МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**Национальный исследовательский ядерный университет «МИФИ»**

---

**Институт интеллектуальных кибернетических систем**

**КАФЕДРА КИБЕРНЕТИКИ (№ 22)**

Направление подготовки 09.03.04 Программная инженерия

# Пояснительная записка

к научно-исследовательской работе студента на тему:

# Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

Группа Б21-504

Студент _____ Попов З.А.

Руководитель _____ Зайцев К.С.

Научный консультант _____ Дунаев М.Е.

Оценка руководителя 12

Оценка комиссии _____

Члены комиссии

_____ _____

_____ _____

_____ _____

_____ _____

**Москва 2025**

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**Национальный исследовательский ядерный университет «МИФИ»**

---

**Институт интеллектуальных кибернетических систем**

**КАФЕДРА КИБЕРНЕТИКИ (№ 22)**

# Задание на НИР

Студенту гр. Б21-504 Попов Захар Андеевич

## ТЕМА НИР

Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

## ЗАДАНИЕ

| № п/п | Содержание работы | Форма отчетности | Срок исполнения | Отметка о выполнении<br>Дата, подпись |
|---|---|---|---|---|
| **1.** | **Аналитическая часть** | | | |
| 1.1. | Анализ и сравнение архитектур построения серверных приложений | | | |
| 1.2. | Анализ и сравнение паттернов проектирования модулей системы | | | |
| 1.3. | Анализ инструментов применяемых для реализации системы | | | |
| **2.** | **Теоретическая часть** | | | |
| 2.1. | Разработка моделей микросервисов | | | |
| 2.2. | Разработка общей модели системы | | | |
| **3.** | **Инженерная часть** | | | |
| 3.1. | Проектирование модулей на основе разработанных моделей | | | |
| 3.2. | Проектирование системы на основе разработанных модели взаимодействия | | | |
| **4.** | **Реализация** | | | |
| 4.1. | Реализация системы | | | |

## ЛИТЕРАТУРА

Дата выдачи задания: 26.12.2024

Руководитель _____ Зайцев К.С.

Студент _____ Попов З.А.

# Реферат

Общий объем основного текста, без учета приложений 54 страниц, с учетом приложений 65. Количество использованных источников 30. Количество приложений 2.

МИКРОСЕРВИСЫ, GOLANG, РАСПРЕДЕЛЕННЫЕ ТРАНЗАКЦИИ, POSTGRESQL, REDPABDA

Целью данной работы является Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

В первой главе проводится обзор и анализ предметно области

Во второй главе описываются использованные и разработанные модели системы и модулей. Третья глава посвященна разработки архитектуры системы и выбору програмных средств и технологий для реализации. В четвертой главе приводится реализация системмы и ее тестирование

# Содержание

**Введение** **4**

**1 Анализ подходов к разработке серверных приложений ИИ ассистентов** **5**

1.1 Анализ применения ИИ ассистентов в медицине . . . . . 5

1.1.1 Анализ основных областей применения ИИ ассистентов . . . . . 5

1.1.2 Обзор существующих систем поддержки принятия врачебных решений с применением искусственного интеллекта . . . . . 6

1.2 Анализ основных архитектур для построения серверных приложений . . . 7

1.2.1 Микросервисная архитектура . . . . . 8

1.2.2 Монолитная архитектура . . . . . 10

1.2.3 Сравнение монолитной и микросервисной архитектур . . . . . 11

1.3 Анализ и сравнение паттернов проектирования модулей системы . . . . . 11

1.3.1 Луковая архитектура . . . . . 13

1.3.2 Hexagonal Architecture . . . . . 14

1.3.3 Чистая архитектура . . . . . 15

1.4 Выводы . . . . . 16

1.5 Постановка цели и задач ВКР . . . . . 17

**2 Теоретическая часть** **18**

2.1 Основные подходы к авторизации в серверных приложениях . . . . . 18

2.2 Модель представления узлов и сегментов образований в щитовидной железе 20

2.3 Функциональных требований к системе Интеллектуального ассистента врача щитовидной железы . . . . . 20

2.3.1 Выявление предметных областей системы . . . . . 21

2.4 Моделирование системы . . . . . 22

2.4.1 Компонент авторизации и аутентификации . . . . . 22

2.4.2 Компонент Управление УЗИ снимками . . . . . 23

2.4.3 Компонент Управление пользователями . . . . . 24

2.4.4 Модель системы . . . . . 25

2.5 Обощенная модель компонента системы . . . . . 25

2.6 Выводы . . . . . 26

**3 Результаты проектирования системы Интеллектуального ассистента врача щитовидной железы 28**

3.1 Разработка микросервисной архитектуры системы . . . . . 28

3.1.1 Паттерн API Composition . . . . . 28

3.1.2 Способы взаимодействия между микросервисами . . . . . 29

3.1.3 Итоговая архитектура системы Интеллектуального ассистента врача щитовидной железы . . . . . 29

3.2 Выбор технологий и инструментов для технической реализации системы . 30

3.2.1 Языка программирования . . . . . 30

3.2.2 Реляционные хранилища данных . . . . . 32

3.2.3 Хранилище УЗИ снимков . . . . . 33

3.2.4 Синхронное взаимодействие между микросервисами . . . . . 36

3.2.5 Асинхронное взаимодействие . . . . . 38

3.3 Выводы . . . . . 39

**4 Реализация и тестирование системы 40**

4.1 Реализация компонентов системы в виде микросервисов . . . . . 40

4.1.1 Сущности предметной области . . . . . 40

4.1.2 Логические сценарии использования . . . . . 41

4.1.3 Интерфейсы взаимодействия с внешними системами . . . . . 42

4.1.4 Контракты микросервисов и API системы . . . . . 43

4.1.5 Топики стриминга сообщений для асинхронного общения . . . . . 44

4.1.6 Структура хранилища узи снимков . . . . . 44

4.2 Состав и структура реализованного программного обеспечения . . . . . 44

4.2.1 Структура приложения . . . . . 45

4.3 Основные сценарии работы пользователя . . . . . 45

4.4 Тестирование системы . . . . . 46

4.4.1 Юнит тестирование . . . . . 46

4.4.2 Интеграционное тестирование . . . . . 49

4.5 Сквозное тестирование . . . . . 50

4.6 Выводы . . . . . 53

**Заключение 54**

**Приложения** **55**

**A Основные части компонента управление УЗИ снимками** **55**

**B API системы** **64**

# Введение

Экспертиза щитовидной железы узла является очень сложной задачей и требует высококвалифицированных медицинских работников. Специальная система классификации EU-TIRADS используется для определения типа узла, основанного на различных особенностях узла, таких как размер, форма и источник эха. Тем не менее, ручная классификация узлов в соответствии с системой EU-TIRADS скучна и может привести к ошибкам.

Из-за широкого использования методов глубокого обучения в различных областях жизни эти методы также используются для решения проблем диагностических эндокринных заболеваний. Чтобы эффективно использовать эти методы в ежедневной работе работников здравоохранения, проводящих эндокринные исследования, необходимо разработать серверные приложения для интуитивного взаимодействия между медицинским персоналом и технологиями искусственного интеллекта.

Целью работы является разработка серверного приложения для упрощения классификации заболеваний в соответствии с EU-TIRADS. Чтобы достичь этой цели, вам необходимо будет разработать архитектуру для вашего приложения для вашего сервера или финансировать существующие, предоставить приложения на единой цифровой платформе российской федерации «ГосТЕХ» и предоставить приложения для измерения скорости обработки и количества запросов, выполняемых одновременно.

Аналогичные технологии уже были реализованы в виде дополнительного программного обеспечения, интегрированного в ультразвуковую диагностику или прикладное программное обеспечение. Однако на данный момент такая технология не представлена на российском рынке. Следовательно, разработка серверных приложений для упрощения классификации заболеваний в EU-TIRADS является важным шагом в разработке эндокринной диагностики. Помимо развития области, российские разработки сопособствуют импортозамещению иностранных подобных технологий.

# 1. Анализ подходов к разработке серверных приложений ИИ ассистентов

## 1.1 Анализ применения ИИ ассистентов в медицине

### 1.1.1 Анализ основных областей применения ИИ ассистентов

Государственное направление по улучшению всех жизненных сфер путем оцифровки, приводит к многочисленным инновациям и крупномасштабным ИТ-проектам. Реализация информационных технологий происходит в области здравоохранения это ключевой показатель уровня жизни страны.

Первоначально цифровизация медицины была предназначена для повышения диагностической эффективности и сокращения времени, необходимого для предоставления медицинских услуг. Рутина и сложные задачи передаются на производительность устройства и выполняются с использованием различных технических достижений. С этой целью были созданы многие системы и платформы, которые сокращают время для реализации стандартных диагностических процессов. В этом списке системы с искусственным интеллектом (ИИ) могут быть важны для замены ручной работы интеллектуальными навыками из различных отраслей. Искусственный интеллект - это область науки и техники, которая позволяет компьютерам и программам выполнять интеллектуальные задачи. У него есть возможность выполнять различные задачи, учиться при их использовании и адаптироваться к новым задачам и контекстам. Кроме того, ИИ может выступать в качестве дополнительной аналитической перспективы, позволяя упростить многие процессы с помощью анализа данных и вопросов поиска схемы.

Искусственный интеллект может изменить систему здравоохранения, автоматизируя многие из ее аспектов, позволяя сотрудникам сосредоточиться на основных задачах, связанных с уходом и лечением пациентов. Системы здравоохранения на основе искусственного интеллекта являются отличительной скоростью, дополненными простым использованием и снижением затрат, что делает их неотъемлемой частью разработки.

Сфера здоровья, в которой активно используется искусственный интеллект, может быть разделена на четыре основных частей.

– используется графическая информация (анализ результатов радиационной диагно-

стики).

- принятие и диагноз на основе различных метаданных.
- Дистанционные консультации и обеспечение контроля здоровья пациентов.
- Исследование лекарств, создание и реализация.

### 1.1.2 Обзор существующих систем поддержки принятия врачебных решений с применением искусственного интеллекта

В России значительное число медицинских компаний и стартапов также используют искусственный интеллект для поддержки медицинских решений. Основными областями работы являются анализ исследований на флуоресцентных (FLG), X-Ray -изображениях (RG) и компьютерной томографии (КТ).
Крупнейшие ИИ медицинские системы:

- MDDC создается для поддержки медицинских решений в соответствии с первоначальным диагнозом, инструментами и лабораториями. Система интегрирована в различные системы данных и источники данных и использование технологий искусственного интеллекта для обработки и анализа данных о здоровье.
- Celsus - это система поддержки медицинских решений, основанных на технологиях искусственного интеллекта, позволяя вам анализировать цифровые медицинские изображения, обнаруживать объекты и автоматически формировать результаты. Основными областями разработки являются анализ изображений X-RAY и флюорографии, анализ КТ и легких.
- «ФтизисБиоМед», продукты и услуги широко используются в практической медицине, чтобы снизить неправильный диагноз и обеспечить раннюю диагностику на ранних стадиях заболевания. Среди наиболее важных событий в компании можно выделить услуги для анализа медицинских изображений (флурограмма и X-Rays) на основе ИИ.

Основываясь на анализе лидеров в области исследований искусственного интеллекта в области медицины, мы можем сделать вывод, что ультразвуковой анализ щитовидной железы является одной из наиболее актуальной задач. Выбор области исследований и скачок в финансировании проектов медицинских технологий произошел в 2020 году, объясняющий эпидемиологическую ситуацию с Covid-19. Самым важным решением, принятым в то время, был постановление правительства о оцифровке системы здравоохранения

и создании различных государственных платформ здравоохранения поэтому решение создать полную систему для диагностики гормональных заболеваний в соответствии с изображением узи снимков является наиболее перспективным способом достижения цели на работе.

Кроме того, основные направления задач выполняются с помощью систем, которые можно различить это обнаружение патологии на основании графических результатов клинических исследований и предварительной диагностики. Этот выбор обусловлен сложностью диагностического процесса и необходимостью высокой подготовки для экспертов.

## 1.2 Анализ основных архитектур для построения серверных приложений

Традиционная архитектура на основе клиент-серверной модели является одной из наиболее распространенных архитектур для создания серверных приложений. Эта модель предполагает разделение приложения на две составляющие - клиентскую и серверную. Клиентская составляющая отвечает за интерфейс взаимодействия с пользователем, а серверная - за обработку запросов и предоставление данных. Основным преимуществом клиент-серверной модели является возможность создания распределенной архитектуры, которая позволяет создавать приложения с большими объемами данных или с большим количеством пользователей. Также этот подход позволяет легко масштабировать и обновлять серверное приложение без влияния на клиентскую составляющую. Одним из недостатков клиент-серверной модели является ее зависимость от сети. Отказ сети или недоступность сервера может привести к невозможности доступа к приложению. Еще одним недостатком является необходимость настройки и поддержки серверной инфраструктуры, что может требовать отдельных затрат и увеличить стоимость разработки и поддержки серверного приложения. Для реализации клиент-серверной модели на практике используются различные технологии и инструменты, как проприетарные, так и открытые. Например, для создания серверных приложений могут использоваться языки программирования, такие как Golang, Java, Python, Ruby, PHP, а для создания клиентских приложений - HTML, CSS, JavaScript. Существует также множество фреймворков, библиотек и инструментов, которые предназначены для упрощения разработки серверных приложений на основе клиент-серверной модели. Например, Node.js является популярным фреймворком для создания серверных приложений на JavaScript, а Django и Flask - для Python. Также используются такие средства, как базы данных, предназначенные для хра-

нения данных приложения, или сервисы и инструменты для автоматизации и управления развертыванием и масштабированием серверных приложений. Кроме того, существуют такие платформы, как Яндекс Облако и Selectel, которые предоставляют в облаке готовые инструменты для создания и развертывания серверных приложений на основе клиент-серверной модели.

### 1.2.1 Микросервисная архитектура

Микросервисная архитектура является относительно новым подходом к разработке серверных приложений[1]. Она характеризуется модульностью, то есть каждый модуль является отдельным сервисом, который может быть развернут и масштабирован отдельно от других модулей. Микросервисная архитектура рассматривает приложение как совокупность небольших сервисов, которые представляют отдельные функции приложения. Каждый сервис обычно работает независимо от других сервисов, используя API для обмена данными между другими сервисами. Основное преимущество микросервисной архитектуры заключается в ее модульности. Это дает возможность быстро отвечать на изменения требований и легко масштабировать приложение по мере необходимости. Другим преимуществом микросервисной архитектуры является возможность использования различных технологий и языков программирования для различных сервисов. Это позволяет использовать наилучшее решение для каждой отдельной задачи. Микросервисная архитектура может быть также более устойчивой и надежной, поскольку имеется возможность аварийного отключения отдельных сервисов, позволяя уменьшить влияние отказа других компонентов на работу приложения в целом. Существует несколько типов микросервисных архитектур, которые различаются по тому, как организованы и взаимодействуют между собой микросервисы. К одному из распространенных типов микросервисных архитектур относится оркестрованная архитектура [8] – это такая архитектура, в которой существует централизованный компонент или слой, называемый оркестратором, который координирует действия и взаимодействие других микросервисов в системе. Оркестратор отвечает за управление жизненным циклом запросов от клиентов, контроль последовательности выполнения операций и обработку бизнес-логики интеграции между сервисами. Ещё одним из распространённых типов микросервисных архитектур является event-driven архитектура – архитектура, в которой микросервисы взаимодействуют через отправку и прием событий. Каждый сервис может публиковать события, на которые другие сервисы могут, соответственно, подписываться и реагировать. К её преимуществам

относятся: гибкость (компоненты могут быть легко добавлены или удалены без влияния на другие части системы), масштабируемость (каждый компонент может масштабироваться независимо для обеспечения высокой производительности), отказоустойчивость (отказ одного компонента не блокирует работу всей системы). Однако тут возникает сложность в обеспечении согласованности данных между различными сервисами, управлении событиями и обработке ошибок. Также стоит упомянуть про подход API Gateway – данный тип архитектуры использует централизованный шлюз, который является единой точкой входа для всех запросов клиентов к микросервисам или другим системам. Шлюз может выполнять несколько функций, к которым относятся: аутентификация, авторизация, маршрутизация запросов, преобразование протоколов и форматов данных, кеширование, балансировка трафика и другие. Данный подход позволяет сделать архитектуру приложения более гибкой, безопасной и легкой в управлении, а также позволяет централизованно управлять всеми взаимодействиями клиентов с микросервисами, упрощая развертывание и поддержку системы.

Микросервисная архитектура предусматривает разбиение приложения на небольшие независимые сервисы, каждый из которых выполняет строго определённые задачи. Эти сервисы взаимодействуют через сети с использованием API или брокеров сообщений.

**Преимущества микросервисной архитектуры**:

- **Гибкость масштабирования**: отдельные сервисы масштабируются независимо, что оптимизирует использование ресурсов.
- **Свобода выбора технологий**: команды могут использовать разные языки программирования и инструменты для реализации отдельных сервисов.
- **Лучшая управляемость**: изолированные сервисы упрощают внесение изменений и поддержку.
- **Повышенная устойчивость**: сбой одного сервиса не влияет на работу других.
- **Параллельная разработка**: команды могут работать над разными сервисами одновременно.

**Недостатки микросервисной архитектуры**:

- **Сложность проектирования**: требует тщательного планирования взаимодействия между сервисами.
- **Проблемы с согласованностью данных**: переход к eventual consistency добавляет сложности.
- **Высокие инфраструктурные затраты**: управление множеством сервисов требует

дополнительных инструментов, таких как Kubernetes.

- **Увеличение задержек**: сетевое взаимодействие между сервисами медленнее, чем вызовы в памяти.
- **Сложности диагностики**: распределённый характер системы затрудняет выявление и исправление ошибок.

### 1.2.2 Монолитная архитектура

Монолитная архитектура представляет собой подход, при котором всё приложение разрабатывается как единый, неделимый блок. В нём объединены пользовательский интерфейс, бизнес-логика и уровень работы с данными. Такой подход исторически был основным в разработке ПО.

**Преимущества монолитной архитектуры**:

- **Простота разработки**: начальный этап разработки требует меньше усилий на планирование структуры.
- **Единая кодовая база**: централизованное хранение и управление кодом упрощает процесс разработки и интеграции изменений.
- **Простая инфраструктура**: одно монолитное приложение, не требует дополнитнельных расходов на инфраструктуру.
- **Высокая производительность**: отсутствие сетевого взаимодействия между компонентами ускоряет их взаимодействие.
- **Поддержка транзакционности**: встроенные механизмы работы с транзакциями позволяют обеспечить согласованность данных.

**Недостатки монолитной архитектуры**:

- **Сложность масштабирования**: увеличение нагрузки требует масштабирования всего приложения, даже если это касается лишь одного его компонента.
- **Ограничения технологического выбора**: единая технологическая платформа ограничивает возможности внедрения новых инструментов.
- **Рост сложности кода**: по мере увеличения объёма функциональности поддержка системы становится сложнее.
- **Долгое развертывание**: внесение малейших изменений требует пересборки и перезапуска приложения.
- **Уязвимость к сбоям**: сбой в одном компоненте приводит к отказу всей системы.

### 1.2.3 Сравнение монолитной и микросервисной архитектур

Для более наглядного понимания различий между монолитной и микросервисной архитектурами представим их основные аспекты в таблице:

| **Аспект** | **Монолит** | **Микросервисы** |
|---|---|---|
| Сложность разработки | Низкая | Высокая |
| Масштабируемость | Ограниченная | Гибкая |
| Устойчивость к сбоям | Низкая | Высокая |
| Технологический выбор | Ограниченный | Гибкий |
| Развертывание | Простое | Сложное |
| Поддержка транзакционности | Простая | Сложная |
| Задержки взаимодействия | Минимальные | Увеличенные |
| Затраты на инфраструктуру | Низкие | Высокие |

Таблица 1.1 – Сравнение монолитной и микросервисной архитектур

**Выводы**: Монолитная архитектура лучше подходит для проектов с ограниченным функционалом и небольшими командами, где важны простота и быстрота разработки. Микросервисы же оправданы в сложных, масштабируемых системах с распределёнными командами и высокими требованиями к отказоустойчивости.[2]

## 1.3 Анализ и сравнение паттернов проектирования модулей системы

Связанность (англ. Coupling) и согласованность (англ. Cohesion) являются фундаментальными концепциями в области разработки программного обеспечения[3], которые оказывают значительное влияние на читаемость, поддержку, тестируемость и переиспользование кода. В данной работе исследуются эти понятия, их виды, а также взаимосвязь между ними с целью формирования рекомендаций по их применению для повышения эффективности разработки.

#### Связанность (Coupling)

Связанность характеризует степень зависимости одного программного модуля от других. Чем выше уровень связанности, тем сильнее изменения в одном модуле отражаются на других. Стремление к низкой связанности (low coupling) является ключевым принципом проектирования, позволяющим минимизировать количество зависимостей между модулями и повысить их автономность.

Существует несколько типов связанности, классифицируемых по степени их зависимости:

- **Content coupling** (содержательная связанность): возникает, когда один модуль полагается на внутреннюю реализацию другого модуля, например, доступ к его локальным данным. Изменения в одном модуле требуют изменений в другом.
- **Common coupling** (общая связанность): модули используют общие данные, например, глобальные переменные. Изменение этих данных затрагивает все модули, которые с ними работают.
- **External coupling** (внешняя связанность): несколько модулей зависят от общего внешнего ресурса, такого как сервис или API. Изменения в этом ресурсе могут привести к некорректной работе модулей.
- **Control coupling** (управляющая связанность): один модуль управляет поведением другого через передачу управляющих сигналов, например, флагов, изменяющих поведение функций.
- **Stamp coupling** (штампованная связанность): модули обмениваются сложными структурами данных, однако используют лишь их часть. Это может приводить к побочным эффектам из-за неполной или изменяемой информации.
- **Data coupling** (связанность по данным): наименее инвазивный тип, при котором модули взаимодействуют исключительно через передачу конкретных данных. Такой подход минимизирует взаимозависимость.

#### Согласованность (Cohesion)

Согласованность характеризует степень внутренней связанности функциональных элементов модуля. Высокая согласованность подразумевает, что все компоненты модуля ориентированы на выполнение одной конкретной задачи или функции, что улучшает читаемость и управляемость кода.[**HenryKafura1981ComplexityMetrics**]

Типы согласованности включают:

- **Functional cohesion** (функциональная согласованность): модуль объединяет весь необходимый функционал для выполнения одной задачи.
- **Sequential cohesion** (последовательная согласованность): результаты одной функции используются в качестве входных данных для другой.
- **Communication cohesion** (коммуникационная согласованность): все элементы модуля работают с одними и теми же данными.
- **Procedural cohesion** (процедурная согласованность): элементы модуля выполняются в определенной последовательности, необходимой для достижения результата.

- **Temporal cohesion** (временная согласованность): функции модуля сгруппированы на основе их выполнения в определенный момент времени (например, при обработке ошибок).
- **Logical cohesion** (логическая согласованность): элементы модуля объединены логически общей функцией, но могут отличаться по своей природе (например, обработка различных типов ввода).
- **Coincidental cohesion** (случайная согласованность): элементы модуля не связаны логически, что делает модуль хаотичным и сложным для понимания.

#### Взаимосвязь между Coupling и Cohesion

Связанность и согласованность находятся в сложной взаимозависимости[4]. Уменьшение связанности обычно способствует увеличению согласованности и наоборот. Например, избыточная согласованность может привести к росту зависимости между модулями, что увеличивает связанность. Таким образом, ключевым аспектом является нахождение оптимального баланса, который обеспечит поддержку высокого уровня согласованности при минимально возможной связанности.

Применение принципов низкой связанности и высокой согласованности позволяет создавать поддерживаемые, масштабируемые и понятные программные системы[5]. Их использование способствует не только повышению качества кода, но и ускорению разработки, что особенно важно в современных условиях высокой конкуренции и быстрого изменения требований.

### 1.3.1 Луковая архитектура

**Луковая архитектура** была предложена Джеффри Палермо в 2008 году как расширение гексагональной архитектуры[6]. Её цель — облегчить поддержку приложений за счёт разделения аспектов и строгих правил взаимодействия слоёв.

#### Основная идея

Луковая архитектура предполагает многослойную организацию системы, где каждый слой зависит только от внутреннего слоя. Центральным слоем всегда является независимый слой, который представляет собой сердцевину архитектуры. Остальные слои располагаются вокруг него, создавая структуру, похожую на лук.

**Ключевые принципы**:

– Приложение строится вокруг независимой объектной модели.
– Внутренние слои определяют интерфейсы, внешние — реализуют их.
– Зависимости направлены к центру.
– Код предметной области изолирован от инфраструктуры.

#### Слои

**Domain Model**: содержит основные сущности и поведение предметной области, например, валидацию данных.

**Domain Services**: реализует бизнес-логику, которая не связана с конкретными сущностями, например, расчёт стоимости заказа.

**Application Services**: оркестрирует бизнес-логику, например, обработку запроса на создание заказа.

**Infrastructure**: реализует интерфейсы и адаптеры для работы с базами данных, внешними API и другими ресурсами.

#### Преимущества

– Независимость предметной области от инфраструктуры.
– Гибкость замены внешних слоёв.
– Высокая тестируемость.

#### Недостатки

– Сложность обучения.
– Необходимость создания множества интерфейсов.

### 1.3.2 Hexagonal Architecture

**Hexagonal Architecture** (также известная как Ports and Adapters) была предложена Alistair Cockburn[7]. Её цель — устранить зависимость бизнес-логики от внешних систем и сделать приложение более гибким и тестируемым.

#### Основная идея

Hexagonal Architecture представляет приложение в виде шестиугольника, где каждая сторона представляет интерфейс (порт) для взаимодействия с внешними системами (адаптерами). Центр архитектуры — это бизнес-логика, изолированная от деталей реализации.

**Ключевые принципы**:

- Бизнес-логика полностью изолирована от инфраструктуры.
- Взаимодействие с внешним миром происходит через порты и адаптеры.
- Внешние системы подключаются через адаптеры, соответствующие определённым портам.

**Слои**

**Core (Business Logic)**: центральная часть архитектуры, содержащая доменные сущности и бизнес-правила.

**Ports**: интерфейсы, через которые осуществляется взаимодействие между Core и внешними системами.

**Adapters**: реализация портов для подключения конкретных технологий, таких как базы данных, веб-сервисы и т.д.

**Преимущества**

- Гибкость: легко заменять внешние системы, не затрагивая бизнес-логику.
- Тестируемость: изолированная бизнес-логика упрощает написание модульных тестов.
- Чёткое разделение обязанностей.

**Недостатки**

- Сложность проектирования и реализации.
- Дополнительные накладные расходы на создание портов и адаптеров.

### 1.3.3 Чистая архитектура

**Чистая архитектура**, предложенная Робертом Мартином, основывается на идеях Onion и Hexagonal архитектур[6]. Её ключевая цель — изолировать бизнес-логику от деталей реализации.[8]

**Базовые принципы**

- Использование принципов SOLID.
- Разделение системы на слои.
- Независимость бизнес-логики от инфраструктуры.

– Тестируемость бизнес-логики без внешних зависимостей.

**Слои**

**Domain**: содержит общие бизнес-правила, структуры и интерфейсы.

**Application**: реализует конкретные сценарии использования, например, обработку запросов.

**Presentation**: отвечает за взаимодействие с пользователем через REST API, CLI и т.д.

**Infrastructure**: содержит детали реализации, такие как доступ к базе данных или внешние сервисы.

**Пересечение границ**

В чистой архитектуре зависимости направлены внутрь. Внешние слои реализуют интерфейсы, определённые внутренними слоями. Это достигается за счёт принципа инверсии зависимостей (DIP).

**Преимущества**

– Независимость от фреймворков и инфраструктуры.
– Простота тестирования.
– Переносимость и возможность разделения на микросервисы.

**Недостатки**

– Требуется строгое разделение бизнес-правил.
– Возможность излишнего усложнения структуры.

**Вывод**

**Чистая архитектура** представляет собой эволюцию луковой и гексагональной архитектур, сохраняя их принципы, но делая больший акцент на SOLID и изоляции бизнес-логики. Она подходит для сложных систем, требующих высокой гибкости и тестируемости, но может быть избыточной для небольших приложений.

## 1.4 Выводы

1. Разобраны основные методы применения ИИ в медицине. Особенности применения ИИ ассистентов в работе врача, основные аналоги систем ИИ ассистентов представленных на рынке.

2. Выполнен анализ архитектурных стилей построения систем, с учетом требований к динамической расширяемости отдельных модулей системы, выбрана микросервисная архитектура с применением API Gateway.
3. Проанализированы основные архитектурные паттерны построения приложений. Clean Architecture за счет преобраладания SOLID, разбиения на слои и преимуществ Dependency inversion выбрана как основа написания микросервисов.

## 1.5 Постановка цели и задач ВКР

Целью данной ВКР является разработка системы серверного приложения комплексной системы «Интеллектуальный ассистент врача УЗИ».

Задачи на ВКР:

1. Разграничение функционала серверного приложения на доменные области.
2. Проектирование системы Интеллектуальный ассистент врача УЗИ, с учетом разграничения на доменные области.
3. Реализация микросервисов в соответствии с принципами Clean Architecture.
4. Тестирование разработанной системы и ее производительности. Устранение недостатков при наличии.

# 2. Теоретическая часть

## 2.1 Основные подходы к авторизации в серверных приложениях

В современном веб-разработке авторизация пользователей является важной частью обеспечения безопасности и управления доступом к ресурсам[9]. Существует несколько подходов к авторизации, каждый из которых имеет свои особенности, преимущества и недостатки. Два самых популярных метода авторизации — это использование сессий и JSON Web Tokens (JWT). В этом обзоре мы рассмотрим основные аспекты каждого из этих подходов.

**Авторизация с использованием сессий** Авторизация с использованием сессий предполагает классический подход к управлению состоянием пользователя. Когда пользователь выполняет вход в систему, сервер создает сессию и сохраняет информацию о пользователе, а также уникальный идентификатор сессии (обычно в виде куки) в браузере.

**Преимущества**

- **Безопасность**: Сессии могут быть безопаснее, так как данные о пользователе хранятся на сервере, и доступ к ним ограничен.
- **Управление сессиями**: Сервер может контролировать время жизни сессии и управлять активными сессиями (например, отключать их по запросу пользователя).
- **Простота реализации**: Для простых приложений реализация сессий может быть проще и привычнее для разработчиков.

**Недостатки**

- **Сложности с масштабированием**: В распределенных системах требуется дополнительная архитектура для хранения сессий (например, база данных или кэш), что может усложнить разработку.
- **Зависимость от состояния**: Поскольку сессии хранят состояние на сервере, это может ограничить возможность использования микросервисной архитектуры.

**Авторизация с использованием JWT** JSON Web Tokens (JWT) — это подход к авторизации, основанный на использовании токенов[10]. Когда пользователь входит в систему, сервер генерирует токен, который содержит зашифрованную информацию о пользователе и отправляет его клиенту. Клиент хранит этот токен и включает его в заголовки запросов при доступе к защищённым ресурсам.

**Преимущества**

- **Безсостояние**: JWT не требуют хранения состояния на сервере. Токен самостоятельно хранит информацию о пользователе, что упрощает масштабирование и работы в распределенных системах.
- **Безопасность**: Токены можно подписывать и шифровать, что добавляет уровень защиты данных.
- **Гибкость**: JWT можно использовать для разных типов приложений, включая веб-приложения, мобильные приложения и API.

**Недостатки**

- **Управление сроком действия**: JWT имеют фиксированный срок действия, и их нужно будет обновлять. Отзыв токена может быть сложнее реализовать.
- **Размер**: Токены могут быть крупнее, чем идентификаторы сессий, что может отрицательно сказаться на производительности при частом использовании в заголовках запросов.

JWT более удобный и простой в реализации из-за того что не требуется хранение состояния на сервере.

Рассмотрим что из себя представляет сам JWT токен:

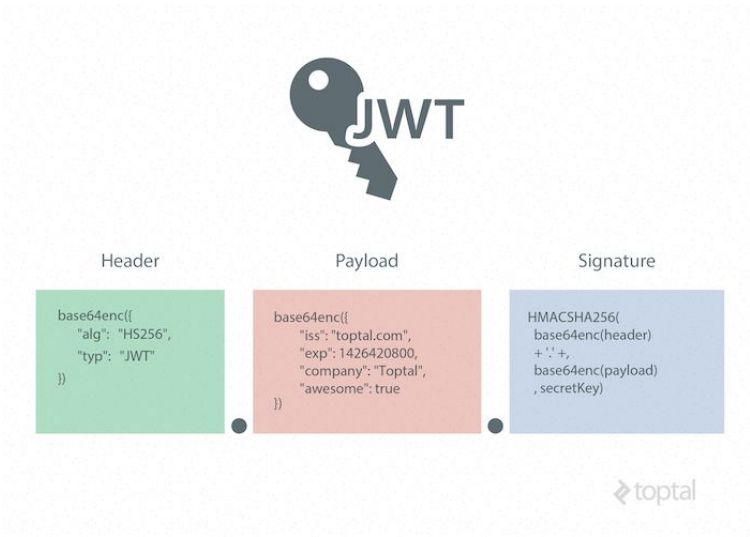


Рисунок 2.1 – JWT токен

JWT токен состоит из трех частей, разделенных точками:

- **Header (Заголовок)** - содержит тип токена и алгоритм шифрования. Обычно используется алгоритм HS256.

– **Payload (Полезная нагрузка)** - содержит утверждения (claims) о пользователе, такие как идентификатор, роль, время истечения токена и другие не часто меняющиеся пользовательские данные.
– **Signature (Подпись)** - создается путем шифрования заголовка и полезной нагрузки с использованием секретного ключа. Подпись гарантирует целостность данных и аутентичность токена.

Каждая часть токена кодируется в формате Base64. Полученные части соединяются точками, образуя единую строку - JWT токен.

## 2.2 Модель представления узлов и сегментов образований в щитовидной железе

Интеллектуальная часть ассистента анализирует УЗИ снимки щитовыдной железы на наличие образований как злокачественных, так и доброкачественных. Как правило узи сники представляют собдой кинопетлю щитовидной железы снятой под различными углами.
Исходя из этого модель представления описывает физические узлы, снятые под разными углами. Проекции этих узлов, мы будем обозначать контурами.

схема представления:

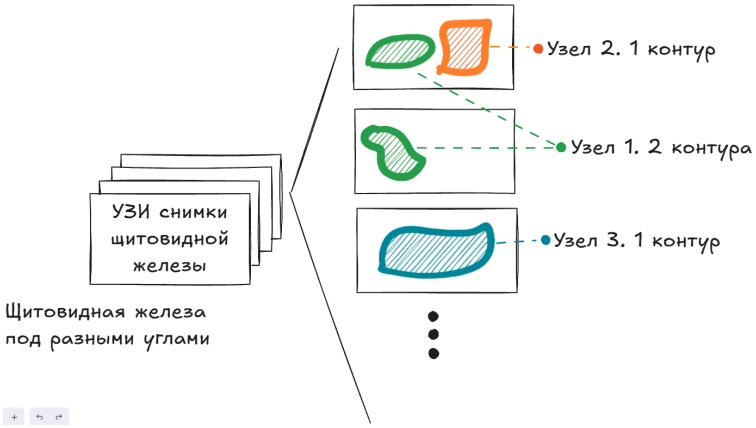


Рисунок 2.2 – Модель представления узлов и сегментов

## 2.3 Функциональных требований к системе Интеллектуального ассистента врача щитовидной железы

Система Интеллектуального ассистента врача щитовидной железы является комплексным сервисом для врача и пациента, позволяющая анализировать УЗИ снимки щитовыдной железы. Для начала, в системе требуется возможность регистрировать аккаунты

врачей и пациентов. Необходима система авторизации и аутентификации пользотелей реализованная по JWT схеме. Необходимо хранить информацию о врачах, пациентах и медицинских карточках пациентов. Для части системы отвечающей за УЗИ следует выдвинуть следующие требования. Система должна загружать снимки УЗИ и сохранять их в хранилище. Система должна передавать интеллектуальной части ИИ ассистента снимок УЗИ на обработку. Система должна получать результаты обработки от интеллектуальной части ИИ ассистента и сохранять их в базу данных. Результат обработки УЗИ интеллектуальной частью системы, должен соответствовать модели представления узлов и сегментов.

Таким образом, можно сформировать следующие функциональные требования к системе:

- Регистрация врачей и пациентов в системе
- Аутентификация и авторизация врачей и пациентов посредством JWT
- Хранение и предоставление информации о врачах, пациентах и медицинских карточках пациентов
- Загрузка и хранение УЗИ снимков
- Разбиение УЗИ снимков на единичные кадры
- Передача УЗИ снимков на обработку интеллектуальной части системы. Сохранение результатов обработки в виде модели узлов и сегментов

### 2.3.1 Выявление предметных областей системы

Исходя из функциональных требований к системе, можно выявить следующие предметные области:

**Авторизация и аутентификация** - область отвечающая за регистрацию и аутентификацию пользователей в системе. Вся логика по обработке, хранению и предоставлению JWT также осуществляется в этой области. Дальнейшая интеграции с сторонними системами в которых требуется учетные данные пользователей, будут осущетвлять посредтсвом ресурсов этой области.

**Управление пользователями** - область отвечающая за хранение и предоставление информации о врачах и пациентах. Реализует связь между пациентами и врачами, позволяет создавать историю анализов УЗИ снимков у пациентов, смотреть вердиткты ассистента и врача на различные анализы

**Управление УЗИ снимками** - область ответственная за УЗИ снимки: флоу их обработ-

ки, хранения результатов обработки, предоставление результатов врачам и пациентам. Предоставляет функционал изменения, создания и удаления узлов и сегментов.

**Интеллектуальная часть** - область отвечающая за обработку УЗИ снимков, классификацию узлов и сегментов.

## 2.4 Моделирование системы

### 2.4.1 Компонент авторизации и аутентификации

Функциональные требования к компоненту аутентификации:

- Хранение базы данных пользователей, паролей, почт и другой чувствительной информации
- Шифрование паролей с применением соли и хэша
- Возможность регистрировать пользователей, врачей и пациентов
- Создание, верификации и подпись JWT ключей для авторизации
- Обменивать логин и пароль пользователя на JWT токен
- Обновления JWT токена

Исходя из функциональных требований, модель базы данных:

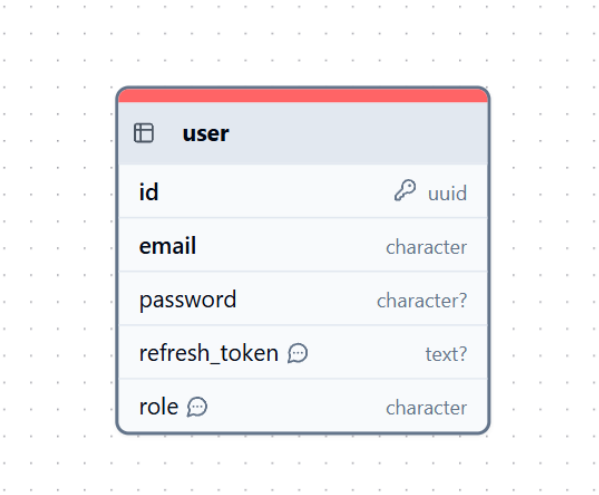


Рисунок 2.3 – Модель базы данных компонента авторизации и аутентификации

Общая модель компонента авторизации и аутентификации:

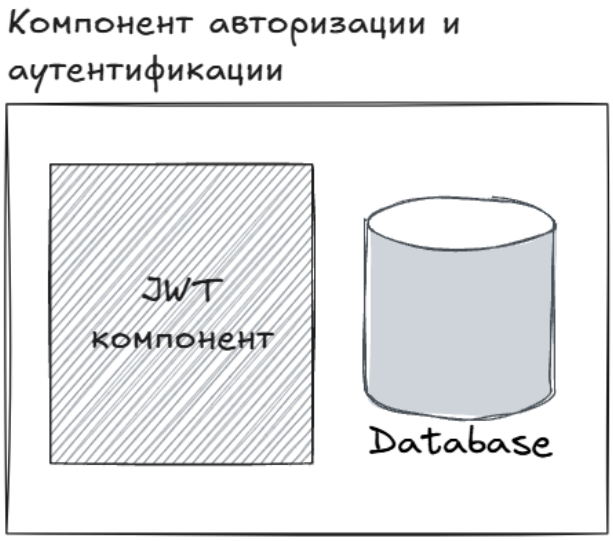


Рисунок 2.4 – Модель компонента авторизации и аутентификации

### 2.4.2 Компонент Управление УЗИ снимками

Функциональные требования к компоненту Управление УЗИ снимками:

- Принимать кинопетлю кадров узи, разбивать и сохранять ее по кадрам.
- Передавать интеллектуальной части системы узи, для обработки.
- Сохранять результаты обработки узлов и сегментов в базу данных.
- Предоставлять возможность просматривать результаты обработки узлов и сегментов.
- Предоставлять возможность совершать CRUD операции над узлами, сегментами, узи, ti-rads, images.

Тогда для соответствия требованиям, модель базы данных:

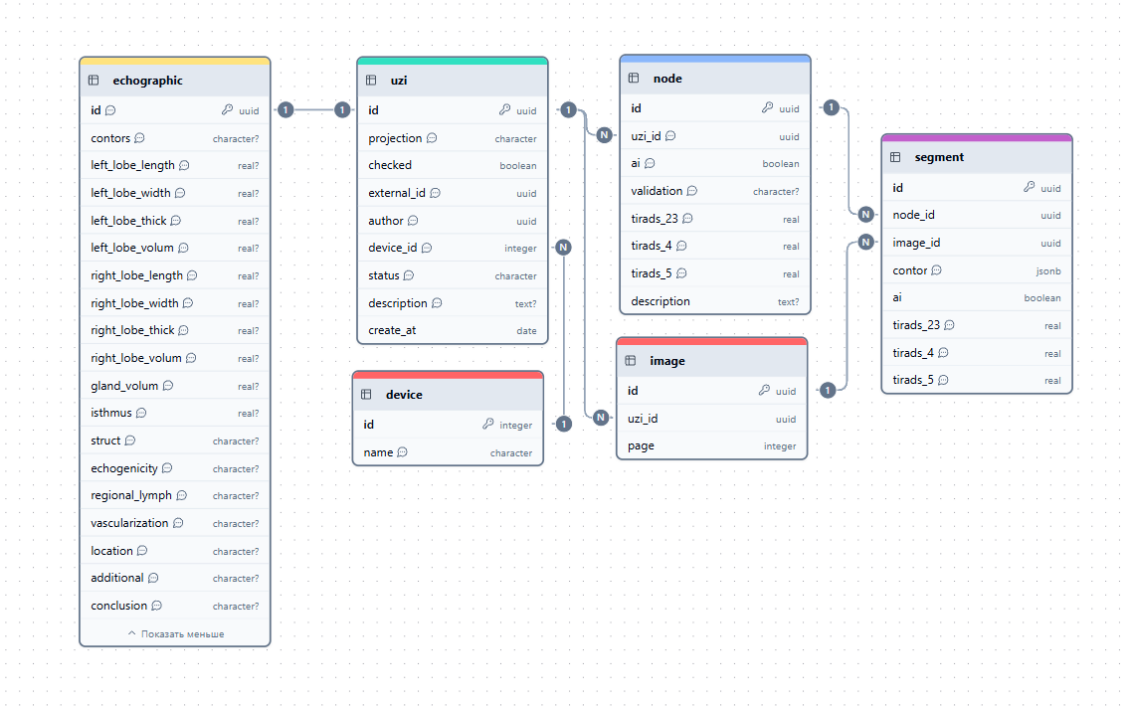


Рисунок 2.5 – Модель базы данных компонента Управление УЗИ снимками

Общая модель компонента Управление УЗИ снимками:

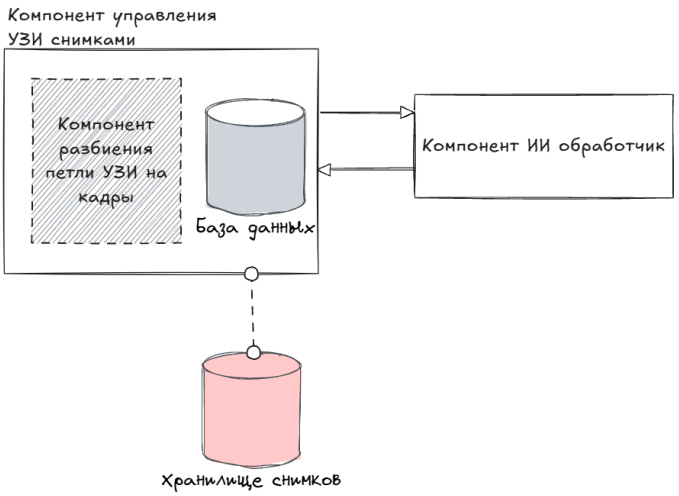


Рисунок 2.6 – Модель компонента Управление УЗИ снимками

### 2.4.3 Компонент Управление пользователями

Функциональные требования к компоненту Управление пользователями:

- Хранение публичной информации о врачах и пациентах
- Хранить историю болезни пациентов, связанную с УЗИ снимками

– Возможность врачам вести историю болезни пациентов, добавлять и удалять пациентов

Исходя из функциональных требований, модель базы данных:

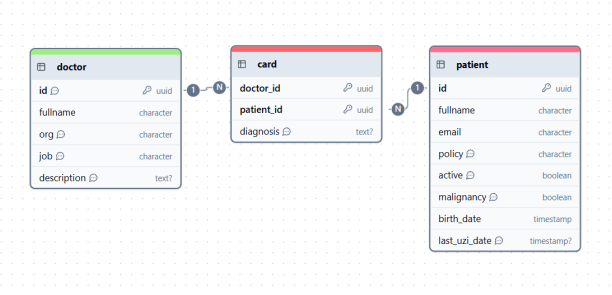


Рисунок 2.7 – Модель базы данных компонента Управление пользователями

### 2.4.4 Модель системы

На основе функциональных требований к системе и разделение ее на предметные области[11], спроектируем модель системы.

В соответствии с предметными областями, разобьем систему на модули: модуль аутентификации, модуль управления врачей и пациентов, модуль управления УЗИ снимков, модуль интеллектуальной части.

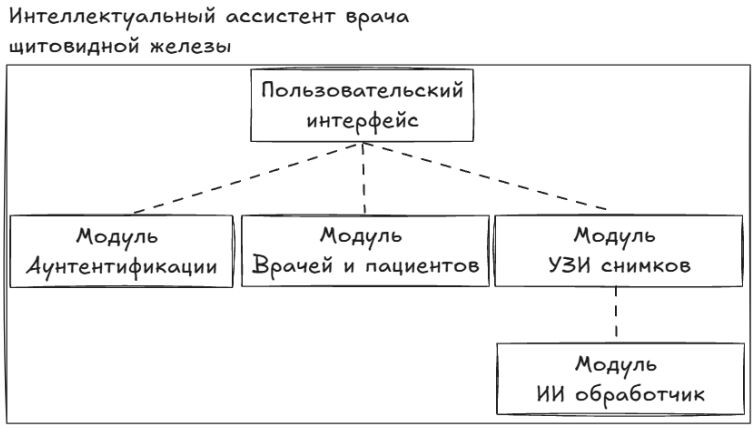


Рисунок 2.8 – Модель системы

## 2.5 Обощенная модель компонента системы

Каждый компонент системы проектируется с учетом подхода Clean Architecture, разбиваясь на слои. Так как система разбита на компоненты по предметным областям, каждый компонент получается достаточно изолированным поэтому не требует нарушения

принципов разбиения компонента на слои для реализации. ветствовать приведенной.

Обощенная модель компонента системы будет представлять собой модель типового микросервиса.

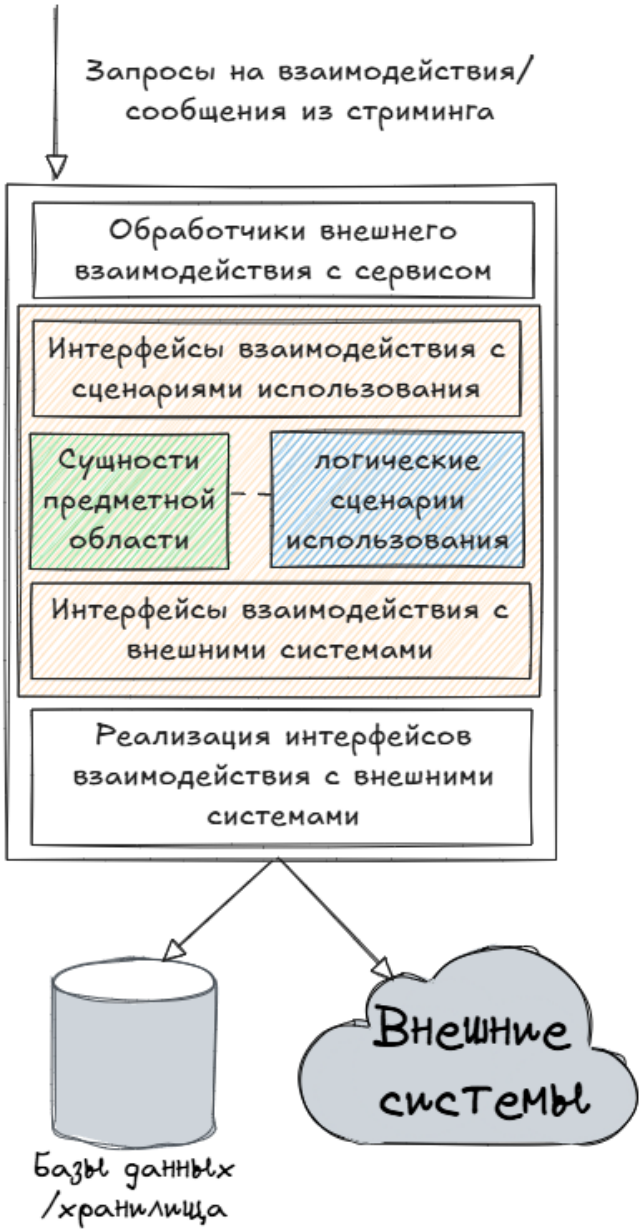


Рисунок 2.9 – Модель компонента системы

## 2.6 Выводы

1. Разработаны функциональные требования к системе, на основании их, произведенно разбиение системы на компоненты.
2. Разработана модель системы, представляющее объединение компонентов системы
3. Разработаны функциональные требования и модели каждого компонента в рамках

взаимодействия в системе.

4. На основе правил и принципов чистой архитектуры, разработана обобщенная модель компонента

# 3. Результаты проектирования системы Интеллектуального ассистента врача щитовидной железы

## 3.1 Разработка микросервисной архитектуры системы

### 3.1.1 Паттерн API Composition

Каждому спроектированному компоненту системы соответствует свой микросервис. Общение между микросервисами осуществляется синхронным и асинхронным способом. Однако в распределенных системах следует избегать лишних зависимостей между микросервисами, это может повлеч за собой сложности в масштабировании и обслуживании системы, проблемы с каскадными ретраями, и в целом делает изолированный и обособленные части системы зависимыми от других частей системы.

Для удовлетворения требовани о регистрации пользователей системы нам требуется сообщить информацию о новом пользователе в 2 компонента: компоненту авторизации и аутентификации и компоненту управления пользователями.

Для избежания лишней зависимости между компонентами, мы используем паттерн Composition-API. API Composition паттерн — это подход в архитектуре микросервисов, который позволяет выносить логику взаимодействия между микросервисами в отдельный сервис более верхнего уровня.

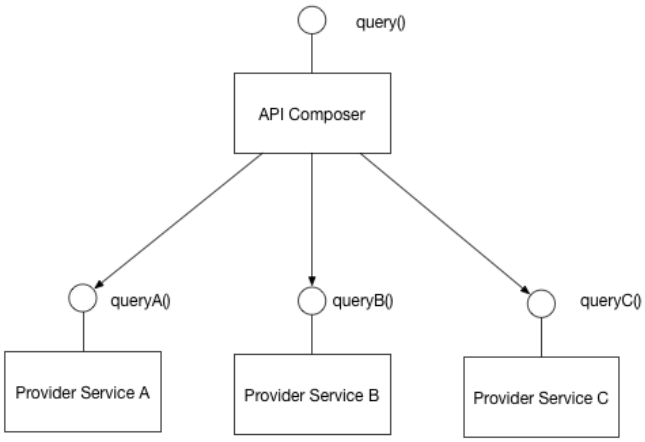


Рисунок 3.1 – API Composition паттерн

### 3.1.2 Способы взаимодействия между микросервисами

Основные запросы которые будут поступать в ИИ ассистент врача, это запросы на получение того или иного ресурса. Данные запросы реализуются посредством обычных синхронных интернет запросов, в API Composition. Далее из API Composition запросы передаются в микросервисы.

Однако стоит рассмотреть сценарий загрузки узи снимков на обработку в систему. Обработка узи снимка занимает значительное для пользователя время. Нужен механизм, который избавит пользователя от необходимости активного ожидания результатов обработки узи снимка.

Решением этой проблемы - во время загрузки узи снимка, система ставит асинхронную задачу на обработку узи снимка. Если задача поставлена успешно, пользователь получает идентификатор задачи, после чего может посредством отдельных запросов узнать статус задачи. В данном случае ожидание пользователя будет сведено к загрузке узи снимка. В таком случае, схема начала обработки узи снимка будет выглядеть следующим образом:

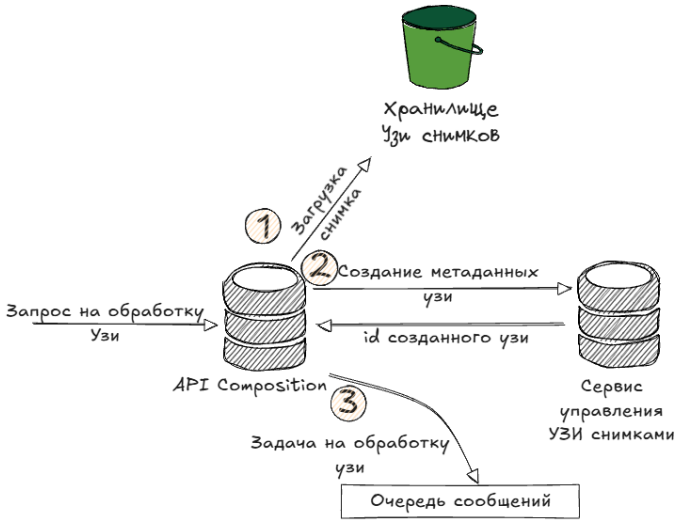


Рисунок 3.2 – Схема асинхронной обработки узи снимка

### 3.1.3 Итоговая архитектура системы Интеллектуального ассистента врача щитовидной железы

Применяя паттерны API Composition при проектировании, а также учитывая синхронное и асинхронное взаимодействие между микросервисами, мы получаем следующую архитектуру системы:

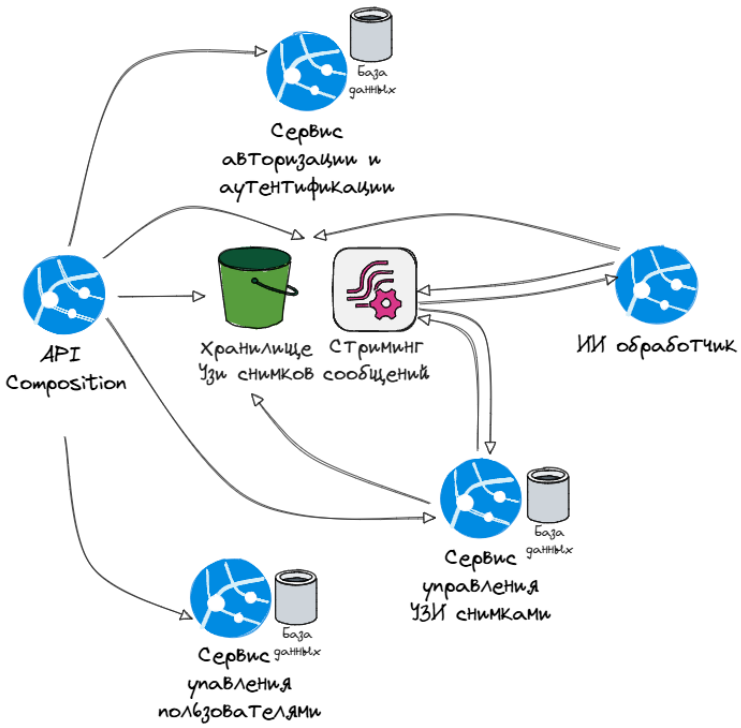


Рисунок 3.3 – Итоговая архитектура микросервисной системы

## 3.2 Выбор технологий и инструментов для технической реализации системы

### 3.2.1 Языка программирования

#### Философия и дизайн языков

**Python** следует философии простоты и читаемости кода, воплощённой в принципе *"Zen of Python"*. Язык спроектирован так, чтобы код был максимально понятным и выразительным, что делает его идеальным для быстрого прототипирования и обучения программированию.

**Java** основан на принципах объектно-ориентированного программирования и философии *"write once, run anywhere"*. Язык обеспечивает строгую типизацию и надёжность выполнения программ за счёт виртуальной машины Java (JVM).

**Go** создавался с целью объединить простоту синтаксиса с высокой производительностью. Разработчики Google стремились создать язык, который был бы эффективен для создания масштабируемых сетевых приложений и системного программирования.

**Производительность и скорость выполнения**

По скорости выполнения языки располагаются в следующем порядке:

1. **Go** — компилируемый язык, обеспечивающий производительность, близкую к C/C++
2. **Java** — благодаря JIT-компиляции и оптимизациям JVM показывает высокую производительность
3. **Python** — интерпретируемый язык с относительно низкой скоростью выполнения

Однако важно отметить, что Python компенсирует низкую скорость выполнения возможностью использования оптимизированных библиотек, написанных на C/C++, таких как NumPy и Pandas.

**Синтаксис и удобство разработки**

**Python** обладает наиболее лаконичным и интуитивным синтаксисом. Отсутствие точек с запятой, использование отступов для обозначения блоков кода и богатая стандартная библиотека делают разработку быстрой и приятной.

**Java** требует больше *boilerplate*-кода, но обеспечивает чёткую структуру и типобезопасность. Современные версии Java (начиная с Java 8) значительно упростили синтаксис благодаря лямбда-выражениям и другим нововведениям.

**Go** сочетает простоту синтаксиса с мощными возможностями. Язык имеет минималистичный дизайн, но при этом включает встроенную поддержку параллельного программирования через горутины и каналы.

**Экосистема и библиотеки**

**Python** обладает самой обширной экосистемой для задач машинного обучения, науки о данных и веб-разработки. PyPI содержит сотни тысяч пакетов практически для любых задач.

**Java** имеет зрелую экосистему корпоративной разработки с такими фреймворками, как Spring, Hibernate, Apache Commons. Maven и Gradle обеспечивают надёжное управление зависимостями.

**Go** обладает растущей, но пока более ограниченной экосистемой. Встроенный пакетный менеджер и стандартная библиотека покрывают большинство базовых потребностей, особенно для сетевого и системного программирования.

**Области применения**

**Python**: машинное обучение, анализ данных, веб-разработка (Django, Flask), автоматизация, скриптинг

**Java**: корпоративные приложения, Android-разработка, веб-сервисы, большие распределённые системы

**Go**: микросервисы, сетевые приложения, DevOps-инструменты, системное программирование

**Параллельное программирование**

**Go** предоставляет наиболее элегантную модель параллельного программирования с горутинами и каналами, основанную на принципах CSP (Communicating Sequential Processes).

**Java** использует традиционную модель потоков с возможностями из пакета `java.util.concur` а также современные решения вроде Project Loom.

**Python** ограничен Global Interpreter Lock (GIL), что затрудняет истинное многопоточное выполнение, хотя существуют способы обхода через multiprocessing и asyncio.

**Заключение**

Учитывая наши требования и специфику приложений, мы выбираем Golang как язык программирования для ассистента.

### 3.2.2 Реляционные хранилища данных

**Реляционные базы данных**: Реляционные базы данных, такие как MySQL и PostgreSQL, предлагают структурированный способ хранения данных с использованием таблиц, что позволяет легко реализовать связи между ними. Они подходят для большинства приложений, требующих согласованности и целостности данных.

**Общая характеристика**

PostgreSQL и MySQL являются двумя наиболее популярными системами управления базами данных с открытым исходным кодом. Каждая из них имеет свои особенности и области применения.

**Основные различия**

**Архитектура и подход** PostgreSQL представляет собой объектно-реляционную СУБД, следующую стандартам SQL и поддерживающую сложные типы данных. MySQL изна-

чально создавалась как быстрая и простая реляционная СУБД.

**Производительность** MySQL традиционно показывает лучшие результаты в простых операциях чтения и веб-приложениях. PostgreSQL превосходит в сложных запросах, аналитике и операциях записи с высокой конкуренцией.

**Функциональность** PostgreSQL предлагает расширенные возможности: поддержку JSON, массивов, пользовательских типов данных, оконных функций и процедурных языков. MySQL имеет более простой набор функций, но достаточный для большинства веб-приложений.

**Области применения**

*PostgreSQL* оптимален для:

- Сложных аналитических систем
- Приложений с высокими требованиями к целостности данных
- Проектов, требующих расширенной функциональности SQL

*MySQL* предпочтителен для:

- Веб-приложений и CMS
- Проектов с простыми запросами и высокой нагрузкой на чтение
- Систем, где важна простота администрирования

**Заключение**

Учитывая наши требования и специфику приложений, мы выбираем PostgreSQL как основное решение для хранения данных в нашей архитектуре.

### 3.2.3 Хранилище УЗИ снимков

В современном мире информационных технологий выбор подходящей системы хранения данных становится критически важным решением. Каждый тип хранилища имеет свои особенности, преимущества и области применения. Рассмотрим три основных подхода к организации хранения данных.

**Объектное хранилище**

Объектное хранилище представляет собой архитектуру, где данные хранятся в виде объектов в плоском пространстве имен. Каждый объект содержит сами данные, метаданные и уникальный идентификатор. **Ключевые особенности**:

- Масштабируемость до петабайтов данных
- REST API для доступа к данным

- Отсутствие традиционной файловой иерархии
- Встроенная репликация и обеспечение целостности данных

**Преимущества**:

- Практически неограниченная масштабируемость
- Высокая надежность благодаря распределенной архитектуре
- Экономическая эффективность для больших объемов данных
- Идеально подходит для веб-приложений и API

**Недостатки**:

- Невозможность модификации объектов (только замена)
- Более высокая латентность по сравнению с локальными системами
- Ограниченная поддержка POSIX-операций

**Примеры использования**: Amazon S3, облачные сервисы, резервное копирование, хранение мультимедиа контента.

**Файловое хранилище**

Традиционные файловые системы организуют данные в виде иерархической структуры папок и файлов. Это наиболее привычный и широко используемый подход к хранению данных. **Ключевые особенности**:

- Иерархическая структура каталогов
- Прямой доступ к файлам через файловую систему
- Поддержка стандартных операций чтения/записи
- Интеграция с операционными системами

**Преимущества**:

- Простота использования и понимания
- Быстрый доступ к данным
- Полная совместимость с существующими приложениями
- Поддержка всех стандартных файловых операций

**Недостатки**:

- Ограниченная масштабируемость
- Уязвимость к отказам оборудования
- Сложность резервного копирования больших объемов данных
- Производительность снижается при работе с миллионами файлов

**Примеры использования**: Локальные диски, NAS-системы, файловые серверы предприятий.

### Распределенные файловые системы

Распределенные файловые системы объединяют преимущества традиционного файлового хранилища с возможностями распределенной архитектуры, обеспечивая масштабируемость и отказоустойчивость. **Ключевые особенности**:

- Данные распределены по множеству узлов
- Прозрачный доступ к файлам как к локальным
- Автоматическая репликация и восстановление
- Горизонтальное масштабирование

**Преимущества**:

- Высокая доступность и отказоустойчивость
- Масштабируемость путем добавления новых узлов
- Сохранение привычного файлового интерфейса
- Автоматическое распределение нагрузки

**Недостатки**:

- Сложность настройки и администрирования
- Потенциальные проблемы с консистентностью данных
- Зависимость от сетевой инфраструктуры
- Более высокая стоимость внедрения

**Примеры использования**: Hadoop HDFS, GlusterFS, Ceph, высоконагруженные системы, big data аналитика.

### Сравнительная таблица

| **Критерий** | **Объектное** | **Файловое** | **Распределенные** |
|---|---|---|---|
| Масштабируемость | Очень высокая | Малая | Высокая |
| Производительность | Средняя | Высокая | Высокая |
| Надежность | Очень высокая | Низкая | Высокая |
| Простота использования | Средняя | Высокая | Низкая |
| Стоимость | Низкая | Средняя | Высокая |

Таблица 3.1 – Сравнение нереалиционных баз данных

**Выбор подходящего решения**

Выбор типа хранилища зависит от конкретных требований: **Объектное хранилище** идеально для веб-приложений, резервного копирования и архивирования данных, где требуется высокая масштабируемость при умеренных требованиях к производительности. **Файловое хранилище** остается оптимальным выбором для локальных приложений, небольших и средних объемов данных, где важна простота и производительность. **Распределенные файловые системы** подходят для корпоративных решений с высокими требованиями к доступности и необходимостью обработки больших объемов данных с сохранением файлового интерфейса. Современные IT-инфраструктуры часто используют гибридный подход, сочетая различные типы хранилищ в зависимости от специфики задач и требований к данным.

### 3.2.4 Синхронное взаимодействие между микросервисами

Микросервисы могут синхронно общаться между собой с использованием различных технологий и протоколов. В этом контексте рассмотрим четыре популярных метода взаимодействия: HTTP/REST API, gRPC, GraphQL и WebSocket.

В большинстве случаев для микросервисов, которые обмениваются фиксированными данными, использование GraphQL может быть излишним, поскольку этот подход предназначен для динамического запроса данных. В сценариях, где структуры данных заранее известны и не требуют изменения, REST API будет более простым и понятным решением. То же касается и WebSocket: двунаправленный стриминг данных может быть избыточным для многих бизнес-приложений, где достаточно стандартного запроса и ответа.

Таким образом, основные методы, которые стоит рассмотреть для синхронного взаимодействия микросервисов, — это HTTP/REST API и gRPC.

**HTTP/REST API** HTTP/REST API — это один из самых распространенных способов синхронного взаимодействия между микросервисами. Каждый микросервис предоставляет набор эндпоинтов, к которым другие сервисы могут обращаться для выполнения операций и получения данных. Используя стандартные методы HTTP (GET, POST, PUT, DELETE), микросервисы обмениваются сообщениями с четко определенными правилами и структурой.

**Преимущества использования HTTP/REST**

– **Простота**: REST API легко реализовать и документировать. Он основан на понятных

стандартах HTTP и может использоваться практически на любой платформе.

- **Читаемость**: Структура URL и использование HTTP-методов делают интерфейс API интуитивно понятным и удобным для работы разработчиков.
- **Совместимость**: REST API может легко взаимодействовать с разными языками программирования и платформами, что делает его универсальным решением для микросервисной архитектуры.

**gRPC** gRPC, разработанный Google, представляет собой высокопроизводительный фреймворк для удаленных вызовов процедур (RPC), который поддерживает множество языков программирования. Он использует HTTP/2 для передачи данных, что обеспечивает преимущества, такие как многопоточность и меньшие задержки при передаче информации.

**Преимущества использования gRPC**

- **Производительность**: Использование HTTP/2 позволяет gRPC эффективно обрабатывать множество параллельных запросов, что делает его более производительным, чем традиционные REST API.
- **Статическая типизация**: gRPC использует Protocol Buffers для описания структуры данных, что позволяет разработчикам строго определять, какой тип данных будет передаваться. Это обеспечивает дополнительную безопасность и позволяет избежать ошибок при взаимодействии между сервисами.
- **Автогенерация кода**: Удобство разработки достигается благодаря автоматической генерации клиентского кода на разных языках, что ускоряет процесс создания микросервисов.

**Сравнение HTTP/REST API и gRPC**

| **Характеристика** | **HTTP/REST API** | **gRPC** |
|---|---|---|
| Протокол | HTTP/1.1 | HTTP/2 |
| Структура данных | JSON/XML | Protobuff |
| Типизация | Динамическая | Статическая |
| Производительность | Низкая | Высокая |
| Поддержка потоковой передачи | Ограниченная | Полная |
| Автогенерация клиента | Нет | Да |

Таблица 3.2 – Сравнение HTTP/REST API и gRPC

**Заключение** Нашей системе не требуется возможность динамической типизации, поэтому выбор сделать в сторону gRPC.

### 3.2.5 Асинхронное взаимодействие

Асинхронное взаимодействие между микросервисами позволяет увеличить производительность и гибкость распределенных систем. В этом подходе микросервисы могут обмениваться данными без необходимости дожидаться ответа, что снижает задержки и повышает общую эффективность системы. Рассмотрим основные методы асинхронного взаимодействия.

**Основные методы асинхронного взаимодействия**

- **Очереди сообщений**: Использование систем обмена сообщениями, таких как RabbitMQ, Apache Kafka или Redpanda, позволяет отправлять сообщения между микросервисами без прямого связывания. Один сервис может отправлять сообщения в очередь, а другой — извлекать их и обрабатывать по мере возможности. Это гарантирует, что сервисы могут работать независимо друг от друга и не блокируют друг друга в случае высокой нагрузки.
- **Событийно-ориентированная архитектура**: В этой архитектуре микросервисы реагируют на события, происходящие в системе. События могут генерироваться различными компонентами и служить сигналами для других микросервисов о том, что произошло что-то важное (например, изменение состояния, завершение задачи и т. д.). Это позволяет строить более гибкие и масштабируемые системы.
- **HTTP-события (Webhooks)**: Использование вебхуков позволяет микросервису отправлять HTTP-запросы в другие сервисы при наступлении определённых событий. Это простой способ интеграции, позволяющий уведомлять другие службы о произошедших изменениях или событиях.

Выбор в сторону очередей сообщений также обуславливается необходимостью наличия механизма Dead Letter Queue (DLQ), который позволяет обрабатывать ошибки при отправке и получении сообщений. Кроме того, мы стремимся автоматизировать хранение сообщений, что делает использование очередей сообщений предпочтительным решением.

**Описание систем очередей сообщений**

- **Apache Kafka**: - Kafka — это распределенная платформа для потоковой передачи данных, которая обеспечивает высокую пропускную способность и низкую задержку. Она работает по принципу публикации и подписки, позволяя множеству клиентов читать и писать сообщения. - **Назначение**: Идеально подходит для обработки потоков данных в реальном времени и хранения больших объемов событий с воз-

можностью их долговременного хранения.

- **Redpanda**: - Redpanda — это высокопроизводительная, совместимая с Kafka, распределенная система сообщений, оптимизированная для работы с потоками данных в реальном времени. - **Назначение**: Обеспечивает низкую задержку и простоту настройки, что делает её предпочтительной для решений, требующих высокой производительности.
- **Apache Pulsar**: - Pulsar — это распределенная система потоковой передачи данных, которая поддерживает многопоточность и управление потоками. Она предоставляет гибкий подход к очередям сообщений и событиям. - **Назначение**: Отличается поддержкой долгосрочного хранения сообщений и сложных сценариев работы с георасределёнными данными.

**Заключение** Асинхронное взаимодействие, организованное через очереди сообщений, является эффективным способом повышения производительности и устойчивости микросервисов. Учитывая возможности по снижению задержки и простой настройке, мы выбрали Redpanda как предпочтительное решение для организации асинхронного взаимодействия между микросервисами.

## 3.3 Выводы

1. Разработана оптимизированная микросервисная архитектура с применением патерна API Composition, минимизирующего зависимости между компонентами системы.
2. Сформирован технологический стек реализации: Golang (основной язык), PostgreSQL (реляционное хранилище), объектное хранилище minio S3 (для УЗИ-данных), gRPC (синхронное взаимодействие) и Redpanda (асинхронные очереди)
3. Реализованы специализированные механизмы взаимодействия, включая асинхронную модель обработки УЗИ-снимков с возвратом идентификатора задачи и поддержкой Dead Letter Queue для устойчивой работы

# 4. Реализация и тестирование системы

## 4.1 Реализация компонентов системы в виде микросервисов

### 4.1.1 Сущности предметной области

В каждом микросервисе существует своя предметная область. Все ключевые сущности заданы в виде структур, для оперирования ими в логике микросервиса.

node.go

uzi > internal > domain > node.go

```go
package domain

import "github.com/google/uuid"

type Node struct {
    Id          uuid.UUID
    Ai          bool
    UziID       uuid.UUID
    Validation  *NodeValidation
    Tirads23    float64
    Tirads4     float64
    Tirads5     float64
    Description *string
}

```

Рисунок 4.1 – Сущности узла образования в щитовидной железе


Рисунок 4.2 – Статусы узла образования в щитовидной железе

Сущности предметной области не имеют зависимостей от внешних частей или структур системы.

### 4.1.2 Логические сценарии использования

Вся логика которая имеется в микросервисе реализована в этом слое приложения. Никакая логика не допустима в другом месте. Такой подход к разработке позволяет легко отлаживать и контролировать систему, она не «растякается» по всему коду.

```go
func (s *service) DeleteNode(ctx context.Context, id uuid.UUID) error {
    node, err := s.dao.NewNodeQuery(ctx).GetNodeByID(id)
    if err != nil {
        return fmt.Errorf("get node by id: %w", err)
    }
    if node.Ai {
        return ErrChangeAiNode
    }

    ctx, err = s.dao.BeginTx(ctx)
    if err != nil {
        return fmt.Errorf("begin transaction: %w", err)
    }
    defer func() { _ = s.dao.RollbackTx(ctx) }()

    if err := s.dao.NewSegmentQuery(ctx).DeleteSegmentsByUziID(id); err != nil {
        return fmt.Errorf("delete node segments: %w", err)
    }

    if err := s.dao.NewNodeQuery(ctx).DeleteNodeByID(id); err != nil {
        return fmt.Errorf("delete node: %w", err)
    }

    if err := s.dao.CommitTx(ctx); err != nil {
        return fmt.Errorf("commit transaction: %w", err)
    }

    return nil
}
```

Рисунок 4.3 – Реализация удаления узла образования

### 4.1.3 Интерфейсы взаимодействия с внешними системами

Для сохранения правила Dependency Inversion, все взаимодействия с внешними системами реализованы через интерфейсы, чей уровень абстракции такого же уровня, что и уровень абстракции вызывающего интерфейс. Рассмотрим пример интерфейса для взаимодействия с базой данных.

```go
type Repository interface {
    InsertNodes(nodes ...entity.Node) error

    GetNodeByID(id uuid.UUID) (entity.Node, error)
    GetNodesByImageID(id uuid.UUID) ([]entity.Node, error)
    GetNodesByUziID(id uuid.UUID) ([]entity.Node, error)

    UpdateNode(node entity.Node) error

    DeleteNodeByID(id uuid.UUID) error
}
```

Рисунок 4.4 – Интерфейс взаимодействия с базой данных для узлов образований

### 4.1.4 Контракты микросервисов и API системы

Все микросервисы взаимодействуют с друг другом посредством gRPC. Описания контрактов взаимодействия делается посредством proto файлов. В них описываются rpc методы микросервиса, которые он реализует. Описаны ожидаемые структуры данных, которые используются в rpc методах.

```
service UziSrv {
  // DEVICE
  rpc createDevice(createDeviceIn) returns (createDeviceOut);
  rpc getDeviceList(google.protobuf.Empty) returns (GetDeviceListOut);

  // UZI
  rpc createUzi(CreateUziIn) returns (CreateUziOut);
  rpc getUziById(GetUziByIdIn) returns (GetUziByIdOut);
  rpc getUzisByExternalId(GetUzisByExternalIdIn) returns (GetUzisByExternalIdOut);
  rpc getUzisByAuthor(GetUzisByAuthorIn) returns (GetUzisByAuthorOut);
  rpc getEchographicByUziId(GetEchographicByUziIdIn) returns (GetEchographicByUziIdOut);
  rpc updateUzi(UpdateUziIn) returns (UpdateUziOut);
  rpc updateEchographic(UpdateEchographicIn) returns (UpdateEchographicOut);
  rpc deleteUzi(DeleteUziIn) returns (google.protobuf.Empty);
```

Рисунок 4.5 – Контракт взаимодействия с микросервисом управления УЗИ снимками

```
message CreateUziIn {
  UziProjection projection = 100;
  string external_id = 200;
  string author = 300;
  int64 device_id = 400;
  optional string description = 500;
}

message CreateUziOut { string id = 100; }
```

Рисунок 4.6 – Элементы контракта взаимодействия с микросервисом управления УЗИ снимками

Далее рассмотрим API системы в целом. Этот API должен в полной мере покрывать все функциональные требования к системе, относящиеся к взаимодействию с системой. API системы описывается файлом в формате OpenAPI 3.0.

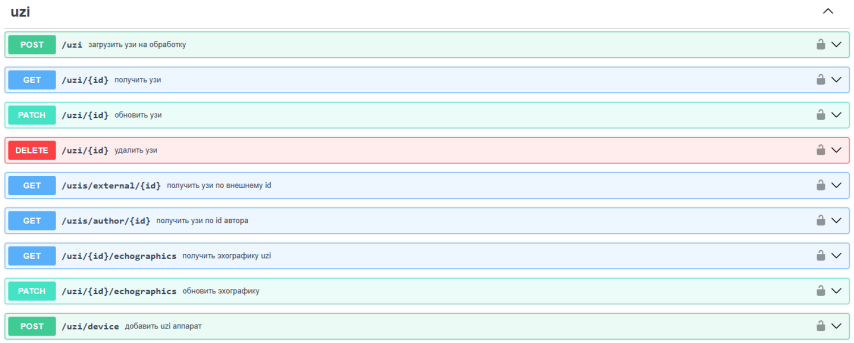


Рисунок 4.7 – Часть API всей системы

### 4.1.5 Топики стриминга сообщений для асинхронного общения

Для использования асинхронного общения через RedPanda, создадим 3 топика:

- **uziupload** - узи загруженно в S3. Запустит разбиение узи на кадры.
- **uzisplitted** - узи разбито на кадры. Запустит обработку узи нейро моделью. В топик пишет сервис управления УЗИ после разбиения узи на кадры.
- **uziprocessed** - узи обработанно нейромоделью. Узи сегментировано и классифицировано

### 4.1.6 Структура хранилища узи снимков

Структура S3 minio представлена в виде файловой системы, позволяет быстро искать необходимые файлы, зная их идентификатор. Микросервисы знают об устройстве S3, поэтому при необходимости передчи в другой микросервис узи снимков, достаточно передавать id снимка.

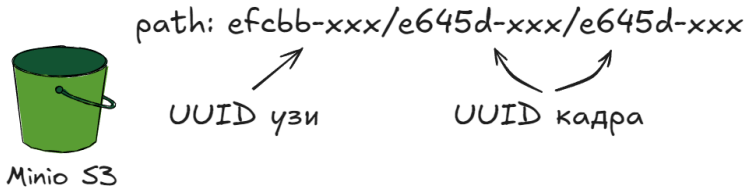


Рисунок 4.8 – Структура S3 хранилища

## 4.2 Состав и структура реализованного программного обеспечения

Реализованно серверное приложение, запускаемое как для локального использование, так и в серверном виде. Приложений отвечает на HTTP запросы, соответствует всем функциональным требованиям к системе.

### 4.2.1 Структура приложения

Приложение состоит из 5 микросервисов:

- Composition API - точка входа в систему. Обрабатывает запросы от пользователя и передает их в соответствующие микросервисы.
- Микросервис Авторизации и Аутентификации - отвечает за аутентификацию и авторизацию пользователей.
- Микросервис Управления УЗИ снимками - отвечает за загрузку, разбиение на кадры, обработку узи нейромоделью.
- Микросервис Управления Медицинскими данными - отвечает за хранение и обработку медицинских данных.
- Микросервис Интеллектуальной части - отвечает за обработку узи нейромоделью.

Каждый микросервис написанный на golang содержит:

- go.mod и go.sum файлы, описывающие библиотекчные зависимости для приложения
- sql файлы миграций. Сами миграции осуществлялись за счет утилиты Goose. SQL файлы в каждом микросервисе расположены по пути db/migrations
- proto файлы описывающие контракты для взаимодествия с внешними системами
- taskfile - файлы описывающие команды необходимые для локальной сборки микросервиса, миграции базы данных, генерации кода и т.д
- Dockerfile - файл для сборки docker образа микросервиса
- service.yml - файл содержащий конфигурацию сервиса

## 4.3 Основные сценарии работы пользователя

Основной сценарий работы ПО - отправка запросов на обработку узи изображений и полученные данных с обработанных изображений. Для взаимодействием с сервисом предоставляется подробное swagger api.

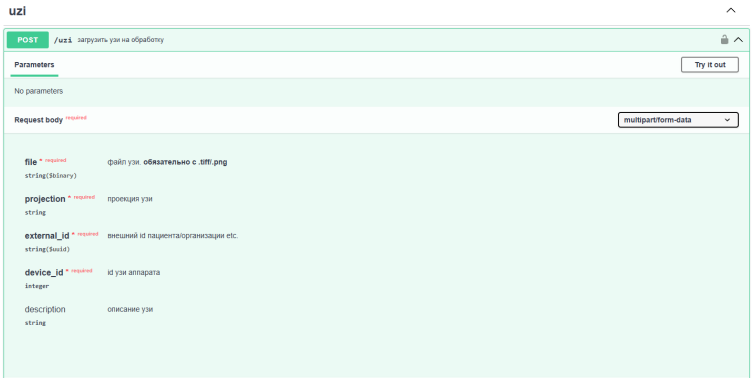


Рисунок 4.9 – Запрос на загрузку и анализ uzi снимка

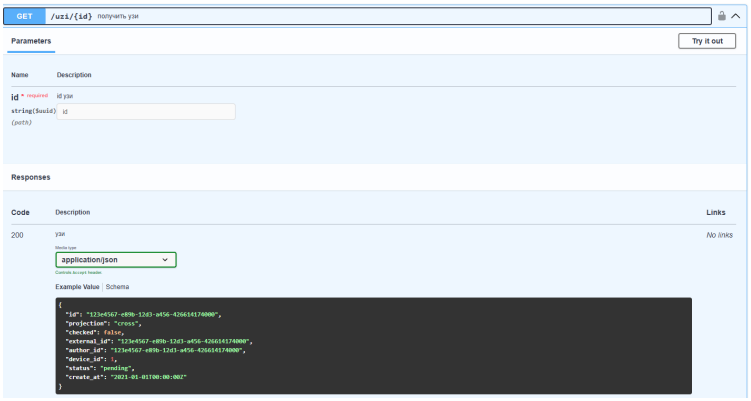


Рисунок 4.10 – Запрос на получение информации по загруженному узи снимку

## 4.4 Тестирование системы

Тестирование программного обеспечения является критически важным этапом жизненного цикла разработки, обеспечивающим качество, надежность и соответствие требованиям готового продукта. Существует множество подходов к тестированию, каждый из которых решает определенные задачи и применяется на различных этапах разработки.

### 4.4.1 Юнит тестирование

Юнит тестирование представляет собой тестирование отдельных компонентов или модулей программы в изоляции от других частей системы. Этот вид тестирования обычно выполняется разработчиками и направлен на проверку корректности работы небольших участков кода, таких как функции, методы или классы.

Основные преимущества юнит тестирования включают раннее обнаружение дефектов, упрощение отладки и рефакторинга, а также создание документации к коду в виде тесто-

вых сценариев. Для автоматизации модульного тестирования используются специальные фреймворки, такие как JUnit для Java, pytest для Python, или Jest для JavaScript.

Для модульного тестирования использовалась библиотека testing из стандартной библиотеки golang. Пакет testing предоставляет необходимые инструменты для написания и выполнения тестов. Он интегрирован с утилитой go test, которая автоматически обнаруживает тесты в файлах с суффиксом _test.go и выполняет их. Благодаря гибкой системе отчетов можно логировать дополнительную информацию, пропускать тесты при определенных условиях или запускать их параллельно для ускорения проверки. С помощью флага -cover, можно выявить непротестированные участки.

Так же для проверки значений результатов использовалась библиотека testify. Она позволяет проверять значения результатов, такие как числа, строки, булевые значения, структуры данных и т.д. Помимо проверки значений результатов, testify добавляет функционал мок объектов. Мок объекты - позволяют задать поведения заранее определенного интерфейса. С помощью мок объекта, можно проверить что во время теста, у интерфейса вызывается определенный метод с определенными параметрами. Помимо этого, мок объекту можно задать его поведение при ответе, когда вызывается описанный метод.

Пример юнит тестирования в проекте:

```go
package slicer

func Flatten2DArray[T any](slice [][]T) []T {
    cnt := 0
    for _, v := range slice {
        cnt += len(v)
    }

    res := make([]T, 0, cnt)
    for _, v := range slice {
        res = append(res, v...)
    }

    return res
}
```

Рисунок 4.11 – Функция для для перевода 2-ух мерных массивов в 1-мерный

Данная функция используется при разбиениее кинопетли на кадры. Библиотека для работы с tiff форматом, будет использовать двумерный массив, который нам нужно будет перевести в 1-мерный.

Тесты для данной функции:

```go
func TestFlatten2DArray_Success_DataSet(t *testing.T) {
    t.Parallel()

    tests := []struct {
        name     string
        input    [][]int
        expected []int
    }{
        {
            name:     "empty 2D array",
            input:    [][]int{},
            expected: []int{},
        },
        {
            name:     "single empty row",
            input:    [][]int{{}},
            expected: []int{},
        },
        {
            name:     "multiple rows",
            input:    [][]int{{1, 2}, {3, 4}, {5, 6}},
            expected: []int{1, 2, 3, 4, 5, 6},
        },
        {
            name:     "mixed lengths",
            input:    [][]int{{1}, {2, 3}, {}, {4, 5, 6}},
            expected: []int{1, 2, 3, 4, 5, 6},
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()

            result := slicer.Flatten2DArray(tt.input)
            assert.Equal(t, tt.expected, result)
        })
    }
}
```

Рисунок 4.12 – Юнит тесты на функцию Flatten2DArray

Данный тест используется методику ”Табличных тестов”. Испольуется массив анонимных структур, в которых описываются ряд сценариев для тестирования.

### 4.4.2 Интеграционное тестирование

Интеграционное тестирование представляет собой тестирование взаимодействия между различными компонентами или модулями программы. Этот вид тестирования обычно выполняется разработчиками и направлен на проверку корректности взаимодействия между различными частями системы.

Основные преимущества интеграционного тестирования включают раннее обнаружение дефектов, упрощение отладки и рефакторинга, а также создание документации к коду в виде тестовых сценариев.

Основным сценарием интеграционного тестирования, является тестирования взаимодействия микросервиса с базой данных или брокером сообщений.
Взаимодействие с базой данных происходит в слое Репозитория. От основной логики приложения он отделен интерфейсом.

```
type Repository interface {
    InsertNodes(nodes ...entity.Node) error

    GetNodeByID(id uuid.UUID) (entity.Node, error)
    GetNodesByImageID(id uuid.UUID) ([]entity.Node, error)
    GetNodesByUziID(id uuid.UUID) ([]entity.Node, error)

    UpdateNode(node entity.Node) error

    DeleteNodeByID(id uuid.UUID) error
}
```

Рисунок 4.13 – Интерфейс взаимодействия с базой данных для узлов образований

Реализация интерфейса непосредственно взаимодействует с базой данных:

```go
func (q *repo) UpdateNode(node entity.Node) error {
    query := q.QueryBuilder().
        Update(table).
        SetMap(sq.Eq{
            columnValidation: node.Validation,
            columnTirads23:   node.Tirads23,
            columnTirads4:    node.Tirads4,
            columnTirads5:    node.Tirads5,
        }).
        Where(sq.Eq{
            columnID: node.Id,
        })

    _, err := q.Runner().Execx(q.Context(), query)
    if err != nil {
        return err
    }

    return nil
}
```

Рисунок 4.14 – Интерфейс взаимодействия с базой данных для обновления узла образований

Тесты на слой взаимодействия с базой данных, прямым образом проверяют корректность работы сервиса с базой данных, так как более нигде кроме слоя репозитория, сервис с базой данных не взаимодействует.

```go
func TestNodeRepository_UpdateNode_DBError(t *testing.T) {
    t.Parallel()
    dao, dbClose := utils.SetUpPostgres(t)
    defer dbClose()

    ctx := context.Background()
    repo := node.NewRepo(ctx, dao)

    dbClose()
    _, err := repo.UpdateNode(0, entity.Node{})
    require.Error(t, err)
}
```

Рисунок 4.15 – Тесты проверяющий отказ базы данных при запросе

## 4.5 Сквозное тестирование

Сквозное тестирование представляет собой тестирование взаимодействия между различными компонентами или модулями программы. Этот вид тестирования обычно вы-

полняется разработчиками и направлен на проверку корректности взаимодействия между различными частями системы.

Основные преимущества сквозного тестирования включают раннее обнаружение дефектов, упрощение отладки и рефакторинга, а также создание документации к коду в виде тестовых сценариев.

Так как каждый микросервис предоставляет свой контракт взаимодействия, то для хорошего покрытия сквозного тестирования, требуется тест на каждую ручку. Рассмотрим пример сквозного тестирования на примере микросервиса управления УЗИ снимками. Микросервис предоставляет контракт, позволяющий ставить узи на обработку, а затем получаеть результаты обработки в различных разрезах.
Обработку узи снимка можно представить в следующем виде:

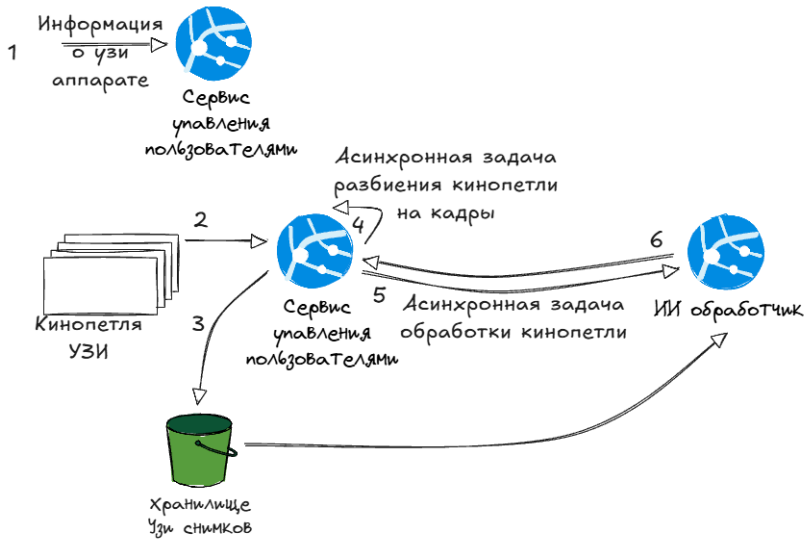


Рисунок 4.16 – Схема обработки узи снимка

Разобьем на следующие этапы:

- Загрузка информации о узи аппарате
- Загрузка узи снимка в S3
- Разбиение узи снимка на кадры
- Обработка узи снимка нейромоделью
- Получение результатов обработки узи снимка

Каждая группа ручек из контракта проверяет определенный набор этапов. Мы не можем проводить тесты на системе с уже обработанным снимком, так как это не будет отра-

жать реальное поведение системы в использовании. Поэтому для каждого теста, нужно выполнения соответствующих ряда этапов. Для примера возьмем ручки, проверяющие разбиение узи снимка на кадры. Для этих тестов обработка нейромоделью узи не требуется, так как проверяться не будет, но оно увеличит время проведения тестов.

Для решения данной задачи применем паттерн ”Цепочка ответственности”. В нем каждый этап выделяется в отдельную состовляющую, имеющую зависимости от остальных этапов, если такие присутствуют. Использование данного подхода, позволит индивидуально или группе тестов указывать необходимые для них этапы, автоматизированно выполнять их, собирая информацию о поведении системы, и не выполнять лишние этапы, нагружая систему лишним трафиком.

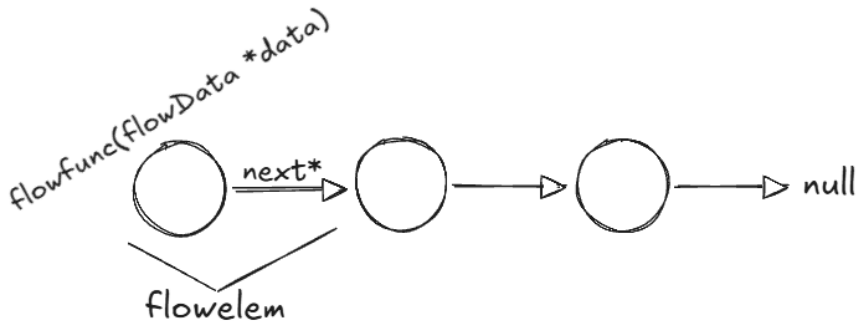


Рисунок 4.17 – Цепочка ответственности для сквозного тестирования

Пример сквозного тестирования для разбиения узи снимка на кадры:

```go
//go:build e2e

package image_test

import (
    "github.com/stretchr/testify/require"

    pb "uzi/internal/generated/grpc/service"
    "uzi/tests/e2e/flow"
)

func (suite *TestSuite) TestGetImagesByUziId_Success() {
    data, err := flow.New(suite.deps, flow.DeviceInit, flow.UziInit, flow.TiffSplit).Do(suite.T().Context())
    require.NoError(suite.T(), err)

    getResp, err := suite.deps.Adapter.GetImagesByUziId(
        suite.T().Context(),
        &pb.GetImagesByUziIdIn{UziId: data.Uzi.Id.String()},
    )
    require.NoError(suite.T(), err)
    require.Equal(suite.T(), len(data.Images), len(getResp.Images))
    for i, image := range getResp.Images {
        require.Equal(suite.T(), data.Images[i].Id.String(), image.Id)
        require.Equal(suite.T(), data.Uzi.Id.String(), image.UziId)
        require.Equal(suite.T(), int64(data.Images[i].Page), image.Page)
    }
}
```

Рисунок 4.18 – Сквозное тестирование для разбиения узи снимка на кадры

## 4.6 Выводы

В результате проведенного анализа, моделирования и проектирования реализованна система ”Интеллектуальный ассистент врача”. Примененные для реализации интсрументы и технологии соответствуют выбранным в прошлой главе.

Серверное приложение состоит из 4 микросервисов связанных с друг другом посредством синхронного и асинхронного общения. Примененный паттерн Composition-Api обеспечивает единую точку входа для взаимодействия с системой. Для использования системы представлено понятное API.

В рамках тестирования, написаны модульные, интеграционные и сквозные тесты. Для полноценного свозного тестирования системы, освновные сценарии использования разбиты на обособленные шаги, реализован механизм воспроизведения определенных шагов для каждого теста в отдельности.

# Заключение

В результате проведенного анализа, моделирования и проектирования реализованна система ”Интеллектуальный ассистент врача”

В аналитической части выполнен анализ архитектурных стилей построения систем, с учетом требований к динамической расширяемости отдельных модулей системы, выбрана микросервисная архитектура. Были проанализированы основные архитектурные паттерны построения приложений. Clean Architecture за счет преобраладания SOLID, разбиения на слои и преимуществ Dependency inversion выбрана как основа написания микросервисов.

В теоретической части была выделенны основные предметные области системы, смоделирована система состоящая из компонентов ответственных за предметные области. Каждый компонент смоделирован в отдельности с учетом схемы базы данных, а также построена модель взаимодействия компонентов меж собой в рамках системы. Модели соответствуют и учитывают все функциональные требования к модулям системы и системе в целом.

В третей главе разработана итоговая архитектура системы, выбраны инструменты и технологии для реализации.

В четвертой главе представлена реализация системы, описаны основные детали реализации, приведены фрагменты кода. Описаны методы и сценарии тестирования системы, разработаны подходы для воспроизводимости сценарием проведения тестирования.

**В рамках ВКР** были проведены анализ проблемной области, моделирование системы, проектирование системы, программная реализация и тестирование серверного приложения «Интеллектуальный ассистент врача УЗИ».