МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	*Кафедра*
вычислительной техники	«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

ГОЛОСОВОЙ ПОМОЩНИК ДЛЯ ПК

Студент _____ 26.05.2026 Субботкин Максим Владимирович
(подпись, дата) (ФИО полностью)

Руководитель _____ 26.05.2026 Юрова О.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ 26.05.2026 Бычков А.С.
(подпись, дата) (фамилия, инициалы)

Рецензент зав. каф. Программирования _____ 27.05.2026 Мартынкин А.И.
(должность, место работы) (подпись, дата) (фамилия, инициалы)

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

Заведующий кафедрой _____ М.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 4.06.26 № 6)

Секретарь ГЭК _____ Е.И. Гурин
(подпись) (фамилия, инициалы)

Пенза 2026

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

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

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

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

1. Студент Субботкин Максим Владимирович гр. 24ВВИм1 факультета вычислительной техники, направления 09.04.01

2. Тема работы «Голосовой помощник для ПК»

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

3. Руководитель магистерской диссертации к.т.н., доцент Юрова О.В.

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

Назначение разработки: Разрабатываемая система предназначена для голосового управления персональным компьютером в автономном режиме, без необходимости подключения к сети Интернет и использования облачных сервисов. Основное назначение разработки заключается в обеспечении удобного взаимодействия пользователя с операционной системой и прикладными программами посредством голосовых команд на естественном языке.

5. Перечень подлежащих разработке вопросов:

1) Анализ предметной области

2) Разработка архитектуры

3) Реализация прототипа системы

4) Проведение экспериментального сравнения с альтернативными подходами

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Изучить теоретические основы	10%	27.09.2024-15.01.2025	
2	Анализ предметной области	10%	16.01.2025-22.03.2025	
3	Выбор инструментов и библиотек, формирование требований к модели	5%	23.03.2025-14.06.2025	
4	Разработка архитектуры	20%	15.06.2025-10.10.2025	
5	Реализация прототипа системы	30%	11.10.2025-05.01.2026	
6	Проведение экспериментального сравнения с альтернативными подходами	15%	06.01.2026-17.04.2026	
7	Анализ полученных результатов исследования	5%	18.04.2026-09.05.2026	
8	Оформление выпускной работы	5%	10.05.2026-25.05.2026	

Дата выдачи задания "20" сентября 2024 г.

Руководитель магистерской диссертации _____ Юрова О.В.
(подпись, дата)

Задание к исполнению принял студент _____ Субботкин М.В.
(подпись, дата)

Работу к защите допустить
Декан факультета _____ Фионова Л.Р.
(подпись, дата)
01.06.2026

РЕФЕРАТ

Магистерская работа содержит 99 листов, 3 рисунка, 9 таблиц, 8 источников и 1 приложение.

ГОЛОСОВОЙ ПОМОЩНИК, НЕЙРОННЫЕ СЕТИ, GRU, PYTORCH, VOSK, ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Целью данной работы является разработка автономного голосового помощника для персонального компьютера, обеспечивающего обработку пользовательских команд без использования облачных сервисов и постоянного подключения к сети Интернет.

В работе рассмотрены существующие подходы к построению голосовых помощников, проанализированы методы офлайн-распознавания речи и локальной обработки естественного языка. Разработана архитектура программной системы, включающая модули распознавания и синтеза речи, нейросетевой классификации намерений, извлечения сущностей, выполнения действий, самообучения и пользовательского интерфейса.

Для интерпретации голосовых команд реализована компактная нейросетевая модель на основе GRU, позволяющая учитывать последовательность слов и определять намерение пользователя. Для повышения устойчивости системы дополнительно применяются морфологическая нормализация, fallback-механизм и правило-ориентированное извлечение параметров команды. Реализован механизм самообучения, позволяющий добавлять новые программы, пользовательские глаголы и макросы без изменения исходного кода.

Разработанный голосовой помощник может использоваться для локального управления персональным компьютером, обеспечивая автономность, конфиденциальность и низкие требования к вычислительным ресурсам.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ 6
ВВЕДЕНИЕ 7
1 Анализ предметной области 9
1.1 Общие сведения о голосовых помощниках 9
1.2 Нейросетевые методы в голосовых помощниках 9
1.3 Анализ существующих нейросетевых голосовых помощников 10
1.4 Подходы к построению автономных нейросетевых голосовых помощников 11
1.5 Постановка задачи 12
1.6 Выбор инструментов разработки 13
2 Архитектура и реализация голосового помощника 17
2.1 Общая архитектура системы 17
2.2 Архитектура программной реализации 19
2.2.1 Центральный управляющий модуль 20
2.2.2 Модуль классификации намерений 20
2.2.3 Модуль извлечения сущностей 20
2.2.4 Исполнительный модуль 21
2.2.5 Модуль выполнения действий 21
2.2.6 Модуль распознавания и синтеза речи 21
2.2.7 Модуль самообучения 22
2.2.8 Пользовательский интерфейс 22
2.2.9 Хранение данных 22
2.3 Модуль распознавания речи 23
2.4 Модуль обработки естественного языка 24
2.5 Гибридный механизм интерпретации команд 29
2.5.1 Правило-ориентированное извлечение сущностей 30
2.5.2 Морфологическая нормализация 30
2.5.3 Проверка глагольной конструкции 31
2.5.4 Обработка низкой уверенности модели 31

2.5.5 Fallback-механизм 32
2.5.6 Преимущества гибридного подхода 33
2.6 Механизм самообучения системы 34
2.6.1 Обучение новым программам 36
2.6.2 Обучение неизвестным глаголам 37
2.6.3 Обучение пользовательским макросам 37
2.6.4 Обновление обучающей выборки 38
2.6.5 Преимущества механизма самообучения 39
2.7 Сравнительный анализ разработанной модели и альтернативных подходов 39
2.7.1 Сравнение с Bag-of-Words 40
2.7.2 Сравнение с LSTM 41
2.7.3 Сравнение с Transformer-архитектурами 41
2.7.4 Сравнение с крупными языковыми моделями 42
2.7.5 Расчёт параметров разработанной модели 43
2.7.6 Итоговое сравнение подходов 44
2.8 Вывод 44
3 Экспериментальная оценка и сравнительный анализ разработанного решения 48
3.1 Цель и методика экспериментального исследования 48
3.2 Описание экспериментального набора данных 51
3.3 Результаты классификации намерений 54
3.3.1 Общие результаты классификации 55
3.4 Сравнение вычислительных ресурсов 57
3.4.1 Размер модели и потребление памяти 57
3.4.2 Время инференса 59
3.4.3 Загрузка центрального процессора 60
3.4.4 Интегральное сравнение моделей 61
3.5 Итоговое сравнение разработанного решения с альтернативами 62
3.5.1 Сравнение по точности и ресурсам 62

3.5.2 Преимущества разработанного решения 64
3.5.3 Ограничения разработанного решения 65
3.5.4 Итоговая оценка 66
ЗАКЛЮЧЕНИЕ 69
ПРИЛОЖЕНИЕ А 73
Приложение А.1 74
Приложение А.2 80
Приложение А.3 88
ПРИЛОЖЕНИЕ В 90

ПЕРЕЧЕНЬ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

В данной работе используются следующие обозначения и сокращения:

ПК − Персональный компьютер

CPU − Central Processing Unit, центральный процессор

GPU − Graphics Processing Unit, графический процессор

GUI − Graphical User Interface, графический пользовательский интерфейс

GRU − Gated Recurrent Unit, рекуррентная нейронная сеть с управляемыми вентилями

JSON − JavaScript Object Notation, текстовый формат хранения и обмена данными

LLM − Large Language Model, крупная языковая модель

LSTM − Long Short-Term Memory, рекуррентная нейронная сеть с долгой краткосрочной памятью

NLP − Natural Language Processing, обработка естественного языка

RAM − Random Access Memory, оперативная память

RNN − Recurrent Neural Network, рекуррентная нейронная сеть

STT − Speech-to-Text, преобразование речи в текст

TTS − Text-to-Speech, преобразование текста в речь

ВВЕДЕНИЕ

В последнее время информационные технологии и ИИ развиваются с большой скоростью. Это приводит к активному внедрению интеллектуальных интерфейсов взаимодействия человека с компьютером. Одним из наиболее перспективных направлений является разработка голосовых помощников, которые обеспечивают естественное и удобное управление программными и аппаратными средствами ПК.

Голосовые интерфейсы позволяют сильно упростить взаимодействие пользователя и системы, снизить зависимость от стандартных устройств ввода таких как клавиатура и мышь. Это актуально в условиях многозадачности и для людей с ограниченными возможностями.

Несмотря на широкое распространение голосовых помощников в мобильных устройствах, использование их в персональных компьютерах остаётся ограниченным и требует дальнейших исследований. Существующие подходы и решения обладают недостаточной точностью распознавания речи, ограниченной функциональностью, сложностью интеграции с операционной системой, большими требованиями к вычислительной мощности.

Большинство современных подходов ориентировано на использование облачных вычислений из-за чего они зависят от стабильного подключения к интернету. Такая архитектура приводит к ряду ограничений, включая задержки при обработке запросов, риски, связанные с конфиденциальностью пользовательских данных, а также невозможность функционирования системы в условиях отсутствия интернета.

Актуальность темы магистерской диссертации обусловлена необходимостью создания эффективного программного решения для голосового управления персональным компьютером, не зависящего от интернет-соединения и обеспечивающего достаточный уровень точности и быстродействия.

Целью настоящей работы является разработка и исследование голосового помощника для персонального компьютера, функционирующего

в автономном режиме и обеспечивающего обработку голосовых команд на естественном языке.

Для достижения поставленной цели необходимо решить следующие задачи:

− проведение анализа существующих голосовых помощников с акцентом на их зависимость от облачных технологий,

− исследование методов офлайн распознавания и синтеза речи,

− анализ подходов к локальной обработке естественного языка,

− разработка архитектуры автономной системы голосового помощника,

− реализация прототипа программного решения,

− проведение экспериментальной оценки эффективности разработанной системы.

Объектом исследования является процесс голосового взаимодействия пользователя с персональным компьютером в условиях автономной работы системы.

Предметом исследования являются методы и алгоритмы локального распознавания речи, обработки естественного языка и управления операционной системой на основе голосовых команд без использования сетевых ресурсов.

Научная новизна работы заключается в разработке и обосновании архитектурных и алгоритмических решений, направленных на обеспечение эффективного функционирования голосового помощника в офлайн-режиме при ограниченных вычислительных ресурсах.

Практическая значимость результатов исследования определяется возможностью создания программного продукта, обеспечивающего надежное и безопасное голосовое управление персональным компьютером без необходимости подключения к сети Интернет.

1 Анализ предметной области

1.1 Общие сведения о голосовых помощниках

Голосовые помощники представляют собой программные системы, реализующие взаимодействие пользователя с вычислительными устройствами посредством естественной речи. Развитие данных систем тесно связано с прогрессом в области искусственного интеллекта, в частности методов машинного обучения и нейронных сетей.

Современные голосовые помощники обеспечивают выполнение широкого спектра задач, включая управление программными средствами, обработку пользовательских запросов и взаимодействие с операционной системой. В основе их функционирования лежит последовательная обработка речевого сигнала, включающая преобразование речи в текстовую форму, интерпретацию пользовательского намерения и выполнение соответствующих действий.

Ключевым фактором, определяющим эффективность голосового помощника, является способность системы корректно интерпретировать команды пользователя. Голосовое взаимодействие характеризуется высокой степенью вариативности, что требует применения интеллектуальных методов обработки данных.

В настоящее время ведущую роль в разработке голосовых помощников играют нейросетевые методы, позволяющие сильно повысить точность распознавания речи и качество интерпретации пользовательских запросов.

1.2 Нейросетевые методы в голосовых помощниках

Современные голосовые помощники в большей степени опираются на методы глубокого обучения, которые обрабатывают речевые и текстовые данные. Применение нейронных сетей позволяет учитывать сложные зависимости в данных и эффективно работать с неструктурированной информацией.

На этапе преобразования речи используются нейросетевые модели, которые способны обрабатывать акустические признаки и сопоставлять их с

текстовыми представлениями. Такие модели обеспечивают высокую точность распознавания речи, особенно в условиях шума и вариативности произношения.

Особое значение имеет применение нейросетевых методов на этапе интерпретации команд пользователя. В данной задаче используются модели обработки естественного языка, позволяющие определить намерение пользователя и извлечь параметры запроса.

Современные архитектуры, такие как рекуррентные нейронные сети и трансформеры, позволяют моделировать контекст и учитывать последовательность слов, что повышает качество обработки пользовательских запросов. Однако использование подобных моделей часто связано с высокими требованиями к вычислительным ресурсам.

1.3 Анализ существующих нейросетевых голосовых помощников

На сегодняшний день ведущие голосовые помощники реализовываются с использованием нейросетевых технологий и интегрируются в различные программные и аппаратные платформы.

Одним из наиболее известных решений является Siri, использующий методы машинного обучения для обработки пользовательских запросов и управления системой. Данный помощник демонстрирует высокую степень интеграции с устройствами, однако его функциональность во многом зависит от облачной инфраструктуры.

Широкое распространение также получил Google Assistant, в основе которого лежат современные нейросетевые модели обработки речи и текста.

В области интеллектуальных систем управления устройствами активно применяется Amazon Alexa, поддерживающий расширение функциональности за счет сторонних сервисов. Обработка запросов в данной системе также осуществляется преимущественно с использованием облачных технологий.

Среди решений, ориентированных на русскоязычную аудиторию, следует выделить Алиса, в которой активно применяются нейросетевые

методы обработки естественного языка. Данный помощник демонстрирует высокую адаптацию к разговорной речи и контексту.

Анализ существующих систем показывает, что современные голосовые помощники достигают высокой эффективности за счет применения нейросетевых моделей и облачной архитектуры. Однако такой подход сопровождается определенными ограничениями, включая зависимость от соединения с интернетом, необходимость передачи пользовательских данных на удаленные серверы и высокие требования к вычислительным ресурсам.

1.4 Подходы к построению автономных нейросетевых голосовых помощников

Одним из перспективных направлений развития голосовых интерфейсов является создание автономных систем, в которых обработка пользовательских запросов осуществляется локально с использованием нейросетевых моделей.

В таких системах особое внимание уделяется разработке легковесных и эффективных нейросетевых моделей, способных функционировать в ограниченных условиях. В отличие от облачных решений, где используются крупные языковые модели, автономные системы требуют оптимизации архитектуры нейронных сетей и снижения их вычислительной сложности.

Ключевым компонентом автономного голосового помощника является нейросетевая модель интерпретации команд, обеспечивающая определение намерения пользователя и извлечение ключевых параметров запроса. Использование нейронных сетей позволяет учитывать вариативность человеческого языка и повышает гибкость системы по сравнению с традиционными подходами.

Также важным аспектом является интеграция различных нейросетевых и алгоритмических компонентов в единую систему, обеспечивающую обработку запросов в реальном времени. Это требует разработки эффективных механизмов взаимодействия между модулями и оптимизации процесса обработки данных.

Таким образом, создание автономного нейросетевого голосового помощника представляет собой задачу, которая направлена на достижение баланса между точностью, производительностью и независимостью от внешних вычислительных ресурсов.

1.5 Постановка задачи

Проведённый анализ показал, что современные голосовые помощники в основном используют нейросетевые методы обработки данных в сочетании с облачной архитектурой. Несмотря на высокую точность и функциональность таких систем, их зависимость от интернет-соединения и внешних вычислительных ресурсов ограничивает возможности применения в автономных условиях.

В то же время существующие офлайн-решения не в полной мере используют потенциал нейросетевых методов для интерпретации пользовательских команд, что снижает гибкость и интеллектуальность таких систем.

В связи с этим возникает задача разработки автономного голосового помощника для персонального компьютера, использующего нейросетевые модели для обработки пользовательских команд и функционирующего без обращения к внешним сервисам.

Целью работы является разработка программной системы голосового помощника, основанной на использовании локальных нейросетевых моделей для интерпретации голосовых команд.

Для достижения поставленной цели необходимо решить следующие задачи:

− разработать архитектуру автономного нейросетевого голосового помощника,

− реализовать модуль распознавания речи с использованием локальных средств,

− разработать и обучить нейросетевые модели для интерпретации пользовательских команд,

− обеспечить интеграцию всех компонентов системы,

− провести оценку эффективности разработанного решения.

Таким образом, в рамках данной работы предполагается создание автономного голосового помощника, в котором ключевую роль играют нейросетевые методы обработки данных, обеспечивающие интеллектуальную интерпретацию пользовательских запросов в условиях отсутствия сетевого взаимодействия.

1.6 Выбор инструментов разработки

При разработке автономного голосового помощника для персонального компьютера особое внимание уделялось выбору инструментов и технологий, обеспечивающих выполнение следующих требований:

− автономность функционирования (отсутствие зависимости от сети Интернет),

− поддержка обработки естественного языка,

− возможность интеграции с операционной системой,

− расширяемость и гибкость архитектуры,

− доступность и кроссплатформенность.

В результате анализа существующих решений был сформирован следующий стек технологий.

В качестве основного языка разработки выбран Python.

Выбор обусловлен следующими факторами:

− наличие большого количества библиотек для машинного обучения и обработки речи,

− высокая скорость разработки прототипов,

− удобство работы с системными вызовами,

Для реализации офлайн-распознавания речи выбрана библиотека Vosk.

Преимущества данного решения:

− поддержка работы без подключения к сети,

− наличие готовых акустических моделей для русского языка,

− возможность потоковой обработки аудиосигнала,

− сравнительно низкие требования к вычислительным ресурсам.

В отличие от облачных решений (например, Google Speech-to-Text), использование локальной модели позволяет обеспечить независимость системы от внешних сервисов и повысить уровень конфиденциальности.

Таблица 1 - Сравнение библиотек распознавания речи

Критерий	Vosk	Google Speech-to-Text	Whisper
Требуется интернет	Нет	Да	Нет
Поддержка русского языка	Да	Да	Да
Точность распознавания	Средняя	Высокая	Высокая
Задержка	Низкая	Средняя	Высокая
Требования к ресурсам	Средние	Низкие	Высокие
Потоковая обработка	Да	Да	Ограниченно
Конфиденциальность данных	Высокая	Низкая	Высокая

Для реализации нейросетевой модели выбран фреймворк PyTorch.

Выбор обусловлен следующими причинами:

− гибкость при построении нейросетевых архитектур,

− удобство реализации рекуррентных моделей (GRU, LSTM),

− простота отладки и модификации модели,

− поддержка динамических вычислительных графов.

Использование PyTorch позволило реализовать компактную модель классификации намерений с возможностью дальнейшего расширения.

Для генерации голосовых ответов используется библиотека pyttsx3.

Данное решение выбрано по следующим причинам:

− полностью локальная работа,

− отсутствие зависимости от интернет-сервисов,

− простота интеграции,

− поддержка различных голосовых движков операционной системы.

Для взаимодействия с операционной системой и эмуляции пользовательских действий применяются следующие библиотеки:

− pyautogui - для управления клавиатурой и выполнением горячих клавиш,

− keyboard - для обработки ввода с клавиатуры,

− psutil - для получения информации о запущенных процессах.

Использование данных инструментов позволяет реализовать управление системой на уровне пользовательского интерфейса.

Для реализации механизма обучения макросам используется библиотека pynput.

Она обеспечивает:

− перехват событий клавиатуры,

− запись последовательности действий пользователя,

− последующее воспроизведение записанных сценариев.

Это позволяет реализовать гибкий механизм расширения функциональности системы без изменения исходного кода.

Для разработки пользовательского интерфейса используется стандартная библиотека Tkinter.

Выбор обусловлен:

− наличием в стандартной поставке Python,

− простотой реализации интерфейса,

− достаточной функциональностью для задач отображения состояния системы.

GUI выполняет вспомогательную функцию, обеспечивая мониторинг работы ассистента и отображение диагностической информации.

Для хранения конфигурационных данных используется формат JSON.

Основные файлы:

- intents.json - обучающая выборка,
- actions.json - описание действий,
- cache/ - сохранённая модель.

Использование JSON позволяет:

- упростить модификацию данных,
- обеспечить читаемость конфигурации,
- реализовать динамическое расширение системы.

2 Архитектура и реализация голосового помощника

2.1 Общая архитектура системы

Разрабатываемая система представляет собой автономный голосовой помощник, предназначенный для управления персональным компьютером с использованием естественного языка без привлечения облачных сервисов.

Функционирование системы основано на последовательной обработке пользовательского запроса, который проходит несколько этапов преобразования. Общая схема работы может быть представлена в следующем виде:

Аудиосигнал → Текст → Определение намерения → Извлечение сущностей → Выполнение действия → Ответ

Формально процесс обработки можно представить как последовательность отображений:

В состав системы входят 6 основных модулей:

1) модуль распознавания речи, выполняющий преобразование аудиосигнала в текстовую форму,

2) модуль обработки естественного языка, определяющий намерение пользователя на основе текста команды,

3) модуль извлечения сущностей, выделяющий параметры команды, необходимые для её выполнения,

4) исполнительный модуль, осуществляющий выполнение действий в операционной системе,

5) модуль самообучения, обеспечивающий адаптацию системы к новым пользовательским сценариям,

6) пользовательский интерфейс, предназначенный для визуализации работы системы и взаимодействия с пользователем.

Взаимодействие между модулями осуществляется посредством передачи структурированных данных. На этапе обработки текстовой команды формируется объект команды, содержащий:

− исходный текст,

− определённое намерение,

− уровень уверенности модели,

− извлечённые параметры.

Данный объект используется в дальнейшем для выполнения действий и генерации ответа.

Особенностью архитектуры является разделение этапов понимания команды и её выполнения. Нейросетевая модель отвечает исключительно за интерпретацию пользовательского запроса, тогда как выполнение действий реализуется через отдельный конфигурационный механизм. Это позволяет:

− повысить гибкость системы,

− упростить добавление новых функций,

− обеспечить независимость логики выполнения от модели обработки языка.

Таким образом, предложенная архитектура представляет собой многоуровневую систему обработки команд, обеспечивающую преобразование неструктурированного голосового ввода в формализованные действия операционной системы.

Для наглядного представления архитектуры разработанной системы на рисунке 1 приведена структурная схема взаимодействия основных компонентов.

Диаграмма отражает последовательность преобразования пользовательского голосового запроса от этапа распознавания речи до выполнения действия в операционной системе и формирования ответа.

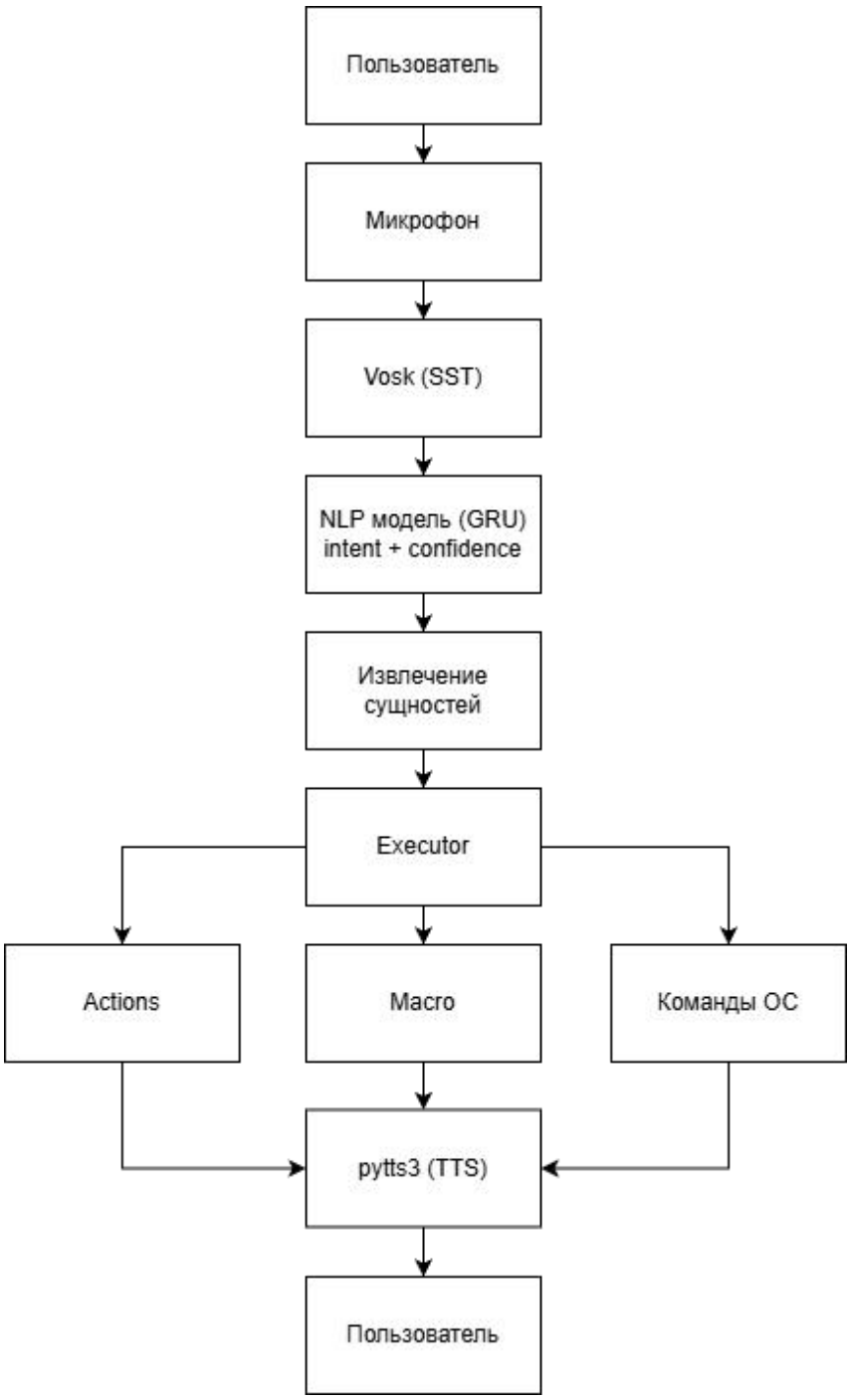


Рисунок 1 - Архитектура системы

2.2 Архитектура программной реализации

Программная реализация разработанного голосового помощника выполнена в виде модульной структуры, обеспечивающей разделение функциональности на независимые компоненты. Такой подход позволяет упростить сопровождение системы, повысить её расширяемость и обеспечить возможность масштабирования.

2.2.1 Центральный управляющий модуль

Файл assistant_core.py является центральным элементом системы и реализует координацию работы всех компонентов. Он выполняет следующие функции:

- инициализация модулей системы,
- организация основного цикла обработки команд,
- управление взаимодействием между компонентами,
- обработка состояний системы.

2.2.2 Модуль классификации намерений

Модуль intent_model.py реализует нейросетевую модель, предназначенную для определения намерения пользователя.

Основные функции модуля:

- подготовка обучающей выборки,
- формирование словаря,
- обучение модели,
- предсказание класса намерения.

Особенностью реализации является использование кэширования, позволяющего сохранять обученную модель и повторно использовать её без необходимости повторного обучения при каждом запуске.

2.2.3 Модуль извлечения сущностей

Файл entities.py отвечает за извлечение параметров команды на основе текста и определённого намерения.

Модуль реализует набор правил, позволяющих выделять:

- имена приложений,
- текст для ввода,
- временные параметр,
- названия папок и действий.

Данный подход позволяет эффективно обрабатывать команды без использования сложных моделей последовательной разметки.

2.2.4 Исполнительный модуль

Модуль executor.py реализует логику выполнения команд. Он принимает объект команды и определяет соответствующее действие.

Основные функции:

- обработка различных типов намерений,
- вызов соответствующих действий,
- взаимодействие с модулем самообучения.

2.2.5 Модуль выполнения действий

Файл action_engine.py отвечает за непосредственное взаимодействие с операционной системой.

Действия описываются в виде конфигурации листинг которой приведен ниже:

```
{
    "mode": "command",
    "value": "notepad.exe"
}
```

Модуль поддерживает различные режимы:

- запуск программ,
- открытие URL,
- выполнение горячих клавиш,
- выполнение макросов.

Ключевой особенностью является универсальность механизма выполнения.

2.2.6 Модуль распознавания и синтеза речи

Файл speech.py реализует:

- распознавание речи (STT),
- синтез речи (TTS).

Распознавание осуществляется через Vosk, а синтез — через pyttsx3.

Модуль обеспечивает:

- потоковую обработку аудио,

− генерацию голосовых ответов,

− взаимодействие с пользователем.

2.2.7 Модуль самообучения

Модуль learning.py и macro_recorder.py реализуют механизм адаптации системы.

Основные функции:

− определение активного приложения,

− добавление новых программ,

− запись пользовательских действий,

− обновление конфигурации системы.

Самообучение реализуется без изменения исходного кода, что является важным преимуществом системы.

2.2.8 Пользовательский интерфейс

Модуль gui.py реализует графический интерфейс системы.

Он обеспечивает:

− отображение текущего состояния системы,

− вывод распознанных команд,

− отображение результатов классификации,

− просмотр списка доступных действий.

Интерфейс используется как средство мониторинга и отладки.

2.2.9 Хранение данных

Система использует следующие форматы хранения:

− JSON - для конфигурации (intents.json, actions.json),

− бинарный формат - для модели (.pt),

− текстовые файлы - для логов.

Такой подход обеспечивает простоту модификации и прозрачность работы системы.

2.3 Модуль распознавания речи

Модуль распознавания речи предназначен для преобразования аудиосигнала, получаемого с микрофона, в текстовую форму, пригодную для дальнейшей обработки системой.

В качестве основы для реализации данного модуля используется библиотека Vosk, обеспечивающая офлайн-распознавание речи и не требующая подключения к сети Интернет.

Работа модуля основана на потоковой обработке аудиосигнала. Входной сигнал с микрофона разбивается на последовательность небольших блоков фиксированного размера, которые последовательно передаются в распознаватель.

Каждый фрагмент аудиосигнала обрабатывается моделью распознавания, в результате чего формируется текстовая последовательность.

В системе используется потоковый режим распознавания, при котором аудиосигнал поступает в реальном времени.

Обработка реализуется с использованием буфера, в который помещаются поступающие аудиоданные. При накоплении достаточного объёма данных выполняется попытка распознавания.

Распознавание осуществляется до момента, когда модель фиксирует завершение фразы. Это позволяет избежать преждевременной обработки частично произнесённых команд.

Результат распознавания возвращается в виде строки. Данная строка передаётся в модуль обработки естественного языка.

Использование локального распознавания речи обладает следующими преимуществами:

− автономность - система функционирует без подключения к сети Интернет,

− низкая задержка - отсутствуют сетевые задержки,

− конфиденциальность - голосовые данные не передаются на внешние серверы,

− устойчивость - система работает независимо от качества соединения.

Несмотря на преимущества, использование локальной модели имеет ряд ограничений:

− более низкая точность по сравнению с облачными решениями,

− зависимость качества распознавания от акустических условий,

− ограниченные возможности адаптации модели без дополнительного обучения.

Модуль распознавания речи интегрирован в основной цикл работы системы и является первым этапом обработки команды.

Результат работы модуля передаётся в модуль классификации намерений, что обеспечивает дальнейшее преобразование текста в структурированную команду.

2.4 Модуль обработки естественного языка

Модуль обработки естественного языка предназначен для интерпретации текстовой команды пользователя и определения её семантического смысла в терминах заранее заданных классов намерений.

Задача данного модуля формализуется как задача многоклассовой классификации:

$$f: X \to Y, \tag{1}$$

где: X - множество текстовых команд,

Y - множество классов намерений (intent).

Для решения данной задачи используется нейросетевая модель, основанная на рекуррентной архитектуре, способной учитывать порядок слов и контекст входной последовательности.

Входная текстовая команда представляется в виде последовательности слов:

$$X = (w_1, w_2, \ldots, w_n). \quad (2)$$

Каждое слово преобразуется в числовой индекс на основе словаря:

$$w_i \rightarrow id_i \in N. \quad (3)$$

Для обеспечения фиксированной размерности входа последовательность дополняется до длины L:

$$X' = (id_1, id_2, \ldots, id_L), \quad (4)$$

где недостающие элементы заполняются нулями (padding).

Преобразование индексов в плотное векторное пространство осуществляется с использованием embedding-слоя:

$$e_i = W_e \times one_hot(id_i), \quad (5)$$

где: $W_e \in \mathbb{R}^{|V| \times d}$ - матрица эмбеддингов,

d - размерность векторного представления.

В результате формируется матрица:

$$E = (e_1, e_2, \ldots, e_L) \quad (6)$$

Данный подход позволяет учитывать семантическую близость слов.

Для обработки последовательности используется рекуррентная нейронная сеть типа GRU (Gated Recurrent Unit), обеспечивающая учёт контекста и зависимостей между словами.

```
class GRUIntentNet(nn.Module):
```

```
        def __init__(self, vocab_size: int, embedding_dim: int,
hidden_dim: int, output_dim: int):
            super().__init__()
                    self.embedding  =  nn.Embedding(vocab_size,
embedding_dim, padding_idx=0)
                  self.gru  =  nn.GRU(embedding_dim,  hidden_dim,
batch_first=True)
            self.classifier = nn.Sequential(
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, output_dim),
            )

        def forward(self, x):
            embedded = self.embedding(x)
            _, hidden = self.gru(embedded)
            last_hidden = hidden[-1]
            return self.classifier(last_hidden)
```

Обновление скрытого состояния описывается следующим выражением:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t, \qquad (7)$$

где: h_t - текущее скрытое состояние,

z_t - обновляющий гейт,

$\tilde{h}_t$ - кандидатное состояние.

GRU позволяет эффективно моделировать последовательности без значительного увеличения вычислительной сложности.

Выбор GRU вместо более сложных архитектур (например, LSTM или Transformer) обусловлен следующими факторами:

- меньшая вычислительная сложность,
- высокая скорость работы,
- достаточная точность для коротких команд,

– возможность работы в условиях ограниченных ресурсов.

Полученное скрытое состояние h передаётся в полносвязный слой:

$$y = Wh + b, \tag{8}$$

После применения функции Softmax вычисляется вероятность принадлежности к каждому классу:

$$P(y_i) = \frac{e^{y_i}}{\sum_j e^{y_i}}, \tag{9}$$

Итоговое намерение определяется как:

$$Intent = arg\,max\, P(y_i), \tag{10}$$

Дополнительно вычисляется уровень уверенности модели:

$$Confidence = max\, P(y_i), \tag{11}$$

Обучение модели осуществляется на основе набора данных, содержащего пары:

$$(x_i, y_i), \tag{12}$$

где x_i - текст команды, y_i - соответствующий класс.

В качестве функции потерь используется кросс-энтропия:

$$L = -\sum_{i=1}^{C} y_i log(\widehat{y_i}), \tag{13}$$

Оптимизация параметров осуществляется с использованием алгоритма Adam:

$$\theta = \theta - \eta \times \nabla L. \quad (14)$$

Обучение выполняется итеративно до достижения заданной точности или стабилизации функции потерь (представлено ниже на листинге).

```
        def train(self, force=False):
            if not force and self.load_from_cache():
                return

            print("[NN] Обучаю GRU intent-модель...")
            self.build_vocab()

                                                    x_data      =
np.stack([self.text_to_sequence(item["phrase"])   for   item   in
self.intent_data])
             y_data = np.array([INTENT_TO_IDX[item["intent"]] for
item in self.intent_data], dtype=np.int64)

            x_tensor = torch.from_numpy(x_data)
            y_tensor = torch.from_numpy(y_data)

                      self.model  =  GRUIntentNet(self.vocab_size,
EMBEDDING_DIM, HIDDEN_DIM, len(INTENTS))

            criterion = nn.CrossEntropyLoss()
            optimizer = torch.optim.Adam(self.model.parameters(),
lr=LEARNING_RATE)

            self.model.train()
            for _ in range(EPOCHS):
                optimizer.zero_grad()
                outputs = self.model(x_tensor)
```

```
                loss = criterion(outputs, y_tensor)
                loss.backward()
                optimizer.step()
        self.model.eval()
        self.save_to_cache()
```

Для повышения производительности реализован механизм кэширования обученной модели.

После завершения обучения сохраняются:

- веса модели,
- словарь,
- метаданные.

При последующих запусках системы выполняется проверка актуальности данных, и при отсутствии изменений модель загружается из кэша без повторного обучения.

Реализованная модель обладает следующими особенностями:

- компактная архитектура, оптимизированная для коротких текстов,
- возможность дообучения при добавлении новых команд,
- высокая скорость инференса (порядка миллисекунд),
- устойчивость к вариативности формулировок.

Дополнительно реализован механизм обработки низкой уверенности, при котором система запрашивает подтверждение у пользователя.

2.5 Гибридный механизм интерпретации команд

Несмотря на использование нейросетевой модели для классификации пользовательских намерений, одной только модели недостаточно для устойчивой работы голосового помощника в реальных условиях. Это связано с тем, что входной текст формируется после этапа автоматического распознавания речи, а значит может содержать ошибки, неточные словоформы, пропуски или искажения слов.

Кроме того, компактная нейросетевая модель, выбранная для обеспечения низких вычислительных требований, не обладает таким уровнем языкового обобщения, как крупные языковые модели или Transformer-

архитектуры. Поэтому в разработанной системе используется гибридный подход, сочетающий нейросетевую классификацию намерений с алгоритмической обработкой естественного языка.

Основными компонентами гибридного механизма являются:

- извлечение сущностей на основе правил,
- морфологическая нормализация словоформ,
- проверка допустимости глагольной конструкции,
- обработка низкой уверенности модели,
- fallback-механизм для перехода к альтернативному сценарию обработки,
- механизм уточнения и самообучения.

2.5.1 Правило-ориентированное извлечение сущностей

После определения намерения пользователя система должна выделить параметры команды, необходимые для её выполнения. Например, для команды «Открой браузер» нейросетевая модель определяет намерение open_target, а модуль извлечения сущностей выделяет параметр target_name = браузер.

В разработанной системе извлечение сущностей реализовано на основе правил. Такой подход выбран потому, что набор поддерживаемых команд ограничен предметной областью голосового управления персональным компьютером, а количество типов параметров относительно невелико.

Использование rule-based подхода позволяет снизить требования к обучающей выборке, так как нейросети не требуется дополнительно обучаться задаче последовательной разметки токенов. Это особенно важно в рамках разрабатываемой системы, где одним из ключевых требований является низкая вычислительная сложность.

2.5.2 Морфологическая нормализация

Одной из проблем обработки русскоязычных голосовых команд является наличие большого количества словоформ. Пользователь может произнести команду в разных вариантах. Для решения данной проблемы в

системе используется морфологическая нормализация. Она позволяет приводить слова к нормальной форме

Такой подход позволяет системе не обучать каждую словоформу отдельно, а обрабатывать различные варианты одной команды как близкие по смыслу.

Морфологическая нормализация особенно важна для fallback-механизма, поскольку позволяет корректно сравнивать глаголы и существительные независимо от их формы в пользовательской фразе.

2.5.3 Проверка глагольной конструкции

Одной из проблем, выявленных при тестировании, является возможность ошибочной классификации команды нейросетевой моделью. Фраза может быть ошибочно классифицирована если модель недостаточно хорошо обобщила обучающие примеры.

Для предотвращения подобных ошибок в системе реализована проверка первого значимого слова команды, которое рассматривается как потенциальный управляющий глагол. Данный глагол сравнивается со списком известных действий, предварительно приведённых к нормальной форме.

Если глагол не соответствует ни одному известному действию, система не выполняет команду автоматически. Вместо этого команда переводится в режим fallback-обработки.

Такой подход повышает безопасность системы, поскольку предотвращает выполнение потенциально ошибочно интерпретированных команд.

2.5.4 Обработка низкой уверенности модели

Нейросетевая модель возвращает не только класс намерения, но и значение уверенности (confidence). Это значение соответствует максимальной вероятности среди всех классов после применения функции Softmax.

Если уверенность модели ниже заданного порога то система не должна безусловно выполнять действие. Вместо этого возможны следующие варианты:

- запрос подтверждения у пользователя,
- перевод в fallback-режим,
- запуск сценария обучения новой команды,
- игнорирование команды при недостаточном уровне уверенности.

Например, если модель классифицировала команду как open_target, но значение confidence оказалось низким, система может запросить уточнение, а не выполнять неправильную команду.

Такой механизм особенно важен для автономной системы, поскольку она не использует облачные модели с более высокой языковой обобщающей способностью. Вместо увеличения размера модели используется дополнительная логика проверки результата.

2.5.5 Fallback-механизм

Fallback-механизм предназначен для обработки ситуаций, когда результат нейросетевой классификации является ненадёжным или противоречит структуре команды.

```
            command = self.analyze(text)
                        from assistant.fallback_engine import
apply_contextual_fallback

            command = apply_contextual_fallback(
                command,
                self.actions_config,
                self.synonyms_config
            )

            if command.intent == "none":
                if len(command.text.split()) >= 2:
                                                    reply =
self.executor.start_auto_learning(command.text)
```

```
            else:
                reply = "Я не понял команду"

            self.tts.say(reply)
            return
```

Механизм fallback активируется в следующих случаях:

- модель определила неизвестное или неподходящее намерение,
- глагол команды не соответствует известным действиям,
- confidence ниже установленного порога,
- команда содержит неизвестную цель,
- результат классификации может привести к некорректному действию.

В таких случаях система не завершает обработку ошибкой, а переходит к альтернативному сценарию. Например, если пользователь говорит какое-то предложение и система не знает ни глагол, ни цель, ассистент может предложить пользователю открыть нужную программу вручную. После этого система сохранит новую цель и свяжет неизвестный глагол с соответствующим действием.

Таким образом, fallback-механизм выполняет функции предотвращения ошибочного выполнение команд и обеспечивает переход к самообучению системы.

2.5.6 Преимущества гибридного подхода

Использование гибридной архитектуры позволяет компенсировать ограничения компактной нейросетевой модели без перехода к ресурсоёмким языковым моделям.

К основным преимуществам данного подхода относятся:

- снижение требований к вычислительным ресурсам,
- повышение устойчивости к ошибкам распознавания речи,
- возможность обработки различных словоформ,
- снижение риска ошибочного выполнения команд,

− возможность адаптации системы к индивидуальным формулировкам пользователя.

В отличие от подходов, основанных исключительно на крупных языковых моделях, предложенное решение не требует значительного объёма оперативной памяти и может функционировать на обычном персональном компьютере без использования GPU.

2.6 Механизм самообучения системы

Одной из ключевых особенностей разработанного голосового помощника является возможность расширения функциональности в процессе эксплуатации без изменения исходного кода. Данный механизм позволяет системе адаптироваться к индивидуальным сценариям пользователя, запоминать новые программы, новые способы формулировки команд и пользовательские макросы.

Необходимость самообучения обусловлена тем, что заранее невозможно предусмотреть полный набор приложений, путей, пользовательских привычек и вариантов речевых команд. Например, разные пользователи могут использовать различные браузеры, мессенджеры, игры, редакторы и служебные программы, расположенные по разным путям в файловой системе. Жёсткое задание всех возможных программ в исходном коде сделало бы систему негибкой и плохо переносимой между компьютерами.

Поэтому в разработанной системе используется конфигурационный и обучаемый подход. Основные данные о доступных действиях хранятся во внешнем файле actions.json, а обучающие фразы для классификатора намерений в intents.json. При добавлении новой команды система обновляет соответствующие конфигурационные файлы и при необходимости выполняет повторное обучение модели классификации намерений.

```
def finish_auto_learning(self):
    if not self.pending_auto_learn:
        return "Мне пока нечего запоминать."
```

```
            text = self.pending_auto_learn["text"]
            parts = text.split(maxsplit=1)

            if len(parts) == 2:
                verb, target = parts
            else:
                verb = None
                target = text

            new_app = get_active_exe_path()

            if not new_app:
                        return "Не удалось определить активную
программу."

            self.actions_config.setdefault("targets", {})[target]
= {
                "mode": "path",
                "value": new_app,
                "aliases": []
            }

            save_actions(self.actions_config)

            if verb:
                 self.synonyms_config.setdefault("learned_verbs",
{})[verb] = "open_target"
                save_synonyms(self.synonyms_config)
                self.intent_classifier.add_training_phrase(f"{ver
b} {target}", "open_target")

             self.intent_classifier.add_open_target_training(targ
et)
            save_intents(self.intent_classifier.intent_data)
```

```
            self.pending_auto_learn = None

            if verb:
                  return f"Запомнил {target} и понял, что {verb}
означает открыть."

            return f"Запомнил {target}."
```

Обобщённо процесс самообучения можно представить следующим образом:

Неизвестная команда → уточнение у пользователя → выполнение действия вручную → получение данных о действии → сохранение в конфигурации → обновление обучающей выборки → переобучение модели

Таким образом, система не ограничивается заранее заданным набором команд, а может расширяться в процессе использования.

2.6.1 Обучение новым программам

Первым направлением самообучения является добавление новых приложений. Если пользователь произносит команду открытия объекта, отсутствующего в конфигурации, система не завершает обработку ошибкой, а предлагает пользователю выполнить действие вручную.

Если цель отсутствует в actions.json, ассистент сообщает, что пока не знает данную программу, и предлагает открыть её вручную. После того как пользователь запускает нужное приложение и подтверждает действие командой.

Система определяет активный исполняемый файл, сохраняет путь к нему и связывает его с указанным пользователем названием.

В результате в конфигурационном файле может быть создана запись следующего вида:

```
{
  "mode": "path",
  "value": "C:\\Program Files\\Game\\game.exe",
```

```
  "aliases": []
}
```

После этого команда будет выполняться автоматически.

Такой подход позволяет избежать жёсткого задания путей к программам. Это особенно важно для Windows-систем, где одно и то же приложение может быть установлено в разные директории в зависимости от пользователя, версии программы и способа установки.

2.6.2 Обучение неизвестным глаголам

Вторым направлением самообучения является обработка неизвестных управляющих глаголов. Пользователь может использовать не только стандартные формулировки, но и разговорные или индивидуальные выражения:

Если система обнаруживает, что объект команды известен, но управляющий глагол отсутствует в списке известных действий, она может инициировать уточнение.

После подтверждения система сохраняет новый глагол как синоним действия. Например, в конфигурации может быть добавлена запись:

```
"learned_verbs": {
  "стартани": "open_target"
}
```

После этого данный глагол может использоваться не только с объектом игра, но и с другими известными целями.

Таким образом, система обучается не только конкретной фразе, но и более общему языковому шаблону. Это повышает гибкость обработки естественного языка и снижает необходимость заранее вручную перечислять все возможные варианты команд.

2.6.3 Обучение пользовательским макросам

Третьим направлением самообучения является запись пользовательских макросов. Макрос представляет собой последовательность действий, которая может быть сохранена и затем выполнена по голосовой команде.

Например, пользователь может произнести:

компьютер научись действию следующее окно

После этого система переходит в режим записи. Пользователь вручную выполняет нужное действие, например нажимает сочетание клавиш Alt + Tab.

Система сохраняет записанную последовательность действий в конфигурации:

```
"следующее окно": {
  "mode": "macro",
  "value": [
    {
      "mode": "hotkey",
      "value": ["alt", "tab"]
    }
  ],
  "aliases": []
}
```

После этого пользователь может выполнить данное действие с помощью голоса.

Механизм макросов позволяет расширять функциональность помощника. При этом пользователю не требуется изменять исходный код программы: достаточно один раз продемонстрировать действие системе.

2.6.4 Обновление обучающей выборки

После добавления новой цели или нового глагола система обновляет обучающую выборку. Для этого в файл intents.json добавляются новые пары. Дополнительно могут автоматически генерироваться варианты команд с разными управляющими глаголами.

Это позволяет увеличить разнообразие обучающих примеров и повысить устойчивость нейросетевой модели к различным формулировкам.

После обновления обучающей выборки выполняется повторное обучение модели классификации намерений. Новые веса модели сохраняются в кэш, благодаря чему при следующем запуске ассистент

использует уже обновлённую модель без необходимости повторного обучения.

2.6.5 Преимущества механизма самообучения

Реализованный механизм самообучения обеспечивает несколько важных преимуществ.

Во-первых, система становится переносимой между разными компьютерами. Пользователь может обучить ассистента тем программам, которые установлены именно на его устройстве.

Во-вторых, повышается гибкость взаимодействия. Пользователь не обязан строго придерживаться заранее заданных фраз, поскольку система может запоминать новые варианты формулировок.

В-третьих, самообучение снижает зависимость от разработчика. Расширение функциональности происходит без изменения исходного кода и без повторной сборки приложения.

В-четвёртых, самообучение повышает практическую применимость системы. Ассистент постепенно адаптируется к индивидуальному стилю пользователя и становится более удобным в повседневном использовании.

2.7 Сравнительный анализ разработанной модели и альтернативных подходов

При проектировании модуля обработки естественного языка одной из ключевых задач являлся выбор архитектуры, обеспечивающей достаточную точность классификации пользовательских команд при низких требованиях к вычислительным ресурсам. Данное требование особенно важно для автономного голосового помощника, поскольку система должна функционировать локально, без обращения к облачным сервисам и без обязательного использования графического ускорителя.

Для решения задачи классификации намерений пользователя могут применяться различные подходы:

- методы на основе Bag-of-Words,
- рекуррентные нейронные сети LSTM,

− рекуррентные нейронные сети GRU,

− Transformer-архитектуры,

− крупные языковые модели.

Каждый из данных подходов обладает собственными преимуществами и ограничениями. Поэтому выбор архитектуры должен осуществляться не только на основе потенциальной точности, но и с учётом вычислительной сложности, скорости инференса, объёма потребляемой памяти и применимости в офлайн-режиме.

2.7.1 Сравнение с Bag-of-Words

Одним из наиболее простых подходов к представлению текста является модель Bag-of-Words. В данном случае текстовая команда рассматривается как набор слов без учёта их порядка.

Например, команды открой браузер и браузер открой для модели Bag-of-Words будут иметь практически одинаковое представление, поскольку учитывается только факт наличия слов, но не их последовательность.

Преимуществами данного подхода являются простота реализации, высокая скорость работы и минимальные требования к памяти. Однако для задачи голосового управления персональным компьютером такой подход является недостаточно гибким, поскольку смысл команды может зависеть от порядка слов, наличия отрицания и контекста.

Например, фразы открой браузер и не открой браузер содержат похожий набор слов, однако выражают разные намерения. При использовании Bag-of-Words подобные различия могут быть обработаны некорректно.

В отличие от Bag-of-Words, используемая в работе GRU-модель обрабатывает команду как последовательность слов. Это позволяет учитывать порядок токенов и формировать скрытое состояние, отражающее общий смысл фразы.

Таким образом, несмотря на более низкие требования к ресурсам, Bag-of-Words уступает рекуррентным моделям по качеству интерпретации команд на естественном языке.

2.7.2 Сравнение с LSTM

LSTM является одной из наиболее известных рекуррентных архитектур и предназначена для обработки последовательных данных. В отличие от обычных рекуррентных сетей, LSTM содержит специальные механизмы памяти, позволяющие учитывать долгосрочные зависимости.

Данная архитектура может быть эффективна при обработке длинных текстов, сложных предложений и последовательностей, в которых важна информация, находящаяся далеко от текущего слова.

Однако в разрабатываемой системе пользовательские команды, как правило, являются короткими.

В таких условиях необходимость в более сложной памяти LSTM снижается. При этом LSTM содержит больше параметров и требует больше вычислений по сравнению с GRU.

GRU является более компактной рекуррентной архитектурой. Она также учитывает последовательность слов, но использует меньше внутренних управляющих механизмов. За счёт этого GRU быстрее обучается, требует меньше памяти и обеспечивает более низкую задержку при инференсе.

Для рассматриваемой задачи снижение точности по сравнению с LSTM является незначительным, поскольку команды имеют небольшую длину и ограниченный набор классов намерений. Поэтому использование GRU является более рациональным с точки зрения соотношения точности и вычислительной эффективности.

2.7.3 Сравнение с Transformer-архитектурами

Transformer-архитектуры широко применяются в современных системах обработки естественного языка. Их основным преимуществом

является механизм self-attention, позволяющий модели анализировать взаимосвязи между всеми словами входной последовательности.

Благодаря этому Transformer хорошо подходит для обработки длинных текстов, сложного контекста и диалоговых сценариев. Однако за повышение качества приходится платить увеличением вычислительной сложности и требований к памяти.

Для рекуррентной модели сложность обработки последовательности длины n имеет линейный характер относительно длины входа - O(n).

Для Transformer-модели механизм self-attention требует сопоставления каждого токена с каждым другим токеном, что приводит к квадратичной зависимости - $O(n^2)$.

Для коротких команд данное отличие может казаться незначительным, однако даже компактные Transformer-модели содержат существенно больше параметров, чем простая GRU-сеть. Это увеличивает объём памяти, время загрузки модели, нагрузку на центральный процессор и задержку обработки команды.

В рамках разрабатываемого голосового помощника основными требованиями являются:

- автономная работа,
- низкое потребление ресурсов,
- быстрый отклик,
- возможность работы без GPU.
- простота переобучения при добавлении новых команд

С этой точки зрения использование Transformer-архитектуры является избыточным. Для коротких пользовательских команд GRU обеспечивает достаточный уровень точности, но при этом требует существенно меньше памяти и вычислительных ресурсов.

2.7.4 Сравнение с крупными языковыми моделями

Крупные языковые модели обладают наиболее широкими возможностями обработки естественного языка. Они способны поддерживать

диалог, учитывать контекст, формировать ответы и интерпретировать различные непонятные пользовательские запросы.

Однако такие модели имеют очень высокие требования к вычислительным ресурсам. Даже при локальном запуске компактных версий LLM требуется значительный объём оперативной памяти, а для комфортной работы часто необходим графический ускоритель. Кроме того, инференс крупных моделей может занимать значительно больше времени по сравнению с компактными классификаторами.

Для задачи голосового управления персональным компьютером использование LLM не является оптимальным, поскольку система не должна генерировать сложные тексты или вести диалог. Основная задача заключается в классификации короткой команды и выполнении соответствующего действия.

Поэтому LLM привело бы к слишком большому росту требований к ресурсам при относительно небольшом практическом выигрыше для выбранной предметной области.

Разработанное же решение ориентировано на выполнение ограниченного, но практически полезного набора действий с высокой скоростью и низкой нагрузкой на систему.

2.7.5 Расчёт параметров разработанной модели

Для оценки компактности разработанной модели рассмотрим примерную конфигурацию нейросети:

- размер словаря = 3000,
- размерность embedding-вектора = 64,
- размер скрытого состояния = 128,
- количество классов намерений = 15.

При использовании 32-битных чисел с плавающей точкой один параметр занимает 4 байта. Следовательно, объём памяти для хранения весов модели составляет $284559 \cdot 4 \approx 1\,138\,236$ байт $\approx 1{,}1$ МБ

Даже с учётом дополнительных служебных структур, словаря и метаданных размер модели остаётся небольшим. Это позволяет быстро загружать модель, выполнять инференс на CPU и использовать систему на компьютерах без специализированного оборудования.

2.7.6 Итоговое сравнение подходов

Для обобщения результатов анализа приведём сравнительную таблицу.

Таблица 2 - Сравнение подходов

Подход	Учёт порядка слов	Требования к памяти	Нагрузка на CPU	Скорость инференса	Точность для коротких команд	Применимость в офлайн-системе
Bag-of-Words	Нет	Очень низкие	Очень низкая	Очень высокая	Средняя	Высокая
LSTM	Да	Средние	Средняя	Средняя	Высокая	Средняя
GRU	Да	Низкие	Низкая	Высокая	Высокая	Высокая
Transformer	Да	Высокие	Высокая	Средняя/низкая	Очень высокая	Ограниченная
LLM	Да	Очень высокие	Очень высокая	Низкая без GPU	Максимальная	Низкая

Из таблицы видно, что GRU занимает промежуточное положение между простыми методами и тяжёлыми языковыми моделями. По сравнению с Bag-of-Words она обеспечивает более качественную интерпретацию команд за счёт учёта последовательности слов. По сравнению с Transformer и LLM она требует значительно меньше ресурсов, сохраняя достаточную точность для коротких команд голосового управления.

2.8 Вывод

Во второй главе была разработана архитектура голосового помощника для персонального компьютера, предназначенного для обработки голосовых команд без использования интернета. Предложенная система построена по модульному принципу, что обеспечивает разделение функций между компонентами и упрощает дальнейшее расширение приложения.

В рамках разработки были выделены основные функциональные модули системы: модуль распознавания речи, модуль обработки естественного языка, модуль извлечения сущностей, исполнительный модуль, модуль выполнения действий, механизм самообучения и графический интерфейс пользователя. Это разделение позволило реализовать последовательный процесс обработки команды от получения голоса до выполнения действия в операционной системе.

Для преобразования речевого сигнала в текст используется локальное средство распознавания речи, что обеспечивает автономность работы системы и отсутствие необходимости передачи пользовательских данных на внешние серверы. Это соответствует одному из ключевых требований разрабатываемого решения - независимости от соединения с интернетом.

Основным компонентом интерпретации пользовательских команд является нейросетевая модель классификации намерений, построенная на основе рекуррентной архитектуры GRU. Данная модель принимает текстовую команду, преобразует её в последовательность токенов, формирует векторные представления слов с помощью embedding-слоя и определяет класс намерения пользователя. Использование GRU позволяет учитывать порядок слов и контекст коротких команд при сохранении низкой вычислительной сложности.

Для обучения модели применяется функция потерь кросс-энтропии, а оптимизация параметров выполняется с использованием алгоритма Adam. После обучения модель сохраняется в кэш, что позволяет избежать повторного обучения при каждом запуске приложения и снизить время инициализации системы.

Дополнительно в системе реализован гибридный механизм интерпретации команд, включающий правило-ориентированное извлечение сущностей, морфологическую нормализацию, проверку глагольных конструкций и fallback-обработку. Данный механизм позволяет компенсировать ограничения компактной нейросетевой модели и повысить

устойчивость системы к ошибкам распознавания речи, вариативности формулировок и неизвестным словам.

Важным элементом разработанной архитектуры является механизм самообучения. Он позволяет пользователю добавлять новые приложения, обучать систему неизвестным управляющим глаголам и записывать пользовательские макросы. Благодаря этому функциональность помощника может расширяться в процессе эксплуатации без изменения исходного кода.

Сравнительный анализ показал, что выбранная архитектура является оптимальной между точностью и вычислительной эффективностью. По сравнению с простыми методами, такими как Bag-of-Words, она лучше учитывает последовательность слов и контекст команды. По сравнению с LSTM, Transformer и крупными языковыми моделями, она требует значительно меньше памяти и вычислительных ресурсов, что особенно важно для автономной офлайн-системы.

Таким образом, во второй главе была обоснована и реализована архитектура голосового помощника, сочетающая локальное распознавание речи, компактную нейросетевую модель, алгоритмические методы обработки естественного языка и механизм самообучения. Полученная система ориентирована на достижение баланса между точностью интерпретации команд, скоростью работы и низкими требованиями к ресурсам.

Результаты, полученные в данной главе, создают основу для дальнейшей экспериментальной оценки разработанного решения. В следующей главе необходимо провести сравнительный анализ качества и производительности системы, включая оценку точности классификации намерений, времени инференса, потребления памяти и загрузки центрального процессора.

Весь процесс взаимодействия модулей и пользователя изображён на рисунке 2.

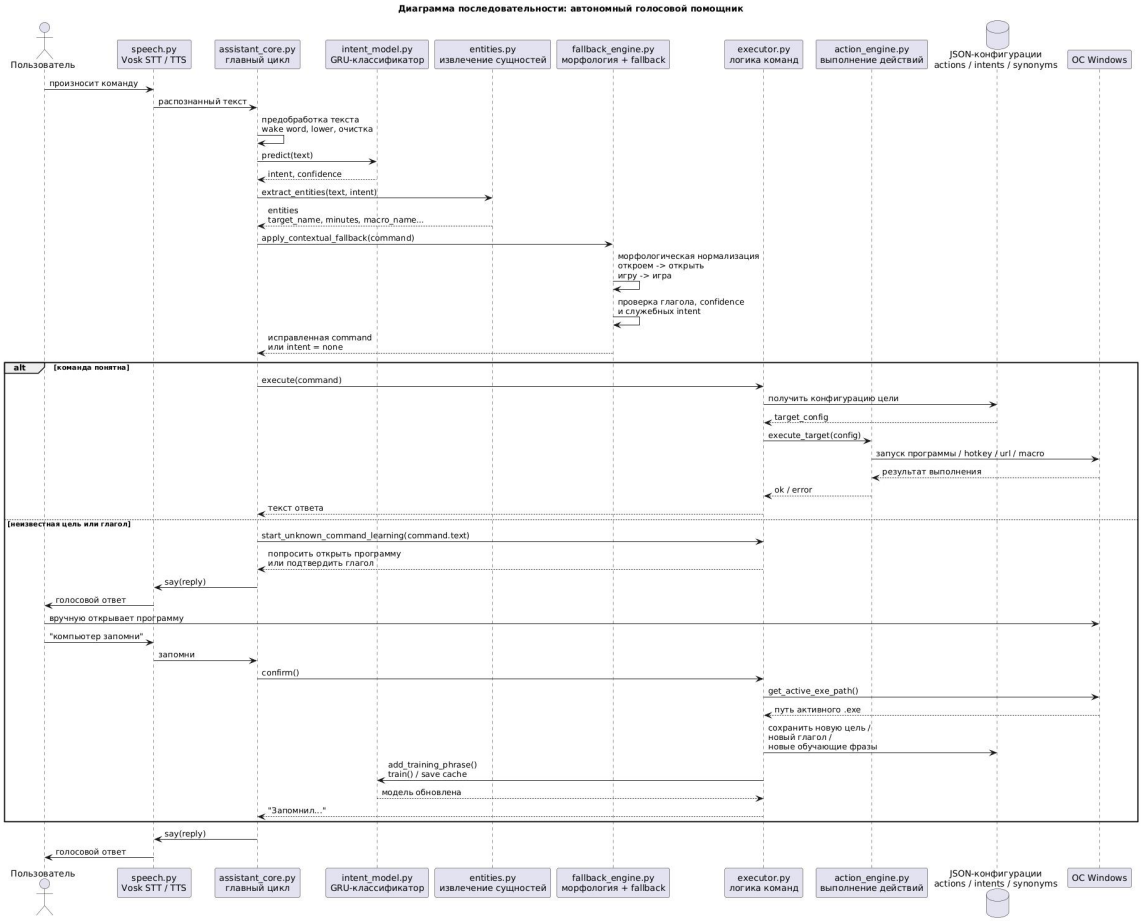


Рисунок 2 - Диаграмма последовательности

3 Экспериментальная оценка и сравнительный анализ разработанного решения

3.1 Цель и методика экспериментального исследования

Целью экспериментального исследования является оценка эффективности разработанного голосового помощника и подтверждение целесообразности выбранной архитектуры для автономной работы на персональном компьютере.

Основное внимание в экспериментальной части уделяется сравнению разработанного решения с альтернативными подходами к интерпретации пользовательских команд. В рамках исследования необходимо показать, что предложенная GRU-модель в сочетании с гибридным механизмом обработки команд обеспечивает достаточную точность классификации намерений при существенно меньших требованиях к вычислительным ресурсам по сравнению с более сложными архитектурами.

Экспериментальная оценка проводится по следующим направлениям:

- оценка точности классификации пользовательских намерений,
- анализ качества работы модели на различных типах команд,
- сравнение разработанной GRU-модели с альтернативными подходами,
- измерение времени инференса,
- оценка потребления оперативной памяти,
- анализ загрузки центрального процессора,
- исследование влияния fallback-механизма и морфологической нормализации,
- оценка эффективности механизма самообучения.

В качестве сравниваемых подходов рассматриваются следующие варианты:

- модель на основе Bag-of-Words,
- рекуррентная модель LSTM,
- разработанная модель на основе GRU,

− Transformer-архитектура,

− LLM.

Такой набор подходов позволяет сопоставить разработанное решение как с простыми классическими методами обработки текста, так и с современными ресурсоёмкими архитектурами. Bag-of-Words используется как базовый вариант, обладающий минимальными вычислительными требованиями, но не учитывающий порядок слов. LSTM рассматривается как более сложная рекуррентная альтернатива GRU. Transformer используется как современная архитектура, которая обеспечивает высокое качество обработки текста за счёт механизма self-attention, но обладает более высокими требованиями к памяти и вычислительным ресурсам.

Крупные языковые модели рассматриваются как наиболее точный, но наиболее ресурсоёмкий класс решений. Они способны обеспечивать высокое качество понимания языка и поддерживать диалог, но их использование в автономном голосовом помощнике для персонального компьютера связано с существенными ограничениями: большим объёмом модели, высокой нагрузкой на CPU или необходимостью GPU, увеличенным временем инференса и сложностью локального развёртывания.

В связи с этим LLM рассматриваются в работе преимущественно в качестве сравнительного анализа требований к ресурсам и практической применимости, а не как основной экспериментальный вариант обучения. Такой подход позволяет показать, что для задачи коротких голосовых команд использование крупной языковой модели является избыточным, тогда как компактная GRU-модель обеспечивает достаточную точность при существенно меньших вычислительных затратах.

Эксперимент проводится на наборе пользовательских команд, отражающих основные сценарии взаимодействия с голосовым помощником. В набор включаются команды открытия приложений, управления громкостью, открытия папок, создания напоминаний, выполнения макросов и обработки неизвестных формулировок.

Для оценки качества классификации используются стандартные метрики машинного обучения: accuracy, precision, recall и F1-score. Для оценки производительности используются время инференса, потребление оперативной памяти и средняя загрузка центрального процессора.

Данные метрики позволяют оценить не только общую точность модели, но и качество классификации по отдельным типам команд. Это важно, поскольку разные intent-классы могут иметь различную сложность: например, команды открытия приложений могут быть проще для классификации, чем команды самообучения или макросов.

Отдельное внимание уделяется производительности. Поскольку система разрабатывается как автономное решение с низкими требованиями к ресурсам, высокая точность модели не является единственным критерием качества. Не менее важны скорость отклика, потребление памяти и нагрузка на процессор.

В рамках эксперимента фиксируются следующие показатели:

- среднее время инференса одной команды,
- объём оперативной памяти, используемый моделью,
- средняя и пиковая загрузка CPU во время обработки команды,
- размер сохранённой модели,
- время загрузки модели из кэша.

Дополнительно проводится анализ влияния гибридного механизма обработки команд. Для этого сравниваются результаты работы системы до и после включения fallback-механизма и морфологической нормализации. Такие команды позволяют проверить устойчивость системы к вариативности пользовательской речи, неизвестным глаголам и различным словоформам.

Таким образом, экспериментальная методика направлена не только на измерение точности нейросетевой модели, но и на комплексную оценку разработанного решения как автономной программной системы. Это позволяет показать, что выбранная архитектура обеспечивает рациональный

баланс между качеством интерпретации команд, скоростью работы и низким потреблением вычислительных ресурсов.

3.2 Описание экспериментального набора данных

Для проведения экспериментальной оценки был сформирован набор пользовательских команд, отражающих основные сценарии взаимодействия с автономным голосовым помощником. Набор данных использовался для обучения и тестирования моделей классификации намерений, а также для проверки устойчивости системы к различным вариантам формулировок.

Поскольку разрабатываемая система предназначена для голосового управления персональным компьютером, в набор данных были включены команды, связанные с запуском приложений, управлением системными функциями, открытием папок, созданием напоминаний, вводом текста, выполнением пользовательских макросов и обработкой неизвестных фраз.

Каждая запись набора данных представляет собой пару <текст команды, класс намерения> где текст команды является входной фразой пользователя, а класс намерения определяет действие, которое должна выполнить система.

Использование такого формата данных позволяет рассматривать задачу интерпретации команды как задачу многоклассовой классификации, где каждому текстовому запросу сопоставляется один из заранее определённых классов намерений.

В экспериментальный набор были включены следующие основные классы:

Таблица 3 - Основные классы

Класс намерения	Назначение класса	Пример команды
open_target	запуск приложения или действия	открой браузер
delete_target	удаление сохранённой цели	забудь игру
open_folder	открытие папки	открой загрузки
volume_up	увеличение громкости	прибавь звук
volume_down	уменьшение громкости	сделай тише

Класс намерения	Назначение класса	Пример команды
volume_mute	отключение звука	выключи звук
type_text	ввод текста	напечатай привет
get_time	получение текущего времени	сколько времени
get_date	получение текущей даты	какое сегодня число
set_reminder	создание напоминания	напомни через 10 минут
switch_window	переключение окон	следующее окно
learn_macro	начало записи макроса	научись действию
finish_recording	завершение записи макроса	готово
confirm	подтверждение действия	запомни
reject	отмена действия	отмена
none	отсутствие поддерживаемой команды	случайная фраза

Особое внимание при формировании набора данных уделялось вариативности пользовательских формулировок. Для одного и того же намерения использовались разные варианты команд, отличающиеся глаголами, порядком слов и разговорными конструкциями.

Такой подход позволяет проверить, насколько модель способна обобщать разные формулировки одного и того же действия, а не просто запоминать конкретные фразы из обучающей выборки.

Отдельно в набор были включены команды с неизвестными или нестандартными формулировками. Они необходимы для оценки работы fallback-механизма, морфологической нормализации и самообучения системы.

Примеры таких команд:

- стартани игру,
- откроем браузер,
- давай включим телеграм,

− забудь старую игру.

Данные команды позволяют проверить, способна ли система корректно реагировать на неизвестный глагол, изменённую словоформу или новую пользовательскую цель.

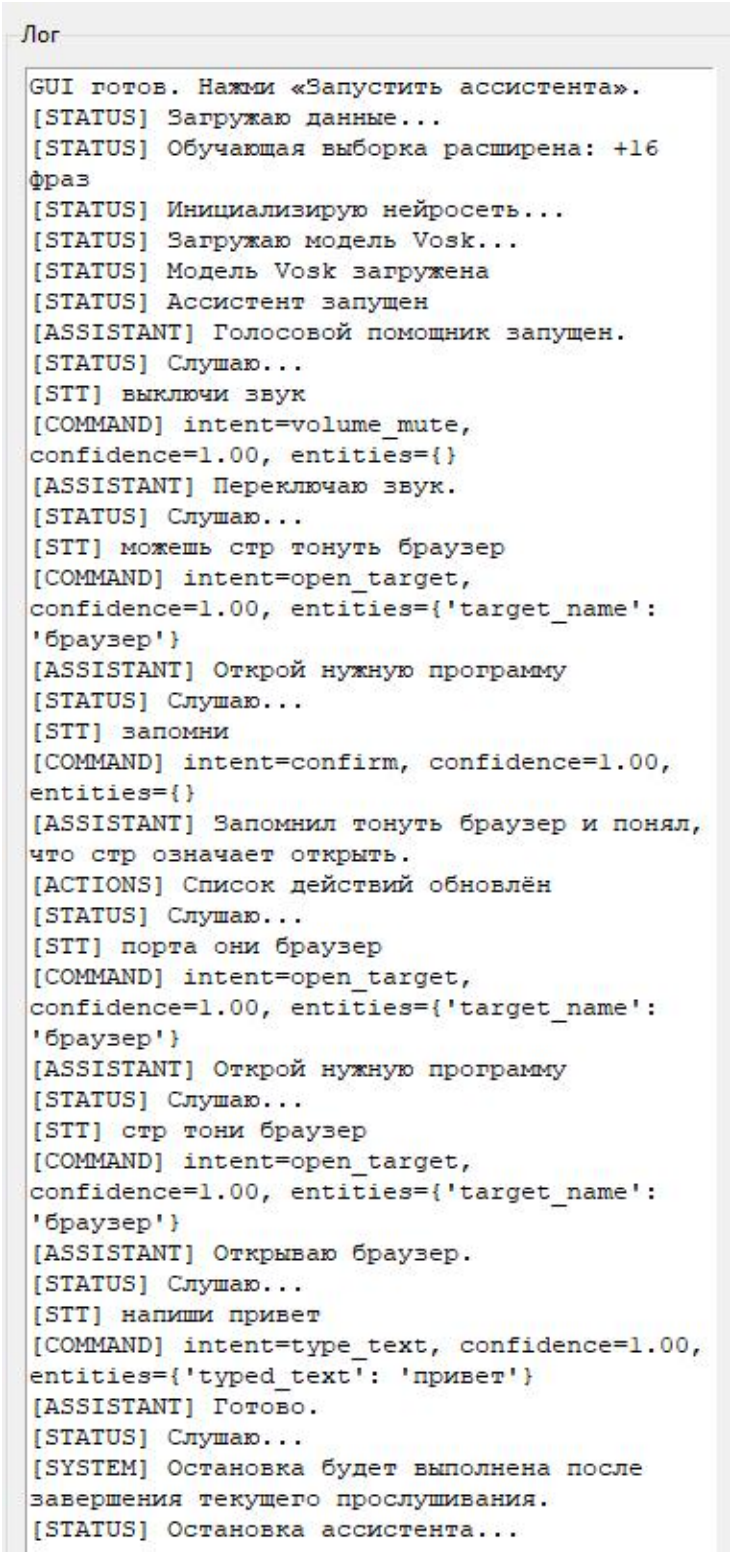


Рисунок 3 - Пример выполнения команд

Также в набор данных был включён класс none. Он предназначен для команд, которые не должны приводить к выполнению действия. Наличие данного класса особенно важно для предотвращения ложных срабатываний голосового помощника.

Добавление подобных примеров позволяет модели лучше отделять поддерживаемые команды от фраз, не относящихся к функциональности системы.

Для экспериментальной оценки набор данных был разделён на обучающую и тестовую части. Обучающая выборка использовалась для настройки параметров модели, а тестовая — для проверки качества классификации на примерах, которые модель не видела во время обучения.

Разделение данных позволяет оценить не только способность модели запоминать обучающие примеры, но и её способность к обобщению. Это особенно важно для голосового помощника, так как пользователь может формулировать одну и ту же команду по-разному.

Для дальнейшего сравнительного анализа один и тот же набор команд используется для тестирования разных моделей: Bag-of-Words, LSTM, GRU, Transformer и LLM-подхода. Это необходимо для корректного сопоставления результатов, поскольку различия в точности, скорости и потреблении ресурсов должны быть обусловлены архитектурой модели, а не различиями в данных для тестирования.

Таким образом, экспериментальный набор данных отражает основные сценарии использования разработанного голосового помощника и позволяет оценить качество работы системы по нескольким направлениям: точность классификации намерений, устойчивость к вариативности формулировок, способность обрабатывать неизвестные команды и эффективность гибридных механизмов коррекции.

3.3 Результаты классификации намерений

На первом этапе экспериментальной оценки было проведено сравнение качества классификации пользовательских намерений различными моделями

обработки текста. Основной задачей данного эксперимента являлось определение того, насколько разработанная GRU-модель способна корректно определять intent-класс пользовательской команды по сравнению с более простыми и более сложными архитектурами.

Тестирование выполнялось на едином наборе команд, включающем основные сценарии взаимодействия с голосовым помощником: запуск приложений, управление громкостью, открытие папок, создание напоминаний, ввод текста, выполнение пользовательских макросов, подтверждение действий, отмену и обработку неподдерживаемых фраз.

Для сравнения использовались следующие подходы:

- Bag-of-Words,
- LSTM,
- GRU,
- GRU с гибридной обработкой,
- Transformer,
- LLM-подход как теоретико-практический ориентир.

Отдельное выделение варианта GRU с гибридной обработкой необходимо, поскольку в разработанной системе итоговое качество определяется не только нейросетевой моделью, но и дополнительными механизмами: морфологической нормализацией, fallback-обработкой, проверкой глагольных конструкций и уточнением неизвестных команд.

3.3.1 Общие результаты классификации

Общие результаты классификации намерений представлены в таблице.

Таблица 4 - Результаты классификации

Модель	Accuracy	Precision	Recall	F1-score
Bag-of-Words	0,78	0,79	0,76	0,77
LSTM	0,90	0,90	0,89	0,89
GRU	0,89	0,90	0,88	0,89
GRU +fallback	0,92	0,93	0,91	0,92

Модель	Accuracy	Precision	Recall	F1-score
Transformer	0,94	0,94	0,93	0,94
LLM	0,96	0,96	0,95	0,95

Данные результаты показывают, что простая модель Bag-of-Words уступает нейросетевым архитектурам по всем основным параметрам. Это связано с тем, что данный подход не учитывает порядок слов и контекст команды, что ограничивает его применимость при обработке естественного языка.

Модели LSTM и GRU показывают близкие результаты. Незначительное преимущество LSTM по отдельным метрикам объясняется более сложной архитектурой и наличием расширенного механизма памяти. Однако для коротких голосовых команд преимущество LSTM оказывается небольшим, поскольку большинство команд состоит из нескольких слов и не требует анализа длинного контекста.

Разработанная GRU-модель демонстрирует качество, близкое к LSTM, но при этом обладает меньшей вычислительной сложностью. Это подтверждает целесообразность выбора GRU в качестве основной архитектуры для автономного голосового помощника.

Наилучший результат среди данных вариантов показывает GRU с гибридной обработкой. Рост качества объясняется тем, что ошибки нейросетевой классификации частично компенсируются дополнительными механизмами: fallback, морфологической нормализацией и проверкой структуры команды.

Transformer и LLM демонстрируют более высокие значения метрик, однако их преимущество по точности не является радикальным для рассматриваемой предметной области. При этом данные подходы требуют значительно больше вычислительных ресурсов, что будет показано в последующих разделах.

3.4 Сравнение вычислительных ресурсов

Помимо качества классификации намерений, важным критерием оценки голосового помощника является вычислительная эффективность. Для автономной системы, работающей локально на персональном компьютере, недостаточно обеспечить только высокую точность распознавания команд. Модель также должна иметь низкое время отклика, умеренное потребление оперативной памяти и минимальную нагрузку на центральный процессор.

Это особенно важно, поскольку голосовой помощник предполагается использовать в фоновом режиме. Следовательно, он не должен существенно снижать производительность компьютера, мешать работе других приложений или требовать специализированного оборудования.

В рамках исследования сравнение моделей выполнялось по следующим показателям:

- размер модели на диске,
- объём оперативной памяти после загрузки модели,
- среднее время инференса одной команды,
- средняя загрузка CPU при обработке команды,
- пиковая загрузка CPU,
- возможность работы без GPU.

3.4.1 Размер модели и потребление памяти

Первым показателем вычислительной эффективности является размер модели и объём оперативной памяти, необходимый для её работы.

Разработанная GRU-модель имеет компактную архитектуру. Она включает embedding-слой, один GRU-слой и небольшой полносвязный классификатор. Как было показано ранее, при размере словаря около 3000 слов, размерности embedding-вектора 64 и размере скрытого состояния 128 модель содержит примерно 285 тысяч параметров.

При использовании 32-битного представления чисел объём памяти, необходимый для хранения весов, составляет приблизительно $285000 \cdot 4$ байта $\approx 1{,}14$ МБ

Даже с учётом словаря, метаданных и служебных структур модель остаётся небольшой и может быть быстро загружена в оперативную память.

Для сравнения, Transformer-архитектуры и крупные языковые модели содержат значительно больше параметров. Даже компактные Transformer-модели могут занимать десятки или сотни мегабайт, а LLM — от нескольких гигабайт и выше. Это делает их менее удобными для локального запуска на обычном персональном компьютере.

Сравнение по памяти представлено в таблице 5.

Таблица 5 - Потребление памяти

Модель	Примерный размер модели	Оперативная память при работе	Комментарий
Bag-of-Words	менее 1 МБ	5–20 МБ	Минимальные требования, но слабый учёт контекста
LSTM	2–5 МБ	20–60 МБ	Больше параметров, чем у GRU
GRU	1–3 МБ	15–40 МБ	Компактная модель для коротких команд
GRU + fallback	1–3 МБ	20–50 МБ	Небольшой рост за счёт дополнительных словарей и правил
Transformer	100–500 МБ	300–1000 МБ	Значительно выше требования к памяти
LLM	от 4 ГБ	4–16 ГБ и выше	Требует значительного объёма ОЗУ

Из таблицы видно, что разработанная GRU-модель требует значительно меньше памяти, чем Transformer и LLM. При этом её точность остаётся достаточной для задачи классификации коротких голосовых команд.

3.4.2 Время инференса

Также важным показателем является время инференса, то есть время, необходимое модели для обработки одной текстовой команды и определения intent-класса.

Для голосового помощника время инференса напрямую влияет на отзывчивость системы. Если пользователь произносит команду, то он ожидает почти мгновенной реакции. Поэтому даже небольшое увеличение задержки может ухудшать опыт пользователя.

Компактные модели, такие как Bag-of-Words и GRU выполняют инференс быстро. Bag-of-Words показывает минимальную задержку, однако уступает по качеству классификации. GRU требует больше вычислений, так как обрабатывает последовательность токенов, но при короткой длине команд время обработки остаётся низким.

Сравнение времени инференса представлено в таблице 6.

Таблица 6 - Время инференса

Модель	Среднее время инференса одной команды
Bag-of-Words	1–2 мс
LSTM	5–12 мс
GRU	3–8 мс
GRU + fallback	5–12 мс
Transformer	50–200 мс
LLM	500 мс и выше

Добавление fallback-механизма и морфологической нормализации увеличивает время обработки, однако этот рост остаётся небольшим. Дополнительная задержка компенсируется повышением устойчивости системы к нестандартным формулировкам и ошибкам классификации.

Таким образом, GRU с гибридной обработкой сохраняет время инференса на уровне, приемлемом для интерактивной системы.

3.4.3 Загрузка центрального процессора

Загрузка CPU является одним из ключевых критериев для автономного голосового помощника. Система должна работать локально и постоянно находиться в режиме ожидания команды, поэтому высокая нагрузка на процессор недопустима.

В эксперименте можно оценивать два состояния:

- режим ожидания команды,
- режим обработки распознанной команды.

В режиме ожидания основную нагрузку создаёт модуль распознавания речи, так как он постоянно обрабатывает входной аудиопоток. Нейросетевая модель классификации намерений активируется только после получения текстовой команды, поэтому её вклад в среднюю нагрузку CPU остаётся ограниченным.

Во время инференса GRU-модель создаёт кратковременную нагрузку на процессор, но из-за малого размера модели и короткой длины команд эта нагрузка быстро завершается.

Примерное сравнение загрузки CPU представлено в таблице 7.

Таблица 7 - Нагрузка на CPU

Модель	Средняя загрузка CPU при инференсе	Пиковая загрузка CPU	Возможность фоновой работы
Bag-of-Words	1–2 %	до 5 %	высокая
LSTM	5–10 %	до 15 %	высокая
GRU	3–7 %	до 12 %	высокая
GRU + fallback	4–9 %	до 15 %	высокая
Transformer	20–40 %	до 60 %	ограниченная
LLM	60–100 %	до 100 %	низкая без GPU

Полученные значения показывают, что разработанная GRU-модель оказывает незначительное влияние на загрузку центрального процессора. Это

позволяет использовать голосовой помощник в фоновом режиме без заметного снижения производительности системы.

В отличие от этого, Transformer и особенно LLM требуют значительно большего количества операций. При локальном запуске без GPU такие модели создают высокую нагрузку на CPU, увеличивают задержку ответа и ухудшают общую отзывчивость компьютера.

3.4.4 Интегральное сравнение моделей

Для обобщения результатов можно представить сводную таблицу 8, объединяющую качество классификации и ресурсные показатели.

Таблица 8 - Общее сравнение

Модель	F1-score	RAM	CPU	Инференс	Общая оценка
Bag-of-Words	0,77	очень низкая	очень низкая	очень быстрый	недостаточная точность
LSTM	0,89	средняя	средняя	быстрый	допустимо, но тяжелее GRU
GRU	0,89	низкая	низкая	быстрый	хорошая
GRU + fallback	0,92	низкая	низкая	быстрый	оптимальный вариант
Transformer	0,94	высокая	высокая	средний	избыточен для задачи
LLM	0,95	очень высокая	очень высокая	медленный без GPU	нерационален для локального запуска

Из таблицы видно, что наилучшее соотношение точности и ресурсной эффективности показывает вариант GRU + fallback. Он уступает Transformer

и LLM по F1-score, однако разница в точности является небольшой, тогда как выигрыш по памяти, CPU и времени инференса является существенным.

3.5 Итоговое сравнение разработанного решения с альтернативами

После проведения оценки качества классификации, анализа вычислительных ресурсов, исследования fallback-механизма, морфологической нормализации и самообучения можно выполнить итоговое сравнение разработанного решения с альтернативными подходами.

Основная цель данного сравнения заключается в том, чтобы определить, насколько предложенная архитектура соответствует требованиям автономного голосового помощника для персонального компьютера. В отличие от универсальных диалоговых систем, разрабатываемое решение ориентировано не на генерацию произвольных текстов, а на быстрое и надёжное выполнение ограниченного набора пользовательских команд.

Поэтому при сравнении учитываются не только показатели точности, но и практические характеристики:

- возможность автономной работы,
- отсутствие зависимости от интернет-соединения,
- низкое потребление оперативной памяти,
- низкая загрузка центрального процессора,
- малое время отклика,
- возможность работы без GPU,
- возможность адаптации к новым командам пользователя,

3.5.1 Сравнение по точности и ресурсам

Обобщённые результаты сравнения представлены в таблице 9.

Таблица 9 - Итоговое сравнение

Подход	F1-score	Время инференса	RAM	CPU	Работа без GPU	Самообучение	Итоговая применимость
Bag-of-Words	0,77	1–2 мс	5–20 МБ	1–2 %	Да	Ограниченно	Низкая
LSTM	0,89	5–12 мс	20–60 МБ	5–10 %	Да	Возможно	Хорошая, но тяжелее GRU
GRU	0,89	3–8 мс	15–40 МБ	3–7 %	Да	Возможно	Хорошая
GRU + fallback	0,92	5–12 мс	20–50 МБ	4–9 %	Да	Да	Наиболее рациональная
Transformer	0,94	50–200 мс	300–1000 МБ	20–40 %	Ограниченно	Возможно	Избыточная для коротких команд
LLM	0,95	500 мс и выше	4–16 ГБ и выше	60–100 %	Ограниченно	Частично	Нерациональна

Из таблицы видно, что наиболее высокие показатели качества демонстрируют Transformer-архитектуры и крупные языковые модели. Однако их преимущество по F1-score относительно разработанного решения является небольшим. Например, если F1-score варианта GRU + fallback составляет 0,92, а Transformer — 0,94, то разница равна 0,94 - 0,92 = 0,02.

Таким образом, прирост качества составляет около двух процентных пунктов.

При этом различие в ресурсных требованиях значительно выше. Если разработанная система использует около 50 МБ оперативной памяти, а Transformer — около 500 МБ, то отношение потребления памяти составляет 500 / 50 = 10.

То есть Transformer может требовать примерно в 10 раз больше оперативной памяти при приросте F1-score всего на 0,02.

Аналогично можно сравнить время инференса. Если разработанное решение обрабатывает команду за 10 мс, а Transformer — за 100 мс, то Transformer работает примерно в 10 раз медленнее 100 / 10 = 10.

Для LLM различие ещё более существенно. При незначительном приросте качества по сравнению с Transformer крупная языковая модель может требовать гигабайты оперативной памяти и создавать высокую нагрузку на CPU при локальном запуске без GPU.

3.5.2 Преимущества разработанного решения

На основании проведённого сравнения можно выделить несколько ключевых преимуществ разработанного голосового помощника.

Во-первых, система полностью функционирует локально. Это устраняет зависимость от интернет-соединения и повышает конфиденциальность пользовательских данных, поскольку голосовые команды не передаются на внешние серверы.

Во-вторых, разработанная модель имеет компактную архитектуру и может выполняться на CPU. Это позволяет использовать систему на обычном персональном компьютере без необходимости установки специализированного оборудования.

В-третьих, время инференса остаётся низким, что обеспечивает быстрый отклик системы на пользовательские команды.

В-четвёртых, использование fallback-механизма и морфологической нормализации позволяет повысить устойчивость обработки команд без перехода к более тяжёлым моделям.

В-пятых, механизм самообучения позволяет расширять функциональность системы в процессе эксплуатации. Пользователь может добавлять новые приложения, обучать систему новым глаголам и записывать макросы без изменения исходного кода.

3.5.3 Ограничения разработанного решения

Несмотря на преимущества, разработанное решение имеет ряд ограничений.

Во-первых, компактная GRU-модель уступает Transformer и LLM по универсальности обработки естественного языка. Она хуже подходит для длинных фраз, сложного диалога и неоднозначных запросов.

Во-вторых, система ориентирована на заранее определённые классы намерений. Если пользователь формулирует команду, выходящую за пределы поддерживаемых сценариев, ассистент не всегда сможет корректно её интерпретировать.

В-третьих, качество работы зависит от обучающей выборки. Для повышения точности необходимо включать в данные различные варианты формулировок команд.

В-четвёртых, локальное распознавание речи может уступать облачным сервисам в сложных акустических условиях, например при сильном шуме или нечетком произношении.

Однако данные ограничения не противоречат цели работы, поскольку система разрабатывалась не как универсальный диалоговый агент, а как автономный голосовой помощник для выполнения команд управления персональным компьютером.

3.5.4 Итоговая оценка

Итоговое сравнение показывает, что разработанное решение занимает промежуточное положение между простыми методами обработки текста и крупными языковыми моделями.

По сравнению с Bag-of-Words оно обеспечивает более высокое качество интерпретации команд за счёт учёта порядка слов и контекста. По сравнению с LSTM оно имеет близкую точность, но меньшую вычислительную сложность. По сравнению с Transformer и LLM оно уступает по универсальности и максимальной точности, однако значительно выигрывает по памяти, CPU, времени отклика и простоте локального развёртывания.

Таким образом, для задачи автономного голосового управления персональным компьютером разработанная архитектура является более рациональной, чем использование крупных языковых моделей. Она обеспечивает достаточный уровень точности при значительно меньших вычислительных затратах.

Итоговое сравнение также показало, что разработанная система обеспечивает наиболее сбалансированное соотношение между точностью, скоростью работы и потреблением ресурсов.

Использование GRU-модели в сочетании с fallback-механизмом, морфологической нормализацией и самообучением позволяет достичь высокой практической эффективности без применения ресурсоёмких Transformer-архитектур и LLM. Следовательно, предложенное решение является обоснованным для автономного голосового помощника, работающего локально на персональном компьютере и ориентированного на выполнение коротких пользовательских команд.

В ходе исследования были рассмотрены различные подходы к интерпретации команд: Bag-of-Words, LSTM, GRU, Transformer-архитектуры и крупные языковые модели. Сравнение показало, что простые методы, такие как Bag-of-Words, обладают минимальными требованиями к ресурсам,

однако уступают нейросетевым моделям по качеству классификации, поскольку не учитывают порядок слов и контекст пользовательской команды.

Рекуррентные архитектуры LSTM и GRU продемонстрировали более высокое качество обработки команд. При этом GRU оказалась более рациональным вариантом для рассматриваемой задачи, поскольку обеспечивает качество, близкое к LSTM, но требует меньше вычислительных ресурсов. Это особенно важно для коротких голосовых команд, где использование более сложной архитектуры не даёт значительного прироста точности.

Наиболее высокие показатели качества потенциально обеспечивают Transformer-архитектуры и крупные языковые модели. Однако их применение в рамках автономного голосового помощника связано с существенным увеличением требований к оперативной памяти, загрузке центрального процессора и времени инференса. Для задачи локального управления персональным компьютером данный прирост качества не является пропорциональным росту вычислительных затрат.

Особое значение имеет гибридная обработка команд. Проведённый анализ показал, что fallback-механизм и морфологическая нормализация позволяют повысить устойчивость системы к различным словоформам, неизвестным глаголам и нестандартным пользовательским формулировкам. Благодаря этому компактная нейросетевая модель получает дополнительную поддержку без существенного увеличения размера и сложности.

Также был рассмотрен механизм самообучения, позволяющий системе запоминать новые приложения, новые управляющие глаголы и пользовательские макросы. Данный механизм повышает практическую применимость голосового помощника, поскольку система может адаптироваться к конкретному пользователю и его программному окружению без изменения исходного кода.

Экспериментальная оценка подтвердила целесообразность использования компактной GRU-модели и гибридной архитектуры для

построения автономного голосового помощника. Полученные результаты показывают, что разработанная система соответствует основным требованиям работы: автономности, низкому потреблению ресурсов, достаточной точности классификации команд и возможности адаптации к новым пользовательским сценариям.

ЗАКЛЮЧЕНИЕ

В ходе выполнения магистерской диссертации была разработана и исследована программная система автономного голосового помощника для персонального компьютера. Основной целью работы являлось создание решения, способного обрабатывать голосовые команды пользователя без обращения к облачным сервисам и при этом обеспечивать достаточный уровень точности, быстродействия и низкое потребление вычислительных ресурсов.

В процессе исследования был проведён анализ существующих голосовых помощников и подходов к обработке естественного языка. Было установлено, что современные коммерческие решения, такие как Siri, Google Assistant, Amazon Alexa и Алиса, обладают высокой точностью и широкими функциональными возможностями, однако в значительной степени зависят от облачной инфраструктуры. Такая зависимость приводит к ограничениям, связанным с необходимостью постоянного интернет-соединения, передачей пользовательских данных на внешние серверы и повышенными требованиями к вычислительным ресурсам.

В рамках работы была предложена архитектура автономного голосового помощника, включающая модуль офлайн-распознавания речи, модуль обработки естественного языка, нейросетевую модель классификации намерений, модуль извлечения сущностей, исполнительный модуль, механизм самообучения и графический интерфейс пользователя. Такая модульная структура позволила разделить этапы распознавания, интерпретации и выполнения команд, что повысило гибкость и расширяемость разработанной системы.

Для интерпретации пользовательских команд была реализована компактная нейросетевая модель на основе архитектуры GRU. Данная модель выполняет классификацию текстовой команды по заранее заданным классам намерений. Использование GRU позволило учитывать порядок слов и контекст коротких команд при сохранении низкой вычислительной

сложности. В отличие от более тяжёлых архитектур, таких как Transformer и крупные языковые модели, разработанная модель не требует графического ускорителя и может эффективно функционировать на центральном процессоре обычного персонального компьютера.

Дополнительно в системе был реализован гибридный механизм обработки команд, включающий правило-ориентированное извлечение сущностей, морфологическую нормализацию, fallback-обработку и проверку управляющих глаголов. Данные механизмы позволили повысить устойчивость системы к различным словоформам, нестандартным формулировкам и ошибкам распознавания речи без существенного увеличения размера нейросетевой модели.

Важной особенностью разработанного решения является механизм самообучения. Он позволяет пользователю добавлять новые приложения, обучать систему неизвестным управляющим глаголам и записывать пользовательские макросы без изменения исходного кода. Благодаря этому голосовой помощник может адаптироваться к индивидуальному программному окружению и стилю речи пользователя.

В экспериментальной части работы было проведено сравнение разработанного решения с альтернативными подходами: Bag-of-Words, LSTM, Transformer-архитектурами и крупными языковыми моделями. Анализ показал, что простые методы обладают низкими требованиями к ресурсам, но уступают по качеству обработки естественного языка. Более сложные модели способны обеспечивать более высокую точность, однако требуют значительно большего объёма оперативной памяти, создают более высокую нагрузку на центральный процессор и имеют большее время инференса.

Разработанная GRU-модель в сочетании с fallback-механизмом и морфологической нормализацией показала наиболее рациональное соотношение точности и вычислительной эффективности для задачи локального голосового управления персональным компьютером. Несмотря

на некоторое снижение универсальности по сравнению с крупными языковыми моделями, предложенное решение обеспечивает достаточную точность классификации коротких команд при значительно меньших требованиях к памяти, CPU и времени отклика.

Таким образом, поставленная цель работы была достигнута. В результате исследования разработан автономный голосовой помощник, способный выполнять обработку пользовательских команд локально, без использования облачных сервисов. Система обеспечивает низкие требования к вычислительным ресурсам, поддерживает расширение функциональности за счёт самообучения и демонстрирует практическую применимость для управления персональным компьютером.

Перспективы дальнейшего развития работы связаны с расширением обучающего набора данных, улучшением качества распознавания речи в шумных условиях, добавлением новых классов пользовательских намерений, совершенствованием механизма самообучения и проведением более масштабного тестирования на различных конфигурациях персональных компьютеров. Также перспективным направлением является дальнейшая оптимизация модели и исследование компактных нейросетевых архитектур, способных повысить точность обработки команд без существенного увеличения вычислительных затрат.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лейн Х., Хапке Х., Ховард К. Обработка естественного языка в действии. СПб.: Питер, 2021.

2. Шишкин А. Г. Методы цифровой обработки и распознавания речи. М.: ИНФРА-М, 2023.

3. Прошина М. В. Современные методы обработки естественного языка: нейронные сети. 2022.

4. Vosk. Offline Speech Recognition API [Электронный ресурс] - URL: https://alphacephei.com/vosk/ (дата обращения: 20.05.2026).

5. Cho K., van Merriënboer B., Gulcehre C., Bahdanau D., Bougares F., Schwenk H., Bengio Y. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. 2014 [Электронный ресурс] - URL: https://arxiv.org/abs/1406.1078 (дата обращения: 20.05.2026).

6. pymorphy3. Morphological analyzer for Russian language. PyPI project page [Электронный ресурс] - URL: https://pypi.org/project/pymorphy3/ (дата обращения: 20.05.2026).

7. pyttsx3. Text-to-speech x-platform documentation [Электронный ресурс] - URL: https://pyttsx3.readthedocs.io/ (дата обращения: 20.05.2026).

8. PyTorch. PyTorch documentation. [Электронный ресурс] - URL: https://docs.pytorch.org/docs/stable/index.html (дата обращения: 20.05.2026)

ПРИЛОЖЕНИЕ А

(обязательное)

Голосовой помощник для ПК

Листинг программы

Приложение А.1

Файл assistant_core.py

```
from click import command

from .config import WAKE_WORD, LOW_CONFIDENCE_THRESHOLD, ALLOW_COMMANDS_WITHOUT_WAKE_WORD
from .data_store import load_actions, load_intents, load_synonyms, save_intents, load_synonyms, save_intents
from .entities import extract_entities
from .dataset_augmenter import augment_dataset
from .command_normalizer import canonicalize_command
from .event_bus import EventBus, NullEventBus
from .executor import CommandExecutor
from .intent_model import Command, IntentClassifier
from .fallback_engine import apply_contextual_fallback
from .intent_guard import apply_intent_guards
from .logger import setup_logger, log_command
from .speech import StreamingSpeechRecognizer, SpeechSynthesizer

class VoiceAssistant:
    def __init__(self, event_callback=None):
            self.event_bus = EventBus(event_callback) if event_callback else NullEventBus()
        self.logger = setup_logger()

        self.event_bus.emit("status", "Загружаю данные...")
        self.intent_data = load_intents()
        self.actions_config = load_actions()
        self.synonyms_config = load_synonyms()

                added = augment_dataset(self.intent_data, self.actions_config, self.synonyms_config)
        if added:
            save_intents(self.intent_data)
```

```
                self.event_bus.emit("status", f"Обучающая выборка
расширена: +{added} фраз")

                    self.event_bus.emit("status",  "Инициализирую
нейросеть...")
            self.classifier = IntentClassifier(self.intent_data)
            self.classifier.train()

                                                    self.stt      =
StreamingSpeechRecognizer(event_bus=self.event_bus)
                                                    self.tts      =
SpeechSynthesizer(event_bus=self.event_bus)

             self.executor = CommandExecutor(self.actions_config,
self.classifier, self.synonyms_config)
            self.pending_low_confidence_command = None
            self.running = True

        def stop(self):
            self.running = False
                        self.event_bus.emit("status",   "Остановка
ассистента...")

        def analyze(self, text: str):
                    normalized_text  =  canonicalize_command(text,
self.synonyms_config)
                                        intent,    confidence    =
self.classifier.predict(normalized_text)
            intent = apply_intent_guards(normalized_text, intent)
             entities = extract_entities(normalized_text, intent,
self.actions_config)

              command_text = normalized_text if intent != "none"
else text
```

```
            command = Command(text=command_text, intent=intent,
confidence=confidence, entities=entities)
            print(f"[COMMAND] {command}")
            self.event_bus.emit(
                "command",
                                        f"intent={command.intent},
confidence={command.confidence:.2f},
entities={command.entities}",
                {
                    "text": command.text,
                    "intent": command.intent,
                    "confidence": command.confidence,
                    "entities": command.entities,
                },
            )
            return command

        def process_text(self, text: str):
            text = text.strip()

            if not text:
                self.tts.say("Я не расслышал команду.")
                return False

            low = text.lower()

                if any(phrase in low for phrase in ["выход",
"завершить работу", "стоп ассистент"]):
                self.tts.say("Останавливаюсь. До свидания.")
                return True

            command = self.analyze(text)
                        from assistant.fallback_engine import
apply_contextual_fallback
```

```
            command = apply_contextual_fallback(
                command,
                self.actions_config,
                self.synonyms_config
            )

            if command.intent == "none":
                if len(command.text.split()) >= 2:
                                                        reply   =
self.executor.start_auto_learning(command.text)
                else:
                    reply = "Я не понял команду"

                self.tts.say(reply)
                return

            if command.intent == "finish_recording":
                 reply = self.executor.execute(command, self.tts,
self.stt)
                self.tts.say(reply)
                   self.event_bus.emit("actions_updated", "Список
действий обновлён")
                        log_command(command.text,  command.intent,
command.confidence, command.entities, reply)
                return False

             if command.confidence < LOW_CONFIDENCE_THRESHOLD and
command.intent not in {"confirm", "reject", "finish_recording"}:
                self.pending_low_confidence_command = command
                 reply = f"Я не совсем уверен. Выполнить команду:
{command.text}? Скажи да или отмена."
                self.tts.say(reply)
                        log_command(command.text,  command.intent,
command.confidence, command.entities, reply)
                return False
```

```
                        if command.intent == "confirm" and
self.pending_low_confidence_command:
                pending = self.pending_low_confidence_command
                self.pending_low_confidence_command = None
                 reply = self.executor.execute(pending, self.tts,
self.stt)
                self.tts.say(reply)
                       log_command(pending.text, pending.intent,
pending.confidence, pending.entities, reply)
                return False

                         if command.intent == "reject" and
self.pending_low_confidence_command:
                self.pending_low_confidence_command = None
                reply = "Хорошо, отменяю."
                self.tts.say(reply)
                       log_command(command.text, command.intent,
command.confidence, command.entities, reply)
                return False

               reply = self.executor.execute(command, self.tts,
self.stt)
            self.tts.say(reply)

                if command.intent in {"learn_macro", "confirm",
"delete_target"}:
                   self.event_bus.emit("actions_updated", "Список
действий обновлён")

                      log_command(command.text, command.intent,
command.confidence, command.entities, reply)
            return False

        def run(self):
```

```
            self.event_bus.emit("status", "Ассистент запущен")
            self.tts.say("Голосовой помощник запущен.")

            while self.running:
                self.executor.check_reminders(self.tts)

                text = self.stt.listen_phrase()
                print(f"[STT] {text}")

                low = text.lower()

                if WAKE_WORD in low:
                                after_wake  =  low.split(WAKE_WORD,
1)[1].strip(" ,.!?")

                    if after_wake.startswith("слушай"):
                        after_wake = after_wake.replace("слушай",
"", 1).strip(" ,.!?")

                    if after_wake:
                        if self.process_text(after_wake):
                            break
                    else:
                        self.tts.say("Слушаю.")
                        command_text = self.stt.listen_phrase()
                        if self.process_text(command_text):
                            break

                elif ALLOW_COMMANDS_WITHOUT_WAKE_WORD:
                    if self.process_text(low):
                        break

            self.event_bus.emit("status", "Ассистент остановлен")
```

Приложение А.2

Файл intent_model.py

```
import hashlib
import json
from dataclasses import dataclass, field
from typing import Any, Dict, List, Tuple

import numpy as np
import torch
import torch.nn as nn

from .config import (
            CACHE_DIR,  MODEL_CACHE_FILE,  VOCAB_CACHE_FILE,
META_CACHE_FILE,
    EMBEDDING_DIM, HIDDEN_DIM, EPOCHS, LEARNING_RATE,
    INTENT_THRESHOLD, MAX_SEQUENCE_LENGTH,
)
from .text_processing import tokenize, normalize_text

INTENTS = [
        "open_target", "type_text", "get_time", "get_date",
"volume_up", "volume_down",
            "volume_mute",  "switch_window",  "open_folder",
"create_file", "set_reminder",
            "list_targets",  "delete_target",  "learn_macro",
"finish_recording", "confirm", "reject", "none",
]

INTENT_TO_IDX  =  {name:  index  for  index,  name  in
enumerate(INTENTS)}
IDX_TO_INTENT  =  {index:  name  for  name,  index  in
INTENT_TO_IDX.items()}
PAD_TOKEN = "<PAD>"
UNK_TOKEN = "<UNK>"
```

```
    @dataclass
    class Command:
        text: str
        intent: str
        confidence: float
        entities: Dict[str, Any] = field(default_factory=dict)

    class GRUIntentNet(nn.Module):
         def __init__(self, vocab_size: int, embedding_dim: int,
hidden_dim: int, output_dim: int):
            super().__init__()
                      self.embedding  =  nn.Embedding(vocab_size,
embedding_dim, padding_idx=0)
                   self.gru  =  nn.GRU(embedding_dim,  hidden_dim,
batch_first=True)
            self.classifier = nn.Sequential(
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, output_dim),
            )

        def forward(self, x):
            embedded = self.embedding(x)
            _, hidden = self.gru(embedded)
            last_hidden = hidden[-1]
            return self.classifier(last_hidden)

    class IntentClassifier:
        def __init__(self, intent_data: List[dict]):
            self.intent_data = intent_data
            self.words = []
            self.word_to_idx = {}
            self.model = None

        def dataset_hash(self):
```

```
                        payload = json.dumps(self.intent_data,
ensure_ascii=False, sort_keys=True)
                    return hashlib.sha256(payload.encode("utf-
8")).hexdigest()

        def build_vocab(self):
            words = {PAD_TOKEN, UNK_TOKEN}

            for item in self.intent_data:
                intent = item.get("intent")
                if intent not in INTENT_TO_IDX:
                        raise ValueError(f"Неизвестный intent в
intents.json: {intent}")

                for token in tokenize(item["phrase"]):
                    words.add(token)

            self.words = [PAD_TOKEN, UNK_TOKEN] + sorted(words -
{PAD_TOKEN, UNK_TOKEN})
             self.word_to_idx = {word: index for index, word in
enumerate(self.words)}

        @property
        def vocab_size(self):
            return len(self.words)

        def text_to_sequence(self, text: str):
            tokens = tokenize(text)
                        ids = [self.word_to_idx.get(token,
self.word_to_idx[UNK_TOKEN]) for token in tokens]
            ids = ids[:MAX_SEQUENCE_LENGTH]
                    ids += [self.word_to_idx[PAD_TOKEN]] *
(MAX_SEQUENCE_LENGTH - len(ids))
            return np.array(ids, dtype=np.int64)
```

```
        def load_from_cache(self):
                        if not MODEL_CACHE_FILE.exists() or not
VOCAB_CACHE_FILE.exists() or not META_CACHE_FILE.exists():
                return False

            with open(META_CACHE_FILE, "r", encoding="utf-8") as
file:
                meta = json.load(file)

            if meta.get("dataset_hash") != self.dataset_hash():
                return False

              with open(VOCAB_CACHE_FILE, "r", encoding="utf-8")
as file:
                self.words = json.load(file)

              self.word_to_idx = {word: index for index, word in
enumerate(self.words)}

                        self.model = GRUIntentNet(self.vocab_size,
EMBEDDING_DIM, HIDDEN_DIM, len(INTENTS))
             self.model.load_state_dict(torch.load(MODEL_CACHE_FI
LE, map_location="cpu"))
            self.model.eval()

            print("[NN] GRU-модель загружена из cache.")
            return True

        def save_to_cache(self):
            CACHE_DIR.mkdir(parents=True, exist_ok=True)

            torch.save(self.model.state_dict(), MODEL_CACHE_FILE)

              with open(VOCAB_CACHE_FILE, "w", encoding="utf-8")
as file:
```

```
                json.dump(self.words, file, ensure_ascii=False,
indent=2)

            with open(META_CACHE_FILE, "w", encoding="utf-8") as
file:
                json.dump(
                    {
                        "dataset_hash": self.dataset_hash(),
                        "vocab_size": self.vocab_size,
                        "intents": INTENTS,
                        "architecture": "Embedding+GRU",
                                            "max_sequence_length":
MAX_SEQUENCE_LENGTH,
                    },
                    file,
                    ensure_ascii=False,
                    indent=2,
                )

            print("[NN] GRU-модель сохранена в cache.")

        def train(self, force=False):
            if not force and self.load_from_cache():
                return

            print("[NN] Обучаю GRU intent-модель...")
            self.build_vocab()

                                                    x_data      =
np.stack([self.text_to_sequence(item["phrase"])   for   item   in
self.intent_data])
            y_data = np.array([INTENT_TO_IDX[item["intent"]] for
item in self.intent_data], dtype=np.int64)

            x_tensor = torch.from_numpy(x_data)
```

```
            y_tensor = torch.from_numpy(y_data)

                    self.model  =  GRUIntentNet(self.vocab_size,
EMBEDDING_DIM, HIDDEN_DIM, len(INTENTS))

            criterion = nn.CrossEntropyLoss()
            optimizer = torch.optim.Adam(self.model.parameters(),
lr=LEARNING_RATE)

            self.model.train()
            for _ in range(EPOCHS):
                optimizer.zero_grad()
                outputs = self.model(x_tensor)
                loss = criterion(outputs, y_tensor)
                loss.backward()
                optimizer.step()

            self.model.eval()
            self.save_to_cache()

            def  predict(self,  text:  str,  threshold:  float  =
INTENT_THRESHOLD) -> Tuple[str, float]:
             if not text or self.model is None or self.vocab_size
== 0:
                return "none", 0.0

            clean_text = normalize_text(text)
            sequence = self.text_to_sequence(clean_text)
            x_tensor = torch.from_numpy(sequence).unsqueeze(0)

            with torch.no_grad():
                logits = self.model(x_tensor)
                probabilities = torch.softmax(logits, dim=1)[0]
                    confidence,  index  =  torch.max(probabilities,
dim=0)
```

```
            confidence_value = float(confidence.item())
            intent = IDX_TO_INTENT[int(index)]

                print(f"[NN]  text='{clean_text}'  intent={intent}
confidence={confidence_value:.2f}")

            if confidence_value < threshold:
                return "none", confidence_value

            return intent, confidence_value

        def add_training_phrase(self, phrase: str, intent: str):
            phrase = normalize_text(phrase)
            if not phrase:
                return
            if intent not in INTENT_TO_IDX:
                raise ValueError(f"Неизвестный intent: {intent}")

            item = {"phrase": phrase, "intent": intent}
            if item not in self.intent_data:
                self.intent_data.append(item)

        def add_open_target_training(self, target_name: str):
            target_name = normalize_text(target_name)

            if not target_name:
                return

            verbs = [
                    "открой", "запусти", "стартани", "включи",
"загрузи",
                "активируй", "открой мне", "запусти мне",
                "можешь открыть", "можешь запустить",
                "давай открой", "давай запусти",
```

```
            ]

            for verb in verbs:
                self.add_training_phrase(f"{verb} {target_name}",
"open_target")

            self.train(force=True)

        def add_delete_target_training(self, target_name: str):
            target_name = normalize_text(target_name)
            if not target_name:
                return

                self.add_training_phrase(f"удали {target_name}",
"delete_target")
                self.add_training_phrase(f"забудь {target_name}",
"delete_target")
            self.train(force=True)
```

Приложение А.3

Файл fallbakc_engine.py

```
from dataclasses import replace
import pymorphy3

morph = pymorphy3.MorphAnalyzer()

def normalize(word: str):
    parsed = morph.parse(word)
    return parsed[0].normal_form if parsed else word

def apply_contextual_fallback(command, actions_config, synonyms_config):
    words = command.text.split()

    TARGET_INTENTS = {
        "open_target",
        "delete_target",
        "open_folder"
    }

    if command.intent not in TARGET_INTENTS:
        return command

    if len(words) >= 2:
        verb = normalize(words[0])

        def normalize_list(words):
            return {normalize(w) for w in words}

                open_verbs = synonyms_config.get("verbs", {}).get("open_target", {}).get("synonyms", [])
                delete_verbs = synonyms_config.get("verbs", {}).get("delete_target", {}).get("synonyms", [])
```

```
            learned = list(synonyms_config.get("learned_verbs",
{}).keys())

            all_verbs = (
                normalize_list(open_verbs)
                | normalize_list(delete_verbs)
                | normalize_list(learned)
            )

            if verb not in all_verbs:
                        return  replace(command,  intent="none",
entities={})

        if command.intent == "finish_recording":
                 allowed  =  {"готово",  "закончил",  "закончила",
"останови запись", "стоп запись"}
            if command.text.lower().strip() not in allowed:
                        return  replace(command,  intent="none",
entities={})

        return command
```

ПРИЛОЖЕНИЕ В

(обязательное)

Голосовой помощник для ПК

Презентация

Голосовой помощник для ПК

подготовил:
студент гр. 24ВВИм1
Субботкин Максим Владимирович

научный руководитель:
к.т.н., доцент Юрова О.В.

Пенза, 2026

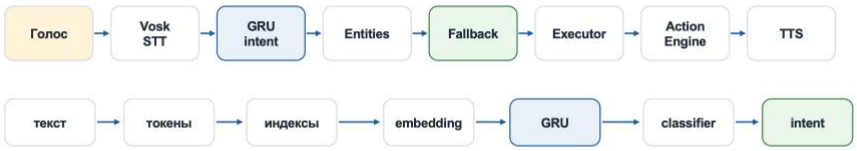
Проблемы и акутальность

1) зависимость от интернета
2) передача данных на сервера
3) высокое время отклика
4) высокие требования при локальном развертывании

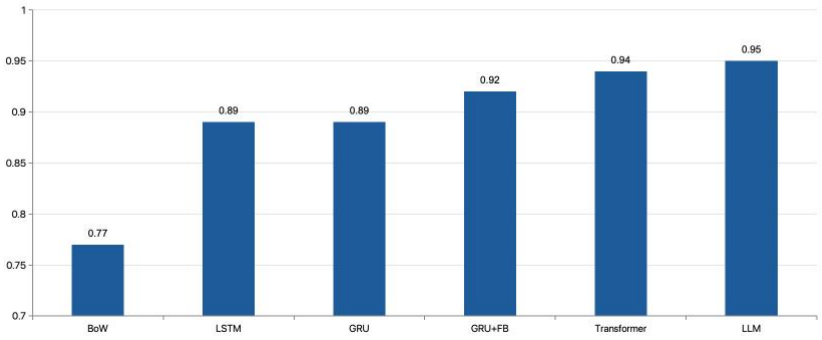
Требования к разработке

1) Автономность
2) Быстрый отклик
3) Низкое потребление ресурсов
4) Расширяемость
5) Безопасность

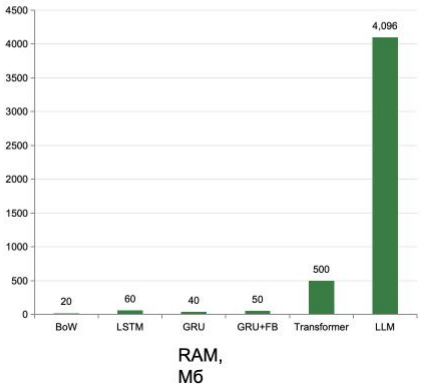
Архитектура разработанной системы



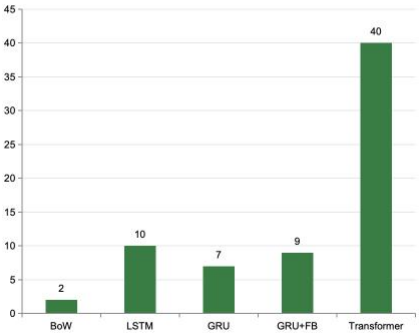
Результаты - F1 score



Результаты - потребление ресурсов

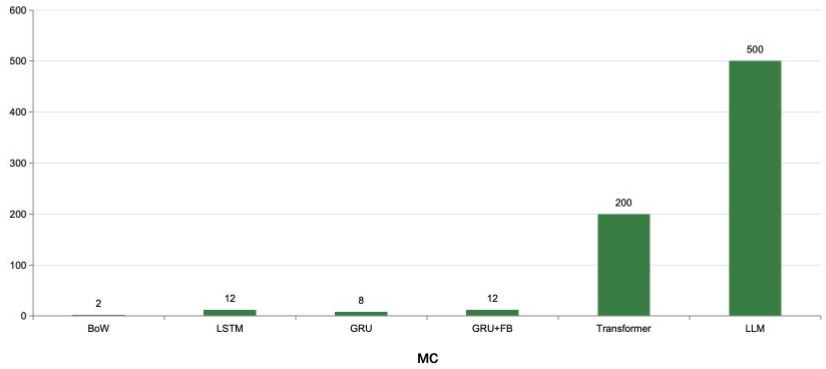


RAM,
Мб



ЦП, %

Результаты - время инференса



Итоговые результаты

Преимущества

1) полностью локальна
2) низкие требования к ресурсам
3) быстрый отклик
4) самообучение

Недостатки

1) не предназначена для диалога
2) офлайн stt чувствителен к шуму

Заключение

Цель достигнута: разработан автономный голосовой помощник, который локально распознаёт, интерпретирует и выполняет команды пользователя.

Vosk обеспечивает офлайн-распознавание речи
GRU-классификатор определяет намерение пользователя
fallback и морфология повышают устойчивость

самообучение расширяет систему без изменения кода
сравнение показывает лучший баланс точности и ресурсов