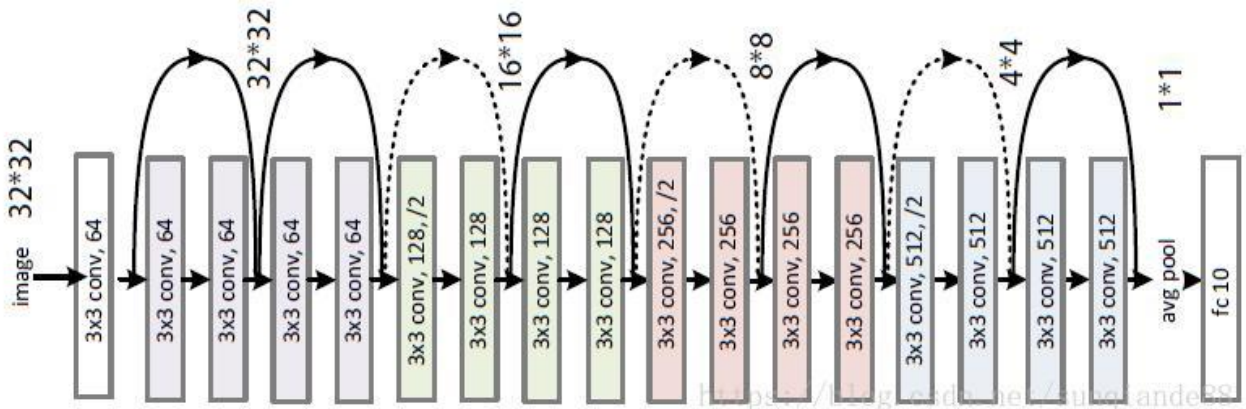

Рисунок 1.5 – Структура resnet18

DetectNet_v2 – нейронна мережа, яка використовується разом з Transfer Learning Toolkit від компанії Nvidia. Вона може використовувати хребти Resnet18, Darknet19 та інші. Практики показують, що найкращим способом використання даної мережі буде використання Nvidia Deepstream SDK. Для цього потрібно спочатку навчити модель через Transfer Learning Toolkit, а згодом експортувати в зашифрований формат (encrypted format .etlt). На рисунку 1.6 зображено процес від тренування моделі, до експортування в формат, необхідний для фреймворку Nvidia Deepstream SDK.

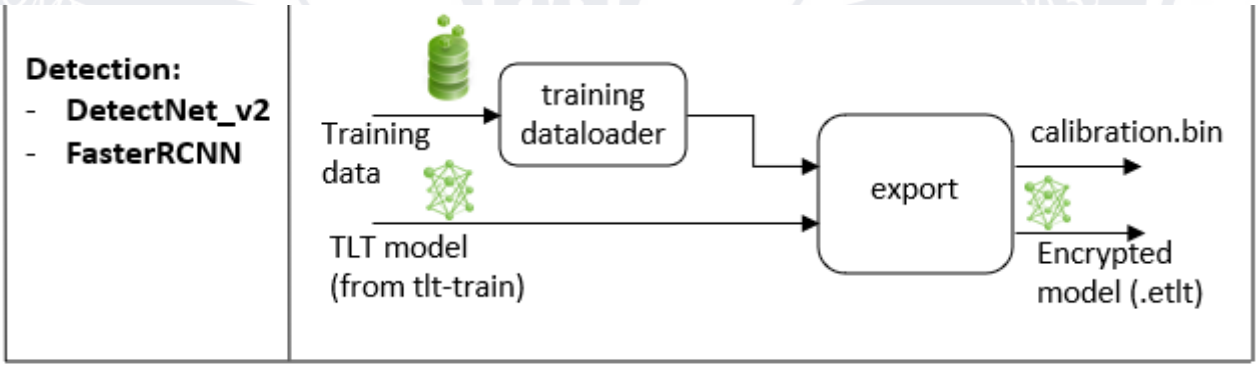

Рисунок 1.6 – Процес експорту моделі detectnet_v2 в encrypted формат

1.4. Nvidia Deepstream SDK

Оскільки завдання роботи полягає в реалізації сервісу, детекції масок на обличчях, доцільно було б використовувати фреймворк, який є найбільш швидким. Швидкість розпізнавання нейронної мережі разом зі швидкістю

передачі інформації з кадрів відео є найголовнішим в побудові сервісу. Використання Nvidia Deepstream SDK дасть найкращий результат для даного сервісу.

Nvidia Deepstream SDK – набір інструментів потокової аналітики для створення додатків на базі штучного інтелекту від компанії Nvidia. Deepstream організований за принципом блоків-плагінів з апаратним прискоренням GPU. DeepStream пропонує виняткову пропускну здатність для широкого спектру моделей штучного інтелекту, на основі виявлення об'єктів, класифікації зображень та сегментації екземплярів. Щоб зменшити зусилля на розробку та збільшити пропускну здатність, розробники можуть використовувати високоточні попередньо навчені моделі з TAO Toolkit та розгортати їх за допомогою DeepStream. Підтримуються різні декодери, фільтри відео та інші складні елементи вхідних даних. На рисунку 1.7 зображений простий пайплайн використання моделі розпізнавання (плагін gst-nvinfer) та різних допоміжних плагінів для ренедеру, декодингу, конвертування зображень з відеоряду.

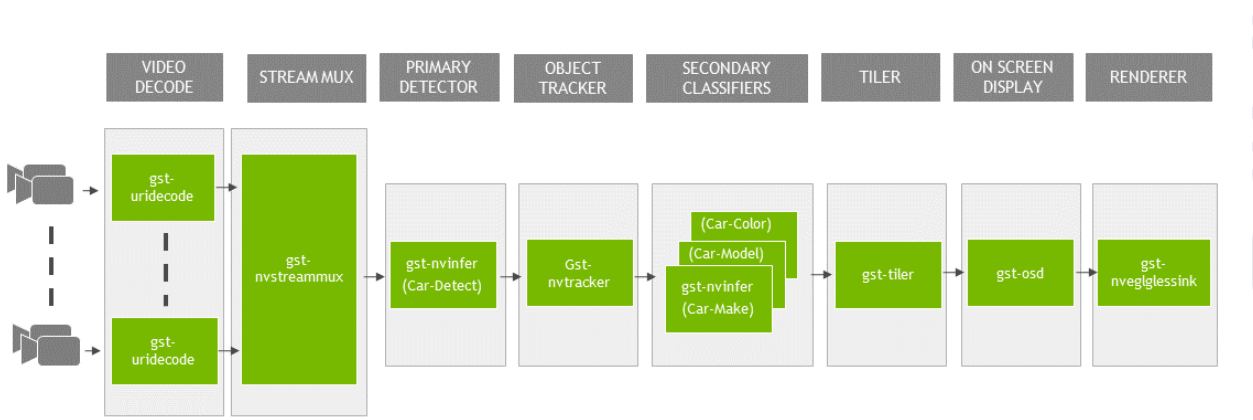

Рисунок 1.7 – Приклад пайплайну Deepstream.

Використання Nvidia Deepstream SDK пропонує гнучкість від швидкого створення прототипів до рішень повного рівня виробництва. Завдяки інтеграції з NVIDIA Triton Inference Server можна розгортати моделі у рідних платформах, таких як PyTorch і TensorFlow для детекції та інференсу. Використовуючи NVIDIA TensorRT для високої пропускної здатності з опціями підтримки

кількох графічних процесорів, кількох потоків і пакетної підтримки, можна досягти максимально можливої продуктивності.

Для реального розгортання додатків/сервісів IVA віддалене керування та контроль додатків є критичними. DeepStream SDK може працювати в будь-якому хмарному середовищі. Він відповідає вимогам Інтернету речей, таким як ефективний двонаправлений обмін повідомленнями між периферією та хмарою, безпека, інтелектуальний запис і оновлення моделі штучного інтелекту в режимі реального часу.

Завдяки двосторонньому обміну повідомленнями між периферією та хмарою можна додати більший контроль для випадків використання, таких як віддалені триґери для запису подій, зміна робочих параметрів та конфігурації програми або запити системних журналів.

Функція розумного запису в додатку DeepStream дозволяє заощадити цінний дисковий простір за допомогою вибіркового запису, що забезпечує швидший пошук.

Безперервне оновлення по повітрю (OTA) для всієї програми або окремих моделей штучного інтелекту з будь-якого хмарного реєстру, щоб постійно підвищувати точність із нульовим простоєм.

Для безпечного зв’язку з пристроями IoT DeepStream забезпечує двосторонню аутентифікацію TLS на основі сертифікатів SSL та зашифроване спілкування на основі аутентифікації з відкритим ключем.

DeepStream пропонує інтерфейс інтеграції IoT з Redis, Kafka, MQTT і AMQP , а також інтеграцію під ключ з AWS IoT і Microsoft Azure IoT.

Також є можливість створювати високопродуктивні хмарні додатки DeepStream за допомогою контейнерів NVIDIA NGC. Використовуючи DeepStream, можна масштабно розгортати та керувати контейнерними додатками за допомогою Kubernetes і Helm Charts.

Апаратне прискорення працює тільки на відеокартах від компанії Nvidia. Deepstream підтримує розробку додатків на таких мовах програмування, як C, C++, Python і використовує платформу GStreamer.

Gstreamer – платформа для побудови мультимедійних додатків, який використовує блоки-плагіни, які поєднуються між собою і створюють один єдиний пайплайн. GStreamer підтримує широкий спектр компонентів обробки медіа, включаючи просте відтворення аудіо, відтворення відео, запис, потокове передавання та редагування. Конструкція пайплайну служить основою для створення багатьох типів мультимедійних додатків, таких як відеоредактори, транскодери, потокові медіа-мовники та медіаплеєри.

За допомогою цього фреймворку можна створювати додатки різної складності, наприклад аудіо чи відео програвачі (рис. 1.8).

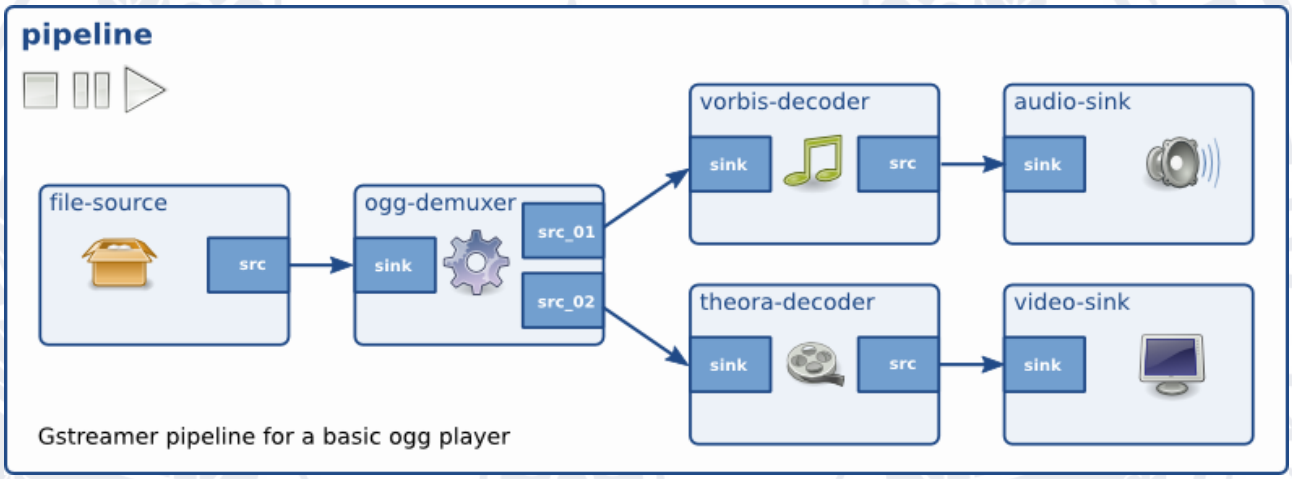

Рисунок 1.8 – Приклад Gstreamer пайплайну для аудіопрогравача

Deepstream також використовує такі технології, як CUDA та TensorRT. CUDA – це програмно-апаратна архітектура паралельних обчислень, яка дозволяє істотно збільшити обчислювальну продуктивність, завдяки використанню графічних процесорів фірми Nvidia. CUDA SDK надає можливість включати в текст програм на C виклик підпрограм, що виконуються на графічних процесорах Nvidia.

CUDA має кілька переваг перед традиційними обчисленнями загального призначення на GPU з використанням графічних API:

- розсіяне читання – код може читати з довільних адрес у пам’яті;
- уніфікована віртуальна пам'ять (CUDA 4.0 і вище);

- уніфікована пам'ять (CUDA 6.0 і вище);
- спільна пам’ять – CUDA надає швидку область спільної пам’яті, яку можна спільно використовувати між потоками. Це можна використовувати як кеш, керований користувачем, що забезпечує більшу пропускну здатність, ніж це можливо;
- швидше завантаження та зчитування на GPU та з нього;
- повна підтримка цілочисельних і порозрядних операцій, включаючи пошук цілих текстур;
- на картах серії RTX 20 і 30 ядра CUDA використовуються для функції під назвою «RTX IO», де ядра CUDA різко скорочують час завантаження відеоконтенту.

Обчислення відбуваються за допомогою додаткової бібліотеки для C, яка суттєво змінює процес виконання програми (рис. 1.9).

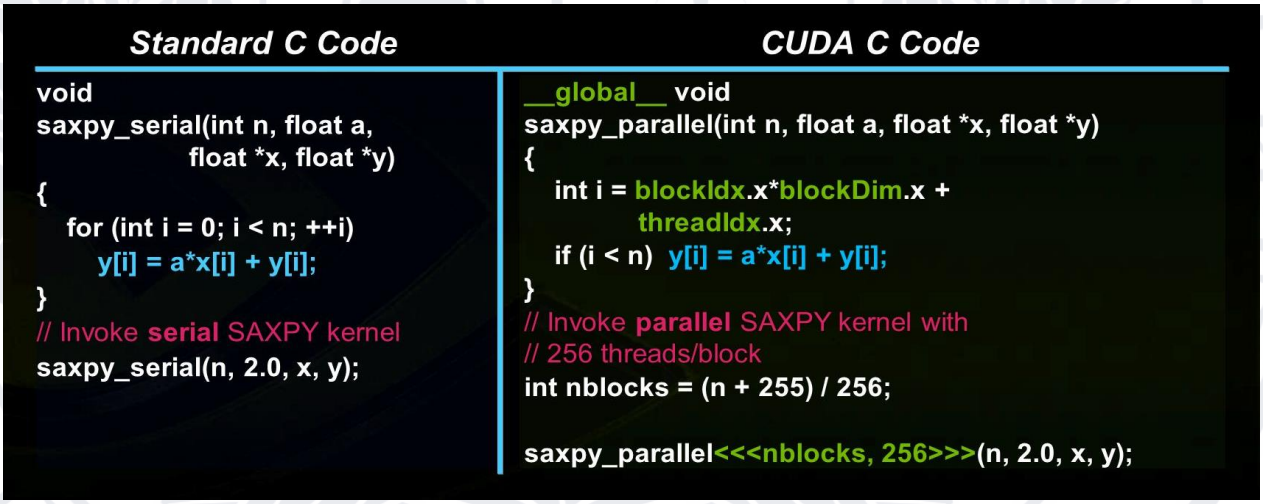

Рисунок 1.9 – Різниця між звичайним C кодом та кодом за допомогою CUDA

Пакет TensorRT дозволяє оптимізувати виконання та навчання моделей, що збільшує швидкість у 36 разів, в порівнянні з платформами, що працюють тільки на процесорі. Даний пакет створений за допомогою CUDA SDK та дозволяє навчати моделі з квантуванням INT8, INT16 тощо. З особливостей TensorRT можна виділити кращу точність виконання, використання злиття

шарів та тензорів, автоналаштування ядра відеокарти, динамічну тензорну пам’ять, паралельність виконання (завдяки CUDA).

TensorRT інтегровано з PyTorch і TensorFlow, тому легко можна досягти в 6 разів швидше виконання детекції за допомогою 1 рядка коду. Якщо проводити навчання моделі на власному або спеціальному фреймворку, відмінному від двох вищенаписаних, потрібно використовувати TensorRT C++ API, щоб імпортувати та прискорити моделі.

Для використання моделей, Deepstream використовує плагін Gst-nvinfer. Він приймає пакети NV12/RGBA буферу з потоку кадрів. Даний плагін можна конфігурувати, вказавши, наприклад, шлях до моделі, яка буде використовуватись, а також швидкість виконання програми, яка задається у кількості батчей за один раз (кількість розпізнавання кадрів за одну ітерацію) і залежить від потужності відеокарти та відеопам’яті. Вхідні дані подаються на плагін gst-nvinfer, але за виконання коду відповідає низькорівневе API nvds_infer, яке встановлюється разом із Deepstream. Дане API тісно співпрацює з CUDA та оптимізує виконання детекції на рівні відеокарти (рис. 1.10).

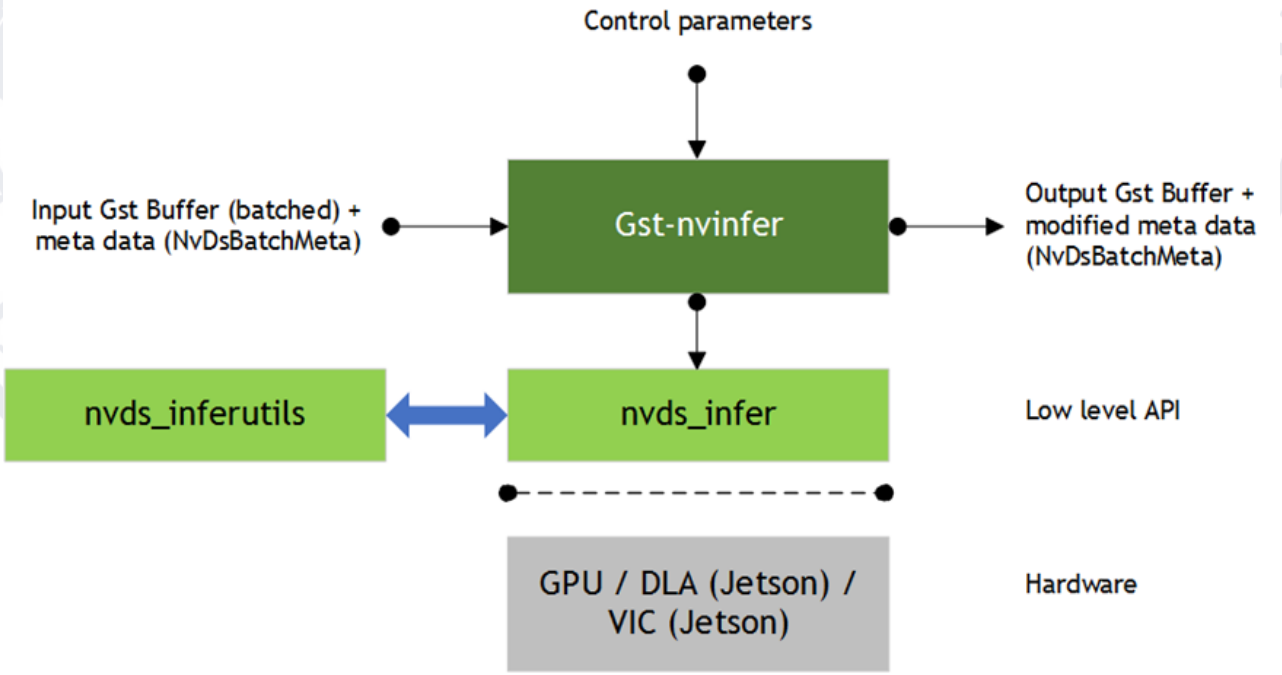

Рисунок 1.10 – Архітектура плагіну nvinfer

Плагін Gst-nvinfer може мати вихідні дані тензора, згенеровані механізмом виведення TensorRT, як метадані. Він додається як NvDsInferTensorMeta в список NvDsFrameMeta для основного (повнокадрового) режиму або в список NvDsObjectMeta для вторинного (об'єктного) режиму. За допомогою цих метаданих можна отримувати дані про розпізнанні об'єкти та їх кількість.

Плагін Gst-nvinfer може працювати в трьох режимах:

- Основний режим: працює на повних кадрах
- Вторинний режим: працює з об'єктами, доданими в мета компонентами вище потоку
- Режим попередньо обробленого тензорного введення: працює з тензорами, приєднаними до компонентів, що знаходяться вгорі

При роботі в режимі введення попередньо обробленого тензора попередня обробка всередині Gst-nvinfer повністю пропускається. Плагін шукає GstNvDsPreProcessBatchMeta, приєднаний до вхідного буфера, і передає тензор, як функцію виведення TensorRT без будь-яких змін. Наразі цей режим підтримує обробку на повному кадрі. GstNvDsPreProcessBatchMeta додається за допомогою плагіна Gst-nvdspreprocess.

Коли плагін працює як вторинний класифікатор разом із трекером, він намагається підвищити продуктивність, уникаючи повторного визначення одних і тих же об'єктів у кожному кадрі. Gst-nvinfer робить це шляхом кешування результатів класифікації на відеокарті з унікальним ідентифікатором об'єкта - ключем. Висновок про об'єкт робиться лише тоді, коли його вперше бачать у кадрі (на основі його ідентифікатора об'єкта) або коли розмір (обмежувальна область) об'єкта збільшується на 20% або більше.

Ця оптимізація можлива лише тоді, коли трекер додається, як елемент upstream. Плагін підтримує інтерфейс IPlugin для користувацьких шарів, а також інтерфейс для користувацьких функцій для аналізу вихідних даних детекторів об'єктів та ініціалізації вхідних шарів, які не є зображеннями.

Для повноцінної роботи пайплайну використовується також плагін nvstreammux. Gst-nvstreammux формує пакет кадрів з кількох джерел введення. Під час підключення джерела вводу до nvstreammux (мюксера) потрібно отримати нову панель (пад) за допомогою gst_element_get_request_pad та шаблону панелі sink_%u. Мюксер формує пакетний буфер кадрів пакетного розміру (розмір пакета вказується за допомогою властивості об'єкта gst). Якщо вихідний формат і формат введення мюксеру однакові, він пересилає кадри з цього джерела, як частину вихідного пакетного буфера мюксеру. Кадри повертаються до джерела, коли мюксер повертає свій вихідний буфер.

Якщо роздільна здатність не однакова, мюксер масштабує кадри з входу в пакетний буфер, а потім повертає вхідні буфери до компонента вихідного потоку. Мюксер переміщує пакет вниз за пайплайном, коли буфер заповнений, або досягнуто ліміт формування пакет. Мюксер використовує циклічний алгоритм для збору кадрів із джерел: він намагається зібрати середнє значення (розмір пакету/кількість джерел) кадрів на пакет з кожного джерела (якщо всі джерела активні та їх частота кадрів однакова). Однак кількість змінюється для кожного джерела, залежно від частоти кадрів джерел. Мюксер видає єдину роздільну здатність (тобто всі кадри в пакеті мають однакову роздільну здатність). Цю роздільну здатність можна вказати за допомогою властивостей ширини та висоти. Мюксер масштабує всі вхідні кадри до цієї роздільної здатності. Властивість enable-padding можна встановити на true, щоб зберегти вхідне співвідношення сторін під час масштабування шляхом заповнення чорними смугами. Для платформ DGPU, графічний процесор, який буде використовуватися для масштабування та виділення пам’яті, можна вказати за допомогою властивості gpu-id.

Оберненим плагіном до мюксера (nvstreammux) можна вважати демюксер (gst-nvstreamdemux), який створюється автоматично. Плагін Gst-nvstreamdemux демуксує пакетні кадри в окремі буфери. Він створює окремий буфер Gst для кожного кадру в пакеті, при цьому не копіюючи відеокадри. Кожен буфер Gst містить покажчик на відповідний кадр у пакеті. Плагін переміщує необроблені

об’єкти Gst Buffer на пад (pad), що відповідає джерелу кожного кадру. Демюксер отримує цю інформацію через NvDsBatchMeta, приєднану до Gst-nvstreammux.

Оригінальні позначки часу буфера (PTS) окремих кадрів також приєднуються до Gst Buffer. Оскільки копій кадрів немає, вхідний буфер Gst повертається не відразу, а коли всі непакетні об’єкти Gst-буфера, демуксовані з вхідного пакетного Gst-буфера.

Окрім розпізнавання об’єктів, потрібно ще відстежувати об’єкт у відеоряді. Цим займається плагін gst-nvtracker. Цей плагін дозволяє пайплайну Deepstream використовувати бібліотеку відстеження низького рівня для відстеження виявлених об’єктів із постійними (можливо унікальними) ідентифікаторами з часом. Він підтримує будь-яку низькорівневу бібліотеку, яка реалізує NvDsTrackerAPI, включаючи довідкові реалізації, надані бібліотекою NvMultiObjectTracker : NvDCF, DeepSORT і трекери IOU. Як частина цього API, плагін запитує бібліотеку низького рівня щодо можливостей та вимог формату введення, типу пам’яті та підтримки пакетної обробки. На основі цих запитів плагін потім перетворює буфери вхідних кадрів у формат, який запитує бібліотека відстеження низького рівня. Наприклад, трекери NvDCF і DeepSORT використовують NV12 або RGBA, тоді як IOU взагалі не вимагає буферів відеокадрів.

Можливості низькорівневої бібліотеки трекерів також включають підтримку пакетної обробки кількох вхідних потоків. Пакетна обробка зазвичай є більш ефективною, ніж обробка кожного потоку незалежно, особливо коли прискорення на основі графічного процесора виконується низькорівневою бібліотекою.

Низькорівневі можливості також включають підтримку передачі даних попереднього кадру, що включає дані відстежених об’єктів, згенеровані в минулих кадрах, але ще не повідомлені як вихідні дані. Це може бути у випадку, коли низькорівневий трекер зберігає дані відстеження об’єктів, згенеровані в попередніх кадрах, лише внутрішньо через, скажімо, низьку

впевненість відстеження, але пізніше вирішив повідомити через, скажімо, підвищену довіру.

Якщо дані минулого кадру витягуються з низькорівневого трекера, вони повідомлятимуться як метадані користувача, NvDsPastFrameObjBatch. Це можна ввімкнути за допомогою enable-past-frame параметра конфігурації. На рисунку 1.11 зображена структура плагіну-трекера.

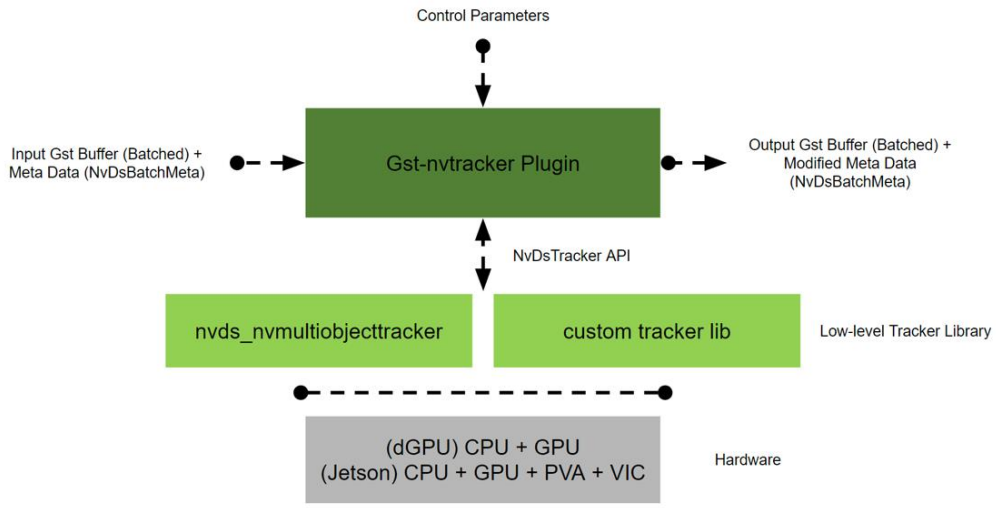

Рисунок 1.11 – Структура плагіну gst-nvtracker

1.5. Клієнт-серверна архітектура

Клієнт-серверна архітектура є однією з популярних архітектур програмного забезпечення та є домінуючою концепцією у створенні розподілених мережевих застосунків і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти (рис. 1.12):

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами.

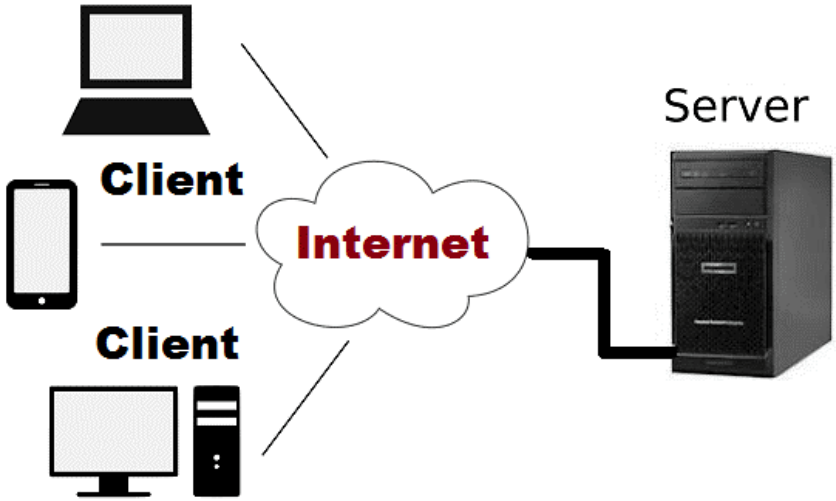

Рисунок 1.12 – Основні компоненти клієнт-серверної архітектури

Найчастіше технологія передачі між клієнтом та сервером використовує REST підхід. REST (від англ. Representational State Transfer, «передача репрезентативного стану») це підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів. REST використовує різні способи передачі даних, таких як, JSON, XML, форми тощо (рис. 1.13).

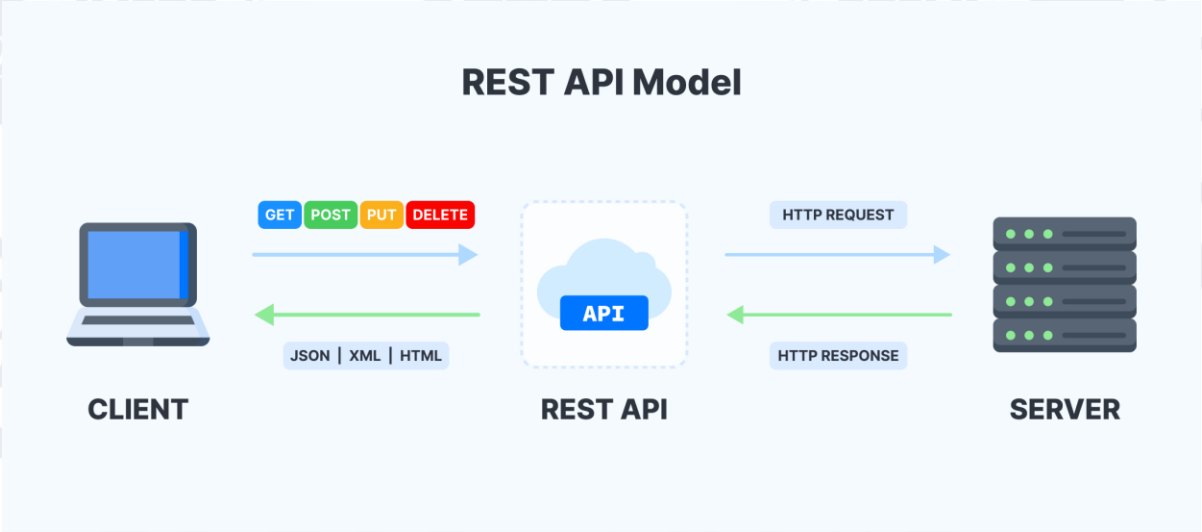

Рисунок 1.13 – REST API підхід

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером. Логічно можна відокремити три рівні операцій:

- рівень представлення даних, який по суті являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
- прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;
- рівень управління даними, який забезпечує зберігання даних та доступ до них.

Дворівнева клієнт-серверна архітектура передбачає взаємодію двох програмних модулів — клієнтського та серверного. В залежності від того, як між ними розподіляються наведені вище функції, розрізняють:

- модель тонкого клієнта, в рамках якої вся логіка застосунку та управління даними зосереджена на сервері. Клієнтська програма забезпечує тільки функції рівня представлення;
- модель товстого клієнта, в якій сервер тільки керує даними, а обробка інформації та інтерфейс користувача зосереджені на стороні клієнта. Товстими клієнтами часто також називають пристрої з обмеженою потужністю: кишенькові комп'ютери, мобільні телефони та ін.

Задача даної роботи є побудова сервісу, який буде працювати на клієнт-серверній архітектурі за допомогою REST. Тобто, автор повинен створити сервери (сервіси), які будуть приймати інформацію від користувачів, та клієнтський додаток, яким буде користуватись користувач. Взаємодія між цими сервісами буде реалізована через HTTP методи.

HTTP – протокол передачі даних, що використовується в комп'ютерних мережах. Назва скорочена від HyperText Transfer Protocol, протокол передачі гіпертекстових документів. Компоненти, що використовують HTTP, можуть самостійно здійснювати збереження інформації про стан, пов'язаний з останніми запитами та відповідями. Браузер, котрий посилає запити, може

відстежувати затримки відповідей. Сервер може зберігати IP-адреси та заголовки запитів останніх клієнтів. Проте, згідно з протоколом, клієнт та сервер не мають бути обізнаними з попередніми запитами та відповідями, у протоколі не передбачена внутрішня підтримка стану й він не ставить таких вимог до клієнта та сервера.

Кожен запит/відповідь складається з трьох частин:

- стартовий рядок;
- заголовки;
- тіло повідомлення, що містить дані запиту, запитаний ресурс або опис проблеми, якщо запит не виконано.

Обов'язковим мінімумом запиту є стартовий рядок. Починаючи з HTTP/1.1 обов'язковим став заголовок Host: (щоб розрізнити кілька доменів, які мають одну й ту ж IP-адресу).

Існують такі HTTP методи, як GET, PUT, POST, DELETE та інші. Всі вони використовуються для обмеження типів запитів. Наприклад, запит з методом GET поверне URI та інші деталі ресурсів колекції, запит з методом POST – створить новий ресурс чи запис в сервісі (колекції).

1.6. Архітектура сервісу

Спираючись на всі дослідження та знання автора, є доцільним побудувати архітектуру майбутнього застосунку для детекції медичних масок на обличчях. Весь застосунок складається із п’яти сервісів (серверів), дві бази даних та додаткових застосунків для полегшення виконання коду: проксі, черга RabbitMQ (рис. 1.14).

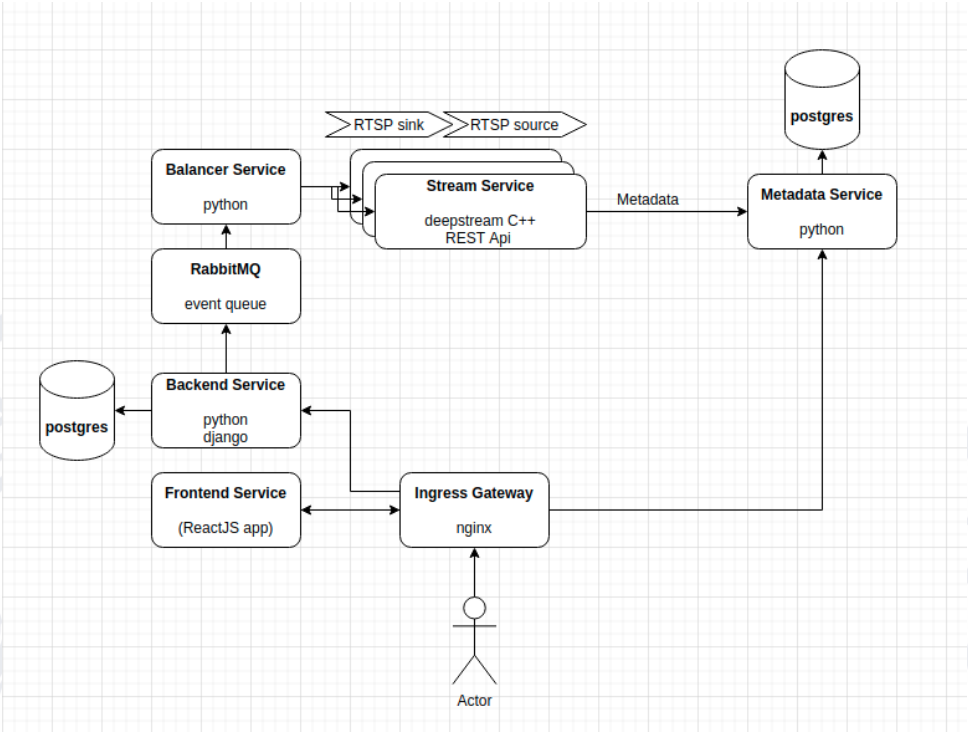

Рисунок 1.14 – Архітектура сервісу

Коротко розглянемо кожний сервіс окремо:

- Stream service (від англ. сервіс потоку) – використовує технологію Nvidia Deepstream SDK та написаний на мові програмування C++. Даний сервіс буде використаний, як основний, за допомогою якого можна відстежувати медичні маски на обличчях людей.
- Metadata service (від англ. сервіс метаданих) – сервіс для отримання даних із сервісу потоку та зберігання їх до бази даних. Використовує мову програмування Python та взаємодіє з базою даних PostgreSQL.
- Balancer service (від англ. сервіс балансування) – побудований на мові Python та використовується для балансування сервісів потоку.
- Backend service – сервіс, який є основою керування всього застосунку, використовує базу даних PostgreSQL та чергу подій RabbitMQ.
- Frontend service – клієнтський сервіс для взаємодії з застосунком через користувацький інтерфейс. Полегшує роботу з застосунком та написаний на мові програмування Javascript з застосуванням фреймворку React.

Маючи готову архітектуру, варто описати як саме працюватиме застосунок. Наприклад, університет вирішив використати даний додаток для підрахунку кількості студентів, що носять маски в приміщенні. Для цього, адміністрація університету реєструється через frontend сервіс та надає доступ до своїх відеокамер. Всі дані про камери, користувачів зберігаються в базі даних. Далі, адміністратори запускають програму, запит посилається спочатку на backend сервіс, де перевіряються дані з камер, користувачі. Потім запит йде на чергу подій, яка почергово посилає запит на сервіс балансування. Сервіс балансування виділяє з-поміж усіх сервісів потоку, один вільний сервіс, на якому можна запустити детекцію об’єктів (масок) з відеоряду камери. В реальному часі сервіс потоку знаходить маски на обличчях людей та рахує кількість людей з масками та без масок. Ці дані надходять запитом на сервіс метаданих, який взаємодіє з базою даних, куди і заносить цю інформацію. Надалі, адміністратор може отримати з бази даних інформацію про кількість людей, що були у масках, за певний період через frontend сервіс.

Варто оглянути СУБД PostreSQL. PostgreSQL – це потужна система об’єктно-реляційних баз даних з відкритим вихідним кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, які безпечно зберігають і масштабують найскладніші робочі навантаження даних.

PostgreSQL заслужив міцну репутацію завдяки своїй перевіреній архітектурі, надійності, цілісності даних, надійному набору функцій, розширюваності та відданості спільноти відкритих вихідних кодів, які стоять за програмним забезпеченням, щоб постійно надавати продуктивні та інноваційні рішення. PostgreSQL працює на всіх основних операційних системах, має ACID - сумісність з 2001 року та має потужні доповнення, такі як популярний розширювач геопросторової бази даних PostGIS.

Нижче наведено невичерпний список різноманітних функцій, які можна знайти в PostgreSQL:

- Типи даних
    - Примітиви: цілі, числові, рядкові, логічні

- Структуровані: дата/час, масив, діапазон, UUID
- Документи: JSON/JSONB, XML, ключ-значення (Hstore)
- Геометрія: точка, лінія, коло, многокутник
- Налаштування: композитні, користувацькі типи

- Цілісність даних
  - Унікальний, ненульовий
  - Первинні ключі
  - Зовнішні ключі
  - Обмежувальні виключення
  - Явні блокування, рекомендаційні блокування
- Паралелізм, продуктивність
  - Індексування
  - Розширене індексування: GiST, SP-Gist, KNN Gist, GIN, BRIN, індекси покриття, фільтри Блума
  - Складний планувальник запитів/оптимізатор, сканування лише для індексів, багатоколонкова статистика
  - Транзакції, вкладені транзакції
  - Багатоверсійний контроль паралельності (MVCC)
  - Паралелізація запитів на читання та побудова індексів B-дерева
  - Усі рівні ізоляції транзакцій, визначені в стандарті SQL, включаючи Serializable
  - Компіляція виразів «точно вчасно» (JIT).
- Надійність, аварійне відновлення
  - Попереднє ведення журналу (WAL)
  - Реплікація: асинхронна, синхронна, логічна
  - Відновлення на момент часу (PITR), активний режим очікування
  - Таблиці
- Безпека
  - Аутентифікація: GSSAPI, SSPI, LDAP, SCRAM-SHA-256, сертифікати тощо

- Надійна система контролю доступу
- Безпека стовпців і рядків
- Багатофакторна аутентифікація за допомогою сертифікатів і додаткових методів

- Розширюваність
  - Збережені функції та процедури
  - Процедурні мови: PL/PGSQL, Perl, Python (і багато інших)
  - Підтримка SQL/JSON
  - Налаштування інтерфейсу зберігання для таблиць
  - Багато розширень, які надають додаткову функціональність, включаючи PostGIS
- Повнотекстовий пошук

Було доведено, що PostgreSQL має високу масштабованість, як за величезною кількістю даних, якими він може керувати, так і за кількістю одночасних користувачів, які він може вмістити. У виробничих середовищах є активні кластери PostgreSQL, які керують багатьма терабайтами даних, і спеціалізовані системи, які керують петабайтами.

Через великий список переваг, автор вирішив використати дану СУБД для реалізації поставленої задачі.

1.7. Додаткові можливості застосунку

Оскільки основний сервіс – розпізнавання медичних масок, то існують додаткові можливості реалізації деяких фішок. Наприклад, система зберігання обрізаних фотографій людей, котрі знаходяться без маски у приміщенні. Дану функцію легко реалізувати, завдяки сервісу метаданих, до якого будуть приходить ті кадри з відеокамери, на яких людина без маски.

Ще однією корисною функцією буде система «штрафів». Тобто, можна розробити додатковий застосунок для смартфонів, в якому кожен користувач буде реєструватись за допомогою обличчя (на зразок FaceID). Далі варто навчити додаткову модель, яка буде визначати з відеокамер обличчя, які схожі

на обличчя у базі даних. А далі – відсилати на пошту користувача попередження чи повідомляти керівництво університету про порушення.

РОЗДІЛ 2

РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

2.1. Реалізація сервісу потоку

Основним фреймворком для реалізації сервісу потоку (stream service) є Nvidia Deepstream SDK. Оскільки даний фреймворк підтримує лише 3 мови програмування: Python, C, C++, автор буде користуватись останньою задля покращення швидкості роботи та легкості написання коду. Компілюватись програма буде за допомогою cmake – обгортки над компілятором, яких генерує файли типу CMakeList.txt та Makefile.

Перше, на що варто звернути увагу, це модель навченої нейронної мережі для розпізнавання. Аби зменшити час тренування моделі та навантаження на відеокарту, доцільно буде використати попередньо навчену модель. Її можна завантажити з офіційного сервісу компанії Nvidia – NGC model registry, попередньо зареєструвавшись через браузер. Автор вважає, що найкраще завантажувати попередньо навченою модель detectnet_v2, оскільки інші моделі потребують попередню обробку вхідних даних за допомогою віднімання середнього значення та зміни порядку BGR каналів. Таким чином, використання інших попередньо навчених моделей може призвести до меншої ефективності.

Наступним кроком є завантажити датасет та приготувати до навчання. Обрані автором набори даних (FDDB, WiderFace, MAFA) описані вище, у першому розділі. Задля зменшення навантаження та часу тренування, можна використовувати не усі картинки з датасетів, а лише певну кількість. Наприклад, краще узяти близько 3000 картинок облич із масками та 3000 картинок облич без масок, що було і зроблено автором. Об’єднавши ці дві категорії, маємо 6000 картинок для навчання. Оскільки завантажені набори у різних форматах, потрібно привести їх до єдиного, наприклад, KITTI формату.

KITTI формат – це формат датасету, в якому окрім даних, присутній текстовий файл із мітками до кожного об’єкту на картинці.

Конвертувавши набори даних до єдиного формату, варто створити tfrecords. TFrecords – формат для зберігання послідовності двійкових записів. Цей формат зберігає усі дані у двійковому вигляді, задля полегшення тренування моделі. Ці записи варто згенерувати лише один раз. Створити tfrecords можна за допомогою програми tlt-dataset-convert, яка є частиною Transfer Learning Toolkit, описаного у першому розділі.

Важливим кроком є правильно сконфігурувати параметри тренування: розмір батчу (batch size), кількість епох, рейтинг навчання тощо. Адже саме правильно обрані параметри гарантують найкращу точність моделі, а також, швидкість тренування. На рисунку 2.1, автор вирішив використати стандартні параметри навчання.

```
training_config {
  batch_size_per_gpu: 24
  num_epochs: 120
  learning_rate {
    soft_start_annealing_schedule {
      min_learning_rate: 5e-06
      max_learning_rate: 5e-04
      soft_start: 0.10000000149
      annealing: 0.699999988079
    }
  }
  regularizer {
    type: L1
    weight: 3.00000002618e-09
  }
  optimizer {
    adam {
      epsilon: 9.99999993923e-09
      beta1: 0.899999976158
      beta2: 0.999000012875
    }
  }
  cost_scaling {
    initial_exponent: 20.0
    increment: 0.005
    decrement: 1.0
  }
  checkpoint_interval: 10
}
```

Рисунок 2.1 – Налаштування параметрів тренування

Тренування відбувається за допомогою програми tlt-train (Transfer Learning Toolkit). В аргументи програми потрібно вказати конфіг параметрів

тренування, вихідну папку з моделю, попередньо навчену модель (ваги) та бажану кількість задіяних відеокарт для навчання (рис. 2.2).

```
!tlt-train detectnet_v2 -e $SPECS_DIR/detectnet_v2_train_resnet18_kitti.txt \
                        -r $USER_EXPERIMENT_DIR/experiment_dir_unpruned \
                        -k $KEY \
                        -n resnet18_detector \
                        --gpus $NUM_GPUS
```

Рисунок 2.2 – Приклад запуску тренування моделі

Після завершення процесу навчання, є кілька можливих варіантів для покращення точності моделі і уникнення перенавчання. Першим варіантом є оцінка натренерованої моделі за допомогою програми tlt-evaluate, в яку потрібно передати навчену модель та конфіг параметрів навчання. Другим варіантом є зменшення розміру моделі, щоб покращити швидкість детекції в режимі реального часу (рис. 2.3). Для зменшення розміру слід використати програму tlt-prune, яка приймає навчену модель, критерій вирівнювання, поріг для обрізки моделі, директорії зберігання обрізаної моделі. Виконавши обидва етапи автор отримав 80% точності моделі.

```
!tlt-prune -m $USER_EXPERIMENT_DIR/experiment_dir_unpruned/weights/resnet18_detector.tlt \
           -o $USER_EXPERIMENT_DIR/experiment_dir_pruned/resnet18_nopool_bn_detectnet_v2_pruned.tlt
           -eq union \
           -pth 0.8 \
           -k $KEY
```

Рисунок 2.3 – Приклад обрізання моделі

Оскільки, така точність моделі є замалою для задачі великого масштабу, варто перетренувати і оцінити обрізану модель ще раз, за допомогою програм описаних вище. Слід тільки змінити конфіги параметрів для перетренування, та зберігти бекап моделі, оскільки можливе зменшення точності при неправильних параметрах.

Останнім кроком буде експортування моделі під формат використання TensorRT фреймворку. TensorRT – це бібліотека C++ для високопродуктивного

розпізнавання на графічних процесорах NVIDIA. Після експортування, модель готова для використання.

Оскільки навчена модель експортована та готова до детекції, потрібно написати сервіс потоку, який буде використовувати дану модель. Головний фреймворк сервісу потоку використовує Deepstream SDK, який будується з певних елементів. На рисунку 2.4 показано повний пайплайн використання даного фреймворку. Даний пайплайн складається із 11 елементів, які почергово пов’язані один з одним. Кожен елемент-плагін виконує свою роль, наприклад, плагін uridecodebin зчитує інформацію із rtsp потоку камери та покадрово відсилає дані на наступний плагін. Основний плагін Nvinfer буде використовуватись моделлю, як обгортка для детекції масок. Також задля покращення роботи потрібно додати плагіни-відеоконвертери та енкодери (nvvideoconvert, x264enc). Розпізнанні дані з кадрів будуть відсилатись на сервіс метаданих через плагін datasender. Udpsink – є простим плагіном візуалізації контенту.

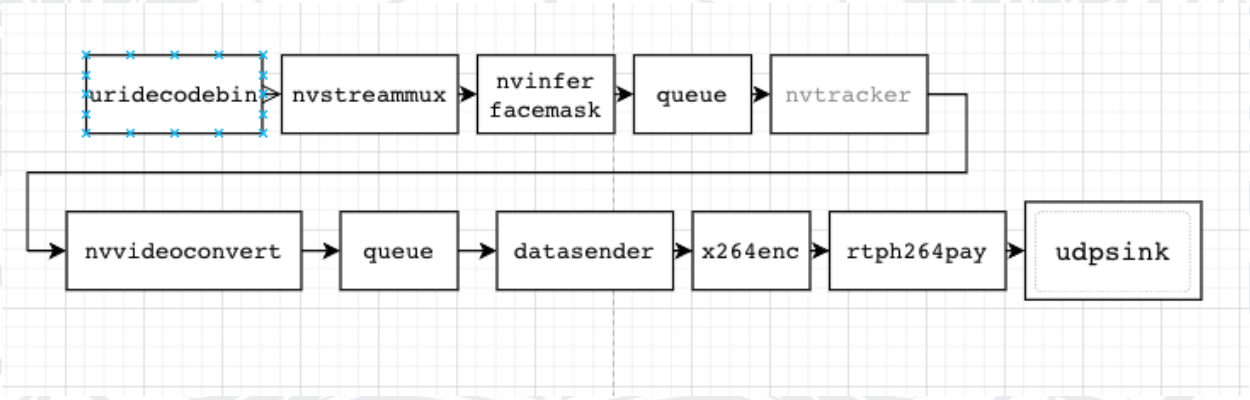

Рисунок 2.4 – Пайплайн Deepstream SDK сервісу потоку

Побудувавши даний пайплайн за допомогою C++ є можливість локально запускати сервіс для тестування моделі. Але найкращим варіантом буде автоматизувати сервіс, для використання клієнтами. Для цього потрібно написати обгортку над пайплайном.

Варто зазначити, що можливість передачі візуалізації детекції з камери клієнту реалізовується тільки за допомогою вихідного rtsp потоку. Це є

обмеження фреймворку Deepstream. Щоб реалізувати передачу потоку, потрібно створити rtsp сервер, який працювати на виділеному сервісом порті та передавати посилання через проксі користувачу.

2.2. Реалізація сервісу метаданих

Збір даних та статистики є головною метою поставленої задачі, адже без кінцевого результату важко оцінити якість роботи застосунку. Сервіс метаданих побудований таким чином, щоб отримувати дані за допомогою HTTP запитів з плагіну сервісу потоку та зберігати у базі даних PostgreSQL. Для цього, потрібно розробити декілька API ендпойнтів, на які будуть приходити дані з сервісу потоку. API ендпойнт (з англ. API endpoint) – точка прийому запиту на стороні серверу (API). Автор пропонує розробити мінімум два ендпойнти:

1. Ендпойнт типу POST (створити), для отримання даних із сервісу потоку і занесення у базу даних.
2. Ендпойнт типу GET (отримати), для отримання даних із бази даних.

Доцільно буде створити ендпойнти для керування операціями з базою даних, наприклад, видалити запис, редагувати запис, створити нову колекцію тощо.

Оскільки за архітектурою застосунку, сервіс метаданих використовує мову програмування Python, потрібно визначити найкращий фреймворк для REST API. З-серед популярних можна виділити Flask, Django, FastAPI. Останній найкраще підійде через швидкість та простоту написання коду.

Для взаємодії з базою даних найкращою практикою буде використання ORM технології. ORM (від англ. Об’єктно-реляційне відображення) – технологія програмування, яка зв’язує бази даних з концепціями об’єктно-орієнтованих мов програмування, створюючи «віртуальну об’єктну базу даних». Для використання цієї технології підійде фреймворк sqlalchemy, який підтримується мовою програмування Python. Тобто не потрібно писати процедури для баз даних, ініціалізацію таблиць, фреймворк sqlalchemy зробить це автоматично, при наявності відповідного коду.

Існує задача підрахунку кількості людей в масках, яка реалізована на сервісі потоку. Дані заносяться в базу даних через сервіс метаданих в одну таблицю, яка містить поля з датою та кількістю людей за цю дату, що були у масках, а також айді стріму та загальну кількість людей за цю дату (рис. 2.5).

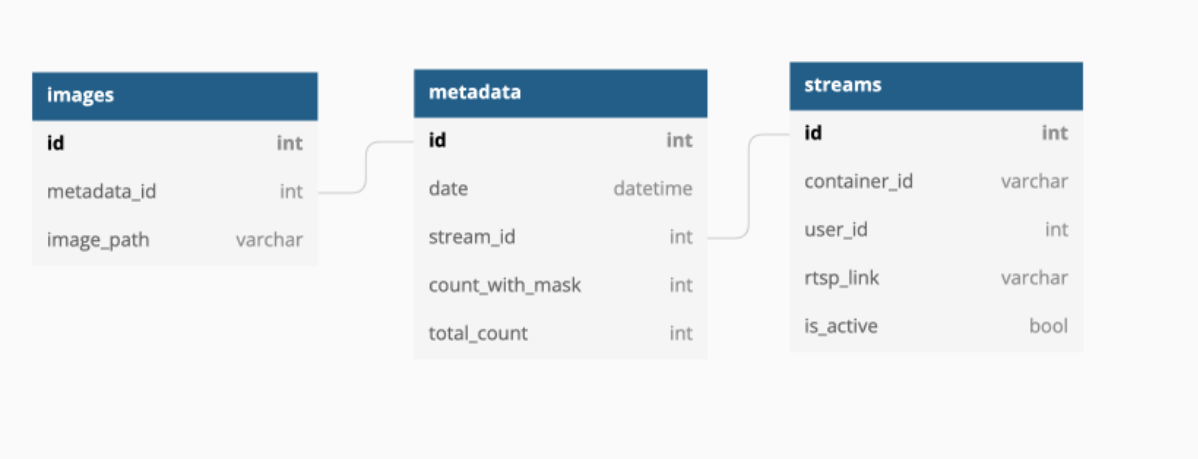

Рисунок 2.5 – Схема бази даних для сервісу метаданих

Також можна реалізувати зберігання фотографій людей, що були без масок в певний проміжок часу. Для цього потрібно буде на сервісі метаданих додати додатковий шар реалізації зберігання знімків у сховищі за допомогою Ceph. Ceph – сховище файлів (зображень, відео, текстових файлів), яке зберігає дані у кластері локально. Є недоцільним використання бази даних для зберігання зображень, оскільки це перевантажить базу даних та сповільнить її роботу. На рисунку 2.4, у схемі бази даних, є таблиця для зберігання посилань на локальне сховище зображень, які були збережені у Ceph.

2.3. Реалізація сервісу балансування

Користувач не може вільно оперувати контейнерами сервісу потоку, без допоміжних сервісів, оскільки важко реалізувати оперування пайплайнами через програмний код Deepstream SDK. Для цього варто розробити додатковий сервіс – сервіс балансування.

Сервіс балансування знаходить вільні незадіяні контейнери сервісу потоку за допомогою модулів-обгорток над програмою Docker та Kubernetes. Пошук незадіяних контейнерів відбувається за аналізом використання пам’яті відеокарти. Якщо активність відеокарти низька та відеопам’ять не споживається – даний контейнер сервісу потоку є вільним та незадіяним, в подальшому його можна використовувати для нових відеокамер.

Управління сервісом балансування здіснюється автоматично сервісом бекенду (backend service). Користувач ніяк не може сприяти напряму на сервіс балансування, тим самим можна обмежити права використання сервісу потоку.

Функціонал сервісу балансування також має містити можливість видалення та перезапуск контейнерів сервісів потоку, адже існує шанс перевантаження пам’яті відеокарти через надмірну кількість працюючих контейнерів. Щоб уникнути перевантаження, потрібно встановити ліміт на використання відеокамер чи збільшити кількість можливих відеокарт для застосунку.

2.4. Реалізація бекенд сервісу

Основою будь-якого застосунку на клієнт-серверній архітектурі є сервіс, який оперує усіма іншими сервісами. Таким сервісом можна назвати бекенд сервіс, який виконує авторизацію користувачів, вмикання чи вимикання сервісу потоку та отримує дані з сервісу метаданих.

Даний сервіс написаний мовою програмування Python та використовує фреймворк FastAPI для реалізації HTTP запитів. Бекенд сервіс містить в собі декілька ендпойнтів:

1. Ендпойнти для реєстрації та авторизації користувачів в застосунку. Для цього використовуються типи POST, GET, DELETE запитів. Інформація про користувачів під час реєстрації зберігається у базу даних PostgreSQL, у таблицю «User».
2. Ендпойнти для оперування сервісом потоку: додавання відеокамери, початок розпізнавання масок з відеокамери, видалення відеокамери.

Усі дані, які приходять від користувачів, записуються в базу даних. Схема бази даних представлена на рисунку 2.6.

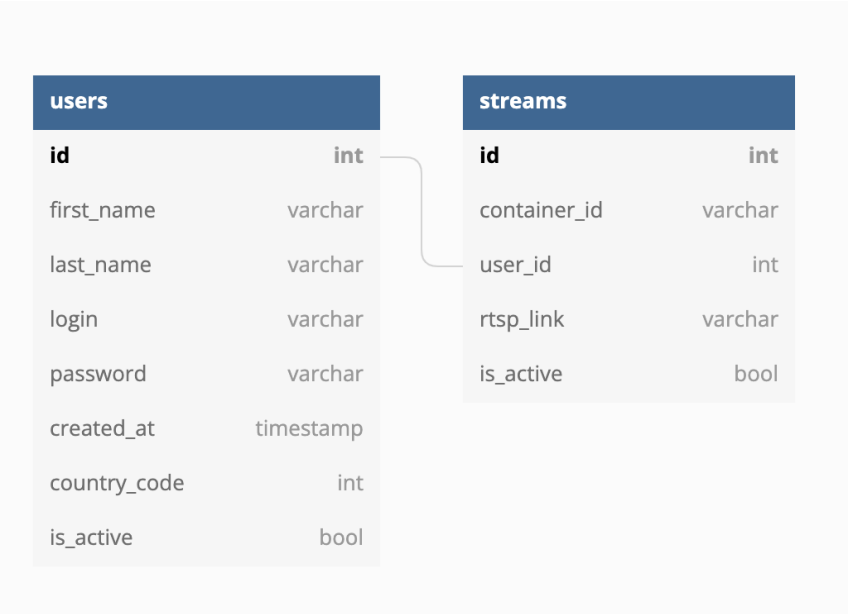

Рисунок 2.6 – Схема бази даних для бекенд сервісу

Для реалізації задачі, достатньо мати лише 2 таблиці – користувачі (users) та потоки (streams). У таблиці користувачів потрібно створити поля прізвища та імені користувача, логіну користувача, а також пароль, який зберігається у зашифрованому форматі. Доцільним буде використання полів created_at, country_code, is_active, що означають час створення користувача, країна користувача, активний чи видалений користучач відповідно. Таблиця потоків має містити поля з айді користувача, який створив потік, посилання на rtsp потік з відеокамери, активність потоку та айді контейнера сервісу потоків, в якому запущено дану відеокамеру.

Важливою частиною оптимізації посилання запитів є система черг або платформа обміну повідомленнями. За це відповідає платформа RabbitMQ. Дана платформа дозволяє надсилати запити почергово в особливому порядку відправлення. Оскільки без системи черг дуже легко перетворити сервіс потоків і балансування в непрацюючі сервіси через перевантаження, є надто важливим використовувати паралельні потоки та систему черг.

2.5. Створення проксі-балансувальника

Задля обмеження доступу користувача до усіх сервісів, окрім користувацького інтерфейсу, є доцільним використання nginx-проксі. При належній конфігурації проксі, можна дозволити доступ лише до сервісу фронтенду (рис. 2.7). Всі інші сервіси будуть доступні тільки всередині локальної мережі застосунку.

```
server {
  listen 80;

  auth_basic "Administrator's Area";
  auth_basic_user_file /etc/nginx/.htpasswd;

  location / {
    auth_basic off;
    proxy_pass http://ui:80/;
    proxy_set_header X-Forwarder-For $remote_addr;
  }
```

Рисунок 2.7 – Конфігурація nginx-проксі

Nginx підтримує також простий метод балансування сервісів бекенд, який реалізований в почерговій відправці запитів на репліки сервісів бекенду. Це дозволить не перевантажувати одну репліку (контейнер) із бекенд сервісом, а використовувати всі можливі ресурси.

Досить важливою частиною є додавання SSL сертифікатів у сервіс проксі, для використання захищенного протоколу HTTPS. На цьому етапі потрібно створити сертифікати TLS/SSL для необхідного нам домену, розміщених в Nginx. Отримати сертифікати можна через Let's Encrypt – центр сертифікації. Nginx підтримує кінцеві вузли SSL, тому можна налаштувати SSL без зміни конфігурації файлів Nginx.

Для отримання сертифікату, потрібно використати програму Certbot для генерування сертифікатів TLS/SSL. При запуску Certbot, необхідно вказати адресу електронної пошти і прийняти умови обслуговування. Після цього програма зв’язується із сервером Let's Encrypt, відправляється запит з метою підтвердити, власність домену, для якого запитуєте сертифікат.

Сгенеровані сертифікати варто додати до конфігу nginx, вказавши перенаправлення з порту 80 (http) до порту 443 (https), а також домен бажаного сайту (рис. 2.8).

```
server {
    listen 443 ssl http2;
    server_name mask.donnu.edu.ua;
    location / {
            proxy_pass http://ui;
        }

    location /process {
            proxy_pass http://api;
    }

    location /csv {
            proxy_pass http://api;
    }

    location /json {
            proxy_pass http://api;
    }

    ssl_certificate /etc/letsencrypt/live/mask.donnu.edu.ua/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/mask.donnu.edu.ua/privkey.pem;
    ssl_trusted_certificate /etc/letsencrypt/live/mask.donnu.edu.ua/chain.pem;
    ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
}
```

Рисунок 2.8 – Оновлений конфіг nginx-проксі з сертифікатами

2.6. Розробка сервісу фронтенду

Головним та єдиним користувацьким інтерфейсом є сервіс фронтенду (frontend service). Він має вміщувати в собі форми реєстрації та авторизації користувачів, а також сторінку з відображенням відеопотоку та статистики даних користувачів.

Сервіс реалізований за допомогою мови програмування Javascript та фреймворком React. За допомогою даної мови та HTML можна легко

інтегрувати rtsp потік в сторінку сайту, тим самим візуалізувавши детекцію масок на обличчях в реальному часі. Варто зазначити, що всі сервіси застосунку мають бути в єдиній локальній мережі.

Користувацький інтерфейс має бути інтуїтивно зрозумілим та лаконічним та містити в собі тільки необхідні елементи. Додавання форм та елементів відбувається за допомогою React.

Інтерфейс має бути оптимізований під будь-яку платформу та встановлюватись як додаток. Для цього потрібно реалізувати PWA технологію – коли можна завантажити сайт, як додаток, на будь-яку платформу.

PWA – це веб-додаток, створений з використанням певних технологій для досягнення завантаження сайту, як додатку. PWA повинен бути:

- надійним – програма завантажується і показується відразу, незалежно від статусу та якості мережі;
- швидким – взаємообмін даними через мережу повинен відбуватись швидко, користувацький інтерфейс повинен бути плавним;
- привабливим – гарний інтерфейс для користувача є досить важливим для комфортної і приємної роботи.

Головним сервісом PWA є service worker – шар проксі між фронтендом та бекендом. Всі запити браузера проходять через service worker. Даних поділ на два незалежні шари дозволить зробити перехід звичайного веб-сайту в PWA максимально простим.

Зі сховищ у service worker'a є доступ до cache storage для веб-ресурсів, та IndexDB для даних. Але найголовніше, повна свобода для реалізації бізнес логіки.

PWA вимагає, щоб усі ресурси сайту передавалися за протоколом HTTPS. Критично, щоб на сайті не було посилань на незахищені ресурси - деякі браузери просто не відображатимуть сайт у цьому випадку.

Важливо сконфігурувати JSON файл, який буде визначати для браузера назву програми, іконку, як виглядатиме розмір та деякі інші параметри. Даний

файл дозволяє встановити PWA як окрему програму на домашній екран смартфона.

Завдяки стандартній бібліотеці React Router, можна створити маршрутизацію сторінок у сервісі. Компоненти типу «router» відображають певні вказані компоненти при певних значеннях адресної строки. В якості бібліотеки для посилання HTTP запитів, була використана Axios. Передача запитів відбувається форматом JSON, в якому вказуються дані користувача для реєстрації та авторизації, а також різні параметри для відеокамери та потоків. Варто зазначити, що саме при реєстрації та авторизації, на сервісі фронтенду, зашифровуються дані про пароль, задля безпеки користувача.

Реалізація авторизації виконана через кукі (з англ. cookie). За допомогою хуків можна отримати дані з куків та сесії браузера, чи був авторизований користувач (рис. 2.9).

```
export const checkLoginThunk = () => {
    return (dispatch: any) => {
        const temp = document.cookie
        if(temp.match( matcher: /session=.+/)){
            dispatch(actions.setIsLogined(true))
        }else {
            dispatch(actions.setIsLogined(false))
        }
    }
}
```

Рисунок 2.9 – Алгоритм перевірки авторизації користувача

HTTP-cookie — у комп'ютерній термінології поняття, яке використовується для опису інформації у вигляді текстових або бінарних даних, отриманих від вебсайту на вебсервері, яка зберігається у клієнта, тобто браузера, а потім відправлена на той самий сайт, якщо його буде повторно відвідано.

2.7. Контейнеризація та розгортання

Важливим етапом є контейнеризація усіх сервісів за допомогою Docker. Для кожного сервісу потрібно написати ізольований образ контейнеру, щоб виконання сервісу було незалежне від середовища користувача.

Перевагами Docker можна назвати:

1. Постійність. Використання платформи, яка працює однаково в різних середовищах, полегшує роботу. Усі команди та код працює однаково, незалежно від сервера, машини чи операційної системи, яку вони використовують.

2. Автоматизація. Існує багато завдань, які, як розробник, можуть стати повторюваними і монотонними, якщо виконувати їх вручну. Контейнери Docker дозволяють планувати виконання ряду завдань, коли вони потрібні, без ручного втручання людини. Це економить час, зусилля та полегшує робоче навантаження.

3. Стабільність. Docker розроблений на основі операційної системи Linux і має ядро Linux у кожному контейнері, незалежно від системи, на якій він працює. У минулому це могло викликати деякі незначні проблеми зі стабільністю під час запуску контейнерів на системах Mac або Windows. У наші дні, незважаючи на те, що Docker часто оновлюється, середовище залишається стабільним на будь-якій системі чи пристрої.

4. Економить пам’ять. Попередником контейнерів була віртуальна машина (VM). ВМ працюють так само, як і контейнери, але беруть фізичні сервери, використовуючи величезні обсяги фізичного серверного простору та тонни пам’яті. Контейнери Docker використовують лише код програми та її залежностей і можуть повністю працювати в хмарному середовищі, тобто вони набагато менші та не потребують великих фізичних серверів.

Ізольовані образи пишуться власною мовою програми Docker у файлах з назвою Dockerfile. Найкращою практикою є використання готових образів, які можна знайти в середовищі DockerHub, та додаванням до них свого коду та залежностей.

Для сервісу потоку варто взяти deepstream образ, як основний, який зберігається у регістрах компанії Nvidia. В цьому образі вже є превстановлені фреймворки CUDA, TensorRT та Deepstream, встановлення яких займає багато часу. Варто додати до готового образу встановлення Gstreamer, Python, Cmake, а також скопіювати код до образу. Після написання докерфайлу (рис. 2.10), потрібно скомпілювати даний файл за допомогою команди docker build, вказавши назву образу та допоміжні аргументи.

```
FROM nvcr.io/nvdeepstream/deepstream6_ea/deepstream:6.0-ea-21.06-triton

# -- Gstreamer & python3.6 installation
RUN apt-get update && apt-get -y upgrade && apt-get install -y \
    vim git wget unzip \
    build-essential pkg-config gcc g++ make cmake software-properties-common

RUN add-apt-repository -y ppa:deadsnakes/ppa && apt-get update && apt-get install -y \
    libgstreamer1.0-0 gstreamer1.0-dev gstreamer1.0-tools gstreamer1.0-doc \
#    TODO: remove unnecessary libs (gstreamer is already installed in deepstream:6.0-ea-21.06-triton)
#    gstreamer1.0-plugins-base gstreamer1.0-plugins-good \
#    gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-libav \
#    gstreamer1.0-doc gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa \
#    gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-pulseaudio gstreamer-1.0 \
    python3.6 python3.6-dev python-dev python3-dev python3-pip python-dev \
    python3.6-venv python3-gi python3-gi-cairo python3-gst-1.0 python-gi-dev \
    libgirepository-1.0-1 libgirepository1.0-dev libgstreamer-plugins-base1.0-dev \
    libcairo2-dev gir1.2-gstreamer-1.0 libboost-all-dev libssl-dev

# -- Upgrade cmake
RUN wget https://github.com/Kitware/CMake/releases/download/v3.19.2/cmake-3.19.2.tar.gz && \
    tar -zxvf cmake-3.19.2.tar.gz && \
    cd cmake-3.19.2 && \
    ./bootstrap && \
    make && \
    make install && \
    cd .. && rm -rf cmake-3.19.2 && rm cmake-3.19.2.tar.gz
```

Рисунок 2.10 – Докерфайл для сервісу потоку.

Залежності інших сервісів мінімальні, тому можна використовувати готові образи Python чи NodeJS, лише додаючи до образу код сервісу. На рисунку 2.11 зображено докерфайл для фронтенд сервісу, який складається лише з головного образу NodeJS, установки залежностей-модулів та копіювання коду.

```
FROM node:12.18.1 AS builder
ENV NODE_ENV=production

WORKDIR /app

COPY package.json ./
RUN npm install --production

COPY . .
RUN npm run build

FROM nginx:latest

COPY default.conf /etc/nginx/conf.d/default.conf
COPY --from=builder /app/build /usr/share/nginx/html

ENTRYPOINT nginx
```

Рисунок 2.11 – Докерфайл фронтенд сервісу

Образи для бази даних та системи черг RabbitMQ слід використати стандартні, від авторів даних програм.

Маючи всі готові образи сервісів, варто об’єднати їх в одну працюючу систему, з однією мережою, в якій кожен сервіс може спілкуватись один з одним. Для цього є два способи реалізації об’єднання: docker-compose та Kubernetes.

Docker-compose – це інструмент для визначення та запуску багатоконтейнерних образів Docker. Compose налаштовується за допомогою файлу конфігурацій типу YAML/YML. В налаштуваннях можна вказати назви контейнерам, кількість реплік, приєднані диски до контейнерів, мережі, якими спілкуватимуться контейнери тощо.

Kubernetes – це система оркестрування контейнерів з можливістю автоматизації розгортання, масштабування контейнерів Docker.

Різниця в compose і Kubernetes в тому, що compose є набагато простішим в управлінні і розгортанні і підходить для швидкого запуску легких програм. В свою чергу Kubernetes побудований для складніших за архітектурою застосунків та дозволяє набагато простіше і краще масштабувати контейнери та створювати репліки.

Оскільки сервісу потоку потрібно автоматичне та якісне масштабування контейнерів, варто обрати Kubernetes, як основну систему керування мультиконтейнерами.

Швидкість розгортання, транспортування робочого навантаження та гарне поєднання зі способом роботи DevOps, є причиною використання масштабування. Контейнери можуть значно спростити надання ресурсів розробникам із обмеженим часом.

Kubernetes полегшує тягар налаштування, розгортання, керування та моніторингу навіть найбільших контейнерних програм. Це також допомагає IT-фахівцям керувати життєвими циклами контейнерів і пов'язаними життєвими циклами додатків, а також проблемами, включаючи високу доступність і балансування навантаження.

Важливою основною Kubernetes є оператори. Оператори — це клієнти API Kubernetes, які контролюють користувацькі ресурси. Ця можливість дозволяє автоматизувати такі завдання, як розгортання, резервне копіювання та оновлення, переглядаючи події без редагування коду Kubernetes.

Kubernetes використовує кластерну систему. Кластер — це група або група вузлів, які запускають контейнерні програми.

Под Kubernetes - найменший блок екосистеми Kubernetes, який можна розгорнути. Под конкретно представляє групу з одного або кількох контейнерів, які працюють разом у кластері.

Вузол або нода складається з фізичних або віртуальних машин у кластері. Ці машини мають все необхідне для запуску контейнерів програм, включаючи середовище виконання контейнера та інші важливі служби.

Kubectl — це інтерфейс командного рядка (CLI) для керування операціями на кластерах Kubernetes. Це робиться шляхом зв'язку з Kubernetes API.

Щоб сконфігурувати автоматичне створення кластеру Kubernetes, потрібно скористатись скриптом створення кластеру, що полегшить подальшу роботу (рис. 2.12).

```
# ----- Clean previous deployment ----- #

sudo kubeadm reset -f
sudo rm -dR "${HOME}/.kube" 2> /dev/null && echo "Removed ${HOME}/.kube"
sudo rm -dR "${CLUSTER_DIR}" 2> /dev/null && echo "Removed ${CLUSTER_DIR}"
sudo rm -dR /etc/cni/net.d 2> /dev/null && echo "Removed /etc/cni/net.d"


# ----- Prepare the machine ----- #

sudo swapoff -a
sudo iptables -P FORWARD ACCEPT
mkdir -vp "${CLUSTER_DIR}"


# ----- Add kube-vip static pod to manifests ----- #
docker run --network host --rm plndr/kube-vip:v0.3.7 manifest pod --interface $KUBE_VIP_INTERFACE \
           --vip $KUBE_VIP --controlplane --services --arp --leaderElection \
        | sudo tee  /etc/kubernetes/manifests/kube-vip.yaml > /dev/null

# ----- Initialize the control plane ----- #

envsubst < template/initial-configuration.tmpl.yaml > "${INIT_CONFIG}"
sudo kubeadm init --config "${INIT_CONFIG}" --upload-certs

mkdir -p $HOME/.kube
sudo cp -f /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

if [[ "${SCHEDULE_PODS_ON_MASTER}" == "true" ]]
then
    sudo kubectl taint nodes --all node-role.kubernetes.io/master-
fi
```

Рисунок 2.12 – Скрипт створення кластеру Kubernetes

За аналогією створення образів докер, необхідно створити конфігурації для подів Kubernetes. Для цього використовуються файли типу YAML/YML. Для кожного сервісу потрібно створити свій конфіг та вказати необхідні параметри запуску, додаткові залежності, кількість контейнерів та реплік подів. Приклад файлу конфігурації для сервісу проксі зображено на рисунку 2.13.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gateway-ingress
  namespace: production1
  annotations:
    ingress.kubernetes.io/ssl-redirect: "false"
    nginx.org/rewrites: "serviceName=web-service rewrite=/;serviceName=admin-service rewrite=/"
spec:
  ingressClassName: nginx
  defaultBackend:
    service:
      name: default-backend
      port:
        number: 80
```

Рисунок 2.13 – Файл конфігурації сервісу проксі

Щодо реплікації та масштабування сервісів, автор вирішив масштабувати сервіси таким чином:

- сервіс потоку – 1 поди по 3 контейнери
- сервіс метаданих – 1 под з 3 контейнерами
- бази даних – 1 под з 3 контейнерами
- сервіс бекенду та балансування – 1 под з 3 контейнерами
- сервіс фронтенд – 1 под з 3 контейнерами.

Варто зазначити, що найкращим способом було б розгортання декілька нодів (міні-кластерів) на різних локальних машинах, але для цього знадобляться додаткові ресурси обчислення, яких немає у автора.

Для моніторингу активності сервісів та використання відеопам’яті, варто додати конфігурацію Prometheus та Grafana. Prometheus — система моніторингу, розроблена спеціально для середовища, що динамічно змінюється. Крім того, вона може використовуватися для традиційної інфраструктури, наприклад, на фізичних серверах з програмами, розгорнутими безпосередньо на ОС. За допомогою мови PromQL можна надсилати запити до Prometheus та отримувати статистику використання відеопам’яті на певних вузлах Kubernetes.

Додаткові плагіни дозволяють збільшити можливості збору статистики, наприклад плагін до СУБД PostgreSQL, який дозволить відслідковувати швидкість запису та кількість під’єднаних сервісів.

Grafana – це веб-додаток для аналітики та візуалізації даних з Prometheus. За допомогою додатку можна налаштувати візуально побачити статистики, зібрані за допомогою програми Prometheus, та встановити певні обмеження на сервіси, які будуть сповіщатись (рис. 2.14).

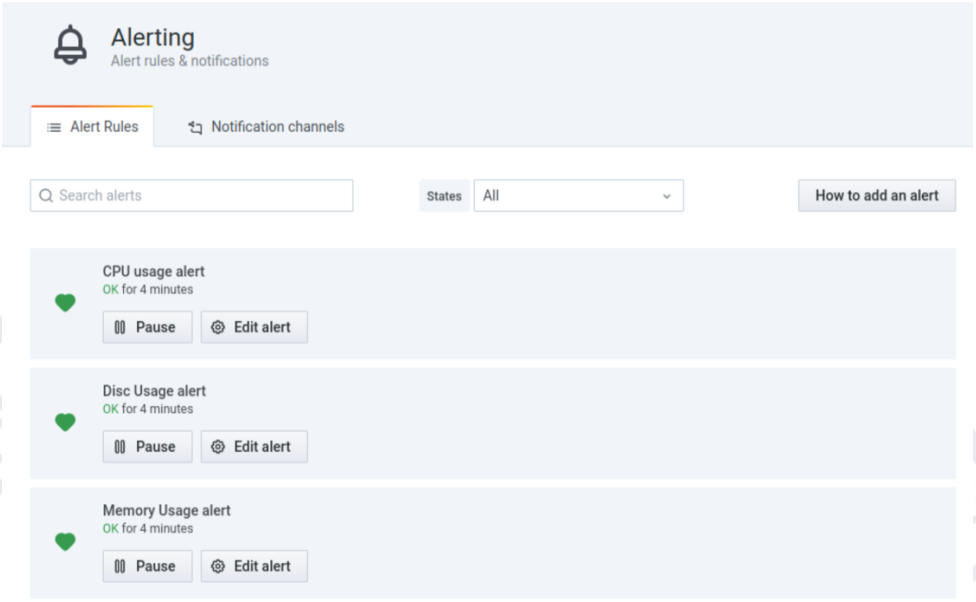

Рисунок 2.14 – Система сповіщень використання CPU, пам’яті

На рисунку 2.15 можна побачити візуалізацію збору статистики для СУБД PostgreSQL.

Рисунок 2.15 – Збір статистики з поду СУБД

Після написання конфігурацій для кожного сервісу, можна розгорнути застосунок на локальній машині за допомогою команд Kubernetes.

ВИСНОВОК

Автором було підготовлено дослідження нейронних мереж та фреймворків для використання детекції медичних масок на обличчях, в ході якого з’ясовано, що найшвидшим фреймворком є Nvidia Deepstream SDK в парі з нейронною мережою detectnet_v2. Для реалізації застосунку, було використано клієнт-серверну архітектуру та побудовано сервіси, які взаємодіють між собою та надають інформацію клієнту.

Встановлено, що основною проблемою даної роботи, яку не можна вирішити теоретично – є потужність машинних ресурсів (відеокарт, процесорів), які необхідні для швидкого та точного розпізнавання моделі. І хоча Nvidia Deepstream SDK є найшвидшим варіантом використання моделей для розпізнавання, використовуючи технології відеокарт від Nvidia, залежість від ресурсів пам’яті відеокарти залишається.

Систематизуючи матеріал даної роботи, можна дійти до висновку, що застосунок на базі клієнт-серверної системи для детекції медичних масок на обличчях є складним, але можливим з точки зору реалізації, за наявністю потужних ресурсів.

СПИСОК ДЖЕРЕЛ

Що таке детекція об’єктів?

URL: https://www.mathworks.com/discovery/object-detection.html

Object detection

URL: https://en.wikipedia.org/wiki/Object_detection

Multiple object class detection

URL:https://web.archive.org/web/20150418092531/http://www.mmp.rwth-aachen.de/publications/pdf/mikolajczyk-multiclass-cvpr06.pdf

Detecting faces in images: a survey

URL: http://homepages.cae.wisc.edu/~ece738/notes/Yang02.pdf

Face detection and recognition

URL: https://facedetection.com/

Artificial neural network

URL: https://en.wikipedia.org/wiki/Artificial_neural_network

An Intuitive Explanation of Convolutional Neural Networks

URL: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Deep Learning

URL: https://en.wikipedia.org/wiki/Deep_learning

Applications of artificial intelligence

URL: https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence

List of artificial intelligence projects

URL: https://en.wikipedia.org/wiki/List_of_artificial_intelligence_projects

Scale space

URL: https://en.wikipedia.org/wiki/Scale_space#Deep_learning_and_scale_space

Згорткова нейронна мережа

URL: https://uk.wikipedia.org/wiki/Згорткова_нейронна_мережа

Виявлення обличчя

URL: https://uk.wikipedia.org/wiki/Виявлення_обличчя

Огляд ResNet-18

URL: https://nl.mathworks.com/help/deeplearning/ref/resnet18.html

ResNet-18 pre-trained model for pytorch

URL: https://www.kaggle.com/datasets/pytorch/resnet18

Overview Nvidia detectnet_v2

URL:https://docs.nvidia.com/metropolis/TLT/tlt-user-guide/text/object_detection/detectnet_v2.html

Overview Nvidia FasterRCNN

URL:https://docs.nvidia.com/metropolis/TLT/tlt-user-guide/text/object_detection/fasterrcnn.html

Yolov4 Nvidia overview

URL:https://docs.nvidia.com/metropolis/TLT/tlt-user-guide/text/object_detection/yolo_v4.html

YOLOv5 документація

URL: https://docs.ultralytics.com/

YOLOv5 порівняння з Faster RCNN

URL: https://towardsdatascience.com/yolov5-compared-to-faster-rcnn-who-wins-a771cd6c9fb4

Deepstream документація

URL: https://developer.nvidia.com/deepstream-getting-started

Gstreamer documentation

URL: https://gstreamer.freedesktop.org/documentation/

Python Library reference

URL: https://docs.python.org/3/library/index.html

C++ documentation

URL: https://docs.microsoft.com/ru-ru/cpp/?view=msvc-170

Клієнт-серверна архітектура

URL: https://uk.wikipedia.org/wiki/Клієнт-серверна_архітектура

Архітектурні шаблони програмного забезпечення

URL:https://uk.wikipedia.org/wiki/Архітектурні_шаблони_програмного_забезпечення

Сервер

URL: https://uk.wikipedia.org/wiki/Сервер

Клієнт

URL: https://uk.wikipedia.org/wiki/Клієнт_(інформатика)

Об’єктно-реляційне відображення

URL: https://uk.wikipedia.org/wiki/Об’єктно-реляційне_відображення

Ідеальна архітектура – міф чи реальність?

URL: https://dou.ua/lenta/articles/how-to-choose-right-architecture/

TensorRT documentation

URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html

CUDA Toolkit documentation

URL: https://docs.nvidia.com/cuda/

Implementing a Real-time, AI-Based, Face Mask Detection application for COVID-19

URL: https://developer.nvidia.com/blog/implementing-a-real-time-ai-based-face-mask-detector-application-for-covid-19/

TAO Toolkit Get started

URL: https://developer.nvidia.com/tao-toolkit-get-started

Pruning Models with NVIDIA TLT

URL: https://developer.nvidia.com/blog/transfer-learning-toolkit-pruning-intelligent-video-analytics/

Preparing models for object detection with real and synthetic data and NVIDIA TAO Toolkit

URL: https://developer.nvidia.com/blog/preparing-models-for-object-detection-with-real-and-synthetic-data-and-tao-toolkit/

Javascript documentation

URL: https://developer.mozilla.org/ru/docs/Web/JavaScript

Початок роботи з React

URL: https://uk.reactjs.org/docs/getting-started.html

Get started with Docker

URL: https://docs.docker.com/get-started/

Kubernetes документація

URL: https://kubernetes.io/uk/docs/home/

Prometheus documentation

URL: https://prometheus.io/docs/introduction/overview/

Grafana documentation

URL: https://grafana.com/docs/