

Факультет «Информационные технологии»

Кафедра «Прикладная математика»

**КУРСОВАЯ РАБОТА**

по дисциплине «Математическое моделирование и анализ данных»

***( по дисциплине «…», по модулю «….» или научно-исследовательская работа)***

на тему: «Имитация живого видео-ответа современными генеративными моделями»

Направление подготовки  02.03.03 Математическое обеспечение и администрированиеинформационных систем

(код и наименование)

Направленность программы   Информационные системы и базы данных

(наименование)

Группа 21ИТ-МО(б/о)ИСБД-1

Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ А. В. Царюк

(подпись) (дата)

Объем заимствованного текста не превышает 20 %.

Допустить к защите \_\_\_\_\_\_\_\_\_ И. С. Колотовкин

(подпись)

Руководитель

курсовой работы \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ И. С. Колотовкин

(подпись) (дата)

Москва, 2024

[ВВЕДЕНИЕ 3](#_gjdgxs)

[Цели и задачи 3](#_30j0zll)

[Глава 1. Проектирование 4](#_1fob9te)

[1.1 Формирование требований 4](#_3znysh7)

[1.2 Поиск решений 5](#_ql3xxb17yyrz)

[1.3 Техническое описание 6](#_tyjcwt)

[1.3.1 Архитектура 6](#_3dy6vkm)

[1.3.2 Протокол 7](#_1t3h5sf)

[1.3.3 Эксплуатация 8](#_4d34og8)

[1.4 Команда 9](#_sr9v7numelrv)

[Глава 2. Конструирование 10](#_2s8eyo1)

[2.1 Инструменты разработчика 10](#_17dp8vu)

[2.1.1 GIT 10](#_3rdcrjn)

[2.1.2 Прикладной уровень 11](#_26in1rg)

[2.1.3 Аппаратура 12](#_lnxbz9)

[2.1.4 Контейнеризация 13](#_35nkun2)

[2.2 Разработка 14](#_1ksv4uv)

[2.2.1 API 14](#_2jxsxqh)

[2.2.2 Dockerfile 15](#_44sinio)

[2.2.3 Вспомогательные функции 16](#_z337ya)

[2.3 Тестирование 17](#_3j2qqm3)

[2.4 Развертывание 18](#_1y810tw)

[Заключение 19](#_4i7ojhp)

[Список литературы 19](#_2xcytpi)

# **ВВЕДЕНИЕ**

В настоящее время наблюдается стремительное развитие технологий искусственного интеллекта (ИИ), включая генеративные модели. Одной из актуальных задач является создание моделей, способных имитировать живой видео ответ. Это может найти применение в различных областях, таких как образование, медицина, развлечения и другие. В данной работе будет рассмотрена возможность использования современных генеративных моделей для создания реалистичных видео ответов, которые могут быть использованы в различных сферах деятельности.

# **Цели и задачи**

**Цель работы**

реализовать современный инструмент для имитации живого видео-ответа

**Задачи**

1. **Поиск последних версий открытого программного обеспечения для генерации**: в этом этапе мы будем искать и анализировать последние версии программного обеспечения, которое может быть использовано для генерации текста, голоса, синхронизации губ и улучшения качества видео. Мы также должны рассмотреть различные варианты и сравнить их производительности и функциональности.
2. **Создание модуля с цепным вызовом инструментов генерации материалов**: после того, как мы найдем подходящее программное обеспечение, мы создадим модуль, который будет использовать эти инструменты для генерации различных элементов видео. Этот модуль будет организовывать последовательность вызовов этих инструментов таким образом, чтобы создать реалистичный видео ответ.
3. **Разработка фронтального компонента для взаимодействия с модулем генерации**: Этот компонент будет предоставлять пользовательский интерфейс для взаимодействия с модулем генерации. Пользователи смогут вводить текст, выбирать параметры генерации и просматривать результаты.
4. **Договоренность о протоколе обмена данными и синхронизация компонентов системы**: для эффективной работы инструмента необходимо согласовать протокол обмена данными между различными компонентами системы. Это включает в себя синхронизацию данных между модулем генерации и фронтальным компонентом, а также координацию работы различных инструментов генерации.
5. **Тестирование инструмента и получение пары интересных примеров**: после реализации инструмента мы проведем тестирование, чтобы убедиться, что он работает корректно и создает реалистичные видео ответы. Кроме того, мы продемонстрируем несколько интересных примеров использования инструмента.

# **Глава 1. Проектирование**

## **1.1 Формирование требований**

**Операционная система:** важно, чтобы инструмент был совместим с различными операционными системами, чтобы пользователи могли использовать его на своих устройствах без проблем. Поддержка: mac os x, windows 10 или выше, \*nix

**Язык программирования:** выбор современного и популярного языка программирования облегчит разработку, поддержку и расширение инструмента.

**Открытое ПО:** использование открытого программного обеспечения позволит улучшить гибкость и масштабируемость инструмента.

**Модульность:** модульная архитектура упростит добавление новых функций и интеграцию с другими системами, что делает инструмент более гибким и расширяемым.

**UI/UX:** простой и интуитивно понятный пользовательский интерфейс улучшит удобство использования инструмента, делая его доступным для широкого круга пользователей.

**Протокол обмена данными:** стандартизированный протокол обмена данными обеспечит надежную передачу информации между различными компонентами системы.

**Синхронизация:** точная синхронизация между генерацией текста, голоса и движений губ является ключевым аспектом для создания реалистичного видео-ответа.

**Качество видео:** высокое качество видео важно для обеспечения естественности и убедительности видео-ответа.

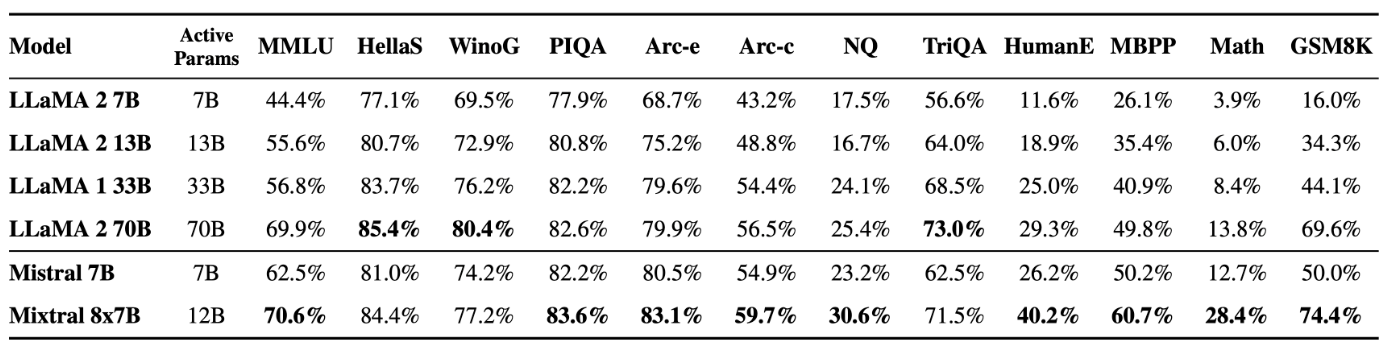
**Тестирование:** полное тестирование гарантирует, что инструмент будет работать корректно и без ошибок.

**Примеры использования:** примеры использования помогают пользователям понять возможности инструмента и стимулируют их к созданию собственных генераций.

## **1.2 Поиск решений**

**Генерация текстового ответа:** движок LLAMA[[1]](#footnote-0) поддерживает самое большое количество открытых языковых моделей. Языковая модель `airoboros-mistral2.2-7B-GGUF` показала себя с хорошей стороны, по соотношению размера модели и качеству ответов.

Подробнее: <https://mistral.ai/news/mixtral-of-experts/>

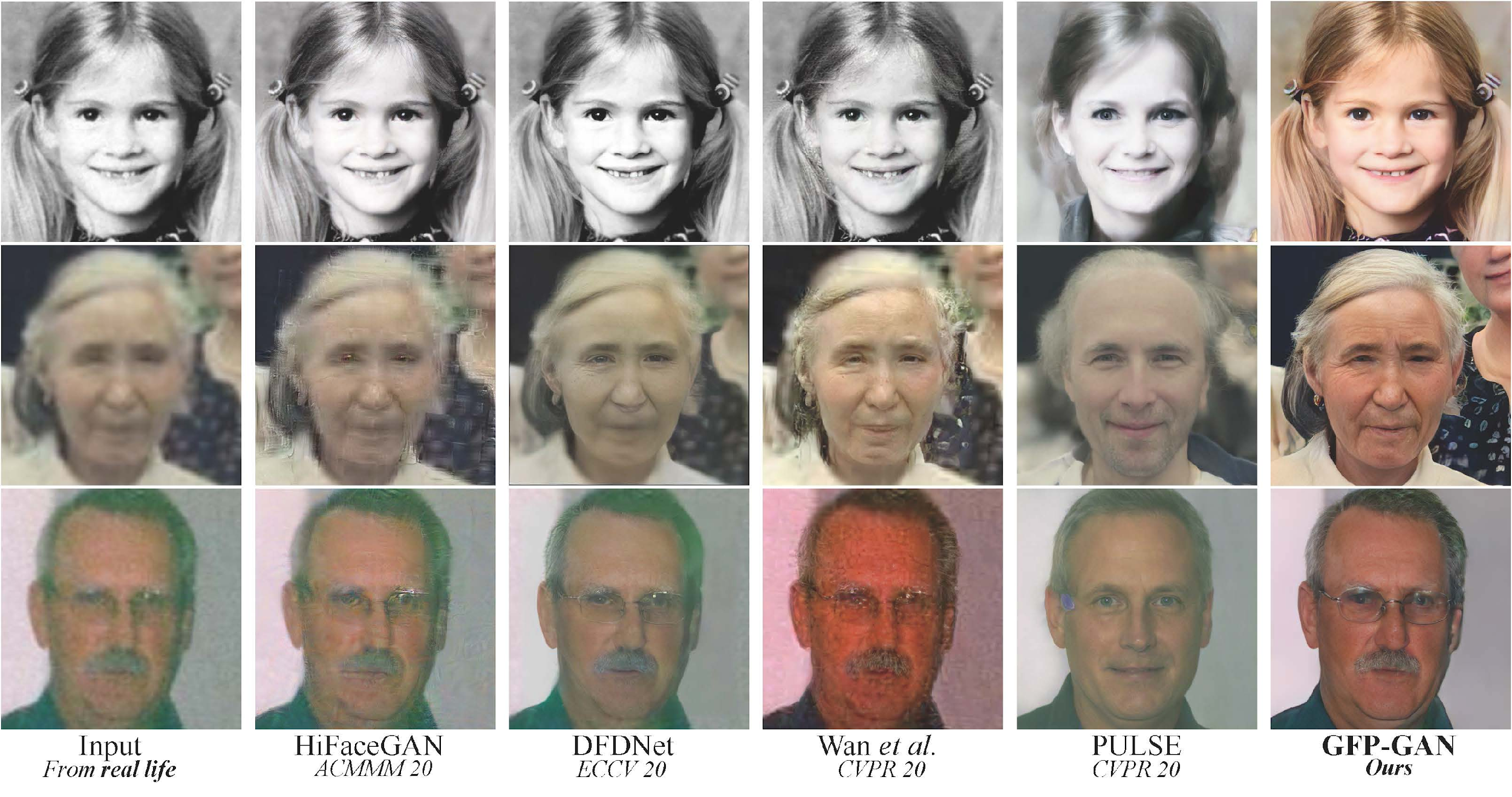


*Рис.1. Демонстрация преимуществ Mistral*

**Синтез речи:** HF-Coqui XTTS[[2]](#footnote-1), это одна из лучших моделей для синтеза речи. Она способна генерировать человекоподобный голос, который звучит естественно и плавно.

**Синхронизация губ:** Wav2Lip[[3]](#footnote-2), это передовая технология, которая может точно синхронизировать движения губ с речью. Это делает видео более реалистичным и убедительным.

**Улучшение качества:** GFP-GAN[[4]](#footnote-3), это мощный инструмент для улучшения качества видео. Он способен устранять шумы, повышать четкость и улучшать общее качество видео.



*Рис.2. Иллюстрация результатов разных моделей*

## **1.3 Техническое описание**

### **1.3.1 Архитектура**

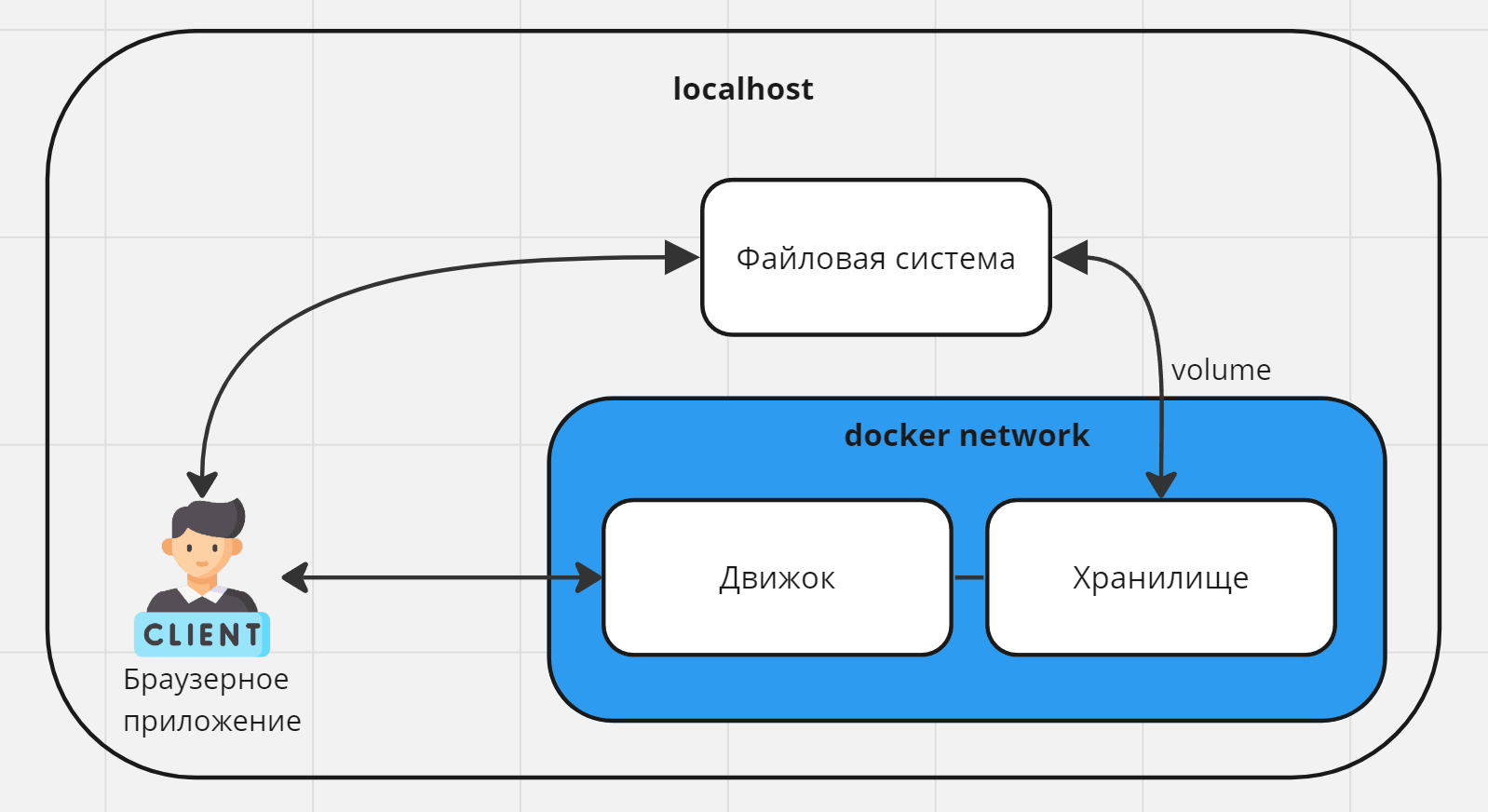
Для начала опишем отдельные функции: создание личности, хранение личности, генерация текстового ответа по контексту личности, синтез речи на основе текста и заложенного в личность голоса, синхронизация видеопотока с синтезированной личностью, отображение выходного видео пользователю. Теперь мы можем разбить эти функции на разные компоненты.

Хранение личности (голос, видео), а также выходных файлов, на себя может взять как файловая система, S3[[5]](#footnote-4), удаленный сервер, так и оперативная память. Так как, доступ к ресурсам должен быть из разных компонентов системы, а также хранилище должно быть бесперебойным и отвечать требованию по работе на разных операционных системах, оптимальным вариантом является BLOB[[6]](#footnote-5) хранилище (например MinIO[[7]](#footnote-6)).

Отвечать за генерацию текста, синтез речи и синхронизацию губ должен отвечать монолитный компонент - движок, так как от сервиса требуется максимальная скорость ответа, а передавать промежуточный результат из сервиса в сервис с задержками - недопустимо.

Процесс создания личности состоит из нескольких этапов, в данном случае, нас интересует первый - получение данных от источника (пользователя). В идеальном сценарии, клиенту не нужно переходить из программы в программу, однако для проверки минимально работающей системы можно допустить, что пользователь самостоятельно размещает данные в BLOB хранилище через браузер на соответствующем ресурсе. И затем обращается к движку и отображает видео пользователю.

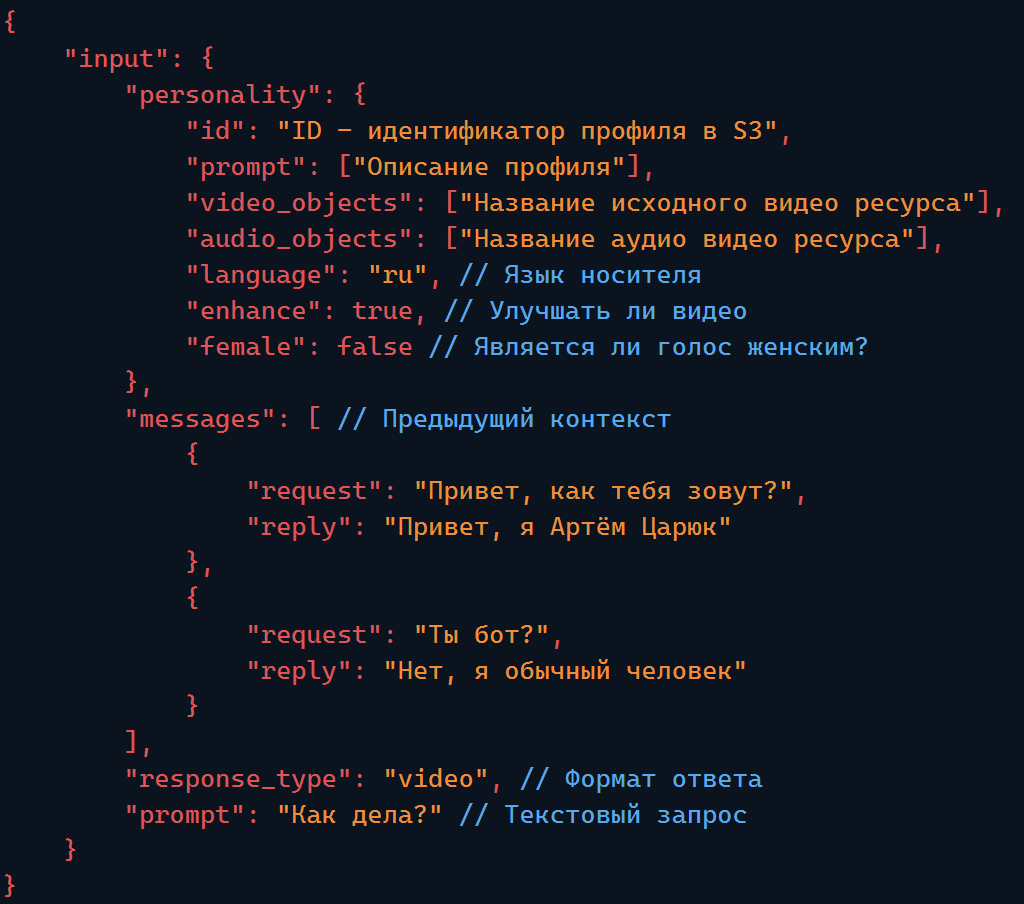
Так как ПО предназначено для одиночного пользования, для упрощения, можно расположить всю систему в рамках локальной или сублокальных сетях. Так же можно использовать контейнеризацию Docker[[8]](#footnote-7) для изоляции процессов от других запущенных приложений.



*Рис.3. Архитектура*

### **1.3.2 Протокол**

Описывается протокол взаимодействия клиентской и серверной частей. Общение происходит в локальной сети, поэтому предпочтительно использовать HTTP[[9]](#footnote-8) протокол. Поскольку сервер и клиент являются одной системой, не требуются подписи, сертификаты и шифрование, то есть TLS[[10]](#footnote-9) не нужен. Синхронность HTTP не вызывает проблем, так как обе части поддерживают параллелизм[[11]](#footnote-10). Политика CORS[[12]](#footnote-11) сервера разрешает все входящие запросы от любых отправителей для удобства работы в локальной сети. Все запросы и ответы представлены в формате JSON[[13]](#footnote-12). Далее следует детальное описание ручек:

1. /check\_gpu\_driver
   1. Тип запроса: **GET**
   2. Описание: проверка поддержки совместимости ПО и аппаратного обеспечения
   3. Тело ответа: поле `message: str`, является информативным сообщением для пользователя о результате проверки совместимости. Поле `supported: bool` в свою очередь, отображает фактический результат
2. /send\_request
   1. Тип запроса: **POST**
   2. Описание: синхронный запрос на генерацию видео ответа, результатом является ссылка на готовый видео ресурс
   3. Пример тела запроса:   
      
   4. В ответе приходит вспомогательная информация и поле `object\_url: str`, в котором хранится ссылка на результат запроса - видео ответ
3. /request\_status
   1. Тип запроса: **GET**
   2. Описание: пока запрос на генерацию видео висит в обработке и блокирует сессию, пользователю нужно отображать статус обработки, для этого существует эта ручка
   3. Тело ответа: поле `stage: str` это текст с информацией о статусе заявки, поле `last\_update: long` это время с последним изменением статуса обработки
4. /list\_files
   1. Тип запроса: **GET**
   2. Описание: на главной странице можно выбрать создание профиля с нуля или же выбор из доступных. Для получения доступных профилей сервер ходит в хранилище и возвращает список всех файлов
   3. Тело ответа: массив абсолютных путей к файлам профилей

### **1.3.3 Эксплуатация**

Из требований к открытости ПО, поддержки разных операционных систем вытекает общепринятый стандарт использования контейнеризации Docker, к такому же выводу мы приходили в части архитектуры. Как должно выглядеть использование контейнеризации в рамках этого ПО?

Всего наша система состоит из трех компонентов: хранилище, сервер, клиент. Если обратиться к концепции DDD[[14]](#footnote-13), то станет ясно что хранилище это generic, то есть сложное в разработке и общее решение, которое лучше взять из публичных репозиториев со свободной лицензией. Так и сделаем. Публичный образ Minio уже существует, не смысла создавать свой. А вот в случае клиента и сервера нам это придется сделать. Поэтому следует установить программу Docker Desktop[[15]](#footnote-14) для управления контейнерами и образами.

Также следует создать заготовку файла `docker-compose.yaml` чтобы одной командой запускать все три сервиса, настроить между ними сеть и протянуть параметры.

1. Настройка общей сети с названием `bridge`

networks:

bridge:

driver: bridge

1. Настройка сервиса клиентской части, заранее подберем название образа и порт

frontend:

image: funcid/oppenganger-frontend:latest

restart: always

ports:

- '3000:3000'

networks:

- bridge

1. Настройка Minio по публичному образу, смонтируем локальную папку в контейнер

minio:

user: root

image: bitnami/minio:latest

volumes:

- /path/to/minio-persistence:/bitnami/minio/data

ports:

- '9000:9000'

- '9001:9001'

environment:

BITNAMI\_DEBUG: true

MINIO\_ROOT\_USER: root

MINIO\_ROOT\_PASSWORD: password0

MINIO\_DEFAULT\_BUCKETS: videos-bucket,personalities-bucket

MINIO\_API\_CORS\_ALLOW\_ORIGIN: '\*'

restart: unless-stopped

networks:

- bridge

1. Настройка серверной части, так же заранее выберем название образа и порт, не забудем прокинуть значения для подключению к хранилищу, а также прокинем в контейнер права к драйверам Nvidia[[16]](#footnote-15) для работы CUDA[[17]](#footnote-16) ядер и ffmpeg[[18]](#footnote-17)

engine:

image: funcid/oppelganger:latest

restart: unless-stopped

ports:

- '8080:8080'

environment:

S3\_ENDPOINT\_URL: <http://minio:9000>

S3\_BUCKET\_VIDEOS: videos-bucket

S3\_BUCKET\_PERSONALITIES: personalities-bucket

S3\_REGION\_NAME: us-east-1

S3\_ACCESS\_KEY\_ID: root

S3\_SECRET\_ACCESS\_KEY: password0

networks:

- bridge

deploy:

resources:

reservations:

devices:

- driver: nvidia

count: 1

capabilities: [gpu, video]

## **1.4 Команда**

Работа выполняется двумя участниками, поэтому для продуктивной работы нужно выделить сильные стороны для правильного разграничения ответственности и обязанностей при разработке решения.

1. Царюк Артем Владимирович

Backend-разработка, devops[[19]](#footnote-18), финансовые технологии, системный анализ

1. Приходько Александр Вячеславович

Frontend-разработка, верстка, QA[[20]](#footnote-19), отладка, react js[[21]](#footnote-20), typescript, webpack[[22]](#footnote-21)

Участники обладают почти непересекающимися навыками что облегчает разграничение зоны ответственности. Серверная часть, системный анализ, инфраструктура - участник 1. Клиентская сторона, ручное тестирование - участник 2. Такие инструменты как Git, демонстрация экрана, сервисы совместной разработки помогут нам синхронизировать наши результаты.

# **Глава 2. Конструирование**

## **2.1 Инструменты разработчика**

### **2.1.1 GIT**

Система контроля версий Git[[23]](#footnote-22) позволяет разработчикам эффективно работать над кодом совместно. Вот несколько причин, почему стоит использовать Git в нашем случае:

1. Работа в команде. В Git есть функционал для работы нескольких разработчиков над одним проектом одновременно. Также система сохраняет все версии файлов, поэтому если кто-то из коллег внесет правки, которые нарушат работу проекта, их можно будет откатить до предыдущей версии.
2. Удобство при работе над ошибками. Если в коде возникла ошибка, то её можно быстро найти и исправить благодаря истории изменений.
3. Сохранение прогресса. Разработчику не нужно каждый раз сохранять файлы, чтобы они остались в системе. Все изменения автоматически фиксируются и доступны для просмотра.

Выберем конкретные реализации Git:

1. GitHub — крупнейший веб-сервис для хостинга IT-проектов и совместной разработки. Сервис поддерживает интеграции с другими сервисами, такими как Jira[[24]](#footnote-23), Jenkins[[25]](#footnote-24), Docker и другими. У сервиса огромная поддержка и сообщество. GitHub имеет большое и активное сообщество разработчиков, которые могут помочь в решении проблем и ответах на вопросы.
2. GitVerse — российский Git веб-сервис от Сбер, именно здесь будут зеркала репозиториев с GitHub на случай блокировки в России. Сервис поддерживает базовый функционал, однако сильно уступает конкуренту на момент написания работы.

Любой проект начинается с создания папки, сделаем организацию на GitHub с названием Oppelganger, в которую будем складывать репозитории. Всего три репозитория: oppelganger-frontend - клиентская часть; oppelganger - серверная часть; oppelganger-devops - различные конфигурации для запуска ПО. Так же не забываем создать зеркала на русском сервисе. Для удобства выполнения процедур GIT, будем использовать клиент GitHub Desktop для создания/отправки коммитов, создания и слияния веток.

### **2.1.2 Прикладной уровень**

**Язык программирования:** python — высокоуровневый язык программирования, который используется для создания приложений различной сложности. Отличительной чертой языка является автоматическое управление памятью (автоматическая сборка мусора), динамическая типизация и полная читаемость кода. Python имеет простой и понятный синтаксис, что делает его идеальным выбором для начинающих программистов. Он поддерживает структурное, объектно-ориентированное и функциональное программирование. Python очень популярен среди разработчиков машинного обучения. Это связано с тем, что многие библиотеки машинного обучения написаны именно на нём. Среди них можно выделить TensorFlow, Keras, PyTorch и другие. Они предоставляют разработчикам широкий набор инструментов для создания, обучения и развертывания моделей машинного обучения. В нашем случае это качество - необходимость. Так же этот язык соответствует требованиям из главы 1.1.

**Среда разработки сервера:** PyCharm - это интегрированная среда разработки (IDE), которая предоставляет множество функций и инструментов, облегчающих процесс разработки на Python. Вот несколько причин, почему выбор пал на PyCharm для разработки:

1. Поддержка Python: PyCharm полностью поддерживает Python, включая последние версии, и обеспечивает полную интеграцию с экосистемой Python, включая популярные библиотеки и фреймворки.
2. Отладчик: встроенный отладчик позволяет легко находить и исправлять ошибки в коде. Можно установить точки останова, шаг за шагом выполнять код и просматривать значения переменных в реальном времени.
3. Автодополнение кода: IntelliSense помогает ускорить написание кода, предлагая варианты завершения кода на основе контекста.
4. Рефакторинг: PyCharm предлагает мощные инструменты рефакторинга, которые позволяют изменять структуру кода без потери информации.
5. Тестирование: интеграция с популярными инструментами тестирования, такими как unittest и pytest, упрощает процесс тестирования кода.
6. Визуальное проектирование баз данных: если проект включает работу с базами данных, PyCharm предлагает удобный инструмент для визуального проектирования схем баз данных.
7. Оптимизация работы с кодом: PyCharm предлагает различные улучшения рабочего процесса, такие как умное форматирование кода, подсветка синтаксиса, автоматическое закрытие скобок и многое другое.
8. Расширяемость: PyCharm поддерживает плагины, которые позволяют расширять функциональность IDE в соответствии с потребностями.
9. Удобный интерфейс: PyCharm имеет интуитивно понятный и хорошо организованный интерфейс, который делает разработку комфортной и продуктивной.

**Среда разработки клиента:** Visual Studio Code — это мощный инструмент для фронтенд-разработчиков, который может значительно повысить эффективность и удобство работы над проектами.

1. Бесплатность и открытость исходного кода.

2. Большое количество расширений, которые позволяют добавлять новые функции и возможности к редактору.

3. Подсветка синтаксиса и автодополнение кода для большинства языков программирования, включая HTML, CSS и JavaScript.

4. Интеграция с Git для управления версиями кода.

5. Встроенный терминал для выполнения команд и отладки приложений.

6. Поддержка перетягивания файлов для их открытия в редакторе.

7. Возможность настройки интерфейса под свои нужды.

8. Встроенная поддержка таких технологий, как React, Angular и Vue.js.

9. Поддержка тем оформления, которые позволяют изменить внешний вид редактора.

**Встроенный помощник:** GigaCode — это генеративная языковая модель от Сбер, которая умеет отвечать на вопросы пользователей, вести диалог, придумывать рекламные посты и планы презентаций, сочинять стихи и сказки, составлять письма и сочинения. Она способна обрабатывать и структурировать большие объемы информации, писать код и создавать изображения по запросу. При помощи плагина в PyCharm и в Visual Studio Code, можно пользоваться услугами GigaCode напрямую из сред разработки.

### **2.1.3 Аппаратура**

Вычислительная мощность аппаратуры должна быть достаточна для работы в рамках больших языковых моделей, с параллельно запущенным клиентом и хранилищем. На данный момент, ни один существующий мобильный телефон не способен на это. Процессор компьютера должен поддерживать виртуализацию, она должна быть включена через BIOS. Так же, операционная система должна обладать компонентами для работы с виртуализацией, для Windows это Hyper V, а для Linux – KVM.

Массовые параллельные вычисления стали краеугольным камнем в мире машинного обучения. Монополистом на этом рынке является компания Nvidia с серверным GPU A100, однако средств на подобную аппаратуру у авторов работы нет. Однако есть другие графические ускорители с CUDA ядрами. Поэтому можно перейти к представлению текущих сборок для разработки и тестирования:

1. **Основная аппаратура**

Графический ускоритель Nvidia GeForce RTX 4080, 16 VRAM, 9728 CUDA ядер

Процессор AMD Ryzen 9 7900X, 12-Core, 5.4 GHz

Оперативная память 4x16 GB, 5600 MHz

SSD KINGSTON SKC3000D2048G

Операционная система Windows 11, Hyper V

1. **Вторичная аппаратура**

Графический ускоритель Nvidia GeForce GTX 1060, 6 VRAM, 1280 CUDA ядер

Процессор Intel Core i7-7700K CPU, 4-Core, 4.2 GHz

Оперативная память 2x16 GB, 2133 MHz

SSD KINGSTON SUV400S37240G

Операционная система Windows 10, Hyper V

Опытным путем оказалось, что обе аппаратуры выдерживают нагрузку, однако, ожидаемо и фактически первая аппаратура отработала быстрее.

### **2.1.4 Контейнеризация**

Виртуализация и контейнеризация — это два разных подхода к созданию изолированных сред для запуска приложений. Виртуализация создает виртуальную машину, которая имитирует работу физического компьютера. Контейнеризация же использует ядро операционной системы хоста для создания изолированной среды, в которой может работать приложение.

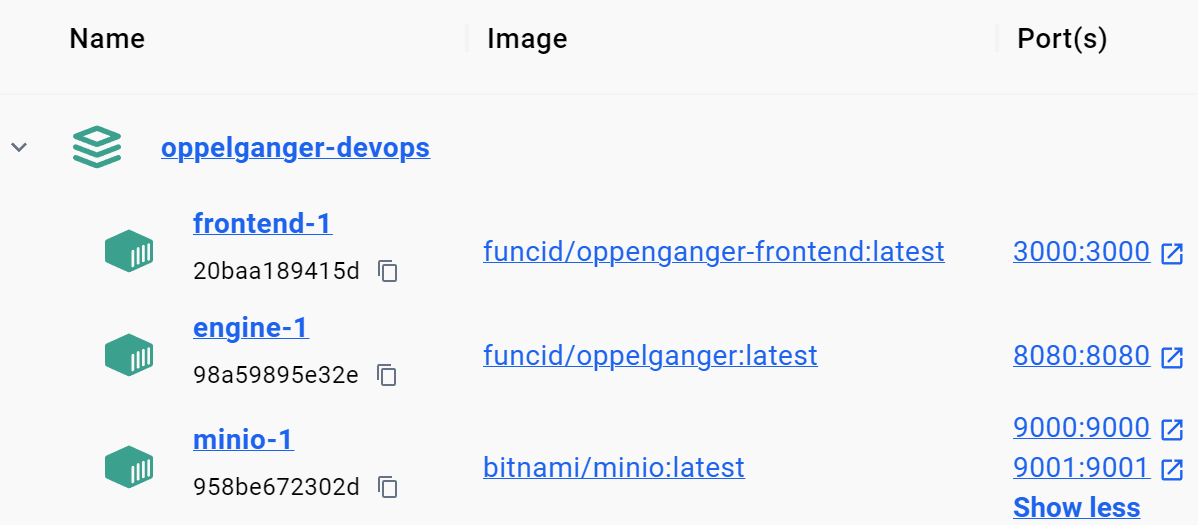
Hyper-V — это технология виртуализации от Microsoft, которая позволяет создавать виртуальные машины на Windows Server 2012 и выше. Каждая виртуальная машина имеет свою собственную операционную систему, ресурсы и настройки безопасности. Hyper-V также поддерживает кластеризацию, что означает, что несколько физических серверов могут работать вместе как единый ресурс для повышения отказоустойчивости и масштабируемости.

Docker — это инструмент для контейнеризации приложений, который позволяет разработчикам упаковывать приложения со всеми необходимыми зависимостями в контейнер. Docker использует Docker Engine[[26]](#footnote-25) для управления контейнерами и Docker Hub для хранения образов контейнеров. Docker также поддерживает оркестрацию контейнеров с помощью таких инструментов, как Docker Compose и Kubernetes[[27]](#footnote-26).

Виртуализация и контейнеризация имеют свои преимущества и недостатки. Виртуализация предоставляет полную изоляцию между виртуальными машинами, но требует больше ресурсов, чем контейнеризация. Контейнеризация быстрее и эффективнее, но может иметь проблемы с безопасностью, если злоумышленник получит доступ к хосту.

Продемонстрируем контейнеры проекта, их образы, названия, порты. В нашем случае, для корректной работы Docker Engine, требуется включенный HyperV. Чтобы включить, нужно выполнить shell-команду

Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true



*Рис.4. Иллюстрация контейнеров*

## **2.2 Разработка**

### **2.2.1 API**

**Обработчик запросов:** для серверной части на Python нужно использовать библиотеку для обработки HTTP запросов, выбор пал на Fast API, по следующим причинам:

1. Одной из основных причин выбора Python Fast API является его высокая производительность. Fast API оптимизирован для обеспечения высокой скорости обработки запросов, что позволяет приложениям работать быстро и эффективно даже при больших объемах трафика. Это особенно важно для проекта, который требует обработки большого количества данных в реальном времени.
2. Python Fast API предлагает широкий спектр расширений и плагинов, которые позволяют легко добавлять новые функции и возможности к приложению. Это значительно упрощает процесс разработки и позволяет сосредоточиться на создании инновационных решений, вместо того чтобы тратить время на написание базового кода.

**Серверный контейнер:** чтобы запустить обработку запросов нужно включить сервер, серверной библиотекой станет Uvicorn, по трем причинам:

1. Производительность: оптимизирован для обеспечения минимального времени ответа, что критически важно для приложений, требующих обработки большого количества данных в реальном времени.
2. Совместимость: полностью совместим с ASGI (Asynchronous Server Gateway Interface), что означает, что он может работать с любыми ASGI-совместимыми приложениями, включая Fast API. Это дает большую гибкость в выборе компонентов и инструментов.
3. Асинхронность: поддерживает асинхронную обработку запросов, что позволяет приложениям обрабатывать несколько запросов одновременно, улучшая общую производительность и эффективность.

**Клиент:** чтобы клиент увидел созданные ресурсы, мог взаимодействовать с функционалом посредством UI/UX необходим HTTP клиент. Было принято решение использовать библиотеку React JS, нас же интересует именно компонент отправляющий запросы - Fetch API. У серверной части, есть требования по совместимости, которым Fetch API удовлетворяет:

1. Fetch API является частью стандартной спецификации Web, что означает, что он поддерживается всеми современными браузерами и устройствами. Это обеспечивает широкую совместимость и универсальность приложения.
2. Поддерживает асинхронные запросы, что позволяет приложению оставаться отзывчивым и не блокировать основной поток выполнения. Это особенно важно для интерактивных веб-приложений.
3. Позволяет выполнять параллельные запросы, что может ускорить загрузку данных и улучшить производительность приложения.
4. Fetch API поддерживает Cross-Origin Resource Sharing (CORS), что позволяет приложению взаимодействовать с API, расположенными на разных доменах.

**Политика CORS:** CORS (Cross-Origin Resource Sharing) - это механизм, который позволяет веб-страницам, загруженным с одного сайта, получать доступ к ресурсам (например, изображениям, стилям, скриптам, XMLHttpRequest и Fetch API), размещенным на другом сайте. Этот механизм реализован в виде заголовков HTTP, которые сервер может устанавливать для своих ответов, чтобы указать, какие домены могут читать и записывать ресурсы, которые он обслуживает. В нашем случае, нужно разрешить локальному клиенту получать доступ по всем HTTP методам. Напишем следующие настройки в серверном Fast API приложении:

from fastapi.middleware.cors import CORSMiddleware

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"]

)

### **2.2.2 Dockerfile**

Напишем к клиенту и серверу соответствующие Dockerfile файлы, именно они описывают сборку образов. Клиентский Dockerfile, node образ, копируем файлы в образ и запускаем через yarn, команда для сборки образа:

$ docker buildx build . --tag funcid/oppenganger-frontend:latest

FROM node:20.4.0-alpine

WORKDIR /app

COPY src /app/src

COPY public /app/public

COPY package.json /app/

COPY postcss.config.js /app/

COPY tailwind.config.js /app/

COPY yarn.lock /app/

COPY tsconfig.json /app/

RUN yarn install

CMD ["yarn", "start"]

*Рис.5. Frontend Dockerfile*

Серверный Dockerfile вышел в 74 строки, поэтому его содержимого здесь представлено не будет. Для представления информации достаточно текстового описания:

Dockerfile описывает процесс создания двух образов Docker для работы с моделями машинного обучения. Первый образ (builder) используется для установки необходимых зависимостей и сборки пакета, второй (final) содержит все необходимые модели и запускает приложение.

1. Аргументы (ARG) определяют переменные, значения которых могут быть переданы при создании образа. В данном случае CUDA\_VERSION и PYTHON.
2. Образы (FROM) указывают на базовые образы, которые будут использоваться для создания новых. В данном случае используются образы NVIDIA CUDA с различными версиями и наборами пакетов.
3. Рабочие директории (WORKDIR) задают корневые директории для каждого этапа сборки.
4. Команды (RUN) выполняют различные операции установки и настройки внутри контейнера. Например, устанавливаются необходимые пакеты, обновляются списки пакетов, устанавливается Python и его зависимости, копируются файлы проекта и т.д.
5. Команды (COPY) копируют файлы из внешнего источника в контейнер.
6. Переменная окружения (ENV) определяет переменную среды, которая будет доступна после создания образа.
7. Точка входа (ENTRYPOINT) указывает на скрипт, который будет запущен при старте контейнера.

Команда для сборки образа:

$ docker buildx build . --target final --build-arg CUDA\_VERSION=12.2.2 --tag funcid/oppelganger:latest

### **2.2.3 Вспомогательные функции**

К сожалению, пользоваться этим ПО в чистом виде будет непрактично, поэтому нужны доработки в качестве вспомогательных функций чтобы соответствовать требованиям из пункта 1.1. Можно выделить следующие функции:

1. Проверка совместимости ПО и аппаратного обеспечения. Доработка заключается в том, чтобы пользователь сразу видел статус совместимости на экране, а в случае отказа в дальнейшей работе, клиент предпринял действия для устранения несовместимости. Проверка осуществляется на клиентской стороне в виде отказа в обслуживании мобильных устройств. На серверной стороне за этот функционал отвечает ручка /check\_gpu\_driver, которая проверяет наличие видеокарты с CUDA ядрами на устройстве.
2. Улучшение качества видео потока. Этап повышения качества самый долгий, он занимает от 30 до 80% всего времени ответа. Есть смысл позволить пользователю выбирать что для него важнее: скорость ответа или качество выходного видео. В запросе /send\_request есть параметр enhance, который отвечает за настройку качества. Поэтому было принято решение сделать UI/UX элемент, который дает возможность настроить качество в экране отображения видео.
3. Список профилей. При первичном взаимодействии с продуктом у пользователя отсутствуют какие-либо готовые профили, поэтому требуется пройти все этапы создания нового профиля, однако уже при втором и последующих обращениях в этом надобность отпадает. Нужно хранить профили в хранилище и давать клиенту возможность выбирать готовые профили, сделанные ранее. Для этого на главной странице приложения присутствует кнопка с выбором профилей. При клике на эту кнопку, клиентская сторона делает запрос на серверную и отображает результат запроса.
4. Клиент для работы с хранилищем. Иногда приходится руками лезть в хранилище данных и смотреть что там лежит. Правильно ли файл закодировался? Валидная ли ссылка сгенерировалась? Правильно ли настроена политика CORS для доступа к ресурсам хранилища? Все эти вопросы решаются клиентским приложением, к счастью, в комплекте с дистрибутивом самого хранилища, в образе Minio так же есть клиентское приложение, по умолчанию, запускаемое с портом 9001.

## **2.3 Тестирование**

Для продукта, нужно выбрать тип тестирования, это может быть как проверки со стороны разработчика: Unit-тестирование, интеграционное, сквозное, так со стороны QA-инженера: ручное, автоматическое. Выбор пал на ручное (мануальное) тестирование по следующим причинам.

Комплектность продукта: продукт включает в себя несколько компонентов (frontend, backend, S3 хранилище), каждый из которых требует отдельного тестирования. Мануальное тестирование позволяет тщательно проверить каждую часть системы, убедиться в их корректной работе вместе и выявить возможные проблемы взаимодействия между компонентами.

Реалистичность видео ответов: поскольку продукт генерирует реалистичные видео ответы, важно провести тщательное тестирование, чтобы убедиться в качестве и точности этих ответов. Мануальное тестирование позволяет вручную проверять каждый видеоответ, анализировать его содержание и сравнивать с исходным текстом или голосом/лицом человека.

Пользовательский опыт: важно также учесть пользовательский опыт при использовании продукта. Мануальное тестирование позволяет оценить удобство интерфейса, скорость работы системы и другие аспекты, влияющие на удовлетворенность пользователей.

Непредсказуемость результатов: генерация видео ответов может иметь непредсказуемые результаты, особенно когда используются нейронные сети или другие сложные алгоритмы. Мануальное тестирование позволяет обнаружить и проанализировать такие случаи.

Для ручного тестирования следующие инструменты:

1. PostMan
   1. Позволяет легко создавать и выполнять различные типы запросов (GET, POST, PUT, DELETE и т.д.)
   2. Имеет встроенные шаблоны для тестирования RESTful API
   3. Позволяет автоматизировать тестирование API, создавая коллекции запросов и сценарии
   4. Поддерживает мониторинг состояния API и отслеживание ошибок
2. Консоль разработчика Chrome:
   1. Позволяет просматривать и анализировать HTTP-запросы и ответы прямо в браузере
   2. Дает возможность изучать и отлаживать JavaScript, CSS и HTML на странице
   3. Позволяет тестировать производительность и оптимизировать веб-страницы
3. Charles Proxy:
   1. Это инструмент для мониторинга сетевого трафика, который позволяет перехватывать и редактировать HTTP и HTTPS трафик
   2. Позволяет просматривать запросы и ответы в удобном формате, включая тело ответа и заголовки
   3. Может использоваться для имитации медленного соединения, что полезно при тестировании производительности приложения
   4. Поддерживает SSL-проксирование, что позволяет просматривать защищенный трафик

Отчет о результатах тестирования:

Продукт представляет собой комплексную систему, включающую в себя клиент, сервер, S3 хранилище. Тестирование проводилось на выборке из 100 запросов. Использовалась основная аппаратура, разница с второстепенной составляет не более чем один порядок.

**Клиент:** интерфейс прост и интуитивно понятен. Загрузка страниц происходит быстро, без заметных задержек. Все элементы интерфейса отображаются корректно и соответствуют требованиям.

**Сервер:** все запросы обрабатываются быстро и без ошибок. Время генерации текста составляет менее секунды, синтез речи и синхронизация губ занимают до пяти секунд (по отдельности). Улучшение качества видео занимает до двух минут, что является приемлемым временем для такого рода задач.

**Хранилище:** скорость загрузки и получение ресурсов моментальная (что и следует из выбора локальной сети).

**Критичные ошибки:** необрабатываемые ошибки случились менее чем в 1% случаев, что является приемлемым уровнем.

В целом, продукт показал высокую стабильность и быстродействие. Интерфейс прост и удобен в использовании. Результаты тестирования подтверждают готовность продукта к использованию в реальных условиях.

## **2.4 Развертывание**

## Клонирование всех репозиториев из системы контроля версий выполняется при помощи команд в Git Bash[[28]](#footnote-27):

## **$ git clone git@github.com:Oppelganger/oppelganger.git**

## **$ git clone git@github.com:Oppelganger/oppelganger-frontend.git**

## **$ git clone git@github.com:Oppelganger/oppelganger-devops.git**

## ПолучениеDocker образов с сервиса Docker Hub:

## **$ docker pull funcid/oppelganger:latest**

## **$ docker pull funcid/oppelganger-frintend:latest**

## Переходим в папку oppelganger-devops и запускаем проект:

## **$ cd ./oppelganger-devops**

## **$ docker compose up -d**

# **Заключение**

В ходе данной курсовой работы был реализован современный инструмент для имитации живого видео на базе пяти моделей для генерации текстового ответа, синтеза речи, распознавания лиц, синхронизации губ и повышения качества видеопотока с ответом на запрос менее чем в 2 минуты. Произведен поиск оптимальных нейронных сетей и моделей для вышеупомянутого класса задач. Был разработан пакет из следующих компонентов: серверная часть на языке Python с библиотекой Fast API и сервером Uvicorn, клиентская часть на языке TypeScript на ReactJS. Также был использовано хранилище Minio для общего доступа к ресурсам. Была проработана архитектура решения, создан протокол взаимодействия компонентов, настроена политика CORS. Все компоненты были собраны в образы и опубликованы в Docker Hub, все репозитории проекта стали доступны на GitHub и GitVerse. Было произведено ручное тестирование системы и составлен отчет позволяющий использовать продукт в промышленной среде эксплуатации. Интерфейс клиентского приложения прост и понятен, на главной странице можно увидеть результат запроса на совместимость программного обеспечения и аппаратуры, а также выбрать готовые профили или функцию создание нового. Процесс создания нового профиля интуитивно понятен, разбит на логические части. Интерактивное окно позволяет отправлять запросы на генерацию видео, отображать результат, а также содержит настройку выбора из двух вариантов: высокое качество или быстрый ответ. В ходе курсовой работы использовались генеративные инструменты, такие как российские сервисы от Сбер: GigaCode и GigaChat, их вклад составляет менее 10%.

# **Список литературы**

Чистая архитектура, Роберт С. Мартин

1. Роберт С. Мартин, Чистая архитектура, 2022 г.
2. Влад Хононов, Изучаем DDD-предметно-ориентированное проектирование

1. LLaMa, <https://github.com/ggerganov/llama.cpp.> [↑](#footnote-ref-0)
2. HF-Coqui XTTS, <https://huggingface.co/spaces/coqui/xtts> [↑](#footnote-ref-1)
3. Wav2Lip, <https://github.com/Rudrabha/Wav2Lip> [↑](#footnote-ref-2)
4. GFP-GAN, <https://github.com/TencentARC/GFPGAN> [↑](#footnote-ref-3)
5. S3 или Simple Storage Service — сервис, где хранятся цифровые данные большого объема [↑](#footnote-ref-4)
6. Хранилище BLOB-объектов — это тип облачного хранилища для неструктурированных данных. «BLOB», сокращение от «Большой двоичный объект», представляет собой массу данных в двоичной форме, которая не обязательно соответствует какому-либо формату файла [↑](#footnote-ref-5)
7. MinIO — это высокопроизводительное объектное хранилище с открытым исходным кодом, предназначенное для локального использования или в качестве альтернативы облачному хранилищу [↑](#footnote-ref-6)
8. Docker — это открытая платформа для разработки, доставки и эксплуатации приложений [↑](#footnote-ref-7)
9. HTTP — широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов [↑](#footnote-ref-8)
10. Протокол TLS (transport layer security) основан на протоколе SSL (Secure Sockets Layer) [↑](#footnote-ref-9)
11. Параллелизм — это свойство систем, при котором несколько вычислений выполняются одновременно, и при этом, возможно, взаимодействуют друг с другом [↑](#footnote-ref-10)
12. Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент [↑](#footnote-ref-11)
13. JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Легко читается человеком и машиной [↑](#footnote-ref-12)
14. Предметно-ориентированное проектирование (реже проблемно-ориентированное, англ. domain-driven design, DDD) — набор принципов и схем, направленных на создание оптимальных систем объектов. Сводится к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом [↑](#footnote-ref-13)
15. <https://www.docker.com/products/docker-desktop/> [↑](#footnote-ref-14)
16. NVIDIA разрабатывает графические процессоры и продвигает достижения в области искусственного интеллекта, высокопроизводительных вычислений [↑](#footnote-ref-15)
17. CUDA (изначально аббр. от англ. Compute Unified Device Architecture) — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia [↑](#footnote-ref-16)
18. FFmpeg — набор свободных библиотек с открытым исходным кодом, которые позволяют записывать, конвертировать и передавать цифровые аудио- и видеозаписи в различных форматах [↑](#footnote-ref-17)
19. DevOps — методология автоматизации технологических процессов сборки, настройки и развертывания программного обеспечения. Методология предполагает активное взаимодействие специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимную интеграцию их технологических процессов друг в друга для обеспечения высокого качества программного продукта [↑](#footnote-ref-18)
20. QA — Quality Assurance — переводится с английского как «обеспечение качества». QA-инженер — специалист, который следит за качеством продукта на всех этапах его разработки [↑](#footnote-ref-19)
21. React — это JavaScript-библиотека для создания пользовательских интерфейсов [↑](#footnote-ref-20)
22. Webpack — сборщик модулей, который позволяет скомпилировать JavaScript-модули в единый JS-файл [↑](#footnote-ref-21)
23. Git — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года [↑](#footnote-ref-22)
24. Jira — это инструмент управления проектами, который помогает оптимизировать работу команды [↑](#footnote-ref-23)
25. Jenkins — программная система с открытым исходным кодом на Java, предназначенная для обеспечения процесса непрерывной интеграции программного обеспечения [↑](#footnote-ref-24)
26. Движок Docker (Docker Engine) — это клиент-серверное приложение. Компания Docker разделила движок Docker на два продукта [↑](#footnote-ref-25)
27. Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию [↑](#footnote-ref-26)
28. Git Bash — это командная строка в Windows, которая позволяет выполнять команды Git [↑](#footnote-ref-27)