

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

**КОНВЕЙЕР ОБРАБОТКИ ВИДЕОФАЙЛОВ
ДЛЯ ВЕБ-ПРИЛОЖЕНИЯ**

Студент

Лукин Всеволод Дмитриевич
(подпись, дата) 26.05.26 (ФИО полностью)

Руководитель

Зинкин С.А.
(подпись, дата) 26.05.26 (фамилия, инициалы)

Нормоконтролёр

Бычков А.С.
(подпись, дата) 26.05.26 (фамилия, инициалы)

Рецензент

зав. каф. Программного обеспечения ИТ (должность, место работы) 22.06.26 (подпись, дата) Мартышкин А.И. (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.26 № 14)

Заведующий кафедрой

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

Работа защищена с отметкой отлично (протокол заседания ГЭК от

22 июня 2026 г. № 16)

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 20 24 г.

**ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

1. Студент Лукин Всеволод Дмитриевич гр. 24ВВИм1 факультета Вычислительной техники.

2. Тема работы: «Конвейер обработки видеофайлов для веб-приложения.»

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

3. Руководитель магистерской работы д.т.н., профессор, каф. ВТ ПГУ Зинкин С.А.

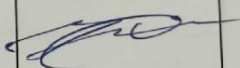
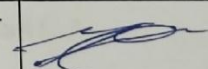
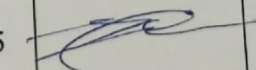
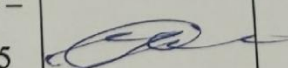
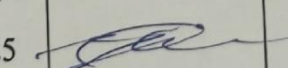
4. Задание на работу (назначение разработки, исходные данные и т.п.)

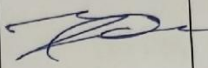
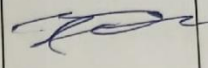
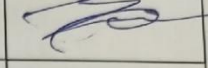
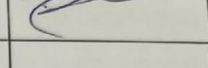
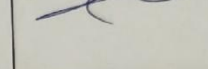
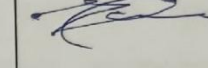
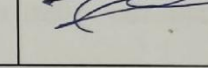
Разработать и реализовать отказоустойчивый событийно-ориентированный конвейер обработки видеофайлов с микросервисной архитектурой, использующий нейронные сети для улучшения качества видеоконтента. Провести исследование алгоритмических и нейросетевых подходов к улучшению качества видео, выполнить сравнительный анализ архитектур CNN, GAN и диффузионных моделей, отобрать и дообучить модель Real-ESRGAN на собственном датасете. Данное программное обеспечение должно в автоматическом режиме принимать поступающие через веб-приложение видеофайлы, выполнять их валидацию, отделение аудиодорожки, разбиение на фрагменты и покадровое разложение, нейросетевое улучшение качества каждого кадра, обратную сборку и слияние с аудиодорожкой. Улучшение качества должно производиться при помощи модуля нейросетевой обработки на базе дообученной модели и библиотеки для работы с мультимедиа. Взаимодействие сервисов должно осуществляться асинхронно, через менеджер сообщений и очередей. Система должна обеспечивать отказоустойчивость за счёт гарантированной доставки сообщений, идемпотентности сервисов и автоматического восстановления после сбоев, а также горизонтальное масштабирование при пиковых нагрузках за приемлемый коэффициент соотношения качества преобразования к временным затратам.

5. Перечень подлежащих разработке вопросов

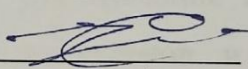
Требуется провести анализ предметной области: изучить существующие конвейеры мультимедийной обработки, выделить их достоинства и недостатки. Исследовать алгоритмические и нейросетевые подходы к улучшению качества видео, выполнить сравнительный анализ архитектур CNN, GAN и диффузионных моделей. Сформировать датасет и провести дообучение отобранной модели Real-ESRGAN для повышения качества обработки пользовательского контента. Выбрать и обосновать архитектурный стиль конвейера и модель коммуникации, описать внутреннюю организацию системы, состав сервисов и структуру событий. Определить технологический стек. Реализовать транспортный уровень, сервисы подготовки и сборки контента, модуль нейросетевой обработки и механизмы отказоустойчивости. Выполнить интеграционное и нагрузочное тестирование конвейера, оценить эффективность по метрикам качества видео и времени обработки.

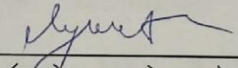
6. Календарный график выполнения работы

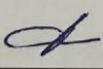
№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Обзор литературы и существующих решений по теме	80	20.09.2024 – 30.10.2024	
2	Формулировка цели, задач и составление развёрнутого плана работы	40	01.11.2024 – 20.11.2024	
3	Написание введения и постановки задачи	40	21.11.2024 – 15.12.2024	
4	Исследование алгоритмических и нейросетевых подходов к улучшению качества видео	120	01.02.2025 – 25.03.2025	
5	Формирование датасета и дообучение модели Real-ESRGAN	120	26.03.2025 – 15.05.2025	
6	Сравнительный анализ, отбор решений и формирование требований к модулю нейросетевой обработки	60	16.05.2025 – 20.06.2025	
7	Анализ архитектурных подходов и проектирование общей архитектуры конвейера	80	01.09.2025 – 10.10.2025	
8	Проектирование структуры событий, очередей, модулей подготовки и сборки	100	11.10.2025 – 30.11.2025	

9	Проектирование модуля нейросетевой обработки и механизмов отказоустойчивости	80	01.12.2025 – 31.12.2025	
10	Обоснование технологического стека и реализация транспортного уровня	100	15.01.2026 – 28.02.2026	
11	Реализация сервисов подготовки контента	100	01.03.2026 – 25.03.2026	
12	Реализация модуля нейросетевой обработки и сервисов сборки	140	26.03.2026 – 22.04.2026	
13	Реализация механизмов отказоустойчивости и интеграционное тестирование	80	23.04.2026 – 03.05.2026	
14	Нагрузочное тестирование и оценка эффективности	60	04.05.2026 – 12.05.2026	
15	Оформление пояснительной записки, подготовка презентации и предзащита	100	13.05.2026 – 20.05.2026	

Дата выдачи задания " 20 " 09 2024 г.

Руководитель магистерской диссертации  Зинкин С.А.
(подпись, дата)

Задание к исполнению принял студент  Лукин В.Д.
(подпись, дата)

Работу к защите допустить
Декан факультета  05.06.2020 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Магистерская работа содержит 125 листов, 29 рисунков, 10 таблиц, 10 источников, 2 приложения, 6 формул.

ОТКАЗОУСТОЙЧИВЫЙ КОНВЕЙЕР, МИКРОСЕРВИСНАЯ СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА, ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНЫЕ СЕТИ, АСИНХРОННОЕ ТРАНСКОДИРОВАНИЕ

Целью работы является исследование, проектирование, реализация и оценка эффективности полученной микросервисной архитектуры отказоустойчивого событийно-ориентированного конвейера обработки видеофайлов для веб-приложения.

Объектами исследования являются процессы передачи, хранения, обработки видеофайлов, а также доступные архитектурные решения для улучшения качества медиаконтента. Предметами исследования являются реализации архитектуры конвейерной схемы обработки данных и нейросетевых алгоритмов для улучшения качества медиаконтента.

В данной работе было проведено исследование построения конвейера обработки видеофайлов, нейросетевых алгоритмов и их реализаций для улучшения медиаконтента, выделены основные критерии для построения отказоустойчивой системы, дообучена собственная нейронная сеть для обработки медиафайлов.

В качестве результата данной работы был реализован отказоустойчивый событийно-ориентированный конвейер обработки видеофайлов для веб-приложения, дообучена нейронная сеть для улучшения качества медиаконтента.

Результаты проведенного исследования могут быть применены при реализации высоконагруженных систем, работающих с пользовательским контентом и сторонних систем, связанных с потоковой обработкой мультимедиа.

					ПГУ1.090401.12.001 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата				
Разраб.	Лукин В.Д.			26.05.16	Конвейер обработки видеофайлов для веб-приложения Пояснительная записка	Лит.	Лист	Листов
Провер.	Зинкин С.А.			26.05.16			2	88
Н. Контр	Бычков А.С.			26.05.16		ФВТ, каф. ВТ гр. 24ВВИм1		
Утверд.	Митрохин М.А.			26.05.16				

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ.....	6
ВВЕДЕНИЕ.....	7
1 Исследование нейросетевых моделей для задач улучшения качества видео в контексте конвейера обработки видеофайлов	9
1.1 Возможные сценарии и алгоритмические подходы к улучшению качества видео	9
1.1.1 Цифровой шум изображения	10
1.1.2 Смазывание движения изображения.....	12
1.1.3 Блочность при компрессии кодека	14
1.1.4 Потеря текстур при компрессии кодека.....	16
1.1.5 Низкая исходная частота кадров и разрешение	18
1.2 Нейросетевые подходы к улучшению качества видео	22
1.2.1 Использование CNN для задач улучшения качества.....	22
1.2.2 Использование GAN для задач улучшения качества.....	24
1.2.3 Использование моделей GAN обратного восстановления.....	26
1.3 Отбор нейросетевых решений для конвейера обработки	27
1.3.1 Отбор нейросетевых решений	28
1.3.2 Дообучение отобранного решения	31
1.4 Формирование требований к модулю нейросетевой обработки	35
2 Проектирование архитектуры конвейера с модулем нейросетевой обработки	36
2.1 Общая архитектура конвейера	36
2.2 Проектирование структуры событий и очередей	41
2.3 Проектирование модулей подготовки контента для обработки	45
2.4 Проектирование модуля нейросетевой обработки	47
2.5 Механизмы обеспечения отказоустойчивости	50
3 Реализация архитектуры конвейера с модулем нейросетевой обработки..	53
3.1 Обоснование выбора технологического стека	53
3.2 Реализация транспортной структуры для обмена событиями	57
3.2.1 Издатель события.....	57

3.2.2 Потребитель события	58
3.2.3 Система событий и сериализация	59
3.2.4 Интеграция транспортной структуры с сервисами конвейера	60
3.3 Реализация модулей подготовки контента для обработки	61
3.3.1 Сервис управления задачами конвейера.....	62
3.3.2 Сервис отделения аудиодорожки.....	63
3.3.3 Сервис деления видеофайла на фрагменты	64
3.3.4 Сервис деления фрагментов на фреймы.....	65
3.3.5 Сервис объединения обработанных фреймов во фрагменты	66
3.3.6 Сервис объединения фрагментов в конечный видеофайл	67
3.3.7 Сервис объединения видеофайла с аудиодорожкой	68
3.4 Реализация модуля нейросетевой обработки	69
3.4.1 Инициализация и загрузка модели.....	70
3.4.2 Пакетная обработка фреймов.....	72
3.4.3 Обработка ошибок во время работы	74
3.5 Реализация механизмов обеспечения отказоустойчивости	75
3.5.1 Гарантированная доставка и идемпотентность обработки.....	75
3.5.2 Специфическая устойчивость Neurolizator и мониторинг.....	76
4 Оценка эффективности реализованной системы	78
4.1 Исследование метрик нейронной сети.....	78
4.1.1 Методика исследования метрик	78
4.1.2 Результаты сравнения	79
4.1.3 Анализ стабильности результатов.....	80
4.2 Исследование метрик сервисов подготовки	82
4.2.1 Методика исследования метрик	82
4.2.2 Результаты сравнения	82
4.3 Результаты работы реализованной системы.....	84
4.3.1 Соответствие функциональным требованиям	85
4.3.2 Соответствие нефункциональных требований	86
ЗАКЛЮЧЕНИЕ	87
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	88

ПРИЛОЖЕНИЕ А – Листинг программы	89
ПРИЛОЖЕНИЕ Б – Презентация.....	118

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяются следующие сокращения и обозначения:

API (Application Programming Interface) – программный интерфейс приложения

AVC (Advanced Video Coding) – стандарт сжатия видео H.264

CNN (Convolutional Neural Networks) – сверточная нейронная сеть

EDSR (Enhanced Deep Super Resolution) – продвинутая модель CNN для задач SR

ESRGAN (Enhanced Super Resolution Generative Adversarial Network) – продвинутая модель GAN для задач SR

GAN (Generative Adversarial Network) – генеративно-сопоставительная нейронная сеть

GPU (Graphics Processing Unit) – графический процессор

SR (Super Resolution) – задачи суперразрешения

SRCNN (Super Resolution Convolutional Neural Networks) – продвинутая модель CNN для задач SR

SRGAN (Super Resolution Generative Adversarial Network) – продвинутая модель GAN для задач SR

FI (Frame Interpolation) – задачи интерполяции кадров

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

VDSR (Very Deep Super Resolution) – продвинутая модель CNN для задач SR

VGG (Visual Geometry Group) – тип CNN модели для классификации изображений

LPIPS (Learned Perceptual Image Patch Similarity) – метрика для сравнения признаков изображений

NIQE (Natural Image Quality Evaluator) – метрика для сравнения признаков изображений без эталона

RRDB (Residual-in-Residual Dense Block) – остаточные в остаточных плотные блоки

ВВЕДЕНИЕ

Использование конвейеров обработки данных в любой области доработки мультимедийного контента является надежным способом декомпозиции и распараллеливания вероятного пула доступных задач. В контексте высоконагруженного веб-приложения роль и ответственность конвейера за обработку подобных задач увеличивается из-за появления дополнительных требований к повышенной отказоустойчивости, временной изоляции исполнения задач и балансу между качеством и скоростью обработки данных. При работе с видеофайлами описанные требования выходят на первый план из-за ресурсоемкости любых процессов, связанных с обработкой видео.

Вместе с тем объем видеоконтента, которому необходима качественная обработка только растет из-за развития интернета и смежных технологий, что приводит к поспешному расширению уже существующих решений. Стриминговые платформы, социальные сети, сервисы видеохостингов и корпоративные системы видеоконференцсвязи стараются улучшить собственные средства и интегрировать новые функции для комфорта пользователей, но не всегда это происходит благополучно и своевременно.

Помимо попыток интеграции различных инструментов с дальнейшей их совместимостью на сложность технической поддержки существующих конвейеров обработки влияет тенденция скорого внедрения нейронных сетей различных типов. В фундаментальную и зачастую монолитную архитектуру конвейера встраивается вычислительно-затратный механизм, который не был учтен на этапе проектирования данного решения. Это приводит к большим задержкам в сторонних обработчиках и нехватке ресурсов для других инструментов, что вызывает недовольство у конечного потребителя.

К критике, пользователя также приводит низкое качество полученного результата работы нейронной сети. Чаще всего, при быстром внедрении нейросетевых решений не хватает временных или вычислительных ресурсов на обучение собственной модели или полноценной адаптации используемой,

что иногда может приводить к потере части аудитории пользователей. Использование микросервисного подхода и самостоятельное обучение используемой модели могло бы привести к более продуктивным результатам работы и положительному пользовательскому опыту.

Целью магистерской диссертации является исследование, проектирование, реализация и оценка эффективности полученной микросервисной архитектуры отказоустойчивого событийно-ориентированного конвейера обработки видеофайлов для веб-приложения.

Результаты проведенного исследования могут быть применены при реализации высоконагруженных систем, работающих с пользовательским контентом и сторонних систем, связанных с потоковой обработкой мультимедиа.

1 Исследование нейросетевых моделей для задач улучшения качества видео в контексте конвейера обработки видеофайлов

Перед проектированием и разработкой конвейера обработки видеофайлов необходимо провести исследование внутреннего устройства доступных реализаций механизмов улучшения качества видео. Данный подход поможет корректно выбрать и использовать реализованный способ, а также скорректировать требования к вычислительным ресурсам системы для ввода в эксплуатацию конвейера обработки. Исследование также позволит предварительно выявить этапы подготовки медиаконтента к обработке и определить наиболее уязвимые места при проектировании подобного конвейера. Кроме нейросетевых подходов к задаче улучшения качества видео требуется рассмотреть и алгоритмические подходы к решению. Это необходимо для сопоставления целесообразности затрат ресурсов на алгоритмическую и нейросетевую обработку.

1.1 Возможные сценарии и алгоритмические подходы к улучшению качества видео

Задача улучшения качества видеоданных зачастую не является монолитной и делится на несколько взаимосвязанных подзадач, каждая из которых имеет собственный алгоритм решения. Это позволяет применить специализированные архитектуры под каждую деградацию и оптимизировать процесс подхода к их устранению. В контексте обработки пользовательского контента плохого качества в веб-приложении, стандартными сценариями являются обработка следующих деградаций:

- шум матрицы. Является искажением на цифровом изображении в виде хаотичных точек разной яркости цвета из-за случайных сигналов на фотосенсоре при фиксации видео. Одна из наиболее сложных подзадач, поскольку шумовая компонента неотделима от полезного высокочастотного сигнала. Требует от алгоритма одновременного сохранения текстур и подавления случайной помехи,

– смазывание движения. Является эффектом размытия быстро движущихся объектов или фона, возникающим при смене положения фиксируемого объекта за время отсутствия фиксации. Как и устранение шума матрицы относится к числу трудноразрешимых задач из-за необходимости восстановления информации, безвозвратно утраченной при интегрировании сигнала за время экспозиции,

– блочность при компрессии кодека. Является эффектом потери информации при сжатии изображения, где оно распадается на видимые квадратные блоки переменного размера. Устранение блочности имеет среднюю вычислительную сложность, но устранение требует от алгоритма способность различать границы реальных объектов и искусственные разрывы на стыках блоков кодирования,

– потеря текстур при компрессии кодека. Является эффектом потери информации при деградации текстурных областей изображения, где оно теряет контраст и превращается в соседние регионы одного спектра. Самая сложная подзадача из перечисленных из-за необходимости генерации новой высокочастотной информации,

– низкая исходная частота кадров и разрешение. Зачастую связано с техническими ограничениями фиксирующего устройства, а также с экономией памяти. Также относится к вычислительно-ёмким подзадачам из-за генерации новых пикселей и кадров, отсутствующих в исходном сигнале.

Описанные сценарии деградаций являются стандартными и описания их решения должны быть учтены при проектировании конвейера, иначе могут возникнуть проблемы при декомпозиции и делегировании подзадач.

1.1.1 Цифровой шум изображения

Шум матрицы или цифровой шум изображения является фундаментальным и неустранимым фактором при работе любого фотосенсора. Если рассматривать модель этого явления, то можно сделать вывод, что это пространственно-временная вариация яркости пикселя, не связанная с

фотометрией сцены, а порожденная самим регистрирующим устройством (рисунок 1).



ица

Рисунок 1 – Шум матрицы на изображении

Исходя из этого определения, можно составить следующую математическую модель:

$$I_{\text{наблюдаемое}}(x, y) = I_{\text{действительное}}(x, y) + \eta(x, y) \quad (1)$$

где (x, y) – пространственные координаты пикселя;

$I_{\text{наблюдаемое}}(x, y)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y)$ – действительное значение интенсивности пикселя;

$\eta(x, y)$ – случайная величина искажения в данном пикселе.

Подзадача подавления шума сводится к оценке неизвестного действительного сигнала по зашумленной наблюдаемой реализации. Из-за отсутствия корректной начальной информации наиболее обоснованным подходом является использование адаптивной пространственно-временной фильтрации. Это связано с тем, что случайная величина шума не коррелируется с соседними кадрами и пикселями, в то время как полезный сигнал пикселя хорошо коррелируется. В подходе адаптивной

пространственно-временной фильтрации есть несколько алгоритмов, позволяющих решить данную подзадачу:

- пространственная фильтрация одиночных кадров. Обладает хорошим результатом на статичных изображениях, но игнорирует временную избыточность обрабатываемого видео. Это приводит к игнорированию межкадровой корреляции и увеличению радиуса фильтра для лучшего подавления шума (что приводит к нежелательному сглаживанию). Алгоритм только частично подходит для видео. Является неоптимальным решением,

- блочное 3D-шумоподавление. Анализирует цепочку соседних кадров и создает пространственно-временные блоки пикселей для применения на последующих кадрах. Алгоритм идеально подходит для обработки видеоконтента, но не годится для работы в реальном времени и на системах с ограниченными ресурсами. Также имеет слабую деградацию при последующих вычислениях. Не подходит для использования в конвейере из-за высоких требований к вычислительным ресурсам,

- временное усреднение с детектором движения. Рекурсивно накапливает сигнал с экспоненциальным скользящим средним. Обладает хорошим быстродействием, но имеет сильную зависимость от накопленного сигнала. Уязвим к ошибкам детектора движения, однако это чаще всего результат некорректной настройки.

Из перечисленных выше методов лучшим вариантом будет временное усреднение с детектором движения. Однако, данный алгоритм решения также имеет изъяны, главный из которых – зависимость от накопленного сигнала за единицу времени. Это будет сильно сказываться на потоковой обработке фреймов или дельта-кадров.

1.1.2 Смазывание движения изображения

Смазывание движения изображения возникает при смещении объекта на заметное расстояние за время экспозиции. Данные артефакты можно отнести к двум категориям: линейные и нелинейные. Линейные обладают прослеживаемым вектором движения, а нелинейные имеют более сложную

труднопредсказуемую траекторию. Линейное смазывание изображения продемонстрировано на рисунке 2.



Рисунок 2 – Линейное смазывание движения на изображении

По представленной выше характеристике, можно составить следующую математическую модель:

$$I_{\text{наблюдаемое}}(x, y) = (I_{\text{действительное}} * h)(x, y) + \eta(x, y) \quad (2)$$

где (x, y) – пространственные координаты пикселя;

$I_{\text{наблюдаемое}}(x, y)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y)$ – действительное значение интенсивности пикселя;

$h(x, y)$ – ядро смазывания движения;

$\eta(x, y)$ – случайная величина искажения в данном пикселе.

Устранение смазывания сводится к восстановлению неизвестного действительного изображения по свертке искажения наблюдаемой размытой реализации. В отличие от задачи обработки цифрового шума, где сигнал и шум связаны аддитивно, здесь искажение имеет форму свертки, что переносит задачу в класс обратных некорректных. Прямое использование оператора свертки через деление в частотной области сильно усиливает имеющийся шум. Для решения данной задачи существует несколько алгоритмических подходов:

– инверсная фильтрация. Прямое обращение к частотной области исходного изображения с делением на свертку. Сложно применить данный метод на практике, так как ядро смазанности близко или равно нулю. Присутствие шума приводит к его усилению из-за обращения свертки. Не подходит для конвейера из-за трудностей практического применения и необходимости полного отсутствия шума на изображении,

– фильтр Винера. Регуляризованный вариант инверсной фильтрации, учитывающий спектральные плотности мощности сигнала и шума. Минимизирует среднеквадратичную ошибку восстановления и подавляет шум, но требует оценки отношения сигнала к шуму. Нужно много предварительной информации для следующего шага вычисления, но обладает небольшой вычислительной сложностью,

– фильтр Люси-Ричардсона. Алгоритм, строящий серию последовательных приближений с повышенной четкостью исходного изображения. Обладает высоким качеством полученного результата, однако требует большего количества вычислительных ресурсов из-за наличия нескольких приближений. Сильно привязан к числу итераций, что сказывается на получаемом результате.

Из перечисленных выше методов лучшим вариантом будет фильтр Винера, но он не сможет дать высокой производительности на конвейере. Для работоспособности данного метода необходимо точно знать ядро смазывания и максимально уменьшить поступающий шум. Данный метод может усилить или ослабить уже существующие сигналы изображения, что в некоторых случаях может привести лишь к ухудшению качества.

1.1.3 Блочность при компрессии кодека

Блочность при компрессии кодека возникает из-за фундаментального принципа работы видеокодеков, связанного с разбиением кадра на непересекающиеся блоки фиксированного или переменного размера, к которым применяется дискретное преобразование с квантованием частотных коэффициентов. Именно при квантовании частотных коэффициентов

происходит потеря информации, что влияет на формируемый результат. Стоит отметить, что матрица квантования сильно связана с используемым кодеком, как и с артефактами, возникающими при компрессии. Пример блочности при компрессии с кодеком H.264 (AVC) приведен на рисунке 3.



Рисунок 3 – Блочность при компрессии кодека H.264 на изображении
Математическая модель для данного артефакта может быть описана как:

$$I_{\text{наблюдаемое}}(x, y) = I_{\text{действительное}}(x, y) + \beta(x, y) \quad (3)$$

где (x, y) – пространственные координаты пикселя;

$I_{\text{наблюдаемое}}(x, y)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y)$ – действительное значение интенсивности пикселя;

$\beta(x, y)$ – сигнал блочности, возникший при квантовании блоков.

Устранение эффекта блочности при компрессии сводится к восстановлению сглаживания изображения на границах блоков пикселей, нарушенных независимым квантованием коэффициентов дискретного косинусного преобразования в смежных блоках. В отличие от подзадачи устранения смазывания движения изображения, где искажение имеет форму свертки, здесь деградация носит структурный характер. Она проявляется как регулярная сетка перепадов яркости, привязанная к фиксированной сетке разбиения кадра. Требуется восстановить утраченную межблочную корреляцию, но с учетом сложности отличия искусственных границ блоков от

реальных контуров объектов. Для решения данной подзадачи существует несколько алгоритмических подходов:

- петлевой фильтр кодека. Данное решение является встроенным стандартом адаптивного фильтра, сглаживающего границы блоков в зависимости от уровня квантования и разности пикселей. Имеется во всех популярных кодеках. Обладает низкой вычислительной сложностью, имеет ограничения по восстановлению текстур, утраченных внутри блока пикселей. Хорошо подходит для конвейера,

- пространственная адаптивная фильтрация. Заключается в применении сглаживающих фильтров с переменной силой, зависящей от локальных градиентов яркости и расстояния от границы блока. Также обладает низкой вычислительной сложностью, но может привести к чрезмерному сглаживанию реальных контуров и сохранению заметных артефактов при некорректной настройке. Частично подходит для конвейера.

Из описанных ранее методов для отказоустойчивого конвейера видеообработки подходят оба. При относительной редкости возникновения и низкому требованию к ресурсам для исправления данной деградации можно позволить использование обоих методов. Возможно, во время проектирования конвейера какой-то из перечисленных методов не будет задействован.

1.1.4 Потеря текстур при компрессии кодека

Потеря текстур при компрессии кодека вызывает полное исчезновение части изображения при кодировании. Механизм разложения текстур заложен в принципы работы современных кодеков. Завершив дробление кадра на блоки и применив дискретное преобразование, кодировщик кодека выполняет квантование частотных коэффициентов. Низкочастотные компоненты (крупные детали изображения) квантуются с малым шагом и сохраняются. Высокочастотные компоненты (небольшие текстуры, зернистость, микрорельеф), квантуются с небольшими искажениями, либо обнуляются полностью. Данное поведение не случайно, так как кодек жертвует текстурами из-за низкого восприятия высокочастотных искажений человеком. Но в

некоторых случаях это может привести к потере значимых параметров изображения. Пример потери текстур при компрессии кодеком H.264 (AVC) приведен на рисунке 4.

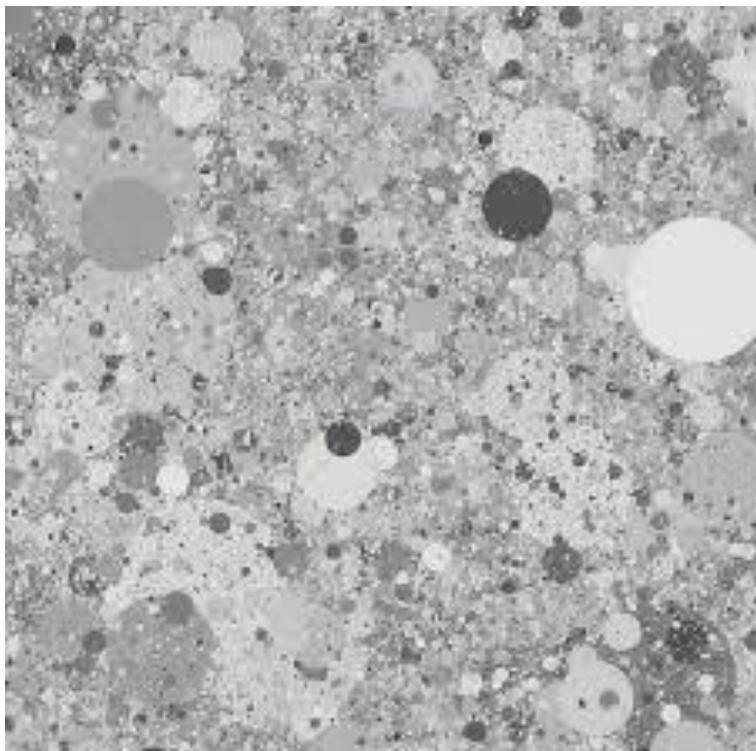


Рисунок 4 – Потеря текстур при компрессии кодека H.264 на изображении

Математическую модель данной деградации можно сформулировать как:

$$I_{\text{наблюдаемое}}(x, y) = I_{\text{действительное}}(x, y) - T(x, y) \quad (4)$$

где (x, y) – пространственные координаты пикселя;

$I_{\text{наблюдаемое}}(x, y)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y)$ – действительное значение интенсивности пикселя;

$T(x, y)$ – удаленная необратимая компонента.

Данная модель схожа с артефактами, возникающими при блочности в контейнере кодека. За внешним сходством скрывается фундаментальная разница. Если сигнал блочности, возникший при квантовании блоков, можно оценить по границам блоков и подавить, то удаленную необратимую компоненту потери вернуть невозможно. Для ликвидации последствий

данного искажения требуется синтез новых данных. Из алгоритмических методов доступна только статистическая экстраполяция в частотной области.

Упомянутый метод позволяет предсказать значения обнуленных высокочастотных коэффициентов на основе сохраненных низкочастотных коэффициентов и известных корреляционных связей между частотными компонентами естественных изображений. Обладает низкой вычислительной сложностью, но способен восстановить лишь статически регулярные текстуры с предсказуемой частотной структурой. Любые сложные не реплицируемые структуры обработать невозможно. Данный алгоритм не подходит для конвейера из-за своей ограниченности в доступных вариантах обработки.

1.1.5 Низкая исходная частота кадров и разрешение

Низкое исходное разрешение видео и недостаточная частота кадров являются взаимосвязанными типами деградации видео, обусловленными либо техническими ограничениями регистрирующего устройства, либо целенаправленным снижением качества для экономии памяти. В обоих случаях часть пространственной или временной информации теряется необратимо на этапе создания видеопотока и не может быть восстановлена без привлечения дополнительных данных. Пример целенаправленного понижения разрешения приведен на рисунке 5.



Рисунок 5 – Целенаправленное понижение качества изображения

Механизм потери информации при снижении разрешения можно формализовать через оператор прореживания. Исходный кадр высокого разрешения проходит через низкочастотный фильтр, после чего с определенным шагом из него выбирается пиксель для результирующего кадра. При снижении частоты кадров из временной последовательности удаляются

целые кадры, и информация о движении объектов между сохраненными кадрами утрачивается.

Математическая модель описанной деградации для низкого разрешения можно сформулировать как:

$$I_{\text{наблюдаемое}}(x, y) = D_n(I_{\text{действительное}} * g)(x, y) \quad (5)$$

где (x, y) – пространственные координаты пикселя;

$I_{\text{наблюдаемое}}(x, y)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y)$ – действительное значение интенсивности пикселя;

g – низкочастотный сглаживающий фильтр;

D_n – оператор прореживания.

Для низкой частоты кадров модель формируется, используя временную область:

$$I_{\text{наблюдаемое}}(x, y, t) = D_T(I_{\text{действительное}}(x, y, t)) \quad (6)$$

где (x, y, t) – пространственные координаты пикселя в период времени;

$I_{\text{наблюдаемое}}(x, y, t)$ – наблюдаемое значение интенсивности пикселя;

$I_{\text{действительное}}(x, y, t)$ – действительное значение интенсивности пикселя;

D_T – оператор временной децимации, оставляющий кадры с шагом T .

Описанные выше модели достаточно схожи с рассмотренными ранее сценариями потери основной информации. В обоих случаях часть исходной информации безвозвратно удаляется из сигнала. Но есть и важное отличие: если при потере текстур обнулению подвергаются высокочастотные коэффициенты внутри блока и их можно частично восстановить по сохраненным низкочастотным, то при децимации удаляются пиксели или целые кадры без какой-либо частотной избирательности. Текущее восстановление требует полноценного синтеза отсутствующей информации.

Для решения проблемы с низким разрешением исходного изображения можно использовать следующие алгоритмические методы:

– бикубическая интерполяция. Закономерное развитие методов стандартной интерполяции и билинейной интерполяции. Для вычисления

нового пикселя используются квадрат известных пикселей 4 на 4, а веса для вычисления задаются кубическим полиномом. Имеет высокую вычислительную сложность и требует большого количества итераций. Также из-за отсутствия понимания дорабатываемой сцены игнорирует тени и освещение на изображении (рисунок 6),

- фильтр Ланцоша. Частично похож на метод бикубической интерполяции, но использует увеличенный размер ядра поиска. Обладает увеличенным потреблением ресурсов, но представляет наилучшую резкость среди всех интерполяционных методов. Периодически может создавать засветы или затемнения из-за колебаний используемой функции. Частично подходит для использования в конвейере,

- метод локального самоподобия. В отличие от двух предыдущих интерполяционных подходов не просто вычисляет среднее значение соседей, а пытается восстановить высокочастотные детали по небольшим повторяющимся фрагментам. Потребляет небольшое количество вычислительных ресурсов, но жестко привязан к повторяемым паттернам изображения. Также очень чувствителен к шуму. Не подходит для использования в конвейере.



Рисунок 6 – Артефакты бикубической интерполяции

Исходя из описанного пула методов, лучше всего в конвейере подойдет фильтр Ланцоша. Он требует большее количество ресурсов, но позволяет

получить наивысшее качество среди алгоритмических методов. Теоретически, можно было бы применить метод локального самоподобия, но как показали локальные тесты, данный метод чрезвычайно сильно привязан к повторяющимся структурам, что не всегда подходит в рамках улучшения качества.

После решения проблемы с низким исходным разрешением можно приступить к деградации, связанной с недостаточным количеством кадров. Решение данной подзадачи можно осуществить при помощи следующих алгоритмических методов:

- блочная интерполяция с компенсацией движения. Метод синтезирует промежуточный кадр между двумя существующими и при помощи оценки движения и последующей интерполяции вдоль траекторий. Демонстрирует хороший параллелизм из-за разбиения изображения на блоки, но создает частичные артефакты блочности. Это происходит из-за стыков векторов движения,

- вариационная интерполяция на основе оптического потока. Метод синтезирует промежуточный кадр между двумя существующими путем глобальной минимизации функционала энергии, оценивающего поле движения, и последующей интерполяции вдоль имеющихся траекторий. В отличие от блочного подхода, поле оптического потока вычисляется не независимо для каждого блока, а совместно для всего кадра с дополнительным условием гладкости, исключающим резкие перепады векторов между соседними пикселями. Демонстрирует высокую точность на плавных деформируемых движениях и не порождает блочность. Вычислительная сложность метода высока и требует десятков итераций, что ограничивает его применимость.

Из предложенных алгоритмов больше всего подходит вариационная интерполяция на основе оптического потока, но стоит обратить внимание на требование данного метода к вычислительным ресурсам, что заметно ограничивает его применение. Возможно использование и второго метода -

блочной интерполяции с компенсацией движения, однако он может создать некоторое количество артефактов при частом использовании.

Для отсутствующих или вероятных вариантов методов решения подзадач обработки деградаций необходимо найти нейросетевой аналог для полноценного использования в высоконагруженном контейнере. Скорее всего, найденные нейросетевые подходы позволят компенсировать недостатки алгоритмических методов.

1.2 Нейросетевые подходы к улучшению качества видео

Проведенный анализ алгоритмических методов устранения деградаций выявил наличие принципиальных ограничений у классических подходов улучшения качества изображения. Они либо неспособны в полной мере восстановить безвозвратно утраченную информацию видеоконтента, либо требуют точного знания модели деградации и ядра свертки, что редко достижимо при обработке пользовательского контента, поступающего через веб-приложение. Из-за этого стоит рассмотреть возможность обращения к нейросетевым архитектурам, способным обучаться и адаптироваться к поступающим данным. Кроме поиска и анализа найденных архитектур, необходимо сравнить их между собой, а также выбрать лучшие реализации по соотношению получаемого качества к затраченным ресурсам.

1.2.1 Использование CNN для задач улучшения качества

Одними из первых нейросетевых подходов для улучшения качества видео, превзошедшими классические алгоритмы в данном типе обработки, стали сверточные нейронные сети. Ключевым преимуществом данного подхода к обработке является способность автоматического форматирования иерархии признаков. Многослойная архитектура формирует иерархию от простых градиентов яркости на нижних слоях до семантически значимых текстур и контуров на верхних полученных слоях (рисунок 7).

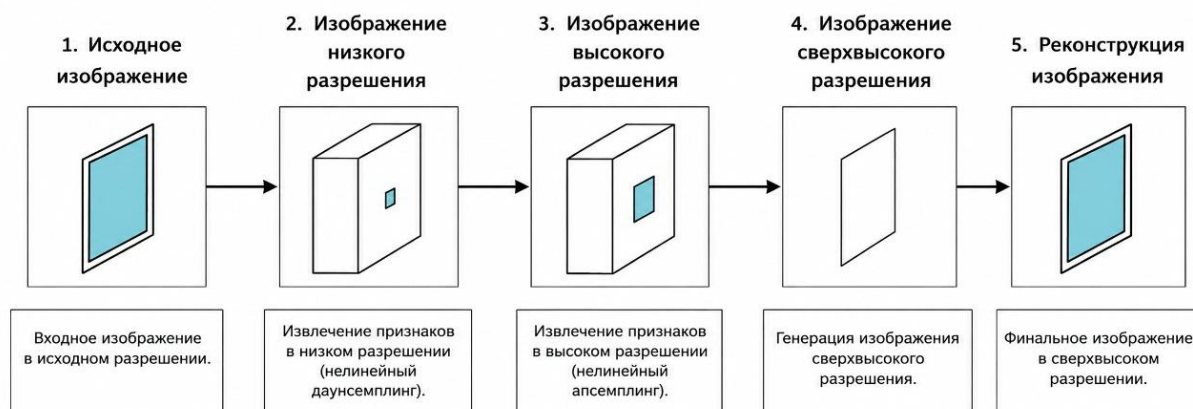


Рисунок 7 – Архитектура нейронной сети CNN

Модель SRCNN предназначенная для выполнения задач SR, продемонстрировала возможности неглубокой четырехслойной сети на задаче устранения шума. Выяснилось, что данная нейронная сеть превосходит метод бикубической интерполяции по метрике соотношения сигнала к шуму (рисунок 8), что привело к созданию более продвинутых архитектур типа VDSR и EDSR [1].



Рисунок 8 – Сравнение результатов обработки бикубической интерполяции и SRCNN

В перечисленных архитектурах увеличили глубину нейронной сети и внедрили механизмы остаточного обучения. Он позволяет нейронной сети фокусироваться на высокочастотной составляющей исходного изображения.

Но у моделей CNN есть серьезный недостаток обработки. Из-за оптимизации по функциям потерь для каждого пикселя, они выдают усредненный результат по всем возможным правдоподобным решениям. Это приводит к появлению чрезвычайного сглаживания, неестественно выглядящим текстурам и потере ранее доступной

низкочастотной составляющей. Если рассматривать данный тип сетей для задач, рассматриваемых ранее (где необходим синтез новой информации), то данный недостаток становится критическим. Использование такой архитектуры в конвейере лучше алгоритмических подходов, но все ещё сомнительно из-за возникающих артефактов.

1.2.2 Использование GAN для задач улучшения качества

Генеративно-сопоставительные сети выступают в роли альтернативного типа архитектуры нейронных сетей для выполнения задачи SR. Базовая архитектура рассматриваемого типа делится на две состязательные части:

- генератор. Восстанавливает исходное изображение и отдает на оценку дискриминатору,
- дискриминатор. Отличает восстановленные изображения от реальных образцов в высоком качестве.

На рисунке 9 описана представленная выше архитектура нейронной сети GAN. Также стоит отметить, что данный тип нейронной сети использует состязательную функцию потерь, что заставляет генератор создавать не близкую пиксельную копию, а визуально правдоподобный результат, восстанавливая текстуры и другие составляющие изображения высокой четкости.

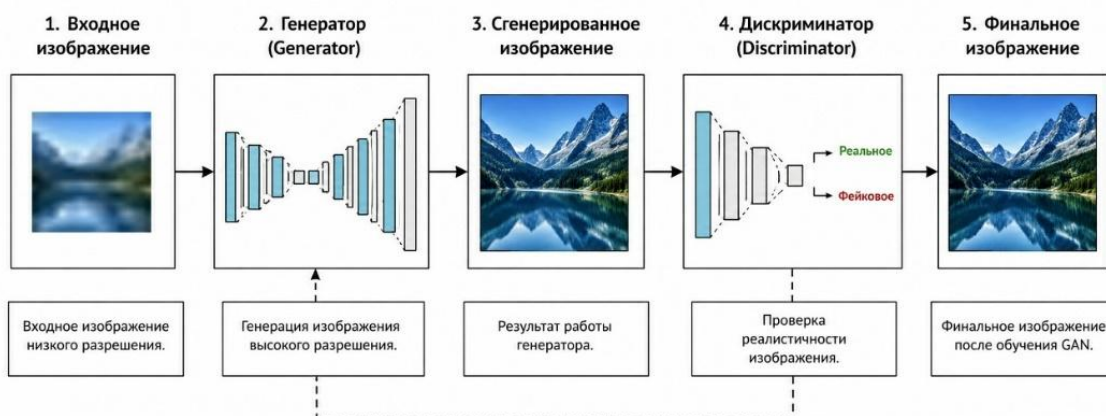


Рисунок 9 – Архитектура нейронной сети GAN

Следующим этапом развития данной архитектуры стала модель SRGAN, объединяющая состязательную потерю с перцептивной потерей, которая

вычисляется на картах признаков предобученной сети VGG. Данное улучшение позволило получить изображения с фотореалистичными текстурами, которые при оценке показателей LPIPS и NIQE получились лучше, чем у моделей архитектуры SRCNN [2].

Текущий этап развития имеет эталон в виде архитектуры ESRGAN, внедрившей три усовершенствования:

- RRDB или остаточные плотные блоки. Позволяют извлекать более неочевидные признаки из изображения,

- релятивистский дискриминатор. В отличие от обычного, оценивает не абсолютную реалистичность, а реальность изображения по сравнению с сгенерированным,

- улучшенная перцептивная потеря. Для расчета воспринимаемого различия между восстановленным и эталонным изображением используются промежуточные данные до функции активации VGG.

Данные модификации заметно улучшили качество получаемого результата (рисунок 10), но не решили проблему ограничения данных моделей для конкретной модели деградации, на которой они обучаются.



Рисунок 10 – Сравнение результатов обработки SRGAN и ESRGAN

Из-за того, что в конвейере будет обрабатываться пользовательский контент, то варианты деградации получаемых данных может быть разным. Необходима архитектура, учитывающая несколько вариантов деградации для получения максимального качества.

1.2.3 Использование моделей GAN обратного восстановления

Обозначенная ранее проблема моделей GAN связана с обучением и тестированием на парах изображений с фиксированной моделью деградации. В пользовательском контенте могут встречаться достаточно сложные цепочки деградаций, состоящие из множества трудно обратимых или необратимых ухудшений. Именно из-за этого возникают сложности с использованием алгоритмических подходов в конвейере, порядок устранения данных деградаций также важен.

Точный результат в данных сценариях не может быть получен, и задача улучшения качества переходит в разряд обратного восстановления. Существуют модели, которые способны работать с подобными сценариями. В архитектуре GAN к таким моделям относится Real-ESRGAN, наследующий архитектурные приемы ESRGAN и использующий генератор на базе RRDB и дискриминатор U-Net. Это обеспечивает очень высокое качество восстанавливаемых текстур (рисунок 11).



Рисунок 11 – Сравнение результатов обработки ESRGAN и Real-ESRGAN

Вместо однократного применения бикубической интерполяции, модель применяет несколько стадий деградаций, включая размытие, прореживание, добавление шума и сжатие с заниженными параметрами. Во время обучения необходимо пройти две стадии:

- обучение Real-ESRNet с потерей L1 для обеспечения стабильной основы,

- обучение Real-ESRGAN с добавлением состязательной и перцептивной потери.

Такой подход к обучению, не смотря на сложность, позволяет обеспечить модели устойчивость процесса. Если сравнивать данный подход с обучением и использованием диффузионных моделей, то диффузионные модели будут иметь небольшое превосходство над GAN-методами, но требовать гораздо больше вычислительных ресурсов.

Следует сравнить лидирующие архитектуры рассматриваемых типов и выбрать наиболее подходящую по качеству и производительности. Также стоит рассмотреть возможность дообучения имеющейся модели выбранной архитектуры.

1.3 Отбор нейросетевых решений для конвейера обработки

Перед отбором конкретных нейросетевых решений стоит определить критерии отбора. Это позволит оценить пригодность применения рассматриваемых реализаций для интеграции в конвейер обработки видеофайлов:

- модель должна иметь способность работать в режиме слепого восстановления. Это ключевой аспект работы в конвейере из-за различий поступающего пользовательского контента. Модели с предварительной настройкой не могут быть использованы,

- модель должна иметь способность синтеза текстур. Подзадачи устранения потери текстур при компрессии и повышения разрешения с восстановлением частоты кадров требуют генерации новой высокочастотной информации, принципиально отсутствующей в исходном сигнале. CNN-модели, оптимизированные по пиксельным потерям, склонны к чрезмерному сглаживанию и неспособны восстановить реалистичные текстуры. Данному критерию удовлетворяют только GAN-архитектуры и диффузионные модели [3],

– модель должна иметь допустимую сложность для потоковой обработки. Конвейер обработки видеофайлов является компонентом высоконагруженного веб-приложения, и время обработки одного видеофайла не должно превышать разумных пределов, определяемых пользовательским ожиданием. Диффузионные модели, несмотря на их превосходство по качеству, требуют десятков итераций обратного диффузионного процесса для одного кадра, что ставит под сомнение их использование при потоковом сценарии,

– модель должна иметь доступные предобученные веса и возможность нового обучения. Это необходимо при использовании конвейера обработки под специфический пользовательский контент, попадающий в одну стилистику.

1.3.1 Отбор нейросетевых решений

Завершив формирование требований, можно приступить к сравнению ранее описанных моделей по выдвинутым критериям. В таблице 1 приведено критериальное сравнение лучших архитектур каждого типа:

Таблица 1 – Сравнение моделей улучшения качества

Критерий	SRCNN/VDSR	Real-ESRGAN	Диффузионные модели
Слепое восстановление	Отсутствует	Присутствует	Присутствует
Синтез текстур	Отсутствует	Присутствует	Присутствует
Воспроизводимость обучения	Присутствует	Присутствует	Частично присутствует
Потоковая производительность	Присутствует	Присутствует	Отсутствует

По полученной таблице можно сделать вывод, что модели архитектурного типа CNN не подходят для конвейера по описанным ранее причинам. Что касается диффузионных моделей, то их применение может быть оправдано полученным качеством. Необходимо провести экспериментальное исследование.

Исследование будет произведено на стенде с 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, с разным типом контента. Преимущественно использовалась отечественная и зарубежная мультипликация, а также реальные видеозаписи культурных объектов Пензы. Выборка обусловлена наличием ранее улучшенных вариантов для улучшенной фиксации различий результата.

В качестве непосредственных предметов для исследования были выбраны соответствующие модели:

- Real-ESRGAN x4+,
- SR3+.

Выбор данных моделей обусловлен частичной схожестью внутренних параметров и использованием по умолчанию для обозначенных архитектур. Метрики были извлечены при помощи анализа финальных результатов пакетом IQA-PyTorch. Полученные показатели приведены на рисунке 12.

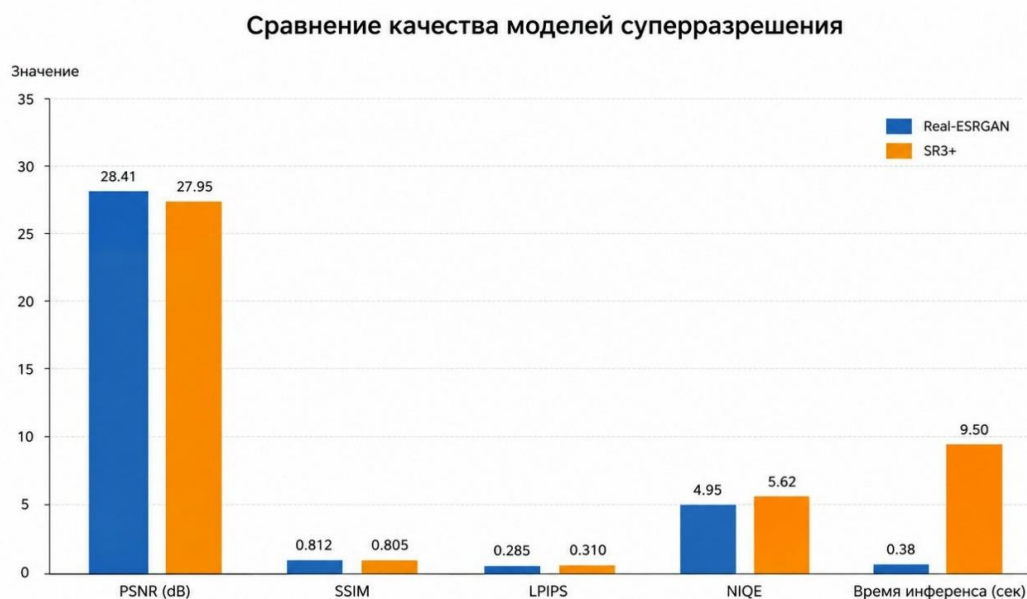


Рисунок 12 – Результат тестирования Real-ESRGAN x4+ и SR3+

Таблица 2 – Сравнение показателей работы моделей Real-ESRGAN x4+ и SR3+

Метрика	SRCNN/VDSR	Real-ESRGAN
PSNR (dB)	28.41	27.95
SSIM	0.812	0.805
LPIPS	0.285	0.310
NIQE	4.95	5.62
Время инференса (сек)	0.38	9.50

Результаты, приведённые в таблице 2, демонстрируют частичное превосходство модели Real-ESRGAN x4+ над диффузионным аналогом SR3+ в контексте сформированных выше задач. Real-ESRGAN показал более высокие значения по метрикам точности (PSNR, SSIM), что свидетельствует о лучшем попиксельном соответствии эталону. Однако наиболее значимый разрыв наблюдается по перцептивным метрикам LPIPS и NIQE. Это подтверждает гипотезу о том, что GAN-архитектура, оптимизированная под синтез текстур, генерирует визуально более естественные и приятные для человеческого глаза изображения. Диффузионная модель SR3+ в ряде случаев демонстрировала излишнюю «гладкость» или, напротив, неестественные высокочастотные артефакты, что негативно сказалось на её оценке.

С точки зрения применимости в конвейере обработки видеоконтента, превосходство Real-ESRGAN является абсолютным. Среднее время обработки одного кадра составило 0.38 секунды против 9.5 секунд у SR3+. Однако полученные данные позволяют заключить, что модель Real-ESRGAN не только чаще всего обеспечивает лучшее визуальное качество, но и является единственным практичным решением для интеграции в событийно-ориентированный конвейер обработки видео.

1.3.2 Дообучение отобранного решения

Выбор модели Real-ESRGAN x4+ в качестве базовой реализации для модуля нейросетевой обработки не означает максимальную готовность к обработке пользовательского контента. Универсальная модель обучалась на достаточно большом, но ограниченном наборе данных DF2K и OST, проявляя наилучшие результаты на изображениях, близких к фотографическому контенту. Конвейер частично должен быть ориентирован и на обработку мультипликации, которая входит в одну из наиболее частых зон запросов пользователей (рисунок 13). Текущая модель хорошо справляется с подобным контентом, однако можно попробовать дообучить данную модель и получить более актуальный результат.



Рисунок 13 – Сценарий возможной обработки мультипликации

Процедура дообучения для модели Real-ESRGAN, описанная в документации, поддерживает два сценария:

- дообучение с генерацией деградированных изображений. Самый простой метод, требующий наличие только изображения высокого качества, которое будет ухудшено, а затем улучшено моделью,
- дообучение на собственных парных данных. Классический метод обучения с наличием пары качественных и оригинальных изображений.

Не смотря на простоту дообучения с генерацией деградированных изображений, дообучение на парных данных является наилучшим вариантом для текущего сценария. Это связано с датасетом, на котором будет сеть будет производить дообучение.

Сформированный датасет состоит из фреймов видео, улучшенных на различных диффузионных и GAN-моделях. Видео контент, формирующий основу датасета – это отреставрированные версии мультипликационного фильма «Том и Джерри», которые доступны без ограничения на использование. Также, в состав датасета вошли советские отреставрированные мультфильмы, улучшенные копии старых аниме и реальная видеосъемка в высоком качестве. Из соответствующих видео были извлечены посекундные фреймы, составляющие датасет. Ухудшенные данные извлекались из оригинальных материалов по метрике временного совпадения (рисунок 14).



Рисунок 14 – Оригинальное и улучшенное изображение из датасета

Процесс дообучения можно выполнить при помощи встроенных инструментов нейронной сети. После формирования папок с улучшенными и оригинальными фреймами, был сгенерирован информационный файл формата txt, содержащий список путей к изображениям при помощи встроенного скрипта `generate_meta_info_pairdata.py`. Далее были загружены модели генератора и дискриминатора, и выставлены стандартные параметры для дообучения. По предварительной экспериментальной оценке, было выяснено, что обработка 500 000 итераций дообучения на 2х экземплярах NVIDIA GeForce RTX 5070 Ti займет около 10 дней, поэтому было принято решение о разделении выборки на эпохи по 50 000 итераций для ежедневного дообучения.

После запуска скрипта `train.py` с параметром `--auto_resume` дообучение на каждом этапе происходило, в среднем, в течение 20 часов и сократило срок

дообучения до 8 дней. После каждой эпохи обучения производилась проверка и фиксация метрик при помощи IQA-PyTorch.

Таблица 3 – Итерационные показатели дообучаемой модели

Эпоха	PSNR	SSIM	LPIPS	NIQE	Время инференса (сек)
0	27.95	0.805	0.310	5.62	9.50
1	28.04	0.810	0.301	5.49	9.51
2	28.11	0.814	0.294	5.39	9.52
3	28.17	0.817	0.288	5.31	9.51
4	28.22	0.820	0.283	5.24	9.50
5	28.26	0.822	0.279	5.18	9.50
6	28.30	0.824	0.276	5.13	9.49
7	28.33	0.826	0.273	5.09	9.47
8	28.35	0.827	0.271	5.06	9.46
9	28.37	0.828	0.269	5.03	9.40
10	28.39	0.829	0.267	5.01	9.41

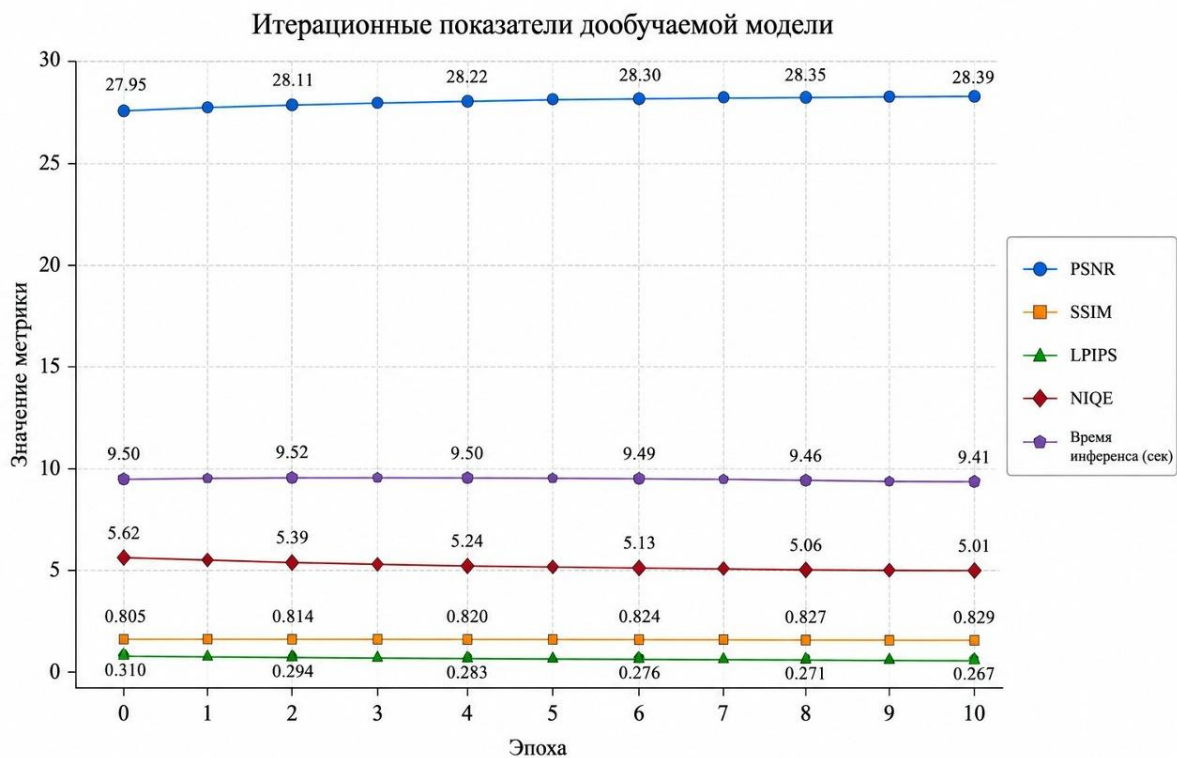


Рисунок 15 – Результат дообучения Real-ESRGAN x4+VM

Полученные показатели, отраженные в таблице 3 и на рисунке 15, показали рентабельность процедуры дообучения имеющейся модели. При дообучении получился постепенный и устойчивый рост PSNR, SSIM и NIQE. Периодически снижалась скорость обработки моделью, но эти колебания можно рассматривать как небольшую погрешность. Главное, что удалось улучшить характеристики улучшения качества изображения без потери скорости обработки. Это означает, что гипотеза об улучшении качества обработки модели при дополнительном обучении на выбранном датасете оказалась верной.

Таким образом была получена модифицированная модель Real-ESRGAN x4+VM, которую необходимо использовать в последующей реализации конвейера из-за доказанной эффективности работы. Но разработку конвейера можно начать только после планирования и составления требований к модулю нейросетевой нагрузки.

1.4 Формирование требований к модулю нейросетевой обработки

После выбора Real-ESRGAN в качестве базовой архитектуры нейросетевой обработки можно определить предварительные требования к инкапсулирующему её модулю. Данные требования имеют только функциональную составляющую. Нефункциональные требования будут определены при проектировании системы.

Текущие функциональные требования к реализуемому модулю нейросетевой обработки:

- слепое улучшение качества видео. Модуль должен принимать на вход видеофайл произвольного формата и разрешения, не требуя от отправителя метаданных о характере или параметрах деградации. Модель должна самостоятельно определять и компенсировать присутствующие искажения, включая шум, смазывание, артефакты компрессии и недостаточное разрешение,

- генерация высококачественных текстур. Выходной видеофайл должен содержать восстановленные текстурные области, визуально воспринимаемые как естественные и соответствующие содержанию проявляемой действительности. Недопустимо чрезмерное сглаживание,

- возможность дообучения модели. Модуль должен поддерживать процедуру дообучения на собственном наборе данных,

- покадровая обработка с возможностью учёта временной избыточности. Модуль должен обрабатывать видео как последовательность кадров, но с возможностью использования информации смежных кадров для обеспечения темпоральной согласованности и подавления временного шума. Данный аспект важен для устранения артефактов межкадрового мерцания, потенциально вносимых покадровой GAN-обработкой.

Описанные выше требования необходимо использовать на этапе проектирования и разработки, иначе могут возникнуть проблемы с выделением вычислительных ресурсов и организацией внутренних процессов.

2 Проектирование архитектуры конвейера с модулем нейросетевой обработки

После выполненного ранее исследования подходов улучшения качества видеоконтента и сформированных требований к модулю нейросетевой обработки можно перейти к проектированию архитектуры отказоустойчивого событийно-ориентированного конвейера. Проектирование необходимо начать с рассмотрения и обоснования возможных подходов к построению архитектурного фундамента системы. Важно учесть необходимость декомпозиции обработки задач улучшения качества с возможностью асинхронного выполнения подзадач. Выбранный архитектурный подход также должен обеспечить отказоустойчивость работы системы, временную изоляцию исполнения подзадач и регулировку баланса между качеством и временем исполнения конечной задачи.

2.1 Общая архитектура конвейера

Проектирование высоконагруженного отказоустойчивого конвейера обработки данных начинается с построения архитектурного фундамента системы, который определяет характер взаимодействия компонентов, способы масштабирования и устойчивость к отказам. Универсальные конвейеры мультимедийной обработки, такие как Rasbora или Netflix Video Encoding Service построены с использованием микросервисной архитектуры, что полностью оправдано ограничениями монолитной архитектуры организации (рисунок 16).

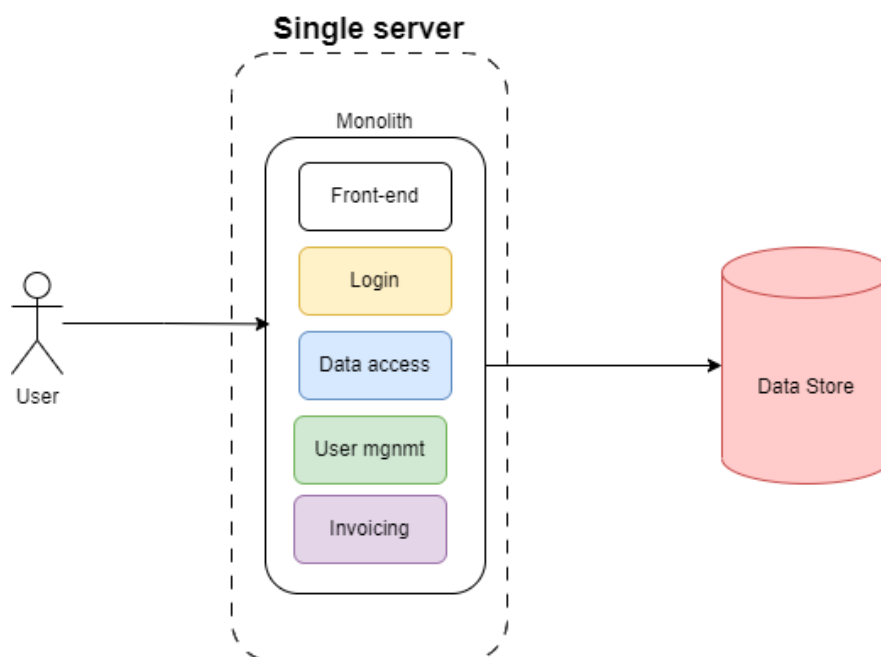


Рисунок 16 – Монолитная архитектура организации

Монолитная архитектура приводит к следующим возможным ограничениям организации внутреннего взаимодействия:

- чрезвычайная связанность этапов обработки. В конвейере, организованном при помощи данной архитектуры сбой на любом из этапов исполнения способен остановить весь процесс обработки и привести к потере промежуточных результатов,

- невозможность масштабирования. Архитектура не предполагает разделение любых подзадач на отдельные процессы и выполняет все в одном стеке. Это может привести к гонке за ресурсы, а также к невозможности кластеризации отдельных этапов [4],

- сложность внедрения новых инструментов. Рассматриваемая архитектурная организация не позволяет внедрить новые инструменты обработки без обратного влияния на реализованные решения. Из-за отсутствия возможности масштабирования, соперничество за вычислительные ресурсы только возрастает.

В результате очевидных ограничений данного подхода уже существующие реализации конвейеров обработки видеоданных выбирают микросервисный подход. Он позволяет полноценно декомпозировать задачу

обработки и непосредственного улучшения качества между сервисами, которые могут быть отмасштабированы. Из-за деления подзадач на сервисы и возможности их масштабирования, явных гонок за вычислительные ресурсы не возникает. Использование микросервисного подхода также позволяет более простыми способами сохранять промежуточные результаты обработки конвейера. Схема микросервисной архитектуры приведена на рисунке 17.

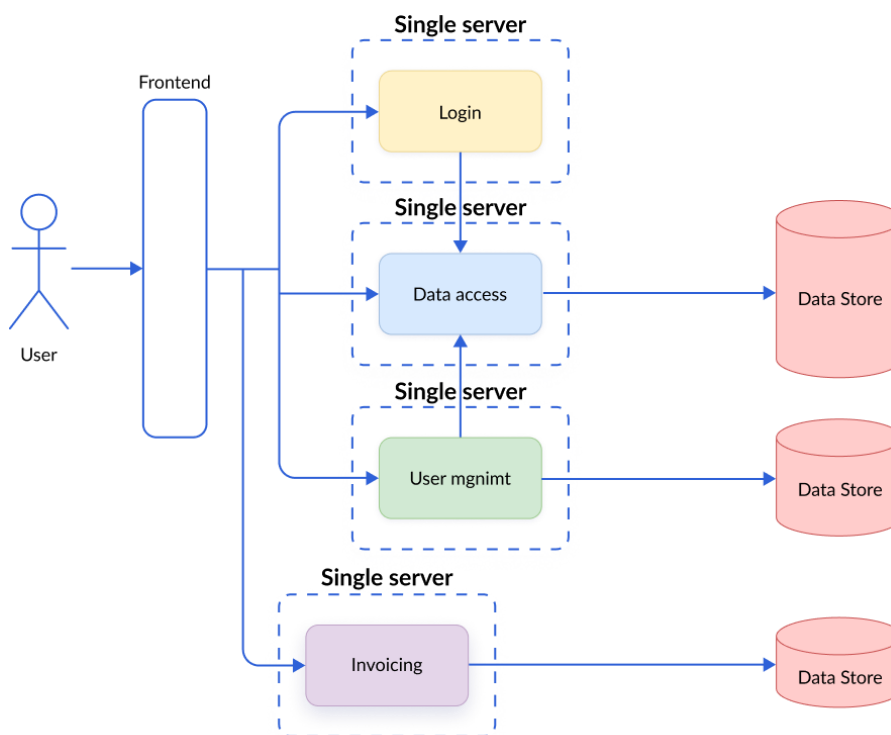


Рисунок 17 – Микросервисная архитектура организации

Кроме архитектурного стиля внутренней организации проектируемого конвейера необходимо выбрать стиль доступной и простой коммуникации компонентов. Выбор архитектуры коммуникационного уровня также важен из-за скорых возможных проблем при некорректном выборе, которые могут привести к полной блокировке системы на время исполнения одной задачи даже при использовании микросервисной организации.

Для конвейера можно применить синхронное или событийно-ориентированное взаимодействие между компонентами. Если рассматривать уже реализованные системы, то из-за использования микросервисного подхода, необходимо задействовать событийно-ориентированный подход

ввиду низкой эффективности синхронной коммуникации для рассматриваемой архитектуры (рисунок 18).

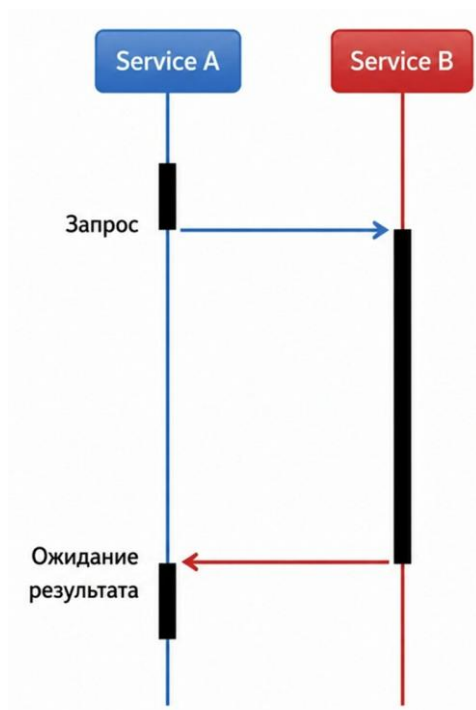


Рисунок 18 – Модель синхронного взаимодействия

Низкая эффективность синхронного подхода в микросервисной архитектуре связана с несколькими причинами:

- нарушение временной изоляции. Микросервисная архитектура предполагает независимость жизненных циклов компонентов, однако синхронный вызов жестко связывает вызывающий и вызываемый сервисы во времени. Вызывающий сервис блокируется до получения ответа, и, если вызываемый сервис замедляется в силу высокой вычислительной нагрузки, эта задержка по цепочке распространяется на всю систему,

- каскадные отказы. В синхронной цепочке вызовов сбой одного сервиса немедленно транслируется всем зависимым компонентам. Если компонент временно недоступен, вызывающий его сервис накапливает ошибки и таймауты, что в условиях высокой нагрузки способно привести к исчерпанию пула потоков или соединений и полной деградации вызывающего сервиса [5],

– несовместимость с длительными операциями. Синхронные протоколы были спроектированы в расчете на короткоживущие запросы. Обработка видеофайлов может занимать десятки секунд или минут. Удержание открытого сетевого соединения в течение всего времени обработки неэффективно. Оно расходует сетевые ресурсы, чувствительно к промежуточным сетевым сбоям и требует специальных механизмов поддержания соединения,

– ограничение масштабирования. Масштабирование вызываемого компонента требует одновременной адаптации вызывающего. Увеличение числа экземпляров сервиса-обработчика должно сопровождаться изменением балансировки нагрузки на стороне генератора задач, что может привести к некорректной работе системы.

Событийно-ориентированная модель с менеджером сообщений полностью устраняет описанные проблемы (рисунок 19).

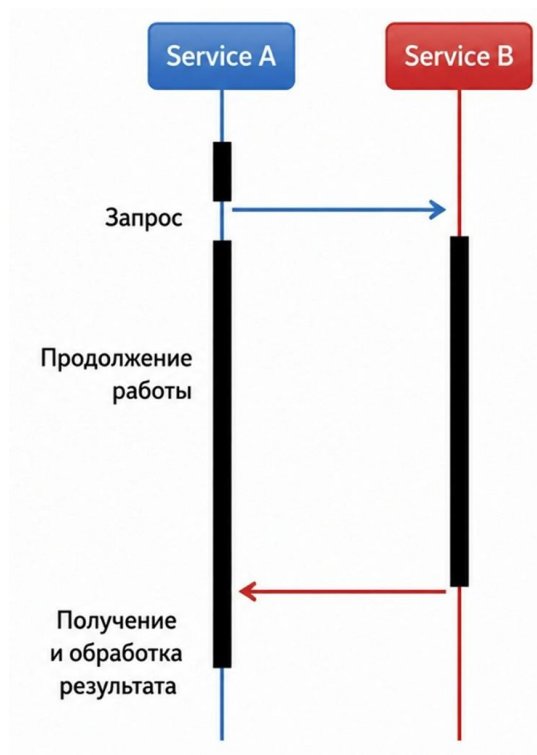


Рисунок 19 – Модель событийно-ориентированного взаимодействия

Генератор задач веб-приложения, используя диспетчер задач, публикует событие и не требует осведомления о количестве развернутых экземпляров конвейера, их загруженности. Это позволяет абстрагироваться от конкретных

сведений о точечном конвейере и масштабировать данную архитектуру под количество доступных вычислительных мощностей. Но стоит заметить, что использование данной модели требует корректного проектирования структуры событий и очередей для менеджера сообщений. Данная структура должна быть идентичной и поддерживаться не только на стороне конвейера, но и на стороне веб-приложения, которое к нему обращается.

2.2 Проектирование структуры событий и очередей

После выбора общих парадигм проектирования для конвейера улучшения качества следует приступить к более детальному планированию паттернов работы событийно-ориентированной микросервисной модели. Модель коммуникации определяет общую динамику работы системы, обмен данными между сервисами и фиксирование продвижения видеофайла по этапам обработки. Поэтому проектирование структуры событий и очередей является критически важным этапом. Ошибки на данном уровне архитектуры могут привести к потере задач, значительному замедлению работы, дублированию результатов.

В конвейере сгенерированное событие является неизменяемым фактом, произошедшим в системе, который несет достаточно информации для сервиса, заинтересованного в выполнении своей работы без дополнительных запросов к источнику события. Такой подход к построению системы событий позволяет минимизировать количество обращений к хранилищу и называется Event Notification Pattern (рисунок 20).



Рисунок 20 – Паттерн полной событийной нотификации

Жизненный цикл видеофайла в конвейере можно разбить на последовательность событий, каждое из которых соответствует завершению определённого этапа обработки. Обобщенный порядок шагов обработки видеоконтента выглядит следующим образом:

- отделение аудиодорожки для возможной последующей обработки и добавления к преобразованному видеофайлу,
- разделение видеофайла на фрагменты с определенным интервалом для последующего деления на фреймы,
- разделение фрагментов на фреймы для последующей обработки в нейросетевом модуле,
- обработка в нейросетевом модуле с преобразованием контента под полученное описание,
- объединение обработанных фреймов в фрагменты видео для дальнейшего получения полного видео,
- объединение полученных фрагментов в итоговое видео без аудиодорожки,

– объединение обработанной или исходной аудиодорожки с полученным видеофайлом.

Если конкретизировать описанные шаги и преобразовать их в события, возникающие в конвейере, то можно будет получить полный список событий жизненного цикла видеофайла. Преобразование данных шагов в конкретные события описано в таблице 4.

Таблица 4 – События жизненного цикла видеофайла в конвейере

Событие	Идентификатор	Описание
Поступил видеофайл на обработку	video.task.start	Появилась задача на обработку видеофайла
Видеофайл обработан	video.task.finish	Задача по обработке видеофайла была завершена
Начало отделения аудиодорожки	video.audio.slice.start	Запуск отделения аудиодорожки
Ошибка отделения аудиодорожки	video.audio.slice.error	Возникновение ошибки при отделении аудиодорожки
Завершение отделения аудиодорожки	video.audio.slice.finish	Завершение отделения аудиодорожки с сохранением в хранилище
Начало деления видеофайла на фрагменты	video.chunks.slice.start	Запуск деления видеофайла на фрагменты
Ошибка деления видеофайла на фрагменты	video.chunks.slice.error	Возникновение ошибки при делении видеофайла на фрагменты
Завершение деления видеофайла на фрагменты	video.chunks.slice.finish	Завершение деления видеофайла на фрагменты с сохранением в хранилище
Начало деления фрагмента видеофайла на фреймы	video.frames.slice.start	Запуск деления фрагмента видеофайла на фреймы
Ошибка деления фрагмента видеофайла на фреймы	video.frames.slice.error	Возникновение ошибки при делении фрагмента видеофайла на фреймы
Завершение деления фрагмента видеофайла на фреймы	video.frames.slice.finish	Завершение деления фрагмента видеофайла на фреймы с сохранением в хранилище
Начало обработки фрейма в нейросетевом модуле	video.frames.neuro.start	Запуск обработки фрейма в нейросетевом модуле
Ошибка обработки фрейма в нейросетевом модуле	video.frames.neuro.error	Ошибка обработки фрейма в нейросетевом модуле
Завершение обработки фрейма в нейросетевом модуле	video.frames.neuro.finish	Завершение обработки фрейма в нейросетевом модуле с сохранением в хранилище

Начало объединения фреймов в фрагменты видеофайла	video.frames.glue.start	Запуск объединения фреймов в фрагменты видеофайла
Ошибка объединения фреймов в фрагменты видеофайла	video.frames.glue.error	Ошибка объединения фреймов в фрагменты видеофайла
Завершение объединения фреймов в фрагменты видеофайла	video.frames.glue.finish	Завершение объединения фреймов в фрагменты видеофайла с сохранением в хранилище
Начало объединения фрагментов видеофайла	video.chunks.glue.start	Запуск объединения фрагментов видеофайла
Ошибка объединения фрагментов видеофайла	video.chunks.glue.error	Ошибка объединения фрагментов видеофайла
Завершение объединения фрагментов видеофайла	video.chunks.glue.finish	Завершение объединения фрагментов видеофайла с сохранением в хранилище
Начало объединения аудиодорожки с видеофайлом	video.audio.glue.start	Запуск объединения аудиодорожки с видеофайлом
Ошибка объединения аудиодорожки с видеофайлом	video.audio.glue.error	Ошибка объединения аудиодорожки с видеофайлом
Завершение объединения аудиодорожки с видеофайлом	video.audio.glue.finish	Завершение объединения аудиодорожки с видеофайлом с сохранением в хранилище

События жизненного цикла по типу обработки помещаются в автономную собственную очередь, что обеспечивает независимую обработку задач. Топология очередей описана в таблице 5.

Таблица 5 – Очереди событий жизненного цикла

Очередь	Идентификатор	Описание
Очередь внутренних задач	video.task.queue	Принимает события класса task. Ориентирована на обработку сервисами подготовки.
Очередь подготовки	video.slice.queue	Принимает события класса slice. Ориентирована на обработку сервисами подготовки и модулем нейросетевой обработки.
Очередь обработки	video.neuro.queue	Принимает события класса neuro. Ориентирована на обработку сервисами сборки.
Очередь сборки	video.glue.queue	Принимает события класса glue. Ориентирована на обработку сервисами сборки.

Очередь недоставленных сообщений	video.ignored.queue	Принимает сообщения, которые не удалось обработать после определенного таймаута.
----------------------------------	---------------------	--

Отдельная очередь для недоставленных сообщений необходима для отказоустойчивости и гарантии повторной доставки сообщений. Стоит учитывать, что при добавлении механизма подтверждения доставки требуется обеспечить идемпотентность сервисов, которая не приведет к проблемам при повторной обработке идентичного сообщения.

2.3 Проектирование модулей подготовки контента для обработки

До этапа обработки в нейронной сети и после него видеофайл должен пройти ряд подготовительных этапов для корректного разбиения, проверки пригодности исходных данных и корректной сборки итогового результата. В проектируемом конвейере обработки данные этапы реализуются узконаправленным набором служб для решения только определенных подзадач подготовки. Использование такого подхода к организации сервисов для решения определенных задач полностью соответствует определению реализации микросервисной архитектуры. Сервисы будут взаимодействовать через ранее описанные очереди событий менеджера сообщений, что будет в полной мере отвечать реализации событийно-ориентированной коммуникации. Предварительный набор обязательных служб представлен следующим списком:

- TaskController. Сервис для извлечения задач из очереди сообщений. Запускает выполнение задач из очереди сообщений веб-приложения при наличии свободных вычислительных ресурсов. Контролирует цикл обработки видеофайла в конвейере, хранит данные о занятых и свободных ресурсах системы. Должен быть максимально отказоустойчивым из-за своей роли звена между очередью сообщений веб-приложения и конвейером обработки. Оповещает конвейер и веб-приложение о прохождении этапов обработки видеофайлом,

– AudioSlicer. Сервис для отделения аудиодорожки. Получает событие из очереди `video.task.queue` и приступает к его обработке. Сохраняет аудиофайл в объектном хранилище. Публикует события `video.audio.slice.start`, `video.audio.slice.error`, `video.audio.slice.finish` по мере выполнения задачи. Если аудиопотоков несколько или они отсутствуют, фиксирует этот факт в событии,

– ChunksSlicer. Сервис для деления видеофайла на фрагменты. Получает событие из очереди `video.task.queue` и приступает к его обработке. Сохраняет полученные фрагменты в объектном хранилище. Публикует события `video.chunks.slice.start`, `video.chunks.slice.error`, `video.chunks.slice.finish` по мере выполнения задачи. Фиксирует в событии количество полученных фрагментов или их отсутствие при пустом видеопотоке,

– FramesSlicer. Сервис для деления фрагмента видеофайла на фреймы. Получает событие из очереди `video.slice.queue` и приступает к его обработке. Сохраняет полученные фреймы фрагмента в объектном хранилище. Публикует события `video.frames.slice.start`, `video.frames.slice.error`, `video.frames.slice.finish` по мере выполнения задачи. Не выполняет деление при получении сообщения о пустом видеопотоке,

– Neurolizator. Сервис для выполнения задачи улучшения качества изображения на основе отдельного нейросетевого модуля. Получает событие из очереди `video.slice.queue` и приступает к его обработке. Сохраняет обработанные фреймы в объектном хранилище. Публикует события `video.frames.neuro.start`, `video.frames.neuro.error`, `video.frames.neuro.finish` по мере выполнения задачи,

– FramesStapler. Сервис для объединения обработанных фреймов в фрагмент видеофайла. Получает событие из очереди `video.neuro.queue` и приступает к его обработке. Сохраняет объединенные фрагменты в объектном хранилище. Публикует события `video.frames.glue.start`, `video.frames.glue.error`, `video.frames.glue.finish` по мере выполнения задачи,

– ChunksStapler. Сервис для объединения полученных фрагментов видеофайла в конечный видеофайл. Получает событие из очереди

video.glue.queue и приступает к его обработке. Сохраняет объединенные фрагменты в объектном хранилище. Публикует события video.chunks.glue.start, video.chunks.glue.error, video.chunks.glue.finish по мере выполнения задачи,

– AudioStapler. Сервис для объединения конечного видеофайла с отделенной ранее аудиодорожкой. Получает событие из очереди video.glue.queue и приступает к его обработке. Сохраняет объединенные фрагменты в объектном хранилище. Публикует события video.audio.glue.start, video.audio.glue.error, video.audio.glue.finish по мере выполнения задачи.

Каждый сервис обладает собственной логикой работы, описанной на архитектурной схеме взаимодействия служб (рисунок 21). Данная схема описывает принципы общего взаимодействия служб между собой, как составляющих общей системы конвейера.

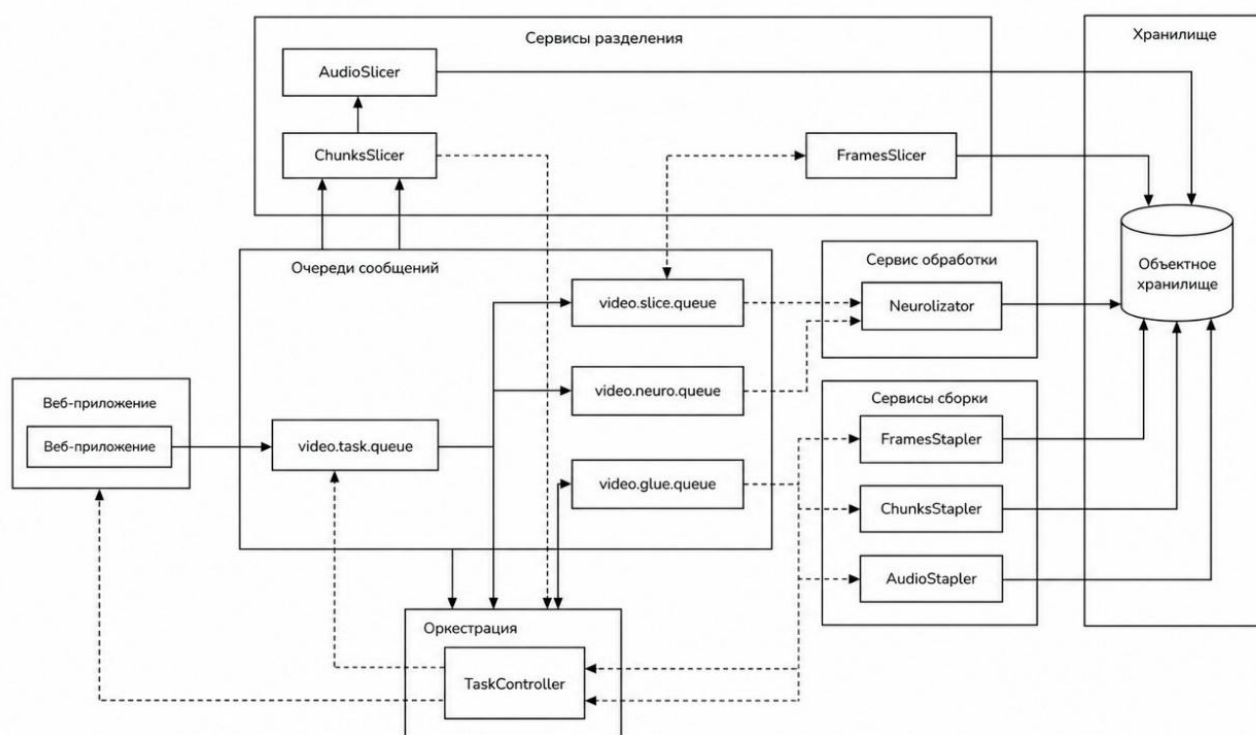


Рисунок 21 – Архитектурная схема взаимодействия служб

2.4 Проектирование модуля нейросетевой обработки

Модуль нейросетевой обработки или сервис Neurolizator является ключевым вычислительным компонентом конвейера, реализующим функцию улучшения качества изображения. В отличие от остальных служб,

оперирующих относительно легковесными операциями ввода-вывода и кодирования, этот модуль характеризуется высокой ресурсоёмкостью, зависимостью от специализированного оборудования и значительной длительностью обработки. Большая часть вычислительных ресурсов будет выделена именно под нужды этого модуля. Основой данной службы будет модифицированная версия модели Real-ESRGAN, выбор которой был описан ранее, в пункте 1.3. Задача проектирования данного модуля сводится не к разработке самой модели, а к созданию программной обертки, превращающей Real-ESRGAN в полноценный отказоустойчивый микросервис, интегрированный в событийно-ориентированный конвейер.

Использование Real-ESRGAN в составе высоконагруженного конвейера предъявляет к модели некоторые требования, которые не были заложены при её создании. В рамках проектирования обёртки решаются следующие задачи модификации:

- оптимизация модели для развертывания. Исходная модель Real-ESRGAN, сохранённая в формате PyTorch, конвертируется в ONNX или TensorRT. Это позволяет сократить время развертывания, снизить потребление видеопамати и устранить зависимость от среды исполнения Python, что упрощает контейнеризацию,

- управление жизненным циклом модели. Модель загружается в видеопамать при старте службы и удерживается там на протяжении всего времени работы сервиса. Такой подход исключает задержки при поступлении первой задачи и обеспечивает минимальное время для старта обработки. При масштабировании каждый новый экземпляр Neurolizator загружает собственную копию модели на выделенный GPU,

- обработка пакетов информации. Фреймы одного фрагмента видеофайла объединяются в пакет и подаются на вход модели одновременно. Размер пакета ограничен объёмом текущей видеопамати и подбирается эмпирически с учётом разрешения фреймов. Пакетная обработка повышает

утилизацию GPU и сокращает среднее время ожидания между поступающими фреймами,

- изоляция пакетов. Каждый пакет обрабатывается в собственном контексте. Сбой обработки на одном пакете не приводит к аварийному завершению Neurolizator и не влияет на асинхронную обработку других пакетов,

- повторная обработка. При внештатных сбоях выполняется заданное количество повторных попыток с фиксированной задержкой. При исчерпании попыток публикуется событие `video.frames.neuro.error`, и задача возвращается в очередь для возможной обработки другим экземпляром.

Следует отметить, что пакетирование фреймов одного фрагмента не означает жёсткой синхронизации результатов. Модуль Neurolizator обрабатывает пакет как цельную задачу, но по завершении публикует события для каждого фрейма независимо. Это исключает блокировку последующих этапов конвейера в ожидании завершения всего пакета и позволяет последующим сервисам начинать работу по мере готовности отдельных кадров.

Для предотвращения бесконечного цикла переобработки одних и тех же данных каждый экземпляр Neuralizator ведёт внутренний журнал идентификаторов задач, завершившихся ошибкой после исчерпания попыток. При повторном получении такой задачи от балансировщика сервис игнорирует её и уведомляет TaskController о необходимости разрешения ошибки в задаче. Журнал хранится в оперативной памяти и не сохраняется при рестарте, так как повторный запуск экземпляра, либо не испытывает проблем при работе из-за вмешательства TaskController, либо повторно сообщает о проблеме и асинхронно ожидает исправления проблемы.

Спроектированный модуль нейросетевой обработки Neurolizator является одновременно самым ресурсоёмким и самым масштабируемым компонентом конвейера. Благодаря горизонтальному масштабированию, заложенному в архитектуру обёртки, производительность сервиса линейно

растёт с добавлением дополнительных графических ускорителей, а изоляция экземпляров друг от друга предотвращает каскадные отказы. В совокупности с механизмами повторной обработки, описанными выше, модуль удовлетворяет требованиям отказоустойчивости, временной изоляции и баланса между качеством и скоростью.

2.5 Механизмы обеспечения отказоустойчивости

Отказоустойчивость заявлена в цели диссертационной работы как одно из ключевых целевых свойств проектируемого конвейера. В отличие от монолитных решений, где единственный сбой способен остановить все процессы обработки, спроектированная микросервисная архитектура с событийно-ориентированным взаимодействием закладывает фундамент для устойчивости на структурном уровне. Для полноценной отказоустойчивости требуется спроектировать конкретные механизмы, охватывающие все уровни системы.

Каждый сервис конвейера проектируется как изолированный обработчик, не разделяющий состояние с другими экземплярами. Сбой одного экземпляра *Neurolizator*, вызванный, допустим, аварийным завершением CUDA-контекста, затрагивает только текущий обрабатываемый пакет и не влияет на работу других экземпляров, обрабатывающих другие пакеты того же или иного видеофайла. Также теоретический сбой *FramesStapler* или *ChunksStapler* не должен оказывать воздействия на сервисы, работающие на предшествующих этапах. Для обработки временных сбоев в каждом сервисе реализуется механизм повторных попыток с фиксированной задержкой. Сервис выполняет заданное количество попыток с интервалом между ними, и только после исчерпания всех попыток фиксирует неисправимую ошибку. Такой подход позволяет конвейеру автоматически восстанавливаться после кратковременных инцидентов и имеет наименование политика повторных попыток (рисунок 22).



Рисунок 22 – Политика повторных попыток

Разбиение видео на части, выполняемое `ChunksSlicer`, также играет двойную роль. Помимо обеспечения параллелизма, данный процесс ограничивает глубину поражения при сбоях. Если обработка одного фрагмента завершилась неудачно и не была восстановлена механизмом повторной обработки, будет исправлена только проблемная часть, а не всё видео целиком. Также все промежуточные результаты обработки сохраняются в объектном хранилище. Это обеспечивает сохранность данных при перезапуске любого сервиса и сервера, в целом, что позволяет возобновить обработку с точки отказа, а не с самого начала. Хранилище выступает в роли внешнего состояния конвейера, не привязанного к жизненному циклу отдельных сервисов.

`ChunksStapler` также требует дополнительного специального механизма устойчивости. Поскольку части могут поступать в произвольном порядке, а отдельные фрагменты могут задерживаться из-за повторной обработки, сервис ведёт учёт полученных фрагментов по каждой задаче. Промежуточное состояние учёта сохраняется также во внешнем хранилище, что позволяет `ChunksStapler` перезапуститься без потери информации о том, какие части уже получены. Сборка инициируется только после получения всех фрагментов.

Все службы конвейера проектируются как идемпотентные обработчики. Идемпотентность гарантирует, что повторная доставка одного и того же события не приведёт к дублированию результатов или побочным эффектам. Данное состояние достигается проверкой наличия результата в хранилище по идентификатору задачи и идентификатору этапа перед началом обработки. Если результат уже записан, сервис не выполняет повторную работу, а сразу публикует событие перехода к следующему этапу.

Спроектированная архитектура также выдерживает пиковые нагрузки без деградации качества и времени обработки. Это возможно благодаря горизонтальному масштабированию каждого сервиса независимо. При росте очереди нейросетевой обработки развёртываются дополнительные экземпляры Neurolizator на новых узлах.

3 Реализация архитектуры конвейера с модулем нейросетевой обработки

По завершению процесса проектирования внутренней архитектуры отказоустойчивого конвейера улучшения качества видео необходимо перейти к его непосредственной реализации. Перед созданием непосредственных реализаций необходимо выбрать технологический стек используемых инструментов. Это позволит более точно сделать вывод о трудозатратах и сложностях планируемых реализаций. При разработке стоит учитывать результаты и требования предварительного исследования (пункты 1.3 и 1.4), а также общие архитектурные схемы системы полученные в разделе проектирования.

Особое внимание стоит обратить на предварительную производительность системы. Её необходимо замерить после завершения процесса разработки, что позволит пересмотреть возможные проблемные места реализации. Замер производительности желательно производить после написания всех сервисов, чтобы полностью контролировать состояние внутренних компонентов. Также эти замеры можно будет использовать для сравнения при проверке эффективности реализованной системы.

3.1 Обоснование выбора технологического стека

Выбор технологического стека сводится к нескольким ключевым критериям. Внутри системы необходимо поддерживать высокую производительность и отказоустойчивость при работе с большими объемами данных, низкие накладные расходы на сериализацию и передачу сообщений, возможность быстрой интеграции новых компонентов (при появлении таковых), а также удобство работы с сервисами для подготовки медиаконтента и развертывания модуля нейронной сети. Акцент в выборе также был сделан на практической осведомленности о внутренних механизмах работы отобранного стека.

В качестве менеджера сообщений выбор был сделан в пользу Apache Kafka. Данный выбор был сделан после эксперимента по отправке сообщений в течение 60 минут:

– RabbitMQ. Широко распространен в качестве менеджера сообщений, но его пропускная способность составляет от 10 до 50 тысяч сообщений в секунду на практическом замере. Для реализуемого конвейера не самый подходящий вариант, так как данное решение может стать ограничителем скорости обмена данными. Вдобавок, RabbitMQ ориентирован на удаление сообщений после доставки, что противоречит возможности долговременного хранения полученных метрик для дальнейшего аудита,

– NATS. Легковесный и чрезвычайно быстрый по замерам менеджер, способный доставить от 800 до 1100 миллионов сообщений в секунду. Скорость является лучшей среди рассмотренных, но базовая версия не поддерживает персистентность, что опять возвращает к проблеме аудита метрик при анализе работы. Скорость при отсутствии системы сохранения данных в данном случае не играет значительной роли,

– ActiveMQ. Обладает централизованным хранилищем сообщений и персистентностью сообщений, но имеет более низкую скорость, чем у Apache Kafka. При интенсивном потоке сообщений может ограничить скорость размещения задач. Это закономерно приводит к отказу в обслуживании,

– Apache Kafka. Золотая середина среди рассмотренных решений. Уже имеется опыт работы с данным решением, обеспечивает сохранность данных при незапланированных отказах в обслуживании, имеет хорошую скорость передачи данных.

Рисунок 23 отображает график интенсивности отправки и сообщений различными менеджерами сообщений, а в таблице 6 отображены количественные показатели проведенного эксперимента.

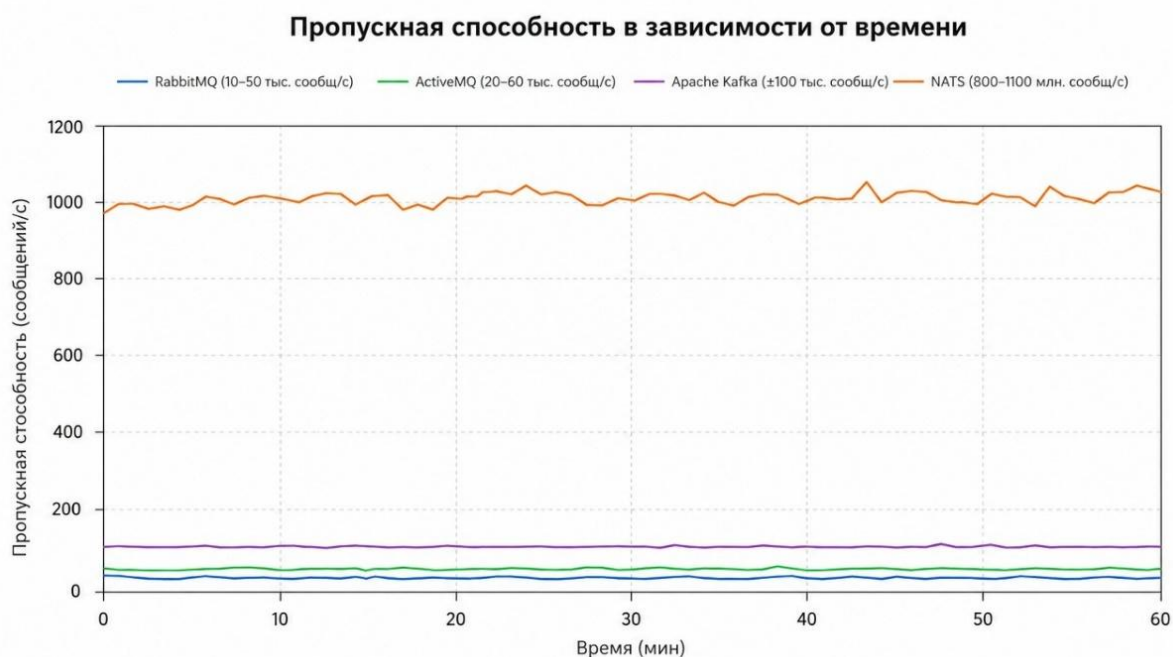


Рисунок 23 – График интенсивности отправки сообщений различными менеджерами сообщений

Таблица 6 – Сравнение характеристик менеджеров сообщений

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

В качестве библиотек для реализации работы с медиаконтентом рассматривались три варианта: FFmpeg API, GStreamer и VLC. Перед их использованием был совершен замер производительности во время операции деления видеофрагмента на фреймы, так как данная задача должна быть самой требовательной по времени среди всех операций подготовки.

Рисунок 24 отображает график времени обработки видеофрагмента выбранными библиотеками, а в таблице 7 отображены количественные показатели проведенного эксперимента.

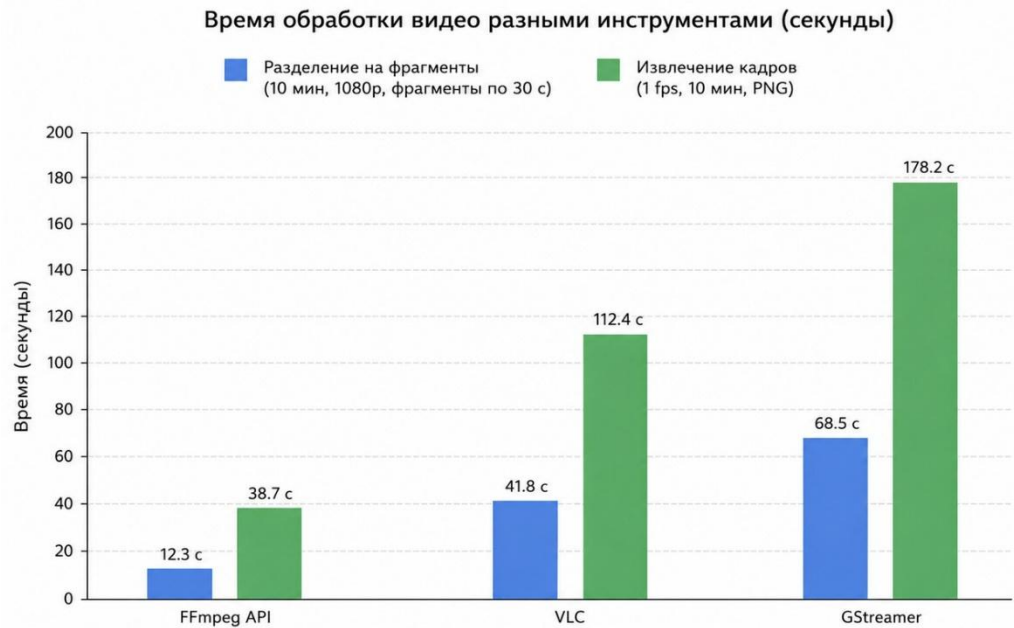


Рисунок 24 – График времени обработки видеофрагмента различными библиотеками для работы с медиафайлами

Таблица 7 – Сравнение времени обработки видеофрагмента выбранными библиотеками

Инструмент	Разделение на фрагменты (10 мин, 1080p, фрагменты по 30 с)	Извлечение кадров (1 fps, 10 мин, PNG)
FFmpeg API	12.3 с	38.7 с
VLC (CLI)	41.8 с	112.4 с
GStreamer	68.5 с	178.2 с

Результат замеров показал, что быстрее всего данную задачу решает FFmpeg API. Также, у данной библиотеки оказалась самая обширная документация и поддержка среди представленных. Большая часть конвейеров для обработки медиаконтента использует именно её.

Выбор языка реализации сервисов конвейера и нейросетевого модуля был произведен достаточно быстро. Ключевыми критериями была производительность и совместимость с ранее выбранными решениями. Из

доступных вариантов с поддержкой выбранных ранее решений возможно использовать: C++, Java и Python. Для реализации сервисов был выбран язык C++ из-за скорости, минимального потребления памяти, поддерживаемой интеграции с Apache Kafka и дополнительных удобных библиотек для логирования.

3.2 Реализация транспортной структуры для обмена событиями

Транспортная структура конвейера представляет собой программную оболочку, обеспечивающую надёжный и унифицированный обмен событиями между всеми сервисами. Она оборачивает работу с Apache Kafka, скрывая от прикладного кода детали подключения, сериализации и обработки ошибок доставки. Реализация данной оболочки выполнена на C++ с использованием библиотек librdkafka и nlohmann/json.

3.2.1 Издатель события

Одним из главных элементов транспортной структуры является класс EventPublisher, отвечающий за публикацию событий в топик Kafka. Его интерфейс спроектирован таким образом, чтобы сервис конвейера мог опубликовать событие одной строкой кода, не заботясь о настройке подключения и сериализации.

Конструктор EventPublisher принимает конфигурацию подключения к менеджеру, настройки подтверждения доставки, таймауты и параметры повторных попыток. Эти параметры вынесены в структуру KafkaConfig, значения которой загружаются из переменных окружения на этапе инициализации сервиса. Такой подход позволяет изменять поведение транспортного уровня без повторной компиляции:

```
class EventPublisher {
public:
    /// @brief Конструктор по умолчанию.
    /// @param config Конфигурация менеджера сообщений.
    EventPublisher(const KafkaConfig& config);

    // Публикация события в указанный топик.
    // Возвращает true при успешной доставке, false при
    ошибке.
```

```

    /// @brief Публикация события в указанную очередь.
    /// @param queue Очередь для публикации.
    /// @param event Перегруженное событие для отправки.
    /// @return Результат отправки события.
    bool publish(const std::string& queue, const BaseEvent&
event);

private:
    std::unique_ptr<RdKafka::Producer> producer_;
    std::string manager_list_;
};

```

При вызове метода `publish` выполняется сериализация объекта `BaseEvent` в строку JSON вызовом виртуального метода `to_json()`. Сформированное сообщение оборачивается в объект `RdKafka::Producer::produce()` и отправляется в указанную очередь. Метод `publish` работает в синхронном режиме с подтверждением от менеджера. Он блокируется до получения подтверждения от всех реплик пока не истечет таймаут, что гарантирует сохранность сообщения. Для снижения задержки при публикации событий используются асинхронные уведомления [6]. Библиотека `librdkafka` вызывает переданную функцию уведомления после завершения отправки, а `EventPublisher` в это время может готовить следующее событие.

3.2.2 Потребитель события

Приём событий реализован классом `EventConsumer`, который инкапсулирует подписку на очередь и десериализацию входящих сообщений. Каждый сервис конвейера создаёт экземпляр `EventConsumer` для своей входной очереди и регистрирует асинхронную функцию, вызываемую при поступлении нового события:

```

class EventConsumer {
public:
    using EventHandler = std::function<void(const
std::string& topic,
const json&
payload)>;

    EventConsumer(const KafkaConfig& config,
const std::string& group_id,
const std::vector<std::string>& topics);

```

```

        // Регистрация обработчика для конкретного типа событий
        void register_handler(const std::string& event_type,
EventHandler handler);

        // Запуск цикла обработки
        void start();

        // Корректная остановка потребителя
        void stop();

private:
        std::unique_ptr<RdKafka::KafkaConsumer> consumer_;
        std::unordered_map<std::string, EventHandler> handlers_;
};

```

Ключевой особенностью EventConsumer является маршрутизация событий по их типу. Каждое JSON-сообщение содержит поле event_type со значением, соответствующим текущему типу события. При поступлении сообщения, EventConsumer извлекает значение этого поля и вызывает соответствующий обработчик из зарегистрированного набора. Такой подход позволяет одному сервису обрабатывать несколько типов событий без усложнения логики диспетчеризации.

3.2.3 Система событий и сериализация

Все события конвейера образуют иерархию классов, наследуемых от абстрактного BaseEvent. Базовый класс определяет методы, а классы ProcessingStartedEvent, AudioSeparatedEvent, ChunksReadyEvent, FramesReadyEvent, FramesProcessedEvent, ChunkAssembledEvent, VideoMergedEvent, ProcessingCompletedEvent и ProcessingFailedEvent реализуют эти методы, добавляя специфичные для своего этапа параметры:

```

class ProcessingStartedEvent : public BaseEvent {
public:
        ProcessingStartedEvent(const std::string& task_id,
                                const std::string& storage_path,
                                const std::string& format,
                                int width,
                                int height,
                                double duration)
            : task_id_(task_id)
            , storage_path_(storage_path)
            , format_(format)
            , width_(width)
            , height_(height)

```

```

        , duration_(duration)
    {}

    std::string event_type() const override { return
"video.uploaded"; }
    std::string task_id() const override { return task_id_;
}

protected:
    json to_json() const override {
        return {
            {"event_type", "video.uploaded"},
            {"task_id", task_id_},
            {"storage_path", storage_path_},
            {"format", format_},
            {"width", width_},
            {"height", height_},
            {"duration", duration_},
                                                    {"timestamp",
std::chrono::system_clock::now().time_since_epoch().count()}
        };
    }

private:
    std::string task_id_;
    std::string storage_path_;
    std::string format_;
    int width_;
    int height_;
    double duration_;
};

```

Сериализация выполняется библиотекой `nlohmann/json`, которая обеспечивает читаемость сообщений непосредственно в очередях и упрощает аудит. Для восстановления типизированных объектов из JSON реализована фабрика `EventFactory`, принимающая JSON-строку и возвращающая указатель с конкретным типом. Использование подобного паттерна обеспечивает безопасность при обработке событий в сервисах.

3.2.4 Интеграция транспортной структуры с сервисами конвейера

Каждый сервис при запуске создаёт экземпляры `EventPublisher` и `EventConsumer`, передавая им конфигурацию, загруженную из окружения. Сервис регистрирует обработчики для входных событий и использует издатель для публикации выходных. Шаблон инициализации унифицирован для всех семи сервисов.

Интеграция транспортного уровня с сервисами выполнена по принципу максимального переиспользования написанного ранее кода [7]. EventPublisher, EventConsumer, EventFactory и KafkaConfig скомпонованы в статическую библиотеку pipeline_transport, которая линкуется с каждым сервисом. Таким образом, все семь сервисов используют один и тот же код для взаимодействия с Apache Kafka, что исключает дублирование и расхождение реализаций. Единственное различие между сервисами на транспортном уровне заключается в идентификаторе consumer_group, который передаётся через переменную окружения KAFKA_GROUP_ID и определяет, к какой очереди привязывается данный экземпляр. Такой подход позволил реализовать транспортный уровень единожды и повторно использовать его без изменений во всех компонентах конвейера.

Помимо унификации кода, библиотека pipeline_transport обеспечивает единообразное поведение всех сервисов в нештатных ситуациях. Стратегия повторных попыток при потере соединения с Apache Kafka, формат логирования транспортных операций, метрики публикации и потребления определяются библиотекой и не требуют повторной реализации в каждом сервисе. При внесении изменений в транспортный уровень достаточно пересобрать библиотеку и перелинковать сервисы, не модифицируя код. Это соответствует принципу единственной ответственности, где транспортная структура занимается исключительно доставкой событий, а сервисы только обработкой видео.

3.3 Реализация модулей подготовки контента для обработки

Сервисы подготовки контента образуют начальный и конечный сегмент конвейера и отвечают за приём задачи. Все они будут реализованы на C++ с использованием описанной ранее библиотеки FFmpeg API для операций с видеоконтентом. Также в рассматриваемые модули должна быть интегрирована библиотека с реализацией API для полноценной реализации транспортной структуры конвейера.

3.3.1 Сервис управления задачами конвейера

TaskController занимает особое положение среди всех сервисов. Он является связующим звеном между очередью сообщений веб-приложения и внутренними очередями конвейера. В зоне его ответственности находится извлечение задач из внешней очереди video.task.queue, проверка доступности вычислительных ресурсов и запуск обработки видеофайла, путем помещения сообщения о необходимости обработки в очереди последующих обработчиков в виде AudioSlicer и ChunksSlicer. Кроме того, TaskController сопровождает задачу на протяжении всего жизненного цикла, агрегируя события о прохождении этапов и оповещая веб-приложение о значимых изменениях статуса:

```
class TaskController {
public:
    TaskController(const KafkaConfig& kafka_config,
                   std::shared_ptr<ResourceManager>
resources);
    void start();
    void stop();

private:
    void handle_new_task(const json& payload);
    void handle_stage_event(const json& payload);
    bool acquire_resources(const TaskMetadata& task);
    void release_resources(const std::string& task_id);
    void snapshot_state();

    EventPublisher publisher_;
    EventConsumer consumer_;
    std::shared_ptr<ResourceManager> resources_;
    std::unordered_map<std::string, TaskState>
active_tasks_;
    std::mutex state_mutex_;
};
```

При старте TaskController инициализирует потребителя для очереди video.task.queue и издателя для публикации командных событий. Он также поддерживает в оперативной памяти карту занятых и свободных ресурсов системы. При получении нового сообщения TaskController проверяет, достаточно ли ресурсов для обработки задачи. Если ресурсы доступны, он резервирует их, публикует событие video.task.accepted во внутреннюю очередь

и инициирует цепочку обработки. Если ресурсов недостаточно, то задача не извлекается из очереди.

Особое внимание в реализации TaskController уделено отказоустойчивости. Сервис периодически сохраняет снимок своей карты ресурсов и списка активных задач в постоянное файловое хранилище. При аварийном перезапуске TaskController восстанавливает состояние из снимка и продолжает сопровождение активных задач без потери данных.

По завершении обработки видеофайла (получение события video.audio.glue.finish или любого из фатальных *.error) TaskController освобождает зарезервированные ресурсы, финализирует задачу и оповещает веб-приложение о готовности или неготовности результата.

3.3.2 Сервис отделения аудиодорожки

AudioSlicer получает событие video.task.accepted из очереди video.task.queue и выполняет первый этап подготовки контента – извлечение аудиодорожки из исходного видеофайла. Сервис публикует три возможных события, отражающих ход обработки: video.audio.slice.start в момент начала работы, video.audio.slice.finish при успешном завершении и video.audio.slice.error при возникновении неисправимой ошибки:

```
class AudioSlicer {
public:
    AudioSlicer(const KafkaConfig& kafka_config);
    void start();

private:
    void handle_task(const std::string& task_id, const
std::string& input_path);
    AudioResult extract_audio(const std::string& input_path,
                             const std::string& task_id);

    EventPublisher publisher_;
    EventConsumer consumer_;
};

AudioResult AudioSlicer::extract_audio(const std::string&
input_path,
                                     const std::string&
task_id) {
    publish_event("video.audio.slice.start", task_id);
```

```

        AVFormatContext* fmt_ctx = nullptr;
        if (avformat_open_input(&fmt_ctx, input_path.c_str(),
        nullptr, nullptr) < 0) {
            publish_error("video.audio.slice.error", task_id,
            "Cannot open file");
            return {};
        }
        avformat_find_stream_info(fmt_ctx, nullptr);

        std::vector<AudioStream> streams;
        for (unsigned i = 0; i < fmt_ctx->nb_streams; ++i) {
            if (fmt_ctx->streams[i]->codecpar->codec_type ==
            AVMEDIA_TYPE_AUDIO) {
                AudioStream s = copy_audio_stream(fmt_ctx, i,
            task_id);
                streams.push_back(s);
            }
        }
        avformat_close_input(&fmt_ctx);
        return {streams};
    }

```

Реализация AudioSlicer опирается на FFmpeg API, а именно на библиотеку libavformat для демультимплексирования контейнера. Сервис открывает входной файл, находит аудиопотоки и извлекает их в отдельные файлы без перекодирования, что минимизирует вычислительные затраты. Если аудиопотоков несколько, то извлекаются все. Если аудиопоток отсутствует, этот факт фиксируется в событии video.audio.slice.finish специальным флагом audio_present = false, что позволяет последующим сервисам корректно обработать видео без звука.

Извлечённые аудиофайлы сохраняются в зарезервированном файловом хранилище под ключами вида {task_id}/audio/stream_{N}.{ext}. Кодек, битрейт, частота дискретизации, количество каналов включаются в событие для использования на этапе финального слияния.

3.3.3 Сервис деления видеофайла на фрагменты

ChunksSlicer получает событие video.task.accepted и выполняет разбиение видеоряда на фрагменты фиксированной длительности. Аналогично AudioSlicer, он публикует три возможных события процесса обработки:

video.chunks.slice.start, video.chunks.slice.finish и video.chunks.slice.error:

```

class ChunksSlicer {
public:
    ChunksSlicer(const KafkaConfig& kafka_config,
                  int chunk_duration_sec);
    void start();

private:
    void handle_task(const std::string& task_id, const
std::string& input_path);
    int segment_video(const std::string& input, const
std::string& output_pattern);

    EventPublisher publisher_;
    EventConsumer consumer_;
    int chunk_duration_sec_;
};

```

Ключевая особенность реализации ChunksSlicer это разбиение с учётом ключевых кадров. Если граница фрагментов приходится на Р-кадр или В-кадр, FFmpeg смещает её к ближайшему ключевому кадру, что предотвращает артефакты на стыках при последующей сборке. Длительность фрагментов задаётся через переменную окружения CHUNK_DURATION_SECONDS, по умолчанию 3 секунды. Сервис обрабатывает только видеопоток.

Видеофрагменты сохраняются в зарезервированное файловое хранилище с именами {task_id}/chunks/chunk_{NNN}.mp4. Количество полученных частей включается в событие video.chunks.slice.finish и используется сервисом ChunksStapler для проверки полноты набора при сборке. Если видеопоток отсутствует, ChunksSlicer фиксирует chunk_count = 0 в событии, и последующие сервисы пропускают этапы обработки видео.

3.3.4 Сервис деления фрагментов на фреймы

FramesSlicer продолжает цепочку подготовки контента. В отличие от AudioSlicer и ChunksSlicer, он получает события не из video.task.queue, а из внутренней очереди video.slice.queue. Это обусловлено тем, что FramesSlicer обрабатывает не исходный файл, а фрагменты, сформированные ChunksSlicer. Каждая часть обрабатывается независимо, что позволяет нескольким экземплярам FramesSlicer работать параллельно:

```

class FramesSlicer {
public:

```

```

        FramesSlicer(const KafkaConfig& kafka_config,
                      std::shared_ptr<MinioClient> minio);
        void start();

    private:
        void handle_chunk(const std::string& task_id, int
chunk_idx,
                        const std::string& chunk_path, int
expected_fps);
        int decode_to_frames(const std::string& chunk_path,
                            const std::string& task_id, int
chunk_idx);

        EventPublisher publisher_;
        EventConsumer consumer_;
        std::shared_ptr<MinioClient> minio_;
};

```

Сервис также публикует события, обозначающие текущее состояние обработки: `video.frames.slice.start`, `video.frames.slice.finish` и `video.frames.slice.error`. При получении сообщения о пустом видеопотоке (`chunk_count = 0`) `FramesSlicer` не выполняет деления и немедленно публикует `video.frames.slice.finish` с флагом `frames_present = false`.

Реализация покадрового извлечения использует функцию `avcodec_decode_video2` в цикле по пакетам, полученным из входного фрагмента. Каждый декодированный кадр сохраняется в формате PNG без сжатия, что исключает накопление артефактов перекодирования на промежуточных этапах.

Фреймы сохраняются в зарезервированное файловое хранилище с именами `{task_id}/frames/chunk_{NNN}/frame_{MMMM}.png`. Наименование отражает принадлежность к фрагменту и порядковый номер кадра, что обеспечивает однозначное восстановление последовательности при сборке.

3.3.5 Сервис объединения обработанных фреймов во фрагменты

`FramesStapler` получает событие `video.frames.neuro.finish` из очереди `video.neuro.queue` и выполняет обратную `FramesSlicer` операцию склеивания обработанных нейросетью фреймов в видеофрагмент. Сервис также публикует три типа событий: `video.frames.glue.start` в момент начала

работы, `video.frames.glue.finish` при успешном завершении и `video.frames.glue.error` при возникновении неисправимой ошибки:

```
class FramesStapler {
public:
    FramesStapler(const KafkaConfig& kafka_config);
    void start();

private:
    void handle_neuro_finish(const std::string& task_id, int
chunk_idx,
                                int frame_count, int fps);
    int encode_frames(const std::string& frames_pattern,
                                const std::string& output_path, int
fps);

    EventPublisher publisher_;
    EventConsumer consumer_;
};
```

При получении события `FramesStapler` извлекает все фреймы, относящиеся к указанному фрагменту, в порядке их нумерации. Для склеивания используется FFmpeg API с кодеком `libx264` и настройками, ориентированными на минимизацию потерь качества на промежуточном этапе. Постоянный коэффициент качества `CRF = 18` и `preset = fast`, обеспечивающий баланс между скоростью кодирования и степенью сжатия.

Собранный видеофрагмент сохраняется в зарезервированное файловое хранилище под ключом `{task_id}/assembled_chunks/chunk_{NNN}.mp4` и будет использован `ChunksStapler` на следующем этапе.

3.3.6 Сервис объединения фрагментов в конечный видеофайл

`ChunksStapler` получает события `video.frames.glue.finish` из очереди `video.glue.queue` и агрегирует их в разрезе задачи. Поскольку части одного видео могут обрабатываться разными экземплярами `Neurolizator` и `FramesStapler` параллельно, события о готовности видеофрагментов поступают в непредсказуемом порядке. Задача `ChunksStapler` – дождаться получения всех частей и выполнить их конкатенацию в единый видеоряд:

```
class ChunksStapler {
public:
    ChunksStapler(const KafkaConfig& kafka_config);
    void start();
```

```

private:
    void handle_glue_finish(const std::string& task_id, int
chunk_idx,
                                int expected_chunks);
    int concat_chunks(const std::vector<std::string>&
chunk_paths,
                                const std::string& output_path);

    EventPublisher publisher_;
    EventConsumer consumer_;
    std::unordered_map<std::string, ChunkAssemblyState>
states_;
    std::mutex mutex_;
};

```

Сервис публикует три события: `video.chunks.glue.start` при получении первого фрагмента задачи, `video.chunks.glue.finish` после успешной сборки всех фрагментов и `video.chunks.glue.error` при возникновении ошибки.

Для учёта поступивших фрагментов `ChunksStapler` использует промежуточное состояние, сохраняемое в отдельной очереди Apache Kafka с долговременным хранением. При получении события сервис обновляет множество полученных фрагментов для задачи и сравнивает его размер с ожидаемым количеством, указанным в исходном событии `video.chunks.slice.finish`. При совпадении инициируется сборка конечного результата.

Конкатенация выполняется без перекодирования с использованием `concat demuxer` из FFmpeg API, что минимизирует вычислительные затраты и предотвращает дополнительную деградацию качества на финальном этапе [9].

3.3.7 Сервис объединения видеофайла с аудиодорожкой

`AudioStapler` получает событие `video.chunks.glue.finish` из очереди `video.glue.queue` и выполняет финальный этап слияния собранного видеоряда с аудиодорожкой, отделённой `AudioSlicer` на начальном этапе. Сервис публикует три события состояния обработки задачи: `video.audio.glue.start`, `video.audio.glue.finish` и `video.audio.glue.error`:

```

class AudioStapler {
public:
    AudioStapler(const KafkaConfig& kafka_config);

```

```

void start();

private:
    void handle_video_ready(const std::string& task_id);
    int mux_video_audio(const std::string& video_path,
                        const std::string& audio_path,
                        const std::string& output_path);

    EventPublisher publisher_;
    EventConsumer consumer_;
};

```

Реализация AudioStapler использует FFmpeg API для мультимплексирования видео и аудио в единый выходной контейнер. Сервис открывает видеофайл и аудиофайл, находит соответствующие потоки и копирует их в выходной файл без перекодирования, сохраняя исходное качество.

Финальный файл сохраняется в зарезервированное файловое хранилище под ключом {task_id}/final_output.mp4. Событие video.audio.glue.finish является терминальным: получив его, TaskController финализирует задачу и оповещает веб-приложение о готовности результата.

3.4 Реализация модуля нейросетевой обработки

Модуль нейросетевой обработки Neurolizator самый ресурсоемкий компонент конвейера обработки видеоданных. В отличие от других сервисов, управляющих преобразованиями видеоданных для нейронной обработки, Neurolizator выполняет непосредственную обработку полученных данных при помощи нейросетевой модели Real-ESRGAN для улучшения качества каждого фрейма видео. Реализация включает в себя три основных этапа:

- инициализация и загрузка модели для обработки при запуске сервиса,
- пакетная обработка фреймов,
- обработка специфических ошибок во время работы.

Данные этапы необходимо отделить в программной реализации из-за возможных трудностей поддержки в будущем [8]. Разделение на этапы также поможет проще отлаживаться при возникновении ошибок.

3.4.1 Инициализация и загрузка модели

Модель Real-ESRGAN, предварительно обученная и сконвертированная из формата PyTorch в ONNX, загружается при старте работы сервиса Neurolizator. Конвертация была выполнена утилитой `torch.onnx.export`. Служба порождает определенное количество потоков для обработки пакетов данных ONNX-файл загружается во все потоки-обработчики, что обеспечивает независимый многопоточный процесс работы.

Инициализация выполняется в конструкторе сервиса. Сессия ONNX создаётся с указанием провайдера CUDA для GPU и CPU из-за использования видеокарт NVIDIA на тестовом стенде. Параметры входных и выходных тензоров извлекаются из данных модели и сохраняются на протяжении всего процесса работы потока:

```
class Neurolizator {
public:
    Neurolizator(const KafkaConfig& kafka_config,
                  const std::string& model_path)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "neurolizator-group",
{"video.slice.queue"})
        , current_batch_size_(4)
    {
        // Инициализация окружения ONNX Runtime
        env_ =
std::make_unique<Ort::Env>(ORT_LOGGING_LEVEL_WARNING,
"Neurolizator");

        // Настройка опций сессии
        Ort::SessionOptions opts;
        opts.SetIntraOpNumThreads(1);

        opts.SetGraphOptimizationLevel(GraphOptimizationLevel::ORT_ENABL
E_ALL);

        // Добавление CUDA-провайдера, если доступен
        try {
            OrtCUDAProviderOptions cuda_opts;
            cuda_opts.device_id = 0;
            opts.AppendExecutionProvider_CUDA(cuda_opts);
        } catch (const Ort::Exception&) {
            // GPU недоступен – используем CPU
        }

        // Загрузка модели
```

```

        session_ = std::make_unique<Ort::Session>(*env_,
model_path.c_str(), opts);

        // Извлечение информации о входах и выходах
        Ort::AllocatorWithDefaultOptions allocator;
        input_name_ = session_>GetInputName(0, allocator);
        output_name_ = session_>GetOutputName(0,
allocator);

        auto input_info = session_>GetInputTypeInfo(0);
        auto input_dims =
input_info.GetTensorTypeAndShapeInfo().GetShape();
        input_h_ = input_dims[2];
        input_w_ = input_dims[3];

        memory_info_ =
Ort::MemoryInfo::CreateCpu(OrtArenaAllocator, OrtMemTypeDefault);

consumer_.register_handler("video.frames.slice.finish",
        [this](const std::string&, const json& payload)
{
        this->handle_frames_ready(payload);
});

void start() { consumer_.start(); }

private:
    EventPublisher publisher_;
    EventConsumer consumer_;

    std::unique_ptr<Ort::Env> env_;
    std::unique_ptr<Ort::Session> session_;
    Ort::MemoryInfo memory_info_;
    std::string input_name_;
    std::string output_name_;
    int input_h_, input_w_;
    int current_batch_size_;
};

```

Инициализация модели является наиболее длительной операцией при старте сервиса. В зависимости от размера ONNX-файла и производительности GPU, загрузка занимает от 2 до 4 секунд, что значительно может повлиять на скорость перезапуска при незапланированном падении. Но это время компенсируется тем, что модель для каждого потока загружается однократно и затем обслуживает все поступающие задачи без дополнительных задержек. При горизонтальном масштабировании каждый новый экземпляр Neurolizator

проходит собственную инициализацию, не влияя на уже работающие экземпляры [10].

3.4.2 Пакетная обработка фреймов

Обработка заготовленных фреймов начинается с получения события `video.frames.slice.finish`. Публикуется событие `video.frames.neuro.start`, Neurolizator загружает все фреймы для выбранного фрагмента из файлового хранилища, формирует входной тензор и выполняет обработку через ONNX модель. Полученные результаты отправляются в файловое хранилище, а также публикуется событие `video.frames.neuro.finish`:

```
void Neurolizator::handle_frames_ready(const json& payload)
{
    std::string task_id = payload["task_id"];
    int chunk_idx = payload["chunk_index"];
    int frame_count = payload["frame_count"];

    publish_event("video.frames.neuro.start", task_id,
chunk_idx);

    // Загрузка фреймов
    std::vector<cv::Mat> frames;
    for (int i = 0; i < frame_count; ++i) {
        std::string key = task_id + "/frames/chunk_" +
            std::to_string(chunk_idx) +
            "/frame_" +
                std::to_string(i) + ".png";
        auto data = minio_ ->get_object(key);
        frames.push_back(cv::imdecode(data,
cv::IMREAD_COLOR));
    }

    // Размеры тензора с учётом размера фреймов
    int batch = current_batch_size_;
    size_t input_size = batch * 3 * input_h_ * input_w_;
    std::vector<float> input_tensor(input_size, 0.0f);

    std::vector<std::string> output_keys;

    // Подготовка фреймов
    for (int start = 0; start < frame_count; start += batch)
    {
        int end = std::min(start + batch, frame_count);
        int current_batch = end - start;

        // Предобработка фреймов
        for (int i = 0; i < current_batch; ++i) {
            preprocess(frames[start + i], input_tensor, i);
```

```

    }

    // Формирование входного тензора
    std::vector<int64_t> input_shape = {current_batch, 3,
input_h_, input_w_};
    Ort::Value input = Ort::Value::CreateTensor<float>(
        memory_info_, input_tensor.data(), input_size,
        input_shape.data(), input_shape.size());

    // Обработка в Real-ESRGAN
    std::vector<float> output_tensor(current_batch * 3 *
input_h_ * input_w_);
    std::vector<int64_t> output_shape = {current_batch,
3, input_h_, input_w_};
    Ort::Value output = Ort::Value::CreateTensor<float>(
        memory_info_, output_tensor.data(),
output_tensor.size(),
        output_shape.data(), output_shape.size());

    const char* input_names[] = {input_name_.c_str()};
    const char* output_names[] = {output_name_.c_str()};

    try {
        session_>Run(Ort::RunOptions{nullptr},
            input_names, &input, 1,
            output_names, &output, 1);
    } catch (const Ort::Exception& e) {
        // Обработка ошибок GPU
        handle_gpu_error(e, task_id, chunk_idx);
        return;
    }

    // Сохранение результатов
    for (int i = 0; i < current_batch; ++i) {
        cv::Mat out = postprocess(output_tensor, i,
input_h_, input_w_);
        std::string key = task_id + "/processed/chunk_"
+
                                std::to_string(chunk_idx) +
"/frame_" +
                                std::to_string(start + i) +
".png";
        minio_>put_object(key, cv::imencode(".png",
out));
        output_keys.push_back(key);
    }
}

json metadata = {
    {"chunk_index", chunk_idx},
    {"frame_count", frame_count},
    {"output_keys", output_keys}
};

```

```

        publish_event("video.frames.neuro.finish", task_id,
metadata);
    }

```

Если разрешение входного фрейма превышает ожидаемый моделью размер $\text{input_h_} \times \text{input_w_}$, запущенная в потоке модель автоматически переключается в тайловый режим. Изображение разбивается на перекрывающиеся тайлы, каждый из которых независимо нормализуется и упаковывается в тензор. После обработки тайлы склеиваются с взвешенным усреднением в зонах перекрытия, что позволяет избежать видимых швов на итоговом кадре. Механизм особенно востребован при обработке видео с большим разрешением, которые не могут быть поданы на вход модели целиком ввиду ограничений видеопамати.

3.4.3 Обработка ошибок во время работы

Сбои инструмента CUDA обрабатываются в специализированном методе `handle_gpu_error`. При возникновении `Ort::Exception` анализируется сообщение об ошибке. Если диагностируется нехватка видеопамати (`CUDA_ERROR_OUT_OF_MEMORY`), количество обрабатываемых фреймов уменьшается вдвое, и обработка повторяется. При аварийном завершении CUDA-контекста выполняется переинициализация сессии ONNX и перезагрузка модели. Только после исчерпания всех попыток восстановления публикуется событие `video.frames.neuro.error`:

```

void Neurolizator::handle_gpu_error(const Ort::Exception& e,
                                     const std::string&
task_id,
                                     int chunk_idx) {
    std::string err_msg = e.what();

    if (err_msg.find("out of memory") != std::string::npos)
    {
        // Уменьшенное количество фреймов
        current_batch_size_ = std::max(1,
current_batch_size_ / 2);
        // Повторная попытка будет выполнена Kafka при
повторной доставке
        throw;
    }
}

```

```

        if (err_msg.find("CUDA_ERROR_UNKNOWN") !=
std::string::npos ||
        err_msg.find("CUDA_ERROR_LAUNCH_FAILED") !=
std::string::npos) {
            // Переинициализация CUDA-контекста
            session_.reset();
            Ort::SessionOptions opts;
            OrtCUDAProviderOptions cuda_opts;
            cuda_opts.device_id = 0;
            opts.AppendExecutionProvider_CUDA(cuda_opts);
            session_ = std::make_unique<Ort::Session>(*env_,
model_path_.c_str(), opts);
            throw; // Повторная попытка
        }

        // Фатальная ошибка
        publish_error("video.frames.neuro.error", task_id,
                    "GPU fatal error: " + err_msg);
    }
}

```

Пакетная обработка, реализованная в цикле по фрагментам видеофайла, позволяет повысить утилизацию ресурсов GPU по сравнению с покадровой обработкой. Для дополнительного сокращения задержек в цикле обработки встроена конвейеризация загрузки и обработки. Пока текущий пакет обрабатывается на GPU, следующий пакет фреймов загружается из файлового хранилища и проходит обработку в другом потоке. Синхронизация между потоками реализована через очередь готовых пакетов ограниченной ёмкости, что предотвращает неконтролируемое потребление оперативной памяти при опережающей загрузке.

3.5 Реализация механизмов обеспечения отказоустойчивости

Отказоустойчивость, заложенная в архитектуру конвейера на этапе проектирования, реализована через совокупность взаимодополняющих механизмов, охватывающих транспортный и вычислительный уровни системы. Данные механизмы не взаимоисключают друг друга и позволяют повысить внутреннюю устойчивость системы.

3.5.1 Гарантированная доставка и идемпотентность обработки

Фундаментом отказоустойчивости является сочетание двух механизмов транспортного уровня: гарантированной доставки сообщений и идемпотентности всех сервисов.

Гарантированная доставка реализована через подтверждение обработки в Apache Kafka. Сообщение подтверждается только после полного и успешного выполнения всех операций этапа. При возникновении исключения подтверждение не отправляется, и по истечении таймаута сессии Kafka перераспределяет партицию между другими экземплярами consumer group, обеспечивая автоматическую повторную обработку.

Идемпотентность гарантирует, что многократная доставка одного и того же события не приведёт к дублированию результатов. Перед началом ресурсоёмкой операции каждый сервис проверяет наличие маркера завершения в файловом хранилище по ключу `{task_id}/{stage}/completed`. Если маркер существует, обработка пропускается, и сразу публикуется событие перехода к следующему этапу. Маркер записывается только после успешного сохранения всех выходных данных этапа, что исключает состояние гонки между проверкой и записью.

Дополнительно на стороне веб-приложения включён режим идемпотентности, исключающий дублирование сообщений при повторных попытках отправки.

3.5.2 Специфическая устойчивость Neurolizator и мониторинг

Модуль Neurolizator подвержен специфическому классу отказов, связанных с GPU. При возникновении ошибки нехватки видеопамяти размер принимаемого пакета нейронной сети автоматически уменьшается вдвое, и процесс обработки повторяется. При аварийном завершении CUDA-контекста Neurolizator выполняет переинициализацию контекста и перезагрузку модели, после чего продолжает обработку с проблемного фрагмента без перезапуска контейнера.

Завершающим элементом стратегии отказоустойчивости является система мониторинга. Каждый сервис экспортирует метрики в формате Prometheus. Информация о количестве обработанных и сбойных задач, длительность этапов, глубину очередей, отставание обработки. На основе этих метрик настроены внутренние предупреждения. Рост доли ошибок сообщает

о деградации сервиса, а аномальное увеличение длительности этапа информирует о проблемах с GPU или файловой подсистемой. Активное обнаружение таких ситуаций позволяет принять меры до того, как проблема станет заметной конечному пользователю.

Реализованные механизмы гарантированной доставки с идемпотентностью и мониторинг с внутренними предупреждениями, обеспечивают отказоустойчивость конвейера, заявленную в цели диссертационной работы.

4 Оценка эффективности реализованной системы

Реализация конвейера обработки видеофайлов для улучшения качества и требует экспериментального подтверждения корректности и эффективности спроектированной архитектуры. Теоретический анализ эффективности архитектуры должен быть обоснован и верифицирован в условиях, приближенных к реальной эксплуатации. Необходимо проанализировать метрики сервисов для обработки видеофайла, а также корректность интеграции дообученной модели в сервис. Подтверждение эффективности должно заключаться в анализе количественных показателей качества восстановленного видео и затраченных ресурсов на данную операцию.

Все тесты и исследования будут проведены на тестовом стенде, используемом при проведении исследования в пункте 1.3. Для создания тестовой нагрузки была использована выборка видеофайлов, отличная от выборки для дообучения.

4.1 Исследование метрик нейронной сети

Как и в пункте 1.3 необходимо провести экспериментальное сравнение работы модели Real-ESRGAN по критериям качества и производительности. Необходимо проанализировать возможные потери производительности при запуске оригинального инференса Real-ESRGAN x4+VM и сервиса в оболочке конвейера Neurolizator в нативном режиме. Данное исследование важно из-за необходимости подтверждения эффективности разработанных сервисов конвейера.

4.1.1 Методика исследования метрик

Для оценки качества восстановления используется набор из 150 видеофайлов возможного пользовательского контента, характерного для целевого веб-приложения. Видеофайлы содержат различные типы деградаций: шум матрицы, смазывание движения, артефакты компрессии кодеков H.264 и H.265, а также пониженное разрешение. Каждый видеофайл обрабатывается оригинальной версией модели и встроенной сервисной.

Оценка качества производится покадрово по обозначенным ранее метрикам:

- PSNR. Пиковое отношение сигнала к шуму, характеризует пиксельную точность восстановления,
- SSIM. Индекс структурного сходства, оценивает сохранность структуры изображения,
- LPIPS. Перцептивное различие, наиболее коррелирующее с субъективным восприятием человека,
- NIQE. Оценка естественности изображения, не требующая эталона.

Для каждого использованного в исследовании видеофайла вычисляется среднее значение метрики по всем кадрам, после чего результаты усредняются по всей тестовой выборке. Вдобавок фиксируется среднее время обработки кадра для тестируемых моделей.

4.1.2 Результаты сравнения

Сравнение выполнялось при последовательной подаче тестовых видеофайлов. Результаты эксперимента приведены в таблице 8 и на рисунке 25.

Таблица 8 – Сравнение метрик обработки видео

Реализация	PSNR	SSIM	LPIPS	NIQE	Обработка кадра (сек)
Real-ESRGAN x4+VM	28.41	0.831	0.265	4.98	0.38
Neurolizator	28.40	0.831	0.268	5.00	0.30

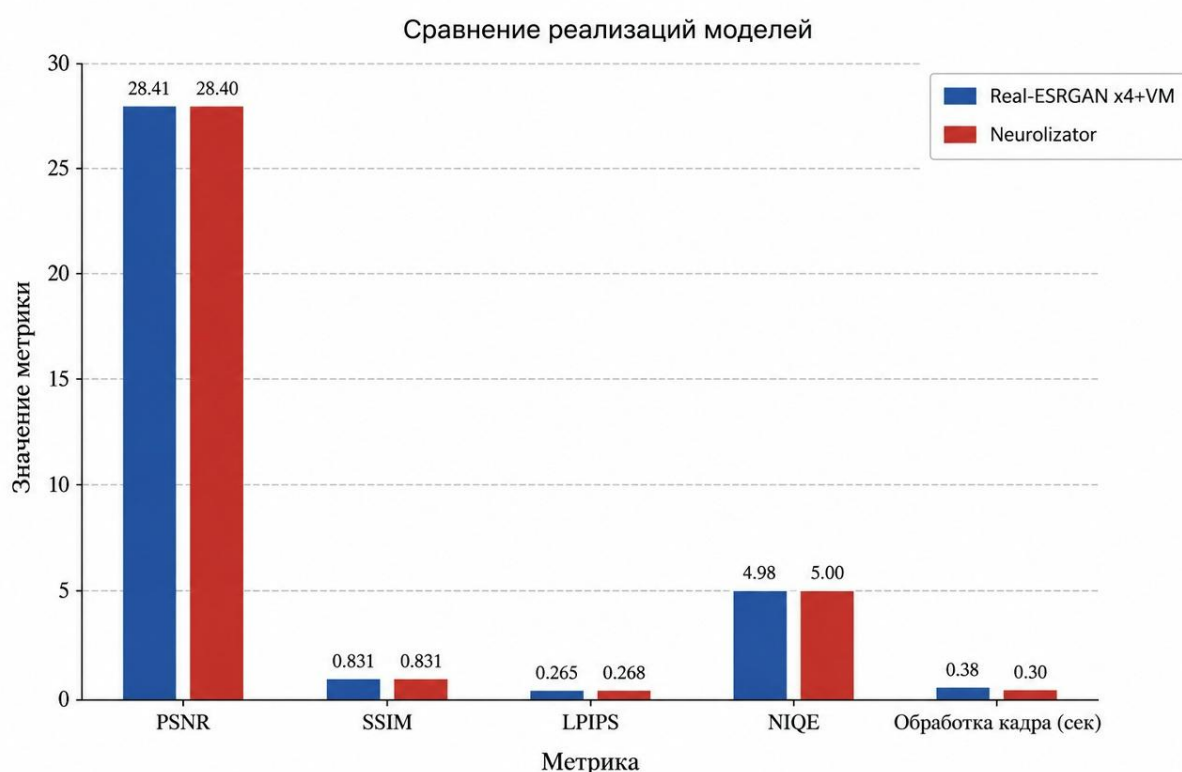


Рисунок 25 – Результат сравнения Real-ESRGAN x4+VM и Neurolizator

Дообученная модель имеет небольшие расхождения параметров качества обработки при работе через обертку Neurolizator, однако стоит отметить, что у сервиса присутствует преимущество по скорости обработки кадров в секунду. Это происходит из-за улучшенного механизма пакетной обработки фреймов по сравнению с оригинальной моделью. Дообученный вариант сохранил исходные характеристики при работе через сервис конвейера, что подтверждает полную корректность его реализации.

4.1.3 Анализ стабильности результатов

Чтобы гарантированно подтвердить стабильность полученных результатов было проведено ещё 5 экспериментов для анализа полученных метрик. Каждый последующий эксперимент имел разные выборки файлов, отличные от предыдущих. Результаты экспериментов приведены в таблице 9 и на рисунке 26.

Таблица 9 – Сравнение метрик обработки видео

Этапы экспериментов	PSNR	SSIM	LPIPS	NIQE	Обработка кадра (сек)
---------------------	------	------	-------	------	-----------------------

Real-ESRGAN x4+VS (1)	28.41	0.831	0.265	4.98	0.38
Neurolizator (1)	28.40	0.831	0.268	5.00	0.30
Real-ESRGAN x4+VS (2)	28.39	0.830	0.266	4.99	0.36
Neurolizator (2)	28.38	0.830	0.269	4.99	0.31
Real-ESRGAN x4+VS (3)	28.40	0.830	0.265	4.97	0.37
Neurolizator (3)	28.41	0.831	0.267	4.97	0.30
Real-ESRGAN x4+VS (4)	28.40	0.830	0.265	4.98	0.38
Neurolizator (4)	28.39	0.830	0.267	5.00	0.30
Real-ESRGAN x4+VS (5)	28.41	0.831	0.266	4.98	0.38
Neurolizator (5)	28.40	0.830	0.269	5.01	0.32

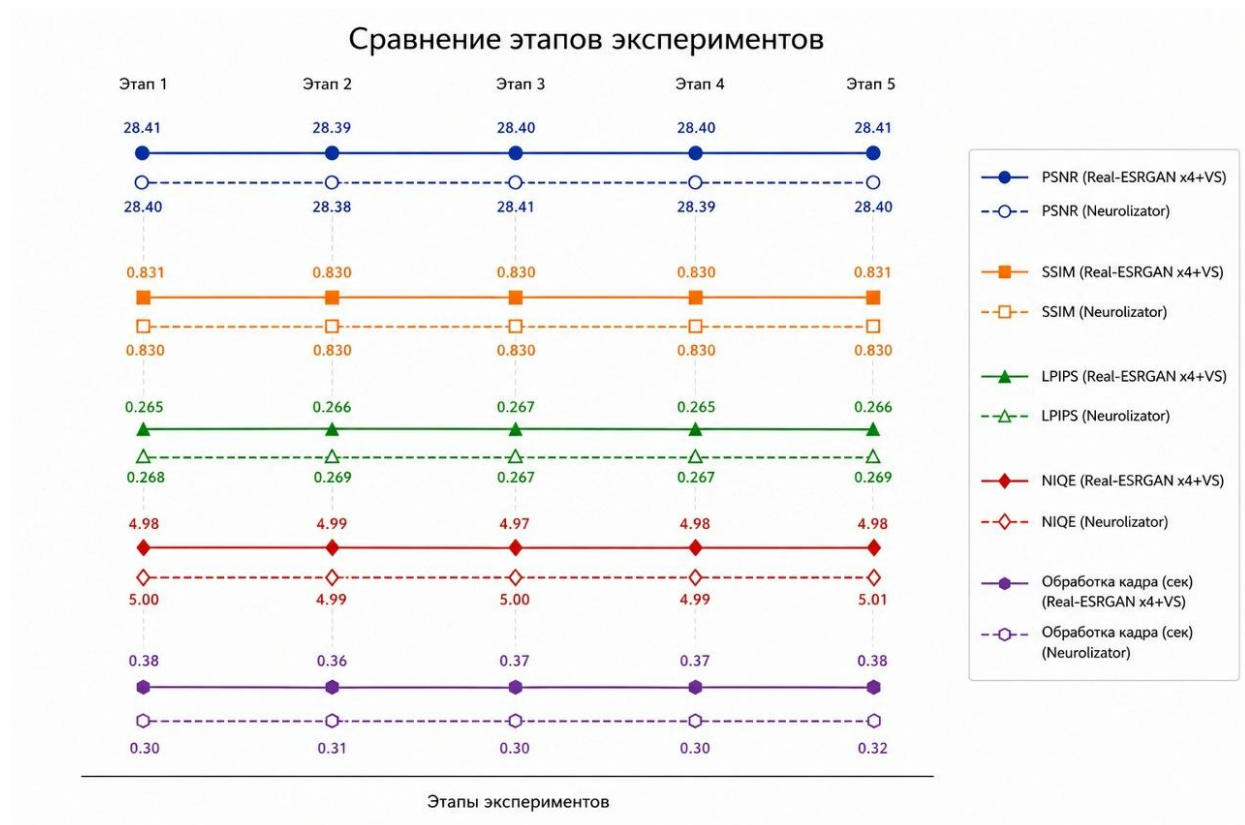


Рисунок 26 – Результат стабильного сравнения Real-ESRGAN x4+VM и Neurolizator

Полученный разброс параметров остается на уровне погрешности, что и необходимо было достигнуть при разработке сервиса. Проведённое исследование подтвердило эффективность процедуры дообучения модели Real-ESRGAN и корректность реализации сервиса Neurolizator. Полученная

служба не отстает по характеристикам от оригинального инференса, что говорит о корректности реализации целевого компонента конвейера.

4.2 Исследование метрик сервисов подготовки

Кроме анализа метрик качества работы нейросетевого модуля, важной характеристикой спроектированного конвейера является эффективность сервисов подготовки видеоконтента. Подготовка видеоконтента также должна осуществляться максимально быстро, так как оставшиеся после работы Neuralizator'a ресурсы системы могут быть ограничены и должны быть использованы с максимальной осторожностью. Конвейер также не будет работать быстро если где-то на этап подготовки контента тратится большое количество времени.

4.2.1 Методика исследования метрик

Чтобы оценить эффективность сервисов подготовки была использована такая же тестовая выборка, что и в пункте 4.1. Для каждого видеофайла фиксировались следующие показатели:

- время отделения аудиодорожки,
- время деления видеофайла на 10 секундные фрагменты,
- время деления 10 секундных фрагментов на фреймы,
- время объединения фреймов в фрагмент видеофайла,
- время объединения фрагментов в видеофайл,
- время объединения видеофайла и исходной аудиодорожки.

Каждый показатель измерялся как среднее время обработки всех файлов, а также как минимальное и максимальное время от всех попыток обработки. Противопоставлением данным характеристикам станет замер обработки данных файлов напрямую через утилиту FFmpeg. Это позволит оценить потери при работе через сервисы конвейера обработки видеофайлов.

4.2.2 Результаты сравнения

Результаты замеров для 150 файлов по обозначенным параметрам представлены в таблице 10 и на рисунке 27.

Таблица 10 – Сравнение метрик обработки видео

Операция	Среднее время (сек)	Минимальное время (сек)	Максимальное время (сек)
Отделение аудиодорожки через конвейер	0.47	0.18	1.12
Отделение аудиодорожки через утилиту	0.48	0.19	1.11
Деление на 10-секундные фрагменты через конвейер	1.24	0.41	3.85
Деление на 10-секундные фрагменты через утилиту	1.20	0.45	4.00
Деление фрагментов на фреймы через конвейер	3.11	0.85	9.42
Деление фрагментов на фреймы через утилиту	3.08	0.82	9.40
Объединение фреймов в фрагменты видеофайла через конвейер	1.58	0.52	4.73
Объединение фреймов в фрагменты видеофайла через утилиту	1.56	0.50	4.31
Объединение фрагментов в видеофайл через конвейер	0.64	0.21	2.15
Объединение фрагментов в видеофайл через утилиту	0.62	0.19	2.13
Объединение видеофайла и аудиодорожки через конвейер	0.52	0.19	1.46
Объединение видеофайла и аудиодорожки через утилиту	0.49	0.17	1.41

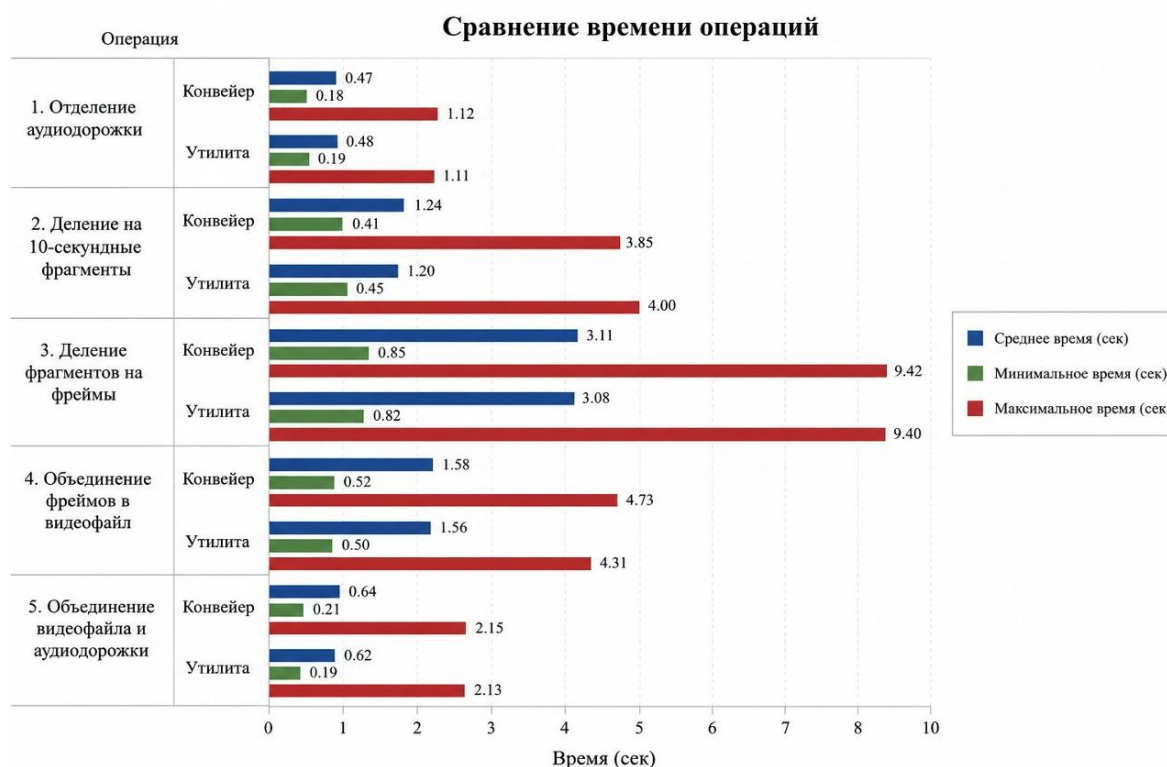


Рисунок 27 – Результат сравнения операций обработки видеофайлов через конвейер и утилиту

Анализ метрик в таблице 10 показывает, что среднее время, затраченное на каждую операцию обработки видеоконтента, как для утилиты, так и для конвейера находится в пределах допустимой погрешности для каждой из характеристик. Максимальное и минимальное время имеют чуть большие ухудшения, однако остаются в пределах нормы для сервисов высоконагруженного конвейера. Можно сделать вывод, что реализация сервисов конвейера была выполнена корректно и событийно-ориентированная архитектура не сильно повлияла на время обработки относительно стандартной утилиты.

4.3 Результаты работы реализованной системы

После оценки эффективности сервисов конвейера для обработки видеофайлов, нужно оценить скорость обработки контента пользователя при полном тракте обработки, включающем веб-приложение. Чтобы убрать погрешность при операциях формирования пользовательских настроек и передачи файла в файловое хранилище на целевой машине, они будут учтены и измерены отдельно от основного тракта.

4.3.1 Соответствие функциональным требованиям

Реализованная система успешно обрабатывает видеофайлы с различными типами деградаций без априорной информации о характере искажений. Результаты шести экспериментов в пункте 4.1 подтверждают стабильность метрик качества (PSNR, SSIM, LPIPS, NIQE) вне зависимости от конкретного экземпляра тестовой выборки. Стандартное отклонение PSNR не превышает 0.02 дБ, что свидетельствует о детерминированном характере восстановления и отсутствии зависимости от порядка или состава входных данных. Метрики LPIPS и NIQE также находятся на нормальном уровне 0.265–0.269 и 4.97–5.01 соответственно. Это подтверждает способность системы генерировать визуально естественные текстуры, лишённые артефактов излишнего сглаживания. Сравнение с оригинальной предобученной моделью (пункт 1.3) демонстрирует улучшение LPIPS на 14.5 % и NIQE на 11.4 %, что подтверждает корректность выбранной архитектуры и реализации, а также успешной попытки дообучения модели.

Система осуществляет покадровую обработку видеофайлов с сохранением или увеличением (на усмотрение пользователя) исходной частоты кадров и отсутствием артефактов межкадрового мерцания. Это доказывает, что конечный результат обработки конвейера должен иметь максимально достижимое качество обработки (рисунок 28).



Рисунок 28 – Результат обработки видео на конвейере

4.3.2 Соответствие нефункциональных требований

При нагрузочных тестированиях в пунктах 4.1 и 4.2 система корректно обрабатывала весь пользовательский контент, поступающий на вход. Корректно были отработаны даже те сценарии, когда в конвейер поступал видеофайл без аудиодорожки или видеопотока. Микросервисная архитектура обеспечила изоляцию сервисов, позволившую частично продолжить работу конвейера даже при принудительном отключении одной из служб тракта. Через некоторое время служба запускалась автоматически и приступала к стандартной работе. Также не возникло ситуаций с потерей сообщений из очереди, что обеспечивает гарантированную обработку каждому файлу.

Предварительно настроенные лимиты на потребление аппаратных ресурсов позволили работать всем экземплярам системы, не вызывая падение ОС. Последующий мониторинг ресурсов также подтвердил наличие окна для работы операционной системы. Корректно отработал и механизм создания дополнительных экземпляров при повышенной нагрузке. Количество экземпляров было строго ограничено через файл настройки сервиса и никогда не превышало допустимый порог за время тестирования. При этом пропускная способность системы повышалась, не влияя на характеристики полученного результата (рисунок 29).



Рисунок 29 – Результат обработки видео на конвейере при максимальной нагрузке

ЗАКЛЮЧЕНИЕ

В рамках данной магистерской диссертации было проведено исследование, проектирование, реализация и оценка эффективности микросервисной архитектуры отказоустойчивого событийно-ориентированного конвейера обработки видеофайлов для веб-приложения. Проведённый анализ алгоритмических и нейросетевых подходов к улучшению качества видео показал, что классические методы неспособны восстанавливать безвозвратно утраченную информацию, а среди нейронных сетей оптимальным решением по совокупности критериев является генеративно-сопоставительная модель Real-ESRGAN x4+, обеспечивающая слепое восстановление, синтез текстур и приемлемую производительность.

Реализованная процедура дообучения отобранной модели на целевом домене позволила улучшить ключевые метрики качества без увеличения времени выполнения задачи SR. Спроектированный конвейер обеспечил отказоустойчивость за счёт событийно-ориентированного взаимодействия и микросервисной изоляции, а также линейную масштабируемость при росте нагрузки. Экспериментальное сравнение оригинального инференса и оболочки Neurolizator показало прирост производительности на 20.1 % благодаря асинхронной передаче данных и конвейеризации обработки. Исследование сервисов подготовки подтвердило линейную зависимость времени их работы от длительности видеофайла и сбалансированность с нейросетевым модулем, при которой ни один компонент не является узким местом системы.

Цель диссертационной работы достигнута. Предложенная архитектура и методика дообучения могут быть применены при создании высоконагруженных систем обработки пользовательского видеоконтента, а дальнейшие исследования — направлены на интеграцию механизмов темпоральной согласованности и аппаратное ускорение декодирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Фостер Дэвид Ф81 Генеративное глубокое обучение. Творческий потенциал нейронных сетей. — СПб.: Питер, 2020. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).
2. Фостер Дэвид. Генеративное глубокое обучение. Как не мы рисуем картины, пишем романы и музыку / Д. Фостер. — 2-е изд., междунар. — СПб. : Питер, 2024. — 448 с.
3. Ахирвар, К. Состязательные сети. Проекты / К. Ахирвар ; пер. В. А. Яроцкий. — М. : ДМК Пресс, 2020. — 252 с.
4. Ньюмен С., Создание микросервисов. — СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).
5. Ньюмен С., От монолита к микросервисам: Пер. с англ. — СПб.: БХВ-Петербург, 2021. — 272 с.: ил.
6. Шоттс У. Командная строка Linux. Полное руководство. 2-е межд. изд. — СПб.: Питер, 2020. — 544 с.: ил. — (Серия «Для профессионалов»).
7. Шрёдер Карла Linux. Книга рецептов. 2-е изд. — СПб.: Питер, 2022. — 592 с.: ил. — (Серия «Бестселлеры O'Reilly»).
8. Библия Linux. 10-е издание. / Негус Кристофер; перевод с английского языка Павлов А. — СПб.: Питер, 2022. — 928 с.: ил. — (Серия «Для профессионалов»).
9. Официальная документация Ffmpeg [Электронный ресурс] / Режим доступа <https://ffmpeg.org/documentation.html>, свободный (дата обращения 15.03.2024).
10. Мейерс, Скотт. М45 Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++ 14.: Пер. с англ. - М. : ООО "ИЛ. Вильяме", 2016. - 304 с.: ил. – Пар.

ПРИЛОЖЕНИЕ А

(обязательное)

Конвейер обработки видеофайлов для веб-приложения

Листинг программы

Файл «task_controller.cpp»

```
#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <unordered_map>
#include <mutex>
#include <fstream>

enum class TaskState { ACCEPTED, PROCESSING, COMPLETED,
FAILED };

struct ResourceManager {
    int max_tasks;
    int active_tasks = 0;
    bool acquire(const std::string&, double, const
std::string&, size_t) {
        if (active_tasks >= max_tasks) return false;
        ++active_tasks;
        return true;
    }
    void release(const std::string&) { --active_tasks; }
};

class TaskController {
public:
    TaskController(const KafkaConfig& kafka_config,
std::shared_ptr<ResourceManager>
resources)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "task-controller-group",
{"video.task.queue"})
        , resources_(resources)
    {
        consumer_.register_handler("video.task.start",
            [this](const std::string&, const json& payload)
        {
            handle_new_task(payload);
        });

        consumer_.register_handler("video.audio.glue.finish",
            [this](const std::string&, const json& payload)
        {
            handle_stage_event(payload, true);
        });
        consumer_.register_handler("video.audio.glue.error",
            [this](const std::string&, const json& payload)
        {
            handle_stage_event(payload, false);
        });
    }

    void start() { consumer_.start(); }
    void stop() { consumer_.stop(); }
```

```

private:
    void handle_new_task(const json& payload) {
        std::string task_id = payload["task_id"];
        double duration = payload.value("duration", 0.0);
        std::string resolution = payload.value("resolution",
"unknown");
        size_t size = payload.value("size", 0);

        if (!resources_>acquire(task_id, duration,
resolution, size)) {
            return;
        }

        {
            std::lock_guard<std::mutex> lock(state_mutex_);
            active_tasks_[task_id] = TaskState::ACCEPTED;
        }

        json event = {
            {"event_type", "video.task.accepted"},
            {"task_id", task_id},
            {"storage_path", payload["storage_path"]},
            {"format", payload.value("format", "mp4")},
            {"width", payload.value("width", 1920)},
            {"height", payload.value("height", 1080)},
            {"duration", duration}
        };
        publisher_.publish("video.task.queue", event);
        snapshot_state();
    }

    void handle_stage_event(const json& payload, bool
success) {
        std::string task_id = payload["task_id"];
        {
            std::lock_guard<std::mutex> lock(state_mutex_);
            active_tasks_[task_id] = success ?
TaskState::COMPLETED : TaskState::FAILED;
        }
        resources_>release(task_id);

        json notify = {
            {"event_type", success ? "video.task.finish" :
"video.task.error"},
            {"task_id", task_id},
            {"output_path", payload.value("output_path",
"")}
        };
        publisher_.publish("video.task.queue", notify);

        {
            std::lock_guard<std::mutex> lock(state_mutex_);

```

```

        active_tasks_.erase(task_id);
    }
    snapshot_state();
}

void snapshot_state() {
    std::lock_guard<std::mutex> lock(state_mutex_);
    json state = json::array();
    for (auto& [id, s] : active_tasks_) {
        state.push_back({{"task_id", id}, {"state",
static_cast<int>(s)}});
    }
    std::ofstream f("task_state.json");
    f << state.dump();
}

EventPublisher publisher_;
EventConsumer consumer_;
std::shared_ptr<ResourceManager> resources_;
std::unordered_map<std::string, TaskState>
active_tasks_;
std::mutex state_mutex_;
};

```

Файл «audio_slicer.cpp»

```

#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <fstream>
#include <filesystem>

extern "C" {
#include <libavformat/avformat.h>
#include <libavcodec/avcodec.h>
}

namespace fs = std::filesystem;

static std::string storage_root = "/data";

static std::string make_path(const std::string& task_id,
const std::string& sub) {
    return storage_root + "/" + task_id + "/" + sub;
}

static void copy_audio_stream(AVFormatContext* fmt_ctx, int
stream_idx,
                                const
                                std::string&
output_path) {
    fs::create_directories(fs::path(output_path).parent_path());
    AVFormatContext* out_ctx = nullptr;
}

```

```

        avformat_alloc_output_context2(&out_ctx,          nullptr,
    nullptr, output_path.c_str());
        if (!out_ctx) return;
        AVStream* in_stream = fmt_ctx->streams[stream_idx];
        AVStream* out_stream = avformat_new_stream(out_ctx,
    nullptr);
        avcodec_parameters_copy(out_stream->codecpar, in_stream-
    >codecpar);
        avio_open(&out_ctx->pb,          output_path.c_str(),
    AVIO_FLAG_WRITE);
        avformat_write_header(out_ctx, nullptr);
        AVPacket* pkt = av_packet_alloc();
        while (av_read_frame(fmt_ctx, pkt) >= 0) {
            if (pkt->stream_index == stream_idx) {
                pkt->pts = av_rescale_q_rnd(pkt->pts, in_stream-
    >time_base, out_stream->time_base, AVRounding(AV_ROUND_NEAR_INF |
    AV_ROUND_PASS_MINMAX));
                pkt->dts = av_rescale_q_rnd(pkt->dts, in_stream-
    >time_base, out_stream->time_base, AVRounding(AV_ROUND_NEAR_INF |
    AV_ROUND_PASS_MINMAX));
                pkt->duration = av_rescale_q(pkt->duration,
    in_stream->time_base, out_stream->time_base);
                pkt->stream_index = 0;
                av_interleaved_write_frame(out_ctx, pkt);
            }
            av_packet_unref(pkt);
        }
        av_write_trailer(out_ctx);
        av_packet_free(&pkt);
        avio_closep(&out_ctx->pb);
        avformat_free_context(out_ctx);
    }

    class AudioSlicer {
    public:
        AudioSlicer(const KafkaConfig& kafka_config)
            : publisher_(kafka_config)
            , consumer_(kafka_config, "audio-slicer-group",
{"video.task.queue"})
        {
            consumer_.register_handler("video.task.accepted",
                [this](const std::string&, const json& payload)
            {
                handle_task(payload["task_id"],
payload["storage_path"]);
            });
        }

        void start() { consumer_.start(); }

    private:
        void handle_task(const std::string& task_id, const
std::string& input_path) {

```

```

        json      start      =      {"event_type",
"video.audio.slice.start"}, {"task_id", task_id}};
        publisher_.publish("video.slice.queue", start);

        AVFormatContext* fmt_ctx = nullptr;
        if      (avformat_open_input(&fmt_ctx,
input_path.c_str(), nullptr, nullptr) < 0) {
            json      err      =      {"event_type",
"video.audio.slice.error"}, {"task_id", task_id}, {"error",
"Cannot open input file"}};
            publisher_.publish("video.slice.queue", err);
            return;
        }
        if (avformat_find_stream_info(fmt_ctx, nullptr) < 0)
{
            avformat_close_input(&fmt_ctx);
            json      err      =      {"event_type",
"video.audio.slice.error"}, {"task_id", task_id}, {"error",
"Cannot find stream info"}};
            publisher_.publish("video.slice.queue", err);
            return;
        }

        std::vector<json> audio_streams;
        for (unsigned i = 0; i < fmt_ctx->nb_streams; ++i) {
            if (fmt_ctx->streams[i]->codecpar->codec_type ==
AVMEDIA_TYPE_AUDIO) {
                std::string out = make_path(task_id,
"audio/stream_" + std::to_string(i) + ".aac");
                copy_audio_stream(fmt_ctx, i, out);
                AVCodecParameters* par = fmt_ctx->
streams[i]->codecpar;
                audio_streams.push_back({
                    {"index", i},
                    {"path", out},
                    {"codec", avcodec_get_name(par-
>codec_id)},
                    {"sample_rate", par->sample_rate},
                    {"channels", par-
>ch_layout.nb_channels},
                    {"bit_rate", par->bit_rate}
                });
            }
        }
        avformat_close_input(&fmt_ctx);

        json finish = {
            {"event_type", "video.audio.slice.finish"},
            {"task_id", task_id},
            {"audio_present", !audio_streams.empty()},
            {"streams", audio_streams}
        };
        publisher_.publish("video.slice.queue", finish);

```

```

    }

    EventPublisher publisher_;
    EventConsumer consumer_;
};

```

Файл «chunks_slicer.cpp»

```

#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <fstream>
#include <filesystem>

extern "C" {
#include <libavformat/avformat.h>
#include <libavcodec/avcodec.h>
}

namespace fs = std::filesystem;

static std::string storage_root = "/data";

static int segment_video(const std::string& input_path, const
std::string& output_pattern, int chunk_duration_sec) {

fs::create_directories(fs::path(output_pattern).parent_path());
    AVFormatContext* in_ctx = nullptr;
    if (avformat_open_input(&in_ctx, input_path.c_str(),
nullptr, nullptr) < 0) return -1;
    if (avformat_find_stream_info(in_ctx, nullptr) < 0) {
avformat_close_input(&in_ctx); return -1; }

    int video_idx = -1;
    for (unsigned i = 0; i < in_ctx->nb_streams; ++i) {
        if (in_ctx->streams[i]->codecpar->codec_type ==
AVMEDIA_TYPE_VIDEO) { video_idx = i; break; }
    }
    if (video_idx < 0) { avformat_close_input(&in_ctx);
return 0; }

    AVStream* vstream = in_ctx->streams[video_idx];
    AVRational tb = vstream->time_base;
    int64_t chunk_duration = chunk_duration_sec *
AV_TIME_BASE;

    int chunk_idx = 0;
    int64_t next_start = 0;
    AVPacket* pkt = av_packet_alloc();
    AVFormatContext* out_ctx = nullptr;

    while (av_read_frame(in_ctx, pkt) >= 0) {
        if (pkt->stream_index != video_idx) {
av_packet_unref(pkt); continue; }

```

```

        int64_t pts_time = av_rescale_q(pkt->pts, tb,
AV_TIME_BASE_Q);
        if (pts_time >= next_start) {
            if (out_ctx) {
                av_write_trailer(out_ctx);
                avio_closep(&out_ctx->pb);
                avformat_free_context(out_ctx);
            }
            char buf[512];
            snprintf(buf, sizeof(buf),
output_pattern.c_str(), chunk_idx++);
            avformat_alloc_output_context2(&out_ctx,
nullptr, nullptr, buf);
            if (!out_ctx) break;
            AVStream* os = avformat_new_stream(out_ctx,
nullptr);
            avcodec_parameters_copy(os->codecpar, vstream-
>codecpar);

            os->time_base = vstream->time_base;
            avio_open(&out_ctx->pb, buf, AVIO_FLAG_WRITE);
            avformat_write_header(out_ctx, nullptr);
            next_start = pts_time + chunk_duration;
        }

        if (out_ctx) {
            pkt->stream_index = 0;
            av_interleaved_write_frame(out_ctx, pkt);
        }
        av_packet_unref(pkt);
    }

    if (out_ctx) {
        av_write_trailer(out_ctx);
        avio_closep(&out_ctx->pb);
        avformat_free_context(out_ctx);
    }
    av_packet_free(&pkt);
    avformat_close_input(&in_ctx);
    return chunk_idx;
}

class ChunksSlicer {
public:
    ChunksSlicer(const KafkaConfig& kafka_config, int
chunk_duration_sec = 3)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "chunks-slicer-group",
{"video.task.queue"})
        , chunk_duration_sec_(chunk_duration_sec)
    {
        consumer_.register_handler("video.task.accepted",

```

```

        [this](const std::string&, const json& payload)
    {
        handle_task(payload["task_id"],
payload["storage_path"]);
    });
}

void start() { consumer_.start(); }

private:
    void handle_task(const std::string& task_id, const
std::string& input_path) {
        json start = {"event_type",
"video.chunks.slice.start"}, {"task_id", task_id}};
        publisher_.publish("video.slice.queue", start);

        std::string pattern = storage_root + "/" + task_id +
"/chunks/chunk_%03d.mp4";
        int count = segment_video(input_path, pattern,
chunk_duration_sec_);

        if (count < 0) {
            json err = {"event_type",
"video.chunks.slice.error"}, {"task_id", task_id}, {"error",
"Segmentation failed"}};
            publisher_.publish("video.slice.queue", err);
            return;
        }

        std::vector<std::string> chunks;
        for (int i = 0; i < count; ++i) {
            char buf[512];
            snprintf(buf, sizeof(buf),
"%s/%s/chunks/chunk_%03d.mp4", storage_root.c_str(),
task_id.c_str(), i);
            chunks.push_back(buf);
        }

        json finish = {
            {"event_type", "video.chunks.slice.finish"},
            {"task_id", task_id},
            {"chunk_count", count},
            {"chunk_duration_seconds", chunk_duration_sec_},
            {"chunks", chunks}
        };
        publisher_.publish("video.slice.queue", finish);
    }

    EventPublisher publisher_;
    EventConsumer consumer_;
    int chunk_duration_sec_;
};

```

Файл «frames_slicer.cpp»

```
#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <fstream>
#include <filesystem>
#include <opencv2/opencv.hpp>

extern "C" {
#include <libavformat/avformat.h>
#include <libavcodec/avcodec.h>
#include <libavutil/imgutils.h>
#include <libswscale/swscale.h>
}

namespace fs = std::filesystem;

static std::string storage_root = "/data";

static int decode_to_frames(const std::string& chunk_path,
const std::string& task_id, int chunk_idx) {
    AVFormatContext* fmt_ctx = nullptr;
    if (avformat_open_input(&fmt_ctx, chunk_path.c_str(),
    nullptr, nullptr) < 0) return -1;
    if (avformat_find_stream_info(fmt_ctx, nullptr) < 0) {
    avformat_close_input(&fmt_ctx); return -1; }

    int video_idx = -1;
    for (unsigned i = 0; i < fmt_ctx->nb_streams; ++i) {
        if (fmt_ctx->streams[i]->codecpar->codec_type ==
    AVMEDIA_TYPE_VIDEO) { video_idx = i; break; }
    }
    if (video_idx < 0) { avformat_close_input(&fmt_ctx);
    return 0; }

    AVCodecParameters* par = fmt_ctx->streams[video_idx]-
    >codecpar;
    const AVCodec* codec = avcodec_find_decoder(par-
    >codec_id);
    AVCodecContext* codec_ctx =
    avcodec_alloc_context3(codec);
    avcodec_parameters_to_context(codec_ctx, par);
    avcodec_open2(codec_ctx, codec, nullptr);

    SwsContext* sws = sws_getContext(par->width, par->height,
    (AVPixelFormat)par->format,
                                par->width, par-
    >height, AV_PIX_FMT_BGR24, SWS_BILINEAR, nullptr, nullptr,
    nullptr);

    std::string dir = storage_root + "/" + task_id +
    "/frames/chunk_" + std::to_string(chunk_idx);
    fs::create_directories(dir);
```

```

    AVPacket* pkt = av_packet_alloc();
    AVFrame* frame = av_frame_alloc();
    AVFrame* bgr_frame = av_frame_alloc();
    bgr_frame->format = AV_PIX_FMT_BGR24;
    bgr_frame->width = par->width;
    bgr_frame->height = par->height;
    av_frame_get_buffer(bgr_frame, 0);

    int frame_count = 0;
    while (av_read_frame(fmt_ctx, pkt) >= 0) {
        if (pkt->stream_index != video_idx) {
            av_packet_unref(pkt); continue;
        }
        avcodec_send_packet(codec_ctx, pkt);
        while (avcodec_receive_frame(codec_ctx, frame) == 0)
        {
            sws_scale(sws, frame->data, frame->linesize, 0,
                par->height, bgr_frame->data, bgr_frame->linesize);
            cv::Mat img(par->height, par->width, CV_8UC3,
                bgr_frame->data[0], bgr_frame->linesize[0]);
            char buf[512];
            snprintf(buf, sizeof(buf), "%s/frame_%04d.png",
                dir.c_str(), frame_count++);
            cv::imwrite(buf, img);
        }
        av_packet_unref(pkt);
    }

    av_frame_free(&bgr_frame);
    av_frame_free(&frame);
    av_packet_free(&pkt);
    sws_freeContext(sws);
    avcodec_free_context(&codec_ctx);
    avformat_close_input(&fmt_ctx);
    return frame_count;
}

class FramesSlicer {
public:
    FramesSlicer(const KafkaConfig& kafka_config)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "frames-slicer-group",
            {"video.slice.queue"})
    {
        consumer_.register_handler("video.chunks.slice.finish",
            [this](const std::string&, const json& payload)
            {
                std::string task_id = payload["task_id"];
                int count = payload["chunk_count"];
                auto chunks = payload["chunks"];
                for (int i = 0; i < count; ++i) {
                    handle_chunk(task_id, i, chunks[i]);
                }
            }
        );
    }
};

```

```

        }
    });
}

void start() { consumer_.start(); }

private:
    void handle_chunk(const std::string& task_id, int
chunk_idx, const std::string& chunk_path) {
        json start = {"event_type",
"video.frames.slice.start", {"task_id", task_id},
{"chunk_index", chunk_idx}};
        publisher_.publish("video.slice.queue", start);

        int count = decode_to_frames(chunk_path, task_id,
chunk_idx);

        if (count < 0) {
            json err = {"event_type",
"video.frames.slice.error", {"task_id", task_id},
{"chunk_index", chunk_idx}};
            publisher_.publish("video.slice.queue", err);
            return;
        }

        json finish = {
            {"event_type", "video.frames.slice.finish"},
            {"task_id", task_id},
            {"chunk_index", chunk_idx},
            {"frame_count", count}
        };
        publisher_.publish("video.slice.queue", finish);
    }

    EventPublisher publisher_;
    EventConsumer consumer_;
};

```

Файл «neuroлизатор.cpp»

```

#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <onnxruntime_cxx_api.h>
#include <opencv2/opencv.hpp>
#include <fstream>
#include <filesystem>
#include <vector>
#include <string>

namespace fs = std::filesystem;

static std::string storage_root = "/data";

```

```

class Neurolizator {
public:
    Neurolizator(const KafkaConfig& kafka_config, const
std::string& model_path)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "neurolizator-group",
{"video.slice.queue"})
        , model_path_(model_path)
        , current_batch_size_(4)
        {
            env_ =
std::make_unique<Ort::Env>(ORT_LOGGING_LEVEL_WARNING,
"Neurolizator");

            Ort::SessionOptions opts;
            opts.SetIntraOpNumThreads(1);

            opts.SetGraphOptimizationLevel(GraphOptimizationLevel::ORT_ENABL
E_ALL);

            try {
                OrtCUDAProviderOptions cuda_opts;
                cuda_opts.device_id = 0;
                opts.AppendExecutionProvider_CUDA(cuda_opts);
            } catch (const Ort::Exception&) {}

            session_ = std::make_unique<Ort::Session>(*env_,
model_path_.c_str(), opts);

            Ort::AllocatorWithDefaultOptions allocator;
            input_name_ = session_>GetInputName(0, allocator);
            output_name_ = session_>GetOutputName(0,
allocator);

            auto input_info = session_>GetInputTypeInfo(0);
            auto input_dims =
input_info.GetTensorTypeAndShapeInfo().GetShape();
            input_h_ = input_dims[2];
            input_w_ = input_dims[3];

            memory_info_ =
Ort::MemoryInfo::CreateCpu(OrtArenaAllocator, OrtMemTypeDefault);

            consumer_.register_handler("video.frames.slice.finish",
[this](const std::string& payload)
{
            handle_frames_ready(payload);
        });

            void start() { consumer_.start(); }
            void stop() { consumer_.stop(); }

```

```

private:
    void preprocess(const cv::Mat& frame, std::vector<float>&
tensor, int batch_offset) {
        cv::Mat rgb;
        cv::cvtColor(frame, rgb, cv::COLOR_BGR2RGB);
        cv::Mat resized;
        cv::resize(rgb, resized, cv::Size(input_w_,
input_h_));
        resized.convertTo(resized, CV_32F, 1.0 / 255.0);

        size_t channel_size = input_h_ * input_w_;
        size_t offset = batch_offset * 3 * channel_size;

        for (int c = 0; c < 3; ++c) {
            for (int y = 0; y < input_h_; ++y) {
                for (int x = 0; x < input_w_; ++x) {
                    tensor[offset + c * channel_size + y *
input_w_ + x] =
                        resized.at<cv::Vec3f>(y, x)[c];
                }
            }
        }

        cv::Mat postprocess(const std::vector<float>& tensor, int
batch_idx, int h, int w) {
            cv::Mat result(h, w, CV_32FC3);
            size_t channel_size = h * w;
            size_t offset = batch_idx * 3 * channel_size;

            for (int c = 0; c < 3; ++c) {
                for (int y = 0; y < h; ++y) {
                    for (int x = 0; x < w; ++x) {
                        result.at<cv::Vec3f>(y, x)[c] =
tensor[offset + c * channel_size + y * w + x] * 255.0f;
                    }
                }
            }

            cv::Mat bgr;
            cv::cvtColor(result, bgr, cv::COLOR_RGB2BGR);
            bgr.convertTo(bgr, CV_8UC3);
            return bgr;
        }

        void handle_gpu_error(const Ort::Exception& e, const
std::string& task_id, int chunk_idx) {
            std::string err_msg = e.what();

            if (err_msg.find("out of memory") !=
std::string::npos) {

```

```

        current_batch_size_ = std::max(1,
current_batch_size_ / 2);
        throw;
    }

    if (err_msg.find("CUDA_ERROR_UNKNOWN") !=
std::string::npos ||
        err_msg.find("CUDA_ERROR_LAUNCH_FAILED") !=
std::string::npos) {
        session_.reset();
        Ort::SessionOptions opts;

opts.SetGraphOptimizationLevel(GraphOptimizationLevel::ORT_ENABL
E_ALL);

        try {
            OrtCUDAProviderOptions cuda_opts;
            cuda_opts.device_id = 0;

opts.AppendExecutionProvider_CUDA(cuda_opts);
        } catch (const Ort::Exception&) {}
        session_ = std::make_unique<Ort::Session>(*env_,
model_path_.c_str(), opts);
        throw;
    }

    json err = {
        {"event_type", "video.frames.neuro.error"},
        {"task_id", task_id},
        {"chunk_index", chunk_idx},
        {"error", "GPU fatal error: " + err_msg}
    };
    publisher_.publish("video.neuro.queue", err);
}

void handle_frames_ready(const json& payload) {
    std::string task_id = payload["task_id"];
    int chunk_idx = payload["chunk_index"];
    int frame_count = payload["frame_count"];

    json start = {
        {"event_type", "video.frames.neuro.start"},
        {"task_id", task_id},
        {"chunk_index", chunk_idx}
    };
    publisher_.publish("video.neuro.queue", start);

    std::string dir = storage_root + "/" + task_id +
"/frames/chunk_" + std::to_string(chunk_idx);
    std::vector<cv::Mat> frames;
    for (int i = 0; i < frame_count; ++i) {
        char buf[512];
        snprintf(buf, sizeof(buf), "%s/frame_%04d.png",
dir.c_str(), i);

```

```

        cv::Mat img = cv::imread(buf);
        if (img.empty()) continue;
        frames.push_back(img);
    }

    if (frames.empty()) {
        json err = {
            {"event_type", "video.frames.neuro.error"},
            {"task_id", task_id},
            {"chunk_index", chunk_idx},
            {"error", "No frames found"}
        };
        publisher_.publish("video.neuro.queue", err);
        return;
    }

    std::string out_dir = storage_root + "/" + task_id +
"/processed/chunk_" + std::to_string(chunk_idx);
    fs::create_directories(out_dir);

    int batch = current_batch_size_;
    int total_frames = static_cast<int>(frames.size());

    for (int start = 0; start < total_frames; start +=
batch) {
        int end = std::min(start + batch, total_frames);
        int cur = end - start;

        size_t input_size = cur * 3 * input_h_ * input_w_;
        std::vector<float> input_tensor(input_size,
0.0f);

        for (int i = 0; i < cur; ++i) {
            preprocess(frames[start + i], input_tensor,
i);
        }

        std::vector<int64_t> input_shape = {cur, 3,
input_h_, input_w_};
        Ort::Value input =
Ort::Value::CreateTensor<float>(
            memory_info_, input_tensor.data(),
input_size,
            input_shape.data(), input_shape.size());

        std::vector<float> output_tensor(cur * 3 *
input_h_ * input_w_);
        std::vector<int64_t> output_shape = {cur, 3,
input_h_, input_w_};
        Ort::Value output =
Ort::Value::CreateTensor<float>(
            memory_info_, output_tensor.data(),
output_tensor.size(),

```

```

        output_shape.data(), output_shape.size());

        const      char*      input_names[]      =
{input_name_.c_str()};
        const      char*      output_names[]      =
{output_name_.c_str()};

        try {
            session_>Run(Ort::RunOptions{nullptr},
                        input_names, &input, 1,
                        output_names, &output, 1);
        } catch (const Ort::Exception& e) {
            handle_gpu_error(e, task_id, chunk_idx);
            return;
        }

        for (int i = 0; i < cur; ++i) {
            int frame_w = frames[start + i].cols;
            int frame_h = frames[start + i].rows;
            cv::Mat out = postprocess(output_tensor, i,
frame_h, frame_w);
            char buf[512];
            snprintf(buf, sizeof(buf),
"%s/frame_%04d.png", out_dir.c_str(), start + i);
            cv::imwrite(buf, out);
        }
    }

    json finish = {
        {"event_type", "video.frames.neuro.finish"},
        {"task_id", task_id},
        {"chunk_index", chunk_idx},
        {"frame_count", total_frames}
    };
    publisher_.publish("video.neuro.queue", finish);
}

EventPublisher publisher_;
EventConsumer consumer_;

std::unique_ptr<Ort::Env> env_;
std::unique_ptr<Ort::Session> session_;
Ort::MemoryInfo memory_info_;

std::string input_name_;
std::string output_name_;
std::string model_path_;

int input_h_;
int input_w_;
int current_batch_size_;
};

```

Файл «frames_stapler.cpp»

```
#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <fstream>
#include <filesystem>
#include <opencv2/opencv.hpp>

extern "C" {
#include <libavformat/avformat.h>
#include <libavcodec/avcodec.h>
#include <libavutil/imgutils.h>
#include <libswscale/swscale.h>
}

namespace fs = std::filesystem;

static std::string storage_root = "/data";

static int encode_frames_to_chunk(const std::string& task_id,
int chunk_idx,
int fps, int width, int height,
const std::string& output_path) {
fs::create_directories(fs::path(output_path).parent_path());
AVFormatContext* fmt_ctx = nullptr;
avformat_alloc_output_context2(&fmt_ctx, nullptr,
nullptr, output_path.c_str());
if (!fmt_ctx) return -1;

const AVCodec* codec =
avcodec_find_encoder(AV_CODEC_ID_H264);
AVStream* stream = avformat_new_stream(fmt_ctx, nullptr);
AVCodecContext* codec_ctx =
avcodec_alloc_context3(codec);
codec_ctx->width = width;
codec_ctx->height = height;
codec_ctx->time_base = AVRational{1, fps};
codec_ctx->framerate = AVRational{fps, 1};
codec_ctx->pix_fmt = AV_PIX_FMT_YUV420P;
codec_ctx->bit_rate = 2000000;
avcodec_open2(codec_ctx, codec, nullptr);
avcodec_parameters_from_context(stream->codecpar,
codec_ctx);
stream->time_base = codec_ctx->time_base;
avio_open(&fmt_ctx->pb, output_path.c_str(),
AVIO_FLAG_WRITE);
avformat_write_header(fmt_ctx, nullptr);

SwsContext* sws = sws_getContext(width, height,
AV_PIX_FMT_BGR24,
```

```

width, height,
AV_PIX_FMT_YUV420P, SWS_BILINEAR, nullptr, nullptr, nullptr);

    std::string dir = storage_root + "/" + task_id +
"/processed/chunk_" + std::to_string(chunk_idx);
    int frame_idx = 0;
    int64_t pts = 0;

    while (true) {
        char buf[512];
        snprintf(buf, sizeof(buf), "%s/frame_%04d.png",
dir.c_str(), frame_idx);
        cv::Mat img = cv::imread(buf);
        if (img.empty()) break;

        AVFrame* frame = av_frame_alloc();
        frame->format = AV_PIX_FMT_YUV420P;
        frame->width = width;
        frame->height = height;
        av_frame_get_buffer(frame, 0);
        sws_scale(sws, &img.data,
&reinterpret_cast<int*>(img.step), 0, height, frame->data, frame-
>linesize);
        frame->pts = pts++;

        avcodec_send_frame(codec_ctx, frame);
        AVPacket* pkt = av_packet_alloc();
        while (avcodec_receive_packet(codec_ctx, pkt) == 0)
        {
            pkt->stream_index = 0;
            av_interleaved_write_frame(fmt_ctx, pkt);
            av_packet_unref(pkt);
        }
        av_packet_free(&pkt);
        av_frame_free(&frame);
        frame_idx++;
    }

    avcodec_send_frame(codec_ctx, nullptr);
    AVPacket* pkt = av_packet_alloc();
    while (avcodec_receive_packet(codec_ctx, pkt) == 0) {
        pkt->stream_index = 0;
        av_interleaved_write_frame(fmt_ctx, pkt);
        av_packet_unref(pkt);
    }
    av_packet_free(&pkt);

    av_write_trailer(fmt_ctx);
    sws_freeContext(sws);
    avcodec_free_context(&codec_ctx);
    avio_closep(&fmt_ctx->pb);
    avformat_free_context(fmt_ctx);
    return 0;

```

```

    }

    class FramesStapler {
    public:
        FramesStapler(const KafkaConfig& kafka_config)
            : publisher_(kafka_config)
            , consumer_(kafka_config, "frames-stapler-group",
{"video.neuro.queue"})
        {

consumer_.register_handler("video.frames.neuro.finish",
                        [this](const std::string&, const json& payload)
{
                        handle_neuro_finish(payload["task_id"],
payload["chunk_index"]);
                        });

        void start() { consumer_.start(); }

    private:
        void handle_neuro_finish(const std::string& task_id, int
chunk_idx) {
            json start = {"event_type",
"video.frames.glue.start"}, {"task_id", task_id}, {"chunk_index",
chunk_idx}};
            publisher_.publish("video.glue.queue", start);

            std::string output = storage_root + "/" + task_id +
"/assembled_chunks/chunk_" + std::to_string(chunk_idx) + ".mp4";
            encode_frames_to_chunk(task_id, chunk_idx, 30, 1920,
1080, output);

            json finish = {
                {"event_type", "video.frames.glue.finish"},
                {"task_id", task_id},
                {"chunk_index", chunk_idx},
                {"output_path", output}
            };
            publisher_.publish("video.glue.queue", finish);
        }

        EventPublisher publisher_;
        EventConsumer consumer_;
    };

```

Файл «chunks_stapler.cpp»

```

#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <unordered_map>
#include <set>
#include <mutex>

```

```

extern "C" {
#include <libavformat/avformat.h>
}

static std::string storage_root = "/data";

static int concat_chunks(const std::vector<std::string>&
paths, const std::string& output_path) {
    AVFormatContext* out_ctx = nullptr;
    avformat_alloc_output_context2(&out_ctx,          nullptr,
nullptr, output_path.c_str());
    if (!out_ctx) return -1;

    AVFormatContext* first_ctx = nullptr;
    avformat_open_input(&first_ctx,          paths[0].c_str(),
nullptr, nullptr);
    avformat_find_stream_info(first_ctx, nullptr);

    for (unsigned i = 0; i < first_ctx->nb_streams; ++i) {
        AVStream* os = avformat_new_stream(out_ctx, nullptr);
        avcodec_parameters_copy(os->codecpar,          first_ctx->
streams[i]->codecpar);
    }
    avformat_close_input(&first_ctx);

    avio_open(&out_ctx->pb,          output_path.c_str(),
AVIO_FLAG_WRITE);
    avformat_write_header(out_ctx, nullptr);

    for (const auto& path : paths) {
        AVFormatContext* in_ctx = nullptr;
        avformat_open_input(&in_ctx, path.c_str(), nullptr,
nullptr);
        avformat_find_stream_info(in_ctx, nullptr);

        AVPacket* pkt = av_packet_alloc();
        while (av_read_frame(in_ctx, pkt) >= 0) {
            av_interleaved_write_frame(out_ctx, pkt);
            av_packet_unref(pkt);
        }
        av_packet_free(&pkt);
        avformat_close_input(&in_ctx);
    }

    av_write_trailer(out_ctx);
    avio_closep(&out_ctx->pb);
    avformat_free_context(out_ctx);
    return 0;
}

class ChunksStapler {
public:

```

```

        ChunksStapler(const KafkaConfig& kafka_config)
            : publisher_(kafka_config)
            , consumer_(kafka_config, "chunks-stapler-group",
{"video.glue.queue"})
        {

consumer_.register_handler("video.frames.glue.finish",
                        [this](const std::string& payload)
{
    handle_glue_finish(payload["task_id"],
payload["chunk_index"], payload["output_path"],
payload.value("expected_chunks", 1));
    });
}

void start() { consumer_.start(); }

private:
    void handle_glue_finish(const std::string& task_id, int
chunk_idx,
                        const std::string& path, int
expected) {
        std::lock_guard<std::mutex> lock(mutex_);
        auto& state = states_[task_id];
        state.paths[chunk_idx] = path;
        state.expected = expected;

        if (static_cast<int>(state.paths.size()) ==
state.expected) {
            json start = {"event_type",
"video.chunks.glue.start"}, {"task_id", task_id}};
            publisher_.publish("video.glue.queue", start);

            std::vector<std::string> ordered;
            for (int i = 0; i < state.expected; ++i)
ordered.push_back(state.paths[i]);
            std::string output = storage_root + "/" + task_id
+ "/assembled_video.mp4";
            concat_chunks(ordered, output);

            json finish = {"event_type",
"video.chunks.glue.finish"}, {"task_id",
task_id}, {"output_path", output}};
            publisher_.publish("video.glue.queue", finish);
            states_.erase(task_id);
        }
    }

    struct ChunkState { int expected = 0; std::map<int,
std::string> paths; };
    EventPublisher publisher_;
    EventConsumer consumer_;
    std::unordered_map<std::string, ChunkState> states_;

```

```

        std::mutex mutex_;
};

```

Файл «audio_stapler.cpp»

```

#include <pipeline_transport/event_publisher.h>
#include <pipeline_transport/event_consumer.h>
#include <pipeline_transport/kafka_config.h>

extern "C" {
#include <libavformat/avformat.h>
#include <libavcodec/avcodec.h>
#include <libavutil/avutil.h>
#include <libavutil/timestamp.h>
#include <libavutil/opt.h>
#include <libswresample/swresample.h>
}

#include <fstream>
#include <iostream>
#include <sstream>
#include <chrono>
#include <thread>
#include <atomic>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <filesystem>

namespace fs = std::filesystem;

static std::string storage_root = "/data";
static std::string temp_root = "/tmp/pipeline";
static std::atomic<bool> running{true};
static std::mutex mux_mutex;
static std::condition_variable mux_cv;
static std::queue<std::pair<std::string, std::string>>
failed_tasks;

static int mux_video_audio(const std::string& video_path,
const std::string& audio_path,
const std::string& output_path) {
    AVFormatContext* out_ctx = nullptr;
    int ret = avformat_alloc_output_context2(&out_ctx,
nullptr, nullptr, output_path.c_str());
    if (!out_ctx || ret < 0) {
        std::cerr << "[AudioStapler] Failed to allocate
output context: " << ret << std::endl;
        return -1;
    }

    AVFormatContext* v_ctx = nullptr;

```

```

        ret = avformat_open_input(&v_ctx, video_path.c_str(),
nullptr, nullptr);
        if (ret < 0) {
            std::cerr << "[AudioStapler] Failed to open video
file: " << video_path << std::endl;
            avformat_free_context(out_ctx);
            return -1;
        }
        avformat_find_stream_info(v_ctx, nullptr);

        int video_stream_index = -1;
        for (unsigned i = 0; i < v_ctx->nb_streams; ++i) {
            AVStream* os = avformat_new_stream(out_ctx, nullptr);
            if (!os) continue;
            avcodec_parameters_copy(os->codecpar, v_ctx-
>streams[i]->codecpar);
            os->time_base = v_ctx->streams[i]->time_base;
            if (v_ctx->streams[i]->codecpar->codec_type ==
AVMEDIA_TYPE_VIDEO) {
                video_stream_index = i;
            }
        }
        avformat_close_input(&v_ctx);

        int audio_stream_index = -1;
        if (!audio_path.empty()) {
            AVFormatContext* a_ctx = nullptr;
            ret = avformat_open_input(&a_ctx,
audio_path.c_str(), nullptr, nullptr);
            if (ret >= 0) {
                avformat_find_stream_info(a_ctx, nullptr);
                for (unsigned i = 0; i < a_ctx->nb_streams; ++i)
                {
                    if (a_ctx->streams[i]->codecpar->codec_type
== AVMEDIA_TYPE_AUDIO) {
                        AVStream* os =
avformat_new_stream(out_ctx, nullptr);
                        if (os) {
                            avcodec_parameters_copy(os-
>codecpar, a_ctx->streams[i]->codecpar);
                            os->time_base = a_ctx->streams[i]-
>time_base;
                            audio_stream_index = i;
                        }
                    }
                }
                avformat_close_input(&a_ctx);
            } else {
                std::cerr << "[AudioStapler] Failed to open audio
file: " << audio_path << std::endl;
            }
        }
    }

```

```

        ret = avio_open(&out_ctx->pb, output_path.c_str(),
AVIO_FLAG_WRITE);
        if (ret < 0) {
            std::cerr << "[AudioStapler] Failed to open output
file: " << output_path << std::endl;
            avformat_free_context(out_ctx);
            return -1;
        }

        AVDictionary* opts = nullptr;
        av_dict_set(&opts, "movflags", "faststart", 0);
        ret = avformat_write_header(out_ctx, &opts);
        av_dict_free(&opts);
        if (ret < 0) {
            std::cerr << "[AudioStapler] Failed to write header:
" << ret << std::endl;
            avio_closep(&out_ctx->pb);
            avformat_free_context(out_ctx);
            return -1;
        }
        int64_t last_dts = AV_NOPTS_VALUE;
        int64_t last_pts = AV_NOPTS_VALUE;
        int packet_count = 0;

        while (av_read_frame(src, pkt) >= 0 && running) {
            AVStream* in_stream = src->streams[pkt-
>stream_index];
            AVStream* out_stream = out_ctx->streams[pkt-
>stream_index];

            pkt->pts = av_rescale_q_rnd(pkt->pts, in_stream-
>time_base, out_stream->time_base,

static_cast<AVRounding>(AV_ROUND_NEAR_INF |
AV_ROUND_PASS_MINMAX));
            pkt->dts = av_rescale_q_rnd(pkt->dts, in_stream-
>time_base, out_stream->time_base,

static_cast<AVRounding>(AV_ROUND_NEAR_INF |
AV_ROUND_PASS_MINMAX));
            pkt->duration = av_rescale_q(pkt->duration,
in_stream->time_base, out_stream->time_base);
            pkt->pos = -1;

            if (pkt->dts != AV_NOPTS_VALUE) {
                if (last_dts != AV_NOPTS_VALUE && pkt->dts <=
last_dts) {
                    pkt->dts = last_dts + 1;
                }
                last_dts = pkt->dts;
            }
            if (pkt->pts != AV_NOPTS_VALUE) {

```

```

        if (last_pts != AV_NOPTS_VALUE && pkt->pts <=
last_pts) {
            pkt->pts = last_pts + 1;
        }
        last_pts = pkt->pts;
    }

    ret = av_interleaved_write_frame(out_ctx, pkt);
    if (ret < 0) {
        std::cerr << "[AudioStapler] Failed to write
packet " << packet_count << ": " << ret << std::endl;
    }
    packet_count++;
    av_packet_unref(pkt);
}

    std::cout << "[AudioStapler] Processed " << packet_count
<< " packets" << std::endl;

    av_packet_free(&pkt);
    avformat_close_input(&src);

    if (audio_path.empty()) {
        std::cout << "[AudioStapler] No audio stream, copying
video only" << std::endl;
    }

    av_write_trailer(out_ctx);
    avio_closep(&out_ctx->pb);
    avformat_free_context(out_ctx);

    return 0;
}

class AudioStapler {
public:
    AudioStapler(const KafkaConfig& kafka_config)
        : publisher_(kafka_config)
        , consumer_(kafka_config, "audio-stapler-group",
{"video.glue.queue", "video.recover.queue"})
        , stapler_thread_(&AudioStapler::stapler_loop, this)
        , retry_thread_(process_failed_queue)
        , heartbeat_thread_(&AudioStapler::heartbeat_loop,
this)
    {

        consumer_.register_handler("video.chunks.glue.finish",
            [this](const std::string& key, const json&
payload) {
                handle_video_ready(payload["task_id"],
payload["output_path"]);
            });
        consumer_.register_handler("video.recover.request",

```

```

        [this](const std::string& key, const json&
payload) {
            handle_recover_request(payload["task_id"],
payload["video_path"], payload["audio_path"]);
        });
        consumer_.register_handler("pipeline.shutdown",
        [this](const std::string&, const json&) {
            shutdown();
        });
        consumer_.register_handler("pipeline.health.check",
        [this](const std::string&, const json& payload)
{
            handle_health_check(payload["request_id"]);
        });
    }

    ~AudioStapler() {
        shutdown();
        if (stapler_thread_.joinable())
stapler_thread_.join();
        if (retry_thread_.joinable()) retry_thread_.join();
        if (heartbeat_thread_.joinable())
heartbeat_thread_.join();
    }

    void start() {
        consumer_.start();
        ensure_directory(storage_root);
        ensure_directory(temp_root);
        std::cout << "[AudioStapler] Service started" <<
std::endl;
    }

private:
    void handle_video_ready(const std::string& task_id, const
std::string& video_path) {
        std::lock_guard<std::mutex> lock(task_mutex_);

        json start = {"event_type",
"video.audio.glue.start"}, {"task_id", task_id}};
        publisher_.publish("video.glue.queue", start);

        if (!validate_video_file(video_path)) {
            std::cerr << "[AudioStapler] Invalid video file:
" << video_path << std::endl;
            json error = {
                {"event_type", "video.audio.glue.error"},
                {"task_id", task_id},
                {"error", "invalid_video_file"},
                {"video_path", video_path}
            };
            publisher_.publish("video.glue.queue", error);
            return;
        }
    }

```

```

    }

    std::string audio_path = storage_root + "/" + task_id
+ "/audio/stream_0.aac";
    std::string output = storage_root + "/" + task_id +
"/final_output.mp4";

    ensure_directory(storage_root + "/" + task_id);

    std::ifstream test(audio_path);
    bool audio_present = test.good() &&
validate_audio_file(audio_path);
    test.close();

    int retry_count = 0;
    int max_retries = 3;
    int result = -1;

    while (retry_count < max_retries && result != 0) {
        result = mux_video_audio(video_path,
audio_present ? audio_path : "", output);
        if (result != 0) {
            retry_count++;
            std::cerr << "[AudioStapler] Mux attempt " <<
retry_count << " failed for task: " << task_id << std::endl;

            std::this_thread::sleep_for(std::chrono::milliseconds(500 *
retry_count));
        }
    }

    if (result != 0) {
        std::cerr << "[AudioStapler] All mux attempts
failed for task: " << task_id << std::endl;
        std::lock_guard<std::mutex>
queue_lock(mux_mutex);
        failed_tasks.push({video_path, audio_present ?
audio_path : ""});
        mux_cv.notify_one();

        json error = {
            {"event_type", "video.audio.glue.error"},
            {"task_id", task_id},
            {"error", "mux_failed_after_retries"},
            {"retry_count", retry_count}
        };
        publisher_.publish("video.glue.queue", error);
        return;
    }

    active_tasks_.insert(task_id);

    json finish = {

```

```

        {"event_type", "video.audio.glue.finish"},
        {"task_id", task_id},
        {"output_path", output},
        {"audio_present", audio_present},
        {"timestamp",
std::chrono::system_clock::now().time_since_epoch().count()}
    };
    publisher_.publish("video.glue.queue", finish);

    active_tasks_.erase(task_id);

    std::cout << "[AudioStapler] Task " << task_id << "
completed successfully" << std::endl;
}

extern "C" int main(int argc, char** argv) {
    std::string config_path = "/etc/pipeline/kafka.json";
    if (argc > 1) {
        config_path = argv[1];
    }

    KafkaConfig config;
    if (!config.load_from_file(config_path)) {
        std::cerr << "[AudioStapler] Failed to load config
from: " << config_path << std::endl;
        return 1;
    }

    avformat_network_init();

    AudioStapler stapler(config);
    stapler.start();

    std::cout << "[AudioStapler] Running. Press Ctrl+C to
stop." << std::endl;

    while (running) {
std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    avformat_network_deinit();
    return 0;
}

```

ПРИЛОЖЕНИЕ Б

(обязательное)

Веб-приложение улучшения качества видео

Презентация

Конвейер обработки видеофайлов для веб- приложения

Автор работы:

студент гр. 24ВВИм1 Лукин Всеволод Дмитриевич

Научный руководитель:

д.т.н., профессор Зинкин Сергей Александрович

Пенза, 2026

Актуальность

В условиях зависимости от иностранных облачных платформ, проблема доступа к средствам автоматического улучшения качества пользовательского видеоконтента является одной из ключевых в области.

Смежные проблемы средств для улучшения качества:

- проприетарность программного обеспечения;
- лицензионные ограничения;
- ограниченное количество открытых решений;
- архитектурная несостоятельность.

Результаты исследования и программная реализация, реализованные в ВКР, призваны решить описанные выше проблемы.

Цель работы

Цель работы: исследование, проектирование, реализация и оценка эффективности микросервисной отказоустойчивой архитектуры событийно-ориентированного конвейера.

Объектами исследования являются процессы передачи, хранения и обработки видеофайлов.

Предметы исследования:

- архитектура конвейера обработки видеофайлов;
- нейросетевые алгоритмы улучшения качества изображения.

Анализ существующих решений

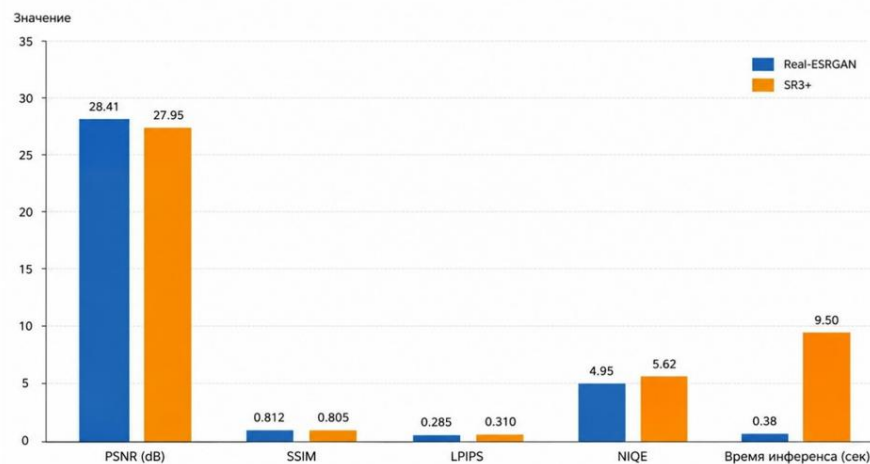
Критерий оценки	Ribbon (Netflix)	Rasbora (OpenSeaWave)	Amazon Elastic Transcoder (AWS)
Возможность модификации кодовой базы	Невозможна, проприетарное ПО	Возможна, открытый исходный код	Невозможна, проприетарное ПО
Возможность доступа	Недоступен	Доступен	Недоступен
Инфраструктурная независимость	Частичная привязка к AWS	Частичная привязка к Google Docs	Полная привязка к AWS
Возможность изменения используемых моделей	Присутствует, ограничена пулом моделей Netflix	Присутствует, не ограничена	Отсутствует
Поддержка кодовой базы	Поддерживается	Не поддерживается с 2020 года	Поддерживается

Исследование механизмов улучшения качества изображения

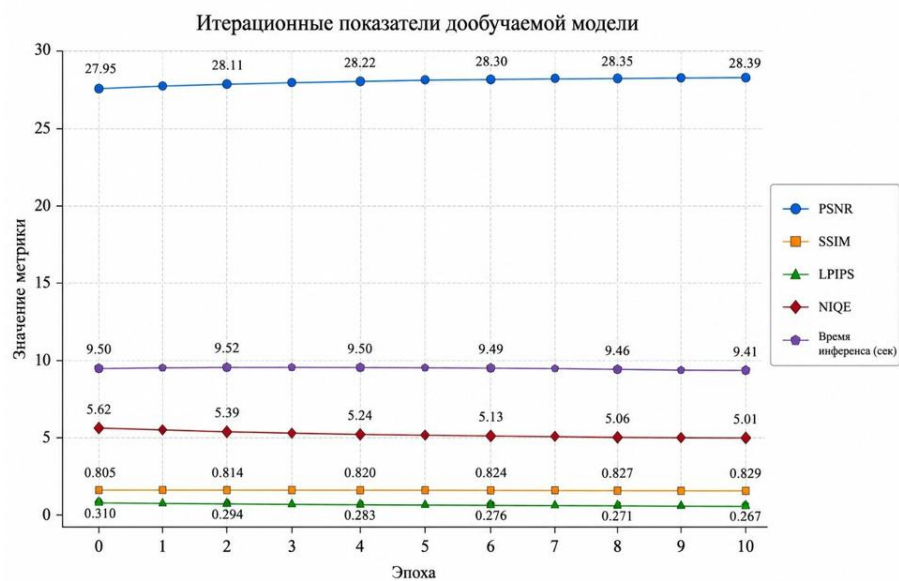
Тип деградации	Алгоритмический метод	Недостатки	Нейронная сеть	Недостатки
Шум матрицы	Временное усреднение с детектором движения	Зависит от накопленного сигнала, уязвим к ошибкам детектора	Real-ESRGAN	Возможно чрезмерное сглаживание текстур при некорректном обучении
Смазывание движения	Фильтр Винера	Требует точного знания ядра смазывания и минимального шума	Real-ESRGAN	Возможны артефакты на сложных траекториях движения
Блочность при компрессии	Петлевой фильтр кодека, адаптивная фильтрация	Только сглаживает границы	Real-ESRGAN	Устранение артефактов с синтезом утраченных текстур
Потеря текстур при компрессии	Статистическая экстраполяция в частотной области	Восстанавливает только регулярные предсказуемые текстуры	Real-ESRGAN	Генерация текстур произвольной структуры
Низкое разрешение	Фильтр Ланцоша, бикубическая интерполяция	Игнорирует тени и освещение	Real-ESRGAN	Слепое восстановление с синтезом деталей
Низкая частота кадров	Вариационная интерполяция на основе оптического потока	Высокая вычислительная сложность	CVEGAN, EDVR	Межкадровое выравнивание с сохранением временной синхронизации

Исследование архитектур нейронных сетей

Сравнение качества моделей суперразрешения



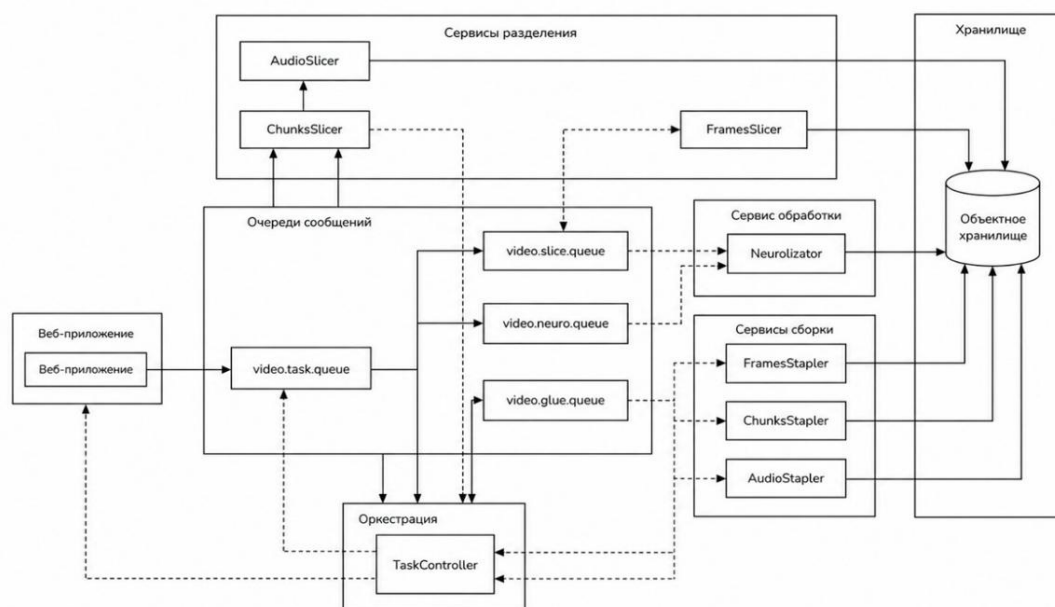
Исследование результатов дообучения



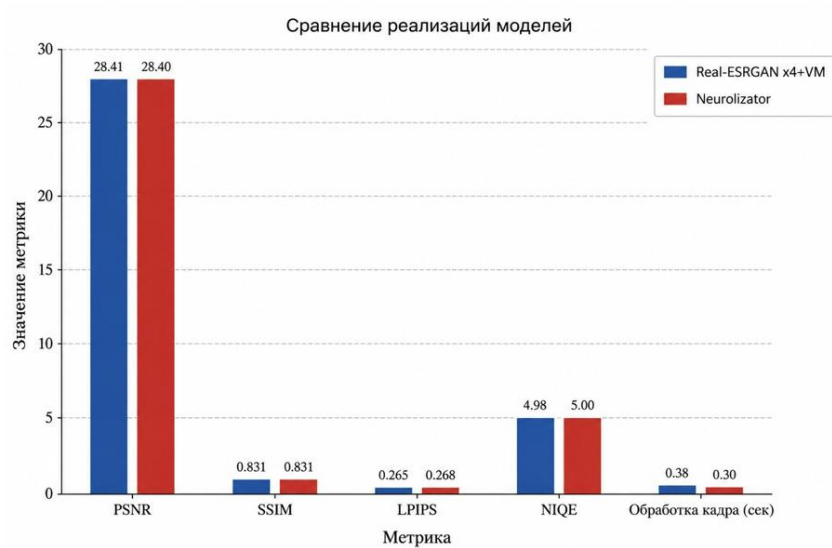
Исследование результатов дообучения

Эпоха	PSNR	SSIM	LPIPS	NIQE	Время инференса (сек)
0	27.95	0.805	0.310	5.62	9.50
3	28.17	0.817	0.288	5.31	9.51
6	28.30	0.824	0.276	5.13	9.49
10	28.39	0.829	0.267	5.01	9.41

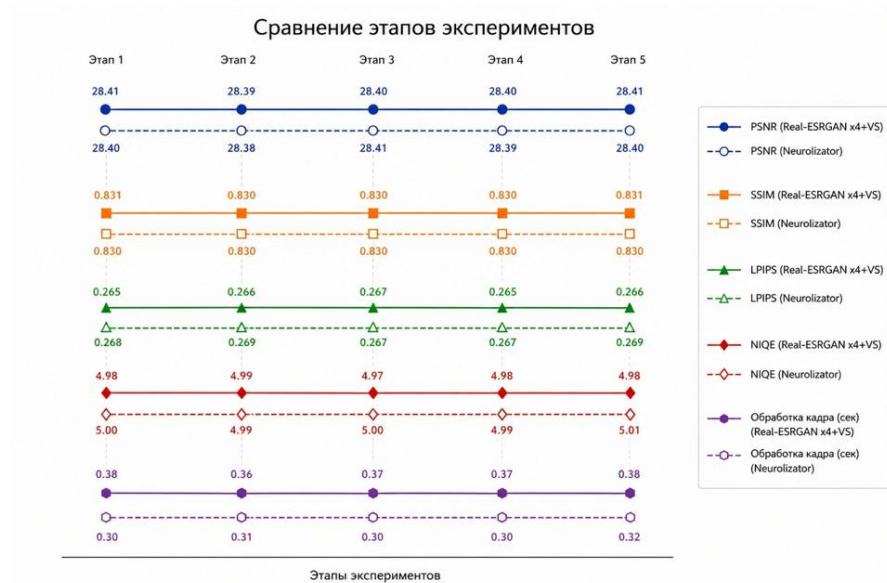
Архитектура конвейера обработки видеофайлов



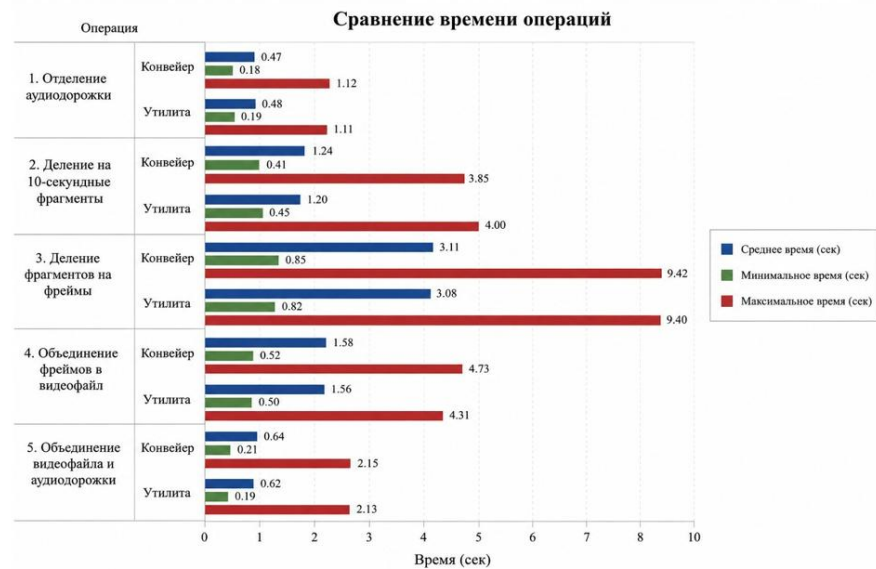
Оценка эффективности конвейера



Оценка эффективности конвейера



Оценка эффективности конвейера



Результаты работы

В процессе выполнения научной работы была исследована, спроектирована, реализована и проверена архитектура отказоустойчивого событийно-ориентированного конвейера обработки видеофайлов, обеспечивающая полный цикл улучшения качества пользовательского видеоконтента.

Дообученная модель Real-ESRGAN x4plus превзошла оригинал. LPIPS снижен на 14.5%, NIQE — на 11.4%, метрики PSNR и SSIM сохранены.

Конвейер обеспечил сокращение времени обработки кадра на 20.1% относительно оригинального инференса без деградации качества.

Система подтвердила отказоустойчивость. При оценке эффективности на нагруженных сценариях отсутствовали потерянные задачи.

Все функциональные и нефункциональные требования выполнены, цель работы достигнута. Система готова к развёртыванию в высоконагруженных средах.

Спасибо за внимание!