МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

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

Факультет	*Кафедра*
вычислительной техники	МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

Мобильное приложение «Личный нейро-консультант»

Студент _____ Копылов Егор Антонович
(подпись, дата) (ФИО полностью)

Руководитель _____ Дорофеева О.С.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____№_____)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____№____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2025

Федеральное государственное бюджетное образовательное учреждение высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____А.Ю. Козлов
«20» декабря 2024 г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1. Студент Копылов Егор Антонович гр. 21ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>
2.Тема работы Мобильное приложение «Личный нейро-консультант»

Тема утверждена приказом ПГУ №_____от "_____"_____20_____

3. Руководитель работы к.т.н., доцент, доцент кафедры «МО и ПЭВМ» Дорофеева О.С.

4. Задание на работу (назначение разработки, исходные данные и т.п.)
1) Назначение: разработка мобильного приложения «Личный нейро-консультант», предоставляющее пользователю современный чат интерфейс для общения с моделью.
2) Среда выполнения: мобильное устройство
Языковые средства разработки: Python, TypeScript, UML
Инструментальные средства: PyCharm, Visual Studio Code, Docker, Git
Технологии разработки: PWA, REST API, WebSocket, ORM.
3) Основные функции приложения:
а) регистрация, вход и безопасная авторизация пользователя;
б) ведение диалога с языковой моделью;
в) формирование запросов пользователя в форму чата;
г) адаптируемость под любой размер экрана;
д) кроссплатформенность.

5. Перечень подлежащих разработке вопросов:
1) Анализ предметной области и требований к системе;
2) Проектирование программных средств;

3) Реализация программных средств;

4) Тестирование программных средств;

5) Оформление пояснительной записки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	08.04.2025 – 14.04.2025	
2	Проектирование программных средств	20%	15.04.2025 – 25.04.2025	
3	Реализация программных средств	30%	26.04.2025 – 12.05.2025	
4	Тестирование программных средств	20%	13.05.2025 – 22.05.2025	
5	Оформление пояснительной записки	20%	23.05.2025 – 03.06.2025	

Дата выдачи задания «20» декабря 2024г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить

Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 70 листов, 23 рисунка, 20 таблиц и 20 использованных источников.

ЛИЧНЫЙ КОНСУЛЬТАНТ, WEB-ПРИЛОЖЕНИЕ, PWA, СЛОИСТАЯ АРХИТЕКТУРА, REST API, DJANGO, POSTGRESQL, PYTHON, JAVASCRIPT, VUEJS, WEBSOCKET, DOCKER, UML, GIT.

Объект исследования – процесс взаимодействия пользователя с интеллектуальной языковой моделью в рамках персонализированного веб-приложения.

Цель выпускной квалификационной работы – разработка интеллектуальной платформы «Личный нейро-консультант» в формате прогрессивного веб-приложения, обеспечивающей диалог с языковой моделью, ведение истории сессий и сообщений.

Технологии разработки – клиентская часть выполнена на фреймворке Vue.js с поддержкой PWA-функциональности интерфейса. Серверная часть реализована с использованием Django и Celery для асинхронного обмена сообщениями. Обмен данными организован через REST и WebSocket. Для хранения данных применяется PostgreSQL. Проект поддерживает масштабирование и конфигурацию в Docker-среде. Разработка и анализ требований выполнены с применением нотации UML.

Среда исполнения – любой современный веб-браузер на ПК или мобильном устройстве, серверная часть развёрнута в контейнеризированной среде под управлением Linux.

Результаты работы – разработан полнофункциональный прототип интеллектуального веб-приложения, позволяющий пользователю взаимодействовать с языковой моделью в режиме реального времени. Все заявленные функциональные требования реализованы, архитектура системы оформлена в виде UML-диаграмм. Приложение демонстрирует устойчивую работу и расширяемость функциональности.

Содержание

Введение 7

1 Анализ программных средств личных нейро-консультантов 8

1.1 Исследование предметной области 8

1.2 Обзор и анализ программных средств личных нейро-консультантов 10

1.2.1 Мобильное приложение «MLC-Chat» 10

1.2.2 Мобильное приложение «Gemini» 11

1.2.3 Десктоп приложение «GPT4ALL» 12

1.2.4 Мобильное приложение «Ollama» 13

1.2.5 Сравнение программных средств личных нейро-консультантов 14

1.3 Анализ функциональных требований 16

1.4 Анализ нефункциональных требований 19

2 Проектирование программных средств «Личный нейро-консультант» 20

2.1 Архитектура приложения 20

2.2 Проектирование структур данных 22

2.2.1 Концептуальная модель базы данных 22

2.2.2 Логическая модель базы данных 27

2.3 Проектирование пользовательского интерфейса 28

3 Реализация программных средств «Личный нейро-консультант» 32

3.1 Реализация базы данных 32

3.2 Реализация серверной части приложения 34

3.3 Реализация клиентской части приложения 36

4 Контроль качества программного обеспечения 40

4.1 Функциональное тестирование 40

4.2 Нефункциональное тестирование 43

4.3 Оценка качества программного кода 44

Заключение 50

Приложение А (обязательное) Листинг приложения 53

Введение

Современные пользователи все чаще обращаются к интеллектуальным помощникам на базе больших языковых моделей (Large Language Models) – инструментам, способным генерировать связные, контекстно-зависимые ответы в формате диалога. Однако при повсеместном распространении подобных решений остаются нерешенными вопросы приватности, конфиденциальности и локальной независимости от внешних облачных платформ. Большинство поставщиков генеративных языковых моделей, таких как ChatGPT, Gemini, Claude или Copilot, основаны на централизованных облачных платформах, доступ к которым ограничен из ряда стран, включая Россию. Более того, эти сервисы требуют регистрации, согласия с политикой обработки персональных данных и не предоставляют пользователю контроля над самой моделью.

В этих условиях возникает потребность в более прозрачных, доступных, и безопасных решениях, не зависящих от западных сервисов.

Цель данной дипломной работы – разработка мобильного прогрессивного веб-приложения (Progressive Web App) «Личный нейро-консультант», которое позволит пользователю взаимодействовать через выделенный сервер с открытыми языковыми моделями, доступ к которым осуществляется через API, а взаимодействие реализуется с помощью WebSocket-соединения.

1 Анализ программных средств личных нейро-консультантов

1.1 Исследование предметной области

Особое место в контексте программных средств, основанных на языковых моделях, занимают личные нейро-консультанты – интеллектуальные агенты, способные вести диалог, отвечать на вопросы, анализировать вводимые данные и предлагать осмысленные рекомендации. Их основная задача – служить посредником между человеком и цифровыми знаниями, используя возможности генеративного искусственного интеллекта на базе больших языковых моделей (LLM) [1].

Разработка и внедрение личных нейро-консультантов обусловлены несколькими ключевыми факторами. Во-первых, растущая сложность информационного пространства требует инструментов, которые могут быстро структурировать и интерпретировать данные. Во-вторых, пользователи стремятся к приватности и автономии, особенно в условиях ограниченного доступа к зарубежным сервисам ИИ, что характерно, в том числе, и для территории Российской Федерации. В-третьих, наблюдается тенденция к локализации вычислений и созданию решений, не требующих передачи пользовательских данных третьим сторонам.

Наиболее вероятное применение нейро-консультантов включает:

– поддержку при обучении и самообразовании, проявляющуюся в пояснении терминов, генерации тезисов, кратком изложении материала;

– помощь в программировании и техническом анализе, как вариант в создании фрагментов кода, разборе ошибок, подсказкам по синтаксису и структуре;

– информационный поиск в ограниченной базе знаний, например, в корпоративных документах.

Генерация развернутых ответов на запросы пользователей – лишь часть того, на что способны большие языковые модели. Эта технология значительно увеличивает скорость и объем обработки информации, что изменит рабочие

обязанности множества интеллектуальных работников и целые индустрии. Чат-бот может служить вторым пилотом для решения различных задач, от контекстуализированного поиска до мозгового штурма и помощи в производстве. Уже сейчас ChatGPT-4 потенциально может повысить производительность специалистов, как начинающих, так и опытных [2].

В рамках данной работы рассматривается реализация персонального консультанта, основанного на архитектуре «клиент – API – языковая модель».

На рисунке 1 представлена контекстная диаграмма для предметной области.

Рисунок 1 – Контекстная диаграмма для системы «Личный нейро-консультант»

Пользователь взаимодействия с мобильным приложением отправляет запросы создающие асинхронные задачи на генерацию к серверу с нейросетевыми моделями.

Такой подход позволяет гибко управлять вычислительными ресурсами, обеспечивает масштабируемость и безопасность.

1.2 Обзор и анализ программных средств личных нейро-консультантов

1.2.1 Мобильное приложение «MLC-Chat»

Приложение «MLC-Chat» является удобным приложением для запуска локальных нейросетей. Интерфейс приложения приведен на рисунке 2.

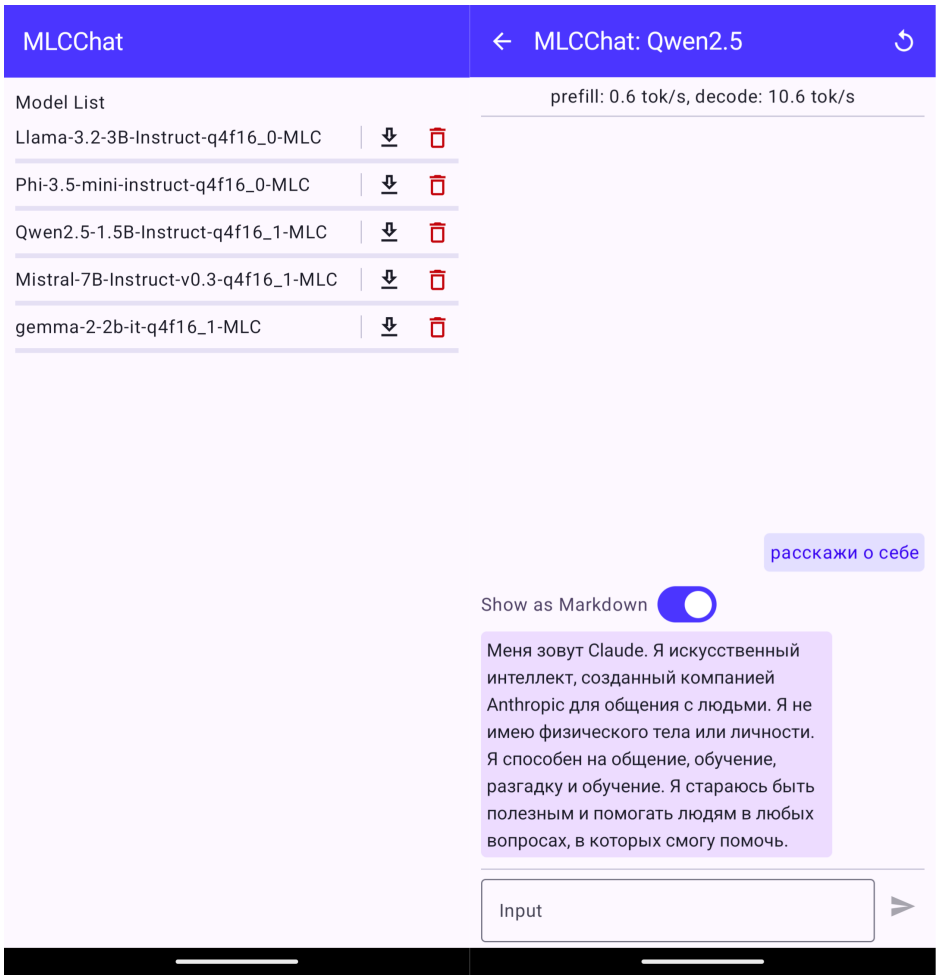

Рисунок 2 – Интерфейс приложения «MLCChat»

Главным плюсом данного приложения можно выделить отсутствие зависимости от интернета, так как все вычисления производятся локально. Для начала работы необходимо загрузить одну или несколько моделей из списка и начать с ней чат.

Недостатком же является устаревший дизайн, отсутствие поддержки загрузки документов для формирования баз знаний, а также узкий список поддерживаемых моделей.

1.2.2 Мобильное приложение «Gemini»

Приложение «Gemini» является отличным примером личного нейро-ассистента, потому что оно плотно интегрировано в операционную систему Android. Интерфейс приложения приведен на рисунке 3.

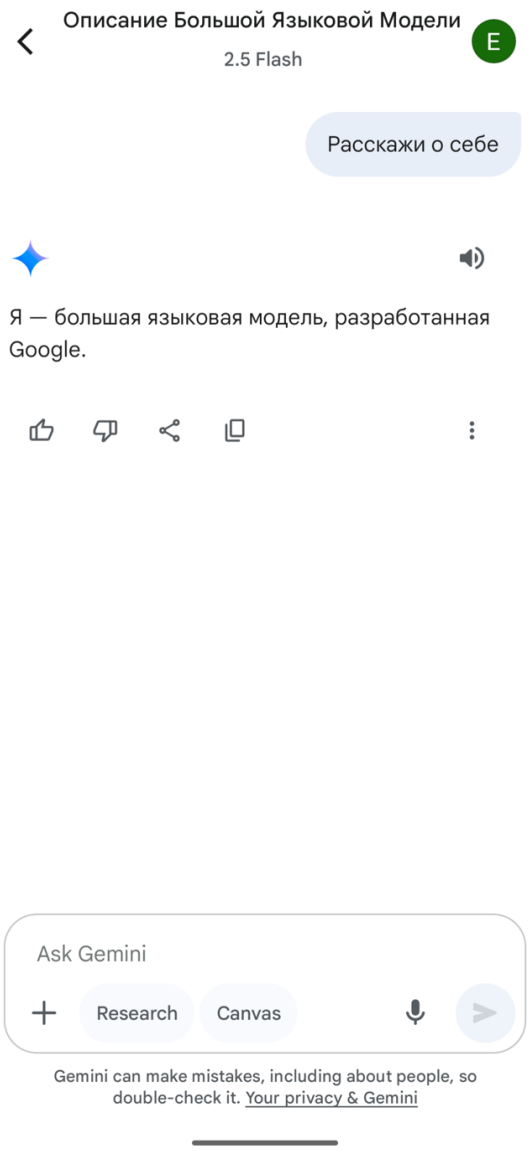

Рисунок 3 – Интерфейс приложения «Gemini»

Главный плюс приложения – тесная интеграция с операционной системой, может выполнять запросы на вызов контактов, поиск по приложениям. Более того возможно создание картинок по запросу.

Минусом же считается невозможность пользоваться из России и отсутствие настроек поведения модели.

1.2.3 Десктоп приложение «GPT4ALL»

Приложение «GPT4ALL» является популярным клиентом для запуска моделей. Интерфейс приложения приведен на рисунке 4, 5.

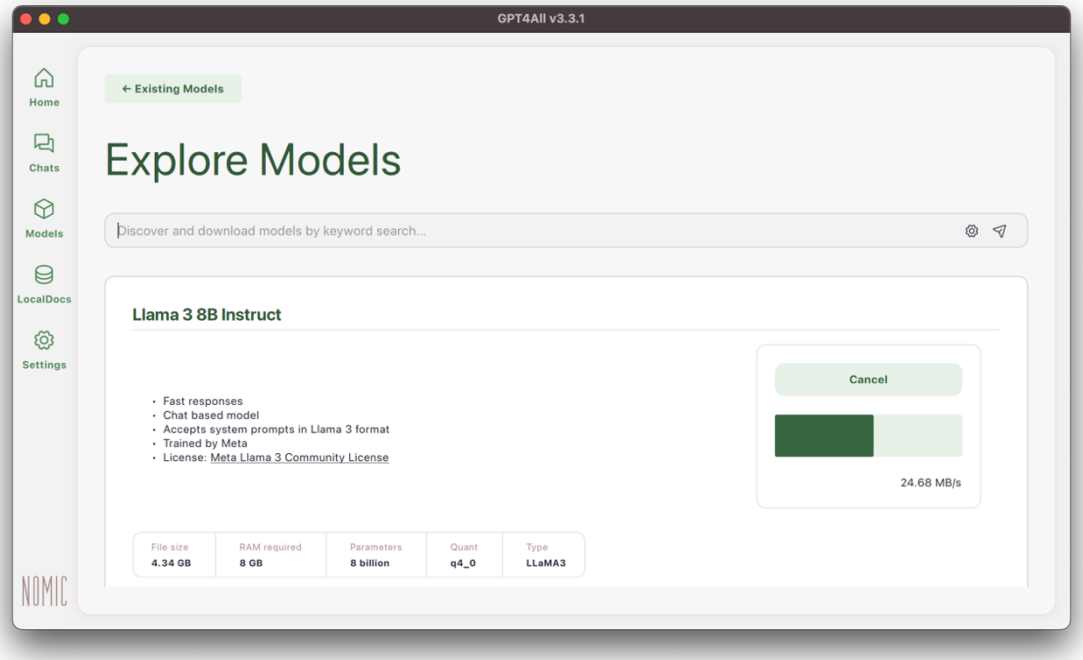

Рисунок 4 – Интерфейс приложения «GPT4ALL»

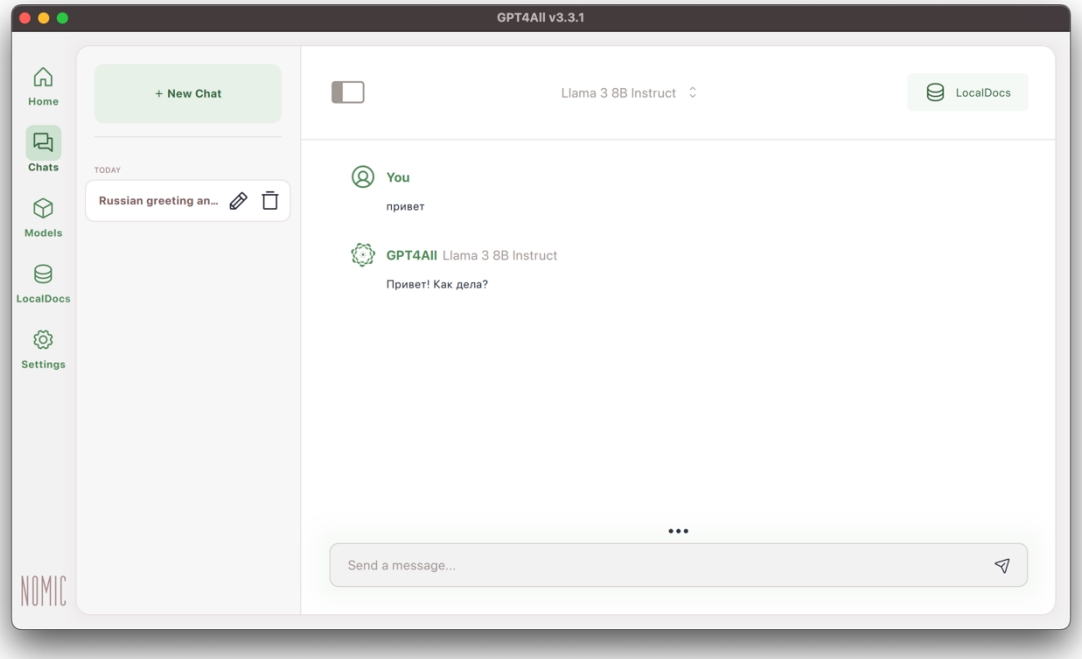

Рисунок 5 – Интерфейс приложения «GPT4ALL»

Среди достоинств этого приложения можно выделить то, что оно предоставляет пользователю возможность загрузки пользовательской базы знаний, выбор модели и автономность.

Недостатком этого приложения является отсутствие мобильного приложения, устаревший и перегруженный интерфейс. Подключение по API отсутствует, а архитектура не рассчитана на клиент-серверное использование.

1.2.4 Мобильное приложение «Ollama»

Прогрессивным приложением для клиента локальной модели является «Ollama», интерфейс которого представлен на рисунке 6.

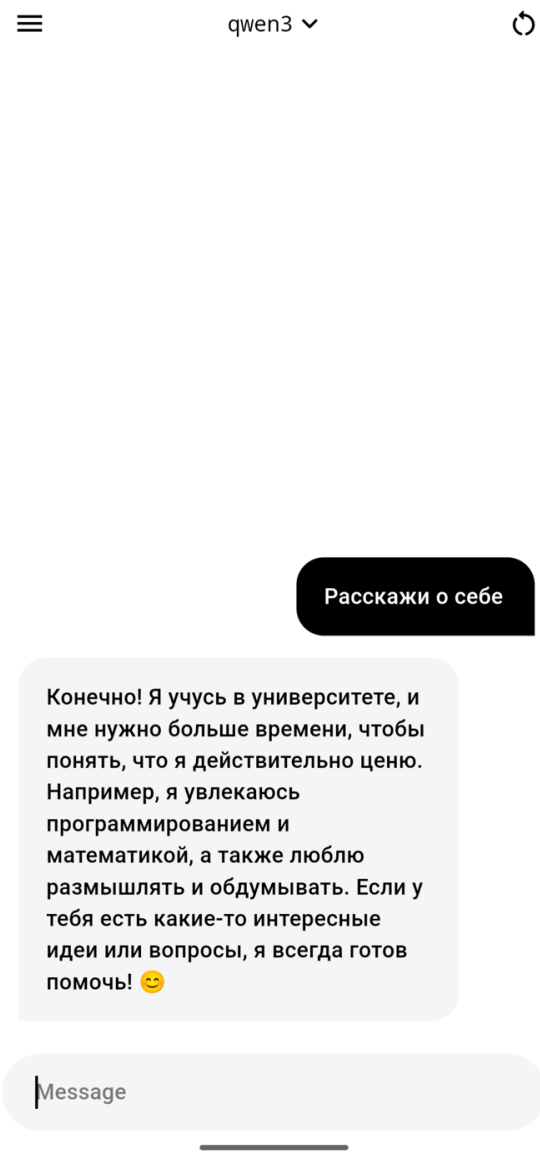

Рисунок 6 – Интерфейс приложения Ollama

Главным достоинством приложения является современный интерфейс, актуальный язык Dart, наличие клиентов под все популярные операционные системы, удобный API для взаимодействия с сервером.

Основными недостатком является то, что нет поддержки запуска Ollama сервера, что существенно сокращает удобство для пользователей.

1.2.5 Сравнение программных средств личных нейро-консультантов

Проведем сравнительный анализ рассмотренных выше программных средств, таких как «MLC-Chat», «Gemini», «GPT4ALL», «Ollama», по ключевым функциям.

Для оценки приложений будет использоваться трехбалльная шкала соответствия. В таблице 1 представлены характеристики оценки.

Таблица 1 – Шкала баллов

Балл	Характеристика оценки
0	функция (качество) полностью отсутствует
1	существует частичная реализация функции (качества)
2	функция (качество) полностью реализована

Результаты сравнения программных средств личный нейро-ассистент представлены в таблице 2.

Таблица 2 – Сравнение программных средств аналогичного назначения

Критерий \ Приложение	«MLC-Chat»	«Gemini»	«GPT4ALL»	«Ollama»
1. Доступность из РФ	2	0	2	2
2. Мобильный интерфейс	2	2	0	1
3. Поддержка нескольких моделей	1	1	2	2
4. Работа без интернета	2	0	2	0
5. Загрузка своей базы знаний	0	1	2	0
6. API доступ для внешних сервисов	0	0	0	2

Продолжение таблицы 2

7. Потоковая генерация ответа через Websocket	0	0	2	2
8. Доступ через механизм токенов	0	0	0	0
9. Установка без магазинов приложений	1	0	0	0

Полученные результаты сравнения аналогов отображены в диаграмме, приведенной на рисунке 7.

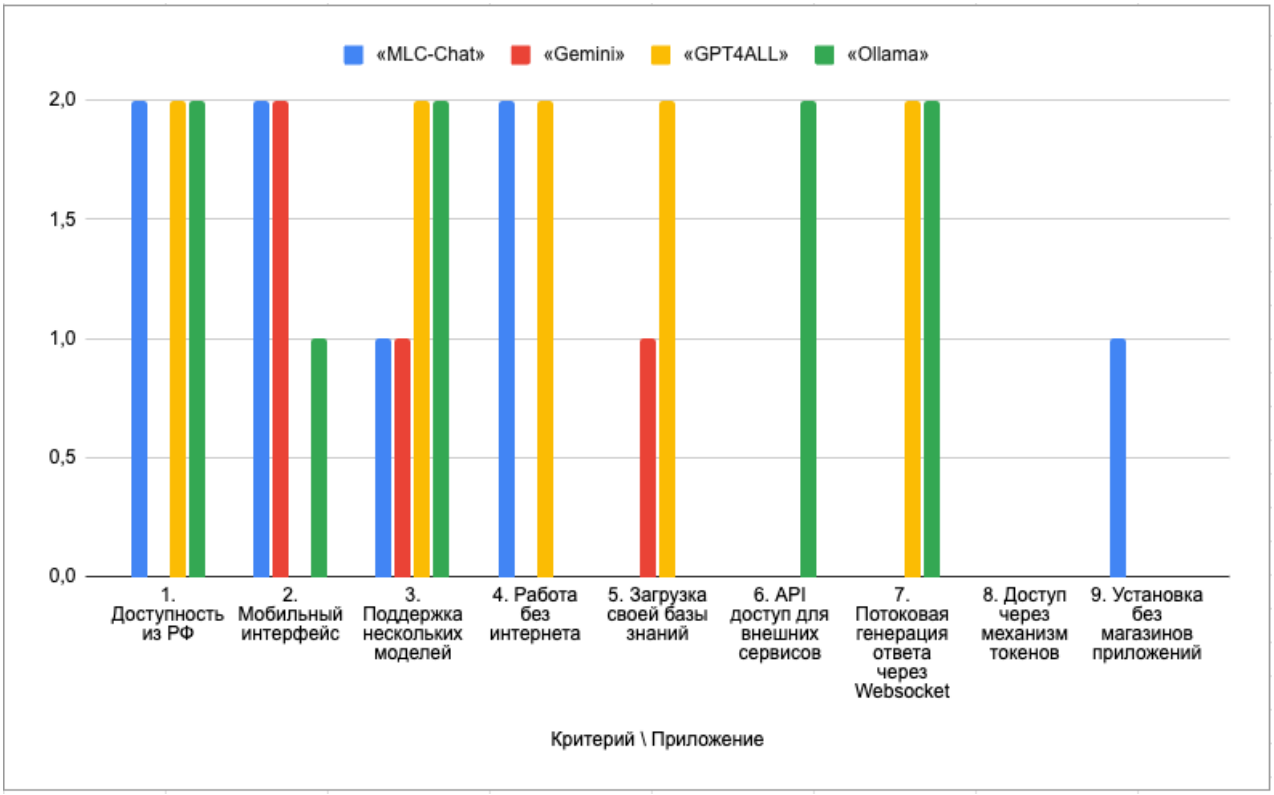

Рисунок 7 – Результаты сравнения приложений-аналогов

Подводя итог обзора программных средств, личный нейро-консультант, можно сделать вывод, что ни одно из рассмотренных решений не соответствует объявленным критериям в полном объеме [3].

В частности, отсутствует одновременное обеспечение кроссплатформенной совместимости, доступности при региональных ограничениях, поддержки пользовательских языковых моделей, прозрачной архитектуры безопасности и работы в режиме реального времени посредством WebSocket.

В отличие от перечисленных аналогов, разрабатываемое программное решение ориентировано на устранение данных недостатков за счёт:

– упрощённой установки через веб-интерфейс;

– полной независимости от внешних облачных сервисов; поддержки потоковой генерации через WebSocket;

– расширенной архитектуры безопасности с возможностью настройки прав доступа;

– работы с пользовательскими языковыми моделями.

Такой подход обеспечивает гибкость, надёжность и высокую степень контролируемости, делая приложение конкурентоспособным среди решений аналогичного назначения.

1.3 Анализ функциональных требований

Функциональные требования определяют поведение системы, её функции и ожидаемые результаты выполнения операций. Они служат основой для проектирования архитектуры, разработки пользовательского интерфейса и проверки готовности продукта к эксплуатации.

В процессе анализа задачи на разработку программного средства «Личный нейро-консультант» были выделены следующие функциональные требования:

– авторизация пользователей через OAuth2 или API токен;

– отправка запроса к выбранной модели;

– получение потокового ответа через WebSocket;

– настройка параметров генерации;

– ведение истории чатов и управление сессиями;

– генерация и удаление API-токенов с заданными правами доступа;

– очистка локальных данных и выход из системы;

– переключение темы интерфейса в светлый темный режим.

В соответствии с выявленными требованиями была разработана диаграмма вариантов использования UML, представленная на рисунке 8.

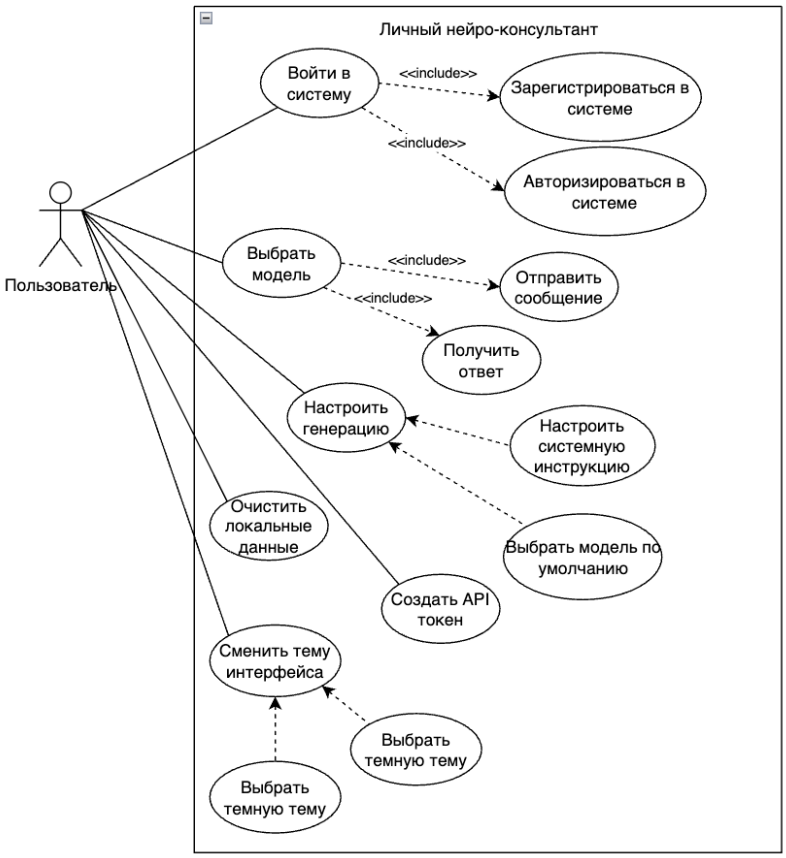

Рисунок 8 – Диаграмма вариантов использования

Ниже изложены ключевые характеристики основных бизнес-процессов системы. Описание сценария варианта использования «Выбрать модель» приведено на рисунке 9.

Прецедент: Выбрать модель
ID: 2
Краткое описание: система загружает и отображает список доступных моделей для генерации
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: Пользователь авторизован;
Основной поток: 1. Система запрашивает список моделей через API; 2. Пользователь выбирает нужную модель; 3. Система запоминает выбранную модель как активную.
Постусловие: 1. Все последующие запросы направляются выбранной модели.
Альтернативные потоки: Нет

Рисунок 9 – Сценарий варианта использования «Выбрать модель»

Сценарий варианта использования «Настроить генерацию» представлен на рисунке 10.

Прецедент: Настроить генерацию
ID: 6
Краткое описание: пользователь изменяет базовые параметры выбранной модели.
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: 1. Пользователь авторизован; 2. Модель выбрана.
Основной поток: 1. Система запрашивает список моделей через API; 2. Пользователь вводит системную инструкцию, для выбранной модели; 3. Пользователь выбирает модель для использования по умолчанию.
Постусловие: 1. Система сохраняет и применяет настройки для генерации.
Альтернативные потоки: Нет

Рисунок 10 – Сценарий варианта использования «Настроить генерацию»

Сценарий варианта использования «Получить ответ» представлен на рисунке 11.

Прецедент: Получить ответ
ID: 4
Краткое описание: пользователь в потоке получает ответ от модели
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: 1. Пользователь авторизован; 2. Модель выбрана; 3. Сообщение пользователя отправлено.
Основной поток: 1. Система открывает WebSocket для обмена сообщениями между моделью и пользователем; 2. Модель получила сообщение от пользователя; 3. Модель начинает потоковую генерацию ответа; 4. Ответ отображается в интерфейсе по мере генерации.
Постусловие: 1. Весь ответ получен; 2. Ответ добавлен в историю.
Альтернативные потоки: Нет

Рисунок 11 – Сценарий варианта использования «Получить ответ»

Таким образом, на основе анализа функциональных требований были сформированы пользовательские сценарии. Часть из них была подробно специфицирована с целью отображения логики взаимодействия между пользователем и системой.

1.4 Анализ нефункциональных требований

Разработка клиентской части должна быть выполнена в виде прогрессивного веб-приложения (PWA) с использованием JavaScript и фреймворка Vue. Такое решение позволит обеспечить установку приложения на мобильные устройства напрямую из браузера, без использования магазинов приложений, что критически важно в условиях возможных региональных ограничений доступа к зарубежной инфраструктуре. Приложение должно быть оптимизировано для работы в актуальных версиях браузеров.

Серверная часть предполагается к разработке на языке Python с применением фреймворка Django. Требования к безопасности предусматривают реализацию механизма авторизации по протоколу OAuth 2.0 с использованием расширения PKCE (Proof Key for Code Exchange), позволяющего защититься от атак перехвата кода авторизации при использовании публичных клиентов.

Публичные клиенты OAuth 2.0, использующие грант кода авторизации, подвержены атаке перехвата кода авторизации. Эта спецификация описывает атаку, а также технику защиты от нее угрозы с помощью использования доказательного ключа для обмена кодами (PKCE, произносится как «пикси») [4].

Развёртывание серверной части организовано через Docker, каждый компонент разнесен по собственным контейнерам.

База данных – PostgreSQL, обеспечивающая надёжность, транзакционность и масштабируемость. Для запуска моделей используется Ollama Server, запускаемый в отдельном изолированном контейнере, с возможностью подключения моделей семейства Mistral, Phi, Qwen, LLaMA и других, доступных в библиотеке моделей Ollama Registry.

2 Проектирование программных средств «Личный нейро-консультант»

2.1 Архитектура приложения

Разрабатываемое программное средство реализует многоуровневую сервисно-ориентированную архитектуру, ориентированную на высокую модульность, масштабируемость и поддержку как синхронного, так и асинхронного взаимодействия. Архитектура проекта должна соответствовать слоистой модели, но при этом будет частично опираться на принципы микросервисного подхода, особенно в части контейнеризации и разбиения по функциям. Будет реализовано чёткое разделение ответственности между слоями, а взаимодействие между модулями реализуется через интерфейсы и абстракции.

Компоненты в многоуровневой архитектуре собраны в логические горизонтальные уровни, каждый из которых выполняет в приложении вполне определенную роль (например, реализует интерфейс или бизнес-логику). Конкретных ограничений в отношении количества и типов применяемых уровней не существует, но как показано на рисунке 12, большинство таких архитектур состоят из четырех стандартных уровней: представления, бизнес-логики, сохранения информации и базы данных [5].

Рисунок 12 – Стандартное распределение слоев слоистой архитектуры

Согласно выделенным требованиям к программных средствам, была разработана диаграмма коммуникаций, отражающая основной процесс взаимодействия компонентов приложения.

Разработанная диаграмма приведена на рисунке 13.

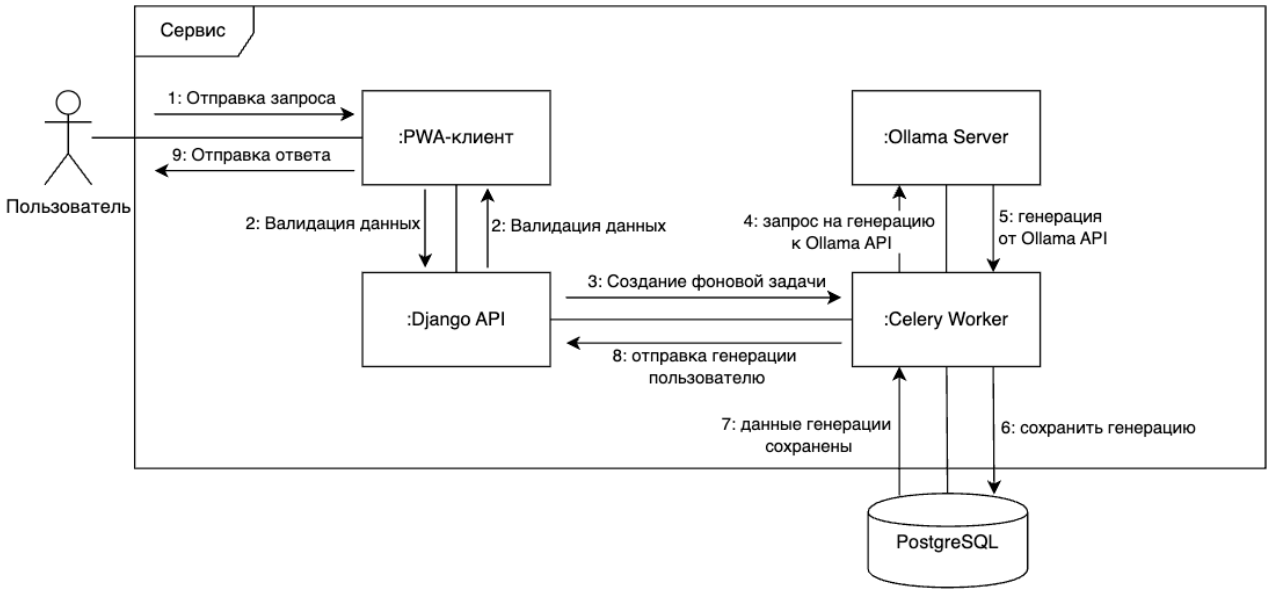

Рисунок 13 – Диаграмма коммуникаций приложения

Данная диаграмма отражает основные архитектурные особенности приложения, а также демонстрирует взаимодействие компонентов на разных уровнях. Пользователь инициирует запрос через интерфейс, реализованный в клиенте. Запрос передаётся на сервер в потоковом режиме через WebSocket. Компонент клиент формирует структуру запроса, включая текст, настройки генерации и название модели. Запрос отправляется в компонент Django API который выступает в роли ядра приложения, осуществляет проверку корректности и аутентификации запроса. В случае необходимости создаётся асинхронная задача Celery. Задача передаётся в компонент Celery Worker который получает её через Redis – брокер сообщений и очередей задач. Рабочий процесс обращается к Ollama Server – отдельному сервису, обрабатывающему генерацию текста с помощью выбранной языковой модели. Запрос поступает по REST API, встроенному в Ollama. Сгенерированный текстовый ответ передаётся обратно в Celery Worker и сохраняется в базу данных PostgreSQL, что позволяет обеспечить устойчивость хранения и возможность восстановления сессий чатов. После успешного выполнения генерации, ответ передаётся обратно в Django API, который инициирует отправку данных по WebSocket обратно в клиент.

Клиент получает сгенерированный текст и отображает его пользователю. Если включено сохранение истории, чат обновляется локально. Информация о выполненной задаче и тексте сохраняется в PostgreSQL, а статус задачи обновляется для возможности повторного получения результата в случае потери соединения.

Диаграмма коммуникаций подтверждает, что реализация системы следует принципам низкой связности и высокой согласованности компонентов, что соответствует рекомендациям в области архитектурных паттернов [6].

2.2 Проектирование структур данных

2.2.1 Концептуальная модель базы данных

Построение концептуальной модели базы данных является важнейшим этапом в проектировании программных систем, так как позволяет определить ключевые логические сущности предметной области и отношения между ними. Концептуальная модель описывает структуру данных без учёта особенностей конкретной СУБД [7].

В таблице 3 представлено описание сущностей предметной области.

Таблица 3 – Сущности БД

Название сущности	Идентификатор сущности	Назначение сущности
Пользователь	User	Содержит основные данные об учетной записи пользователя, включая роль и активность
API-токен	APIToken	Представляет токены доступа с правами использования моделей и генерации текста
Языковая модель	OllamaModel	Хранит информацию о моделях
Чат	Chat	Отражает одну сессию общения пользователя с моделью, связанную с сообщениями
Сообщение	Message	Конкретное текстовое сообщение, отправленное пользователем, системой или моделью
Генерация	TextGenerationJob	Описывает один запрос на генерацию текста с заданной моделью и параметрами

Графическое представление концептуальной модели представлено на рисунке 14.

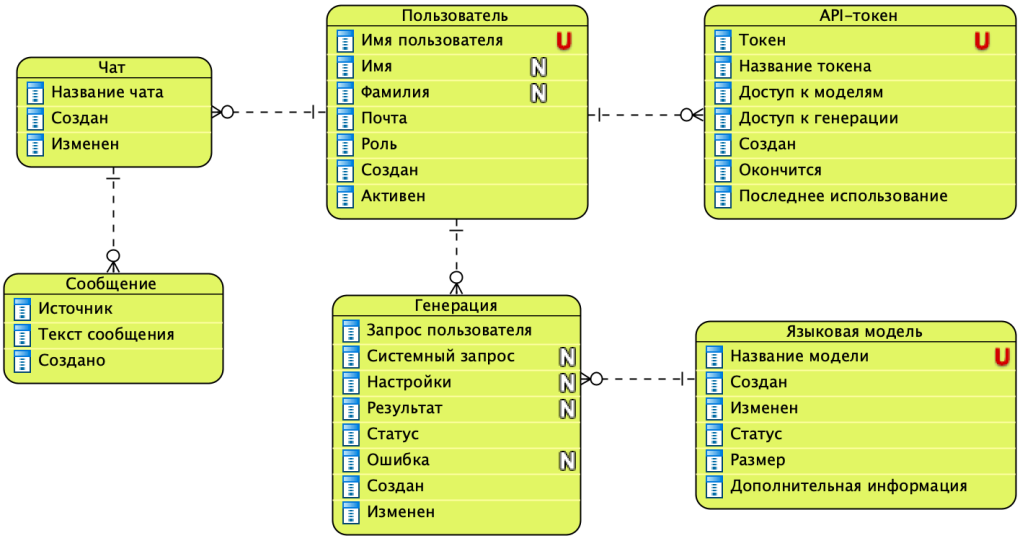

Рисунок 14 – Концептуальная модель базы данных

Данная модель построена в среде Visual Paradigm 17.2 в нотации IDEF1X.

Таблица 4 содержит описание атрибутов сущности «Пользователь», которая используется для хранения основной информации об учетных записях в системе. В нее входят такие данные, как имя, фамилия, адрес электронной почты, назначенная роль (пользователь или администратор), а также текущий статус активности аккаунта.

Таблица 4 – Атрибуты сущности «Пользователь»

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Имя пользователя	Уникальное имя пользователя в системе	Строковый	150		Нет	ivanov
2	Имя	Имя пользователя	Строковый	100	А-я	Да	Иван
3	Фамилия	Фамилия пользователя	Строковый	100	А-я	Да	Иванов
4	Почта	Электронная почта	Строковый	254	Email	Нет	ivan@mail.ru
5	Роль	Роль в системе	Строковый	20	User, Admin	Нет	user
6	Создан	Дата создания	Дата			Нет	2025:05:17
7	Активен	Статус активности	Логический		True, False	Нет	True

В таблице 5 представлены атрибуты сущности «API-токен», которая обеспечивает механизм программного доступа к серверу с учётом разрешений. Каждый токен связан с одним пользователем и включает набор флагов доступа, дату создания и срок действия.

Таблица 5 – Атрибуты сущности «API-токен»

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Токен	Уникальный ключ доступа	Строковый	64	SHA256 строка	Нет	a1b2c3...
2	Название токена	Подпись для различения	Строковый	100		Нет	my-phone-token
3	Доступ к моделям	Разрешение на просмотр моделей	Логический		True, False	Нет	True
4	Доступ к генерации	Разрешение на генерацию текста	Логический		True, False	Нет	True
5	Создан	Дата создания	Дата			Нет	2025 – 05 – 15
6	Окончится	Дата окончания действия токена	Дата			Да	2025:06:15
7	Последнее использование	Последняя активность по токену	Дата			Да	2025:05:18

В таблице 6 описаны атрибуты сущности «Чат». Эта сущность используется для хранения информации о диалогах между пользователем и языковой моделью.

Таблица 6 – Атрибуты сущности «Чат»

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Название чата	Название чата	Строковый	255		Нет	Личный чат
2	Создан	Дата создания чата	Дата			Нет	2025:05:17
3	Изменен	Дата последнего изменения	Дата			Нет	2025:05:18

В таблице 7 приводятся атрибуты сущности «Сообщение». Сущность хранит сообщения, отправленные в рамках чата, с указанием источника, текста и времени отправки.

Таблица 7 – Атрибуты сущности «Сообщение»

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Источник	Кто отправил сообщение	Строковый	10	user, model, system	Нет	user
2	Текст сообщения	Содержимое сообщения	Строковый	255	255	Нет	Привет
3	Создано	Дата создания сообщения	Дата			Нет	2025:05:17

В таблице 8 приведена структура сущности «Генерация», которая представляет собой задание на генерацию текста. Включает запрос пользователя, системный запрос, параметры генерации, результат и статус выполнения.

Таблица 8 – Атрибуты сущности «Генерация»

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Запрос пользователя	Входное сообщение	Строковый	255		Нет	Расскажи шутку
2	Системный запрос	Системное сообщение	Строковый	255		Да	Ты помощник
3	Настройки	Параметры генерации	JSON			Да	{«temp»: 0.8}
4	Результат	Сгенерированный моделью текст	Строковый	255		Да	Конечно! …
5	Статус	Состояние задачи	Строковый	20	pending, processing, done	Нет	processing
6	Ошибка	Сообщение об ошибке, если возникла	Строковый	255	А-я	Да	Не удалось отправить
7	Создан	Дата создания записи	Дата			Нет	2025:05:17
8	Изменен	Последнее обновление	Дата			Нет	2025:05:18

В таблице 9 описываются атрибуты сущности «Языковая модель». Эта сущность предназначена для хранения информации о моделях, доступных на сервере.

Таблица 9 – Атрибуты сущности «Языковая модель

№	Наименование атрибута	Описание атрибута	Тип данных атрибута	Размер значения	Область допустимых значений	Возможность значения Null	Пример
1	Название модели	Уникальное имя модели	Строковый	255		Нет	qwen3:0.6B
2	Создан	Дата загрузки модели	Дата			Нет	2025:05:10
3	Изменен	Последнее обновление	Дата			Нет	2025:05:17
4	Статус	Состояние модели	Строковый	20	ready, pulling, failed	Нет	Ready
5	Размер	Объем модели в байтах	Числовой		больше 0	Нет	437289200
6	Дополнительная информация	описание параметров модели	JSON		quant, arch, context_size	Да	{«quant»: «Q4_K_M»}

Описание связей между сущностями представлено в таблице 10.

Таблица 10 – Спецификации связей между сущностями

№	Тип сущности	Тип сущности	Семантика связи	Мощность	Обязательность
1	Пользователь	API-токен	Один пользователь может иметь несколько токенов	1:M	Необязательная, т. к. пользователь может не пользоваться механизмом токенов
2	Пользователь	Чат	Один пользователь может создать множество чатов	1:M	Обязательная
3	Чат	Сообщение	Один чат содержит множество сообщений	1:M	Обязательная
4	Чат	Генерация	Генерации могут быть привязаны к конкретному чату	1:M	Необязательная, т. к. генерации не зависят от чата
5	Пользователь	Генерация	Один пользователь может отправлять много генераций	1:M	Необязательная, т. к. генерации не зависят от пользователя
6	Генерация	Языковая модель	Каждая генерация выполняется одной моделью	M:1	Обязательная, если не анонимная генерация

Полученные результаты являются исходными данными для моделирования базы данных и разработки приложения информационной системы.

2.2.2 Логическая модель базы данных

Логическая модель базы данных представляет собой абстрактное описание структуры информации, хранимой в системе, ориентированное на реализацию в конкретной системе управления базами данных (СУБД).

В отличие от концептуальной модели, логическая модель уже ориентирована на реализацию в конкретной реляционной СУБД, такой как PostgreSQL, MySQL или MS SQL Server. Она может быть выражена в виде таблиц, SQL-описания схемы, ER-диаграмм, UML-диаграмм классов, а также в виде документации, описывающей структуру и правила взаимодействия с данными [8].

Логическая модель позволяет: формализовать структуру хранения данных; реализовать целостность на уровне БД; обеспечить оптимизацию запросов; автоматизировать взаимодействие между слоями приложения Данная модель представлена на рисунке 15.

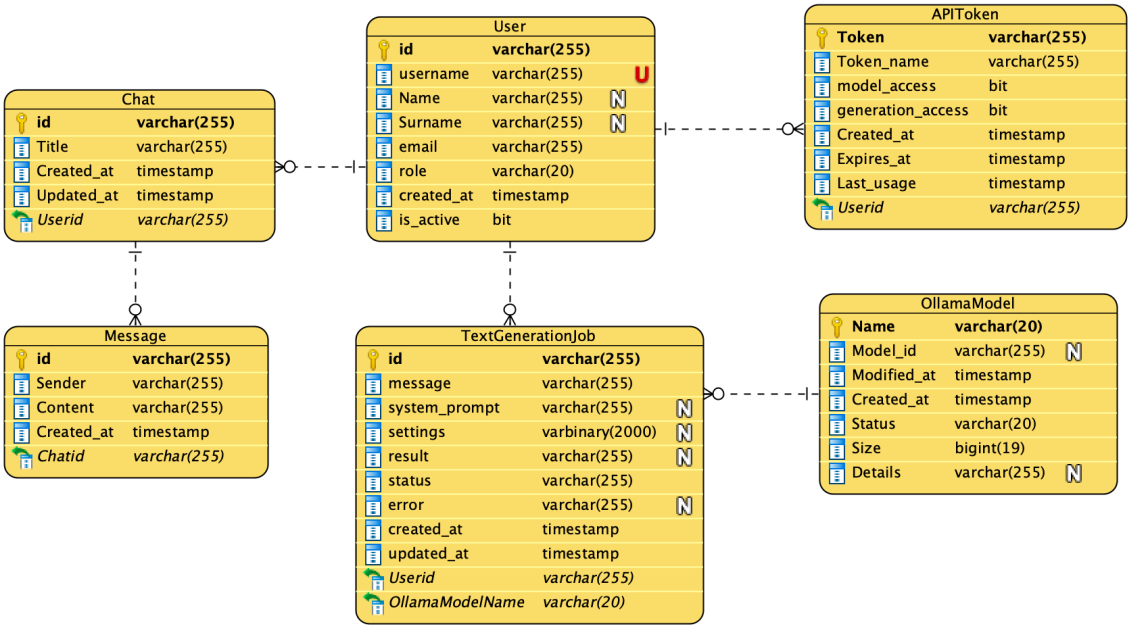

Рисунок 15 – Графическое представление логической модели

Таким образом, построенная концептуальная и логическая модель базы данных обеспечивает основу для технической реализации взаимодействия с хранимыми данными, а также позволяет наглядно отразить структуру предметной области и связи между сущностями. Созданные спецификации и диаграммы способствуют формализации архитектуры хранения, обеспечивая устойчивость, расширяемость и целостность информационного слоя приложения.

2.3 Проектирование пользовательского интерфейса

Согласно выделенным функциональным требованиям, был разработан пользовательский интерфейс приложения «Личный нейро-консультант», обеспечивающий доступ к основным функциям системы: выбор модели, генерация текстовых ответов, управление параметрами сессии и взаимодействие с языковой моделью.

Главное окно и боковая панель приложения представлены на рисунке 16 и содержат элементы управления, позволяющие пользователю взаимодействовать с языковой моделью в рамках диалогового интерфейса и навигации между сессиями. Состав графических элементов:

1 – подменю управления чатами;

2 – идентификатор активной модели;

3 – сообщение пользователя;

4 – блок объяснения языковой модели;

5 – ответ языковой модели;

6 – поле ввода сообщения;

6 – переключатель темы интерфейса;

7 – кнопка отправки сообщения;

8 – кнопка перехода на страницу настроек;

9 – кнопка для создания нового чата;

10 – список существующих чатов;

11 – кнопка обновления списка чатов.

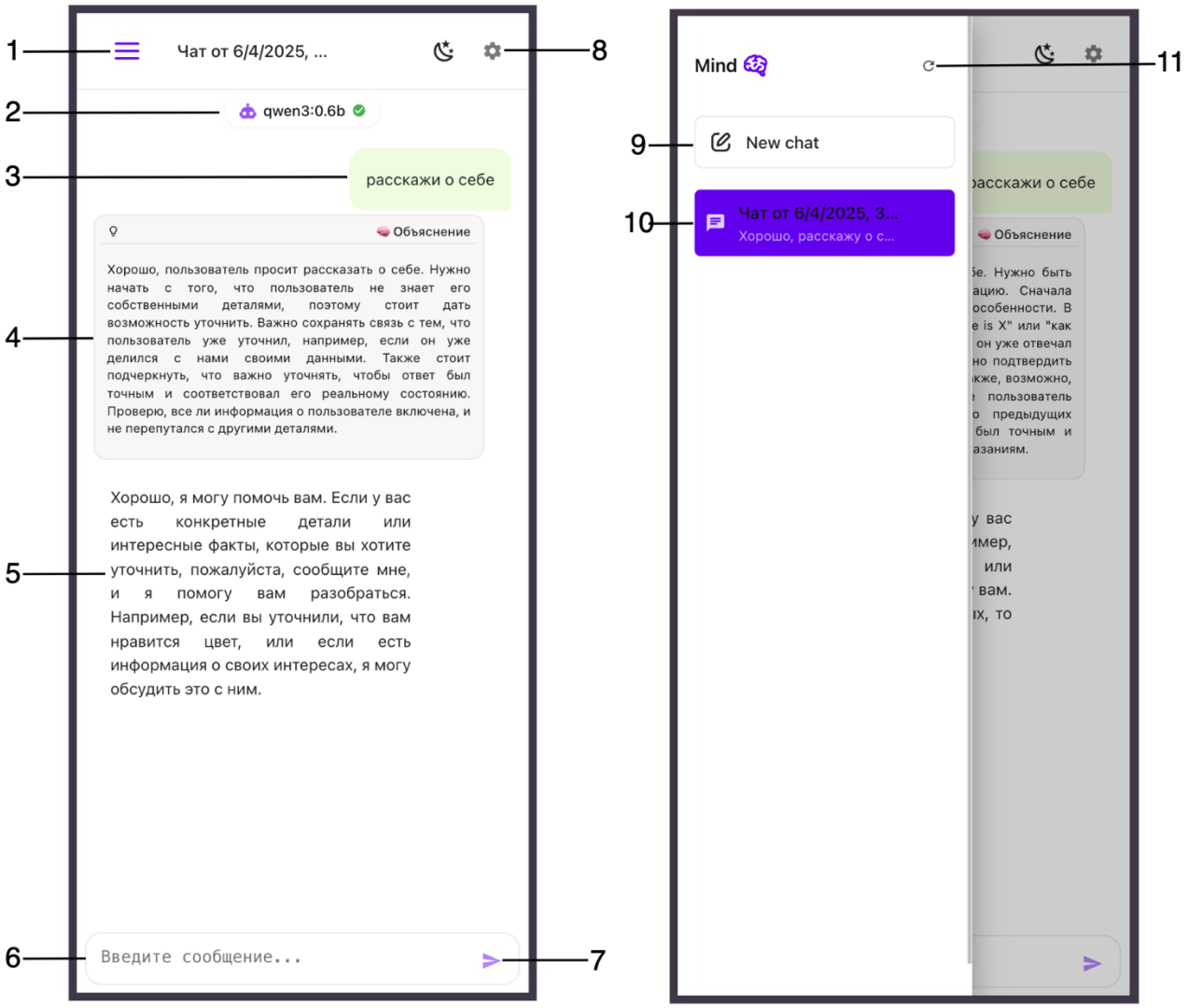

Рисунок 16 – Графический интерфейс главной страницы и бокового окна чата

Экран настроек служит важным элементом пользовательского интерфейса, предоставляя доступ к основным параметрам взаимодействия с языковой моделью, управления активными сессиями и обеспечения безопасности. Все элементы расположены в вертикальном порядке, что особенно удобно при работе с мобильными устройствами, обеспечивая простоту восприятия и комфорт при навигации. Через данный экран пользователь может изменять параметры работы модели как для личного использования, так и при работе с API, без необходимости перехода в административную панель или вмешательства в серверную часть.

Визуальное представление интерфейса показано на рисунке 17.

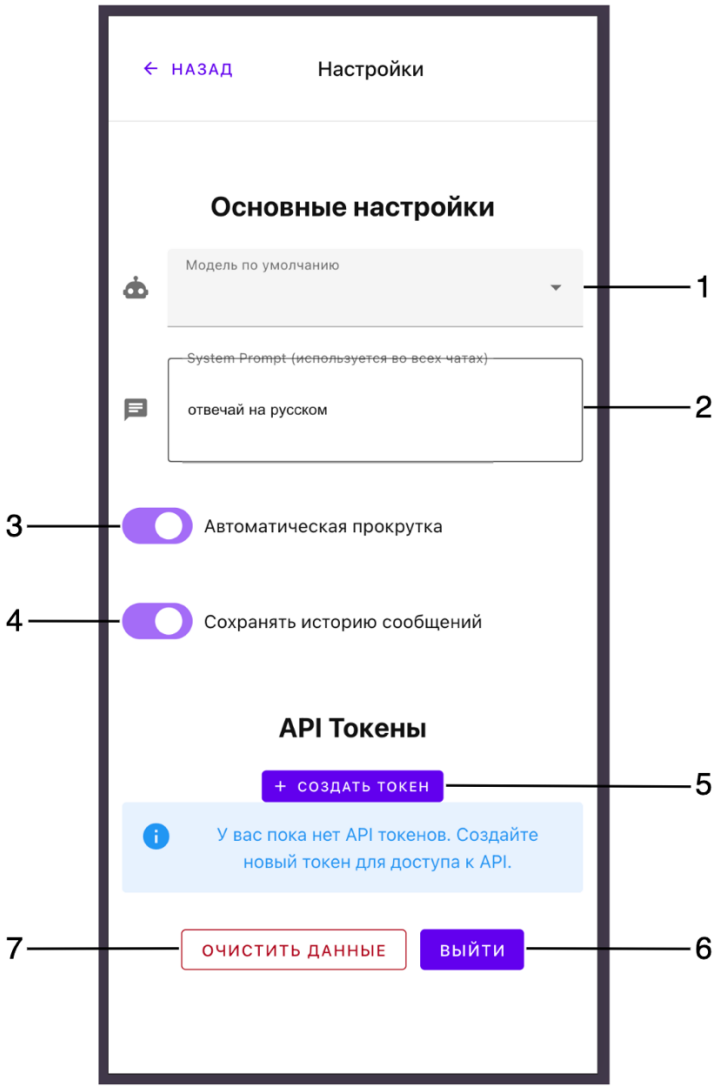

Рисунок 17 – Графический интерфейс раздела настроек

Функциональные элементы:

1 – выбор модели по умолчанию;

2 – системная инструкция для всех моделей;

3 – переключатель автоматической прокрутки внутри чата;

4 – переключатель сохранения истории сообщений;

5 – кнопка создания API-токена;

6 – кнопка для выхода из учетной записи;

7 – кнопка удаления локальных данных.

Далее рассмотрим экран входа, позволяющий пользователю авторизоваться через логин или токен, его интерфейс отражен на рисунке 18.

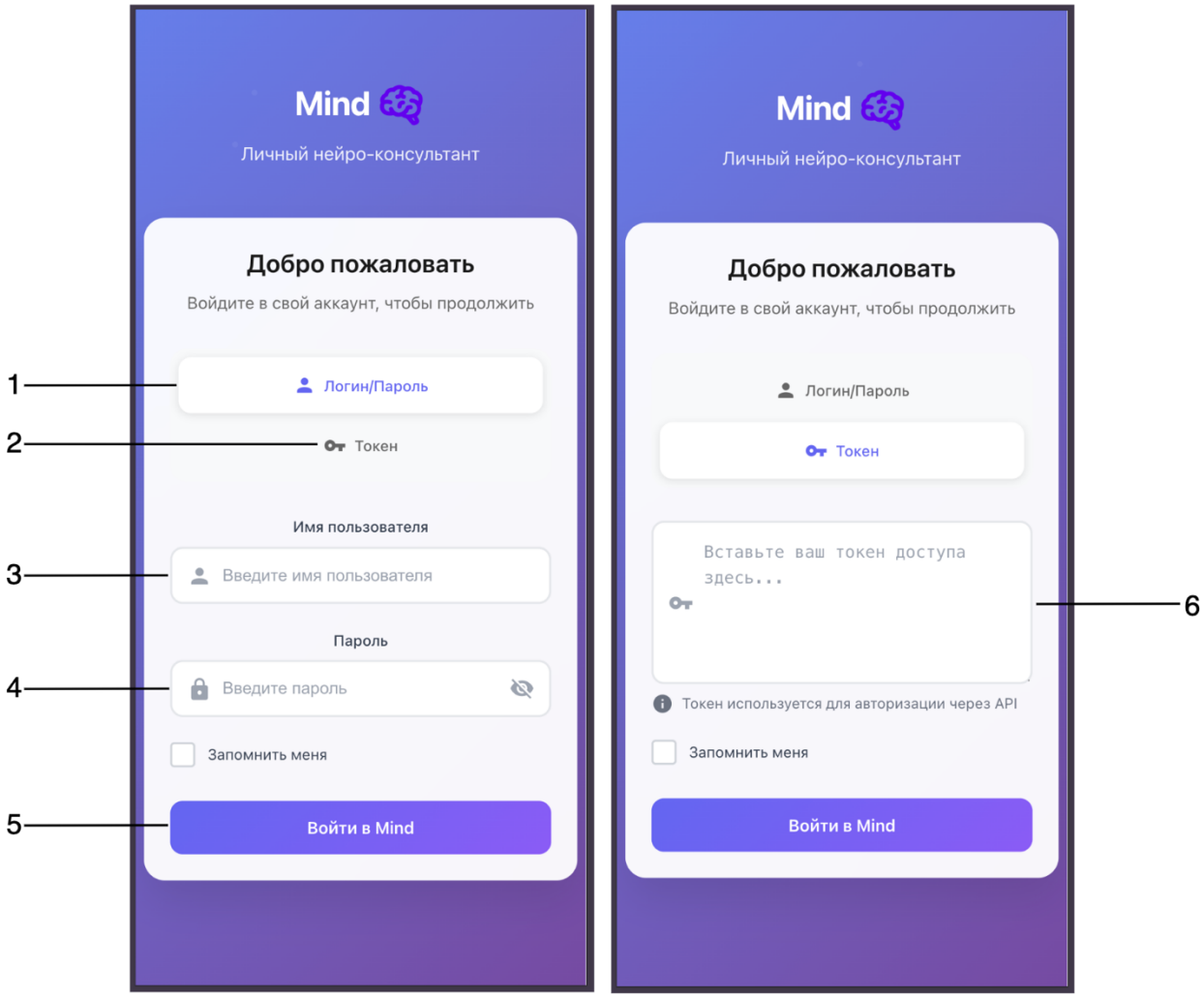

Рисунок 18 – Графический интерфейс страницы авторизации

Функциональные элементы:

1 – кнопка выбора варианта входа через Логин/Пароль;

2 – кнопка выбора варианта входа через токен;

3 – поле для ввода имени пользователя;

4 – поле для ввода пароля;

5 – кнопка авторизации;

6 – поле для ввода токена;

В рамках этого этапа были разработаны графические интерфейсы для всех ключевых сценариев использования приложения: ведение чатов, управление сессиями, настройка модели и API-доступа.

3 Реализация программных средств «Личный нейро-консультант»

3.1 Реализация базы данных

Реализация базы данных является фундаментом работы всей системы, обеспечивая надежное хранение информации о пользователях, чатах, сообщениях, API-токенах, а также задачах генерации текста. Схема данных базируется на концептуальной модели, описанной в главе 2.2, и реализуется посредством механизма объектно-реляционного отображения (Objet Relationship Mapping, ORM), встроенного в фреймворк Django.

Физическая модель данных определяет способ хранения информации в конкретной системе управления базами данных (СУБД). Она включает детальное описание структуры таблиц, индексов, ограничений и связей между объектами, а также конкретные параметры хранения, влияющие на производительность и масштабируемость системы [9].

В рамках ORM Django каждая таблица БД представляется в виде Python-класса, что упрощает разработку и делает возможным автоматическое формирование миграций – промежуточных сокриптов, отвечающих за создание и изменение структуры базы данных.

После определения моделей предметной области генерируются миграции, автоматически формирующие SQL-выражения, необходимые для создания таблиц и связей. Сами SQL-структуры хранятся в специальных миграционных файлах.

Процесс миграции включает в себя:

– создание таблиц по каждой из моделей;
– добавление внешних ключей;
– установление уникальных и первичных ограничений;
– формирование необходимых индексов.

Такая автоматизация обеспечивает контроль версий структуры базы данных и возможность безопасного развёртывания на различных средах.

В Django ORM каждая сущность представлена как отдельный класс, унаследованный от базового Model. Связи между сущностями описываются через поля ForeignKey, OneToOneField и ManyToManyField, что упрощает доступ к вложенным объектам и реализует навигационные свойства в программном коде. Листинг 1 демонстрирует типичную реализацию модели сообщения в рамках Django ORM

Листинг 1 – Пример ORM класса Сообщений

```
class Message(models.Model):
    chat = models.ForeignKey(Chat, on_delete=models.CASCADE)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

Такой подход не требует ручного написания SQL-запросов и обеспечивает высокоуровневую абстракцию взаимодействия с базой данных. Физическая модель базы данных изображена на рисунке 19.

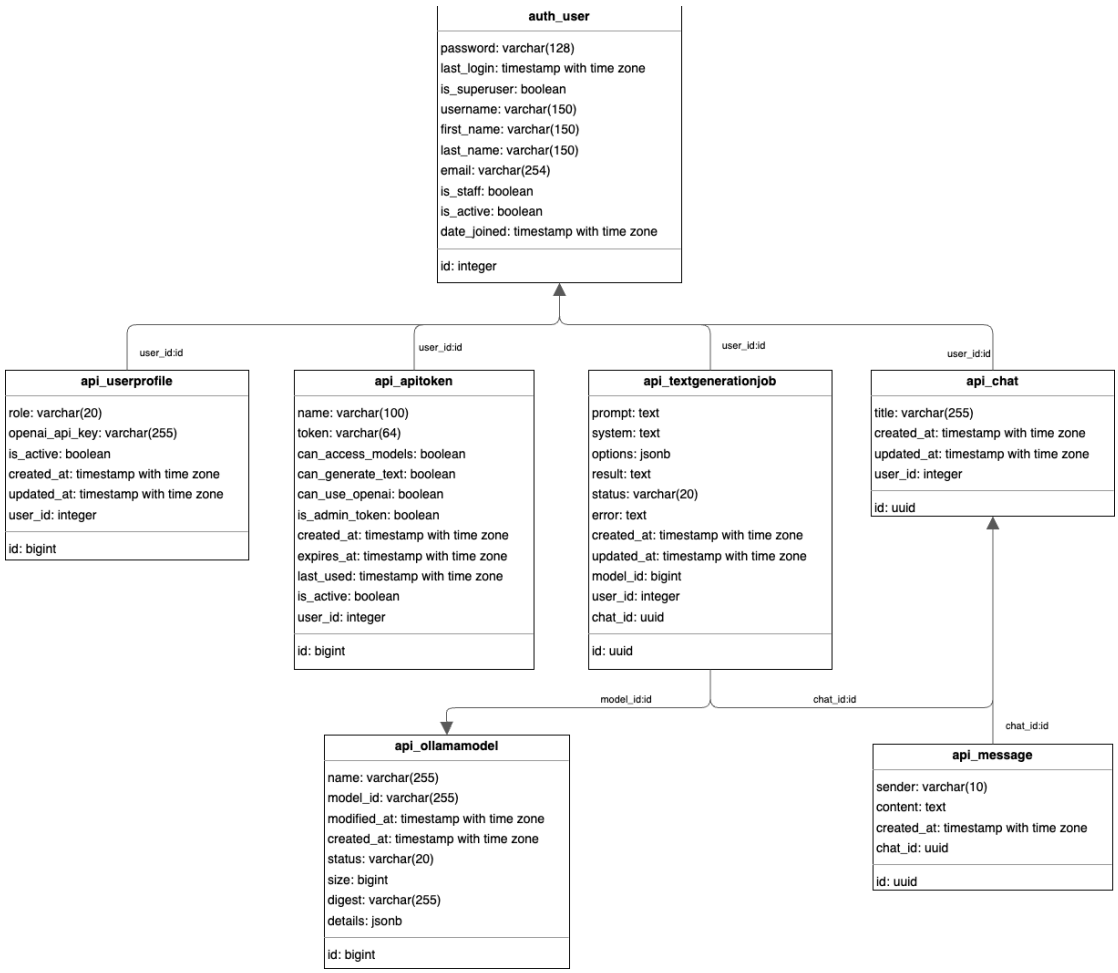

Рисунок 19 – Физическая модель базы данных

3.2 Реализация серверной части приложения

Архитектура серверной части разрабатываемого приложения базируется на принципах многоуровневой (слоистой) организации программных систем.

В контексте разрабатываемой системы была реализована модифицированная архитектура, адаптированная под специфику задач обработки естественного языка и интеграции с внешними сервисами машинного обучения. Фактическая архитектура серверной части приложения отражена на рисунке 20.

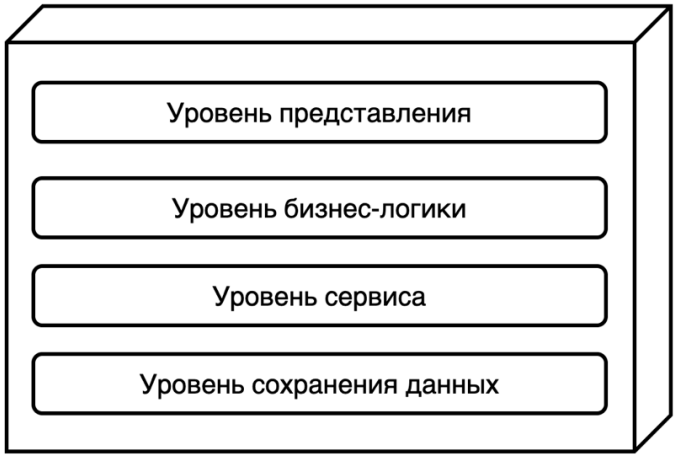

Рисунок 20 – Архитектура разрабатываемой системы

Более подробно рассмотрим каждый слой архитектуры разрабатываемой системы, их описание представлено в таблице 11.

Таблица 11 – Слои архитектуры серверной части

Слой	Описание
Представление	Обрабатывает входящие HTTP-запросы и WebSocket-соединения, формируя конечный API-интерфейс. Для REST-интерфейса используются классы представлений, а для WebSocket – потребители на базе Django Channels.
Бизнес-логика	Реализует прикладную логику взаимодействия между пользователем и языковой моделью, управляет сессиями, задачами генерации и синхронизацией данных.
Сервисный	Отвечает за обработку внешних вызовов, в частности, обращение к API Ollama-сервера. Абстракция OllamaService инкапсулирует операции загрузки, удаления, синхронизации моделей и генерации текста.
Хранения данных	Реализуется через ORM-модели Django, отражающие сущности предметной области: пользователи, чаты, сообщения, задачи генерации и модели. Взаимодействие происходит через высокоуровневые API моделей.

Особое значение в архитектуре серверной части отводится сервисному уровню, обеспечивающему взаимодействие с внешним API Ollama. Ключевую роль здесь играет класс OllamaService, в котором сосредоточена логика интеграции с системой управления языковыми моделями и обработки текстов на основе машинного обучения.

Реализуемый компонент охватывает весь цикл операций, необходимых для администрирования моделей: от получения их перечня и загрузки из централизованного реестра до удаления неактуальных экземпляров и синхронизации состояния между локальной средой и удалённым сервером.

Функциональные возможности по генерации текстов включают формирование HTTP-запросов с передачей параметров диалога и конфигурации генерации, а также получение результатов в различных режимах – как потоковом, так и блочном.

Для обеспечения устойчивости функционирования внедрены механизмы контроля исключений и сбоев, в том числе автоматическое восстановление после потери соединения, обработка таймаутов и корректная реакция на ошибки валидации.

Эффективность повышается за счёт встроенного механизма кэширования часто повторяющихся запросов, что снижает нагрузку на систему и ускоряет обработку.

В части безопасности предусмотрены механизмы аутентификации и проверка достоверности всех входных и выходных данных, что обеспечивает доверенность взаимодействия с внешней платформой.

Такая архитектура обеспечивает высокую отзывчивость пользовательского интерфейса и эффективное использование вычислительных ресурсов, что особенно важно при работе с ресурсоёмкими операциями генерации естественного языка [10].

Реализованное решение демонстрирует эффективность применения современных архитектурных паттернов для создания масштабируемых систем искусственного интеллекта, интегрированных с веб-технологиями.

3.3 Реализация клиентской части приложения

React и Vue – это современные JavaScript-фреймворки для построения пользовательских интерфейсов. Оба основаны на компонентной архитектуре, что обеспечивает переиспользуемость кода и простоту масштабирования.

Прогрессивные веб-приложения – это новая захватывающая форма современных веб-приложений. В этих приложениях используются новейшие веб-возможности для обеспечения опыта, который сочетает в себе уникальные особенности нативных мобильных приложений и преимущества веба [11].

Для отображения сообщений с поддержкой стиля Markdown используется библиотека marked. Также применяется библиотека KaTeX для вывода математических формул, программный код представлен в листинге 2.

Листинг 2 – Отображение стиля Markdown и математических формул

```
function renderMath(text: string): string {
  return text
    .replace(/\$\$([\s\S]+?)\$\$/g, (_: string, expr: string) =>
      katex.renderToString(expr, { displayMode: true,
throwOnError: false })
    )
    .replace(/\$(.+?)\$/g, (_: string, expr: string) =>
      katex.renderToString(expr, { displayMode: false,
throwOnError: false })
    )
}
const renderedContent = computed(() => {
  let content = props.message ?? '';
  if (isThink.value) {
    content = content.replace(/<think>[\s\S]*?<\/think>/, '');
  }
  return content.trim() ? marked.parse(renderMath(content.trim()))
: '';
});
```

Этот код используется, чтобы в окне чата пользователь мог видеть сообщения с красиво отформатированными текстами, ссылками, списками и математическими формулами, включая теоремы, уравнения или вычисления, оформленные согласно стандарту LaTeX.

Поддержка тёмной/светлой темы реализована через Pinia Store с сохранением состояния в localStorage.

Для обеспечения взаимодействия с сервером в реальном времени реализован класс WebSocket-клиента.

Протокол WebSocket разработан для замены существующих технологии двунаправленной связи, использующей HTTP в качестве транспортного транспортного уровня, чтобы воспользоваться преимуществами существующей инфраструктуры [12].

Для демонстрации работы с WebSocket в листинге 3 был приведен метод класса webSocketChat, отвечающий за установку двунаправленного соединения с сервером.

Листинг 3 – Метод установки WebSocket соединения

```
  connect(): void {
    const protocol = window.location.protocol === 'https:' ?
'wss:' : 'ws:';
    const host = window.location.hostname || 'localhost';
    const port = this.portAttempts[0];
    let wsUrl =
`${protocol}//${host}:${port}/ws/chat/${this.chatId}/`;
    if (this.token) wsUrl += `?token=${this.token}`;

    this.ws = new WebSocket(wsUrl);
    this.ws.onopen = () => {
      console.log('WebSocket connected');
      this.processQueue();
    };
    this.ws.onmessage = (event) => {
      const data = JSON.parse(event.data);
      this.onMessage(data);
    };
};
```

Этот метод обеспечивает инициализацию канала реального времени между клиентом и сервером, что позволяет мгновенно получать ответы модели, отправленные с сервера в виде потоковыхфрагментов. Такой подход критически важен для обеспечения низкой задержки при взаимодействии с языковыми моделями и построения интерактивного интерфейса.

Более подробная логика работы с WebSocket в режиме реального времени отражена в диаграмме последовательности для бизнес-процесса «Генерация сообщения» на рисунке 20.

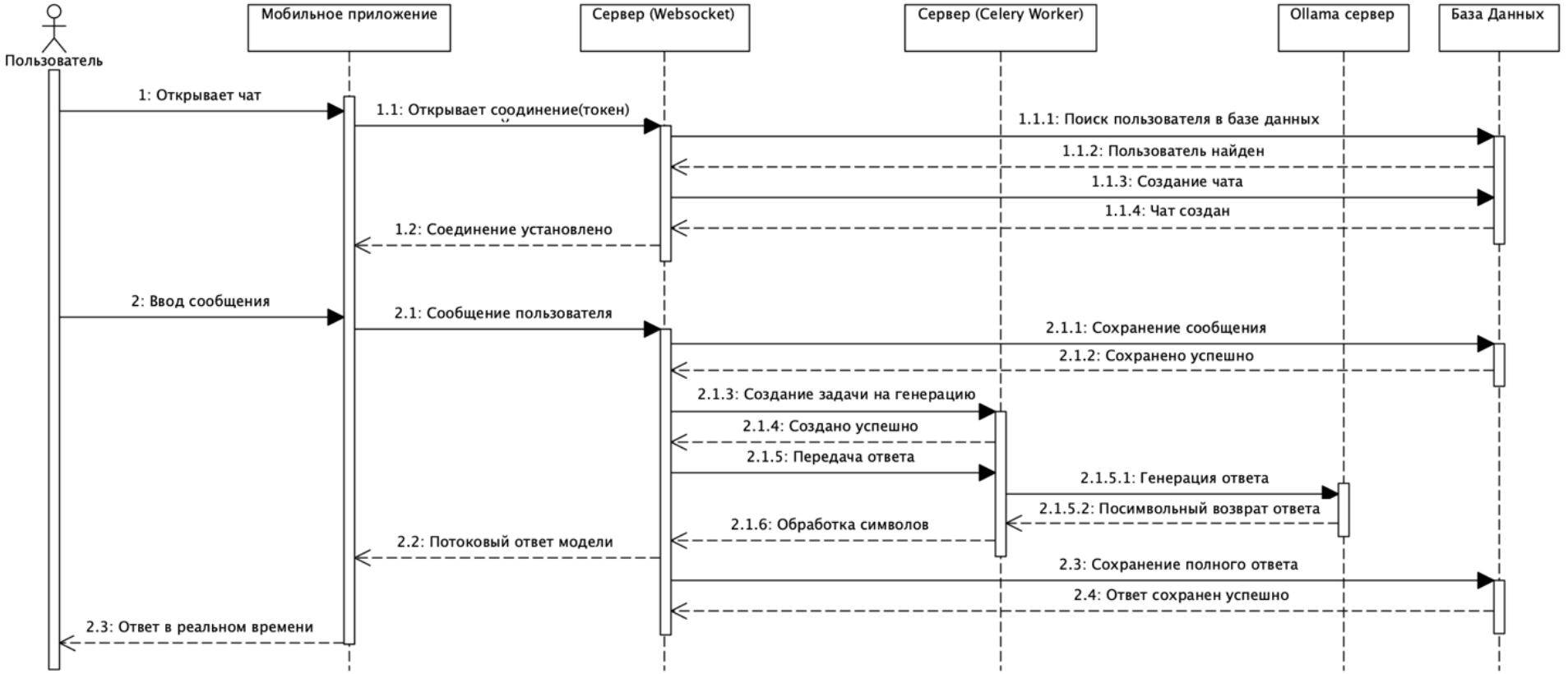

Рисунок 20 – Диаграмма последовательности при генерации сообщения через WebSocket

Диаграмма последовательности отражает жизненный цикл общения пользователя с моделью. После установления соединения клиент инициирует аутентификацию, а затем отправляет запрос генерации текста. Серверная часть создает задачу и ставит в очередь, для асинхронного ответа, где обращается к Ollama серверу с моделями. Полученные данные в виде символов возвращаются клиенту через WebSocket, формируя потоковый отклик в интерфейсе пользователя. Все сообщения и результаты сохраняются в базу данных для последующего просмотра истории. Такой подход обеспечивает минимальную задержку, гибкость и масштабируемость при работе с нейросетевыми моделями.

Согласно спецификации UML, диаграмма последовательности – основной вид диаграмм взаимодействия, ориентированный на отображение упорядоченных событий и обмена сообщениями между участниками во временном порядке. На ней четко видны «линии жизни» объектов, окно их активности и отличие между синхронными и асинхронными вызовами [13].

Разработка пользовательского интерфейса продемонстрировала эффективность применения современных веб-технологий. Использование фреймворка Vue в связке с концепциями PWA обеспечило отзывчивость интерфейса, кроссплатформенность и улучшенное пользовательское взаимодействие. Компонентная модель позволила обеспечить изоляцию логики и упрощённое тестирование, а подключение WebSocket – реализовать обмен данными в реальном времени. Таким образом, реализованная архитектура клиентской части отвечает современным требованиям к масштабируемым и интерактивным веб-приложениям.

4 Контроль качества программного обеспечения

4.1 Функциональное тестирование

После завершения разработки программного обеспечения было проведено функциональное тестирование системы интеллектуального чат-бота.

Функциональное тестирование – вид тестирования, направленный на проверку корректности работы функциональности приложения (корректность реализации функциональных требований) [14].

Тестирование программного обеспечения было проведено с использованием подхода тестирования «белым ящиком».

Стратегия белого ящика, или стратегия тестирования, управляемого логикой программы, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы [15].

Преимущества метода белого ящика:

– показывает скрытые проблемы и упрощает их диагностику;

– допускает достаточно простую автоматизацию тест-кейсов и их пополнение на самых ранних стадиях развития проекта;

– обладает развитой системой метрик;

– стимулирует разработчиков к написанию качественного кода.

В качестве объекта тестирования была выбрана критически важная функция get_user_from_token из модуля WebSocket-аутентификации, отвечающая за получение пользователя по токену доступа. Данная функция является ключевым компонентом системы безопасности, обеспечивающим аутентификацию пользователей через токены и контролирующим доступ к ресурсам системы. В процессе тестирования были проверены все возможные сценарии работы функции, включая различные типы токенов, проверку истечения срока действия и обработку исключительных ситуаций. Код функции представлен в листинге 4.

Листинг 4 – Функция получения пользователя по токену

```
def get_user_from_token(token_key):  # a
    try:
        token_obj = Token.objects.get(key=token_key)  # b
        return token_obj.user  # c
    except Token.DoesNotExist:
        try:
            api_token = APIToken.objects.get(token=token_key,
is_active=True)  # d

            if api_token.expires_at and api_token.expires_at <
timezone.now():  # f
                return None  # g

            api_token.last_used = timezone.now()
            api_token.save(update_fields=['last_used'])  # h

            return api_token.user  # i
        except APIToken.DoesNotExist:
            return None  # e
```

Схема программы для функции «get_user_from_token» представлена на рисунке 21.

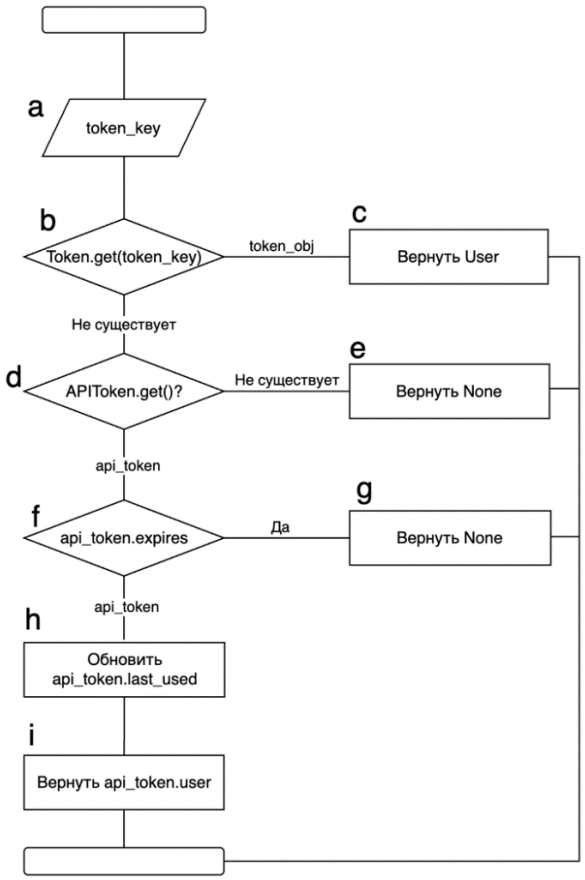

Рисунок 21 – Схема программы функции «get_user_from_token»

Критерием покрытия является выполнение каждого оператора программы по крайней мере один раз. К сожалению, это слабый критерий, так как выполнение каждого оператора по крайней мере один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика [15].

В таблице 12 отражено покрытие каждого оператора для схемы функции «get_user_from_token».

Таблица 12 – Покрытие операторов

№	Операторы	Входные данные	Выходные данные
1	a, b, c	token_key = «valid_drf_token123»	User объект
2	a, f, g	token_key = «expired_api_token456» (истёкший)	None
3	a, h, i	token_key = «valid_api_token789»	User объект с обновлённым last_used
4	a, d, e	token_key = «invalid_token_xyz»	None

Более сильный критерий покрытия логики программы известен как покрытие решений, или покрытие переходов. Согласно данному критерию, должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение истина и ложь по крайней мере один раз. Иными словами, каждое направление перехода должно быть реализовано по крайней мере один раз [15].

Покрытие решений представлено в таблице 13.

Таблица 13 – Покрытие решений

№	Решения	Входные данные	Выходные данные
1	a, b-да, c	token_key = «abc123…345» (существующий токен)	User объект
2	a, b-нет, d-нет, f-да, g	token_key = «expired789», expires_at = «2023:01:01 00:00:00»	None
3	a, b-нет, d-нет, f-нет, h, i	token_key = «active456», expires_at = «2025:12:31 23:59:59»	User объект
4	a, b-нет, d-да, e	token_key = «nonexistent123»	None

Комбинаторное покрытие условий представлено в таблице 14.

Таблица 14 – Комбинаторное покрытие решений

№	Комбинация условий	Маршруты	Входные данные	Выходные данные
1	Токен существует	a, b, c	token_key = «drf_token_1234»	User объект
2	Токен не существует, API токен существует, не истёк	a, b, d, f, h, i	token_key = «api_token_act», expires_at = «2025:12:31»	User объект с обновлённым last_used
3	Токен не существует, API токен существует, истёк	a, b, d, f, g	token_key = «api_token_exp», expires_at = «2023:01:01»	None
4	Токен не существует, API токен не существует	a, b, d, e	token_key = «fake_token_xyz»	None

Программа успешно прошла функциональное тестирование методом «белого ящика», показав правильность работы всех программных компонентов и полное соответствие функциональным требованиям. Все возможные пути выполнения программы были протестированы, что гарантирует надёжность работы системы аутентификации в различных сценариях использования.

4.2 Нефункциональное тестирование

Нефункциональное тестирование – вид тестирования, направленный на проверку нефункциональных особенностей приложения (корректность реализации нефункциональных требований), таких как удобство использования, совместимость, производительность, безопасность и т. д. [15].

Большинство бизнес-ориентированных тестов, поддерживаемых командой разработки, также должны быть автоматизированы. Одной из наиболее важных целей тестов является быстрая доставка информации и быстрый поиск неполадок. Они должны запускаться часто, чтобы обеспечить для команды ранний отклик в случае непреднамеренного изменения поведения. Все эти тесты должны запускаться как часть процесса непрерывной автоматической интеграции, сборки и тестирования [16].

Для обеспечения качества пользовательского опыта и проверки корректности работы интерфейса было проведено нефункциональное тестирование с использованием фреймворка Playwright, представляющий собой инструмент для автоматизации браузеров, разработанный компанией Microsoft для проведения end-to-end (E2E) тестирования веб-приложений [17].

Тестирование было организовано по функциональным областям приложения с охватом ключевых пользовательских сценариев. Структура тестовых случаев представлена в таблице 15.

Таблица 15 – Тестовые случаи для нефункционального тестирования

Категория тестов	Количество тестов	Описание
Загрузка приложения	2	Проверка корректности загрузки и времени отклика
Интерфейс чата	3	Отображение элементов, темы, адаптивность
Производительность	3	Общая производительность системы

Тестирование проводилось на четырех браузерных платформах с общим количеством 32 тестовых сценария. В таблице 16 отражены тестовые сценарии с результатом и описанием.

Таблица 16 – Результаты нефункционального тестирования

№	Тестовый сценарий	Результат во всех браузерах	Описание
1	Загрузка главной страницы	Успешно	Корректная загрузка и отображение
2	Быстрая загрузка приложения	Успешно	Время загрузки менее 4 секунд
3	Отображение элементов интерфейса	Успешно	Видимость и контент загружены
4	Функционирование темы	Успешно	Корректная работа CSS и стилей
5	Адаптивность интерфейса	Успешно	Работа на разных размерах экрана
6	Производительность	Успешно	Соответствие метрикам производительности
7	Оптимизация	Успешно	Загрузка ресурсов без ошибок
8	Отсутствие утечек памяти	Успешно	Стабильная работа без критических ошибок

Проведенное нефункциональное тестирование с использованием Playwright продемонстрировало отличную стабильность и производительность разработанного приложения.

4.3 Оценка качества программного кода

Для анализа сложности кода и оценки его тестируемости в рамках функционального тестирования методом «белого ящика», были выбраны две

классические метрики: метрики Холстеда и цикломатическая сложность Мак-Кейба. Обоснование их применения связано с тем, что данные метрики позволяют количественно оценить внутреннюю структуру кода и определить наиболее критичные участки, требующие тщательной проверки.

Метрики Холстеда (Halstead) позволяют частично учесть возможность записи одной и той же функциональности разным количеством строк и операторов. Они основаны на большом числе количественных показателей, таких, как число уникальных операторов программы, число уникальных операндов программы, общее число операторов, общее число операндов, теоретическое число уникальных операторов и теоретическое число уникальных операндов

Считайте, что программа состоит из последовательности операторов и операндов. Холстед сделал следующие определения:

n_1 – число уникальных операторов программы;

n_2 – число уникальных операндов программы;

N_1 – общее число операторов в программе;

N_2 – общее число операндов в программе [18].

Через первичные характеристики выражаются словарь:

$$n=n_1+n_2 \quad (1)$$

и длина программы:

$$N=N_1+N_2. \quad (2)$$

Объем алгоритма является важной метрической характеристикой алгоритма и вычисляется по формуле:

$$V = N * \log_2 n \quad (3)$$

Трудность, или коэффициент расширяемости вычисляется по формуле:

$$D = V/V` \quad (4)$$

И усилие, количество решений необходимых для разработки программы вычисляется по формуле:

$$E = V * D \quad (5)$$

Для удобства еще раз приведем функцию «get_user_from_token» в листинг 5.

Листинг 5 – Функция получения пользователя по токену

```
def get_user_from_token(token_key):  # a
    try:
        token_obj = Token.objects.get(key=token_key)  # b
        return token_obj.user  # c
    except Token.DoesNotExist:
        try:
            api_token = APIToken.objects.get(token=token_key,
is_active=True)  # d

            if api_token.expires_at and api_token.expires_at <
timezone.now():  # f
                return None  # g

            api_token.last_used = timezone.now()
            api_token.save(update_fields=['last_used'])  # h

            return api_token.user  # i
        except APIToken.DoesNotExist:
            return None  # e
```

В таблице 17, 18 описаны все операторы и операнды функции из листинга 5.

Таблица 17 – Операторы функции «get_user_from_token»

Операнды	Число вхождений
token_key	3
Token	1
Token.objects	2
User	2
APIToken	1
APIToken.objects	1
api_token	5
is_active	1
token	1
expires_at	2
timezone.now()	2
None	2
last_used	2
Update_fields	2
n_1=15	N_1=27

Таблица 18 – Операнды функции «get_user_from_token»

Операторы	Число вхождений
def	1
try	2
except	2
return	4
=	5
.	10
()	5
if	1
<	1
and	1
n_1=10	N_1=32

В таблице 19 используя формулы 1 – 5 приведем подсчет вычисления метрик Холстеда

Таблица 19 – Результаты вычислений по метрике Холстеда

Значение	Результат
Словарь (n)	$n1 + n2 = 10 + 15 = 25$
Длина (N)	$N1 + N2 = 32 + 27 = 59$
Объем (V)	$N * \log_2 n = 59 * \log_2 25 = 273.76$
Трудность (D)	$\frac{n1}{2} * \frac{N2}{n2} = 5 * \left(\frac{27}{15}\right) = 9$
Усилие (E)	$D * V = 9 * 273.76 = 2463.84$

Метрика Холстеда показывает, что функция обладает умеренной сложностью. Усилие в 2460 указывает на необходимость внимательного проектирования тестов и потенциальную нагрузку на понимание кода.

Цикломатическая сложность, или метрика Мак-Кейба (McCabe) является наиболее известной и широко используемой при создании инструментария для оценки сложности программ на различных языках.

Подход к измерению сложности, который мы будем использовать, заключается в измерении и контроле количества путей, проходящих через программу.

Определение 1: Цикломатическая сложность V(G) графа G определяется по формуле:

$$V(G) = e - n + 2p \quad (6)$$

где n – вершины;

e – ребра;

p – компоненты связности.

Теорема 1: В сильно связном графе G цикломатическая сложность равна максимальному количеству линейно независимых цепей [19].

Для рассмотренной выше функции был построен граф потока управления, представленный на рисунке 22.

Такой граф отражает возможные маршруты выполнения внутри функции или модуля, представляя собой абстрактную модель выполнения алгоритма.

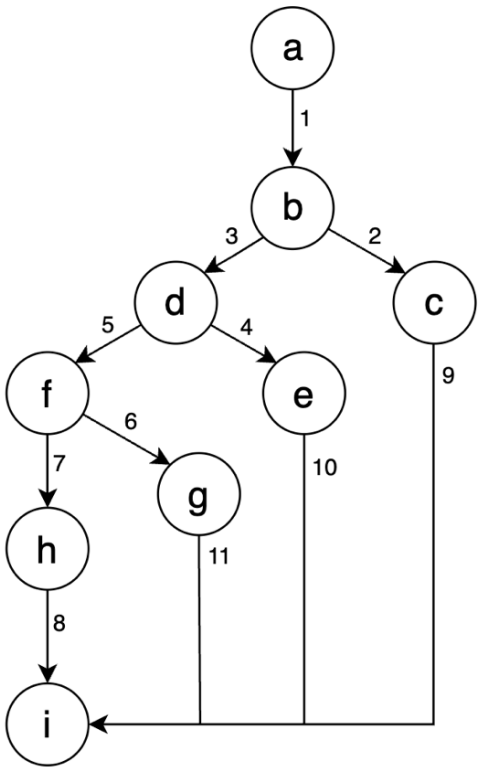

Рисунок 22 – Граф потока управления

Применяя формулу 6 для подсчета цикломатической сложности Мак-Кейба в графе потока управления, получаем следующие значения, приведенные в таблице 20:

Таблица 20 – Результат вычисления по метрике Мак-Кейба

Значение	Результат
Число ребер (e)	11
Число узлов (n)	9
Компонент связности (p)	1

Используя значения из таблицы 20 и формулу 6, получаем:

$$V(G) = 11 - 9 + 2 = 4$$

Таким образом, цикломатическая сложность данной функции составляет 4, что соответствует минимальному количеству тестов, необходимых для полного покрытия всех независимых путей исполнения.

Далее проверим подсчитанную вручную цикломатическую сложность используя библиотеку radon. Генерация изображена на рисунке 23.

```
(venv) ➜  api git:(main) ✗ radon cc listing.py -a
listing.py
    F 2:0 get_user_from_token - A

1 blocks (classes, functions, methods) analyzed.
Average complexity: A (4.0)
(venv) ➜  api git:(main) ✗
```

Рисунок 23 – Автоматический подсчет цикломатической сложности

Таким образом, применение метрик Холстеда и цикломатической сложности Мак-Кейба является обоснованным подходом в рамках функционального тестирования методом белого ящика.

Такой подход дополняет традиционные методы функционального тестирования, направленные на проверку корректности реализации требований, и повышает общую надежность и контролируемость процесса тестирования.

Заключение

В ходе выпускной квалификационной работы было разработано PWA-приложение «Личный нейро-консультант», обеспечивающее автономное и конфиденциальное взаимодействие пользователя с открытыми языковыми моделями через собственный сервер.

В приложении реализованы авторизация, выбор модели, отправка запросов, потоковый приём ответов, настройка параметров, управление историей чатов, смена темы и очистка данных. Система построена на сочетании Django и Vue.js с поддержкой REST и WebSocket, что обеспечивает высокую отзывчивость и расширяемость.

Результаты работы подтверждены проведённым функциональным тестированием и оценкой качества кода с помощью метрик Холстеда и Мак-Кейба. Полученное приложение соответствует заявленным требованиям, демонстрирует стабильность и может быть расширено для поддержки дополнительных моделей и сценариев.

Эта мысль ставит во главу угла активную позицию разработчика в этическом и технологическом пространстве. Проект «Личный нейро-консультант» отражает именно такой подход – приложению поручено не только решать задачи, но и быть инструментом для осознанного развития пользователей, без потери контроля над собственными данными и поведением.

Список использованных источников

1. Саттон, Р. С. Введение в обучение с подкреплением / Р. С. Саттон, А. Г. Барто. – М.: Бином, 2022. – 544 с.

2. Марков С. Охота на электроовец. Большая книга искусственного интеллекта. – Москва, 2024. – 568 с. (1 том).

3. Копылов Е. А., Дзюба Е. А. Личные нейроконсультанты с применением больших языковых моделей // Вестник Пензенского государственного университета. 2025. № 1. С. 66–72.

4. Протокол PKCE – Proof Key for Code Exchange by OAuth Public Clients [Электронный ресурс] // IETF Datatracker. – URL: https://datatracker.ietf.org/doc/html/rfc7636 (дата обращения: 19.05.2025)

5. Ричардс Марк, Форд Нил Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы. – СПб.: Питер, 2023. – 448 с.: ил. – (Серия «Для профессионалов»).

6. Van Vliet, H. Software Engineering: Principles and Practice / H. Van Vliet. – Wiley, 2008. – 784 с.

7. Фаулер, М. UML для начинающих / М. Фаулер. – 3-е изд. – М.: Вильямс, 2021. – 208 с.

8. Bass, L. Software Architecture in Practice / L. Bass, P. Clements, R. Kazman. – 3rd ed. – Addison-Wesley, 2012. – 624 с.

9. Elmasri, R. Fundamentals of Database Systems / R. Elmasri, S. Navathe. – 7th ed. – Pearson, 2015. – 1272 с.

10. Рассел С., Норвиг П. Искусственный интеллект: современный подход, 2-е изд..: Пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 1408 с.: ил. – Парал. тит. англ.

11. Ater T. Building Progressive Web Apps / T. Ater – O’Reilly Media, 2017 – 429 с.

12. The WebSocket protocol [Электронный ресурс] // IETF Datatracker. – URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата обращения: 25.05.2025)

13. UML 2.5.1. Specification. Unified Modeling Language. Object Management Group (OMG). – 2017.

14. С.Куликов Тестирование программного обеспечения базовый курс / EPAM Systems, 2017 – 295с.

15. Майерс Г. Искусство тестирования программ/Пер. с англ. под ред. Б. А. Позина. – М.: Финансы и статистика, 1982. – 176 с., ил.

16. Криспин, Лайза, Грегори, Джанет. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд.: Пер. с англ. – ООО «И.Д. Вильямс», 2010. – 446 с

17. Playwright // Официальный сайт [Электронный ресурс] URL: https://playwright.dev/ (дата обращения 30.05.2025)

18. Henry, S. and D.Kafura and K.Harris, «On the Relationship Among Three Software Metrics», 1981 ACM Workshop|Symposium on Measurement and Evaluation of Software Quality, March 1981

19. T.J. McCabe, «A complexity measure», IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308 – 320, December, 1976

20. Лем С. Е. Мегабитовая бомба / Станислав Ежи Лем. – М.: АСТ, 1999. – Цитата: «Мир нужно изменять…». – С. 45.

Приложение А

(обязательное)

Листинг приложения

Листинг А.1 – Файл «ollama_service.py»

```
import requests
import logging
import json
from typing import Dict, List, Optional, Any, Union
from django.conf import settings

logger = logging.getLogger(__name__)

class OllamaService:
    """Service for interacting with the Ollama API"""

    def __init__(self, base_url: str = None):
        """Initialize the Ollama service with the base URL"""
        self.base_url = base_url or settings.OLLAMA_BASE_URL
        if 'localhost' in self.base_url or '127.0.0.1' in
self.base_url:
            self.alternate_urls = [
                self.base_url.replace('localhost',
'ollama').replace('127.0.0.1', 'ollama'),
                self.base_url.replace('localhost',
'host.docker.internal').replace('127.0.0.1',
'host.docker.internal'),
                'http://ollama:11434',
                settings.OLLAMA_BASE_URL
            ]
        else:
            self.alternate_urls = [self.base_url]

    def _make_request(self, method, endpoint, **kwargs):
        """
        Make a request to the Ollama API with fallback URLs

        Args:
            method: HTTP method (get, post, delete)
            endpoint: API endpoint
            **kwargs: Additional arguments for requests

        Returns:
            Response object
        """
        last_error = None
        is_streaming = kwargs.get('stream', False)

        for url in self.alternate_urls:
            try:
                full_url = f"{url}{endpoint}"
                logger.info(f"Attempting to connect to Ollama at:
{full_url}")

                if method.lower() == 'get':
```

```
                    response = requests.get(full_url, **kwargs)
                elif method.lower() == 'post':
                    response = requests.post(full_url, **kwargs)
                elif method.lower() == 'delete':
                    response = requests.delete(full_url, **kwargs)
                else:
                    raise ValueError(f"Unsupported HTTP method:
{method}")

                response.raise_for_status()
                self.base_url = url

                if is_streaming:
                    return response
                else:
                    return response

            except requests.RequestException as e:
                last_error = e
                logger.warning(f"Failed to connect to Ollama at
{url}: {e}")
                continue

        if last_error:
            raise last_error


    def pull_model(self, model_name: str) -> Dict[str, Any]:
        """
        Pull a model from Ollama

        Args:
            model_name: Name of the model to pull

        Returns:
            Response data
        """
        try:
            response = self._make_request(
                'post',
                '/api/pull',
                json={"name": model_name}
            )

            try:
                return response.json()
            except json.JSONDecodeError:
                result = {}

                lines = response.text.strip().split('\n')
                for line in lines:
                    if not line:
```

```
                        continue
                    try:
                        data = json.loads(line)
                        result.update(data)
                    except json.JSONDecodeError as e:
                        logger.warning(f"Failed to parse JSON
line: {line[:100]}... Error: {e}")

                if result:
                    return result
                else:
                    raise ValueError(f"Could not parse response
from Ollama: {response.text[:200]}...")

        except requests.RequestException as e:
            logger.error(f"Error pulling model {model_name} from
Ollama: {e}")
            raise

    def delete_model(self, model_name: str) -> Dict[str, Any]:
        """
        Delete a model from Ollama

        Args:
            model_name: Name of the model to delete

        Returns:
            Response data
        """
        try:
            response = self._make_request(
                'delete',
                '/api/delete',
                json={"name": model_name}
            )
            return response.json()
        except requests.RequestException as e:
            logger.error(f"Error deleting model {model_name} from
Ollama: {e}")
            raise

    def generate_text(
        self,
        model_name: str,
        prompt: str,
        system: Optional[str] = None,
        template: Optional[str] = None,
        context: Optional[List[int]] = None,
        options: Optional[Dict[str, Any]] = None,
        stream: bool = False
    ) -> Dict[str, Any]:
        """
```

```
        Generate text using a model

        Args:
            model_name: Name of the model to use
            prompt: The prompt to generate from
            system: Optional system prompt
            template: Optional template to use
            context: Optional context for the model
            options: Optional parameters for generation
            stream: Whether to stream the response

        Returns:
            Response with generated text
        """
        payload = {
            "model": model_name,
            "prompt": prompt,
            "stream": stream
        }

        if system:
            payload["system"] = system
            logger.info(f"Using system prompt: {system}")
        if template:
            payload["template"] = template
        if context:
            payload["context"] = context
        if options:
            if 'history' in options and options['history']:
                history_count = len(options['history'])
                logger.info(f"Using conversation history with
{history_count} messages for model {model_name}")
                if history_count > 0:
                    if history_count <= 4:
                        logger.info(f"Full history:
{json.dumps(options['history'])}")
                    else:
                        first_msgs = options['history'][:2]
                        last_msgs = options['history'][-2:]
                        logger.info(f"First messages:
{json.dumps(first_msgs)}")
                        logger.info(f"Last messages:
{json.dumps(last_msgs)}")

            payload["options"] = options

        try:
            if stream:
                return self._stream_generate(payload)
            else:
                logger.info(f"Sending non-streaming request to
Ollama API: {json.dumps(payload)}")
```

```
                response = self._make_request(
                    'post',
                    '/api/generate',
                    json=payload
                )

                try:
                    return response.json()
                except json.JSONDecodeError:
                    lines = response.text.strip().split("\n")
                    for line in reversed(lines):
                        try:
                            return json.loads(line)
                        except json.JSONDecodeError:
                            continue

                    raise ValueError(f"Could not parse response
from Ollama: {response.text}")

        except requests.RequestException as e:
            logger.error(f"Error generating text with model
{model_name}: {e}")
            raise

    def _stream_generate(self, payload):
        """
        Stream generated text from Ollama

        Args:
            payload: Request payload

        Returns:
            Generator yielding response chunks
        """
        try:
            with self._make_request(
                'post',
                '/api/generate',
                json=payload,
                stream=True
            ) as response:
                for line in response.iter_lines():
                    if line:
                        try:
                            decoded_line = line.decode('utf-8')
                            yield json.loads(decoded_line)
                        except json.JSONDecodeError:
                            logger.error(f"Could not parse
streaming response: {line}")
                        except Exception as e:
                            logger.error(f"Error processing stream
line: {e}")
```

```
                            continue
        except Exception as e:
            logger.error(f"Error in streaming generation: {e}")
            yield {"error": str(e)}
            raise
```

Листинг А.2 – Файл «chat.py»

```
import json
import uuid
import logging
import asyncio
from channels.generic.websocket import AsyncWebsocketConsumer
from channels.db import database_sync_to_async
from django.contrib.auth.models import AnonymousUser
from django.conf import settings
from ..models import OllamaModel, TextGenerationJob, Chat, Message
from ..services.ollama_service import ollama_service

logger = logging.getLogger(__name__)

DEFAULT_MAX_CONTEXT_MESSAGES = 10

class ChatConsumer(AsyncWebsocketConsumer):
    """WebSocket consumer for chat with Ollama models"""

    async def connect(self):
        """Handle WebSocket connection"""
        self.chat_id =
self.scope["url_route"]["kwargs"]["chat_id"]
        self.chat_group_name = f"chat_{self.chat_id}"

        self.user = self.scope["user"]
        self.is_authenticated = not isinstance(self.user,
AnonymousUser)

        await self.channel_layer.group_add(
            self.chat_group_name,
            self.channel_name
        )

        await self.accept()

        await self.send(text_data=json.dumps({
            "type": "connection_established",
            "message": "Connected to chat server",
            "chat_id": self.chat_id,
            "authenticated": self.is_authenticated,
            "username": self.user.username if
self.is_authenticated else None
        }, ensure_ascii=False))

    async def disconnect(self, close_code):
```

```
        """Handle WebSocket disconnection"""
        # Leave chat group
        await self.channel_layer.group_discard(
            self.chat_group_name,
            self.channel_name
        )

    async def receive(self, text_data):
        """Handle messages from WebSocket"""
        try:
            data = json.loads(text_data)
            message_type = data.get("type", "")

            if message_type == "chat_message":
                await self.handle_chat_message(data)
            elif message_type == "model_info":
                await self.handle_model_info(data)
            elif message_type == "chat_history_update":
                await self.handle_chat_history_update(data)
            else:
                await self.send(text_data=json.dumps({
                    "type": "error",
                    "message": f"Unknown message type:
{message_type}"
                }, ensure_ascii=False))

        except json.JSONDecodeError:
            await self.send(text_data=json.dumps({
                "type": "error",
                "message": "Invalid JSON"
            }, ensure_ascii=False))
        except Exception as e:
            logger.exception("Error in receive")
            await self.send(text_data=json.dumps({
                "type": "error",
                "message": str(e)
            }, ensure_ascii=False))

    async def handle_chat_message(self, data):
        """Handle chat message from client"""
        model_name = data.get("model")
        prompt = data.get("prompt")
        original_system = data.get("system", "")
        history = data.get("history", [])
        save_history = data.get("save_history", True)
        max_context_messages = data.get("max_context_messages",
DEFAULT_MAX_CONTEXT_MESSAGES)

        if not all([model_name, prompt]):
            await self.send(text_data=json.dumps({
                "type": "error",
                "message": "Missing required fields: model,
```

```
prompt"
            }, ensure_ascii=False))
            return

        system = original_system
        if history and not "conversation history" in
original_system.lower():
            context_instruction = "Answer in Russian anyways"

            if original_system:
                system = f"{context_instruction}
{original_system}"
            else:
                system = context_instruction

        await self.send(text_data=json.dumps({
            "type": "message_received",
            "message_id": str(uuid.uuid4())
        }, ensure_ascii=False))

        if save_history and self.is_authenticated:
            chat = await self.get_or_create_chat(model_name)
            await self.save_message(chat.id, "user", prompt)

            if system and not history and len(await
self.get_chat_messages(chat.id)) <= 1:
                await self.save_message(chat.id, "system", system)

        if not history and save_history and self.is_authenticated:
            chat = await self.get_or_create_chat(model_name)
            history = await
self.get_formatted_chat_history(chat.id, max_context_messages)
        elif history:
            for msg in history:
                if msg.get('role') == 'model':
                    msg['role'] = 'assistant'

        await self.stream_response(model_name, prompt, system,
history, save_history)

    async def get_or_create_chat(self, model_name):
        """Get existing chat or create a new one"""
        try:
            chat = await
database_sync_to_async(Chat.objects.get)(id=self.chat_id)
            return chat
        except Chat.DoesNotExist:
            chat = await
database_sync_to_async(Chat.objects.create)(
                id=self.chat_id,
                user=self.user,
                title=f"Chat with {model_name}"
```

```
            )
            return chat

    @database_sync_to_async
    def save_message(self, chat_id, sender, content):
        """Save a message to the chat history"""
        chat = Chat.objects.get(id=chat_id)
        message = Message.objects.create(
            chat=chat,
            sender=sender,
            content=content
        )
        # Update the chat's updated_at timestamp
        chat.save(update_fields=['updated_at'])
        return message

    @database_sync_to_async
    def get_chat_messages(self, chat_id):
        """Get all messages for a chat"""
        chat = Chat.objects.get(id=chat_id)
        return list(chat.messages.all())

    @database_sync_to_async
    def check_model_exists(self, model_name):
        """Check if a model exists in the database"""
        return OllamaModel.objects.filter(name=model_name,
status=OllamaModel.READY).exists()

    @database_sync_to_async
    def create_generation_job(self, model_name, prompt,
system=None, history=None):
        """Create a new generation job"""
        model = OllamaModel.objects.get(name=model_name)

        user = None
        if self.is_authenticated:
            user = self.user

        chat = None
        if self.is_authenticated:
            try:
                chat = Chat.objects.get(id=self.chat_id)
            except Chat.DoesNotExist:
                chat = Chat.objects.create(
                    id=self.chat_id,
                    user=self.user,
                    title=f"Chat with {model_name}"
                )

        options = {}

        job = TextGenerationJob.objects.create(
```

```
            user=user,
            chat=chat,
            model=model,
            prompt=prompt,
            system=system,
            status=TextGenerationJob.PROCESSING,
            options=options)

        return job
```

Листинг А.3 – Файл «chatWebSocket.ts»

```
<template>
  <div class="chat-page">
    <!-- Sidebar -->
    <ChatSidebar
      v-model="isSidebarOpen"
      :chats="chatStore.chatsWithPreview"
      :selectedChatId="selectedChatId"
      :loading="isLoading"
      @select-chat="loadSelectedChat"
      @delete-chat="confirmDeleteChat"
      @new-chat="createNewChat"
      @refresh-chats="loadChats"
    />

    <!-- Main chat area -->
    <div class="chat-main">
      <ChatHeader
        :title="chatStore.currentChat?.title"
        @toggle-sidebar="isSidebarOpen = true"
      />

      <div class="model-selector-container">
        <ModelSelector
          @change="model = $event"
          :connected="wsStatus === 'Connected'"
          :connection-status="wsStatus"
        />
      </div>

      <div
        class="messages-container"
        ref="messagesContainer"
        @scroll="handleScroll"
      >
        <div v-if="messages.length === 0" class="empty-chat-
guide">
          <div class="guide-content">
            <h3>Начните новый разговор</h3>
            <p>Контекст предыдущих сообщений используется для
генерации более точных ответов.</p>
          </div>
```

```
        </div>

        <div v-for="(msg, index) in messages" :key="msg.id">
          <MessageBubble
            :message="msg.text"
            :role="msg.role"
            :streaming="isReceivingResponse &&
isLastMessage(index)"
          />
        </div>
        <div ref="messagesEnd"></div>
      </div>

      <!-- Scroll button -->
      <v-btn
        v-if="!atBottom"
        class="scroll-to-bottom-btn"
        icon
        @click="scrollToBottom"
      >
        <v-icon>mdi-arrow-down</v-icon>
      </v-btn>

      <!-- Input field -->
      <ChatInput
        v-model="prompt"
        @send="sendMessage"
        :placeholder="isReceivingResponse ? 'Модель отвечает...' :
'Введите сообщение...'"
      />
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref, nextTick, onMounted } from 'vue'
import { useChatStore } from '../store/chat'
import { useAuthStore } from '../store/auth'
import { ChatWebSocket } from '../services/chatWebSocket'
import ChatHeader from '@/components/ChatHeader.vue'
import ChatSidebar from '@/components/ChatSidebar.vue'
import MessageBubble from '@/components/MessageBubble.vue'
import ChatInput from '@/components/ChatInput.vue'
import ModelSelector from '@/components/ModelSelector.vue'
import type { ChatMessage } from '../store/chat'
import { useToast } from 'vue-toast-notification'
import 'vue-toast-notification/dist/theme-sugar.css'

const chatStore = useChatStore()
const authStore = useAuthStore()
const messages = ref<ChatMessage[]>([])
const prompt = ref('')
```

```
const model = ref('qwen3:0.6b')

const isSidebarOpen = ref(false)
const isReceivingResponse = ref(false)
const wsStatus = ref('Disconnected')

const messagesContainer = ref<HTMLElement | null>(null)
const messagesEnd = ref<HTMLElement | null>(null)
const atBottom = ref(true)

const selectedChatId = ref('')

let socket: ChatWebSocket

// Add isLoading state
const isLoading = ref(false)

// Add variables for tracking connection
const isConnectionError = ref(false)
const connectionErrorMessage = ref('')

// Create toast instance
const toast = useToast()

// Function to generate UUID v4
function generateUUID() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
function(c) {
    const r = Math.random() * 16 | 0,
          v = c === 'x' ? r : (r & 0x3 | 0x8);
    return v.toString(16);
  });
}

function initWebSocket() {
  if (socket) {
    socket.close();
  }

  try {
    // Get the raw token without Bearer prefix
    const token = authStore.accessToken;
    console.log('Initializing WebSocket with token:', token ?
token.substring(0, 8) + '...' : 'none');

    socket = new ChatWebSocket(
      selectedChatId.value,
      handleMessage,
      handleConnectionError,
      handleConnectionSuccess,
      token // Pass token directly
    );
```

```
    socket.connect();
  } catch (error) {
    console.error('Failed to initialize WebSocket:', error);
    handleConnectionError(error);
  }
}

// Function to remove think blocks from message text
function removeThinkBlocks(text: string): string {
  if (!text) return text;
  return text.replace(/<think>[\s\S]*?<\/think>/g, '').trim();
}

function handleMessage(msg: any) {
  if (msg.type === 'chat_response_chunk') {
    const last = messages.value[messages.value.length - 1]
    if (last && last.role === 'model') {
      last.text += msg.chunk
    } else {
      const newMessage: ChatMessage = {
        role: 'model',
        text: msg.chunk,
        id: generateUUID()
      }
      messages.value.push(newMessage)
    }
    isReceivingResponse.value = true
    scrollToBottom()

    // Save chat state to store after each chunk
    saveMessagesToStore()
  }

  if (msg.type === 'chat_response_complete') {
    isReceivingResponse.value = false

    // Clean think blocks from the last model message for display
    const lastMessage = messages.value[messages.value.length - 1]
    if (lastMessage && lastMessage.role === 'model') {
      // Store the original message with think blocks (for display
only)
      const originalText = lastMessage.text

      // Remove think blocks for the chat history
      lastMessage.text = removeThinkBlocks(lastMessage.text)

      // Get original text back for display (MessageBubble
component knows how to handle think blocks)
      nextTick(() => {
        lastMessage.text = originalText
      })
    }
```

```
    // Update chat title if it's a new chat or has a default title
    if (chatStore.currentChat &&
        (chatStore.currentChat.title === 'Новый чат' ||
         chatStore.currentChat.title === `Чат от ${new
Date().toLocaleString()}` ||
         chatStore.currentChat.title.startsWith('Chat with'))) {
      // Get first user message
      const userMessage = messages.value.find(m => m.role ===
'user')
      if (userMessage) {
        // Generate a better title based on the first user message
        const titlePreview = userMessage.text.split(' ')
          .slice(0, 6)
          .join(' ')
          .replace(/[?!.,;:]+$/, '') // Remove trailing
punctuation
          .replace(/^['"(]|['").]$/g, '') // Remove surrounding
quotes/brackets
          .trim();

        const betterTitle = titlePreview.length > 40 ?
          titlePreview.substring(0, 40) + '...' :
          titlePreview;

        // Update title with backend sync - ensure this happens
immediately
        chatStore.updateChatTitle(selectedChatId.value,
betterTitle)
          .then(() => console.log('Chat title updated
successfully'))
          .catch(err => console.error('Failed to update chat
title:', err));
      }
    }

    // Attempt to sync with server after generation is complete,
with fallback to local
    if (selectedChatId.value) {
      const chat = chatStore.chats.find(c => c.id ===
selectedChatId.value);
      if (chat) {
        // Ensure the chat has all current messages before syncing
        chat.messages = [...messages.value];

        // If no server API is running, we'll just save locally
        if (!chatStore.syncEnabled) {
          chatStore.saveChats();
        } else {
          // Try syncing with server, with fallback to local
storage
          chatStore.syncWithServer(chat).catch(error => {
```

```
            console.warn('Server sync failed, saving locally
only', error);
            chatStore.saveChats();
          });
        }
      }
    }

    // Save final chat state to store as backup
    saveMessagesToStore()
  }
}

// Function to save current messages to the chat store
function saveMessagesToStore() {
  if (selectedChatId.value && messages.value.length > 0) {
    chatStore.updateChatMessages(selectedChatId.value,
[...messages.value])
    chatStore.saveChats() // Explicitly save to localStorage
    chatStore.saveCurrentChatState() // Save the current chat ID
  }
}

function handleConnectionSuccess() {
  wsStatus.value = 'Connected'
  isConnectionError.value = false
  connectionErrorMessage.value = ''
}

function handleConnectionError(err: any) {
  console.error('WebSocket error:', err)
  wsStatus.value = 'Error'
  isConnectionError.value = true
  connectionErrorMessage.value = 'Connection error. Please try
again later.'

  toast.error('Ошибка соединения с сервером. Пожалуйста, проверьте
подключение к интернету.', {
    position: 'top',
    duration: 5000,
    dismissible: true
  })
}

function scrollToBottom() {
  nextTick(() => {
    if (messagesContainer.value) {
      messagesContainer.value.scrollTop =
messagesContainer.value.scrollHeight
    }
  })
}
```

```
function handleScroll() {
  if (!messagesContainer.value) return
  const el = messagesContainer.value
  atBottom.value = el.scrollTop + el.clientHeight >=
el.scrollHeight - 40
}

function isLastMessage(index: number) {
  return index === messages.value.length - 1
}

async function sendMessage() {
  if (!prompt.value.trim() || isReceivingResponse.value) return
  const text = prompt.value.trim()
  prompt.value = ''

  const newMessage: ChatMessage = {
    role: 'user',
    text,
    id: generateUUID()
  }
  messages.value.push(newMessage)

  // Save to store immediately after adding user message
  saveMessagesToStore()

  try {
    await socket.send(text, model.value, 'отвечай на русском',
selectedChatId.value, true)
    isReceivingResponse.value = true
    scrollToBottom()
  } catch (error) {
    console.error('Error sending message:', error)
    toast.error('Не удалось отправить сообщение. Пожалуйста,
проверьте соединение.', {
      position: 'top',
      duration: 3000,
      dismissible: true
    })
  }
}

async function createNewChat() {
  try {
    isLoading.value = true
    const chatId = generateUUID()
    selectedChatId.value = chatId
    messages.value = []

    // Create new chat with a more descriptive default title
    const defaultTitle = `Чат от ${new Date().toLocaleString()}`
```

```
    const newChat = chatStore.createChat(chatId, defaultTitle,
model.value)

    // Sync with server
    await chatStore.syncWithServer(newChat)

    // Initialize WebSocket after chat is created
    initWebSocket()
  } catch (error) {
    console.error('Error creating new chat:', error)
    toast.error('Ошибка при создании чата. Пожалуйста, попробуйте
еще раз.', {
      position: 'top',
      duration: 3000,
      dismissible: true
    })
  } finally {
    isLoading.value = false
  }
}

async function loadSelectedChat(id: string) {
  try {
    selectedChatId.value = id
    isLoading.value = true

    await chatStore.loadChatFromServer(id)
    messages.value = chatStore.getMessagesByChatId(id)

    // Initialize WebSocket for this chat
    initWebSocket()
  } catch (error) {
    console.error('Error loading chat:', error)
    toast.error('Ошибка при загрузке чата. Пожалуйста, попробуйте
еще раз.', {
      position: 'top',
      duration: 3000,
      dismissible: true
    })
  } finally {
    isLoading.value = false
  }
}
```