ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Машинное обучение и высоконагруженные системы»

УДК 004.42

Отчет о программном проекте на тему:

Разработка сервиса для классификации изображений овощей

Выполнил:

студент
Павлов Сергей Владимирович _____ _____
(подпись) (дата)

Принял руководитель проекта:

Кантонистова Елена Олеговна
Доцент
Факультета компьютерных наук НИУ ВШЭ

_____ _____
(подпись) (дата)

Москва 2023

Содержание

Аннотация

В данной работе обозревается процесс разработки и вывода в эксплуатацию сервиса для классификации изображений: от подготовки сериализованного экземпляра модели для получения предсказаний до размещения пользовательского интерфейса в виде web-страницы в открытом доступе.

В качестве набора данных для основы проекта используется известный Vegetable Image Dataset из сообщества Kaggle. Выбор набора данных обусловлен его готовностью к использованию без дополнительной обработки и наличием качественных решений по его классификации от участников сообщества.

Кодовая база сервиса представляет собой два отдельных приложения: сервер API и web-интерфейс, размещенные в отдельных git репозиториях. Оба приложения являются контейнеризированными и обернуты в Helm-чарты для установки в кластерах Kubernetes. Автоматизация развертывания новых версий выполнена с помощью GitHub Actions.

Акцент в данной работе сделан на качестве следующих ключевых элементов: общей программной архитектуры, кодовых баз приложений сервиса и автоматизации процессов CI/CD. Для полноценного ознакомления с работой рекомендуется просмотр содержимого исходного кода приложений.

Ключевые слова

Классификация изображений, разработка, CI/CD, API, UI, Kubernetes, Docker, Helm, S3, PostgreSQL, Python, PyTorch, ONNX, TypeScript, Litestar, Dramatiq, RabbitMQ, Vue, Vuetify, git, GitHub

1 Введение

1.1 Постановка задачи

В рамках данной работы планируется разработать и вывести в поддерживаемую эксплуатацию сервис для классификации изображений на базе открытого набора данных с фотографиями овощей [14]. План работы состоит из следующих подзадач:

1. Решить задачу классификации овощей по виду овоща (огурец, помидор и т.д.):
   1.1. Поскольку данная задача уже хорошо отработана коллегами из сообщества Kaggle, воспользоваться примерами лучших архитектур для получения хорошего качества предсказаний.
   1.2. Имплементировать выбранную архитектуры модели на стеке, используемом в данной работе (Python [19], PyTorch [21]), обучить модель, оценить качество.
   1.3. Подготовить сериализованный объект с моделью для использования в сессиях инференса в в рантайме сервиса.
2. Найти или разработать алгоритм, позволяющий определить доминантный «цвет» изображения (красный, зеленый и т.д.) и разметить при помощи него исходный набор данных. Также включить в процесс инференса разметку новых изображений с использованием этого алгоритма.
3. Разработать сервер API, позволяющий:
   3.1. Отправлять запрос с комбинацией параметров «количество + вид + цвет» и получать вывод изображений овощей, соответствующих предоставленной комбинации параметров. Предусмотреть адекватную визуализацию отсутствия подходящих изображений в базе.
   3.2. Загружать пользовательское изображение овоща и получать вывод его разметки по виду (полученной через сессию инференса) и цвету (вычисленной через имплементацию алгоритма).
4. Разработать web-интерфейс сервиса для взаимодействия с сервером API через браузер.
5. Настроить автоматизацию процессов CI/CD для сервера API и web-интерфейса.
6. Выполнить развертывание разработанной связки API + UI на демонстрационном стенде с публичным сетевым адресом и провести живую презентацию работы сервиса.

1.2 Ожидаемые результаты

В качестве результата задачи классификации ожидается модель на базе PyTorch, позволяющая предсказывать вид овоща с качеством по accuracy score не менее 0.97 на данных для валидации. Модель должна быть сериализована в формате, подходящем для запуска сессии инференса для загруженных пользователями изображений.

В качестве результата разработки сервиса ожидается приложение со следующей архитектурой:

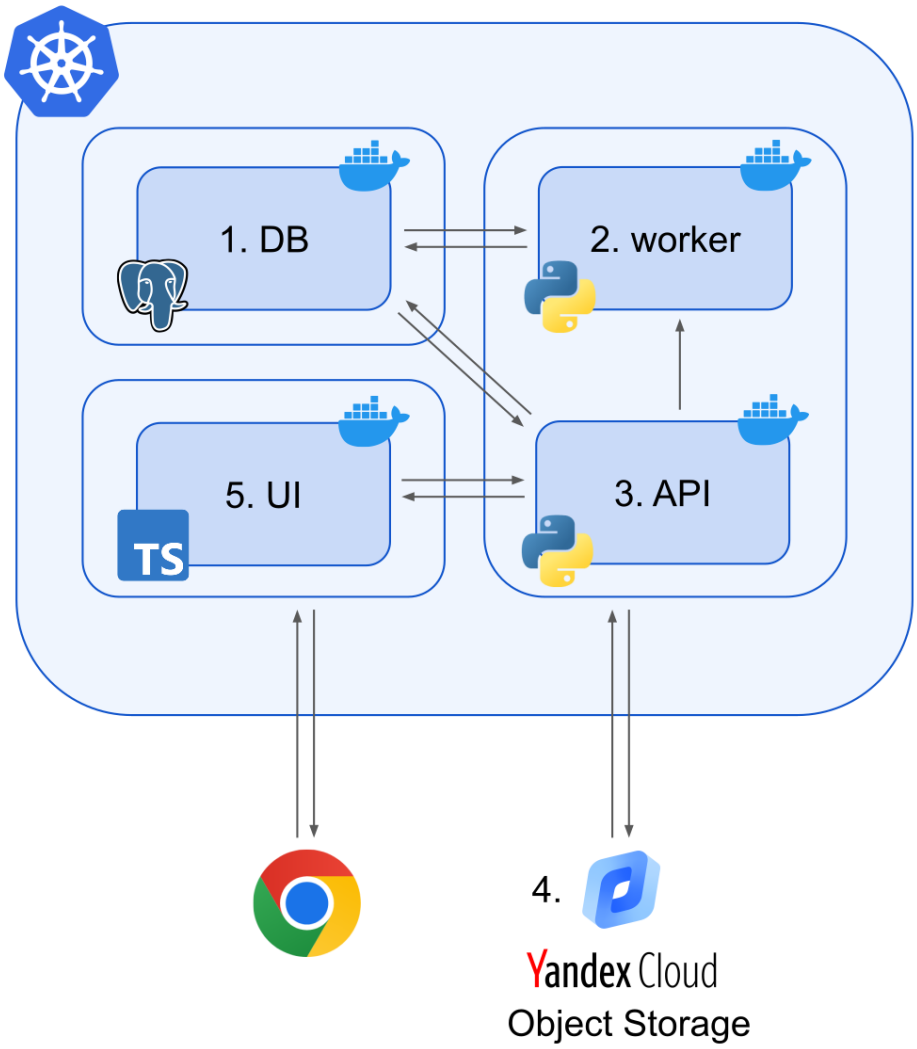

Рис. 1.1: Начальный план архитектуры для разрабатываемого сервиса.

1. На рисунке 1.1: «1. DB». Реляционная база данных (например, PostgreSQL [24]) для хранения метаинформации об изображениях (путь к файлу и разметка изображения по виду и цвету) и справочника вида овощей.

2. На рисунке 1.1: «2. worker». Вспомогательный рантайм Python для выполнения задач в фоновом режиме на базе библиотеки Celery [27] или аналогичного инструмента. В частности, для разметки загруженного пользователем изображения без задержки ответа на POST запрос.

3. На рисунке 1.1: «3. API». Сервер API на базе одного из современных асинхронных web-фреймворков на Python, обеспечивающий основной функционал взаимодействия с сервисом и вызывающий исполнение фоновых задач worker-ом.

4. На рисунке 1.1: «4. Object Storage». Внешнее облачное файловое хранилище для исходного набора фотографий и загруженных пользователями изображений. Сервис хранилища должен предоставлять удобный API для загрузки / чтения файлов. Также должна быть возможность генерировать временные ссылки для скачивания файлов, чтобы скачивание могло происходить в рантайме клиентского приложения. Под эти критерии подходит, например, совместимое с интерфейсом S3 хранилище от Yandex Cloud [13].

5. На рисунке 1.1: «5. UI». Веб-сервер упакованных статических файлов для отрисовки пользовательского интерфейса в браузере. Выполнить кодовую базу UI планируется на TypeScript [33] с использованием одного из современных SPA-фреймворков.

6. Сборка образов приложений при помощи GitHub Actions [22] с последующим хранением артефактов в GitHub Container Registry (ghcr.io) [23].

7. Автоматическое развертывания новых релизов приложений в кластере Kubernetes на личной машине автора через установку Helm-чартов [20] с помощью GitHub Actions.

8. Доступ к сервису по протоколу HTTPS через поддомен на личном домене автора.

2 Краткий обзор существующих решений задачи классификации

2.1 Пример 1. InceptionV3 transfer learning

Решение [26] с использованием архитектуры Inception V3 [43], обученной на открытом наборе данных ImageNet [41]. В качестве предобработки исходного набора изображений автор применил следующие преобразования:

- повышение насыщенности цвета
- повышение контрастности
- повышение резкости

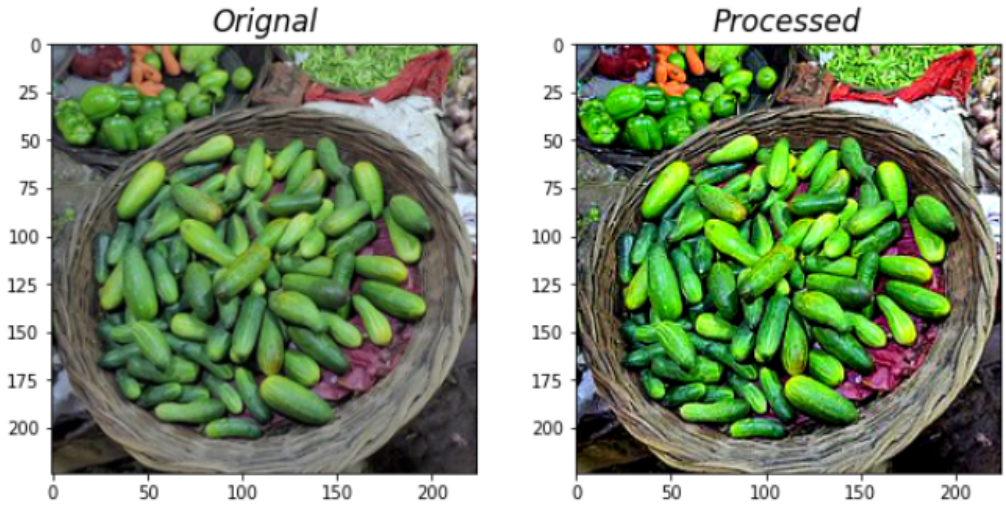

Рис. 2.1: Результат преобразований.

Поверх выходного слоя InceptionV3 были добавлены следующие слои:

- слой 2D average pooling
- полносвязный слой с функцией активации ReLU
- слой Dropout с коэффициентом 0.2
- выходной полносвязный слой для классификации с функцией Softmax

Обучение производилось в 5 эпох. В качестве функции потерь была использована категориальная кросс-энтропия. В качестве оптимизационного алгоритма - Adam. Полученная модель на наборе данных для валидации показала средний accuracy score в размере 0.992. Худшее значение среди всех классов - 0.97. Лучшее значение среди всех классов - 1.00.

2.2 Пример 2. EfficientNet-b0 transfer learning

Решение [39] с использованием модели EfficientNet-b0 [44], предобученной на ImageNet. В качестве случайных аугментаций для исходного набора данных были использованы следующие преобразования:

- горизонтальное отражение изображения
- изменение высоты изображения с коэффициентом 0.2
- изменение ширины изображения с коэффициентом 0.2
- поворот изображения с коэффициентом 0.2
  и заполнением пустоты ближайшим пикселем
- приближение / отдаление изображения с коэффициентом 0.2

Поверх выходного слоя EfficientNet-b0 были добавлены следующие слои:

- слой 2D average pooling
- выходной полносвязный слой для классификации с функцией Softmax

В качестве функции потерь была использована категориальная кросс-энтропия. В качестве оптимизационного алгоритма - Adam. Обучение производилось в 2 этапа по 5 эпох. Сначала были использованы 20% от исходного набора данных для обучения. После этого шага модель на наборе данных для валидации показала средний accuracy score в размере 0.992.

На втором этапе веса из предыдущего обучения использовались в качестве начальных весов модели, а набор данных для обучения был уже полным (100% от исходного набора данных для обучения). Архитектура самой модели относительно предыдущего этапа осталась без изменений. Результатом второго этапа обучения стал средний accuracy score в размере 0.998 на наборе данных для валидации.

2.3 Пример 3. CNN

Решение [30] с использованием стандартного подхода - построения пользовательской архитектуры сверточной нейронной сети.

Преобразования исходного набора данных и случайные аугментации не использовались. Выбранная архитектура слоев нейронной сети выглядела следующим образом:

Таблица 2.1: Архитектура сверточной нейронной сети.

Тип слоя	Размерность выхода	Количество параметров
Conv2D	(None, 150, 150, 32)	896
MaxPooling2D	(None, 75, 75, 32)	0
Conv2D	(None, 75, 75, 64)	18496
MaxPooling2D	(None, 37, 37, 64)	0
Flatten	(None, 87616)	0
Dense / Linear	(None, 128)	11214976
Dropout	(None, 128)	0
Dense / Linear	(None, 128)	16512
Dense / Linear	(None, 15)	1935

Обучение производилось в 100 эпох с условием остановки при отсутствии улучшения качества модели на протяжении 5 эпох. В качестве функции потерь была использована категориальная кросс-энтропия. В качестве оптимизационного алгоритма - Adam.

Обучение длилось 15 эпох. Полученная модель на наборе данных для тестирования показала средний accuracy score в размере 0.955.

3 Разведочный анализ данных

3.1 Примеры классов

Рис. 3.1: Примеры изображений каждого класса из набора данных для обучения.

3.2 Обзор данных

Исходный набор данных состоит из изображений 21000 изображений 15 видов овощей. Классы сбалансированы ровно, то есть на каждый вид овоща приходится всего 1400 изображений. Исходный набор данных изначально разделен на группы для обучения, тестирования и валидации (train, test, validation) в отношении 70% / 15% / 15%. Это 15000 / 3000 / 3000 изображений соответственно (1000 / 200 / 200 на каждый класс).

Ручной просмотр изображений не позволил выявить откровенные выбросы, требующие исключения из обучения. Но точно имеют место угловые кейсы, когда несколько разных видов овощей попадаются вместе на одной фотографии, что может привести к ошибкам в классификации.

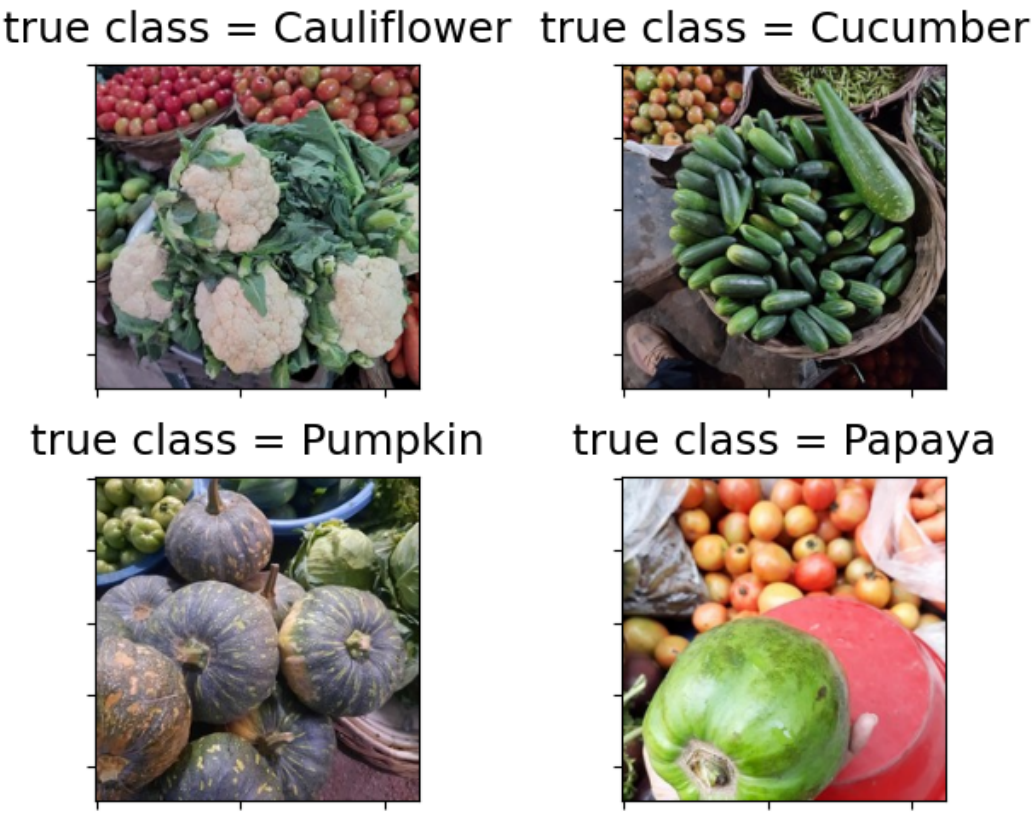

Рис. 3.2: Примеры сложных для классификации изображений.

4 Описание baseline модели

4.1 Архитектура модели

В качестве основания baseline модели использовалась модель с архитектурой Inception V3, предобученная на наборе данных ImageNet. В качестве весов использовался объект `DEFAULT` из пространства имен `torchvision.models.Inception_V3_Weights` . Перечисленные сущности скачивались из онлайн хаба pytorch/vision версии 0.10.0.

Последний слой модели (классификатор) был заменен на полносвязный слой с размерностью выхода, соответствующей количеству классов в используемом в данной работе наборе данных – 15.

Для всех изображений из набора данных было применено преобразование увеличения размера картинки до 299 на 299 пикселей, поскольку именно такую размерность модель Inception V3 ожидает на вход.

В качестве функции потерь была использована категориальная кросс-энтропия. В качестве оптимизационного алгоритма - Adam. Также использовался планировщик StepLR с понижением коэффициента скорости обучения на 0.1 каждые 5 эпох. Размер батча был равен 64.

Обучение модели производилось в бесплатном рантайме Kaggle Notebook с использованием GPU.

4.2 Результаты обучения модели

Таблица 4.1: Результаты обучение baseline модели.

Номер эпохи	Accuracy score на тестовых данных	Accuracy score на валидационных данных
1	89.13%	98.60%
2	97.38%	99.07%
3	98.29%	99.30%
4	98.61%	99.50%
5	98.94%	99.60%

5 Описание плана экспериментов с моделью

5.1 Оптимизация преобразований исходного набора данных

В рамках baseline модели использовалось только одно преобразование – увеличение размера изображений с 224^2 до 299^2. Даже такой простейший подход показал отличный результат – 99.60% accuracy score на данных для валидации.

Тем не менее в библиотеке torchvision имеется метод `transforms` из пространства имен `torchvision.models.Inception_V3_Weights.IMAGENET1K_V1` , возвращающий набор готовых преобразований для инференса на архитектуре Inception V3. Можно попробовать включить в пайплайн применение этих преобразований и оценить изменения качества предсказаний.

6 Результаты экспериментов с моделью

6.1 Оптимизация преобразований исходного набора данных

В пайплайн были добавлены рекомендованные для Inception V3 документацией библиотеки PyTorch преобразования.

Список преобразований:

- Увеличение размера изображения до 342^2
- Central crop размера (299, 299)
- Масштабирование числовых значений пикселей на шкалу [0.0, 1.0]
- Нормализация со средним [0.485, 0.456, 0.406]
  и стандартным отклонением [0.229, 0.224, 0.225]

С данными преобразованиями обучение модели показало следующий результат:

Таблица 6.1: Результаты обучение baseline модели с использованием преобразований.

Номер эпохи	Accuracy score на тестовых данных	Accuracy score на данных для валидации
1	89.57%	98.43%
2	97.57%	99.17%
3	98.17%	99.37%
4	98.56%	99.63%
5	98.71%	99.57%

Можно резюмировать, что применение к входным данным рекомендованных для данной архитектуры преобразований значимо не повлияло на качество модели.

Финальная архитектура модели и логика обучения ее обучения хранятся в формате Jupyter Notebook в контроле версий проекта [1].

7 Алгоритм определения доминантного цвета

7.1 Палитры цветов

Доминантный цвет изображений определяется по двум разным популярным цветовым палитрам отдельно (по каждой палитре свой доминантный цвет).

7.1.1 RGB

Таблица 7.1: Палитра цветов RGB.

hex	red	green	blue
#FF0000	255	0	0
#FF8000	255	128	0
#FFFF00	255	255	0
#80FF00	128	255	0
#00FF00	0	255	0
#00FF80	0	255	128
#00FFFF	0	255	255
#0080FF	0	128	255
#0000FF	0	0	255
#8000FF	128	0	255
#FF00FF	255	0	255
#FF0080	255	0	128

Рис. 7.1: Палитра цветов RGB.

7.1.2 RYB

Таблица 7.2: Палитра цветов RYB.

hex	red	green	blue
#FE2712	254	39	18
#FC600A	252	96	10
#FB9902	251	153	2
#FCCC1A	252	204	26
#FEFE33	254	254	51
#B2D732	178	215	50
#66B032	102	176	50
#347C98	52	124	152
#0247FE	2	71	254
#4424D6	68	36	214
#8601AF	134	1	175
#C21460	194	20	96

Рис. 7.2: Палитра цветов RYB.

7.2 Описание алгоритма

Алгоритм определения доминантного цвета основан на вычислении евклидова расстояния в трехмерном пространстве (reg, green, blue) между пикселями изображений и точками, соответствующими цветам в выбранной палитре.

Цвет, для которого его среднее расстояние является наименьшим среди всех цветов палитры, выбирается в качестве доминантного, и его hex код записывается в базу данных.

7.3 Имплементация алгоритма

```python
from typing import Protocol

import numpy as np
from numpy.typing import NDArray


class Base(Protocol):
    _colors: dict[str, NDArray[np.uint8]]
    # ключ - hex код цвета
    # значение - трехмерный массив, где каждая точка - координата цвета

    @classmethod
    def get_dominant_color(cls, image: NDArray[np.uint8]) -> str:
        dominant_color, min_distance = "", float("+inf")

        for color_hex, color_array in cls._colors.items():
            score = np.mean(
                np.sqrt(
                    np.sum(
                        np.square(np.subtract(image, color_array)),
                        axis=2,
                    )
                )
            )

            if score < min_distance:
                min_distance = score
                dominant_color = color_hex

        return dominant_color
```

Рис. 7.3: Имплементация базового utility-класса для вычисления доминантного цвета.

Логика алгоритма вынесена в отдельную библиотеку [2] для переиспользования:

- в среде основного проекта для разметки исходного набора данных
- в среде сервера API для разметки загружаемых пользователями изображений

8 Описание реализованной архитектуры сервиса

8.1 Организация кодовой базы проекта

В родительском репозитории [3] данной работы находится среда Python для запуска разметки исходного набора данных по доминантным цветам. Там же хранятся исходники данного текста в исходном формате LaTeX и сопутствующие артефакты.

Кодовые базы приложений сервера API и web-интерфейса подключены в родительский репозиторий в качестве подмодулей git [4].

8.2 Хранение изображений

Исходный набор данных скачан с публичного реестра Kaggle и размещен в приватном хранилище Yandex Cloud Object Storage. Данный тип хранилища имеет интерфейс, совместимый с AWS S3, поэтому для доступа к изображениям из среды приложения используется Python-библиотека boto3 [38].

В рамках сервера API реализован интерфейс [5] для чтения и записи изображений, а также для получения временной ссылки на скачивание одного изображения без авторизации.

8.3 Сериализация экземпляра обученной модели

После обучения модели ее объект сериализуется в формат, совместимый с ONNX Runtime [32], с помощью метода `torch.onnx.export` . Сериализованный экземпляр модели хранится в том же Object Storage, что используется и для хранения изображений.

8.4 Развертывание сервиса

Сервер API [6] и web-интерфейс [7] являются отдельными приложениями и, соответственно, их релизы устанавливаются отдельными Helm-чартами [8, 9] в кластер Kubernetes на личной машине автора данной работы.

Автоматизация сборки образов и развертывания / обновления релизов выполнена на базе GitHub Actions в репозиториях сервера API [10] и web-интерфейса [11]. Образы приложений хранятся в GitHub Container Registry [12].

8.5 Схема архитектуры сервиса

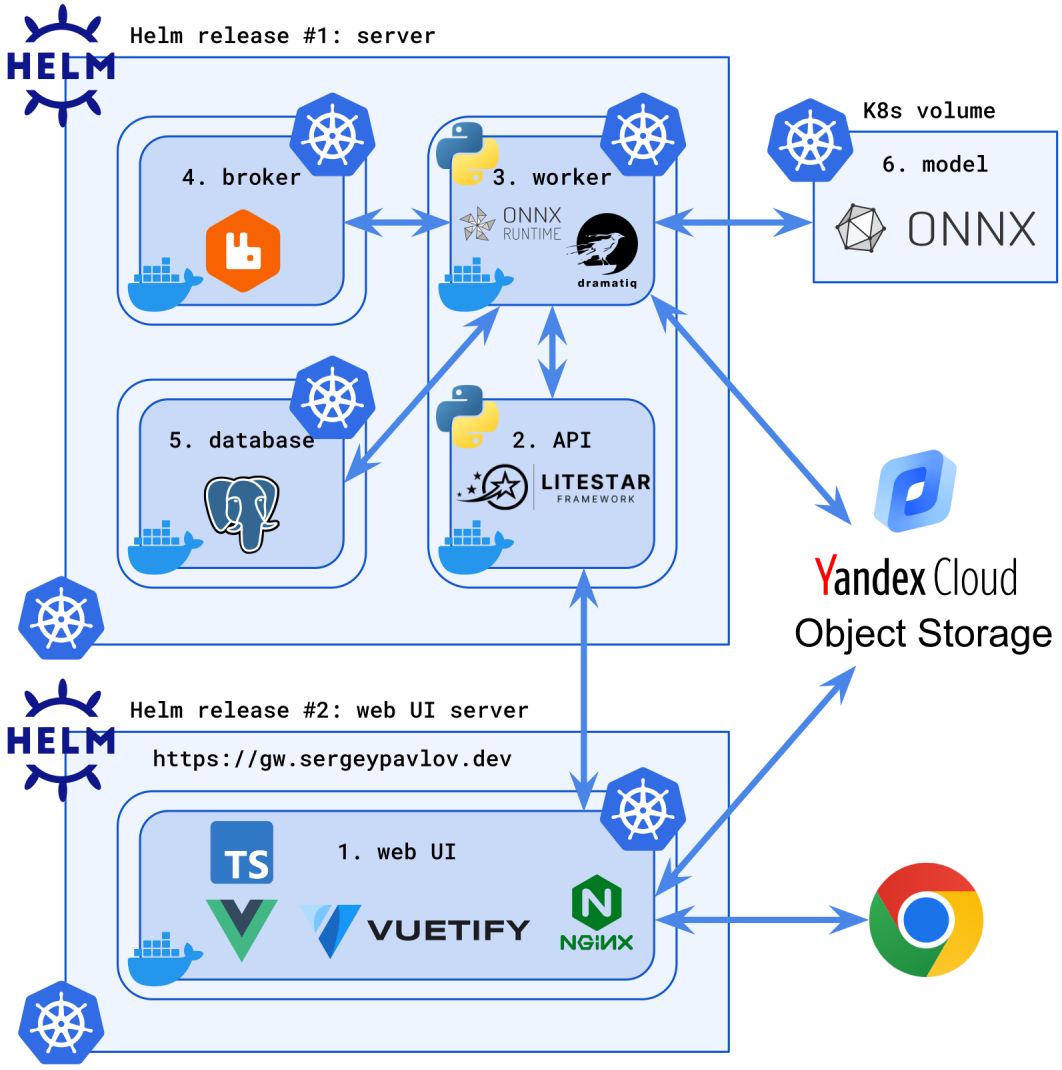

Рис. 8.1: Схема финальной архитектуры сервиса.

8.5.1 Web UI

На рисунке 8.1: «1. web UI». Для пользователя точкой входа для взаимодействия с приложением является web-интерфейс. Запросы статических файлов браузером для рендера web-интерфейса обсуживаются web-сервером NGINX [17].

Логика web-интерфейса выполнена на языке TypeScript на базе фреймворка Vue.js [46] и с использованием библиотеки готовых UI-компонентов Vuetify [45] (следует канонам Material UI).

Выбор данного стека обусловлен исключительно желанием автора данной работы попробовать frontend технологии, с которыми он ранее не имел дела в контексте основной профессиональной деятельности. Иначе можно было бы воспользоваться самым популярным

web-фреймворком для single page applications в экосистеме JavaScript, которым на текущий момент заслуженно остается [28] React [40].

Работа с Vue в данном проекте завершена с положительным впечатлением. Особо стоит отметить state management библиотеку Pinia [35]. Данное решение показалось более удобным по сравнению с аналогами из экосистемы React (например, Redux [36] или Redux Toolkit [36]).

В web-интерфейсе реализовано две вкладки (рис. 8.2) для двух основных элементов функционала:

- просмотр исходного набора данных с возможностью фильтрации по:
  - количеству выводимых изображений (от 1 до 9)
  - классу овоща
  - доминантному цвету изображения по палитре RGB или RYB
- загрузка пользовательского изображения и визуализация результата его классификации и разметки по цвету

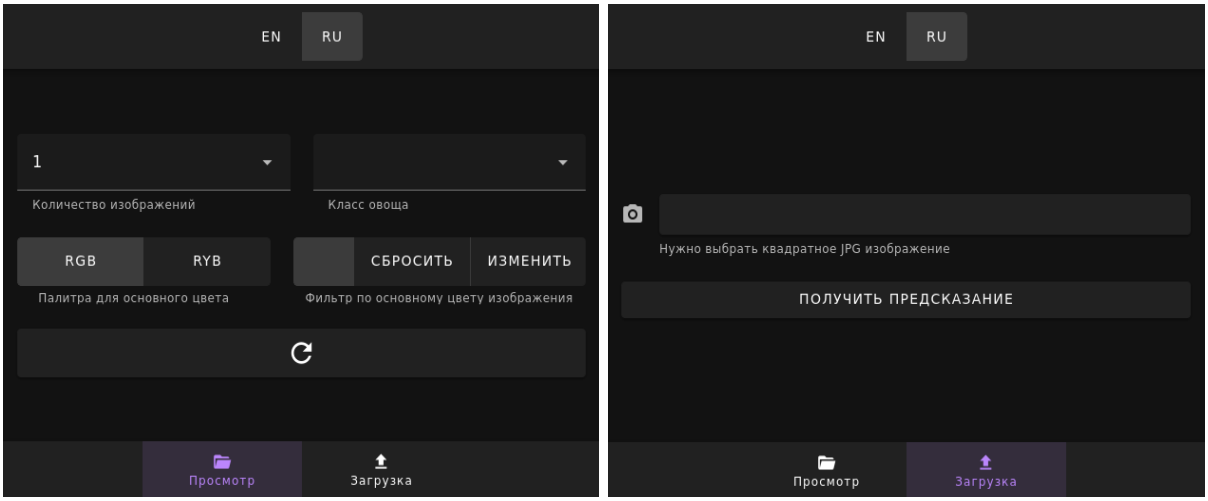

Рис. 8.2: Вкладки web-интерфейса.

8.5.2 Сервер API

На рисунке 8.1: «2. API». Действия, выполняемые пользователем в web-интерфейсе, отправляют запросы из браузера пользователя к серверу API приложения. Логика сервера API выполнена на языке Python с использованием web-фреймворка Litestar [29].

Litestar (ранее называвшийся Starlite) по перечню функционала очень похож на популярный в данный момент FastAPI [18], и изначально оба фреймворка были основаны на легковесном ASGI-совместимом фреймворке Starlette [42] (Litestar на данный момент уже освобожден от этой зависимости, что является плюсом). Также в Litestar (в отличие от FastAPI) есть поддержка class-based контроллеров, что является классическим подходом ООП к дизайну CRUD API для сущностей объектной модели приложения.

Скромные потребности данного проекта фреймворком Litestar полностью удовлетворены. Были выявлены некоторые шероховатости в части совместимости генерируемой схемы OpenAPI с поставляемым в комплекте Swagger UI, но это легко исправится в новых версиях.

Рантайм сервера API обеспечивается процессами WSGI HTTP сервера Gunicorn [25] с использованием worker класса UvicornWorker. Это необходимо для совместимости с интерфейсом ASGI, который является расширением интерфейса WSGI для web-приложений, использующих асинхронный функционал Python.

8.5.3 Процессы для выполнения фоновых задач

На рисунке 8.1: «3. worker». Контейнер с процессом для запуска фоновых задач работает как sidecar относительно контейнера с сервером API. В рамках рантайма сервера API предусмотрена отправка сигналов на выполнение двух типов фоновых задач:

1. сохранение загруженного пользователем изображения в S3-хранилище
2. разметка загруженного пользователем изображения по классу овоща по доминантному цвету, а также запись этих результатов в базу данных

Исполнение фоновых задач осуществляется процессами, управляемыми с помощью Python-библиотеки Dramatiq [16].

Выбор в пользу Dramatiq вместо мейнстримового Celery обусловлен не только желанием попробовать новое, но и промышленным опытом автора данной работы с Celery. Для простых кейсов Celery работает отлично. Но стоит углубиться, например, в пользовательскую сериализацию аргументов, передаваемых в фоновую задачу через очередь сообщений, как начинаются трудности. Не помогает и тот факт, что кодовая база Celery довольно непроста для чтения по причине тяжелейшего архитектурного наследия.

По опыту разработки в данном проекте можно резюмировать, что по крайней мере с простыми кейсами Dramatiq справляется не хуже.

Для инференса внутри фоновых задач используются сессии ONNX Runtime. Сам файл с моделью помещается в Persistent Volume [34] в процессе инициализации основного приложения и считывается процессом фоновой задачи для каждого запуска сессии инференса.

8.5.4 Менеджер очередей сообщений

На рисунке 8.1: «4. broker». Менеджер очередей сообщений RabbitMQ [31], обрабатывающий сигналы на запуск фоновых задач, поступающие из рантайма сервера API.

Здесь подходит любой брокер, чей протокол совместим с выбранной библиотекой для управления фоновыми задачами. Helm-чарт RabbitMQ работает из коробки с минимальной конфигурацией авторизации. Потребности и желания пробовать что-то новое для этого функционала не было.

8.5.5 База данных

На рисунке 8.1: «5. database». База данных PostgreSQL (схема на рис. 8.3). Хранит информацию об изображениях из исходного набора данных и о загруженных пользователем в таблице IMAGE. Есть справочная таблица CATEGORY со списком всех классов овощей, на одну строку которой ссылается каждая запись из таблицы IMAGE через `FOREIGN KEY`. Также таблица IMAGE содержит hex коды доминантных цветов изображения по палитрам RGB и RYB.

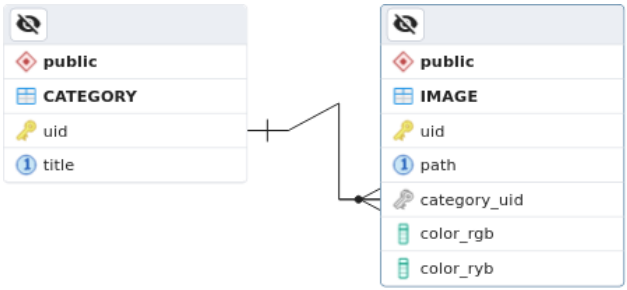

Рис. 8.3: Схема базы данных.

Данные базы хранятся в локальном Persistent Volume сервера. При удалении релиза сервера API данные сохраняются и могут быть использованы при повторной установке. Жизненный цикл данного volume управляется Helm-чартом PostgreSQL, входящим в чарт сервера API в качестве зависимости.

8.5.6 Файл с моделью

На рисунке 8.1: «6. model». Persistent Volume для хранения сериализованного экземпляра модели, упакованного в совместимый с ONNX Runtime формат.

8.6 Заключение

Рассмотренная в данной работе архитектура сервиса для классификации изображений ограничена по функционалу первоначальной постановкой задачи и представляет собой удобную базу для расширения и/или адаптации под данные из других коллекций и предметных областей.

Горизонтальное масштабирование поддерживается простыми правками конфигурации Helm-чартов:

- настройкой репликации (`replicas`) развертываний [15]
- настройкой запросов (`requests`) и ограничений (`limits`)
  на потребление ресурсов кластера [37]

На период актуальности данный сервис будет доступен для тестирования онлайн по адресу `https://gw.sergeypavlov.dev`.

Список литературы

[1] URL: https://github.com/DramatikMan/MLHL-gradwork/blob/main/notebook/model_baseline.ipynb (дата обр. 28.05.2023).

[2] URL: https://gitlab.com/DramatikMan/mlhl-gradwork-color-utils (дата обр. 28.05.2023).

[3] URL: https://github.com/DramatikMan/MLHL-gradwork (дата обр. 28.05.2023).

[4] URL: https://github.com/DramatikMan/MLHL-gradwork/tree/main/app (дата обр. 28.05.2023).

[5] URL: https://github.com/DramatikMan/MLHL-gradwork-server/blob/main/gwserver/core/s3/s3.py (дата обр. 28.05.2023).

[6] URL: https://github.com/DramatikMan/MLHL-gradwork-server (дата обр. 28.05.2023).

[7] URL: https://github.com/DramatikMan/MLHL-gradwork-web-UI (дата обр. 28.05.2023).

[8] URL: https://github.com/DramatikMan/MLHL-gradwork-server/tree/main/chart (дата обр. 28.05.2023).

[9] URL: https://github.com/DramatikMan/MLHL-gradwork-web-UI/tree/main/chart (дата обр. 28.05.2023).

[10] URL: https://github.com/DramatikMan/MLHL-gradwork-server/actions/workflows/ci.yml (дата обр. 28.05.2023).

[11] URL: https://github.com/DramatikMan/MLHL-gradwork-web-UI/actions/workflows/ci.yml (дата обр. 28.05.2023).

[12] URL: https://github.com/DramatikMan?ecosystem=container&tab=packages (дата обр. 28.05.2023).

[13] Yandex project © 2023 Intertech Services AG. *Getting started with Yandex Object Storage*. URL: https://cloud.yandex.com/en-ru/docs/storage/quickstart (дата обр. 28.05.2023).

[14] Israk Ahmed. *Vegetable Image Dataset*. URL: https://kaggle.com/datasets/misrakahmed/vegetable-image-dataset (дата обр. 28.05.2023).

[15] *Deployments | Kubernetes. Scaling a Deployment*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment (дата обр. 28.05.2023).

[16] *Dramatiq: background tasks — Dramatiq 1.14.2 documentation*. URL: https://dramatiq.io (дата обр. 28.05.2023).

[17] Inc. F5. *Installing NGINX Open Source*. URL: https://docs.nginx.com/nginx/admin-guide/installing-nginx/installing-nginx-open-source (дата обр. 28.05.2023).

[18] *FastAPI*. URL: https://fastapi.tiangolo.com/lo (дата обр. 28.05.2023).

[19] Python Software Foundation. *About Python™ | Python.org*. URL: https://python.org/about (дата обр. 28.05.2023).

[20] The Linux Foundation. *Helm | Docs*. URL: https://helm.sh/docs (дата обр. 28.05.2023).

[21] The Linux Foundation. *PyTorch documentation — PyTorch 2.0 documentation*. URL: https://pytorch.org/docs/stable/index.html (дата обр. 28.05.2023).

[22] Inc. GitHub. *Understanding GitHub Actions*. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions (дата обр. 28.05.2023).

[23] Inc. GitHub. *Working with the Docker registry*. URL: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-docker-registry (дата обр. 28.05.2023).

[24] The PostgreSQL Global Development Group. *PostgreSQL: About*. URL: https://postgresql.org/about (дата обр. 28.05.2023).

[25] *Gunicorn - Python WSGI HTTP Server for UNIX*. URL: https://gunicorn.org (дата обр. 28.05.2023).

[26] Raghav Gupta. *Vegetable Classification Using Transfer Learning*. URL: https://kaggle.com/code/theeyeschico/vegetable-classification-using-transfer-learning (дата обр. 28.05.2023).

[27] *Introduction to Celery*. URL: https://docs.celeryq.dev/en/stable/getting-started/introduction.html (дата обр. 28.05.2023).

[28] The 2022 State of JS survey. *Front-end Frameworks. Retention, interest, usage, and awareness ratio over time*. URL: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks (дата обр. 28.05.2023).

[29] *Litestar | Effortlessly Build Performant APIs*. URL: https://litestar.dev (дата обр. 28.05.2023).

[30] Chitwan Manchanda. *Vegetable Image Classification Using CNN*. URL: https://kaggle.com/code/chitwanmanchanda/vegetable-image-classification-using-cnn (дата обр. 28.05.2023).

[31] *Messaging that just works — RabbitMQ*. URL: https://rabbitmq.com (дата обр. 28.05.2023).

[32] Microsoft. *Python | onnxruntime*. URL: https://onnxruntime.ai/docs/get-started/with-python.html (дата обр. 28.05.2023).

[33] Microsoft. *TypeScript: JavaScript With Syntax For Types*. URL: https://typescriptlang.org (дата обр. 28.05.2023).

[34] *Persistent Volumes | Kubernetes*. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes (дата обр. 28.05.2023).

[35] *Pinia | The intuitive store for Vue.js*. URL: https://pinia.vuejs.org (дата обр. 28.05.2023).

[36] *React Redux | React Redux*. URL: https://react-redux.js.org (дата обр. 28.05.2023).

[37] *Resource Management for Pods and Containers | Kubernetes*. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers (дата обр. 28.05.2023).

[38] Amazon Web Services. *Boto3 documentation*. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html (дата обр. 28.05.2023).

[39] Ashwin Shetgaonkar. *Vegetable Clf | Transfer learning | Error Analysis*. URL: https://kaggle.com/code/ashwinshetgaonkar/vegetable-clf-transfer-learning-error-analysis (дата обр. 28.05.2023).

[40] Meta Open Source. *React*. URL: https://react.dev (дата обр. 28.05.2023).

[41] Princeton University Stanford Vision Lab Stanford University. *About ImageNet*. URL: https://image-net.org/about.php (дата обр. 28.05.2023).

[42] *Starlette*. URL: https://starlette.io (дата обр. 28.05.2023).

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens и Zbigniew Wojna. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[44] Mingxing Tan и Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].

[45] *Vuetify — A Vue Component Framework*. URL: https://vuetifyjs.com/en (дата обр. 28.05.2023).

[46] Evan You. *Vue.js - The Progressive JavaScript Framework | Vue.js*. URL: https://vuejs.org (дата обр. 28.05.2023).