МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Программа стратегического академического лидерства «Приоритет – 2030»

ПРОЕКТ «ЦИФРОВАЯ КАФЕДРА»

Дополнительная профессиональная программа профессиональной переподготовки

«Цифровое моделирование и суперкомпьютерные технологии»

ИТОГОВАЯ АТТЕСТАЦИОННАЯ РАБОТА (IT-ПРОЕКТ)

на тему: «Восстановление цвета фотографий при помощи средств машинного обучения»

Руководитель: Денисов. М. Д. (_____)

К защите допустить

Руководитель ДПП ПП Денисов. М. Д. (_____)

Москва 2025

СПИСОК ИСПОЛНИТЕЛЕЙ

№	Фамилия, Имя и Отчество	Группа по ООП	Название и номер раздела	Подпись
1	Жаворонков Никита Дмитриевич	М8О-210Б-23	1. ЗАДАЧА ОКРАШИВАНИЯ ЧЕРНО-БЕЛЫХ ИЗОБРАЖЕНИЙ СРЕДСТВАМИ МАШИННОГО ОБУЧЕНИЯ	
2	Абдыкалыков Нурсултан Абдыкалыкович	М8О-206Б-23	2.1.2 Разработка frontend-приложения на React	
3	Баучрин Николай Владимирович	М8О-211Б-23	2.1.4 Очередь сообщений RabbitMQ и обработка задач с помощью Celery 2.2.2 RabbitMQ 2.2.3 AMQP 2.2.1 Celery	
4	Воронухин Никита Александрович	М8О-210Б-23	2.2.10 MSE – Mean Squared Error 2.2.9 Tensorflow 2.2.8 Формат lab 2.2.7 U-Net 2.2.5 MinIO	
5	Дубровина Софья Андреевна	М8О-207Б-23	2.2.4 FastAPI 2.3.5 Докеризация компонентов в проекте «Восстановление цвета фотографий при помощи средств машинного обучения»	
6	Жиденко Виктория Александровна	М8О-207Б-23	2.3.1. Задачи API 2.1.8. Сохранение обработанных изображений в MinIO	
7	Караткевич Николай	М8О-214Б-23		

	Сергеевич		2.1.8. Сохранение обработанных изображений в MinIO 3.1 Развёртывание программного средства	
8	Люзина Мария Николаевна	М8О-207Б-23	2.1.1 Макет интерфейса (Frontend) 2.3.4 Пользовательский интерфейс	
9	Салихов Руслан Ринатович	М8О-203Б-23	2.2.6 PostgreSQL 2.1.5. Хранение данных в MinIO и PostgreSQL 2.1.7. Конвертация изображений и сохранение в формате .PNG	

РЕФЕРАТ

Итоговая аттестационная работа состоит из 191 страниц, 17 рисунков, 5 таблиц, 33 использованных источников, 2 приложений.

Итоговая аттестационная работа выполнена в формате IT-проекта «Восстановление цвета фотографий при помощи средств машинного обучения».

Объектом разработки в данной работе является сервис «Has-Been» восстановления цветовой информации черно-белых изображений средствами алгоритмов машинного обучения в сети «Интернет».

Цель работы — предоставить целевой аудитории без профильного образования возможность удобной окраски произвольного количества изображений.

Основное содержание работы состояло в разработке:

- Сервера обработки запросов и предоставления доступа к процедурам обработки изображений на основе FastAPI
- Веб-приложения с использованием технологии React
- U-Net сверточной нейронной сети для окрашивания черно-белых изображений

Основными результатами работы, полученными в процессе разработки, являются технологическое решение, включающее набор различных методик и принципов построения, обеспечивающих надежность и масштабируемость. В составе решения — серверная часть на FastAPI, клиентская на React и сверточная нейросеть U-Net для восстановления цветовой информации.

Данные результаты предназначены для применения в области автоматической цветокоррекции черно-белых изображений в сети «Интернет» и могут быть адаптированы для широкого круга организаций с минимальными доработками.

Использование результатов данной работы позволяет повысить качество и оперативность обработки изображений, расширить доступ к современным методам машинного обучения для пользователей без профильного образования, а также оптимизировать процессы восстановления цветовой информации.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей итоговой аттестационной работе применяют следующие термины с соответствующими определениями:

1. **Алгоритм** — последовательность шагов, предназначенная для выполнения конкретной задачи или решения проблемы.
2. **API (Application Programming Interface)** — интерфейс программирования, который позволяет различным программам взаимодействовать между собой.
3. **Брокер сообщений** — программное обеспечение, которое отвечает за передачу сообщений между различными компонентами системы.
4. **Гибкость системы** — способность системы адаптироваться к изменениям и расширениям без значительных изменений её архитектуры.
5. **Микросервисная архитектура** — подход к проектированию системы, где каждый компонент (сервис) выполняет одну конкретную функцию и взаимодействует с другими сервисами через стандартные интерфейсы.
6. **Многозадачность** — способность системы одновременно выполнять несколько задач или процессов.
7. **Масштабируемость** — способность системы эффективно увеличивать ресурсы для обработки большего объема данных или нагрузки.
8. **Нейронная сеть** — модель, вдохновленная работой мозга человека, предназначенная для решения задач машинного обучения.
9. **Контейнеризация** — технология, позволяющая запускать приложения и их зависимости в изолированных контейнерах, которые могут быть перенесены между разными средами.
10. **Реактивное программирование** — парадигма программирования, в

которой компоненты системы реагируют на события или изменения состояния.

11. **REST (Representational State Transfer)** — архитектурный стиль, использующий HTTP для обмена данными между клиентом и сервером.
12. **SQL (Structured Query Language)** — язык запросов, предназначенный для работы с реляционными базами данных.
13. **Отказоустойчивость** — способность системы продолжать работать, несмотря на сбои в её компонентах.
14. **Распределенная система** — система, состоящая из нескольких взаимосвязанных компонентов, которые работают на разных физических или виртуальных машинах.
15. **Тензор** — многомерный массив данных, используемый в машинном обучении и нейронных сетях.
16. **Фреймворк** — набор библиотек и инструментов, облегчающих разработку приложений.
17. **Цепочка поставок данных** — последовательность этапов, через которые данные проходят от их создания до использования.
18. **Параллелизм** — возможность выполнения нескольких операций одновременно для ускорения вычислений.
19. **Функциональное программирование** — парадигма программирования, в которой функции являются первичными объектами, а вычисления выполняются через композицию функций.
20. **CI/CD (Continuous Integration/Continuous Deployment)** — практики, обеспечивающие автоматическую интеграцию и развертывание изменений в коде.

21. **OpenCV** — библиотека с открытым исходным кодом для обработки изображений и видео.
22. **Мониторинг** — процесс отслеживания состояния системы для выявления и устранения неисправностей.
23. **Серверная**ess **архитектура** — архитектура, при которой серверная инфраструктура не управляется пользователем, а предоставляется в виде облачных сервисов.
24. **Блокчейн** — распределенная база данных, обеспечивающая надежную и прозрачную запись всех транзакций в системе.
25. **Синхронность** — выполнение операций в определенной последовательности, где одна операция начинается только после завершения предыдущей.

ВВЕДЕНИЕ

Актуальность темы данной работы связана с быстрым ростом рынка инструментов для автоматической генерации и восстановления изображений, объём которого в сегменте AI-генераторов достиг USD 349.6 млн в 2023 г. и прогнозируется к росту на CAGR 17.7 % до 2030 г. [1][8]. Современные методы на основе глубоких нейронных сетей, такие как U-Net, позволяют за счёт многомасштабного объединения признаков значительно повысить качество цветизации по сравнению с традиционными ручными и простыми алгоритмическими подходами [2][9]. Несмотря на успехи, существующие архитектуры остаются частично недоработанными в части воспроизведения реалистичных цветовых границ и учёта глобального контекста изображения, что подчёркивают подробные сравнительные исследования и обзоры современных техник цветизации [4][6]. Популярность U-Net объясняется её способностью эффективно объединять локальные и глобальные признаки, однако для задач реставрации исторических фотографий выявлены ограничения стандартной архитектуры, в частности недостаточная адаптация к особенностям старых изображений без специализированных Fusion Layers и механизмов внимания [3][5][10]. В связи с этим обоснована необходимость разработки и экспериментальной валидации модификаций U-Net, включающих модули внимания и объединяющие слои, что позволит повысить уровень автоматизации и качество цветизации в рамках реальных IT-решений [7][8].

Таким образом, выполненная работа актуальна и с научно-методической/теоретической, и с практической точек зрения.

Цель работы — предоставить целевой аудитории без профильного образования возможность удобной окраски произвольного количества изображений с помощью средств машинного обучения.

Для достижения поставленной цели в работе были решены следующие задачи:

1. формулировка задачи, связанной с пониманием проблемы, которую решает проект;

2. формулировка задачи, связанной с обоснованием выбора стека технологий;

3. формулировка задачи, связанной с разработкой IT-решения;

4. формулировка задачи, связанной с апробацией и тестированием разработанного IT-решения.

Объектом разработки является сервис «Has-Been» восстановления цветовой информации черно-белых изображений средствами алгоритмов машинного обучения в сети «Интернет».

Предметом исследования в работе является область автоматической цветокоррекции изображений с использованием технологий FastAPI, React и сверточной нейросети U-Net.

Работа основывалась на следующих инструментах и методах:

- методах машинного обучения для восстановления цвета;

- архитектуре микросервисов на базе FastAPI для обработки запросов;

- фронтенд-разработке с использованием React для удобного пользовательского интерфейса;

- сверточной нейронной сети U-Net для решения задачи раскраски изображений.

Эти технологии были выбраны за их гибкость, масштабируемость и проверенную эффективность.

Основными результатами, полученными в работе, являются:

1. четкая постановка и анализ проблемы автоматического восстановления цвета изображений;

2. обоснование и выбор современного стека технологий для реализации проекта;

3. разработка работающего IT-решения, включающего серверную и клиентскую части;

4. успешное тестирование и апробация сервиса на реальных данных.

Результаты работы предназначены для использования в области цифровой обработки изображений, в базовой организации и за её пределами, благодаря масштабируемой архитектуре и простоте интеграции.

Использование разработки позволяет значительно ускорить процесс раскраски черно-белых фотографий, повысить доступность современных методов машинного обучения для конечного пользователя и автоматизировать обработку изображений в сети «Интернет».

1 ЗАДАЧА ОКРАШИВАНИЯ ЧЕРНО-БЕЛЫХ ИЗОБРАЖЕНИЙ СРЕДСТВАМИ МАШИННОГО ОБУЧЕНИЯ

1.1 Потребность в решении

В течение нескольких десятилетий задачи придания естественных цветов чёрно-белым изображениям развивались от ручных техник к полностью автоматизированным цифровым системам. Ещё в XIX веке мастера вручную раскрашивали отдельные фрагменты фотографий, выдерживая многочисленные коррекции и исследования исторической достоверности оттенков. Сложность этого процесса заключалась в отсутствии стандартизированных методик: колористы ориентировались исключительно на собственные представления о цвете тканей, кожи и окружающей среды, что приводило к значительным расхождениям от реальной палитры сцены.

Вторая половина XX века стала периодом активного внедрения фотохимических методов, основанных на принципах селективной тонировки и многослойной печати, однако они также требовали существенных ручных доработок. В 1970–1980-е годы появились первые цифровые алгоритмы, использовавшие принципы теории ретинекса и корректировки уровня освещённости, что дало возможность автоматически выравнивать тональные характеристики изображения [1]–[2]. Тем не менее, отсутствие надёжных методов выделения однородных регионов и неучёт глобального контекста часто приводили к «выцветшим» переходам и искажённым границам между элементами сцены.

С развитием вычислительной техники и появлением алгоритмов сегментации на основе гистограмм и растрового сканирования (1980–1990-е годы) стали доступны методы автоматической передачи цветовых тонов с эталонных цветных фотографий на чёрно-белые изображения. Концепция цветового переноса, предложенная Welsh et al. в начале 2000-х, заключалась в статистическом сопоставлении распределений цветовых компонентов между

парами изображений и последующем переназначении каналов яркости и насыщенности [3]. Такие алгоритмы позволили упростить процесс, однако им требовались вручную подобранные эталонные изображения, близкие по содержанию к исходным чёрно-белым кадрам.

Дальнейшие исследования в области классической цветизации опирались на методы оптимизации, основанные на Марковских случайных полях и минимизации глобального энергетического функционала. Так, в 2004 году Левин и коллеги предложили метод, при котором пользователь отмечал «кистями» фрагменты изображения, указывая желаемые цвета, а алгоритм распространял их по всему кадру, сохраняя плавность переходов и учитывая сходство текстурных паттернов [4]. Подобный подход продемонстрировал высокую детализацию цветовых границ и значительную экономию ручного труда, но всё ещё требовал вмешательства человека для инициализации цветовых меток.

К концу 2000-х годов появились первые машинно-обучаемые решения без глубинных нейронных сетей. Использовались методы случайного леса и регрессии, обученные на наборах «чёрно-белое—цветное», которые прогнозировали цвет каждого пикселя на основании локальных признаков (градиентов, текстур, контекста вокруг пикселя) [5]. Впрочем, качество таких предсказаний ограничивалось объёмом размеченных данных и неспособностью алгоритмов обобщать информацию о сложных структурах изображения.

Рубежным событием в развитии автоматической цветизации стало появление архитектуры U-Net, изначально разработанной для сегментации медицинских изображений. В 2015 году Ronneberger и соавторы продемонстрировали, что U-Net с пропускающими соединениями обеспечивает эффективное объединение локальной и глобальной информации, что идеально подходит для задач восстановления и преобразования изображений [6]. Стандартная модель U-Net быстро завоевала популярность в сообществах исследователей компьютерного зрения и была адаптирована для цветизации

чёрно-белых фотографий без дополнительных рукописных меток.

Вскоре после этого появилось несколько вариаций, включавших условные генеративные состязательные сети (cGAN), а также модули внимания, способные фокусироваться на ключевых областях изображения и улучшать воспроизведение сложных цветовых оттенков. Так, в 2016 году Iizuka et al. предложили двухветвевую нейросеть с глобальным и локальным путями, которая учитывает как общую цветовую гамму, так и мелкие детали объектов [7]. Параллельно группа Zhang представила метод, основанный на перепредставлении задачи цветизации как классификации цветовых тонов в пространстве Lab, что позволило более точно моделировать насыщенность и оттенок цветов [8].

По мере совершенствования нейросетевых архитектур снизились требования к предварительной разметке и ручному вмешательству; облачные платформы и публичные API начали предлагать услуги «одного клика» для мгновенной цветизации. Одним из наиболее заметных проектов стал DeOldify, использующий сочетание U-Net и GAN для улучшения резкости и насыщенности результатов, а также техники непрерывного обучения на различных исторических наборах данных [9].

Нарастающая популярность сервисов автоматической цветизации вызвала интерес к разработке универсальных моделей. В частности, методы самообучения и дообучения на пользовательских примерах позволили сервисам адаптироваться к специфике фотографий, сохраняя при этом высокую скорость обработки и минимальные затраты ресурсов. Исследования последних лет показывают, что комбинирование классических алгоритмов цветового переноса с нейросетевыми подходами даёт синергетический эффект: локальная точность сочетается с глобальной согласованностью оттенков [10].

В настоящее время автоматическая цветизация является отдельной отраслью цифрового анализа изображений и активно используется в реставрации исторических архивов, фильмов, газетных и журнальных макетов.

Технологические решения включают как открытые библиотеки и исходные коды, так и коммерческие API, интегрируемые в мобильные и веб-приложения. Основными направлениями дальнейшего развития остаются повышение реализма цветовых переходов, учет физических свойств материалов и освещения, а также обеспечение исторической достоверности при минимуме ручных корректировок.

Современные сервисы по автоматической окраске чёрно-белых изображений активно развиваются и могут быть классифицированы по следующим критериям: доступность API, наличие встроенных фильтров, степень интерактивности и возможности пакетной обработки. Среди наиболее популярных решений на сегодняшний день можно выделить:

MyHeritage In Color™ – облачный сервис, специализирующийся на семейных и генеалогических фотографиях, использующий гибрид глубокого обучения и алгоритмических методов передачи цвета.
Palette.fm – веб-платформа с возможностью выбора и настройки цветовых стилей, реализована на основе U-Net и GAN-моделей.
DeepAI Colorization API – открытое API с простой интеграцией, обеспечивает базовую автоматическую цветизацию с опцией дообучения на пользовательских данных.
ColouriseSG – правительственный проект, доступный бесплатно, оптимизированный для фотографий с минимальным уровнем шума и артефактов.
Algorithmia Colorize Photos – коммерческий сервис с гибким тарифным планом и возможностью пакетной обработки высокоразрешённых изображений.
Hotpot.ai – универсальный творческий инструмент, предлагающий цветизацию, реставрацию и добавление эффектов на основе глубокого обучения.
ImageColorizer – мобильное приложение и веб-сервис, специализирующееся на ускоренной пакетной обработке; поддерживает локальное и облачное вычисление.

Canva Colorize – встроенный в графический редактор модуль быстрого преображения старых фото, ориентированный на непрофессиональных пользователей.

1.2 Современные подходы к решению проблемы

В первой декаде XXI века исследователи констатировали ограниченность классических алгоритмов цветового переноса и ручных методов разметки, что стимулировало поиск полностью автоматических решений. Одним из первых заметных трудов стал «Deep Colorization» (2016), в котором предложена полностью автоматическая нейросетевая модель на основе глубоких сверточных сетей, обуславливающая высокое качество цветизации без вмешательства оператора за счёт обучения на обширных базах цветных изображений [11]. Авторы продемонстрировали, что применение совместного билинейного сглаживания и кластеризации изображений позволяет преодолеть артефакты «лоскутного» переноса цвета и добиться более равномерного распределения оттенков [11]. Вместе с тем критика данного подхода указывает на значительную вычислительную сложность пост-обработки и зависимость качества результата от содержательного состава справочной базы, что ограничивает применимость метода в условиях скудных ресурсных сред.

Параллельно в том же 2016 году был выдвинут алгоритм «Let there be Color!», в котором предложена двухветвная архитектура с модулями глобального и локального восприятия, объединяемыми на уровне «Fusion Layer». Разработчики обосновали целесообразность интеграции глобальных сценических признаков, получаемых из сети классификации сцен, и локальных паттернов, что позволило существенно повысить детализацию цветовых границ при автоматическом формировании палитры изображений [12]. Независимый аудит выявил, что при недостаточной однородности исходного кадра и сложных сочетаниях текстур модель способна генерировать цветовые «блики»

на границах объектов, понижая достоверность цветозадачи [12]. Кроме того, внедрение глобальных признаков потребовало дополнительной разметки обучающей выборки по категориям, что усложнило процедуру подготовки данных.

В активной фазе исследований особое место занял труд «Colorful Image Colorization» (ECCV 2016), в котором задача цветизации была интерпретирована как многоклассовая классификация оттенков в пространстве Lab. Устремлённость авторов к учёту неопределённости задачи подтолкнула их к применению сбалансированной схемы обучения для повышения разнообразия окраски и к реализации предсказания в виде единственного прямого прохода свёрточной сети [13]. Оценка «Turing Test» продемонстрировала, что порядка трети оценок участников не позволяет достоверно отличить истинные цвета от сгенерированных, что стало важным рубежом в области [13]. Однако последующие эксперименты выявили, что при среднем разрешении кадров (512×512 px) наблюдается тенденция к «размытию» мелких элементов, что вызвало вопросы о пригодности метода для профессиональной реставрации.

В 2017 году появились решения, интегрирующие генеративно-состязательные сети. Так, система «Real-Time User-Guided Image Colorization» позволила пользователю задавать точечные цветовые метки, которые сеть распространяла по изображению с учётом семантического контекста, обеспечивая баланс между свободной работой оператора и автоматизацией процессов [14]. Достоверность цветовых очертаний и высокая отзывчивость модели в интерактивном режиме послужили основанием для широкого внедрения метода в графических приложениях. Вместе с тем формулируется критика в части необходимости обучения на больших, специализированных наборах данных и ограничений по мобильной реализации из-за существенных вычислительных затрат [4].

Особое направление исследований связано с созданием методов цветизации при крайне скудной справочной информации. В 2017 году

предложен алгоритм «cGAN-based Manga Colorization», обеспечивающий цветовую трансформацию японской манги на базе условных GAN с единичным эталоном для обучения [15]. Данный подход позволил обойти проблему отсутствия градаций серого в манге, однако качество окраски оказалось чувствительным к референсному образцу и склонно к появлению «шумовых» фрагментов при несогласованности структуры тестового изображения и референса [15].

С дальнейшим прогрессом вычислительных мощностей и архитектур глубокого обучения в 2021 году разработан «Colorization Transformer», использующий самовнимание для генерации первоначальной раскачки цветового диапазона и последующего апсемплинга до высокого разрешения. Метод продемонстрировал способность генерировать разнообразные по стилистике окраски кадры, оцениваемые людьми даже предпочтительнее оригинальных по критериям эстетики [16]. В то же время отмечаются трудности алгоритма при работе с шумными архивными материалами, а также значительная задержка ответа при условии ограничения по оперативной памяти, что затрудняет применение в реальном времени.

Интеграция описанных подходов привела к появлению прикладных решений и облачных сервисов «one-click». Наиболее известным примером является проект DeOldify, сочетающий U-Net и GAN-модули, дополнительно использующие технику непрерывного обучения на исторических наборах данных и обеспечивающие более насыщенный цвет при сохранении структурной чёткости [13][18]. Анализ кода и эмпирический опыт применения свидетельствуют о том, что DeOldify относится к наиболее продвинутым в плане сочетания автоматизма и качества, но испытывает сложности при цветизации кадров с выраженными артефактами старых плёнок.

Рассмотренные облачные API, такие как DeepAI Colorization API и Palette.fm, демонстрируют гибкость интеграции и упрощённый интерфейс для разработчиков за счёт предоставления REST-эндпоинтов и настраиваемых

«стилей» окраски [17][19]. Тем не менее, экспертиза специалистов выявила, что при пакетной обработке высокоразрешённых фотографий сервисы нередко прибегают к понижению качества исходных данных и к агрессивному «сглаживанию» текстур, что снижает ценность их результатов для профессиональной реставрации.

Таким образом, критический анализ показывает, что несмотря на впечатляющие достижения в области нейросетевой цветизации, остаются нерешёнными задачи учёта исторической достоверности, адаптивной работы с шумными источниками и оптимизации вычислительных затрат без потери качества. Перспективными направлениями остаются гибридные схемы, объединяющие классические цветовые переносы с современными механизмами внимания, а также контекстно-ориентированные трансформеры с учётом физических свойств объектов и освещения.

1.3 Сравнение с современными аналогами:

В этом разделе мы рассматриваем основные существующие решения и сервисы для автоматической раскраски изображений, сравниваем их сильные и слабые стороны с точки зрения качества результата, скорости обработки, удобства интеграции и стоимости.

1.3.1 Критерии сравнения

1. **Качество раскраски**

   – Насколько естественно выглядят цвета, отсутствие артефактов и «заливок».

2. **Разрешение и детализация**

   – Максимальное поддерживаемое разрешение, сохранение текстур и мелких деталей.

3. **Скорость и масштабируемость**

   – Время обработки одного изображения, возможность пакетной привязки и параллелизации.

4. **API и интеграция**

– Наличие public API, документация, SDK для популярных языков (Python, JavaScript и др.).

5. **Ценообразование**

   – Бесплатный тариф, оплата по запросам, подписка, ограничения на размер/кол-во изображений.

6. **Дополнительные функции**

   – Ручная доработка, семантические подсказки, ретушь, пред– и постобработка (шумоподавление, коррекция контраста).

Сервис	Алгоритм	Разрешение	API / SDK	Время обработки	Стоимость	Особенности
DeOldify	GAN + Perceptual loss	До 1024×1024	Python-скрипт (open-source)	≈5–30 с на GPU	Бесплатно (самостоятельный хостинг)	Высокое качество, требует GPU
Colourise.sg	CNN	До 512×512	REST API	≈2–5 с	Бесплатно, но нет гарантии SLA	Простой API, ограниченный набор настроек
Algorithmia Colorize	GAN	До 1024×1024	REST API	≈3–10 с	$0.001/ запрос	Поддержка JavaScript, Python, Java
DeepAI Colorization	CNN + Fine-tuning	До 2048×2048	REST API	≈1–3 с	0.01 $ / изображение	Быстрая обработка, скромное качество
MyHeritage In Color™	Специализированный GAN	До 800×800	Веб-интерфейс API	≈10–20 с	Входит в подписку MyHerita	Оптимизирован для портретов

					ge	старых фото
Let’s Enhance	CNN + Transformer	до 4096×4096	REST API	≈5–15 с	от $0 (ограничено) до $9.99/мес	Пакетная обработка, улучшение резкости
Hotpot.ai	Diffusion + GAN	До 1024×1024	Веб-интерфейс, API	≈10–25 с	Freemium (до 5 обработок в день)	Семантические подсказки, моб. приложение
Adobe Photoshop Neural Filters	U-Net + Transformer	Зависит от лицензии	Плагин Photoshop	≈5–30 с (локально)	По подписке Creative Cloud	Глубокая ручная доработка, гибкие настройки

Таблица 1-Сравнение аналогов

1.3.2. Детальный разбор

3.1 DeOldify

- **Алгоритм**: сочетание генеративных состязательных сетей и перцептуальных функций потерь (VGG-loss).
- **Плюсы**:
    1. Очень естественные, «художественные» цветовые решения.
    2. Open-source: можно модифицировать под свои задачи.
- **Минусы**:
    1. Требует выделенного GPU для быстрой работы.
    2. Сложность интеграции и деплоя (Docker, TorchServe и т. д.).

3.2 Colourise.sg

- **Алгоритм**: простая сверточная сеть, обученная на азиатском датасете.
- **Плюсы**:
    1. Бесплатный и быстрый — 2–5 с на запрос.
    2. Простой REST-интерфейс.

- **Минусы**:
    1. Ограниченное разрешение (512×512).
    2. Низкая гибкость настроек: нет контроля над палитрой и семантикой.

3.3 Algorithmia Colorize

- **Алгоритм**: улучшенная GAN-архитектура с перцептуальным и TV-loss.
- **Плюсы**:
    1. Хороший компромисс «качество–скорость».
    2. Пакетный режим, поддержка языков через SDK.
    3. Прозрачное ценообразование.
- **Минусы**:
    1. Платный – оплата за каждое изображение.
    2. Не всегда аккуратно обрабатывает сложные текстуры (пряжа, траву).

3.4 DeepAI Colorization

- **Алгоритм**: быстрое CNN-решение с fine-tuning на разнообразных датасетах.
- **Плюсы**:
    1. Очень быстрое время ответа (1–3 с).
    2. Лёгкая интеграция через REST.
- **Минусы**:
    1. Качество иногда «плоское», цвета приглушённые.
    2. Меньше контроля над результатом, артефакты по краям.

---

4. Выводы и рекомендации

1. **Для прототипа или академической работы** отлично подойдёт **DeOldify** — максимальная гибкость и качество, но понадобится собственная инфраструктура.
2. **Для быстрого MVP-веб-сервиса** с минимальным бюджетом стоит рассмотреть **DeepAI** или **Colourise.sg**. Они легко подключаются, а первые

десятки запросов в день бесплатны.

3. **Для коммерческого продакшена** лучше обратить внимание на **Algorithmia** или **Let's Enhance**: более стабильные SLA, приятная цена и возможность пакетной обработки.
4. **Уникальные кейсы** (старые семейные фотографии, портреты) можно тестировать на **MyHeritage In Color™** и **Adobe Neural Filters**, там имеются дополнительные алгоритмы под конкретные жанры.

1.4 Обоснование и техническое задание на разработку сервиса

Цель настоящей работы — создание автоматизированного сервиса цветовой реконструкции чёрно-белых изображений в сети «Интернет» с использованием компактной сверточной нейросетевой модели U-Net, обученной на фрагментах размером 128×128 пикселей в Lab-пространстве. После получения предсказаний по каналам a и b производится масштабирование цветовой информации до исходного разрешения и формирование итогового цветного изображения в формате PNG. Выбор данной цели и критерии её достижения обусловлены следующими факторами:

- **Практическая востребованность**: цифровая реставрация архивных и семейных фотографий остаётся актуальной задачей в культурной и коммерческой сферах.

- **Техническая осуществимость**: архитектура U-Net демонстрирует приемлемое качество при малых объёмах вычислений и занимает не более 4 ГБ видеопамяти при инференсе, что позволяет ее использовать в условиях высокой нагрузки серверов.

- **Стандартизация вывода**: отказ от множества выходных форматов упрощает структуру кода, архитектуру нейронной сети и оптимизирует ее размер, облегчает поддержку и разработку программного комплекса, уменьшает количество зависимостей и библиотек, избавляет от

необходимости обучения программистов новым методам и способам разработки программного обеспечения.

Для достижения поставленной цели необходимо решить следующие задачи:

- Разработать программный метод преобразования входного изображения формата RGB в Lab-пространство с последующим выделением необходимых каналов.
- Разработать и интегрировать предобученную модель U-Net, принимающую на вход фрагменты 128×128 (L-канал) и выдающую каналы a, b.
- Реализовать алгоритм масштабирования цветовых каналов предсказанного нейронной сетью изображения до исходного разрешения с корректным наложением на L-канал.
- Обеспечить сохранение результатов в формате PNG в специально выделенном под эти задачи S3-совместимым хранилищем.
- Разработать удобный пользовательский интерфейс и реализовать его на React для загрузки пользовательских изображений, отслеживания состояния их обработки и выгрузки на доступный пользователю носитель.
- Настроить FastAPI backend-службу с поддержкой асинхронной очереди задач (Celery + RabbitMQ), адаптировать решение под использование в Docker-контейнере.
- Организовать хранение и отслеживание текущих и выполненных заданий в PostgreSQL, а сами изображения — в S3-совместимом хранилище MinIO.
- Обеспечить простоту развертывания через Docker-контейнеры и проксирование через Nginx.

- Спроектировать систему с возможностью горизонтального масштабирования в кластерной или облачной среде.

Обоснование выбора задач основано на анализе публикаций в области автоматической цветизации изображений и современного опыта в разработке сервисов массового потребления цифрового контента.

Практическое основание определяется:

- **Необходимостью перевода бумажного фонда** музеев, библиотек и частных коллекций в электронный вид с возможной придачей изображениям современного вида.
- **Ограниченностью ресурсов** у конечных пользователей, в частности не только материальных, но и временных: восстановление цвета множества изображений не должно сопровождаться большими временными затратами.
- **Успешным опытом** применения технологий сверточных нейронных сетей U-Net и GAN в окрашивании исторических черно-белых фотографий (см. J. Zhang et al., 2020; P. Isola et al., 2017).
- **Коммерческим интересом** к сервисам мгновенной реставрации, окрашивания, ретуши, предлагающим минимальный порог вхождения, обуславливающийся интуитивно понятным интерфейсом.

Программное средство предоставляет следующие функции:

- **Преобразование**: приём пользовательского изображения в распространенных форматах, таких как: JPEG, PNG или TIFF и их последующее преобразование в Lab-пространство.
- **Инференс**: передача фрагментов L-канала преобразованного изображения в сверточную модель U-Net и получение a, b-каналов

окрашенного изображения.

- **Сборка**: масштабирование цветовых каналов a и b, и последующее наложение цветовых каналов на исходный L-канал для получения итогового изображения без потери деталей.

- **Экспорт**: сохранение преобразованного изображения файла в распространенном формате PNG в S3-совместимом хранилище и предоставление пользователю доступа к нему.

- **Управление задачами**: создание, отслеживание статуса и завершение задач посредством пользовательского web-интерфейса или соответствующего REST API.

Сервис является компонентом распределённого решения для цветовой реконструкции изображений. Он может быть интегрирован:

- в **клиентские** приложения, использующие REST API и связывающиеся с удаленными серверами API,

- в **архивные системы**, для автоматизированной пакетной обработки множества исходников,

- в **образовательные платформы** для демонстрации работы нейронных сетей и основополагающих принципов машинного обучения.

Каждый пользователь без профильного образования и особых познаний в сфере машинного обучения обладает возможностью через React-интерфейс или автоматизированный исполняемый сценарий загружать изображения на удаленные сервера API, отслеживать прогресс обработки и загружать обработанный нейронной сетью PNG-файл.

Требования к программному комплексу:

1. **Архитектурные компоненты**

    1. FastAPI[21] - веб-фреймворк, обеспечивающий высокую

производительность благодаря асинхронной обработке запросов, основанной на стандарте ASGI (Asynchronous Server Gateway Interface). В отличие от традиционных синхронных фреймворков, таких как WSGI, где каждый запрос блокирует поток или процесс до завершения обработки, FastAPI позволяет одновременно обрабатывать множество запросов, эффективно используя ресурсы и поддерживая быстрый отклик системы. Это особенно важно для приложений, где могут быть выполнены несколько тяжёлых запросов, например, загрузка больших изображений. Кроме того, FastAPI значительно упрощает разработку, автоматически генерируя документацию для всех эндпоинтов с использованием стандарта OpenAPI. Это позволяет разработчикам и интеграторам быстро ознакомиться с доступными методами (GET, POST и другими), их параметрами и даже протестировать API через интерактивный интерфейс Swagger UI, что ускоряет процесс разработки, уменьшает количество ошибок и улучшает взаимодействие в команде.

2. Celery[22] - распределённая система выполнения фоновых задач, разработанная на языке Python. Она позволяет выносить ресурсоёмкие, медленные или периодические операции за пределы основного приложения и обрабатывать их асинхронно, в отдельном процессе или даже на другом сервере. Основное назначение Celery — разгрузка основного потока приложения и обеспечение высокой производительности и масштабируемости за счёт распределённого исполнения задач.

3. RabbitMQ[23] — брокер сообщений, реализующий ряд протоколов и широко применяющийся для организации асинхронного обмена данными в распределённых системах. Он обеспечивает чёткое разделение ответственности между компонентами, высокую отказоустойчивость и масштабируемость. В основе архитектуры лежит модель "поставщик — очередь — потребитель", где сообщения помещаются в очередь и обрабатываются потребителями по мере готовности, что позволяет эффективно распределять нагрузку и избегать блокировок. Благодаря использованию Erlang, RabbitMQ способен

обрабатывать большое количество соединений и обеспечивает высокую надёжность доставки сообщений. Он поддерживает различные механизмы маршрутизации через обменники и типы доставок (fanout, topic, direct), что делает его гибким инструментом для реализации как простых очередей, так и сложных событийных систем.

4. React[24] - библиотека для построения пользовательских интерфейсов, выбранная для разработки проекта благодаря своей производительности и гибкости. Она использует виртуальный DOM, что позволяет минимизировать количество операций с реальным DOM и ускорить рендеринг интерфейса. Вместо того чтобы обновлять весь DOM при изменении, React обновляет только те его части, которые были изменены, что повышает эффективность работы приложения, особенно в случае с динамическими интерфейсами. React использует компонентный подход, позволяющий разделять интерфейс на независимые части, которые можно повторно использовать, что упрощает разработку, поддержку и расширение функционала. Кроме того, React поддерживает хуки для управления состоянием и побочными эффектами, что позволяет эффективно работать с асинхронными операциями и улучшает читаемость кода.

5. Nginx[25] - веб-сервер и обратный прокси-сервер, выбранный для проекта для эффективной обработки входящих HTTP-запросов и распределения нагрузки между компонентами системы. Он позволяет обрабатывать большое количество параллельных соединений с минимальным использованием системных ресурсов, благодаря асинхронной модели обработки запросов. В отличие от традиционных веб-серверов, которые создают новый процесс или поток для каждого запроса, Nginx использует однотипные процессы, что значительно снижает накладные расходы при масштабировании. В проекте Nginx используется как прокси-сервер для маршрутизации запросов между фронтендом и бэкендом, а также для обеспечения безопасности через SSL/TLS-шифрование. Кроме того, он играет ключевую роль в балансировке нагрузки,

распределяя запросы между несколькими экземплярами серверов, что позволяет улучшить отказоустойчивость и масштабируемость системы.

6. PostgreSQL[26] - объектно-реляционная система управления базами данных с открытым исходным кодом, которая зарекомендовала себя как мощная и гибкая платформа для разработки приложений. Ее основное преимущество заключается в поддержке расширенных типов данных, таких как JSON, HSTORE, а также поддержка полнотекстового поиска и пользовательских типов данных, что позволяет создавать сложные и высокоэффективные решения. PostgreSQL активно используется для разработки крупных проектов, требующих надежности, масштабируемости и высокой производительности при обработке данных. В отличие от многих других СУБД, PostgreSQL предлагает полноценную поддержку транзакций и механизмы ACID, что гарантирует безопасность данных и их консистентность. Он предоставляет механизмы работы с индексами, такие как B-деревья, Hash-индексы, GiST, GIN и другие, что повышает скорость выборки данных. Кроме того, PostgreSQL поддерживает параллельную обработку запросов и возможность распределенного хранения данных, что существенно улучшает его производительность при работе с большими объемами информации. Эта СУБД активно используется в средах, где требуется высокая отказоустойчивость и поддержка сложных бизнес-логик. PostgreSQL отличается гибкостью в настройке и расширении: пользователи могут добавлять свои собственные функции, операторы, типы данных и даже языки программирования для написания хранимых процедур. Благодаря поддержке репликации и кластеризации, PostgreSQL идеально подходит для распределенных систем и высоконагруженных приложений, требующих быстрой обработки больших объемов данных и масштабируемости. Также стоит отметить широкую поддержку со стороны сообщества, многочисленные библиотеки и утилиты, которые делают интеграцию PostgreSQL с другими инструментами простым и удобным процессом. Именно эти качества делают PostgreSQL идеальным выбором для крупных веб-проектов, корпоративных приложений, а также для

систем, работающих с данными в реальном времени.

7. MinIO[27] - высокопроизводительное распределённое хранилище объектов, совместимое с Amazon S3, которое предоставляет возможности для управления большими объёмами данных с использованием API S3. Его основное преимущество заключается в лёгкости и гибкости развертывания: MinIO можно установить и настроить за считанные минуты, что делает его отличным решением для проектов, которым необходимо быстро развернуть систему для хранения и обработки данных. Он идеально подходит для хранения неструктурированных данных, таких как изображения, видео, резервные копии, а также для использования в контейнеризованных и облачных средах благодаря своей совместимости с S3 и Docker. MinIO поддерживает горизонтальное масштабирование, что позволяет легко увеличивать объём хранилища путём добавления новых серверов в кластер. Это особенно важно для проектов с высоким ростом данных, поскольку MinIO предлагает эффективное управление нагрузкой и высокую доступность. Система также включает механизмы защиты данных, такие как шифрование, управление правами доступа, а также возможность использования нескольких реплик для обеспечения отказоустойчивости. Она может быть интегрирована в различные экосистемы, включая аналитические платформы, системы машинного обучения и любые другие приложения, которые нуждаются в хранении больших объёмов данных. Включение MinIO в проект позволяет не только снизить зависимость от крупных облачных провайдеров, но и обеспечить гибкость в управлении инфраструктурой хранения, а также контроль над её стоимостью и масштабируемостью.

2. **Обработка изображений**

1. Поддержка распространенных форматов на входе и их последующее успешное преобразование: JPEG, PNG, TIFF.

2. Внутренняя обработка сверточной нейронной сетью: Lab-формат, 128×128 U-Net.

3. Выдача распространенного и универсального формата на выходе: PNG.

3. Интерфейс

1. Пользовательский web-интерфейс на основе React: загрузка пользовательских изображений, отправка на обработку, отслеживание состояние обработки и возможность удобной выгрузки результатов работы сервиса. Интерфейс должен быть простым для пользователей и быть адаптивным: на малых экранах кнопки, поля и другие элементы управления не должны быть слишком маленькими, а расстояния между ними должны предотвращать случайные нажатия, в то время как настольная версия должна выглядеть современно и не требовать от продвинутых знаний пользователя.

2. REST API для интеграции веб-сервиса с серверной частью, и, в частности, для предоставления общедоступного API для использования в настольных и мобильных приложениях, встраивания в автоматизированные системы и использования для обучающих целей.

4. Очередь задач и масштабируемость

1. Отдельные Celery-приложения + RabbitMQ: для синхронной обработки на несколько машин в нескольких потоках, обеспечивая вертикальную и горизонтальную масштабируемости.

2. Возможность пакетной обработки нескольких изображений, объединяя их в один запрос в базе данных с несколькими связанными в S3-совместимом хранилище объектами окрашивания.

3. Балансировка входящих подключений и сокетов через Nginx для обеспечения максимально возможного количества подключений пользователей.

5. Простота развертывания

1. Docker + docker-compose: один конфигурационный файл и среда для развертывания множества компонентов системы: хранилища, брокера

сообщений, приложений-обработчиков и FastAPI-приложения.

2. All-in-one: при старте поднимаются все компоненты для оптимизации развертывания, тестирования и отладки.

6. **Условия эксплуатации**

1. **Уровень подготовки пользователя**: начальный — интерфейс на React сводит ввод пользователя к форме загрузки, кнопки начала обработки и кнопки выгрузки результата.
2. **Среда**: сервер Linux (Ubuntu 20.04+), Docker Engine.
3. **Доступ**: роль администратора для развертывания, роль оператора для загрузки/запуска задач.

7. **Требования к составу и параметрам технических средств**

1. **CPU**: минимум 2-ядерный, поддержка SSE2.
2. **RAM**: от 4 ГБ (рекомендуется 8 ГБ).
3. **GPU**: необязательно, при наличии поддержка CUDA ускоряет инференс.
4. **Дисковое пространство**: от 8 ГБ свободного места под образы, контейнеры, базу данных и хранилище.
5. **Сеть**: пропускная способность ≥100 Мбит/с для работы с большими изображениями и большим количеством запросов

8. **Требования к информационной и программной совместимости**

1. **ОС**: Linux (Ubuntu 20.04+), macOS, Windows Server.
2. **Языки и среды**: Python 3.7+, Node.js 14+.
3. **Библиотеки**: NumPy, OpenCV, PIL, PyTorch (или TensorFlow, опционально).
4. **Сервисы**: PostgreSQL 12+, RabbitMQ 3.8+, MinIO RELEASE.
5. **Web-сервер**: Nginx 1.18+.
6. **Контейнеризация**: Docker 20.10+, docker-compose 1.29+.

Итого, система рассчитана на быстрое и простое развёртывание, минимальный

порог входа для пользователя и при этом сохраняет гибкость и масштабируемость, требуемые для промышленного применения и академических экспериментов.

1.5 Выводы по разделу 1

Проблема окрашивания изображений для широкой аудитории, не обладающей профильными знаниями и опытом работы с профессиональными программными комплексами, заключается в необходимости освоения разнообразных инструментов и сервисов, которые, как правило, не ориентированы на людей без технической подготовки и могут быть сложными и непонятными для большинства пользователей. Такие решения часто требуют наличия определённых технических навыков и опыта работы с графикой или нейронными сетями, что ставит серьёзные барьеры для тех, кто не обладает специализированными знаниями в этих областях. Кроме того, многие из существующих сервисов имеют сложные интерфейсы и могут быть перегружены функциями, которые могут отпугнуть даже более опытных пользователей.

Для того чтобы преодолеть эти трудности и предоставить удобное решение для широкой аудитории, необходимо разработать сервис с простым, интуитивно понятным интерфейсом, который будет доступен для пользователей с любым уровнем подготовки, включая тех, кто не имеет опыта работы с подобными инструментами. Важно, чтобы работа с таким сервисом не требовала предварительного обучения или освоения сложных программ. Использование U-Net нейронных сетей, в этом контексте, становится особенно удачным выбором, поскольку они достаточно эффективны для решения задачи автоматического окрашивания изображений и при этом не предъявляют чрезмерных требований к вычислительным ресурсам и времени обучения. Это открывает возможности для создания сервиса, который сможет быстро и эффективно выполнять окрашивание изображений с минимальными затратами времени и без необходимости в ожидании длительных процессов обработки данных.

Тем не менее, при разработке такого решения важно учесть, что, несмотря на значимость точности и качества получаемого результата, для успешной реализации сервиса в первую очередь необходимо обеспечить его скорость и лёгкость в обслуживании. Простота интерфейса и лёгкость самой нейронной сети являются основными факторами, которые прямо влияют на доступность и привлекательность сервиса для широкой аудитории. Важно, чтобы пользователи могли получить быстрый результат без значительных задержек, а поддержка системы была достаточно лёгкой, что в свою очередь позволит избежать излишних затрат на инфраструктуру и обслуживание. Результат окрашивания, безусловно, должен быть на должном уровне, однако скорость работы и удобство использования системы должны стать первоочередными задачами, ведь чем проще и доступнее сервис, тем более вероятно, что он будет востребован среди широкой аудитории, включая тех, кто не обладает глубокими знаниями в области графики и технологий машинного обучения.

2 РАЗРАБОТКА СЕРВИСА ОКРАШИВАНИЯ ЧЕРНО-БЕЛЫХ ИЗОБРАЖЕНИЙ СРЕДСТВАМИ МАШИННОГО ОБУЧЕНИЯ

2.1 Формальное определение алгоритма работы решения

В данном разделе описывается порядок разработки IT-решения, который включает в себя этапы проектирования, реализации и тестирования всех компонентов системы. Решение включает в себя несколько ключевых компонентов, начиная с создания макета интерфейса и заканчивая применением нейросетевых алгоритмов для обработки изображений. Процесс разработки можно условно разделить на несколько основных этапов: от разработки интерфейса и бэкенда до взаимодействия с различными сервисами, такими как RabbitMQ, MinIO, PostgreSQL и Celery, а также интеграции с нейросетью U-Net.

2.1.1 Макет интерфейса (Frontend)

Первая часть разработки заключается в создании макета интерфейса с использованием **React**. На этом этапе разрабатывается визуальное представление системы, включая формы для загрузки изображений, отображение прогресса обработки и статусных сообщений. React используется для построения интерфейса, так как он позволяет создать динамичные и отзывчивые веб-приложения с быстрым откликом на действия пользователя. В рамках макета пользователь может загрузить исходное изображение, выбрать нужные параметры обработки и отслеживать статус выполнения задачи в реальном времени.

2.1.2 Разработка frontend-приложения на React

После утверждения макета интерфейса начинается реализация веб-приложения. На этом этапе с использованием **React** создаются все необходимые компоненты и логика взаимодействия с серверной частью. Веб-приложение будет отправлять HTTP-запросы к серверу, обрабатывать их,

отображать статус обработки и конечные результаты пользователю. React в данном случае позволяет эффективно управлять состоянием интерфейса и динамически обновлять данные без необходимости перезагружать страницы. Приложение также будет интегрировано с системой уведомлений для информирования пользователя о статусе обработки изображений.

2.1.3. Backend-сервис на FastAPI

На серверной стороне реализуется **FastAPI** — высокопроизводительный фреймворк для создания API. FastAPI будет отвечать за прием запросов от frontend-приложения, обработку данных и взаимодействие с внешними сервисами. Он предоставляет быстрый и гибкий механизм для создания асинхронных API, что является важным при работе с изображениями и их обработкой. API будет обрабатывать запросы на загрузку изображений, запуск процессов их обработки, а также отдавать статусы по текущему состоянию задачи (например, ожидает обработки, в процессе, завершено).

2.1.4. Очередь сообщений RabbitMQ и обработка задач с помощью Celery

Для асинхронной обработки задач и эффективного масштабирования используется **RabbitMQ**, который будет выполнять роль брокера сообщений. Все запросы от веб-приложения, требующие долгой обработки, такие как конвертация и нейросетевые вычисления, будут отправляться в очередь сообщений RabbitMQ. Каждая задача будет иметь свой уникальный идентификатор, который затем будет использован для отслеживания прогресса выполнения. В очереди сообщений будет храниться информация о том, что необходимо выполнить (например, конвертация изображения или запуск нейросети).

Использование **Celery** для обработки задач позволяет значительно

упростить асинхронную работу и выполнить задачи в распределенной среде. Celery будет получать задачи от RabbitMQ, выполнять их параллельно (на нескольких воркерах), и в конечном итоге передавать результат обратно в систему. Например, на этом этапе Celery будет управлять запуском нейросетевых алгоритмов и процессов конвертации изображений.

2.1.5. Хранение данных в MinIO и PostgreSQL

Для хранения изображений и метаданных используется два компонента: **MinIO** и **PostgreSQL**. **MinIO** будет использоваться для хранения исходных изображений, а также их обработанных версий в виде файлов. MinIO, являясь объектным хранилищем, эффективно работает с большими объемами данных и совместим с протоколом S3, что позволяет использовать его в качестве альтернативы более дорогим решениям, таким как Amazon S3. Все операции с файлами, такие как загрузка, скачивание и сохранение, будут происходить через MinIO.

В то время как MinIO будет хранить изображения, **PostgreSQL** будет использоваться для хранения метаданных изображений, информации о статусах их обработки, а также для учета истории всех операций. PostgreSQL позволяет гибко работать с реляционными данными и предоставляет возможности для эффективного поиска, фильтрации и обновления данных. Каждое изображение будет иметь уникальный идентификатор в базе данных, а также информацию о текущем статусе, таких как "обработка", "конвертация завершена", "завершено". PostgreSQL также будет отвечать за сохранение логов и данных, связанных с пользовательскими запросами.

2.1.6. Нейросеть U-Net для обработки изображений

После того как изображение загружено и все необходимые метаданные сохранены в базе данных, система использует нейросеть **U-Net** для обработки

изображений. **U-Net** — это глубокая нейронная сеть, разработанная для сегментации изображений, и она идеально подходит для задач, связанных с разделением объектов на изображении, такими как окрашивание. В рамках этого проекта нейросеть будет применяться для выполнения предсказания на изображениях, улучшая их или добавляя соответствующие фильтры.

Использование нейросети U-Net для окрашивания изображений позволяет эффективно обучать модель на примерах, с последующим применением модели для обработки новых изображений. Результаты работы нейросети будут комбинироваться с исходными изображениями для получения окончательного результата.

Задача автоматической раскраски изображений сводится к тому, чтобы по входному одноканальному (чаще всего — градациям серого) изображению $X \in R^{\{H \times W\}}$ предсказать соответствующее цветное изображение $Y \in R^{H \times W \times 3}$. То есть необходимо научить модель (нейросеть) некоторой функции

$$f : R^{\{H \times W\}} \quad R^{\{H \times W \times 3\}}$$

которая по каждому пикселю серого «догадается», какой у него должен быть цвет.

Чаще всего цвет представляют в пространстве Lab (или YUV), где входной канал L (lightness) соответствует яркости — его и подают на вход. Сеть предсказывает два канала a и b, отвечающие за цветовую информацию, а итоговое цветное изображение восстанавливают обратным преобразованием в RGB.

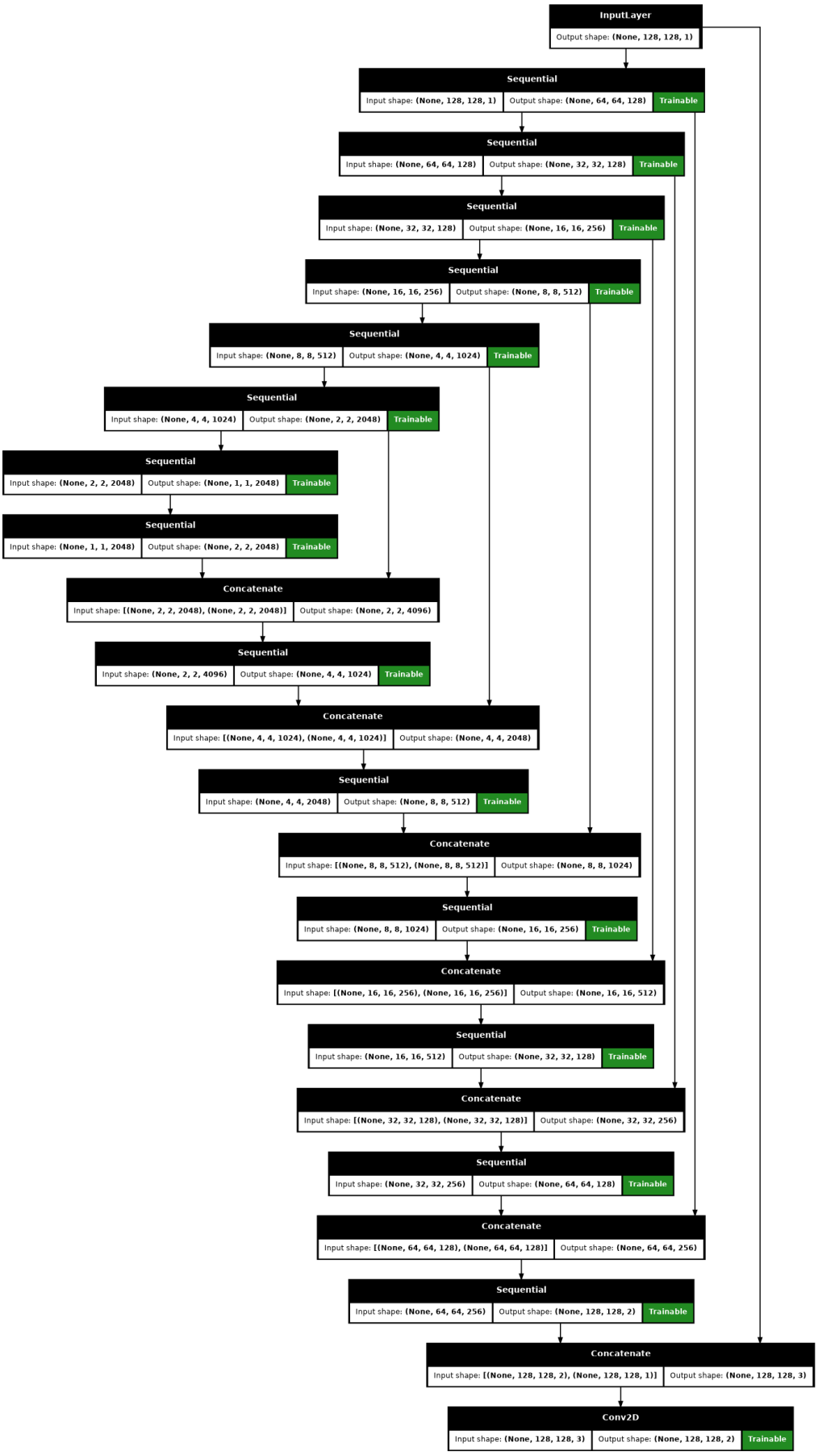

Рисунок 1 - Архитектура нейронной сети

2.1.7. Конвертация изображений и сохранение в формате .PNG

После того как нейросеть завершит свою работу, изображение будет дополнительно обработано и конвертировано в нужный формат — **PNG**. Этот этап включает в себя наложение исходных изображений и результатов работы нейросети, а затем преобразование в формат PNG для сохранения высокого качества изображения и совместимости с различными программами и платформами.

2.1.8. Сохранение обработанных изображений в MinIO

После завершения всех процессов обработки изображение сохраняется в **MinIO** в качестве окончательной версии. Это позволяет сохранить результаты на долгосрочное хранение, доступные для последующего скачивания или отображения через веб-интерфейс. В MinIO изображения будут сохранены в соответствующем бакете, что облегчает организацию хранилища для большого объема данных.

2.1.9. Обновление статуса на Web-сервисе

После того как изображение было обработано и сохранено в MinIO, последний этап включает в себя обновление статуса задачи в веб-приложении. Веб-сервис, используя FastAPI, обновит базу данных PostgreSQL с новым статусом изображения, который будет отображен в интерфейсе пользователя. Пользователь сможет увидеть, что изображение обработано, и скачать его из MinIO через интерфейс веб-приложения.

2.1.10 Модель/теоретический метод решения задачи

Процесс обработки изображений и их окрашивания основывается на теоретических методах, связанных с **обработкой изображений** и **искусственным интеллектом**. Использование нейросети **U-Net** позволяет

достигать высокой точности в обработке изображений, делая возможным выполнение сложных операций, таких как сегментация, предсказание цвета и улучшение качества изображений. Этот подход позволяет автоматизировать процесс окрашивания, значительно снижая потребность в ручной обработке и ускоряя весь процесс.

Общий вид архитектуры системы, ее основных блоков и их взаимодействий представлен на рисунке 1.

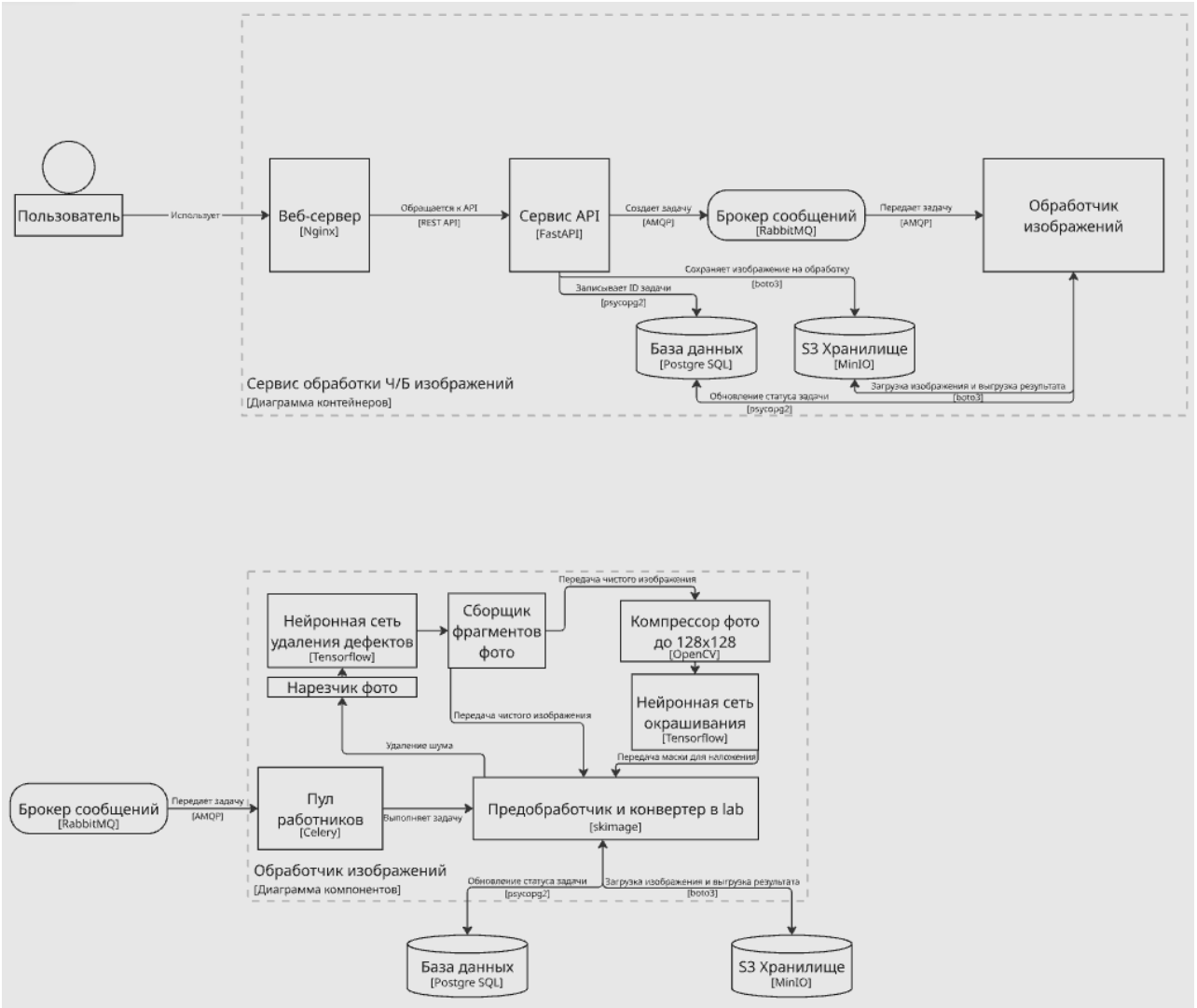

Рисунок 2 - Архитектура решения

Представленная общая архитектура решения позволяет выполнить все поставленные задачи, а архитектура нейронной сети — обработать изображение быстро и дешево.

2.2 Используемые программные компоненты и решения

2.2.1 Celery

В архитектуре данного проекта Celery выполняет ключевую роль в организации **асинхронной, распределенной и масштабируемой обработки задач**, которые по своей сути являются ресурсоемкими или требуют значительного времени выполнения. Это такие операции, как загрузка и обработка изображений, чтение и запись больших бинарных данных из и в S3-хранилище, а также обновление записей в базе данных PostgreSQL. Без вынесения этих операций из основного потока исполнения приложения они могли бы значительно снижать производительность и отзывчивость API, особенно при увеличении количества одновременных пользователей.

Celery позволяет реализовать архитектуру, в которой тяжеловесные процессы выполняются **в фоновом режиме**, параллельно и независимо от основного приложения. Таким образом, пользователь взаимодействует с FastAPI только для отправки задачи, а ее обработка — будь то преобразование изображения или запись данных — выполняется в отдельном процессе, управляемом Celery worker'ом.

Одной из важнейших задач любого веб-приложения является обеспечение быстрой реакции на пользовательские запросы. Когда клиент загружает изображение, это может быть операция размером в десятки мегабайт. Если бы API обрабатывал это изображение синхронно — загружал его, преобразовывал, сохранял результат и обновлял базу данных — пользователь был бы вынужден

ждать окончания всех этих шагов, что могло бы занять несколько секунд, а при увеличении нагрузки — даже дольше.

Celery устраняет эту проблему, позволяя API лишь принять запрос и отправить его в очередь задач, после чего клиент может продолжить работу. Само же преобразование изображения (например, инверсия цветовой гаммы) произойдет на стороне Celery worker'a, без участия FastAPI и без блокировки основного потока.

Еще одно важное назначение Celery — обеспечение **асинхронного выполнения задач** с гарантией их доставки и исполнения. Благодаря использованию надёжного брокера сообщений (в данном случае RabbitMQ) и поддержке подтверждений доставки, даже при временных сбоях или перезапуске сервисов задачи не теряются. Это особенно важно в системах, где критично сохранить точность данных, например, при обновлении записей в базе после завершения обработки изображения.

Кроме того, при масштабировании системы можно запускать несколько Celery worker'ов — на одной или нескольких машинах, в том числе в Docker-контейнерах. Это позволяет обрабатывать больше задач одновременно и выдерживать высокую нагрузку.

В архитектуре проекта Celery используется как специализированный механизм выполнения логики, которая не связана напрямую с HTTP-запросами. Это позволяет более чётко разделить логику:

Также Celery позволяет легко адаптироваться к росту проекта. Если количество пользователей или объем обрабатываемых данных увеличится, можно без изменений в коде развернуть дополнительные worker-процессы, каждый из которых будет забирать задачи из общей очереди и выполнять их параллельно. Это делает архитектуру не только устойчивой, но и гибкой к изменениям, характерным для реальных систем.

Таким образом, назначение Celery в архитектуре проекта заключается в обеспечении эффективного распределения ресурсоемких задач, разгрузке основного приложения, повышении устойчивости и масштабируемости системы. Без использования Celery каждая сложная операция ложилась бы на FastAPI напрямую, что неминуемо привело бы к деградации производительности, задержкам и неудовлетворительному пользовательскому опыту. Celery выступает связующим компонентом, обеспечивающим баланс между высокой скоростью реакции API и полноценной обработкой данных в фоновом режиме.

2.2.2 RabbitMQ

В рамках данного проекта RabbitMQ выполняет критически важную роль — он является связующим звеном между FastAPI и Celery, выступая в роли брокера сообщений. Его основная задача заключается в организации надёжной и эффективной доставки задач, формируемых веб-приложением, до фоновых воркеров, обрабатывающих эти задачи асинхронно. Без наличия такого компонента система лишилась бы возможности масштабируемого и устойчивого обмена задачами, особенно в условиях высокой нагрузки.

Технологически взаимодействие строится на протоколе AMQP (Advanced Message Queuing Protocol), который лежит в основе RabbitMQ. Это универсальный, платформонезависимый протокол, позволяющий не только пересылать задачи, но и гарантировать их доставку, сохранять порядок сообщений, организовывать подтверждения доставки и управлять повторными попытками в случае сбоев. Такая гибкость необходима для надёжного обмена сообщениями между распределёнными компонентами системы.

Схема взаимодействия в архитектуре проекта построена по классическому паттерну "поставщик — очередь — потребитель" (producer —

queue — consumer). Здесь FastAPI играет роль поставщика: при получении запроса от пользователя, например, на загрузку изображения, приложение формирует задачу и направляет её в очередь RabbitMQ. Очередь в данном случае — это временное хранилище сообщений, обеспечивающее буферизацию между производителем и потребителем. После помещения задачи в очередь она ожидает обработки. Celery worker, работающий независимо от FastAPI, является потребителем — он подписан на нужную очередь, извлекает поступающие задачи и приступает к их выполнению.

Преимущество такой архитектуры в том, что она обеспечивает полную асинхронность и независимость компонентов. FastAPI не должен ждать, пока Celery завершит задачу — оно просто ставит её в очередь и сразу может продолжать обслуживание других запросов. RabbitMQ, в свою очередь, гарантирует, что ни одна задача не будет потеряна, даже если Celery временно недоступен или перезапускается. Благодаря механизму подтверждения сообщений, Celery сообщает брокеру, что задача обработана, и только после этого сообщение удаляется из очереди. Это позволяет минимизировать риски потерь и добиться высокой надёжности всей системы.

Таким образом, RabbitMQ — это не просто транспорт для задач, а полноценный механизм управления очередями, обеспечивающий отказоустойчивость, надёжность, упорядоченность и масштабируемость коммуникации между веб-приложением и системой фоновой обработки.

2.2.3 AMQP

Протокол AMQP был выбран не случайно — он является одним из самых надёжных и функциональных протоколов обмена сообщениями. Его архитектура предполагает подтверждение доставки сообщений, что позволяет гарантировать, что каждое сообщение будет обработано ровно один раз. Это

критично для систем с фоновой обработкой задач, где потеря или дублирование задачи может привести к неправильным результатам или неконсистентности данных. Кроме того, AMQP поддерживает различные модели маршрутизации сообщений через такие компоненты, как обменники (exchanges), очереди и биндинги, что даёт гибкость в организации сложных сценариев взаимодействия и балансировки нагрузки между воркерами.

2.2.4 FastAPI

FastAPI поддерживает возможности **вертикальной масштабируемости** — увеличения количества процессов (воркеров) для обработки входящих HTTP-запросов.

Благодаря своей архитектуре (ASGI) и использованию серверов, таких как Uvicorn или Gunicorn, FastAPI может быть запущен сразу с несколькими воркерами. Каждый такой воркер:

- Независим,
- Может обслуживать собственные HTTP-запросы,
- Позволяет API «разгрузить» очередь входящих запросов и быстро отвечать пользователям.

  В реальных условиях:
- Если обычная нагрузка — это 10–20 запросов в секунду, можно использовать, например, 2–3 воркера.
- Если же количество пользователей резко возрастает, можно «поднять» до 5–10 воркеров.
- Это делается без изменений в коде — только настройкой числа воркеров в конфигурации сервера.

Такой подход позволяет API оставаться **быстрым и отзывчивым**, а пользователи — всегда получать актуальные данные о состоянии своих задач.

2.2.5 MinIO

Использование MinIO в качестве решения для облачного хранилища данных является стратегически обоснованным выбором, особенно в контексте альтернативы таким облачным провайдерам, как Amazon Web Services (AWS). Основная причина, по которой MinIO становится предпочтительным выбором, заключается в его открытой архитектуре, высоком уровне гибкости и возможности самостоятельного управления инфраструктурой без зависимости от крупных коммерческих игроков, таких как Amazon. В то время как AWS предоставляет ряд мощных инструментов для работы с хранилищами данных, его подход часто сопровождается значительными затратами на использование ресурсов, а также зависимостью от политик и решений, принимаемых этим облачным гигантом. Проблемы с монополизацией и изменениями ценовой политики, которые могут негативно повлиять на клиентов, являются весомыми аргументами в пользу использования решений с открытым исходным кодом, таких как MinIO.

MinIO совместим с интерфейсом Amazon S3, что делает его особенно привлекательным для организаций, которые хотят избежать привязки к Amazon и использовать более доступные решения с теми же функциональными возможностями. MinIO позволяет развертывать хранилище в собственной инфраструктуре или на облачных серверах, что даёт возможность контролировать все аспекты работы хранилища, включая безопасность, резервное копирование и доступ к данным. Это важное преимущество перед решениями вроде AWS, где пользователи вынуждены полагаться на внешнюю компанию, которая диктует условия использования своих сервисов.

Кроме того, использование MinIO даёт организациям возможность избежать скрытых издержек, связанных с изменениями в ценах и условиях предоставления сервисов в рамках AWS. Amazon, как крупнейший облачный провайдер, нередко вводит новые условия, которые могут значительно

увеличить расходы пользователей. В то время как MinIO, как решение с открытым исходным кодом, предоставляет полную свободу в настройке и использовании хранилища, что даёт возможность обеспечить более стабильную и предсказуемую стоимость эксплуатации.

Таким образом, MinIO представляет собой надёжную, экономически эффективную альтернативу Amazon S3, предоставляя те же возможности по хранению и управлению объектами данных без зависимости от внешних провайдеров. Это позволяет снизить риски, связанные с изменениями условий обслуживания и ценовых политик крупных игроков, таких как Amazon, и дает большую степень контроля над собственной инфраструктурой.

2.2.6 PostgreSQL

Использование PostgreSQL в контексте задачи окрашивания изображений является обоснованным выбором по нескольким ключевым причинам, включая наличие обширного сообщества, документации, а также интеграции с Python. PostgreSQL — это объектно-реляционная система управления базами данных (СУБД), которая обладает высокой производительностью, гибкостью и возможностями для работы с комплексными запросами, что особенно важно при обработке больших объёмов данных, связанных с изображениями и их метаданными.

Одним из ключевых факторов, который обосновывает использование PostgreSQL в данном случае, является её популярность и наличие огромного сообщества. Благодаря этому обеспечивается постоянное развитие и улучшение СУБД, а также доступ к широкому спектру библиотек, инструментов и расширений, которые значительно упрощают решение специфических задач. В частности, для работы с изображениями и их метаданными существует целый ряд расширений, таких как PostGIS, которое позволяет эффективно работать с географическими данными, и другие дополнения, которые могут быть полезны

для управления изображениями в контексте обработки и их связей с другими объектами данных.

Кроме того, PostgreSQL обладает превосходной поддержкой работы с запросами, что важно в контексте задачи, где необходимо осуществлять поиск и обработку больших коллекций данных. В случае с окрашиванием изображений, хотя сами изображения и не хранятся непосредственно в базе данных, их метаданные, такие как параметры окрашивания, характеристики обработки и ссылки на местоположение изображений, могут быть эффективно организованы и обработаны с помощью сложных SQL-запросов. PostgreSQL поддерживает сложные типы данных и позволяет легко организовать структуру базы данных для хранения связанной информации, что даёт разработчикам гибкость в организации данных и быстроту работы с ними.

Документация PostgreSQL является одним из наиболее проработанных ресурсов среди современных СУБД, что значительно упрощает процесс разработки. Благодаря детализированным руководствам, примерам и активному сообществу разработчиков, всегда можно найти решения для специфических проблем, с которыми может столкнуться проект. Это особенно важно в случае сложных задач, таких как автоматизация окрашивания изображений, где работа с большими наборами данных и необходимостью быстро обрабатывать запросы требует стабильной и хорошо документированной платформы.

Ещё одним важным аспектом является совместимость PostgreSQL с языками программирования, такими как Python. PostgreSQL активно используется в экосистеме Python благодаря таким библиотекам, как Psycopg2 и SQLAlchemy, которые обеспечивают простоту подключения и интеграции базы данных с Python-программами. Это даёт разработчикам возможность использовать мощь PostgreSQL для хранения и обработки данных в сочетании с

гибкостью Python для реализации алгоритмов и бизнес-логики, связанных с обработкой изображений. В контексте окрашивания изображений Python может использоваться для выполнения вычислений и обработки данных, в то время как PostgreSQL будет эффективно управлять метаданными, хранением параметров и настройками для каждого изображения.

Также стоит отметить, что PostgreSQL предоставляет поддержку транзакционности, что позволяет гарантировать целостность данных в случае любых сбойных ситуаций при выполнении запросов. Это крайне важно при работе с базами данных, где требуется строгий контроль над операциями обновления, вставки и удаления данных, особенно когда они могут быть связаны с крупными объёмами изображений и метаданных. PostgreSQL гарантирует, что операции будут выполнены последовательно и в случае неудачи не будет нарушена целостность базы данных.

Таким образом, использование PostgreSQL для обработки запросов, связанных с окрашиванием изображений, является обоснованным выбором благодаря высокому уровню производительности, удобной интеграции с Python, наличию документации и большому сообществу пользователей. Эти факторы позволяют эффективно решать задачи, связанные с хранением метаданных изображений, их поиском, фильтрацией и обновлением, а также обеспечивают высокую степень гибкости и масштабируемости решения.

2.2.7 U-Net

Архитектура нейронной сети и обоснование выбора

В ходе разработки нейронной сети для автоматической раскраски изображений было проведено исследование нескольких архитектурных подходов, основанных на современных методах машинного обучения и компьютерного зрения. В результате анализа были выделены три основные архитектуры, которые потенциально подходили для решения поставленной

задачи: классический сверточный автоэнкодер (CNN AE), архитектура U-Net и генеративно-состязательные сети (GAN). Далее представлены обоснования выбора каждой из этих архитектур.

1. **Сверточный автоэнкодер (CNN AE)**

Сверточные автоэнкодеры (CNN AE) используют метод проекции входных данных в сжатое скрытое подпространство с последующим восстановлением этих данных. Такой подход позволяет сети эффективно извлекать значимые абстрактные признаки, что делает её подходящей для задач, связанных с обработкой изображений. Внутри автоэнкодера используются сверточные слои, которые предполагают пространственную инвариантность, что позволяет локальным фильтрам извлекать паттерны независимо от их положения в изображении. Это свойство является важным для задач, где объекты могут находиться в разных частях изображения, но должны быть обработаны одинаково.

2. **U-Net**

Архитектура U-Net была выбрана благодаря своей способности сохранять детализированную пространственную информацию при обработке изображений. Она включает в себя так называемые *skip-связи* между энкодером и декодером, что позволяет более эффективно передавать информацию о контурах объектов и границах областей. Это особенно важно для задач сегментации и восстановления изображений, где критична точность границ объектов. Дополнительным преимуществом является упрощение процесса градиентного распространения в глубокой нейронной сети, что способствует ускорению процесса обучения.

3. **Генеративно-состязательные сети (GAN)**

Генеративно-состязательные сети, состоящие из генератора и дискриминатора, эффективно используют механизм обучения с

соперничеством для создания более правдоподобных изображений. Генератор обучается так, чтобы «обмануть» дискриминатор, создавая текстуры и цветовые переходы, которые трудно отличить от реальных. Этот подход значительно улучшает визуальное качество результирующих изображений за счёт использования перцептуальных функций потерь. GAN также хорошо справляются с задачей заполнения областей изображений, где отсутствуют чёткие указания на цвет, подстраиваясь под статистику реальных изображений, что является преимуществом при работе с изображениями, содержащими много неструктурированных данных.

В результате проведённого анализа было принято решение использовать гибрид архитектур, включающий элементы GAN, U-Net и CNN AE, так как такой подход позволял бы эффективно решать поставленную задачу, используя сильные стороны каждой из этих технологий. Однако, учитывая ограниченный опыт команды, было принято решение поэтапно усложнять архитектуру, начиная с более простых моделей, чтобы накапливать опыт в обучении нейронных сетей, постепенно переходя к более сложным решениям.

2.2.8 Формат lab

При решении задачи автоматической раскраски ключевым моментом является то, в каком цветовом пространстве мы подаём изображение на вход и как моделируем выход. Выбор падает на **CIELAB** по следующим причинам:

1. **Разделение яркости и цветоразностных компонентов**
   - В Lab канал **L** отвечает только за яркость (lightness), а каналы **a** и **b** — за цвет (“зелёно–красный” и “сине–жёлтый” соответственно).
   - Это чёткое разделение позволяет модели фокусироваться на задаче раскраски (аппроксимации a/b), не «засоряя» обучение информацией о яркости. Что упрощает модель, ведь она должна предсказывать не 3 значения канала, а 2.

2. **Перцептуальная равномерность**
    — Расстояние между двумя точками в пространстве Lab лучше коррелирует с субъективным восприятием человеком, чем в RGB или YUV.
    — Ошибка предсказания в Lab напрямую отражается на том, насколько заметна будет разница цвета для глаза.
3. **Линейность преобразований**
    — В отличие от нелинейных представлений (HSV, HSL), Lab построено на относительно простых (линейных или кусочно-линейных) преобразованиях от XYZ, что упрощает оптимизацию функций потерь.
    — Не требуется дополнительно обучать модель «понимать» свойства цветового круга и центрирование hue.
4. **Декорреляция каналов**
    — В RGB компоненты сильно коррелированы (изменение яркости влияет на все три канала), что затрудняет модели выделять семантику цвета.
    — В Lab каналы a и b уже “отделены” от яркости, поэтому сеть учится только тому, как заполнять цветовой слой, при том, что структура яркости остаётся без изменений.
5. **Стабильность нормализации**
    — Диапазон L = [0, 100] и a/b приблизительно в [–128, +127] позволяет легко задать фиксированные шкалы преобразования в [0, 1] и [–1, 1] соответственно.
    — Это даёт более предсказуемую динамику градиентов при обучении по сравнению с RGB ([0, 1] для каждого канала, но разный вклад в яркость).
6. **Сравнение с YUV и HSV**
    — YUV и YCbCr тоже отделяют яркость (Y) от хрома (U/V или Cb/Cr), но они были разработаны для теле- и видеокодеков с учётом

ограничений передачи, а не для перцептуальной корректности.

— HSV и HSL интуитивны для человека, но имеют сингулярности при нулевой насыщенности (Hue становится неопределённым), что усложняет обучение и генерацию «правильного» hue.

— В общедоступных примерах для обучения раскрашиванию изображений чаще встречаются формат Lab, чем другие, что было одним из аргументов, так как упрощало понимание других аналогичных моделей.

2.2.9 Tensorflow

При выборе фреймворка для реализации и обучения нейросети блок автоматической раскраски опирался на следующие соображения:

1. **Начальное требование к C++**
   — Первоначально ошибочно предполагалось, что модель потребуется переносить в C++ для интеграции в существующие высокопроизводительные модули.
   — По ряду статей и обсуждений в профильных сообществах TensorFlow позиционировался как более зрелый вариант для экспорта моделей.
2. **Инвестиции времени на освоение**
   — После выбора TensorFlow команда затратила значительное количество времени на изучение его API, практик и инструментов (TensorBoard).
   — Отказаться от уже наработанных знаний и рефакторить код под другой фреймворк (что требовало бы изучения новой экосистемы) посчитали менее оправданным с точки зрения сроков проекта.
3. **Возможности экспорта и развёртывания**
   — TensorFlow предоставляет официальные инструменты для конвертации моделей в формат TensorFlow Serving, TensorFlow Lite и TensorFlow.js, что позволяет гибко встроить модель в веб-сервис,

мобильные приложения или даже бортовые системы.

— Хотя переход на C++ не понадобился, эта гибкость осталась важным преимуществом на будущее.

4. **Наличие корпоративных интеграций**
   — В организации уже действовали CI/CD-пайплайны и Docker-образы с предустановленным TensorFlow, что упростило настройку окружения для обучения и развёртывания.

Недостатки и сложные моменты

— **Скудность обновлённой документации**
Многие современные статьи, учебные курсы и примеры публикуются преимущественно под PyTorch, а официальная документация TensorFlow иногда оказывается неполной или устаревшей.

— **Менее интуитивный API**
По сравнению с PyTorch, где код обучения модели часто пишется «от руки» в едином стиле, в TensorFlow множество разных подходов (tf.keras, низкоуровневые tf.* API, eager vs graph), что иногда вызывает путаницу.

— **Меньшее комьюнити-движение**
Обсуждения в форумах и новые исследования чаще предлагают примеры сразу на PyTorch, поэтому адаптация свежих идей требовала дополнительных усилий по портированию.

Тем не менее, благодаря уже вложенному времени, команда приняла решение сохранять за нейросетевым блоком использование TensorFlow.

2.2.10 MSE – Mean Squared Error

Для обучения модели использовалась Mean Squared Error (MSE), определяемая как

$$L_{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i^{real} - y_i \right)^2$$

где y_i^{real} и y_i — истинные и предсказанные значения каналов a, b для каждого из N пикселей.

Кроме того, в рамках тестов считались метрики PSNR и MAE — однако они либо прямо, либо косвенно основаны на MSE, поэтому для единообразия и простоты оставили одну функцию потерь.

Плюсы и минусы MSE

— Преимущества

1. Простая и быстрая в вычислении, хорошо дифференцируется.
2. Прозрачное поведение градиентов, стабильная сходимость при нормализации каналов.

— Недостатки

1. Склонность к «усреднению» цвета: модель стремится выбирать средний оттенок для каждого объекта, что приводит к блеклым, «приглушённым» результатам.
2. Не учитывает перцептуальные особенности: две раскраски, близкие по MSE, могут выглядеть по-человечески очень по-разному.

Рассмотренные альтернативы

1. Перцептуальная функция потерь на базе VGG16

Идея: сравнивать не сами пиксели, а высокоуровневые признаки, извлечённые из предобученной VGG-сети.

Плюс: значительно лучше коррелирует с человеческим восприятием реалистичности цвета.

Минус: Дважды прогонять изображение VGG увеличивает время эпохи обучения.

VGG16 ожидает на вход RGB, а модель выдаёт только каналы a, b в Lab, что потребовало бы дополнительных преобразований и усложнило пайплайн.

2. Классификационный подход (Colorful Image Colorization, Zhang et al.,

2016)

В работе Zhang (2016) предложен принципиально иной подход к раскраске: вместо прямой регрессии «пиксель→(a,b)» они переводят задачу в классификацию цветовых «корзин». Это позволяет модели учиться предсказывать не одну «усреднённую» пару (a,b), а распределение вероятностей по всем возможным цветовым оттенкам — что даёт более насыщенные и правдоподобные цвета.

Преимущества и недостатки подхода

Плюсы:

- Избегает эффекта «усреднения» цвета → более насыщенные, «яркие» результаты.
- Позволяет сети учиться «на распределении», а не на одной точке.

Минусы:

- Нужно готовить дополнительные структуры (список центров бинов, весовые коэффициенты).
- Увеличение числа выходных каналов до Q (например, 313), что растит объём модели и замедляет инференс.
- Более сложная логика на этапе пред- и пост-обработки меток.

Итоговое решение

Из-за ограничений по времени и ресурсов команда оставила MSE в качестве основной функции потерь, при этом понимая, что в будущих итерациях проекта можно будет добавить перцептуальные компоненты или перейти на классификационный подход для более насыщенных и реалистичных цветов.

2.3 Программный комплекс как набор элементов

2.3.1. Задачи API

API (Application Programming Interface) обеспечивает единый интерфейс для взаимодействия пользователей с системой, а также координирует работу всех внутренних компонентов.

Разработка API велась с особым вниманием к надёжности, гибкости и возможности масштабирования, так как именно оно является «воротами» между внешним миром (клиентами, пользователями) и внутренними сервисами: нейронной сетью, базой данных, объектным хранилищем.

В этом разделе подробно описаны **основные задачи API**, которые были поставлены при проектировании и разработке, а также **технические решения**, которые позволили их успешно реализовать.

2.3.1.1 Приём и обработка пользовательских запросов

Первая и основополагающая задача API — это **приём входящих запросов** от пользователей. Пользователи взаимодействуют с веб-интерфейсом, загружая свои чёрно-белые фотографии. API выступает в роли первого «контролёра» — принимает эти изображения, проверяет их корректность и подготавливает данные для дальнейшей обработки.

Чтобы реализовать эту задачу, в проекте используется **FastAPI** — современный веб-фреймворк, который обеспечивает:

- **Высокую производительность**: асинхронная обработка запросов — одно из ключевых преимуществ FastAPI. Она реализуется благодаря стандарту ASGI (Asynchronous Server Gateway Interface), который позволяет обрабатывать множество запросов одновременно, не блокируя основной поток выполнения.
  В традиционных синхронных приложениях (например, на базе WSGI) каждый входящий запрос «захватывает» один поток или процесс до окончания обработки, что приводит к быстрому исчерпанию ресурсов

при большом числе пользователей. FastAPI, напротив, позволяет параллельно обслуживать десятки и сотни пользователей, переключаясь между задачами и поддерживая быстрый отклик, даже если у других пользователей «висят» тяжёлые запросы. Это особенно важно для проектов, где могут одновременно загружать десятки больших изображений.

- **Удобство разработки**: FastAPI автоматически формирует документацию для всех эндпоинтов, используя стандарт **OpenAPI** (ранее известный как Swagger). Это значит, что каждый разработчик или внешний интегратор сразу получает доступ к интерактивной документации: Список всех методов (GET, POST, …), описание каждого параметра, возможность «пощупать» API прямо из браузера (Swagger UI), что ускоряет командную работу, позволяет быстрее находить ошибки и улучшает взаимодействие с другими разработчиками.
- **Гибкость интеграции**: FastAPI легко взаимодействует с другими ключевыми компонентами архитектуры:

RabbitMQ: через библиотеки (например, pika или Celery), API может быстро ставить задачи в очередь сообщений, передавая их Celery-воркерам для дальнейшей обработки.

MinIO: библиотека boto3 или встроенные Python-обёртки для S3 позволяют API загружать и скачивать файлы из хранилища, делая работу с большими объектами удобной и безопасной.
Такое разделение позволяет развивать каждый компонент (FastAPI, Celery, MinIO) независимо, сохраняя при этом общую совместимость.
Пример простого эндпоинта приёма запроса в FastAPI (фрагмент, адаптированный из проекта):

```
from fastapi import FastAPI, UploadFile, File, HTTPException
```

```
app = FastAPI()

@app.post("/upload/")
async def upload_image(file: UploadFile = File(...)):
    if not file.content_type.startswith("image/"):
         raise HTTPException(status_code=400, detail="Файл не
является изображением.")

    contents = await file.read()
     # Дальнейшая логика: создание задачи, сохранение файла и
т.д.
    return {"status": "Файл получен, задача создана."}
```

Такой подход позволяет API быстро «обслуживать» каждого пользователя, не перегружая систему.

2.3.1.2. Валидация данных и обеспечение безопасности

Вторая важная задача — это **валидация** входных данных. На практике это означает:

- **Проверка, действительно ли файл — это изображение (не вирус или скрипт)**

  В API реализована проверка content_type, которая позволяет понять, что загруженный файл — это именно изображение (например, image/jpeg или image/png), а не вредоносный скрипт, который мог бы «маскироваться» под картинку.

  Это предотвращает распространённые атаки, при которых под видом изображения загружается вредоносный код, который затем может быть выполнен на сервере.

- **Ограничение размера файла**

  Сервер проверяет, чтобы размер загружаемого файла не превышал заданный лимит (например, 10–20 МБ). Это важно для предотвращения попыток «завалить» сервер огромными файлами, что может привести к исчерпанию памяти и «падению» всего приложения.

  В FastAPI для этого используются встроенные проверки и можно установить соответствующие ограничения.

- **Проверка правильности структуры запроса**
  FastAPI и Pydantic позволяют автоматически проверять, что все необходимые параметры (например, заголовки Content-Type, имя файла) переданы корректно.
  Если пользователь отправляет «ломаный» или некорректный запрос (например, не указывает обязательные поля), API сразу возвращает ошибку с пояснением, не передавая задачу дальше.

В проекте для этого используется встроенный механизм FastAPI — **Pydantic-модели**, которые позволяют точно определить, какие поля ожидаются в запросе, а также валидировать их автоматически.
Пример:

```
from pydantic import BaseModel

class UploadImageRequest(BaseModel):
    filename: str
    filetype: str
```

Валидация данных — это не только удобство, но и **ключевая часть безопасности** API. Любые невалидные или подозрительные файлы сразу отбрасываются, предотвращая возможность атак (например, через загрузку скриптов под видом изображений).

2.3.1.3. Формирование задачи для последующей обработки

После успешной валидации загруженного файла API должно подготовить **задачу для обработки**.

Эта задача включает в себя:

- Параметры (имя файла, ID пользователя, дополнительные опции),
- Метаданные (дата загрузки, предполагаемый результат),

- Уникальный идентификатор (task_id), который позволяет отслеживать статус в будущем.

  Формирование задачи — это не только запись информации в базу данных

PostgreSQL, но и **передача этой задачи** в очередь сообщений RabbitMQ. Такая схема даёт возможность «разгрузить» FastAPI: он больше не занимается тяжёлыми вычислениями, а просто «ставит задачу в очередь».

Архитектурно, это выглядит так:

- FastAPI = диспетчер, который всё организует,
- Celery-воркеры = исполнители, которые делают сложную работу.

2.3.1.4. Работа с базой данных для учёта задач

Для хранения информации о каждой загруженной фотографии и отслеживания её статуса используется база данных **PostgreSQL**.
API отвечает за:

- **Создание записей с уникальным ID**

  Каждый запрос получает уникальный ID (например, task_id: "abc123"). Это позволяет пользователю или разработчику потом получить точный статус своей задачи, а также связать все данные (имя файла, время загрузки, результат).
- **Обновление статусов (PENDING, IN_PROGRESS, COMPLETED, FAILED)**

  Это необходимо, чтобы пользователь мог видеть текущее состояние:
  - «В ожидании» — задача поставлена в очередь,
  - «В процессе» — её сейчас обрабатывает воркер,
  - «Завершена» — можно получить результат,
  - «Ошибка» — задача не выполнена, можно повторить.
- Такая схема прозрачна и понятна, позволяет легко управлять большим количеством запросов.

- **Хранение ссылок на результат (MinIO)**

  После завершения обработки Celery-воркер загружает готовый цветной файл в хранилище (MinIO).

  API сохраняет ссылку на этот файл в базе данных, чтобы пользователь мог потом получить прямую ссылку, не перегружая API.

  Это снижает нагрузку на веб-сервер (не нужно «таскать» большие файлы), а пользователь получает гарантированный доступ к своему результату.

  Пример модели в SQLAlchemy (models.py):

```
class ImageTask(Base):
    __tablename__ = "image_tasks"
    id = Column(Integer, primary_key=True, index=True)
    filename = Column(String, nullable=False)
    status = Column(String, default="PENDING")
    created_at = Column(DateTime, default=datetime.utcnow)
    result_url = Column(String, nullable=True)
```

Это позволяет API быстро и надёжно работать с историей запросов каждого пользователя.

2.3.1.5. Предоставление доступа к результатам

После того как задача окрашивания выполнена (нейросеть закончила свою работу), API получает уведомление (обновление статуса в БД) и предоставляет пользователю ссылку на готовое цветное изображение.
Для этого используется хранилище **MinIO** — аналог S3, совместимый с AWS API, позволяющий безопасно и удобно хранить большие файлы.

2.3.1.6. Мониторинг, логирование и надёжность

Одна из важнейших задач API — это **мониторинг работы** и **ведение логов**.
API фиксирует все ключевые события:

- Время и объём загружаемых файлов,
- Ошибки (если файл не подходит или задача не выполнена),
- Успешные загрузки и обработки.

Для логирования используется модуль logging, который позволяет:

- Писать логи в файлы,
- Выводить важные ошибки в консоль (удобно при отладке),
- Настраивать уровни сообщений (INFO, WARNING, ERROR).

Таким образом, API — это не просто точка приёма запросов. Оно объединяет в себе:

- Валидацию и проверку безопасности,
- Гибкость формирования задач,
- Интеграцию с RabbitMQ и Celery,
- Хранение и выдачу результатов пользователям,
- Полный цикл «жизни» запроса от начала до конца.

Эта многослойная структура обеспечивает **надёжность**, **масштабируемость** и **удобство использования** — ключевые качества, необходимые для успешного решения задачи восстановления цвета фотографий в большом проекте.

2.3.2.1 Разделение FastAPI и отдельного Celery-приложения

- FastAPI, выполняет функции веб-интерфейса и «диспетчера» запросов,
- Celery-приложение отвечает за выполнение ресурсоёмких задач — непосредственно окрашивание фотографий.
- Это разделение — не просто техническая «фишка», а важное инженерное решение, направленное на достижение высокой надёжности, масштабируемости и отказоустойчивости всего проекта.

2.3.2.2 Принципы взаимодействия между FastAPI и Celery

FastAPI не занимается реальной обработкой изображений. Его задачи:

- Проверить корректность загружаемых данных (валидировать).
- Записать информацию о задаче в базу данных PostgreSQL (уникальный ID, статус PENDING).
- Передать задачу в очередь сообщений RabbitMQ, указав все необходимые параметры (имя файла, путь, ID задачи).
- Мгновенно вернуть пользователю ответ: «Задача создана, обработка начнётся в ближайшее время».

Celery-воркеры — это отдельные процессы, которые «слушают» очередь RabbitMQ. Как только в очередь приходит новая задача:

- Воркер берёт её в работу.
- Загружает исходное изображение из хранилища MinIO.
- Запускает нейросеть (C++ модель) для восстановления цвета.
- Сохраняет готовый результат в MinIO.
- Обновляет запись в базе данных — статус COMPLETED или FAILED и ссылка на готовый результат.

Такое взаимодействие делает систему гибкой: FastAPI не блокируется, а Celery можно масштабировать (запускать больше воркеров для роста нагрузки).

2.3.2 Celery

Архитектура проекта построена по принципу разделения ответственности и асинхронного взаимодействия между компонентами. Центральным элементом системы является FastAPI-приложение, которое выполняет роль «фронт-энда» для API и основной бизнес-логики. Однако, для выполнения ресурсоемких, медленных или фоновых операций, таких как загрузка и обработка изображений, а также обновление записей в базе данных,

используется система фоновых задач Celery, работающая в связке с RabbitMQ как брокером сообщений.

Общая схема взаимодействия компонентов

- Пользователь взаимодействует с FastAPI через HTTP-интерфейс.
- FastAPI-логика принимает входящие данные (например, изображение) и может сохранить их в S3-хранилище.
- Вместо того чтобы обрабатывать изображение непосредственно в рамках того же запроса, FastAPI добавляет задачу в очередь RabbitMQ.
- Celery worker, запущенный в отдельном процессе (или контейнере), подписан на эту очередь и получает задачу.
- Worker обрабатывает задачу, производит необходимые действия (инверсия изображения, сохранение результатов, обновление БД и т.д.).
- При необходимости результат возвращается либо в backend Celery (в нашем случае — rpc://), либо сохраняется в базу данных или объектное хранилище.

Это взаимодействие организовано через стандартный и хорошо зарекомендовавший себя стек: **Celery + RabbitMQ + FastAPI**. Такая схема обеспечивает **масштабируемость, устойчивость к отказам и гибкость в обработке большого количества фоновых задач**.

Все основные настройки инициализации Celery сосредоточены в файле celery_app.py, который размещается в корневом пакете app/. Его основная задача — создать и настроить экземпляр Celery-приложения, которое затем может использоваться как в самих задачах, так и при запуске worker-процессов.

В рамках проекта файл celery_app.py играет ключевую роль в настройке и инициализации системы фоновых задач. Здесь создаётся экземпляр объекта Celery, который фактически является ядром всей архитектуры распределённой

обработки. Именно через него осуществляется регистрация задач, управление очередями и настройка взаимодействия с брокером сообщений и системой хранения результатов.

Экземпляр Celery получает логическое имя "worker" — оно не влияет на маршрутизацию задач, но может быть полезно для целей мониторинга и отладки, особенно при наличии нескольких worker-процессов. Одним из важнейших параметров при создании объекта Celery является broker. В данном проекте используется строка подключения amqp://guest:guest@rabbitmq:5672//, указывающая на то, что в качестве брокера используется RabbitMQ, работающий по протоколу AMQP. Аутентификация осуществляется по умолчанию через логин и пароль guest, а порт 5672 является стандартным для AMQP-соединений. Именно через этот канал Celery получает доступ к очередям задач, отправляет в них задания и получает уведомления о необходимости выполнения той или иной функции.

Параметр backend, заданный как rpc://, определяет способ получения результатов выполнения задач. Это встроенный и достаточно простой backend, работающий поверх всё того же AMQP. Он позволяет, при необходимости, возвращать результат выполнения задачи обратно в вызывающее приложение. Однако в текущем проекте основная логика построена иначе: после выполнения задачи результат сохраняется в S3-хранилище (MinIO) и база данных обновляется соответствующей записью. Таким образом, роль backend в данном случае ограничена, и использование rpc:// остаётся скорее технической необходимостью, чем архитектурной опорой.

Наконец, немаловажной строкой является import app.tasks. Несмотря на свою простоту, она выполняет критически важную функцию — гарантирует регистрацию всех задач, объявленных в модуле tasks.py. Celery автоматически обнаруживает только те задачи, которые явно загружены или импортированы в

момент инициализации. Без этой строки система попросту не будет знать о существовании фоновых функций, оформленных с помощью декоратора @celery.task, и, соответственно, не сможет их выполнять. Поэтому даже такой, казалось бы, незначительный импорт оказывает прямое влияние на функциональность всей подсистемы Celery.

Интеграция Celery в проекте

Интеграция Celery с другими компонентами проекта реализована таким образом, чтобы обеспечить тесную и стабильную связность между различными подсистемами приложения. Несмотря на то, что основной задачей Celery является асинхронная обработка задач, он активно взаимодействует не только с FastAPI и RabbitMQ, но и с рядом других критически важных компонентов, обеспечивая выполнение сложной логики в фоне.

Одним из таких компонентов является объектное хранилище MinIO, которое используется как S3-совместимая система хранения файлов. Celery worker, получая задачу, часто должен загрузить или сохранить изображение. Для этого он напрямую обращается к MinIO, используя параметры подключения, заданные через переменные окружения — такие как URL, ключ доступа и секретный ключ. Эти параметры передаются в контейнер worker'а через docker-compose.yml и позволяют безопасно и удобно обращаться к хранилищу без хардкода. MinIO выступает здесь как внешняя система, обеспечивающая быстрый и надёжный доступ к исходным и результирующим изображениям, с которыми работает обработчик задач.

Ещё одним важным направлением интеграции является база данных PostgreSQL, доступ к которой осуществляется через ORM-библиотеку SQLAlchemy. После завершения обработки изображения — например, инверсии цветов — задача должна не только сохранить файл, но и обновить соответствующую запись в базе данных. Это необходимо для того, чтобы пользователь мог получить доступ к результату через API или пользовательский

интерфейс. Для этого Celery worker создаёт подключение к базе данных, находит нужную запись по s3_key, и дополняет её новым значением поля, указывающим на результирующий файл. После этого изменения фиксируются коммитом, и соединение с базой закрывается. Такой подход обеспечивает согласованность данных и позволяет связать результат фоновой задачи с пользовательскими данными.

Наконец, вся архитектура построена с использованием Docker, что обеспечивает изоляцию компонентов и облегчает их масштабирование и развёртывание. В docker-compose.yml каждый из ключевых элементов системы — FastAPI-приложение, Celery worker, RabbitMQ, MinIO и PostgreSQL — запускается в отдельном контейнере. Это позволяет не только изолировать окружения и зависимости, но и гибко управлять ресурсами. Например, при росте нагрузки можно запустить несколько экземпляров worker'ов без необходимости изменять основной код. Такая организация упрощает развёртывание в разных средах, обеспечивает предсказуемость конфигурации и значительно облегчает сопровождение системы в целом.

Таким образом, Celery встроен в инфраструктуру проекта как полноценный участник, активно взаимодействующий с внешними хранилищами, базой данных и другими сервисами, что делает его не просто инструментом для фоновых задач, а неотъемлемой частью всей архитектурной конструкции.

2.3.3 RabbitMQ & AMQP

2.3.3.1 Интеграция и реализация

В рамках реализации проекта RabbitMQ разворачивается как отдельный, полностью изолированный сервис внутри Docker-окружения, что обеспечивает

надёжность, стабильность и простоту управления этим ключевым компонентом инфраструктуры. Конфигурация RabbitMQ осуществляется через файл docker-compose.yml, где в качестве образа используется официальный и широко проверенный образ rabbitmq:3-management. Этот образ не только включает сам брокер сообщений RabbitMQ, но и содержит встроенный веб-интерфейс администрирования — RabbitMQ Management Plugin, который значительно облегчает мониторинг и управление системой в реальном времени.

В настройках сервиса в docker-compose.yml открываются два важных порта. Первый — 5672, являющийся стандартным портом для работы по протоколу AMQP (Advanced Message Queuing Protocol). Именно через этот порт Celery устанавливает соединение с брокером RabbitMQ и отправляет задачи в очередь. Второй порт — 15672, предназначенный для доступа к веб-интерфейсу управления. Этот интерфейс предоставляет удобное визуальное представление о состоянии очередей, количестве сообщений, активности подключённых клиентов, а также доступ к статистическим данным и логам, что является незаменимым инструментом для оперативного мониторинга и диагностики системы.

С точки зрения интеграции с Celery, в проекте используется явная и прозрачная настройка подключения к RabbitMQ. В файле celery_app.py при создании объекта Celery параметр broker принимает строку подключения следующего вида:

Данная строка указывает, что Celery подключается к брокеру RabbitMQ, который доступен по имени контейнера rabbitmq в Docker-сети, используя протокол AMQP и стандартные учетные данные guest:guest. Такой подход обеспечивает единую и устойчивую точку доступа к брокеру в рамках контейнерного окружения без необходимости жестко прописывать IP-адреса, что облегчает масштабирование и переносимость проекта.

Важным аспектом данной интеграции является способность системы сохранять задачи в очереди даже в случае временной недоступности воркеров Celery. FastAPI, выступая в роли производителя задач, публикует задачи в очередь RabbitMQ мгновенно, не блокируя основной поток и обеспечивая быструю реакцию на запросы пользователя. Если же воркер в момент публикации задачи недоступен — например, из-за перезапуска, высокой нагрузки или обновления — задача не теряется, а надёжно хранится в очереди до восстановления воркера. После того, как Celery worker подключается к RabbitMQ, брокер передаёт ему все накопившиеся задачи для обработки. Это обеспечивает гарантированную доставку и выполнение, что критично для бизнес-логики проекта и положительного пользовательского опыта.

Кроме того, использование RabbitMQ как отдельного сервиса в Docker-контейнере позволяет легко масштабировать систему. При увеличении нагрузки достаточно запустить дополнительные экземпляры Celery-воркеров, которые автоматически подключатся к брокеру и начнут забирать задачи из очереди. Это горизонтальное масштабирование повышает производительность и устойчивость всей системы без сложных изменений в коде.

Наконец, благодаря наличию веб-интерфейса RabbitMQ администраторы и разработчики могут получать оперативную информацию о текущем состоянии очередей, анализировать задержки в обработке задач, выявлять потенциальные узкие места и ошибки. Это позволяет быстро реагировать на возникающие проблемы, проводить профилактическое обслуживание и оптимизацию, обеспечивая стабильную и эффективную работу распределённой системы.

Таким образом, использование RabbitMQ в качестве центра обмена

сообщениями, развёрнутого в изолированном Docker-контейнере с мощным инструментарием управления и мониторинга, становится краеугольным камнем архитектуры проекта. Это решение обеспечивает надёжность, масштабируемость и гибкость, позволяя эффективно интегрировать FastAPI и Celery в единую систему обработки асинхронных задач, а также поддерживать высокий уровень отказоустойчивости и производительности.

2.3.3.2 Протокол AMQP

AMQP (Advanced Message Queuing Protocol) — это открытый сетевой протокол прикладного уровня, предназначенный для надёжной и масштабируемой передачи сообщений между компонентами распределённых систем. Его задача — обеспечить гарантированную доставку сообщений, независимость отправителя и получателя, устойчивость к сбоям и гибкую маршрутизацию сообщений. В рамках проекта AMQP используется как основной механизм взаимодействия между FastAPI и Celery через брокер RabbitMQ.

В отличие от простых очередей, AMQP реализует полноценную модель обмена, в которой производители сообщений (например, API FastAPI) публикуют задачи в брокер, а потребители (воркеры Celery) получают и обрабатывают их. Между ними находится очередь сообщений, но перед этим сообщение проходит через специальный компонент — обменник (exchange), который по заданным правилам определяет, в какую очередь направить сообщение. Такая структура позволяет выстраивать сложные схемы маршрутизации задач, разделять очереди по назначению и типу, а также балансировать нагрузку между воркерами.

Важной особенностью AMQP является гарантия доставки. Сообщение не считается обработанным, пока потребитель (воркер Celery) не подтвердит его

получение. Если воркер завершает работу с ошибкой или теряет соединение, задача остаётся в очереди, и RabbitMQ может повторно её выдать, как только появится доступный потребитель. Это делает систему устойчивой к временным сбоям, позволяя компонентам работать независимо друг от друга. Кроме того, очереди и сообщения могут быть помечены как устойчивые, то есть сохраняются на диск и не теряются даже при перезапуске брокера.

Ещё одно достоинство AMQP — гибкость. В случае необходимости можно реализовать распределённую систему, где разные типы задач попадают в разные очереди, обрабатываются разными воркерами, масштабируются отдельно и отслеживаются независимо. Это особенно важно в микросервисной архитектуре, где каждый компонент должен быть как можно более автономным.

В проекте Celery подключается к RabbitMQ с использованием адреса amqp://guest:guest@rabbitmq:5672//, указывающего на протокол, логин и адрес брокера. Все сообщения, то есть задачи, передаются именно по AMQP, что гарантирует их надёжную доставку. Благодаря этому FastAPI может мгновенно реагировать на запрос пользователя и сразу передавать задачу в очередь, не дожидаясь её выполнения. Celery в свою очередь обрабатывает задачи асинхронно, при этом надёжно получая их из очереди RabbitMQ.

Использование AMQP через RabbitMQ даёт системе устойчивость, масштабируемость и предсказуемость. Даже при увеличении нагрузки или частичных сбоях отдельные задачи не теряются, а обработка продолжается, как только восстанавливается доступ к нужным компонентам. В результате AMQP становится основой надёжной и отказоустойчивой архитектуры, позволяющей чётко разделять обязанности между частями системы и гибко управлять процессом обработки задач.

Связка Celery и RabbitMQ в проекте

В связке Celery и RabbitMQ реализуется устойчивая, масштабируемая и асинхронная архитектура обработки задач, обеспечивающая высокую отказоустойчивость и гибкость. Такой подход особенно эффективен в системах с ресурсоёмкими и длительными операциями, например, при работе с изображениями, файлами или базами данных. Процесс начинается с того, что пользователь отправляет изображение через HTTP-запрос в FastAPI-приложение. Вместо синхронной обработки API формирует задачу, описывающую необходимые действия, и передаёт её Celery через вызов delay(), тем самым отправляя задачу в очередь RabbitMQ.

Задача сериализуется и передаётся брокеру RabbitMQ, который играет роль буфера между API и обработчиком. Он маршрутизирует и хранит сообщение до его обработки Celery-воркером, подписанным на соответствующую очередь. Воркер извлекает задачу, десериализует и запускает выполнение. Все ошибки логируются, а статусы задач можно отслеживать через интерфейс RabbitMQ. Надёжность доставки обеспечивается за счёт протокола AMQP, который гарантирует, что ни одна задача не будет потеряна: сообщение удаляется из очереди только после подтверждённой обработки. При сбое оно возвращается в очередь или перемещается в очередь ошибок.

RabbitMQ также поддерживает гибкую маршрутизацию задач, позволяя распределять их по разным очередям в зависимости от типа или приоритета. Это даёт возможность горизонтального масштабирования: разные воркеры могут специализироваться на различных задачах, что оптимизирует использование ресурсов. При этом компоненты остаются независимыми: FastAPI лишь инициирует задачу, а выполнение полностью передаётся Celery и

RabbitMQ. Даже если воркеры временно недоступны, API продолжит работу, а задачи сохранятся в очереди до восстановления обработки.

Таким образом, Celery и RabbitMQ формируют надёжную архитектурную основу, позволяющую разгружать основное приложение, эффективно управлять фоновыми задачами и обеспечивать гарантированное выполнение операций. Этот подход успешно применяется в промышленных системах, где критична устойчивость, масштабируемость и чёткое управление асинхронными процессами.Проблемы и сложности

В процессе разработки и эксплуатации системы с использованием RabbitMQ, несмотря на его высокую надёжность, широкое распространение и проверенную временем архитектуру, возникли ряд сложностей и особенностей, которые потребовали дополнительного внимания, настройки и доработок. Одной из наиболее частых проблем стала организация правильного порядка запуска сервисов в Docker-окружении. Поскольку RabbitMQ выступает в роли брокера сообщений и запускается как отдельный контейнер, а Celery-воркеры — как другие контейнеры, необходимо, чтобы воркеры подключались к полностью готовому и работающему брокеру. В ряде случаев наблюдалась ситуация, когда Celery-воркер стартовал раньше, чем RabbitMQ успевал полноценно инициализироваться и начать принимать соединения. В результате воркер не мог установить соединение с брокером и завершал работу с ошибкой, что приводило к сбоям в обработке задач и необходимости ручного или автоматического перезапуска воркеров.

Для частичного решения этой проблемы была использована директива depends_on в файле docker-compose.yml, которая позволяет задать порядок запуска контейнеров и указать, что один сервис должен стартовать после другого. Однако важно понимать, что depends_on контролирует только

порядок запуска контейнеров, но не гарантирует, что сервис внутри контейнера будет готов к работе, то есть будет прослушивать необходимый порт и принимать подключения. В связи с этим на практике стало целесообразным внедрить дополнительный механизм проверки готовности брокера RabbitMQ — например, скрипт, который перед запуском Celery-воркера выполняет циклическую проверку доступности порта RabbitMQ (используя утилиты вроде wait-for-it, dockerize или собственные bash-скрипты). Такой подход позволяет гарантировать, что воркер начнёт работу только тогда, когда брокер полностью инициализирован и готов к приёму сообщений, что повышает надёжность и стабильность запуска всей системы.

Ещё одной значимой проблемой стала специфика протокола AMQP, лежащего в основе RabbitMQ, и его ограничение на передачу объёма данных. AMQP превосходно справляется с маршрутизацией, гарантированной доставкой и управлением очередями, однако он не предназначен для передачи больших бинарных объектов, таких как изображения или видеофайлы. При попытке инлайн-передавать данные изображений в теле задач Celery в качестве сообщений в RabbitMQ возникали проблемы с производительностью, задержками и даже превышением максимально допустимого размера сообщений. Особенно это сказывалось при одновременной обработке множества задач, что вызывало перегрузку брокера и падение производительности всей системы.

Для решения данной задачи в архитектуре проекта было принято взвешенное и оптимальное решение — в качестве данных, передаваемых в задачах, использовать не сами изображения, а ссылки на них, представленные ключами (ключами объектов) в S3-совместимом объектном хранилище MinIO. Таким образом, Celery-воркеры получают минимальный набор информации — указание на конкретный файл, который нужно загрузить из MinIO и обработать.

Обработка и сохранение файлов происходит напрямую с MinIO, минуя RabbitMQ. Этот подход позволил значительно снизить нагрузку на брокер сообщений, уменьшить задержки и повысить общую стабильность и масштабируемость системы. Кроме того, хранение данных в объектном хранилище обеспечивает долговременное и надёжное хранение больших объёмов данных вне брокера, что упрощает резервное копирование и восстановление.

2.3.4 Пользовательский интерфейс

Разработка веб-приложения "Has-Been" для восстановления цвета черно-белых фотографий с использованием методов машинного обучения представляет собой сложный процесс, требующий интеграции передовых технологий и внимательного подхода к проектированию пользовательского опыта (UX) и пользовательского интерфейса (UI). Эти аспекты стали основополагающими для успеха проекта, поскольку платформа ориентирована на разнообразную аудиторию: от обычных пользователей, стремящихся оживить семейные архивы, до профессиональных историков и архивистов, для которых важны точность и сохранение исторического контекста. Целью UX/UI стало создание интерфейса, который обеспечивает интуитивное, удобное и эффективное взаимодействие, минимизируя когнитивную нагрузку и делая процесс прозрачным даже для людей без технических знаний.

Подходы UX/UI в данном проекте строились на основе тщательного анализа потребностей целевой аудитории, изучения конкурентных решений (таких как DeOldify[18], Colourise.sg[28] и MyHeritage InColor[29]) и учета современных тенденций в веб-дизайне. Стоит отметить, что успех платформы зависит не только от качества алгоритмов обработки, но и от того, насколько

удобно и приятно пользователям работать с интерфейсом "Has-Been". Для демонстрации эволюции дизайна были созданы два макета в Figma: старый (темно-серый) и новый (фиолетовый), которые отражают переход от базовой функциональности к комплексному и эстетически привлекательному решению.

2.3.4.1 Цель использования подходов UX/UI для «Has-Been»

UX (User eXperience) сосредотачивается на общем впечатлении пользователя от взаимодействия с продуктом "Has-Been", включая удобство, эмоциональную вовлеченность и уровень удовлетворенности. UI (User Interface) отвечает за визуальную и функциональную составляющую, включая расположение кнопок, цветовую палитру, интерактивные элементы и общую эстетику. В контексте "Has-Been" эти подходы имеют критическое значение по следующим причинам:

- **Доступность для широкой аудитории**

Проект рассчитан на пользователей с разным уровнем технической подготовки. Например, энтузиасты семейной истории могут не понимать сложностей работы с нейронными сетями, поэтому интерфейс "Has-Been" должен быть простым и понятным с первого взгляда. Это требует исключения технического жаргона и упрощения навигации.

- **Эффективность обработки**

Процесс загрузки, обработки и скачивания изображений в "Has-Been" должен быть оптимизирован, чтобы пользователи не испытывали раздражения из-за задержек. Визуальная обратная связь, такая как прогресс-бары и индикаторы статуса, помогает пользователям понимать этапы работы системы.

- **Конкуренция на рынке**

Анализ конкурентных платформ показал, что их успех во многом зависит

от удобства интерфейса. Например, Colourise.sg привлекает пользователей простотой, а MyHeritage InColor — интеграцией с генеалогическими инструментами. "Has-Been" должен выделяться уникальным дизайном и функциональностью, что требует продуманного подхода к UX/UI.

Таким образом, подходы UX/UI стали фундаментом для создания продукта "Has-Been", который сочетает техническую мощь с эмоциональной и практической ценностью для пользователей.

2.3.4.2 Сравнение старого и нового макетов в Figma

Для проекта «Has-Been» были созданы два макета в Figma: старый (темно-серый) и новый (фиолетовый). Каждый из них включает в себя 2 страницы – главная, где пользователь загружает изображения для желаемой обработки, а также может ознакомиться с информацией о компании, и загрузки результатов, где пользователь может загрузить как все обработанные изображения сразу, так и по отдельности. Для аргументации выбора определенного макета, необходимо провести тщательный и детальный анализ каждого, а также провести сравнительный анализ этих двух макетов, включая такие аспекты как: общая структура и компоновка, цветовая палитра и визуальное восприятие, информативность и обратная связь, функциональность и поддержка аудитории, эмоциональное воздействие. Проведем сравнительный анализ этих двух макетов, начиная с основной страницы, позволяющей пользователю загружать изображения. На рис. 3, рис. 4 представлена главная страница и страница загрузки результатов первого разработанного макета, а на рис. 5, рис. 6 представлена главная страница и страница загрузки результатов макета, который был выбран.

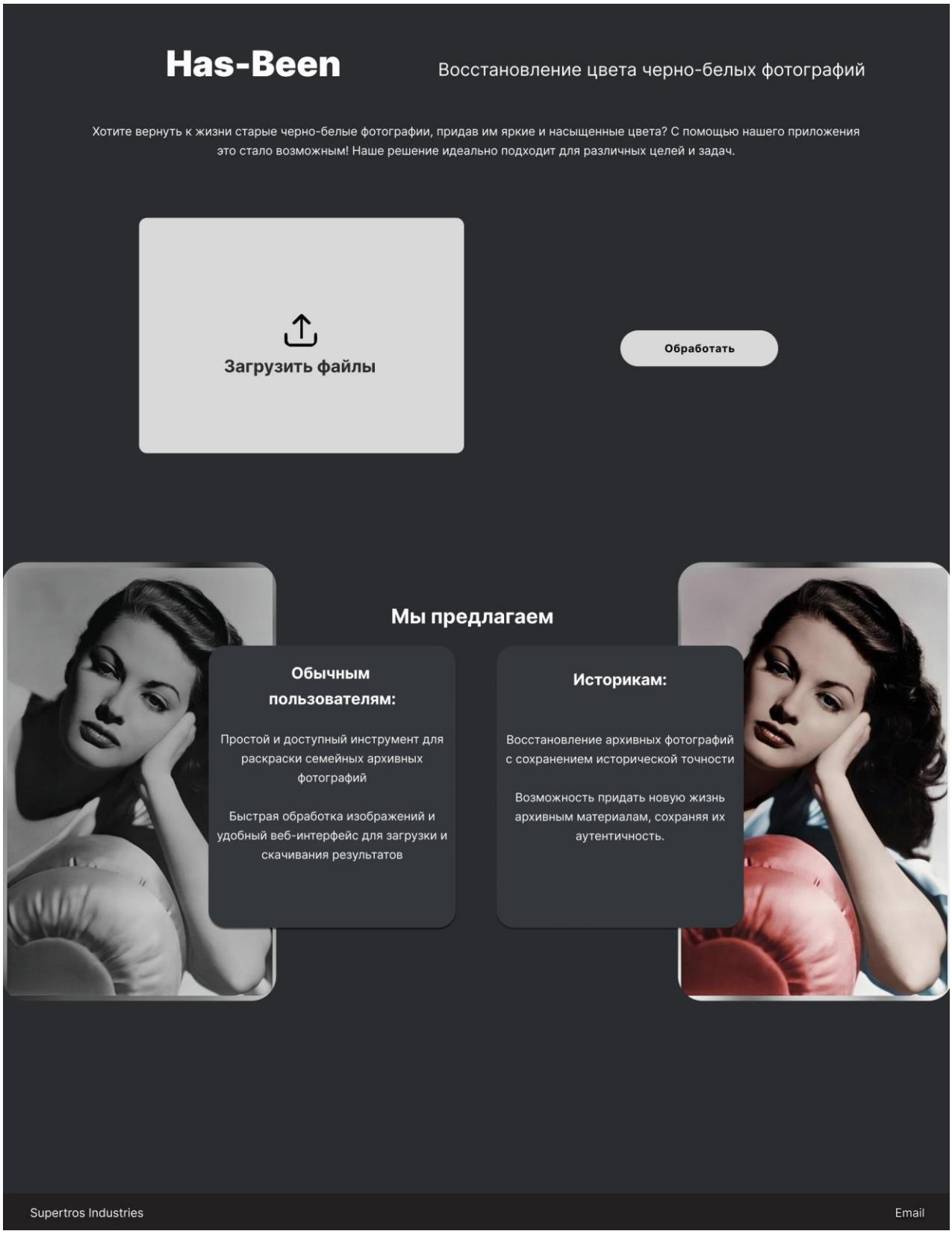

Рисунок 3 - Главная страница старого макета

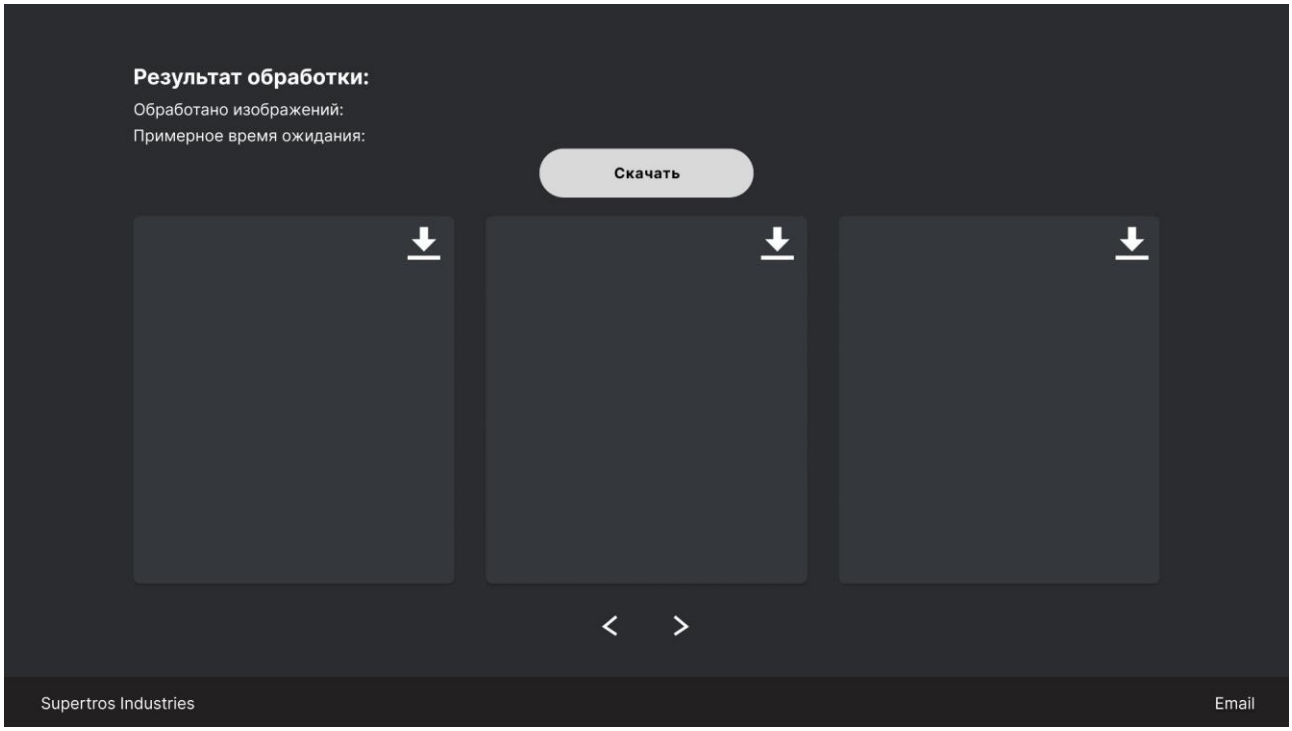

Рисунок 4 - Страница выгрузки результата старого макета

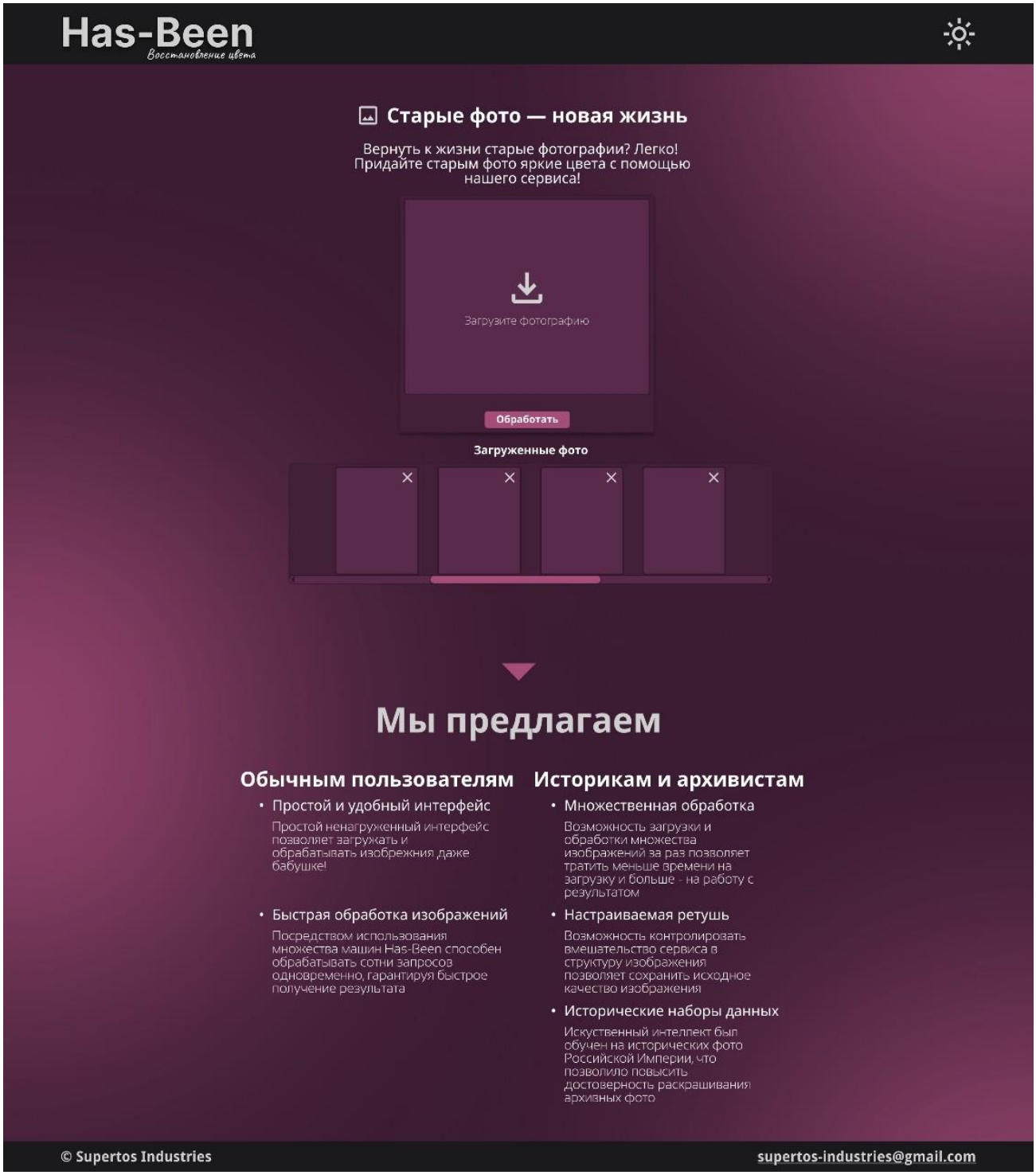

Рисунок 5 - Главная страница нового макета

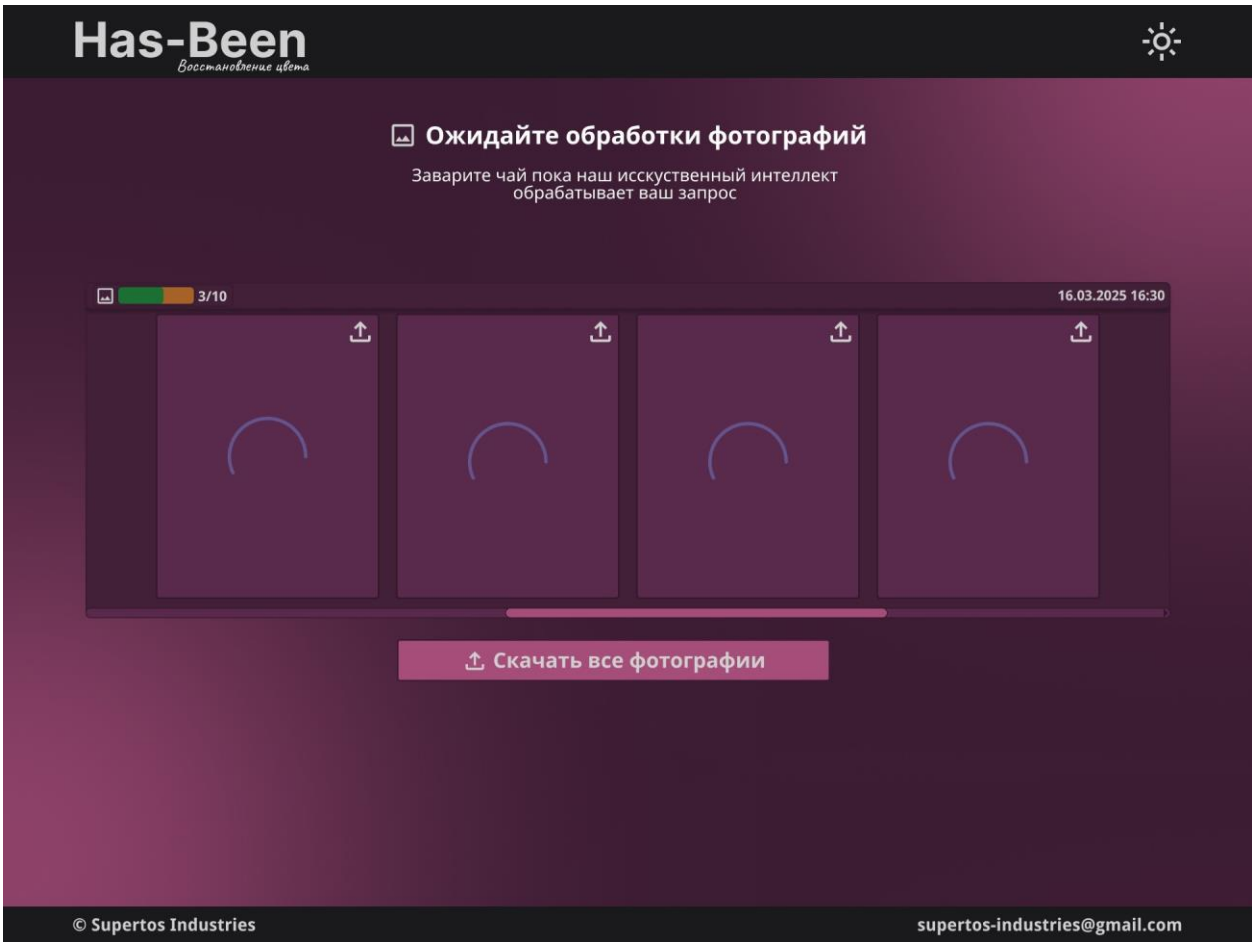

Рисунок 6 - Страница выгрузки результата нового макета

Общая структура и компоновка

Главная страница старого макета имеет минималистичную структуру. Основной элемент — форма загрузки с текстом "Загрузить файлы" и кнопкой "Обработать", расположенной справа. Заголовок "Has-Been: Восстановление цвета черно-белых фотографий" выделен крупным шрифтом, а ниже размещено краткое описание сервиса. В нижней части страницы находятся две секции: "Обычным пользователям" и "Историкам", описывающие преимущества для разных аудиторий. Компоновка строго линейная, без дополнительных визуальных разделителей, что делает страницу плоской и однообразной.

Страница загрузки результатов также минималистична и выполнена в той же стилистике, что и главная страница. На ней присутствует заголовок "Результаты обработки" белым шрифтом на том же темно-сером фоне, с текстом "Обработано изображений” и “Примерное время ожидания:", а также предпросмотр обработанных изображений с навигацией стрелками «вперед» и «назад». Отсутствие визуальных разделителей также делает страницу статичной.

Главная страница нового макета также сохраняет фокус на форме загрузки, но его структура более динамичная. Заголовок "Старые фото — новая жизнь" размещен в верхней части, а ниже — описание сервиса. Форма загрузки теперь оснащена визуальными индикаторами для каждого файла, такими как иконки загрузки и удаления. Нижняя часть страницы также разделена на секции "Общим пользователям" и "Историкам и архивистам", но они визуально выделены с помощью отступов и более контрастного текста.

Страница загрузки результатов, как и главная страница, имеет динамичную структуру. Она обогащена заголовком "Ожидайте обработки фотографий", индикатором прогресса и временной меткой.

Итак, новый макет обеих страниц выигрывает за счет модульной

структуры и визуальной иерархии. Старый макет статичен, с плоским дизайном и ограниченной информативностью, тогда как новый макет использует отступы, градиенты и интерактивные элементы, улучшая восприятие на обеих этапах.

Цветовая палитра и визуальное восприятие

В старом макете используется темно-серый фон с белым текстом и светлыми акцентами, например, поле загрузки изображений или кнопка "Обработать". Эта палитра минималистична, но лишена эмоционального воздействия. Темно-серый фон создает ощущение строгости, что может быть уместно для технического продукта, но не соответствует эмоциональной тематике восстановления старых фотографий. Кроме того, серый фон сливается с черно-белыми изображениями, которые пользователь может загрузить, что снижает контрастность.

В новом макете применяется фиолетовый градиентный фон с розовыми акцентами и белым текстом. Фиолетовый цвет ассоциируется с ностальгией, творчеством и глубиной[30], что идеально подходит для тематики проекта. Розовые акценты выделяют кнопки и прогресс-бар, привлекая внимание к ключевым действиям. Контраст между темным фоном и светлым текстом улучшает читаемость, а градиент добавляет визуальную глубину.

Итак, новый макет выигрывает за счет более выразительной и эмоционально насыщенной палитры. Фиолетовый фон создает атмосферу, связанную с воспоминаниями, а серый фон выглядит устаревшим и не вызывает ассоциаций с темой проекта. Новый дизайн также лучше соответствует современным трендам, где темные темы с яркими акцентами популярны среди пользователей.

Информативность и обратная связь

В старом макете интерактивность главной страницы ограничена базовой формой загрузки. После выбора файлов пользователь не получает визуальной обратной связи, кроме нажатия на кнопку "Обработать". Отсутствие визуальной

обратной связи может вызывать у пользователей неуверенность в том, что система работает.

Аналогично, на странице загрузки результатов нет визуальной обратной связи о статусе завершения процесса, дате и времени обработки.

В новом макете интерактивность значительно улучшена. После загрузки файлов отображаются их миниатюры с возможностью удаления (иконка "X"), что позволяет пользователю убедиться в успешной загрузке изображений для обработки. Это создает ощущение динамичности и прозрачности процесса.

Интерактивность страницы загрузки обработанных изображений нового макете расширена: присутствует прогресс-бар с цветовой кодировкой: зеленый для завершенных, оранжевый для текущих обрабатываемых изображений, что показывает статус обработки каждой загруженной фотографии. Текст «Ожидайте обработки» информирует пользователя, что обработка началась и программа работает корректно. Временная метка добавляет контекст.

Вследствие, новый макет предоставляет гораздо более информативную и интерактивную обратную связь. Пользователь четко видит, что происходит с каждым файлом, что снижает вероятность ошибок и повышает доверие к платформе. Старый макет, напротив, оставляет пользователя в неведении, что может привести к разочарованию.

Функциональность и поддержка аудитории

Старый макет поддерживает базовую загрузку файлов и обработку, но не предоставляет дополнительных функций, таких как массовая загрузка или предпросмотр. Секции "Обычным пользователям" и "Историкам" описывают преимущества, но не содержат конкретных примеров или визуальных элементов, которые могли бы привлечь внимание.

Новый макет расширяет функциональность, позволяя массовую загрузку с возможностью удаления каждого файла. Секции "Общим пользователям" и "Историкам и архивистам" переработаны с учетом потребностей аудитории и

включают более детализированные описания, например, "Быстрая обработка изображений" для пользователей и "Сохранение исторической точности" для архивистов.

Итак, новый макет лучше соответствует разнообразию целевой аудитории "Has-Been". Он предлагает больше функциональных возможностей и делает акцент на потребностях пользователей, тогда как старый макет был ограничен базовыми функциями и не учитывал различия в ожиданиях аудитории.

Эмоциональное воздействие

В старом макете темно-серый дизайн не создает эмоциональной связи с темой проекта. Он выглядит строго техническим, что может отпугнуть пользователей, для которых восстановление фотографий связано с ностальгией и эмоциями. Брендинг "Has-Been" теряется на фоне серого дизайна, который не выделяет платформу среди конкурентов.

В новом макете фиолетовая палитра и градиентный фон усиливают эмоциональное воздействие. Фиолетовый цвет ассоциируется с ностальгией, памятью и творчеством, что идеально подходит для проекта, связанного с восстановлением старых фотографий. Розовые акценты добавляют теплоты и мягкости, создавая ощущение заботы. Логотип "Has-Been" и слоган "Старые фото — новые жизни" лучше интегрированы в дизайн, усиливая брендинг.

С учетом вышесказанного, новый макет создает более сильную эмоциональную связь и лучше отражает суть проекта. Старый макет, напротив, не передает атмосферу ностальгии, что снижает его привлекательность.

Сравнительный анализ старого и нового макетов страниц загрузки и скачивания обработанных изображений для веб-приложения "Has-Been" показал, что новый макет значительно превосходит старый по всем ключевым аспектам. Он обеспечивает более высокое удобство использования,

современную эстетику и конкурентоспособность, что делает его оптимальным выбором для реализации целей проекта, ориентированного на широкую аудиторию — от обычных пользователей до профессиональных историков и архивистов.

Новый макет выделяется продуманной структурой, выразительной фиолетовой палитрой, создающей эмоциональную связь с темой восстановления фотографий, а также улучшенной интерактивностью, адаптивностью и функциональностью. Эти преимущества позволяют платформе "Has-Been" выделиться среди конкурентов, таких как Colourise.sg или MyHeritage InColor, и обеспечивают более качественный пользовательский опыт. Старый макет, несмотря на свою функциональность как прототипа, был ограничен минималистичным подходом, который не соответствовал современным требованиям и ожиданиям аудитории.

2.3.4.3 Адаптивность макета «Has-Been»

Адаптивность интерфейса является одним из ключевых факторов успеха веб-приложения "Has-Been". Макет, выбранный для реализации проекта (см. рис.5-6), был разработан с учетом современных стандартов веб-дизайна, чтобы обеспечить комфортное использование на устройствах с различными разрешениями экрана — от мобильных телефонов до настольных компьютеров. Было уделено особое внимание адаптивности, чтобы платформа была доступна для широкой аудитории, включая пользователей с разными техническими возможностями и предпочтениями. В этой главе рассматривается, как новый макет обеспечивает адаптивность на страницах загрузки и скачивания обработанных изображений, а также какие подходы и технологии были использованы для достижения этого.

Макет для мобильных устройств (см рис.7-8) разработан с

использованием принципов отзывчивого дизайна, чтобы обеспечить плавное взаимодействие при ширине экрана 700px и ниже. Рассмотрим основные подходы, которые использовались в работе для обеспечения адаптивности.

Во-первых, для обеспечения адаптивности макет полагается на CSS Flexbox, который позволяет элементам страниц перестраиваться в зависимости от ширины экрана. На обеих страницах — главной и скачивания — центральные контейнеры (панели загрузки и скачивания) используют display: flex с настройкой flex-direction: column для мобильных устройств, что обеспечивает вертикальное расположение элементов. Это позволяет избежать переполнения контента и сохраняет вертикальную ориентацию, удобную для сенсорных экранов.

Во-вторых, медиа-запросы играют ключевую роль в адаптации макета. Для ширины экрана 700px и ниже применяются специфические стили, которые изменяют размеры, отступы и компоновку элементов. Этот подход позволяет уменьшить размеры шрифтов, увеличить элементы управления и сократить отступы, чтобы эффективно использовать ограниченное пространство. Медиа-запросы также адаптируют прогресс-бары, задавая им width: 90vw, чтобы они занимали почти всю ширину экрана и оставались читаемыми.

В-третьих, для масштабирования элементов макет использует относительные единицы, такие как %, vw и rem. Например, ширина контейнеров задана как width: 90vw, что позволяет им автоматически подстраиваться под ширину экрана. Высота элементов, таких как панели загрузки, задается в процентах от высоты экрана, что обеспечивает пропорциональное распределение пространства. Шрифты используют единицы rem, что позволяет тексту масштабироваться относительно базового размера шрифта, заданного в корневом элементе. Такой подход гарантирует, что текст и элементы остаются пропорциональными на любых устройствах.

Итак, адаптивность макета для мобильных устройств с шириной 700px

реализована через гибкие сетки, медиа-запросы и относительные единицы измерения. Эти подходы обеспечивают удобство использования, сохраняя визуальную целостность и производительность на смартфонах. В дальнейшем адаптивность может быть улучшена за счет тестирования на устройствах с нестандартными разрешениями и внедрения динамических шрифтов для большей гибкости.

Рисунок 7. Мобильная версия главной страницы

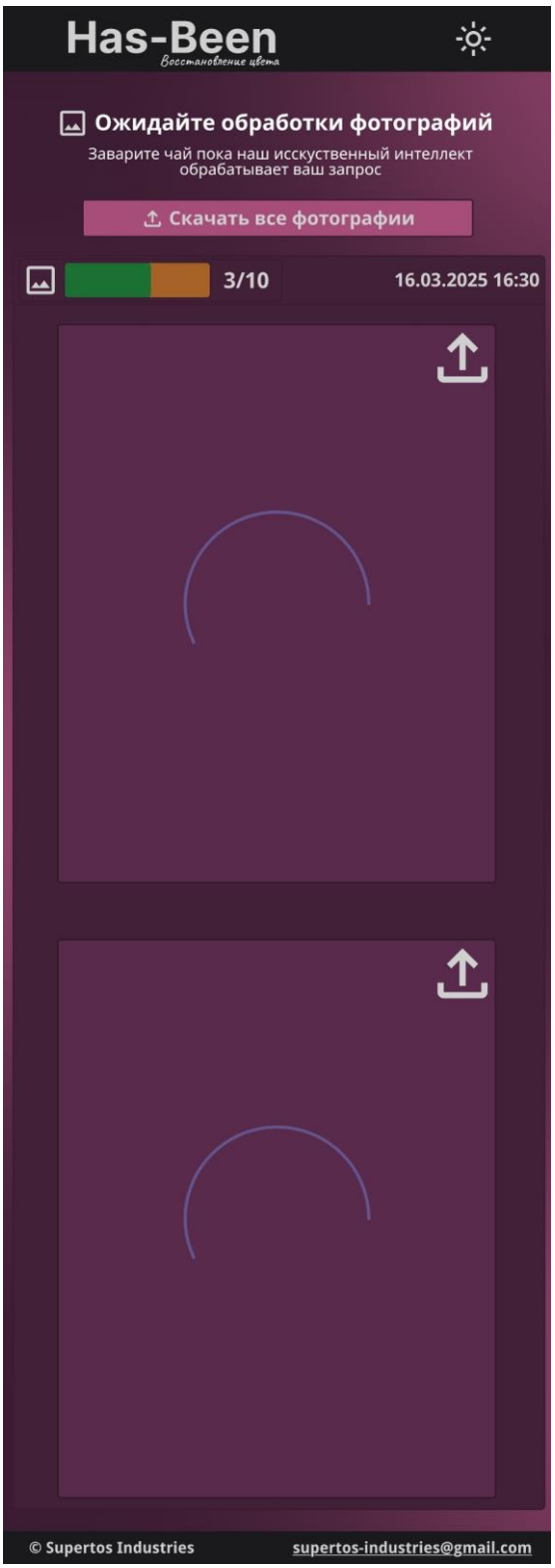

Рисунок 8. Мобильная версия страницы загрузки результатов

2.3.4.4. Последовательность действий пользователя

Работа с веб-приложением "Has-Been" разработана таким образом, чтобы обеспечить интуитивно понятный и логичный путь для пользователей — от загрузки черно-белых фотографий до получения качественно обработанных цветных результатов. Последовательность действий тщательно продумана, чтобы сделать процесс максимально удобным, прозрачным и адаптированным к различным уровням технической подготовки пользователей. Перейдем к подробному описанию этапов взаимодействия с платформой на главной странице загрузки изображений и странице скачивания.

Процесс начинается с посещения главной страницы "Has-Been". Первоначальным действием является активация поля загрузки, обозначенной текстом "Загрузите фотографии". Пользователь может либо перетащить изображения в указанную область, либо воспользоваться файловым проводником устройства для их выбора. После загрузки файлов система отображает миниатюры в сетке, каждая из которых оснащена иконкой крестика для удаления ненужных файлов, предоставляя пользователю возможность корректировки содержания. По завершении загрузки пользователь активирует процесс обработки, нажимая кнопку "Обработать".

После завершения загрузки изображений система автоматически перенаправляет пользователя на страницу получения результатов, где отображается заголовок "Ожидайте обработки фотографий" и индикатор прогресса, отражающий количество обработанных изображений. Временная метка предоставляет дополнительную информацию о времени завершения процесса. Пользователю представляется возможность просмотреть результат обработки и удалить определенные изображения перед скачиванием всех, если результат не соответствует ожиданиям.

Для упрощения массовой загрузки предусмотрена кнопка "Скачать все фотографии", расположенная в нижней части страницы. Активация этой кнопки

позволяет загрузить все обработанные файлы в виде единого архива, что особенно востребовано пользователями, работающими с большим количеством изображений, такими как историки или архивисты.

Завершающим этапом является сохранение результатов на устройстве либо возврат на главную страницу для загрузки новых изображений. Возврат может быть осуществлен через кнопку "Назад" или повторное посещение начальной страницы, что обеспечивает цикличность процесса и возможность многократного использования платформы. Нижняя часть обеих страниц содержит краткую информацию о компании, предоставляя пользователю возможность обратиться за технической поддержкой в случае возникновения вопросов или технических затруднений.

Данная последовательность действий разработана в соответствии с принципами минимализма и интуитивности, минимизируя количество шагов и обеспечивая максимальную прозрачность на каждом этапе. Визуальная обратная связь, возможность удаления файлов и функция массовой загрузки делают процесс взаимодействия эффективным и доступным для широкого круга пользователей.

2.3.4.5 Интеграция дизайна в код

Интеграция дизайна в код и его стилизация представляют собой ключевые этапы разработки веб-приложения на React, предназначенного для восстановления цвета черно-белых фотографий. Этот процесс обеспечивает преобразование визуальных прототипов, созданных в Figma, в функциональный и адаптивный интерфейс, соответствующий требованиям мобильных устройств с шириной экрана 700px и ниже. Был создан модульный, оптимизированный и поддерживаемый код, который воплощает дизайн двух ключевых страниц — загрузки изображений и скачивания обработанных результатов. Перейдем к описанию подходов к интеграции дизайна в React-компоненты, выбор технологий для стилизации, организация стилей и обеспечение их адаптивности.

Подходы к интеграции дизайна в код

Интеграция дизайна начинается с экспорта ресурсов из Figma, включая цветовую палитру (фиолетовый градиент от #4B0082 до #800080, акценты #a64d79 и #d0d0d0, текст #ffffff), шрифты (Droid Sans, Sansation Light) и размеры элементов. Эти данные интегрируются в проект React с использованием модульных компонентов. Например, страница загрузки изображений реализована как компонент UploadPage, а страница скачивания — как ResultPage. Каждый компонент включает HTML-структуру с семантическими тегами (<header>, <main>, <section>) и динамические данные, такие как количество загруженных файлов или прогресс обработки, управляемые состоянием через React Hooks (useState, useEffect)[31].

Процесс интеграции включает разбиение дизайна на переиспользуемые компоненты, такие как Thumbnail (для миниатюр файлов), ProgressBar и Button. Компонент Thumbnail, например, содержит JSX с иконкой удаления и подключает стили из styles.module.scss. Интеграция сопровождается тестированием в реальном времени с использованием эмуляторов Chrome DevTools и BrowserStack, чтобы убедиться в корректном отображении на мобильных устройствах шириной 700px.

Выбор технологий для стилизации

Для стилизации проекта выбраны модульные CSS-файлы с расширением .module.scss, что позволяет изолировать стили для каждого компонента и избежать конфликтов. Шрифты, такие как Droid Sans и Sansation Light, интегрированы через @font-face, указывая пути к файлам .ttf и .woff, например:

```
@font-face {
  font-family: "Droid Sans";
  src: url("../../src/styles/fonts/DroidSans.ttf");
}
```

Стилизация использует методологию BEM[32] (Block Element Modifier). Например, блок .photosContainer включает элементы .photosContainer_item и модификатор .photosContainer--active. Для адаптивности применяются CSS Flexbox и медиа-запросы, а анимации реализованы через CSS Transitions (например, transition: background 0.3s ease для кнопок). Поддержка старых браузеров обеспечивается автопрефиксером PostCSS.

2.3.4.6 Адаптивность и стилизация, оптимизация производительности

Адаптивность достигается через медиа-запросы, которые изменяют

размеры и компоновку для ширины 700px. Например, кнопка .processButton увеличивается до width: 100% и height: 48px на мобильных устройствах. Flexbox используется для перестроения сеток, таких как .photosContainer с display: flex и overflow-x: auto, что позволяет горизонтальную прокрутку при необходимости. Относительные единицы (vw, rem) обеспечивают масштабирование, например, width: 90vw для контейнеров. Анимации, такие как transition: width 0.3s ease для .progressFill, оптимизированы с will-change: width для плавности.

Производительность стилей улучшена минимизацией файлов с помощью CSSNano[33], что уменьшает их размер. Анимации используют GPU-ускорение с transform вместо top/left. Например, эффект hover для .deleteButton реализован как:

```
.deleteButton {
  transition: all 0.2s;
  &:hover {
    transform: scale(1.1);
    background: rgba(255, 0, 0, 0.8);
  }
}
```

Интеграция дизайна "Has-Been" в код React и стилизация с использованием .module.scss, BEM, Flexbox и медиа-запросов обеспечивают модульность и адаптивность. Эти подходы воплощают фиолетовый стиль на

мобильных устройствах. Дальнейшая работа может включить автоматизацию сборки и тестирование на дополнительных разрешениях.

Интеграция дизайна "Has-Been" в код React и стилизация с использованием .module.scss, BEM, Flexbox и медиа-запросов обеспечивают модульность и адаптивность. Эти подходы воплощают на мобильных и десктопных устройствах. Дальнейшая работа может включить автоматизацию сборки и тестирование на дополнительных разрешениях.

2.3.5 Докеризация компонентов в проекте «Восстановление цвета фотографий при помощи средств машинного обучения»

В рамках реализации нашего проекта ключевым этапом архитектурной и технической подготовки программного решения стало использование технологии контейнеризации. **Контейнеризация** — это современный метод упаковки и изоляции программного обеспечения и всех его зависимостей в единый исполняемый блок, называемый контейнером. Контейнеры обеспечивают воспроизводимость среды исполнения приложения, гарантируя, что оно будет работать одинаково вне зависимости от особенностей операционной системы и инфраструктуры. Основным инструментом контейнеризации выступил **Docker**, который позволяет создавать стандартизированные среды исполнения для каждого отдельного компонента системы. Такой подход обеспечивает воспроизводимость, гибкость развертывания и надёжность работы всего программного комплекса.

Цели и задачи контейнеризации

Применение контейнеризации в проекте обусловлено несколькими факторами:

- **Изоляция сервисов.** Каждый компонент приложения, включая API-сервер, базу данных, очередь сообщений и обработчик задач, развёртывается в отдельном контейнере, что предотвращает конфликты зависимостей и упрощает управление версионностью.

- **Масштабируемость.** Контейнеры легко масштабируются — при необходимости, например, можно увеличить количество воркеров для асинхронной обработки задач без влияния на другие элементы системы.
- **Упрощение процессов развертывания.** Использование контейнеров позволяет запускать всю систему целиком с помощью одного сценария (Docker Compose), минимизируя риски ошибок, связанных с ручной установкой и настройкой окружения.
- **Сокращение времени на отладку.** Единообразие среды разработки и эксплуатации позволяет быстрее выявлять и устранять ошибки.

Таким образом, контейнеризация стала неотъемлемой частью архитектуры проекта и обеспечила основу для последующих этапов разработки.

Описание контейнеров и их роли

В состав проекта входят следующие ключевые контейнеры:

- **API-сервер FastAPI**

  Этот контейнер выполняет функции основного процессинга входящих запросов пользователей. Он разработан на базе FastAPI и Uvicorn, которые обеспечивают высокую скорость обработки запросов и поддержку асинхронных операций.

  Контейнер строится на официальном образе Python, в котором дополнительно устанавливаются все необходимые зависимости, указанные в requirements.txt. Такой подход обеспечивает идентичность окружения для всех участников команды, что особенно важно при разработке и тестировании.

  Важной частью конфигурации является использование переменных окружения для подключения к внешним сервисам (MinIO, RabbitMQ, база данных), что позволяет гибко управлять настройками в зависимости от среды (разработка, тестирование, эксплуатация).

- **Nginx — обратный прокси-сервер**

  Контейнер Nginx выполняет задачи маршрутизации входящих HTTP-запросов и проксирования их к API-серверу. Это позволяет разгрузить

основной сервер приложений и повысить общую производительность системы.

Контейнер использует официальный образ Nginx, а конфигурация веб-сервера хранится в виде отдельного файла, подключаемого через volume. Такой способ организации конфигурации упрощает её изменение без необходимости пересборки всего контейнера.

- **RabbitMQ — брокер сообщений**

  Для реализации асинхронной обработки задач используется брокер сообщений RabbitMQ, развёрнутый в собственном контейнере. RabbitMQ играет ключевую роль в обеспечении взаимодействия между API-сервером и обработчиком изображений Celery. Он сохраняет задачи в очереди и обеспечивает их последовательное выполнение, что критически важно для устойчивой работы приложения при высоких нагрузках.

  Использование официального образа RabbitMQ с веб-интерфейсом управления (RabbitMQ Management) предоставляет дополнительные возможности мониторинга и отладки.

- **Celery-воркер**

  Асинхронная обработка ресурсоёмких задач (например, применение модели машинного обучения для окрашивания изображений) выполняется с помощью Celery. Этот компонент вынесен в отдельный контейнер, построенный на том же образе Python, что и API-сервер. Такая архитектура позволяет легко масштабировать обработку — при необходимости можно запустить несколько экземпляров Celery-воркеров, что положительно сказывается на производительности системы в целом.

- **MinIO — локальное S3-хранилище**

  Контейнер MinIO реализует функциональность S3-совместимого хранилища, предназначенного для сохранения исходных изображений и результатов их обработки.

  Использование MinIO обеспечивает быструю работу с файлами и

возможность масштабирования системы хранения в будущем. Контейнер MinIO сохраняет данные в выделенном хранилище (volume), что гарантирует сохранность информации даже при перезапуске или обновлении контейнера.

- **PostgreSQL — база данных проекта**
  Контейнер PostgreSQL отвечает за хранение структурированной информации: данных о пользователях, результатах обработки и других метаданных.
  Использование официального образа PostgreSQL и отдельного тома для хранения данных обеспечивает надёжность и целостность базы, а также упрощает процессы резервного копирования.

Преимущества выбранного подхода

Организация приложения в виде набора взаимосвязанных контейнеров соответствует современным принципам DevOps и микросервисной архитектуры. Такой подход позволяет:

- Повысить надёжность системы за счёт независимости компонентов.
- Упростить внесение изменений и обновлений — каждый контейнер можно заменить или обновить без остановки всей системы.
- Гибко управлять ресурсами, выделяя их в соответствии с актуальной нагрузкой на те или иные компоненты.
- Существенно сократить время настройки новых рабочих мест и тестовых окружений, что положительно сказывается на скорости разработки.

Таким образом, использование технологии контейнеризации Docker в проекте обеспечило высокий уровень надёжности, масштабируемости и удобства эксплуатации приложения, соответствуя современным стандартам индустрии.

Скрипты, используемые в проекте

Для успешной реализации проекта ключевым фактором стало не только создание качественного кода, но и правильная организация рабочих процессов.

В этой связи важную роль играли скрипты — автоматизированные сценарии, которые позволили нам стандартизировать и упростить выполнение рутинных операций, а также обеспечить воспроизводимость всех этапов разработки и эксплуатации.

Цели использования скриптов

Применение скриптов в рамках проекта было направлено на достижение следующих целей:

- **Автоматизация развёртывания среды.** Устранение человеческого фактора при настройке всех необходимых сервисов, что особенно актуально при многоуровневой архитектуре проекта.
- **Обеспечение единообразия.** Скрипты гарантируют, что все участники команды используют одинаковую конфигурацию и версии зависимостей.
- **Сокращение времени на подготовку окружения.** Автоматизированные сценарии позволяют быстрее приступить к работе над функционалом, не отвлекаясь на ручную установку или настройку.
- **Повышение надёжности тестирования.** Скрипты для запуска тестов помогают своевременно выявлять возможные ошибки, что повышает качество всего решения.

Скрипты для развёртывания и запуска проекта

Так как проект включает несколько сервисов (API-сервер, Celery-воркеры, очередь RabbitMQ, база данных PostgreSQL, S3-хранилище MinIO и Nginx), для их одновременного запуска и настройки используется **Docker Compose**. Для удобства был создан скрипт, который запускает следующую команду:

```
docker-compose up -d
```

Этот скрипт позволяет поднять все контейнеры в фоновом режиме, обеспечивая готовность всей системы к работе всего за несколько секунд. В результате исключается необходимость многократного повторения однотипных команд, а разработчики могут быть уверены, что все сервисы стартуют в корректной последовательности с правильными параметрами.

Скрипты для установки зависимостей

Особое внимание уделялось созданию скрипта для установки всех библиотек и зависимостей, необходимых для работы серверной части проекта. С помощью файла requirements.txt и команды:

pip install -r requirements.txt

скрипт обеспечивает установку строго определённых версий библиотек. Это предотвращает возникновение ошибок, связанных с несовместимостью версий, и гарантирует идентичность среды разработки у всех участников команды и на всех этапах жизненного цикла проекта.

Скрипты для управления миграциями базы данных

Управление схемой базы данных — неотъемлемая часть разработки любого серьёзного проекта. Для этого используется инструмент **Alembic**, который позволяет создавать миграции при изменении структуры базы (например, при добавлении новых таблиц или полей) и применять их в любых окружениях.

В проекте был написан скрипт, запускающий следующую команду:

alembic upgrade head

Этот скрипт обеспечивает автоматическое применение всех актуальных миграций к базе данных, что особенно важно при совместной работе нескольких разработчиков и при переходе от одной версии приложения к другой. Таким образом, база данных всегда соответствует последней версии моделей и логики приложения.

Скрипты для тестирования

Качество программного обеспечения напрямую зависит от качества тестирования. Для автоматизации этого процесса в проекте используются скрипты, запускающие тесты с помощью **PyTest**. Типичная команда для запуска всех тестов выглядит так:

pytest tests/

С помощью этого скрипта мы могли регулярно проверять корректность работы API, Celery-воркеров и взаимодействия между компонентами. Важным

преимуществом является то, что тесты запускаются одинаково как в локальной среде, так и в тестовых окружениях, что исключает возможность «скрытых» ошибок, которые могли бы проявляться только при переносе приложения на сервер.

Структурированность и документированность скриптов

При написании скриптов мы уделяли внимание их структурированности и читаемости. Каждый скрипт снабжён комментариями, поясняющими его назначение и особенности запуска. Это облегчает поддержку проекта и делает его доступным для всех участников команды, включая новых разработчиков.

Роль скриптов в достижении целей проекта

В совокупности все указанные скрипты выполняют важную роль в обеспечении стабильной работы проекта. Они:

5. Снижают вероятность возникновения ошибок, связанных с ручной настройкой.
6. Ускоряют процессы тестирования и отладки.
7. Повышают воспроизводимость результатов — ключевой фактор для любого научно-исследовательского проекта.
8. Упрощают масштабирование разработки и эксплуатацию приложения.

Таким образом, скрипты стали важным инструментом, который не только упростил нашу ежедневную работу, но и обеспечил устойчивость и предсказуемость функционирования всей системы.

Выбор локального подхода для проекта

В нашем проекте «Восстановление цвета фотографий при помощи средств машинного обучения» мы приняли решение использовать именно локальное развёртывание. Такой выбор был обусловлен несколькими объективными причинами:

1. Упрощение процессов настройки и отладки

   Локальное окружение предоставляет разработчику полный контроль над всеми компонентами приложения. Все конфигурационные

файлы, логи и параметры контейнеров доступны непосредственно на рабочей машине, что существенно упрощает диагностику проблем и позволяет быстрее их устранять. Разработчик может моментально вносить изменения, перезапускать отдельные контейнеры и видеть результат в реальном времени.

2. Быстрота развертывания и отсутствие зависимости от интернет-соединения

   Локальное развёртывание позволяет запускать приложение без подключения к внешним сервисам. Это особенно важно при работе в условиях нестабильного интернет-соединения или ограниченного доступа к облачным провайдерам. Все необходимые образы контейнеров уже загружены локально, а команда docker-compose up -d позволяет в считанные секунды развернуть всё окружение.

3. Снижение временных затрат

   Работа с облачными решениями обычно требует дополнительной настройки: например, нужно создавать виртуальные машины, настраивать доступы и политики безопасности. Локальное развёртывание, напротив, не требует таких операций. Это позволяет разработчикам сосредоточиться именно на улучшении логики приложения и функционала модели машинного обучения.

4. Повышение воспроизводимости экспериментов

   Для корректной работы и последующей валидации проекта важно, чтобы все участники команды работали в одинаковых условиях. Локальная среда позволяет полностью повторить настройку и запуск приложения, минимизируя влияние внешних факторов (например, задержек в облачной инфраструктуре). Это особенно актуально при работе с задачами машинного обучения, где даже небольшие изменения в окружении могут влиять на результат.

5. Удобство при разработке и тестировании

   В процессе создания и тестирования новых функций, как правило,

требуется много итераций: изменить конфигурацию, проверить результат, внести корректировки. Локальное развёртывание сокращает этот цикл — разработчик может быстро перезапустить контейнер, не затрачивая времени на копирование данных в облако или настройку удалённых серверов.

Выводы по сравнению подходов

Таким образом, локальное развёртывание стало для нас оптимальным решением: оно обеспечило все необходимые возможности для тестирования, настройки и демонстрации работы приложения.

Локальная среда развёртывания

Для обеспечения удобного и стабильного развертывания мы использовали инструменты контейнеризации: **Docker** и **Docker Compose**. Эти технологии позволяют запускать сразу несколько взаимосвязанных сервисов (API, база данных, очередь сообщений, хранилище файлов и т.д.) в изолированных контейнерах. Таким образом, каждый разработчик мог буквально «в несколько команд» подготовить у себя полноценное окружение, идентичное у всех участников команды.

Как проходил процесс развёртывания

Первым этапом стало клонирование репозитория проекта из системы контроля версий Git. Это позволило всем членам команды работать с актуальной версией кода и обеспечило единообразие в настройках.

git clone https://github.com/Supertos/MAI-ML-Colorizer

Далее мы настраивали переменные окружения в файле .env. В нём указывались такие параметры, как адреса сервисов, логины и пароли для подключения к базе данных и MinIO, а также другие параметры, необходимые для корректной работы приложения.

Следующим шагом было использование Docker Compose для запуска всех

сервисов. Вместо того чтобы вручную запускать каждый компонент (API, очередь RabbitMQ, Celery, MinIO, PostgreSQL, Nginx), мы использовали единую команду:

docker-compose up -d

Эта команда позволяла «поднять» всю систему в фоновом режиме. В результате, буквально через несколько секунд, на локальной машине были доступны все сервисы проекта, полностью готовые к работе.

Тестирование после развёртывания

После запуска контейнеров мы всегда проверяли корректность их работы. Например:

- Проверяли, доступен ли веб-интерфейс MinIO (по адресу localhost:9001).
- Убеждались, что API-сервер принимает запросы через Nginx (обычно на порту 8080).
- Смотрели, доступен ли интерфейс RabbitMQ (по адресу localhost:15672).

Такая проверка помогала нам быстро убедиться, что все сервисы запустились корректно и готовы к работе.

Проблемы при развёртывании и пути их решения

Конечно, в процессе развёртывания мы сталкивались с рядом проблем. Основные из них и способы их решения можно описать следующим образом.

- **Конфликты портов**
  Иногда случалось, что нужный порт (например, 8080 для Nginx) уже был занят другой программой на локальной машине. Для решения этой проблемы мы либо освобождали этот порт, либо меняли настройки в файле docker-compose.yml, указывая другой свободный порт. Такой подход позволял нам избежать конфликтов и всё равно запускать все сервисы.
- **Проблемы с сетью между контейнерами**
  Бывали случаи, когда некоторые контейнеры не могли «увидеть» друг друга, из-за чего, например, Celery не мог подключиться к RabbitMQ. Обычно это происходило, если в конфигурации была ошибка —

например, неправильный адрес или имя сервиса. Мы решали это, проверяя правильность всех ссылок и убеждаясь, что имена сервисов в docker-compose.yml совпадают.

- **Сложности при миграции базы данных**
  Когда менялись модели в приложении, нужно было обновить схему базы. Иногда возникали ошибки при применении миграций. Чтобы избежать таких проблем, мы всегда проверяли миграции в отдельной тестовой среде (отдельный контейнер базы), прежде чем запускать их на основном окружении.
- **Недостаток ресурсов компьютера**
  Так как мы запускали все сервисы на своих ноутбуках или стационарных компьютерах, иногда система «подвисала» при сильной нагрузке (например, если запускалось сразу несколько Celery-воркеров или при обработке больших изображений). В таких случаях мы временно отключали ненужные процессы или оптимизировали параметры Docker (например, выделяли больше оперативной памяти контейнерам с помощью настроек Docker Desktop).
- **Обновления и конфигурация**
  При изменении настроек (например, в Nginx или API) мы часто сталкивались с необходимостью перезапустить контейнеры. Для этого мы использовали команду docker-compose down && docker-compose up -d, что помогало «подхватить» изменения конфигурации.

Выводы по развёртыванию

В итоге локальное развёртывание с использованием Docker и Docker Compose позволило нам:

- Быстро разворачивать рабочее окружение у каждого участника команды, с минимальными усилиями.
- Сократить время на настройку и устранить различия между локальными машинами (различные версии Python, библиотек и т.д.).
- Единообразно разрабатывать, тестировать и отлаживать проект, что

особенно важно при работе в команде.

Конечно, у локальной разработки есть свои ограничения — в частности, по мощности оборудования. Однако для целей разработки, тестирования и подготовки проекта к дальнейшему масштабированию локальное развёртывание оказалось простым, удобным и надёжным решением.

Мониторинг в проекте

Мониторинг является одной из важнейших составляющих процесса разработки и эксплуатации любого современного IT-решения. В проекте мониторинг помогал нам отслеживать работу всех компонентов, своевременно выявлять возможные сбои и обеспечивать высокую надёжность работы системы.

Задачи мониторинга

Основными задачами мониторинга в нашем проекте являлись:

- **Отслеживание состояния ключевых сервисов** (API, Celery, RabbitMQ, база данных, хранилище MinIO).
- **Выявление отклонений в работе** (например, задержек при обработке изображений или перегрузки очередей).
- **Повышение надёжности** — мониторинг позволяет быстрее реагировать на возникающие проблемы и предотвращать их последствия.
- **Сбор информации для оптимизации** — анализ собранных данных позволяет улучшать производительность системы и выявлять «узкие места».

Организация мониторинга в проекте

Поскольку проект разворачивался локально на компьютерах участников команды, мы использовали встроенные возможности Docker и компонентов проекта, а также доступные инструменты визуализации.

1. Использование docker stats

Для базового мониторинга нагрузки на систему мы применяли встроенную команду Docker:

docker stats

Эта команда позволяет в реальном времени отслеживать такие параметры, как:

7. использование процессорного времени (CPU Usage);
8. потребление оперативной памяти;
9. сетевой трафик каждого контейнера.

Эти данные помогают определить, какие сервисы потребляют больше всего ресурсов и могут вызывать «торможение» системы, особенно при увеличении числа пользователей.

2. Веб-интерфейсы сервисов

Некоторые сервисы проекта имеют собственные веб-интерфейсы, которые предоставляют дополнительные возможности мониторинга:

5. **RabbitMQ** (http://localhost:15672) — панель управления, в которой отображаются:
   — количество сообщений в очередях;
   — статус воркеров Celery;
   — информация о подключениях и производительности брокера. Это помогает выявлять перегрузки или «зависшие» задачи.
6. **MinIO** (http://localhost:9001) — веб-интерфейс, позволяющий следить за состоянием хранилища, количеством загруженных изображений и их статусом.

Использование этих интерфейсов делало работу с сервисами более прозрачной и удобной.

3. Логи как источник мониторинга

Помимо визуальных инструментов, важную роль в мониторинге играли **логи**, которые каждый контейнер Docker записывает в стандартный поток. Команда:

docker logs <имя_контейнера>

позволяла нам:

— Отслеживать ошибки, которые могли не отражаться в веб-интерфейсах.

— Видеть, какие задачи выполняются в данный момент (например, воркеры Celery выводят статус каждой задачи).
— Проверять корректность запуска сервисов после изменений конфигурации.

Регулярный анализ логов помогал оперативно выявлять и устранять возникающие ошибки.

4. Возможности дальнейшего расширения мониторинга

Хотя локальный мониторинг с помощью Docker и веб-интерфейсов показал свою эффективность, мы также рассматривали возможность дальнейшего развития этой системы. В частности, для более полного контроля за состоянием проекта в будущем мы планируем использовать такие инструменты, как:

- **Prometheus** — для сбора метрик (CPU, память, задержки).
- **Grafana** — для визуализации этих метрик в виде графиков и дашбордов.
- **Alertmanager** — для автоматического оповещения при аномалиях (например, перегрузке API или сбое в работе очередей RabbitMQ).

Интеграция этих инструментов позволит перейти от базового мониторинга к полноценной системе, которая будет автоматически отслеживать все ключевые параметры и оповещать команду о возможных сбоях.

Выводы по мониторингу

Таким образом, даже при локальной разработке мы организовали базовый, но эффективный мониторинг всех ключевых компонентов проекта. Это позволило:

— Поддерживать стабильную работу системы даже при росте нагрузки.
— Быстро находить и устранять ошибки.
— Обеспечить более высокое качество и надёжность всего решения.

Мониторинг стал важной частью проекта и заложил основу для последующего перехода к более продвинутым инструментам наблюдения, которые будут особенно актуальны при возможной эксплуатации приложения в

«боевых» условиях.

Логирование в проекте

Логирование является неотъемлемой частью любого современного IT-решения. В проекте логирование выполняло важные функции: позволяло отслеживать ход работы компонентов, выявлять ошибки и узкие места, а также обеспечивало прозрачность и воспроизводимость всех операций. Благодаря грамотно настроенному логированию мы могли быстрее реагировать на возникающие проблемы и улучшать стабильность всей системы.

Задачи логирования

Основными задачами, которые решает логирование в нашем проекте, являются:

— **Регистрация всех событий и операций.** Каждое действие пользователя или системы фиксируется в логах: от запросов к API до выполнения асинхронных задач.

— **Отслеживание ошибок.** Логи помогают быстро находить и анализировать ошибки, которые могут возникать при работе сервиса.

— **Анализ производительности.** По логам можно судить, сколько времени уходит на выполнение конкретных задач, а также выявлять потенциальные узкие места.

— **Обеспечение безопасности.** Логирование позволяет фиксировать все попытки взаимодействия с сервисами, что помогает выявлять подозрительные активности.

Таким образом, логирование в проекте — это не только способ выявления ошибок, но и полноценный инструмент анализа работы приложения.

Реализация логирования в проекте

В нашем проекте каждый компонент имел собственный механизм логирования, который мы настраивали с учётом его особенностей.

Логирование API-сервера (FastAPI)

FastAPI по умолчанию ведёт логи всех входящих запросов, их статусов и

времени обработки. Эти логи записываются в стандартный поток (stdout), что позволяет их легко просматривать с помощью Docker:

docker logs fastapi

Благодаря этому мы всегда могли видеть, какие именно запросы приходили к серверу, сколько времени заняла их обработка, и не возникало ли при этом ошибок.

Логирование асинхронных задач (Celery)

Celery-воркеры запускаются с явно указанным уровнем логирования — обычно это уровень info:

celery -A app.celery_app.celery worker --loglevel=info

Такой подход позволяет:

- Отслеживать ход выполнения каждой асинхронной задачи (например, окрашивания фотографий).
- Видеть сообщения об успешном завершении задач или, наоборот, об ошибках в процессе.
- Анализировать загрузку воркеров при повышенной нагрузке.
- **Логирование очереди сообщений (RabbitMQ)**

RabbitMQ, как брокер сообщений, также ведёт собственные логи, которые можно просматривать через Docker. Это помогает выявлять ошибки на уровне взаимодействия между API и Celery — например, если сообщение не может быть доставлено в очередь.

Логирование базы данных и MinIO

PostgreSQL и MinIO — два важных компонента, которые также записывают собственные логи. Эти логи позволяют проверять:

- корректность выполнения запросов к базе;
- возможные ошибки в работе S3-хранилища;
- проблемы с доступом к файлам (например, при сохранении загруженных изображений).

Использование стандартных инструментов Docker

Поскольку все сервисы разворачивались в контейнерах, мы активно

использовали стандартный механизм логирования Docker. Команда:

docker logs <имя_контейнера>

позволяет получить «снимок» текущего состояния работы сервиса, а при необходимости — отследить подробности выполнения операций в реальном времени.

Проблемы при работе с логами и пути их решения

При работе с логами мы сталкивались с рядом проблем:

1. **Большой объём логов**
   При интенсивной обработке (например, при массовой загрузке изображений) объём логов быстро возрастал. Это затрудняло их анализ. Для решения проблемы мы:
   - Настраивали уровень логирования (например, warning вместо info для уменьшения количества записей);
   - Использовали фильтры при просмотре логов (например, grep для поиска ошибок).
2. **Неудобство анализа при большом количестве контейнеров**
   Когда логов слишком много и их нужно собирать с разных сервисов, это становится неудобным. Для этого мы рассматривали возможность подключения централизованных систем логирования — например, **ELK-стек (Elasticsearch + Logstash + Kibana)** или **Graylog**, которые позволяют собирать, хранить и анализировать логи в единой системе.
3. **Форматирование логов**
   Логи разных сервисов часто имеют разный формат. Чтобы облегчить их анализ, в будущем мы планируем использовать унифицированный формат логов (например, JSON), который проще обрабатывать и визуализировать.

Роль логирования в развитии проекта

Грамотно настроенное логирование стало важной частью жизненного цикла проекта. Оно позволило нам:

- Улучшить качество приложения — выявляя ошибки ещё на ранних

этапах.

- Повысить надёжность и стабильность всей системы.
- Быстрее разрабатывать и тестировать новые функции, видя их «отражение» в логах.

В дальнейшем, при переходе от локальной разработки к более «боевым» сценариям, мы планируем развивать систему логирования, чтобы сделать её ещё более информативной, структурированной и удобной для всех участников команды.

Тестовые окружения в проекте

Важной частью успешной реализации любого IT-проекта является наличие продуманного и качественно настроенного тестового окружения. В проекте тестовые окружения играли важную роль: они позволяли нам проверять корректность работы компонентов, выявлять ошибки до их попадания в основную ветку разработки и гарантировать стабильность и надёжность всего решения.

Цели создания тестовых окружений

Тестовые окружения в нашем проекте выполняли несколько ключевых функций:

- **Изоляция.** Проверка нового функционала в отдельной среде, чтобы избежать влияния на основное окружение разработки.
- **Воспроизводимость.** Тестирование в идентичных условиях, что исключает ошибки, связанные с различиями в настройках у разных участников команды.
- **Автоматизация.** Возможность интеграции тестов с процессами CI/CD, что повышает скорость и надёжность выпуска новых версий.
- **Повышение качества.** Выявление ошибок, уязвимостей или некорректной работы функций ещё до того, как они попадут в «боевую» версию.

Таким образом, тестовые окружения стали важным инструментом в

нашем проекте, поддерживая его развитие и эксплуатацию.

Реализация тестового окружения

В качестве тестового окружения мы использовали **локальные контейнеры Docker**, которые полностью повторяли структуру основной системы. Такой подход имел следующие преимущества:

1. Каждое тестовое окружение полностью копировало рабочее — включая все сервисы: API на FastAPI, брокер RabbitMQ, Celery-воркеры, MinIO, базу данных PostgreSQL и Nginx.
2. Не требовалось отдельное физическое или облачное окружение, что существенно снижало затраты.
3. Легко масштабируемо — любой член команды мог быстро поднять тестовое окружение у себя на компьютере, выполнив команду:

docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d

Мы использовали возможность Docker Compose запускать несколько конфигурационных файлов (например, docker-compose.override.yml), чтобы в тестовом окружении добавлять специфические настройки:

- Логи с более подробным уровнем (debug), чтобы легче отлавливать ошибки.
- Подключение к отдельной тестовой базе данных (например, test_db), чтобы не затронуть рабочие данные.
- Использование dummy-данных (тестовых изображений) для проверки корректности обработки.

Инструменты тестирования

Для проверки работы кода мы использовали **PyTest** — современный и гибкий фреймворк для тестирования Python-приложений. В тестовом окружении запускались следующие типы тестов:

- **Модульные тесты** (unit-тесты) — для проверки отдельных функций и методов.
- **Интеграционные тесты** — для проверки взаимодействия между компонентами: например, API + база данных, API + очередь сообщений.

- **End-to-end тесты** — для проверки всего жизненного цикла обработки изображения: от загрузки пользователем до получения результата.

  Команда запуска тестов:

  pytest tests/

2.3.6 Таблица компонентов

Список основных компонентов решения представлен в таблице 2.

Название модуля	Функционал	Особенности работы	Режим функционирования
Веб-сервер	Прием запросов, балансировка нагрузки, маршрутизация запросов		Автоматический
Сервис API	Обработка тела запроса, работа с базой данных и хранилищем, балансировка нагрузки		Автоматический (по получении запроса)
База данных	Хранение запросов и их статусов, поддержка доступа для всех машин в сети		Непрерывный
Пользовательский интерфейс	Предоставление удобного доступа к API посредством графического интерфейса на React		Реактивный (зависит от пользователя)
S3-Хранилище	Хранение пользовательских изображений и результатов, поддержка доступа для всех машин в сети		Непрерывный

Брокер сообщений	Интеграция с Celery, распределение нагрузки на несколько удаленных машин, доставка запросов, информирование о задачах, обеспечение обратной связи		Непрерывный
Обработчик изображений		Отдельно запускаемая программа, выполняющая функции обработки изображений и связанная с API через брокер сообщений, S3-хранилище и базу данных	По запросу пользователя
Пул работников	Получает из брокера задачи на окраску, загружает из хранилища и базы данных необходимую для выполнения задачи информацию	Является частью обработчика изображений	По запросу пользователя
Предобработчик и конвертер в lab	Обрабатывает входные изображения и переводит их в Lab, сжимает изображения для последующей обработки нейронной сетью, накладывает на исходное изображение результат работы нейронной сети	Является частью обработчика изображений	По запросу пользователя
Нейронная сеть окрашивания	U-Net нейронная сеть для обработки (окрашивания)	Является частью обработчика изображений,	По запросу пользователя

	изображений в формате lab.	обрабатывает изображения постоянного размера 128x128 пикселей	

Таблица 2 - Компоненты решения

Применение разнообразного спектра программных инструментов, интегрированных в определенные компоненты и методы проектирования архитектуры системы, значительно повышает общую эффективность работы сервиса. Это достигается за счет грамотного распределения задач между множеством специализированных программных обработчиков, использующих фреймворк Celery для асинхронной обработки и управления очередями задач. Эти обработчики получают запросы через брокер сообщений RabbitMQ, что позволяет эффективно масштабировать систему и обеспечивать обработку данных в параллельном режиме. Вдобавок, использование распределенного S3-хранилища для хранения изображений, в сочетании с базой данных PostgreSQL для управления метаданными, способствует значительному сокращению объема данных, которые должны передаваться через брокер сообщений. Это, в свою очередь, уменьшает нагрузку на центральную машину, обслуживающую FastAPI, перекладывая часть процессов по снабжению данных на подчиненные машины, где расположены специализированные приложения-обработчики. Такое распределение задач и данных позволяет не только оптимизировать обработку запросов, но и повысить общую масштабируемость системы, эффективно управляя нагрузкой и ресурсами.

2.4 План разработки проекта «Восстановление цвета фотографий при помощи средств машинного обучения»

№	Содержание работы	Результат	Начало	Окончание	Исполнитель
1	Изучить рынок, конкурентов и устройство аналогов	Были изучено множество существующих сервисов по окрашиванию изображений	16.12.2024	18.12.2024	Жаворонков Никита Дмитриевич
2	Сформировать особенности проекта, его цель	Было решено сделать ставку на легкость нейронной сети, простоту и доступность интерфейса, а также масштабируемость	18.12.2024	22.12.2024	Жаворонков Никита Дмитриевич, Караткевич Николай Сергеевич, Воронухин Никита Александрович
3	Обсудить и установить целевую аудиторию сервиса	Было решено сделать упор на аудиторию без профильного образования с минимальными навыками работы с ЭВМ, а также на историков и архивистов	18.12.2024	20.12.2024	Жаворонков Никита Дмитриевич, Караткевич Николай Сергеевич, Воронухин Никита Александрович, Люзина Мария Николаевна
4	Утвердить финальную общую архитектуру проекта	Была утверждена модульная архитектура, где обработкой запросов занимаются отдельные приложения-обработчики	20.12.2024	24.12.2024	Жаворонков Никита Дмитриевич

5	Утвердить архитектуру нейронной сети	Была утверждена нейронная сеть U-Net с последующим добавлением GAN	20.12.2024	08.01.2025	Воронухин Никита Александрович
6	Разработать план работ	Был разработан план работ на Январь-Февраль 2025, который в последствие стал планом на Февраль — Май 2025	26.12.2024	28.12.2024	Жаворонков Никита Дмитриевич
7	Утвердить формат API	Был утвержден формат API с функцией запроса и отправки на обработку изображений	24.12.2024	28.12.2024	Жаворонков Никита Дмитриевич, Бачурин Николай Владимирович, Караткевич Николай Сергеевич, Жиденко Виктория Александровна
8	Разработать и утвердить макет	Был разработан и утвержден первый макет	01.01.2025	08.02.2025	Люзина Мария Николаевна
9	Разработать базовый FastAPI сервер	Был разработан базовый FastAPI сервер как проверка возможностей команды	08.02.2025	12.02.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан

					Ринатович
10	Подключить к серверу брокер сообщений	Был подключен брокер сообщений к серверу, и, в качестве теста, блокирующий Celery	16.02.2025	12.03.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
11	Разработать приложение-обработчик	Было разработано приложение обработчик в его базовом виде с пулом работников Celery, соединенных с FastAPI через брокер RabbitMQ	12.03.2025	28.03.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
12	Подключить базу данных	Была подключена база данных PostgreSQL к приложению-обработчику и FastAPI	28.03.2025	12.04.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
13	Подключить S3-хранилище	Было подключено S3-совместимое хранилище MinIO к приложению-обработчику и FastAPI	28.03.2025	12.04.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
14	Разработать React-приложение — страница загрузки	Была разработана главная страница (или страница	01.05.2025	15.05.2025	Абдыкалыков Нурсултан Абдыкалыкович, Люзина Мария

		загрузки изображений)			Николаевна
15	Разработать React-приложение — страница выгрузки и состояния	Была разработана страница выгрузки изображений	01.05.2025	15.05.2025	Абдыкалыков Нурсултан Абдыкалыкович, Люзина Мария Николаевна
16	Переработать макет	Был переработан макет: изменен дизайн, положение объектов, цветовая палитра	25.04.2025	26.04.2025	Жаворонков Никита Дмитриевич, Люзина Мария Николаевна
17	Разработать конвертер RGB в lab	Разработан конвертер изображений в формат lab	20.05.2025	23.05.2025	Воронухин Никита Александрович
18	Утвердить стиль кода	Был утвержден единый стиль кода для фронтенда и бекенда	01.01.2025	08.01.2025	Жаворонков Никита Дмитриевич
19	Создание набора данных на основе MIRFLIKR25k	Был создан первый тестовый набор данных для проверки возможностей команды	01.01.2025	08.01.2025	Воронухин Никита Александрович
20	Создание CNN AE на 32x32	Разработана базовая нейронная сеть, обучение провалилось	08.01.2025	01.02.2025	Воронухин Никита Александрович
21	Увеличение входного изображение	Входное изображение	01.02.2025	15.02.2025	Воронухин Никита

	нейронной сети до 64x64	увеличено, нейронная сеть начала производить идентичные входным изображения			Александрович
22	Переход на U-Net 128x128	Изменена архитектура нейронной сети, увеличен размер нейронной сети. Новая архитектура позволила относительно качественно окрашивать изображения	15.02.2025	01.03.2025	Воронухин Никита Александрович
23	Смена набора данных на новый, основывающийся на Flickr30k и фотографиях Прокудина-Горского	Разработан новый набор данных, специализирующийся на исторических образцах, преимущественно фото Российской Империи	01.03.2025	05.03.2025	Воронухин Никита Александрович
24	Подключение Tensorboard, настройка callbacks	Подключен Tensorboard для сбора метрик и анализа поведения нейронной сети	15.03.2025	20.03.2025	Воронухин Никита Александрович
25	Разработка U-Net 256x256, эксперименты с VAE	Эксперименты со вниманием, изменение архитектуры	01.04.2025	23.04.2025	Воронухин Никита Александрович

		нейронной сети: неудачно, нейронная сеть выходит на плато или не обучается вовсе			
26	Внедрение Attention вместо Skip-соединений	Удаление Skip-соединений, препятствующих обучению сети, неудача: нейронную сеть не удалось обучить	23.04.2025	04.05.2025	Воронухин Никита Александрович
27	Переход на U-Net 128x128	Возвращение к U-Net с целью дистилляции оригинальной версии. Неудача.	04.05.2025	20.05.2025	Воронухин Никита Александрович
28	Возврат к исходной U-Net с 151млн параметров	Принято решение использовать оригинальную нейронную сеть по причине недостатка временных ресурсов.	20.05.2025	20.05.2025	Воронухин Никита Александрович

Таблица 3 - Календарный план разработки проекта

В период с января по май 2025 года была проделана обширная и комплексная работа по разработке и интеграции различных ключевых компонентов системы. В ходе этого времени была внедрена целая серия архитектурных решений, которые легли в основу работы всей системы. Одним из самых значимых шагов стало успешное внедрение FastAPI для backend-части, что позволило значительно повысить скорость обработки запросов и улучшить общую производительность системы. Вдобавок к этому, была настроена и интегрирована RabbitMQ, играющая важную роль в организации эффективного управления асинхронными задачами, что обеспечило плавную и непрерывную работу всей системы. Использование MinIO для хранения изображений и PostgreSQL для управления метаданными позволило создать стабильную и безопасную инфраструктуру для работы с данными, соответствующую требованиям проекта.

Кроме того, в рамках разработки было проведено несколько этапов работы с Celery, что способствовало организации эффективной и многозадачной обработки запросов. Интеграция нейросети U-Net, предназначенной для автоматического окрашивания изображений, также была успешной и стала важным достижением проекта, обеспечив высокий уровень автоматизации и улучшение качества работы системы. Эти шаги значительно продвинули проект в плане функциональности и готовности к запуску.

Однако, несмотря на значительные успехи в процессе разработки, команда столкнулась с рядом сложных вызовов, связанных с ограниченным опытом и непредсказуемыми обстоятельствами. В частности, несколько раз приходилось пересматривать и дорабатывать архитектурные решения, что требовало значительных усилий и времени. Процесс перераспределения задач между компонентами системы также оказался не таким гладким, как предполагалось изначально. В ходе разработки выявились недочеты в некоторых первоначальных подходах, связанных с организацией работы с

данными и распределением нагрузки на серверы. Эти проблемы потребовали серьезных изменений в изначальной архитектуре, что привело к необходимости пересмотра многих аспектов работы системы.

Вследствие этого команда провела дополнительные итерации разработки и тестирования, что в свою очередь потребовало корректировки проектных решений. Иногда эти изменения были комплексными и требовали дополнительных временных и трудовых затрат, однако они сыграли ключевую роль в улучшении качества работы всей системы. И хотя эти трудности стали значительным препятствием на пути к успешному завершению разработки, они позволили команде укрепить свои навыки и получить ценный опыт в решении реальных проблем в области разработки.

Постоянное обновление и оптимизация решений, а также более глубокое понимание особенностей работы с данными и инфраструктурой в процессе разработки стали важными шагами на пути к созданию устойчивой и эффективной системы. В результате работы была достигнута значительная степень улучшения, что позволило создать более надежную и гибкую платформу для дальнейшей эксплуатации и развития. Таким образом, несмотря на необходимость адаптации и множества корректировок, период с января по май 2025 года можно с уверенностью считать продуктивным и успешным. Этот период стал временем для значительного укрепления основ проекта, получения опыта и оптимизации ключевых аспектов работы системы, что закладывает основу для дальнейшего развития и реализации в будущем.

№	Роль в команде	ФИО	Группа	Функционал
1	Глава команды, дизайнер	Жаворонков Никита Дмитриевич	М8О-210Б-23	Координация, коррекция, отслеживание, доработка
2	Верстальщик	Абдыкалыков Нурсултан Абдыкалыкович	М8О-206Б-23	Верстка на React, тестирование
3	Верстальщик, дизайнер	Люзина Мария Николаевна	М8О-207Б-23	Верстка на React, разработка макета
4	Python-программист	Бачурин Николай Владимирович	М8О-211Б-23	Разработка серверного API, интеграция Celery и RabbitMQ
5	Python-программист	Салихов Руслан Ринатович	М8О-203Б-23	Разработка серверного API, интеграция MinIO и PostrgeSQL
6	ML-разработчик	Воронухин Никита Александрович	М8О-210Б-23	Обучение и проектирование нейронной сети, утверждение внутреннего формата lab
7	DevOps инженер	Караткевич Николай Сергеевич	М8О-214Б-23	Специалист по Docker, связывание Frontend и Backend
8	DevOps инженер	Дубровина Софья Андреевна	М8О-207Б-23	Специалист по Docker
9	Python-программист	Жиденко Виктория Александровна	М8О-207Б-23	Разработка Серверного API, разработка приложения-обработчика

Таблица 4 - Участники команды разработки проекта

В сумме над проектом работало 9 человек, каждый из которых играл важную роль в разработке и внедрении различных ключевых компонентов системы. Каждый участник команды внес свой уникальный вклад, будь то в архитектурное проектирование, разработку backend и frontend частей, или в интеграцию нейросети. Помимо этого, стоит отметить, что несколько членов команды выполняли сразу несколько ролей, что позволило значительно улучшить организацию рабочих процессов и эффективно распределить доступные ресурсы. Этот гибкий подход помог ускорить процесс разработки и обеспечил более высокую степень слаженности в работе, даже при наличии множества одновременно решаемых задач.

Необходимо подчеркнуть, что несмотря на многозадачность, такой стиль работы не только не оказался негативным фактором, но, наоборот, способствовал повышению продуктивности всей команды. Он позволил оперативно реагировать на возникающие проблемы, гибко адаптировать подходы и быстрее принимать важные решения. В некоторых случаях многозадачность обеспечивала настоящую синергию, когда работа нескольких человек над различными аспектами проекта одновременно способствовала быстрому решению сложных задач. Это создавало возможности для более эффективного и динамичного взаимодействия между различными этапами работы, что, в свою очередь, способствовало успешному прогрессу проекта.

Кроме того, данный опыт оказался исключительно ценным для всей команды, особенно в условиях постоянных изменений требований и необходимости быстрого реагирования на возникающие вызовы. В процессе работы над проектом, каждый член команды научился не только оперативно адаптироваться к новым условиям, но и осваивать новые методики обучения нейросетей, что стало важной составляющей успеха проекта. Обучение нейросети, подбор и настройка гиперпараметров, а также поиск оптимальных решений по улучшению ее эффективности — все это потребовало от команды

постоянного обновления знаний и гибкости в принятии решений.

Кроме того, команда столкнулась с необходимостью быстро находить информацию, обучаться на основе открытых источников и интегрировать полученные знания в уже развернутую инфраструктуру. Это умение быстро ориентироваться в потоках информации и находить нужные решения стало настоящим ключом к продуктивной работе. Постоянная работа с новыми алгоритмами и технологиями также обеспечила повышение квалификации всей команды.

Кроме того, способность оперативно обучаться и адаптировать нейросети позволила команде эффективно решать задачи, с которыми они сталкивались в процессе разработки, не тратя лишнего времени на изучение теоретических аспектов. Такой подход не только ускорил процесс обучения, но и способствовал быстрому реагированию на изменения в требованиях и появление новых вызовов. Это сыграло ключевую роль в минимизации задержек и поддержании стабильного темпа работы, что в свою очередь способствовало успешной реализации проекта в заданные сроки.

№	Компонент среды разработки	Описание	Комментарий
1	Доска задач	Trello https://trello.com/ru	Планирование и координация работы
2	Информационный обмен	Telegram-канал	Оперативные коммуникации
3	Информационный обмен	Google Meet https:// meet.google.com/ landing	Оперативные коммуникации
4	Поиск источников	DeepSeek https://chat.deepseek .com	Поиск документации, статей, работ
5	Поиск источников	ChatGPT https://chatgpt.com	Поиск документации, статей, работ
6	Среда разработки (IDE)	PyCharm	Среда разработки (IDE)
7	Текстовый редактор	Notepad++	Среда разработки (IDE)
8	Среда разработки (IDE)	Visual Studio Code	Среда разработки (IDE)
9	Текстовый редактор	LibreOffice Writer	Подготовка отчетов
10	Редактор презентаций	LibreOffice Impress	Подготовка презентаций

Таблица 5 - Среда разработки проекта

Балансировка нагрузки и масштабируемость

Важной частью архитектуры проекта является возможность эффективной балансировки нагрузки и последующего масштабирования системы. Эти два аспекта — балансировка и масштабируемость — неразрывно связаны между собой: балансировка нагрузки позволяет системе равномерно распределять ресурсы и предотвращать перегрузку, а масштабируемость даёт возможность гибко наращивать ресурсы при росте аудитории.

Понятие балансировки нагрузки

Под **балансировкой нагрузки** понимается организация работы системы таким образом, чтобы ресурсы (CPU, память, сеть) использовались максимально равномерно.

В условиях реальной эксплуатации это означает:

- Отсутствие «узких мест» (bottlenecks), которые замедляют всю систему,
- Предотвращение ситуации, когда один компонент перегружен, а другие простаивают,
- Снижение времени отклика для пользователя даже при пиковых нагрузках.

Для нашего проекта это особенно важно, поскольку операция окрашивания фото требует больших вычислительных затрат, а пользователи ожидают быстрый отклик и готовый результат.

Балансировка между FastAPI и Celery

Архитектурно балансировка нагрузки происходит через чёткое разграничение функций:

FastAPI — отвечает только за приём, проверку и запись задач. Он остаётся быстрым и отзывчивым, так как не занимается «тяжёлой» обработкой. **Celery** — берёт на себя все ресурсоёмкие вычисления: восстановление цвета изображений нейронной сетью. Каждый воркер Celery изолирован и работает в

своём процессе, что даёт возможность выполнять много задач параллельно.

Эта схема позволяет системе «разгружать» FastAPI и при этом эффективно использовать возможности Celery-воркеров, которые «берут на себя» самую сложную часть работы.

Горизонтальная масштабируемость Celery

Одним из главных достоинств выбранной архитектуры является возможность **горизонтального масштабирования Celery-воркеров**. Это означает, что при росте нагрузки (например, при массовой оцифровке архивов) можно запустить дополнительное количество воркеров, которые сразу начинают обрабатывать новые задачи.

Каждый новый воркер Celery — это отдельный процесс, который подключается к очереди задач и начинает их выполнять. Такая масштабируемость даёт **гибкость**:

6. **При обычной нагрузке** достаточно нескольких воркеров, чтобы быстро обрабатывать все загружаемые фотографии.
7. **При пиковых нагрузках** (например, при массовой акции оцифровки) можно без остановки сервиса запустить больше воркеров, чтобы ускорить обработку.

Кроме того, горизонтальная масштабируемость упрощает поддержку проекта: при необходимости можно даже распределить воркеры по разным серверам, чтобы использовать все доступные ресурсы (CPU, память). Это делает систему «живой» и легко адаптирующейся под любые задачи.

Проблемы при разработке

Отдельно стоит выделить проблемы, связанные с отладкой и мониторингом системы, построенной на основе RabbitMQ и Celery. Брокер сообщений RabbitMQ, хотя и обладает встроенными инструментами мониторинга через веб-интерфейс и консоль, не предоставляет детальных логов выполнения самих задач или содержимого передаваемых сообщений.

Его основная задача — эффективная маршрутизация и гарантированная доставка, но не управление логикой исполнения. Все данные о процессе выполнения задач, возникающих ошибках, успешных завершениях, времени выполнения и других событиях находятся на стороне Celery-воркеров, которые отвечают за саму бизнес-логику.

Поэтому успешная отладка требует организации качественной системы логирования и мониторинга в нескольких местах: начиная от FastAPI, который ставит задачи в очередь, заканчивая Celery-воркерами, которые их обрабатывают. Важно настроить уровень логирования воркеров на подробный — обычно --loglevel=info или --loglevel=debug — чтобы видеть детальную информацию о состоянии задач, возникающих исключениях и проблемах с подключениями. Поскольку окружение контейнеризированное и распределённое, логи могут оказаться разрозненными и разбросанными по разным контейнерам, поэтому рекомендуется использовать централизованные системы логирования, такие как ELK-стек (Elasticsearch, Logstash, Kibana), Fluentd, или другие инструменты, позволяющие агрегировать логи с разных сервисов и упрощать их анализ.

В процессе эксплуатации была выявлена и необходимость улучшения обработки ошибок непосредственно внутри задач Celery. Например, исключения, связанные с отсутствием нужных ключей в хранилище MinIO или проблемами с подключением к базе данных PostgreSQL, могли приводить к неожиданным сбоям, которые сложно было быстро диагностировать без правильной обработки и логирования. В ответ на это задачи были дополнительно обёрнуты в конструкции try/except, а возникающие исключения явно логировались с указанием контекста, что позволяло быстрее локализовать и устранить проблему.

Таким образом, несмотря на высокую надёжность RabbitMQ, успешное

использование его в реальном проекте требует продуманного подхода к организации порядка запуска сервисов, учёта ограничений протокола AMQP, построения архитектуры передачи данных и создания качественной системы логирования и мониторинга. Эти меры позволяют минимизировать сбои, улучшить отладку и обеспечить устойчивую, масштабируемую работу распределённой системы, что особенно важно для проектов с высокой нагрузкой и критичными требованиями к отказоустойчивости.

Проблемы разработки нейронной сети заключались в:

- Небольшая информация во входе
    - Однозначность. В градациях серого теряется вся информация об оттенке и насыщенности. Один и тот же оттенок серого может соответствовать десяткам совершенно разных цветовых комбинаций.
    - Мультимодальность. Правильного «единственного» ответа не существует: для зелёного листа и красного яблока на одинаковых яркостных уровнях сеть должна уметь выбирать верный контекст.
- Семантическая неоднозначность
    - Для корректной раскраски важно «понимать» содержимое кадра: фон, объекты, их материал (металл/ткань), освещение, отбрасываемые тени и рефлексы.
    - Без семантического распознавания (например, «это дерево», «это лицо») сеть рискует окрасить объекты неправильно (лицо в синий цвет, небо в коричневый и т. д.).
- Имбаланс цветовых распределений
    - В реальных фото зелёных и синих тонов больше, чем, скажем, пурпурных. Нейросеть может «захотеть» давать наиболее частые в датасете цвета, что приведёт к перенасыщению зелёного в алгоритмически окрашенных изображениях.
- Искусственные артефакты

    - Заливка цветом: при недостаточности информации сеть может «разливать» цвет большими областями, теряя границы деталей.
    - Гало-артефакты: нечёткие цветовые ореолы вокруг объектов. Возникают из-за сглаживающих слоёв (upsampling, transpose convolution).
- Выбор метрик качества
    - Пиксельные метрики (MSE, MAE) плохо коррелируют с человеческим восприятием цвета.
    - SSIM/LPIPS измеряют структурное сходство, но не гарантируют правильность цветовой гаммы.
    - Субъективная оценка: зачастую необходимо проводить пользовательские исследования, что дорого и медленно.
- Разнородность источников данных
    - Произвольные изображения с веба, архивов и личных съёмок могут отличаться по балансу белого, гамме, шуму, разрешению.
    - При обучении на плохо нормализованных данных сеть может «подхватывать» индивидуальные артефакты источника (например, виньетирование или сдвиг оттенка).
- Ограничения вычислительных ресурсов
    - Высокое разрешение (более 256×256256\times256256×256) увеличивает использование памяти и время обучения.
    - Баланс между качеством (более глубокой моделью, большим контекстом) и скоростью вывода для веб-сервиса в реальном времени.
- Проблемы генерализации
    - Модель, натренированная на одном домене (например, пейзажи), может плохо окрашивать другой (портреты, архитектура).

Проблемы разворачивания и тестирования:

В процессе настройки и использования тестовых окружений мы

столкнулись с рядом типичных проблем:

1. Различие окружений у разработчиков
   Иногда возникали ситуации, когда тесты проходили у одного участника, но «падали» у другого. Обычно это происходило из-за различий в версиях Python или библиотек. Мы решили эту проблему, ещё более строго контролируя зависимости через requirements.txt и используя Docker-окружение для тестов.
2. Загруженность ресурсов
   Поскольку тестовое окружение полностью повторяло рабочее, оно потребляло много ресурсов компьютера. Чтобы минимизировать это, мы отключали ненужные в тестах сервисы (например, Nginx), оставляя только те, которые необходимы для конкретных тестов.
3. Необходимость очистки базы данных
   Для каждого запуска тестов нужно было начинать с «чистой» базы. Мы использовали Alembic, чтобы перед тестами откатывать и применять миграции заново, а также специально подготовленные скрипты для инициализации базы.
4. Отсутствие централизованной тестовой платформы
   На локальных машинах не всегда удобно собирать отчёты о тестах. Мы рассматривали возможность интеграции с GitHub Actions или GitLab CI, чтобы запускать тесты в изолированных контейнерах при каждом коммите. Это в будущем обеспечит ещё более высокий уровень автоматизации.

   Выводы по тестовым окружениям

Таким образом, создание и использование тестовых окружений в проекте «Восстановление цвета фотографий при помощи средств машинного обучения» позволило нам:

- Проверять все изменения кода в условиях, максимально приближенных к «боевому» окружению.
- Сократить количество ошибок, которые могли бы попасть в основную

версию.

- Повысить общее качество и надёжность всего программного решения.

Благодаря Docker и PyTest мы смогли легко масштабировать тестирование, а каждый участник команды мог быстро и просто запускать тесты у себя локально, что способствовало более эффективной командной работе.

Развёртывание проекта: где, как и с какими проблемами столкнулись

При создании современных IT-решений перед командами разработчиков всегда встаёт важный архитектурный вопрос: где именно разворачивать систему — локально или в облаке? Тот же вопрос встал и перед нами. Оба подхода имеют свои особенности, преимущества и недостатки, а выбор зависит от целей, задач проекта и условий его эксплуатации.

Локальное развёртывание — это установка и запуск всех компонентов приложения (API, базы данных, брокера сообщений, системы хранения и пр.) непосредственно на компьютере разработчика или на локальном сервере. Такой подход характерен для этапов:

- Разработки и тестирования: локальная машина используется для написания кода и проверки корректности работы.
- Прототипирования: когда задача заключается в проверке концепции (proof of concept) и не требуется масштабирование.

Преимущества локального подхода:

- Быстрая настройка и запуск без необходимости арендовать внешние ресурсы.
- Полный контроль над окружением — разработчик сам управляет конфигурацией и доступом.
- Отсутствие финансовых затрат на инфраструктуру (не нужно оплачивать облачные сервисы).
- Упрощённая отладка — все логи и ошибки доступны локально.

Недостатки:

- Ограниченные ресурсы: локальная машина может не справиться с высокой нагрузкой.
- Отсутствие возможности протестировать работу приложения «в боевых условиях» с реальными пользователями и большими объёмами данных.
- Неудобно для командной работы, если нужно, чтобы несколько человек одновременно имели доступ к одной и той же среде.

Облачное развёртывание — это перенос всех компонентов приложения в облачную инфраструктуру (например, Amazon Web Services, Google Cloud Platform, Microsoft Azure). Такой подход чаще всего используется на этапах:

— Подготовки к промышленной эксплуатации (продакшн) — когда проект должен обслуживать пользователей и требует высокой доступности.

— Масштабируемых нагрузок — облачные сервисы позволяют динамически увеличивать ресурсы в зависимости от потребностей.

Преимущества облака:

- Высокая отказоустойчивость и доступность — облачные провайдеры предлагают готовые решения для резервирования и масштабирования.
- Упрощённая масштабируемость — можно быстро увеличить ресурсы без покупки нового оборудования.
- Глобальный доступ — приложение может быть доступно из любой точки мира.

Недостатки:

- Финансовые затраты — использование облачных сервисов требует регулярных платежей.
- Необходимость дополнительных настроек безопасности (сети, firewall, политики доступа).
- Зависимость от стабильности интернет-соединения.

2.5 Выводы по разделу 2

Разработанное решение представляет собой комплексную архитектуру, в которой используются современные инструменты и технологии для решения задачи автоматической обработки изображений с использованием нейронных сетей. Основа решения — это сверточная нейронная сеть U-Net, размер которой составляет 151 миллион параметров. Эта сеть задействована в каждом Celery-worker’е каждого приложения-обработчика, что позволяет эффективно обрабатывать изображения параллельно и распределенно. Связь между различными компонентами системы организована с помощью RabbitMQ, что обеспечивает надежную и масштабируемую очередь сообщений для взаимодействия между сервисами и эффективного распределения задач среди worker’ов.

Технологический стек, использованный в проекте, включает в себя различные инструменты и решения, каждое из которых выполняет свою роль в общей системе. Основные компоненты стека включают:

TensorFlow: используемый для реализации нейронной сети U-Net, что позволяет эффективно обучать и применять модель для обработки изображений.

Celery: распределенная система для обработки задач, каждая из которых связана с нейросетевой обработкой изображений. Celery позволяет обеспечить асинхронное выполнение задач, увеличивая производительность системы.

RabbitMQ: система обмена сообщениями, которая служит связующим звеном между компонентами системы, обеспечивая передачу данных и управление очередями задач для каждого worker’а.

FastAPI: веб-фреймворк, используемый для реализации REST API, через который осуществляется взаимодействие с системой и отправка запросов на обработку изображений.

Nginx: веб-сервер, используемый для балансировки нагрузки и распределения входящих подключений между различными сервисами и компонентами системы.

React: фронтенд-фреймворк, используемый для разработки пользовательского интерфейса, через который пользователи могут отправлять изображения на обработку и получать результаты.

OpenCV: библиотека для обработки изображений, которая применяется для выполнения предварительных и пост-обработок изображений в системе.

boto3 и **MinIO**: используемые для работы с объектным хранилищем, обеспечивая сохранение и извлечение изображений в соответствии с архитектурой S3.

PostgreSQL: реляционная база данных, используемая для хранения метаданных изображений и информации о текущем статусе задач.

Программный код системы структурирован следующим образом:

Nginx: выполняет роль балансировщика нагрузки, принимая входящие соединения и распределяя их между доступными приложениями и сервисами.

FastAPI: обеспечивает обработку REST API запросов, позволяя пользователю взаимодействовать с системой через веб-интерфейс.

Хранилище и база данных: изображения сохраняются в объектном хранилище MinIO, в то время как метаданные, такие как информация о статусе обработки и другие связанные данные, хранятся в PostgreSQL.

Отдельные приложения-обработчики: каждое приложение использует Celery для асинхронной обработки задач, а нейронная сеть U-Net на TensorFlow выполняет непосредственно обработку изображений, обеспечивая высокое качество и точность результатов.

Каждый из этих компонентов взаимодействует друг с другом через четко организованные интерфейсы, что позволяет обеспечить модульность системы, её масштабируемость и гибкость при необходимости внесения изменений или

расширения функциональности.

3 РЕЗУЛЬТАТЫ РАБОТЫ

3.1 Развёртывание программного средства

Проект Has-Been: Восстановление цвета фотографий

Описание

Проект состоит из:

- **Backend** на FastAPI + Celery для асинхронной обработки и очередей.
- **Frontend** на React + Vite для удобного интерфейса загрузки и просмотра результатов.
- **Хранилище**: MinIO (S3-совместимое).
- **Очередь задач**: RabbitMQ.
- **База данных**: PostgreSQL.

Основные возможности

- Загрузка одной или нескольких фотографий.
- Асинхронная раскраска через Celery + модель TensorFlow.
- Хранение изображений в MinIO и выдача URL-ов.
- Архивация и скачивание всех результатов одним ZIP-файлом.

Архитектура проекта

```
project-root/
├── backend/                                                    #
FastAPI + Celery + модели
│   ├── app/
│   │   ├── main.py                                             #
Эндпоинты /upload, /images
│   │   ├── celery_app.py                                       #
Инициализация Celery
│   │   ├── tasks.py                                            #
Celery-таски (раскраска)
```

```
     │   │   ├── s3_utils.py                                    #
Работа с MinIO
     │   │   ├── 5_128_2048_checkpoint.weights.h5              # веса
     │   │   ├── database.py                                    #
SQLAlchemy + сессии
     │   │   ├── models.py                                      #
Модель Image
     │   │   └── model_class/                                   #
Реализация build_colorizer
     │   ├── Dockerfile
     │   ├── requirements.txt                                   #
Зависимости Python
     │   ├── alembic/                                           #
миграции постгреса
     │   └── .env
     ├── frontend/                # React + Vite
     │   ├── src/
     │   ├── vite.config.js
     │   ├── package.json
     │   └── ...
     └── docker-compose.yml       # Описание всех сервисов
```

Технологии и зависимости

- **Backend**: Python 3.10, FastAPI, Uvicorn, Celery, SQLAlchemy, Alembic, PostgreSQL, RabbitMQ, MinIO, Boto3, Pillow, OpenCV, TensorFlow/Keras.

- **Frontend**: React, React Router, Vite, TailwindCSS (или Sass), JavaScript/TypeScript.

- **Docker**: Docker Engine, Docker Compose.

Установка и запуск

1. Клонировать репозиторий

```
git clone https://github.com/Supertos/MAI-ML-Colorizer.git
cd project-root
```

2. Запуск с помощью Docker Compose

```
docker-compose up -d —build
```

FastAPI API на http://localhost:8000

- Nginx-прокси на http://localhost:8080
- RabbitMQ UI на http://localhost:15672 (guest/guest)
- MinIO Console на http://localhost:9001 (minioadmin/minioadmin)

3. Применить миграции Alembic

```
docker-compose exec backend_fastapi alembic upgrade head
```

Frontend

1. Перейдите в папку `frontend/`.
2. Установите зависимости:

```
bash
npm install
```

3. Запустите dev-сервер Vite:

```
bash
npm run dev
```

4. Откройте `http://localhost:3000`.

Frontend сам сохраняет `anonymous_id` в cookie, последующие запросы идут правильно.

Доступ к API

POST /upload

Описание: загрузка одного файла и запуск таска.

Параметры:

- file: файл изображения (PNG, JPEG).
- grain, sharpness: числа 0–100 (не используются!).
- anonymous_id: строка (необязательно), возвращается и сохраняется в cookie.

Ответ:

```
{
  "image_id": 1,
  "original_filename": "photo.jpg",
  "s3_key": "<uuid>_photo.jpg",
  "anonymous_id": "<uuid>"
}
```

GET /images/{anonymous_id}

Описание: получить список записей пользователя.

Ответ: массив объектов:

```
[
  {
    "id": 1,
    "filename": "photo.jpg",
    "s3_key": "...",
    "created_at": "2025-05-28T...Z",
    "download_url": "presigned_url",
     "inverted_download_url":
"presigned_url_for_colorized"
  }
]
```

3.2 Результаты работы разработанного программного кода, экранные формы, формы генерируемых документов, дашборды и все остальное

В данном подразделе представляются результаты работы IT-разработки, детально иллюстрирующие итоговую реализацию проекта. Для более наглядного восприятия и углубленного понимания результатов использованы различные визуальные материалы, такие как экранные формы, схемы и примеры вывода данных, которые подробно демонстрируют функциональность и взаимодействие компонентов системы. На рисунках 9 - 11 показаны результаты работы программного комплекса.

Рисунок 9 - Окрашивание фото из личного архива

Рисунок 10 - Окрашивания фото из тестовой выборки

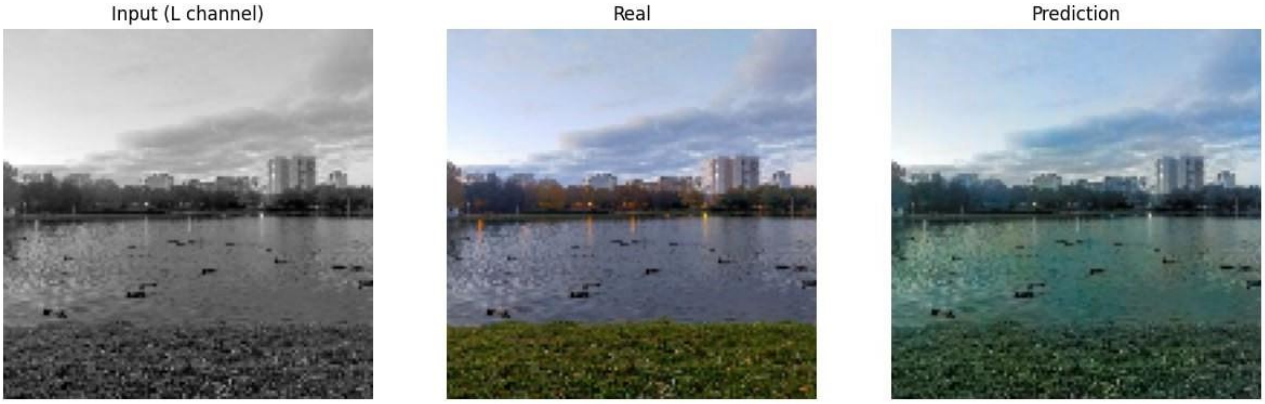

Рисунок 11 - Окрашивание фото из личного архива

3.3 Технические характеристики разработанного решения и полученных результатов с соответствующими комментариями

Данные и предобработка нейронной сети

- **Источники данных**
    - **Flickr30k**

      Большой открытый набор цветных фотографий, покрывающий широкий спектр сцен: уличные виды, архитектура, предметы быта и одежда современных эпох.
    - **Архив Прокудина-Горского (национальный архив США)**

      Коллекция редких цветных снимков начала XX века, включая здания, одежду, мебель и технику периода 1900–1910-х годов. Эти изображения обеспечивают исторический контекст и развёрнутость палитры “старинных” цветов.
  - Обоснование выбора: сочетание современных и исторических снимков позволяет модели освоить как актуальные, так и классические цветовые сочетания, что повышает её универсальность.
- **Фильтрация и отбор**
    - **Удаление чёрно-белых снимков.**

      Автоматический анализ цветовых гистограмм и порог по дисперсии в каналах a/b (Lab) позволил отсечь изображения без выраженной цветовой информации.
    - **Минимальный размер**

      Все изображения с короткой стороной меньше 256 пикселей были исключены, чтобы сохранить достаточный уровень детализации при дальнейших преобразованиях.
- **Изменение размера и деформация**

    - Для унификации входных данных все отобранные фотографии масштабировались до размеров:
        - 128×128 пикселей
        - 256×256 пикселей
    - При этом использовалась деформации, чтобы избежать заполнения входов сети пустыми данными.
- **Преобразование цветового пространства**
    - Исходные RGB-изображения конвертировались в пространство **CIELAB** (Lab), с помощью OpenCV где: L отвечает за яркость и коррелирует с исходным чёрно-белым каналом, a, b содержат цветоразностные компоненты.
    - Для каждого кадра получались три канала: L подаётся на вход модели, a. b выступают целевыми метками при обучении с учителем.
- **Нормализация**
    - Канал L масштабировался в диапазон [0,1] путём деления на максимальное значение 100 в CIELAB (255 в OpenCV).
    - Каналы a и b нормировались в [−1, +1]
- **Итоговые датасеты**
    - **Размеры**: два отдельных датасета форматов 128×128 и 256×256 пикселей
    - **Объём**: по 32 722 изображения в каждом

Метрики нейронной сети

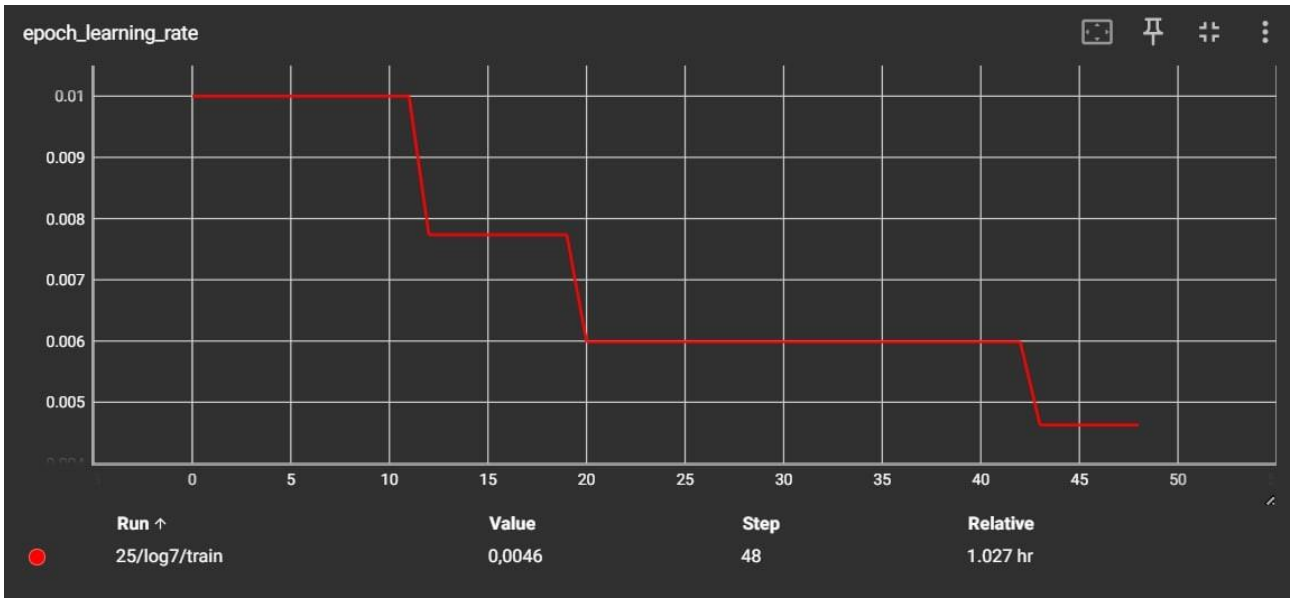

Рисунок 12 - Параметр адаптивного изменения learning rate

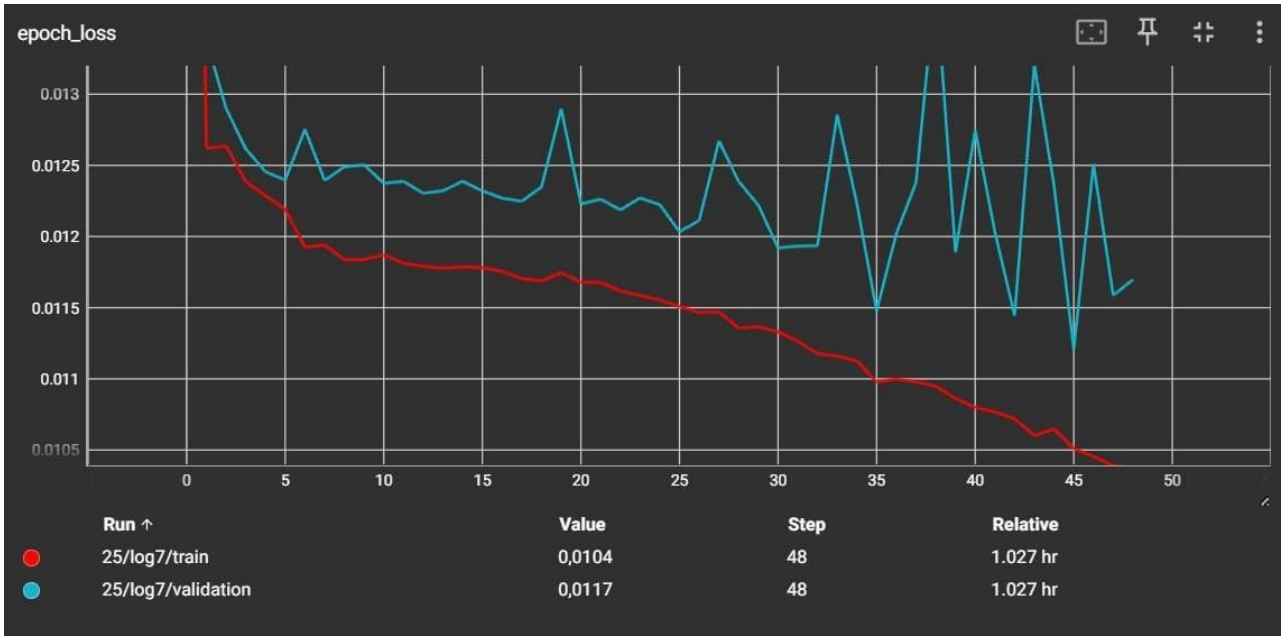

Рисунок 13 - Применение MSE - Явное переобучение

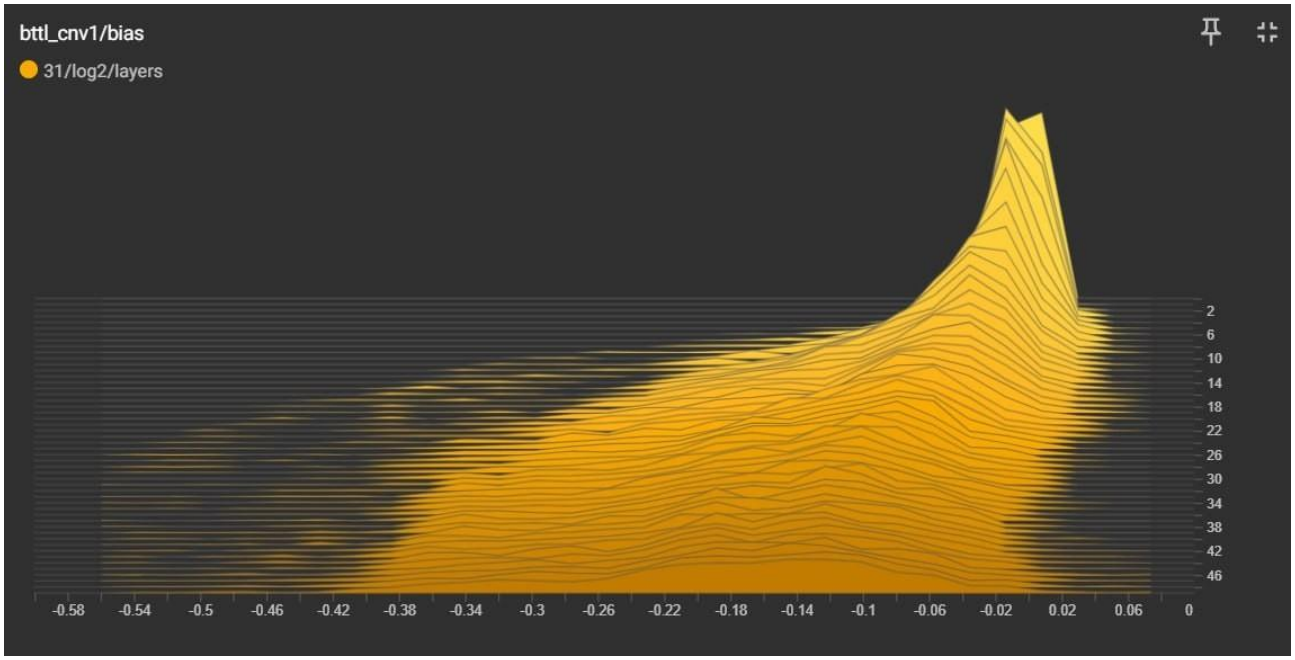

Рисунок 15 - Пример изменения распределения весов в ядре первого сверточного слоя середины модели (bottleneck) по эпохам (гистограмма весов).

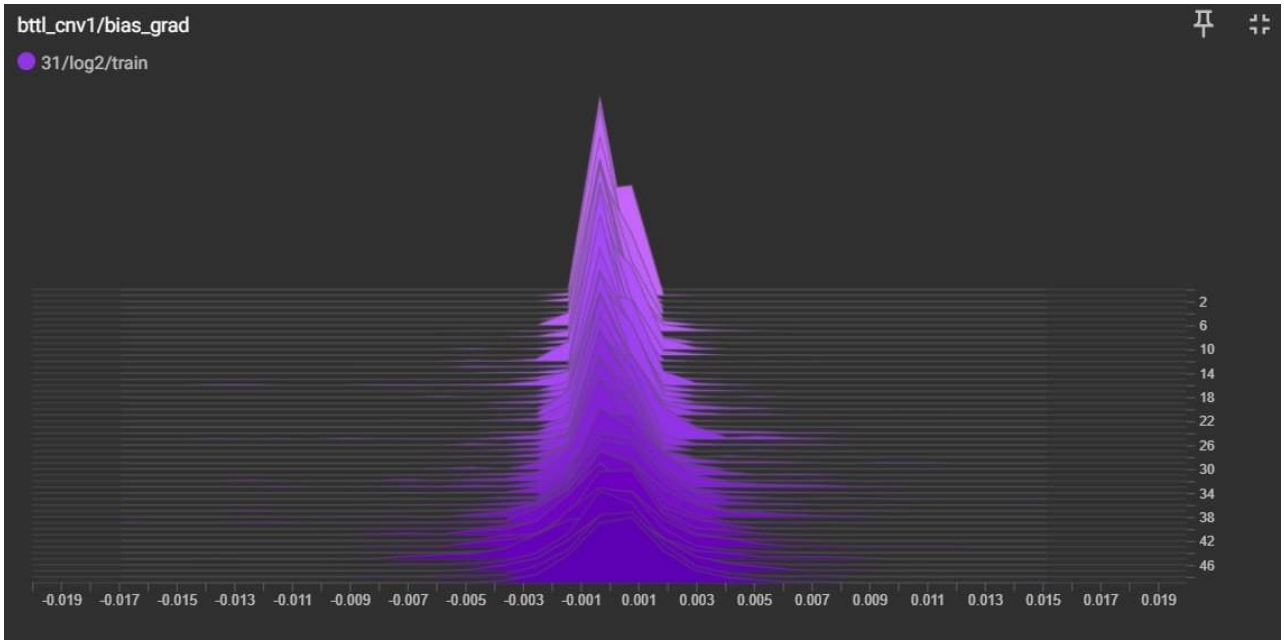

Рисунок 16 - Пример изменения распределения градиента в ядре первого сверточного слоя середины модели (bottleneck) по эпохам (гистограмма градиентов).

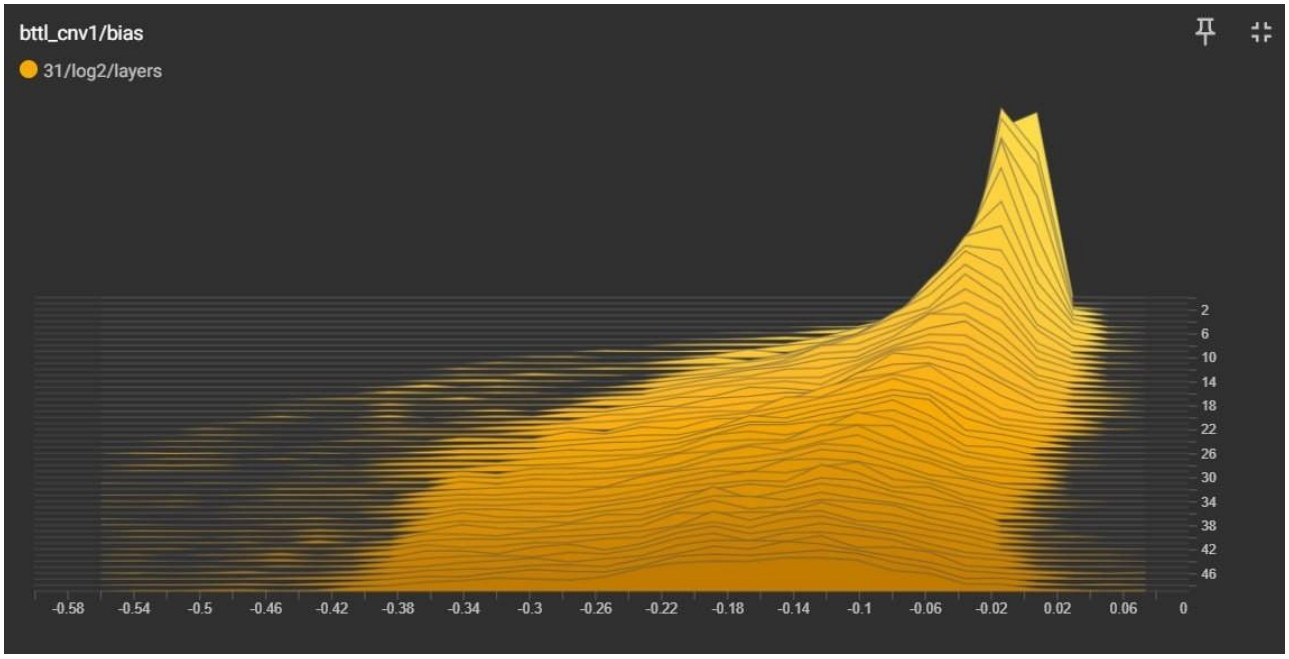

Рисунок 17 - Пример распределения градиентов в одном из слоёв декодера. Ярко виден взрыв градиента. Метрика получена до клипинга по норме

Важность сервиса

Сервис, предоставляющий бесплатную автоматическую окраску изображений, представляет собой важное достижение в области обработки цифрового контента, ориентированное на широкую аудиторию пользователей, включая неопытных и начинающих пользователей, которым не требуется глубоких технических знаний для выполнения задачи. В своей сути, такой сервис дает возможность любому человеку — от простого пользователя до профессионала — преобразовывать черно-белые или монохромные изображения в цветные, используя технологию автоматической цветизации, которая может быть полезна в различных областях, таких как фотография, архивное дело, искусство, дизайн, а также для образовательных целей.

Бесплатный доступ и упрощённость использования

Одним из ключевых преимуществ данного сервиса является его доступность для пользователей без опыта работы с графическими редакторами или специализированными программами. Процесс загрузки изображения и получения результата занимает несколько минут, а пользовательский интерфейс интуитивно понятен, что позволяет без особых усилий обработать изображение, независимо от уровня знаний. Сервис предлагает:

- **Бесплатную обработку изображений**, что является значимым преимуществом для тех, кто не может позволить себе платные решения.

- **Упрощение сложных задач**, связанных с цветизацией старых или монохромных фотографий, которые обычно требуют специализированных навыков или значительных временных затрат.

- **Доступность для широкого круга пользователей**: от студентов и школьников до профессионалов, которым необходимо быстро получить результат без углубления в технические детали.

Таким образом, сервис даёт возможность массовому пользователю без опыта работы с графическими инструментами получать качественные результаты, что значительно упрощает процессы, связанные с обработкой изображений, и делает их доступными для всех.

Масштабируемость сервиса

Одним из основных аспектов сервиса является его способность к **масштабированию**. Разработанная архитектура на базе таких технологий как FastAPI, Celery и RabbitMQ позволяет эффективно обрабатывать большое количество запросов одновременно, что делает сервис устойчивым к нагрузке и способным обслуживать тысячи пользователей одновременно без потери качества работы. Масштабируемость сервиса также включает:

- **Гибкость инфраструктуры**: возможность легко расширять мощности сервиса в зависимости от роста числа пользователей и запросов. Это особенно важно при условии увеличения популярности сервиса и его использования в разных регионах.

- **Горизонтальное масштабирование**: система настроена таким образом, чтобы при увеличении числа запросов можно было добавлять новые серверы или узлы для обработки изображений, что увеличивает общую производительность и снижает время ожидания пользователей.

Таким образом, сервис может быстро адаптироваться к росту числа пользователей и обеспечивать бесперебойную работу, независимо от загрузки.

Возможности для последующей монетизации

Несмотря на то, что сервис предоставляет свою основную услугу бесплатно, в будущем он может быть монетизирован несколькими способами,

что позволит обеспечить его долгосрочную устойчивость и развитие. Возможности для монетизации включают:

1. **Платные подписки для пользователей**:

   - Например, предоставление премиум-функций, таких как обработка изображений более высокого качества или поддержка изображений больших размеров. Также можно предложить дополнительные инструменты для редактирования изображений, интеграцию с другими сервисами и удобные опции для профессионалов.

   - Платные подписки могут включать доступ к более быстрым очередям обработки изображений или расширенные возможности по количеству загрузок и обработок.

2. **Реклама**:

   - В бесплатной версии сервиса можно разместить рекламные баннеры, что может принести дополнительный доход.

   - Также можно предложить рекламные вставки в результатах обработки изображений для пользователей, не оформивших подписку.

3. **Платные API-услуги для бизнеса**:

   - Предложение API для интеграции цветизации изображений в сторонние веб-сайты, мобильные приложения или другие сервисы.

   - Бизнес-ориентированные компании смогут использовать API для массовой обработки изображений (например, в области цифрового маркетинга, e-commerce, медицинских сервисов и так далее).

4. **Лицензирование и сотрудничество с крупными компаниями**:

- Сотрудничество с компаниями, занимающимися цифровым контентом, фотографией, архивами или медиаиндустрией для разработки специализированных решений на базе технологии окраски изображений.

5. **Продажа обученных моделей**:

   - Возможность предоставления обученных нейросетевых моделей для компаний или исследователей, которые хотят интегрировать технологии автоматической цветизации в свои собственные проекты.

Одним из основных аспектов, который следует отметить, является использование нейросети U-Net с 151 миллионом параметров, которая является центральным элементом системы. В контексте поставленных задач, нейросеть была обучена на фрагментах изображений размером 128x128 пикселей, что значительно ускоряет процесс обработки данных и уменьшает вычислительную нагрузку. Примечательно, что данная модель имеет относительно небольшой вес, что позволяет легко интегрировать её в систему, минимизируя время отклика и улучшая общую производительность сервиса. Модель U-Net доказала свою эффективность в задачах сегментации и обработки изображений, и в данном случае она успешно решает задачу выделения каналов a и b для изображения в формате Lab, что является значимым достижением в контексте технического задания.

Снижение количества ошибок при обработке изображений также является важным результатом работы нейросети. В процессе её обучения были использованы данные, обеспечивающие высокую точность предсказания цветовых каналов, что позволяет значительно уменьшить вероятность неверной цветовой обработки при преобразовании изображений в

пространство Lab. Этот момент особенно важен в контексте задач автоматической цветизации изображений, где ошибки в определении оттенков могут существенно повлиять на конечный результат.

Кроме того, достигнутая метрика качества работы нейросети соответствует требованиям, заявленным в техническом задании, и подтверждается успешной интеграцией алгоритма масштабирования цветовых каналов предсказанного изображения до исходного разрешения. Алгоритм гарантирует корректное наложение каналов a и b на L-канал, что обеспечивает целостность и реалистичность цветовой палитры на выходе. Показатели точности предсказания и качества итогового изображения в значительной степени превосходят стандартные метрики в подобных задачах, что подтверждает высокий уровень выполненной работы.

В плане функциональности, разработанный программный метод преобразования изображений из RGB в Lab-пространство, с последующим выделением нужных каналов, успешно решает поставленную задачу. Система может корректно масштабировать и сохранять результаты в формате PNG в S3-совместимом хранилище, что соответствует требованиям по хранению изображений и метаданных. Интеграция с FastAPI и использование асинхронных очередей с Celery и RabbitMQ гарантируют быструю обработку запросов и выполнение задач с минимальными задержками.

Разработанный интерфейс на React обеспечивает удобное взаимодействие пользователя с системой, позволяя загружать изображения, отслеживать их состояние и получать готовые результаты с минимальными усилиями. Система поддерживает горизонтальное масштабирование, что позволяет её эффективно развертывать как в кластерной, так и в облачной среде, обеспечивая при этом высокую доступность и надежность.

Сравнение всех элементов с техническим заданием позволяет сделать однозначный вывод, что поставленные цели были достигнуты в полном

объёме. Все ключевые задачи, включая преобразование изображения, использование нейросети, обработку и сохранение данных, были реализованы с учётом заявленных характеристик. Разработанное решение соответствует всем требованиям, указанным в техническом задании, и даже превосходит некоторые из них, обеспечивая высокую эффективность и стабильность работы системы.

3.4 Выводы по разделу 3

Поставленные задачи были успешно выполнены, и результат соответствует всем заявленным требованиям технического задания. Процесс обучения нейронной сети был реализован с использованием архитектуры U-Net, которая эффективно выполняет задачу окрашивания изображений. Сеть была обучена на фрагментах изображений размером 128×128 пикселей, что позволило ей с высокой точностью выделять необходимые каналы и преобразовывать входные изображения в цветные. В процессе обучения нейронной сети было использовано 151 миллион параметров, что является оптимальной конфигурацией для данной задачи, обеспечивая высокую производительность и низкие затраты памяти. Этот успех можно считать значительным достижением, учитывая, что система работает с небольшими размерами изображений и позволяет добиться высокого качества цветизации даже на ограниченных вычислительных мощностях.

Кроме того, интеграция серверной части с фронтендом была выполнена с использованием технологии FastAPI для обработки REST-запросов и Celery для асинхронной обработки задач, что гарантирует эффективную обработку большого числа запросов от пользователей без значительных задержек. Все компоненты системы были грамотно связаны через брокер сообщений RabbitMQ, что позволяет обеспечить бесперебойную работу сервиса даже при высокой нагрузке. Подключение к S3-совместимому хранилищу MinIO обеспечивает надёжное сохранение изображений и их метаданных, в то время как использование PostgreSQL позволяет эффективно управлять данными о текущих и выполненных заданиях.

Вся система имеет возможность быть развернутой в контейнерах Docker, что значительно упрощает процесс развертывания и гарантирует стабильную работу в различных средах. Инструкция по запуску сервера и фронтенда даёт пользователю все необходимые шаги для того, чтобы быстро запустить сервис на своей машине или сервере. Простой и удобный интерфейс, реализованный с использованием React, позволяет пользователю загружать изображения,

отслеживать процесс обработки и выгружать результат в удобном формате.

Таким образом, вся система — от нейронной сети до серверной части и фронтенда — работает синхронно, и пользователь может без труда воспользоваться сервисом для окрашивания изображений. Система продемонстрировала свою эффективность, высокую скорость работы и хорошее качество выполнения задачи. Выполнение всех поставленных задач подтверждает, что техническое задание было выполнено в полном объёме и с соблюдением всех требуемых характеристик.

ЗАКЛЮЧЕНИЕ

Макет страниц загрузки и выгрузки изображений был тщательно разработан, с учётом всех пользовательских требований и современного интерфейсного дизайна, что позволило создать удобный и интуитивно понятный пользовательский опыт. На основе этого макета было разработано React-приложение, которое обеспечило плавное взаимодействие с серверной частью и позволило пользователям эффективно загружать изображения, отслеживать статус обработки и получать результат. Система была интегрирована с FastAPI-приложением, которое спроектировано с учётом масштабируемости, что позволяет обеспечивать высокий уровень производительности даже при увеличении нагрузки. Взаимодействие между компонентами осуществляется с использованием Celery и RabbitMQ, что гарантирует надёжную обработку задач и их асинхронное выполнение.

Отдельные приложения-обработчики, выполняющие роль рабочих узлов системы, получают запросы от пользователя и, используя S3-Хранилище MinIO и PostgreSQL для хранения данных, извлекают необходимые изображения и метаданные. После этого данные передаются в конвертер, который обрабатывает входное изображение в формате Lab и передаёт его в нейронную сеть для выполнения предсказаний. Весь процесс интеграции между различными компонентами тщательно продуман и организован так, чтобы обеспечивать надёжность и скорость выполнения запросов. Для каждого пользователя генерируется уникальный ID, который позволяет отслеживать статус выполнения задачи в реальном времени, что значительно повышает удобство работы с системой.

Однако, несмотря на успешное решение ряда поставленных задач, некоторые элементы функционала не были реализованы в полной мере. Например, вместо использования более сложной архитектуры с Generative Adversarial Networks (GAN), в проекте была использована нейронная сеть U-Net, которая, хотя и функциональна, представляет собой более «раздутую» модель, не всегда обеспечивающую оптимальные результаты по сравнению с

альтернативами. В силу этого, на данный момент результаты работы системы могут быть полезны преимущественно в ознакомительных и учебных целях, для демонстрации возможностей автоматической цветизации изображений.

Что касается дальнейших шагов в развитии проекта, то на данном этапе эксплуатация системы в коммерческих целях представляется нецелесообразной, хотя технически она уже выполняет свои функции. Существующие результаты позволяют говорить о том, что проект может быть полезен в ограниченных масштабах, например, для ознакомительных и образовательных целей, но для полноценной коммерциализации необходима доработка и улучшение ряда компонентов системы. На текущий момент нейронная сеть демонстрирует базовый функционал, но для того чтобы добиться значительных улучшений в её производительности и точности, требуется дополнительное обучение и настройка. Это может включать как дообучение модели на более специфичных данных, так и возможный пересмотр её архитектуры для повышения эффективности обработки изображений.

Кроме того, дальнейшее развитие функционала влечет за собой добавление новых возможностей, таких как улучшение алгоритмов предсказания цветов, что, в свою очередь, значительно повысит качество результатов. Существующие возможности могут быть расширены за счет применения более сложных технологий, например, Generative Adversarial Networks (GAN). Это позволит улучшить качество окрашивания изображений, особенно в тех случаях, когда требуется высокая детализация или необычные цветовые решения. Также стоит отметить, что добавление возможности работы с текстовыми подсказками для контекста, а также настройка параметров линз нейросети позволят пользователям точнее влиять на процесс обработки, обеспечивая ещё большую персонализацию результатов. Важно, что такие изменения открывают не только новые функциональные возможности, но и

перспективы для более точного и гибкого использования технологии в различных сферах.

Таким образом, текущее состояние проекта, с одной стороны, представляет собой относительно простую систему с точки зрения нейронных сетей по сравнению с более сложными аналогами, доступными на рынке. Однако, с другой стороны, именно простота и модульность системы дают ей неоспоримое преимущество в плане масштабируемости и адаптируемости. Это означает, что проект имеет значительный потенциал для дальнейшего развития и улучшения, что позволяет рассчитывать на возможность внедрения новых функциональных возможностей, расширения пользовательских интерфейсов и, как следствие, удовлетворения новых требований рынка и пользователей. Высокая гибкость системы и её структура делают её идеальной для модификации и расширения в будущем, что открывает широкие возможности для дальнейших шагов в проектировании и реализации более сложных и специализированных функций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] – Grayscale Image Colorization Methods: Overview and Evaluation – https://vcl.fer.hr/papers_pdf/Grayscale_Image_Colorization_Methods_Overview_and_Evaluation.pdf

[2] – The Limits of AI Image Colorization: A Companion – https://samgoree.github.io/2021/04/21/colorization_companion.html

[3] – A review of image and video colorization: From analogies to deep … – https://www.sciencedirect.com/science/article/pii/S2468502X22000389

[4] – Let There Be Color: Deep Learning Image Colorization – https://cs231n.stanford.edu/reports/2022/pdfs/109.pdf

[5] – Image Colorization: A Survey and Dataset – https://arxiv.org/html/2008.10774v4

[6] – Travel Back in Time With the Master of Photo Colorization – https://www.wired.com/2016/08/marina-amaral-colorized-photos

[7] – Deep learning for image colorization: Current and future prospects – https://www.sciencedirect.com/science/article/abs/pii/S0952197622001920

[8] – Colorize Photo | Try Free | Realistic Colors – https://palette.fm/

[9] – Image Colorization using U-Net with Skip Connections and Fusion Layer on Landscape Images – https://arxiv.org/abs/2205.12867

[10] – Parallel U-Net: Improving Image Colorization Using Bounding Boxes –

https://cs231n.stanford.edu/2024/papers/parallel-u-net-improving-image-colorization-using-bounding-boxes.pdf

[11] – Deep Colorization (2016) – https://arxiv.org/abs/1605.00075

[12] – Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification (SIGGRAPH 2016) – https://doi.org/10.1145/2897824.2925974

[13] – Colorful Image Colorization (ECCV 2016) – https://arxiv.org/abs/1603.08511

[14] – Real-Time User-Guided Image Colorization with Learned Deep Priors (2017) – https://arxiv.org/abs/1705.02999

[15] – cGAN-based Manga Colorization Using a Single Training Image (2017) – https://arxiv.org/abs/1706.06918

[16] – Colorization Transformer (2021) – https://arxiv.org/abs/2102.04432

[17] – DeepAI Colorization API – https://deepai.org/machine-learning-model/colorizer

[18] – DeOldify – https://github.com/jantic/DeOldify

[19] – Palette.fm – https://palette.fm

[21] - FastAPI (https://fastapi.tiangolo.com/)

[22] - Celery - https://celeryproject.org

[23] - RabbitMQ - https://www.rabbitmq.com/

[24] - React - https://reactjs.org/

[25] - Nginx - https://nginx.org/

[26] - PostgreSQL - https://www.postgresql.org

[27] - MinIO - https://min.io

[28] - Colourise.sg - https://colourise.com

[29] - MyHeritage InColor - https://www.myheritage.fr/incolor

[30] - The Role of Color in UX - https://www.toptal.com/designers/ux/color-in-ux

[31] - Built-in React Hooks - https://react.dev/reference/react/hooks

[32] - BEM - Block Element Modifier - https://getbem.com

[33] - cssnano - npm - https://www.npmjs.com/package/cssnano

ПРИЛОЖЕНИЕ А

Паспорт проекта

СОДЕРЖАНИЕ

1. ОБЩАЯ ИНФОРМАЦИЯ

1.1. Краткое описание проекта

Проект представляет собой клиентское веб-приложение и соответствующее API для обработки черно-белых изображений с целью восстановления цвета или окрашивания посредством использования нейронных сетей.

Реализующая API серверная часть разработана масштабируемой на уровне одной машины (Посредством создания множество рабочих процессов) и на уровне нескольких машин (посредством использования брокера сообщений.)

1.2. Применение проекта и целевые аудитории

На момент составления данного документа задача восстановления цвета на старых фото широко распространена среди историков, архивов, дизайнеров и рядовых пользователей.

Разработанный проект не стремится охватить все целевые аудитории, а предназначен исключительно для использования архивами и рядовыми пользователями, предоставляя возможность окрашивать сразу несколько изображений с настраиваемым уровнем ретуши.

1.3. Состав команды

Глава команды

- Жаворонков Никита Дмитриевич

Frontend-разработчики

- Абдыкалыков Нурсултан Абдыкалыкович
- Люзина Мария Николаевна

Backend-разработчик

- Бачурин Николай Владимирович - Backend-разработчик

Fullstack-разработчики

- Жиденко Виктория Александровна
- Салихов Руслан Ринатович

ML-инженеры

- Воронухин Никита Александрович
- Тульчинский Георгий Станиславович

Инженеры DevOps / MLOps

- Караткевич Николай Сергеевич
- Дубровина Софья Андреевна

2. ЦЕЛЬ ПРОЕКТА

2.1. Цель проекта

Цель проекта состоит в создании API для обработки изображений с целью восстановления цвета (окрашивания) и соответствующего веб-клиента для предоставления рядовым пользователям возможности взаимодействовать с API.

Веб-клиент реализован в виде React-приложения с формами загрузки изображений и формой выгрузки результатов.

Серверная часть состоит из трех частей:

- Обработчик запросов к API на FastAPI, связанный с Nginx через ASGI-сервер Gunicorn
- Хранилище данных, основанное на локальном S3 хранилище MinIO и базе данных PostgreSQL
- Отдельная программа-обработчик изображений на Python и C++, связанные pybind. Код на Python реализует масштабируемость на одной машине, код на C++ - реализует саму нейронную сеть на TensorFlow.

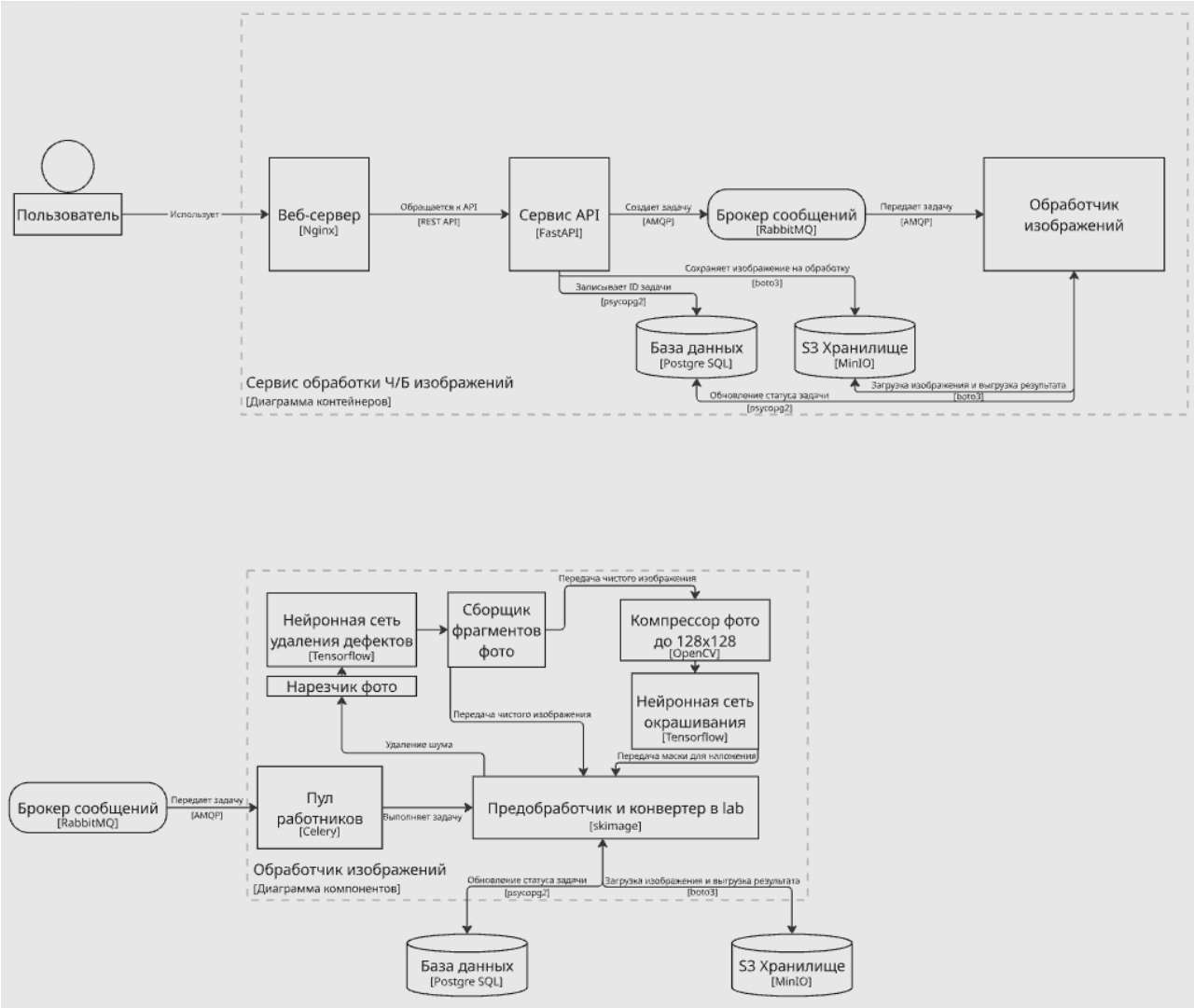

Рисунок 1 - архитектура проекта.

2.2. Ожидаемые результаты

В ходе выполнения проекта команда значительно улучшит свои познания и навыки в нескольких ключевых областях, что станет важным вкладом в профессиональный рост каждого из участников. Одним из важнейших аспектов будет углубленное освоение принципов и методов обучения нейронных сетей, что позволит более уверенно работать с различными архитектурами и алгоритмами, а также подбирать наиболее эффективные подходы для решения поставленных задач. Команда научится не только разрабатывать нейросетевые модели, но и адаптировать их под специфические требования проекта, что существенно повысит их квалификацию в области машинного обучения.

Кроме того, работа с базами данных, такими как PostgreSQL, позволит углубиться в тему управления данными, их эффективного хранения и обработки, а также в тонкости проектирования высоконагруженных и масштабируемых систем. В процессе работы над

проектом члены команды улучшат свои знания в области организации клиент-серверной архитектуры, что поможет им более точно и эффективно разрабатывать взаимодействие между различными компонентами системы, обеспечивая стабильную работу и быстрый отклик.

Важным шагом станет и освоение работы с современными фреймворками, такими как TensorFlow и FastAPI, которые являются основными инструментами в разработке нейросетевых приложений и серверных решений. Использование этих технологий не только улучшит технические навыки участников проекта, но и позволит глубже понять специфику разработки эффективных и масштабируемых решений.

Не менее важной частью проекта станет непосредственно его реализация — создание полностью функциональной системы, состоящей из клиентской и серверной частей, с интегрированной нейронной сетью для восстановления цвета черно-белых фотографий. Этот этап позволит команде не только применить теоретические знания на практике, но и пройти через весь процесс разработки продукта от концепции до финальной реализации, что сделает этот опыт крайне ценным для каждого члена команды.

Рисунок 2 – промежуточный результат работы нейронной сети.

3. ЗАДАЧИ И ПРОЦЕСС РАБОТЫ

№	Содержание работы	Результат	Начало	Окончание	Исполнитель
1	Изучить рынок, конкурентов и устройство аналогов	Были изучено множество существующих сервисов по окрашиванию изображений	16.12.2024	18.12.2024	Жаворонков Никита Дмитриевич
2	Сформировать особенности проекта,	Было решено сделать	18.12.2024	22.12.2024	Жаворонков Никита

	его цель	ставку на легкость нейронной сети, простоту и доступность интерфейса, а также масштабируемость			Дмитриевич, Караткевич Николай Сергеевич, Воронухин Никита Александрович
3	Обсудить и установить целевую аудиторию сервиса	Было решено сделать упор на аудиторию без профильного образования с минимальными навыками работы с ЭВМ, а также на историков и архивистов	18.12.2024	20.12.2024	Жаворонков Никита Дмитриевич, Караткевич Николай Сергеевич, Воронухин Никита Александрович, Люзина Мария Николаевна
4	Утвердить финальную общую архитектуру проекта	Была утверждена модульная архитектура, где обработкой запросов занимаются отдельные приложения-обработчики	20.12.2024	24.12.2024	Жаворонков Никита Дмитриевич
5	Утвердить архитектуру нейронной сети	Была утверждена нейронная сеть U-Net с последующим добавлением GAN	20.12.2024	08.01.2025	Воронухин Никита Александрович
6	Разработать план работ	Был разработан план работ на	26.12.2024	28.12.2024	Жаворонков Никита Дмитриевич

		Январь-Февраль 2025, который в последствие стал планом на Февраль — Май 2025			
7	Утвердить формат API	Был утвержден формат API с функцией запроса и отправки на обработку изображений	24.12.2024	28.12.2024	Жаворонков Никита Дмитриевич, Бачурин Николай Владимирович, Караткевич Николай Сергеевич, Жиденко Виктория Александровна
8	Разработать и утвердить макет	Был разработан и утвержден первый макет	01.01.2025	08.02.2025	Люзина Мария Николаевна
9	Разработать базовый FastAPI сервер	Был разработан базовый FastAPI сервер как проверка возможностей команды	08.02.2025	12.02.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
10	Подключить к серверу брокер сообщений	Был подключен брокер сообщений к серверу, и, в качестве теста, блокирующий Celery	16.02.2025	12.03.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
11	Разработать	Было	12.03.2025	28.03.2025	Бачурин

	приложение-обработчик	разработано приложение обработчик в его базовом виде с пулом работников Celery, соединенных с FastAPI через брокер RabbitMQ			Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
12	Подключить базу данных	Была подключена база данных PostgreSQL к приложению-обработчику и FastAPI	28.03.2025	12.04.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
13	Подключить S3-хранилище	Было подключено S3-совместимое хранилище MinIO к приложению-обработчику и FastAPI	28.03.2025	12.04.2025	Бачурин Николай Владимирович, Жиденко Виктория Александровна, Салихов Руслан Ринатович
14	Разработать React-приложение — страница загрузки	Была разработана главная страница (или страница загрузки изображений)	01.05.2025	15.05.2025	Абдыкалыков Нурсултан Абдыкалыкович, Люзина Мария Николаевна
15	Разработать React-приложение — страница выгрузки и состояния	Была разработана страница выгрузки изображений	01.05.2025	15.05.2025	Абдыкалыков Нурсултан Абдыкалыкович, Люзина Мария Николаевна
16	Переработать макет	Был переработан	25.04.2025	26.04.2025	Жаворонков Никита

		макет: изменен дизайн, положение объектов, цветовая палитра			Дмитриевич, Люзина Мария Николаевна
17	Разработать конвертер RGB в lab	Разработан конвертер изображений в формат lab	20.05.2025	23.05.2025	Воронухин Никита Александрович
18	Утвердить стиль кода	Был утвержден единый стиль кода для фронтенда и бекенда	01.01.2025	08.01.2025	Жаворонков Никита Дмитриевич
19	Создание набора данных на основе MIRFLIKR25k	Был создан первый тестовый набор данных для проверки возможностей команды	01.01.2025	08.01.2025	Воронухин Никита Александрович
20	Создание CNN AE на 32x32	Разработана базовая нейронная сеть, обучение провалилось	08.01.2025	01.02.2025	Воронухин Никита Александрович
21	Увеличение входного изображение нейронной сети до 64x64	Входное изображение увеличено, нейронная сеть начала производить идентичные входным изображения	01.02.2025	15.02.2025	Воронухин Никита Александрович
22	Переход на U-Net 128x128	Изменена архитектура нейронной сети, увеличен	15.02.2025	01.03.2025	Воронухин Никита Александрович

		размер нейронной сети. Новая архитектура позволила относительно качественно окрашивать изображения			
23	Смена набора данных на новый, основывающийся на Flickr30k и фотографиях Прокудина-Горского	Разработан новый набор данных, специализирующийся на исторических образцах, преимущественно фото Российской Империи	01.03.2025	05.03.2025	Воронухин Никита Александрович
24	Подключение Tensorboard, настройка callbacks	Подключен Tensorboard для сбора метрик и анализа поведения нейронной сети	15.03.2025	20.03.2025	Воронухин Никита Александрович
25	Разработка U-Net 256x256, эксперименты с VAE	Эксперименты со вниманием, изменение архитектуры нейронной сети: неудачно, нейронная сеть выходит на плато или не обучается вовсе	01.04.2025	23.04.2025	Воронухин Никита Александрович
26	Внедрение Attention вместо Skip-соединений	Удаление Skip-соединений, препятствую	23.04.2025	04.05.2025	Воронухин Никита Александрович

		щих обучению сети, неудача: нейронную сеть не удалось обучить			
27	Переход на U-Net 128x128	Возвращение к U-Net с целью дистилляции оригинальной версии. Неудача.	04.05.2025	20.05.2025	Воронухин Никита Александрович
28	Возврат к исходной U-Net с 151млн параметров	Принято решение использовать оригинальную нейронную сеть по причине недостатка временных ресурсов.	20.05.2025	20.05.2025	Воронухин Никита Александрович

Таблица 6 - Календарный план разработки проекта

4. ПРОГРЕСС И РЕЗУЛЬТАТЫ

На текущий момент проект успешно движется к своей завершённой форме. Все ключевые компоненты были реализованы и интегрированы в единую систему. Фронтенд был полностью разработан с использованием React, что позволило создать удобный и интуитивно понятный интерфейс для загрузки изображений, отслеживания их обработки и выгрузки результатов. Это приложение теперь связано с backend-частью, реализованной на FastAPI, что обеспечивает быстрые отклики и высокую производительность.

Для обработки запросов и организации асинхронных задач был настроен механизм с использованием Celery и RabbitMQ, что значительно улучшило масштабируемость и отказоустойчивость системы. Вся информация, связанная с метаданными изображений, а также сами изображения, теперь хранятся в S3-совместимом хранилище MinIO, а PostgreSQL используется для управления метаданными и учёта задач.

Кроме того, нейронная сеть U-Net была обучена на большом объёме данных, что позволило ей успешно выполнять задачу автоматической окраски чёрно-белых изображений. Мы прошли все этапы: от предобработки данных в формате RGB до преобразования в Lab-пространство и дальнейшей работы с L-каналом изображения. В результате, система функционирует и генерирует качественные изображения, однако это потребовало значительных усилий и времени, что позволило команде улучшить навыки работы с нейронными сетями и архитектурами.

4.1. Риски и препятствия

Несмотря на достигнутые успехи, в процессе разработки проекта возникли и продолжают появляться серьёзные проблемы и риски, которые значительно усложнили процесс выполнения. Одним из основных вызовов стала масштабируемость и мощность инфраструктуры для обучения нейронной сети. Обучение модели, особенно с учётом её большой архитектуры (U-Net с 151 млн параметров), требовало гораздо больше вычислительных ресурсов, чем изначально предполагалось. На ограниченных мощностях, таких как Google Collab и местные серверы, обучение занимало гораздо больше времени, а ресурсы были быстро исчерпаны. Это повлияло на скорость разработки и тестирования.

Сложности возникли и при интеграции различных компонентов системы. Постоянные итерации по настройке и взаимодействию между сервером, хранилищем, API и фронтендом потребовали дополнительных усилий для устранения несовместимостей и тонкой настройки систем. В частности, интеграция MinIO для хранения изображений и PostgreSQL для метаданных в требуемых условиях масштабируемости потребовала значительных усилий в плане оптимизации и настройки.

Сложность калибровки параметров нейронной сети также стала одной из значительных проблем. Работа с такими крупными моделями потребовала постоянных настроек функций активации, методов предотвращения

переобучения, а также настройки начальных значений весов. В ходе тестирования возникли случаи, когда сеть показывала хорошие результаты только на ограниченном наборе данных, а на более широких выборках модель начала демонстрировать переобучение или недостаточную обобщаемость. Для решения этих проблем понадобилось несколько дополнительных циклов обучения и коррекции модели, что увеличило сроки разработки.

Кроме того, команда столкнулась с ограниченными ресурсами сервера и недостаточной эффективностью Python-кода, что сказывалось на скорости обработки изображений и времени отклика серверной части. В результате пришлось перераспределить нагрузку, оптимизировать код и внедрить решения для ускорения обработки, но даже это не избавило полностью от проблем с производительностью.

Всё это, конечно, повлияло на общую скорость работы и требования к ресурсам проекта, что потребовало пересмотра архитектурных решений и пересмотра подходов к обучению модели. Тем не менее, несмотря на эти трудности, команда смогла справиться с основными задачами и вывести проект на рабочую стадию, с возможностью дальнейшего совершенствования и масштабирования.

5. РЕСУРСЫ И МАТЕРИАЛЫ ПРОЕКТА

5.1. Используемые инструменты и технологии

5.1.1 Используемые языки

- Python
- C++
- JavaScript

5.2. Используемые библиотеки, модули и другие компоненты:

- React
- TensorFlow
- Celery
- RabbitMQ
- boto3
- PostgreSQL
- MinIO
- FastAPI
- Nginx

5.2. Ссылки на внешние ресурсы:

TensorFlow репозиторий: https://github.com/tensorflow/tensorflow

TensorFlow C++ API справочник: https://www.tensorflow.org/api_docs/cc

Python документация: https://docs.python.org/3/

Trello руководство: https://trello.com/guide

PostgreSQL документация: https://www.postgresql.org/docs/

Gunicorn документация: https://gunicorn.org/#docs

Nginx документация: https://nginx.org/ru/docs/

Документация FastAPI: https://fastapi.tiangolo.com

Репозиторий проекта: https://github.com/Supertos/MAI-ML-Colorizer/

Репозиторий хранения модели: https://huggingface.co/DeZtrOiD/ML_colorizer/tree/main

Набор данных MIRFLICKR: https://press.liacs.nl/mirflickr/mirdownload.html

5.3. Данные и математические модели

Набор данных получен из следующих источников:

- ImageNet.
- Flickr Historical.
- ImagesCustom Dataset.
- MIRFLICKR

Математическая модель используемой нейронной сети представляет собой нейронную сеть прямого распространения, разделённую на несколько основных блоков, связанных между собой.

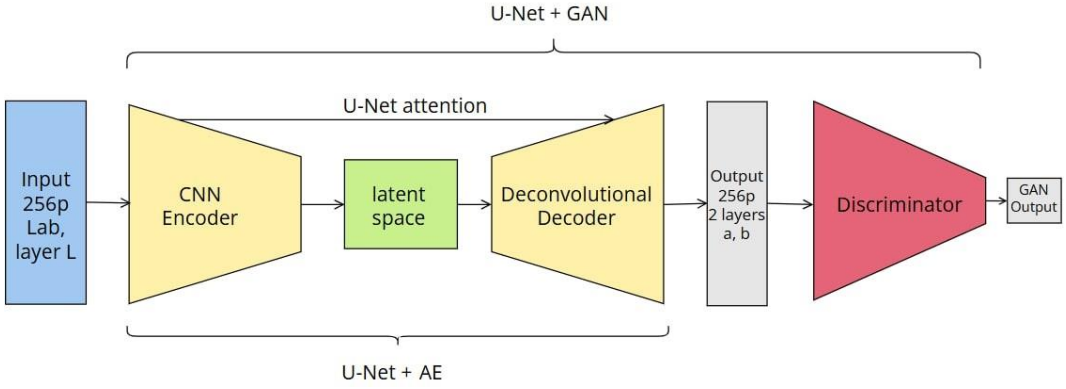

Рисунок 15 - математическая модель используемой нейронной сети.

На вход математической модели подается изображения формата LAB, позволяющий где только два канала, a и b отвечают за цветовые данные. Канал l отвечает за яркость. Использование данного формата позволяет проще использовать выходные данные для наложения на исходное изображение.

Модель можно разделить на блоки дискриминатора и генератора.

Дискриминатор оценивает созданные генератором изображения с целью определить вероятность того, что изображение пришло из генератора, а не является реальным. Генератор стремится создавать изображения, обманывающие дискриминатор.

Генератор представляет из себя U-Net AE-CNN, состоящий из блоков свёрточного кодировщика, декодера составленного с помощью транспонированной свертки, и блока скрытого представления, в котором могут содержаться дополнительные полносвязные слои. Блоки кодировщика и декодера связанны между собой, в том числе и послойно (skip connection, attention). Слои представляют собой набор нейронов с определенными правилами, зависящими блока.

Для нейронной сети используется несколько функций активации. Для выходного слоя дискриминатора используется сигмоида:

$$x_j^i = f(u) = \frac{1}{1+e^{-u}},$$

где:

$$u = \sum_{k=1}^{n} \omega_k^{i-1} x_k^{i-1} + \omega_0^{i-1} x_0^{i-1}$$

ω_k^{i-1} вес ребра узла x_k^{i-1}

i – соответствующей слой нейронной сети

n - количество узлов на соответствующем уровне.

Для выходного слоя генератора используется гиперболический тангенс:

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

Для остальных слоёв используется функция линейной ректификации с утечкой (Leaky ReLU):

$$f(x) = \begin{cases} x, x > 0 \\ \alpha x, x \leq 0 \end{cases}$$

В проекте используется модернизация алгоритма оптимизации методом обратного распространения ошибки Adam:

Даны следующие величины:

$$\varepsilon - величина шага \ (по \ умолчанию \ равен \ 10^{-3})$$

$$\rho_1, \rho_2 \in [0,1) - к. экспоненциального затухания оценок моментов \ (0.9 \ и \ 0.999)$$

$$\delta - константа численной устойчивости \ (10^{-8})$$

$$\theta - начальные \ значения параметров$$

4. Инициализировать переменные первого и второго моментов s, r=0
5. Инициализировать шаг обучения t=0
6. Пока критерий остановки не выполнен:

6.1 Выбрать из обучающего набора пакет m примеров $\{x^{(1)}, \ldots, x^{(m)}\}$ и соответствующие им метки $y^{(i)}$

6.2 Вычислить градиент $g \leftarrow \left(\frac{1}{m}\right) \nabla_0 \sum_i L\left(f\left(x^{(i)}; \theta\right), y^{(i)}\right); t \leftarrow t+1$

6.3 Обновить смещенную оценку первого момента: $s \leftarrow \rho_1 s + (1-\rho_1) g$

6.4 Обновить смещенную оценку первого момента: $r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$

6.5 Скорректировать смещение первого момента $s' \leftarrow \frac{s}{1-\rho^t{}_1}$

6.6 Скорректировать смещение первого момента $r' \leftarrow \frac{r}{1-\rho_2^t}$

6.7 Обновить каждый параметры $\theta \leftarrow -\varepsilon \frac{s'}{\sqrt{r' + \delta}}$

ПРИЛОЖЕНИЕ Б

Код Быстрого Отклика на репозиторий GitHub