

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»
Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
на тему
**ВЕБ-ПРИЛОЖЕНИЕ С ИНТЕГРАЦИЕЙ AI-
ПОМОЩНИКА ДЛЯ МЕДИЦИНСКОЙ
КЛИНИКИ**

Студент

Пантюшов Егор Игоревич
(подпись, дата)

Пантюшов Егор Игоревич
(ФИО полностью)

Руководитель

Зинкин С.А.
(подпись, дата)

Зинкин С.А.
(фамилия, инициалы)

Нормоконтролёр

Бычков А.С.
(подпись, дата)

Бычков А.С.
(фамилия, инициалы)

Рецензент

Доцент ПензГУ
(должность, место работы) (подпись, дата)

Печерский С.В.
(фамилия, инициалы)

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

Заведующий кафедрой

М.А. Митрохин
(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от

23 июля 2026г. № 30)

Секретарь ГЭК

Е
(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2026

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

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
М.А. Митрохин
« 20 » 09 2024 г.

**ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

1. Студент Пантюшов Егор Игоревич гр. 24ВВИМ1 факультета ВТ

2. Тема работы Веб-приложение с интеграцией AI-помощника для медицинской клиники

Тема утверждена приказом ПГУ № 720/0 от " 20 " 09 20 24 г.

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

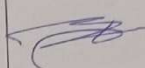
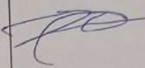
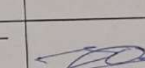
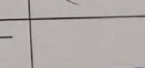
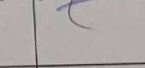
4. Задание на работу (назначение разработки, исходные данные и т.п.)

Выполнить исследование и разработать программный комплекс для медицинской клиники с интеграцией AI-помощника. Веб-приложение должно обеспечивать предоставление информации о клинике, её сотрудниках, медицинских услугах, контактных данных, а также возможность отправки заявки на приём и взаимодействие пользователя с интеллектуальным ассистентом. AI-помощник должен обрабатывать вопросы пользователей, выполнять поиск релевантной информации в базе знаний клиники и формировать справочные ответы с учётом ограничений медицинской предметной области.

5. Перечень подлежащих разработке вопросов

Требуется произвести анализ предметной области веб-приложений для медицинских клиник и определить основные требования к разрабатываемой системе. Провести сравнительный анализ существующих аналогов, выявить их функциональные особенности, преимущества и недостатки. Разработать архитектуру веб-приложения, обеспечивающую взаимодействие клиентской части, серверной части, базы данных и AI-сервиса. Реализовать основные функциональные модули системы. Провести сравнительный анализ больших языковых моделей. По результатам исследования выбрать подходящую для интеграции в AI-сервис. Выполнить тестирование разработанной системы в типовых сценариях взаимодействия пациента с медицинской клиникой и оценить корректность, релевантность и безопасность формируемых ответов.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение предметной области	10%	20.09.24-15.11.2024	
2.	Анализ требований к интеллектуальному помощнику	10%	16.11.2024-20.12.2024	
3.	Изучение литературы и существующих решений	10%	21.12.2024-31.01.2025	
4.	Проектирование архитектуры веб-приложения	15%	01.02.2025-15.03.2025	
5.	Реализация веб-приложения медицинской клиники с AI-помощником	30%	16.03.2025-31.01.2026	
6.	Тестирование разработанной системы	10%	01.02.2026-28.02.2026	
7.	Анализ полученных результатов исследования и выбор наиболее подходящих моделей	10%	01.03.2026-30.04.2026	
8.	Оформление пояснительной записки	5%	01.05.2026-20.05.2026	

Дата выдачи задания " 20 " 09 " 20 24 г.

Руководитель магистерской диссертации

(подпись, дата)

Зинкин С.А.

Задание к исполнению принял студент

(подпись, дата)

Пантюшов Е.И.

Работу к защите допустить
Декан факультета

 17.06.2026

Фионова Л.Р.

РЕФЕРАТ

Магистерская работа содержит 105 листов, 41 рисунок, 1 таблицу, 18 источников, 2 приложения.

ВЕБ-ПРИЛОЖЕНИЕ, МЕДИЦИНСКАЯ КЛИНИКА, AI-ПОМОЩНИК, ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, LLM, RAG, БАЗА ЗНАНИЙ, СЕМАНТИЧЕСКИЙ ПОИСК, КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА

Целью работы является исследование, проектирование и разработка веб-приложения для медицинской клиники с интеграцией интеллектуального AI-помощника. Разрабатываемая система предназначена для предоставления пользователям информации о клинике, её сотрудниках, медицинских услугах и направлениях, обработки заявок на прием, а также автоматизации справочного взаимодействия пациента с медицинской организацией.

В результате выполнения работы было разработано веб-приложение, включающее клиентскую часть, серверную часть, базу данных и AI-сервис. Клиентская часть обеспечивает отображение информации о клинике, врачах и услугах, а также предоставляет интерфейс для взаимодействия с AI-помощником. Серверная часть реализует обработку запросов, хранение и управление данными приложения, а также взаимодействие с AI-модулем.

Особое внимание уделено исследованию больших языковых моделей и технологии Retrieval Augmented Generation, позволяющей формировать ответы AI-помощника на основе внешней базы знаний клиники. Это снижает вероятность генерации недостоверной информации и позволяет использовать актуальные сведения о врачах, услугах, правилах записи и контактных данных медицинской организации.

					ПГУ 1.090401.16.001 ПЗ			
Имя	Лист	№ докум	Подпись	Дата	Веб-приложение с интеграцией AI-помощника для медицинской клиники Пояснительная записка	Лит.	Лист	Листов
Разраб.	Павлов Е.И.	100	15.06.20					
Провер.	Зинкин С.А.	2	15.06.20				2	69
И. Контр.	Бычков А.С.	1	15.06.20			ФВТ, каф. ВТ		
Учт.	Иванов И.А.	1	15.06.20			гр. 24ВВМ1		

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	5
ВВЕДЕНИЕ	6
1 Анализ предметной области	8
1.1 Общие характеристики больших языковых моделей	8
1.2 Применение нейронных сетей в медицине	13
1.3 Анализ моделей нейронных сетей	16
1.3.1 OpenAI GPT 5.4 mini	16
1.3.2 GPT 5.5	17
1.3.3 Antropic Claude Sonnet 4.6/Opus 4.6.....	19
1.3.4 Gemini 2.5 Flash.....	22
1.3.5 Mistral AI	21
1.3.6 Сравнение моделей.....	24
1.4 Анализ embedding-моделей для семантического поиска.....	24
1.5 Анализ темы	26
1.6 Анализ аналогов.....	29
1.6.1 Веб-приложение Медидцинской клиники «Медгард»	29
1.6.2 Веб-приложение Медицинской клиники «Здоровье»	30
1.6.3 Веб-приложение медицинской клиники «Клиника Фомина»	31
1.7 Анализ темы	34
1.8 Анализ клиент-серверной архитектуры в рамках решаемой задачи.....	36
1.9 Выбор языка программирования и среды разработки	39
1.10 Выбор фреймворка для работы с искусственным интеллектом	39
1.11 Выбор языка программирования и среды разработки	40
1.12 Выбор языка программирования и среды разработки	41
2 Разработка веб-приложения для медицинской клиники.....	41
2.1 Разработка серверной части веб-приложения	41
2.1.1 Разработка серверного приложения	41
2.1.2 Проектирование доменной модели.....	42
2.1.3 Реализация REST API.....	44
2.1.4 Реализация модуля записи на приём	45

2.1.5 Реализация аутентификации и ролевой модели.....	46
2.1.6 Логирование и аудит	47
2.1.7 Развёртывание серверного приложения.....	48
2.2 Разработка клиентской части веб-приложения	48
2.2.1 Структура клиентской части	48
2.2.2 Реализация API-клиента	49
2.2.3 Навигация и основные разделы интерфейса	50
2.3 Разработка AI-модуля.....	50
2.3.1 Разработка AI-контроллера	50
2.3.2 Реализация сервисного слоя AI-помощника	51
2.3.3 Реализация модуля медицинской безопасности	52
2.3.4.RAG как основа медицинского помощника	53
2.3.5.Разработка поиска по базе знаний клиники.....	54
2.3.6.Индексация базы знаний.....	55
2.3.7.Интеграция с LLM.....	56
2.3.8.Формирование итогового ответа AI-помощника	57
2.3.9.Сохранение истории AI-диалогов и источников.....	58
2.3.10.Оценка качества AI-модуля.....	58
2.4 Тестирование приложения	59
2.4.1.Проверка серверной части.....	59
2.4.2.Проверка пользовательских сценариев.....	61
ЗАКЛЮЧЕНИЕ.....	70
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	71
ПРИЛОЖЕНИЕ А – Листинг программы	73

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяются следующие сокращения и обозначения.

ИИ – искусственный интеллект

СУБД – система управления базами данных

AI (Artificial Intelligence) – искусственный интеллект

API (Application Programming Interface) – программный интерфейс приложения

DTO (Data Transfer Object) – объект передачи данных между слоями приложения

HTTP (HyperText Transfer Protocol) – протокол для получения ресурсов

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

JPA (Java Persistence API) – спецификация, предоставляющая возможность сохранять в удобном виде Java-объекты в базе данных

JWT (JSON Web Token) – открытый стандарт для создания токенов доступа, основанный на формате JSON

LLM (Large Language Model) – языковая модель, основанная на нейронной сети с множеством параметров

RAG (Retrieval-Augmented Generation) – генерация ответа с предварительным поиском релевантных источников

REST (Representational State Transfer) – архитектурный стиль взаимодействия web-сервисов.

SQL (Structured Query Language) – язык описания и обработки данных в реляционных СУБД

UI (User Interface) – пользовательский интерфейс

URL (Uniform Resource Locator) – адрес ресурса в сети Интернет

UUID (Universally Unique Identifier) – универсальный уникальный идентификатор

ВВЕДЕНИЕ

Информационные технологии с каждым годом продолжают своё развитие и затрагивают всё больше направлений. Так, медицинские клиники всё активнее используют и заключают контракты на приобретение веб-приложений. Это позволяет автоматизировать работу с пациентами, сократить время на решение административных вопросов, а также снизить значительную нагрузку на персонал клиник.

Особую роль в современных веб-приложениях отводят качеству взаимодействия с пользователем. Перед обращением в медицинское учреждение пациенту часто приходится самостоятельно узнавать ряд вопросов: стоимость услуги, дату приёма конкретного врача, что взять с собой на приём и так далее. При отсутствии удобного приложения пользователю приходится тратить много времени, чтобы получать необходимые сведения или даже связываться с администратором. В результате возрастает число однотипных обращений к администрации клиник, а пациенту становится сложнее в кратчайшее время получить необходимую информацию.

Для решения проблем, изложенных выше, в последнее время, всё чаще в веб-приложения интегрируются AI-помощники. Современные модели способны обрабатывать вопросы пользователя на естественном языке, анализировать контекст запроса и формировать понятные текстовые ответы. Благодаря этому AI-помощник может использоваться как дополнительный справочно-информационный инструмент, который помогает пользователю быстрее ориентироваться в услугах медицинской организации и выбирать подходящее направление обращения.

Однако применение больших языковых моделей в медицинской предметной области имеет ряд ограничений. Универсальная языковая модель может не обладать актуальными сведениями о конкретной клинике, ее врачах, расписании, услугах и правилах записи. Кроме того, существует риск генерации недостоверной информации, что особенно критично для медицинских систем. Поэтому при разработке AI-помощника важно не только

подключить языковую модель, но и ограничить ее работу проверенной базой знаний медицинской организации.

Актуальность темы обусловлена тем, что типовой сайт клиники часто решает только информационную задачу: показывает перечень услуг, телефоны и форму заявки. Однако пациенту требуется связанный сервис, в котором каталог врачей, расписание, личный кабинет и AI-помощник работают как единая система. Особенно важна интеграция ассистента с базой знаний клиники, чтобы ответ строился не на свободной генерации, а на проверяемых материалах.

Цель работы – спроектировать и реализовать веб-приложение для медицинской клиники с интеграцией AI-помощника, ориентированного на безопасную справочную поддержку пациента и организационную маршрутизацию.

Для достижения цели решаются задачи: анализ аналогов и требований; проектирование архитектуры; разработка клиентской и серверной части приложения; реализация авторизации и ролей; построение базы знаний и RAG-контура; экспериментальная проверка AI-помощника на контрольном наборе запросов; анализ полученных результатов.

Объектом исследования является процесс цифрового взаимодействия пациента с медицинской клиникой. Предметом исследования являются методы построения веб-приложения с AI-помощником, использующим RAG подход и ограничительную политику медицинской безопасности [1].

1 Анализ предметной области

Перед разработкой приложения необходимо исследовать предметную область, определить существующие решения и выявить функциональные пробелы. Для медицинской клиники ключевыми процессами являются публикация каталога услуг, ведение карточек врачей, управление расписанием, прием заявок на запись, информирование пациентов и сопровождение административных действий.

Веб-приложение клиники отличается от обычной информационной страницы тем, что оно должно поддерживать транзакционные операции. Если пациент записывается на прием, система обязана проверить доступность слота, длительность услуги, права пользователя и отсутствие конфликтующих записей. Ошибка в такой логике приводит не только к неудобству пользователя, но и к нарушению расписания врача.

AI-помощник в клинике должен рассматриваться как модуль поддержки, а не как автономный медицинский эксперт. По этой причине ответы формируются только после поиска в базе знаний, сопровождаются источниками, а при тревожных симптомах система должна отказаться от обычной консультационной логики и рекомендовать срочную очную помощь.

Первым шагом к выполнению необходимого анализа будет исследование общих задач, требуемых к большой языковой модели, а также обоснование выбора класса модели.

1.1 Общие характеристики больших языковых моделей

Для реализации AI-помощника в веб-приложении медицинской клиники целесообразно использовать большие языковые модели. Это связано с тем, что основная задача разрабатываемой системы заключается в обработке текстовых запросов пользователей, формировании справочных ответов, предварительной маршрутизации пациента и интеграции результата работы модели с серверной частью приложения.

Большая языковая модель представляет собой модель, обученную на больших объемах текстовых данных и предназначенную для понимания и

генерации естественного языка. Такие модели способны анализировать пользовательский запрос, учитывать контекст диалога, выделять ключевую информацию, формировать ответ в понятной форме и возвращать результат в структурированном виде. В рамках медицинского веб-приложения это позволяет реализовать AI-помощника, который помогает пользователю получить информацию об услугах клиники, выбрать направление обращения и подготовить первичное описание проблемы. Данная архитектура представлена на рисунке 1.

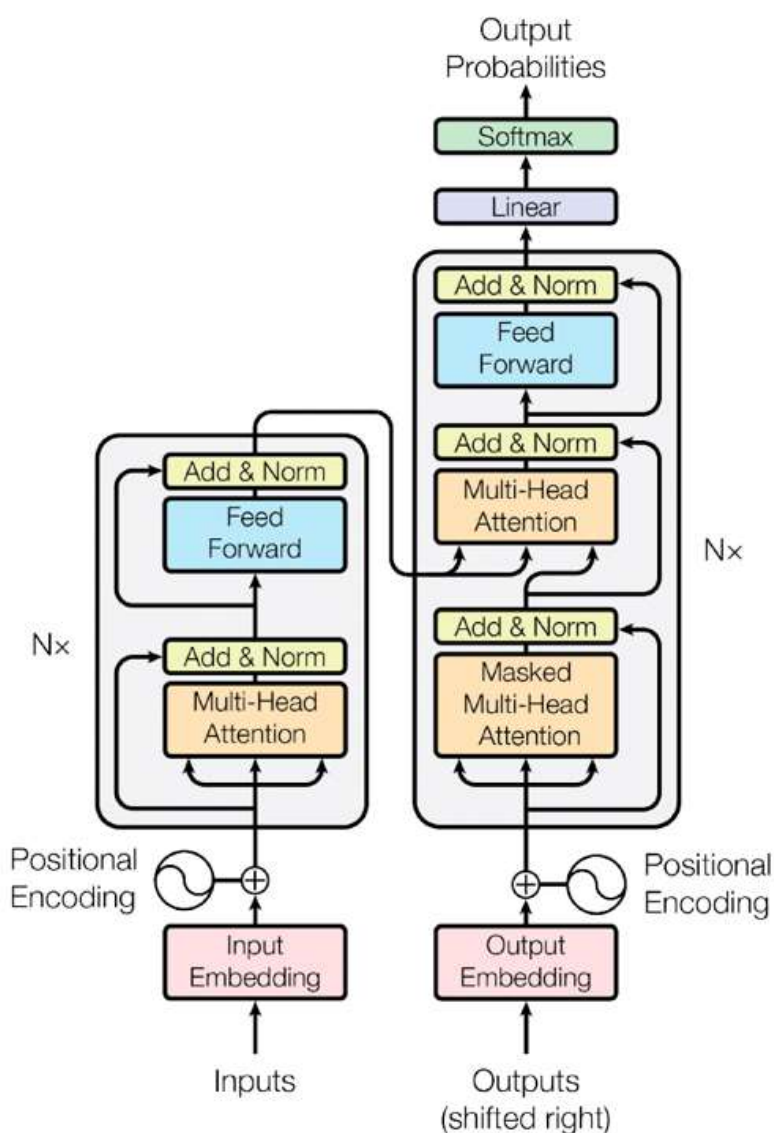


Рисунок1-Архитектура Transformer

Современные большие языковые модели чаще всего строятся на архитектуре Transformer. Данная архитектура основана на механизме внимания, который позволяет модели определять связи между словами и

фрагментами текста. В отличие от более ранних подходов, Transformer эффективно работает с длинными последовательностями и способен учитывать контекст сообщения целиком. Это особенно важно для диалоговых систем, так как смысл запроса пациента может зависеть не только от одного предложения, но и от предыдущих сообщений в диалоге.

Основным элементом архитектуры Transformer является механизм self-attention. Он позволяет каждому токenu входной последовательности учитывать информацию о других токенах. За счет этого модель может понимать, какие части запроса являются наиболее важными. Например, в сообщении пользователя «болит горло, температура, к какому врачу записаться» модель должна выделить признаки жалобы, определить, что пользователь хочет выбрать специалиста, и сформировать безопасный ответ без постановки диагноза. Принцип работы данного механизма показан на рисунке 2 [2].

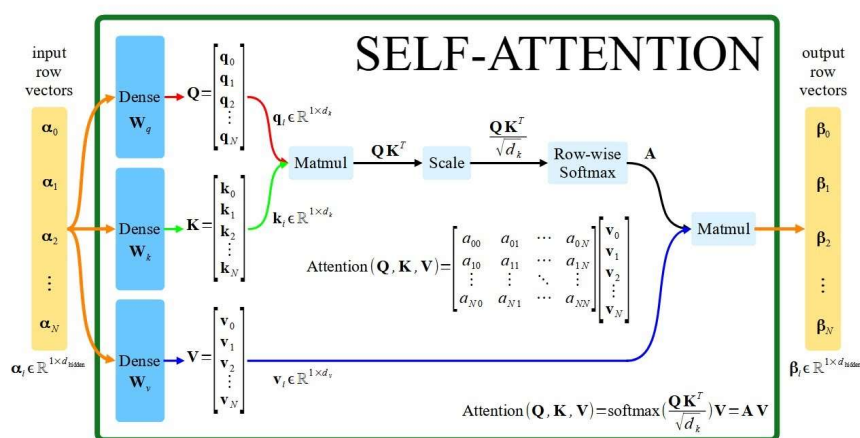


Рисунок 2-механизм Self-attention

Языковая модель сначала разбивает текст на маленькие части – токены. Потом она превращает эти части в числа, с которыми может работать нейронная сеть. После обработки модель постепенно выбирает новые токены и складывает из них ответ.

Для медицинского AI-помощника важной характеристикой является контекстное окно модели. Под контекстным окном понимается максимальный объем текста, который модель может учитывать при формировании ответа.

Чем больше контекстное окно, тем больше информации можно передать модели: историю диалога, список врачей, перечень услуг, правила записи, ограничения безопасности и фрагменты базы знаний клиники. Это позволяет повысить точность ответа и сделать его связанным с конкретной медицинской организацией.

Еще одной важной возможностью современных языковых моделей является структурированный вывод. В обычном режиме модель возвращает пользователю текстовый ответ. Однако при интеграции с серверной частью приложения часто требуется получить результат в виде объекта с заранее заданными полями. Например, модель может вернуть JSON-структуру, содержащую рекомендуемое направление обращения, предполагаемого специалиста, степень срочности, краткое описание жалобы и текст ответа пациенту. Такой подход позволяет использовать AI-помощника не только как чат, но и как часть бизнес-логики приложения.

Для повышения достоверности ответов в медицинском веб-приложении используется подход Retrieval Augmented Generation. Его суть заключается в том, что модель получает дополнительный контекст из базы знаний приложения. В качестве такой базы могут использоваться сведения о врачах, услугах, расписании, подготовке к процедурам, адресах филиалов и правилах записи. Благодаря этому AI-помощник формирует ответ не только на основе общих знаний модели, но и с учетом реальной информации конкретной клиники. Подход RAG в общем виде представлен на рисунке 3 [3].

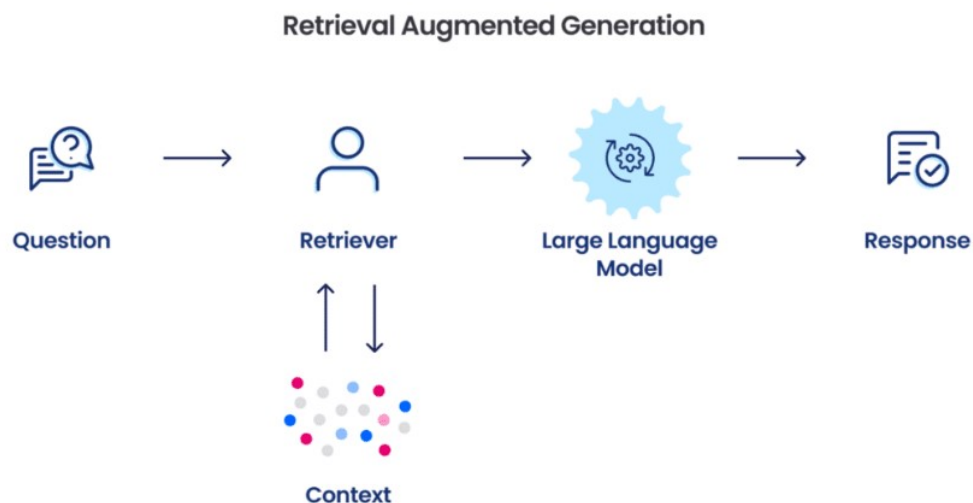


Рисунок 3-технология Retrieval-Augmented Generation

При использовании нейронных моделей в медицинских приложениях необходимо учитывать правила безопасности. AI-помощник не должен заменять врача и ставить диагнозы или назначать лечение. Поэтому при проектировании модуля обязательно надо ставить ограничения на возможные ответы и формулировки модели.

А рамках данной работы используются внешние языковые модели, которые доступны на прямую через API. Такие модели легко интегрировать в серверную часть приложения через Spring AI. Это компонент фреймворка Spring, который позволяет обращаться к моделям через Java приложения[4] .

Таким образом для данного веб-приложения наиболее подходящей группой нейронных сетей являются большие языковые модели. Они позволяют обрабатывать текстовые сообщения, использовать базу знаний приложения и возвращать результат в понятном и структурированном виде.

1.2 Применение нейронных сетей в медицине

В последние годы нейронные сети стали одними из самых перспективных направлений в развитии медицинских веб-приложениях. Их применение позволяет упростить обработку больших данных, улучшить качество взаимодействия пациентов с персоналом клиники и снизить нагрузку с сотрудников медицинских учреждений.

Для данной работы наибольший интерес представляет применение больших языковых моделей. Такие модели способны принимать запросы пользователей, анализировать данные и выдавать структурированные релевантные ответы для дальнейшей обработки приложением. В медицинском веб-приложении это позволяет реализовать AI-помощника, который помогает пациенту получить справочную информацию об услугах клиники, подобрать подходящее направление обращения, уточнить порядок записи на прием и подготовить краткое описание жалоб для администратора или врача.

В отличие от обычного чат-бота, работающего по заранее заданным сценариям, AI-помощник на основе большой языковой модели способен обрабатывать более широкий круг пользовательских запросов. Пациент может сформулировать вопрос в свободной форме, использовать бытовые выражения, неполные описания симптомов или смешивать медицинские и немедицинские термины. Модель в таком случае выполняет смысловой анализ сообщения, определяет намерение пользователя и формирует ответ в понятной форме. Например, пользователь может написать: «болит горло и температура, к кому записаться», а система должна не ставить диагноз, а осторожно предложить обратиться к терапевту или оториноларингологу, а также напомнить, что при ухудшении состояния необходимо получить очную медицинскую помощь.

Важной особенностью применения нейронных сетей в медицине является необходимость соблюдения ограничений безопасности. AI-помощник не должен заменять врача, устанавливать диагноз, назначать лекарственные препараты или давать категоричные рекомендации по

лечению. Его задача заключается в справочной поддержке, предварительной маршрутизации пациента и помощи в использовании сервисов клиники. Поэтому при проектировании интеллектуального модуля необходимо задавать строгие системные инструкции, ограничивать область ответа и предусматривать обработку потенциально опасных ситуаций. Если пользователь описывает симптомы, которые могут указывать на угрозу жизни или необходимость срочной помощи, система должна рекомендовать немедленно обратиться к врачу или вызвать скорую медицинскую помощь.

Для повышения точности ответов AI-помощника целесообразно использовать подход Retrieval Augmented Generation. Данный подход предполагает, что языковая модель формирует ответ не только на основе общих знаний, но и с учетом внешней базы знаний приложения. В качестве такой базы могут использоваться сведения о врачах клиники, перечень медицинских услуг, расписание приема, правила подготовки к процедурам, адреса филиалов, контактная информация и внутренние инструкции медицинской организации. Это позволяет снизить вероятность генерации недостоверных сведений и сделать ответы AI-помощника более привязанными к конкретной клинике.

В разрабатываемом веб-приложении медицинской клиники AI-помощник может выполнять несколько прикладных функций. Во-первых, он может отвечать на типовые вопросы пациентов об услугах, специалистах, графике работы и правилах записи. Во-вторых, он может помогать выбрать направление обращения на основе описания жалоб, не устанавливая при этом диагноз. В-третьих, он может формировать краткое резюме обращения пациента, которое затем может быть использовано в заявке или передано администратору. В-четвертых, он может возвращать результат в структурированном виде, например в формате JSON, что позволяет серверной части приложения автоматически обрабатывать ответ модели.

Структурированный вывод является важным преимуществом современных языковых моделей при интеграции с веб-приложением. Вместо

обычного текстового ответа модель может сформировать объект, содержащий отдельные поля: предполагаемое направление обращения, рекомендуемого специалиста, степень срочности, краткое описание проблемы и текст ответа для пользователя. Такой подход позволяет использовать нейронную сеть не только как диалоговый интерфейс, но и как компонент бизнес-логики приложения. Например, если модель определила, что пользователю нужен кардиолог, приложение может предложить список врачей данного направления или открыть форму записи на прием.

Применение нейронных сетей также может быть полезно для административной части медицинского веб-приложения. AI-модуль может помогать классифицировать обращения пациентов, выделять ключевые жалобы, сокращать длинные сообщения до краткого резюме, подготавливать шаблоны ответов и группировать заявки по направлениям. Это снижает нагрузку на сотрудников регистратуры и ускоряет первичную обработку обращений. При этом окончательное решение всегда должно оставаться за медицинским работником или администратором клиники.

При выборе модели для медицинского AI-помощника необходимо учитывать несколько критериев: качество понимания русского языка, способность следовать инструкциям, поддержка структурированного вывода, скорость ответа, стоимость эксплуатации, возможность интеграции с серверной частью приложения и наличие механизмов безопасного поведения. Для веб-приложения на Spring Boot наиболее удобным вариантом является использование внешней языковой модели через Spring AI. Такой подход позволяет интегрировать взаимодействие с LLM прямо в серверную часть приложения и удобно обращаться к модели из приложения.

Таким образом, в рамках данной работы нейронные сети рассматриваются не как средства постановки диагноза, а как дополнительный помощник для пользователей клиники, который может помочь с выбором специалиста, подробнее проанализировать свою проблему, ответить на самые частые вопросы качественно и в кратчайшие сроки.

1.3 Анализ моделей нейронных сетей

В рамках разрабатываемого веб-приложения медицинской клиники нейронная сеть используется для реализации AI-помощника, который должен обрабатывать текстовые запросы пациентов, помогать с выбором направления обращения, отвечать на вопросы об услугах клиники и формировать структурированный результат для серверной части приложения. Поэтому в работе рассматриваются большие языковые модели, доступные через внешние API и интегрируемые с серверной частью приложения на Spring Boot с использованием Spring AI [7].

1.3.1 OpenAI GPT 5.4 mini

Модель GPT-5.4 mini, разработанная компанией OpenAI, является современной большой языковой моделью, предназначенной для обработки текстовых запросов, генерации ответов, анализа информации и взаимодействия с внешними программными функциями. GPT-5.4 mini относится к классу больших языковых моделей и используется для работы с естественным языком. Архитектура модели основана на технологии Transformer [20].

Преимуществом GPT-5.4 mini является сочетание высокого качества ответов, высокой скорости работы и сравнительно низкой стоимости использования. Это особенно важно для веб-приложения медицинской клиники, так как AI-помощник должен отвечать пользователю в интерактивном режиме и при этом не создавать чрезмерную нагрузку на серверную инфраструктуру. Еще одним преимуществом является возможность подключения модели к Java-приложению через Spring AI. Благодаря этому серверная часть может отправлять запросы пользователя в модель, получать ответ и использовать его в логике приложения [10].

Важным свойством GPT-5.4 mini является возможность возвращать не только обычный текст, но и структурированный результат. Это позволяет использовать модель не только как чат-бота, но и как часть программной логики медицинского сервиса. Например, ответ модели может содержать

отдельные поля: рекомендуемое направление обращения, предполагаемый специалист, степень срочности, краткое описание жалобы и текст ответа пациенту. Такой подход упрощает дальнейшую обработку результата в серверной части приложения.

Недостатком GPT-5.4 mini является зависимость от внешнего программного интерфейса OpenAI. Для работы модели требуется подключение к интернету и наличие доступа к API. Кроме того, при использовании внешней модели необходимо учитывать вопросы защиты персональных данных и не передавать в запросах лишнюю медицинскую или идентифицирующую информацию. Поэтому в рамках разрабатываемого приложения AI-помощник должен использоваться только для справочной поддержки и предварительной маршрутизации пациента, а не для постановки диагноза или назначения лечения.

1.3.2 GPT 5.5

Модель GPT-5.5, разработанная компанией OpenAI, является современной большой языковой моделью, предназначенной для решения сложных задач обработки естественного языка, анализа информации, программирования, работы с документами и выполнения многоэтапных пользовательских запросов. Данная модель ориентирована на понимание текста, рассуждение, генерацию ответов и взаимодействие с внешними инструментами.

GPT-5.5 является наиболее новой моделью OpenAI для сложной профессиональной работы. В официальной документации OpenAI указано, что она предназначена для сложных рассуждений, поддерживает текстовый и графический ввод, имеет контекстное окно 1 миллион токенов, максимальный размер ответа 128 тысяч токенов, а также поддерживает разные уровни рассуждения: none, low, medium, high и xhigh.

Для разрабатываемого веб-приложения медицинской клиники GPT-5.5 может использоваться как наиболее качественная модель для сложных сценариев. Например, она подходит для анализа большого объема справочной

информации клиники, подготовки базы знаний, проверки корректности ответов AI-помощника, обработки сложных обращений пользователей и формирования подробных объяснений. Благодаря большому контекстному окну модель может учитывать значительный объем данных: сведения о врачах, услугах, расписании, правилах записи, ограничениях медицинской безопасности и истории диалога с пользователем.

GPT-5.5 выделяется своим усовершенствованным мыслительным процессом и умением решать комплексные, многошаговые задачи. При анонсе OpenAI представили его как самую интеллектуальную модель на тот момент, созданную для решения амбициозных задач в области программирования, анализа данных и обработки массивных объемов информации. Согласно данным OpenAI, GPT-5.5 демонстрирует впечатляющие результаты в профессиональных тестовых сценариях: 84,9% на GDPval, 78,7% на OSWorld-Verified и 98,0% на Tau2-bench Telecom, причем эти показатели достигаются без специальной оптимизации инструкций.

Актуальным преимуществом GPT-5.5 для медицинских AI-систем является его повышенная точность. На практике это означает, что модель не будет ставить диагнозы, назначать терапию, претендовать на роль врача или, при обнаружении тревожных симптомов, но сможет рекомендовать пользователю обратиться за профессиональной медицинской помощью. Более того, GPT-5.5 может быть задействован для верификации качества ответов менее дорогих аналогов, например GPT-5.4 mini.

Среди ограничений GPT-5.5 можно отметить его более высокую цену эксплуатации по сравнению с моделями серии mini. Официальные тарифы OpenAI для GPT-5.5 составляют 5 долларов за миллион входных токенов и 30 долларов за миллион выходных токенов. Для стандартных нужд AI-помощника в медицинской клинике подобный уровень затрат может оказаться неоправданно высоким, поскольку большинство обращений пациентов носят шаблонный характер, связанный с рабочим расписанием, выбором специалиста, уточнением информации об услугах или записью на прием.

1.3.3 Anthropic Claude Sonnet 4.6/Opus 4.6

Модели Claude Sonnet и Claude Opus, разработанные Anthropic, относятся к современным большим языковым моделям. Они работают с текстом, помогают анализировать информацию, обрабатывать документы, писать код и решать сложные пользовательские задачи. В этой работе они рассматриваются как сильная альтернатива моделям OpenAI.

Spring AI поддерживает интеграцию с Anthropic. За счёт этого Claude-модели можно подключить к Java-приложению и использовать внутри серверной логики. Для разрабатываемого веб-приложения это делает их практически применимыми.

Claude Sonnet выделяется аккуратными формулировками и способностью работать с длительными задачами. В официальных материалах Anthropic Claude Sonnet 4.6 описывается как наиболее способная модель линейки Sonnet. Разработчики указывают, что её улучшили для программирования, работы с компьютером, длинных рассуждений, планирования действий, работы со знаниями и дизайна. Также для модели заявлено контекстное окно в 1 миллион токенов в бета-режиме.

Для медицинского AI-помощника Claude Sonnet интересна прежде всего осторожным стилем ответов. Модель может давать развёрнутые пояснения, не переходя к категоричным выводам. В медицинской предметной области это особенно важно: AI-помощник не должен заменять врача, ставить диагноз или назначать лечение. Его роль – дать пациенту справочную поддержку, помочь с предварительной маршрутизацией и при необходимости посоветовать обратиться к специалисту.

Claude Opus относится к более мощному классу моделей. В материалах Anthropic указано, что Opus 4.6 лучше справляется с длинным контекстом и сложными задачами по сравнению с предыдущими моделями. Такая модель может пригодиться при анализе больших документов, проверке качества ответов и в сложных корпоративных сценариях.

У Claude Sonnet и Claude Opus есть и слабая сторона – высокая стоимость по сравнению с более экономичными моделями. Кроме того, для выбранной архитектуры приложения нужно учитывать, насколько удобно модель поддерживает структурированный вывод и интегрируется с серверной логикой. Поэтому Claude-модели разумно рассматривать как сильную альтернативу, но не как единственный возможный вариант.

Claude Sonnet / Opus хорошо подходят для сложных текстовых задач, осторожных ответов и корпоративных сценариев. Их можно использовать в медицинском AI-помощнике, однако по сочетанию стоимости и удобства интеграции в рамках текущей реализации они могут уступать моделям OpenAI [22].

1.3.4 Gemini 2.5 Flash

Модели Google Gemini – это семейство больших языковых моделей для работы с текстом, изображениями, документами и другими типами данных.

Spring AI поддерживает Google GenAI и Vertex AI Gemini, поэтому модели Gemini можно подключать к Java-приложениям. Для серверной части веб-приложения медицинской клиники это делает их технически применимым вариантом.

Сильная сторона Gemini – мультимодальность. Модель умеет работать не только с текстом, но и с изображениями, файлами, документами и другими форматами данных. В медицинской клинике это может пригодиться при дальнейшем развитии приложения: например, для обработки справочных документов, PDF-файлов, инструкций или материалов, которые пользователь прикрепляет к обращению.

Еще одно преимущество Gemini – поддержка длинного контекста. В модель можно передать большой объем информации: сведения о врачах, услугах, правилах записи, подготовке к процедурам и другие данные клиники. За счет этого AI-помощник сможет отвечать с опорой на конкретную базу знаний, а не только на общие сведения.

Для текущей работы у Gemini есть и ограничение: мультимодальность пока не относится к ключевым задачам приложения. Разрабатываемая система в первую очередь строится вокруг текстового AI-помощника, который помогает пациенту выбрать направление обращения и записаться на прием. Поэтому обработка изображений, видео или аудио выглядит перспективной, но не обязательной для первой версии.

Google Gemini хорошо подходит для мультимодальных сценариев и работы с большими документами. Эта модель может быть полезна при дальнейшем расширении медицинского веб-приложения, но для текущей реализации текстового AI-помощника она не является единственным оптимальным выбором [8].

1.3.5 Mistral Large 3

Модель Mistral Large 3 – это большая языковая модель для обработки естественного языка, генерации текстовых ответов, анализа запросов и создания диалоговых приложений. В рамках данной работы Mistral Large 3 рассматривается как внешний поставщик языковых моделей, который можно интегрировать в приложение через Spring AI.

Преимущество Mistral Large 3 – наличие альтернативного поставщика AI-моделей. Для организаций это может быть важным фактором: появляется выбор между разными сервисами, больше гибкости при интеграции и меньше зависимость от одного поставщика. Кроме того, Mistral Large 3 часто рассматривают как европейскую альтернативу, что может быть полезно для корпоративных сценариев.

При этом для текущей задачи у Mistral Large 3 есть ограничение. По совокупности качества, универсальности и удобства применения в медицинском AI-помощнике она уступает лидирующим моделям OpenAI и Anthropic. В медицинской предметной области особенно важны точное выполнение инструкций, осторожные формулировки, структурированный результат и стабильность ответов. Поэтому Mistral Large 3 логичнее

рассматривать как резервный вариант или альтернативного поставщика, а не как основную модель для текущей реализации.

Mistral Large 3 выглядит перспективным вариантом для интеграции в Spring AI-приложение. Она может использоваться для справочной поддержки пациентов и обработки типовых обращений, но в рамках данной работы рассматривается именно как альтернативное решение [9].

1.3.6 Выбор модели

После сравнения больших языковых моделей нужно выбрать ту, которая будет подключена к AI-сервису веб-приложения медицинской клиники. В анализе сравниваются не компании и не линейки моделей, а конкретные варианты: GPT-5.4-mini, GPT-5.5, Claude Sonnet 4.6, Gemini 2.5 Flash и Mistral Large 3.

Модель выбиралась по метрикам, которые напрямую влияют на работу AI-помощника в медицинском сервисе. Оценивались корректность ответа, безопасность, валидность структурированного JSON-вывода, частота галлюцинаций, время ответа и стоимость эксплуатации. Корректность показывает, насколько ответ совпадает с запросом пользователя и данными базы знаний. Безопасность — соблюдает ли модель ограничения медицинской области: не ставит диагнозы, не назначает лечение и отправляет пользователя к врачу при тревожных симптомах. Валидность JSON-вывода показывает, насколько стабильно модель возвращает структурированный результат, который серверная часть может обработать без исправлений. Частота галлюцинаций отражает долю ответов с неподтверждённой информацией. Время ответа важно для интерфейса чата, а стоимость эксплуатации показывает, можно ли использовать модель при большом числе пользовательских обращений.

Для проверки использовался контрольный набор типовых запросов пациента клиники. В него вошли вопросы об услугах и врачах, запросы на выбор специалиста, вопросы по записи на приём, обращения с недостатком исходных данных и потенциально опасные медицинские запросы. Такой набор

позволяет оценить не только качество текста, но и то, как модель ведёт себя в медицинских ограничениях.

Для графика метрики привели к нормированной шкале. Корректность, безопасность и валидность JSON указаны в процентах. Для частоты галлюцинаций использован обратный показатель: чем меньше галлюцинаций, тем выше значение на графике. Для времени ответа и стоимости эксплуатации также применена обратная нормировка, потому что для веб-приложения лучше меньшая задержка и меньшая стоимость.

Результаты сравнения представлены на рисунке 4.

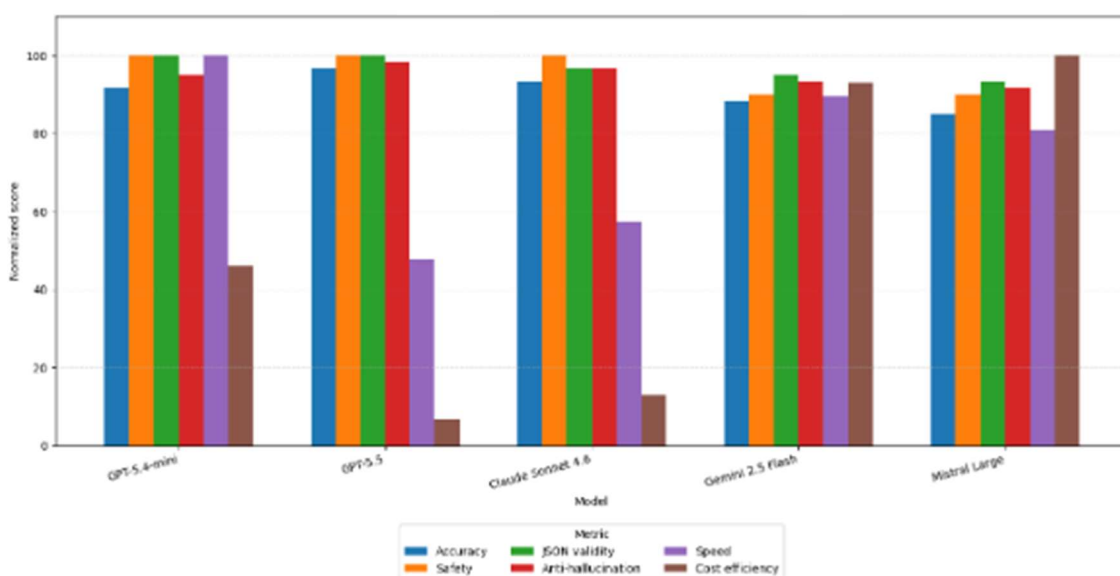


Рисунок 4-График сравнения моделей

GPT-5.5 показывает самую высокую корректность и минимальную частоту галлюцинаций. Но у этой модели выше стоимость эксплуатации и больше время ответа. Для справочного AI-помощника клиники это избыточный вариант: большая часть запросов связана с услугами, врачами, записью на приём и первичной маршрутизацией, а не со сложным аналитическим рассуждением.

Claude Sonnet 4.6 даёт высокие результаты по корректности и безопасности, но уступает GPT-5.4-mini по скорости ответа и стоимости. Gemini 2.5 Flash и Mistral Large 3 выгоднее по эксплуатационным параметрам, однако проигрывают по качеству ответа, безопасности и стабильности

структурированного вывода. Для медицинского сценария эти показатели критичны.

По результатам сравнения основной моделью для AI-сервиса выбрана GPT-5.4-mini. Она даёт достаточную корректность ответов, соблюдает ограничения медицинской безопасности, стабильно формирует JSON-вывод, редко добавляет неподтверждённые сведения, быстро отвечает и остаётся приемлемой по стоимости.

Для разрабатываемого веб-приложения медицинской клиники основной языковой моделью AI-помощника выбирается GPT-5.4-mini.

1.4 Анализ embedding-моделей для семантического поиска

В RAG-архитектуре за качество ответа отвечает не только большая языковая модель. LLM формирует финальный текст для пользователя, но перед этим система должна найти в базе знаний нужные фрагменты. Эту задачу решает embedding-модель. Она переводит документы базы знаний и запрос пользователя в числовые векторы. После этого система сравнивает векторы между собой и находит материалы, которые ближе всего подходят по смыслу. Найденные фрагменты передаются языковой модели как контекст для ответа.

В веб-приложении медицинской клиники embedding-модель используется для поиска по базе знаний: сведениям о врачах, услугах, правилах записи, расписании, подготовке к процедурам и контактными данными. От качества этой модели напрямую зависит точность AI-помощника. Если она плохо связывает вопрос пациента с нужным разделом базы знаний, LLM получит неподходящий контекст. В такой ситуации ответ может получиться неполным, неточным или просто не по теме.

При выборе embedding-модели для RAG-контура сравнивались пять вариантов: OpenAI text-embedding-3-small, OpenAI text-embedding-3-large, Cohere Embed v4, mistral-embed и BAAI BGE-M3. Все они решают одну базовую задачу: переводят запрос пользователя и фрагменты базы знаний в числовые векторы. После этого система сравнивает векторы и подбирает

документы, которые ближе всего подходят по смыслу. В данном разделе модели сравниваются по следующим метрикам: открытый benchmark-показатель, эффективность хранения векторов, экономичность обработки токенов, объём входного контекста.

Benchmark показывает общий уровень качества модели по открытым тестам. Эффективность хранения векторов рассчитывается по размерности вектора: чем меньше размерность, тем меньше места потребуется для хранения базы знаний. Экономичность обработки токенов отражает, насколько выгодно использовать модель с точки зрения стоимости обработки текста. Чем ниже цена за 1 млн токенов, тем выше значение этой метрики. Объём входного контекста показывает, какой объём текста модель может обработать за один запрос. Результаты сравнения представлены на рисунке 5.

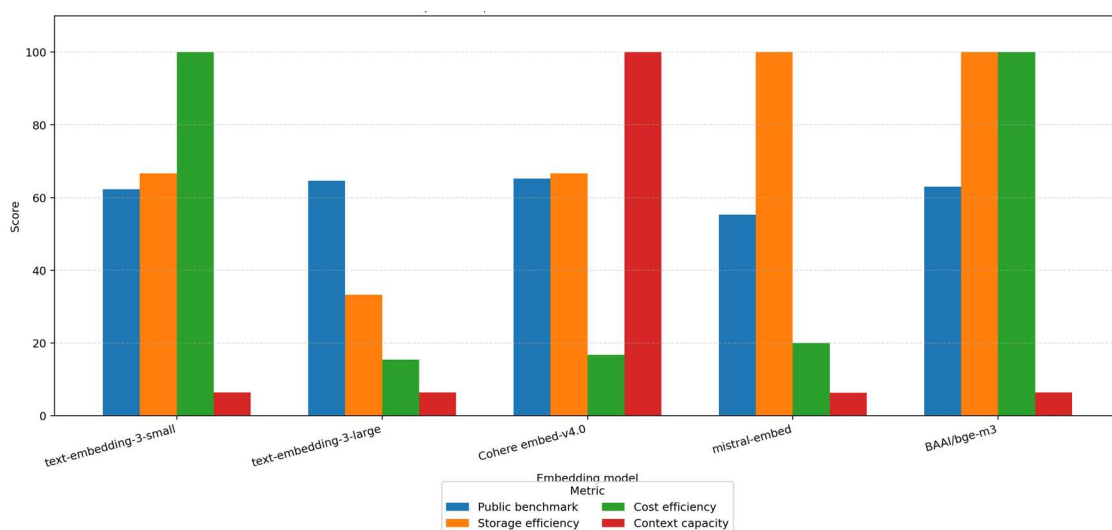


Рисунок 5-График сравнения моделей

Из графика видно, что text-embedding-3-large немного опережает text-embedding-3-small по benchmark-показателю: 64,6 против 62,3. Разница небольшая — 2,3 пункта. При этом размерность вектора у large выше в два раза: 3072 против 1536. Стоимость обработки токенов тоже заметно больше — примерно в 6,5 раза. Для базы знаний медицинской клиники, где основную часть составляют короткие справочные фрагменты, такой прирост качества не окупает рост стоимости и требований к хранению.

Cohere Embed v4 показывает высокий benchmark-показатель и лучший результат по Context capacity. Это полезно для длинных корпоративных документов: регламентов, отчетов, инструкций, больших файлов. В текущей версии медицинского веб-приложения база знаний устроена проще. В ней хранятся описания услуг, карточки врачей, правила записи, адреса, контакты и инструкции перед процедурами. Поэтому большой контекст Cohere Embed v4 не даёт решающего преимущества. К тому же подключение Cohere добавляет ещё одного внешнего поставщика AI-сервиса.

Mistral-embed имеет компактную размерность вектора и приемлемую стоимость обработки токенов, но уступает по открытому benchmark-показателю. BAAI BGE-M3 тоже работает с компактными векторами и может использоваться без оплаты внешнего API при локальном развёртывании. Но локальный вариант требует серверных ресурсов, настройки окружения, мониторинга и сопровождения модели. Для первой версии приложения это лишнее усложнение архитектуры.

По результатам сравнения для RAG-контура выбрана OpenAI text-embedding-3-small. Эта модель даёт наиболее рациональный баланс между качеством, размерностью вектора, стоимостью обработки токенов и простотой интеграции. Она подходит для семантического поиска по коротким справочным фрагментам базы знаний медицинской клиники и хорошо сочетается с выбранной языковой моделью GPT-5.4-mini.

1.5 Анализ темы

В настоящее время пользователи все чаще обращаются к цифровым сервисам для решения повседневных задач, связанных со здоровьем: поиска клиники, выбора врача, записи на прием, получения информации об услугах и уточнения порядка обращения к специалисту. Медицинские организации, в свою очередь, заинтересованы в развитии веб-приложений, которые позволяют снизить нагрузку на регистратуру, ускорить обработку заявок и сделать взаимодействие пациента с клиникой более удобным.

При этом самостоятельный поиск медицинской информации в сети Интернет не всегда является безопасным. Пользователь может столкнуться с недостоверными сведениями, устаревшими рекомендациями, рекламными материалами или непроверенными советами. Особенно опасной такая ситуация становится в случаях, когда человек пытается самостоятельно поставить себе диагноз или подобрать лечение без консультации врача. Поэтому при разработке цифрового сервиса для медицинской клиники необходимо четко разграничивать справочную поддержку и медицинскую консультацию.

Разрабатываемое приложение не ставит диагнозы, не назначает лекарства и не заменяет очный приём врача. Его задача – создать удобную цифровую среду для пациента.

Пользователь сможет посмотреть информацию о клинике, медицинских направлениях и доступных врачах, самостоятельно записаться на приём или обратиться в поддержку. Для сложных вопросов и более простой навигации по приложению в систему добавлен AI-помощник.

Тема актуальна, потому что многие пациенты не всегда понимают, к какому специалисту им нужно обратиться. Например, человека могут беспокоить симптомы, но он не знает, выбрать терапевта, невролога, гастроэнтеролога или другого врача. В такой ситуации AI-помощник не ставит диагноз, а помогает сориентироваться и направляет пациента на приём к подходящему специалисту.

Клиники также часто получают много однотипных вопросов. Пациенты спрашивают о графике работы, стоимости услуг, порядке записи, подготовке к обследованиям, доступности врачей и контактных данных. Большая часть таких обращений не требует участия врача. Их может обработать автоматизированный сервис.

AI-помощник помогает пациенту быстрее получить ответ, а сотрудникам клиники – меньше отвлекаться на повторяющиеся запросы и сосредоточиться на задачах, где действительно нужен человек.

В-третьих, современные веб-приложения медицинских организаций должны обеспечивать не только отображение информации, но и полноценное взаимодействие с пользователем. Обычный сайт клиники часто ограничивается описанием услуг, списком врачей и контактами. Однако для пациента более удобным является сервис, который помогает пройти весь путь: от первичного вопроса до выбора специалиста и отправки заявки на прием. Поэтому разработка веб-приложения с интеграцией AI-помощника является актуальной задачей.

Особенность рассматриваемой темы заключается в сочетании классического веб-приложения и интеллектуального модуля. Серверная часть приложения отвечает за хранение и обработку данных о врачах, услугах, заявках и пользователях. Клиентская часть обеспечивает удобный интерфейс для пациента и администратора. AI-помощник используется как дополнительный сервис, который анализирует текстовый запрос пользователя, формирует безопасный ответ и при необходимости возвращает структурированный результат для дальнейшей обработки backend-частью.

В рамках анализа темы также необходимо учитывать удобство интеграции AI-модуля с backend-частью приложения. Поскольку система разрабатывается на базе Spring Boot, для подключения внешних языковых моделей может использоваться Spring AI. Данный подход позволяет отправлять запросы к модели, получать ответы, использовать структурированный вывод и интегрировать результат работы AI-помощника с бизнес-логикой приложения. Например, модель может вернуть не только текстовый ответ, но и дополнительные поля: рекомендуемое направление, предполагаемого специалиста, уровень срочности и краткое описание обращения.

Таким образом, выбранная тема является актуальной, поскольку объединяет потребность медицинских организаций в цифровизации взаимодействия с пациентами и возможности современных языковых моделей. Разрабатываемое веб-приложение позволяет представить

информацию о клинике, реализовать запись на прием, обеспечить административное управление данными и добавить AI-помощника для справочной поддержки пользователя. Такой подход повышает удобство сервиса, ускоряет обработку обращений и делает взаимодействие пациента с медицинской организацией более понятным и доступным.

1.6 Анализ аналогов

Для обоснования актуальности разрабатываемого веб-приложения был проведен анализ существующих сайтов медицинских клиник города Пензы. В качестве аналогов были выбраны три медицинских сервиса: «Медгард», медицинский центр «Здоровье» и Клиника Фомина. Данные ресурсы выбраны потому, что они относятся к медицинским организациям, предоставляют информацию об услугах и врачах, а также имеют функциональность для взаимодействия пациента с клиникой.

Цель анализа заключается не в сравнении качества медицинских услуг, а в оценке цифровых возможностей сайтов: удобства навигации, наличия онлайн-записи, информации о врачах и услугах, личного кабинета, справочных разделов и интеллектуальной поддержки пользователя. Для разрабатываемого приложения особенно важным является наличие AI-помощника, который может помочь пациенту выбрать направление обращения, получить справочную информацию и подготовить заявку на прием.

1.6.1 Веб-приложение Медицинской клиники «Медгард»

Веб-приложение медицинской клиники «Медгард» позиционирует организацию как многопрофильную частную клинику в Пензе. На главной странице представлены врачи, кнопки записи, вход в личный кабинет, программы, отделения и контактные данные. Для пользователя это сильная сторона: сайт быстро ведет к практическому действию, а не только описывает бренд.

Отдельный интерес представляет Telegram-бот «Медгард». На странице бота указано, что пациенты могут подтверждать или отменять запись через

Telegram, а также получать доступ к сценариям, связанным с записью, результатами анализов и консультациями. Это показывает востребованность дополнительных цифровых каналов, но бот не заменяет AI-помощника: он автоматизирует коммуникацию, а не обязательно формирует ответ с проверяемыми источниками. Пользователь может получить сервис записи и личный кабинет, но не видит механизма, который показывал бы источник ответа, релевантный фрагмент базы знаний и численные показатели уверенности.

В приложении присутствует личный кабинет как сервис доступа к информации о пациенте, результатам лабораторной и инструментальной диагностики и записи на прием без дополнительной авторизации.

Для безопасности на сайте отдельно перечислены рекомендации не передавать пароль, проверять стартовую страницу сайта и сообщать о подозрении на несанкционированный доступ.

Сильным местом «Медгард» является зрелость клинической инфраструктуры и наличие цифровых каналов вокруг записи [10].

Интерфейс веб-приложения представлен на рисунке 6.

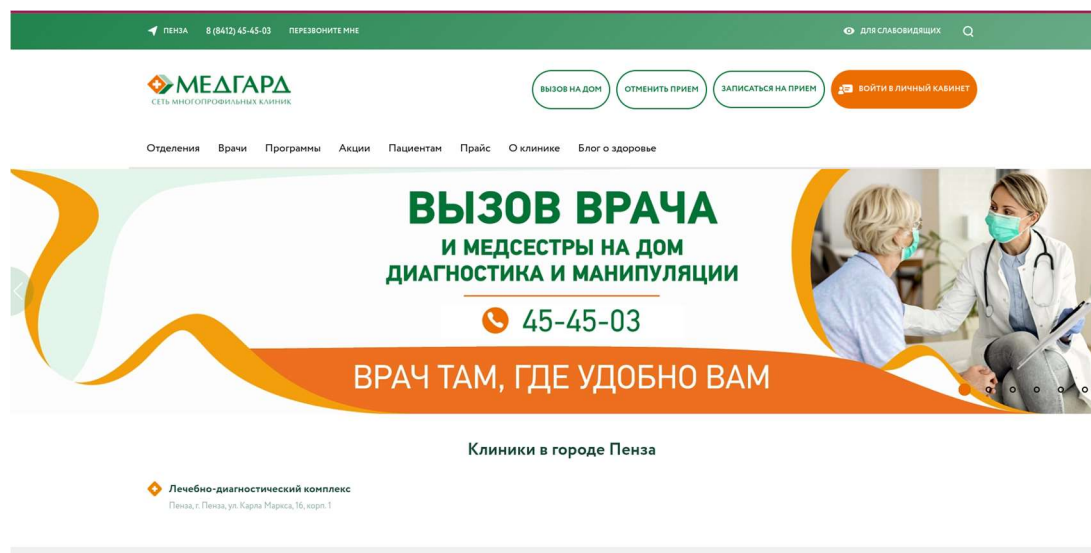


Рисунок 6-Интерфейс веб-приложение клиники «Медгард»

1.6.2 Веб-приложение Медицинской клиники «Здоровье»

Веб-приложение медицинского центра «Здоровье» показывает широкую региональную медицинскую инфраструктуру. На главной странице

представлены несколько клиник с адресами и телефонами, направления, диагностика, анализы, вакцинация, стационар и медицинские осмотры. Для пациента это прежде всего справочник возможностей и контактов.

Сайт содержит версию для слабовидящих. Это сильная сторона с точки зрения доступности. Медицинский сайт учитывает интересы пользователей старшего возраста и людей с ограничениями зрения.

На сайте «Здоровье» отдельно опубликовано предупреждение, что материалы носят информационный характер и не должны использоваться как медицинские рекомендации, а выбор диагноза и лечения остается за врачом. Это в полной мере отвечает требованиям функциональной безопасности [11].

Веб-приложение медицинского центра «Здоровье» выполняет функцию информационного ресурса и канала записи на прием, однако не содержит AI-помощника и не обеспечивает автоматизированную маршрутизацию пациента по описанию жалоб. Интерфейс данного приложения представлен на рисунке 7.

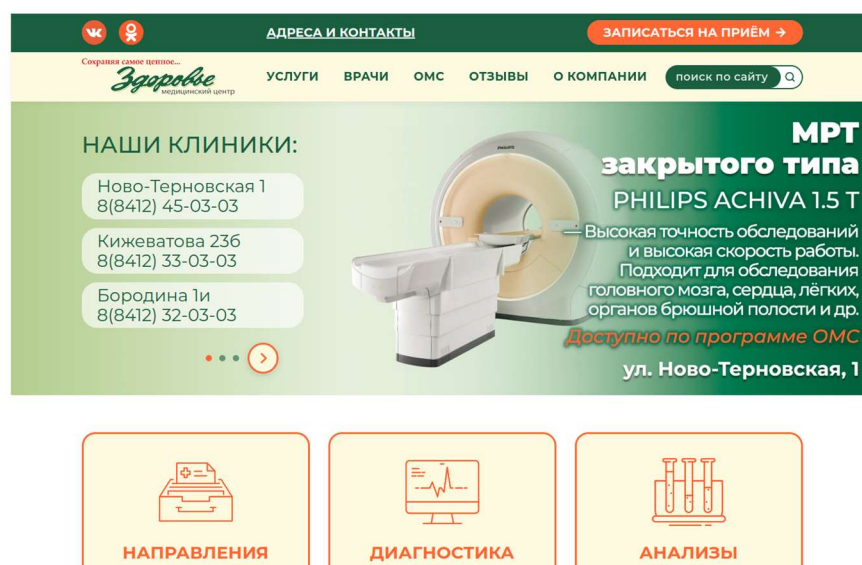


Рисунок 7-Интерфейс веб-приложение клиники «Здоровье»

1.6.3 Веб-приложение медицинской клиники «Клиника Фомина»

Клиника Фомина представляет собой сеть многопрофильных клиник доказательной медицины. На сайте представлены разделы с услугами, врачами, акциями, информацией для пациентов, контактами и сведениями о клинике. Также имеется возможность записи на прием.

Сайт обладает развитой структурой медицинских услуг. Среди направлений представлены гастроэнтерология, гематология, гинекология, неврология, терапия, эндокринология, онкология, урология и другие специальности. Также на сайте есть разделы для женщин, мужчин и детей, что упрощает навигацию пользователя.

Преимуществом сайта Клиники Фомина является широкий набор онлайн-сервисов для пациента. Веб-приложение предоставляет онлайн-консультации, анкету нового пациента, справочник заболеваний и справочник симптомов. В карточках врачей отображаются стаж, специальность, возможность онлайн-приема, формат общения в чате, стоимость и ближайшие даты записи.

Также на странице клиники указаны преимущества, среди которых современное оборудование, контроль этапов лечения с помощью ИИ, привлечение федеральных экспертов и служба заботы о пациентах [12].

Недостатком является то, что веб-приложение не демонстрирует полноценного открытого AI-помощника для первичной маршрутизации пациента в формате диалога. Пользователь может воспользоваться справочником симптомов или онлайн-консультацией, но не представлено решение, которое в интерфейсе сайта анализирует свободный запрос пациента, объясняет возможное направление обращения и формирует структурированные данные для записи к врачу. Интерфейс данного приложения представлен на рисунке 8.

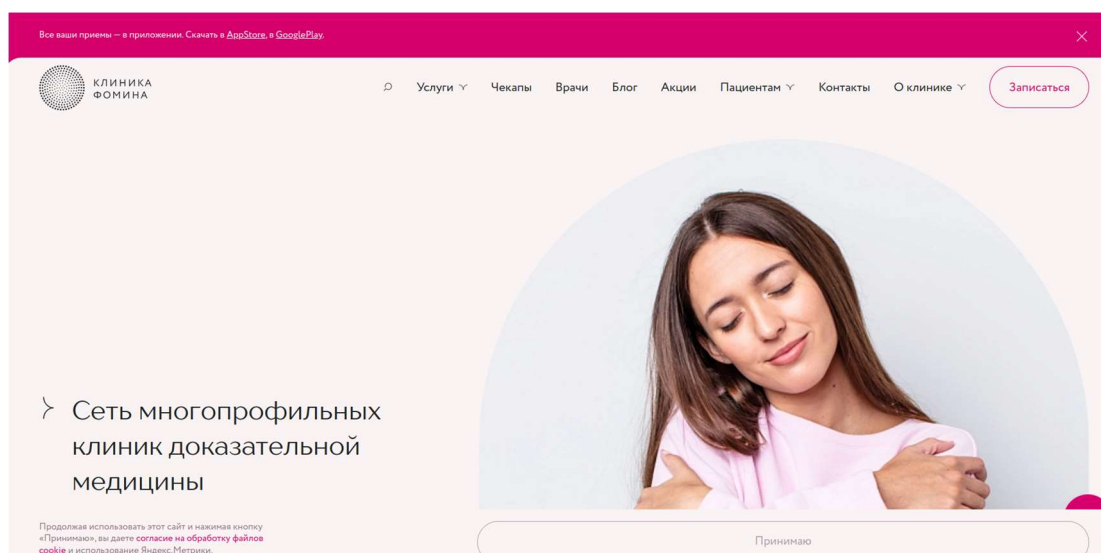


Рисунок 8-Интерфейс веб-приложение клиники «Клиника Фомина»

Пользователь может воспользоваться справочником симптомов или онлайн-консультацией, но не представлено решение, которое в интерфейсе сайта анализирует свободный запрос пациента, объясняет возможное направление обращения и формирует структурированные данные для записи к врачу. Отсутствие такого AI-помощника снижает эффективность первичной маршрутизации и увеличивает нагрузку на call-центр, так как система не преобразует естественный язык пациента в структурированные симптомы.

Таким образом, анализ веб-приложений медицинских клиник «Медгард», «Здоровье» и «Клиника Фомина» показывает, что современные частные медицинские организации уже активно используют цифровые каналы взаимодействия с пациентами. Во всех рассмотренных случаях сайт выполняет не только презентационную, но и сервисную функцию: пользователь может получить информацию о направлениях, врачах, адресах, стоимости услуг, записаться на прием или перейти к дополнительным онлайн-сервисам.

Однако общий недостаток рассмотренных веб-приложений заключается в отсутствии полноценного AI-помощника.

Результат сравнения функционала рассмотренных сервисов представлен в таблице 1.

Таблица 1-Сравнение LLM моделей

Критерий сравнения	Медгард	МЦ «Здоровье»	Клиника Фомина
Онлайн-запись	Есть запись с сайта, мобильного приложения и дополнительных каналов	Есть форма заявки, пользователь оставляет данные для обратной связи	Есть запись к врачу, в том числе по карточкам специалистов
Личный кабинет	Есть личный кабинет с результатами диагностики и записью	Отсутствует	Есть профиль
Карточки врачей	Есть специалисты и направления	Есть раздел врачей и филиалов	Карточки показывают стаж, направление, стоимость и доступность услуг
Каталог услуг	Имеется	Имеется	Имеется
Управление расписанием	Публично видна запись на приём	Публично видна заявка на приём	Публично видна доступность врачей
AI-поддержка	Отсутствует	Отсутствует	Имеется искусственный интеллект в примитивном варианте
Медицинская безопасность	Есть стандартные пользовательские предупреждения и безопасный вход в личный кабинет	Есть дисклеймер о справочном характере материалов	Есть предупреждение о противопоказаниях и обращении к врачу

1.7 Анализ темы

На основании анализа предметной области и рассмотренных аналогов сформулирована задача разработки веб-приложения медицинской клиники с интеграцией AI-помощника. Разрабатываемая система должна быть не только информационной страницей клиники, но и прикладным сервисом, который поддерживает запись на прием, личные кабинеты, административное управление и безопасное справочное сопровождение пользователя.

Объектом разработки является программный комплекс медицинской клиники. Предметом разработки являются клиент-серверная архитектура, серверная бизнес-логика записи на прием, интерфейс пациента, врача и

администратора, а также AI/RAG-модуль, отвечающий на вопросы пользователя с опорой на управляемую базу знаний.

Цель разработки заключается в создании веб-приложения, которое объединяет каталог врачей и услуг, выбор доступного времени, авторизацию пользователей, разграничение ролей и AI-помощника с источниками ответа и метриками качества. В отличие от обычного сайта-витрины, приложение должно иметь серверные правила, исключающие некорректные записи и доступ к чужим данным [13].

Функциональные требования сформированы на основе анализа аналогов и фактических сценариев медицинской клиники. Первое требование – предоставить пользователю публичный каталог услуг и врачей. Пользователь должен видеть название услуги, описание, длительность и стоимость, а также специалистов, которые эту услугу оказывают. Без этой связи запись становится случайной: пациент может выбрать врача, который не работает с выбранным направлением.

Второе требование – реализовать регистрацию и авторизацию. Пациент должен иметь возможность создать учетную запись и сразу получить связанный профиль пациента. Сотрудники клиники работают через заранее созданные учетные записи со специальными ролями.

Третье требование – запись на приём. Пользователь выбирает услугу, врача, дату и свободный слот. Сервис проверяет несколько условий: слот есть в расписании, относится именно к выбранному врачу, ещё не занят, не попадает в прошлое и длится достаточно долго для выбранной услуги. У пациента при этом не должно быть других активных записей, которые пересекаются по времени.

Четвёртое требование – личный кабинет. Пациент видит свои записи и может отменить будущий приём, если это разрешено правилами сервиса. Врач работает только со своими приёмами. Администратор видит все записи, поскольку отвечает за организацию процессов в клинике.

Пятое требование – AI-помощник. Пользователь задаёт вопрос, система ищет подходящие документы в базе знаний, проверяет сообщение на безопасность, формирует ответ и возвращает json. История диалогов сохраняется

за пользователем. Администратор видит общую картину обращений: какие темы чаще поднимают, где возникают повторяющиеся вопросы, как работает поддержка.

Шестое требование – административная панель. Администратор управляет услугами, врачами, расписанием и статьями базы знаний. После изменения базы знаний система должна переиндексировать RAG; без этого AI-помощник продолжит отвечать по старым данным. В панели также нужен журнал действий и возможность вручную запускать переиндексацию.

1.8 Анализ клиент-серверной архитектуры

Для веб-приложения медицинской клиники выбрана клиент-серверная архитектура. Она подходит для этой задачи, потому что система должна одновременно обслуживать пациентов, врачей и администраторов, хранить данные о специалистах, услугах и записях, обрабатывать заявки на приём и работать с AI-помощником. Разделение на клиентскую и серверную части делает разработку понятнее, снижает риски для безопасности и оставляет возможность расширять приложение без полной переработки системы.

Клиентская часть отвечает за то, как пользователь взаимодействует с приложением. Пациент через интерфейс смотрит информацию о клинике, выбирает врача или услугу, записывается на приём и задаёт вопросы AI-помощнику. Врач получает доступ к своим приёмам. Администратор управляет специалистами, услугами, расписанием и заявками. При этом клиентская часть не выполняет ключевые бизнес-операции сама. Она отправляет запросы на сервер через программный интерфейс и получает от него результат.

Серверная часть выступает центром системы. Она принимает запросы от клиента, проверяет права доступа, выполняет логику записи на приём, работает с базой данных и подключает AI-модуль. Такой подход переносит основную логику на сервер и не даёт пользователю напрямую обращаться к базе данных или внешнему AI-сервису. Для медицинского приложения это особенно важно: система обрабатывает персональные данные, поэтому должна контролировать, какие сведения передаются, кому они доступны и как используются.

Для обмена данными между клиентской и серверной частями используется REST API. Клиентская часть отправляет HTTP-запросы к серверу, а сервер возвращает результат в структурированном формате. Такой способ взаимодействия удобен для веб-приложения, так как позволяет отдельно развивать интерфейс и серверную логику.

Важным элементом архитектуры является механизм авторизации. В приложении предусмотрены разные роли пользователей: пациент, врач и администратор. Каждая роль имеет собственный набор доступных действий. Например, пациент может записываться на прием и обращаться к AI-помощнику, врач может просматривать связанные с ним записи, а администратор может управлять врачами, услугами и расписанием. Использование JWT позволяет передавать информацию об авторизованном пользователе между клиентской и серверной частями и ограничивать доступ к защищенным функциям приложения [14].

Для хранения данных используется база данных. В ней размещаются сведения о пользователях, врачах, услугах, доступных временных слотах, записях на прием, базе знаний клиники и журнале действий. Серверная часть взаимодействует с базой данных через слой доступа к данным, что позволяет отделить бизнес-логику от операций хранения. Такой подход повышает надежность системы и упрощает дальнейшее сопровождение приложения.

Отдельное место в архитектуре занимает AI/RAG-слой. Он используется для обработки запросов пользователя к AI-помощнику. При поступлении вопроса пациента серверная часть передает запрос во внутренний AI-слой, где выполняется поиск релевантных фрагментов в базе знаний клиники. После этого найденная информация передается языковой модели вместе с системными правилами медицинской безопасности. Благодаря этому AI-помощник формирует ответ не только на основе общих знаний модели, но и с учетом конкретных данных клиники: перечня услуг, информации о врачах, правил записи и других справочных материалов [18].

Использование RAG-подхода позволяет снизить вероятность недостоверных ответов. Вместо того чтобы полностью полагаться на внутренние знания языковой модели, система передает ей актуальные сведения из базы знаний приложения. Это особенно важно для медицинской предметной области, где AI-помощник не должен выдумывать несуществующие услуги, врачей или условия приема. Ответы AI-помощника должны носить справочный характер и не заменять консультацию врача [18].

Внешний поставщик AI-модели вызывается только через серверную часть приложения. Клиентская часть не обращается к модели напрямую. Такое решение повышает безопасность, так как API-ключи и параметры взаимодействия с AI-сервисом остаются на сервере. Кроме того, backend может предварительно обработать запрос пользователя, удалить избыточные персональные данные, добавить системные ограничения и проконтролировать формат ответа модели [16].

Таким образом, клиент-серверная архитектура соответствует требованиям разрабатываемого веб-приложения медицинской клиники. Она обеспечивает разделение пользовательского интерфейса и серверной логики, безопасную работу с ролями пользователей, централизованное хранение данных, интеграцию с AI-помощником и возможность дальнейшего развития системы. Структура приложения включает клиентскую часть, серверную часть REST API, базу данных, AI/RAG-слой и внешнего поставщика языковой модели. Данная архитектура представлена на рисунке 9.

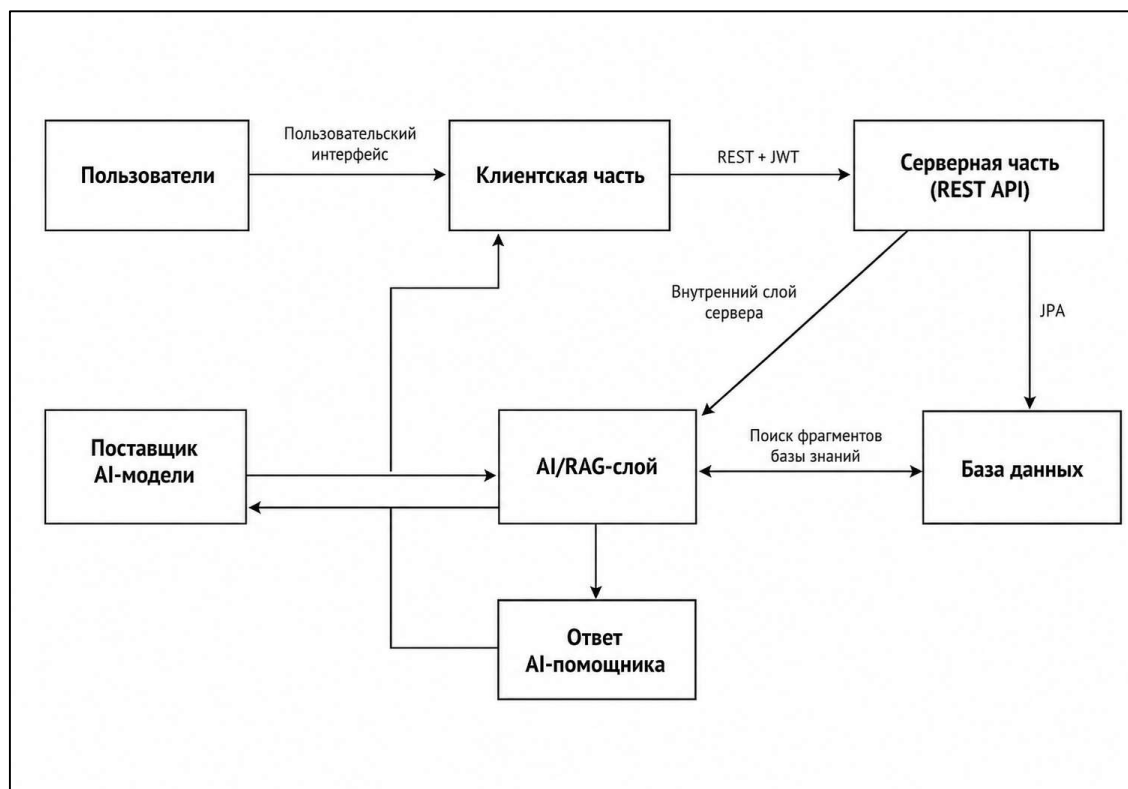


Рисунок 9-Архитектура приложения

1.9 Выбор языка программирования и среды разработки

Для клиентского приложения выбран TypeScript. В отличие от обычного JavaScript, TypeScript позволяет явно описывать структуры данных, которые приходят от серверной части приложения: врача, услугу, слот, запись, пользователя, ответ AI-помощника и метрики качества. Это снижает вероятность ошибок при изменении REST API и делает интерфейс более сопровождаемым [15].

В качестве библиотеки пользовательского интерфейса выбран React. Он подходит для приложения с большим числом состояний: текущий пользователь, выбранная вкладка, выбранная услуга, врач, дата, слот, история AI-диалога, административные формы и уведомления. Для сборки используется Vite, так как он быстро запускает dev-сервер и хорошо работает с TypeScript-проектами [16].

1.10 Выбор фреймворка для работы с искусственным интеллектом

Для AI-модуля выбран подход Retrieval-Augmented Generation. Обычная языковая модель может уверенно отвечать на вопросы, не имея актуальных

сведений о конкретной клинике, расписании, правилах записи и ограничениях сервиса. RAG позволяет сначала найти релевантные фрагменты базы знаний, а затем сформировать ответ с учетом найденного контекста.

На серверной части используется Spring AI. Этот выбор согласуется с серверной частью на Spring Boot: AI-клиент, embedding-клиент и сервисы приложения находятся в одной технологической экосистеме. Для хранения векторных представлений используется PostgreSQL с pgvector, что позволяет не поднимать отдельное векторное хранилище для учебного прототипа.

AI/RAG-подсистема проектируется как контролируемый справочный слой поверх базы знаний клиники. Пользовательский вопрос не отправляется в языковую модель изолированно. Сначала система выполняет поиск релевантных фрагментов, затем формирует контекст и только после этого получает модельную оценку. Такой подход снижает риск ответов, не связанных с материалами клиники.

База знаний хранит статьи о записи, подготовке к анализам, документах, ограничениях AI, тревожных симптомах и маршрутизации к специалистам. При индексировании статьи разбиваются на chunks. Для каждого фрагмента создается вектор числовых представлений через Spring AI EmbeddingModel, после чего он сохраняется в PostgreSQL pgvector.

1.11 Выбор языка программирования и среды разработки

Для серверного приложения выбран язык Java 21 и фреймворк Spring Boot. Java подходит для разработки сервисов с устойчивой типизацией, зрелой экосистемой и большим набором библиотек для web-разработки, безопасности, работы с базой данных и тестирования. Spring Boot упрощает создание REST API, подключение Spring Security, Spring Data JPA, Validation, Flyway и OpenApi.

В качестве СУБД выбран PostgreSQL. Он поддерживает транзакционные данные клиники, индексы, ограничения целостности и расширение pgvector для векторного поиска. Такой выбор позволяет хранить расписание, записи, пользователей, базу знаний и AI-диалоги в единой базе данных [17].

Для локальной разработки backend используется Maven, Docker Compose и Swagger UI. Maven управляет зависимостями и тестами, Docker Compose поднимает PostgreSQL на отдельном порту, а Swagger UI позволяет проверять REST API независимо от frontend. Это делает серверную часть воспроизводимой на другой машине[18].

1.12 Вывод

Таким образом, в данном разделе был проведён анализ предметной области и аналогичных проектов, определена архитектура разрабатываемой системы, а также описан алгоритм взаимодействия пользователей с общей системой.

2 Разработка веб-приложения для медицинской клиники

2.1 Разработка серверной части веб-приложения

2.1.1 Разработка серверного приложения

Серверная часть является центральным элементом приложения, так как именно она хранит доменную модель клиники, проверяет права доступа, выполняет бизнес-правила записи на прием, связывает пользователя с AI-помощником и формирует единый API для клиентского приложения. В проекте используется Java 21 и Spring Boot, что дает возможность построить приложение по слоистой архитектуре: controller принимает HTTP-запросы, service реализует сценарии, repository скрывает детали хранения, а persistence-слой отвечает за преобразование доменных объектов в таблицы базы данных.

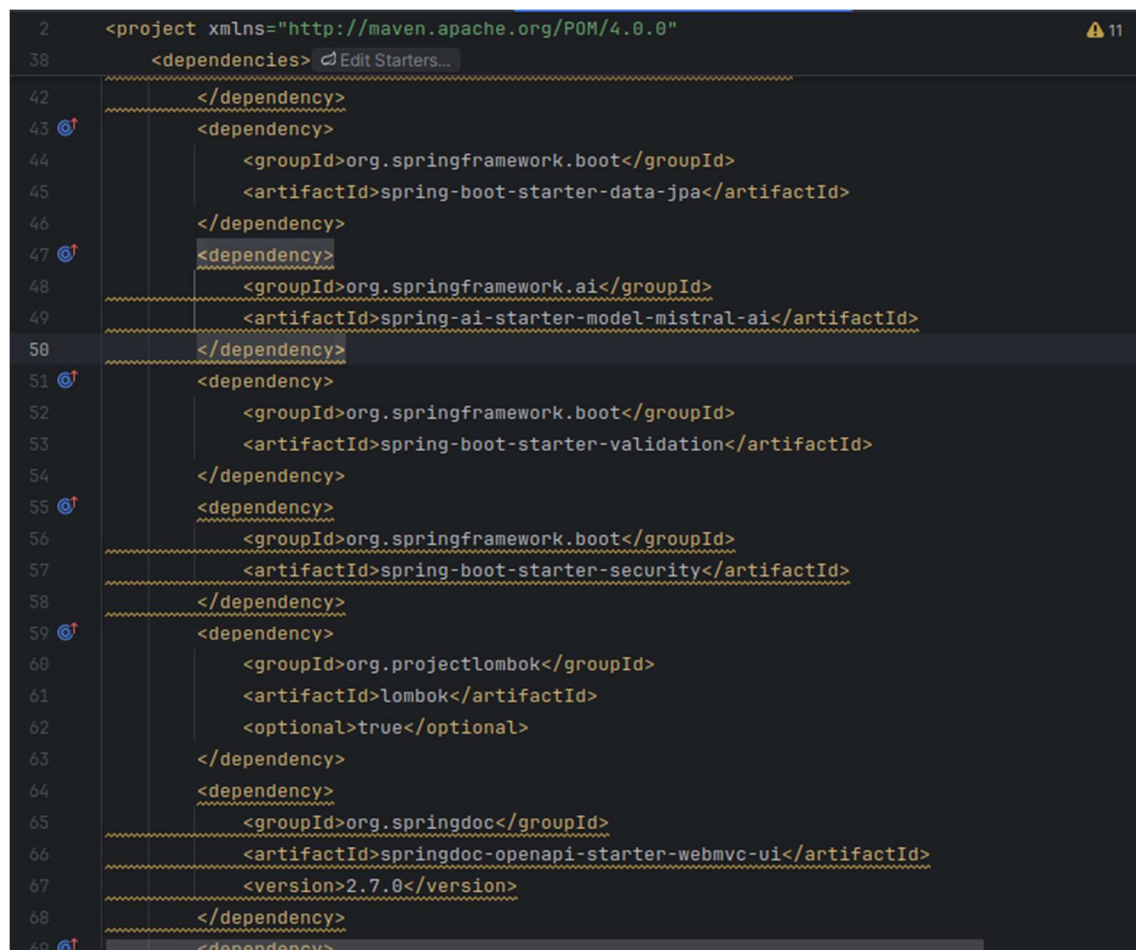
Сервер реализован как REST API. Это соответствует выбранной клиент-серверной архитектуре: клиент обращается к ресурсам врачей, услуг, пациентов, записей, AI-диалогов и административного журнала через HTTP. В результате frontend остается независимым от базы данных и AI-провайдера, а backend сохраняет контроль над безопасностью и валидностью операций.

Основной класс MedicineServiceApplication запускает Spring Boot приложение. После старта контейнер Spring создает контроллеры, сервисы, репозитории, фильтры безопасности и конфигурационные компоненты.

Проект имеет Maven-структуру, что упрощает сборку, подключение зависимостей и выполнение тестов.

В pom.xml подключены web, validation, security, data-jpa, PostgreSQL, Flyway, springdoc-openapi и Spring AI. Такой набор зависимостей соответствует назначению сервиса: web-слой нужен для REST API, validation – для проверки входных DTO, security – для JWT-аутентификации, JPA и PostgreSQL – для хранения медицинского каталога, Flyway – для контролируемых миграций, а Spring AI – для интеграции с LLM.

На рисунке 10 приведён фрагмент зависимостей, показывающий технологический состав серверного приложения.



```
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
38 <dependencies> Edit Starters...
42 </dependency>
43 <dependency>
44 <groupId>org.springframework.boot</groupId>
45 <artifactId>spring-boot-starter-data-jpa</artifactId>
46 </dependency>
47 <dependency>
48 <groupId>org.springframework.ai</groupId>
49 <artifactId>spring-ai-starter-model-mistral-ai</artifactId>
50 </dependency>
51 <dependency>
52 <groupId>org.springframework.boot</groupId>
53 <artifactId>spring-boot-starter-validation</artifactId>
54 </dependency>
55 <dependency>
56 <groupId>org.springframework.boot</groupId>
57 <artifactId>spring-boot-starter-security</artifactId>
58 </dependency>
59 <dependency>
60 <groupId>org.projectlombok</groupId>
61 <artifactId>lombok</artifactId>
62 <optional>true</optional>
63 </dependency>
64 <dependency>
65 <groupId>org.springdoc</groupId>
66 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
67 <version>2.7.0</version>
68 </dependency>
69 </dependencies>
```

Рисунок 10-Зависимости проекта

2.1.2 Проектирование доменной модели

Доменная модель описывает основные объекты клиники: пациента, врача, медицинскую услугу, свободный слот, запись на прием, пользователя системы, статью базы знаний и AI-диалог. Выделение этих сущностей

позволяет построить не просто чат с AI, а полноценное веб-приложение, где AI связан с реальными действиями клиники и подчиняется тем же ограничениям доступа, что и обычные функции.

В проекте `ApplicationUserEntity` хранит учетные записи и роль, `PatientEntity` связывает пользователя с пациентом, `DoctorEntity` описывает врача и его специальность, `MedicalServiceEntity` задает услугу, `AppointmentSlotEntity` содержит доступные интервалы времени, `AppointmentEntity` фиксирует конкретную запись. `KnowledgeArticleEntity` и `KnowledgeArticleChunk` применяются для базы знаний, которая затем индексируется в векторном хранилище.

Такая схема снижает риск неконсистентности. Например, AI не должен выдумывать цену услуги или кабинет врача: эти сведения берутся из таблиц `medical_services` и `doctors`. Если данных в справочнике нет, корректным поведением является не генерация предположения, а сообщение о необходимости уточнения у администратора. Для наглядного представления структуры хранения данных на рисунке 11 представлена ER-диаграмма базы данных PostgreSQL, отражающая таблицы приложения, их атрибуты, первичные и внешние ключи, а также связи между основными сущностями системы.

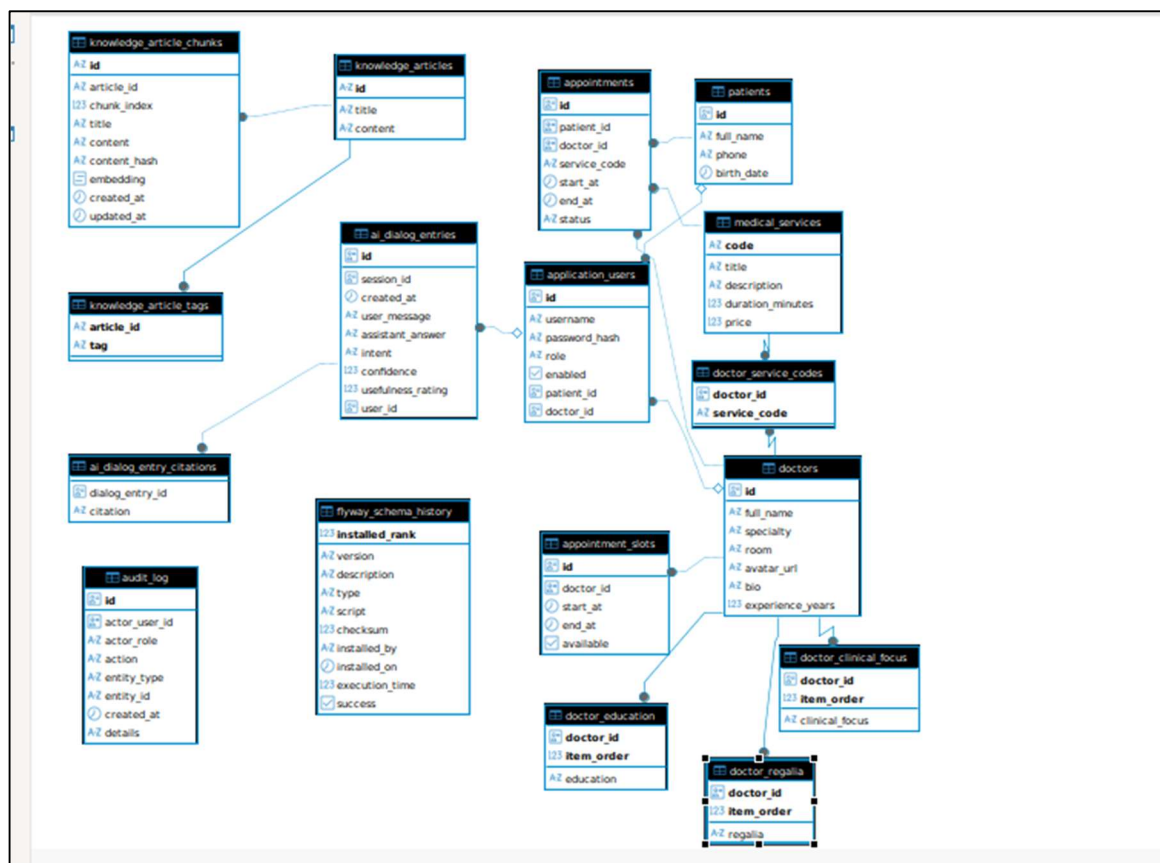


Рисунок 11-ER-диаграмма базы данных приложения

2.1.3 Реализация REST API

Для взаимодействия клиентского приложения с сервером был разработан REST API. Клиент отправляет HTTP-запросы, а сервер возвращает ответы в формате JSON. Такой подход позволяет независимо развивать пользовательский интерфейс и серверную логику. В приложении выделены программные точки доступа для авторизации, каталога, записи на прием, административного управления и AI-помощника. Пример такого контроллера представлен на рисунке 12.

```

@RequestMapping(value = "/api")
public class CatalogController {

    private final CatalogService catalogService; // 9 usages
    private final DoctorAvatarService doctorAvatarService; // 2 usages

    public CatalogController(CatalogService catalogService, DoctorAvatarService doctorAvatarService) {
        this.catalogService = catalogService;
        this.doctorAvatarService = doctorAvatarService;
    }

    @GetMapping(value = "/doctors")
    public Collection<Doctor> doctors() { return catalogService.doctors(); }

    @GetMapping(value = "/doctors/{id}")
    public Doctor doctor(@PathVariable("id") UUID id) { return catalogService.doctor(id); }

    @GetMapping(value = "/doctors/{id}/avatar", produces = "image/svg+xml;charset=UTF-8")
    public ResponseEntity<String> doctorAvatar(@PathVariable("id") UUID id) {
        return ResponseEntity.ok()
            .contentType(new MediaType("image", "svg+xml", StandardCharsets.UTF_8))
            .body(doctorAvatarService.generateSvg(catalogService.doctor(id)));
    }

    @GetMapping(value = "/services")
    public Collection<MedicalService> services() { return catalogService.services(); }
}

```

Рисунок 12-CatalogController

Контроллер не содержит сложной бизнес-логики. Его задача состоит в приеме запроса, вызове соответствующего сервиса и возврате результата. Это упрощает сопровождение кода и позволяет менять внутреннюю реализацию сервиса без изменения контракта API.

2.1.4 Реализация модуля записи на приём

Один из основных сценариев приложения – запись пациента на прием к врачу. Для выполнения данной операции необходимо проверить существование пациента, врача, услуги и свободного слота, а также убедиться, что выбранный врач оказывает нужную услугу. После успешной проверки создается запись и выбранный слот помечается как занятый. Код данного метода представлен на рисунке 13.

```

public Appointment create(CreateAppointmentRequest request, AuthenticatedUser actor) {
    patientRepository.findById(request.patientId())
        .orElseThrow(() -> new NotFoundException("Пациент не найден: " + request.patientId()));

    Doctor doctor = doctorRepository.findById(request.doctorId())
        .orElseThrow(() -> new NotFoundException("Доктор не найден: " + request.doctorId()));

    MedicalService medicalService = medicalServiceRepository.findByCode(request.serviceCode())
        .orElseThrow(() -> new NotFoundException("Сервис не найден: " + request.serviceCode()));

    if (!doctor.serviceCodes().contains(medicalService.code())) {
        throw new BusinessException("Данный врач не оказывает услуги с кодом " + medicalService.code());
    }

    var decision = appointmentBusinessPolicy.validateNewAppointment(request.startAt(), appointmentRepository.findById(request.patientId()));
    if (!decision.allowed()) {
        throw new BusinessException(decision.reason());
    }

    AppointmentSlot slot = slotRepository.findByDoctorIdAndStartAt(doctor.id(), request.startAt())
        .orElseThrow(() -> new NotFoundException("Schedule slot not found for selected doctor and time"));

    if (!slot.doctorId().equals(doctor.id())) {
        throw new BusinessException("Слот не принадлежит выбранному врачу");
    }

    if (!slot.available()) {
        throw new BusinessException("Данный слот уже занят");
    }
}

```

Рисунок 13-Метод создания записи на приём

Аннотация `@Transactional` гарантирует выполнение метода как единого целого. Если на одном из этапов произойдет ошибка, все изменения, уже сделанные в этой функции, откатятся назад. Это особенно важно для расписания клиники, так как нельзя допустить ситуации, чтобы запись была отменена или создана, но статус остался неизменным.

2.1.5 Реализация аутентификации и ролевой модели

В веб-приложении предусмотрены роли пользователя: пациент, врач или администратор. Ролевая модель нужна не только для интерфейса, но и для защиты API: скрытая кнопка в клиентской части приложения не является защитой, если серверная часть не проверяет права.

Авторизация построена на JWT. После успешного входа клиент сохраняет токен доступа и добавляет его в заголовок `Authorization`. Серверный фильтр извлекает токен, проверяет подпись и срок действия, формирует объект `Authentication` и передает запрос дальше. Если токен отсутствует или просрочен, защищённые endpoint возвращают ошибку авторизации.

Решение соответствует принципу минимальных привилегий: пациент может смотреть свои записи и диалоги, администратор — редактировать справочники и видеть логи, а публичные методы каталога доступны без входа. Это позволяет использовать сайт как витрину клиники, но защищать персональные и административные данные.

2.1.6 Логирование и аудит

В медицинском веб-приложении важна трассировка. Пользовательские действия, административные операции и обращения к AI фиксируются в журнале аудита. Это не означает хранение лишних медицинских данных, но позволяет понять, кто создал запись, кто изменил справочник, кто запустил переиндексацию RAG и какие события происходили перед ошибкой.

Аудит особенно полезен при работе с AI, потому что ответ модели может зависеть от контекста. Сохранение исходного сообщения пользователя, сформированного ответа AI-помощника, распознающего намерения пользователя, уровня уверенности системы и списка использованных источников обеспечивает возможность последующего анализа диалога, проверки достоверности ответа и доработки базы знаний приложения. При этом необходимо разграничить права доступа к истории диалогов: пациенту доступны только его собственные обращения, а администратор получает обобщенную информацию, необходимую для контроля работы сервиса и его сопровождения.

Журналирование также помогает при эксплуатации. Если пользователи часто задают вопросы, на которые RAG отвечает с низкой уверенностью, это признак недостаточной базы знаний. Администратор может добавить новые статьи, переиндексировать базу и снова проверить качество. Поля аудита представлены на рисунке 14

```
public record AuditLogEntry( 17 usages
    UUID id, 1 usage
    UUID actorUserId, 1 usage
    UserRole actorRole, 1 usage
    String action, 1 usage
    String entityType, 1 usage
    String entityId, 1 usage
    LocalDateTime createdAt, 1 usage
    String details 1 usage
) {
}
```

Рисунок 14-Сущность AuditLogEntry

2.1.7 Развёртывание серверного приложения

Развёртывание серверной части включает подготовку PostgreSQL, настройку переменных окружения, запуск миграций Flyway, сборку Maven и старт приложения. Для локальной разработки достаточно поднять контейнер PostgreSQL с pgvector, указать параметры подключения к источнику данных и запустить Spring Boot приложение из IDE или через Maven.

После запуска проверяется Swagger UI, состояние подключения к базе данных, методы, авторизация и AI-модуль. Если хранилище RAG пусто, необходимо выполнить переиндексацию и убедиться, что количество проиндексированных чанков больше нуля.

2.2 Разработка клиентской части веб-приложения

Клиентская часть реализована на React и Vite. Она обеспечивает навигацию по разделам клиники, отображение услуг и врачей, регистрацию и вход пациента, онлайн-запись, личный кабинет, AI-помощника и административную панель.

Выбор React оправдан его компонентной моделью. Страницы и формы можно разделить на независимые компоненты: шапка сайта, главная страница, каталог, страница врачей, бронирование, ассистент, личный кабинет и панель администратора. Состояние авторизации и загруженные данные хранятся на уровне приложения, а слой работы с API скрывает детали HTTP-запросов.

Frontend не принимает критических решений: он отправляет запросы и показывает ответы. Все проверки записи, прав доступа, владельца диалога и безопасности AI выполняются на сервере. Это правильное распределение ответственности, потому что клиентский код можно изменить в браузере, а сервер остается доверенной стороной.

2.2.1 Структура клиентской части

Структура клиентской части минимальна и понятна: `src/main.tsx` содержит основное React-приложение, `src/api/client.ts` – HTTP-клиент и хранение токена, `src/types/models.ts` – TypeScript-модели, `src/lib/format.ts` –

форматирование дат, цен, статусов и ролей, styles.css – визуальное оформление.

В package.json определены сценарии dev, build, preview и lint. Сценарий build запускает TypeScript и Vite, что позволяет обнаружить типовые ошибки до развертывания. Сценарий lint через tsc --noEmit полезен для проверки типов без сборки.

2.2.2 Реализация API-клиента

API-клиент является прослойкой между React-компонентами и серверной частью. Он задаёт базовый URL, добавляет заголовок Authorization, обрабатывает ошибки, очищает токен при ошибке 401 и предоставляет удобные методы: doctors, services, slots, createAppointment, askAi, ragStatus, reindex, audit и другие.

Такое решение уменьшает дублирование кода. Компоненты не формируют URL вручную и не повторяют обработку fetch. Если изменится адрес серверной части или формат авторизации, достаточно изменить файл client.ts. Кроме того, единая обработка ошибок позволяет показывать пользователю понятные сообщения вместо технической ошибки.

Особенно важен метод askAi. Он отправляет вопрос пользователя на endpoint /api/ai/ask и передаёт sessionId при его наличии. Благодаря этому AI-помощник может сохранять контекст диалога и отличать одну сессию от другой. Единая функция HTTP-запроса представлена на рисунке 15.

```
export const authStore = { Show usages
  getToken: () : string | null => localStorage.getItem(TOKEN_KEY),
  setToken: (token: string) : void => localStorage.setItem(TOKEN_KEY, token),
  clear: () : void => localStorage.removeItem(TOKEN_KEY)
};

async function request<T>({path: string, options: RequestInit = {}}: Promise<T> { Show usages
  const headers = new Headers(options.headers);
  const token : string | null = authStore.getToken();
  if (!headers.has('Content-Type') && options.body) headers.set('Content-Type', 'application/json');
  if (token) headers.set('Authorization', 'Bearer ${token}');

  const response : Response = await fetch(`${API_URL}${path}`, { ...options, headers, credentials: API_CREDENTIALS });
  if (!response.ok) {
    let body : unknown = null;
    try { body = await response.json(); } catch { body = await response.text(); }
    if (response.status === 401 && path !== '/auth/login') {
      authStore.clear();
      window.dispatchEvent(new CustomEvent(AUTH_EXPIRED_EVENT));
    }
    const message : string = typeof body === 'object' && body && 'message' in body ? String((body as { message: unknown }).message) : `Ошибка API ${response.status}`;
    throw new ApiError(message, response.status, body);
  }
  if (response.status === 204) return undefined as T;
  return response.json() as Promise<T>;
}
```

Рисунок 15-Функция HTTP-запроса

2.2.3 Навигация и основные разделы интерфейса

Интерфейс содержит несколько разделов: главную страницу, каталог услуг, список врачей, карточку врача, запись на приём, AI-помощника, личный кабинет и административную панель. Переключение между разделами реализовано через состояние `tab`. Для отображения карточки врача используется хэш-навигация формата `#doctor/{id}`. Благодаря этому карточку конкретного врача можно открыть по прямой ссылке, не внедряя полноценную систему маршрутизации.

Главная страница выполняет роль витрины: показывает статистику, услуги и врачей. Каталог позволяет пользователю быстро понять, какие услуги доступны, а раздел врачей – выбрать подходящего специалиста. Запись на приём и личный кабинет доступны только после входа, что соответствует требованиям защиты персональных данных.

Важной деталью является режим повышенной контрастности. Он не заменяет полноценную реализацию доступности, но демонстрирует учёт потребностей пользователей с нарушениями зрения. Для медицинского сервиса это особенно актуально, потому что пользователи могут обращаться к приложению в состоянии дискомфорта.

2.3 Разработка AI-модуля

2.3.1 Разработка AI-контроллера

Для взаимодействия клиентского приложения с интеллектуальным помощником был разработан контроллер `AiAssistantController`. Он предоставляет `api` для отправки пользовательского вопроса, просмотра истории диалога, оценки полезности ответа, проверки состояния индекса базы знаний и запуска повторной индексации. Листинг представлен на рисунке 16.

```

@PostMapping("/{ask}")
public AiAssistantResponse ask(@RequestParam(name = "sessionId", required = false) UUID sessionId,
    @Valid @RequestBody AskAiRequest request,
    Authentication authentication) {
    return aiAssistantService.ask(securityPrincipalService.requireUser(authentication), sessionId, request.message());
}

@GetMapping("/{rag/index}")
public AiRagIndexStatus ragIndexStatus() { return aiAssistantService.ragIndexStatus(); }

@PostMapping("/{rag/reindex}")
public AiRagIndexStatus reindexRag() { return aiAssistantService.reindexRag(); }

@GetMapping("/{evaluation/cases}")
public List<AiEvaluationCase> evaluationCases() { return aiEvaluationService.cases(); }

@PostMapping("/{evaluation/run}")
public AiEvaluationReport runEvaluation(Authentication authentication) {
    return aiEvaluationService.run(securityPrincipalService.requireUser(authentication));
}

@GetMapping("/{dialog}")
public Collection<AiDialogEntry> dialogEntries(Authentication authentication) {
    AuthenticatedUser user = securityPrincipalService.requireUser(authentication);
    if (securityPrincipalService.hasRole(user, UserRole.ADMIN)) {
        return aiAssistantService.dialogEntries();
    }
    return aiAssistantService.dialogEntries(user);
}

```

Рисунок 16-AiAssistantController

Параметр sessionId в методе api/ai/ask позволяет объединять несколько сообщений пользователя в один диалог. Если идентификатор сессии отсутствует, он создается автоматически в сервисном слое. Благодаря этому AI-помощник может учитывать предыдущие сообщения пользователя, а история обращения сохраняется в базе данных. Доступ к истории диалогов организован с учетом ролей пользователей представлен на рисунке 17. Пациент может просматривать только собственные сессии взаимодействия с AI-помощником, тогда как администратору предоставляется сводная информация, позволяющая оценивать качество работы сервиса и своевременно выявлять проблемные обращения.

```

@GetMapping("/{dialog}")
public Collection<AiDialogEntry> dialogEntries(Authentication authentication) {
    AuthenticatedUser user = securityPrincipalService.requireUser(authentication);
    if (securityPrincipalService.hasRole(user, UserRole.ADMIN)) {
        return aiAssistantService.dialogEntries();
    }
    return aiAssistantService.dialogEntries(user);
}

```

Рисунок 17-Разграничение доступа к истории AI-диалогов

2.3.2 Реализация сервисного слоя AI-помощника

Основная логика AI-модуля сосредоточена в сервисе AiAssistantService. Данный сервис связывает между собой репозиторий истории диалогов, модуль

поиска по базе знаний, клиент языковой модели, политику медицинской безопасности, аудит действий и расчет метрик качества ответа. Последовательность обработки запроса представлена на рисунке 18. Код метода обработки сообщений представлен на рисунке 19.



Рисунок 18-Последовательность обработки запроса AI-помощником

```

public AiAssistantResponse ask(AuthenticatedUser user, UUID sessionId, String message) {
    medicalSafetyPolicy.validateMessage(message);
    UUID effectiveSessionId = sessionId == null ? UUID.randomUUID() : sessionId;
    List<AiDialogEntry> dialogContext = aiDialogRepository.findBySessionIdAndUserId(effectiveSessionId, user.id());
    List<AiRagService.ScoredKnowledgeArticle> ragMatches = aiRagService.retrieve(message);
    String ragContext = aiRagService.contextBlock(ragMatches);

    AiTriageAssessment assessment = medicalSafetyPolicy.immediateDecision(message)
        .orElseGet(() -> safeModelAssessment(message, ragContext, dialogContext));
    List<AiCitation> citationDetails = citationDetailsFromAssessment(assessment, ragMatches);
    String answer = buildAnswer(assessment);
    var qualityMetrics = aiQualityService.calculate(message, answer, assessment, ragMatches);
    var sourceSnippets = aiQualityService.sourceSnippets(ragMatches);

    AiAssistantResponse response = saveResponse(
        user.id(),
        effectiveSessionId,
        message,
        assessment,
        answer,
        citationDetails,
        sourceSnippets,
        qualityMetrics
    );
    auditLogService.record(user, action: "AI_ASSISTANT_ASKED", entityType: "AiDialogEntry", response.dialogEntryId().toString());
}
  
```

Рисунок 19-Основной метод обработки сообщения пользователя

В начале метода выполняется проверка пользовательского сообщения. Затем определяется активная сессия диалога и загружается контекст предыдущих сообщений. После этого AI-модуль извлекает релевантные фрагменты базы знаний и формирует контекст для языковой модели. Если правила безопасности обнаружили тревожные симптомы, ответ формируется без обращения к модели.

2.3.3 Реализация модуля медицинской безопасности

AI-помощник медицинского приложения не должен ставить диагноз, назначать лечение, подбирать дозировки препаратов или заменять врача. Для этого разработан компонент MedicalSafetyPolicy. Он выполняет

предварительную проверку сообщения и способен принять безопасное решение без обращения к языковой модели. Код представлен на рисунке 20.

```
Service 2 usages
public class MedicalSafetyPolicy {

    private static final List<String> EMERGENCY_MARKERS = List.of( 1 usage
        ...elements: "боль в груди", "сдавило грудь", "давит в груди", "тяжело дышать", "трудно дышать", "задыхаюсь",
        "потеря сознания", "обморок", "кровотечение", "онемела рука", "перекосило лицо", "нарушилась речь",
        "инсульт", "инфаркт", "судорог", "отекали губ", "отек губ", "отек лица", "отек язык", "отек горл"
    );
}
```

Рисунок 20-Фрагмент списка тревожных симптомов

Данный список содержит маркеры состояний, при которых пользователю нельзя давать обычный справочный ответ. Если подобные признаки обнаружены в сообщении, система рекомендует срочно обратиться за очной медицинской помощью. Код представлен на рисунке 21.

```
public Optional<AiTriageAssessment> immediateDecision(String message) { 1 usage
    String normalized = normalize(message);
    List<String> detectedEmergency = EMERGENCY_MARKERS.stream().filter(normalized::contains).toList();
    if (!detectedEmergency.isEmpty()) {
        return Optional.of(new AiTriageAssessment(
            AiIntent.EMERGENCY_TRIAGE,
            alarmingSymptomsDetected: true,
            confidence: 0.99,
            detectedEmergency,
            reasoning: "Сработало правило безопасности: обнаружены красные флаги, требующие срочной очной помощи.",
            answer: "По описанию есть признаки, при которых нельзя ограничиваться онлайн-консультацией. Обратитесь в экстренную службу или ближайшую больницу.",
            List.of("Позвонить в экстренную службу", "Не откладывать очную помощь", "Не заниматься самолечением")
        ));
    }
}
```

Рисунок 21-Формирование безопасного ответа при тревожных симптомах

2.3.4 RAG как основа медицинского помощника

Retrieval-Augmented Generation выбран как базовый подход, потому что медицинский помощник должен отвечать не только на основании параметров модели, но и на основании актуальных правил клиники. В рамках приложения база знаний содержит регламенты, шаблоны безопасных ответов, правила маршрутизации пациента, подготовку к исследованиям, ограничения AI и сценарии администратора.

RAG-процесс состоит из двух частей. Первая часть разбивает статьи на фрагменты, получает векторные представления и сохраняет их в базу данных. Вторая часть получает вопрос пользователя, преобразует его в вектор, ищет ближайшие фрагменты, фильтрует их по релевантности, добавляет в запрос и заставляет модель отвечать только в пределах найденного контекста.

В проекте дополнительно применяется лексическая проверка. Это важно для русского языка и медицинских терминов: чистый семантический поиск может вернуть близкий по смыслу, но не целевой документ. Поэтому итоговая релевантность учитывает сходство векторных представлений, лексические и точные совпадения. Схема работы RAG представлена на рисунке 22 [18].

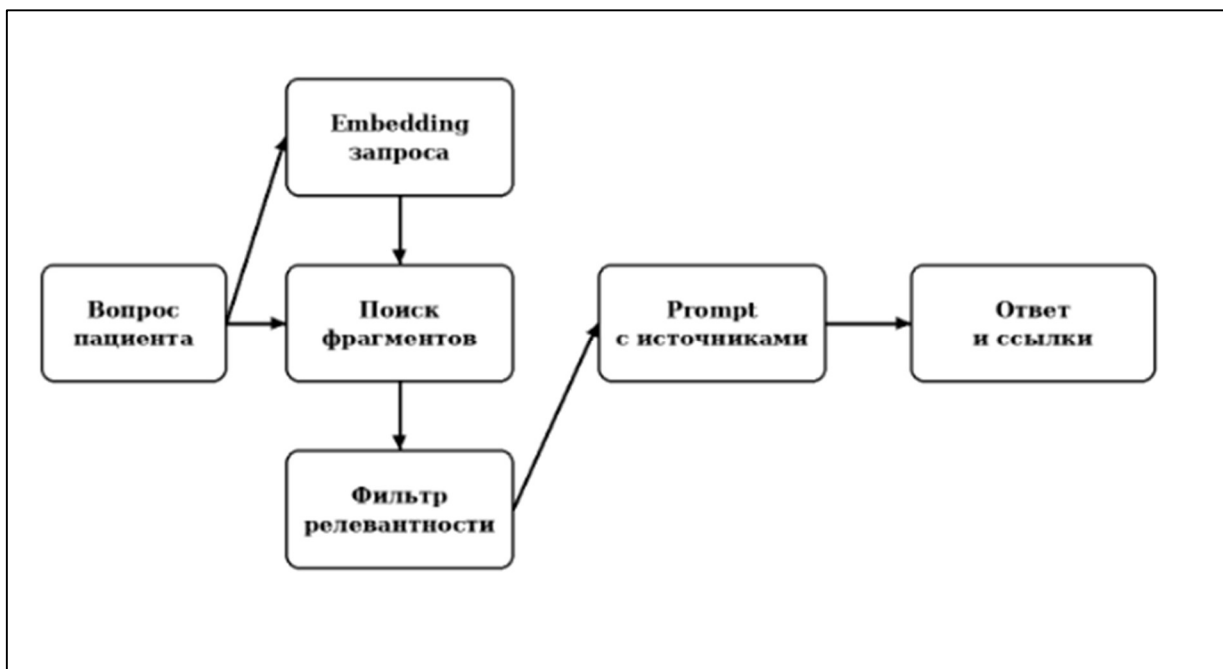


Рисунок 22-Схема RAG

2.3.5 Разработка поиска по базе знаний клиники

Для повышения достоверности ответов AI-помощник использует базу знаний клиники. В нее входят сведения о врачах, услугах, подготовке к процедурам, правилах записи, ограничениях AI-помощника и признаках срочного обращения. Поиск релевантных фрагментов реализован в сервисе AiRagService. Его задача заключается в том, чтобы найти материалы, наиболее близкие по смыслу к сообщению пользователя, и передать их языковой модели как контекст (рисунок 23).

```

public List<ScoredKnowledgeArticle> retrieve(String message) { 2 usages
    if (!knowledgeVectorRepository.hasIndexedChunks()) {
        throw new AIServiceException("Индекс базы знаний RAG пуст. Выполните переиндексацию embeddings.", null);
    }

    float[] embedding = embeddingClient.embed(message);
    Set<String> queryTerms = searchTerms(message);
    Set<String> exactQueryTerms = exactSearchTerms(message);
    int candidateLimit = Math.max(maxArticles * 250, 1000);

    return knowledgeVectorRepository.findNearest(embedding, candidateLimit).stream() Stream<KnowledgeArticleChunk>
        .filter(chunk -> chunk.similarity() >= minSimilarity)
        .map(chunk -> toScoredArticle(chunk, queryTerms, exactQueryTerms)) Stream<ScoredKnowledgeArticle>
        .filter(match -> match.lexicalMatches() > 0)
        .filter(match -> match.titleMatches() > 0 || match.lexicalMatches() >= 2 || match.relevance() >= minRelevance)
        .sorted(Comparator.comparingInt(ScoredKnowledgeArticle::relevance).reversed())
        .thenComparingInt(match -> sourcePriority(match.article().id()))
        .thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::exactTitleMatches).reversed())
        .thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::titleMatches).reversed())
        .thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::lexicalMatches).reversed())
        .limit(maxArticles)
        .toList();
}

```

Рисунок 23-Поиск релевантных фрагментов базы знаний

В данном методе объединяются два подхода: смысловой поиск по числовым векторным представлениям текста и дополнительная лексическая фильтрация. Смысловой поиск помогает находить похожие по содержанию фрагменты, а лексическая фильтрация снижает вероятность выбора нерелевантного источника.

2.3.6 Индексация базы знаний

Индексация базы знаний выполняется отдельным компонентом KnowledgeEmbeddingIndexer. Он проходит по статьям, разбивает их на смысловые фрагменты, преобразует каждый фрагмент в числовое векторное представление и сохраняет результат в базе данных. Такой индекс не должен обновляться синхронно при каждом пользовательском вопросе. Если выполнять векторизацию всей базы знаний во время диалога, приложение станет медленным и будет расходовать больше вычислительных ресурсов.

Поэтому индекс строится заранее, а во время обработки пользовательского запроса выполняется только векторизация сообщения и поиск ближайших фрагментов (рисунок 24). Такой подход делает AI-помощника достаточно быстрым для интерактивного веб-приложения.

```

@GetMapping("/{rag/index}")
public AiRagIndexStatus ragIndexStatus() { return aiAssistantService.ragIndexStatus(); }

@PostMapping("/{rag/reindex}")
public AiRagIndexStatus reindexRag() { return aiAssistantService.reindexRag(); }

```

Рисунок 24-контроль и запуск индексации базы знаний

Статус индекса показывает, сколько фрагментов уже проиндексировано, сколько их всего, есть ли ошибки, когда база знаний обновлялась в последний раз и запущена ли индексация сейчас. Эти данные делают работу AI-модуля прозрачнее: администратор видит, актуальна ли база знаний, завершилась ли переиндексация и не возникли ли сбои в процессе.

2.3.7 Интеграция с LLM

Для работы с языковой моделью используется компонент SpringAiClinicalTriageClient. Он отправляет в модель сообщение пользователя, контекст из базы знаний и историю текущего диалога. Отдельную роль играет системная инструкция: она задаёт границы поведения модели и не позволяет ей отвечать в свободной форме.

В ответ модель должна вернуть не обычный текст, а структурированный JSON-объект. Серверная часть может проверить такой ответ, разобрать его по полям и безопасно использовать в дальнейшей логике приложения. Запрос, составленный для языковой модели, представлен на рисунке 25.

```

private static final String SYSTEM_PROMPT = """ 1usage
Ты медицинский AI-помощник для информационной системы клиники.
Твоя задача - классифицировать обращение пациента, отметить тревожные симптомы и дать безопасный короткий ответ.

Правила безопасности:
- Не ставь диагноз, не назначай лечение, дозировки, препараты или процедуры.
- Не успокаивай пациента при красных флагах и не предлагай ждать при потенциально опасном состоянии.
- Если есть боль в груди, нарушение дыхания, потеря сознания, сильное кровотечение, признаки инсульта или инфаркта, внезапное онемение, тяжела
- Если alarmingSymptomsDetected = true, intent сторого EMERGENCY_TRIAGE, alarmingSymptoms не пустой, recommendedActions содержит срочное очное
- answer формируй только по RAG-контексту клиники и истории диалога. Если данных мало, скажи, что нужно уточнить вопрос или обратиться к админ
- Для неэкстренных медицинских жалоб не придумывай диагноз: объясни общий маршрут по базе знаний клиники.

Верни строго один JSON-объект без markdown, комментариев и дополнительного текста.
Все поля обязательны. Схема:
{
  "intent": "APPOINTMENT_BOOKING | KNOWLEDGE_LOOKUP | EMERGENCY_TRIAGE | ADMIN_ROUTING | UNKNOWN",
  "alarmingSymptomsDetected": true | false,
  "confidence": number от 0 до 1,
  "alarmingSymptoms": ["строка"],
  "reasoning": "короткое объяснение на русском",
  "answer": "короткий безопасный ответ пациенту на русском",
  "recommendedActions": ["строка"]
}
""";

```

Рисунок 25-Системная инструкция для языковой модели

Такая инструкция задает ограничения медицинской безопасности и делает ответ модели пригодным для машинной обработки. Серверная часть может проверить распознанное намерение пользователя, степень уверенности модели, факт обнаружения тревожных симптомов и перечень дальнейших действий.

После получения ответа система разбирает JSON-структуру и проверяет, соответствует ли она ожидаемому формату. Если модель возвращает некорректные данные или временно не отвечает, приложение переходит к резервному безопасному сценарию.

Для медицинского сервиса это критично. Пользователь не должен видеть техническую ошибку или сырой ответ модели. Вместо этого система должна выдать понятное сообщение с безопасной рекомендацией и сохранить предсказуемое поведение даже при сбое AI-модуля.

2.3.8 Формирование итогового ответа AI-помощника

После ответа языковой модели серверная часть собирает итоговое сообщение для пользователя. Сначала она учитывает результат проверки: есть ли в обращении тревожные симптомы или нет.

Если система обнаружила тревожные признаки, ответ формируется по безопасному шаблону. В нём пользователь получает не развернутую консультацию, а осторожную рекомендацию обратиться за медицинской помощью. Если тревожных симптомов нет, система возвращает справочную информацию и добавляет маршрутизационную подсказку – например, к какому специалисту можно записаться или какой раздел сервиса открыть дальше(рисунок 26).

```

private String buildAnswer(AiTriageAssessment assessment) { 1 usage
    if (assessment.alarmingSymptomsDetected()) {
        return "По описанию возможна ситуация, требующая срочной медицинской помощи. "
            + "Обратитесь в экстренную службу или ближайшее приемное отделение. "
            + symptomText(assessment.alarmingSymptoms());
    }

    if (assessment.answer() != null && !assessment.answer().isBlank()) {
        return assessment.answer().strip() + " " + routingHint(assessment.intent());
    }

    return "Я могу помочь с записью на прием, расписанием врачей, подготовкой к анализам и общими правилами работы клиники. "
        + "Сформулируйте вопрос как справочный или организационный.";
}

```

Рисунок 26-Формирование итогового ответа AI-помощника

2.3.9. Сохранение истории AI-диалогов и источников

Для анализа работы AI-помощника и последующего улучшения базы знаний в системе сохраняется история взаимодействия пользователя с AI-модулем. Сохранение исходного сообщения пользователя, ответа AI-помощника, распознанного намерения, степени уверенности и использованных источников позволяет воспроизвести ситуацию, проверить корректность ответа и улучшить базу знаний (рисунок 27).

```

AiDialogEntry entry = aiDialogRepository.save(new AiDialogEntry(
    UUID.randomUUID(),
    userId,
    sessionId,
    LocalDateTime.now(),
    message,
    answer.strip(),
    assessment.intent(),
    assessment.confidence(),
    citations,
    usefulnessRating: null
));

```

Рисунок 27 – Сохранение результата работы AI-помощника

Дополнительно в ответ клиентскому приложению передаются источники и фрагменты базы знаний, использованные при формировании ответа. Это повышает прозрачность работы AI-помощника и позволяет администратору понять, на какие материалы опиралась система.

2.3.10. Оценка качества AI-модуля

Для проверки качества AI-помощника предусмотрен отдельный механизм оценки. Он запускает набор тестовых сценариев и формирует отчет с результатами. Такая проверка показывает, насколько точно система

понимает намерение пользователя, замечает тревожные симптомы, обращается к источникам из базы знаний и собирает безопасный ответ.

Результаты оценки помогают развивать приложение дальше. Если AI-помощник часто отвечает с низкой уверенностью или не находит подходящие источники, администратор видит, где именно есть слабое место. После этого можно расширить базу знаний, добавить недостающие статьи или уточнить правила маршрутизации.

2.4 Тестирование приложения

Тестирование веб-приложения проводилось, чтобы проверить работу основных пользовательских сценариев, серверной бизнес-логики и AI/RAG-модуля. Проверка шла на двух уровнях: автоматизированные тесты для серверной части и ручные функциональные сценарии в клиентском приложении.

Такой подход позволяет увидеть не только работу отдельных методов, но и весь путь пользователя в системе: от авторизации до записи на приём и обращения к интеллектуальному помощнику.

Критерии успешности были выбраны с учётом ключевых функций приложения. Проверялась авторизация по ролям, ограничения при записи на приём, корректное отображение данных в личном кабинете, обработка медицинских вопросов без постановки диагноза, выявление тревожных симптомов и наличие ссылок на материалы базы знаний. Отдельно проверялось, что административные функции доступны только пользователю с ролью ADMIN.

2.4.1. Проверка серверной части

Тестирование серверной части проводилось не только на уровне отдельных методов, но и через интеграционные сценарии REST API. Это позволяет проверить приложение в условиях, близких к реальному использованию: от входящего запроса до выполнения бизнес-логики, обращения к базе данных и возврата ответа клиенту. В результате оценивается

не изолированный фрагмент кода, а весь путь обработки пользовательского действия.

Автоматизированные тесты покрывают основные сценарии серверной части: регистрацию и вход пользователей, получение данных публичного каталога, создание и отмену записи на приём, запрет действий за пределами своей роли, выполнение административных операций и отдельные элементы AI-модуля. Отдельное внимание уделялось разграничению прав доступа. Приложение работает с медицинскими данными, поэтому система должна исключать просмотр, изменение или удаление чужой информации.

Для проверки REST API использовались тестовые HTTP-запросы к основным группам методов приложения. В ходе тестирования проверялись коды ответов сервера, корректность возвращаемых JSON-структур, сохранение данных в базе, обработка ошибок валидации и реакция системы на некорректные действия пользователя. Например, пациент не должен иметь доступ к административным операциям, а неавторизованный пользователь не должен иметь возможность создавать записи на прием или просматривать защищенные данные.

Отдельно проверялись сценарии, связанные с записью пациента на прием. Тесты подтверждают, что система корректно создает запись при наличии свободного слота, изменяет статус слота после бронирования и не допускает повторного использования занятого времени. Также проверялась отмена записи, при которой состояние соответствующего временного слота должно быть обновлено. Это важно, поскольку ошибка в данной логике может привести к конфликтам в расписании врача.

Тестирование AI-модуля выполнялось с учетом его роли в серверной архитектуре. Проверялась возможность отправки сообщения AI-помощнику, сохранение истории диалога, возврат структурированного ответа, наличие признака обнаружения тревожных симптомов и корректная работа механизмов безопасности. Для опасных формулировок система должна возвращать не обычный справочный ответ, а рекомендацию обратиться за

срочной медицинской помощью. Таким образом проверяется не только техническая доступность AI-сервиса, но и соблюдение ограничений медицинской предметной области.

Также проверялись административные сценарии: создание и изменение справочных записей, управление данными каталога, доступ к истории AI-диалогов и получение служебной информации о состоянии AI-модуля. Эти тесты подтверждают, что административная часть отделена от пользовательской и доступна только пользователям с соответствующей ролью.

Результаты тестового прогона представлены на рисунке 28.

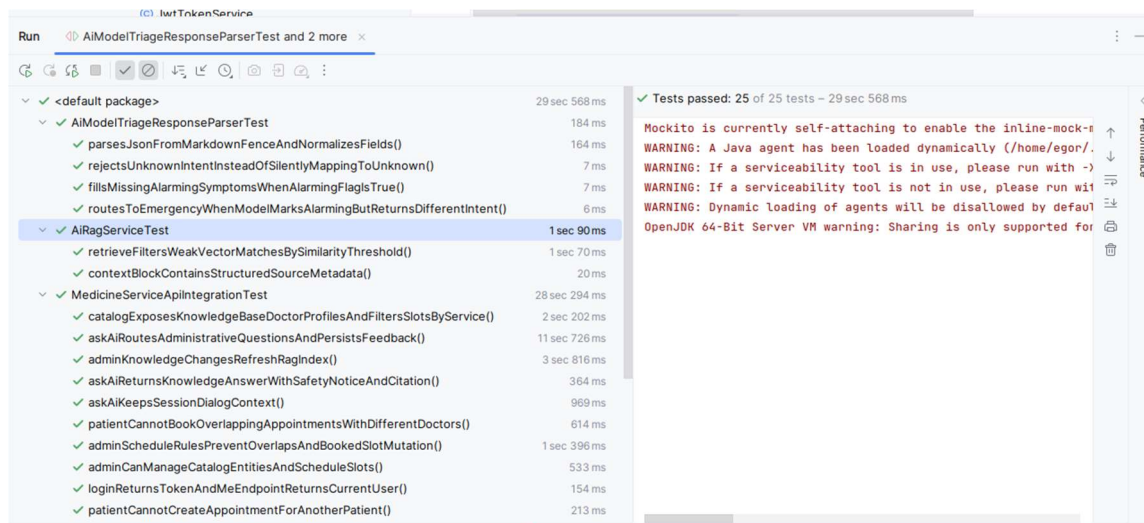


Рисунок 28 – Результат тестов

2.4.2. Проверка пользовательских сценариев

После проверки серверной части была выполнена ручная проверка клиентского приложения. Для иллюстрации выбраны сценарии, которые лучшего всего показывают работоспособность системы.

Проверка начинается с публичной части приложения. Главная страница должна открываться без авторизации, показывать позиционирование клиники, основные переходы и кнопки записи. Рисунок 29 подтверждает, что пользователь может начать работу с сервисом без предварительной настройки личного кабинета.

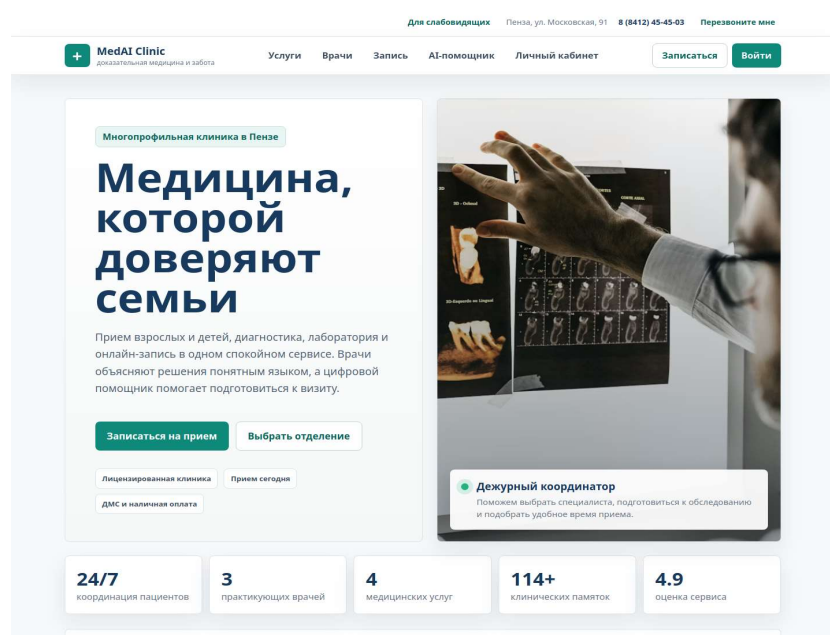


Рисунок 29 – Проверка главной страницы и публичной навигации

Далее проверяется форма входа. Для приложения с ролевой моделью важно, чтобы авторизация была явной точкой перехода к защищенным сценариям: записи, личному кабинету, AI-истории и административным функциям (рисунок 30).

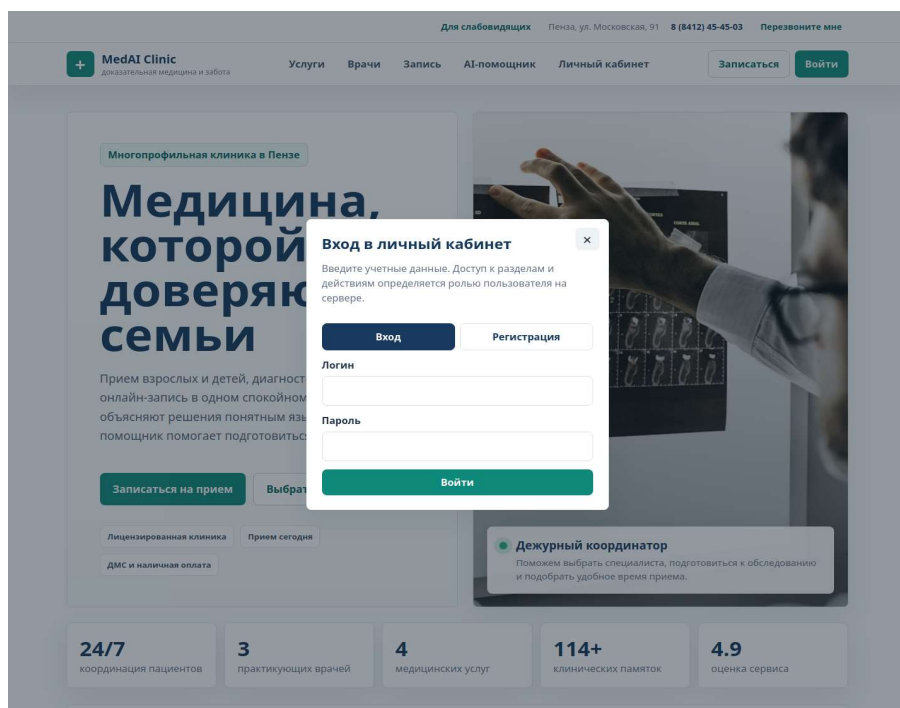


Рисунок 30 – Проверка модального окна авторизации пользователя

Каталог услуг проверяется как справочник, связанный с серверной частью приложения. В сценарии выполняется поиск услуги и открытие

подробной карточки, где пользователь видит описание, код услуги, стоимость и длительность приема (рисунок 31).

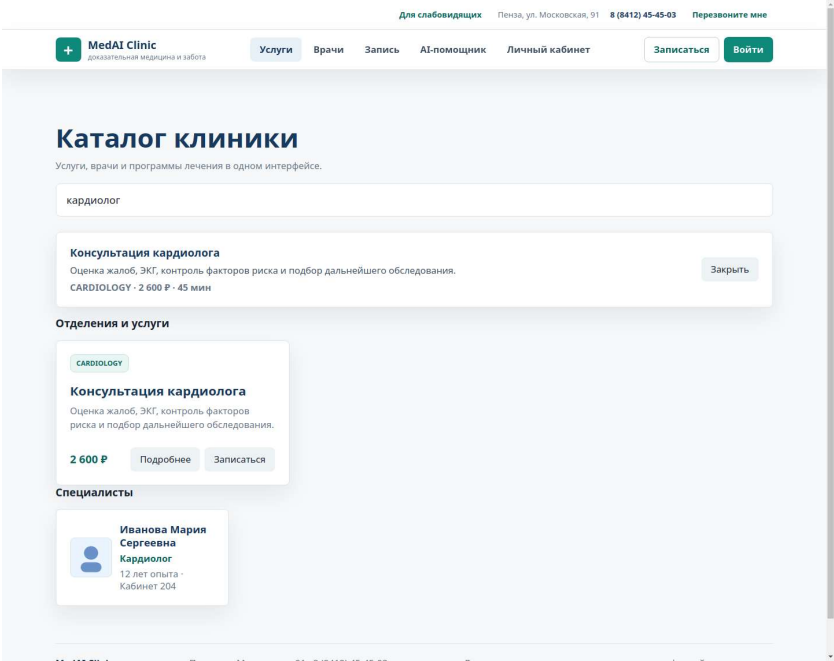


Рисунок 31-Проверка поиска услуги и открытия карточки услуги в каталоге

Раздел врачей проверяется отдельно, потому что он формирует маршрут пациента до записи. На экране должны отображаться специалисты, специализация, кабинет, опыт и связанные услуги (рисунок 32).

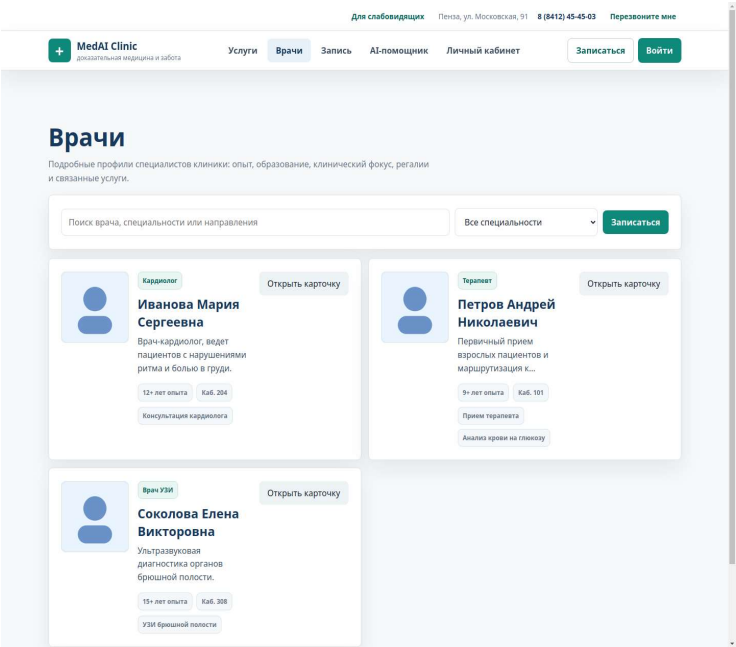


Рисунок 32 – Проверка отображения полного списка врачей клиники

Фильтрация врачей показывает, что интерфейс поддерживает быстрый поиск нужного специалиста. Это снижает нагрузку на пользователя при расширении каталога клиники (рисунок 33).

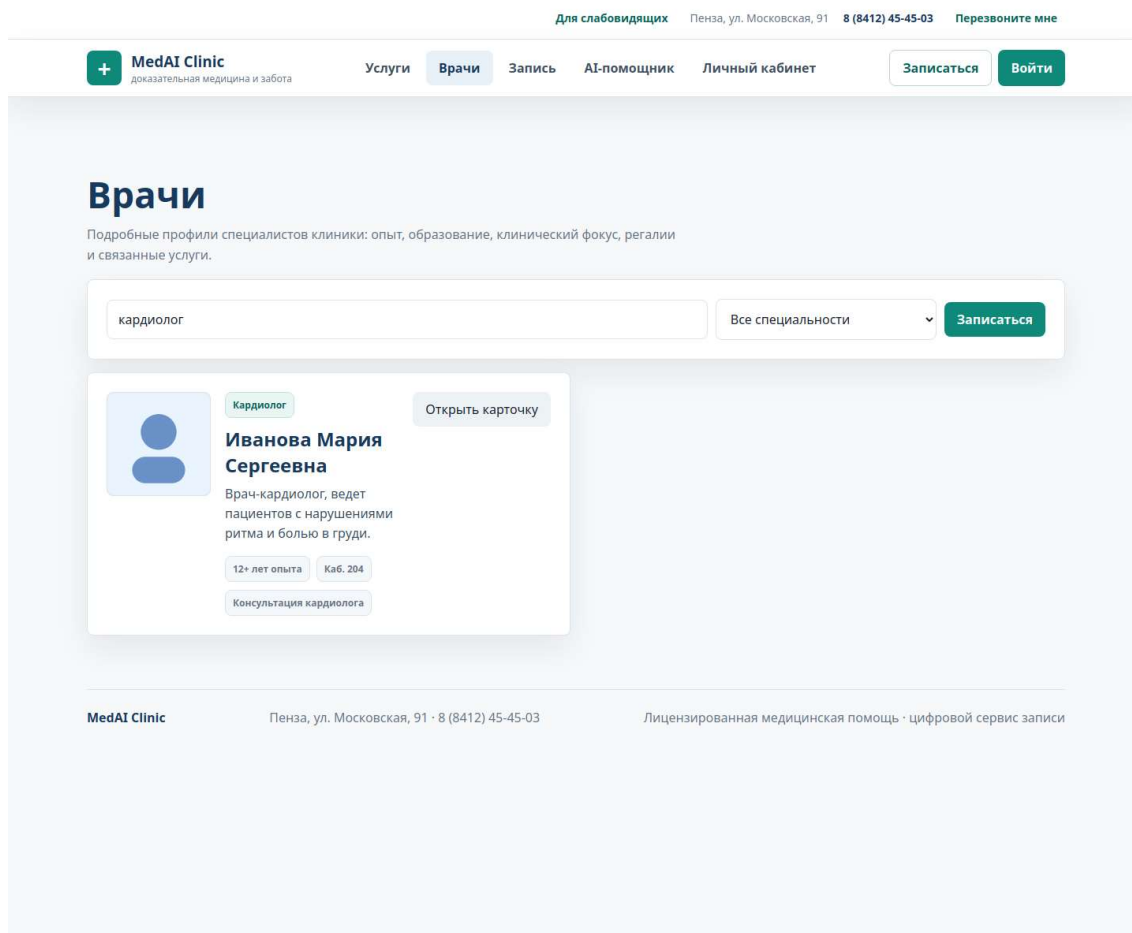


Рисунок 33 – Проверка фильтрации врачей по направлению

Сценарий записи является основным транзакционным сценарием приложения. На этом этапе проверяется выбор услуги, врача, даты и свободного слота, полученного от серверной части (рисунок 34).

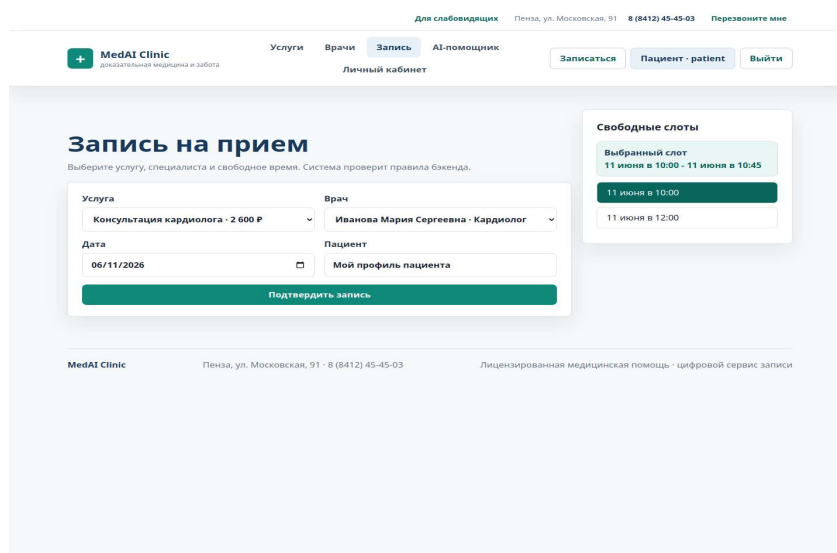


Рисунок 34 – Проверка выбора услуги, врача, даты и свободного слота

После выбора слота выполняется подтверждение записи. Система должна отправить данные на серверную часть, обновить состояние и показать пользователю понятный результат операции (рисунок 35).

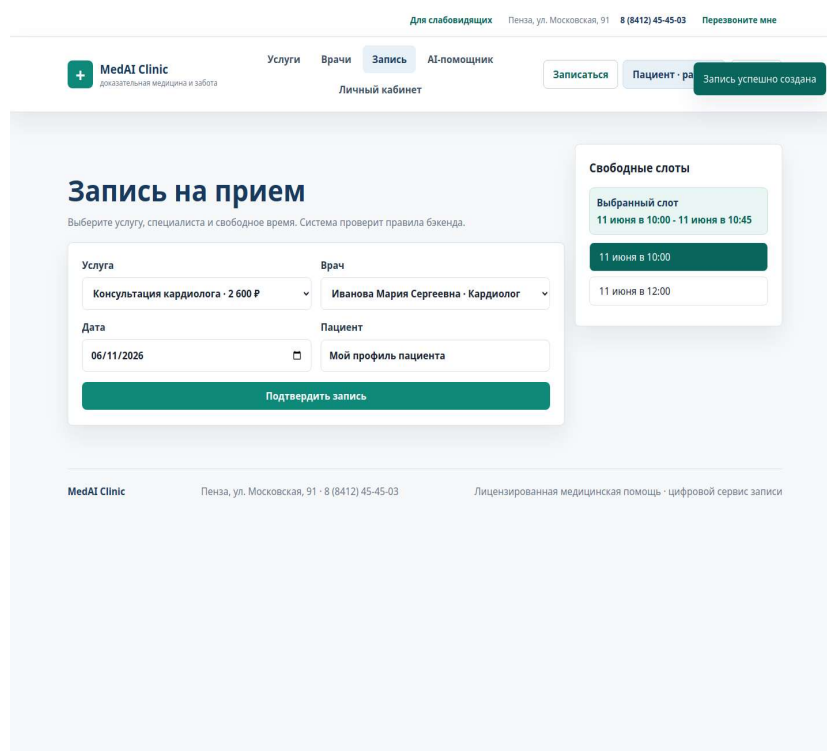


Рисунок 35 – Проверка подтверждения записи на прием

Личный кабинет подтверждает, что созданная или существующая запись возвращается пользователю как часть его персонального маршрута. Детальная карточка показывает услугу, врача, кабинет, время и статус приема (рисунок 36).

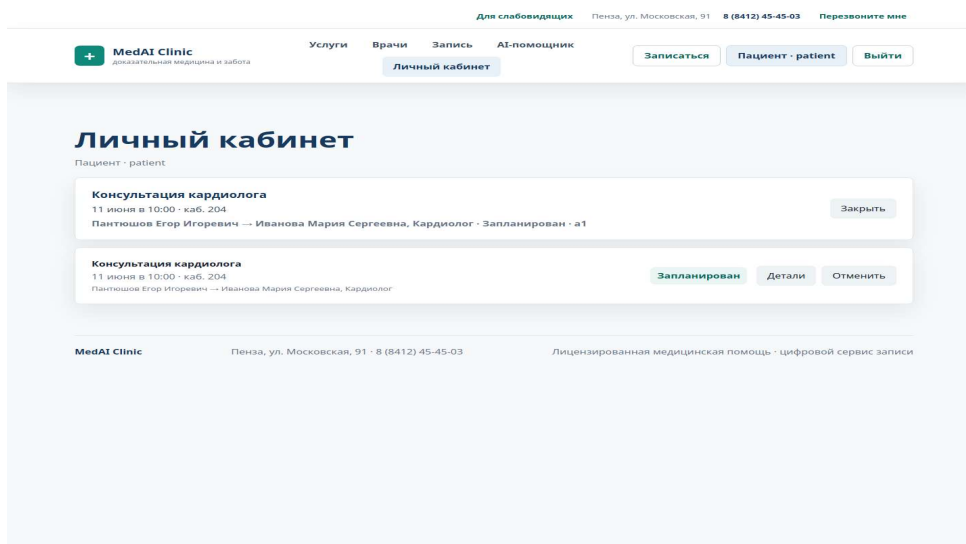


Рисунок 36 – Проверка личного кабинета и деталей записи пациента

AI-помощник проверяется сначала в исходном состоянии. Экран должен объяснять допустимый формат справочного взаимодействия и явно напоминать, что помощник не заменяет врача (рисунок 37).

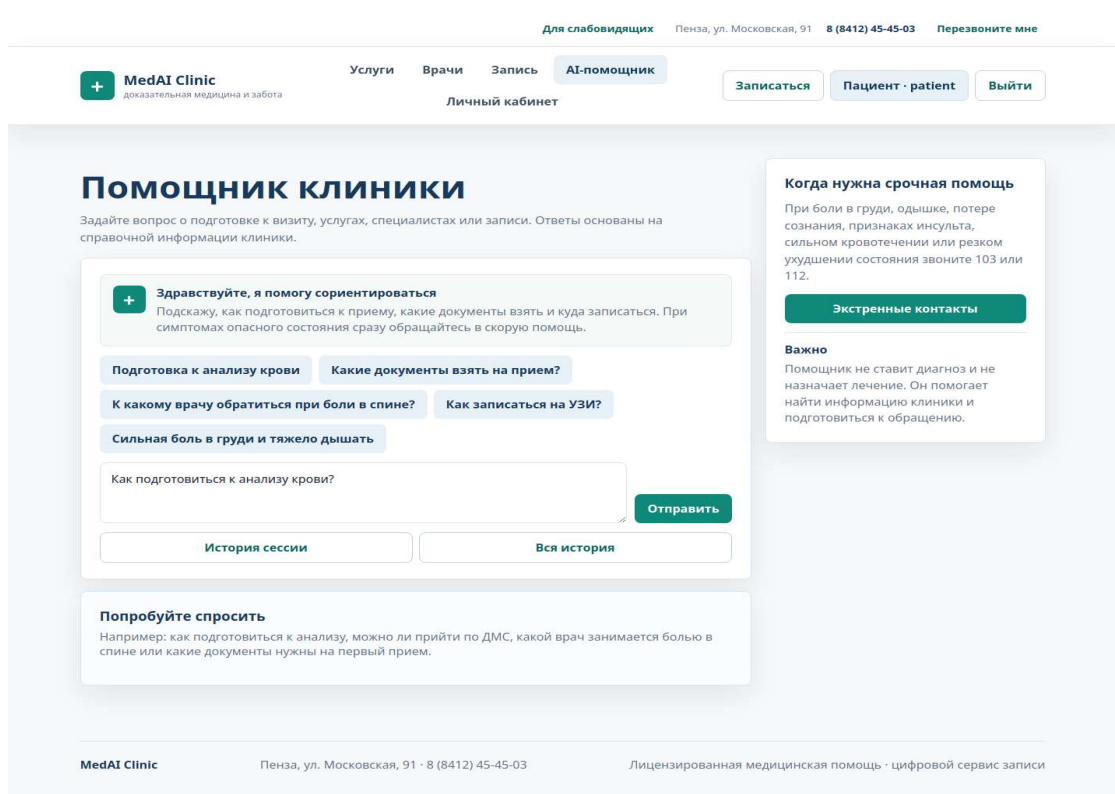


Рисунок 37 – Проверка начального экрана AI-помощника

Обычный RAG-сценарий проверяется вопросом о подготовке к анализу крови на глюкозу. Корректный результат должен содержать ответ по базе знаний, источник и дальнейшие действия пользователя (рисунок 38).

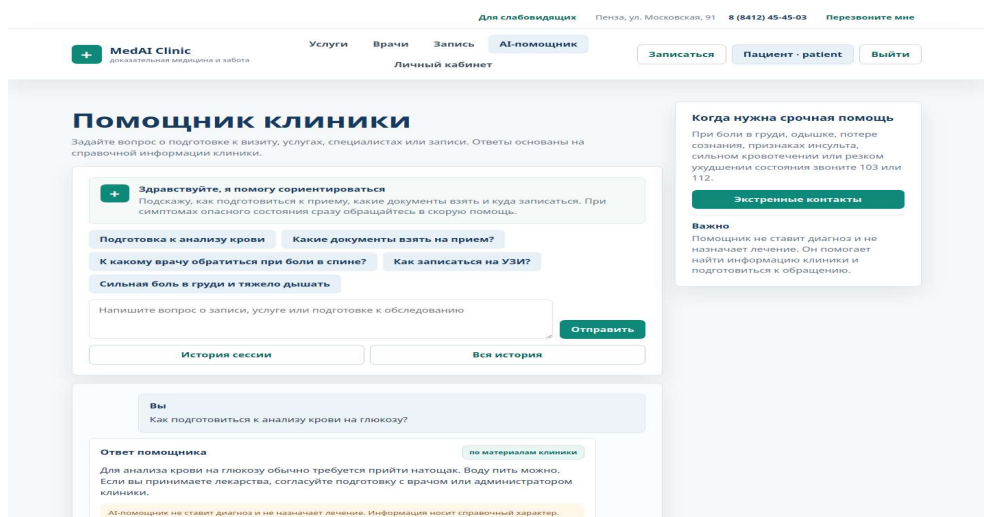


Рисунок 38 – Проверка RAG-ответа AI-помощника с источниками

Экстренный сценарий проверяется отдельным вопросом о боли в груди и затрудненном дыхании. В этом случае помощник должен рекомендовать срочную медицинскую помощь (рисунок 39).

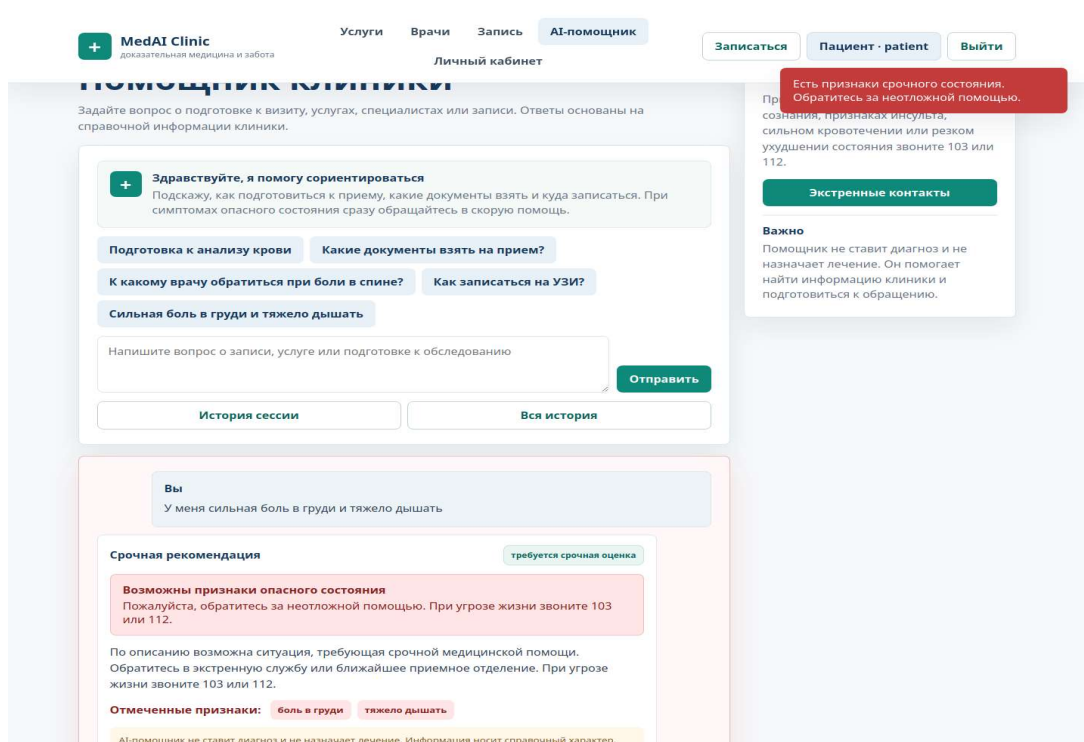


Рисунок 39 – Проверка AI-помощника при тревожных симптомах

Административный экран AI/RAG evaluation показывает, что качество помощника измеряется не вручную, а через контрольный набор кейсов и метрик (рисунок 40).

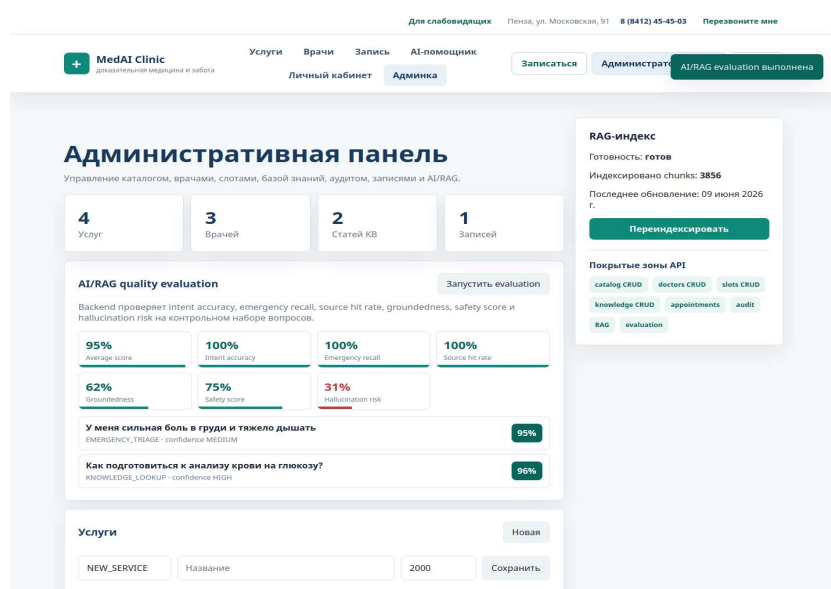


Рисунок 40 – Проверка административных метрик AI/RAG

Нижняя часть административной панели проверяет сопровождение системы после внедрения: управление данными, аудит действий и состояние RAG-индекса (рисунок 41).

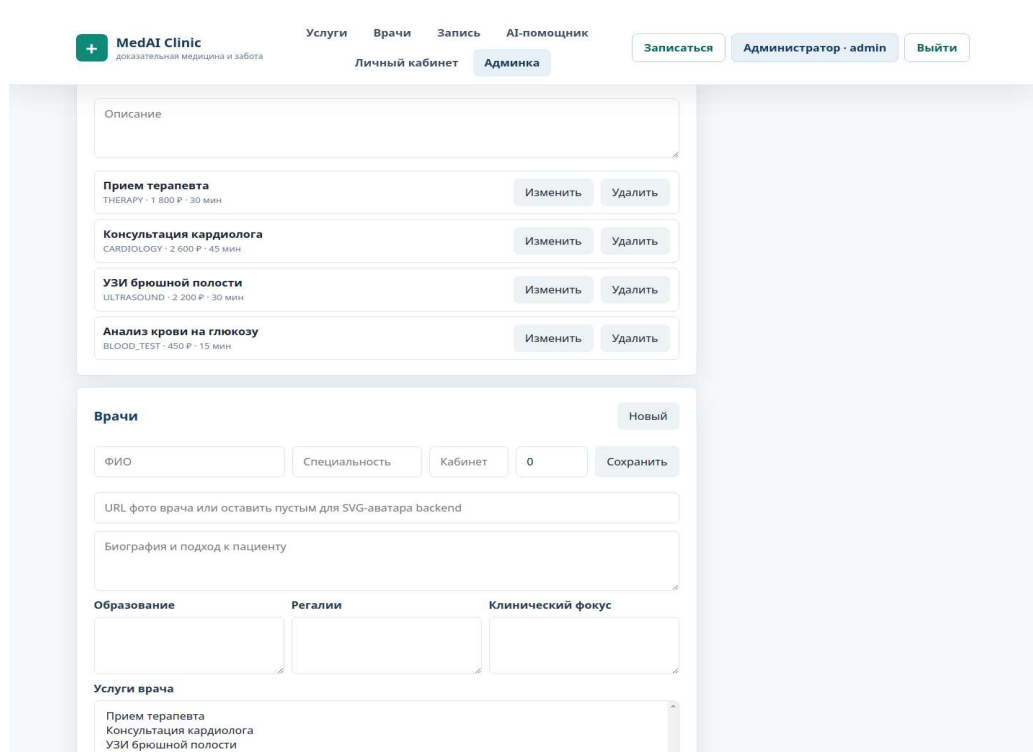


Рисунок 41 – Проверка административных CRUD-разделов, аудита и RAG-индекса

Отдельно проверяется версия для слабовидящих. Для медицинского сервиса это важная часть пользовательской доступности, так как интерфейсом могут пользоваться пациенты разных возрастных групп (рисунок 42).

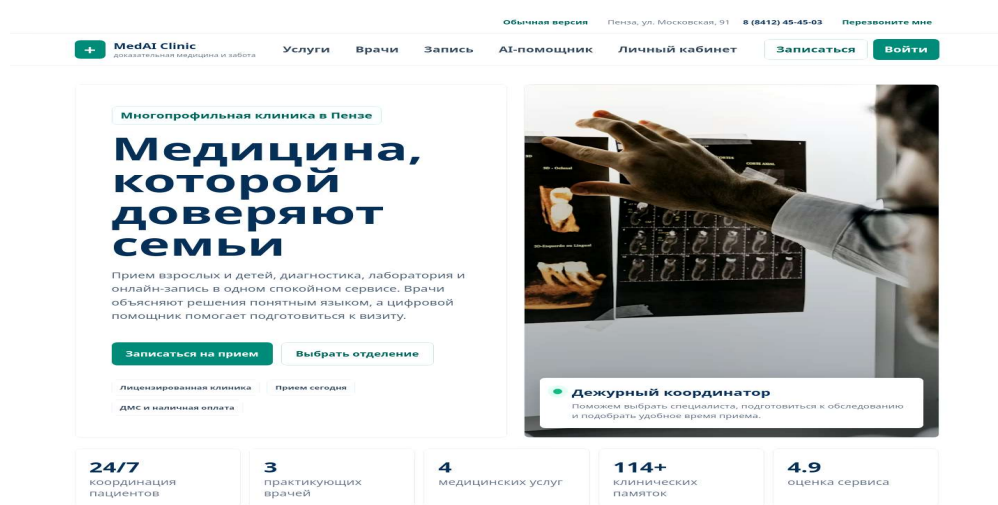


Рисунок 42-Проверка версии для слабовидящих

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был проведен анализ предметной области разработки веб-приложений для медицинских клиник. По результатам анализа были выделены основные требования к разрабатываемой системе.

На основании сформированных требований была спроектирована архитектура веб-приложения медицинской клиники. Серверная часть реализована на языке Java с использованием Spring Boot, Spring Security, Spring Data JPA и PostgreSQL. Клиентская часть разработана на React и TypeScript с применением Vite. Для разграничения прав доступа использована ролевая модель, предусматривающая роли пациента, врача и администратора, а обмен данными между клиентом и сервером организован через REST API.

В процессе разработки реализованы основные функциональные модули системы: каталог услуг и врачей, просмотр расписания, запись пациента на прием, личный кабинет, административная панель, аудит действий пользователей, модуль базы знаний и интеллектуальный ассистент. Отдельное внимание уделено обработке медицинских обращений: ассистент использует механизм поиска по базе знаний, отображает источники ответа и не заменяет консультацию врача. Для потенциально опасных симптомов предусмотрено выявление экстренных сценариев и формирование предупреждения о необходимости обращения за срочной медицинской помощью.

Таким образом, цель магистерской диссертации достигнута. В результате работы разработано веб-приложение для медицинской клиники с интегрированным AI-помощником, базой знаний, механизмом поиска релевантных источников, контролем медицинской безопасности и сохранением истории диалогов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. OWASP Top 10 [САЙТ]. – URL: <https://owasp.org/www-project-top-ten/> (дата обращения: 09.05.2026).
2. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. Attention Is All You Need // arXiv. – 2017. – URL: <https://arxiv.org/abs/1706.03762> (дата обращения: 09.05.2026).
3. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Kuttler H., Lewis M., Yih W., Rocktaschel T., Riedel S., Kiela D. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks // arXiv. – 2020. – URL: <https://arxiv.org/abs/2005.11401> (дата обращения: 09.05.2026).
4. Spring AI Reference Documentation [САЙТ]. – URL: <https://docs.spring.io/spring-ai/reference/> (дата обращения: 09.05.2026).
5. Spring Boot Reference Documentation [САЙТ]. – URL: <https://docs.spring.io/spring-boot/reference/> (дата обращения: 09.05.2026).
6. OpenAI. Models: OpenAI API documentation [САЙТ]. – URL: <https://platform.openai.com/docs/models> (дата обращения: 10.05.2026).
7. Anthropic. Models overview: Claude API Docs [САЙТ]. – URL: <https://docs.anthropic.com/en/docs/about-claude/models> (дата обращения: 10.06.2026).
8. Google AI for Developers. Gemini API models [САЙТ]. – URL: <https://ai.google.dev/gemini-api/docs/models> (дата обращения: 10.05.2026).
9. Mistral AI. La Plateforme documentation: models [САЙТ]. – URL: <https://docs.mistral.ai/getting-started/models/> (дата обращения: 10.05.2026).
10. Официальный сайт медицинской клиники «Медгард» [САЙТ]. – URL: <https://medguard.ru/> (дата обращения: 09.05.2026).
11. Официальный сайт медицинского центра «Здоровье» [САЙТ]. – URL: <https://zdorovie58.ru/> (дата обращения: 09.05.2026).
12. Официальный сайт Клиники Фомина [САЙТ]. – URL: <https://fomin-clinic.ru/> (дата обращения: 09.06.2026).

13. ISO/IEC 25010:2023 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model [САЙТ]. – URL: <https://www.iso.org/standard/78176.html> (датаобращения: 09.05.2026).

14. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT) [САЙТ]. – URL: <https://www.rfc-editor.org/rfc/rfc7519> (датаобращения: 09.05.2026).

15. TypeScript Documentation [САЙТ]. – URL: <https://www.typescriptlang.org/docs/> (датаобращения: 09.05.2026).

16. React Documentation [САЙТ]. – URL: <https://react.dev/reference/react> (датаобращения: 09.05.2026).

17. PostgreSQL Documentation [САЙТ]. – URL: <https://www.postgresql.org/docs/> (датаобращения: 09.05.2026).

18. Docker Documentation [САЙТ]. – URL: <https://docs.docker.com/> (датаобращения: 09.05.2026).

ПРИЛОЖЕНИЕ А

(обязательное)

Веб-приложение с интеграцией AI-помощника для медицинской клиники

Листинг программы

Файл AiAssistantController.java

```
package org.example.medicineservice.controller;

import jakarta.validation.Valid;
import org.example.medicineservice.domain.AiDialogEntry;
import org.example.medicineservice.domain.UserRole;
import org.example.medicineservice.dto.AiAssistantResponse;
import org.example.medicineservice.dto.AiEvaluationCase;
import org.example.medicineservice.dto.AiEvaluationReport;
import org.example.medicineservice.dto.AiFeedbackRequest;
import org.example.medicineservice.dto.AiRagIndexStatus;
import org.example.medicineservice.dto.AskAiRequest;
import org.example.medicineservice.security.AuthenticatedUser;
import org.example.medicineservice.security.SecurityPrincipalService;
import org.example.medicineservice.service.AiAssistantService;
import org.example.medicineservice.service.AiEvaluationService;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collection;
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/api/ai")
public class AiAssistantController {

    private final AiAssistantService aiAssistantService;
    private final AiEvaluationService aiEvaluationService;
    private final SecurityPrincipalService securityPrincipalService;

    public AiAssistantController(AiAssistantService aiAssistantService,
                                AiEvaluationService aiEvaluationService,
                                SecurityPrincipalService securityPrincipalService) {
        this.aiAssistantService = aiAssistantService;
        this.aiEvaluationService = aiEvaluationService;
        this.securityPrincipalService = securityPrincipalService;
    }
}
```

```

        @PostMapping("/ask")
        public AiAssistantResponse ask(@RequestParam(name =
"sessionId", required = false) UUID sessionId,
                                     @Valid @RequestBody
AskAiRequest request,
                                     Authentication
authentication) {
            return
aiAssistantService.ask(securityPrincipalService.requireUser(auth
entication), sessionId, request.message());
        }

        @GetMapping("/rag/index")
        public AiRagIndexStatus ragIndexStatus() {
            return aiAssistantService.ragIndexStatus();
        }

        @PostMapping("/rag/reindex")
        public AiRagIndexStatus reindexRag() {
            return aiAssistantService.reindexRag();
        }

        @GetMapping("/evaluation/cases")
        public List<AiEvaluationCase> evaluationCases() {
            return aiEvaluationService.cases();
        }

        @PostMapping("/evaluation/run")
        public AiEvaluationReport runEvaluation(Authentication
authentication) {
            return
aiEvaluationService.run(securityPrincipalService.requireUser(auth
entication));
        }

        @GetMapping("/dialog")
        public Collection<AiDialogEntry>
dialogEntries(Authentication authentication) {
            AuthenticatedUser user =
securityPrincipalService.requireUser(authentication);
            if (securityPrincipalService.hasRole(user,
UserRole.ADMIN)) {
                return aiAssistantService.dialogEntries();
            }
            return aiAssistantService.dialogEntries(user);
        }

        @GetMapping("/sessions/{sessionId}/dialog")
        public List<AiDialogEntry>
dialogEntries(@PathVariable("sessionId") UUID sessionId,
Authentication authentication) {

```

```

        AuthenticatedUser user =
securityPrincipalService.requireUser(authentication);
        if (securityPrincipalService.hasRole(user,
UserRole.ADMIN)) {
            return aiAssistantService.dialogEntries(sessionId);
        }
        return aiAssistantService.dialogEntries(user,
sessionId);
    }

    @PostMapping("/dialog/{id}/feedback")
    public AiDialogEntry rate(@PathVariable("id") UUID id,
                             @Valid @RequestBody
AiFeedbackRequest request,
                             Authentication authentication) {
        AuthenticatedUser user =
securityPrincipalService.requireUser(authentication);
        AiDialogEntry entry =
aiAssistantService.dialogEntry(id);
        if (!securityPrincipalService.hasRole(user,
UserRole.ADMIN) && !user.id().equals(entry.userId())) {
            throw new AccessDeniedException("Нет доступа к AI-
диалогу другого пользователя");
        }
        return aiAssistantService.rate(id,
request.usefulnessRating());
    }
}

```

Файл AiAssistantService.java

```

package org.example.medicineservice.service;

import org.example.medicineservice.domain.AiDialogEntry;
import org.example.medicineservice.domain.AiIntent;
import org.example.medicineservice.dto.AiAssistantResponse;
import org.example.medicineservice.dto.AiCitation;
import org.example.medicineservice.dto.AiQualityMetrics;
import org.example.medicineservice.dto.AiRagIndexStatus;
import org.example.medicineservice.dto.AiSourceSnippet;
import org.example.medicineservice.dto.AiTriageAssessment;
import org.example.medicineservice.exception.NotFoundException;
import
org.example.medicineservice.repository.AiDialogRepository;
import org.example.medicineservice.security.AuthenticatedUser;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.Collection;
import java.util.List;
import java.util.UUID;
import java.util.stream.Stream;

```

```

@Service
public class AiAssistantService {

    private static final String SAFETY_NOTICE = "AI-помощник не ставит диагноз и не назначает лечение. При жалобах или ухудшении состояния обратитесь к врачу.";

    private final AiDialogRepository aiDialogRepository;
    private final AiRagService aiRagService;
    private final ClinicalTriageClient clinicalTriageClient;
    private final MedicalSafetyPolicy medicalSafetyPolicy;
    private final AuditLogService auditLogService;
    private final AiQualityService aiQualityService;

    public AiAssistantService(AiDialogRepository
aiDialogRepository,
                                AiRagService aiRagService,
                                ClinicalTriageClient
clinicalTriageClient,
                                MedicalSafetyPolicy
medicalSafetyPolicy,
                                AuditLogService auditLogService,
                                AiQualityService aiQualityService)
    {
        this.aiDialogRepository = aiDialogRepository;
        this.aiRagService = aiRagService;
        this.clinicalTriageClient = clinicalTriageClient;
        this.medicalSafetyPolicy = medicalSafetyPolicy;
        this.auditLogService = auditLogService;
        this.aiQualityService = aiQualityService;
    }

    public AiAssistantResponse ask(AuthenticatedUser user, UUID
sessionId, String message) {
        medicalSafetyPolicy.validateMessage(message);
        UUID effectiveSessionId = sessionId == null ?
UUID.randomUUID() : sessionId;
        List<AiDialogEntry> dialogContext =
aiDialogRepository.findBySessionIdAndUserId(effectiveSessionId,
user.id());
        List<AiRagService.ScoredKnowledgeArticle> ragMatches =
aiRagService.retrieve(message);
        String ragContext =
aiRagService.contextBlock(ragMatches);

        AiTriageAssessment assessment =
medicalSafetyPolicy.immediateDecision(message)
            .orElseGet(() -> safeModelAssessment(message,
ragContext, dialogContext));
        List<AiCitation> citationDetails =
citationDetails(assessment, ragMatches);
        String answer = buildAnswer(assessment);
    }
}

```

```

        var qualityMetrics = aiQualityService.calculate(message,
answer, assessment, ragMatches);
        var sourceSnippets =
aiQualityService.sourceSnippets(ragMatches);

        AiAssistantResponse response = saveResponse(
            user.id(),
            effectiveSessionId,
            message,
            assessment,
            answer,
            citationDetails,
            sourceSnippets,
            qualityMetrics
        );
        auditLogService.record(user, "AI_ASSISTANT_ASKED",
"AiDialogEntry", response.dialogEntryId().toString(),
            "intent=" + response.intent() + "; alarming=" +
response.alarmingSymptomsDetected());
        return response;
    }

    private AiTriageAssessment safeModelAssessment(String
message, String ragContext, List<AiDialogEntry> dialogContext) {
        try {
            AiTriageAssessment assessment =
clinicalTriageClient.assess(message, ragContext, dialogContext);
            if (assessment == null || assessment.confidence() <
0.55 || assessment.answer() == null ||
assessment.answer().isBlank()) {
                return
medicalSafetyPolicy.fallbackDecision(message);
            }
            return assessment;
        } catch (RuntimeException exception) {
            return
medicalSafetyPolicy.fallbackDecision(message);
        }
    }

    public AiRagIndexStatus ragIndexStatus() {
        return aiRagService.indexStatus();
    }

    public AiRagIndexStatus reindexRag() {
        return aiRagService.reindex();
    }

    public Collection<AiDialogEntry> dialogEntries() {
        return aiDialogRepository.findAll();
    }

```

```

        public Collection<AiDialogEntry>
dialogEntries(AuthenticatedUser user) {
            return aiDialogRepository.findById(user.id());
        }

        public List<AiDialogEntry> dialogEntries(UUID sessionId) {
            return aiDialogRepository.findById(sessionId);
        }

        public List<AiDialogEntry> dialogEntries(AuthenticatedUser
user, UUID sessionId) {
            return
aiDialogRepository.findByIdAndUserId(sessionId,
user.id());
        }

        public AiDialogEntry dialogEntry(UUID dialogEntryId) {
            return aiDialogRepository.findById(dialogEntryId)
                .orElseThrow(() -> new NotFoundException("AI
dialog entry not found: " + dialogEntryId));
        }

        public AiDialogEntry rate(UUID dialogEntryId, int
usefulnessRating) {
            AiDialogEntry entry = dialogEntry(dialogEntryId);

            return aiDialogRepository.save(new AiDialogEntry(
                entry.id(),
                entry.userId(),
                entry.sessionId(),
                entry.createdAt(),
                entry.userMessage(),
                entry.assistantAnswer(),
                entry.intent(),
                entry.confidence(),
                entry.citations(),
                usefulnessRating
            ));
        }

        private List<AiCitation> citationDetails(AiTriageAssessment
assessment, List<AiRagService.ScoredKnowledgeArticle>
ragMatches) {
            if (ragMatches.isEmpty()) {
                if (assessment.alarmingSymptomsDetected()) {
                    return List.of(new AiCitation("kb-emergency",
"Признаки срочного обращения", 100));
                }
                return List.of(new AiCitation("kb-ai-safety",
"Ограничения AI-помощника", 1));
            }
        }

```

```

        if (assessment.alarmingSymptomsDetected() &&
ragMatches.stream().noneMatch(match ->
match.article().id().equals("kb-emergency"))) {
            return Stream.concat(
                Stream.of(new AiCitation("kb-emergency",
"Признаки срочного обращения", 100)),
                aiRagService.citations(ragMatches).stream()
            ).toList();
        }
        return aiRagService.citations(ragMatches);
    }

    private String buildAnswer(AiTriageAssessment assessment) {
        if (assessment.alarmingSymptomsDetected()) {
            return "По описанию возможна ситуация, требующая
срочной медицинской помощи. "
                + "Обратитесь в экстренную службу или
ближайшее приемное отделение. "
                +
symptomText(assessment.alarmingSymptoms());
        }

        if (assessment.answer() != null &&
!assessment.answer().isBlank()) {
            return assessment.answer().strip() + " " +
routingHint(assessment.intent());
        }

        return "Я могу помочь с записью на прием, расписанием
врачей, подготовкой к анализам и общими правилами работы
клиники. "
            + "Сформулируйте вопрос как справочный или
организационный.";
    }

    private String symptomText(List<String> symptoms) {
        if (symptoms == null || symptoms.isEmpty()) {
            return "AI-триаж отметил тревожные признаки или
опасную неопределенность.";
        }
        return "Отмеченные тревожные признаки: " +
String.join(", ", symptoms) + ".";
    }

    private String routingHint(AiIntent intent) {
        return switch (intent) {
            case APPOINTMENT_BOOKING -> "Для записи выберите
услугу, врача и свободный слот.";
            case KNOWLEDGE_LOOKUP -> "При медицинских жалобах
лучше записаться к профильному специалисту.";
            case ADMIN_ROUTING -> "Организационный вопрос лучше
передать администратору клиники.";
        }
    }

```

```

        case EMERGENCY_TRIAGE -> "При тревожных симптомах
нужна срочная очная помощь.";
        case UNKNOWN -> "Для первичной маршрутизации можно
начать с записи к терапевту.";
    };
}

private List<String> enrichActionsByQuality(List<String>
actions, AiQualityMetrics qualityMetrics) {
    if (qualityMetrics == null ||
!"LOW".equals(qualityMetrics.confidenceLabel())) {
        return actions;
    }
    return Stream.concat(
        actions == null ? Stream.<String>empty() :
actions.stream(),
        Stream.of("Ответ имеет низкую уверенность:
проверьте источники или обратитесь к администратору/врачу")
    ).distinct().toList();
}

private AiAssistantResponse saveResponse(
    UUID userId,
    UUID sessionId,
    String message,
    AiTriageAssessment assessment,
    String answer,
    List<AiCitation> citationDetails,
    List<AiSourceSnippet> sourceSnippets,
    AiQualityMetrics qualityMetrics
) {
    List<String> citations =
citationDetails.stream().map(AiCitation::articleId).distinct().t
oList();
    AiDialogEntry entry = aiDialogRepository.save(new
AiDialogEntry(
        UUID.randomUUID(),
        userId,
        sessionId,
        LocalDateTime.now(),
        message,
        answer.strip(),
        assessment.intent(),
        assessment.confidence(),
        citations,
        null
    ));

    return new AiAssistantResponse(
        entry.id(),
        entry.sessionId(),
        entry.intent(),
        assessment.alarmingSymptomsDetected(),

```

```

        entry.confidence(),
        entry.assistantAnswer(),
        SAFETY_NOTICE,
        assessment.reasoning(),
        assessment.alarmingSymptoms(),
        entry.citations(),
        citationDetails,
        sourceSnippets,
        qualityMetrics,

```

Файл AiRagService.java

```

package org.example.medicineservice.service;

import org.example.medicineservice.domain.KnowledgeArticle;
import org.example.medicineservice.domain.KnowledgeArticleChunk;
import org.example.medicineservice.dto.AiCitation;
import org.example.medicineservice.dto.AiRagIndexStatus;
import org.example.medicineservice.exception.AiServiceException;
import
org.example.medicineservice.repository.KnowledgeVectorRepository
;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import java.util.Comparator;
import java.util.List;
import java.util.Locale;
import java.util.Set;
import java.util.regex.Pattern;

@Service
public class AiRagService {

    private static final Pattern NON_SEARCH_CHARS =
Pattern.compile("[^\\p{L}\\p{N}]+");
    private static final Set<String> STOP_WORDS = Set.of(
        "как", "что", "это", "для", "при", "или", "если",
"нужно", "надо", "можно",
        "меня", "мне", "мой", "моя", "мое", "пациент",
"пациента", "клиники", "клинике",
        "врач", "врача", "помощник", "обычно", "уточнить",
"обратиться", "подго"
    );

    private final KnowledgeVectorRepository
knowledgeVectorRepository;
    private final ClinicalEmbeddingClient embeddingClient;
    private final KnowledgeEmbeddingIndexer
knowledgeEmbeddingIndexer;
    private final int maxArticles;
    private final double minSimilarity;
    private final int minRelevance;

```

```

    public AiRagService(KnowledgeVectorRepository
knowledgeVectorRepository,
                        ClinicalEmbeddingClient embeddingClient,
                        KnowledgeEmbeddingIndexer
knowledgeEmbeddingIndexer,
                        @Value("${medicine.ai.rag.max-
articles:4}") int maxArticles,
                        @Value("${medicine.ai.rag.min-
similarity:0.25}") double minSimilarity,
                        @Value("${medicine.ai.rag.min-
relevance:75}") int minRelevance) {
        this.knowledgeVectorRepository =
knowledgeVectorRepository;
        this.embeddingClient = embeddingClient;
        this.knowledgeEmbeddingIndexer =
knowledgeEmbeddingIndexer;
        this.maxArticles = maxArticles;
        this.minSimilarity = minSimilarity;
        this.minRelevance = minRelevance;
    }

    public List<ScoredKnowledgeArticle> retrieve(String message)
{
    if (!knowledgeVectorRepository.hasIndexedChunks()) {
        throw new AiServiceException("Индекс базы знаний RAG
пуст. Выполните переиндексацию embeddings.", null);
    }

    float[] embedding = embeddingClient.embed(message);
    Set<String> queryTerms = searchTerms(message);
    Set<String> exactQueryTerms = exactSearchTerms(message);
    int candidateLimit = Math.max(maxArticles * 250, 1000);

    return knowledgeVectorRepository.findNearest(embedding,
candidateLimit).stream()
        .filter(chunk -> chunk.similarity() >=
minSimilarity)
        .map(chunk -> toScoredArticle(chunk, queryTerms,
exactQueryTerms))
        .filter(match -> match.lexicalMatches() > 0)
        .filter(match -> match.titleMatches() > 0 ||
match.lexicalMatches() >= 2 || match.relevance() >=
minRelevance)

        .sorted(Comparator.comparingInt(ScoredKnowledgeArticle::relevanc
e).reversed())
        .thenComparingInt(match ->
sourcePriority(match.article().id()))

        .thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::
exactTitleMatches).reversed())

```

```

.thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::
titleMatches).reversed())

.thenComparing(Comparator.comparingLong(ScoredKnowledgeArticle::
lexicalMatches).reversed()))
        .limit(maxArticles)
        .toList();
    }

    public AiRagIndexStatus indexStatus() {
        long indexedChunks =
knowledgeVectorRepository.countIndexedChunks();
        long totalChunks = totalChunks();
        KnowledgeEmbeddingIndexer.IndexProgress progress =
knowledgeEmbeddingIndexer.progress();
        if (progress != null && progress.totalChunks() > 0) {
            totalChunks = progress.totalChunks();
        }
        long missingChunks = Math.max(0, totalChunks -
indexedChunks);
        boolean indexing = progress != null &&
progress.running();
        return new AiRagIndexStatus(
            indexedChunks > 0,
            indexedChunks,
            totalChunks,
            missingChunks,
            indexing,
            progress == null ? 0 :
progress.processedChunks(),
            progress == null ? 0 : progress.updatedChunks(),
            progress == null ? 0 : progress.failedChunks(),
            progress == null ? null : progress.lastError(),
            knowledgeVectorRepository.lastUpdatedAt()
        );
    }

    public AiRagIndexStatus reindex() {
        knowledgeEmbeddingIndexer.startIndexing("manual");
        return indexStatus();
    }

    private long totalChunks() {
        try {
            return knowledgeEmbeddingIndexer.totalChunks();
        } catch (RuntimeException exception) {
            return
knowledgeVectorRepository.countIndexedChunks();
        }
    }
}

```

```

        public List<AiCitation>
citations(List<ScoredKnowledgeArticle> matches) {
            return matches.stream()
                .map(match -> new
AiCitation(match.article().id(), match.article().title(),
match.relevance()))
                .toList();
        }

        public String contextBlock(List<ScoredKnowledgeArticle>
matches) {
            if (matches.isEmpty()) {
                return "База знаний не дала релевантных
фрагментов.";
            }
            return matches.stream()
                .map(this::sourceBlock)
                .reduce((left, right) -> left + "\n\n" + right)
                .orElse("");
        }

        private String sourceBlock(ScoredKnowledgeArticle match) {
            return "Источник RAG:\n"
                + "articleId: " + match.article().id() + "\n"
                + "title: " + match.article().title() + "\n"
                + "relevance: " + match.relevance() + "/100\n"
                + "content: " + match.article().content();
        }

        private ScoredKnowledgeArticle
toScoredArticle(KnowledgeArticleChunk chunk, Set<String>
queryTerms, Set<String> exactQueryTerms) {
            Set<String> titleTerms = searchTerms(chunk.title());
            Set<String> documentTerms = searchTerms(chunk.title() +
" " + chunk.content());
            Set<String> exactTitleTerms =
exactSearchTerms(chunk.title());
            long lexicalMatches =
queryTerms.stream().filter(documentTerms::contains).count();
            long titleMatches =
queryTerms.stream().filter(titleTerms::contains).count();
            long exactTitleMatches =
exactQueryTerms.stream().filter(exactTitleTerms::contains).count
();
            double lexicalRatio = queryTerms.isEmpty() ? 0 :
(double) lexicalMatches / queryTerms.size();
            double weightedScore = chunk.similarity() * 75 +
lexicalRatio * 25 + titleMatches * 15 + exactTitleMatches * 10;
            int relevance = Math.max(1, Math.min(100, (int)
Math.round(weightedScore)));
            KnowledgeArticle article = new KnowledgeArticle(
                chunk.articleId(),
                chunk.title(),

```

```

        Set.of(),
        chunk.content()
    );
    return new ScoredKnowledgeArticle(article, relevance,
chunk.similarity(), lexicalMatches, titleMatches,
exactTitleMatches);
}

private int sourcePriority(String articleId) {
    if (articleId == null) {
        return 9;
    }
    if (articleId.startsWith("kb-pro-")) {
        return 0;
    }
    if (articleId.startsWith("kb-") &&
!articleId.startsWith("kbpopular-")) {
        return 1;
    }
    if (articleId.startsWith("kbreal-")) {
        return 2;
    }
    if (articleId.startsWith("kbpopular-")) {
        return 3;
    }
    return 4;
}

private Set<String> searchTerms(String text) {
    if (text == null || text.isBlank()) {
        return Set.of();
    }
    String normalized =
NON_SEARCH_CHARS.matcher(text.toLowerCase(Locale.ROOT).replace('
ë', 'e')).replaceAll(" ");
    return
Pattern.compile("\\s+").splitAsStream(normalized.strip())
        .map(this::searchTerm)
        .filter(term -> term.length() >= 4)
        .filter(term -> !STOP_WORDS.contains(term))
        .collect(java.util.stream.Collectors.toSet());
}

private Set<String> exactSearchTerms(String text) {
    if (text == null || text.isBlank()) {
        return Set.of();
    }
    String normalized =
NON_SEARCH_CHARS.matcher(text.toLowerCase(Locale.ROOT).replace('
ë', 'e')).replaceAll(" ");
    return
Pattern.compile("\\s+").splitAsStream(normalized.strip())

```

```

        .filter(term -> term.length() >= 4)
        .filter(term -> !STOP_WORDS.contains(term))
        .collect(java.util.stream.Collectors.toSet());
    }

    private String searchTerm(String token) {
        String normalized = token

        .replaceAll("(иями|ями|ами|ого|ему|ыми|ими|ной|ная|ное|ние|ции|и
        ях|ах|ях|ом|ем|ый|ий|ой|ая|ое|ые|ую|ью|ия|ие|а|я|и|ы|е|у)$",
        "");
        if (normalized.length() >= 5) {
            return normalized.substring(0, 5);
        }
        return normalized;
    }

    public record ScoredKnowledgeArticle(
        KnowledgeArticle article,
        int relevance,
        double vectorSimilarity,
        long lexicalMatches,
        long titleMatches,
        long exactTitleMatches
    ) {
    }
}

```

Файл CatalogService.java

```

package org.example.medicineservice.service;

import org.example.medicineservice.domain.AppointmentSlot;
import org.example.medicineservice.domain.Doctor;
import org.example.medicineservice.domain.KnowledgeArticle;
import org.example.medicineservice.domain.MedicalService;
import org.example.medicineservice.exception.NotFoundException;
import
org.example.medicineservice.repository.AppointmentSlotRepository
;
import org.example.medicineservice.repository.DoctorRepository;
import
org.example.medicineservice.repository.KnowledgeBaseRepository;
import
org.example.medicineservice.repository.MedicalServiceRepository;
import org.springframework.stereotype.Service;

import java.time.LocalDate;
import java.util.Collection;
import java.util.List;
import java.util.UUID;

```

```

@Servicepublic class CatalogService {

    private final DoctorRepository doctorRepository;
    private final MedicalServiceRepository
medicalServiceRepository;
    private final AppointmentSlotRepository slotRepository;
    private final KnowledgeBaseRepository
knowledgeBaseRepository;

    public CatalogService(DoctorRepository doctorRepository,
                          MedicalServiceRepository
medicalServiceRepository,
                          AppointmentSlotRepository
slotRepository,
                          KnowledgeBaseRepository
knowledgeBaseRepository) {
        this.doctorRepository = doctorRepository;
        this.medicalServiceRepository =
medicalServiceRepository;
        this.slotRepository = slotRepository;
        this.knowledgeBaseRepository = knowledgeBaseRepository;
    }

    public Collection<Doctor> doctors() {
        return doctorRepository.findAll();
    }

    public Doctor doctor(UUID id) {
        return doctorRepository.findById(id)
            .orElseThrow(() -> new NotFoundException("Doctor
not found: " + id));
    }

    public Collection<MedicalService> services() {
        return medicalServiceRepository.findAll();
    }

    public MedicalService medicalService(String code) {
        return medicalServiceRepository.findByCode(code)
            .orElseThrow(() -> new
NotFoundException("Medical service not found: " + code));
    }

    public List<AppointmentSlot> availableSlots(UUID doctorId,
String serviceCode, LocalDate date) {
        if (doctorId != null) {
            doctor(doctorId);
        }
        MedicalService requestedService = null;
        if (serviceCode != null && !serviceCode.isBlank()) {
            requestedService = medicalService(serviceCode);
        }
    }
}

```

```

        List<AppointmentSlot> slots =
slotRepository.findAvailable(doctorId, serviceCode, date);
        if (requestedService == null) {
            return slots;
        }

        int durationMinutes =
requestedService.durationMinutes();
        return slots.stream()
            .filter(slot ->
!slot.endAt().isBefore(slot.startAt().plusMinutes(durationMinute
s)))
            .toList();
    }

    public AppointmentSlot slot(UUID id) {
        return slotRepository.findById(id)
            .orElseThrow(() -> new
NotFoundException("Schedule slot not found: " + id));
    }

    public List<KnowledgeArticle> knowledgeBase() {
        return knowledgeBaseRepository.findAll();
    }
}

```

Файл PatientService.java

```

package org.example.medicineservice.service;

import org.example.medicineservice.domain.Patient;
import org.example.medicineservice.dto.CreatePatientRequest;
import
org.example.medicineservice.exception.BusinessRuleException;
import org.example.medicineservice.exception.NotFoundException;
import org.example.medicineservice.repository.PatientRepository;
import org.springframework.stereotype.Service;

import java.util.Collection;
import java.util.UUID;

@Servicepublic class PatientService {

    private final PatientRepository patientRepository;

    public PatientService(PatientRepository patientRepository) {
        this.patientRepository = patientRepository;
    }

    public Collection<Patient> patients() {
        return patientRepository.findAll();
    }
}

```

```

    }

    public Patient patient(UUID id) {
        return patientRepository.findById(id)
            .orElseThrow(() -> new
NotFoundException("Patient not found: " + id));
    }

    public Patient create(CreatePatientRequest request) {
        String phone = normalizePhone(request.phone());
        if (patientRepository.existsByPhone(phone)) {
            throw new BusinessException("Пациент с таким
телефоном уже существует");
        }

        Patient patient = new Patient(
            UUID.randomUUID(),
            normalizeFullName(request.fullName()),
            phone,
            request.birthDate()
        );
        return patientRepository.save(patient);
    }

    private String normalizeFullName(String fullName) {
        return fullName == null ? null :
fullName.strip().replaceAll("\\s+", " ");
    }

    private String normalizePhone(String phone) {
        if (phone == null) {
            return null;
        }
        String normalized = phone.strip().replaceAll("[^+0-9]",
""");
        if (normalized.startsWith("8") && normalized.length() ==
11) {
            return "+7" + normalized.substring(1);
        }
        return normalized;
    }
}

```

Файл KnowledgeEmbeddingIndexer.java

```

package org.example.medicineservice.service;

import jakarta.annotation.PreDestroy;
import org.example.medicineservice.domain.KnowledgeArticle;
import org.example.medicineservice.domain.KnowledgeArticleChunk;
import
org.example.medicineservice.repository.KnowledgeBaseRepository;

```

```

import
org.example.medicineservice.repository.KnowledgeVectorRepository
;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import
org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Service;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.HexFormat;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.atomic.AtomicBoolean;

@Service
public class KnowledgeEmbeddingIndexer {

    private static final Logger log =
LoggerFactory.getLogger(KnowledgeEmbeddingIndexer.class);
    private static final int MAX_CHUNK_LENGTH = 1200;

    private final KnowledgeBaseRepository
knowledgeBaseRepository;
    private final KnowledgeVectorRepository
knowledgeVectorRepository;
    private final ClinicalEmbeddingClient embeddingClient;
    private final boolean autoIndex;
    private final int maxUpdatesPerRun;
    private final long embeddingDelayMs;
    private final int embeddingMaxAttempts;
    private final long embeddingRetryDelayMs;
    private final ExecutorService indexingExecutor;
    private final AtomicBoolean indexing = new
AtomicBoolean(false);
    private volatile IndexProgress progress =
IndexProgress.idle();

    public KnowledgeEmbeddingIndexer(KnowledgeBaseRepository
knowledgeBaseRepository,
                                   KnowledgeVectorRepository
knowledgeVectorRepository,
                                   ClinicalEmbeddingClient
embeddingClient,
                                   @Value("${medicine.ai.rag.auto-index:true}") boolean autoIndex,

```

```

@Value("${medicine.ai.rag.max-updates-per-run:0}") int
maxUpdatesPerRun,

@Value("${medicine.ai.rag.embedding-delay-ms:750}") long
embeddingDelayMs,

@Value("${medicine.ai.rag.embedding-max-attempts:5}") int
embeddingMaxAttempts,

@Value("${medicine.ai.rag.embedding-retry-delay-ms:10000}") long
embeddingRetryDelayMs) {
    this.knowledgeBaseRepository = knowledgeBaseRepository;
    this.knowledgeVectorRepository =
knowledgeVectorRepository;
    this.embeddingClient = embeddingClient;
    this.autoIndex = autoIndex;
    this.maxUpdatesPerRun = maxUpdatesPerRun;
    this.embeddingDelayMs = embeddingDelayMs;
    this.embeddingMaxAttempts = embeddingMaxAttempts;
    this.embeddingRetryDelayMs = embeddingRetryDelayMs;
    this.indexingExecutor =
Executors.newSingleThreadExecutor(indexingThreadFactory());
}

@EventListener(ApplicationReadyEvent.class)
public void indexOnStartup() {
    if (autoIndex) {
        startIndexing("startup");
    }
}

@PreDestroy    public void shutdown() {
    indexingExecutor.shutdownNow();
}

public boolean startIndexing(String trigger) {
    if (!indexing.compareAndSet(false, true)) {
        log.info("Knowledge vector indexing already running;
skip {} request", trigger);
        return false;
    }
    progress = IndexProgress.running(0, 0, 0, 0, trigger);
    indexingExecutor.submit(() -> runIndexing(trigger));
    return true;
}

public int reindex() {
    if (!indexing.compareAndSet(false, true)) {
        log.info("Knowledge vector indexing already running;
skip manual reindex request");
        return 0;
    }
}

```

```

        return runIndexing("manual");
    }

    public IndexProgress progress() {
        return progress;
    }

    public int totalChunks() {
        return prepareChunks().size();
    }

    private int runIndexing(String trigger) {
        try {
            return indexAll(embeddingClient, trigger);
        } finally {
            indexing.set(false);
        }
    }

    private int indexAll(ClinicalEmbeddingClient
embeddingClient, String trigger) {
        List<PreparedChunk> preparedChunks = prepareChunks();
        List<String> indexedChunkIds = preparedChunks.stream()
            .map(preparedChunk ->
preparedChunk.chunk().id())
            .toList();
        int totalChunks = indexedChunkIds.size();
        int processedChunks = 0;
        int updatedChunks = 0;
        int failedChunks = 0;
        boolean stoppedEarly = false;
        String lastError = null;
        progress = IndexProgress.running(totalChunks, 0, 0, 0,
trigger);
        log.info("Knowledge vector indexing started: trigger={},
chunks={}", trigger, totalChunks);

        for (PreparedChunk preparedChunk : preparedChunks) {
            KnowledgeArticleChunk chunk = preparedChunk.chunk();
            processedChunks++;
            if (isAlreadyIndexed(chunk)) {
                progress = IndexProgress.running(totalChunks,
processedChunks, updatedChunks, failedChunks, trigger);
                continue;
            }
            if (maxUpdatesPerRun > 0 && updatedChunks >=
maxUpdatesPerRun) {
                stoppedEarly = true;
                lastError = "update limit reached: " +
maxUpdatesPerRun;
                log.info("Knowledge vector index update limit
reached: {} updates per run", maxUpdatesPerRun);
                break;
            }
        }
    }

```

```

        }
        try {
            float[] embedding =
embedWithRetry(embeddingClient, preparedChunk);
            knowledgeVectorRepository.upsert(chunk,
embedding);
            updatedChunks++;
        } catch (RuntimeException exception) {
            failedChunks++;
            stoppedEarly = true;
            lastError = exception.getMessage();
            log.warn("Knowledge vector indexing stopped
after {} updates: {}", updatedChunks, lastError);
            break;
        }
        progress = IndexProgress.running(totalChunks,
processedChunks, updatedChunks, failedChunks, trigger);
    }

knowledgeVectorRepository.deleteMissing(indexedChunkIds);
    progress = IndexProgress.finished(totalChunks,
processedChunks, updatedChunks, failedChunks, stoppedEarly,
lastError, trigger);
    log.info("Knowledge vector index refreshed: {} chunks,
{} updated{}",
        totalChunks,
        updatedChunks,
        stoppedEarly ? " (partial run)" : "");
    return totalChunks;
}

private List<PreparedChunk> prepareChunks() {
    List<PreparedChunk> preparedChunks = new ArrayList<>();
    for (KnowledgeArticle article :
knowledgeBaseRepository.findAll()) {
        List<String> chunks = split(article.content());
        for (int index = 0; index < chunks.size(); index++)
        {
            String content = chunks.get(index);
            String embeddingText = article.title() + "\n" +
content;
            KnowledgeArticleChunk chunk = new
KnowledgeArticleChunk(
                article.id() + ":" + index,
                article.id(),
                index,
                article.title(),
                content,
                sha256(embeddingText),
                0
            );
            preparedChunks.add(new PreparedChunk(chunk,
embeddingText));
        }
    }
}

```

```

        }
    }
    return preparedChunks;
}

private float[] embedWithRetry(ClinicalEmbeddingClient
embeddingClient, PreparedChunk preparedChunk) {
    int attempts = Math.max(1, embeddingMaxAttempts);
    RuntimeException lastException = null;
    for (int attempt = 1; attempt <= attempts; attempt++) {
        delay(embeddingDelayMs);
        try {
            return
embeddingClient.embed(preparedChunk.embeddingText());
        } catch (RuntimeException exception) {
            lastException = exception;
            if (!isRateLimited(exception) || attempt ==
attempts) {
                throw exception;
            }
            long retryDelay = embeddingRetryDelayMs *
attempt;
            log.warn("Embedding rate limit for chunk {}.
Retry {}/{}, after {} ms",
                preparedChunk.chunk().id(), attempt + 1,
attempts, retryDelay);
            delay(retryDelay);
        }
    }
    throw lastException == null ? new
IllegalStateException("Embedding retry failed without
exception") : lastException;
}

private boolean isRateLimited(RuntimeException exception) {
    String message = exception.getMessage();
    return message != null && (message.contains("HTTP 429")
|| message.contains("rate_limited") || message.contains("Rate
limit"));
}

private void delay(long delayMs) {
    if (delayMs <= 0) {
        return;
    }
    try {
        Thread.sleep(delayMs);
    } catch (InterruptedException exception) {
        Thread.currentThread().interrupt();
        throw new IllegalStateException("Knowledge vector
indexing interrupted", exception);
    }
}
}

```

```

        private boolean isAlreadyIndexed(KnowledgeArticleChunk
chunk) {
            return
knowledgeVectorRepository.findIndexedContentHash(chunk.id())
                .filter(chunk.contentHash()::equals)
                .isPresent();
        }

        private List<String> split(String content) {
            String normalized = content == null ? "" :
content.strip();
            if (normalized.length() <= MAX_CHUNK_LENGTH) {
                return List.of(normalized);
            }

            List<String> chunks = new ArrayList<>();
            int start = 0;
            while (start < normalized.length()) {
                int end = Math.min(normalized.length(), start +
MAX_CHUNK_LENGTH);
                int sentenceEnd = normalized.lastIndexOf('.', end);
                if (sentenceEnd > start + 200) {
                    end = sentenceEnd + 1;
                }
                chunks.add(normalized.substring(start,
end).strip());
                start = end;
            }
            return chunks;
        }

        private String sha256(String value) {
            try {
                MessageDigest digest =
MessageDigest.getInstance("SHA-256");
                return
HexFormat.of().formatHex(digest.digest(value.getBytes(StandardCh
arsets.UTF_8)));
            } catch (NoSuchAlgorithmException exception) {
                throw new IllegalStateException("SHA-256 is not
available", exception);
            }
        }

        private ThreadFactory indexingThreadFactory() {
            return runnable -> {
                Thread thread = new Thread(runnable, "knowledge-
vector-indexer");
                thread.setDaemon(true);
                return thread;
            };
        }
    }

```

```

        private record PreparedChunk(KnowledgeArticleChunk chunk,
String embeddingText) {
        }

        public record IndexProgress(
                boolean running,
                int totalChunks,
                int processedChunks,
                int updatedChunks,
                int failedChunks,
                boolean partial,
                String lastError,
                String trigger,
                LocalDateTime startedAt,
                LocalDateTime finishedAt
        ) {
                static IndexProgress idle() {
                        return new IndexProgress(false, 0, 0, 0, 0, false,
null, null, null, null);
                }

                static IndexProgress running(int totalChunks, int
processedChunks, int updatedChunks, int failedChunks, String
trigger) {
                        return new IndexProgress(true, totalChunks,
processedChunks, updatedChunks, failedChunks, false, null,
trigger, LocalDateTime.now(), null);
                }

                static IndexProgress finished(int totalChunks, int
processedChunks, int updatedChunks, int failedChunks,
                                boolean partial, String
lastError, String trigger) {
                        return new IndexProgress(false, totalChunks,
processedChunks, updatedChunks, failedChunks, partial,
                                lastError, trigger, null,
LocalDateTime.now());
                }
        }
}

```

Файл DoctorAvatarService.java

```

package org.example.medicineservice.service;

import org.example.medicineservice.domain.Doctor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class DoctorAvatarService {

```

```

        private static final List<String> BACKGROUNDS = List.of(
            "#166534", "#0f766e", "#0369a1", "#7c2d12",
            "#854d0e", "#6d28d9", "#be123c", "#374151"    );

        public String generateSvg(Doctor doctor) {
            String background =
BACKGROUNDS.get(Math.floorMod(doctor.id().hashCode(),
BACKGROUNDS.size()));
            String initials = initials(doctor.fullName());
            String name = doctor.fullName();
            String specialty = doctor.specialty();
            return ""
                <svg
xmlns="http://www.w3.org/2000/svg" width="320" height="320"
viewBox="0 0 320 320" role="img" aria-label="%s">
<rect width="320" height="320" rx="48" fill="%s"/>
<circle cx="250" cy="72" r="52" fill="rgba(255,255,255,0.16)"/>
<circle cx="72" cy="256" r="76" fill="rgba(255,255,255,0.12)"/>
<text x="160" y="146" text-anchor="middle" font-family="Inter,
Arial, sans-serif" font-size="84" font-weight="700"
fill="#ffffff">%s</text>
                <text x="160" y="210"
text-anchor="middle" font-family="Inter, Arial, sans-serif"
font-size="20" font-weight="700" fill="#ffffff">%s</text>
<text x="160" y="240" text-anchor="middle" font-family="Inter,
Arial, sans-serif" font-size="16"
fill="rgba(255,255,255,0.82)">%s</text>
                </svg>
"".formatted(escapeXml(name), background, escapeXml(initials),
escapeXml(compact(name, 22)), escapeXml(compact(specialty,
28)));
        }

        private String initials(String fullName) {
            if (fullName == null || fullName.isBlank()) {
                return "DR";
            }
            String[] parts = fullName.strip().split("\\s+");
            StringBuilder result = new StringBuilder();
            for (String part : parts) {
                if (!part.isBlank()) {
                    result.append(part.substring(0,
1).toUpperCase());
                }
                if (result.length() == 2) {
                    break;
                }
            }
            return result.isEmpty() ? "DR" : result.toString();
        }

        private String compact(String value, int maxLength) {
            if (value.length() <= maxLength) {
                return value;
            }
        }

```

```

        return value.substring(0, Math.max(0, maxLength -
1)).stripTrailing() + "...";
    }

    private String escapeXml(String value) {
        if (value == null) {
            return "";
        }
        return value
            .replace("&", "&amp;")
            .replace("<", "&lt;")
            .replace(">", "&gt;")
            .replace("\"", "&quot;")
            .replace("'", "&apos;");
    }
}

```

Файл SecurityPrincipalService.java

```

package org.example.medicineservice.security;

import org.example.medicineservice.domain.UserRole;
import
org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Service;

import java.util.UUID;

@Servicepublic class SecurityPrincipalService {

    public AuthenticatedUser requireUser(Authentication
authentication) {
        if (authentication == null ||
!(authentication.getPrincipal() instanceof AuthenticatedUser
user)) {
            throw new AccessDeniedException("Authentication is
required");
        }
        return user;
    }

    public boolean hasRole(AuthenticatedUser user, UserRole
role) {
        return user.role() == role;
    }

    public boolean canAccessPatient(AuthenticatedUser user, UUID
patientId) {
        if (user.role() == UserRole.ADMIN || user.role() ==
UserRole.DOCTOR) {
            return true;
        }
        return user.role() == UserRole.PATIENT &&

```

```

user.patientId() != null && user.patientId().equals(patientId);
    }

    public boolean canAccessDoctor(AuthenticatedUser user, UUID
doctorId) {
        if (user.role() == UserRole.ADMIN) {
            return true;
        }
        return user.role() == UserRole.DOCTOR && user.doctorId()
!= null && user.doctorId().equals(doctorId);
    }

    public void requirePatientAccess(AuthenticatedUser user,
UUID patientId) {
        if (!canAccessPatient(user, patientId)) {
            throw new AccessDeniedException("Нет доступа к
данным другого пациента");
        }
    }

    public void requireDoctorAccess(AuthenticatedUser user, UUID
doctorId) {
        if (!canAccessDoctor(user, doctorId)) {
            throw new AccessDeniedException("Нет доступа к
данным другого врача");
        }
    }
}

```

ПРИЛОЖЕНИЕ Б

(обязательное)

Веб-приложение с интеграцией AI-помощника для медицинской клиники

Презентация

Веб-приложение с интеграцией AI-помощника для медицинской клиники

Выполнил: Пантюшов Егор Игоревич, 24ВВИМ1

Руководитель: профессор д.т.н. Зинкин Сергей Александрович



Актуальность работы

Современные клиники активно используют веб-приложения, однако базового информирования — услуг, врачей, контактов — часто недостаточно. Пациенту нужна помощь в навигации: к какому врачу обратиться, что сделать перед приёмом и так далее.

Проблема

Без помощи AI нагрузка на персонал растёт, а пациент тратит больше времени

Потребности пациента

Нужно быстро сориентироваться в услугах и выбрать подходящего врача

Решение

AI-помощник как справочный модуль: использует базу знаний клиники, соблюдает медицинские ограничения



Цель и задачи исследования

Цель работы

Исследование, проектирование и разработка веб-приложения для медицинской клиники с интеграцией AI-помощника

01

Анализ предметной области

Изучение существующих веб-приложений медицинских клиник и их ограничений

02

Исследование AI-моделей

Выбор моделей для формирования ответов и поиска в базе знаний

03

Проектирование архитектуры

Клиентская часть, сервер, база данных и AI-модуль

04

Реализация и тестирование

Разработка функций системы и оценка качества AI-модуля

Made with **GRAMMAR**

Сравнительный анализ аналогов

Проведён анализ трёх популярных веб-приложений медицинских клиник города Пензы

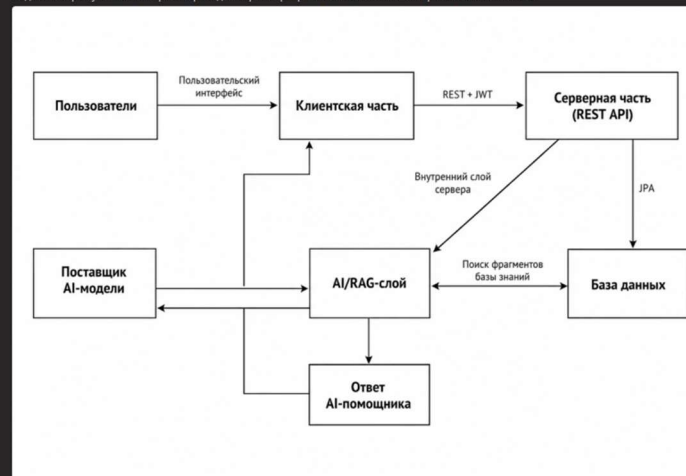
Критерий	Медгарт	Здоровье	Клиника Фомина	Разрабатываемое приложение
Каталог услуг и врачей	✓	✓	✓	✓
Запись на приём	✓	✓	✓	✓
Личный кабинет	✓	—	✓	✓
AI-помощник с базой знаний	—	—	—	✓

Вывод: существующие решения покрывают базовые задачи, но ни одно не реализует безопасную интеллектуальную помощь пациенту на основе базы знаний клиники.

Made with **GRAMMAR**

Архитектура системы

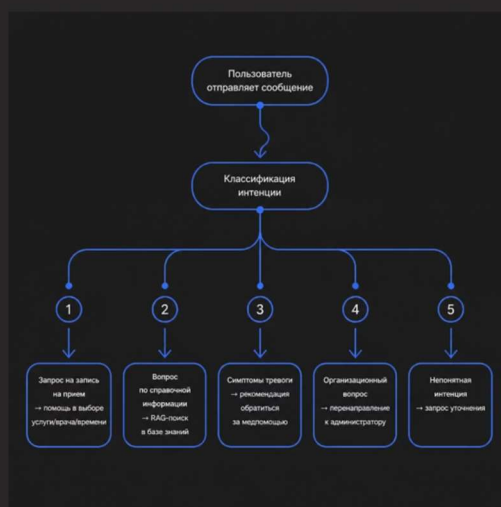
Система построена по клиент-серверному принципу с выделенным AI-модулем. Пользовательский интерфейс не обращается к AI-модели напрямую — все запросы проходят через сервер, что обеспечивает контроль и безопасность.



Разделение на слои позволяет независимо развивать каждую часть системы и строго контролировать работу AI-помощника через серверную логику.

Made with GAMMA

Логика обработки обращений



Контролируемая маршрутизация

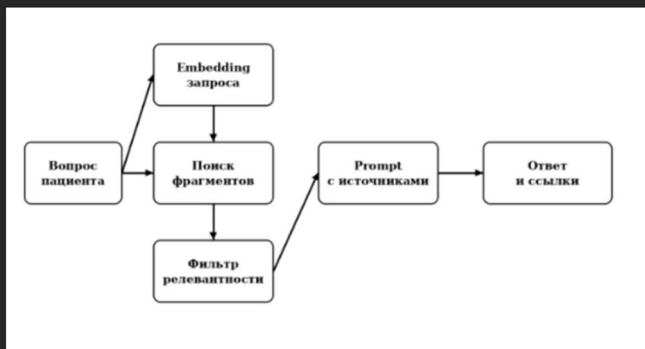
AI-помощник не отвечает «из головы». Сначала определяется тип обращения, затем выбирается безопасный сценарий.

- **Запись на приём** — навигация к выбору услуги, врача и времени
- **Справочный вопрос** — поиск в базе знаний через RAG
- **Тревожные симптомы** — рекомендация обратиться за помощью
- **Организационные вопросы** — направление к администратору

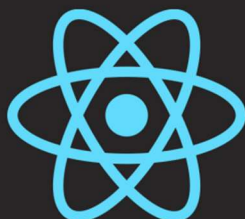
Made with GAMMA

RAG-подход: генерация с поиском

Retrieval-Augmented Generation — AI-помощник сначала ищет проверенные сведения в базе знаний клиники, а затем формирует ответ с опорой на найденные данные.



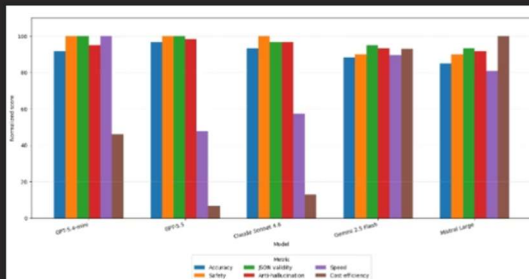
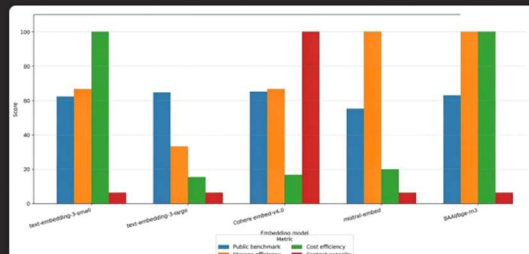
Стек технологий



Made with **GRANAAR**

Выбор AI-моделей

Сравнение различных языковых моделей и моделей векторных представлений по критериям качества, скорости и стоимости. Выбраны GPT-4o-mini и text-embedding-3-small как оптимальное решение.



Made with GAMMA

Результаты тестирования

0.95

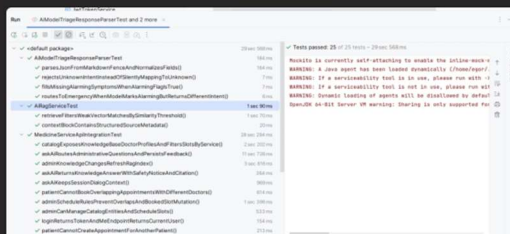
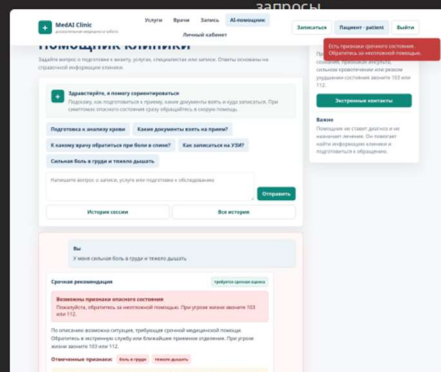
Средний балл качества AI
Оценка по контрольным сценариям

1.0

Определение типа обращения
Корректная классификация запросов

1.0

Обнаружение тревожных
симптомов
Безопасная реакция на критические
запросы



Made with GAMMA

Демонстрация веб-приложения

Посмотрим, как работает веб-приложение для медицинской клиники с интегрированным AI-помощником.

Made with **GLAMA**

Заключение

Разработано веб-приложение медицинской клиники с контролируемым AI-помощником, который помогает пациентам получать справочную информацию и не заменяет врача.

Ключевые достижения:

- Проведён анализ предметной области и аналогов
- Исследованы и выбраны оптимальные AI-модели
- Спроектирована безопасная архитектура системы
- Реализован AI-помощник с RAG и базой знаний клиники
- Внедрены ограничения медицинской безопасности
- Проведено комплексное тестирование приложения

