

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

Кафедра

вычислительной техники

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

**ВЕБ-ПРИЛОЖЕНИЕ УЛУЧШЕНИЯ
КАЧЕСТВА ВИДЕО**

Студент

Верховский Михаил Владимирович
(подпись, дата) (ФИО полностью)

Руководитель

Зинкин С.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр

Бычков А.С.
(подпись, дата) (фамилия, инициалы)

Рецензент

зав. кафе. Программирования Пенз. ГТУ - АИИ 02.06.26 Мартышкин А.И.
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.2026 № 14)

Заведующий кафедрой

Митрохин М.А.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от

09 июня 2026г. № 15)

Секретарь ГЭК

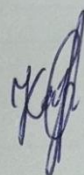
Гурин Е.И.
(подпись) (фамилия, инициалы)

Пенза 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»



«Утверждаю»
Заведующий кафедрой
М.А. Митрохин

« 20 » 09 2024 г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

1. Студент Верховский Михаил Владимирович гр. 24ВВИм1 факультета ВТ

2. Тема работы «Веб-приложение улучшения качества видео»

Тема утверждена приказом ПГУ № 920/о от "20" сентября 2024 г.

3. Руководитель магистерской диссертации д.т.н., профессор Зинкин С.А.

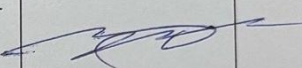
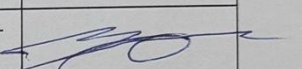
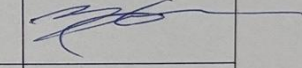
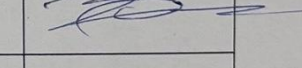
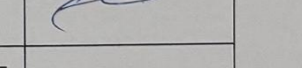
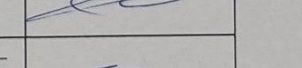
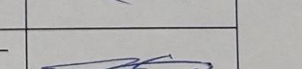
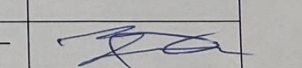
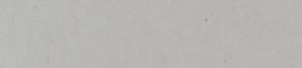
4. Задание на работу (назначение разработки, исходные данные и т.п.)

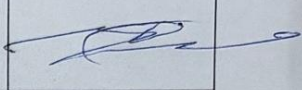
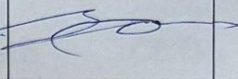
Разработать и реализовать архитектуру веб-приложения для улучшения качества видео с использованием конвейера обработки видеоконтента при помощи генеративно-состязательных сетей (GAN). Приложение должно обеспечивать загрузку видеофайлов и изображений пользователями, асинхронную обработку с помощью конвейера улучшения качества, а также отображение результата. В состав системы необходимо включить подсистему диспетчеризации задач для управления очередью обработки в условиях переменной пользовательской нагрузки. Для обмена сообщениями между компонентами использовать менеджер сообщений. В качестве исходных данных выступают видеофайлы различных форматов, разрешений, длительности и битрейта, загружаемые пользователями через веб-интерфейс.

5. Перечень подлежащих разработке вопросов

Провести исследование влияния характеристик видеоконтента (разрешение, длительность, частота кадров, кодек, битрейт) на вычислительную сложность GAN-обработки.
Сформировать датасет для дообучения модели и вывести формулу оценки приоритета задач.
Выбрать архитектуру веб-приложения, включающую клиентскую часть, серверную часть, менеджер сообщений и диспетчер задач. Спроектировать и реализовать фронтенд-часть, бэкенд-часть. Разработать диспетчер задач. Настроить взаимодействие компонентов.
Провести экспериментальную оценку эффективности реализованной системы по метрикам времени ожидания, распределения задач по приоритетам и глубины очереди

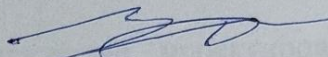
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Исследование влияния характеристик видеоконтента на вычислительную сложность GAN-обработки	23	20.09.2024 – 22.10.2024	
2	Формирование датасета для дообучения модели	45	23.10.2024 – 25.12.2024	
3	Вывод формулы оценки приоритета задач	18	26.12.2024 – 22.01.2025	
4	Анализ алгоритмов диспетчеризации и выбор оптимального подхода	18	23.01.2025 – 17.02.2025	
5	Проектирование общей архитектуры веб-приложения	20	18.02.2025 – 18.03.2025	
6	Проектирование подсистемы диспетчеризации задач	18	19.03.2025 – 11.04.2025	
7	Реализация клиентской части веб-приложения	40	14.04.2025 – 10.06.2025	
8	Реализация серверной части веб-приложения	42	11.06.2025 – 11.08.2025	
9	Реализация диспетчера задач	36	12.08.2025 – 02.10.2025	
10	Настройка взаимодействия компонентов	16	03.10.2025 – 24.10.2025	
11	Проведение экспериментальных	32	27.10.2025 –	

	исследований и оценка эффективности системы		09.12.2025	
12	Оформление магистерской диссертации	100	10.12.2025 – 21.05.2026	

Дата выдачи задания " 20 " 09 2024 г.

Руководитель магистерской диссертации



Зинкин С.А.

(подпись, дата)

Задание к исполнению принял студент



Верховский М.В.

(подпись, дата)

Работу к защите допустить
Декан факультета



05.06.2026

Фионова Л.Р.

(подпись, дата)

РЕФЕРАТ

Магистерская работа содержит 147 листов, 56 рисунков, 21 таблицу, 16 источников, 2 приложения, 6 формул.

ВЕБ-ПРИЛОЖЕНИЕ, ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНЫЕ СЕТИ, ДИСПЕТЧЕРИЗАЦИЯ ЗАДАЧ, АСИНХРОННАЯ ОБРАБОТКА, ФОРМИРОВАНИЕ ДАТАСЕТА

Целью работы является исследование, проектирование, реализация и оценка эффективности полученной архитектуры веб-приложения, предназначенного для улучшения качества видео с использованием ранее разработанного конвейера обработки медиафайлов.

Объектами исследования являются процессы передачи и диспетчеризации поступающей информации в условиях переменной пользовательской нагрузки.

Предметами исследования являются архитектурные реализации веб-приложения с менеджером сообщений и диспетчером задач, алгоритмы диспетчеризации и их зависимость от данных, полученных в процессе дообучения нейронной сети.

В рамках данной работы было проведено исследование существующих архитектурных подходов к построению высоконагруженных и отказоустойчивых веб-приложений для обработки видеофайлов, выполнен анализ внутренних компонентов системы и алгоритмов диспетчеризации задач применительно к результатам, полученным во время дообучения нейронной сети.

Реализовано веб-приложение для асинхронной обработки данных с использованием менеджера сообщений и диспетчера задач, использующего данные результатов дообучения нейронной сети для базового алгоритма диспетчеризации. Результаты могут быть применены в веб-сервисах улучшения качества медиафайлов и других нагруженных задачах машинного обучения.

					ПГУ1.090401.03.001 ПЗ				
Изм.	Лист	№ докум.	Подпись	Дата	Веб-приложение улучшения качества видео Пояснительная записка	Лит.	Лист	Листов	
Разраб.		Верховский М.В.		23.05.26					
Провер.		Зинкин С.А.		23.05.26			2	114	
Н. Контр		Бычков А.С.		23.05.26		ФВТ, каф. ВТ гр. 24ВВИм1			
Утверд.		Митрохин М.А.		23.05.26					

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	5
ВВЕДЕНИЕ	6
1 Исследование влияния характеристик видеоконтента на вычислительную сложность GAN-обработки	8
1.1 Характеристики видеофайла как фактор нагрузки при обработке	8
1.2 Исследование зависимости периода выполнения задач SR от характеристик входного видеоконтента	13
1.2.1 Методика исследования зависимости метрик для стандартной модели	14
1.2.2 Результаты исследования метрик для стандартной модели	15
1.2.3 Формирование датасета для дообучения модели	21
1.2.4 Методика исследования зависимости метрик для дообученной модели	30
1.3 Вывод формулы оценки приоритета	36
1.4 Формирование требований к подсистеме диспетчеризации	41
2 Проектирование веб-приложения с подсистемой диспетчеризации задач ..	43
2.1 Общая архитектура системы	43
2.2 Проектирование клиентской и серверной частей веб-приложения	44
2.2.1 Фронтенд-часть	44
2.2.2 Бэкенд-часть	48
2.3 Проектирование системы обмена сообщениями	50
2.4 Анализ алгоритмов диспетчеризации	54
3 Реализация веб-приложения и подсистемы диспетчеризации	61
3.1 Реализация клиентской и серверной частей веб-приложения	61
3.1.1 Фронтенд-часть	61
3.1.2 Бэкенд-часть	67
3.2 Реализация взаимодействия с менеджером сообщений	74
3.2.1 Топики и схема сообщений	75
3.2.2 Повторная доставка сообщений и обработка отказов	79
3.3 Реализация диспетчера задач	80

3.3.1 Точка входа диспетчера.....	81
3.3.2 Планировщик задач	85
3.3.3 Механизм запланированного устаревания	87
3.3.4 Аудитор метрик.....	89
3.4 Взаимодействие компонентов	92
4 Оценка эффективности реализованной системы	95
4.1 Исследование метрик диспетчера задач	95
4.1.1 Методика проведения эксперимента	96
4.1.2 Оценка времени ожидания задач.....	98
4.1.3 Оценка распределения задач по уровням приоритета	100
4.1.4 Оценка глубины очереди.....	101
4.2 Результаты работы реализованного веб-приложения.....	102
ЗАКЛЮЧЕНИЕ	111
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	113
ПРИЛОЖЕНИЕ А – Листинг программы	115
ПРИЛОЖЕНИЕ Б – Презентация.....	138

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В данной работе применяются следующие сокращения и обозначения

БД – база данных

ОС – операционная система

СУБД – система управления базами данных

ЯП – язык программирования

AI (Artificial Intelligence) – искусственная интеллектуальная система

API (Application Programming Interface) – программный интерфейс приложения

DNN (Deep Neural Networks) – глубокие нейронные сети

FI (Frame Interpolation) – задачи интерполяции кадров

FIFO (First In, First Out) – дисциплина обслуживания очереди «первым пришёл – первым ушёл»

FPS (Frames Per Second) – количество сменяемых кадров за единицу времени

FSD (Feature Sliced Design) – методология архитектурной организации фронтенд-проектов

GAN (Generative Adversarial Network) – генеративно-сопоставительная сеть

GPU (Graphics Processing Unit) – графический процессор

HTTP (HyperText Transfer Protocol) – протокол передачи гипертекста

JSON (JavaScript Object Notation) – текстовый формат для обмена данными, основанный на синтаксисе JavaScript

Kafka – распределённая платформа для потоковой передачи сообщений (менеджер сообщений)

PostgreSQL – объектно-реляционная система управления базами данных

REST (Representational State Transfer) – архитектурный стиль взаимодействия компонентов распределённого приложения

SR (Super Resolution) – задачи суперразрешения

WebSocket – протокол полнодуплексной связи поверх TCP-соединения

Zod – библиотека для валидации схем данных в TypeScript

ВВЕДЕНИЕ

Веб-приложение всегда стоит на важном месте между пользователем и доступными ему вычислительными ресурсами, к которым он обращается. Веб-приложение выступает как посредник, давая тем самым доступ к разным программным сервисам. То, насколько хорошо происходит взаимодействие, прямо влияет на то, с какой скоростью и эффективностью можно улучшить качество видео. Всё это зависит от способности веб-инфраструктуры справляться с большой и неравномерной нагрузкой от пользователей. Нужду в эффективной работе таких систем усиливает рост объёмов видеоконтента и требований к его качеству в сети.

Сейчас множество стриминговых платформ, соцсетей, видеохостингов и корпоративных систем видеоконференций стараются улучшить опыт взаимодействия пользователей с нейронными сетями [1]. Однако это не всегда даёт положительный результат. При интеграции нейронных сетей в сервис разработчики игнорируют требования к качеству обслуживания. Внутренняя диспетчеризация задач как раз может помочь в этом вопросе. Этот подход расставляет приоритеты в соответствии со скоростью обработки контента, что приводит к улучшению использования веб-приложения.

Однако существует довольно мало открытых решений для установки приоритетов обработки видеоконтента, особенно при работе с генеративно-состязательными сетями.

Использование нейронных сетей в тракте обработки приводит к неочевидным зависимостям для диспетчера задач, что грозит некорректным определением приоритета пользовательской задачи и последующим негативным опытом использования веб-приложения. При работе с видеоконтентом можно использовать его метаданные для определения приоритета, но это должно осуществляться в зависимости от используемой модели нейронной сети. Данный подход позволил бы улучшить скорость обработки видеоконтента для улучшения его качества, а также снизил бы нагрузку на смежные элементы системы.

Целью магистерской диссертации является исследование, проектирование, реализация и оценка эффективности полученной архитектуры веб-приложения, предназначенного для улучшения качества видео с использованием ранее разработанного конвейера обработки медиафайлов.

Для достижения поставленной цели необходимо решить следующие задачи исследования:

- провести анализ влияния характеристик видеоконтента на вычислительную сложность GAN-обработки и вывести формулу для расчёта приоритета задач,

- спроектировать архитектуру веб-приложения, интегрирующую клиентскую и серверную части, менеджер сообщений и подсистему диспетчеризации задач,

- реализовать диспетчер задач на основе выбранного алгоритма диспетчеризации,

- оценить эффективность разработанной системы путём экспериментального исследования метрик времени ожидания, распределения задач по приоритетам и глубины очереди.

Результаты проведенного исследования могут быть применены в веб-сервисах улучшения качества медиафайлов и других нагруженных задачах машинного обучения.

1 Исследование влияния характеристик видеоконтента на вычислительную сложность GAN-обработки

Видеофайл является одним из самых ресурсоемких типов мультимедийных данных в современных системах [2]. Сложная пространственно-временная структура заметно отличает видеоконтент от текстового представления или изображения. Для выполнения задач повышения качества подобного контента может потребоваться значительное количество выполненных операций, что может привести к гонке за аппаратные ресурсы и переполнению очереди задач, находящейся после бэкенда. Это создаёт весьма серьёзную угрозу как пользовательскому опыту, так и внутренней отказоустойчивости веб-приложения. По этой причине перед началом проектирования веб-приложения улучшения качества важно определить факторы, вызывающие пиковые нагрузки на вычислительные ресурсы, поскольку они напрямую влияют на выбор внутренней архитектуры балансировки и алгоритмов организации.

1.1 Характеристики видеофайла как фактор нагрузки при обработке

После выбора пользователем видеоматериала для обработки задача с параметрами обработки формируется на уровне бэкенда и поступает в очередь задач, которую периодически анализирует обработчик. Вместе с тем, между модулем формирования задач и очередью отсутствует механизм для диспетчеризации поступающих запросов. При наличии нескольких обработчиков или нескольких версий реализаций бэкенда может произойти конфликт между фиксируемыми запросами на обработку. С другой стороны, может возникнуть проблема на уровне обработчика, когда будут обрабатываться только последние запросы.

Чтобы решить данную проблему, необходимо спроектировать механизм оценки и распределения поступающих задач, опираясь только на доступные к началу цикла обработки характеристики. Такими характеристиками являются параметры видеофайла, которые можно распределить по предварительным градациям влияния на скорость

выполнения задачи улучшения качества. В таблице 1 подробно описаны доступные признаки видеоряда с указанием предварительной оценки влияния.

Таблица 1 – Доступные признаки видеофайла

Признак	Предварительные значения	Оценка влияния
Разрешение	HD, Full HD, 2K, 4K	Критическое, степенная зависимость от площади кадра
Длительность	Короткие (до 60 секунд) Средние (от 1 до 10 минут) Длинные (более 10 минут)	Высокое, линейная зависимость
FPS	24, 30, 45, 60 кадров в секунду	Высокое, линейная зависимость
Кодек сжатия	H.264, HEVC (H.265), AV1	Высокое, зависит от частных случаев
Битрейт	От 20 до 100 Мбит/с	При значениях 60 Мбит/с и больше оказывает сильного влияние. В остальных случаях – среднее.
Глубина цвета	8 бит (SDR) или 10 бит (HDR)	Среднее, не оказывает сильного влияния

При составлении предварительной оценки влияния параметров видео на скорость выполнения задачи повышения качества стоит начать с разрешения. Данный признак является первичным фактором нагрузки, определяющим размерность входного и выходного тензоров для обработки в нейронной сети. При фиксированном масштабирующем коэффициенте ($\times 2$ или $\times 4$) площадь кадра растёт квадратично относительно линейных размеров. Переход от Full HD (≈ 2 Мп) к 4K (≈ 8.3 Мп) увеличивает количество пикселей в 4 раза. Для DNN любого типа это означает:

- расширение карт признаков во всех промежуточных слоях,
- рост вычислительной сложности, пропорциональный произведению высоты и ширины изображения,
- кратное увеличение потребления видеопамати под хранение активаций.

Высокое исходное разрешение требует большей рецептивной области признаков для захвата текстурной информации, что вынуждает углублять архитектуру или использовать механизмы внимания с квадратичной по числу патчей сложностью. На практике с большими весами не загружают кадр 4K целиком, а разбивают его на фреймы, что негативно сказывается на конечном результате работы.

Частота кадров в задачах улучшения качества влияет на накопленный за единицу времени объём вычислений. В классической покадровой постановке каждый кадр обрабатывается независимо, и переход с 30 на 60 FPS означает ровно двукратный рост числа необходимых обработок в секунду. Чаще всего в процессе выполнения задачи улучшения качества для видео между кадрами дополнительно строится оптический поток или вычисляются выравнивающие деформации, что при высокой частоте кадров дает меньшее межкадровое смещение – с одной стороны облегчая выравнивание, с другой – увеличивая чувствительность к шуму. Количество кадров в секунду задает определенный бюджет ресурсов на один кадр и выдержать его в режиме реального времени без специализированного аппаратного ускорения и алгоритмов обработки достаточно сложно.

Кодек сжатия определяет характер и интенсивность артефактов на этапе декодирования перед запуском модели для решения задачи улучшения качества. Различие между кодеками проявляется в следующих аспектах:

- H.264/AVC имеет умеренные блочные артефакты и шум вблизи резких границ, нагрузка на декодер модели для SR должна быть низкой. Пример работы кодека приведен на рисунке 1,

- H.265/HEVC обладает более тонкой структурой блочности, но глубже проникают эффекты размытости и ступенчатости на градиентах. Аппаратное декодирование доступно на большинстве GPU, программное требует значительных ресурсов. Пример работы кодека приведен на рисунке 2,

- AV1 прогрессивный кодек с сильным подавлением высоких частот на низких битрейтах. SR-модели, обученные на упорядоченных датасетах, часто

трактуют сглаженные AV1 области как истинно низкочастотную текстуру и не восстанавливают детали. В таких случаях требуется дообучение модели на смеси кодеков или добавление легкого декодера артефактов во входной конвейер. Пример работы кодера приведен на рисунке 3.



Рисунок 1 – Изображение с кодеком H.264/AVC



Рисунок 2 – Изображение с кодеком HEVC/H.265



Рисунок 3 – Изображение с кодеком AV1

Битрейт является мерой степени сжатия и, косвенно, количества вносимых кодеком искажений. При значениях 60 Мбит/с и выше уровень артефактов для разрешений вплоть до 4K минимален: входные кадры должны сохранять исходную зернистость и тонкую текстуру. При средних битрейтах (10–60 Мбит/с) возможно появление локальных искажений. Это может проявляться в виде возникновения блоков на участках активного движения, потери текстурной детализации на воде и листве. На экстремально низких битрейтах (менее 5 Мбит/с) классический SR без

специализированного обучения склонен усиливать артефакты, делая их более резкими и заметными.

Глубина цвета кардинально не меняет операционный ландшафт, но сдвигает периферийные компоненты. Основные модели SR традиционно работают с 8-битными целочисленными RGB-представлениями, на которых отлажены квантованные (INT8/FP16) модели. Появление 10-битного HDR-материала приводит к следующему:

- объём входного тензора увеличивается на 25% в битовом выражении, но большинство сетей внутренне повышают разрядность до FP32, что нивелирует выигрыш от сжатой разрядности, однако не создаёт взрывного роста параметров,

- ключевое влияние SR на HDR заключено в сохранении градиентов яркости: 10-битный градиент должен оставаться плавным, иначе возникает постеризация на ярких участках.

В сравнении с разрешением или количеством кадров, глубина цвета вносит аддитивную, а не мультипликативную прибавку к вычислительному бюджету и поэтому классифицируется как фактор среднего влияния.

Длительность видео напрямую определяет суммарное время обработки задачи. Для ролика длительностью 1 минута при 30 FPS нужно выполнить 1800 операций SR, для 10-минутного – 18 000. Полный SR многочасового контента может занимать десятки часов на одном GPU, что неприемлемо в сервис-ориентированной архитектуре. Из-за линейного характера сложности длительность, в отличие от разрешения, не ведёт к быстрому росту нагрузки, но является ключевым фактором при расчёте пропускной способности очередей задач, времени ожидания пользователя и стоимости серверного времени.

Также не стоит забывать о ряде внешних факторов, порождаемых архитектурой веб-приложения и моделью взаимодействия с пользователем. Эти факторы не связаны с содержанием кадра, но столь же существенно влияют на стабильность и время ответа, формируя неоднородную и трудно прогнозируемую картину нагрузки.

К одному из таких факторов можно отнести неоднородность вычислительной нагрузки. В многопользовательской среде на одном физическом сервере одновременно обрабатываются задачи с кардинально разными требованиями. Без механизмов изоляции и диспетчеризации возникает частичная или полная блокировка доступных ресурсов. Если один пользователь отправил видео 2К с запросом на двукратное увеличение, модель может занять всю видеопамять и вычислительные блоки GPU. Остальные запросы, даже простые (например, удаление шума на 480p), помещаются в очередь и испытывают недетерминированную задержку. С точки зрения пользователя лёгкой задачи время ответа становится непредсказуемым: вместо ожидаемых 10 секунд он ждёт 5 минут. Отсутствие справедливого разделения ресурсов приводит к ситуации, когда одно критичное задание не может быть выполнено быстрее за счёт вытеснения фоновых.

1.2 Исследование зависимости периода выполнения задач SR от характеристик входного видеоконтента

Чтобы подтвердить выдвинутые ранее гипотезы, связанные с расстановкой приоритетов параметров видеофайла, необходимо провести эмпирическое исследование. Для установки факта зависимости скорости улучшения качества от характеристик поступающих видео будет проведено два эксперимента:

- сбор и анализ признаков выполнения задач SR на стандартной модели,
- сбор и анализ признаков выполнения задач SR при дообучении модели.

Необходимость проведения данных экспериментов обусловлена не только выявлением зависимости периода выполнения задачи SR от характеристик входного видеоконтента. Требуется также проверить факт увеличения точности предсказания при диспетчеризации, если учитывать данные, поступившие во время дообучения нейронной сети. Это позволило

бы сделать вывод о необходимости сбора данных для диспетчера до дополнительной итерации обучения.

1.2.1 Методика исследования зависимости метрик для стандартной модели

В качестве обработчика видеоконтента была выбрана GAN-модель Real-ESRGAN [3]. Выбор данной модели обусловлен результатами смежных исследований нейросетевых моделей для задач улучшения качества. Для данного решения уже существует несколько открытых обученных моделей, что предоставляет возможность для дообучения имеющейся модели. Из пула предложенных моделей был выбран вариант RealESRGAN x4+ из-за наличия подробных описаний процесса дообучения, понятных инструментов анализа и частичной схожести типов используемого датасета.

Для замера и составления карты признаков обработки была написана специализированная утилита на языке Python. Данный язык был выбран из-за низкого порога входа, доступности библиотек для отображения текущего результата сбора признаков и простоты интеграции с существующей реализацией выбранной нейронной сети. Алгоритм работы данной утилиты заключается в следующем:

- выбор директории с образцами для тестирования. Утилите необходима заранее сформированная директория с образцами видео,
- сбор статистики. После выбора расположения в одноименной директории создается XML-файл с статистикой работы модели. Происходит запуск цикла обработки образцов со сбором статистики,
- вывод зависимостей метрик. Встроенные библиотеки языка Python позволяют проанализировать собранные метрики и сформировать результат о параметрах, влияющих на ключевой признак.

Перед проведением первой эмпирической части исследования была сформирована директория с видеоконтентом для снятия метрик. В выборку вошли 700 различных отрывков анимационных фильмов, видеозаписей

культурных достопримечательностей Пензы, а также часть тестовых видео для выбранной модели (рисунок 4).

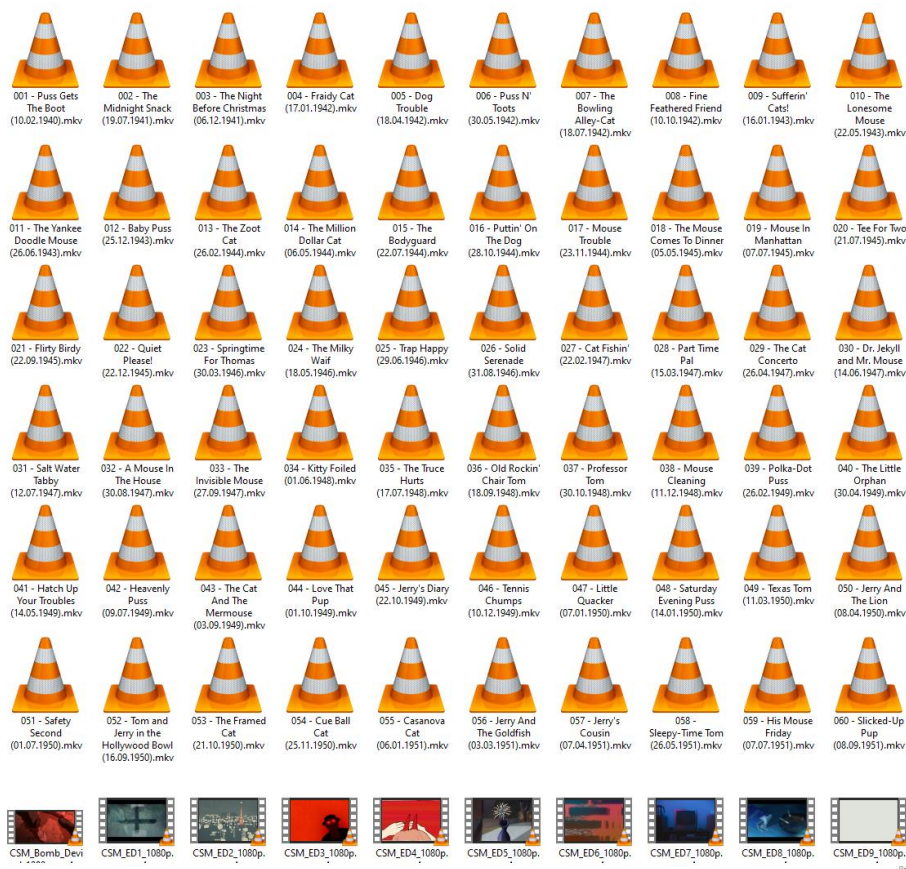


Рисунок 4 – Директория с тестовыми видеоматериалами

В тестовую выборку вошли видео с различными параметрами. Все параметры укладываются в параметры выборки, указанные в таблице 1. Данная выборка распределена неравномерно, в соотношении 6 к 1, из-за более высокой распространённости улучшенного мультипликационного контента, нежели реальных видео повышенного качества с низкокачественным оригиналом.

1.2.2 Результаты исследования метрик для стандартной модели

Чтобы выявить зависимость времени обработки видео от характеристик входного контента проведены эксперименты с различными значениями каждого из параметров (длительности, разрешения, частоты кадров, битрейта и типа кодека).

На рисунке 5 представлена зависимость времени обработки видео от его длительности при следующих параметрах:

– стандартное разрешение: 640 на 480 пикселей,

- извлекаемое разрешение: 1920 на 1080 пикселей,
- частота кадров: 24 кадра в секунду,
- длительность: от 1 до 60 минут.

Зависимость времени обработки видео от длительности видео

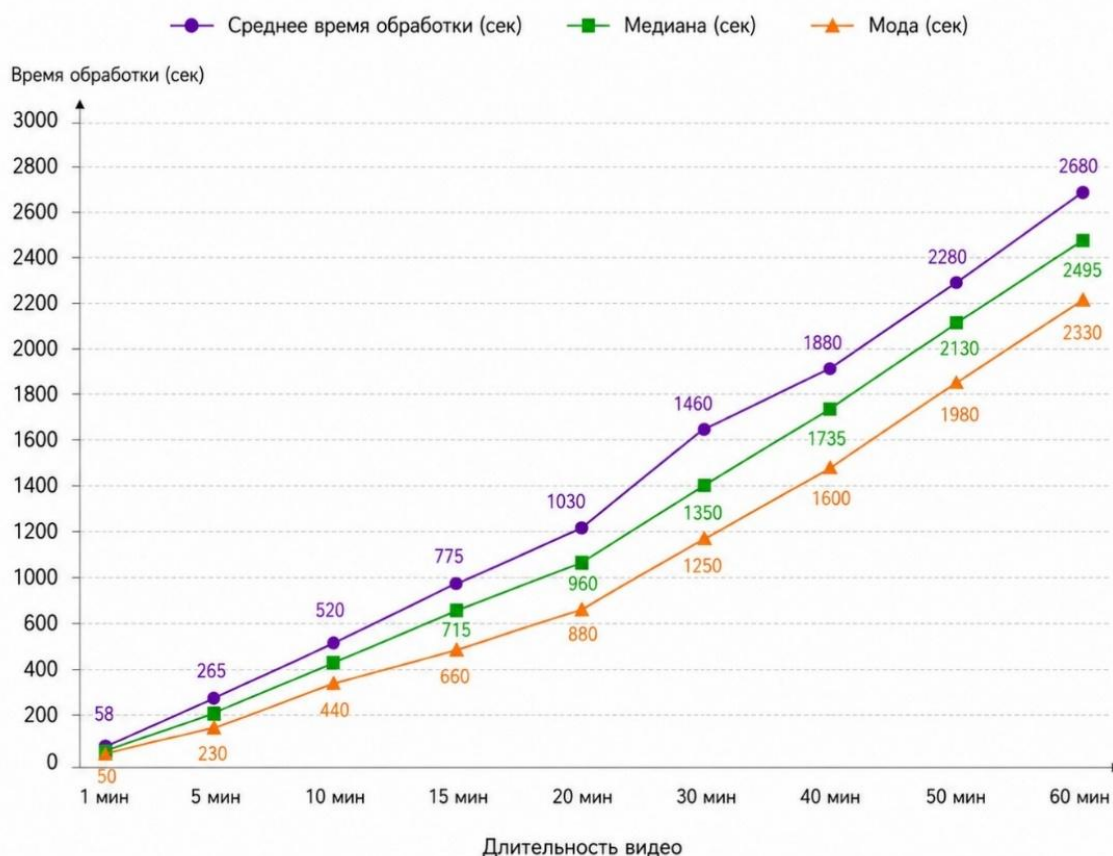


Рисунок 5 – Зависимость времени обработки от длительности видео

В таблице 2 показаны статистические показатели, на основе которых был построен описанный график.

Таблица 2 – Статистические показатели времени обработки видео в зависимости от длительности

Длительность видео	1 мин	5 мин	10 мин	15 мин	20 мин	30 мин	40 мин	50 мин	60 мин
Среднее время (сек)	58	265	520	775	1030	1460	1880	2280	2680
Медиана (сек)	54	243	470	715	960	1350	1735	2130	2495
Мода (сек)	50	230	440	660	880	1250	1600	1980	2330

Таким образом, с увеличением длительности видео время обработки линейно возрастает, что подтверждается собранной статистикой. Это подтверждает вывод, сделанный в таблице 1.

На рисунке 6 видно зависимость времени обработки видео от разрешения.

Параметры следующие:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: 24 кадра в сек.,
- длительность: 10 мин.

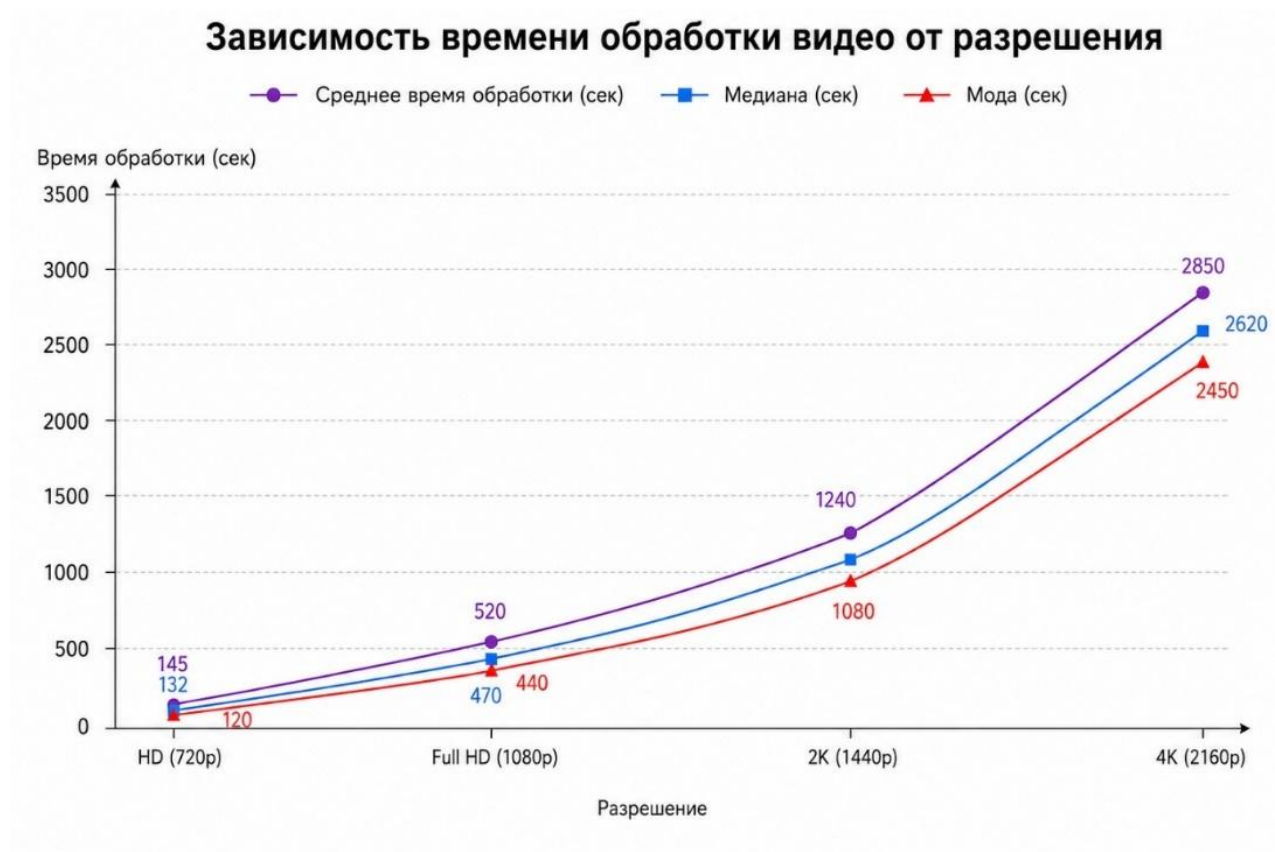


Рисунок 6 – Зависимость времени обработки от разрешения

В таблице 3 исследована зависимость времени обработки от разрешения входных данных на модели Real-ESRGAN x4+.

Данная статистика приводит к выводу о том, что чем больше разрешение, тем всё больше увеличивается время обработки. Зависимость почти квадратичная.

Таблица 3 – Собранные в исследовании показатели времени обработки видео в зависимости от разрешения исходного видео

Разрешение	HD (720p)	Full HD (1080p)	2K (1440p)	4K (2160p)
Среднее время (сек)	145	520	1240	2850
Медиана (сек)	132	470	1150	2620
Мода (сек)	120	440	1080	2450

Здесь видно подтверждение высказанной ранее гипотезы, что параметр разрешения сильно влияет на время обработки видеофайла.

На рисунке 7 показана зависимость времени обработки видео от частоты кадров. Статистические данные проведенного замера представлены в таблице 4. Сбор метрик проводился при следующих параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: от 24 до 60 кадров в секунду,
- длительность: 10 минут.

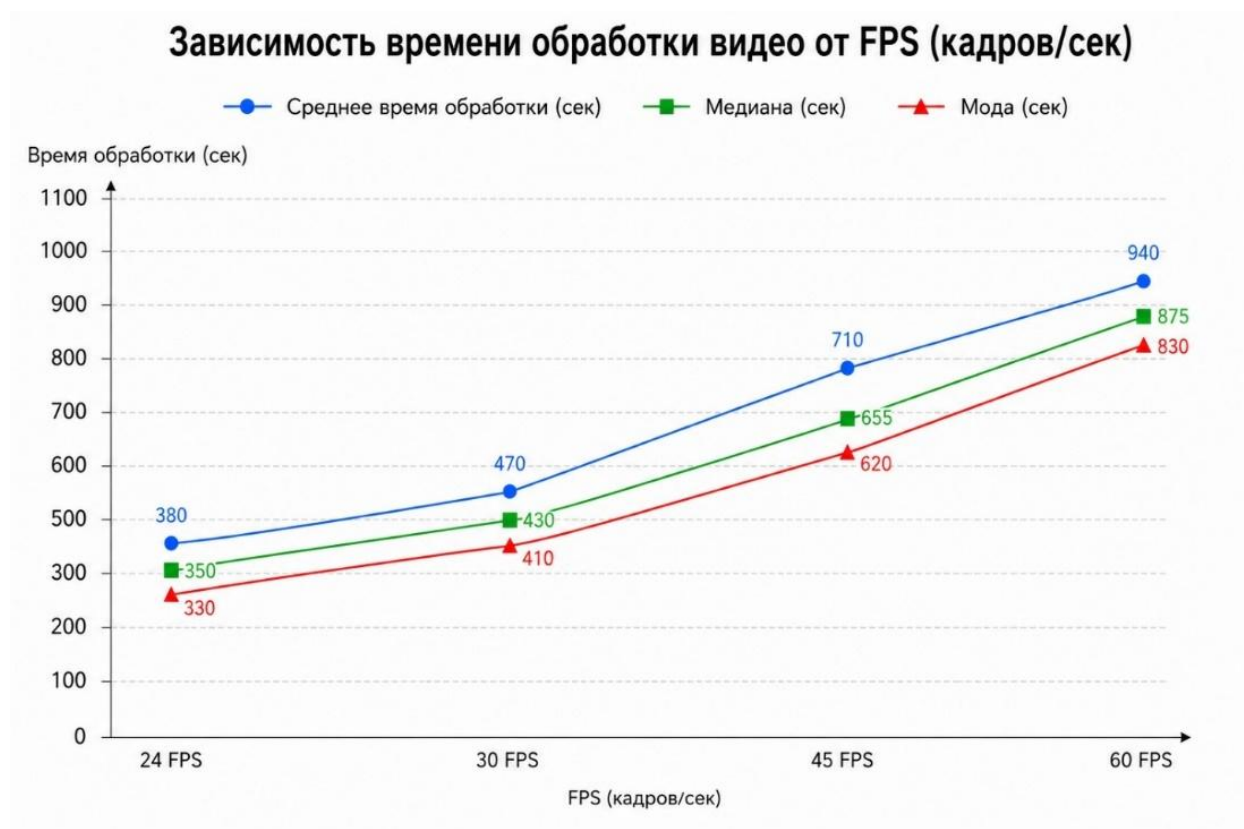


Рисунок 7 – Зависимость времени обработки от количества FPS

Таблица 4 – Статистические показатели времени обработки видео в зависимости от количества кадров в секунду

FPS (кадров/сек)	24	30	45	60
Среднее время (сек)	380	470	710	940
Медиана (сек)	350	430	655	875
Мода (сек)	330	410	620	830

Повышение частоты кадров ведет к значительному увеличению времени обработки. Это подтверждает гипотезу, объявленную в таблице 1.

На рисунке 8 изображена зависимость времени обработки видео от битрейта при заданных параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: 30 кадров в секунду,
- длительность: 10 минут,
- битрейт: от 20 до 100 Мбит/с.

Зависимость времени обработки видео от битрейта (усл. ед.)

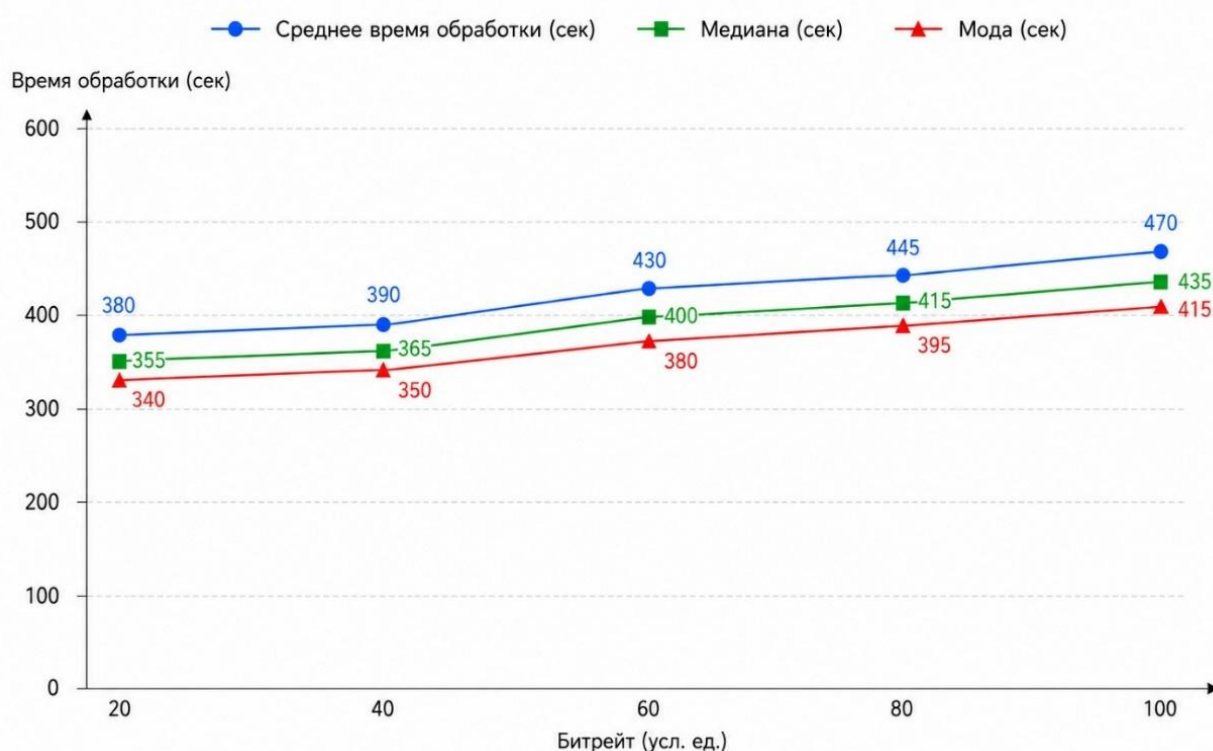


Рисунок 8 – Зависимость времени обработки от битрейта

В таблице 5 приведены статистические показатели, использованные при построении графика.

Таблица 5 – Статистические показатели времени обработки видео в зависимости от битрейта

Битрейт (усл. ед.)	20	40	60	80	100
Среднее время (сек)	380	390	430	445	470
Медиана (сек)	355	365	400	415	435
Мода (сек)	340	350	380	395	415

По анализу полученных метрик битрейт не оказывает значительного влияния на время обработки.

На рисунке 9 показана зависимость времени обработки видео от типа кодека при следующих параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: 30 кадров в секунду,
- длительность: 10 минут,
- битрейт: 50 Мбит/с.

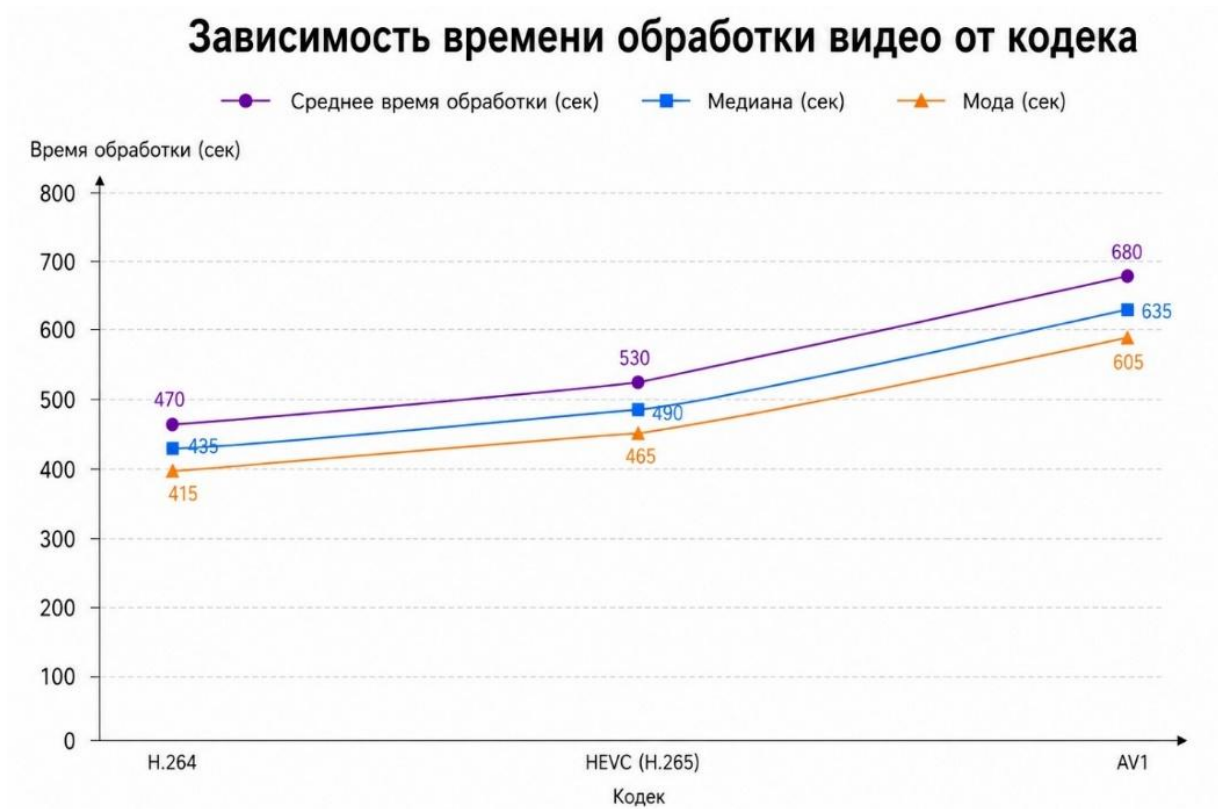


Рисунок 9 – Зависимость времени обработки от кодека

В таблице 6 представлены статистические данные зависимости длительности обработки видео от используемого кодека.

Таблица 6 – Статистические показатели времени обработки видео в зависимости от кодека

Кодек	H.264	HEVC (H.265)	AV1
Среднее время (сек)	470	530	680
Медиана (сек)	435	490	635
Мода (сек)	415	465	605

Проведённые эксперименты показали, что наибольшее влияние на время обработки оказывают длительность, разрешение и частота кадров. Битрейт вносит незначительный вклад, как и кодек. Данный вывод подтверждает корректность гипотез, сделанных в таблице 1. Однако следует проверить сохранение данных зависимостей в процессе дообучения исходной модели.

1.2.3 Формирование датасета для дообучения модели

Формирование репрезентативного обучающего датасета является критическим этапом, в значительной степени определяющим обобщающую способность модели улучшения качества видео. Специфика решаемой задачи диктует необходимость дообучения на выборке, покрывающей различные классы визуальных сцен, текстур и типов анимации, поскольку характер артефактов и искомых высокочастотных деталей для каждого класса существенно различается. Необходимость формирования датасета для дообучения при замерах диспетчеризации заключается в уникальности видеоконтента как материала для дообучения, так и его же уникальности для исследования зависимостей характеристик. Уникальный датасет позволит подобрать такие данные, которые не были использованы в предыдущих итерациях тренировки и практического тестирования нейронной сети.

Применительно к нейронной сети датасет должен соответствовать следующим требованиям:

- репрезентативность. Должны учитываться разнообразные типы контента, с которыми будет вестись работа при реальном использовании.

Будет три типа контента. Сервис должен будет обрабатывать как мультипликацию (советскую и зарубежную), так и документальные видео и фотографии. Если упустить какую-нибудь категорию контента, то модель будет плохо с ним работать,

- сбалансированность. Количество примеров основных категориях (зарубежная и советская мультипликация) не должно уступать другим. Иначе модель сместится в сторону этого типа контента и будет хуже обрабатывать другие,

- качество целевых изображений. Исходные кадры должны иметь высокое разрешение и качество. Использование некачественных эталонов приведёт к тому, что модель будет учиться восстанавливать несуществующие детали,

- объём. Для дообучения модели супер-разрешения было использовано 716 000 уникальных кадров. Разбиение выборки было распределено в процентном соотношении: обучение (70%), валидация (15%) и тестирование (15%),

Для формирования датасета были рассмотрены четыре основных метода сбора данных (таблица 7), каждый из которых имеет свои преимущества и ограничения применительно к задаче улучшения качества мультимедийного контента.

Если использовать только открытые датасеты, то можно столкнуться с тем, что они содержат преимущественно фотографии природы, городские пейзажи и портреты. Платформы, такие как Kaggle и Hugging Face, предоставляют доступ к множеству разнообразных данных, но в них отсутствуют кадры советской и зарубежной мультипликации.

Краудсорсинг в моём случае тоже не лучший выбор, т.к. такой способ сбора датасета не даёт контроля качества, а времени на проверку за каждым привлечённым исполнителем просто нет.

Веб-скрейпинг может проводиться с помощью разных инструментов, например, BeautifulSoup или Scrapy. Но не весь веб-скрейпинг законен.

Важно учитывать лицензирование данных, ведь не все из них доступны для использования.

А вот ручной сбор данных для датасета полностью подконтролен и помогает добиться максимальной релевантности. Этот метод позволяет гарантировать, что каждый кадр соответствует требованиям качества, отобран из легальных источников и принадлежит к нужной категории контента.

В таблице 7 продемонстрирован анализ рассмотренных выше методов сбора данных.

Таблица 7 – Анализ методов сбора данных

Метод	Преимущества	Недостатки	Применимость к задаче
Open-source датасеты	Минимальные временные затраты, данные уже очищены	Отсутствие специфических категорий	Низкая – не покрывает требуемые категории
Веб-скрейпинг	Быстрый сбор больших объёмов	Юридические ограничения, требует ручной очистки	Низкая – высокий риск нерелевантных данных
Краудсорсинг	Масштабируемость, широкий охват	Невозможность контроля качества	Низкая – не подходит для профессионального отбора
Ручной сбор	Максимальное качество, полный контроль	Высокие временные и трудовые затраты	Высокая – обеспечивает эталонное качество

Был выбран ручной метод сбора данных для того, чтобы полностью контролировать содержание датасета на наличие контента из трёх категорий (зарубежная и советская мультипликация, а также немного документальных кадров). Медиафайлы брались из открытых источников по открытой лицензии. Многие мультфильмы зарубежной мультипликации являются общественным достоянием и доступны для открытого использования.

Для формирования обучающих пар кадров с низким и высоким разрешением использовалась утилита FFmpeg [4], которая предназначена для обработки медиа любого формата различными методами. В моём случае она поможет в извлечении кадров по заданным правилам. Из каждого видео извлекается первый кадр каждой секунды. Такой подход выбран потому, что

после апскейлинга и интерполяции общая длительность видео не меняется. Первый кадр каждой секунды в оригинальном и улучшенном видео будет соответствовать одной и той же временной точке, что обеспечивает соответствие пар этих кадров.

Для отбора кадров использовалась следующая команда:

```
ffmpeg -i input.mp4 -vf "select='not(mod(t,1))'" -vsync vfr  
frame_%02d.png.
```

Здесь:

- i input.mp4 – это входной видеофайл,
- vf "select='not(mod(t,1))'" – отбор кадров в начале каждой секунды,
- vsync vfr – сохранение только выбранных кадров,
- frame_%02d.png – выходные файлы с нумерацией.

Данный процесс проводится для двух видео (оригинального и улучшенного).

Для формирования датасета с соответствующими кадрами важно было, чтобы оригинальные и улучшенные кадры строго соответствовали одному моменту времени. Плюс при таком методе отбора кадров не остаётся дублированных изображений.

Итоговый объём датасета показан в таблице 8.

Таблица 8 – Объем полученного датасета

Категория	Количество кадров	Доля
Советская мультипликация	322200	45%
Зарубежная мультипликация	322200	45%
Фотографии в высоком качестве	71600	10%
Итого	716000	100%

На основе сформулированных требований и выбранного подхода (ручной сбор данных) обучающий датасет был скомпонован из трёх категорий исходного материала, отобранных по принципу максимального покрытия характерных паттернов, встречающихся в пользовательском контенте.

В качестве первого источника взяты улучшенные кадры мультфильмов студии «Союзмультфильм» (работы семидесятых и девяностых годов). Она характеризуется следующими показателями:

- плавными контурными линиями,
- акварельными и гуашевыми градиентами и бумажной текстурой,
- отсутствием резких границ между объектами переднего и заднего планов. Это создаёт сложный случай для алгоритмов выделения границ при масштабировании.

Выбор именно улучшенных кадров обусловлен тем, что они предоставляют высококачественный эталон в целевом разрешении (рисунки 10 – 13).



Рисунок 10 – Изначальный кадр из мультфильма «Котенок по имени Гав»



Рисунок 11 – Улучшенный кадр из мультфильма «Котенок по имени Гав»

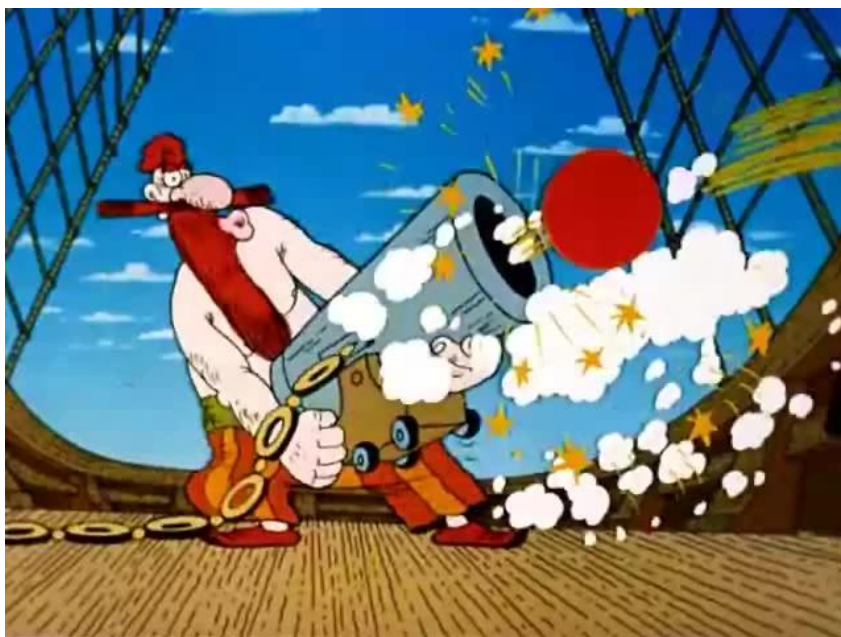


Рисунок 12 – Изначальный кадр из мультфильма «Остров сокровищ»

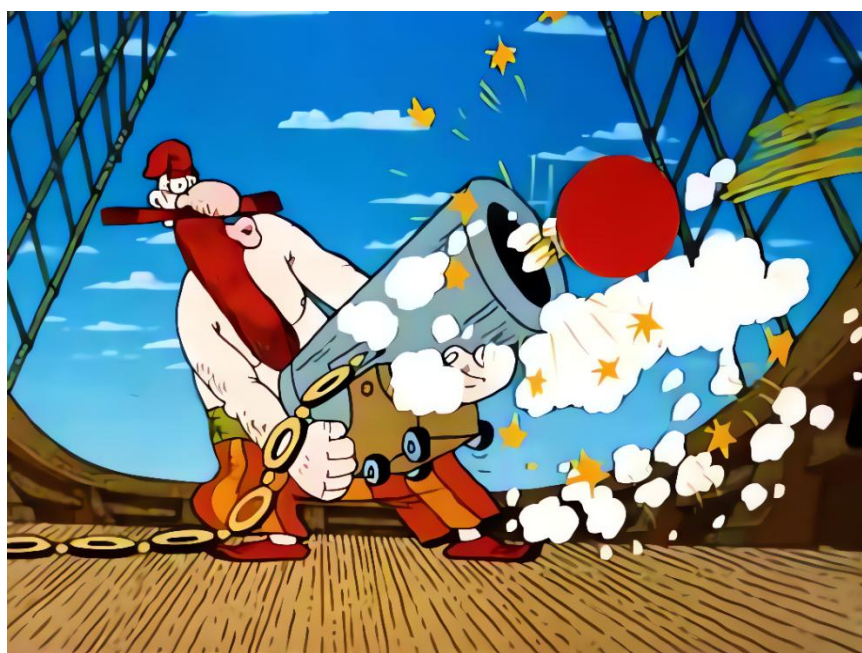


Рисунок 13 – Улучшенный кадр из мультфильма «Остров сокровищ»

В качестве второго источника выбраны документальные кадры. Взяты фрагменты доступных всем архивов, картинки природы, а также городских ландшафтов. Эта категория дополняет датасет следующими признаками:

- сложными сценарии освещения, где есть резкие тени и блики,
- мелкими высокочастотными деталями.

Добавление именно этой категории в датасет дало предотвращение эффекта «анимационного смещения». Потому что, если обучить модель

только на мультфильмах, она будет сглаживать текстуры, не давая выглядеть им реалистично (рисунки 14 – 17).



Рисунок 14 – Первоначальное фото пензенского областного драматического театра



Рисунок 15 – Улучшенное фото пензенского областного драматического театра

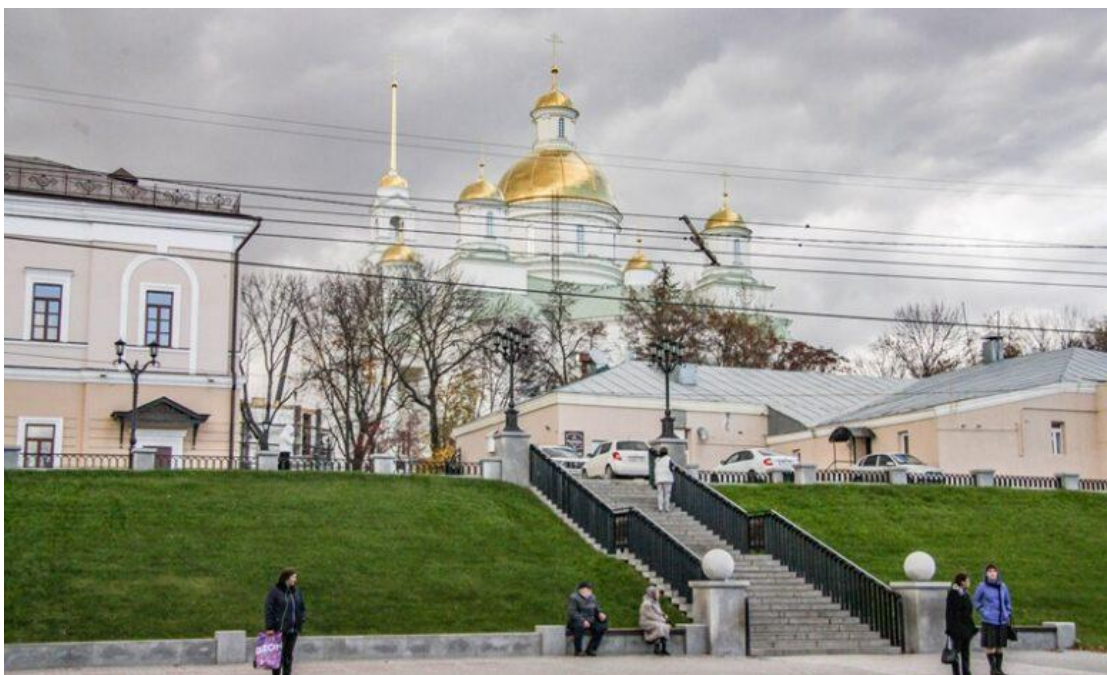


Рисунок 16 – Исходная фотография собора



Рисунок 17 – Улучшенное фото собора

Третьим источником в составлении датасета были выбраны кадры из серии фильмов «Том и Джерри» (рисунки 18, 19), которые вышли в пятидесятых годах двадцатого века. Этот источник выбран по причине того, что он представляет контраст по отношению к «Союзмультфильму», так как у него другой визуальный стиль, а именно:

- более резкие, графичные контуры,
- неменяющаяся толщина обводки,

- плоский насыщенный цвет объектов без градиента,
- высокая скорость передвижения объектов в кадре, выраженное сглаживание. Это увеличивает требования к дополнительной согласованности кадров при обучении.

Совместное использование двух стилистически различных источников анимации (советской живописной и американской графической) направлено на обучение модели инвариантным к стилю рисования признакам апскейла, что снижает вероятность переобучения под конкретную художественную технику.



Рисунок 18 – Исходная версия кадра из мультфильма «Том и Джерри»



Рисунок 19 – Улучшенная версия кадра из мультфильма «Том и Джерри»

Для каждой из трёх категорий применялась единая методика подготовки данных. Исходные кадры выступали в роли целевого изображения высокого разрешения. Соответствующее ему изображение низкого разрешения было извлечено из исходного видеофайла при помощи утилиты FFmpeg. Такой подход позволяет получить строго попиксельно

выровненные пары изображений, необходимые для обучения с учителем, и одновременно обеспечить аугментацию, повышающую адаптацию модели к различным сценариям деградации исходного пользовательского файла.

1.2.4 Методика исследования зависимости метрик для дообученной модели

После формирования датасета и дообучения на нём модели Real-ESRGAN x4+, в процессе которого также были сняты метрики для анализа совпадения зависимостей с экспериментом в пункте 1.2.2, можно приступить к получению метрик зависимостей на дообученной модели.

На рисунке 20 представлена зависимость времени обработки видео от его длительности для дообученной GAN-модели при фиксированных параметрах:

- стандартное разрешение: 640 на 480 пикселей,
- извлекаемое разрешение: 1920 на 1080 пикселей,
- частота кадров: 24 кадра в секунду,
- длительность: от 1 до 60 минут.

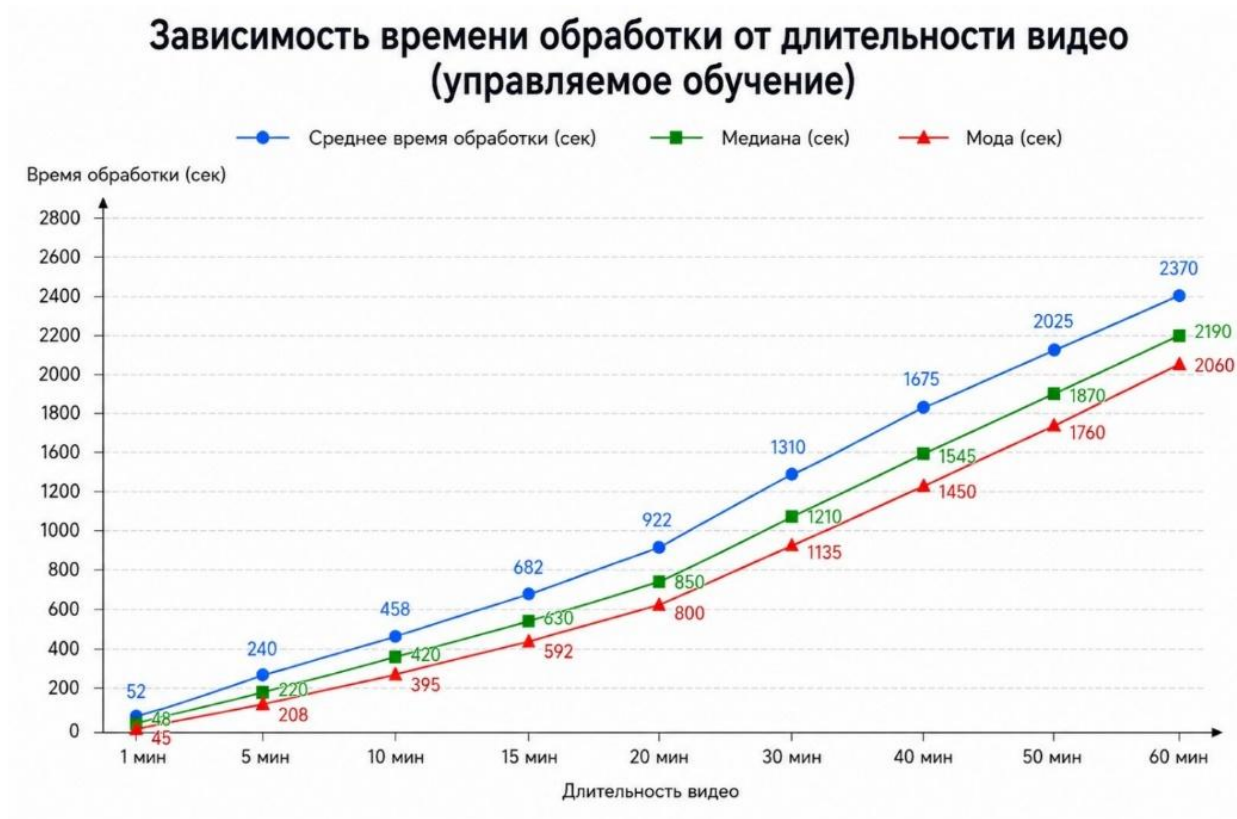


Рисунок 20 – Зависимость времени обработки от длительности видео

В таблице 9 показаны статистические показатели, на основе которых был построен этот график.

Таблица 9 – Статистические показатели времени обработки видео в зависимости от длительности

Длительность видео	1 мин	5 мин	10 мин	15 мин	20 мин	30 мин	40 мин	50 мин	60 мин
Среднее время (сек)	52	240	458	682	922	1310	1675	2025	2370
Медиана (сек)	48	220	420	630	850	1210	1545	1870	2190
Мода (сек)	45	208	395	592	800	1135	1450	1760	2060
Улучшение (среднее, %)	-10%	-9%	-12%	-12%	-10%	-10%	-11%	-11%	-12%

Таким образом, дообученная модель демонстрирует ускорение обработки на 9–12% для всех диапазонов длительности и соответствующее смещение на графике зависимостей.

На рисунке 21 представлена зависимость времени обработки видео от разрешения для дообученной модели при фиксированных параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: 30 кадров в секунду,
- длительность: 10 минут.

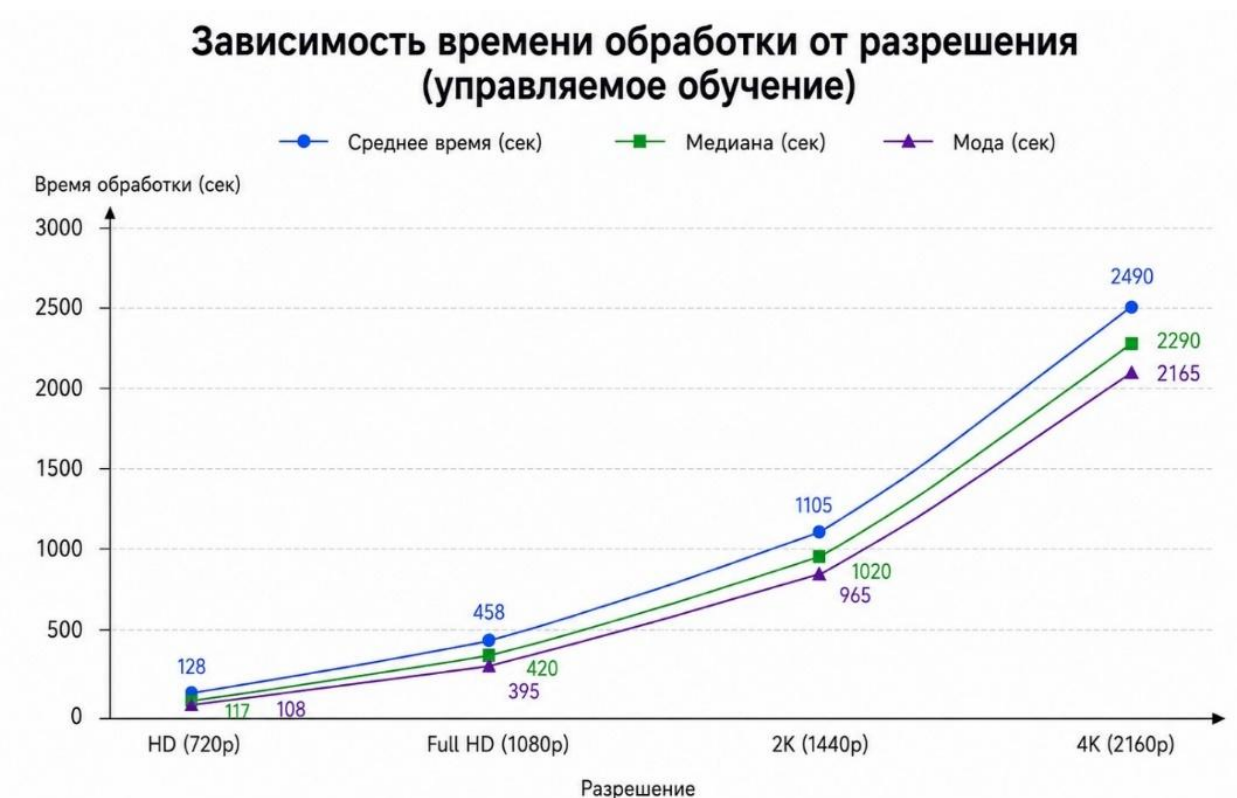


Рисунок 21 – Зависимость времени обработки от разрешения

В таблице 10 приведены статистические показатели, полученные при исследовании дообученной модели.

Таблица 10 – Статистические показатели времени обработки видео в зависимости от разрешения

Разрешение	HD (720p)	Full HD (1080p)	2K (1440p)	4K (2160p)
Среднее время (сек)	128	458	1105	2490
Медиана (сек)	117	420	1020	2290
Мода (сек)	108	395	965	2165
Улучшение (среднее, %)	-12%	-12%	-11%	-13%

Из полученных данных следует, что дообученная модель обеспечивает ускорение обработки на 11–13% для всех исследованных разрешений.

На рисунке 22 показана зависимость времени обработки видео от частоты кадров, а в таблице 11 отображены количественные параметры для его построения. Тестирование выполнялось при фиксированных параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,

- частота кадров: от 24 до 60 кадров в секунду,
- длительность: 10 минут.

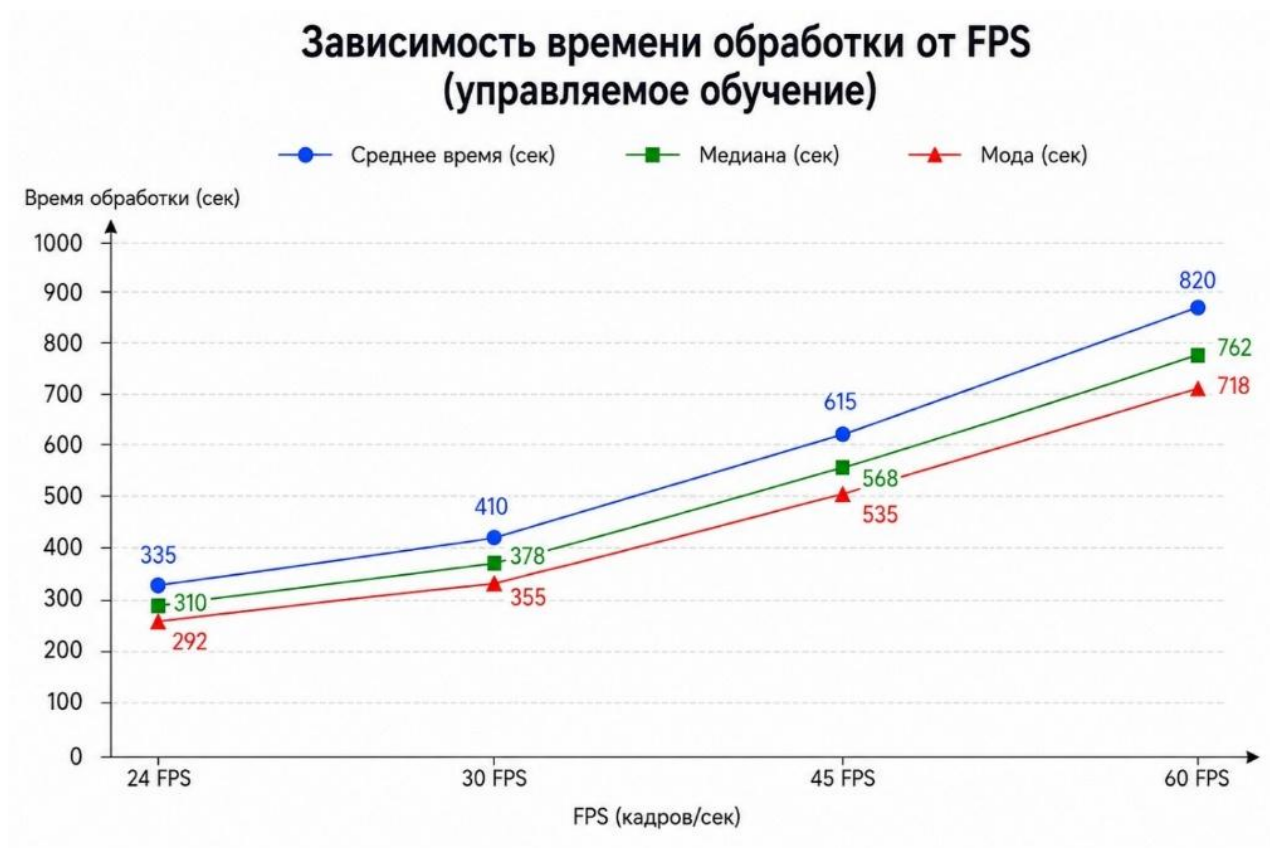


Рисунок 22 – Зависимость времени обработки от FPS

Таблица 11 – Статистические показатели времени обработки видео в зависимости от FPS

FPS (кадров/сек)	24	30	45	60
Среднее время (сек)	335	410	615	820
Медиана (сек)	310	378	568	762
Мода (сек)	292	355	535	718
Улучшение (среднее, %)	-12%	-13%	-13%	-13%

Анализ метрик показывает рост от 12 до 13% при обработке на дообученной модели, что приводит к соответствующим смещениям на графике.

На рисунке 23 изображена зависимость времени обработки видео от битрейта для дообученной модели при фиксированных параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,

- частота кадров: 30 кадров в секунду,
- длительность: 10 минут,
- битрейт: от 20 до 100 Мбит/с.

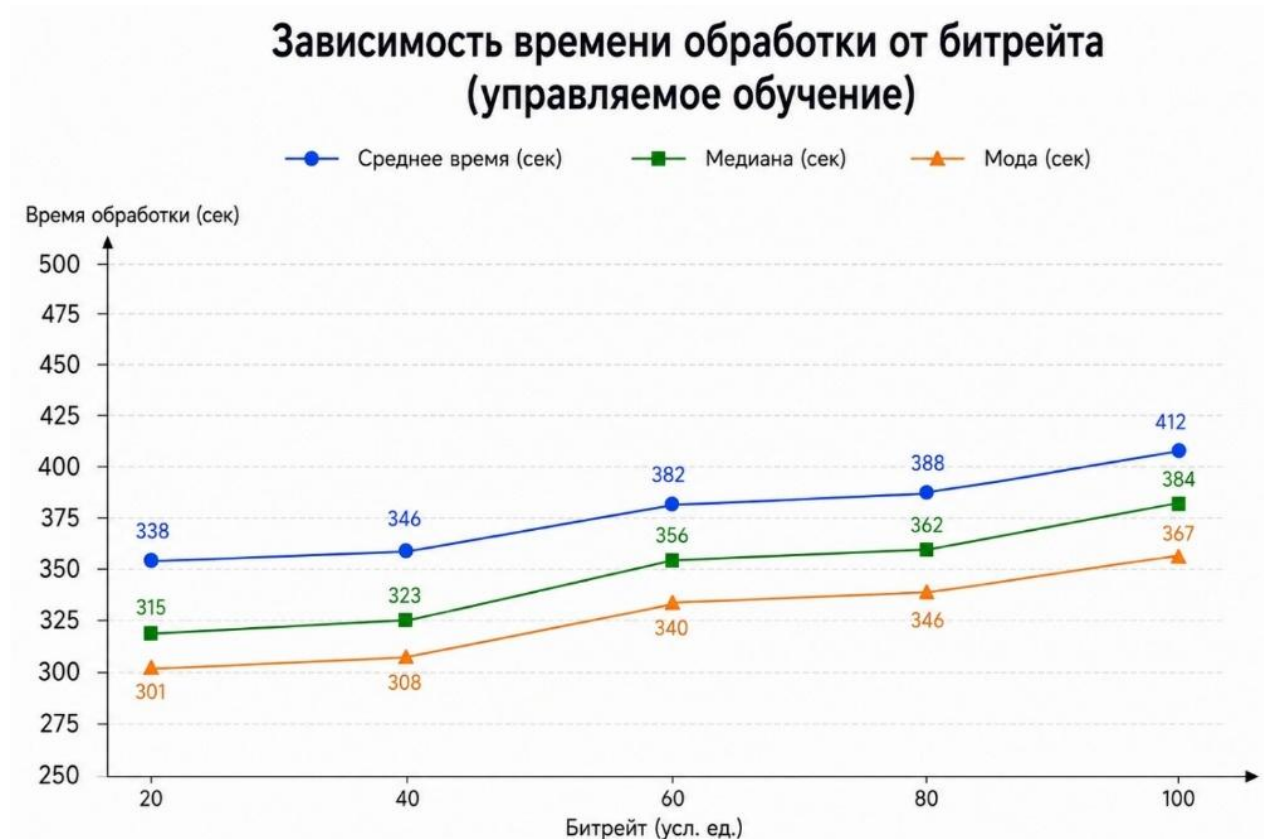


Рисунок 23 – Зависимость времени обработки от битрейта

В таблице 12 приведены статистические показатели, использованные при построении графика.

Таблица 12 – Статистические показатели времени обработки видео в зависимости от битрейта

Битрейт (усл. ед.)	20	40	60	80	100
Среднее время (сек)	338	346	382	388	412
Медиана (сек)	315	323	356	362	384
Мода (сек)	301	308	340	346	367
Улучшение (среднее, %)	-11%	-11%	-11%	-13%	-12%

Смещение битрейта на графике отображает ускорение обработки на 11–13%. Соотношение зависимостей характеристик осталось прежним, на уровне погрешности.

На рисунке 24 показана зависимость времени обработки видео от типа кодека для дообученной модели. Количественные данные продемонстрированы в таблице 13. Тестирование было проведено при следующих параметрах:

- стандартное разрешение: 1920 на 1080 пикселей,
- извлекаемое разрешение: 3840 на 2160 пикселей,
- частота кадров: 30 кадров в секунду,
- длительность: 10 минут,
- битрейт: 50 Мбит/с.

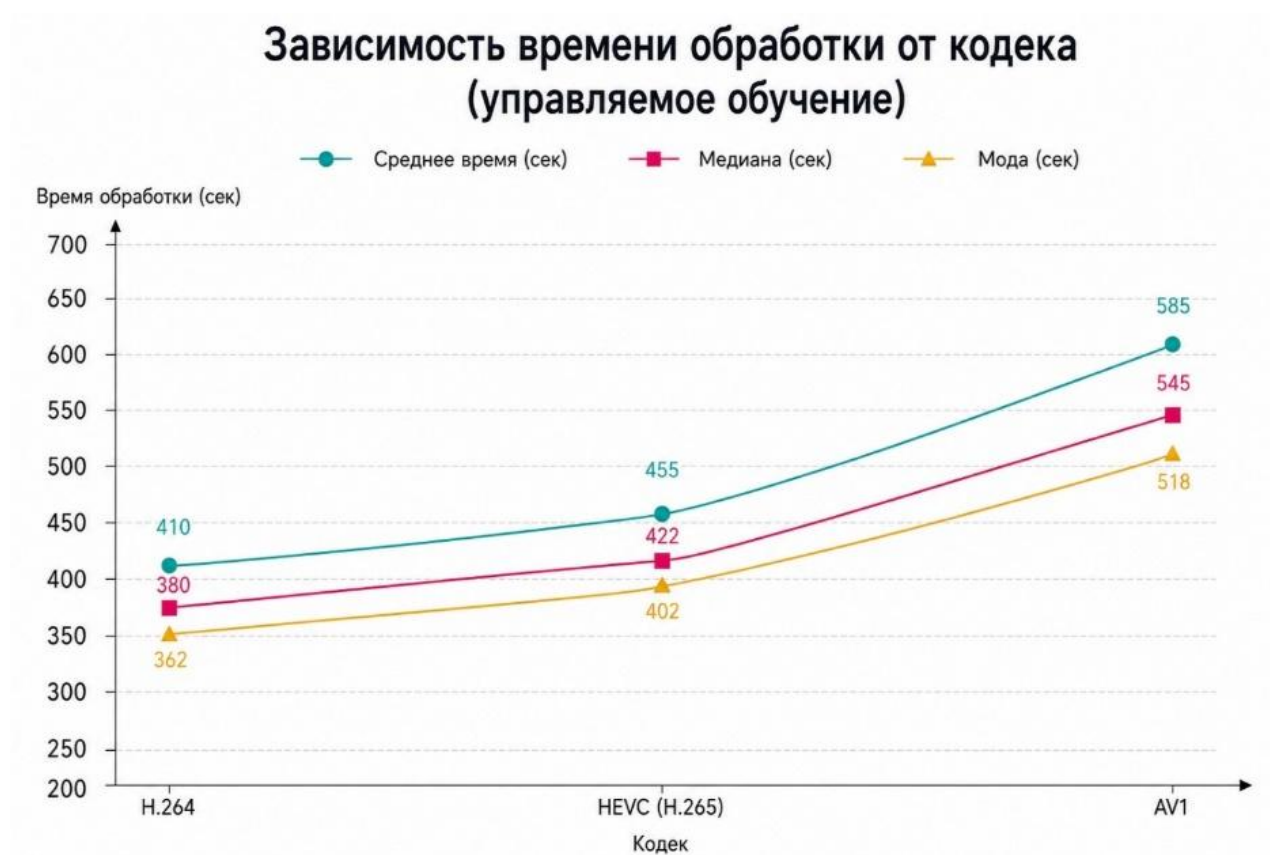


Рисунок 24 – Зависимость времени обработки от кодека

Таблица 13 – Статистические показатели времени обработки видео в зависимости от кодека

Кодек	H.264	HEVC (H.265)	AV1
Среднее время (сек)	410	455	585
Медиана (сек)	380	422	545
Мода (сек)	362	402	518
Улучшение (среднее, %)	-13%	-14%	-14%

Можно заключить, что модель с дообучением наиболее эффективна при работе со сложными кодеками (HEVC, AV1), где ускорение достигает 14%. Соотношение также немного увеличено в пользу этих кодеков.

В результате проведённых экспериментов можно сделать вывод, что GAN-модель, дообученная на целевом датасете (советская и зарубежная мультипликация), обеспечивает ускорение обработки видео на 10–14% по сравнению с базовой моделью. Наибольший эффект достигнут для высоких разрешений (4K), высоких FPS (60) и сложных кодеков (HEVC, AV1). Соответствующие соотношения немного увеличились, однако вектора направления движения приоритетов остались такими же. Это говорит о том, что характеристики обрабатываемого видеоконтента не только влияют на длительность обработки, но и то, что это влияние может быть преобразовано в вид формулы. И данную формулу можно будет использовать для расчёта оценки приоритета в диспетчере задач.

1.3 Вывод формулы оценки приоритета

После подтверждения совпадения векторов зависимостей скорости обработки от характеристик при тестах на классической и дообученной модели можно перейти к выводу формулы для оценки приоритета задачи. Приоритет задачи напрямую зависит от скорости её обработки. Чем короче задача, тем выше у неё приоритет. Характер зависимости от каждого фактора был установлен ранее:

- длительность имеет линейную зависимость,
- количество пикселей (обобщенное разрешение) имеет линейную зависимость,
- частота кадров имеет линейную зависимость,
- битрейт имеет слабую линейную зависимость,
- кодек выступает в качестве слабой аддитивной поправки.

Таким образом, базовая зависимость времени обработки от характеристик видео принимает вид формулы 1:

$$T = k \times W \times H \times F \times D \times K_{\text{битрейт}}(B) \times K_{\text{кодек}}(C), \quad (1)$$

где T – прогнозируемое время обработки, с;

W, H – ширина и высота кадра в пикселях;

F – частота кадров, кадр/с;

D – длительность видео, с;

B – битрейт видео, Мбит/с;

C – тип кодека (H.264, HEVC, AV1);

k – базовый константный коэффициент, с/пиксель;

$K_{\text{битрейт}}(B)$ – безразмерный коэффициент влияния битрейта;

$K_{\text{кодек}}(C)$ – безразмерный коэффициент влияния кодека.

В качестве эталонной точки для вычисления константного коэффициента необходимо принять следующие параметры:

- разрешение – 1920 на 1080,
- частота кадров – 30 кадров в секунду,
- длительность – 600 секунд,
- кодек – H.264 ($K_{\text{кодек}} = 1.0$),
- битрейт – 60 Мбит/с ($K_{\text{битрейт}}(60) = 1.0$).

Среднее время обработки для такого видеофайла составляет 458 секунд по таблице 10. Отсюда вытекает обратимая формула базового константного коэффициента:

$$k = \frac{T}{W \times H \times F \times D} = \frac{458}{1920 \times 1080 \times 30 \times 600} \approx 1.23 \times 10^{-8}.$$

В таблице 12 видна зависимость среднего времени обработки от битрейта при фиксированных прочих параметрах. Далее нормируем параметры относительно значения при 60 Мбит/с. Результат в таблице 14.

Таблица 14 – Коэффициент влияния битрейта $K_{\text{битрейт}}(B)$

Битрейт B , Мбит/с	Среднее время T , с	Кбитрейт = $T(B) / T(60)$
20	338	0.885
40	346	0.906
60	382	1.000
80	388	1.016
100	412	1.079

При аппроксимации линейной зависимостью по методу наименьших квадратов получается следующая запись:

$$K_{\text{битрейт}}(B) = 0.84 + 0.0024 * B, \quad (2)$$

где B – битрейт в Мбит/с ($20 \leq B \leq 100$).

Коэффициент детерминации $R^2 \approx 0.96$. Это подтверждает допустимость линейной аппроксимации для расчётов.

В таблице 13 представлена зависимость среднего времени обработки видеофайла от типа его кодека. В таблице 15 нормированы коэффициенты влияния кодеков относительно H.264.

Таблица 15 – Коэффициент влияния кодека $K_{\text{кодек}}(C)$

Кодек C	Среднее время T , с	$K_{\text{кодек}} = T(C) / T(\text{H.264})$
H.264	410	1.00
HEVC (H.265)	455	1.11
AV1	585	1.43

Коэффициенты константны для каждого типа кодека. Они не зависят от остальных параметров видео, что видно из параллельного смещения графиков, представленных на рисунках 9 и 25 соответственно.

Можно дополнить формулу полученными выше выражениями:

$$T = 1.23 * 10^{-8} * N_{\text{пикселей}} * N_{\text{фреймов}} * (0.84 + 0.0024 * B) * K_{\text{кодек}}(C), \quad (3)$$

где $N_{\text{пикселей}} = W \times H$ – количество пикселей в кадре;

$N_{\text{фреймов}} = F \times D$ – общее количество кадров в видео;

$K_{\text{кодек}}(C) \in \{1.00, 1.11, 1.43\}$ для H.264, HEVC, AV1 соответственно.

Для использования в подсистеме диспетчеризации абсолютное время T неудобно, поскольку алгоритму распределения очередей требуется нормированная величина, позволяющая сравнивать разнородные задачи в единой шкале. Введём коэффициент приоритета $P \in [0.1, 1.0]$, удовлетворяющий следующему семантическому правилу:

– $P = 1.0$ – самая лёгкая задача (минимальные значения всех параметров),

– $P = 0.1$ – самая тяжёлая задача (максимальные значения всех параметров в рамках ограничений сервиса),

– промежуточные значения распределены по логарифмической шкале для сглаживания экспоненциального разброса времён.

В качестве границ нормализации принимаем для T_{min} :

$$N_{пикселей} = 640 * 480 = 3.072 * 10^5 \text{ пикселей},$$

$$N_{фреймов} = 24 * 1 = 24 \text{ кадра},$$

$$B = 20 \text{ Мбит/с (минимальный битрейт)},$$

$$C = H.264 \text{ (самый быстрый кодек)}.$$

В качестве границ нормализации принимаем для T_{max} :

$$N_{пикселей} = 3840 * 2160 = 8.294 * 10^6 \text{ пикселей},$$

$$N_{фреймов} = 60 * 600 = 36\,000 \text{ кадров},$$

$$B = 100 \text{ Мбит/с (максимальный битрейт)},$$

$$C = AV1 \text{ (самый медленный кодек)}.$$

Вычисляем границы:

$$K_{буфрейм}(20) = 0.84 + 0.0024 * 20 = 0.84 + 0.048 = 0.888,$$

$$T_{min} = 1.23 * 10^{-8} * 3.072 * 10^5 * 24 * 0.888 * 1.00 \approx 0.081 \text{ с},$$

$$K_{буфрейм}(100) = 0.84 + 0.0024 * 100 = 0.84 + 0.24 = 1.08,$$

$$T_{max} = 1.23 * 10^{-8} * 8.294 * 10^6 * 36\,000 * 1.08 * 1.43 \approx 17\,020 \text{ с}.$$

Логарифмическая нормализация:

$$\lg T_{min} = \lg 0.081 \approx -1.092,$$

$$\lg T_{max} = \lg 17020 \approx 4.231,$$

$$\Delta = \lg T_{max} - \lg T_{min} = 4.231 - (-1.092) = 5.323.$$

Коэффициент приоритета показан в формуле 4:

$$P = 1 - \frac{0.9 * (\lg T - \lg T_{min})}{\Delta} = 1 - \frac{0.9 * (\lg T + 1.092)}{5.323}. \quad (4)$$

Логарифмируя выражение (3):

$$\lg T = \lg(1.23 * 10^{-8}) + \lg N_{пикселей} + \lg N_{фреймов} + \lg(0.84 + 0.0024 * B) + \lg K_{кодек},$$

$$\lg T = -7.91 + \lg N_{пикселей} + \lg N_{фреймов} + \lg(0.84 + 0.0024 * B) + \lg K_{кодек}.$$

Подстановка в (4):

$$P = 1 - \frac{0.9 * (-7.91 + \lg N_{пикселей} + \lg N_{фреймов} + \lg(0.84 + 0.0024 * B) + \lg K_{кодек} + 1.092)}{5.323},$$

$$P = 1 - \frac{0.9 \times (-6.818 + \lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B) + \lg K_{\text{кодек}})}{5.323},$$

$$P = 1 - 0.169 \times (-6.818 + \lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B) + \lg K_{\text{кодек}}).$$

Раскрывая скобки:

$$P = 1 + 1.152 - 0.169 \times (\lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B) + \lg K_{\text{кодек}}),$$

$$P = 2.152 - 0.169 \times (\lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B) + \lg K_{\text{кодек}}), \quad (5)$$

Итоговая формула коэффициента приоритета:

$$P = 2.152 - 0.169 \times (\lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B)) + \lg K_{\text{кодек}}, \quad (6)$$

где $N_{\text{пикселей}} = W \times H$ – количество пикселей в кадре;

$N_{\text{фреймов}} = F \times D$ – общее количество кадров;

B – битрейт, Мбит/с ($20 \leq B \leq 100$);

$K_{\text{кодек}} = 1.00$ для H.264, 1.11 для HEVC (H.265), 1.43 для AV1;

$\lg$ – десятичный логарифм;

константы 2.152 и 0.169 вычислены для дообученной модели Real-ESRGAN x4+VM.

Когда была получена окончательная формула коэффициента приоритета, была выполнена проверка её работы.

Первой проверкой является рассмотрение задачи с минимальным приоритетом:

$$N_{\text{пикселей}} = 3.072 * 105,$$

$$\lg N_{\text{пикселей}} = 5.487,$$

$$N_{\text{фреймов}} = 24,$$

$$\lg N_{\text{фреймов}} = 1.380,$$

$$0.84 + 0.0024 * 20 = 0.888,$$

$$\lg 0.888 = -0.052,$$

$$K_{\text{кодек}} = 1.00,$$

$$\lg 1.00 = 0,$$

$$P = 2.152 - 0.169 * (5.487 + 1.380 - 0.052 + 0),$$

$$P = 2.152 - 0.169 * 6.815 = 2.152 - 1.152 = 1.000 \approx 1.0.$$

Далее идёт проверка задачи, у которой максимальный приоритет:

$$N_{\text{пикселей}} = 8.294 * 106,$$

$$\lg N_{\text{пикселей}} = 6.919,$$

$$N_{\text{фреймов}} = 36\,000,$$

$$\lg N_{\text{фреймов}} = 4.556,$$

$$0.84 + 0.0024 * 100 = 1.08,$$

$$\lg 1.08 = 0.033,$$

$$K_{\text{кодек}} = 1.43,$$

$$\lg 1.43 = 0.155,$$

$$P = 2.152 - 0.169 * (6.919 + 4.556 + 0.033 + 0.155),$$

$$P = 2.152 - 0.169 * 11.663 = 2.152 - 1.971 = 0.181.$$

И, наконец, третья проверка уже для задачи со средним приоритетом:

$$N_{\text{пикселей}} = 2.074 * 106,$$

$$\lg N_{\text{пикселей}} = 6.317,$$

$$N_{\text{фреймов}} = 18\,000,$$

$$\lg N_{\text{фреймов}} = 4.255,$$

$$0.84 + 0.0024 * 50 = 0.96,$$

$$\lg 0.96 = -0.018,$$

$$K_{\text{кодек}} = 1.11,$$

$$\lg 1.11 = 0.045,$$

$$P = 2.152 - 0.169 * (6.317 + 4.255 - 0.018 + 0.045),$$

$$P = 2.152 - 0.169 * 10.599 = 2.152 - 1.791 = 0.361.$$

Успешное прохождение данных проверок подтверждает корректность разработанной формулы для расчета приоритета задачи улучшения качества видео. Необходимо использовать данную формулу в дальнейшей реализации, а также сформировать список требований к подсистеме диспетчеризации.

1.4 Формирование требований к подсистеме диспетчеризации

После формирования формулы зависимости периода выполнения задач SR от характеристик входного видеоконтента можно сформировать совокупность требований к подсистеме диспетчеризации. Данные требования необходимо обязательно применить на этапе проектирования и разработки рассматриваемой системы, иначе могут возникнуть проблемы с организацией внутренних процессов.

Текущие функциональные требования к реализуемой подсистеме диспетчеризации:

- подсистема диспетчеризации должна уметь извлекать из поступающего запроса следующие параметры: разрешение, длительность, частоту кадров, кодек сжатия и битрейт. Извлечение должно выполняться на этапе валидации запроса, до помещения задачи в очередь обработки,

- на основе извлечённых характеристик и выведенной формулы расчёта приоритета подсистема обязана вычислять оценку с использованием актуальных значений коэффициентов, хранящихся в профиле диспетчера задач. Допустимая погрешность прогноза не должна превышать $\pm 10\%$ от фактического времени обработки для 90% поступающих задач,

- каждой задаче должен быть присвоен числовой приоритет, вычисляемый как функция от прогнозируемого времени и, возможно, уровня обслуживания пользователя,

- подсистема обязана иметь интерфейс для взаимодействия с очередью задач. Данный интерфейс не должен ограничивать скорость оценки приоритета задач,

- для каждой обработанной задачи подсистема должна фиксировать: приоритет, отклонение приоритета, предсказанное время,

- при неполадках с очередью задач, подсистема должна иметь возможность сообщить о возникшей проблеме на бэкэнд для прекращения потока запросов или принудительного перезапуска очереди задач.

Сформулированная совокупность требований образует конкретное техническое задание на реализацию архитектуры диспетчера задач. Выполнение требований обеспечит корректное определение приоритета задач с учётом результатов дообучения модели и актуальных значений переменных формулы. Также эти требования должны гарантировать надёжность, производительность и наблюдаемость подсистемы в условиях переменной пользовательской нагрузки.

2 Проектирование веб-приложения с подсистемой диспетчеризации задач

2.1 Общая архитектура системы

Архитектура веб-приложения (рисунок 25) для загрузки, обработки и просмотра результатов улучшения медиафайлов разделена на фронтенд-часть, бэкенд-часть, менеджер сообщений, диспетчер задач, конвейер обработки, базу данных, локальное файловое хранилище и WebSocket [5].

Фронтенд используется для взаимодействия с пользователем, бэкенд принимает запросы, сохраняет файлы и создаёт задачи, менеджер сообщений обеспечивает асинхронную передачу сообщений, диспетчер определяет порядок обработки, а конвейер выполняет саму обработку медиафайлов. Результаты и статусы возвращаются в бэкенд и передаются во фронтенд в реальном времени через WebSocket. Файлы хранятся в storage, а метаданные задач и пользователей в базу данных.

Такой подход позволяет отделить пользовательский интерфейс от вычислительно тяжёлой обработки, повысить устойчивость системы и упростить её масштабирование [6].

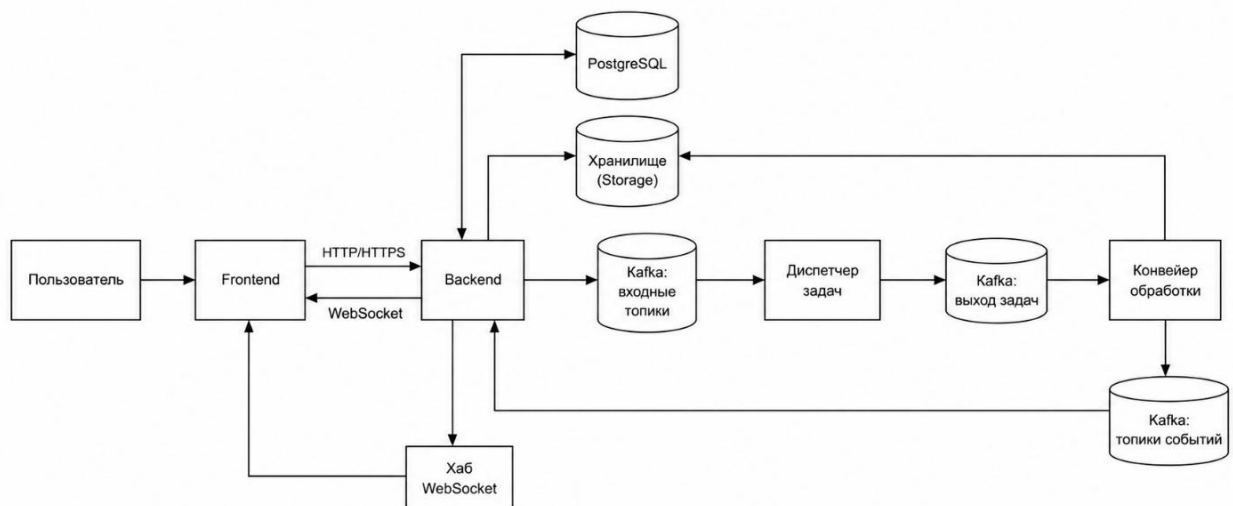


Рисунок 25 – Общая архитектура системы

2.2 Проектирование клиентской и серверной частей веб-приложения

2.2.1 Фронтенд-часть

Одним из основных подходов в написании фронтенда на React является Feature Sliced Design (FSD). Это основная методология построения структуры проекта. Нужна она, чтобы удобно было ориентироваться в компонентах. Явные модули нужны, иначе всё будет обрастать хаотичными связями.

Фича другую фичу использовать не может. Вышележащее может использовать нижележащее. Используются нижние слои, чтобы был линейный поток данных (однаправленный).

Каждый слой внутри себя содержит слайсы (модули). В случае с entities – это конкретные бизнес сущности (пользователь, статья, профиль) и каждая из них внутри себя содержит сегменты (api, компоненты, константы, хелперы), но здесь уже не абстрактные глобальные моменты, а конкретные бизнес сущности, запросы, которые относятся к пользователю, к товару, а также хелперы.

Фичи – это модули, которые несут какую-то бизнес-ценность: форма отправки фидбека, регистрация/авторизация по телефону, изменение номера. То есть это конкретные модули, которые несут какую-то бизнес-ценность. Они уже могут быть самостоятельными. При этом фичи могут использовать как shared-компоненты, так и функционал, который содержится в entities.

Виджеты – это максимально самостоятельные компоненты. Могут использовать внутри себя фичи. Это могут быть сайдбар, хедер, навбар. Они уже содержат в себе фичи. То есть это максимально самостоятельные модули приложения.

Каждый слайс внутри себя содержит сегменты:

- UI – это сами компоненты,
- model – бизнес-логика. Это взаимодействие со стейтом, селекторы, экшны,

- lib – это все хелперы, вспомогательные функции, которые могут использоваться внутри модуля,
- config – конфигурация модуля, если она требуется,
- api – запросы на сервер, которые требуются для данного модуля,
- consts – константы.

Поток данных однонаправленный.

На рисунке 26 показаны слои и сегменты методологии FSD.

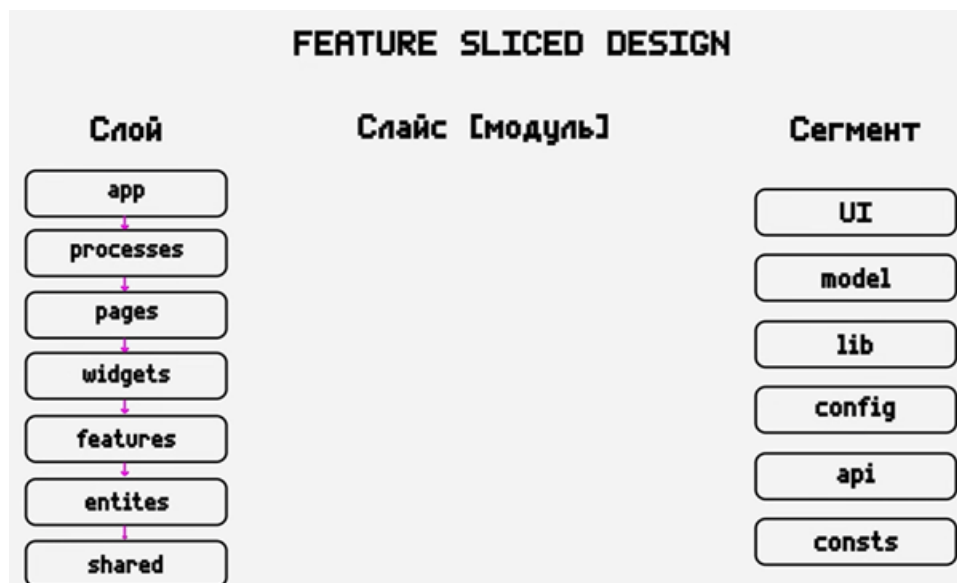


Рисунок 26 – Слои и сегменты методологии FSD

Чем ниже модуль, тем опаснее в него вносить изменения.

Существуют следующие слои:

- shared – это максимально переиспользуемые модули (кнопки, модальные окна, не привязанные к бизнес-логике хелперы, конфигурации). Не привязан к бизнес-логике, можно переместить из одного проекта в другой,
- entities – бизнес сущности, которыми оперирует предметная область (товар, заказ, корзина, пользователь, комментарий или отзыв). В модулях на этом слое располагается всё, что связано с конкретной сущностью. Это разные переиспользуемые хелперы, которые переиспользуются на слоях выше,

– features – пользовательские сценарии, которые несут какую-то бизнес-ценность (подписка на пользователя, поставить лайк, оценка товара с проставлением звёзд от 1 до 5),

– widgets – самостоятельные полноценные блоки для страниц,

– pages – страницы приложения,

– processes – процессы приложения, протекающие над страницами. Это когда есть процесс, который затрагивает несколько страниц. Например, многоэтапная регистрация, когда заполняем одну форму, переходим на следующую и т.д. Для этого нужен процесс, чтобы глобально связать несколько страниц,

– app – инициализирующая логика приложения. Здесь определяются провайдеры, роутеры, глобальная конфигурация, глобальные стили, корневой компонент. То есть это энтри-поинт проекта.

Чем выше слой, тем больше он привязан к предметной области и тем больше в нём бизнес-логики. Чем ниже, тем слой более абстрактный, переиспользуемый и несамостоятельный. Например, страницы максимально самостоятельны и переиспользоваться нигде не могут.

На рисунке 27 показаны все существующие слои из методологии FSD.

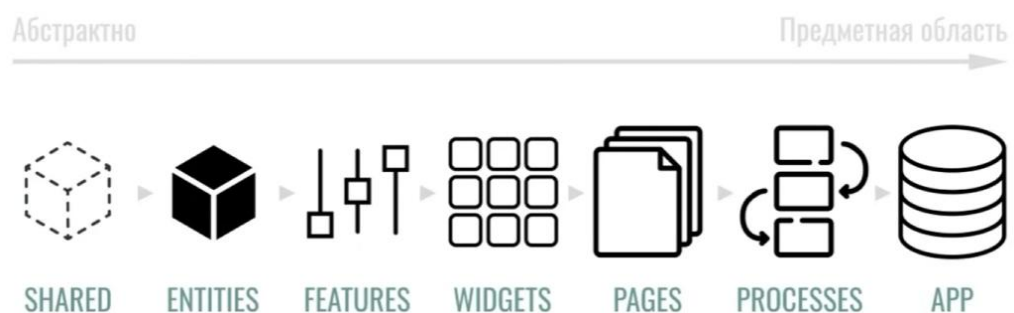


Рисунок 27 – Все существующие слои из методологии FSD

Также есть сценарии использования концепции ООП (полиморфизм, инкапсуляция и наследование) в вебе.

Полиморфизм и абстракцию получаем за счёт слоёв. Нижние слои наиболее абстрактные и их можно переиспользовать. При этом в зависимости от условий тот или иной компонент/модуль может работать по-разному в зависимости от параметров, которые ему передать.

Инкапсуляция (модульность) достигается за счёт public api (публичного интерфейса). К содержимому модуля извне обращаться нельзя. Единственный способ что-то вытащить из модуля – это публичный интерфейс. Он изолирует составные части модуля, которые снаружи не нужны.

Наследование получаем с помощью слоёв. Вышележащие слои используют нижележащие слои и каким-то своим функционалом (за счёт пропсов, функций, api) расширяют.

Обеспечивается слабое зацепление и высокая связанность. В любой момент модуль можно без лишних действий удалить. За счёт таких явных однонаправленных связей достигаются слабые однонаправленные зацепления между модулями.

Модуль внутри себя должен быть сильно связан. Он должен решать одну конкретную задачу. Все кусочки этого модуля должны находиться рядом и не должны быть разбросаны по проекту. Это достигается тем, что модуль внутри себя содержит всё необходимое и у него есть public api, который позволяет предоставить публичный интерфейс, а все внутренности этого модуля остаются внутри, тем самым изолируясь.

В shared-слое нет отдельных слайсов (модулей). Здесь сразу расположены сегменты const, lib, api, ui. В shared ui находятся как раз компоненты переиспользуемые: кнопки, тексты, спиннеры, модалки. В shared lib находятся хуки, хелперы.

В entities-слое (сущности) представлены слоты, в которые фичи уже на уровне выше можно прокинуть. Внутри entities слоя находятся слайсы (конкретные модули), которые относятся к статье, продукту, пользователю. Внутри слайсов находятся сегменты (папочки): api (запросы, всё, что с ними связано. CRUD-операции, которые привязаны к сущности пользователя), components (ui, привязанные к сущности пользователя компоненты), lib (хелперы, хуки), модел (бизнес логика, работа со стейтом, селекторы, типы). Каждый из таких слайсов (модулей) должен обладать public api. Здесь не

отдаются компоненты и хелперы, которые используются внутри, они тут изолируются. Отдаём только то, что нужно.

features-слой – меню с действиями. Конкретно бизнес-функционал, который приводит к результату. Одна фича – это одна функциональность. Например, смена языка, авторизация по email.

widgets-слой – комбинация features и entities. То есть это самостоятельные смысловые блоки, которые комбинируют нижние слои.

page – страница собирается из виджетов, из фичей. Страница всегда остаётся максимально тонкой. Минимум бизнес-логики. В идеальном варианте – это перечисление виджетов и фичей, обёрнутое в layout.

app-слой – инициализирующая логика приложения. Это инициализирующая точка входа в приложение. Здесь хранятся все провайдеры, глобальные типы, стили, всё, что не будет использоваться ниже.

Использование именно методологии Feature Sliced Design в разрабатываемом проекте обеспечивает чёткое разделение ответственности между слоями, упрощает масштабирование фронтенд-части, а также даёт возможность независимо работать с отдельными пользовательскими сценариями. Такой подход особенно важен для веб-приложения с большим количеством экранов (галерея, страницы улучшения видео, личный кабинет) и активным использованием WebSocket-уведомлений.

2.2.2 Бэкенд-часть

Бэкенд-часть веб-приложения будет реализована на Node.js [7], Express.js [8] и TypeScript и сделана по трёхслойному принципу. Такой подход позволяет разделить обработку HTTP-запросов, бизнес-логику и работу с данными. Для валидации входных данных используются схемы Zod, а для доступа к базе данных используется современный типобезопасный ORM Prisma [9], совместимый с Node.js, а также являющийся графической студией для просмотра данных (Prisma Studio). Серверная платформа Node.js и фреймворк Express.js выбраны, потому что с их помощью можно создавать высокопроизводительные веб-приложения, обрабатывающие одновременно большое количество запросов. Также Node.js позволяет разрабатывать

масштабируемые веб-приложения, адаптирующиеся к изменяющимся нагрузкам.

Архитектура бэкенда (рисунок 28) разделена на три основных слоя:

- контроллеры. Отвечают за обработку HTTP-запросов и формирование HTTP-ответов. Контроллеры принимают данные от фронтенда, выполняют первичную проверку и передают управление в соответствующие сервисы. Здесь же происходит разбор multipart/form-data, получение загружаемых файлов и валидация структуры входных данных через схемы Zod. Каждый контроллер отвечает за свою группу эндпоинтов и не содержит сложной бизнес-логики,

- сервисы реализуют основную бизнес-логику приложения. Здесь выполняются операции по созданию задач обработки, формированию конфигурационного файла задачи, отправке сообщений в Kafka, обновлению статусов и синхронизации данных с пользовательским интерфейсом. Здесь же располагается логика обработки событий, приходящих из Kafka, и механизмы взаимодействия с локальным хранилищем файлов. Таким образом, сервисы связывают между собой контроллеры, хранилище файлов, базу данных и механизм асинхронной обработки,

- слой доступа к данным и инфраструктурные сервисы. Тут работа с базой данных и внешними техническими компонентами. В разрабатываемом проекте отдельный слой репозитория не выделен, но его роль выполняет Prisma-обёртка над PostgreSQL, объектно-реляционной СУБД с поддержкой ACID-транзакций и расширенной типизацией. Prisma обеспечивает типизированный доступ к БД. Помимо доступа к базе данных, к инфраструктурным сервисам относятся модули для работы с локальным файловым хранилищем storage, Kafka, WebSocket.

Такое разделение на слои позволяет выполнять независимое тестирование контроллеров, сервисов и инфраструктурных модулей, а также облегчает дальнейшее развитие приложения. Например, при необходимости можно заменить способ хранения файлов, изменить механизм обмена

сообщениями или доработать способ передачи статусов во фронтенд и не затрагивать всю бизнес-логику целиком.

Взаимодействие с внешними системами происходит через сервисы. Локальное файловое хранилище storage используется для размещения исходных файлов, результатов обработки, конфигурационных файлов и служебных маркеров задач. Apache Kafka применяется для асинхронной передачи сообщений между бэкендом, диспетчером задач и конвейером обработки. Для обмена статусами с пользовательским интерфейсом используется WebSocket.

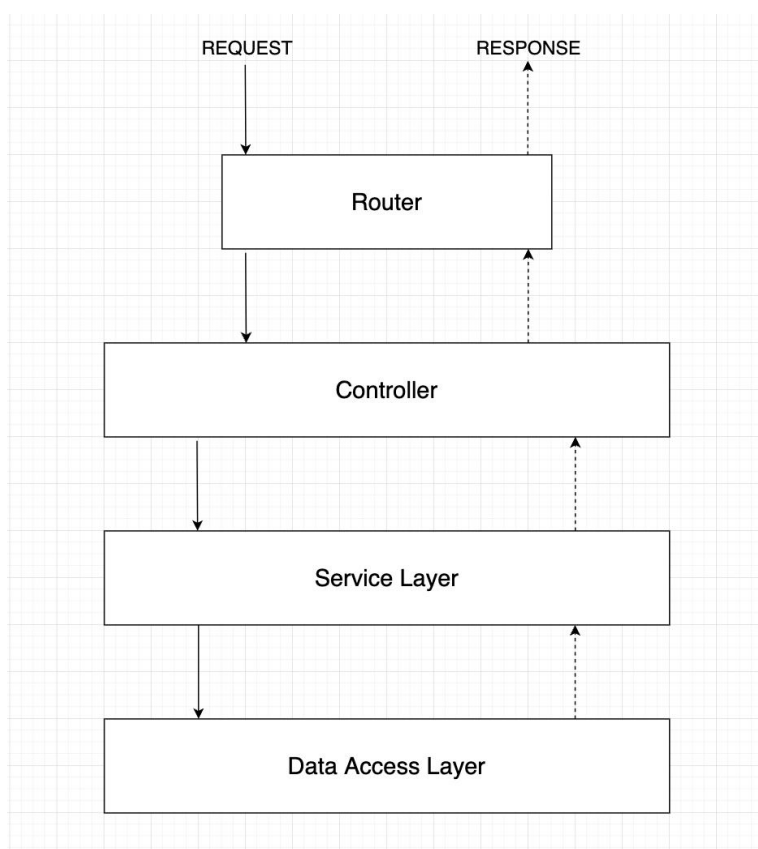


Рисунок 28 – Трёхслойная архитектура

Благодаря этому бэкенд выполняет роль центрального узла управления задачами, не занимаясь непосредственной вычислительной обработкой медиафайлов.

2.3 Проектирование системы обмена сообщениями

Для организации асинхронного взаимодействия между бэкендом и конвейером обработки необходимо использовать менеджер сообщений. Это позволит решить проблемы отказоустойчивости и оптимизировать механизм

взаимодействия. Из проектируемой архитектуры проекта можно выделить следующие требования к менеджеру сообщений:

- надёжность доставки. Задачи на обработку не должны теряться при внезапном отказе компонентов. Менеджер должен сохранять сообщения в файловой системе и давать возможность повторного чтения,
- гарантия порядка сообщений. Должно быть сохранение порядка сообщений в пределах одной партии,
- высокая пропускная способность. Система должна быть рассчитана на одновременную работу многих пользователей, загружающих медиафайлы. Поэтому менеджер должен справляться с пиковыми нагрузками без существенного снижения производительности,
- масштабируемость. Необходимо поддержать возможность увеличения числа конвейеров обработки медиа без вмешательства в другие компоненты.

Было рассмотрено несколько популярных инструментов: RabbitMQ, Active MQ, NATS и Apache Kafka.

RabbitMQ по официальной документации поддерживает гибкую маршрутизацию сообщений через обменники различных типов и использует push-модель доставки [10]. Также поддерживает подтверждения публикации, приоритеты потребителей и очереди с ограничением длины. Но у него низкая пропускная способность (до 50 тыс. сообщений/сек), отсутствует гарантия порядка сообщений. Также сообщения не хранятся на диске в базовой настройке. Для моего проекта перечисленные недостатки являются весомыми.

ActiveMQ обеспечивает надёжную асинхронную коммуникацию между компонентами распределенной системы и поддерживает различные протоколы передачи, по типу AMQP или MQTT. Является достаточно производительным менеджером сообщений с расчетным показателем отправки 21–22 тысяч сообщений в секунду при замерах сообщения до 1 КБ.

NATS, в стандартной конфигурации, легковесный и быстрый менеджер сообщений, с сохранением данных только в оперативной памяти [11]. Доставка сообщений гарантируется «не более одного раза» (at-most-once). Для обеспечения персистентности придётся использовать расширение JetStream, добавляющее хранение на диске, репликацию и доставку «как минимум один раз» (at-least-once). И всё же стандартная конфигурация NATS не соответствует требованиям надёжности из-за перечисленных выше проблем.

Apache Kafka по официальной документации использует pull-модель доставки, сохраняет данные на диск и реплицирует их, что обеспечивает персистентность и отказоустойчивость [12]. Внутри одной партии порядок сообщений гарантированно сохраняется, что позволяет направлять задачи по ключу `userId` в одну партию и обрабатывать их последовательно.

В сравнении характеристик были использованы результаты собственных бенчмарков.

На рисунке 29 продемонстрирован график сравнения пропускной способности менеджеров сообщений.

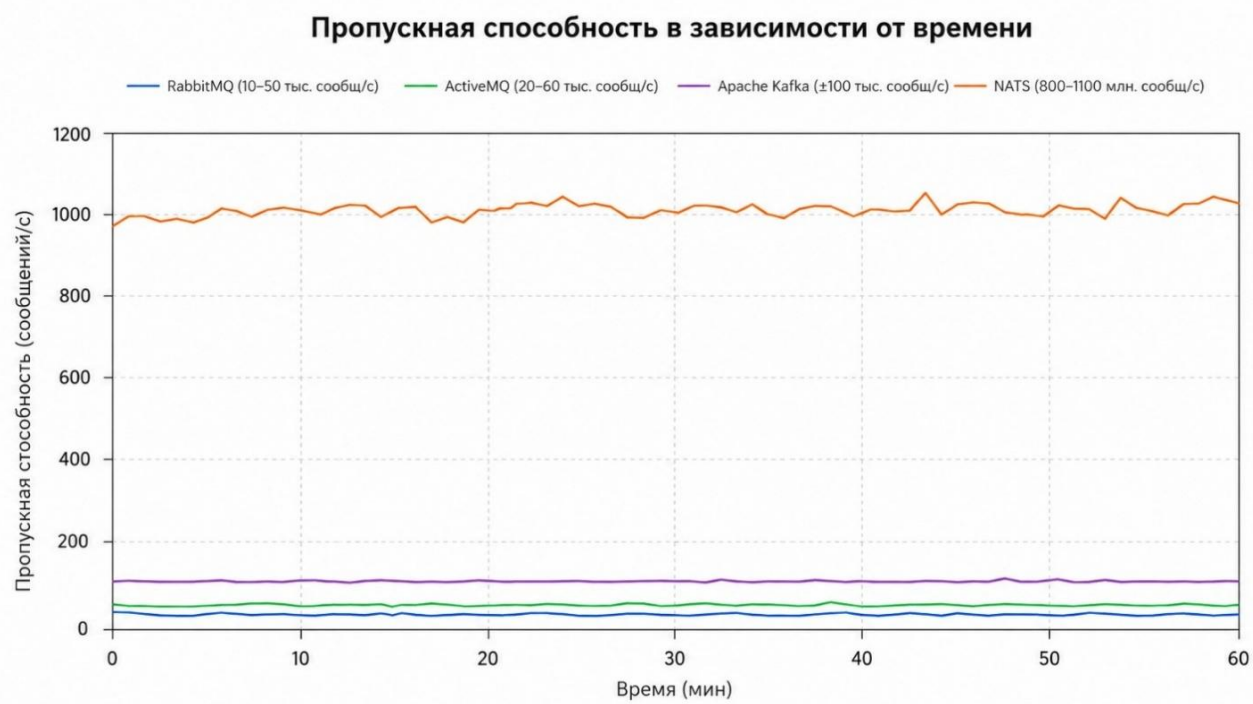


Рисунок 29 – Пропускная способность в зависимости от времени

В таблице 16 представлено сравнение ключевых характеристик рассмотренных менеджеров сообщений.

Таблица 16 – Сравнение характеристик менеджеров сообщений

Решение	Пропускная способность	Персистентность	Причина отказа
RabbitMQ	10–50 тыс. сообщ/с	Частичная (удаление после доставки)	Недостаточная скорость, отсутствие долговременного аудита
ActiveMQ	20–60 тыс. сообщ/с	Есть	Скорость ниже, чем у Kafka, риск деградации при пиковых нагрузках
Apache Kafka	±100 тыс. сообщ/с	Полная (реплицируемый журнал)	—
NATS	800–1100 млн. сообщ/с	Отсутствует в базовой версии	Нет сохранности метрик

В итоге был выбран менеджер сообщений Apache Kafka (рисунок 30), потому что он соответствует большинству требований проекта. NATS Core, несмотря на высокую производительность, не обеспечивает сохранность данных, что критично для надёжной обработки задач. RabbitMQ и ActiveMQ уступают как по производительности, так и по некоторой доступной функциональности.

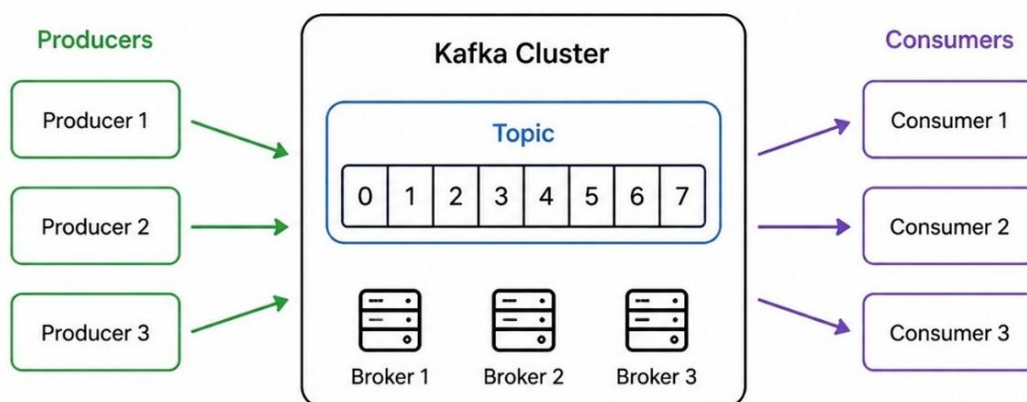


Рисунок 30 – Apache Kafka

При использовании Apache Kafka даже если все конвейеры перестанут работать, задачи не потеряются. Когда они снова запустятся, прочитают всё,

что накопилось в очереди. Это позволяет обеспечить высокую степень отказоустойчивости при возможных отказах сетевого оборудования. Благодаря единой очереди сообщений, проектируемый далее диспетчер задач будет испытывать меньшую нагрузку при получении метрик сформированных задач.

Поддержка возможности быстрого вложения и извлечения сообщений позволит множеству пользователей одновременно отправлять запросы на обработку без конфликтов синхронности и внутренней упорядоченности задач. Apache Kafka должен справляться с этим без заметной потери скорости при будущей оценке эффективности системы.

2.4 Анализ алгоритмов диспетчеризации

Для дальнейшей эффективной обработки сформированных пользовательских запросов в вычислительном кластере были отобраны и проанализированы различные алгоритмы диспетчеризации поступающих задач. Данное сравнение необходимо для определения возможной эффективной стратегии работы, на которой будет основан проектируемый диспетчер задач. Возможно, что не один из представленных алгоритмов не подойдет в конкретном случае для высокой эффективности работы с генеративно-состязательными сетями из-за проблем, описанных в пункте исследования 1.1.

First In, First Out или FIFO является простейшей стратегией диспетчеризации, при которой задачи извлекаются из очереди строго в порядке их поступления (рисунок 31) [13]. Самый простой в реализации подход к приоритезации задач, однако применительно к задаче улучшения качества видео демонстрирует существенные недостатки организации. Главный недостаток – блокировка коротких задач более длительными. Если в очередь первым поступил видеофайл продолжительностью в несколько часов, все последующие короткие ролики будут ожидать его полной обработки. Это приводит к отказу в обслуживании и непредсказуемому времени отклика с точки зрения пользователя. FIFO не учитывает гетерогенность вычислительной сложности задач, характерную для

конвейера улучшения качества видео, что полностью отменяет полезность встраиваемого диспетчера задач.

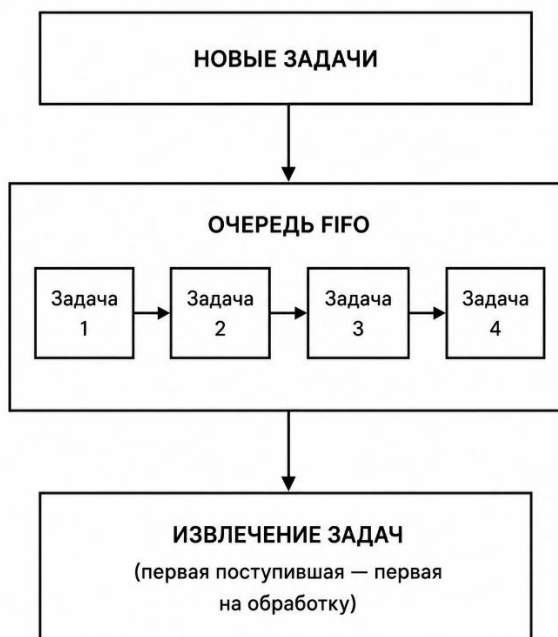


Рисунок 31 – Алгоритм FIFO

Циклический перебор Round Robin – один из самых простых и распространённых алгоритмов. Работает он по принципу кругового распределения задач (рисунок 32), то есть первый запрос направляется на первый сервер, второй – на второй, и так далее по порядку. Когда очередь доходит до последнего сервера, алгоритм возвращается к первому. Round Robin распределяет запросы по доступным серверам по очереди, независимо от количества соединений или нагрузки на сервер [14]. Это его главный недостаток. Если один из серверов занят обработкой тяжёлой задачи, он всё равно будет получать новые запросы в порядке очереди, что может привести к перегрузке и увеличению времени ожидания.

Подход эффективен в тех случаях, когда все запросы обрабатываются примерно за одинаковое время, что в сценарии улучшения качества видео невозможно из-за различия задач по длительности и ресурсоемкости. Также для данного подхода необходимо иметь различные очереди для каждого сервера, что сильно усложняет работу диспетчеру задач из-за распределенного контроля за несколькими очередями. Желательно использовать одну очередь для упрощения выдачи приоритетов.



Рисунок 32 – Алгоритм Round Robin

Алгоритмы приоритетной диспетчеризации или priority-based scheduling предлагают назначение каждой обрабатываемой задаче числового показателя приоритета, на основе которого определяется её положение в очереди (рисунок 33). Приоритет может быть статическим, задаваемым на основе присутствия каких-либо флагов, либо динамическим, рассчитываемым на основе каких-либо получаемых параметров. Приоритетная диспетчеризация, в отличие от FIFO и Round Robin, позволяет решить проблему блокировки коротких задач длинными, если приоритет действительно коррелирует с ожидаемой длительностью обработки.

Главным недостатком данного подхода является необходимость обоснованного выбора формулы для расчета коэффициента приоритета, что может быть затруднительно. Однако в пункте 1.3 уже была рассмотрена составляющая зависимостей характеристик от времени обработки и выведена предварительная формула для расчёта приоритета полученной задачи улучшения качества. Это позволяет рассмотреть данный алгоритм для использования в проектируемом диспетчере задач.

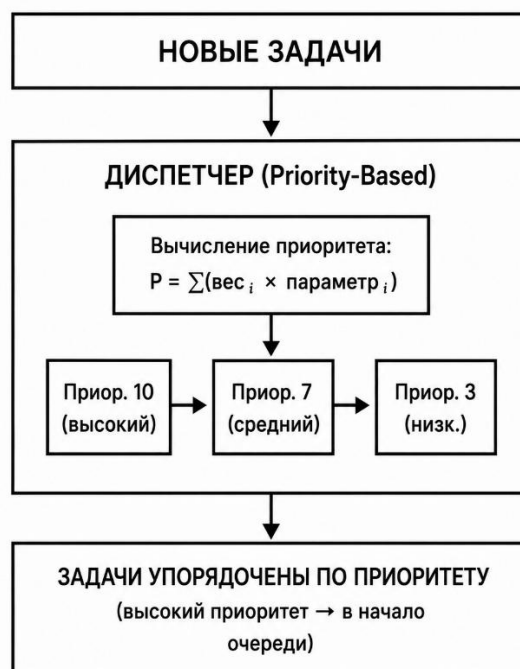


Рисунок 33 – Алгоритм приоритетной диспетчеризации

Взвешенное справедливое квотирование или WFQ – это алгоритм предварительной классификации задач для получения доли пропускной способности в системе (рисунок 34) [15]. При задаче улучшения качества видео применение данного алгоритма значило бы обработку коротких и длинных видео в заданной пропорции. Длительность является примерным критерием для описания работы данной методологии. Пропорции можно применить к любым характеристикам поступающих видео, но это полностью не решает проблему полной блокировки системы одним классом вычислений.

Для реализации данной системы требуется обязательное использование N-количества очередей для реализации квотирования, иначе задачи, полученные в условиях непредсказуемой пользовательской нагрузки, также будут обрабатываться в порядке поступления и частично блокировать существующую очередь. Можно рассмотреть данный подход, однако он значительно уступает приоритетной диспетчеризации в гибкости обработки в рамках одной очереди.

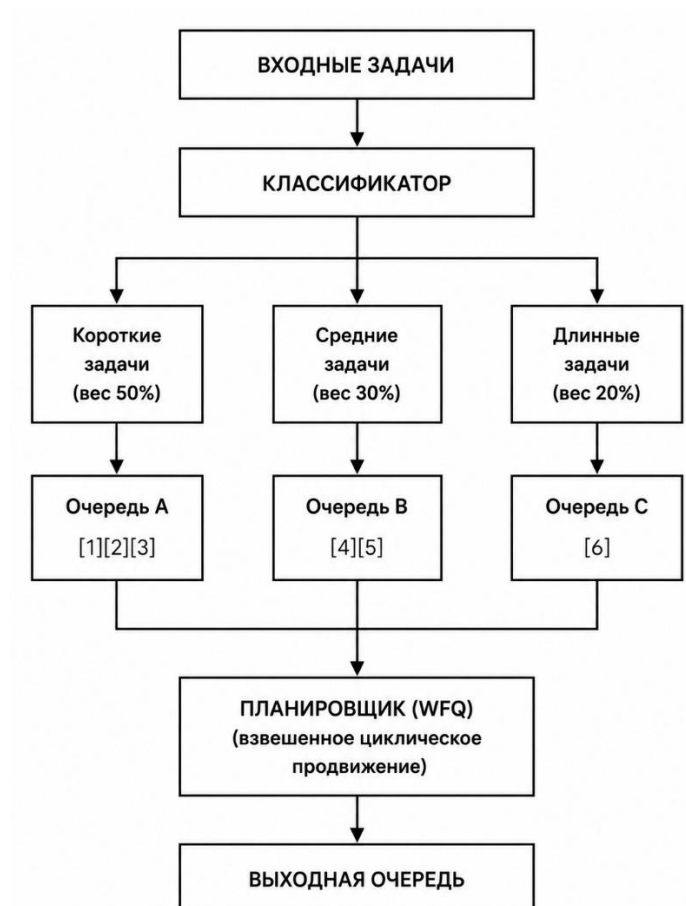


Рисунок 34 – Алгоритм взвешенного справедливого квотирования

Таблица 17 отображает преимущества и недостатки всех вышеописанных алгоритмов.

Таблица 17 – Преимущества и недостатки алгоритмов диспетчеризации задач

Подход к диспетчеризации	Преимущества	Недостатки
FIFO	Максимальная простота реализации, предсказуемое поведение, отсутствие накладных расходов на приоритезацию	Длинные задачи блокируют короткие, не позволяет реализовать дифференцированное обслуживание пользователей, может привести к отказу в обслуживании

Продолжение таблицы 17:

Подход к диспетчеризации	Преимущества	Недостатки
Round Robin	Прост в реализации, равномерно распределяет нагрузку, нет необходимости в централизованной очереди задач	Предполагает одинаковую сложность всех задач, не учитывает загрузку обработчиков, может создать локальные перегрузки для кластеров обработки
PBS	Предотвращает блокировку очереди задач, позволяет реализовать дифференцированное обслуживание пользователей, адаптируется к динамическому профилю нагрузки	Сложен в реализации, высокие накладные расходы на расчет приоритета
WFQ	Гарантирует долю вычислительных ресурсов каждому типу задач, имеет защиту от полной блокировки очереди	Требует предварительного определения класса задач, плохо адаптируется к динамическому профилю нагрузки, в некоторых случаях может вызвать сложность подбора коэффициентов нагрузки

Для наглядного сравнения производительности различных алгоритмов на рисунке 35 представлены результаты экспериментального исследования. Приведены результаты пропускной способности в таблице 18.

Таблица 18 – Эффективность алгоритмов диспетчеризации

Алгоритм	Средняя задержка (сек)	P95 задержка (сек)	Успешность
FIFO	420	1850	94,2%
Round Robin	385	1920	93,8%
PBS	180	1120	98,4%
WFQ	220	1310	96,7%

Сравнение алгоритмов планирования

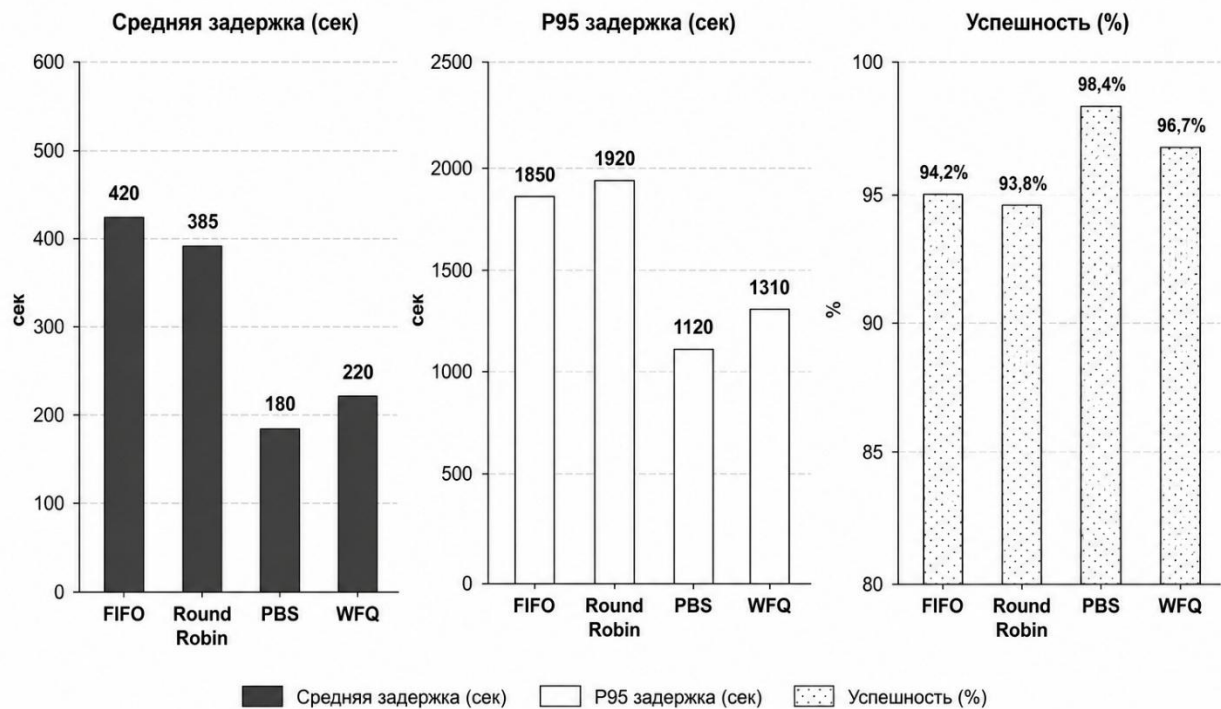


Рисунок 35 – Сравнительный график эффективности алгоритмов диспетчеризации

В конкретном случае для диспетчера задач веб-приложения улучшения качества видео подойдет алгоритм приоритетной диспетчеризации. Несмотря на вычислительную сложность и возможные проблемы с формированием формулы для определения приоритета, данный подход позволяет обеспечить минимальное среднее время ожидания для клиента. Из-за наличия задач, которые могут быть различными по времени выполнения и затраченной вычислительной мощности, данный алгоритм предварительно подходит лучше всего для проектирования и реализации планировщика диспетчера задач.

3 Реализация веб-приложения и подсистемы диспетчеризации

3.1 Реализация клиентской и серверной частей веб-приложения

Веб-приложение реализовано как разделённая на две части система: фронтенд-часть, отвечающая за пользовательский интерфейс, и бэкенд-часть, обеспечивающая обработку запросов, работу с данными, хранение файлов и взаимодействие с внешним конвейером обработки.

3.1.1 Фронтенд-часть

В основе фронтенд-части в моём проекте лежит стек технологий, состоящий из декларативной JavaScript-библиотеки React, которая отвечает за построение пользовательского интерфейса на основе компонентного подхода, языка программирования TypeScript, который гарантирует типобезопасность взаимодействия с серверной частью, а также инструмента сборки Vite, который предоставляет механизм горячей замены модулей (HMR), существенно ускоряющий процесс разработки.

Архитектура фронтенд-части построена на компонентном подходе по методологии FSD (рисунок 36). Пользовательский интерфейс разбит на отдельные страницы, виджеты и переиспользуемые элементы, что упрощает сопровождение приложения и повторное использование логики.

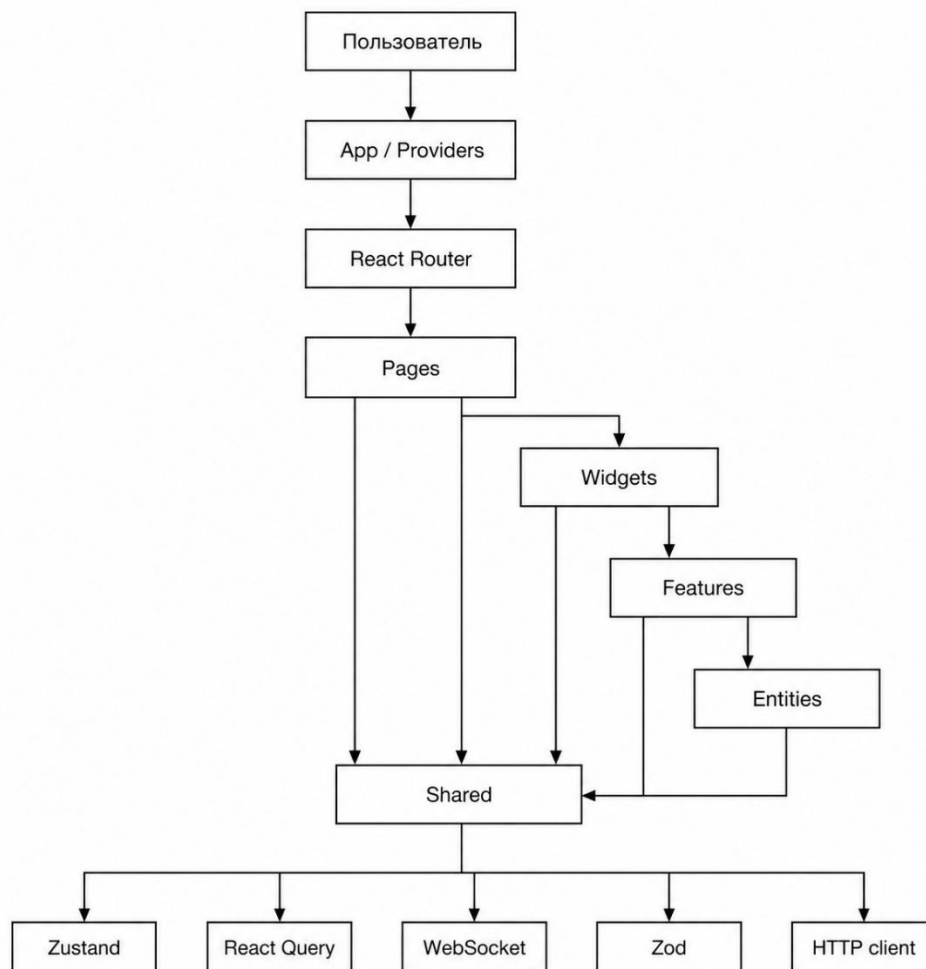


Рисунок 36 – Архитектура фронтенда

В приложении реализованы основные пользовательские сценарии: переход на главную страницу, авторизация и регистрация, просмотр публичной и личной галерей, переход к страницам обработки изображений и видео, а также просмотр профиля пользователя.

3.1.1.1 Структура страниц и компонентный подход

Основные страницы реализованы как отдельные компоненты уровня pages, общие элементы интерфейса вынесены в widgets и shared, а функциональная логика сгруппирована в features и entities.

Центральным элементом структуры является роутер приложения, который связывает пользовательские маршруты с конкретными страницами. В app-router.tsx задаются маршруты на главную страницу, авторизацию, регистрацию, страницу информации, публичную и личную галереи, а также страницы обработки изображений и видео.

Для объединения глобальной инфраструктуры интерфейса используется компонент `AppProviders`. Он подключает `QueryClientProvider`, настройку темы, восстановление сессии и `WebSocket`-соединение для обновления статусов задач. Это делает инициализацию приложения прозрачной и не перегружает отдельные страницы служебной логикой.

Компоненты уровня `pages` отвечают за отображение основных пользовательских сценариев: вход в систему, регистрацию, просмотр галереи, загрузку медиа и запуск обработки. Например, `PageWelcome` служит входной точкой в приложение, `PageImageEnhance` и `PageVideoEnhance` реализуют сценарии отправки медиа на обработку, а `PagePrivateGallery` и `PagePublicGallery` отображают результаты работы системы в виде галерей.

Крупные блоки интерфейса инкапсулированы в виджеты. К ним относится шапка сайта (`AppHeader`), отвечающая за навигацию и отображение пользователя, галерея, модальное окно с деталями медиафайла, виджет статуса задач (`ProgressWidget`), показывающий состояние активных задач в реальном времени, и `toast`-уведомления. Виджеты позволяют не собирать логику прямо на страницах, а использовать готовые блоки, которые можно подключать в нужных местах без дублирования кода.

Отдельную часть составляют элементы общего назначения, расположенные в `shared`. К ним относятся базовые UI-компоненты, хелперы, глобальные библиотеки и механизмы для работы с данными. Например, `RealtimeBridge` подключает `WebSocket` и обновляет состояние интерфейса при поступлении событий о задачах, `SessionBootstrap` восстанавливает сессию пользователя после перезагрузки страницы, а `ThemeBootstrap` применяет тему оформления и язык интерфейса.

Бизнес-логика изолирована в функциональных модулях уровня `features` и `entities`. Например, модуль авторизации содержит мутации для входа, регистрации и обновления профиля, а модуль медиафайлов — запросы к галерее, деталям медиа и операциям вроде добавления в избранное или изменения публичности. Такой подход позволяет отделить бизнес-логику от отображения и делает код более предсказуемым. Модуль загрузки

задач на обработку реализует мутации отправки изображений и видео. После успешного выполнения кеш React Query инвалидируется, чтобы интерфейс обновил список активных задач и личную галерею.

3.1.1.2 Навигация и маршрутизация

Для организации переходов между страницами используется React Router. Маршруты связаны с конкретными экранами приложения, а часть страниц защищена проверкой авторизации. Это позволяет разграничить доступ к публичным и приватным разделам системы.

Публичные страницы доступны без ограничений, а приватные маршруты обёрнуты в компонент RequireAuth. Если токен отсутствует, пользователь перенаправляется на страницу входа. Этот механизм решает задачи безопасности интерфейса контроля доступа к личным данным пользователя.

Так выглядит маршрутизация в данном веб-приложении:

```
export function AppRouter() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path={APP_ROUTES.welcome}
element={<PageWelcome />} />
        <Route path={APP_ROUTES.login} element={<PageLogin
/>} />

        <Route path={APP_ROUTES.register}
element={<PageRegister />} />
        <Route path={APP_ROUTES.about} element={<PageAbout
/>} />

        <Route path={APP_ROUTES.publicGallery}
element={<PagePublicGallery />} />
        <Route
          path={APP_ROUTES.privateGallery}
          element={
            <RequireAuth>
              <PagePrivateGallery />
            </RequireAuth>
          }
        />

      </Routes>
    </BrowserRouter>
  );
}
```

3.1.1.3 Работа с состоянием с использованием технологии Zustand

Библиотека Zustand используется, чтобы удобно хранить глобальное состояние, а также для управления следующими данными: информация о текущем пользователе, язык интерфейса, тема, токен авторизации, активный в обработке медиафайл, а также информацию о последних задачах. Эти данные доступны нескольким компонентам сразу. Чтобы обеспечить сохранение сессии пользователя, а также темы, языка, Zustand сохраняет эти данные в localStorage через механизм persist. Снижается зависимость компонентов друг от друга и всё выстраивается вокруг общего состояния.

3.1.1.4 Инициализация веб-приложения

В приложении используется слой провайдеров, который отвечает за глобальные функции интерфейса: кеширование запросов, восстановление сессии, realtime-канал и настройку темы.

Этот компонент служит основой для работы приложения.

SessionBootstrap проверяет сессию пользователя, ThemeBootstrap устанавливает тему и язык, RealtimeBridge подключает WebSocket для обновления статусов у задач. Так основные технические процессы запускаются только один раз на уровне всего приложения.

3.1.1.5 Работа с данными через React Query

Чтобы загружать и обновлять данные с сервера используется такая технология, как React Query. Эта библиотека помогает делать запросы к галерее, профилю пользователя, информации о конкретном медиафайле, а также списку задач. Также React Query инвалидирует запросы, обеспечивает кеширование и предоставляет удобный способ работать с изменениями.

Пример хука useActiveTasksQuery, который запрашивает список активных задач:

```
export function useActiveTasksQuery(enabled: boolean,
  refetchInterval = 15000) {
  return useQuery({
    queryKey: ['tasks', 'active'],
    queryFn: getActiveTasks,
    enabled,
    refetchInterval: enabled ? refetchInterval : false,
    refetchOnWindowFocus: false,
```

```

    refetchOnReconnect: false,
  });
}

```

После каких-нибудь успешных операций мутации инвалидируют связанные кешы, чтобы интерфейс сразу отображал актуальные данные. После изменения статуса избранного инвалидируются запросы к личной галерее и к конкретному медиафайлу:

```

export function useToggleFavoriteMutation() {
  const queryClient = useQueryClient();
  return useMutation({
    mutationFn: ({ mediaId, isFavorite }) =>
      toggleFavoriteRequest(mediaId, isFavorite),
    onSuccess: async (_data, variables) => {
      await queryClient.invalidateQueries({ queryKey:
        ['gallery', 'private'] });
      await queryClient.invalidateQueries({ queryKey:
        ['media', variables.mediaId] });
    },
  });
}

```

Это особенно полезно для мест, где данные часто меняются после обработки на бэкенде. React Query автоматически управляет состоянием загрузки, ошибок и обновления данных.

3.1.1.6 Валидация данных с помощью Zod

Для проверки структуры данных на клиенте используется Zod. В проекте применяются общие схемы из shared-пакета. Zod используется при разборе событий, приходящих по WebSocket, а также при проверке структуры ответов от бэкенда. Метод `safeParse` проверяет корректность входящего JSON-сообщения; если структура не соответствует схеме, событие игнорируется. Это снижает риск ошибок при обработке данных в интерфейсе и делает поведение приложения более предсказуемым.

3.1.1.7 Обновление интерфейса в реальном времени

Для получения изменений статусов задач без перезагрузки страницы используется WebSocket-соединение. Компонент `RealtimeBridge` открывает соединение от имени текущего пользователя, принимает события и обновляет интерфейс.

При получении сообщения данные проходят валидацию через Zod. При успешной проверке обновляется глобальное состояние задачи и

инвалидируются кеши React Query для активных задач, истории, личной галереи и конкретного медиафайла:

```
socket.addEventListener('message', (event) => {
  const parsed =
taskEventSchema.safeParse(JSON.parse(event.data));
  if (!parsed.success) return;

  setLatestTaskEvent(parsed.data);
  if (parsed.data.type === 'task:completed') {
    setLatestCompletedTaskEvent(parsed.data);
  }
  void queryClient.invalidateQueries({ queryKey: ['tasks',
'active'] });
  void queryClient.invalidateQueries({ queryKey: ['gallery',
'private'] });
  void queryClient.invalidateQueries({ queryKey: ['media',
parsed.data.mediaId] });
});
```

Пользователь сразу видит изменение статуса задачи или появление результата в галерее.

3.1.2 Бэкенд-часть

Бэкенд-часть веб-приложения реализована на базе Node.js, Express.js и TypeScript. Она отвечает за обработку HTTP-запросов, создание и сопровождение задач обработки медиафайлов, работу с PostgreSQL, хранение файлов в локальном storage, а также взаимодействие с Kafka и WebSocket. Такая архитектура (рисунок 36) позволяет отделить пользовательский интерфейс от вычислительно тяжёлой логики обработки и сделать систему более устойчивой к высокой нагрузке.

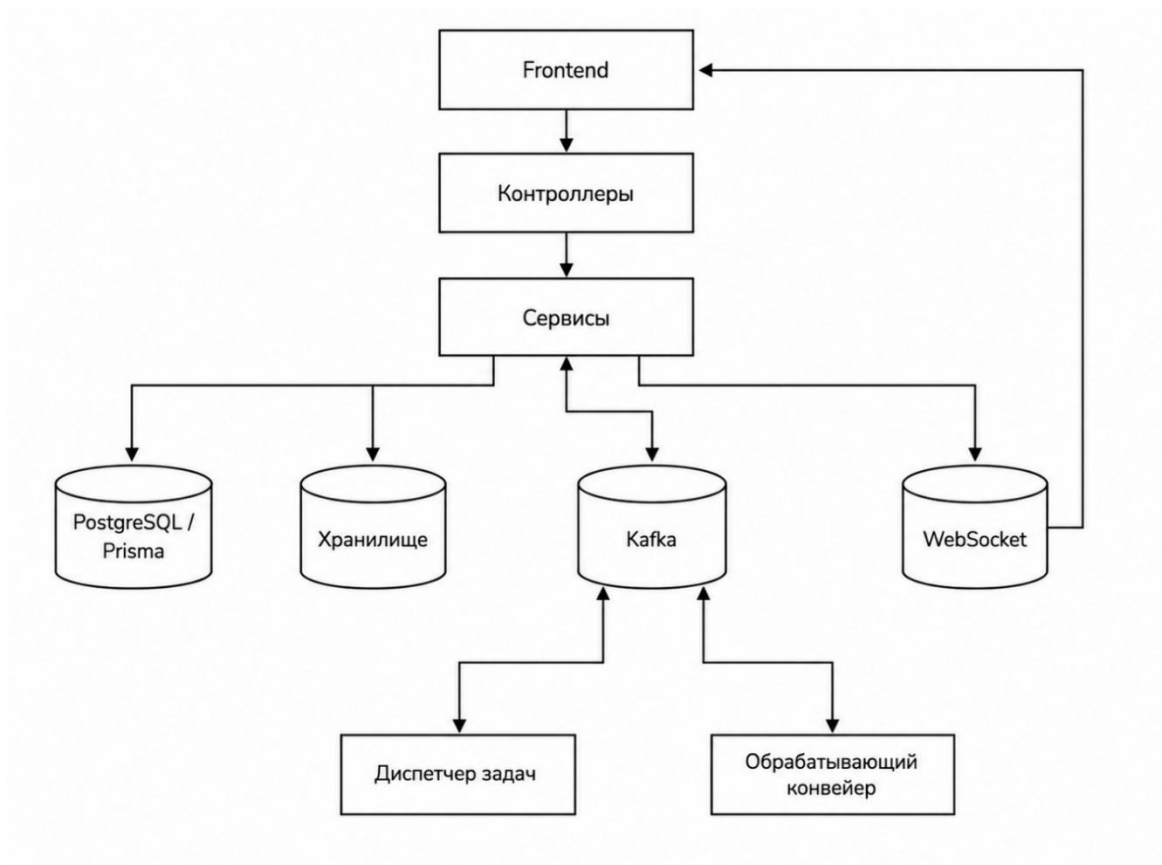


Рисунок 36 – Архитектура бэкенда

3.1.2.1 Контроллеры и HTTP-слой

Контроллеры в бэкенде отвечают за приём запросов от фронтенда, первичную проверку данных и передачу управления в сервисный слой. В проекте реализованы отдельные контроллеры для авторизации, работы с галереями, задачами и загрузкой медиафайлов. Это позволяет разделить ответственность между маршрутами и не смешивать HTTP-обработку с бизнес-логикой.

Например, контроллер загрузки и создания задач на обработку. Реализованы сценарии отправки изображения и видео. Каждый из них принимает загруженный файл, извлекает параметры обработки, создаёт структуру хранения и инициирует создание задачи.

Пример обработки запроса на создание задачи:

```
export async function createImageTaskController(request: Request, response: Response) {
  const file = request.file;
  if (!file || !request.user) {
    response.status(400).json({ message: 'File and auth required' });
  }
}
```

```

        return;
    }

    const input = createImageTaskSchema.parse(request.body);
    const processingConfig = parseSubmittedConfig(
        request.body.config,
        imageProcessingConfigSchema.parse({
            model: input.model,
            targetResolution: input.targetResolution,
        }),
        imageProcessingConfigSchema.parse
    );
}

```

Этот контроллер выполняет только первичную обработку запроса: проверяет наличие файла и авторизации, валидирует тело запроса и извлекает конфигурацию обработки. Более сложные действия, такие как создание записи в базе, формирование пакета задачи и публикация сообщения в Kafka, вынесены в сервисный слой.

3.1.2.2 Сервисный слой и бизнес-логика

Основная бизнес-логика бэкенда сосредоточена в сервисах. Здесь формируются задачи обработки, обновляются статусы, выполняется синхронизация галереи, создаются конфигурационные файлы и публикуются сообщения в Kafka.

Сервис `task.service.ts` отвечает за создание задачи и обновление её состояния. При создании задачи бэкенд сохраняет запись в таблице `EnhanceTask`, формирует пакет задачи и публикует сообщение во входной топик Kafka (`fullqualityai.image-input` или `fullqualityai.video-input`). Бэкенд передаёт только метаданные файла и параметры обработки. Вычисление приоритета выполняется в диспетчере задач.

Фрагмент создания задачи:

```

const task = await db.enhanceTask.create({
  data: {
    id: input.taskId,
    userId: input.userId,
    mediaId: input.mediaId,
    model: input.model,
    targetResolution: input.targetResolution,
    fps: input.fps,
    previewMode: input.previewMode ?? false,
    previewDurationSec: input.previewDurationSec ?? 15,
    stage: 'Queued',
  }
});

```

```

    },
  });

  const routing = buildFileRouting(input.kind,
input.sizeOriginal, {
  previewMode: input.previewMode,
  targetResolution: input.targetResolution,
  durationSec: input.durationSec,
  userTier: input.userTier,
  originalFilename: input.originalFilename,
  codec: input.codec,
});

```

Бэкенд сразу рассчитывает характеристики маршрутизации задачи, т.к. дальнейшая публикация в Kafka строится на основе `routingKey`, который формируется до отправки сообщения. Затем вызывается `writeTaskPackage`, которая сохраняет в `config.json` описание задачи, пути к файлам и параметры обработки.

Функции, выполняемые сервисным слоем:

- создаёт и обновляет задачи в БД,
- готовит конфигурацию для конвейера обработки,
- публикует сообщения во входной топик Kafka,
- синхронизирует состояние с галереей и интерфейсом.

3.1.2.3 Слой доступа к данным и Prisma

Для работы с PostgreSQL используется Prisma. Отдельный слой репозитория не выделен, т.к. роль слоя доступа к данным выполняет Prisma-клиент и сервисы, которые работают с ним напрямую. Это делает структуру бэкенда более компактной и снижает количество промежуточных абстракций.

База данных (рисунок 37) содержит сущности (схема Prisma определяет связи между ними):

- User (связывается с медиафайлами и задачами через отношения один-ко-многим),
- Media (содержит тип файла, путь к исходному и результатному файлу, размеры, длительность и признаки публичности),
- EnhanceTask (хранит статус задачи, прогресс, стадию, параметры обработки и временные отметки).

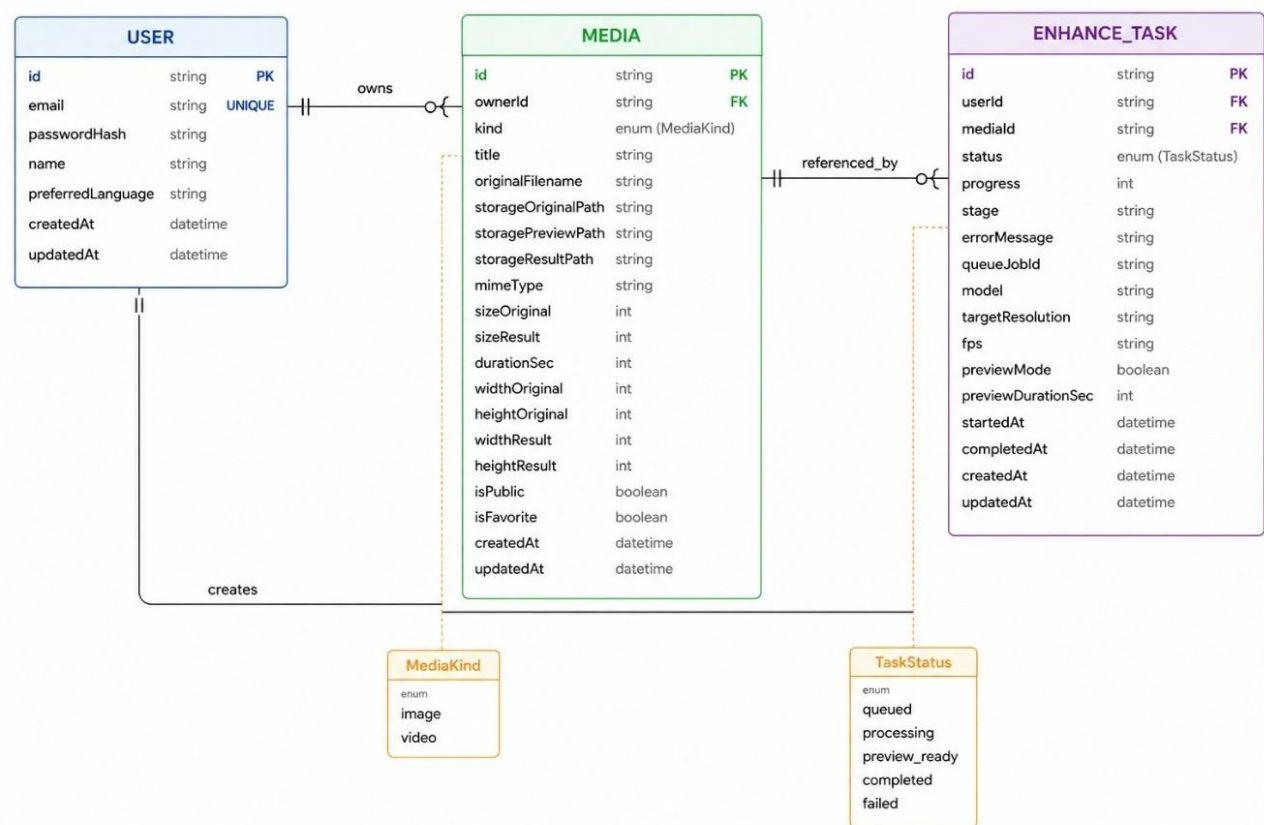


Рисунок 37 – Структура базы данных

При обновлении задачи используется транзакция, чтобы состояние EnhanceTask и Media изменялось согласованно:

```
const task = await db.$transaction(async (tx) => {
  const task = await tx.enhanceTask.update({
    where: { id: input.taskId },
    data: {
      status: input.status,
      progress: input.progress,
      stage: input.stage,
      errorMessage: input.errorMessage,
      startedAt: input.status === 'processing' ? new Date()
: undefined,
      completedAt:
        input.status === 'completed' || input.status ===
'failed' ? new Date() : undefined,
    },
  });

  if (input.previewPath || input.resultPath) {
    await tx.media.update({
      where: { id: task.mediaId },
      data: {
        storagePreviewPath: input.previewPath,
        storageResultPath: input.resultPath,
      },
    });
  }
});
```

```
    return task;
  });
```

3.1.2.4 Инфраструктурные сервисы: storage, Kafka и WebSocket

Бэкенд также включает инфраструктурные модули для взаимодействия с внешними системами. Локальное файловое хранилище storage используется для размещения исходных файлов, результатов обработки, конфигурационных файлов и служебных маркеров задач. Apache Kafka применяется для асинхронной передачи задач между бэкендом, диспетчером задач и конвейером обработки. Для обмена статусами с пользовательским интерфейсом используется WebSocket.

Работа с файловой системой реализована через отдельные сервисы, которые создают рабочую структуру задачи, сохраняют исходные файлы и формируют конфигурационный пакет. В config.json записываются метаданные, пути к рабочим директориям, параметры обработки и routing-данные.

Kafka-интеграция позволяет бэкенду публиковать задачи в топики fullqualityai.image-dispatch и fullqualityai.video-dispatch, а результаты получать из fullqualityai.image-events и fullqualityai.video-events.

Пример отправки задачи в Kafka:

```
await producer.send({
  topic: resolveDispatchTopic(payload.kind),
  messages: [{ key: parsed.routing.routingKey, value:
JSON.stringify(parsed) }],
});
```

Это позволяет маршрутизировать задачи по типу файла и по рассчитанному ключу нагрузки.

Для доставки статусов во фронтенд используется WebSocket. Он хранит активные соединения по userId и рассылает события конкретному пользователю.

3.1.2.5 Передача статусов и уведомлений через WebSocket

Для отображения хода выполнения задач в реальном времени в системе используется WebSocket. Такой подход позволяет передавать изменения статуса от бэкенд-части веб-приложения к фронтенду без постоянного

повторного запроса к серверу. Это особенно важно для обработки видео и изображений, поскольку выполнение таких задач занимает заметное время, и пользователю нужно видеть, на каком этапе находится его запрос.

После получения события о задаче бэкенд обновляет состояние в PostgreSQL и одновременно отправляет уведомление во фронтенд через WebSocket. В проекте эта логика реализована в `ws-hub.ts` и `task-events.service.ts`. Когда бэкенд получает сообщение о прогрессе, готовности preview или завершении обработки, он не только фиксирует новое состояние задачи в базе данных, но и пересылает это же событие клиенту, чтобы интерфейс сразу отразил изменения.

Ниже показан пример рассылки события на стороне бэкенда:

```
export function broadcastTaskEvent(event: TaskEventDto) {
  const targets = connections.get(event.userId);
  if (!targets) return;

  for (const socket of targets) {
    socket.send(JSON.stringify(event));
  }
}
```

Kafka используется для обмена сообщениями между бэкендом, диспетчером и конвейером обработки, а WebSocket для доставки уже обработанных событий в пользовательский интерфейс. Конвейер сообщает о ходе выполнения задачи через events-топик, бэкенд принимает это сообщение, обновляет данные и затем передаёт уведомление во фронтенд в режиме реального времени.

На стороне фронтенда для подключения к этому каналу используется `realtime-bridge`, который поддерживает WebSocket-соединение с бэкендом. Пользователь видит изменения статуса сразу после их обработки сервером: задача может перейти из состояния `queued` в `processing`, затем в `preview_ready` или `completed`, либо в `failed` при возникновении ошибки.

Вот пример обработки приходящего сообщения:

```
socket.addEventListener('message', (event) => {
  const parsed =
taskEventSchema.safeParse(JSON.parse(event.data));
  if (!parsed.success) return;
```

```

    setLatestTaskEvent(parsed.data);
    if (parsed.data.type === 'task:completed') {
        setLatestCompletedTaskEvent(parsed.data);
    }
  });
});

```

Так, WebSocket используется как канал доставки статусов и уведомлений, а Kafka – способ обмена сообщениями между бэкендом, диспетчером и конвейером обработки. Отсутствие пуллинга снижает нагрузку на сервер.

3.2 Реализация взаимодействия с менеджером сообщений

Apache Kafka в архитектуре проекта используется для взаимодействия между бэкендом, диспетчером задач и конвейером обработки видео. При перегрузках она добавляет задачи в буфер и синхронизирует передачу сообщений между компонентами. В проекте для приоритезации задач есть отдельный модуль диспетчеризации.

После загрузки медиафайла бэкенд сохраняет его в локальное файловое хранилище, создаёт запись в PostgreSQL и формирует сообщение с описанием задачи. Это сообщение публикуется во входной топик Kafka. Диспетчер задач считывает сообщение из входного топика, вычисляет приоритет на основе характеристик файла и публикует задачу уже с приоритетом в выходной топик. Конвейер обработки забирает задачу из выходного топика, выполняет инференс модели и публикует события о ходе выполнения в events-топик. Бэкенд получает эти события, обновляет статус задачи в PostgreSQL и уведомляет пользователя через WebSocket.

Для разделения потоков данных по типам контента и направлениям передачи используются различные топики. В сообщениях передаётся не сам медиафайл, а его описание и параметры обработки. Это снижает нагрузку на менеджер сообщений и позволяет использовать Kafka как механизм передачи служебной информации, а не как файловое хранилище. Бэкенд передаёт во входной топик метаданные файла (длительность, разрешение, кодек, FPS, размер) и параметры обработки. Диспетчер добавляет к этой информации вычисленный приоритет и публикует задачу в выходной топик. Конвейер

публикует в events-топик статус задачи, процент выполнения, пути к результатам или текст ошибки.

Kafka повышает устойчивость и гибкость системы. Бэкенд не зависит от времени выполнения обработки и не блокируется на длительных операциях. Диспетчер и конвейер могут быть масштабированы горизонтально за счёт механизма consumer group. При отказе одного из экземпляров диспетчера или конвейера нагрузка автоматически перераспределяется между оставшимися.

3.2.1 Топики и схема сообщений

Для обмена сообщениями между компонентами системы используются три группы топиков, разделяющие потоки данных по этапам обработки и типу медиафайла.

Топики для передачи задач от бэкенда к диспетчеру:

- fullqualityai.image-input – для задач по обработке изображений,
- fullqualityai.video-input – для задач по обработке видео.

Топик для передачи упорядоченных задач от диспетчера к конвейеру:

- fullqualityai.task-output – единый выходной топик, в который диспетчер публикует задачи с вычисленным приоритетом.

Топики для получения результатов (от конвейера к бэкенду):

- fullqualityai.image-events – для событий о ходе обработки изображений,
- fullqualityai.video-events – для событий о ходе обработки видео.

Разделение входных топиков по типу медиафайла позволяет диспетчеру гибко настраивать правила приоритезации для разных типов контента. Единый выходной топик упрощает подписку конвейера.

Передача задач в Apache Kafka реализована в бэкенд-части веб-приложения. После загрузки файла бэкенд формирует описание задачи (не сам медиафайл) и публикует его во входной топик, соответствующий типу медиафайла: fullqualityai.image-input для изображений и fullqualityai.video-input для видео.

Структура сообщения в input-топике:

```
{
  "taskId": "uuid",
  "mediaId": "uuid",
  "userId": "uuid",
  "kind": "image | video",
  "metadata": {
    "durationSec": 120,
    "resolution": "1920x1080",
    "fps": 30,
    "codec": "h264",
    "bitrate": 50,
    "sizeBytes": 10485760
  },
  "taskDir": "/storage/tasks/uuid",
  "configPath": "/storage/tasks/uuid/config.json"
}
```

Приоритет на этом этапе не вычисляется. Бэкенд передаёт в Kafka только исходные метаданные и параметры обработки. Решение о порядке выполнения задач принимается позже, в диспетчере задач.

Схема передачи:

- пользователь загружает файл через интерфейс,
- бэкенд сохраняет файл в хранилище,
- бэкенд создаёт запись о задаче в PostgreSQL,
- бэкенд формирует сообщение с метаданными,
- бэкенд публикует сообщение во входной топик Kafka.

На этом этапе Kafka выступает как надёжный транспортный слой и буфер между бэкендом и диспетчером. Она гарантирует, что сообщение не будет потеряно, даже если диспетчер временно недоступен. При этом Kafka не вмешивается в логику приоритезации и не изменяет порядок сообщений – это полностью ответственность диспетчера.

После вычисления приоритета диспетчер задач публикует задачу в выходной топик `fullqualityai.task-output`. Это единый топик для всех типов медиафайлов (изображений и видео). Диспетчер получает задачу из input-топика, анализирует её характеристики, вычисляет её приоритет по формуле и формирует обновлённую структуру для передачи в выходной топик. В этой структуре дополнительно появляется блок `priority`, содержащий числовую

оценку приоритета, его уровень и ключ маршрутизации. Данный ключ используется для группировки задач в дальнейшем потоке обработки и позволяет конвейеру получать задачи в порядке, определённом диспетчером.

Структура сообщения в output-топике (диспетчер добавляет вычисленный приоритет):

```
{
  "taskId": "uuid",
  "mediaId": "uuid",
  "userId": "uuid",
  "kind": "image | video",
  "priority": {
    "score": 0.35,
    "level": "medium",
    "routingKey": "video:medium:medium:mid"
  },
  "taskDir": "/storage/tasks/uuid",
  "configPath": "/storage/tasks/uuid/config.json"
}
```

Конвейер обработки подписан на этот топик и забирает задачи по мере освобождения вычислительных ресурсов. Приоритет, добавленный диспетчером, используется конвейером для определения очередности обработки.

Схема передачи:

- диспетчер получает задачу из input-топика,
- диспетчер вычисляет приоритет,
- диспетчер публикует задачу с приоритетом в output-топик,
- конвейер забирает задачу из output-топика и начинает обработку.

Output-топик является связующим звеном между диспетчером и конвейером, обеспечивая передачу уже упорядоченных задач.

После завершения обработки медиафайла конвейер публикует в Kafka сообщения о ходе выполнения и результате. Для этого используются топики событий `fullqualityai.image-events` и `fullqualityai.video-events`, разделённые по типу медиафайла. Бэкенд подписывается на эти топики и получает сообщения о прогрессе, готовности preview-версии, успешном завершении или ошибке.

Конвейер формирует события следующих типов:

- task:progress – обновление прогресса выполнения,
- task:preview-ready – готовность preview-версии,
- task:completed – успешное завершение обработки,
- task:failed – ошибка при обработке.

Каждое событие содержит идентификаторы задачи, медиафайла и пользователя, процент выполнения, текущую стадию, статус, а при необходимости ещё и путь к результату или текст ошибки.

Структура сообщения в events-топике:

```
{
  "type": "task:progress | task:preview-ready |
task:completed | task:failed",
  "taskId": "uuid",
  "mediaId": "uuid",
  "userId": "uuid",
  "progress": 45,
  "stage": "Decoding",
  "status": "processing",
  "resultPath": "/storage/results/uuid/result.mp4",
  "errorMessage": null
}
```

При получении сообщения бэкенд определяет его тип и обновляет состояние задачи в PostgreSQL. Для task:progress задача переводится в статус processing, для task:preview-ready – в preview_ready, для task:completed – в completed, для task:failed – в failed. При успешном завершении сохраняются пути к preview-версии и финальному результату. Эти события используются бэкендом для обновления записи в базе данных и передачи актуального состояния пользователю через WebSocket.

Также одновременно изменяются записи в таблицах EnhanceTask и Media. Это гарантирует согласованность данных даже при повторной доставке сообщений.

Схема получения результатов:

- конвейер публикует событие,
- Kafka доставляет сообщение в бэкенд,
- бэкенд определяет тип события и обновляет статус задачи,
- изменения фиксируются в PostgreSQL,

– бэкенд отправляет обновлённое состояние во фронтенд через WebSocket.

Так, Kafka завершает асинхронный цикл обработки, обеспечивая надёжную доставку событий от конвейера обратно в бэкенд.

На рисунке 38 видно, как устроена работа с топиками сообщений в проекте.

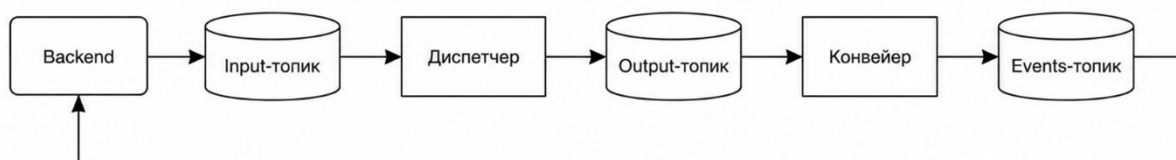


Рисунок 38 – Полный поток сообщений

Исходное описание задачи поступает от бэкенда к диспетчеру. Далее идёт передача задачи с выданным приоритетом от диспетчера к конвейеру обработки. А потом обратно в бэкенд происходит возврат событий о ходе выполнения.

3.2.2 Повторная доставка сообщений и обработка отказов

Kafka работает по схеме с возможностью повторного получения сообщений. Из этого логично, что обработка задач должна спокойно переносить повторную доставку, чтобы сбои на каком-нибудь шаге не приводили к потере данных или изменению целостности состояний.

Сообщение не считается обработанным успешно в случаях, если при его чтении или в процессе обработки случается ошибка. Kafka имеет возможность повторно передать сообщение, если консьюмер закончил работу со сбоем или просто перезагрузился. То есть задачи хранятся в очереди до момента их выполнения по факту.

Перед стартом работы конвейер и диспетчер проверяют не была ли задача уже выполнена. Они ищут файлы результата и отметки об успешном завершении. Повторное выполнение пропускается, если задача ранее обработана. Это убирает риск ошибочного повторного исполнения и появления дубликатов.

В Kafka тоже предусмотрена обработка ошибок. Если менеджер сообщений временно недоступен или возникла ошибка при отправке сообщения, система фиксирует это в журнале и может повторить публикацию либо использовать fallback-механику.

Такая работа с Kafka позволяет использовать асинхронную архитектуру без риска потери данных и делает систему устойчивой к сбоям и перезапускам отдельных компонентов.

3.3 Реализация диспетчера задач

После завершения проектирования архитектурных и алгоритмических подходов для оценки приоритета задач необходимо перейти к реализации диспетчера задач для веб-приложения. Из заключения пункта 2.4 для диспетчера был выбран модифицированный алгоритм приоритетной диспетчеризации с механизмом запланированного устаревания задач. Формула оценки приоритета для исполнения в реализации была выведена в пункте 1.3. Реализация будет выполнена на языке Python [16] из-за предыдущего опыта работы с данным языком, высокой читаемости кода, а также упрощённого построения взаимодействия с бэкендом и очередью сообщений.

Перед непосредственной разработкой диспетчера задач нужно определить структуру внутреннего устройства. Это позволит упростить понимание механизмов работы конечного решения, а также облегчить разработку и дальнейшую поддержку существующего кода. Структура модуля диспетчера задач (рисунок 39) будет выглядеть следующим образом:

- точка входа диспетчера. Инициализирует подключение к очереди сообщений, запускает цикл ожидания и обработки входящих запросов от бэкенда. Обрабатывает возможные необработанные исключения из других модулей и сигналы прерывания от системы,

- вычислитель приоритета или планировщик задач. Реализует формулу расчёта приоритета на основе метаданных видеофайла и результатов дообучения нейронной сети. Распараллеливает вычисления по доступным потокам для внутренней оптимизации и повышения скорости оценки,

– механизм запланированного устаревания. Реализует логику пересчёта приоритета для задач, длительное время ожидающих в очереди. Помещается в отдельный циклический поток с таймаутом срабатывания,

– аудитор метрик. Реализует сбор данных по оценке видеоконтента. Является необходимым отладочным инструментом для анализа эффективности текущего состояния диспетчера.

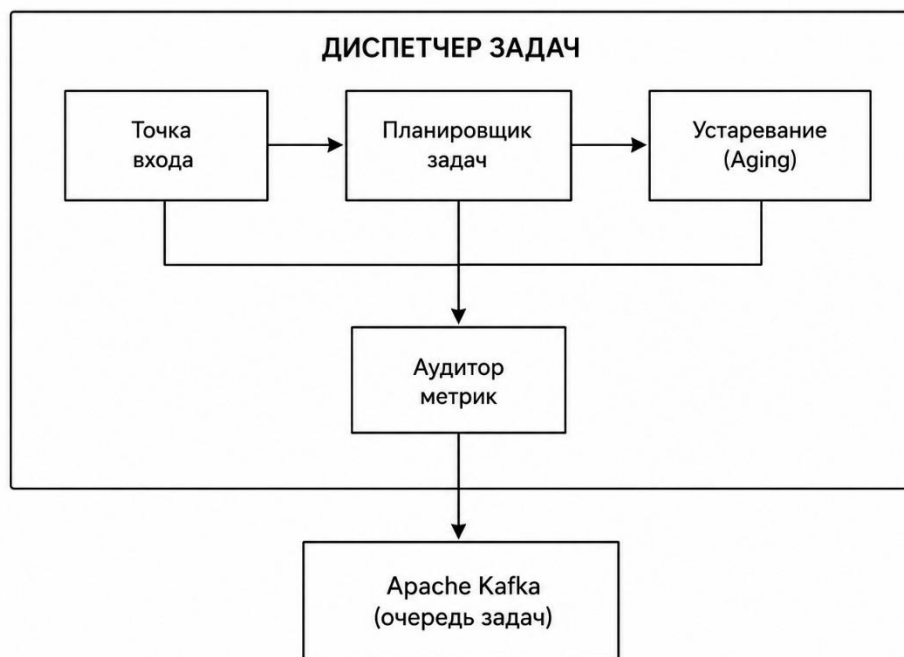


Рисунок 39 – Схема взаимодействия моделей структуры

Модули представленной структуры должны входить в отдельный python-пакет `task_dispatcher`, который будет являться конечной реализацией диспетчера. Все зависимости интегрируемых пакетов должны быть прописаны заранее для использования корректных версий в файле `requirements.txt`.

3.3.1 Точка входа диспетчера

Модуль точки входа является ведущим компонентом диспетчера, координирующим жизненный цикл всех остальных модулей. Его ответственность охватывает инициализацию подключения к очереди сообщений, запуск цикла ожидания и обработки входящих запросов от бэкенда, централизованную обработку исключений из подчинённых

модулей, обработку сигналов прерывания от операционной системы и координацию запуска и остановки фоновых компонентов.

При запуске диспетчер загружает параметры из переменных окружения. Конфигурация включает адрес менеджера сообщений, учётные данные, название очереди, тайм-ауты соединения и параметры повторных попыток. Ниже приведён фрагмент, реализующий загрузку конфигурации и формирование структуры параметров подключения:

```
class DispatcherApp:
    """Точка входа диспетчера задач."""

    def __init__(self, config: Optional[DispatcherConfig] =
None) -> None:
        self.config = config or load_config()
        self._consumer: Optional[Consumer] = None
        self._producer: Optional[Producer] = None
        self._consuming: bool = False
        self._shutdown_requested: bool = False

        # Инициализация подчинённых модулей
        self.scheduler = PriorityScheduler()
        self.aging_mechanism: Optional[AgingMechanism] =
None

        self.metrics_auditor =
MetricsAuditor(port=self.config.metrics_port)

        # Потоки фоновых компонентов
        self._aging_thread: Optional[threading.Thread] =
None

    def connect(self) -> None:
        """Инициализирует потребителя и производителя
Kafka."""
        self._ensure_topics_exist()
        self._create_consumer()
        self._create_producer()
        logger.info(
            "Подключение к Kafka установлено (bootstrap=%s,
group=%s)",
            self.config.kafka_bootstrap_servers,
            self.config.kafka_group_id,
        )
```

Диспетчер выступает в роли потребителя для входного топика, куда бэкенд направляет запросы, и производителя для выходного топика, откуда обработчики видео извлекают упорядоченные по коэффициенту приоритета

задачи. Класс `DispatcherApp` инкапсулирует логику управления клиентами `Apache Kafka`:

```
def _create_consumer(self) -> None:
    """Создаёт и конфигурирует потребителя Kafka."""
    consumer_config = {
        "bootstrap.servers":
self.config.kafka_bootstrap_servers,
        "group.id": self.config.kafka_group_id,
        "auto.offset.reset": "earliest",
        "enable.auto.commit": False,
        "max.poll.interval.ms":
self.config.kafka_max_poll_interval_ms,
        "session.timeout.ms":
self.config.kafka_session_timeout_ms,
    }
    self._consumer = Consumer(consumer_config)
    self._consumer.subscribe([self.config.kafka_input_to
pic])
    logger.info(
        "Потребитель Kafka подписан на топик '%s'",
        self.config.kafka_input_topic,
    )

def _create_producer(self) -> None:
    """Создаёт и конфигурирует производителя Kafka."""
    producer_config = {
        "bootstrap.servers":
self.config.kafka_bootstrap_servers,
        "acks": "all",
        "retries": self.config.kafka_max_retries,
        "retry.backoff.ms":
self.config.kafka_retry_backoff_ms,
        "enable.idempotence": True,
    }
    self._producer = Producer(producer_config)
    logger.info("Производитель Kafka инициализирован")
```

В `Apache Kafka` используется модель опроса `poll`. Точка входа запускает бесконечный цикл, в котором периодически опрашивает менеджер на предмет новых сообщений во входном топике. Каждый входящий запрос содержит метаданные загруженного и провалидированного видеофайла в формате `JSON`. Модуль извлекает полученные данные, десериализует их и передаёт в планировщик для вычисления приоритета, после чего результат публикуется в выходной топик:

```
def _create_producer(self) -> None:
    """Создаёт и конфигурирует производителя Kafka."""
    producer_config = {
```

```

        "bootstrap.servers":
self.config.kafka_bootstrap_servers,
        "acks": "all",
        "retries": self.config.kafka_max_retries,
        "retry.backoff.ms":
self.config.kafka_retry_backoff_ms,
        "enable.idempotence": True,
    }
    self._producer = Producer(producer_config)
    logger.info("Производитель Kafka инициализирован")

```

Было учтено, что Apache Kafka не предоставляет встроенного механизма приоритетных очередей и сообщения в партиции хранятся строго в порядке их публикации. В связи с этим приоритет передаётся в виде пользовательского заголовка сообщения, который извлекается обработчиком. Обработчик при получении сообщения извлекает значение заголовка и определяет наивысший приоритет в очереди. Данный подход позволяет сохранить единый выходной топик, не усложняя архитектуру множеством параллельных топиков под каждый уровень приоритета, и при этом обеспечивает передачу всей необходимой информации для последующей приоритезации.

Точка входа также реализует многоуровневую стратегию обработки ошибок. Ошибки десериализации JSON приводят к логированию и подтверждению смещения, чтобы некорректное сообщение не блокировало обработку последующих. Исключения, возникающие в планировщике, обрабатываются аналогично ошибкам десериализации: задача отклоняется, смещение подтверждается. Такая стратегия гарантирует, что ошибка в обработке одной задачи не блокирует партицию и не влияет на работоспособность диспетчера в целом.

Модуль регистрирует обработчики сигналов SIGTERM и SIGINT, поступающих от операционной системы при штатной остановке диспетчера. При получении сигнала реализуется процедура корректного завершения, обеспечивающая сохранность всех обрабатываемых задач:

```

def _register_signal_handlers(self) -> None:
    """Регистрирует обработчики сигналов ОС."""
    signal.signal(signal.SIGTERM, self._signal_handler)
    signal.signal(signal.SIGINT, self._signal_handler)

```

```

        logger.debug("Обработчики сигналов SIGTERM и SIGINT
зарегистрированы")

    def _signal_handler(self, signum: int, frame) -> None:
        """Обработчик сигналов прерывания."""
        sig_name = signal.Signals(signum).name
        logger.info("Получен сигнал %s. Иницируется
остановка...", sig_name)
        self._shutdown_requested = True
        self._consuming = False

```

Модуль точки входа реализует все необходимые функции для управления жизненным циклом диспетчера. Централизованная обработка исключений и механизм корректного завершения гарантируют, что ни одна задача, находящаяся в процессе обработки, не будет потеряна, а фоновые компоненты будут остановлены согласованно и без нарушения целостности данных.

3.3.2 Планировщик задач

Модуль планировщика является ядром диспетчера. В нём реализована логика расчёта приоритета задачи на основе метаданных видеофайла и данных, полученных в процессе дообучения нейронной сети. Планировщик получает от точки входа словарь метаданных загруженного видеофайла и возвращает целочисленное значение приоритета, которое впоследствии используется обработчиками для определения очередности обслуживания задач. Этот показатель характеризует ожидаемую ресурсоёмкость обработки данного видеофайла нейронной сетью. Ответственность планировщика строго ограничена вычислением приоритета. Он не участвует в валидации файлов, не хранит пользовательские данные и не управляет ходом последующей обработки, что соответствует принципу единой ответственности и позволяет независимо развивать модель приоритизации, не затрагивая смежные компоненты системы.

Планировщик, реализуемый в модуле scheduler.py использует класс PriorityScheduler, который инкапсулирует всю логику вычисления приоритета и предоставляет единственный публичный метод compute_priority. Внутренняя структура класса включает:

- метод `_compute_priority_stub`. Реализует формулу приоритета, заданную в пункте 1.3;
- метод `_validate_metadata`. Проверяет корректность входных данных;
- пул потоков для распараллеливания вычислений при пиковых нагрузках.

```
def compute_priority(self, metadata: Dict[str, Any]) -> int:
    self._validate_metadata(metadata)
    priority = self._compute_priority_stub(metadata)
    logger.debug(
        "Приоритет вычислен: %d (duration=%s,
complexity=%s)",
        priority,
        metadata.get("duration"),
        metadata.get("complexity_score"),
    )
    return priority

def compute_priority_async(self, metadata: Dict[str,
Any]) -> Future:
    return self._executor.submit(self.compute_priority,
metadata)
```

Перед вычислением приоритета планировщик проверяет входные метаданные. Метод `_validate_metadata` выполняет следующие последовательные проверки:

- полученные метаданные являются словарем,
- присутствуют значения во всех полях словаря,
- значения имеют корректные типы данных,
- значения находятся в корректных диапазонах.

Некорректные данные приводят к генерации внутреннего исключения `InvalidMetadataError`, которое перехватывается точкой входа и приводит к отклонению задачи с уведомлением бэкенда. Такой защитный слой предотвращает попадание в выходной топик заведомо некорректных задач.

Метод `_compute_priority_stub` реализует формулу оценки приоритета, описанную на этапе исследования в пункте 1.3. Приоритет вычисляется как взвешенная сумма нормированных параметров метаданных, после чего масштабируется в заданный диапазон. Логика инверсии соответствует проектному решению в пункте 2.4, согласно которому короткие и лёгкие

задачи должны обрабатываться раньше длинных и вычислительно сложных. Конкретные значения весовых коэффициентов и нормировочных констант вынесены в параметры конструктора класса, что позволяет менять их без модификации кода.

В условиях пиковой нагрузки, когда множество пользователей одновременно загружают видео на обработку, бэкенд может направлять диспетчеру значительный поток запросов в короткий промежуток времени. Для решения этой проблемы планировщик использует пул потоков `ThreadPoolExecutor`. Метод `compute_priority_async` принимает метаданные и возвращает объект `Future`, позволяя точке входа не блокироваться на время вычисления. Размер пула потоков конфигурируется через параметр `max_workers`.

Метод `compute_priority` реализован как чистая функция. Он не имеет побочных эффектов, не зависит от глобального состояния и детерминированно возвращает одинаковый результат для идентичных входных данных. Это свойство принципиально важно для внутренней архитектуры диспетчера, где любой экземпляр сервиса вычислит одинаковый приоритет для одной и той же задачи. Класс `PriorityScheduler` является неизменяемым в отношении своего конфигурационного состояния, что дополнительно упрощает параллельное использование одного экземпляра из нескольких потоков.

3.3.3 Механизм запланированного устаревания

Описываемый механизм реализует логику защиты от заикленности приоритетов. В этой ситуации, низкоприоритетные задачи неограниченно долго откладываются из-за непрерывного потока более приоритетных задач. Без этого механизма приоритетная диспетчеризация, не гарантирует обслуживание каждой задачи за конечное время. Механизм устаревания устраняет данный недостаток путём постепенного повышения приоритета задач, длительное время ожидающих обработки.

Данный механизм реализован как независимый циклический поток, который запускается точкой входа при старте диспетчера и работает в

фоновом режиме. Модуль выполняет свою функцию периодически, с фиксированным тайм-аутом срабатывания, значение которого задаётся в конфигурации. Такой подход позволяет отделить логику устаревания от основного цикла обработки запросов. Класс `AgingMechanism` инкапсулирует всю логику устаревания и предоставляет методы для запуска, остановки и контроля состояния фонового потока:

```
class AgingMechanism:
    """Механизм запланированного устаревания приоритета
    задач."""

    # Значения по умолчанию
    DEFAULT_AGING_INTERVAL: int = 60          # интервал
срабатывания, сек.
    DEFAULT_AGING_THRESHOLD: int = 120        # порог
ожидания до старения, сек.
    DEFAULT_AGING_RATE: float = 0.1           # прирост
приоритета в секунду
    DEFAULT_PROMOTION_THRESHOLD: int = 8      # порог
продвижения задачи

    def __init__(
        self,
        consumer: Consumer,
        producer: Producer,
        output_topic: str,
        interval: int = DEFAULT_AGING_INTERVAL,
        aging_threshold: int = DEFAULT_AGING_THRESHOLD,
        aging_rate: float = DEFAULT_AGING_RATE,
        promotion_threshold: int =
DEFAULT_PROMOTION_THRESHOLD,
        metrics_auditor: Optional[Any] = None,
    ) -> None:
        self._consumer = consumer
        self._producer = producer
        self._output_topic = output_topic
        self._interval = interval
        self._aging_threshold = aging_threshold
        self._aging_rate = aging_rate
        self._promotion_threshold = promotion_threshold
        self._metrics_auditor = metrics_auditor

        self._running: bool = False
        self._thread: Optional[threading.Thread] = None
        self._lock = threading.Lock()
        self._aged_tasks_count: int = 0

        logger.info(
            "AgingMechanism инициализирован (interval=%d
сек., "
```

```

        "threshold=%d сек., rate=%.2f, promotion=%d)",
        self._interval,
        self._aging_threshold,
        self._aging_rate,
        self._promotion_threshold,
    )

```

Каждая итерация цикла устаревания включает три последовательных этапа:

- метод `_fetch_waiting_tasks` формирует список задач, ожидающих обработки в выходном топике. Выполняется анализ отставания доступных задач на обработку видео,

- из полученного списка отбираются задачи-кандидаты, время ожидания которых превышает пороговое значение `aging_threshold`. Порог вводится для того, чтобы задачи, опубликованные недавно, не начинали немедленно конкурировать за продвижение с действительно длительно ожидающими задачами. Это предотвращает неоправданный лавинообразный пересчёт приоритетов,

- вычисляется новое значение приоритета по формуле старения:

$$P_{new} = P_{original} + aging_rate \times (waiting_time - aging_threshold),$$

где $P_{original}$ – исходный приоритет задачи;

$aging_rate$ – конфигурируемый коэффициент скорости устаревания;

$waiting_time$ – полное время ожидания задачи в очереди;

$aging_threshold$ – пороговое время, до истечения которого старение не применяется.

Если пересчитанный приоритет достигает или превышает порог продвижения `promotion_threshold`, задача принудительно перепубликуется в выходной топик с обновлённым значением приоритета в заголовке.

3.3.4 Аудитор метрик

Аудитор метрик реализует сбор и экспорт количественных показателей работы диспетчера. Он является отладочным инструментом, позволяющим разработчику анализировать эффективность текущей конфигурации диспетчера и корректировать параметры модели приоритизации. Служит

источником данных для оперативного мониторинга состояния системы в промышленной эксплуатации.

Аудитор реализован классом `MetricsAuditor`, который инкапсулирует регистрацию метрик, их обновление и экспорт через файловую систему. Класс не зависит от других модулей диспетчера и взаимодействует с ними исключительно через вызовы публичных методов, что позволяет тестировать его изолированно. При старте диспетчера точка входа создаёт экземпляр аудитора, фиксирует или создает файл для хранения метрик и передает ссылку на экземпляр `MetricsAuditor` в планировщик и механизм устаревания, которые обновляют метрики в процессе работы:

```
class MetricsAuditor:
    """Аудитор метрик диспетчера задач с сохранением в
    текстовый файл."""

    # Квантование уровней приоритета по диапазонам
    PRIORITY_BUCKETS: Dict[str, tuple] = {
        "low": (0, 3),
        "medium": (4, 6),
        "high": (7, 8),
        "critical": (9, 10),
    }

    # Интервал записи метрик в файл по умолчанию (секунды)
    DEFAULT_FLUSH_INTERVAL: float = 30.0

    def __init__(
        self,
        file_path: str = "metrics/dispatcher_metrics.txt",
        flush_interval: float = DEFAULT_FLUSH_INTERVAL,
        max_waiting_time_samples: int = 10000,
        max_duration_samples: int = 5000,
        max_resolution_samples: int = 5000,
        max_complexity_samples: int = 5000,
        max_computation_time_samples: int = 5000,
    ) -> None:
        self._file_path = file_path
        self._flush_interval = flush_interval
        self._lock = threading.Lock()

        # Ограничения на количество хранимых замеров
        self._max_waiting_time_samples =
max_waiting_time_samples
        self._max_duration_samples = max_duration_samples
        self._max_resolution_samples =
max_resolution_samples
```

```

        self._max_complexity_samples =
max_complexity_samples
        self._max_computation_time_samples =
max_computation_time_samples

        # Счётчики задач по уровням приоритета
self._tasks_by_priority: Dict[str, int] = {
    "low": 0,
    "medium": 0,
    "high": 0,
    "critical": 0,
}

        # Счётчик задач, затронутых механизмом устаревания
self._aged_tasks_count: int = 0

        # Текущая глубина очереди
self._queue_depth: int = 0
self._queue_topic: str = "video_processing_tasks"

        # Замеры времени ожидания задачи в очереди (в
секундах)
self._waiting_times: Dict[str, List[float]] = {
    "low": [],
    "medium": [],
    "high": [],
    "critical": [],
}

        # Замеры характеристик видеоконтента
self._video_durations: List[float] = []
self._video_resolutions: List[float] = []
self._video_complexities: List[float] = []

        # Замеры времени вычисления приоритета
self._computation_times: List[float] = []

        # Счётчики ошибок по типам
self._errors: Dict[str, int] = {}

        # Временная метка запуска аудитора
self._start_time: float = time.time()

        # Поток периодической записи в файл
self._running: bool = False
self._flush_thread: Optional[threading.Thread] =

None

        # Создание директории для файла метрик, если её нет
file_dir = os.path.dirname(self._file_path)
if file_dir:
    os.makedirs(file_dir, exist_ok=True)

        logger.info(

```

```

        "MetricsAuditor инициализирован (file_path='%s',
flush_interval=%.1f сек.)",
        self._file_path,
        self._flush_interval,
    )

```

Метод `_format_metrics` формирует текстовое представление метрик, сгруппированное по разделам. Каждый раздел включает как абсолютные значения счётчиков, так и вычисляемые аналитические значения. Файл полностью перезаписывается при каждой операции `flush`, поэтому его содержимое всегда отражает актуальное состояние метрик на момент последней записи.

Поскольку диспетчер может работать непрерывно в течение длительного времени, неограниченное накопление замеров привело бы к утечке памяти. Для предотвращения этой проблемы все списки замеров имеют ограничение на максимальное количество хранимых элементов. При превышении лимита старые значения отбрасываются, и сохраняются только последние `N` элементов. Такой подход обеспечивает постоянное потребление памяти вне зависимости от длительности работы диспетчера.

3.4 Взаимодействие компонентов

В разрабатываемой системе взаимодействие между пользовательским интерфейсом, бэкенд-частью, Apache Kafka, диспетчером задач, конвейером обработки и файловым хранилищем организовано как асинхронный конвейер обработки медиафайлов. Такой подход позволяет отделить действия пользователя от выполнения вычислительно тяжёлой операции и обеспечить устойчивую работу приложения при одновременной обработке нескольких задач. Пользователь взаимодействует только с фронтенд-интерфейсом, а все дальнейшие действия, связанные с сохранением файла, постановкой задачи в очередь, её диспетчеризацией и обработкой результата, выполняются внутри серверного контура системы.

После загрузки медиафайла фронтенд отправляет запрос в бэкенд, где файл принимается, сохраняется в локальное файловое хранилище `storage` и сопровождается записью в PostgreSQL. На этом же этапе бэкенд формирует структуру задачи, содержащую параметры обработки, пути к исходному

файлу, файлу конфигурации и результатам. Для изображений и видео используются разные потоки передачи задач, поэтому бэкенд определяет тип медиафайла и публикует сообщение в соответствующий входной топик Kafka. Это позволяет не блокировать пользовательский запрос на время выполнения обработки, а быстро перевести задачу в асинхронный режим выполнения.

Далее бэкенд передаёт задачу в Apache Kafka. В этой схеме Kafka выступает как промежуточный слой маршрутизации сообщений между бэкендом, диспетчером задач и конвейером обработки. В очередь отправляется не сам медиафайл, а только служебная информация о задаче: идентификатор задачи, идентификатор медиафайла, тип файла, метаданные, путь к конфигурационному файлу и путь к рабочему каталогу. Диспетчер задач получает это сообщение из входного топика, анализирует характеристики файла, вычисляет приоритет и формирует упорядоченный поток задач для конвейера обработки. Крупные файлы остаются на уровне хранилища storage.

В процессе обработки конвейер читает исходный файл из хранилища, затем создаёт промежуточные и итоговые файлы и сохраняет результат в storage. В зависимости от режима работы это может быть либо финальный обработанный файл, либо предпросмотр. В конце обработки конвейером публикуется сообщение об этом обратно в систему, бэкенд же обновляет статус в БД и передаёт новое состояние задачи во фронт. Так, storage используется как единое файловое пространство для бэкенда и конвейера обработки, а Kafka как средство передачи задач и событий между ними.

Последовательность взаимодействия компонентов в системе можно представить следующим образом:

- пользователь загружает медиафайл через фронтенд,
- фронтенд отправляет файл в бэкенд,
- бэкенд сохраняет файл в хранилище и создаёт запись о задаче в PostgreSQL,

- бэкенд формирует конфигурационный пакет задачи,
- бэкенд публикует сообщение в Kafka,
- диспетчер задач получает сообщение из Kafka, вычисляет приоритет и формирует порядок обработки,
- конвейер обработки получает задачу из Kafka и начинает выполнение,
- конвейер читает и записывает файлы в storage,
- бэкенд получает событие о результате и обновляет состояние задачи,
- фронтенд отображает пользователю текущий статус и готовый результат.

Такая архитектура не создаёт жёстких зависимостей между компонентами, позволяя каждому работать над своей задачей независимо от других. Пользователь работает с удобным интерфейсом, бэкенд управляет задачами и метаданными, Kafka отвечает за передачу сообщений, диспетчер формирует порядок выполнения, конвейер обработки выполняет медиа-операции, а storage используется для хранения исходных и результирующих файлов. В результате система становится более устойчивой, масштабируемой и пригодной для дальнейшего развития.

4 Оценка эффективности реализованной системы

Результат реализации веб-приложения по спроектированной архитектуре требует экспериментального подтверждения корректности и эффективности принятых архитектурных решений. Теоретический анализ эффективности архитектуры должен быть обоснован и верифицирован в условиях, приближенных к реальной эксплуатации. Необходимо также провести тестирование реализованного веб-интерфейса на предмет удобства, работоспособности и отказоустойчивости. Только такая оценка ключевых показателей поможет сформировать вывод о достижении предложенной архитектурой поставленных целей.

Исследование было проведено на тестовом стенде со следующими вычислительными ресурсами:

- CPU: Intel Xeon E5-2699 v5 x2,
- RAM: DDR5 128 GB,
- GPU: NVIDIA 5070 Ti x2,
- SSD: KingSpec 1 TB,
- HDD: Toshiba 5 TB,
- ОС: Ubuntu 24.04 Noble Numbat.

Для создания тестовой нагрузки была использована выборка видеофайлов, отличная от присутствующей в обучении.

4.1 Исследование метрик диспетчера задач

Для оценки эффективности реализованного диспетчера задач с механизмом запланированного устаревания и проверки корректности работы выбранного алгоритма и формулы для оценки приоритета задач улучшения качества было проведено экспериментальное исследование. В ходе исследования был осуществлен сбор метрик, экспортируемых аудитором диспетчера задач, в условиях тестовой нагрузки, имитирующей реальные сценарии использования веб-приложения.

4.1.1 Методика проведения эксперимента

Отправка видеоконтента производилась при помощи библиотеки Playwright и скрипта на Node.js для периодической отправки запросов через веб-форму:

```
const { chromium } = require('playwright');
const fs = require('fs');
const path = require('path');

const URL = 'http://192.168.0.102:20617';
const VIDEOS_DIRECTORY = '/home/user/vid_test/*';
const MIN_DELAY = 1;
const MAX_DELAY = 30;

function getRandomDelay(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min) *
1000;
}

function getRandomVideo(directory) {
  const files = fs.readdirSync(directory).filter(file => {
    const ext = path.extname(file).toLowerCase();
    return ['.mp4', '.webm', '.mov', '.avi',
'.mkv'].includes(ext);
  });

  if (files.length === 0) {
    throw new Error(`В директории ${directory} нет
видеофайлов`);
  }

  const randomFile = files[Math.floor(Math.random() *
files.length)];
  return path.join(directory, randomFile);
}

async function runPeriodicUpload() {
  const browser = await chromium.launch({ headless: false
});

  const context = await browser.newContext();
  const page = await context.newPage();

  console.log(`Директория с видео: ${VIDEOS_DIRECTORY}`);
  console.log(`Диапазон задержки: ${MIN_DELAY}-${MAX_DELAY}
сек.`);

  while (true) {
    try {
      const videoPath = getRandomVideo(VIDEOS_DIRECTORY);
      const delay = getRandomDelay(MIN_DELAY, MAX_DELAY);
```

```

        console.log(`[${new Date().toLocaleTimeString()}]
Отправка: ${path.basename(videoPath)}`);

        await page.goto(URL);

        const fileInput = page.locator('input[type="file"]');
        await fileInput.setInputFiles(videoPath);

        await page.locator('#upload-button').click();

        console.log(`Видео отправлено. Ожидание ${delay /
1000} сек...`);

        } catch (error) {
            console.error('Произошла ошибка:', error.message);
            const delay = getRandomDelay(MIN_DELAY, MAX_DELAY);
        } finally {
            const waitTime = getRandomDelay(MIN_DELAY, MAX_DELAY);
            await new Promise(resolve => setTimeout(resolve,
waitTime));
        }
    }
}
runPeriodicUpload();

```

Таймаут ожидания перед отправкой выбирается в диапазоне от 1 до 30 секунд. При попытках тестирования с большей верхней границей времени система не накапливала сообщения из-за доступных вычислительных ресурсов. Данный генератор формировал задачи из тестовой директории `/home/user/vid_test/*`, где присутствовали видео с различными параметрами и форматами, которые входили в допустимую выборку из таблицы 1. Если сгруппировать их по длительности, то можно выделить три группы:

- короткие видео. Длительность 30 – 120 секунд, показатель сложности 0 – 3 ~ 50% от общего потока задач,
- средние видео. Длительность 120 – 600 секунд, показатель сложности 3 – 7 ~ 20% от общего потока,
- длинные видео. Длительность 600 – 3 600 секунд, показатель сложности 7 – 10 ~ 30% от общего потока.

Процентное распределение типов основано на актуальных в данный момент запросах пользователя. Короткие ролики обладают большим спросом на обработку из-за актуальности клипов и коротких роликов. Средние видео обрабатывают реже всего из-за нетипичного временного диапазона. Длинные

видео востребованы сразу после коротких, так как одна из основных целей пользователя в данном веб-приложении – это реставрация старых фильмов, архивов или сериалов.

Эксперимент проводился в три этапа, каждый длительностью 60 минут:

- диспетчер отключен, очередь с конвейером обработки работают в режиме FIFO,

- диспетчер работает, но отключен механизм старения задач,

- стандартная работа диспетчера.

- Во время тестирования производилась фиксация метрик с интервалом в одну минуту, что позволило получить 60 замеров на каждом из этапов.

4.1.2 Оценка времени ожидания задач

Ключевым показателем эффективности диспетчера задач является время ожидания задачи в очереди от момента публикации до момента извлечения обработчиком. В таблице 19 представлены сравнительные данные по времени ожидания для трёх этапов эксперимента. На рисунке 40 показана гистограмма сравнения времени ожидания задач в очереди для трёх режимов работы диспетчера.

Таблица 19 – Время ожидания задачи в очереди

Показатель	FIFO	Приоритетный (без старения)	Приоритетный (со старением)
Среднее время ожидания, сек.	385,2	172,4	156,8
Медианное время ожидания, сек.	210,5	45,2	42,1
P95 времени ожидания, сек.	1 240,8	2 450,3	890,4
Максимальное время ожидания, сек.	1 820,5	4 320,0	1 650,2
Стандартное отклонение, сек.	412,7	685,3	378,1



Рисунок 40 – Время ожидания задачи в очереди

Алгоритм FIFO демонстрирует худшее среднее и медианное время ожидания. Короткие задачи долго ожидают завершения длинных, попавших в очередь раньше. P95 для FIFO достигает 20.7 минут, а это негативно сказывается на взаимодействии с приложением.

Медианное время ожидания сильно улучшается при использовании приоритетного алгоритма без механизма старения. Но резко ухудшаются важные показатели. Например, максимальное время ожидания достигает 72 минуты, а P95 уже почти 41 минуту. То есть надолго откладываются низкоприоритетные задачи при непрерывном потоке более приоритетных задач. Система ведёт себя непредсказуемо, стандартное отклонение возрастает, происходит увеличение разброса времени ожидания.

А вот уже полноценный диспетчер, у которого присутствует механизм запланированного устаревания, демонстрирует наилучшие результаты по всем показателям. Среднее время ожидания снижено: на 9% по сравнению с приоритетным алгоритмом без старения, а по сравнению с FIFO снижено на 59%. Самое значимое – это снижение P95 времени ожидания до 14,8 минут, что в 2,7 раза лучше, чем у приоритетного алгоритма без механизма старения. Максимальное время ожидания ограничено 27,5 минутами, что

подтверждает эффективность механизма старения в предотвращении бесконечного откладывания задач. Стандартное отклонение также снижено, что указывает на более предсказуемое поведение системы.

4.1.3 Оценка распределения задач по уровням приоритета

Распределение задач по уровням приоритета можно корректно проанализировать только в третьем эксперименте. Результат распределения для 60 минут штатной работы отражен в таблице 20. А на рисунке 41 показана гистограмма распределения задач по уровням приоритета.

Таблица 20 – Распределение задач по уровням приоритета

Уровень приоритета	Количество задач	Доля, %	Преобладающий тип задач
Низкий (0–3)	62	10,3	С (длинные видео)
Средний (4–6)	174	29,0	В (средние видео)
Высокий (7–8)	211	35,2	А (короткие видео)
Критический (9–10)	153	25,5	А (очень короткие видео)

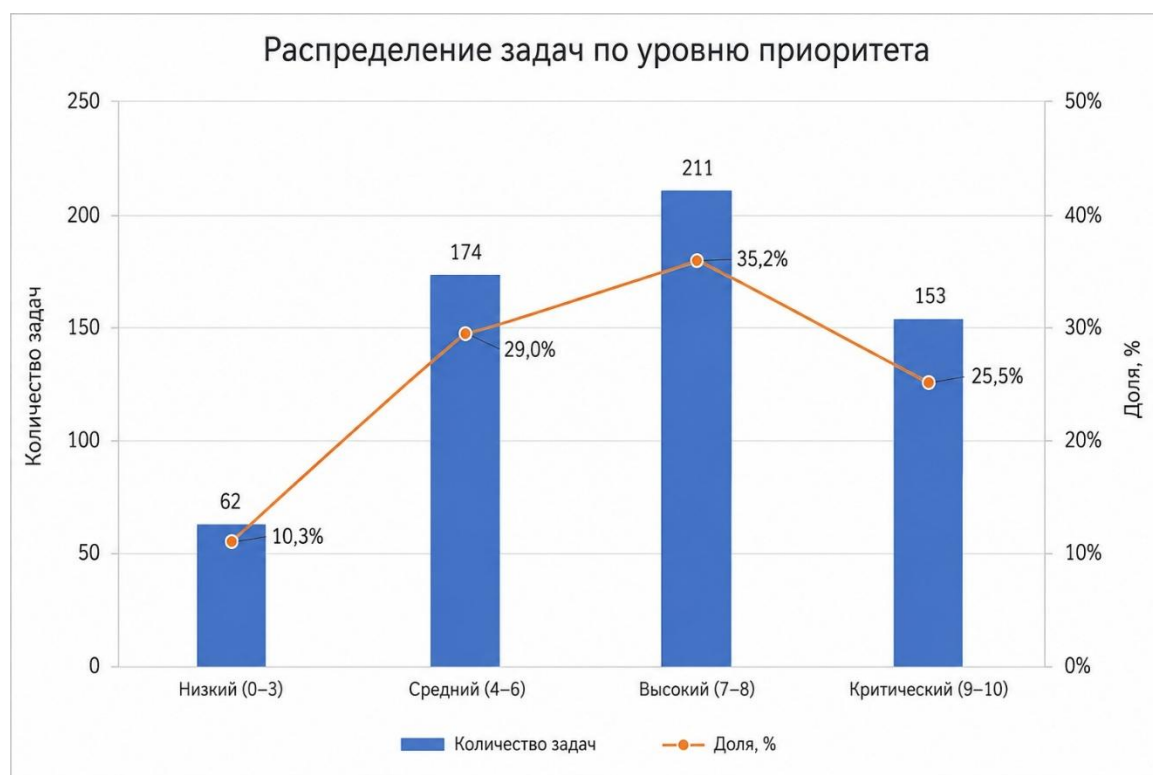


Рисунок 41 – Распределение задач по уровню приоритета

Анализ метрик показывает, что реализованная формула приоритета успешно группирует задачи по их вычислительной сложности. Длинные видео концентрируются в низком диапазоне приоритетов, в то время как

короткие видео получают максимальный приоритет обработки. Средние видео расположены в плавающем среднем диапазоне. Также в этой таблице есть погрешность в виде 2 – 3% от каждой категории из-за присутствия механизма старения.

Данные показания свидетельствуют о корректной работе формулы приоритета. Более ресурсоёмкие задачи получают более низкий приоритет, что обеспечивает приоритетную обработку коротких видео и предотвращает их блокировку длинными.

4.1.4 Оценка глубины очереди

Глубина выходной очереди является одним из самых значимых показателей эффективности работы реализованного алгоритма диспетчеризации. Лучше всего эффективность алгоритма заметна на показателе максимальной глубины. Данная характеристика хорошо отражает количество ожидающих задач в самый нагруженный момент времени для каждого из трех экспериментов. Оценка глубины очереди представлена в таблице 21. Для трёх режимов работы диспетчера на рисунке 42 показано сравнение глубины очереди.

Таблица 21 – Оценка глубины очереди

Показатель	FIFO	Приоритетный (без старения)	Приоритетный (со старением)
Средняя глубина очереди	24,8	18,3	16,5
Максимальная глубина очереди	52	61	38
Минимальная глубина очереди	4	0	1
Стандартное отклонение	12,4	16,8	9,2

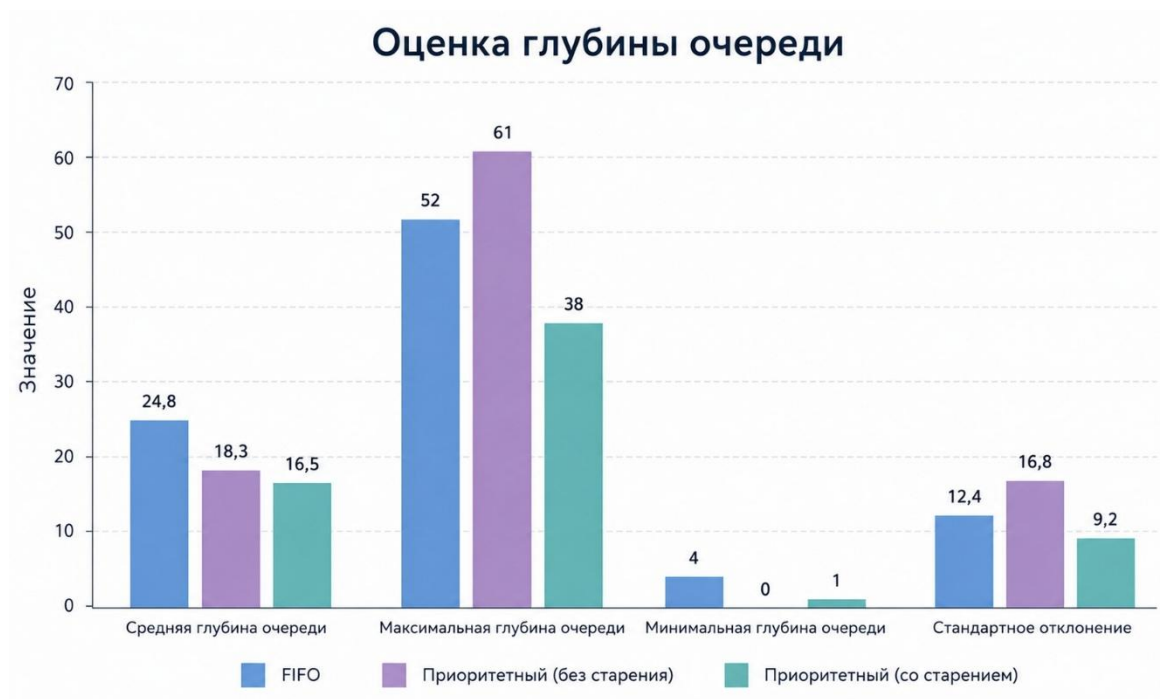


Рисунок 42 – Оценка глубины очереди

Оценка метрик показывает, что приоритетная диспетчеризация со старением обеспечивает наименьшую среднюю глубину очереди. Это объясняется тем, что короткие задачи обрабатываются быстрее и не накапливаются в очереди, ожидая завершения длинных. Максимальная глубина очереди при использовании полноценного диспетчера так же наименьшая, что свидетельствует о лучшей утилизации вычислительных ресурсов обработчиков.

У приоритетного алгоритма без старения наблюдается наибольшая максимальная глубина очереди и стандартное отклонение. Это объясняется накоплением низкоприоритетных задач, которые не могут быть обработаны из-за непрерывного потока высокоприоритетных. Механизм старения устраняет этот эффект, обеспечивая своевременное продвижение застоявшихся задач и предотвращая неконтролируемый рост очереди.

4.2 Результаты работы реализованного веб-приложения

Разработанное веб-приложение реализует полный цикл обработки видео и изображений, включающий следующие основные сценарии: авторизация и регистрация пользователя, загрузка файла и настройка параметров обработки, отслеживание прогресса выполнения задачи,

получение и скачивание результата, просмотр публичной и личной галереи. Далее будет продемонстрированы все основные сценарии взаимодействия и возможности веб-приложения.

Главная страница в светлой и тёмной темах показана на рисунках 43 и 44 соответственно. Посередине страницы расположены два больших интерактивных блока «Изображение» и «Видео». При нажатии на каждый происходит переход на страницу с улучшением качества для изображений или для видео.

Общий хедер содержит следующие навигационные ссылки:

- логотип сайта (кликабельный, ведёт на главную страницу /),
- Home ведёт на главную страницу /,
- About ведёт на /about (страница о сервисе),
- Gallery ведёт на /gallery (публичная галерея примеров),
- My Gallery ведёт на /gallery/my (только для авторизованных пользователей),
- Progress Widget (показывается только при активной задаче),
- Log In / Profile (меняется в зависимости от статуса).

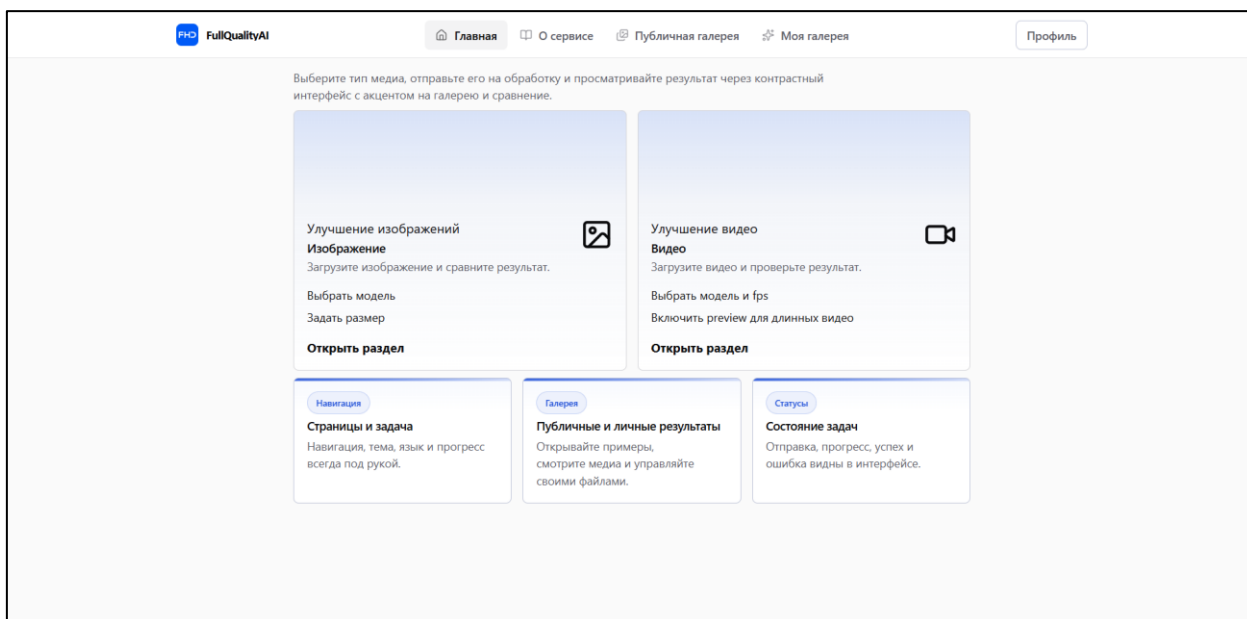


Рисунок 43 – Главная страница

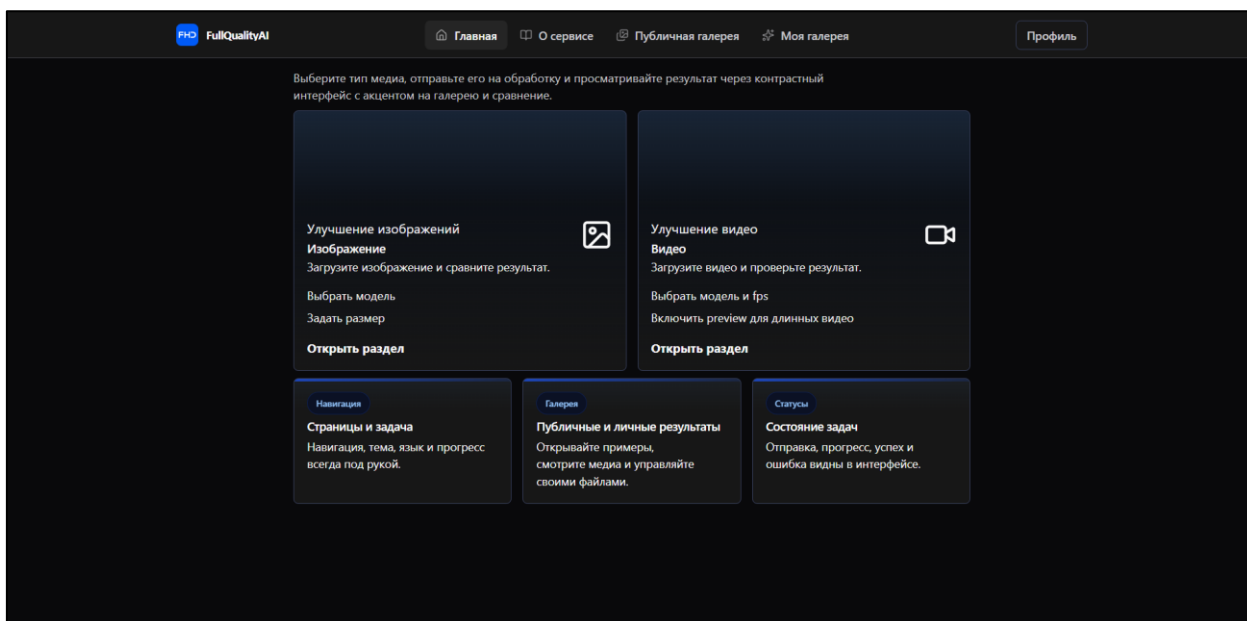


Рисунок 44 – Тёмная тема

Пользователь имеет возможность авторизации в сервисе (рисунок 45). На ней реализована форма входа с полями «Email» и «Пароль», кнопкой входа и ссылкой на страницу регистрации (рисунок 46), а также обработка ошибок ввода и проверки корректности данных.

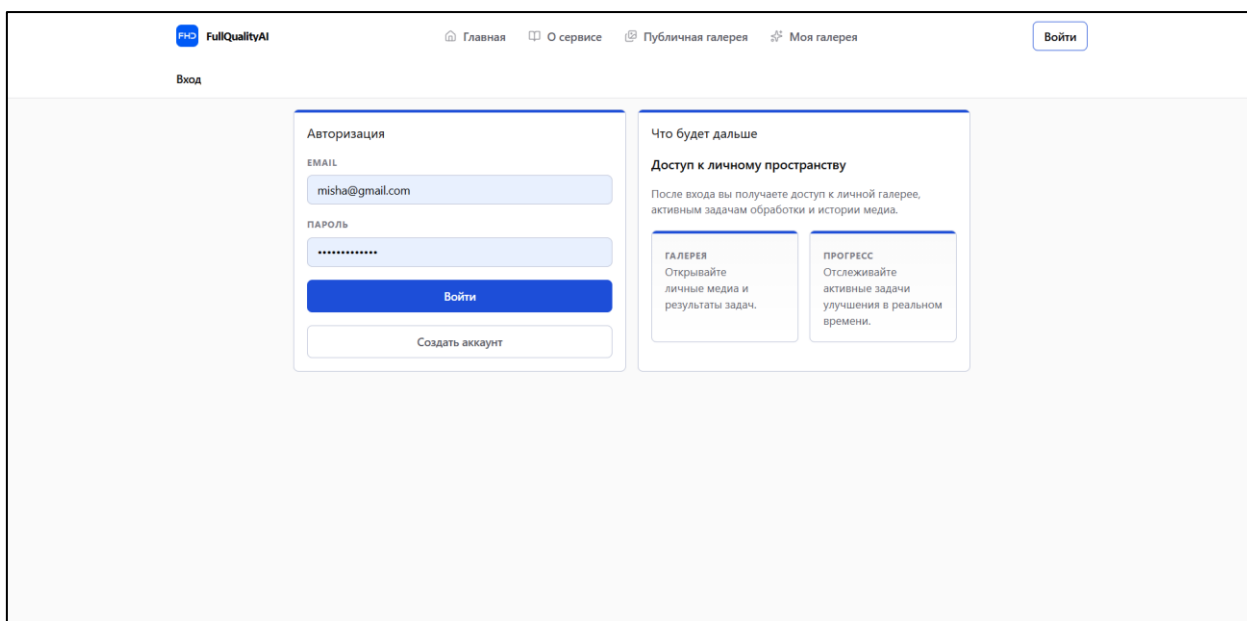


Рисунок 45 – Страница авторизации

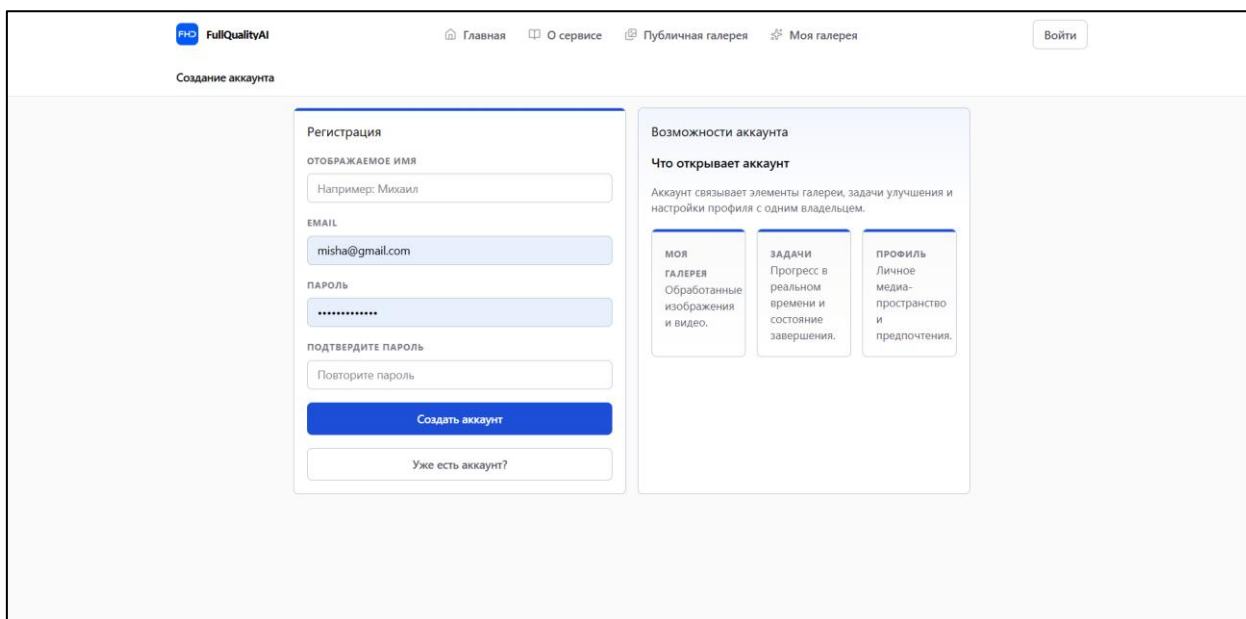


Рисунок 46 – Страница создания аккаунта

После успешного входа пользователю становятся доступны возможности веб-приложения, такие как: улучшение медиа, личная галерея и управление профилем.

Страница профиля показана на рисунке 47. Она содержит информацию о пользователе, а также возможность смены пароля.

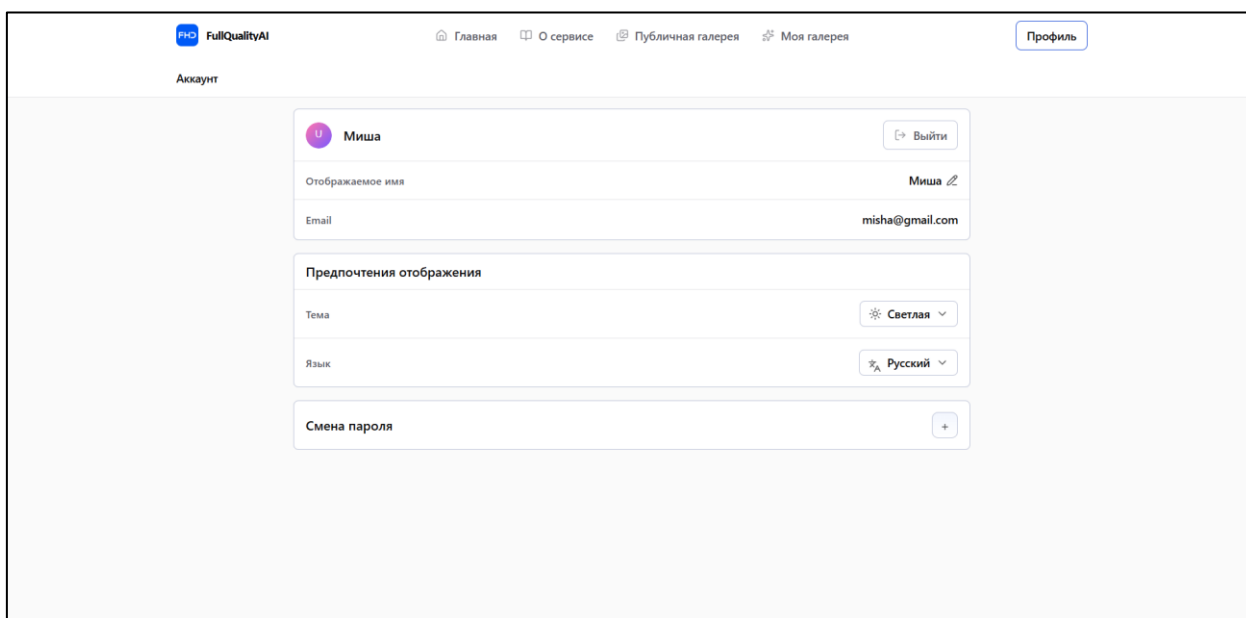


Рисунок 47 – Страница профиля

Основной функционал приложения сосредоточен на страницах обработки изображений и видео. На каждой из страниц улучшения медиа в зоне расположения медиафайла есть интерактивный слайдер «до/после», а в секции слева – поле с настройками.

Страница улучшения изображений (рисунок 48) предоставляет пользователю возможность загрузить файл, выбрать параметры обработки и отправить задачу на сервер.

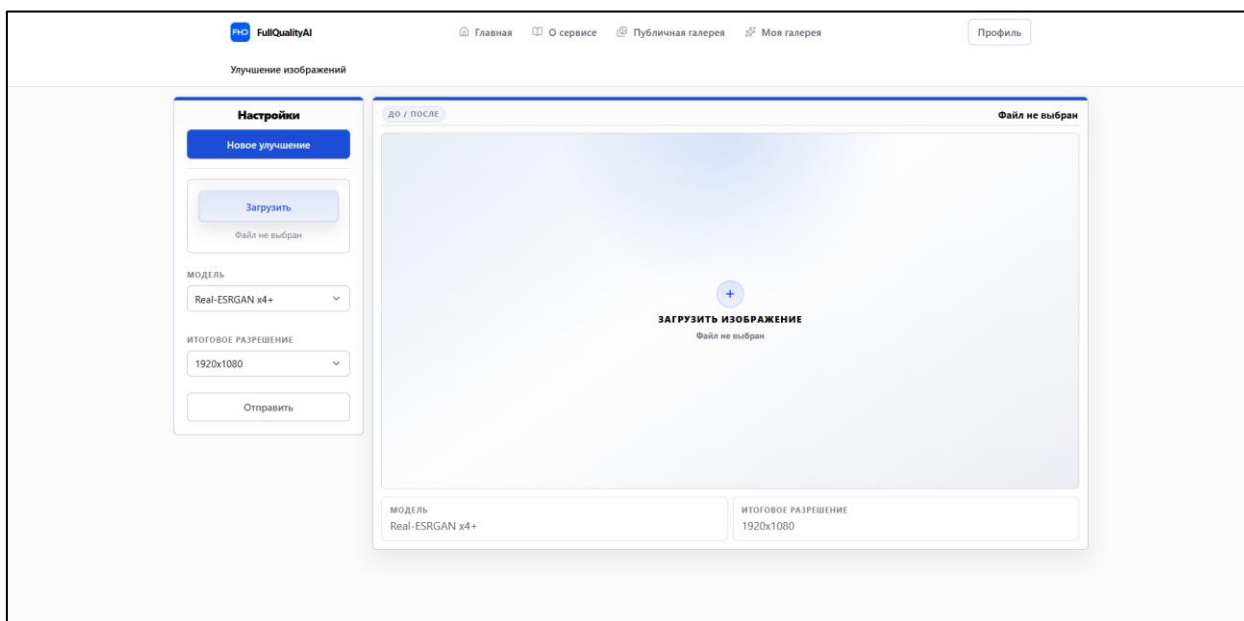


Рисунок 48 – Страница улучшения изображений

Страница улучшения видео устроена аналогично, но дополнительно включает параметры, связанные с предпросмотром (режимом частичной обработки первых 15 минут видео). Это позволяет пользователю сначала получить короткий preview-результат, а затем, при необходимости, запустить полную обработку. Режим предпросмотра выбирается пользователем самостоятельно (рисунок 49).

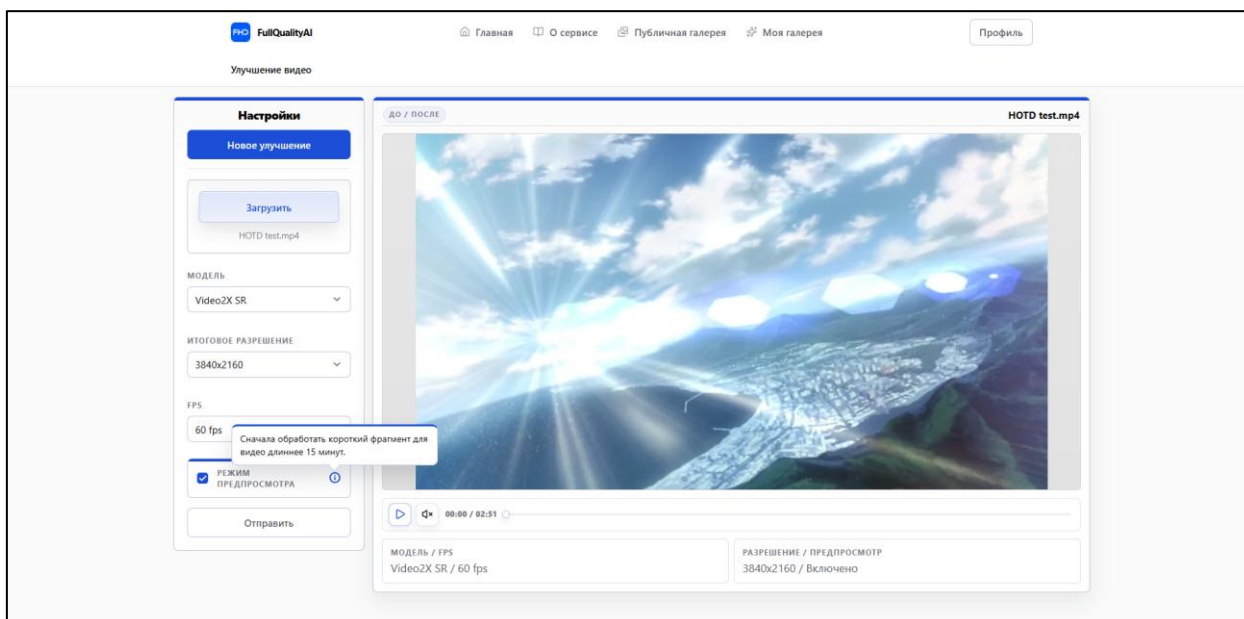


Рисунок 49 – Опция режима предпросмотра

Пользователь в режиме реального времени видит всплывающие оповещения о состоянии задачи.

Результат улучшения видео на странице показан на рисунке 50.

В итоге обработки появляется слайдер, видео разделено на «до» и «после». Оба видео при нажатии на кнопку play воспроизводятся одновременно.

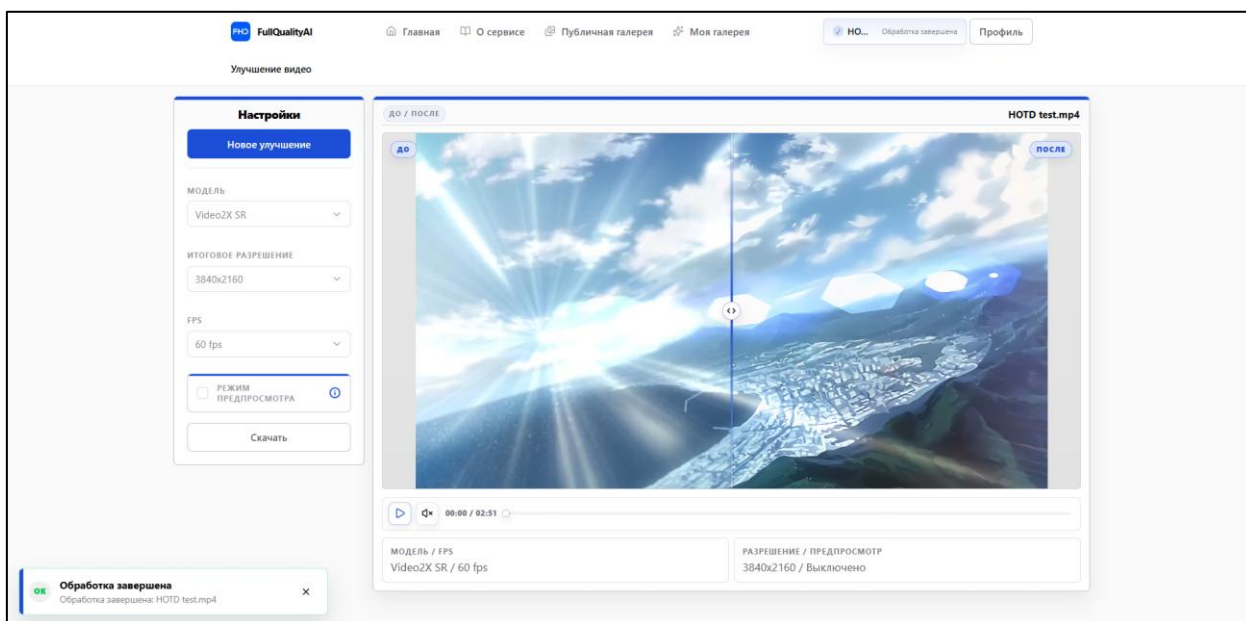


Рисунок 50 – Результат улучшения видео

После обработки задачи пользователь может проверить результат в личной галерее или на самой странице обработки. Также пользователь должен видеть активную задачу в специальном виджете состояния задач (рисунок 51), который находится в хедере веб-приложения и является отдельным элементом интерфейса. Он отображает актуальный статус обработки в реальном времени и позволяет пользователю видеть, на каком этапе находится отправленный файл. В зависимости от состояния задача (или несколько задач) может находиться в очереди, выполняться, быть завершённой либо завершиться с ошибкой.

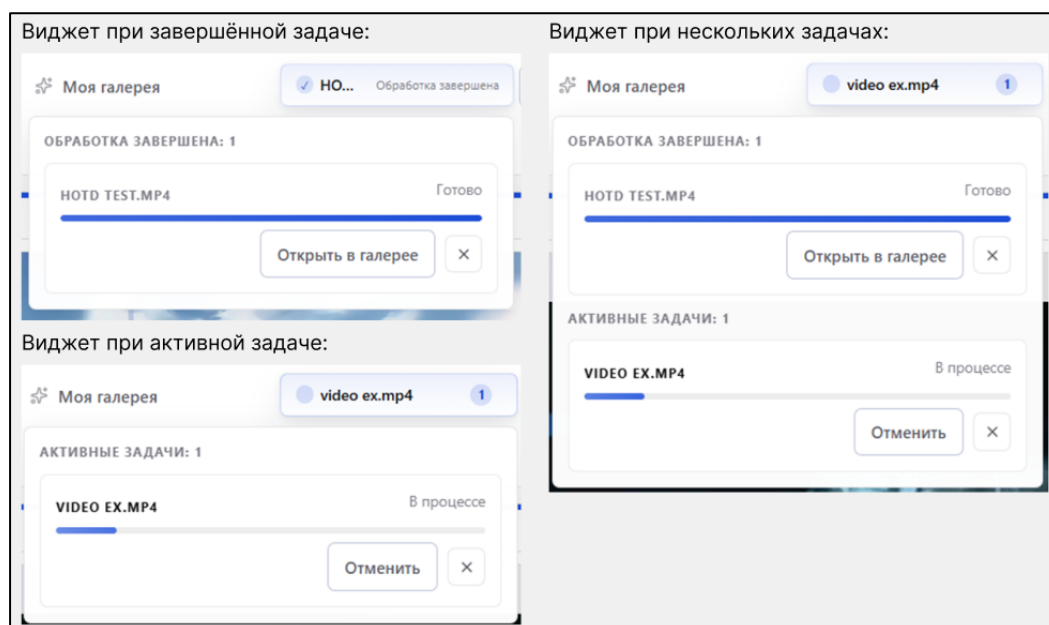


Рисунок 51 – Виджет состояния задач

При клике на название задачи в виджете во время обработки происходит переход на соответствующую страницу улучшения конкретного медиа. Также через виджет можно отменить обработку задачи. Если обработка завершена, то внутри виджета при нажатии на название задачи произойдёт переход уже на страницу личной галереи, где сразу же откроется модальное окно с подробной информацией о медиафайле.

Для авторизованных пользователей реализована личная галерея (рисунок 52) ранее улучшенных видео, изображений со статистикой. В галерее можно выставить фильтры: «Все», «Фото», «Видео», «Избранное».

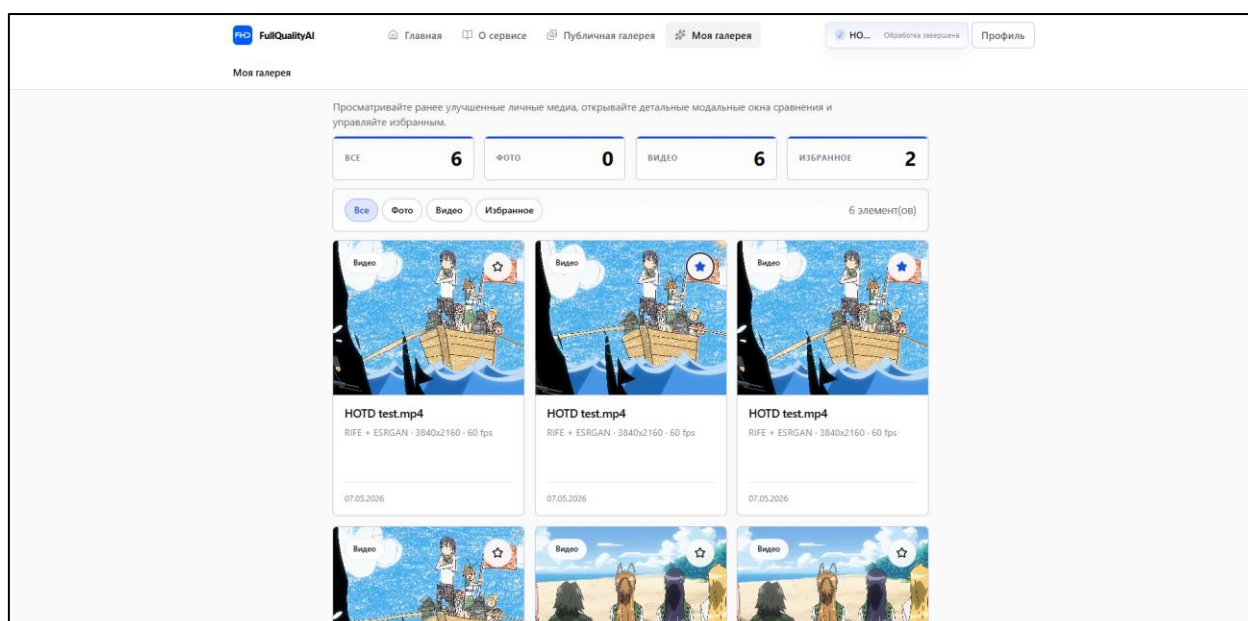


Рисунок 52 – Макет страницы личной галереи

Если пользователь перешёл на страницу личной галереи, кликает на какую-нибудь карточку, медиафайл открывается в модальном окне (рисунок 53). В нём можно увидеть слайдер «до/после», параметры результата, основные кнопки «Скачать результат», «Удалить», «Добавить в избранное», «Сделать публичным/приватным».

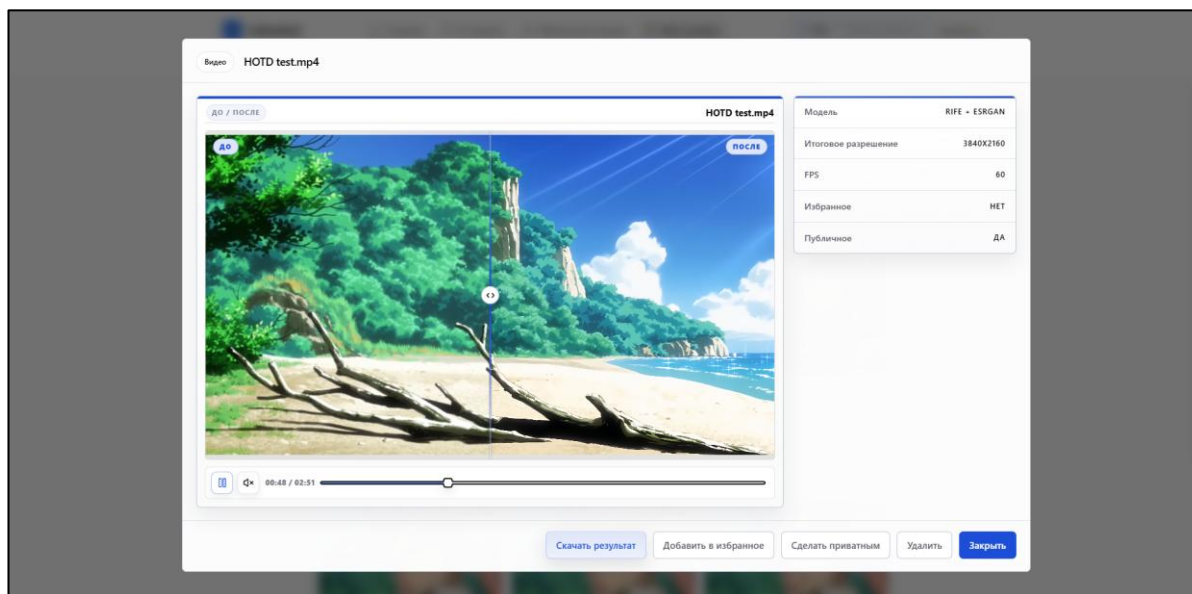


Рисунок 53 – Модальное окно открытого медиафайла личной галереи

Публичная же галерея предназначена для демонстрации примеров уже обработанных медиафайлов и служит витриной возможностей сервиса. На этой странице пользователь может ознакомиться с результатами обработки медиа других пользователей без необходимости авторизации.

Публичная галерея показана на рисунке 54.

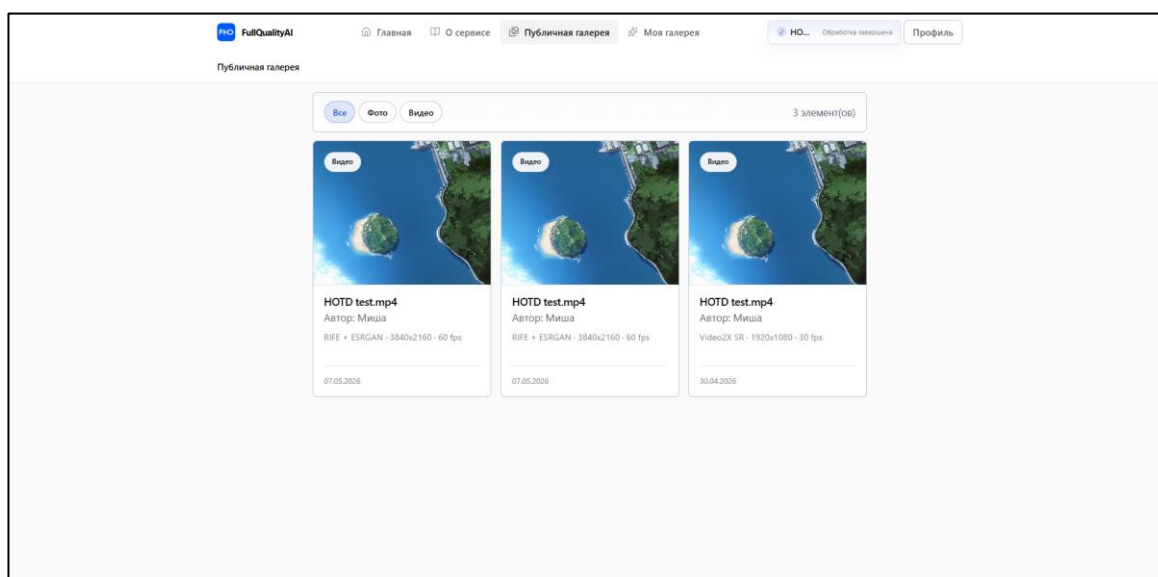


Рисунок 54 – Страница публичной галереи

При клике на карточку открывается такое же модальное окно с большим слайдером «до/после», описанием и возможностью скачивания обработанного медиа (рисунок 55).

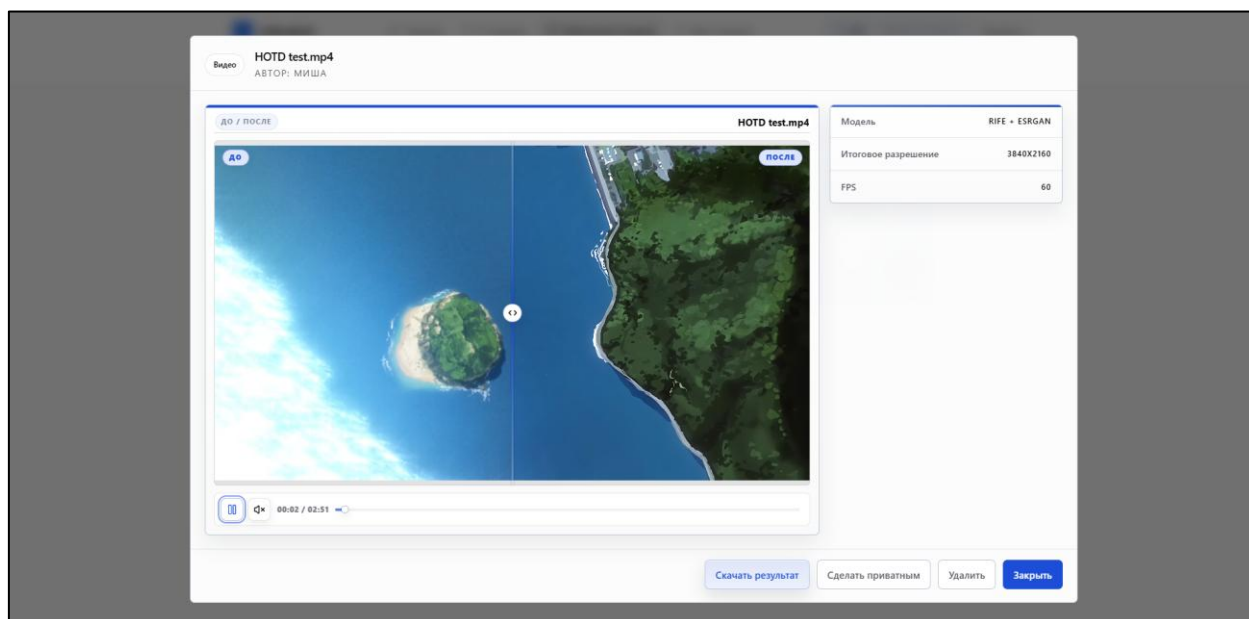


Рисунок 55 – Модальное окно открытого медиафайла публичной галереи

Страница «О сервисе» показана на рисунке 56. Она содержит описание сервиса.

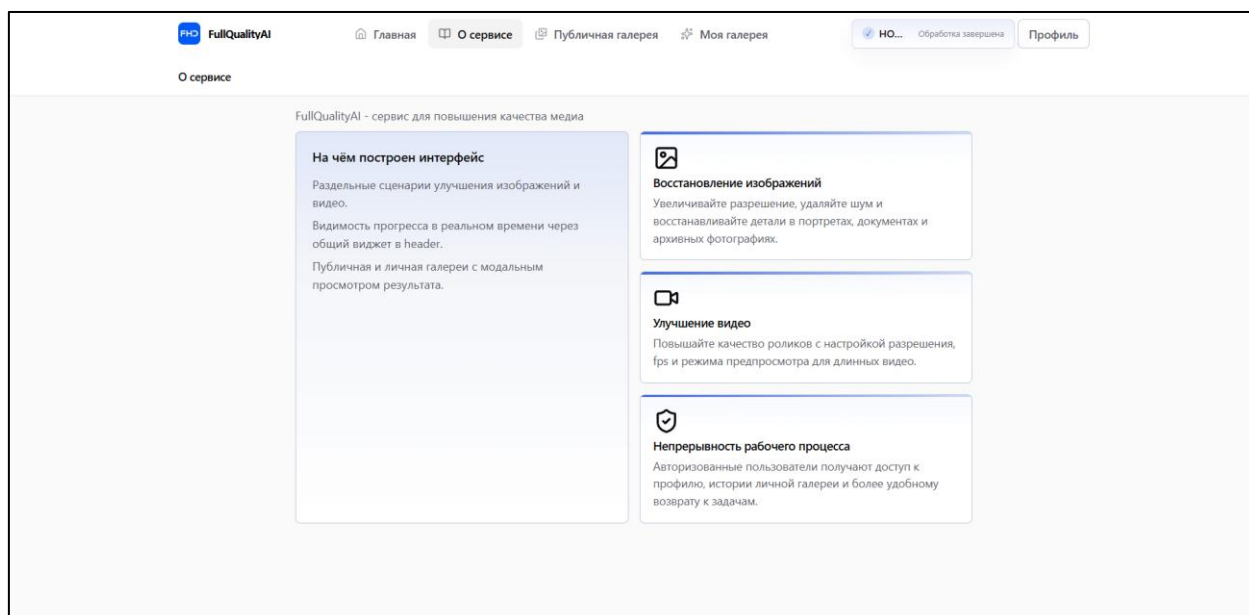


Рисунок 56 – Страница «О сервисе»

ЗАКЛЮЧЕНИЕ

В ходе выполнения магистерской диссертации была достигнута поставленная цель: спроектированы, реализованы и оценены архитектурные решения веб-приложения для улучшения качества видео с использованием ранее разработанного конвейера обработки медиафайлов.

Обоснована актуальность разработки собственного веб-приложения с асинхронной обработкой и выделенным диспетчером задач.

Экспериментально подтверждено, что время обработки видео GAN-моделью зависит от параметров этого видео: от длительности, FPS, разрешения, кодека и битрейта. На основе полученных данных выведена формула оценки приоритета, учитывающая выявленные зависимости. Для дообучения модели был собран и структурирован датасет из кадров советской и зарубежной мультипликации. Выбранный ручной метод сбора данных обеспечил репрезентативность и сбалансированность выборки.

Проведён сравнительный анализ алгоритмов диспетчеризации: FIFO, Round Robin, приоритетной диспетчеризации и WFQ. Для разрабатываемой системы выбран алгоритм приоритетной диспетчеризации с механизмом запланированного устаревания, который показал наилучшие результаты по среднему времени ожидания и стабильности работы при пиковых нагрузках.

Разработано веб-приложение, включающее фронтенд на React и TypeScript, бэкенд на Node.js и Express.js с трёхслойной архитектурой. Apache Kafka используется в качестве транспортного слоя для асинхронной передачи сообщений между компонентами. В рамках работы реализован отдельный диспетчер задач на Python, который вычисляет приоритет на основе характеристик файла и данных дообучения нейронной сети. Диспетчер включает точку входа, планировщик с формулой оценки приоритета, механизм запланированного устаревания для предотвращения бесконечного откладывания низкоприоритетных задач и аудитор метрик для сбора статистики.

Проведённые эксперименты подтвердили эффективность разработанного диспетчера с механизмом старения. Приоритетная

диспетчеризация со старением показала снижение среднего времени ожидания на 59% по сравнению с FIFO (с 385,2 до 156,8 секунд). Медианное время ожидания улучшилось с 210,5 до 42,1 секунды. Система ведёт себя более предсказуемо. Так как зафиксировано в исследовании, что максимальная глубина очереди сократилась на 27%, а показатель стандартного отклонения снизился с 412,7 до 378,1. В моём исследовании подтверждена необходимость применения механизма старения, т.к. без него алгоритм показывает нестабильную работу, где максимальное время ожидания увеличивается до 4320 секунд.

Пользовательский опыт улучшается при приоритезации задач с механизмом старения, происходит это без снижения общей производительности.

Кроме того, в ходе исследования выяснилось, что использование менеджера сообщений Apache Kafka позволяет выровнять нагрузку и исключить потерю задач при пиковых запросах. Пользовательский опыт улучшается при приоритезации задач с механизмом старения, происходит это без снижения общей производительности. Чтобы эффективно спрогнозировать время обработки, необходимо провести учёт характеристик файлов на основе данных, которые были собраны при дообучении модели, а также успешно высчитать коэффициент приоритета задач.

Новизну данного научного исследования составляет предложенный подход к диспетчеризации задач для GAN-обработки видео, который учитывает характеристики файла и использует данные дообучения нейросетевой модели. Разработанное веб-приложение и диспетчер задач применимы для коммерческих сервисов улучшения качества видео, для ML-систем с асинхронной обработкой тяжёлых моделей. Полученные результаты подтверждают эффективность предложенной архитектуры и её полезность в реальных веб-сервисах улучшения качества видео.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Goodfellow, I., et al. Generative Adversarial Nets / I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio // Advances in Neural Information Processing Systems. – 2014. – Vol. 27. – P. 2672–2680.
2. Dong, C., et al. Image Super-Resolution Using Deep Convolutional Networks / C. Dong, C.C. Loy, K. He, X. Tang // IEEE Transactions on Pattern Analysis and Machine Intelligence. – 2016. – Vol. 38, No. 2. – P. 295–307.
3. Wang, X., et al. Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data / X. Wang, L. Xie, C. Dong, Y. Shan // arXiv:2107.10833. – 2021.
4. Официальная документация FFmpeg. [Электронный ресурс] / Режим доступа: <https://ffmpeg.org/documentation.html>, свободный (дата обращения: 05.03.2026).
5. Ньюмен С. Создание микросервисов / С. Ньюмен. – 2-е изд. – Санкт-Петербург : Питер, 2022. – 672 с.
6. Фаулер М. Архитектура корпоративных программных приложений / М. Фаулер. – Москва : Вильямс, 2022. – 544 с.
7. Официальная документация Node.js. [Электронный ресурс] / Режим доступа: <https://nodejs.org/en/docs/>, свободный (дата обращения: 27.03.2026).
8. Официальная документация Express.js. [Электронный ресурс] / Режим доступа: <https://expressjs.com/>, свободный (дата обращения: 15.04.2026).
9. Официальная документация Prisma. [Электронный ресурс] / Режим доступа: <https://www.prisma.io/docs>, свободный (дата обращения: 15.04.2026).
10. Официальная документация RabbitMQ. [Электронный ресурс] / Режим доступа: <https://www.rabbitmq.com/documentation.html>, свободный (дата обращения: 19.04.2026).
11. Официальная документация NATS. [Электронный ресурс] / Режим доступа: <https://docs.nats.io/>, свободный (дата обращения: 22.04.2026).

12. Официальная документация Apache Kafka. [Электронный ресурс] / Режим доступа: <https://kafka.apache.org/documentation/>, свободный (дата обращения: 25.04.2026).
13. Gaujal, B., Navet, N., & Walsh, C. Shortest-path algorithms for real-time scheduling of FIFO tasks with minimal energy use / B. Gaujal, N. Navet, C. Walsh // ACM Transactions on Embedded Computing Systems (TECS). – 2005. – Vol. 4, No. 4. – P. 907–933.
14. Load Balancing Algorithms: Round Robin. Developer.baidu.com [Электронный ресурс] / Режим доступа: <https://developer.baidu.com/article/detail.html?id=3835777> (дата обращения: 05.05.2026).
15. Bennett, J. C. R., & Zhang, H. WF²Q: worst-case fair weighted fair queueing / J. C. R. Bennett, H. Zhang // Proceedings of IEEE INFOCOM '96. Conference on Computer Communications. – 1996. – Vol. 1. – P. 120–128.
16. Python Software Foundation. The Python Standard Library. – [Электронный ресурс] / Режим доступа: <https://docs.python.org/3/library/>, свободный (дата обращения: 07.05.2026).

ПРИЛОЖЕНИЕ А

(обязательное)

Веб-приложение улучшения качества видео

Листинг программы

Код веб-приложения представлен в общем репозитории проекта по ссылке <https://85.237.33.45:27192/gitea/Vsevolod/himeno>.

Файл main.py

```
import logging
import signal
import threading
from typing import Optional

from confluent_kafka import (
    Consumer,
    Producer,
    KafkaError,
    KafkaException,
    Message,
)
from confluent_kafka.admin import AdminClient, NewTopic

from .config import DispatcherConfig, load_config
from .scheduler import PriorityScheduler
from .aging import AgingMechanism
from .metrics import MetricsAuditor

logger = logging.getLogger(__name__)

class DispatcherApp:

    def __init__(self, config: Optional[DispatcherConfig] =
None) -> None:
        self.config = config or load_config()
        self._consumer: Optional[Consumer] = None
        self._producer: Optional[Producer] = None
        self._consuming: bool = False
        self._shutdown_requested: bool = False

        self.scheduler = PriorityScheduler()
        self.aging_mechanism: Optional[AgingMechanism] = None
        self.metrics_auditor =
MetricsAuditor(port=self.config.metrics_port)

        self._aging_thread: Optional[threading.Thread] = None

    def connect(self) -> None:
        self._ensure_topics_exist()
        self._create_consumer()
        self._create_producer()
        logger.info(
            "Подключение к Kafka установлено (bootstrap=%s,
group=%s)",
            self.config.kafka_bootstrap_servers,
            self.config.kafka_group_id,
```

```

    )

    def _ensure_topics_exist(self) -> None:
        admin_client = AdminClient({
            "bootstrap.servers":
self.config.kafka_bootstrap_servers,
        })
        existing_topics =
admin_client.list_topics(timeout=10).topics
        required_topics = [
            self.config.kafka_input_topic,
            self.config.kafka_output_topic,
        ]
        topics_to_create = []
        for topic in required_topics:
            if topic not in existing_topics:
                topics_to_create.append(
                    NewTopic(topic, num_partitions=1,
replication_factor=1)
                )
            if topics_to_create:
                futures =
admin_client.create_topics(topics_to_create)
                for topic, future in futures.items():
                    try:
                        future.result()
                        logger.info("Топик '%s' создан", topic)
                    except Exception as exc:
                        logger.warning(
                            "Не удалось создать топик '%s': %s",
topic, exc
                        )

    def _create_consumer(self) -> None:
        consumer_config = {
            "bootstrap.servers":
self.config.kafka_bootstrap_servers,
            "group.id": self.config.kafka_group_id,
            "auto.offset.reset": "earliest",
            "enable.auto.commit": False,
            "max.poll.interval.ms":
self.config.kafka_max_poll_interval_ms,
            "session.timeout.ms":
self.config.kafka_session_timeout_ms,
        }
        self._consumer = Consumer(consumer_config)

        self._consumer.subscribe([self.config.kafka_input_topic])
        logger.info(
            "Потребитель Kafka подписан на топик '%s'",
            self.config.kafka_input_topic,
        )

    def _create_producer(self) -> None:

```

```

        producer_config = {
            "bootstrap.servers":
self.config.kafka_bootstrap_servers,
            "acks": "all",
            "retries": self.config.kafka_max_retries,
            "retry.backoff.ms":
self.config.kafka_retry_backoff_ms,
            "enable.idempotence": True,
        }
        self._producer = Producer(producer_config)
        logger.info("Производитель Kafka инициализирован")

    def _register_signal_handlers(self) -> None:
        signal.signal(signal.SIGTERM, self._signal_handler)
        signal.signal(signal.SIGINT, self._signal_handler)
        logger.debug("Обработчики сигналов SIGTERM и SIGINT
зарегистрированы")

    def _signal_handler(self, signum: int, frame) -> None:
        sig_name = signal.Signals(signum).name
        logger.info("Получен сигнал %s. Иницируется
остановка...", sig_name)
        self._shutdown_requested = True
        self._consuming = False

    def shutdown(self) -> None:
        logger.info("Начало процедуры graceful shutdown...")
        self._consuming = False

        if self.aging_mechanism is not None:
            logger.debug("Остановка механизма устаревания...")
            self.aging_mechanism.stop()
            if self._aging_thread is not None:
                self._aging_thread.join(timeout=30)
                if self._aging_thread.is_alive():
                    logger.warning(
                        "Поток устаревания не завершился в
течение тайм-аута"
                    )

            logger.debug("Остановка аудитора метрик...")
            self.metrics_auditor.shutdown()

        if self._consumer is not None:
            logger.debug("Закрытие потребителя Kafka...")
            try:
                self._consumer.close()
            except KafkaException as exc:
                logger.warning("Ошибка при закрытии потребителя:
%s", exc)

        if self._producer is not None:
            logger.debug("Сброс буфера производителя Kafka...")
            remaining = self._producer.flush(timeout=30)

```

```

        if remaining > 0:
            logger.warning(
                "%d сообщений не были доставлены в течение
тайм-аута",
                remaining,
            )

        logger.info("Диспетчер остановлен")

def _start_background_components(self) -> None:
    self.aging_mechanism = AgingMechanism(
        consumer=self._consumer,
        producer=self._producer,
        input_topic=self.config.kafka_input_topic,
        output_topic=self.config.kafka_output_topic,
        interval=self.config.aging_interval,
        metrics_auditor=self.metrics_auditor,
    )
    self._aging_thread = threading.Thread(
        target=self.aging_mechanism.run,
        name="aging-thread",
        daemon=True,
    )
    self._aging_thread.start()
    logger.info(
        "Механизм устаревания запущен (интервал=%d сек.)",
        self.config.aging_interval,
    )

    self.metrics_auditor.start()
    logger.info(
        "Аудитор метрик запущен (порт=%d)",
        self.config.metrics_port
    )

def main() -> None:
    logging.basicConfig(
        level=logging.INFO,
        format=(
            '{"timestamp": "%(asctime)s", "level":
"%(levelname)s", '
            '"module": "%(name)s", "message": "%(message)s"}'
        ),
    )

    app = DispatcherApp()
    try:
        app.start()
    except RuntimeError as exc:
        logger.critical("Критическая ошибка при запуске: %s",
exc)
        raise SystemExit(1) from exc

```

```
if __name__ == "__main__":
    main()
```

Файл metrics.py

```
import logging
import threading
import time
import os
from datetime import datetime
from typing import Dict, List, Optional

logger = logging.getLogger(__name__)

class MetricsAuditor:

    PRIORITY_BUCKETS: Dict[str, tuple] = {
        "low": (0, 3),
        "medium": (4, 6),
        "high": (7, 8),
        "critical": (9, 10),
    }

    DEFAULT_FLUSH_INTERVAL: float = 30.0

    def __init__(
        self,
        file_path: str = "metrics/dispatcher_metrics.txt",
        flush_interval: float = DEFAULT_FLUSH_INTERVAL,
        max_waiting_time_samples: int = 10000,
        max_duration_samples: int = 5000,
        max_resolution_samples: int = 5000,
        max_complexity_samples: int = 5000,
        max_computation_time_samples: int = 5000,
    ) -> None:
        self._file_path = file_path
        self._flush_interval = flush_interval
        self._lock = threading.Lock()

        self._max_waiting_time_samples =
max_waiting_time_samples
        self._max_duration_samples = max_duration_samples
        self._max_resolution_samples = max_resolution_samples
        self._max_complexity_samples = max_complexity_samples
        self._max_computation_time_samples =
max_computation_time_samples

        self._tasks_by_priority: Dict[str, int] = {
            "low": 0,
            "medium": 0,
            "high": 0,
            "critical": 0,
```

```

    }

    self._aged_tasks_count: int = 0

    self._queue_depth: int = 0
    self._queue_topic: str = "video_processing_tasks"

    self._waiting_times: Dict[str, List[float]] = {
        "low": [],
        "medium": [],
        "high": [],
        "critical": [],
    }

    self._video_durations: List[float] = []
    self._video_resolutions: List[float] = []
    self._video_complexities: List[float] = []

    self._computation_times: List[float] = []

    self._errors: Dict[str, int] = {}

    self._start_time: float = time.time()

    self._running: bool = False
    self._flush_thread: Optional[threading.Thread] = None

    file_dir = os.path.dirname(self._file_path)
    if file_dir:
        os.makedirs(file_dir, exist_ok=True)

    logger.info(
        "MetricsAuditor инициализирован (file_path='%s',
flush_interval=%.1f сек.)",
        self._file_path,
        self._flush_interval,
    )

    def start(self) -> None:
        if self._running:
            logger.warning("Аудитор метрик уже запущен")
            return
        self._running = True
        self._flush_thread = threading.Thread(
            target=self._flush_loop,
            name="metrics-flush-thread",
            daemon=True,
        )
        self._flush_thread.start()
        logger.info(
            "Аудитор метрик запущен (flush_interval=%.1f сек.)",
            self._flush_interval,
        )

```

```

def shutdown(self) -> None:
    logger.info("Остановка аудитора метрик...")
    self._running = False
    if self._flush_thread is not None:
        self._flush_thread.join(timeout=10)
    self.flush_to_file()
    logger.info("Аудитор метрик остановлен")

def _flush_loop(self) -> None:
    logger.debug("Цикл записи метрик запущен")
    while self._running:
        time.sleep(self._flush_interval)
        if self._running:
            self.flush_to_file()
    logger.debug("Цикл записи метрик завершён")

def record_task_published(self, priority: int) -> None:
    priority_level = self._priority_to_level(priority)
    with self._lock:
        self._tasks_by_priority[priority_level] += 1
    logger.debug(
        "Зафиксирована публикация задачи (priority=%d,
level=%s)",
        priority,
        priority_level,
    )

def record_task_aged(self) -> None:
    with self._lock:
        self._aged_tasks_count += 1
    logger.debug("Зафиксировано старение задачи")

def record_task_started(
    self,
    priority: int,
    waiting_time_seconds: float,
) -> None:
    priority_level = self._priority_to_level(priority)
    with self._lock:
        waiting_list = self._waiting_times[priority_level]
        waiting_list.append(waiting_time_seconds)
        if len(waiting_list) >
self._max_waiting_time_samples:
        self._waiting_times[priority_level] =
waiting_list[
        -self._max_waiting_time_samples:
    ]
    logger.debug(
        "Зафиксировано начало обработки задачи "
        "(priority=%d, waiting_time=%.1f сек.)",
        priority,
        waiting_time_seconds,
    )

```

```

def record_video_metadata(
    self,
    duration: float,
    width: int,
    height: int,
    complexity_score: float,
) -> None:
    resolution_mp = (width * height) / 1_000_000.0
    with self._lock:
        self._video_durations.append(duration)
        self._video_resolutions.append(resolution_mp)
        self._video_complexities.append(complexity_score)

        if len(self._video_durations) >
self._max_duration_samples:
            self._video_durations = self._video_durations[
                -self._max_duration_samples:
            ]
        if len(self._video_resolutions) >
self._max_resolution_samples:
            self._video_resolutions =
self._video_resolutions[
                -self._max_resolution_samples:
            ]
        if len(self._video_complexities) >
self._max_complexity_samples:
            self._video_complexities =
self._video_complexities[
                -self._max_complexity_samples:
            ]

        logger.debug(
            "Зафиксированы метаданные видео "
            "(duration=%.1f сек., resolution=%.2f Мп,
complexity=%.3f)",
            duration,
            resolution_mp,
            complexity_score,
        )

def set_queue_depth(self, depth: int, topic: str = "") ->
None:
    with self._lock:
        self._queue_depth = depth
        if topic:
            self._queue_topic = topic
    logger.debug(
        "Глубина очереди установлена: %d (topic=%s)",
        depth,
        self._queue_topic,
    )

def record_error(self, error_type: str) -> None:
    with self._lock:

```

```

        self._errors[error_type] =
self._errors.get(error_type, 0) + 1
        logger.debug("Зафиксирована ошибка (type=%s)",
error_type)

    def record_priority_computation(self, duration_seconds:
float) -> None:
        with self._lock:
            self._computation_times.append(duration_seconds)
            if len(self._computation_times) >
self._max_computation_time_samples:
                self._computation_times =
self._computation_times[
                    -self._max_computation_time_samples:
                ]
            logger.debug(
                "Зафиксировано время вычисления приоритета: %.4f
сек.",
                duration_seconds,
            )

    def flush_to_file(self) -> None:
        try:
            metrics_text = self._format_metrics()
            with open(self._file_path, "w", encoding="utf-8") as
f:
                f.write(metrics_text)
            logger.debug("Метрики записаны в файл '%s'",
self._file_path)
        except OSError as exc:
            logger.error(
                "Ошибка записи метрик в файл '%s': %s",
                self._file_path,
                exc,
            )

    def _format_metrics(self) -> str:
        with self._lock:
            lines: List[str] = []
            now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            uptime = time.time() - self._start_time

            lines.append("=" * 60)
            lines.append("МЕТРИКИ ДИСПЕТЧЕРА ЗАДАЧ")
            lines.append("=" * 60)
            lines.append(f"Дата и время формирования: {now}")
            lines.append(f"Время работы диспетчера:
{uptime:.0f} сек.")
            lines.append("")

            lines.append("-" * 40)
            lines.append("Распределение задач по уровням
приоритета:")
            total_tasks = sum(self._tasks_by_priority.values())

```

```

        for level in ["low", "medium", "high", "critical"]:
            count = self._tasks_by_priority[level]
            pct = (count / total_tasks * 100) if total_tasks
> 0 else 0.0
            low, high = self.PRIORITY_BUCKETS[level]
            lines.append(
                f" {level:10s} (приоритет {low}-{high}): "
                f"{count:6d} задач ({pct:5.1f}%)"
            )
            lines.append(f" {'ВСЕГО':10s}
{total_tasks:6d} задач")
            lines.append("")

            lines.append("-" * 40)
            lines.append("Механизм запланированного
устаревания:")
            lines.append(f" Количество состаренных задач:
{self._aged_tasks_count}")
            pct_aged = (
                (self._aged_tasks_count / total_tasks * 100)
                if total_tasks > 0
                else 0.0
            )
            lines.append(f" Доля от общего числа задач:
{pct_aged:.2f}%")
            lines.append("")

            lines.append("-" * 40)
            lines.append("Состояние очереди:")
            lines.append(f" Топик:
{self._queue_topic}")
            lines.append(f" Текущая глубина:
{self._queue_depth}")
            lines.append("")

            lines.append("-" * 40)
            lines.append("Время ожидания задач в очереди
(секунды):")
            for level in ["low", "medium", "high", "critical"]:
                samples = self._waiting_times[level]
                if samples:
                    avg_val = sum(samples) / len(samples)
                    min_val = min(samples)
                    max_val = max(samples)
                    sorted_samples = sorted(samples)
                    p50_idx = int(len(sorted_samples) * 0.5)
                    p95_idx = int(len(sorted_samples) * 0.95)
                    p50_val = sorted_samples[p50_idx] if p50_idx
< len(sorted_samples) else sorted_samples[-1]
                    p95_val = sorted_samples[p95_idx] if p95_idx
< len(sorted_samples) else sorted_samples[-1]
                    lines.append(
                        f" {level:10s}: кол-
во={len(samples):5d}, "

```

```

        f"среднее={avg_val:7.1f},
медиана={p50_val:7.1f}, "
        f"P95={p95_val:7.1f},
мин={min_val:7.1f}, макс={max_val:7.1f}"
    )
    else:
        lines.append(f"  {level:10s}: нет данных")
    lines.append("")

    lines.append("-" * 40)
    lines.append("Характеристики загружаемого
видео контента:")

    if self._video_durations:
        lines.append(
            f"  Длительность (сек.):      "
            f"кол-во={len(self._video_durations)}, "

f"среднее={self._mean(self._video_durations):.1f}, "
            f"мин={min(self._video_durations):.1f}, "
            f"макс={max(self._video_durations):.1f}"
        )
    else:
        lines.append("  Длительность (сек.):      нет
данных")

    if self._video_resolutions:
        lines.append(
            f"  Разрешение (Мп):      "
            f"кол-во={len(self._video_resolutions)}, "

f"среднее={self._mean(self._video_resolutions):.2f}, "
            f"мин={min(self._video_resolutions):.2f}, "
            f"макс={max(self._video_resolutions):.2f}"
        )
    else:
        lines.append("  Разрешение (Мп):      нет
данных")

    if self._video_complexities:
        lines.append(
            f"  Сложность (0-1):      "
            f"кол-во={len(self._video_complexities)}, "

f"среднее={self._mean(self._video_complexities):.3f}, "
            f"мин={min(self._video_complexities):.3f}, "
            f"макс={max(self._video_complexities):.3f}"
        )
    else:
        lines.append("  Сложность (0-1):      нет
данных")

    lines.append("")

    lines.append("-" * 40)

```

```

        lines.append("Производительность вычислителя
приоритета:")
        if self._computation_times:
            avg_comp = self._mean(self._computation_times)
            lines.append(
                f"    Время вычисления (сек.): "
                f"кол-во={len(self._computation_times)}, "
                f"среднее={avg_comp:.6f}, "
                f"мин={min(self._computation_times):.6f}, "
                f"макс={max(self._computation_times):.6f}"
            )
        else:
            lines.append("    Время вычисления (сек.):  нет
данных")
        lines.append("")

        lines.append("-" * 40)
        lines.append("Ошибки обработки:")
        if self._errors:
            total_errors = sum(self._errors.values())
            for error_type, count in
sorted(self._errors.items()):
                lines.append(f"    {error_type:30s}:
{count:5d}")
            lines.append(f"    {'ВСЕГО ОШИБОК':30s}:
{total_errors:5d}")
        else:
            lines.append("    Ошибок не зафиксировано")
            lines.append("")

        lines.append("=" * 60)

        return "\n".join(lines)

def _priority_to_level(self, priority: int) -> str:
    for level, (low, high) in self.PRIORITY_BUCKETS.items():
        if low <= priority <= high:
            return level
    return "unknown"

@staticmethod
def _mean(values: List[float]) -> float:
    if not values:
        return 0.0
    return sum(values) / len(values)

def get_tasks_count(self) -> int:
    with self._lock:
        return sum(self._tasks_by_priority.values())

def get_aged_tasks_count(self) -> int:
    with self._lock:
        return self._aged_tasks_count

```

```

def get_queue_depth(self) -> int:
    with self._lock:
        return self._queue_depth

def get_error_counts(self) -> Dict[str, int]:
    with self._lock:
        return dict(self._errors)

def get_metrics_snapshot(self) -> str:
    return self._format_metrics()

```

Файл aging.py

```

import logging
import threading
import time
from typing import Optional, List, Dict, Any

from confluent_kafka import Consumer, Producer, KafkaException,
Message
from confluent_kafka.admin import AdminClient

logger = logging.getLogger(__name__)

class AgingMechanism:

    DEFAULT_AGING_INTERVAL: int = 60
    DEFAULT_AGING_THRESHOLD: int = 120
    DEFAULT_AGING_RATE: float = 0.1
    DEFAULT_PROMOTION_THRESHOLD: int = 8

    def __init__(
        self,
        consumer: Consumer,
        producer: Producer,
        output_topic: str,
        interval: int = DEFAULT_AGING_INTERVAL,
        aging_threshold: int = DEFAULT_AGING_THRESHOLD,
        aging_rate: float = DEFAULT_AGING_RATE,
        promotion_threshold: int = DEFAULT_PROMOTION_THRESHOLD,
        metrics_auditor: Optional[Any] = None,
    ) -> None:
        self._consumer = consumer
        self._producer = producer
        self._output_topic = output_topic
        self._interval = interval
        self._aging_threshold = aging_threshold
        self._aging_rate = aging_rate
        self._promotion_threshold = promotion_threshold
        self._metrics_auditor = metrics_auditor

        self._running: bool = False

```

```

self._thread: Optional[threading.Thread] = None
self._lock = threading.Lock()
self._aged_tasks_count: int = 0

logger.info(
    "AgingMechanism инициализирован (interval=%d сек., "
    "threshold=%d сек., rate=%.2f, promotion=%d)",
    self._interval,
    self._aging_threshold,
    self._aging_rate,
    self._promotion_threshold,
)

def start(self) -> None:
    if self._running:
        logger.warning("Механизм устаревания уже запущен")
        return
    self._running = True
    self._thread = threading.Thread(
        target=self._run_loop,
        name="aging-thread",
        daemon=True,
    )
    self._thread.start()
    logger.info("Механизм устаревания запущен")

def stop(self) -> None:
    logger.info("Остановка механизма устаревания...")
    self._running = False
    if self._thread is not None:
        self._thread.join(timeout=30)
        if self._thread.is_alive():
            logger.warning(
                "Поток устаревания не завершился в течение
тайм-аута"
            )

def _run_loop(self) -> None:
    logger.debug("Цикл устаревания запущен")
    while self._running:
        try:
            time.sleep(self._interval)
            if not self._running:
                break
            self._perform_aging_cycle()
        except Exception as exc:
            logger.exception(
                "Ошибка в цикле устаревания: %s", exc
            )

    logger.debug("Цикл устаревания завершён")

def _perform_aging_cycle(self) -> None:
    logger.debug("Запуск цикла устаревания")

```

```

waiting_tasks = self._fetch_waiting_tasks()
if not waiting_tasks:
    logger.debug("Нет задач, ожидающих обработки")
    return

now = time.time()
candidates = []
for task in waiting_tasks:
    waiting_time = now - task["publish_timestamp"]
    if waiting_time >= self._aging_threshold:
        candidates.append(task)

if not candidates:
    logger.debug("Нет задач-кандидатов на старение")
    return

logger.info("Найдено %d задач-кандидатов на старение",
len(candidates))

for task in candidates:
    try:
        self._age_task(task, now)
    except Exception as exc:
        logger.error(
            "Ошибка при старении задачи (offset=%d):
%s",
            task.get("offset", -1),
            exc,
        )

def _fetch_waiting_tasks(self) -> List[Dict[str, Any]]:
    try:
        admin_client = AdminClient({
            "bootstrap.servers": self._consumer.conf().get(
                "bootstrap.servers", "localhost:9092"
            ),
        })

        consumer_groups = admin_client.list_consumer_groups(
            request_timeout=10
        )
        waiting_tasks = []

        for group in consumer_groups:
            if not
group.group_id.startswith("video_processor"):
                continue
            try:
                offsets =
admin_client.list_consumer_group_offsets(
                    group.group_id, request_timeout=10
                )

```

```

        for topic_partition, offset_metadata in
offsets.items():
            if topic_partition.topic !=
self._output_topic:
                continue
            lag = offset_metadata.lag
            for i in range(lag):
                waiting_tasks.append({
                    "topic": topic_partition.topic,
                    "partition":
topic_partition.partition,
                    "offset": offset_metadata.offset
+ i,
                    "publish_timestamp": time.time()
- (lag - i) * 60,
                    "priority": 0,
                })
            except KafkaException as exc:
                logger.warning(
                    "Ошибка получения смещений для группы
'%s': %s",
                    group.group_id,
                    exc,
                )

                logger.debug("Обнаружено %d ожидающих задач",
len(waiting_tasks))
                return waiting_tasks

            except KafkaException as exc:
                logger.error("Ошибка получения списка ожидающих
задач: %s", exc)
                return []

    def _age_task(self, task: Dict[str, Any], now: float) ->
None:
        waiting_time = now - task["publish_timestamp"]
        original_priority = task.get("priority", 0)

        priority_increase = self._aging_rate * (waiting_time -
self._aging_threshold)
        new_priority = min(
            original_priority + int(priority_increase),
            10,
        )

        logger.debug(
            "Задача (partition=%d, offset=%d): waiting_time=%.1f
сек., "
            "priority %d -> %d",
            task.get("partition", -1),
            task.get("offset", -1),
            waiting_time,
            original_priority,

```

```

        new_priority,
    )

    if new_priority >= self._promotion_threshold:
        self._promote_task(task, new_priority)

def _promote_task(
    self, task: Dict[str, Any], new_priority: int
) -> None:
    try:
        headers = [
            ("x-priority", str(new_priority).encode("utf-
8")),
            ("x-aged", b"true"),
        ]
        self._producer.produce(
            topic=self._output_topic,
            value=task.get("value", b""),
            headers=headers,
            callback=self._on_promotion_delivery,
        )
        self._producer.poll(0)

        with self._lock:
            self._aged_tasks_count += 1

        if self._metrics_auditor is not None:
            self._metrics_auditor.record_task_aged()

        logger.info(
            "Задача продвинута (partition=%d, offset=%d): "
            "новый приоритет=%d",
            task.get("partition", -1),
            task.get("offset", -1),
            new_priority,
        )

    except KafkaException as exc:
        logger.error(
            "Ошибка при продвижении задачи (offset=%d): %s",
            task.get("offset", -1),
            exc,
        )

def _on_promotion_delivery(
    self, err: Optional[Any], msg: Message
) -> None:
    if err is not None:
        logger.error(
            "Ошибка доставки продвинутой задачи (topic=%s):
%s",
            msg.topic(),
            err,
        )

```

```

        else:
            logger.debug(
                "Продвинутая задача доставлена (topic=%s,
offset=%d)",
                msg.topic(),
                msg.offset(),
            )

```

```

@property
def aged_tasks_count(self) -> int:
    with self._lock:
        return self._aged_tasks_count

```

Файл scheduler.py

```

import logging
import math
from concurrent.futures import ThreadPoolExecutor, Future
from typing import Dict, Any

from .exceptions import InvalidMetadataError

logger = logging.getLogger(__name__)

class PriorityScheduler:

    MIN_PRIORITY: int = 0
    MAX_PRIORITY: int = 10

    FORMULA_CONST_A: float = 2.152
    FORMULA_CONST_B: float = 0.169

    CODEC_COEFFICIENTS: Dict[str, float] = {
        "h264": 1.00,
        "h.264": 1.00,
        "hevc": 1.11,
        "h.265": 1.11,
        "av1": 1.43,
    }

    BITRATE_COEFF_A: float = 0.84
    BITRATE_COEFF_B: float = 0.0024

    BITRATE_MIN: float = 20.0
    BITRATE_MAX: float = 100.0

    DEFAULT_FRAMERATE: float = 30.0

    REQUIRED_FIELDS: tuple = ("duration", "width", "height",
"bitrate", "codec")

    def __init__(self, max_workers: int = 4) -> None:

```

```

        self._executor = ThreadPoolExecutor(
            max_workers=max_workers,
            thread_name_prefix="priority-worker",
        )
        logger.info(
            "PriorityScheduler инициализирован
(max_workers=%d)",
            max_workers,
        )

    def compute_priority(
        self,
        metadata: Dict[str, Any],
        framerate: float = DEFAULT_FRAMERATE,
    ) -> int:
        self._validate_metadata(metadata)

        duration_sec = float(metadata["duration"])
        width = int(metadata["width"])
        height = int(metadata["height"])
        bitrate_mbps = float(metadata["bitrate"])

        if "framerate" in metadata and metadata["framerate"] is
not None:
            framerate = float(metadata["framerate"])

        codec = metadata.get("codec", "h264").lower()
        codec_coeff = self.CODEC_COEFFICIENTS.get(codec, 1.00)

        priority_float = self._compute_priority_formula(
            width=width,
            height=height,
            duration_sec=duration_sec,
            framerate=framerate,
            bitrate_mbps=bitrate_mbps,
            codec_coeff=codec_coeff,
        )

        scaled = int(round(
            self.MIN_PRIORITY + (priority_float - 0.1) / (1.0 -
0.1) *
            (self.MAX_PRIORITY - self.MIN_PRIORITY)
        ))

        priority = max(self.MIN_PRIORITY, min(self.MAX_PRIORITY,
scaled))

        logger.debug(
            "Приоритет вычислен: P=%.3f -> priority=%d "
            "(resolution=%dx%d, duration=%.1fc, fps=%.1f,
bitrate=%.1fМбит/с, codec=%s, coeff=%.2f)",
            priority_float,
            priority,
            width,

```

```

        height,
        duration_sec,
        framerate,
        bitrate_mbps,
        codec,
        codec_coeff,
    )
    return priority

def _compute_priority_formula(
    self,
    width: int,
    height: int,
    duration_sec: float,
    framerate: float,
    bitrate_mbps: float,
    codec_coeff: float,
) -> float:
    pixels = float(width * height)
    frames = framerate * duration_sec

    clamped_bitrate = max(self.BITRATE_MIN,
min(self.BITRATE_MAX, bitrate_mbps))
    k_bitrate = self.BITRATE_COEFF_A + self.BITRATE_COEFF_B
* clamped_bitrate

    log_pixels = math.log10(max(pixels, 1.0))
    log_frames = math.log10(max(frames, 1.0))
    log_k_bitrate = math.log10(max(k_bitrate, 1e-10))
    log_k_codec = math.log10(max(codec_coeff, 1e-10))

    log_sum = log_pixels + log_frames + log_k_bitrate +
log_k_codec

    p_value = self.FORMULA_CONST_A - self.FORMULA_CONST_B *
log_sum

    clamped_p = max(0.1, min(1.0, p_value))

    return clamped_p

def compute_priority_async(
    self,
    metadata: Dict[str, Any],
    framerate: float = 30.0,
) -> Future:
    return self._executor.submit(self.compute_priority,
metadata, framerate)

def _validate_metadata(self, metadata: Dict[str, Any]) ->
None:
    if not isinstance(metadata, dict):
        raise InvalidMetadataError(

```

```

        f"Метаданные должны быть словарём, получен
{type(metadata).__name__}"
    )

    missing_fields = [
        field for field in self.REQUIRED_FIELDS if field not
in metadata
    ]
    if missing_fields:
        raise InvalidMetadataError(
            f"Отсутствуют обязательные поля метаданных:
{missing_fields}"
        )

    duration = metadata.get("duration")
    if not isinstance(duration, (int, float)) or duration <=
0:
        raise InvalidMetadataError(
            f"Некорректная длительность: {duration}"
        )

    width = metadata.get("width")
    if not isinstance(width, int) or width <= 0:
        raise InvalidMetadataError(
            f"Некорректная ширина: {width}"
        )

    height = metadata.get("height")
    if not isinstance(height, int) or height <= 0:
        raise InvalidMetadataError(
            f"Некорректная высота: {height}"
        )

    bitrate = metadata.get("bitrate")
    if not isinstance(bitrate, (int, float)) or bitrate < 0:
        raise InvalidMetadataError(
            f"Некорректный битрейт: {bitrate}"
        )

    codec = metadata.get("codec")
    if not isinstance(codec, str) or not codec:
        raise InvalidMetadataError(
            f"Некорректный кодек: {codec}"
        )

    logger.debug("Валидация метаданных пройдена успешно")

    def get_codec_coefficient(self, codec_name: str) -> float:
        return self.CODEC_COEFFICIENTS.get(codec_name.lower(),
1.00)

    def shutdown(self, wait: bool = True) -> None:
        logger.info("Завершение работы пула потоков
планировщика...")

```

```
self._executor.shutdown(wait=wait)
logger.info("Пул потоков планировщика остановлен")
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Веб-приложение улучшения качества видео

Презентация

Веб-приложение улучшения качества видео

Автор работы:

студент гр. 24ВВИм1 Верховский Михаил Владимирович

Научный руководитель:

д.т.н., профессор Зинкин Сергей Александрович

Пенза, 2026

Цель, объект, предмет исследования

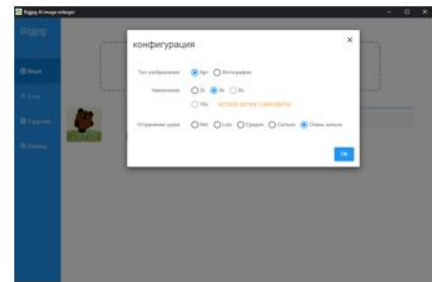
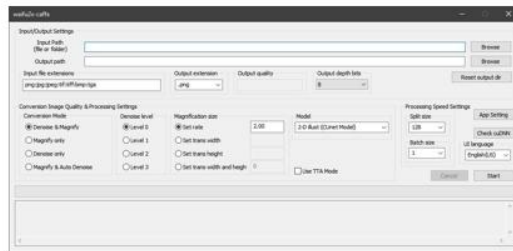
Проблема: при одновременной загрузке видео система перегружается. Короткие видео ждут часами, задачи теряются.

Объект: процессы передачи и диспетчеризации задач в асинхронной системе обработки видео.

Цель: исследование, проектирование, реализация и оценка эффективности архитектуры веб-приложения для улучшения качества видео с акцентом на диспетчеризацию задач и асинхронную обработку.

Предмет: архитектурные решения с менеджером сообщений и диспетчером задач.

Аналоги



3

Актуальность, научная новизна и практическая значимость

Научная новизна: подход к диспетчеризации задач для генеративно-состязательной обработки с учётом характеристик файла и данных обучения модели. Была выведена формула параметрического приоритета и внедрен механизм старения, исключающий бесконечное откладывание низкоприоритетных задач.

Актуальность:

Над проблемой работают Google, Netflix, NVIDIA;
Мало открытых решений для приоритезации задач при работе с GAN;
Особенно актуально в России (аналоги недоступны).

Практическая значимость:

Использование в коммерческих сервисах улучшения качества видео;
Тиражирование для других задач ML с асинхронной обработкой.

4

Задачи исследования

Для достижения цели решались следующие **задачи**:

1. Исследование влияния характеристик видео на время обработки GAN-моделью.
2. Вывод формулы параметрического приоритета.
3. Сравнение алгоритмов диспетчеризации и выбор оптимального.
4. Проектирование и реализация веб-приложения и отдельного диспетчера задач.
5. Экспериментальная оценка эффективности разработанной системы.

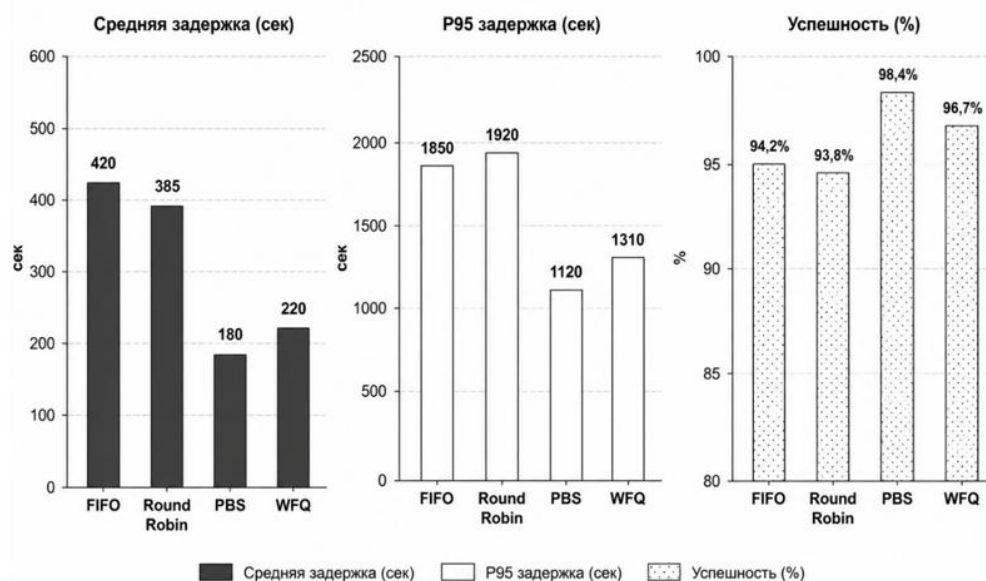
5

Сравнение алгоритмов диспетчеризации

Подход к диспетчеризации	Преимущества	Недостатки
FIFO	Максимальная простота реализации, предсказуемое поведение, отсутствие накладных расходов на приоритезацию	Длинные задачи блокируют короткие, не позволяет реализовать дифференцированное обслуживание пользователей, может привести к отказу в обслуживании
Round Robin	Прост в реализации, равномерно распределяет нагрузку, нет необходимости в централизованной очереди задач	Предполагает одинаковую сложность всех задач, не учитывает загрузку обработчиков, может создать локальные перегрузки для кластеров обработки
PBS	Предотвращает блокировку очереди задач, позволяет реализовать дифференцированное обслуживание пользователей, адаптируется к динамическому профилю нагрузки	Сложен в реализации, высокие накладные расходы на расчет приоритета
WFQ	Гарантирует долю вычислительных ресурсов каждому типу задач, имеет защиту от полной блокировки очереди	Требует предварительного определения класса задач, плохо адаптируется к динамическому профилю нагрузки, в некоторых случаях может вызвать сложность подбора коэффициентов нагрузки

6

Сравнение алгоритмов диспетчеризации



7

Характеристики файла как фактор нагрузки

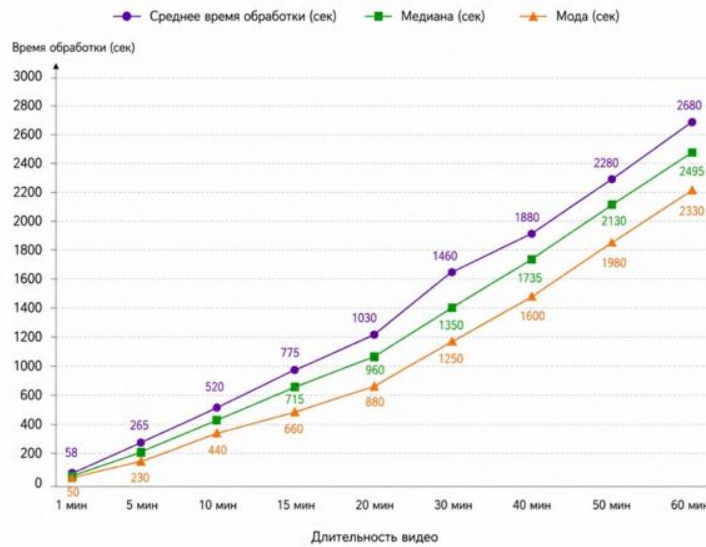
Признак	Предварительные значения	Оценка влияния
Разрешение	HD, Full HD, 2K, 4K	Критическое, степенная зависимость от площади кадра
Длительность	Короткие (до 60 секунд) Средние (от 1 до 10 минут) Длинные (более 10 минут)	Высокое, линейная зависимость
FPS	24, 30, 45, 60 кадров в секунду	Высокое, линейная зависимость
Кодек сжатия	H.264, HEVC (H.265), AV1	Высокое, зависит от частных случаев
Битрейт	От 20 до 100 Мбит/с	При значениях 60 Мбит/с и больше оказывает сильного влияние. В остальных случаях – среднее.
Глубина цвета	8 бит (SDR) или 10 бит (HDR)	Среднее, не оказывает сильного влияния

8

Результаты исследования метрик для стандартной модели

Зависимость времени обработки видео от длительности видео

- стандартное разрешение: 640 на 480 пикселей,
- извлекаемое разрешение: 1920 на 1080 пикселей,
- частота кадров: 24 кадра в секунду,
- длительность: от 1 до 60 минут.

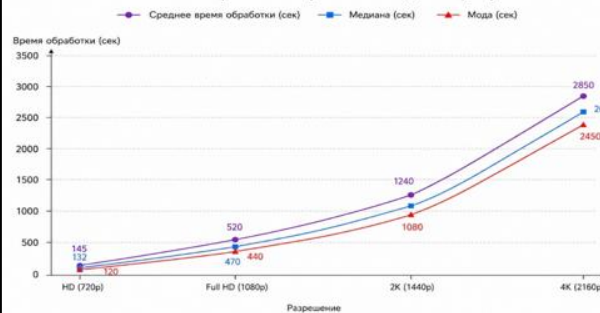


Характеристики виде, действительно, влияют на время обработки. Гипотеза Н1 подтверждена.

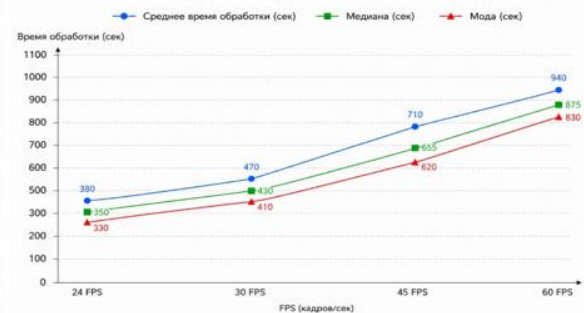
9

Результаты исследования метрик для стандартной модели

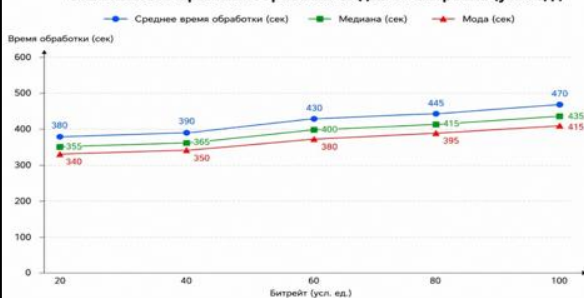
Зависимость времени обработки видео от разрешения



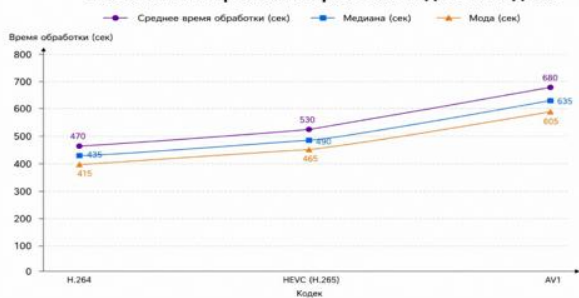
Зависимость времени обработки видео от FPS (кадров/сек)



Зависимость времени обработки видео от битрейта (усл. ед.)



Зависимость времени обработки видео от кодека



10

Формирование датасета



1. Советская мультипликация;
2. Зарубежная мультипликация;
3. Документальные кадры.

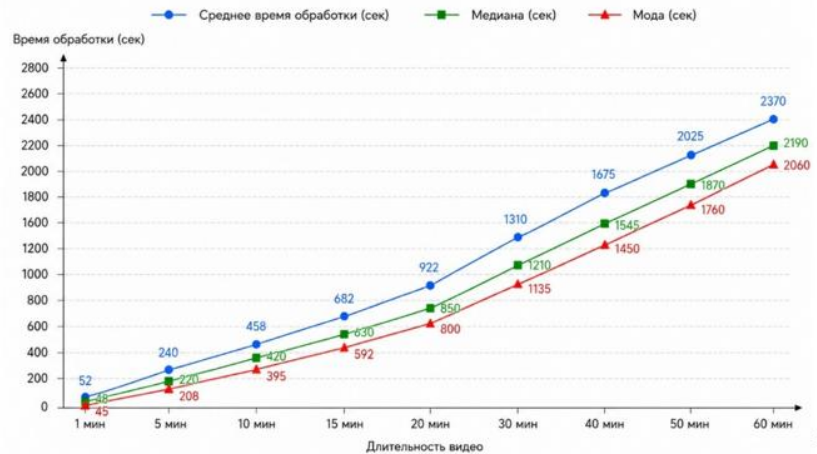
11

Результаты исследования метрик для дообученной модели

Зависимость времени обработки от длительности видео (управляемое обучение)

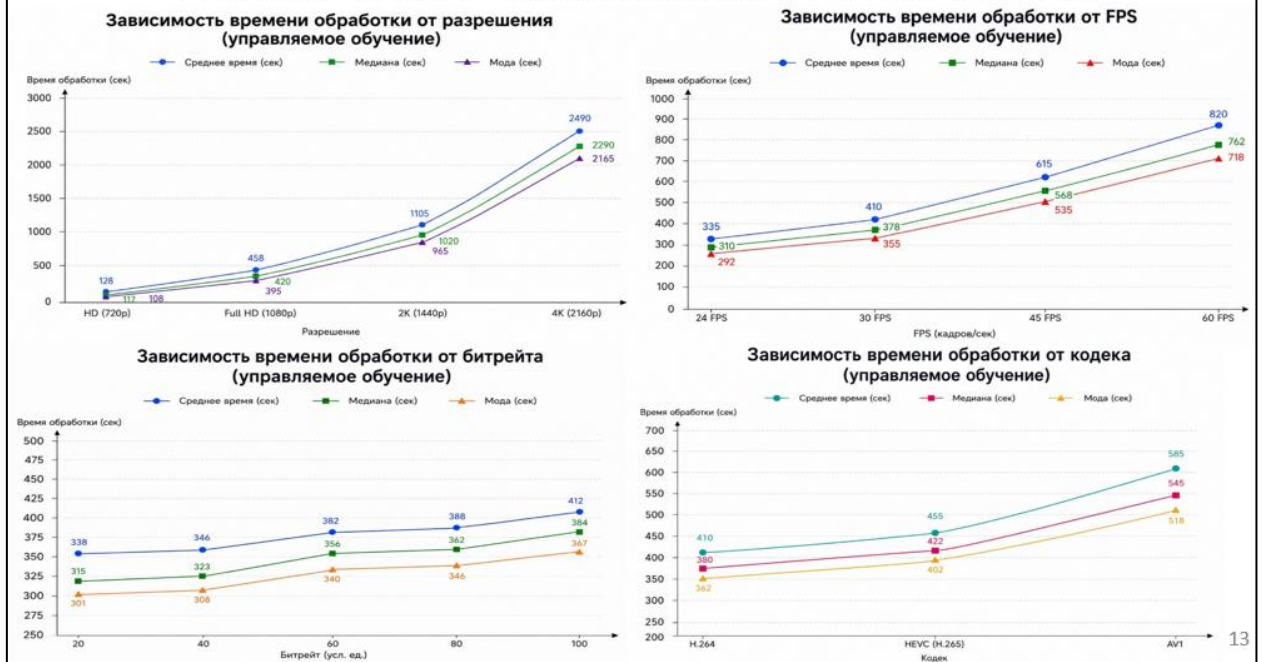
Сравнение графиков до и после дообучения показывает, что ускорение обработки составило 10–14%, а зависимости стали более чёткими.

Гипотеза H2 подтверждена: дообучение модели конкретизирует характеристики и делает зависимость более явной.



12

Результаты исследования метрик для дообученной модели



Формула композитного приоритета

$$P = 2.152 - 0.169 \times (\lg N_{\text{пикселей}} + \lg N_{\text{фреймов}} + \lg(0.84 + 0.0024 \times B)) + \lg K_{\text{кодек}}$$

где $N_{\text{пикселей}} = W \times H$ – количество пикселей в кадре;

$N_{\text{фреймов}} = F \times D$ – общее количество кадров;

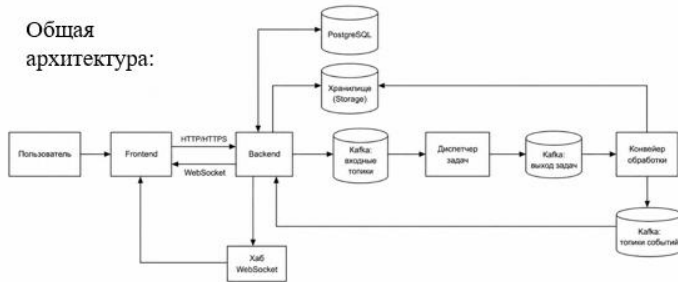
B – битрейт, Мбит/с ($20 \leq B \leq 100$);

$K_{\text{кодек}} = 1.00$ для H.264, 1.11 для HEVC (H.265), 1.43 для AV1;

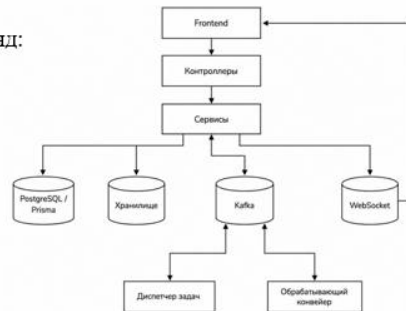
$\lg$ – десятичный логарифм;

Архитектура проекта

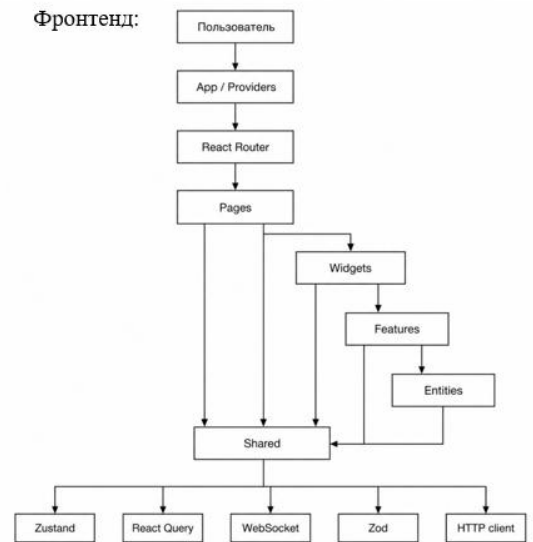
Общая архитектура:



Бэкенд:



Фронтенд:



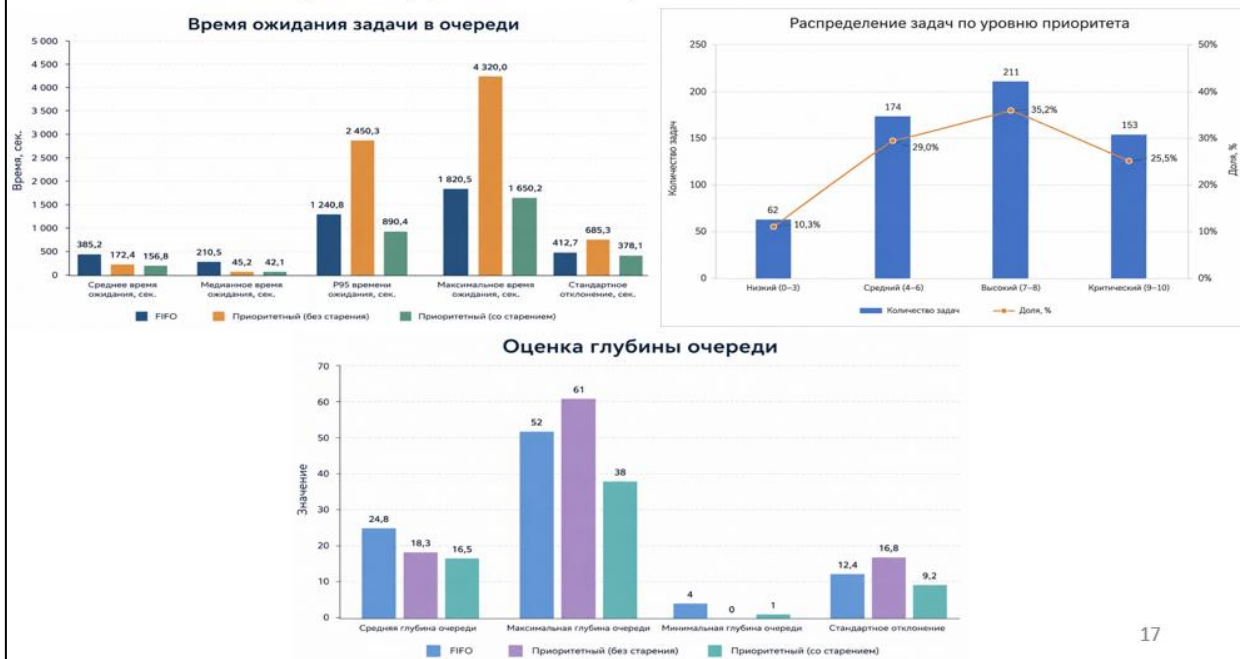
15

Диспетчер задач



16

Оценка эффективности реализованной системы



17

Заключение

1. Проведён анализ существующих решений, выявлены их ограничения.
2. Исследовано влияние характеристик видео на время обработки GAN-моделью.
3. Выведена формула композитного приоритета, учитывающая длительность, разрешение, FPS, кодек и битрейт.
4. Реализовано веб-приложение.
5. Разработан отдельный диспетчер задач на Python с механизмом запланированного устаревания.
6. Экспериментально подтверждена эффективность: среднее время ожидания сократилось на 59%, медианное – с 210 до 42 секунд.

Цель магистерской диссертации достигнута. Все поставленные задачи выполнены.

18