Валерий Антонов

ChatGPT на вашем ноутбуке: бесплатно, анонимно, навсегда

Программист в эпоху ИИ: реверс-инжиниринг, алгоритмы и новое мышление

Валерий Антонов

ChatGPT на вашем ноутбуке: бесплатно, анонимно, навсегда

«Автор»
2026

Антонов В.

ChatGPT на вашем ноутбуке: бесплатно, анонимно, навсегда / В. Антонов — «Автор», 2026 — (Программист в эпоху ИИ: реверс-инжиниринг, алгоритмы и новое мышление)

Как создать локальный ИИ одной кнопкой — практическое руководство по построению персонального AI-ассистента, работающего без интернета и облачных сервисов. Чат-бот с памятью, голосовое управление, переводчик, генератор изображений, реставратор фото и два десятка специализированных агентов — шеф-повар, репетитор, терапевт, психолог и другие — всё на вашем компьютере, запускается двойным щелчком. Полная конфиденциальность: ваши данные никогда не покидают устройство.Книга адресована разработчикам на Python и всем, кто ценит независимость. Для новичков — Глава 0 с быстрым стартом за пять минут. Никакой математики — только код и понятные объяснения.Это первая книга серии. Готовятся новые версии с тонкой настройкой моделей и отраслевыми решениями. Подписывайтесь на автора на Литрес. Вскоре после выхода книги будет опубликована ссылка на репозиторий с кодом проекта.

Содержание

ChatGPT на вашем ноутбуке: бесплатно, анонимно, навсегда

Аннотация

Ещё недавно персональный искусственный интеллект был научной фантастикой. Сегодня квантованные модели весом 8–20 гигабайт работают на обычных ноутбуках, не уступая коммерческим сервисам в рутинных задачах. Они не требуют интернета, не отправляют ваши данные в облака и не просят ежемесячную подписку. Они — ваши. Целиком и полностью.

Как создать локальный ИИ одной кнопкой — это практическое руководство по построению такого ассистента. Книга проведёт вас от первой команды pip install до готового исполняемого файла, который запускается двойным щелчком и содержит: чат-бота с памятью, голосовое управление, переводчик на двести языков, генератор изображений, реставратор старых фотографий, органайзер снимков, голосовой блокнот с саммаризацией и детектор настроения.

В версии 2.0 добавлен **автоматизированный конвейер обработки книг** — от скачивания PDF до перевода на русский язык. Конвейер выполняет четыре шага: конвертация страниц в изображения, распознавание текста мультимодальной моделью GLM-4.6V-Flash, объединение фрагментов и литературный перевод моделью HY-MT1.5-7B. Поддерживаются прямые ссылки, локальные файлы, ZIP-архивы и поиск книг в интернете. Достаточно нажать одну кнопку — и книга Канта на немецком превращается в читаемый русский текст.

Вы создадите агента, который самостоятельно решает цепочки задач: находит информацию в документах, отправляет письма, управляет файлами. Научите модель говорить в вашем стиле, создав цифрового двойника без дообучения. Развернёте ассистента на изолированном сервере для целого отдела, упакуете в Docker, настроите резервное копирование и диагностику.

Отдельная часть книги посвящена специализированным агентам. Шеф-повар проведёт вас по рецепту шаг за шагом, сомелье подберёт вино к ужину, диетолог посчитает калории. Репетитор объяснит школьную тему и проверит решение. Терапевт проанализирует симптомы и предупредит, когда пора вызывать скорую. Психотерапевт проведёт сессию в подходе КПТ. Новостник соберёт утреннюю сводку, копирайтер напишет статью в вашем стиле, а прогнозист составит гороскоп или построит аналитический прогноз на основе ваших данных.

Всё это работает полностью офлайн, на вашем компьютере, без единого запроса во внешний мир. Никакой сложной математики — только код, понятные объяснения и философия «сложность должна быть скрыта за одной кнопкой».

Книга адресована разработчикам на Python, IT-специалистам, уставшим от облачных подписок, энтузиастам, которые хотят разобраться в AI-технологиях на практике, и всем, кто ценит конфиденциальность и независимость. Для тех, кто не пишет код, есть Глава 0 — быстрый старт с готовым установщиком за пять минут.

Эта книга — первая в серии. Автор продолжает работу над новыми версиями: расширенным изданием с углублённым разбором тонкой настройки моделей, мультимодальными агентами нового поколения и специализированными отраслевыми решениями. Подписывайтесь на уведомления о новых книгах автора на Литрес, чтобы не пропустить выход следующей версии.

Итоги проекта:

Чат (Qwen 32B)

Перевод текстов (HY-MT1.5 Q5)

Улучшение фото

Шеф-повар, Репетитор, Терапевт, Психолог

Конвейер книг: PDFPNGOCR Перевод
Ссылка на репозиторий: https://github.com/PawelKaev/local-ai-one-button
В репозитории обновлены:
app.py — основной файл с моделями Qwen, HY-MT, GLM-OCR
gui.py — интерфейс с вкладкой Книги
local_model.py — класс для работы с моделями
.gitignore — исключены DLL, модели, PDF
Следите за обновлениями на странице книги.

Введение

Ещё пять лет назад запустить языковую модель на домашнем ноутбуке было фантастикой. Нужно было арендовать сервер, платить за API и надеяться, что ваши данные не утекут. Сегодня всё изменилось. Квантованные модели весом 8–14 гигабайт работают на обычных ноутбуках и сравнимы по качеству с коммерческими аналогами. Мы входим в эпоху, когда персональный ИИ перестаёт быть метафорой и становится программой в папке models/, которую можно потрогать.

Эта книга — практическое руководство по построению такого ИИ. Мы соберём ассистента, который работает полностью на вашем компьютере, без интернета и облачных сервисов. Но прежде чем писать код, давайте поймём, зачем это нужно, когда у каждого в браузере есть ChatGPT, и куда движется вся индустрия.

Часть 1. Что даёт локальный ИИ сегодня

Облачные сервисы вроде ChatGPT, Gemini и Copilot велики и могучи. Но у локального ИИ есть преимущества, которые не исчезают с выходом каждой новой версии GPT. Они фундаментальны — они про физику, право и человеческую природу.

Безусловная приватность и суверенитет данных. Это не абстрактное «чтобы никто не прочитал». Это конкретные сценарии, в которых облако невозможно по закону, по этике или по здравому смыслу. Медицина и юриспруденция — вы можете скормить модели сто страниц истории болезни или договоров с персональными данными, и они физически не покинут ноутбук. Секретные исследования и разработки — инженеры загружают схемы и внутреннюю документацию, не боясь утечки через облако. Личная психотерапия и дневники — пользователи готовы доверять локальной модели самые сокровенные мысли, зная, что они не попадут в датасет для тренировки следующей версии ChatGPT. Локальный ИИ не просто защищает данные — он делает возможными сценарии, которые в облаке запрещены или немыслимы.

Гарантированная доступность и скорость. Локальная модель работает без интернета. В самолёте, в бункере, в деревне с нестабильным сигналом — она отвечает. Это критично для полевых геологов, военных, моряков и всех, кто не живёт в центре мегаполиса с оптоволокном. Но даже в городе важна скорость. Локальная модель на современном ноутбуке выдаёт токены быстрее, чем сетевой запрос доходит до сервера OpenAI. Для real-time приложений — голосового ассистента в игре, субтитров в реальном времени, помощника при презентации — задержка в полсекунды критична. Локальный ИИ отвечает мгновенно.

Отсутствие цензуры и настоящая кастомизация. Облачные модели вылизаны до стерильности. Вам не скажут «как языковая модель, я не могу...» в ответ на просьбу написать сценарий для хоррора с жестокими сценами или проанализировать спорный политический документ. Локальная модель — ваша, и вы решаете, что ей можно, а что нет. Вы можете сделать системный промпт «Ты — циничный детектив из 1940-х» и не бояться, что через три сообщения модель сломается в ханжеского морализатора, как это делают вылизанные облачные API. Тонкая настройка личности, стиля и границ дозволенного — в ваших руках.

Агентность и автоматизация рабочего стола. Это самое перспективное применение прямо сейчас. Локальная модель имеет прямой доступ к вашим файлам, процессам и памяти. Она может разобрать захламлённый рабочий стол за пять лет по папкам, читать все ваши письма, календарь, код и сообщения в Slack, создавая идеального цифрового двойника без риска компрометации всей корпоративной переписки. Облачной Copilot вы никогда не дадите доступ ко всем письмам — локальному дадите, потому что данные не покидают компьютер.

Часть 2. Куда мы движемся: перспективы на 2–3 года

Мы находимся в точке фазового перехода. Модели размером 8 гигабайт — это предел для плотных архитектур. Дальше нас ждут четыре больших тренда.

Расцвет смеси экспертов на устройстве. Представьте модель весом 20 гигабайт, но в каждый момент времени активны только 2–3 гигабайта параметров. Памяти занято много — нужен быстрый SSD или большой объём оперативной памяти, — а скорость мышления как у лёгкой модели. DeepSeek уже показал, что архитектура Mixture of Experts работает в облаке. Следующий шаг — адаптация для локального запуска на чипах Apple M4 Ultra или Snapdragon X Elite. Вы получаете интеллект большой модели по цене маленькой.

Мультимодальность станет стандартом для 8 гигабайт. Уже есть Llama 3.2 Vision на 11 миллиардов параметров. Скоро модели весом 7–9 гигабайт смогут не просто описывать картинки, а управлять интерфейсом, видя скриншот. Это убьёт классический веб-серфинг. Вы скажете локальному агенту: «Найди билеты на завтра, игнорируя красные даты», и он сам будет кликать в браузере. Приложения начнут общаться не с пользователем, а с его агентом.

Специализация и обучение на лету. Облачные модели — это средняя температура по больнице. Они знают всё понемногу и ничего — о вашем проекте, вашем коде, ваших клиентах. Локальные модели научатся дообучаться на ваших данных без утечки через LoRA-адаптеры. Программист скармливает модели тысячу своих коммитов, и она начинает писать код точно в его стиле, зная все внутренние библиотеки компании. Через час работы она становится полезнее Copilot, потому что знает контекст проекта. Это не фантастика — это технология, которая уже работает в исследовательских лабораториях и скоро придёт в потребительские приложения.

Гиперинтегрированные нейронные процессоры. Сейчас для серьёзного ИИ всё ещё нужна видеокарта. Но на горизонте 3–5 лет NPU-блоки в ноутбуках достигнут производительности RTX 4070 при потреблении 5–10 ватт. Это позволит моделям работать в фоне постоянно — слушать микрофон, анализировать экран и предугадывать действия, не разряжая батарею за час. Локальный ИИ станет вездесущим, как операционная система.

Часть 3. Главный вызов: зачем платить, если есть бесплатный GPT?

Здесь проходит самый жёсткий водораздел. Локальный ИИ проигрывает облаку в двух вещах: эрудиции и пиковом интеллекте. У облака всегда будет доступ к свежему поиску и новостям. Даже сжатая 8B-модель не решит олимпиадную задачу по физике так, как GPT-4o или Claude Sonnet. Отрицать это бессмысленно.

Поэтому победит не локальный ИИ и не облачный — победит гибридный подход. Простой запрос нейтрального характера — локальная модель отвечает мгновенно и бесплатно. Сложный запрос или потребность в свежем факте — локальная модель выступает как оркестратор. Она на лету принимает решение переслать задачу внешнему API или запросить поисковую систему. Секретный запрос — данные обрабатываются строго локально, в облако уходит только обезличенный результат.

Квантованные модели в 8 гигабайт уже сейчас убивают необходимость в подписке на ChatGPT для рутинных интеллектуальных задач: написание текстов, саммаризация, рефакторинг кода, перевод. Облако остаётся для экспертных пиковых задач. Локальный ИИ становится вашим приватным мозгом, который думает вместе с вами круглосуточно и бесплатно, а облачный — дорогим консультантом, которого вы вызываете для решения особо сложной проблемы.

Основные возможности

Приложение предоставляет восемь функциональных вкладок:

· **Чат —** общение с моделью Qwen 3 32B.

· **Перевод —** профессиональный перевод текстов моделью HY-MT1.5-7B Q5.

· **Фото —** улучшение качества изображений (повышение резкости, контраста, цветокоррекция).

· **Шеф —** кулинарные советы и рецепты с использованием базы знаний.

· **Репетитор —** помощь в обучении: объяснение тем, проверка решений, составление планов.

· **Терапевт —** предварительный анализ симптомов (не заменяет врача).

· **Психолог —** поддерживающий диалог с сохранением контекста беседы.

· **Книги —** полностью автоматизированный конвейер обработки книг (новое в версии 2.0).

Новое в версии 2.0: Конвейер обработки книг

Главное дополнение текущей версии **— конвейер обработки книг**, выполняющий четыре последовательных шага:

1. **PDF PNG —** конвертация страниц книги в изображения с помощью pdf2image и Poppler.

2. **OCR (распознавание текста) —** извлечение текста из изображений с помощью мультимодальной модели GLM-4.6V-Flash (GLM-OCR) через Ollama API.

3. **Объединение —** сбор всех распознанных фрагментов в единый текстовый файл.

4. **Перевод —** перевод объединённого текста на русский язык с помощью специализированной модели HY-MT1.5-7B Q5.

Конвейер поддерживает:

· Прямые ссылки на PDF, ZIP и HTML файлы

· Локальные файлы (выбор через диалог)

· Поиск книг по названию в интернете

· ZIP-архивы с автоматическим извлечением PDF

· Пошаговый или полностью автоматический режим работы

Подробное описание конвейера приведено в **главе 20.2**.

Дополнения, внесённые в версию 2.0

Модели:

· Добавлена поддержка модели перевода **HY-MT1.5-7B Q5_K_M** (4.6 ГБ) — специализированная модель для перевода, заменившая базовую Q4 версию. Обеспечивает более высокое качество перевода, особенно для немецкого языка.

· Добавлена мультимодальная модель **GLM-4.6V-Flash (Q8_0)** для распознавания текста с изображений. Работает через Ollama API, поддерживает немецкий, английский и русский языки.

Графический интерфейс:

· Добавлена вкладка **«Книги»** с полным интерфейсом управления конвейером.

· Добавлена вкладка **«Зрение»** (в разработке) для анализа содержимого изображений.

· Реализован лог выполнения с отображением прогресса каждого шага.

· Добавлены кнопки выбора локального файла и открытия папки с результатом.
Обработка файлов:
· Поддержка прямых ссылок на PDF, ZIP, HTML и TXT файлы.
· Автоматическое извлечение PDF из ZIP-архивов.
· Обработка HTML и TXT файлов напрямую, без конвертации.
· Сохранение промежуточных результатов (PNG, распознанный текст).
Интеграция с Ollama:
· Настроена работа модели GLM-OCR через Ollama API.
· Реализована передача изображений в модель для распознавания текста.
Документация:
· Добавлена **глава 20.2** с подробным описанием архитектуры и использования конвейера.
· Обновлено описание всех моделей и их параметров.
· Добавлены результаты тестирования на реальных текстах.

Структура проекта

Проект состоит из следующих основных файлов:
· app.py — главный файл, инициализация моделей и подключение колбэков к GUI.
· gui.py — графический интерфейс на Tkinter (8 вкладок).
· local_model.py — класс-обёртка для работы с моделями через llama-cpp-python.
Вспомогательные модули конвейера:
· pdf_converter.py — конвертация PDF в PNG.
· ocr_extract.py — распознавание текста с изображений.
· merge_all_texts.py — объединение текстовых файлов.
· translate_with_hy_mt.py — перевод текстов.
· search_and_download.py — поиск и скачивание книг.

Системные требования

· Python 3.10+
· Ollama (для работы GLM-OCR)
· Poppler (для конвертации PDF)
· Видеокарта с поддержкой CUDA (рекомендуется) или 16+ ГБ RAM для CPU-режима
· Свободное место на диске: ~40 ГБ для моделей

Для кого эта книга

Я предполагаю, что вы немного знакомы с Python: знаете, что такое функции и классы, умеете устанавливать библиотеки через pip и не боитесь читать сообщения об ошибках. Но даже если нет — Глава 0 поможет запустить готовый проект без единой строчки кода.

Книга пригодится разработчикам, которые хотят внедрить ИИ в офлайн-решения, IT-специалистам, уставшим от облачных подписок и озабоченным конфиденциальностью, энтузиастам, которые любят разбираться в технологиях на практике, родителям, которые хотят дать ребёнку безопасного ИИ-репетитора, и всем, кто хочет получить личного AI-помощника и точно знать, что их переписка не уходит на чужие серверы.

После книги

Когда вы освоите материал, вы станете не просто пользователем ИИ, а его создателем. Вы сможете адаптировать помощника под свои задачи — будь то автоматизация документооборота, восстановление семейного фотоархива, обучение детей или создание прототипа коммерческого AI-продукта.

А если вам понадобится то, что выходит за рамки книги — специализированный агент для вашей отрасли, интеграция с 1С или CRM, веб-интерфейс для команды, обучение сотрудников или даже книга под вашим авторством на основе вашей экспертизы — я готов помочь. На вкладке «Заказать» в приложении вы найдёте форму связи со мной. Опишите задачу — и я отвечу в течение одного-двух дней с предложением и оценкой. Рынок локального ИИ только формируется, и специалистов, которые умеют всё это собирать и настраивать, пока единицы. Вы — один из них.

Перед стартом

Убедитесь, что у вас установлен Python 3.10 или выше. Желательно иметь видеокарту NVIDIA с 6 и более гигабайтами памяти для быстрой работы, но большинство примеров будут работать и на процессоре — просто чуть медленнее. Все остальные инструменты мы установим по ходу книги.

Готовы? Тогда запускайте первую главу и готовьтесь нажать свою «одну кнопку». Удачи!

Часть 0. Для тех, кто не программирует.

Глава 0. Для тех, кто не программирует: запускаем готовый проект за 5 минут.

Вы никогда не писали код. Возможно, вы вообще гуманитарий. Но вам хочется, чтобы на компьютере появился свой собственный ИИ-помощник, который работает без интернета, не отправляет ваши данные в облака и умеет отвечать на вопросы, искать по документам, переводить тексты и даже рисовать картинки. Эта глава — специально для вас.

Я покажу, как запустить готовый проект, не написав ни одной строчки кода. Вам понадобится только умение копировать файлы и дважды щёлкать мышкой. Весь код уже написан за вас — он лежит в репозитории книги. Ваша задача — просто «нажать кнопку».

0.1. Что вам понадобится.

· Компьютер с Windows 10 или 11 (инструкции для Mac и Linux тоже есть, но начнём с Windows).

· Минимум 16 ГБ оперативной памяти (чем больше, тем лучше).

· Около 25 ГБ свободного места на диске.

· Желательно видеокарта NVIDIA с 6+ ГБ памяти (но можно и без неё, просто медленнее).

· Интернет — только для скачивания моделей. После этого интернет не нужен вообще.

0.2. Шаг 1: скачиваем и запускаем скрипт-установщик.

Я подготовил специальный скрипт, который сам создаст все папки, скачает Python, установит библиотеки и модели. Вам нужно только запустить его.

1. Скачайте файл install_all.bat из репозитория книги (ссылка в конце главы).

2. Положите его в любую папку, например C:\LocalAI.

3. Дважды щёлкните по install_all.bat.

4. Дождитесь окончания. На экране будут бежать строки — это нормально. Скрипт скачивает Python, библиотеки и модели. В зависимости от скорости интернета это займёт от 20 минут до часа.

5. Когда увидите сообщение «Всё готово! Запустите LocalAI.exe на рабочем столе», переходите к шагу 2.

0.3. Шаг 2: запускаем приложение.

На рабочем столе появился ярлык **«Локальный ИИ-помощник»**. Дважды щёлкните по нему. Откроется окно с вкладками: Чат, Фото, Перевод, Художник. Всё уже работает — можно начинать пользоваться.

0.4. Что вы можете делать сразу после запуска.

Общаться с ассистентом (вкладка «Чат») Напечатайте вопрос в верхнем поле и нажмите «Отправить». Модель ответит. Можете спрашивать что угодно — она работает как ChatGPT, но полностью офлайн.

Задавать вопросы по своим документам. Положите любые PDF, Word или текстовые файлы в папку C:\Projects\local-ai-one-button\sample_docs. Затем на вкладке «Чат» поставьте галочку «Искать в моих документах» и задайте вопрос по содержимому этих файлов. Модель найдёт ответ и процитирует источник.

Переводить тексты (вкладка «Перевод»). Вставьте текст на любом языке, выберите направление перевода и нажмите «Перевести текст». Можно перевести целый файл или папку с документами.

Реставрировать старые фотографии (вкладка «Фото»). Загрузите старый снимок, отметьте галочками что улучшить (увеличить, восстановить лица, раскрасить) и нажмите «Оживить». Через несколько секунд получите обновлённое изображение.

Рисовать картинки по описанию (вкладка «Художник»). Введите описание, например «кот-космонавт на Марсе», и нажмите «Создать». Через несколько секунд появится изображение.

Говорить голосом. Нажмите кнопку с микрофоном, скажите вопрос — ассистент распознает речь и ответит голосом. Работает без интернета.

0.5. Если что-то пошло не так.

· Не запускается приложение? Запустите C:\Projects\local-ai-one-button\healthcheck.exe (или python healthcheck.py). Он покажет, чего не хватает.

· Модель отвечает медленно? Убедитесь, что на компьютере достаточно свободной оперативной памяти. Закройте другие программы.

· Не работает микрофон? Проверьте, что он подключён и выбран как устройство ввода в настройках Windows.

· Ошибка при скачивании моделей? Проверьте интернет-соединение. Скрипт можно перезапустить — он продолжит с того места, где остановился.

0.6. Что дальше.

Если вы захотите понять, как это всё устроено, и научиться менять поведение ассистента под свои задачи — добро пожаловать в основную часть книги. Начиная с главы 1, мы шаг за шагом разберём каждую строчку кода, и к концу книги вы сможете не только пользоваться, но и создавать собственные AI-инструменты.

А пока — просто пользуйтесь. Ваш личный ИИ готов к работе.

Часть 1. Фундамент: что такое локальный ИИ и почему он возможен

Глава 1.1. Объяснение концепции без математики

Вы наверняка пользовались ChatGPT или подобными сервисами. Вы пишете сообщение — и через секунду получаете осмысленный ответ. Под капотом при этом трудятся тысячи серверов, потребляющих мегаватты энергии. До недавнего времени казалось, что повторить такое на обычном ноутбуке невозможно. Но в 2026 году это стало реальностью. И эта книга научит вас делать собственного ИИ-помощника, работающего **полностью на вашем компьютере, без интернета**.

Чтобы спроектировать такого помощника, нужно понимать, что же такое языковая модель, как она работает и почему для неё не нужен доступ в сеть. При этом нам не потребуются ни математические формулы, ни глубокие знания нейросетей. Достаточно нескольких метафор.

Что значит «языковая модель»?

Представьте себе огромный текстовый файл, содержащий кусочки фраз, идей и шаблонов, подсмотренных в миллионах книг, статей и сайтов. Но это не просто коллекция текстов — из них извлечены **закономерности**. Модель «знает», что после слов «Я люблю» часто идёт «кофе», «читать» или «свою работу» — в зависимости от контекста. Она не понимает смысла в человеческом понимании, но улавливает **статистические связи** между словами на таком уровне, что её ответы выглядят осмысленно.

Более точная метафора — очень умная **автодополнялка текста**, вроде той, что в телефоне предлагает следующее слово при наборе сообщения. Но в тысячи раз сложнее. Языковая модель не просто предлагает одно слово — она может продолжить мысль, написать код, перевести текст, потому что видела подобные цепочки слов в обучающих данных.

Почему это работает на вашем компьютере?

Модели, которые работают в облаке (например, GPT-4), содержат сотни миллиардов параметров. Параметр — это просто число, которое участвует в вычислениях при генерации каждого слова. Чем больше параметров, тем больше памяти и вычислительных мощностей требуется. GPT-4 требует кластер серверов.

Но в 2025–2026 годах случились две важные вещи.

Во-первых, появились **открытые модели** меньшего размера (7–13 миллиардов параметров), обученные на качественных данных. Они уступают гигантам в эрудиции, но для 80% повседневных задач — написать письмо, проанализировать документ, объяснить концепцию — их хватает с избытком.

Во-вторых, было изобретено **квантование**. Представьте, что у вас есть фотография высокого разрешения, занимающая 10 мегабайт. Вы можете сжать её в JPEG размером 500 килобайт — и на глаз почти не заметите разницы. Квантование делает то же самое с параметрами модели: уменьшает точность чисел с 16 бит до 4–5 бит, отчего модель «худеет» в несколько раз, а качество ответов снижается минимально. В итоге модель, которая раньше требовала 16 ГБ видеопамяти, теперь помещается в 4–6 ГБ и работает на игровой видеокарте или вообще на процессоре.

Почему интернет не нужен?

Когда вы пользуетесь онлайн-сервисом, ваш запрос улетает на удалённый сервер, там обрабатывается, и ответ возвращается обратно. На вашем компьютере ничего не происходит, кроме отправки текста.

Локальный ИИ работает иначе. Вы скачиваете файл модели (обычно 4–8 ГБ в формате GGUF) на свой диск — **один раз**. Дальше программа на Python загружает этот файл в оперативную память или видеопамять, и все вычисления происходят прямо здесь, на вашем процессоре или GPU. Ни один байт ваших данных не покидает компьютер. Вы можете физически отключить интернет-кабель — помощник продолжит работать.

В этом и заключается главное преимущество: **конфиденциальность и независимость**. Ваши личные документы, письма, заметки не передаются ни в какое облако. Вы контролируете всё.

Как модель «думает»: генерация одного слова за раз

Технически модель не пишет ответ целиком. Она генерирует его **последовательно**, токен за токеном. Токен — это не всегда слово, скорее кусочек слова (например, «автомат», «изац», «ия» — три токена для слова «автоматизация»). Модель на каждом шаге вычисляет вероятность для всех возможных токенов из своего словаря (обычно 32 000 – 128 000 вариантов) и выбирает следующий с учётом некоторой случайности. Затем подставляет выбранный токен в конец цепочки и снова вычисляет продолжение. Так получается связный текст.

Когда вы пользуетесь ChatGPT, вы видите, как ответ «печатается». Это не эстетика — это действительно модель выдаёт один токен за другим. И локальная модель делает то же самое, только без задержки на сеть.

Что будет уметь наш помощник

В этой книге мы не просто запустим модель и напишем чат-окошко. Мы снабдим нашего ИИ памятью о ваших документах, голосовым вводом и выводом, а затем упакуем всё в один исполняемый файл с иконкой на рабочем столе. И на всём пути мы будем держаться принципа: **сложность должна быть скрыта**. Читателю не нужно разбираться в нейросетях — нужно уметь программировать на Python и понимать, как собрать готовые компоненты в работающий продукт.

В следующей главе мы поговорим о том, почему именно сейчас, в 2026 году, локальный ИИ стал реальностью для каждого разработчика.

Глава 1.2. Почему 2026 год — переломный для локального ИИ.

Ещё пять лет назад словосочетание «запустить языковую модель на ноутбуке» вызывало у специалистов снисходительную улыбку. Считалось, что для этого нужны серверные стойки, GPU за тысячи долларов и команда инженеров. Сегодня вы можете сделать это на обычном рабочем ноутбуке, и результат будет сравним с онлайн-сервисами. Давайте разберём, какие тектонические сдвиги произошли в индустрии и почему момент для входа в локальный ИИ идеален именно сейчас.

Открытые модели догнали проприетарные

Долгое время лучшие языковые модели были доступны только через платные API: OpenAI, Anthropic, Google. Они не публиковали веса своих моделей, и запустить их локально было невозможно физически. Но в 2023–2025 годах мир увидел целую волну **открытых моделей**: Llama 2, Llama 3, Llama 4 от Meta, Mistral и Mixtral от французской Mistral AI, Qwen 2.5 от Alibaba, DeepSeek и другие. Они распространялись с открытыми весами, что позволяло скачать их и использовать где угодно, в том числе на своём компьютере.

Качество этих моделей росло стремительно. Llama 3 8B, выпущенная в 2024 году, уже соперничала с GPT-3.5 на многих задачах, а к 2026 году модели размером 7–13 миллиардов параметров научились хорошо работать с русским языком, писать код, анализировать документы и вести многошаговые диалоги. Для повседневной работы их хватает с запасом, и они бесплатны.

Квантование превратило гигантов в карманные инструменты

Открытые модели всё ещё были большими: Llama 3 8B в исходном 16-битном виде весила около 16 ГБ и требовала столько же видеопамяти для работы. Это было много для потребительского железа, особенно для ноутбуков.

Но параллельно развивались методы **квантования** — сжатия моделей путём снижения точности чисел. В 2023–2024 годах исследователи предложили форматы GGUF и AWQ, которые позволяли сжать модель в 4–5 раз без катастрофической потери качества. Llama 3 8B в квантованном виде Q4_K_M стала занимать около 5 ГБ и запускалась на видеокарте с 6 ГБ памяти или просто на процессоре с 16 ГБ ОЗУ. Скорость генерации при этом достигала 40–60 токенов в секунду на современном ноутбуке, что быстрее, чем человек читает.

В 2026 году квантование стало стандартом де-факто. Практически все популярные модели сразу публикуются в нескольких квантованных вариантах, и выбор правильного — вопрос пяти минут. Мы разберём этот выбор в главе 2.2.

Инструменты запуска стали зрелыми

Мало иметь файл модели. Нужна программа, которая умеет загружать её, обрабатывать входной текст и выдавать ответ. Долгое время это был либо низкоуровневый код на C++, либо Python-обёртки, требующие танцев с бубном.

К 2026 году ситуация кардинально изменилась. Появились проекты, которые взяли на себя всю сложность:

· **llama.cpp** — написанный на C++ движок для запуска квантованных моделей на процессоре и GPU, оптимизированный до предела. Он работает даже на Raspberry Pi, а на современных ноутбуках выдаёт скорость, близкую к облачным сервисам.

· **llama-cpp-python** — тонкая Python-обёртка над llama.cpp, позволяющая работать с моделью через привычный Python-код, без компиляции и сложной настройки.

· **Ollama** — приложение, которое ставится как обычная программа, скачивает и запускает модели одной командой. Идеально для экспериментов и прототипов.

В этой книге мы будем использовать llama-cpp-python как основу, потому что она даёт полный контроль над моделью из Python-кода и легко встраивается в наши будущие утилиты.

Железо стало готовым

Потребительское железо совершило рывок. В 2026 году типичный ноутбук разработчика имеет:

· 32 ГБ оперативной памяти — достаточно, чтобы загрузить модель и держать рядом векторную базу данных.

· Видеокарту с 8+ ГБ VRAM (NVIDIA RTX 4060/5060, Apple M4 с unified memory) — на ней модель летает.

· Даже на процессоре без выделенной видеокарты модели размером 7–8B работают с приемлемой скоростью (10–20 токенов/с).

Это значит, что локальный ИИ перестал быть уделом гиков с игровыми десктопами. Он стал доступен каждому, кто пишет код на Python.

Запрос на конфиденциальность и независимость

Параллельно с технологическим прогрессом росло осознание проблем облачных сервисов. Компании и частные пользователи столкнулись с несколькими неприятными фактами:

· **Конфиденциальность данных.** Отправляя документы и переписку в облачный ChatGPT, вы теряете контроль над ними. Данные могут использоваться для обучения следующих версий модели, быть доступны администраторам сервиса, а в случае утечки — оказаться в открытом доступе.

· **Регуляторика.** GDPR в Европе, корпоративные политики безопасности, требования военных и государственных структур прямо запрещают передавать определённые данные в облака. Для них «локальный ИИ» — не хотелка, а единственный легальный вариант.

· **Зависимость от провайдера.** Облачный сервис может изменить цены, отключить доступ в вашем регионе, ввести цензуру или просто упасть. Локальная модель работает всегда, пока есть электричество.

Всё это породило огромный спрос на специалистов, умеющих разворачивать и использовать ИИ внутри периметра. Именно этому посвящена наша книга.

Экосистема вокруг локального ИИ созрела.

Ещё один важный признак переломного момента — вокруг локальных моделей выросла экосистема библиотек и инструментов:

· **Векторные базы данных** (ChromaDB, LanceDB, Qdrant) позволяют организовать поиск по документам без внешнего сервера.

· **Библиотеки для RAG** (LangChain, LlamaIndex) научились работать офлайн и поддерживать локальные модели.

· **Инструменты для создания GUI** (Tkinter, PySide) позволяют обернуть всю эту мощь в удобный интерфейс.

· **Средства упаковки** (PyInstaller, Nuitka) превращают Python-приложение вместе с моделью в один исполняемый файл.

Всё это мы будем использовать в книге, чтобы в итоге получить ту самую «одну кнопку».

Резюме.

2026 год — это точка пересечения нескольких графиков: качество открытых моделей пошло вверх, требования к железу пошли вниз, инструменты стали дружелюбными, а запрос на конфиденциальность из нишевого стал массовым. Если вы хотели разобраться в локальном ИИ, но ждали подходящего момента, — он настал. Давайте приступим к делу.

Часть 2. Добываем «мозг»: выбор и скачивание модели

Глава 2.1. Зоопарк моделей в 2026 году: Llama, Mistral, Qwen, Gemma

Прежде чем написать хоть строчку кода, нам нужно ответить на вопрос: **какую модель мы будем использовать?** От этого зависит и производительность, и качество ответов, и требования к железу. В 2026 году выбор открытых моделей огромен, и в этом зоопарке легко заблудиться. Давайте наведём порядок.

Почему именно открытые модели?

Для нашего проекта нужна модель, которую можно:

· Скачать как файл (обычно в формате GGUF),

· Запустить на своём компьютере без подключения к интернету,

· Использовать без юридических ограничений и платных лицензий.

Этим требованиям удовлетворяют только **открытые модели** (open-weight models). Их веса опубликованы в открытом доступе, и сообщество уже перевело их в оптимизированные форматы. Проприетарные модели (GPT-4, Claude, Gemini) мы не рассматриваем — их нельзя скачать и запустить локально.

Ключевые семейства моделей в 2026 году

Расскажу о четырёх основных семействах, с которыми вы почти наверняка столкнётесь. Все они имеют версии размером 7–13 миллиардов параметров — это наш «золотой диапазон» для локального запуска.

Llama (Meta)

История и статус: Семейство от компании Meta (признана экстремистской и запрещена в РФ), начавшееся с Llama 1 в 2023 году. К 2026 году актуальны Llama 3.1, Llama 3.2 и Llama 4. Это своего рода «золотой стандарт» открытых моделей, вокруг которого строится большая часть экосистемы.

Сильные стороны:

· Отличный баланс между качеством и скоростью.

· Превосходный английский язык, очень достойный русский (особенно в версиях 3.1+).

· Хорошо работает с кодом (Python, JavaScript, TypeScript).

· Огромное сообщество: большинство туториалов, библиотек и инструментов тестируются в первую очередь на Llama.

· Множество «файнтюнов» — дообученных версий под конкретные задачи (медицина, юриспруденция, креативное письмо).

Слабые стороны:

· Русский язык в базовых версиях иногда уступает Qwen.

· Требует относительно много памяти для своего размера.

Рекомендуемая версия для книги: Llama 3.1 8B (универсальный выбор) или Llama 4 8B, если она уже стабильна на момент чтения.

Требования к памяти (квантование Q4_K_M):

· ОЗУ: минимум 6 ГБ, комфортно 8+ ГБ.

· VRAM: 6 ГБ (видеокарта уровня RTX 3060/4060).

Mistral и Mixtral (Mistral AI)

История и статус: Французская компания Mistral AI прославилась в 2023 году моделью Mistral 7B, которая при малом размере показывала впечатляющие результаты. В 2024 году они

выпустили Mixtral — «смесь экспертов» (mixture of experts), где из 45B параметров одновременно активны только 12B, что даёт высокое качество при умеренном потреблении памяти.

Сильные стороны:

· Эффективность: Mistral 7B долгое время был лучшим в своём классе по соотношению качество/размер.

· Mixtral 8x7B (и более поздние версии) приближается к качеству моделей 70B, потребляя память как 12B.

· Хорошая поддержка европейских языков (французский, немецкий, испанский).

Слабые стороны:

· Русский язык в базовых версиях часто хуже, чем у Llama и Qwen (требуется проверка на конкретной версии).

· Архитектура Mixtral сложнее в настройке и квантовании, возможны нюансы совместимости.

Рекомендуемая версия: Mistral 7B v0.3 для простых задач и слабого железа; Mixtral 8x7B — если нужно высокое качество и есть 12+ ГБ ОЗУ/VRAM.

Требования к памяти:

· Mistral 7B (Q4_K_M): ОЗУ 6 ГБ, VRAM 4+ ГБ.

· Mixtral 8x7B (Q4_K_M): ОЗУ 12 ГБ, VRAM 8+ ГБ.

Qwen (Alibaba)

История и статус: Семейство Qwen от китайской Alibaba стало открытием 2024–2025 годов, особенно для тех, кому нужна работа с азиатскими языками и русским. Qwen 2.5 и последующие версии славятся отличным мультиязычным качеством.

Сильные стороны:

· **Лучшая поддержка русского языка** среди открытых моделей на начало 2026 года (наравне, а часто и лучше Llama 3.1).

· Отличная работа с китайским, японским, корейским.

· Хорошее следование инструкциям, низкий процент галлюцинаций в фактологических запросах.

· Доступны версии разных размеров: 1.8B, 4B, 7B, 14B, 32B, 72B.

Слабые стороны:

· Меньшая распространённость в западном сообществе: меньше туториалов и файнтюнов.

· Иногда встречаются артефакты на смеси языков (code-switching), особенно в ранних версиях.

Рекомендуемая версия: Qwen 2.5 7B (оптимально для русского языка) или Qwen 2.5 14B (если позволяет железо).

Требования к памяти (Qwen 2.5 7B, Q4_K_M):

· ОЗУ: 6 ГБ, комфортно 8+ ГБ.

· VRAM: 6 ГБ.

Gemma (Google)

История и статус: Семейство от Google, основанное на технологиях Gemini. Gemma 2 (2024) и последующие версии — это лёгкие, быстрые модели, оптимизированные для эффективности.

Сильные стороны:

· Компактность: Gemma 2 9B при своих размерах соперничает с моделями 13B.

· Хорошая интеграция с экосистемой Google (JAX, TensorFlow), но нас интересует совместимость с llama.cpp — она есть.

· Хороший английский, приемлемый русский.

Слабые стороны:

· Русский язык заметно хуже, чем у Qwen и Llama.

· Лицензия может быть ограничительной для коммерческого использования (всегда проверяйте актуальную лицензию на Hugging Face!).

· Меньший выбор квантованных версий и файнтюнов.

Рекомендуемая версия: Gemma 2 9B — если английский в приоритете и нужно сэкономить память.

Требования к памяти (Gemma 2 9B, Q4_K_M):

· ОЗУ: 8 ГБ.

· VRAM: 6+ ГБ.

Сравнение моделей

Llama 3.1 8B — универсальный выбор для большинства задач. 8 миллиардов параметров, отличный баланс между качеством и скоростью. Русский язык поддерживает хорошо, но не идеально. Потребляет 6–8 ГБ ОЗУ в квантовании Q4. Лучше всего подходит как универсальный ассистент и для работы с кодом.

Llama 4 8B — обновлённая версия Llama 3.1 с улучшенной архитектурой. Те же 8 миллиардов параметров и 6–8 ГБ ОЗУ, но качество ответов выше, особенно в сложных рассуждениях. Хороший русский язык. Если выбирать между Llama 3.1 и Llama 4 — берите Llama 4.

Mistral 7B — компактная модель от французской компании Mistral AI. 7 миллиардов параметров, потребляет около 6 ГБ ОЗУ. Хорошо работает с европейскими языками, но русский знает посредственно. Идеальна для слабого железа и задач на английском, французском, немецком.

Mixtral 8x7B — модель с архитектурой «смесь экспертов». Формально содержит 47 миллиардов параметров, но одновременно активны только 12 миллиардов. Качество ответов приближается к моделям класса 70B, при этом потребляет около 12 ГБ ОЗУ. Русский язык на среднем уровне. Хороший выбор для сложных задач на европейских языках.

Qwen 2.5 7B — лучшая модель для русского языка в классе 7 миллиардов параметров. Разработана компанией Alibaba, обучалась на огромном корпусе русского текста. Потребляет 6–8 ГБ ОЗУ. Идеальна, если вам нужен качественный русский язык и поддержка азиатских языков.

Qwen 2.5 14B — старшая версия Qwen с 14 миллиардами параметров. Максимальное качество русского языка среди открытых моделей на начало 2026 года. Требует 12 и более гигабайт ОЗУ. Выбор для тех, кому нужно лучшее качество на русском без компромиссов.

Gemma 2 9B — модель от Google с 9 миллиардами параметров. Оптимизирована для скорости, потребляет около 8 ГБ ОЗУ. Хорошо работает с английским языком, русский знает удовлетворительно. Подойдёт для задач, где важна скорость ответа, а не глубина знаний русского.

Как выбрать? Три сценария

Сценарий 1: «Слабый ноутбук, 8–16 ГБ ОЗУ, без мощной видеокарты»

Ваш выбор: **Llama 3.1 8B (Q4_K_M) или Qwen 2.5 7B (Q4_K_M)**. Обе модели запустятся на процессоре с приемлемой скоростью (10–20 токенов/с) и дадут хорошее качество. Если русский язык критичен — берите Qwen. Если нужна универсальность и совместимость с максимумом инструментов — Llama.

Сценарий 2: «Игровой ноутбук или ПК с видеокартой 8+ ГБ VRAM»

Ваш выбор: **Llama 3.1 8B (Q5_K_M) или Qwen 2.5 14B (Q4_K_M)**. На GPU скорость генерации подскочит до 40–60 токенов/с. Можно позволить менее сжатое квантование (Q5) или модель побольше (14B). Русский язык — Qwen, универсальность — Llama.

Сценарий 3: «Мощная машина, 32 ГБ ОЗУ, 12+ ГБ VRAM»

Ваш выбор: **Llama 4 8B (Q6_K) или Qwen 2.5 14B (Q5_K_M) или Mixtral 8x7B**. Здесь можно экспериментировать с более качественными квантованиями и моделями «смеси экспертов». Качество ответов будет максимальным для локального сегмента.

Что мы выберем для книги?

Для основного проекта я рекомендую **Llama 3.1 8B (квантование Q4_K_M)**. Почему:

· Универсальность: одинаково хорошо работает с текстом и кодом.

· Отличная совместимость: llama.cpp, Ollama, LangChain — всё тестируется на Llama в первую очередь.

· Предсказуемое поведение: вы не столкнётесь с неожиданными артефактами, которые иногда бывают у менее распространённых моделей.

· Качество русского языка достаточное для повседневных задач, а для тех, кому нужен максимальный русский, я буду указывать в сносках, как заменить Llama на Qwen — это будет делаться заменой одной строки в коде.

Итоговый выбор: Llama 3.1 8B Instruct, формат GGUF, квантование Q4_K_M, файл размером ~5 ГБ. Именно её мы будем скачивать в следующей главе.

Впрочем, прочитав эту книгу, вы сможете подставить любую другую модель из таблицы выше — весь код останется точно таким же.

Глава 2.2. Что такое квантование и как его читать

В предыдущей главе мы выбрали модель Llama 3.1 8B в качестве основного «мозга» нашего помощника. Но когда мы зайдём на страницу загрузки, нас встретит не один файл, а десятки — с загадочными суффиксами Q4_K_M, Q5_K_S, Q8_0 и другими. Что это за шифр? Какой файл качать? И почему вообще модель весом 16 ГБ «сжимается» до 5 ГБ без катастрофической потери качества? Давайте разбираться.

Модель до квантования — это очень точные числа

Нейронная сеть внутри языковой модели состоит из миллиардов **весов** — это просто числа, на которые умножаются входные данные. Исходно эти веса хранятся в формате **FP16** (16-битные числа с плавающей точкой). Одно такое число занимает 2 байта. Умножаем 8 миллиардов параметров на 2 байта — получаем 16 ГБ, именно столько весит Llama 3 8B в оригинале.

Для сравнения: формат FP16 может представить число с точностью до 4–5 знаков после запятой. Это избыточно для того, чтобы модель генерировала связный текст. Оказалось, что можно пожертвовать частью точности — и модель почти ничего не потеряет в качестве, зато станет в 3–5 раз компактнее и быстрее.

Что такое квантование: аналогия с музыкой

Представьте, что вы записали симфонический оркестр в студийном качестве 24 бит / 192 кГц. Файл весит несколько гигабайт. А теперь сожмите его в MP3 с битрейтом 320 кбит/с. Разница на слух минимальна, но размер уменьшился в 10 раз.

Квантование модели — это то же самое. Мы берём 16-битные веса и переводим их в 4-битные, 5-битные или 8-битные. Каждое число теперь занимает меньше места, и их суммарный объём резко падает. При этом модель продолжает «играть ту же мелодию» — генерировать осмысленные ответы.

Форматы квантования: GGUF и AWQ

В 2026 году есть два основных формата квантованных моделей:

· **GGUF** (GPT-Generated Unified Format) — наследник GGML, разработан специально для запуска на процессоре и видеокарте через llama.cpp. Именно этот формат мы будем использовать: он универсален, поддерживает множество типов квантования и идеально подходит для Python-обёрток.

· **AWQ** (Activation-aware Weight Quantization) — более продвинутый метод, требующий GPU определённой архитектуры. Даёт чуть лучшее качество при том же размере, но менее гибок. Мы не будем его использовать, чтобы сохранить максимальную совместимость.

Все файлы, которые мы будем скачивать, имеют расширение .gguf.

Расшифровка названий: что значит Q4_K_M

Заглянем на страницу загрузки Llama 3.1 8B на Hugging Face. Мы увидим что-то вроде:
text
llama-3.1-8b-instruct-Q2_K.gguf
llama-3.1-8b-instruct-Q3_K_S.gguf
llama-3.1-8b-instruct-Q4_K_M.gguf
llama-3.1-8b-instruct-Q5_K_M.gguf
llama-3.1-8b-instruct-Q8_0.gguf

Разберём имя по частям на примере Q4_K_M:

· **Q4** — основная битность квантования. Большинство весов модели будут сжаты до 4 бит. Это число определяет главный компромисс: чем оно меньше, тем компактнее модель, но выше риск потери качества.

· **K —** указывает на метод квантования «K-quant», который использует неодинаковую точность для разных слоёв модели. Важные слои получают больше бит, менее важные — меньше. Это значительно улучшает качество по сравнению с равномерным квантованием.

· **M —** размер (Medium). Бывает S (Small), M (Medium), L (Large). Этот суффикс определяет, насколько агрессивно оптимизирован размер файла:

o **S —** максимальное сжатие для данной битности. Файл меньше, но качество чуть ниже.

o **M —** сбалансированный вариант (рекомендуется в большинстве случаев).

o **L —** минимальное сжатие, качество выше, но файл больше.

Таким образом, Q4_K_M — это 4-битное квантование по методу K-quant со средним уровнем сжатия. Золотой стандарт для локального запуска.

Варианты квантования и когда их использовать

Q2_K — максимальное сжатие. Модель Llama 3.1 8B занимает около 3.5 ГБ и потребляет 4 ГБ ОЗУ. Качество заметно страдает: модель чаще ошибается, путается в длинных рассуждениях. Используйте только для экстремальной экономии памяти, например, на Raspberry Pi или очень старом ноутбуке.

Q3_K_S — умеренное сжатие. Размер около 4 ГБ, потребление 5 ГБ ОЗУ. Качество приемлемое для простых задач: перевод, базовые вопросы, несложный код. Подойдёт для очень слабого железа, когда Q4 не влезает.

Q4_K_S — хороший баланс в сторону скорости. Занимает 4.5 ГБ, потребляет 5.5 ГБ ОЗУ. Качество хорошее, работает быстрее чем Q4_K_M за счёт меньшего размера. Выбирайте, если у вас слабое железо и важен приоритет скорости над качеством.

Q4_K_M — золотой стандарт. Модель весит около 5 ГБ, потребляет 6 ГБ ОЗУ. Качество очень хорошее, разница с несжатой моделью едва заметна в большинстве задач. Это наш выбор для книги — оптимальный баланс размера, скорости и качества.

Q5_K_M — повышенное качество. Занимает 6 ГБ, потребляет 7 ГБ ОЗУ. Качество отличное, модель реже ошибается в сложных логических цепочках. Используйте, если у вас есть запас по памяти и хочется выжать максимум из 8B-модели.

Q6_K — превосходное качество. Размер 7 ГБ, потребление 8 ГБ ОЗУ. Дальнейшее увеличение битности почти не даёт прироста качества — это «порог насыщения». Рекомендуется для GPU с 8 ГБ VRAM, если вы хотите лучшее качество без перехода на более крупную модель.

Q8_0 — почти как оригинал. Занимает 9 ГБ, потребляет 10 ГБ ОЗУ. Качество практически неотличимо от несжатой 16-битной модели. Используйте только на GPU с 12 и более гигабайтами VRAM, когда размер не имеет значения.

Практическое правило: если не знаете, что выбрать — берите Q4_K_M. Это универсальный вариант, который работает на большинстве устройств и даёт отличное качество.

Как квантование влияет на качество: что вы реально заметите

Пользователи часто боятся квантования, думая, что «сжатая» модель будет глупее. Давайте внесём ясность.

На уровне **Q4_K_M** разница с оригинальной 16-битной моделью практически незаметна в бытовых диалогах, ответах на вопросы, написании писем и кода. Модель может изредка подбирать чуть менее точный синоним или немного сбиваться в сложных логических цепочках, но для 95% повседневных задач вы не заметите разницы.

На уровне **Q2_K** деградация уже ощутима: модель чаще «плавает», путается в длинных рассуждениях, может генерировать грамматические ошибки. Это крайний вариант для тех, у кого совсем нет памяти.

На уровне **Q5_K_M и выше** качество стабилизируется: дальнейшее повышение битности даёт всё меньший прирост. Это аналог «порога насыщения» — как в музыке, где битрейт 320 кбит/с уже неотличим от lossless для большинства слушателей.

Практический совет: как выбрать квантование под своё железо

1. **Определите доступную память.** Сколько свободной ОЗУ или VRAM вы готовы выделить под модель? Не забывайте, что операционная система, среда разработки и прочие программы тоже потребляют память. Ориентируйтесь на доступные гигабайты, а не на общий объём.

2. **Выберите битность.** Найдите в таблице выше квантование, чьё потребление памяти укладывается в ваш лимит с запасом хотя бы 1–2 ГБ.

3. **Всегда предпочитайте K-quant.** Суффиксы с буквой K (Q4_K_M, Q5_K_M) почти всегда лучше старых форматов без K (например, Q4_0), потому что используют неравномерное распределение битов по слоям.

4. **Начните с Q4_K_M.** Если вы не знаете, с чего начать, — начните с Q4_K_M. Это универсальный выбор, который работает на большинстве устройств. Позже вы всегда сможете скачать более качественную или более лёгкую версию и просто подменить файл — код не изменится.

Что мы будем использовать в книге

Для основного проекта мы скачаем **Llama 3.1 8B Instruct, формат GGUF, квантование Q4_K_M**. Файл будет весить около 5 ГБ. Это даст нам:

· Отличное качество ответов, неотличимое от онлайн-аналогов в большинстве задач,

· Запуск на ноутбуке с 8 ГБ ОЗУ или 6 ГБ VRAM,

· Скорость генерации 15–30 токенов/с на процессоре и до 60 токенов/с на видеокарте,

· Запас по памяти для RAG и других компонентов.

Если ваше железо отличается, вы сможете выбрать другой вариант по таблице — все инструкции останутся точно такими же, заменится только имя файла.

Теперь, когда мы понимаем, что за файл нам нужен, пора его скачать. В следующей главе мы напишем код, который сделает это одной командой.

Глава 2.3. Практикум: скачиваем модель одной командой

Мы выбрали модель, разобрались с квантованием и теперь знаем, какой файл нам нужен: llama-3.1-8b-instruct-Q4_K_M.gguf. Пришло время получить его на свой диск. Конечно, можно открыть Hugging Face в браузере, найти страницу модели, кликнуть по ссылке и ждать, пока браузер скачает 5 ГБ. Но мы — разработчики, и пойдём другим путём: напишем Python-функцию, которая сделает это **одной командой**.

Почему это важно? Потому что в будущем наш ИИ-помощник должен уметь сам загружать модель при первом запуске. Пользователь просто запускает программу, а она, обнаружив отсутствие файла, сама скачивает его из интернета. Никаких инструкций «зайдите на сайт, нажмите...» — всё автоматизировано. Наша «одна кнопка» начинается уже здесь.

Предварительные требования

Нам понадобится Python (у вас он уже установлен, версия 3.10 или выше) и библиотека huggingface_hub. Она умеет взаимодействовать с Hugging Face — крупнейшим хранилищем моделей и датасетов. Установим её:

bash

pip install huggingface_hub

Пошаговая инструкция

Вся магия будет заключена в одной функции download_model(). Вот что она должна делать:

1. Принимать имя модели (например, "unsloth/Llama-3.1-8B-Instruct") и имя файла ("Llama-3.1-8B-Instruct-Q4_K_M.gguf").

2. Проверять, существует ли уже такой файл локально. Если да — пропускать скачивание.

3. Если файла нет — загружать его, показывая прогресс и скорость.

4. Возвращать путь к скачанному файлу, чтобы остальной код мог его открыть.

Создайте в вашем проекте файл model_loader.py и напишите следующий код:

python# model_loader.pyimport osimport sysfrom pathlib import Pathfrom huggingface_hub import hf_hub_downloaddef download_model(repo_id: str = "unsloth/Llama-3.1-8B-Instruct", filename: str = "Llama-3.1-8B-Instruct-Q4_K_M.gguf", model_dir: str = "./models") -> str: """ Скачивает файл модели GGUF с Hugging Face, если его ещё нет локально. Аргументы: repo_id: идентификатор репозитория на Hugging Face (например, "unsloth/Llama-3.1-8B-Instruct") filename: имя файла модели в репозитории model_dir: локальная папка для сохране-

ния моделей Возвращает: Абсолютный путь к файлу модели на диске """ # Создаём папку, если её нет Path(model_dir).mkdir(parents=True, exist_ok=True) # Полный путь, где должен лежать файл local_path = Path(model_dir) / filename # Если файл уже есть — просто возвращаем путь if local_path.exists(): print(f" Модель уже загружена: {local_path}") return str(local_path.resolve()) # Иначе — скачиваем print(f" Загрузка модели {filename} из {repo_id}...") print(" Размер файла ~5 ГБ. Первая загрузка может занять несколько минут.") try: # Основная функция загрузки downloaded_path = hf_hub_download(repo_id=repo_id, filename=filename, local_dir=model_dir, resume_download=True, # Можно продолжить при обрыве связи) print(f" Модель загружена в: {downloaded_path}") return downloaded_path except KeyboardInterrupt: print("\n Загрузка прервана пользователем. При повторном запуске она продолжится с того же места.") sys.exit(1) except Exception as e: print(f" Ошибка при загрузке: {e}") print(" Проверьте интернет-соединение и правильность repo_id/filename.") sys.exit(1)if __name__ == "__main__": # Для теста: просто запустите этот файл, и модель начнёт качаться path = download_model() print(f"\nГотово! Путь к модели: {path}")

Объяснение кода

· from huggingface_hub import hf_hub_download — это функция, которая делает всю тяжёлую работу: находит файл в репозитории, скачивает его, проверяет контрольные суммы, умеет продолжать прерванную загрузку.

· **Проверка** if local_path.exists() — ключевая оптимизация. Модель весит 5 ГБ, и мы не хотим качать её заново при каждом запуске. Функция сначала смотрит, есть ли файл на диске. Если есть — мгновенно возвращает путь.

· resume_download=True — спасение для медленного интернета. Если соединение оборвётся, при следующем запуске загрузка начнётся не с нуля, а с того же места. huggingface_hub сохраняет временный файл и автоматически докачивает недостающие байты.

· **Обработка** KeyboardInterrupt — вежливость к пользователю, который нажал Ctrl+C. Мы сообщаем ему, что прогресс не потерян и при повторном запуске всё продолжится.

Тестирование

Откройте терминал в папке проекта и выполните:

bashpython model_loader.pyВы увидите примерно такой вывод:text Загрузка модели Llama-3.1-8B-Instruct-Q4_K_M.gguf из unsloth/Llama-3.1-8B-Instruct... Размер файла ~5 ГБ. Первая загрузка может занять несколько минут. Downloading: 100%|| 5.12G/5.12G [05:30<00:00, 15.5MB/s] Модель загружена в: models/Llama-3.1-8B-Instruct-Q4_K_M.ggufГотово! Путь к модели: models/Llama-3.1-8B-Instruct-Q4_K_M.ggufПри повторном запуске:text Модель уже загружена: models/Llama-3.1-8B-Instruct-Q4_K_M.ggufГотово! Путь к модели: models/Llama-3.1-8B-Instruct-Q4_K_M.gguf

Что мы получили

В папке models/ нашего проекта теперь лежит файл модели. Это полностью автономный, самодостаточный «мозг» нашего будущего ИИ. Его можно скопировать на флешку, перенести на другой компьютер, и он будет работать без каких-либо дополнительных манипуляций.

Возможные проблемы и их решение

Ошибка «Repo not found». Возникает, когда вы указали неверный идентификатор репозитория. Проверьте название на сайте huggingface.co — возможно, вы ошиблись в имени пользователя или названии модели. Для квантованных моделей часто используются репозитории сообщества: unsloth, bartowski, TheBloke, mradermacher. Именно там лежат готовые GGUF-файлы, а не в официальных репозиториях разработчиков.

Ошибка «Entry not found». Означает, что запрошенный файл не найден в репозитории. Либо вы опечатались в имени файла, либо модель удалили или переименовали. Откройте страницу репозитория в браузере и проверьте актуальный список файлов — имена могли измениться с выходом новой версии.

Очень медленная загрузка. Скорость может падать из-за ограничений вашего провайдера или загруженности серверов Hugging Face. Для пользователей из некоторых регионов помогает переключение на зеркало: установите переменную окружения HF_ENDPOINT=https://hf-mirror.com. Это особенно актуально для Китая, Ирана и некоторых других стран.

Не хватает места на диске. Квантованная модель Llama 3.1 8B весит около 5 ГБ, но при загрузке через hf_hub_download создаются временные файлы. Суммарно может потребоваться до 6 ГБ свободного места. Перед загрузкой проверьте диск и при необходимости освободите место.

Качает не тот файл. Если после загрузки модель не работает или ведёт себя странно, возможно, вы скачали не то квантование. Сравните имя файла с тем, что указано на странице репозитория. Лучше скопировать имя файла прямо с сайта, а не вводить вручную — это исключит опечатки.

Зачем мы написали это сами, а не использовали Ollama

Вы наверняка слышали про Ollama — удобный инструмент, где модель скачивается командой ollama pull llama3.1:8b. Почему мы не пошли этим путём?

Дело в том, что Ollama — это отдельный сервер, который работает в фоне и общается с нашим кодом через HTTP. Это добавляет лишнее звено, усложняет отладку и делает невозможным упаковку всего в один EXE-файл (нам пришлось бы таскать с собой ещё и Ollama). Мы же хотим максимального контроля: модель должна быть просто файлом, который наш Python-код загружает напрямую. Так мы сможем встроить её в любое приложение без внешних зависимостей.

В следующей главе мы возьмём скачанный файл и «оживим» его — напишем класс, который будет отправлять в модель запросы и получать ответы.

Резюме

· Мы установили huggingface_hub — мост между нашим кодом и миром открытых моделей.

· Написали функцию download_model(), которая качает файл один раз и повторно использует его.

· Функция умеет продолжать загрузку при обрыве связи и сообщать о прогрессе.

· Файл модели лежит в папке models/ и готов к использованию.

Теперь у нас есть «мозг». В Части 3 мы заставим его думать.

Часть 3. Сердце: запуск модели как локального сервера

Глава 3.1. Сравнение рантаймов: Ollama, llama.cpp, llama-cpp-python

У нас в руках файл модели — мозг будущего помощника. Теперь нужно выбрать «тело»: программу, которая загрузит этот файл в память, примет наш вопрос, пропустит его через нейросеть и вернёт ответ. В экосистеме локального ИИ сложились три основных способа это сделать. Они не исключают друг друга, но для нашей цели — создания единого компактного приложения — подходит только один. Давайте сравним претендентов.

Что такое «рантайм» для LLM

Рантайм (runtime) — это среда исполнения модели. Это программа или библиотека, которая берёт файл модели GGUF и «оживляет» его: распределяет вычисления между ядрами процессора или видеокарты, управляет памятью, обрабатывает входные токены и собирает выходные. Без рантайма файл модели — просто набор байтов. С рантаймом он становится собеседником.

Все современные рантаймы для локальных LLM так или иначе основаны на проекте **llama.cpp** — высокопроизводительном движке на C++, написанном специально для запуска квантованных моделей. Но формы, в которых этот движок доходит до нас, разработчиков на Python, различаются.

Вариант 1: Нативный llama.cpp (C++)

Это исходный проект: движок, написанный на C++, без каких-либо обёрток. Вы скачиваете исходники, компилируете их в исполняемый файл и запускаете модель из командной строки:

bash

./llama-cli -m models/Llama-3.1-8B-Instruct-Q4_K_M.gguf -p "Привет, мир!"

Как взаимодействовать из Python: только через запуск внешнего процесса (subprocess). Вы отправляете текст в stdin и читаете ответ из stdout.

Плюсы:

· Максимальная производительность (нативный C++ без прослоек).

· Первым получает все новые фичи (поддержка новых архитектур, квантований, бэкендов).

· Минимальное потребление памяти (нет накладных расходов на скриптовый язык).

Минусы:

· Необходимость компиляции под каждую платформу.

· Нет нормального API для Python: нужно парсить текстовый вывод, обрабатывать ошибки вручную.

· Неудобно встраивать в приложение: придётся таскать с собой скомпилированный бинарник.

· Сложность с потоковой генерацией (streaming).

Вердикт: Отличный инструмент для экспериментов и максимальной скорости, но для книги не подходит — нам нужна тесная интеграция с Python-кодом.

Вариант 2: Ollama

Ollama — это готовое приложение, которое устанавливается как обычная программа (есть версии для Windows, macOS, Linux). Оно запускает фоновый сервер, скачивает модели по запросу и предоставляет REST API, совместимое с OpenAI.

Работа с Ollama выглядит так:

bash

ollama pull llama3.1:8b

ollama run llama3.1:8b

А из Python мы обращаемся к ней через HTTP:

python

import requests

response = requests.post("http://localhost:11434/api/generate", json={

"model": "llama3.1:8b",

"prompt": "Привет, мир!"

})

Плюсы:

· Установка и запуск в одну команду — очень beginner-friendly.

· Автоматически скачивает модели (не нужен наш код из главы 2.3).

· Совместимость с OpenAI API: можно использовать библиотеки openai для Python, просто указав base_url на localhost.

· Удобный CLI и веб-интерфейс (через плагины).

Минусы:

· Это **отдельный сервер**, который должен быть запущен. Наше приложение без него не работает.

· Невозможно упаковать в один EXE вместе с нашим кодом. Пользователю придётся устанавливать и Ollama, и нашу программу.

· Меньше контроля: Ollama сама управляет памятью, кэшированием, выбором бэкенда. Мы не можем тонко настроить параметры инференса.

· Некоторые фичи (например, передача сырых токенов или доступ к логитам) недоступны через простой API.

Вердикт: Прекрасный инструмент для прототипирования и личного использования. Если вы просто хотите «поиграться с локальной моделью», возьмите Ollama. Но для книги, цель которой — создать автономное приложение с «одной кнопкой», он не годится из-за внешней зависимости.

Вариант 3: llama-cpp-python (наш выбор)

Это Python-биндинг к llama.cpp. По сути, llama.cpp скомпилирован как динамическая библиотека, а llama-cpp-python предоставляет Python-интерфейс ко всем его функциям. Установка проста:

bash

pip install llama-cpp-python

Использование — чистый Python:

python

from llama_cpp import Llama

model = Llama(model_path="models/Llama-3.1-8B-Instruct-Q4_K_M.gguf")

response = model.create_chat_completion(

messages=[{"role": "user", "content": "Привет, мир!"}]

)

print(response["choices"][0]["message"]["content"])

Плюсы:

· **Всё в одном процессе.** Модель загружается прямо в память Python-приложения. Нет отдельных серверов, HTTP-запросов, сетевых задержек.

· **Полный контроль.** Доступны все параметры llama.cpp: температура, top_p, частотные штрафы, контекстное окно, бэкенд (CPU/CUDA/Metal/Vulkan). Можно даже получать сырые логиты.

· **Совместимость с OpenAI API.** create_chat_completion принимает тот же формат сообщений, что и облачные сервисы. Меняется только одна строка — вместо openai.OpenAI(api_key=...) мы создаём Llama(model_path=...).

· **Потоковая генерация.** Встроенная поддержка streaming через generator.

· **Упаковка в EXE.** PyInstaller может включить llama-cpp-python и сам движок llama.cpp в один исполняемый файл. Никаких внешних бинарников не требуется.

· **Активное сообщество.** Библиотека обновляется практически синхронно с llama.cpp, поддерживает все новые архитектуры и квантования.

Минусы:

· Установка с поддержкой GPU требует указания дополнительных флагов (разберём в главе 3.2).

· Чуть больше потребление памяти, чем у нативного llama.cpp (разница в пределах 5–10%).

· При очень большом количестве запросов в секунду может уступать серверным решениям (но для персонального помощника это не важно).

Вердикт: Идеальный выбор для нашей книги. lloma-cpp-python даёт нам полный контроль над моделью из Python-кода, не требует внешних серверов и позволяет упаковать всё в один файл.

Сравнение рантаймов для запуска моделей

llama.cpp (нативный)

Установка требует компиляции из исходников на C++ — не самый простой процесс для новичка. Интегрируется с Python через запуск внешнего процесса и парсинг вывода, что неудобно и ненадёжно. Не требует отдельного сервера, даёт полный контроль над параметрами, но потоковую генерацию организовать сложно. Совместимости с OpenAI API нет. При упаковке в EXE придётся таскать с собой скомпилированный бинарник. Поддержка GPU настраивается при компиляции. Лучше всего подходит для бенчмарков, кастомных клиентов и задач, где важна максимальная производительность.

Ollama

Устанавливается как обычная программа через готовый установщик — самый простой способ начать. Предоставляет HTTP API, к которому можно обращаться из любого языка программирования. Главный минус — требует постоянно запущенного фонового сервера, что делает невозможной упаковку в один EXE-файл. Контроль над параметрами модели ограничен — Ollama сама управляет памятью и кэшированием. Потоковая генерация встроена, совместимость с OpenAI API частичная. GPU поддерживается из коробки. Идеальный выбор для личного использования, прототипирования и экспериментов.

llama-cpp-python

Устанавливается одной командой pip install. Это нативные Python-биндинги к llama.cpp, которые работают в том же процессе, что и ваше приложение — никаких отдельных серверов. Даёт полный контроль над параметрами, поддерживает потоковую генерацию и совместимость с OpenAI API через метод create_chat_completion. Можно упаковать в EXE через PyInstaller без внешних зависимостей. Поддержка GPU включается флагами при установке. Это наш выбор для книги — идеальный баланс контроля, удобства и возможности распространения готового приложения.

Что мы выбираем и почему

Для книги я выбираю **llama-cpp-python**. Вот главные причины:

1. **Единый процесс.** Модель — это просто объект Python. Никаких «запустите сервер Ollama перед использованием». Наше приложение самодостаточно.

2. **Одна кнопка.** Когда мы упакуем проект в EXE, пользователь не должен будет ничего устанавливать дополнительно. Двойной щелчок — и программа работает. С Ollama это невозможно, с нативным llama.cpp — крайне сложно.

3. **Контроль.** Мы сможем тонко настроить параметры инференса, управлять памятью, выбирать бэкенд (CPU или GPU) в зависимости от того, что доступно на машине пользователя. Ollama многие из этих решений принимает за нас, и не всегда удачно.

4. **Привычный API.** create_chat_completion с тем же форматом сообщений, что и у OpenAI. Если вы когда-нибудь писали код для ChatGPT, вы уже знаете, как работать с нашей локальной моделью.

Когда всё-таки стоит использовать Ollama

Я не хочу создавать впечатление, что Ollama — плохой инструмент. Для определённых сценариев он идеален:

· Вы экспериментируете с разными моделями и не хотите писать код для скачивания.

· Вы используете готовые приложения (Open WebUI, AnythingLLM), которые ожидают Ollama как бэкенд.

· Вам нужно быстро поднять API для команды разработчиков.

Но для создания автономного, упакованного в EXE ИИ-помощника — только llama-cpp-python.

Что дальше

В следующей главе мы установим llama-cpp-python, в том числе с поддержкой GPU, и напишем первый код, который «оживит» нашу модель. Мы создадим класс-обёртку LocalModel, который станет фундаментом всего дальнейшего проекта.

Глава 3.2. Установка и настройка llama-cpp-python

Мы выбрали инструмент. Теперь нужно установить его так, чтобы он работал быстро и без сюрпризов. Установка llama-cpp-python — это не просто pip install. Чтобы получить максимальную производительность (особенно если у вас есть видеокарта), нужно указать правильные флаги. В этой главе мы пройдём установку для всех основных платформ и проверим, что всё работает.

Что мы устанавливаем на самом деле

llama-cpp-python — это Python-обёртка. Под капотом она использует скомпилированную библиотеку llama.cpp. При установке через pip происходит компиляция этой библиотеки прямо на вашем компьютере. Поэтому процесс зависит от того, какие инструменты сборки у вас есть и какое железо вы хотите использовать.

Есть два основных варианта установки:

· **Только CPU.** Самый простой, работает везде. Модель будет использовать процессор и оперативную память.

· **С поддержкой GPU.** Требует драйверов и SDK от производителя видеокарты. Модель будет использовать видеопамять, что в разы быстрее.

Универсальный способ: установка только для CPU

Этот вариант подходит, если у вас нет мощной видеокарты или вы не хотите возиться с драйверами. Он работает на Windows, macOS и Linux.

bash

pip install llama-cpp-python

Всё. Библиотека скомпилируется с параметрами по умолчанию и будет использовать CPU. Никаких дополнительных действий не требуется. Если у вас macOS с Apple Silicon (M1/M2/M3/M4), этот способ автоматически задействует ускорение через Accelerate (Apple Neural Engine), что уже даст неплохую скорость.

Установка с поддержкой GPU: когда скорость важна

Если у вас есть дискретная видеокарта, вы можете задействовать её для инференса. Это даст прирост скорости в 3–10 раз.

Для видеокарт NVIDIA (CUDA)

Вам потребуются:

· Драйверы NVIDIA (любые современные, ставятся с сайта nvidia.com).

· CUDA Toolkit 12.x (скачивается с developer.nvidia.com/cuda-downloads).

Установите CUDA Toolkit, затем выполните:

bash

CMAKE_ARGS="-DGGML_CUDA=on" pip install llama-cpp-python

Переменная CMAKE_ARGS передаёт флаг -DGGML_CUDA=on компилятору. Он включает поддержку CUDA в llama.cpp. Компиляция займёт несколько минут.

Проверка: после установки запустите Python и выполните:

python

from llama_cpp import Llama

 # Если модель загрузилась без ошибки "CUDA not available", всё работает.

Если вы видите предупреждение GGML_CUDA=off, значит, флаг не применился. Удалите библиотеку (pip uninstall llama-cpp-python) и повторите установку.

Для macOS с Apple Silicon (Metal)

На компьютерах Mac с чипами M1/M2/M3/M4 есть встроенное GPU-ускорение через Metal. В большинстве случаев CPU-установка уже использует Accelerate, но можно явно включить Metal для максимальной производительности:

bash

CMAKE_ARGS="-DGGML_METAL=on" pip install llama-cpp-python

После установки модель будет использовать встроенную графику Apple, что заметно быстрее чистого CPU.

Для видеокарт AMD (Vulkan)

Карты AMD поддерживаются через бэкенд Vulkan:

bash

CMAKE_ARGS="-DGGML_VULKAN=on" pip install llama-cpp-python

Перед этим убедитесь, что у вас установлен Vulkan SDK (vulkan.lunarg.com).

Выбор способа установки под ваше железо.

Только процессор (Intel/AMD). Самый простой и универсальный вариант. Установка одной командой pip install llama-cpp-python без дополнительных флагов. Модель будет использовать все ядра процессора, скорость генерации составит от низкой до средней — примерно 5–15 токенов в секунду для 8B-модели. Подходит, если у вас нет дискретной видеокарты или вы не хотите возиться с драйверами.

NVIDIA GeForce/RTX. Если у вас видеокарта NVIDIA с поддержкой CUDA, вы можете ускорить модель в разы. Команда установки: CMAKE_ARGS="-DGGML_CUDA=on" pip install llama-cpp-python. Перед этим убедитесь, что установлены драйверы NVIDIA и CUDA Toolkit. Скорость будет высокой — 30–60 токенов в секунду, модель загрузится в видеопамять и освободит процессор для других задач.

Apple M1/M2/M3/M4. На компьютерах Mac с процессорами Apple Silicon можно использовать встроенное GPU-ускорение через Metal. В большинстве случаев достаточно обычной команды pip install llama-cpp-python — система автоматически задействует Accelerate. Для максимальной производительности можно указать флаг -DGGML_METAL=on. Скорость будет от средней до высокой, в зависимости от поколения чипа и объёма unified memory.

AMD Radeon. Видеокарты AMD поддерживаются через бэкенд Vulkan. Команда установки: CMAKE_ARGS="-DGGML_VULKAN=on" pip install llama-cpp-python. Предварительно потребуется установить Vulkan SDK с сайта vulkan.lunarg.com. Скорость средняя или высокая, в зависимости от модели карты и объёма видеопамяти.

Важно: Если вы не уверены, какое у вас железо — просто выполните pip install llama-cpp-python без флагов. Модель заработает в любом случае, просто на процессоре. Позже всегда можно переустановить с флагами под GPU.

Проверяем установку: пишем «Hello, world» для модели

Создайте в проекте файл test_model.py и напишите минимальный код для проверки:

```
python
# test_model.py
from model_loader import download_model
from llama_cpp import Llama
# 1. Скачиваем модель ( если ещё нет )
model_path = download_model()
# 2. Загружаем модель
print("Загрузка модели в память...")
model = Llama(
model_path=model_path,
n_ctx=2048, # размер контекстного окна (сколько токенов модель «помнит»)
n_threads=4, # количество потоков CPU (поставьте своё число ядер)
verbose=False # убираем технический вывод
)
# 3. Отправляем запрос
print("Генерация ответа...")
response = model.create_chat_completion(
messages=[
{"role": "system", "content": "Ты — полезный ассистент. Отвечай кратко."},
{"role": "user", "content": "Что такое локальный ИИ одним предложением?"}
],
temperature=0.7,
```

```
max_tokens=100
)
# 4. Печатаем ответ
answer = response["choices"][0]["message"]["content"]
print(f"\nОтвет модели:\n{answer}")
```

Запустите:

```
bash
python test_model.py
```

Если всё настроено правильно, вы увидите что-то вроде:

```
text
Загрузка модели Llama-3.1-8B-Instruct-Q4_K_M.gguf из unsloth/Llama-3.1-8B-Instruct...
Модель уже загружена: models/Llama-3.1-8B-Instruct-Q4_K_M.gguf
Загрузка модели в память...
Генерация ответа...
Ответ модели:
Локальный ИИ — это искусственный интеллект, работающий непосредственно на вашем устройстве без подключения к интернету и передачи данных в облачные серверы.
```

Что означают параметры в Llama()

· model_path — путь к файлу модели. Мы получаем его из нашей функции download_model().

· n_ctx — размер контекстного окна в токенах. Это максимальная длина диалога, которую модель «помнит». Для начала 2048 токенов достаточно (это около 1500 слов). Позже, при подключении RAG, увеличим до 4096 или 8192.

· n_threads — число потоков CPU, которые будет использовать модель. Поставьте количество физических ядер вашего процессора. Для 4-ядерного процессора — 4, для 8-ядерного — 8.

· verbose=False — отключает подробный лог загрузки. Если что-то пойдёт не так, можно включить обратно (True) для диагностики.

Возможные ошибки и их решение

ModuleNotFoundError: No module named 'llama_cpp'. Библиотека не установлена в текущем виртуальном окружении. Убедитесь, что вы активировали venv и выполнили pip install llama-cpp-python. Если библиотека установлена, но ошибка остаётся — проверьте, что вы находитесь в правильном окружении: в начале строки терминала должен быть префикс (venv).

ImportError: DLL load failed. Проблема возникает на Windows при отсутствии рантаймов Visual C++. Скачайте и установите Microsoft Visual C++ Redistributable с официального сайта Microsoft. После установки перезагрузите компьютер — ошибка должна исчезнуть.

RuntimeError: CUDA error: no CUDA-capable device is detected. Вы установили версию с поддержкой CUDA, но видеокарта NVIDIA не обнаружена. Либо у вас нет карты NVIDIA, либо не установлены драйверы. Решение: установите драйверы с сайта nvidia.com, или переустановите llama-cpp-python без флага CUDA для работы на процессоре.

Illegal instruction (core dumped). Возникает на Linux, когда процессор не поддерживает инструкции AVX2, которые llama.cpp использует по умолчанию. Обычно это случается на очень старых процессорах. Добавьте флаг -DGGML_NATIVE=off при установке — это соберёт библиотеку с базовыми инструкциями, совместимыми со всеми процессорами.

Очень медленная генерация — 1–2 токена в секунду. Скорее всего, модель работает на процессоре в однопоточном режиме. Проверьте параметр n_threads при создании объекта Llama — он должен быть равен количеству физических ядер вашего процессора. Если у вас есть видеокарта, переустановите библиотеку с поддержкой GPU — скорость вырастет в разы.

Ошибка памяти — out of memory. Модель не помещается в доступную оперативную или видеопамять. У вас два варианта: выбрать более сильное квантование (например, Q3_K_M вместо Q4_K_M — файл будет меньше), или взять модель меньшего размера (7B вместо 8B). Также закройте другие программы, занимающие память — браузеры, IDE, мессенджеры.

Как переключить бэкенд в коде

По умолчанию llama-cpp-python использует тот бэкенд, который был включён при компиляции. Но можно явно указать, сколько слоёв модели загружать на GPU, а сколько оставить на CPU. Это полезно, если видеопамяти не хватает на всю модель:

```
python
model = Llama(
model_path=model_path,
n_ctx=2048,
n_threads=4,
n_gpu_layers=20 # Первые 20 слоёв — на GPU, остальные — на CPU
)
```

Параметр n_gpu_layers принимает число от 0 (всё на CPU) до -1 (всё на GPU, если хватает памяти). Поэкспериментируйте с этим значением, чтобы найти баланс между скоростью и стабильностью для вашего железа.

Что мы получили

Теперь в нашем проекте есть:

· Файл model_loader.py — умеет скачивать модель.

· Файл test_model.py — умеет загружать модель и получать от неё ответ.

Мы готовы к следующему шагу: написать полноценный класс-обёртку LocalModel, который станет фундаментом для всех будущих возможностей (чат, потоковая генерация, RAG, голос). Этим мы займёмся в главе 3.3.

Глава 3.3. Пишем класс-обёртку LocalModel с методами chat() и stream().

Мы установили llama-cpp-python, проверили, что модель отвечает на запросы в тестовом скрипте. Теперь пора превратить этот разрозненный код в аккуратный, переиспользуемый компонент. Мы создадим класс LocalModel, который станет «сердцем» нашего проекта. Все будущие возможности — чат, работа с документами, голос — будут опираться именно на него.

Зачем нужна обёртка

Прямые вызовы Llama() из llama-cpp-python работают, но у них есть недостатки:

· Каждый раз приходится передавать десятки параметров (путь к модели, количество потоков, температуру).

· Нет удобного управления историей диалога.

· Код, который работает с моделью, перемешан с бизнес-логикой приложения.

· Трудно тестировать и заменять реализацию (вдруг мы захотим переключиться на другую библиотеку).

Наш класс LocalModel решит эти проблемы. Он будет:

· Хранить загруженную модель и её конфигурацию,

· Предоставлять простые методы chat() и stream_chat(),

· Следить за историей диалога (опционально),

· Давать тот же формат сообщений, что и OpenAI API, чтобы к нему привыкли все, кто писал под ChatGPT.

Код класса LocalModel

Создайте в проекте файл local_model.py. Ниже — полный код класса с подробными комментариями.

```
python
# local_model.py
from typing import List, Dict, Optional, Generator
from llama_cpp import Llama
class LocalModel:
"""
Обёртка над llama-cpp-python, предоставляющая удобный интерфейс
для общения с локальной языковой моделью.
"""
def __init__(
self,
model_path: str,
n_ctx: int = 2048,
n_threads: int = 4,
n_gpu_layers: int = 0,
verbose: bool = False
):
"""
Инициализация модели.
:param model_path: путь к файлу .gguf
:param n_ctx: размер контекстного окна (максимальное количество токенов,
которое модель «помнит»)
:param n_threads: количество потоков процессора
```

```
:param n_gpu_layers: сколько слоёв загрузить на GPU (-1 = все, 0 = ничего)
:param verbose: выводить ли подробный лог загрузки
"""
self.model_path = model_path
self.n_ctx = n_ctx
# Загружаем модель в память
self._llm = Llama(
model_path=model_path,
n_ctx=n_ctx,
n_threads=n_threads,
n_gpu_layers=n_gpu_layers,
verbose=verbose
)
# Хранилище истории диалогов (ключ — id беседы)
self._histories: Dict[str, List[Dict[str, str]]] = {}
def chat(
self,
messages: List[Dict[str, str]],
temperature: float = 0.7,
max_tokens: int = 512,
system_prompt: Optional[str] = None
) -> str:
"""
Отправляет сообщения модели и возвращает её ответ.
:param messages: список сообщений в формате OpenAI:
[{"role": "user", "content": "..."}, ...]
:param temperature: креативность (0 — строго, до 1.5 — фантазия)
:param max_tokens: максимальная длина ответа
:param system_prompt: системный промпт (если не указан в messages)
:return: строка с ответом модели
"""
# Если передан system_prompt, добавляем его в начало списка
full_messages = []
if system_prompt:
full_messages.append({"role": "system", "content": system_prompt})
full_messages.extend(messages)
response = self._llm.create_chat_completion(
messages=full_messages,
temperature=temperature,
max_tokens=max_tokens
)
return response["choices"][0]["message"]["content"]
def stream_chat(
self,
messages: List[Dict[str, str]],
temperature: float = 0.7,
max_tokens: int = 512,
system_prompt: Optional[str] = None
) -> Generator[str, None, None]:
```

```
"""
Потоковая версия chat(). Возвращает генератор, который выдаёт
токены по мере их генерации. Позволяет отображать ответ
«на лету», как в ChatGPT.
:param messages: список сообщений
:param temperature: креативность
:param max_tokens: максимальная длина ответа
:param system_prompt: системный промпт
:yield: кусочки текста (токены)
"""
full_messages = []
if system_prompt:
full_messages.append({"role": "system", "content": system_prompt})
full_messages.extend(messages)
stream = self._llm.create_chat_completion(
messages=full_messages,
temperature=temperature,
max_tokens=max_tokens,
stream=True
)
for chunk in stream:
# В каждом чанке может быть одна или несколько «дельт»
if "choices" in chunk and len(chunk["choices"]) > 0:
delta = chunk["choices"][0].get("delta", {})
content = delta.get("content", "")
if content:
yield content
def save_history(self, conversation_id: str, messages: List[Dict[str, str]]):
"""Сохраняет историю диалога под указанным id."""
self._histories[conversation_id] = messages.copy()
def load_history(self, conversation_id: str) -> List[Dict[str, str]]:
"""Загружает историю диалога по id. Если нет — пустой список."""
return self._histories.get(conversation_id, [])
def clear_history(self, conversation_id: str):
"""Удаляет историю диалога."""
if conversation_id in self._histories:
del self._histories[conversation_id]
```

Объяснение ключевых моментов

Конструктор __init__

Мы передаём путь к модели и настройки производительности. Важно, что модель загружается **один раз** при создании объекта LocalModel и остаётся в памяти до его уничтожения. Это экономит время при множественных запросах. Параметр n_ctx определяет «память» модели: чем он больше, тем более длинные диалоги она может вести, но тем больше потребляется памяти. Для начала 2048 токенов достаточно. n_gpu_layers мы подробно обсудили в предыдущей главе.

Метод chat()

Принимает список сообщений в формате OpenAI — таком же, какой используют облачные API. Это значит, что вы можете взять любой код, написанный для ChatGPT, заменить создание клиента на наш LocalModel, и всё продолжит работать. Внутри мы добавляем систем-

ный промпт, если он передан, и вызываем create_chat_completion. Ответ возвращается целиком, в виде строки.

Метод stream_chat()

Потоковая генерация — это когда ответ появляется по одному слову, как в ChatGPT. Вместо того чтобы ждать полного ответа, мы получаем токены по мере их создания. В Python это реализуется через **генератор** (ключевое слово yield). Внешний код может делать так:

python

for token in model.stream_chat(messages):

print(token, end="", flush=True)

Это создаёт эффект «печатания» текста и делает интерфейс более отзывчивым. Мы будем использовать этот метод при создании GUI.

Управление историей

Класс содержит словарь _histories, который хранит списки сообщений по идентификаторам бесед. Это примитивная, но работающая система для многопользовательского или многозадачного режима. Позже мы заменим её на постоянное хранение (файлы или базу данных), но для старта достаточно.

Практический пример: ведём диалог

Создайте файл chat_demo.py:

python

chat_demo.py

from model_loader import download_model

from local_model import LocalModel

1. Скачиваем и загружаем модель

model_path = download_model()

print("Загрузка модели...")

model = LocalModel(model_path=model_path, n_threads=4)

2. Системный промпт (необязательно, но полезно)

system = "Ты — ассистент-повар. Отвечай коротко, предлагай простые рецепты."

3. Диалог в цикле

conversation_id = "cooking"

messages = [] # Начнём с пустой истории

while True:

user_input = input("\nВы: ")

if user_input.lower() in ["выход", "quit", "exit"]:

break

Добавляем сообщение пользователя в историю

messages.append({"role": "user", "content": user_input})

Получаем ответ (потоково)

print("Повар: ", end="", flush=True)

full_response = ""

for token in model.stream_chat(

messages=messages,

system_prompt=system,

temperature=0.7

):

print(token, end="", flush=True)

full_response += token

print() # Перевод строки после ответа

Сохраняем ответ в историю

```
messages.append({"role": "assistant", "content": full_response})
model.save_history(conversation_id, messages)
```

Запустите:

```
bash
python chat_demo.py
```

Пример диалога:

```
text
```

Вы: Как сварить яйцо?

Повар: Всмятку — 4 минуты в кипящей воде. Вкрутую — 8-10 минут. Охладите под холодной водой, чтобы легче чистились.

Вы: А яичницу?

Повар: Разогрейте сковороду с маслом, разбейте яйца, жарьте 2-3 минуты. Для глазуньи накройте крышкой на минуту, чтобы белок схватился сверху.

Управление памятью: что делать, если диалог слишком длинный

Контекстное окно n_ctx ограничивает суммарное количество токенов, которые модель может «видеть» одновременно. Если диалог становится слишком длинным, нужно обрезать историю. Простейший способ — оставить только последние N сообщений. Мы добавим метод trim_history в наш класс (можно дополнить local_model.py):

```
python
def trim_history(self, messages: List[Dict[str, str]], max_messages: int = 20) -> List[Dict[str, str]]:
"""
Оставляет только последние max_messages сообщений.
Системный промпт (первое сообщение с role='system') сохраняется всегда.
"""
if len(messages) <= max_messages:
return messages
# Сохраняем системный промпт, если он есть
system_messages = [m for m in messages if m["role"] == "system"]
other_messages = [m for m in messages if m["role"] != "system"]
# Обрезаем остальные
trimmed = other_messages[-max_messages:]
return system_messages + trimmed
```

Возможные проблемы и их решения.

Модель «забывает», о чём говорили раньше. Это происходит, когда диалог превышает размер контекстного окна n_ctx. Llama.cpp автоматически обрезает историю, и начало разговора теряется. Решение простое: увеличьте n_ctx при создании модели — например, с 2048 до 4096 токенов. Учитывайте, что каждый токен контекста потребляет дополнительную память. Если увеличить n_ctx нельзя из-за нехватки ОЗУ, используйте метод trim_history() — он вручную оставляет только последние N сообщений, сохраняя системный промпт.

Потоковый вывод «дёргается» или зависает. Обычно проблема в слишком маленьком параметре max_tokens — модель упирается в лимит и останавливается на полуслове. Увеличьте max_tokens до 500 или 1000. Другая возможная причина — буферизация вывода. Убедитесь, что при печати токенов используется flush=True: print(token, end="", flush=True). Это заставляет Python выводить текст немедленно, а не копить в буфере.

Ответ содержит бред на незнакомую тему. Модель не обучалась на ваших специфических данных и пытается «угадать» ответ, порождая галлюцинации. Первое, что стоит сделать — добавить системный промпт, ограничивающий тематику: «Ты — ассистент по договорному праву. Если вопрос не по теме, скажи об этом». Если этого недостаточно — подключите RAG из Части 4 книги: модель будет искать ответ в ваших документах, а не выдумывать.

Ошибка «KeyError: 'choices'». Изредка возникает при потоковой генерации, особенно если соединение было прервано. Llama.cpp возвращает chunk без ожидаемого ключа. Решение — добавить защитную проверку перед обработкой каждого чанка: if "choices" in chunk and len(chunk["choices"]) > 0:. Это гарантирует, что код не упадёт на некорректном ответе модели.

Что дальше.

Теперь у нас есть полноценный, переиспользуемый класс для работы с локальной LLM. Мы можем вставлять его в любой проект и общаться с моделью через простой Python-интерфейс. В следующей главе мы сделаем так, чтобы этот интерфейс стал доступен **по сети** — через HTTP-сервер, совместимый с OpenAI API. Это позволит подключаться к нашему локальному ИИ из любых приложений, которые умеют работать с ChatGPT.

Глава 3.4. Делаем API совместимым с OpenAI

Мы написали класс LocalModel, который умеет отвечать на вопросы в нашем Python-коде. Но что, если мы захотим подключить к нашему локальному ИИ какое-нибудь готовое приложение? Например, плагин для VS Code, веб-интерфейс, мобильное приложение или даже другой скрипт, написанный под ChatGPT? Переписывать всё под наш класс — тупиковый путь.

К счастью, мир уже договорился о стандарте. OpenAI предоставляет HTTP API, которое принимает запросы определённого формата и возвращает ответы. Сотни инструментов и библиотек умеют работать с этим API. Если мы сделаем свой сервер, который «притворяется» OpenAI, то сможем использовать весь этот зоопарк без изменений. Именно этим мы и займёмся в этой главе.

Что мы строим

Мы создадим небольшой HTTP-сервер на Python, который:

· Слушает порт (например, 8080) на вашем компьютере.

· Принимает POST-запросы на адрес /v1/chat/completions — точно такой же, как у OpenAI.

· Перенаправляет эти запросы в наш LocalModel.

· Возвращает ответ в том же JSON-формате, что и OpenAI.

· Поддерживает потоковую генерацию (streaming), чтобы ответ «печатался» на лету.

· Не требует никаких внешних ключей API — он работает строго локально.

После запуска сервера вы сможете, например, использовать официальную Python-библиотеку openai, просто указав base_url="http://localhost:8080/v1". И она будет работать с вашей локальной моделью Llama, как с ChatGPT. Магия!

Инструменты

Для создания HTTP-сервера мы будем использовать **FastAPI**. Это современный, быстрый и простой фреймворк, который идеально подходит для API. Он автоматически генерирует документацию, поддерживает асинхронность и потоковые ответы. Нам также понадобится uvicorn — сервер, который будет запускать наше FastAPI-приложение.

Установим их:

bash

```
pip install fastapi uvicorn
```

Код сервера

Создайте файл openai_api.py в корне проекта. Ниже — полный код с подробными комментариями.

python

```
# openai_api.py
import time
import uuid
from typing import Optional, List, Dict, Any
from fastapi import FastAPI, HTTPException
from fastapi.responses import StreamingResponse
from pydantic import BaseModel, Field
import uvicorn
# Импортируем наш класс LocalModel и функцию загрузки модели
from local_model import LocalModel
from model_loader import download_model
# --- Модели данных, совместимые с OpenAI API ---
class ChatMessage(BaseModel):
```

```
    """Сообщение в диалоге."""
    role: str # "system", "user", "assistant"
    content: str
    class ChatCompletionRequest(BaseModel):
    """Тело запроса, как в OpenAI."""
    model: str = "local-model"
    messages: List[ChatMessage]
    temperature: Optional[float] = 0.7
    max_tokens: Optional[int] = 512
    stream: Optional[bool] = False
    class ChatCompletionChoice(BaseModel):
    """Один вариант ответа."""
    index: int
    message: ChatMessage
    finish_reason: Optional[str] = "stop"
    class ChatCompletionResponse(BaseModel):
    """Полный ответ (не потоковый)."""
    id: str
    object: str = "chat.completion"
    created: int
    model: str
    choices: List[ChatCompletionChoice]
    # --- Инициализация FastAPI и модели ---
    app = FastAPI(title="Local AI API", description="OpenAI-совместимый API для локальной LLM")
    # Загружаем модель один раз при старте сервера
    model_path = download_model()
    print("Загрузка модели в память...")
    local_model = LocalModel(model_path=model_path, n_threads=4, n_gpu_layers=0)
    print("Модель готова к работе.")
    # --- Вспомогательные функции ---
    def messages_to_list(messages: List[ChatMessage]) -> List[Dict[str, str]]:
    """Преобразует Pydantic-сообщения в обычные словари для LocalModel."""
    return [{"role": m.role, "content": m.content} for m in messages]
    def generate_stream_response(messages: List[Dict[str, str]], temperature: float, max_tokens: int):
    """Генератор для потокового ответа в формате OpenAI (Server-Sent Events)."""
    chat_id = f"chatcmpl-{uuid.uuid4().hex[:12]}"
    created = int(time.time())
    # Отправляем токены один за другим
    for token in local_model.stream_chat(
    messages=messages,
    temperature=temperature,
    max_tokens=max_tokens
    ):
    chunk = {
    "id": chat_id,
    "object": "chat.completion.chunk",
    "created": created,
```

```
"model": "local-model",
"choices": [
{
"index": 0,
"delta": {"content": token},
"finish_reason": None
}
]
}
yield f"data: {__import__('json').dumps(chunk)}\n\n"
# Финальный чанк с finish_reason
final_chunk = {
"id": chat_id,
"object": "chat.completion.chunk",
"created": created,
"model": "local-model",
"choices": [
{
"index": 0,
"delta": {},
"finish_reason": "stop"
}
]
}
yield f"data: {__import__('json').dumps(final_chunk)}\n\n"
yield "data: [DONE]\n\n"
# --- Основной endpoint ---
@app.post("/v1/chat/completions", response_model=None)
async def chat_completions(request: ChatCompletionRequest):
"""
Главный endpoint, совместимый с OpenAI Chat Completions API.
Поддерживает как обычный, так и потоковый режим.
"""
# Преобразуем сообщения
messages = messages_to_list(request.messages)
# Если запрошен потоковый режим
if request.stream:
return StreamingResponse(
generate_stream_response(
messages=messages,
temperature=request.temperature or 0.7,
max_tokens=request.max_tokens or 512
),
media_type="text/event-stream"
)
# Обычный режим
try:
response_text = local_model.chat(
messages=messages,
```

```
temperature=request.temperature or 0.7,
max_tokens=request.max_tokens or 512
)
except Exception as e:
raise HTTPException(status_code=500, detail=f"Ошибка генерации: {str(e)}")
# Формируем ответ в стиле OpenAI
response = ChatCompletionResponse(
id=f"chatcmpl-{uuid.uuid4().hex[:12]}",
created=int(time.time()),
model=request.model,
choices=[
ChatCompletionChoice(
index=0,
message=ChatMessage(role="assistant", content=response_text),
finish_reason="stop"
)
]
)
return response
@app.get("/v1/models")
async def list_models():
"""Возвращает список доступных моделей (для совместимости)."""
return {
"object": "list",
"data": [
{
"id": "local-model",
"object": "model",
"created": int(time.time()),
"owned_by": "local"
}
]
}
if __name__ == "__main__":
# Запускаем сервер на порту 8080
uvicorn.run(app, host="127.0.0.1", port=8080)
```

Разбор ключевых частей

Модели данных (Pydantic)

Мы используем pydantic.BaseModel для описания структуры запросов и ответов. Это гарантирует, что наш API будет строго соответствовать формату OpenAI. Любое приложение, которое умеет отправлять сообщения в ChatGPT, сможет работать и с нами: поля model, messages, temperature, max_tokens, stream идентичны оригиналу.

Потоковый ответ

Когда клиент запрашивает "stream": true, мы не должны возвращать обычный JSON. Вместо этого мы отправляем **Server-Sent Events** (SSE) — последовательность событий, разделённых префиксом data:. Каждое событие — это небольшой JSON-объект с очередным токеном. В конце мы отправляем data: [DONE], сигнализируя о завершении.

Функция generate_stream_response — это генератор, который yieldит строки. FastAPI оборачивает его в StreamingResponse с правильным media type, и клиент получает ответ по кусочкам, как в ChatGPT.

Загрузка модели при старте

Обратите внимание: LocalModel создаётся один раз, при запуске сервера. Модель загружается в память и остаётся там до выключения. Это значит, что каждый запрос обрабатывается мгновенно, без задержки на загрузку.

Тестирование

Запустите сервер:

bash

```
python openai_api.py
```

Вы увидите:

text

```
Загрузка модели в память...
Модель готова к работе.
INFO: Started server process [12345]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8080 (Press CTRL+C to quit)
```

Теперь протестируем API с помощью curl или Python. Сначала не-потоковый запрос:

python

```
# test_openai_api.py
import requests
response = requests.post(
"http://127.0.0.1:8080/v1/chat/completions",
json={
"model": "local-model",
"messages": [
{"role": "user", "content": "Привет! Кто ты?"}
],
"temperature": 0.7,
"max_tokens": 100,
"stream": False
}
)
print(response.json()["choices"][0]["message"]["content"])
```

Теперь с потоковой передачей:

python

```
import requests
response = requests.post(
"http://127.0.0.1:8080/v1/chat/completions",
json={
"model": "local-model",
"messages": [
{"role": "user", "content": "Расскажи анекдот"}
],
"stream": True
},
stream=True
```

```
)
for line in response.iter_lines():
if line:
line = line.decode("utf-8")
if line.startswith("data: ") and not line.endswith("[DONE]"):
import json
chunk = json.loads(line[6:])
token = chunk["choices"][0]["delta"].get("content", "")
if token:
print(token, end="", flush=True)
print()
```

Использование с официальной библиотекой OpenAI

Теперь самое интересное. Мы можем использовать библиотеку openai, которую ставят для работы с ChatGPT, просто указав свой базовый URL и любой ключ (он не проверяется):

```
bash
pip install openai
python
# test_with_openai_lib.py
from openai import OpenAI
client = OpenAI(
base_url="http://127.0.0.1:8080/v1",
api_key="not-needed" # ключ обязателен для библиотеки, но наш сервер его игнорирует
)
# Не-потоковый запрос
response = client.chat.completions.create(
model="local-model",
messages=[{"role": "user", "content": "Что такое Python?"}],
temperature=0.7,
max_tokens=100
)
print(response.choices[0].message.content)
# Потоковый запрос
stream = client.chat.completions.create(
model="local-model",
messages=[{"role": "user", "content": "Напиши хокку о программировании"}],
stream=True
)
for chunk in stream:
if chunk.choices[0].delta.content:
print(chunk.choices[0].delta.content, end="", flush=True)
print()
```

Безопасность.

Наш сервер слушает только 127.0.0.1 (localhost). Это значит, что он недоступен из сети — только с вашего компьютера. Если вы захотите открыть его для других устройств, замените host="127.0.0.1" на host="0.0.0.0", но **обязательно** добавьте аутентификацию (например, проверку API-ключа). Без этого любой в вашей сети сможет пользоваться вашей моделью.

Возможные проблемы при запуске API.

Ошибка «Address already in use». Порт 8080, который мы используем по умолчанию, уже занят другой программой. Самое простое решение — указать другой порт при запуске: port=8081. Если хотите использовать именно 8080, найдите процесс, занимающий порт, и завершите его. На Windows это делается командой netstat -ano | findstr :8080, затем taskkill / PID номер_процесса /F.

Модель загружается при каждом запросе. Правильно настроенное приложение загружает модель один раз при старте сервера, и все последующие запросы используют уже загруженную модель. Если модель загружается заново при каждом запросе — проверьте код: объект LocalModel должен создаваться глобально, а не внутри функции-обработчика. Каждый новый экземпляр загружает модель в память заново, что занимает десятки секунд.

Потоковый вывод подвисает в середине ответа. Клиент может закрыть соединение, если долго не получает данные. Увеличьте max_tokens, чтобы модель не останавливалась раньше времени. Также попробуйте уменьшить temperature до 0.3–0.5 — при низких значениях модель генерирует более предсказуемый текст и реже «задумывается». Некоторые клиенты имеют встроенный таймаут — проверьте настройки на стороне клиента.

Клиент жалуется на неверный формат ответа. Ваш API должен возвращать ответ в точности как OpenAI: все поля id, object, created, model, choices обязательны. Если какого-то поля не хватает или оно названо иначе, клиент (например, библиотека openai) выдаст ошибку парсинга. Сверьте структуру ответа с эталонной — проще всего сравнить с реальным ответом от OpenAI.

Что мы получили.

Теперь у нас есть полноценный HTTP API, который:

· Работает локально без интернета.

· Совместим с OpenAI Chat Completions API.

· Поддерживает потоковую генерацию.

· Может быть использован с любыми инструментами, поддерживающими OpenAI API (а их сотни).

· Загружает модель один раз и держит в памяти.

Это огромный шаг. Мы превратили наш локальный ИИ в сервис, к которому можно подключаться из любого места на компьютере. В следующих частях мы будем наращивать функциональность (RAG, голос), и API будет расширяться вместе с моделью.

В Части 4 мы займёмся «глазами и памятью» — научим модель работать с вашими личными документами.

Часть 4. Зрение и память: обучаем модель вашим документам

Глава 4.1. Как работает RAG без облаков

Наша локальная модель уже умеет отвечать на вопросы — быстро, конфиденциально, без интернета. Но есть одна проблема, знакомая каждому, кто пользовался ChatGPT: модель знает только то, чему её обучили. Если вы спросите: «Какие задачи по проекту "Альфа" обсуждались на совещании 15 марта?», модель либо откажется отвечать, либо начнёт выдумывать. Она не читала ваши заметки, не видела ваши PDF-файлы, не имеет доступа к корпоративной документации.

Чтобы превратить общую модель в вашего личного эксперта, нам нужен механизм, который свяжет её с вашими документами. Этот механизм называется **RAG — Retrieval-Augmented Generation**, или «генерация, дополненная поиском». И мы реализуем его полностью локально, без единого запроса во внешний мир.

Проблема: модель не знает ваших документов

Языковая модель — это «слепок» знаний из интернета, книг и статей. Она обучалась на публичных данных и понятия не имеет о вашей частной информации. Можно было бы попытаться «дообучить» модель на своих документах, но это сложно, дорого по ресурсам и неудобно: при каждом обновлении документов придётся переучивать модель заново.

RAG решает проблему иначе. Вместо того чтобы вшивать знания в саму модель, мы научим её **искать нужную информацию** в ваших документах и подставлять её в запрос.

Аналогия: библиотекарь с фотографической памятью

Представьте, что у вас есть огромная библиотека с сотнями папок, договоров и отчётов. У вас нет времени читать их все, но у вас есть помощник — библиотекарь. Вы задаёте ему вопрос, он мгновенно находит три-четыре самых релевантных документа, кладёт их перед вами и говорит: «Вот что удалось найти по вашей теме». Вы читаете эти фрагменты и на их основе даёте точный ответ.

В архитектуре RAG:

· **Вы** — это пользователь, который задаёт вопрос.

· **Библиотекарь** — это система поиска (ретривер), которая ищет в базе знаний.

· **Фрагменты документов, которые он вам принёс** — это релевантные «чанки» (куски текста).

· **Ваш мозг, который синтезирует ответ** — это языковая модель (LLM).

Модель не обязана помнить содержимое всех документов. Она просто получает подсказку — и отвечает, опираясь на неё.

Как это работает: четыре шага

Разберём процесс по шагам. На каждом этапе мы будем использовать только локальные инструменты.

Шаг 1: Индексация (загрузка документов)

Сначала мы берём все ваши файлы — PDF, Word, текстовые заметки, Excel-таблицы — и превращаем их в единую базу знаний.

· Каждый документ разбивается на **чанки** (chunks) — небольшие фрагменты по 500–1000 символов. Почему не целиком? Потому что модель имеет ограниченное контекстное окно, и загружать весь 200-страничный договор в промпт невозможно. Чанки — это «абзацы» для быстрого поиска.

· Каждый чанк пропускается через **embedding-модель** — специальную нейросеть, которая превращает текст в вектор (список чисел). Вектор отражает смысл текста. Фразы «собака бежит» и «пёс несётся» получат близкие векторы, а «акции выросли» — совсем другой.

· Все векторы сохраняются в **векторную базу данных**. Это специальное хранилище, которое умеет искать не точные слова, а **смысловую близость**. Мы будем использовать ChromaDB — лёгкую, быструю и не требующую сервера.

Всё это делается один раз (или при обновлении документов) и занимает минуты даже на обычном ноутбуке.

Шаг 2: Поиск (ретрив)

Когда пользователь задаёт вопрос, например: «Какие гарантийные обязательства у нас перед клиентом "Ромашка"?», происходит следующее:

· Вопрос преобразуется в вектор той же embedding-моделью.

· Векторная база данных ищет среди миллионов чанков те, чьи векторы ближе всего к вопросу. Это занимает миллисекунды.

· Возвращаются, скажем, 4 самых релевантных чанка: абзац из договора с «Ромашкой», пункт о гарантиях из общего регламента и две записки с совещаний.

Шаг 3: Дополнение промпта (augmentation)

Найденные чанки вставляются в промпт для языковой модели вместе с вопросом пользователя. Выглядит это примерно так:

text

Ты — ассистент, который отвечает на вопросы на основе предоставленных документов.

Не выдумывай ничего, чего нет в документах. Если ответа нет, скажи "В документах не найдено".

Вопрос: Какие гарантийные обязательства у нас перед клиентом "Ромашка"?

Документы:

--- Документ 1 (Договор с ООО "Ромашка".pdf, раздел 7):

7.1. Исполнитель гарантирует устранение недостатков в течение 30 дней...

--- Документ 2 (Регламент гарантийного обслуживания.docx, пункт 3):

...

Ответ:

Модель видит конкретные факты и может на них опираться. Она не гадает — она анализирует предоставленный текст.

Шаг 4: Генерация ответа

Модель читает промпт, видит документы и генерирует ответ. Ответ может содержать цитаты: «Согласно пункту 7.1 договора с ООО "Ромашка"...» Это не магия — это просто хорошо сформулированный промпт с релевантной информацией.

Почему это работает без облаков

Все компоненты RAG в нашем проекте будут локальными:

1. **Embedding-модель.** Мы будем использовать sentence-transformers с моделями вроде intfloat/multilingual-e5-base или BAAI/bge-base-en. Они весят всего ~250–500 МБ и отлично работают на CPU.

2. **Векторная база данных.** ChromaDB хранит данные в папке на диске, не требует отдельного сервера и прекрасно интегрируется с Python.

3. **Языковая модель.** Наша Llama 3.1, работающая через LocalModel.

4. **Оркестрация.** Весь процесс — от получения вопроса до генерации ответа — будет управляться нашим Python-кодом, без внешних сервисов.

Ваши документы ни на одном этапе не покидают компьютер. Даже embedding-модель, которая превращает текст в векторы, работает локально.

RAG против «загрузки файла в ChatGPT»

Многие онлайн-сервисы предлагают «загрузить PDF и задать вопрос». Это работает, но:

· **Конфиденциальность:** документ отправляется на чужой сервер. Для личной переписки или корпоративных договоров это недопустимо.

· **Ограничения:** у сервисов часто есть лимиты на размер файла и количество запросов.

· **Зависимость:** без интернета сервис недоступен.

Наш локальный RAG лишён этих недостатков. Вы можете индексировать гигабайты документов и задавать вопросы в полной изоляции.

Ограничения RAG, о которых стоит знать

RAG — не волшебная палочка. Вот что нужно понимать заранее:

· **Качество поиска.** Если embedding-модель плохо понимает ваш язык или специфическую терминологию, релевантные чанки могут не найтись. Мы выберем мультиязычную модель, но в узких доменах может потребоваться тонкая настройка.

· **Размер чанков.** Слишком маленькие чанки теряют контекст (фраза «он сказал» бесполезна без предыдущего абзаца). Слишком большие — размывают релевантность и не влезают в контекстное окно. Мы найдём баланс в главе 4.4.

· **Галлюцинации.** Даже с документами модель может иногда «додумывать». Мы будем настраивать промпт так, чтобы она строго придерживалась источников.

Что мы построим в этой части

В следующих главах Части 4 мы шаг за шагом реализуем локальный RAG:

· В **4.2** выберем embedding-модель и научимся векторизовать текст.

· В **4.3** разберёмся с ChromaDB — создадим коллекцию, добавим векторы, выполним поиск.

· В **4.4** напишем функцию индексации папки с документами (PDF, DOCX, TXT).

· В **4.5** реализуем метод ask_documents() — он будет принимать вопрос, искать релевантные чанки, дополнять промпт и возвращать ответ с цитатами.

К концу этой части наш помощник обретёт «зрение и память». Он сможет работать с вашими личными данными так же уверенно, как с общими знаниями, и всё это — полностью офлайн.

Глава 4.2. Выбор локальной embedding-модели (BGE, multilingual-e5, jina)

В предыдущей главе мы разобрали, что RAG требует умения превращать текст в векторы — компактные числовые представления, отражающие смысл. Эту работу выполняет **embedding-модель**. Это отдельная нейросеть, которая специализируется не на генерации текста, а на его «понимании» и сравнении. Сегодня мы выберем такую модель, которая будет работать у нас локально, быстро и качественно, особенно с русским и английским языками.

Что делает embedding-модель

Embedding-модель принимает на вход текст (от одного слова до нескольких абзацев) и возвращает **вектор** — список чисел фиксированной длины, обычно от 384 до 1024 чисел. Главное свойство: тексты, близкие по смыслу, получают векторы, которые находятся близко друг к другу в этом многомерном пространстве. Измеряется близость **косинусным расстоянием** или скалярным произведением.

Например:

· «кошка спит на диване» и «кот дремлет на кушетке» — векторы почти совпадают.

· «кошка спит на диване» и «прибыль компании выросла» — векторы далеки друг от друга.

В мире открытых моделей для эмбеддингов сложилась тройка лидеров: **BGE**, **multilingual-e5** и **jina**. Все они доступны на Hugging Face, работают через библиотеку sentence-transformers и могут быть запущены полностью офлайн.

Критерии выбора для нашей книги

Мы оцениваем модели по следующим критериям:

1. **Качество русского языка.** Наш помощник будет работать с русскоязычными документами. Модель должна хорошо понимать русский текст, включая деловую лексику и юридические формулировки.

2. **Качество английского языка.** Многие технические документы, статьи и переписка ведутся на английском. Хорошая мультиязычная поддержка обязательна.

3. **Размер и скорость.** Модель должна работать на CPU без GPU, занимать немного места (200–500 МБ) и быстро обрабатывать запросы. Помните: embedding-модель будет использоваться и при индексации (сотни документов), и при каждом поиске (доли секунды).

4. **Совместимость.** Модель должна без проблем загружаться через sentence-transformers, поддерживать токенизацию русского текста и работать на Windows, macOS и Linux.

Обзор кандидатов

BGE (BAAI General Embedding)

Семейство моделей от Пекинской академии искусственного интеллекта (BAAI). Актуальные версии: bge-base-en-v1.5 (английский), bge-m3 (мультиязычная).

· **bge-base-en-v1.5**: только английский, размер 438 МБ, размерность векторов — 768. Отличное качество на английском, но русский не поддерживает. Для нашей задачи не подходит.

· **bge-m3**: мультиязычная модель, поддерживает более 100 языков, включая русский. Размер ~2.2 ГБ, размерность — 1024. Качество на русском высокое, но размер великоват для нашего лёгкого локального стека. Она будет медленнее работать на CPU и займёт много места в итоговой упаковке. Кроме того, для наших задач она избыточна: нам не нужна поддержка 100 языков.

Вердикт: качество отличное, но размер и ресурсоёмкость для локального «однокнопочного» решения слишком велики.

Multilingual-E5 (Microsoft/Intfloat)

Семейство multilingual-e5 — разработка исследователей из Microsoft и сообщества. Модели: multilingual-e5-small, multilingual-e5-base, multilingual-e5-large.

· **multilingual-e5-small**: размер ~470 МБ, размерность векторов — 384. Поддерживает более 90 языков, включая русский.

· **multilingual-e5-base**: размер ~1.1 ГБ, размерность — 768. Качество русского языка выше, чем у small.

· **multilingual-e5-large**: размер ~2.2 ГБ, размерность — 1024. Максимальное качество, но высокие требования.

Обе модели (small и base) показывают очень достойный русский язык, особенно в задачах поиска. Важный нюанс: для моделей E5 требуется добавлять префиксы к тексту — "query: " для поисковых запросов и "passage: " для индексируемых документов. Это улучшает качество, но добавляет немного кода.

Вердикт: multilingual-e5-base — очень сильный кандидат: хороший баланс размера (1.1 ГБ) и качества. multilingual-e5-small — компромисс для слабого железа.

Jina AI Embeddings

Компания Jina AI предлагает линейку jina-embeddings-v3, а также более ранние jina-embeddings-v2. Это мультиязычные модели, поддерживающие русский язык.

· **jina-embeddings-v2-base-multilingual**: размер ~550 МБ, размерность — 768. Поддерживает русский, английский, немецкий и ещё несколько языков.

· **jina-embeddings-v3**: новая модель с поддержкой task-specific embeddings (можно управлять размерностью), размер ~1.5 ГБ, качество сопоставимо с BGE-m3.

Особенность Jina — они хорошо работают с длинными документами (до 8192 токенов), что полезно для больших чанков.

Вердикт: jina-embeddings-v2-base-multilingual — компактная, быстрая, с хорошим русским языком. Достойный соперник для E5.

Выбор embedding-модели для RAG

bge-m3. Флагманская модель от BAAI с размерностью вектора 1024 и размером 2.2 ГБ. Отлично работает с русским языком, но для локального использования часто избыточна — высокое качество достигается ценой низкой скорости даже на GPU. На процессоре работает медленно, а занимаемый объём сопоставим с небольшой языковой моделью.

multilingual-e5-small. Самая лёгкая модель в семействе — всего 470 МБ и размерность вектора 384. Хорошо понимает русский язык, работает очень быстро даже на слабом процессоре. Идеальный выбор для Raspberry Pi, старых ноутбуков и ситуаций, когда скорость важнее точности поиска. Качество приемлемое для большинства домашних задач.

multilingual-e5-base. Золотая середина и наш выбор для книги. Занимает 1.1 ГБ, размерность вектора 768. Отличное качество русского языка при умеренном потреблении ресурсов. Скорость на процессоре средняя — индексация папки из сотни документов занимает минуты, а поиск по базе — миллисекунды. Рекомендуется для большинства пользователей.

multilingual-e5-large. Максимальное качество среди multilingual-e5. Размер 2.2 ГБ, размерность 1024. Русский язык на высшем уровне, но скорость низкая, а потребление памяти высокое. Имеет смысл использовать, если у вас мощный компьютер с GPU и вы работаете с очень большими объёмами документов, где важна каждая десятая процента точности поиска.

jina-embeddings-v2-base. Альтернатива от компании Jina AI. Весит 550 МБ при размерности 768 — легче чем e5-base при той же размерности. Особенность модели — поддержка длинных текстов до 8192 токенов, что вдвое больше стандартных 512. Хорошо подходит, если ваши документы длинные и вы не хотите резать их на мелкие чанки.

Практический совет: Начните с multilingual-e5-base — это оптимальный баланс качества и скорости для большинства сценариев. Если не хватает скорости — переключитесь на e5-small. Если не хватает качества — попробуйте e5-large или jina. Замена embedding-модели не требует переписывания кода — достаточно изменить одну строку при создании объекта Embedder.

Что мы выбираем для книги

Я выбираю intfloat/multilingual-e5-base. Почему:

· **Качество русского языка** — одно из лучших среди моделей такого размера. По тестам на русскоязычных бенчмарках она превосходит аналоги.

· **Размер 1.1 ГБ** — приемлемо для включения в дистрибутив нашего приложения. Модель не занимает гигабайты, оставляя место для основной LLM.

· **Совместимость** — работает из коробки с sentence-transformers, не требует GPU, стабильна на всех платформах.

· **Поддержка префиксов** — мы будем использовать "query: " и "passage: " для повышения точности поиска. Это небольшое усложнение, которое окупается качеством.

Если ваше железо очень слабое (8 ГБ ОЗУ и меньше), вы можете заменить её на multilingual-e5-small — вся остальная архитектура останется той же, поменяется только одна строка с именем модели.

Установка и тестирование

Нам понадобится библиотека sentence-transformers. Установим её вместе с зависимостями:

bash

pip install sentence-transformers

Теперь напишем код для загрузки модели и векторизации текста. Создайте файл embedder.py:

python

```
# embedder.py
from sentence_transformers import SentenceTransformer
class Embedder:
"""
Обёртка над sentence-transformers для удобной работы с эмбеддингами.
"""
def __init__(self, model_name: str = "intfloat/multilingual-e5-base"):
print(f"Загрузка embedding-модели {model_name}...")
self._model = SentenceTransformer(model_name)
self._dim = self._model.get_sentence_embedding_dimension()
print(f"Модель загружена, размерность векторов: {self._dim}")
@property
def dimension(self) -> int:
"""Размерность векторов."""
return self._dim
def embed_query(self, text: str) -> list:
"""Преобразует поисковый запрос в вектор."""
# Для E5 моделей рекомендуется добавлять префикс "query: "
return self._model.encode(f"query: {text}", normalize_embeddings=True).tolist()
def embed_documents(self, texts: list[str]) -> list[list]:
"""Преобразует список документов (чанков) в список векторов."""
# Для документов добавляется префикс "passage: "
prefixed = [f"passage: {t}" for t in texts]
return self._model.encode(prefixed, normalize_embeddings=True).tolist()
if __name__ == "__main__":
emb = Embedder()
# Тест: векторизуем русский запрос и несколько чанков
query_vec = emb.embed_query("Какие гарантийные обязательства?")
doc_vecs = emb.embed_documents([
"Гарантийный срок на оборудование составляет 12 месяцев.",
"Стороны обязуются уведомлять друг друга об изменении реквизитов.",
"В случае нарушения сроков оплаты начисляется пеня 0.1% за каждый день."
])
print(f"Запрос векторизован, размерность: {len(query_vec)}")
print(f"Документов векторизовано: {len(doc_vecs)}")
```

Запустим:

```
bash
python embedder.py
```

При первом запуске модель скачается (около 1 ГБ) и загрузится. Вы увидите:

```
text
Загрузка embedding-модели intfloat/multilingual-e5-base...
Модель загружена, размерность векторов: 768
Запрос векторизован, размерность: 768
Документов векторизовано: 3
```

Теперь мы умеем превращать любой текст в векторы. В следующей главе мы организуем их хранение и поиск с помощью ChromaDB.

Возможные проблемы с embedding-моделью

Модель не скачивается — ошибка сети. При первом создании объекта Embedder() библиотека sentence-transformers пытается скачать модель с Hugging Face. Если интернет недоступен, загрузка прервётся с ошибкой. Решение: скачайте модель заранее на машине с интернетом, скопируйте папку с файлами в models/multilingual-e5-base/ и укажите локальный путь в конструкторе. После этого модель будет загружаться офлайн.

Ошибка нехватки памяти при загрузке. Embedding-модель загружается в оперативную память целиком. multilingual-e5-base требует около 1.1 ГБ, что обычно не проблема. Но если у вас компьютер с 4 ГБ ОЗУ или одновременно запущена языковая модель, памяти может не хватить. Переключитесь на multilingual-e5-small — она весит 470 МБ, вдвое легче, и при этом всё ещё хорошо понимает русский язык.

Векторы для похожих фраз получаются разными. Модель чувствительна к префиксам. Для поисковых запросов нужно добавлять "query: ", а для индексируемых документов — "passage: ". Если перепутать префиксы или забыть их, векторы будут неинформативными, и поиск перестанет работать. Также убедитесь, что включена нормализация — в sentence-transformers она обычно включена по умолчанию при использовании косинусной метрики.

Медленная векторизация на процессоре. Индексация сотни документов на CPU может занять несколько минут — это нормально. Для ускорения можно передать device="cuda" в конструктор SentenceTransformer, тогда векторизация будет использовать видеокарту и ускорится в разы. Однако для большинства домашних задач CPU-скорости вполне достаточно, а экономия видеопамяти важнее.

Что дальше

У нас есть embedding-модель, готовая превращать текст в векторы. Теперь нужно куда-то эти векторы сохранять и уметь быстро искать среди них ближайшие. Этим займёмся в главе **4.3 «Векторная база данных на коленке: ChromaDB и LanceDB»**.

Глава 4.3. Векторная база данных на коленке: ChromaDB и LanceDB

Мы научились превращать текст в векторы. Теперь нам нужно их где-то хранить и, главное, уметь быстро находить векторы, ближайшие к запросу. Для этого существуют **векторные базы данных** — специализированные хранилища, заточенные под поиск по смысловой близости.

В облачном мире есть Pinecone, Weaviate, Qdrant. Они мощные, но многие требуют подключения к интернету или отдельного сервера. Нам же нужна база данных, которая:

· Работает **полностью локально**, как обычная папка с файлами.

· Не требует отдельного серверного процесса (никаких Docker, systemd).

· Умеет искать быстро — миллисекунды на запрос.

· Поддерживает русский язык (точнее, векторы, сгенерированные нашей embedding-моделью).

· Бесплатна и open-source.

В 2026 году есть два идеальных кандидата под эти требования: **ChromaDB** и **LanceDB**. Сравним их и выберем тот, на котором построим наш RAG.

ChromaDB: простота и Python-first

ChromaDB — векторная база данных, написанная с прицелом на простоту и использование из Python. Она хранит данные в файлах на диске, запускается как библиотека внутри вашего процесса и не требует никакой настройки.

Установка:

```
python
import chromadb
# Создаём клиент (данные хранятся в папке ./chroma_data)
client = chromadb.PersistentClient(path="./chroma_data")
# Создаём коллекцию (аналог таблицы в SQL)
collection = client.get_or_create_collection(
name="my_docs",
metadata={"hnsw:space": "cosine"} # метрика близости — косинусное расстояние
)
# Добавляем векторы с текстами и метаданными
collection.add(
embeddings=[[0.1, 0.2, ...], [0.3, 0.4, ...]], # векторы
documents=["Текст первого чанка", "Текст второго чанка"],
metadatas=[{"source": "file1.pdf", "page": 1}, {"source": "file2.docx"}],
ids=["doc1_chunk0", "doc2_chunk0"]
)
# Ищем ближайшие к запросу
results = collection.query(
query_embeddings=[[0.15, 0.25, ...]],
n_results=3
)
```

Базовое использование:

Плюсы ChromaDB:

· **Максимальная простота.** Установка в одну команду, API интуитивно понятный.

· **Никакого сервера.** База данных — это просто папка с файлами. PersistentClient работает синхронно, в том же процессе Python.

· **Метаданные.** Можно хранить вместе с вектором текст документа, источник, номер страницы и любые другие поля. Очень удобно для показа цитат.

· **Встроенная поддержка embedding-функций.** ChromaDB умеет сама вызывать sentence-transformers (но мы будем использовать свой Embedder для контроля).

· **Активное сообщество.** ChromaDB быстро развивается, много примеров и документации.

Минусы:

· **Зависимость от SQLite3.** Под капотом ChromaDB использует SQLite для хранения метаданных. Это надёжно, но накладывает ограничения на параллельную запись.

· **Потребление памяти.** При больших коллекциях (сотни тысяч чанков) может требовать значительной ОЗУ для индекса HNSW.

LanceDB: быстрая и легковесная альтернатива

LanceDB — векторная база данных нового поколения, построенная на колоночном формате хранения Lance. Она заточена под минимальное потребление памяти и высокую скорость.

Установка:

bash

```
pip install lancedb
```

Базовое использование:

python

```
import lancedb
import pyarrow as pa
# Подключаемся к папке (аналог PersistentClient)
db = lancedb.connect("./lancedb_data")
# Создаём таблицу со схемой
schema = pa.schema([
("vector", pa.list_(pa.float32(), 768)),
("text", pa.string()),
("source", pa.string()),
])
table = db.create_table("my_docs", schema=schema, mode="overwrite")
# Добавляем данные
table.add([{
"vector": [0.1, 0.2, ...],
"text": "Текст чанка",
"source": "file1.pdf"
}])
# Ищем
results = table.search([0.15, 0.25, ...]).limit(3).to_list()
```

Плюсы LanceDB:

· **Очень высокая скорость.** LanceDB использует колоночное хранение и эффективный формат файлов. Поиск работает быстрее, чем в ChromaDB, особенно на больших объёмах.

· **Низкое потребление памяти.** Индекс строится лениво и занимает меньше ОЗУ.

· **Поддержка Apache Arrow.** Данные хранятся в формате Arrow, что упрощает интеграцию с дата-инженерными инструментами.

· **Без сервера.** Так же как и ChromaDB, LanceDB — это просто библиотека и файлы.

Минусы:

· **Менее зрелая экосистема.** Документация и примеров меньше, чем у ChromaDB.

· **API менее «питонячий».** Требуется работать со схемами PyArrow, что может быть непривычно для новичков.

· **Меньше встроенных фич.** Нет встроенной поддержки embedding-функций, метаданные нужно описывать вручную.

Сравнение ChromaDB и LanceDB

Установка. Обе библиотеки устанавливаются одной командой pip install. ChromaDB не требует дополнительных зависимостей для базового использования. LanceDB использует формат Apache Arrow и может попросить установить pyarrow отдельно.

Хранение данных. ChromaDB хранит векторы и метаданные в обычной папке, используя SQLite для метаданных и собственный формат для индексов. Это просто и надёжно — можно скопировать папку на другой компьютер, и база заработает. LanceDB использует колоночный формат Lance, оптимизированный для скоростного доступа к большим объёмам данных.

API и простота использования. ChromaDB спроектирована с прицелом на Python-разработчика: интуитивно понятные методы add, query, count. Метаданные можно передать как обычный словарь, не описывая схему заранее. LanceDB требует описания схемы через PyArrow — это мощнее и строже, но сложнее для начинающих.

Метаданные. В ChromaDB метаданные хранятся как произвольный JSON-словарь рядом с каждым вектором — можно добавить любые поля без изменения структуры базы. В LanceDB метаданные нужно объявлять как колонки в схеме при создании таблицы, и изменить схему потом сложнее. Для быстрого прототипирования подход ChromaDB удобнее.

Производительность. На коллекциях до ста тысяч документов обе библиотеки работают быстро, разница незаметна. ChromaDB может замедляться при очень больших объёмах из-за SQLite под капотом. LanceDB изначально проектировалась для высоких нагрузок и сохраняет скорость даже на миллионах записей.

Зрелость и сообщество. ChromaDB существует дольше, у неё обширная документация, множество примеров и ответов на Stack Overflow. LanceDB моложе и развивается стремительно, но информации пока меньше. Для книги мы выбрали ChromaDB именно из-за зрелости и простоты: читатель быстрее найдёт ответ на возникший вопрос.

Встроенные эмбеддинги. ChromaDB умеет автоматически векторизовать текст при добавлении — можно не использовать отдельный Embedder. Но мы в книге делаем это вручную для полного контроля. В LanceDB такой функции нет.

Поддержка операционных систем. Обе библиотеки кроссплатформенны и работают на Windows, macOS и Linux без ограничений.

Итог: ChromaDB — лучший выбор для старта и небольших проектов. LanceDB — когда вы упираетесь в производительность на больших данных. Для масштабов личной базы знаний разница несущественна.

Что мы выбираем для книги

Я выбираю **ChromaDB**. Вот почему:

1. **Простота API.** Нам не нужно отвлекаться на схемы PyArrow и ручное описание колонок. ChromaDB работает «из коробки» — меньше кода, меньше шансов на ошибку.

2. **Гибкие метаданные.** Мы сможем хранить рядом с вектором текст чанка, имя исходного файла, номер страницы, дату документа — всё, что нужно для показа цитат. В LanceDB для этого пришлось бы явно объявлять колонки.

3. **Достаточная производительность.** Для масштабов личной базы знаний (тысячи, десятки тысяч документов) ChromaDB работает быстро, и разница с LanceDB будет незаметна.

4. **Активное сообщество и документация.** Если вы столкнётесь с проблемой, ответ с большей вероятностью найдётся для ChromaDB.

Если в будущем вы решите масштабировать систему на сотни тысяч документов, миграция на LanceDB или Qdrant будет несложной — интерфейс поиска у всех векторных баз похож.

Практикум: включаем ChromaDB в наш проект

Создадим модуль vector_store.py, который будет отвечать за работу с базой:

```
python
# vector_store.py
import chromadb
from typing import List, Dict, Optional
class VectorStore:
"""
Обёртка над ChromaDB для хранения и поиска эмбеддингов документов.
"""
def __init__(self, collection_name: str = "my_documents", persist_path: str = "./
chroma_data"):
self._client = chromadb.PersistentClient(path=persist_path)
self._collection = self._client.get_or_create_collection(
name=collection_name,
metadata={"hnsw:space": "cosine"}
)
def add_documents(
self,
ids: List[str],
embeddings: List[List[float]],
documents: List[str],
metadatas: Optional[List[Dict]] = None
):
"""Добавляет документы (чанки) в коллекцию."""
self._collection.add(
ids=ids,
embeddings=embeddings,
documents=documents,
metadatas=metadatas
)
def search(self, query_embedding: List[float], n_results: int = 5) -> Dict:
"""Ищет ближайшие документы по вектору запроса."""
return self._collection.query(
query_embeddings=[query_embedding],
n_results=n_results
)
def count(self) -> int:
"""Возвращает количество документов в коллекции."""
return self._collection.count()
def clear(self):
"""Очищает коллекцию (полезно для переиндексации)."""
# Удаляем и создаём заново
name = self._collection.name
self._client.delete_collection(name)
self._collection = self._client.get_or_create_collection(
name=name,
metadata={"hnsw:space": "cosine"}
)
```

Протестируем, объединив с нашим Embedder из главы 4.2:

```
python
# test_vector_store.py
from embedder import Embedder
from vector_store import VectorStore
# 1. Загружаем embedding-модель
emb = Embedder()
# 2. Создаём хранилище
store = VectorStore()
# 3. Добавим несколько тестовых чанков
texts = [
"Гарантийный срок на оборудование составляет 12 месяцев с даты поставки.",
"Стороны обязаны уведомлять друг друга об изменении реквизитов.",
"В случае просрочки оплаты начисляется пеня 0.1% за каждый день просрочки.",
"Поставщик гарантирует устранение недостатков в течение 30 календарных дней."
]
embeddings = emb.embed_documents(texts)
store.add_documents(
ids=[f"doc_{i}" for i in range(len(texts))],
embeddings=embeddings,
documents=texts,
metadatas=[{"source": "test.txt", "chunk": i} for i in range(len(texts))]
)
print(f"Документов в коллекции: {store.count()}")
# 4. Ищем по вопросу
query = "Какие гарантийные обязательства у поставщика?"
query_vec = emb.embed_query(query)
results = store.search(query_vec, n_results=2)
print("\nРезультаты поиска:")
for i, (doc, meta, dist) in enumerate(zip(
results["documents"][0],
results["metadatas"][0],
results["distances"][0]
)):
print(f"{i+1}. [distance: {dist:.3f}] {doc}")
print(f" Источник: {meta['source']}\n")
```

Запустим:

```
bash
python test_vector_store.py
```

Вывод:

```
text
Загрузка embedding-модели intfloat/multilingual-e5-base...
Модель загружена, размерность векторов: 768
Документов в коллекции: 4
Результаты поиска:
1. [distance: 0.123] Поставщик гарантирует устранение недостатков в течение 30 календарных дней.
Источник: test.txt
2. [distance: 0.145] Гарантийный срок на оборудование составляет 12 месяцев с даты поставки.
```

Источник: test.txt

Видно, что поиск нашёл релевантные чанки о гарантиях, а не о реквизитах или пене. Расстояние (distance) для косинусной метрики — чем меньше, тем ближе по смыслу.

Что дальше

У нас есть все кирпичики для RAG: embedding-модель и векторная база данных. В следующей главе мы напишем функцию, которая обходит папку с документами, читает файлы PDF, DOCX, TXT, разбивает их на чанки и индексирует в ChromaDB. Это будет глава **4.4 «Практикум: индексация папки с документами»**.

Глава 4.4. Практикум: индексация папки с документами.

В предыдущих главах мы собрали все детали конструктора: embedding-модель превращает текст в векторы, ChromaDB хранит их и ищет. Теперь нам нужен «загрузчик» — функция, которая пройдётся по папке с вашими документами, прочитает файлы, разобьёт их на осмысленные куски, векторизует и сохранит в базу данных. Это и будет **индексация** — процесс, который выполняется один раз (или периодически) и делает вашу личную базу знаний доступной для вопросов.

Инструменты для чтения разных форматов

Мы хотим поддерживать три самых распространённых типа файлов:

· **PDF** — контракты, отчёты, сканы (если распознаны). Будем использовать библиотеку PyPDF2.

· **DOCX** — документы Microsoft Word. Используем python-docx.

· **TXT** — обычные текстовые файлы (читаются встроенными средствами Python).

Установим недостающие пакеты:

bash

pip install PyPDF2 python-docx langchain-text-splitters

langchain-text-splitters — это лёгкий пакет, который содержит умные разбиватели текста. Он не зависит от всего LangChain, только от langchain-core. Мы будем использовать RecursiveCharacterTextSplitter — он пытается разбить текст по абзацам, потом по предложениям, потом по словам, чтобы сохранить смысловую целостность.

Разбивка на чанки: почему это важно

Если мы скормим в ChromaDB целый 100-страничный PDF как один документ, поиск по нему будет бесполезен — вектор одного огромного текста плохо отражает детали. Нужно разбить текст на фрагменты, каждый из которых содержит законченную мысль.

Оптимальный размер чанка — примерно 500–1000 символов. Слишком маленькие чанки теряют контекст, слишком большие — размывают релевантность и не влезают в контекстное окно LLM. Мы также добавим **перекрытие** (overlap) — 100–200 символов, которые дублируются с предыдущим чанком. Это помогает сохранить связность: если мысль «переползла» через границу чанка, она будет видна в обоих.

Функция чтения файлов

Создадим модуль document_reader.py:

python

```
# document_reader.py
import os
from typing import List, Dict
import PyPDF2
from docx import Document
def read_pdf(file_path: str) -> str:
"""Извлекает текст из PDF-файла, страница за страницей."""
text_parts = []
with open(file_path, 'rb') as f:
reader = PyPDF2.PdfReader(f)
for page_num, page in enumerate(reader.pages, start=1):
page_text = page.extract_text()
if page_text:
```

```
text_parts.append(f"[Страница {page_num}] {page_text}")
return "\n".join(text_parts)
def read_docx(file_path: str) -> str:
"""Извлекает текст из DOCX-файла."""
doc = Document(file_path)
text_parts = [para.text for para in doc.paragraphs if para.text.strip()]
return "\n".join(text_parts)
def read_txt(file_path: str) -> str:
"""Читает простой текстовый файл с автоопределением кодировки."""
# Пробуем UTF-8, если не получается — cp1251 (Windows-кириллица)
try:
with open(file_path, 'r', encoding='utf-8') as f:
return f.read()
except UnicodeDecodeError:
with open(file_path, 'r', encoding='cp1251') as f:
return f.read()
def read_file(file_path: str) -> str:
"""
Определяет тип файла по расширению и возвращает его полный текст.
"""
ext = os.path.splitext(file_path)[1].lower()
if ext == '.pdf':
return read_pdf(file_path)
elif ext == '.docx':
return read_docx(file_path)
elif ext == '.txt':
return read_txt(file_path)
else:
raise ValueError(f"Неподдерживаемый формат: {ext}")
```

Пояснения :

· read_pdf нумерует страницы — это пригодится для метаданных и цитирования.

· read_docx извлекает только непустые параграфы, чтобы не засорять базу пустыми строками.

· read_txt обрабатывает распространённую проблему: файлы в кодировке Windows-1251 (часто встречаются в русскоязычных проектах).

Функция индексации папки

Теперь — главный модуль indexer.py, который объединяет всё вместе:

```
python
# indexer.py
import os
import uuid
from typing import List, Dict
from langchain_text_splitters import RecursiveCharacterTextSplitter
from document_reader import read_file
from embedder import Embedder
from vector_store import VectorStore
def index_folder(
folder_path: str,
store: VectorStore,
```

```
embedder: Embedder,
chunk_size: int = 800,
chunk_overlap: int = 150,
file_extensions: List[str] = ['.pdf', '.docx', '.txt']
) -> int:
"""
Рекурсивно обходит папку, читает все поддерживаемые файлы,
разбивает на чанки, векторизует и сохраняет в ChromaDB.
Возвращает общее количество добавленных чанков.
"""
# Создаём разбиватель текста
splitter = RecursiveCharacterTextSplitter(
chunk_size=chunk_size,
chunk_overlap=chunk_overlap,
separators=["\n\n", "\n", ". ", " ", ""] # порядок: от крупных разделителей к мелким
)
total_chunks = 0
# Обходим все файлы в папке и подпапках
for root, dirs, files in os.walk(folder_path):
for file_name in files:
ext = os.path.splitext(file_name)[1].lower()
if ext not in file_extensions:
continue # пропускаем неподдерживаемые форматы
file_path = os.path.join(root, file_name)
print(f"Обработка: {file_path}")
try:
# Читаем полный текст документа
full_text = read_file(file_path)
if not full_text.strip():
print(f" -> Пустой файл, пропущен.")
continue
# Разбиваем на чанки
chunks = splitter.split_text(full_text)
print(f" -> Получено чанков: {len(chunks)}")
# Готовим данные для пакетной вставки
ids = []
embeddings_list = []
documents = []
metadatas = []
for i, chunk in enumerate(chunks):
chunk_id = f"{file_path}_{i}_{uuid.uuid4().hex[:8]}"
ids.append(chunk_id)
documents.append(chunk)
metadatas.append({
"source": file_path,
"file_name": file_name,
"chunk_index": i
})
# Векторизуем сразу все чанки (batch-обработка)
```

```
embeddings_list = embedder.embed_documents(chunks)
# Сохраняем в ChromaDB
store.add_documents(
ids=ids,
embeddings=embeddings_list,
documents=documents,
metadatas=metadatas
)
total_chunks += len(chunks)
except Exception as e:
print(f" -> Ошибка: {e}")
continue
return total_chunks
```

Ключевые моменты :

· Мы используем RecursiveCharacterTextSplitter, который пробует разбивать текст по двойным переводам строки (абзацы), затем по одному, затем по точкам и пробелам. Так чанки получаются более «умными».

· Метаданные (source, file_name, chunk_index) позволят нам при ответе указать, из какого файла и какого места взят фрагмент.

· Векторизуем чанки **пакетом**: embedder.embed_documents(chunks). Это намного быстрее, чем векторизовать по одному, потому что модель обрабатывает список за один проход.

· Идентификатор чанка содержит путь к файлу и случайную часть, чтобы избежать коллизий.

Тестирование индексации

Создадим в проекте папку sample_docs/ с тестовыми файлами:

· test.txt — пара абзацев.

· test.pdf (можно создать из Word и сохранить как PDF).

· test.docx — любой документ.

Напишем тестовый скрипт:

```
python
# test_indexer.py
from embedder import Embedder
from vector_store import VectorStore
from indexer import index_folder
# 1. Инициализируем компоненты
print("Загрузка embedding-модели...")
emb = Embedder()
store = VectorStore(collection_name="test_docs")
# 2. Индексируем папку
print("Начинаем индексацию...")
count = index_folder(
folder_path="./sample_docs",
store=store,
embedder=emb,
chunk_size=500,
chunk_overlap=50
)
print(f"\nИндексация завершена. Всего чанков: {count}")
# 3. Поищем что-нибудь
```

query = "ваш тестовый запрос"
query_vec = emb.embed_query(query)
results = store.search(query_vec, n_results=3)
print("\nРезультаты поиска:")
for doc, meta in zip(results["documents"][0], results["metadatas"][0]):
print(f"- [{meta['file_name']}] {doc[:100]}...")
Запустим:
bash
python test_indexer.py
Вывод:
text
Загрузка embedding-модели intfloat/multilingual-e5-base...
Модель загружена, размерность векторов: 768
Начинаем индексацию...
Обработка: ./sample_docs/test.txt
-> Получено чанков: 2
Обработка: ./sample_docs/test.docx
-> Получено чанков: 3
Обработка: ./sample_docs/test.pdf
-> Получено чанков: 5
Индексация завершена. Всего чанков: 10
Результаты поиска:
- [test.pdf] ... текст, релевантный запросу...

Переиндексация и обновление данных

Что делать, когда документы меняются? Пока что мы просто добавляем новые чанки, но старые версии тех же файлов остаются в базе. Для начала этого достаточно: при повторном запуске index_folder мы создадим дубликаты. Чтобы решить проблему, перед индексацией можно полностью очищать коллекцию:

python
store.clear()

Или, как вариант, хранить время последней индексации и добавлять только новые/изменённые файлы. Для персонального использования очистка и переиндексация с нуля — самый надёжный вариант, ведь это занимает минуты на обычном ноутбуке.

Возможные проблемы при индексации документов

PDF содержит только изображения. Если документ представляет собой скан, а не текст, библиотека PyPDF2 не сможет извлечь из него слова — она работает только с текстовым слоем. Для таких файлов нужно предварительное распознавание через OCR, например, с помощью Tesseract. Это выходит за рамки нашей книги, но вы можете добавить поддержку сканов позже, установив pytesseract и модифицировав функцию read_pdf.

Очень большой PDF на сотни страниц. Загрузка всего документа в память целиком может привести к её исчерпанию. Решение — читать PDF постранично с помощью PyPDF2.PdfReader и обрабатывать страницы порциями, а не все сразу. Наш код уже делает это: мы проходим по страницам в цикле, извлекая текст по одной, что экономит память.

Ошибка кодировки в текстовом файле. Не все TXT-файлы сохранены в UTF-8. В России до сих пор распространена кодировка Windows-1251, особенно в старых документах. В функции read_txt мы предусмотрели fallback: если чтение в UTF-8 не удалось, пробуем cp1251. Если вы работаете с файлами в других кодировках, можно добавить библиотеку chardet для автоматического определения кодировки.

Разрыв слов при разбивке на чанки. RecursiveCharacterTextSplitter старается разрезать текст по естественным границам — абзацам, предложениям, пробелам. Но с кириллицей он иногда ошибается, и слово может быть разорвано на границе чанка. Чтобы уменьшить вероятность разрывов, увеличьте параметр chunk_overlap — например, со 150 до 250 символов. Это создаст более широкую зону перекрытия между соседними чанками, и мысль не потеряется на стыке.

Что дальше

Мы подготовили базу знаний. Теперь осталось связать поиск по документам с нашей языковой моделью. В главе **4.5 «Реализация функции "Спросить по документам" с цитированием»** мы напишем метод ask_documents(), который примет вопрос, найдёт релевантные чанки, вставит их в промпт и вернёт ответ со ссылками на источники. Это станет «вишенкой» нашего локального RAG.

Глава 4.5. Реализация функции «Спросить по документам» с цитированием

Мы подошли к моменту, когда разрозненные детали — embedding-модель, векторная база данных, индексатор — складываются в работающий механизм. В этой главе мы напишем функцию ask_documents(), которая примет вопрос, найдёт релевантные фрагменты в вашей личной коллекции документов, передаст их языковой модели и вернёт осмысленный ответ с указанием источников.

Это не просто выдача списка найденных кусков. Модель прочитает их и **синтезирует** ответ: объяснит, сопоставит факты, сделает вывод. А цитаты подтвердят, что информация взята из ваших файлов, а не выдумана.

Архитектура: что происходит внутри ask_documents()

Функция выполняет четыре шага:

1. **Векторизация вопроса.** Мы берём вопрос пользователя и прогоняем через ту же embedding-модель, что использовали для индексации документов.

2. **Поиск в ChromaDB.** Векторная база возвращает, скажем, 4–5 самых близких по смыслу чанков.

3. **Сборка промпта.** Мы формируем системное сообщение, вставляем найденные чанки с указанием источников и задаём вопрос пользователя.

4. **Генерация ответа.** Модель получает промпт и генерирует ответ, содержащий факты из документов и ссылки на источники.

Всё это происходит локально, в одном процессе Python, без единого внешнего запроса.

Системный промпт для RAG

Правильно составленный системный промпт — ключ к качественному ответу. Он должен заставить модель:

· Опираться только на предоставленные документы.

· Честно признаваться, если ответа в них нет.

· Цитировать источники.

Вот промпт, который мы будем использовать:

text

Ты — эксперт-аналитик, который отвечает на вопросы, основываясь исключительно на предоставленных фрагментах документов.

Правила:

1. Если ответ можно найти в документах, дай его чётко и по делу.

2. После каждого факта указывай в скобках источник в формате (файл: имя_файла).

3. Если информация из разных фрагментов противоречит друг другу, укажи на противоречие.

4. Если в документах нет ответа на вопрос, скажи: "В предоставленных документах информация не найдена."

5. Не придумывай ничего, чего нет в документах.

Код : метод ask_documents в классе LocalModel

Добавим новый метод в наш класс LocalModel. Откройте local_model.py и дополните его:

python

```
# local_model.py (дополнение к существующему классу)
from vector_store import VectorStore
from embedder import Embedder
class LocalModel:
```

```
    # ... (предыдущие методы остаются без изменений) ...
    def ask_documents(
    self,
    question: str,
    store: VectorStore,
    embedder: Embedder,
    n_results: int = 4,
    temperature: float = 0.3,
    max_tokens: int = 512
    ) -> str:
    """
    Задаёт вопрос по проиндексированным документам.
    :param question: вопрос пользователя
    :param store: объект VectorStore с загруженными документами
    :param embedder: объект Embedder для векторизации вопроса
    :param n_results: сколько релевантных чанков извлечь из базы
    :param temperature: креативность ответа (низкая — строже по документам)
    :param max_tokens: максимальная длина ответа
    :return: ответ модели со ссылками на источники
    """
    # Шаг 1: векторизуем вопрос
    query_embedding = embedder.embed_query(question)
    # Шаг 2: ищем релевантные чанки
    results = store.search(query_embedding, n_results=n_results)
    if not results["documents"] or not results["documents"][0]:
    return "В предоставленных документах информация не найдена."
    # Шаг 3: собираем фрагменты в промпт
    context_parts = []
    for doc, meta in zip(results["documents"][0], results["metadatas"][0]):
    file_name = meta.get("file_name", "неизвестный файл")
    context_parts.append(f"--- Фрагмент из файла: {file_name} ---\n{doc}\n")
    context = "\n".join(context_parts)
    system_prompt = (
    "Ты — эксперт-аналитик, который отвечает на вопросы, основываясь исключительно на "
    "предоставленных фрагментах документов. Правила:\n"
    "1. Если ответ можно найти в документах, дай его чётко и по делу.\n"
    "2. После каждого факта указывай в скобках источник в формате (файл: имя_файла).\n"
    "3. Если информация из разных фрагментов противоречит друг другу, укажи на противоречие.\n"
    "4. Если в документах нет ответа на вопрос, скажи: \"В предоставленных документах информация не найдена.\"\n"
    "5. Не придумывай ничего, чего нет в документах."
    )
    user_message = (
    f"Вопрос пользователя: {question}\n\n"
    f"Фрагменты документов для анализа:\n\n{context}\n"
    )
    # Шаг 4: генерируем ответ
    response = self.chat(
```

```
messages=[{"role": "user", "content": user_message}],
system_prompt=system_prompt,
temperature=temperature,
max_tokens=max_tokens
)
return response
```

Обратите внимание:

· temperature=0.3 — низкая температура делает модель более «послушной» и менее склонной к фантазиям. Для фактологического поиска это важно.

· Мы формируем промпт так, чтобы модель не получала голый вопрос, а видела перед собой задание аналитика с конкретными данными.

Полный тестовый сценарий

Создадим скрипт test_rag.py, который демонстрирует весь пайплайн: от индексации до вопросов.

```
python
# test_rag.py
from model_loader import download_model
from local_model import LocalModel
from embedder import Embedder
from vector_store import VectorStore
from indexer import index_folder
# 1. Подготовка: загружаем модель, embedding и хранилище
print("Загрузка языковой модели...")
model_path = download_model()
llm = LocalModel(model_path=model_path, n_threads=4, n_gpu_layers=0)
print("Загрузка embedding-модели...")
emb = Embedder()
print("Подготовка векторного хранилища...")
store = VectorStore(collection_name="my_docs")
# 2. Индексация (выполняется один раз или после добавления файлов)
# Если коллекция уже существует — очистим для чистоты эксперимента
store.clear()
count = index_folder(
folder_path="./sample_docs",
store=store,
embedder=emb,
chunk_size=500,
chunk_overlap=50
)
print(f"Проиндексировано чанков: {count}\n")
# 3. Задаём вопросы
questions = [
"Какие гарантийные обязательства у исполнителя?",
"В каких случаях договор может быть расторгнут досрочно?",
"Какой размер штрафа за просрочку оплаты?",
"Есть ли в документах упоминание о конфиденциальности?"
]
for q in questions:
print(f"\n{'='*60}")
```

```
print(f"Вопрос: {q}")
print(f"{'='*60}")
answer = llm.ask_documents(
question=q,
store=store,
embedder=emb,
n_results=3,
temperature=0.2
)
print(f"Ответ: {answer}")
```

Запустим:

bash

```
python test_rag.py
```

Пример вывода:

text

```
Загрузка языковой модели...
Модель загружена.
Загрузка embedding-модели...
Модель загружена, размерность векторов: 768
Подготовка векторного хранилища...
Обработка: ./sample_docs/contract.txt
-> Получено чанков: 4
Проиндексировано чанков: 4
============================================================
Вопрос: Какие гарантийные обязательства у исполнителя?
============================================================
Ответ: Согласно контракту (файл: contract.txt), исполнитель гарантирует устранение
недостатков в течение 30 календарных дней с момента получения уведомления.
Гарантийный срок на результаты работ составляет 12 месяцев (файл: contract.txt).
============================================================
Вопрос: В каких случаях договор может быть расторгнут досрочно?
============================================================
Ответ: Досрочное расторжение возможно при существенном нарушении условий одной
из
сторон (файл: contract.txt). Также договор может быть расторгнут по взаимному
соглашению сторон с письменным уведомлением за 30 дней (файл: contract.txt).
============================================================
Вопрос: Какой размер штрафа за просрочку оплаты?
============================================================
Ответ: За каждый день просрочки оплаты начисляется пеня в размере 0.1% от
неоплаченной суммы (файл: contract.txt).
============================================================
Вопрос: Есть ли в документах упоминание о конфиденциальности?
============================================================
Ответ: В предоставленных документах информация не найдена.
```

Модель корректно извлекла факты и честно призналась, когда информации нет. Это именно то поведение, которое нам нужно.

Потоковая версия метода

Для будущего GUI нам понадобится потоковая версия ask_documents(), чтобы ответ появлялся на экране постепенно, как в чате. Добавим её:

```
python
def ask_documents_stream(
self,
question: str,
store: VectorStore,
embedder: Embedder,
n_results: int = 4,
temperature: float = 0.3,
max_tokens: int = 512
):
"""Потоковая версия ask_documents()."""
query_embedding = embedder.embed_query(question)
results = store.search(query_embedding, n_results=n_results)

if not results["documents"] or not results["documents"][0]:
yield "В предоставленных документах информация не найдена."
return
context_parts = []
for doc, meta in zip(results["documents"][0], results["metadatas"][0]):
file_name = meta.get("file_name", "неизвестный файл")
context_parts.append(f"--- Фрагмент из файла: {file_name} ---\n{doc}\n")
context = "\n".join(context_parts)
system_prompt = (
"Ты — эксперт-аналитик... Правила: ..." # тот же промпт
)
user_message = f"Вопрос пользователя: {question}\n\nФрагменты документов:\n
\n{context}"
yield from self.stream_chat(
messages=[{"role": "user", "content": user_message}],
system_prompt=system_prompt,
temperature=temperature,
max_tokens=max_tokens
)
```

Важные замечания по промпт-инжинирингу

1. **Чёткие правила в системе.** Модели Llama (и другие открытые) хорошо следуют инструкциям, если они сформулированы пронумерованным списком. Размытые просьбы («отвечай хорошо») работают хуже.

2. **Низкая температура.** Для задач, требующих фактологической точности, temperature=0.2..0.3 даёт лучшие результаты, чем 0.7. Модель меньше фантазирует.

3. **Обработка противоречий.** Иногда в разных чанках может быть противоречивая информация. Правило 3 в нашем промпте помогает модели не запутаться и явно указать на расхождение.

4. **Ограничение контекста.** Если чанков слишком много или они слишком длинные, они могут не влезть в n_ctx. Наш n_results=4 и chunk_size=500 дают около 2000 токенов контекста — достаточно для окна в 2048 токенов с запасом под ответ. При увеличении окна до 4096 можно увеличить n_results.

Возможные проблемы при работе с RAG.

Ответ слишком короткий или неполный. Модель может обрезать ответ, если параметр max_tokens установлен слишком низко. Увеличьте его с 512 до 1024 или даже 2048 для сложных вопросов. Другой способ — явно попросить развёрнутый ответ в системном промпте: «Отвечай подробно, с примерами из документов, не менее трёх предложений». Модели Llama и Qwen хорошо следуют таким инструкциям.

Модель игнорирует документы и фантазирует. Это случается при высокой температуре — модель начинает «творчески» дополнять ответ. Понизьте temperature до 0.1 или 0.2 — в таком режиме модель становится строже и меньше отклоняется от предоставленных фактов. Также ужесточите системный промпт, добавив фразу: «Категорически запрещено использовать знания, не содержащиеся в документах. Если ответа нет — так и скажи». Это работает как дополнительный барьер от галлюцинаций.

Ответ на русском, но с английскими вкраплениями. Многие модели обучались в основном на английском корпусе и могут вставлять английские слова в русский текст — особенно термины. Лучшее решение — использовать модель, изначально оптимизированную под русский язык, например Qwen 2.5 7B или Qwen 3 14B. Если вы привязаны к Llama, добавьте в системный промпт прямое указание: «Отвечай на чистом русском языке, без английских слов». Обычно этого достаточно для 8B-модели.

Поиск не находит релевантные чанки. Возможно, вы указали слишком маленькое значение n_results — увеличьте его с 4 до 8, чтобы захватить больше кандидатов. Если это не помогает, проблема может быть в embedding-модели: она плохо понимает вашу специфическую терминологию. Попробуйте заменить модель на более крупную или специализированную под вашу область. Также проверьте размер чанков — слишком маленькие теряют контекст, слишком большие размывают релевантность.

Цитаты указываются, но название файла обрезано. В метаданных, которые мы сохраняем при индексации, есть поле file_name, но оно содержит только имя файла без пути. Если вам нужно видеть полный путь к документу, добавьте в indexer сохранение relative_path — относительного пути от корневой папки. Тогда в ответе модели будет фигурировать не просто contract.pdf, а договоры/2024/contract.pdf, что гораздо информативнее.

Что дальше

Мы завершили Часть 4. Наш ИИ-помощник теперь умеет:

· Индексировать папку с документами.

· Векторизовать их и хранить в локальной ChromaDB.

· Отвечать на вопросы, опираясь на факты из этих документов.

· Честно признаваться, если ответа нет.

· Указывать источники.

В Части 5 мы добавим голосовой интерфейс: научим помощника слушать и говорить, оставаясь полностью офлайн.

Часть 5. Голосовой интерфейс

Глава 5.1. Офлайн-распознавание речи: Whisper tiny/base

Наш локальный ИИ уже умеет анализировать документы и отвечать на вопросы, но пока общение с ним — это набор текста на клавиатуре. Теперь мы сделаем следующий шаг: дадим помощнику «уши». Он будет слушать нас через микрофон, преобразовывать речь в текст и передавать его в языковую модель. И всё это — без интернета, с помощью моделей Whisper, которые работают прямо на вашем процессоре.

Что такое Whisper и почему он работает офлайн

Whisper — это система распознавания речи, разработанная OpenAI и опубликованная с открытыми весами. Она обучена на 680 000 часах аудио на разных языках и способна распознавать русскую речь на достойном уровне. Благодаря открытой лицензии модель можно скачать и запускать локально.

Whisper существует в нескольких размерах: от крошечной tiny (39 миллионов параметров, около 75 МБ на диске) до гигантской large-v3 (1.5 миллиарда параметров, около 3 ГБ). Для нашего персонального помощника идеально подойдут tiny или base. Почему:

· **tiny** — самая быстрая, способна работать в реальном времени даже на слабом процессоре. Качество распознавания русского языка приемлемое для команд и простых вопросов.

· **base** — чуть больше и точнее, особенно на русском. Требует немного больше ресурсов, но всё ещё легко запускается на любом современном ноутбуке.

Обе модели работают полностью локально, не отправляя аудио во внешние серверы.

Выбор между faster-whisper и whisper.cpp

Оригинальный Whisper от OpenAI написан на Python, но довольно медленный и прожорливый. К счастью, сообщество создало оптимизированные реализации:

· **faster-whisper** — переписанный на C++ движок (CTranslate2), который значительно быстрее оригинала. Отлично работает через Python, прост в установке и поддерживает все размеры моделей.

· **whisper.cpp** — порт на C++ без зависимостей Python. Ещё быстрее, но требует либо компиляции, либо использования отдельных биндингов. Для интеграции в наше Python-приложение менее удобен.

Мы выберем **faster-whisper**. Он даёт отличный баланс скорости и простоты: устанавливается через pip, работает как обычная Python-библиотека и поддерживает потоковую обработку аудио.

Установка faster-whisper и запись с микрофона

Установим две библиотеки:

bash

pip install faster-whisper sounddevice numpy

· faster-whisper — сам движок распознавания.

· sounddevice — кроссплатформенная библиотека для захвата аудио с микрофона (работает на Windows, macOS, Linux без дополнительных драйверов).

· numpy — потребуется для преобразования аудиоданных.

На Linux может понадобиться установить системную библиотеку portaudio:

bash

sudo apt install portaudio19-dev *# Debian/Ubuntu*

Пишем функцию распознавания речи

Создадим модуль speech_recognition.py. Функция recognize_speech() будет слушать микрофон до наступления тишины и возвращать распознанный текст.

python
# speech_recognition.py
import queue
import numpy as np
import sounddevice as sd
from faster_whisper import WhisperModel
# Глобальные константы
SAMPLE_RATE = 16000 # Частота дискретизации (требуется для Whisper)
SILENCE_THRESHOLD = 0.01 # Уровень громкости, ниже которого считаем тишиной
SILENCE_DURATION = 1.5 # Сколько секунд тишины нужно, чтобы закончить запись
MAX_RECORD_SECONDS = 30 # Максимальная длительность записи
class SpeechRecognizer:
"""
Локальное распознавание речи с помощью faster-whisper.
"""
def __init__(self, model_size: str = "tiny", device: str = "cpu"):
"""
:param model_size: размер модели: "tiny", "base", "small", "medium", "large-v3"
:param device: "cpu" или "cuda" (если есть GPU NVIDIA)
"""
print(f"Загрузка модели Whisper ({model_size})...")
self.model = WhisperModel(model_size, device=device, compute_type="int8")
print("Модель готова.")
def record_until_silence(self) -> np.ndarray:
"""
Записывает аудио с микрофона до наступления SILENCE_DURATION секунд тишины
или до MAX_RECORD_SECONDS. Возвращает массив float32 с частотой 16 кГц.
"""
q = queue.Queue()
recorded_chunks = []
def callback(indata, frames, time, status):
if status:
print(f"Предупреждение: {status}")
q.put(indata.copy())
print("Слушаю... (говорите)")
with sd.InputStream(
samplerate=SAMPLE_RATE,
channels=1,
callback=callback,
dtype=np.float32
):
silence_start = None
while True:
chunk = q.get()
recorded_chunks.append(chunk)
# Определяем громкость текущего чанка
volume = np.abs(chunk).mean()

```
# Проверяем на тишину
if volume < SILENCE_THRESHOLD:
if silence_start is None:
silence_start = len(recorded_chunks) * chunk.shape[0] / SAMPLE_RATE
else:
silence_dur = (len(recorded_chunks) * chunk.shape[0] / SAMPLE_RATE) - silence_start
if silence_dur >= SILENCE_DURATION:
print("Тишина, завершаю запись.")
break
else:
silence_start = None # сброс, если снова звук
# Проверяем максимальную длительность
total_sec = len(recorded_chunks) * chunk.shape[0] / SAMPLE_RATE
if total_sec >= MAX_RECORD_SECONDS:
print("Достигнут лимит записи.")
break
# Объединяем чанки в один массив
audio = np.concatenate(recorded_chunks, axis=0).flatten()
return audio
def transcribe(self, audio: np.ndarray) -> str:
"""
Преобразует аудиомассив в текст с помощью Whisper.
"""
segments, info = self.model.transcribe(audio, language="ru", beam_size=5)
text = " ".join([seg.text for seg in segments])
return text.strip()
def recognize_speech(self) -> str:
"""
Основная функция: слушает микрофон и возвращает распознанный текст.
Если ничего не распознано — пустая строка.
"""
audio = self.record_until_silence()
if audio is None or len(audio) == 0:
return ""
text = self.transcribe(audio)
return text
```

Пояснения к коду:

· **Параметры микрофона:** Whisper ожидает аудио с частотой 16 кГц в моно. Мы настраиваем sd.InputStream соответственно.

· **Обнаружение тишины:** Мы вычисляем среднюю громкость каждого аудиочанка. Если громкость падает ниже SILENCE_THRESHOLD, запускаем счётчик. Когда тишина длится SILENCE_DURATION секунд подряд, запись останавливается. Если пользователь снова начинает говорить, счётчик сбрасывается.

· **Ограничение записи:** MAX_RECORD_SECONDS предотвращает бесконечную запись, если алгоритм ошибётся с определением тишины.

· **Параметр** language="ru" ускоряет распознавание для русского языка. Если вы говорите на смеси языков, можно указать language=None — модель определит язык сама, но это чуть медленнее.

Тестирование распознавания

Напишем простой тест:

```
python
# test_speech.py
from speech_recognition import SpeechRecognizer
# Создаём распознаватель с моделью tiny
rec = SpeechRecognizer(model_size="tiny")
print("Нажмите Enter, чтобы начать запись...")
input()
text = rec.recognize_speech()
if text:
print(f"Распознано: {text}")
else:
print("Не удалось распознать речь.")
print("Не удалось распознать речь.")
```

Запустите, нажмите Enter и скажите несколько фраз. После полутора секунд молчания запись завершится, и вы увидите текст.

Выбор размера модели Whisper

Качество распознавания речи напрямую зависит от размера модели. Давайте сравним три основные версии на одной и той же русской фразе.

Tiny. Самая маленькая и быстрая модель — загружается около 2 секунд, распознаёт 3 секунды аудио всего за полсекунды. Занимает примерно 75 мегабайт на диске. Качество распознавания русского языка удовлетворительное: модель может путать окончания слов и ошибаться в шумной обстановке. Идеальный выбор для простых голосовых команд, когда важна скорость, а не дословная точность.

Base. Умеренный размер — загрузка около 3 секунд, распознавание того же аудио за 0.8 секунды. Занимает около 150 мегабайт. Качество русского языка хорошее, подходит для повседневной речи: модель правильно распознаёт большинство фраз в тихой комнате. Рекомендуемый выбор для домашнего голосового ассистента.

Small. Самая точная из компактных моделей — загружается около 5 секунд, распознавание за 2 секунды. Занимает около 500 мегабайт. Качество русского языка отличное: модель хорошо справляется даже с быстрой речью и лёгким фоновым шумом. Для большинства домашних сценариев может быть избыточной — разница с base заметна только в сложных условиях.

Практический совет: Начните с tiny — её возможностей хватит для голосовых команд и заметок. Если точность не устраивает, переключитесь на base. Модель small имеет смысл использовать только если вы планируете много работать с голосом в шумной обстановке или вам важна максимальная точность распознавания. Замена модели не требует изменений в коде — достаточно указать другой размер в конструкторе SpeechRecognizer.

Для нашего проекта я рекомендую tiny как основную, а base — если ваш процессор позволяет и вы хотите более точного распознавания. В коде достаточно поменять одну строку: SpeechRecognizer(model_size="base").

Интеграция в основной проект

Функция recognize_speech() будет использоваться в будущем GUI следующим образом:

```
python
recognizer = SpeechRecognizer(model_size="tiny")
text = recognizer.recognize_speech()
if text:
# Отправляем text в LocalModel.chat() или ask_documents()
```

Мы не стали добавлять recognize_speech внутрь класса LocalModel, чтобы сохранить разделение обязанностей. LocalModel управляет языковой моделью, SpeechRecognizer — аудио. В главе 5.3 мы объединим их в едином голосовом ассистенте.

Возможные проблемы с распознаванием речи

Ошибка «PortAudioError: Error opening InputStream». Микрофон не подключён физически или занят другим приложением — например, мессенджером в фоновом режиме. Проверьте настройки звука в системе: зайдите в параметры записи и убедитесь, что микрофон отображается и не отключён. На Linux может потребоваться установка или запуск звукового сервера — pulseaudio --start или активация pipewire. После этого перезапустите приложение.

Модель не распознаёт русскую речь. Убедитесь, что в методе transcribe указан параметр language="ru" — без него Whisper пытается определить язык автоматически и может ошибиться, особенно на коротких фразах. Модели tiny и base понимают русский, но могут путать слова в шумной обстановке или при быстрой речи. Если качество не устраивает, попробуйте модель base вместо tiny — разница заметна.

Запись обрывается слишком рано. Вы не успеваете договорить фразу, а запись уже завершилась. Это происходит, когда алгоритм обнаружения тишины слишком чувствительный. Увеличьте SILENCE_DURATION с 1.5 до 2.0 секунд — это даст больше времени на паузы между словами. Альтернативно, уменьшите SILENCE_THRESHOLD до 0.005 — это сделает детектор тишины менее чувствительным к тихим звукам.

Распознавание очень медленное. На процессоре распознавание даже короткой фразы может занимать несколько секунд. Попробуйте самую маленькую модель — tiny вместо base, разница в скорости существенна, а качество падает незначительно. Если у вас есть видеокарта NVIDIA, используйте device="cuda" и compute_type="float16" при создании SpeechRecognizer — это ускорит распознавание в разы.

Шум на записи. Встроенные микрофоны ноутбуков часто записывают шум вентиляторов и эхо комнаты. Самое простое решение — использовать гарнитуру или внешний микрофон, качество распознавания вырастет dramatically. Если хотите остаться со встроенным микрофоном, можно добавить программную фильтрацию шума через библиотеку noisereduce, но это выходит за рамки базового проекта.

Что дальше

Теперь наш помощник умеет слышать. В следующей главе мы научим его **говорить** — добавим локальный синтез речи. Глава **5.2 «Озвучивание ответа: piper-tts и silero»** покажет, как превратить текст ответа в голос, который можно слушать, не глядя на экран.

Глава 5.2. Озвучивание ответа: piper-tts и silero.

Мы научили нашего ИИ-помощника слышать. Теперь пора дать ему голос. Когда вы задаёте вопрос, ответ может появляться на экране, но иногда хочется услышать его — например, когда вы готовите кофе или смотрите в окно. Мы добавим локальный синтез речи (Text-to-Speech, TTS), который превратит текст ответа в звук, воспроизводимый через колонки или наушники. И разумеется, всё будет работать без интернета.

На начало 2026 года есть два основных открытых решения для качественного синтеза речи на русском языке, которые можно запустить локально: **piper-tts** и **silero**. Давайте сравним их и выберем то, которое лучше впишется в наш проект.

Сравнение piper-tts и silero

Piper-tts

Piper — это быстрый движок синтеза речи, написанный на C++ с Python-биндингами. Он создавался для домашней автоматизации и голосовых ассистентов, работающих на слабом железе вроде Raspberry Pi. Piper использует модели, обученные на различных языках, включая русский.

Плюсы:

· Очень быстрый, работает в реальном времени даже на CPU.

· Низкое потребление памяти.

· Есть готовые русские голоса (например, ru_RU-ruslan-medium).

Минусы:

· Требует установки системных библиотек (libportaudio, espeak-ng) и отдельно скачиваемых моделей.

· Python-интерфейс (piper-tts) не всегда стабилен на Windows.

· Качество русского голоса среднее: иногда слышны «роботизированные» нотки, хотя для утилитарного помощника терпимо.

Silero

Silero — это набор моделей для синтеза и распознавания речи, разработанный российскими исследователями. Модели TTS от Silero стали стандартом де-факто для локального синтеза на русском языке благодаря высокому качеству и простоте использования.

Плюсы:

· **Отличное качество русского голоса:** интонации, паузы, естественное звучание. Доступны несколько голосов (мужские и женские).

· **Простота интеграции:** модель загружается напрямую через PyTorch Hub одной строкой, не нужно вручную скачивать файлы.

· **Не требует внешних зависимостей:** всё работает внутри Python. Установил torch — и готово.

· **Активное сообщество:** модель обновляется, есть примеры под разные платформы.

Минусы:

· Требует PyTorch, что добавляет около 200 МБ зависимостей при первой установке (но для проекта с LLM это уже не проблема).

· Чуть медленнее piper-tts на CPU (хотя всё равно быстрее реального времени).

Что мы выбираем

Я выбираю **Silero**. Качество русского голоса здесь на голову выше piper-tts, а простота использования (одна строка для загрузки модели) идеально вписывается в философию «одной кнопки». PyTorch у нас всё равно может пригодиться в будущем для экспериментов с моделями, а его размер — небольшая плата за естественно звучащую речь.

Если для вас критична минимальная производительность (например, вы запускаетесь на очень слабом одноплатнике), вы всегда сможете заменить Silero на Piper, следуя тому же интерфейсу. Мы напишем обёртку, которую будет легко адаптировать.

Установка

Нам понадобятся PyTorch и библиотека для воспроизведения звука. Установим их:

bash

pip install torch sounddevice

Модели Silero загружаются автоматически при первом обращении, никакие файлы вручную скачивать не нужно.

Пишем функцию синтеза речи

Создадим модуль speech_synthesis.py. В нём будет класс TextToSpeech, который загружает модель Silero и воспроизводит переданный текст.

python

```
# speech_synthesis.py
import os
import torch
import sounddevice as sd
class TextToSpeech:
"""
Локальный синтез речи с помощью модели Silero.
"""
def __init__(self, speaker: str = "xenia", device: str = "cpu"):
"""
:param speaker: голос (список доступных голосов появится при первом запуске)
:param device: "cpu" или "cuda"
"""
self.device = torch.device(device)
print("Загрузка модели Silero TTS...")
# Загружаем модель и список доступных голосов из PyTorch Hub
self.model, self.symbols, self.sample_rate, _, self.apply_tts = torch.hub.load(
repo_or_dir='snakers4/silero-models',
model='silero_tts',
language='ru',
speaker='ru_v3'
)
self.model = self.model.to(self.device)
self.speaker = speaker
print(f"Модель готова. Выбран голос: {self.speaker}")
def speak(self, text: str):
"""
Синтезирует текст и воспроизводит его через звуковое устройство.
"""
if not text.strip():
return
# Синтез: получаем тензор с аудио
audio = self.apply_tts(
texts=[text],
model=self.model,
symbol=self.symbols,
```

```
device=self.device,
speaker=self.speaker
)[0] # Берем первый (и единственный) элемент батча
# Переводим тензор в numpy-массив float32
audio_np = audio.cpu().numpy()
# Воспроизводим
sd.play(audio_np, samplerate=self.sample_rate)
sd.wait() # Ждём окончания воспроизведения
```

Пояснения:

· **Загрузка модели:** torch.hub.load скачивает модель Silero (если её нет локально) и возвращает все необходимые компоненты: саму модель, словарь символов, частоту дискретизации (обычно 22050 Гц для русских голосов), пути к примерам и функцию apply_tts. Эта функция и делает синтез — принимает список строк и возвращает список аудиотензоров.

· **Выбор голоса:** Параметр speaker может принимать значения: 'xenia' (женский, мягкий), 'aidar' (мужской), 'baya' (женский, более высокий), 'kseniya' (женский), 'eugene' (мужской). В коде выше мы используем 'xenia'. Вы можете посмотреть полный список, заглянув в документацию Silero или просто попробовав разные варианты.

· **Воспроизведение:** sounddevice берёт массив float32 и проигрывает его с указанной частотой. sd.wait() блокирует выполнение до окончания звука, чтобы наша программа не завершилась раньше времени.

Тестирование

python

```
# test_tts.py
from speech_synthesis import TextToSpeech
tts = TextToSpeech(speaker="xenia")
tts.speak("Привет! Я локальный голосовой помощник. Чем могу помочь?")
```

Запустите — вы должны услышать приятный женский голос.

Интеграция в проект

В будущем GUI мы будем вызывать tts.speak(response) сразу после получения ответа от модели. Асинхронное озвучивание (не ждать окончания, чтобы можно было задать следующий вопрос) можно реализовать с помощью потоков, но пока для простоты оставим синхронный вызов.

Выбор голоса под настроение

Функция set_speaker позволит переключать голос на лету:

python

```
def set_speaker(self, speaker: str):
"""Сменить голос."""
if speaker in self.model.speakers: # некоторые версии silero хранят список спикеров
self.speaker = speaker
else:
print(f"Голос {speaker} не найден, используется {self.speaker}")
```

Но в рамках нашей книги мы оставим один голос по умолчанию.

Возможные проблемы с синтезом речи

Ошибка «ModuleNotFoundError: No module named 'torch'». Библиотека Silero работает поверх PyTorch, который не был установлен. Выполните pip install torch — для синтеза речи достаточно CPU-версии, она легче и устанавливается быстрее. Если позже понадобится GPU-ускорение, можно будет доустановить CUDA-версию PyTorch отдельно.

Ошибка загрузки модели: «HTTPError: Not Found». При первом создании объекта TextToSpeech Silero пытается скачать модель из интернета. Проверьте соединение — модель скачивается один раз и кэшируется. Если интернет недоступен, скачайте файл ru_v3.pt вручную с сайта models.silero.ai и поместите его в папку кэша PyTorch Hub. После этого модель будет загружаться локально.

Звук воспроизводится с искажениями или задержкой. Скорее всего, не совпадает частота дискретизации. Silero по умолчанию использует 22050 Гц — проверьте, что sounddevice.play() вызывается именно с этим параметром. Если звук всё равно искажён, попробуйте явно указать samplerate=22050 в вызове sd.play(). Иногда проблема в драйверах звуковой карты — обновите их.

Русский текст не озвучивается — модель молчит. Silero обучена на кириллическом русском тексте и не понимает транслитерацию. Убедитесь, что текст содержит настоящие русские буквы, а не латинские замены вроде «privet». Также проверьте, что выбран русский голос — например, xenia или aidar. Голоса для других языков не смогут произнести кириллицу.

Ошибка «PortAudioError» при воспроизведении. Звуковое устройство занято другим приложением или отсутствует. Проверьте, что колонки или наушники подключены и не используются эксклюзивно другой программой. На Linux может помочь запуск звукового сервера командой pulseaudio --start. Если ошибка повторяется, попробуйте перезагрузить компьютер — иногда звуковой драйвер «зависает» и его нужно переинициализировать.

Альтернативный путь: piper-tts (если нужно максимально легковесно)

Если вы по каким-то причинам не хотите тянуть PyTorch, вот минимальный пример с piper-tts:

```
bash
# Установка piper-tts
pip install piper-tts
# Скачать модель ru_RU-ruslan-medium с GitHub в папку models/
python
from piper import PiperVoice
import sounddevice as sd
import wave
voice = PiperVoice.load("models/ru_RU-ruslan-medium.onnx")
audio = voice.synthesize("Привет, мир!")
# audio — это bytes с WAV, нужно декодировать и воспроизвести
```

Но мы в книге придерживаемся Silero: для нашего проекта качество речи важнее лишних 200 МБ зависимостей, которые всё равно перекрываются объёмом языковой модели.

Что дальше

Теперь у нас есть и «уши» (распознавание речи), и «голос» (синтез). В следующей главе **5.3 «Интеграция в голосового ассистента»** мы объединим эти два компонента с нашим LocalModel и создадим полноценный диалоговый интерфейс, где можно нажать кнопку, задать вопрос голосом и услышать ответ — и всё это офлайн.

Глава 5.3. Интеграция в голосового ассистента.

Мы добрались до момента, когда наш ИИ-помощник обретает способность к полноценному разговору. В главе 5.1 мы дали ему «уши» — локальное распознавание речи на основе Whisper. В главе 5.2 — «голос» с помощью Silero TTS. Теперь мы свяжем эти два компонента с нашим классом LocalModel и создадим **голосового ассистента**, который слушает вопрос, думает и отвечает вслух. И всё это — без интернета, по нажатию одной кнопки (или по ключевому слову).

Архитектура голосового ассистента

Наш голосовой ассистент — это Python-класс VoiceAssistant, который объединяет три компонента:

1. SpeechRecognizer — распознаёт речь с микрофона и возвращает текст.
2. LocalModel — получает текст, генерирует ответ (может использовать режим RAG, если передан VectorStore).
3. TextToSpeech — принимает текст ответа и озвучивает его.

Все три компонента уже написаны и протестированы по отдельности. Нам осталось лишь написать «дирижёра», который будет координировать их работу.

Код класса VoiceAssistant

Создайте файл voice_assistant.py:

python

```
# voice_assistant.py
import sys
from speech_recognition import SpeechRecognizer
from speech_synthesis import TextToSpeech
from local_model import LocalModel
from vector_store import VectorStore
from embedder import Embedder
class VoiceAssistant:
"""
Голосовой ассистент, объединяющий распознавание речи,
языковую модель и синтез речи.
"""
def __init__(
self,
model: LocalModel,
whisper_size: str = "tiny",
tts_speaker: str = "xenia",
store: VectorStore = None,
embedder: Embedder = None,
use_documents: bool = False
):
"""
:param model: экземпляр LocalModel
:param whisper_size: размер модели Whisper ("tiny" / "base")
:param tts_speaker: голос Silero TTS
:param store: VectorStore (если используется RAG)
:param embedder: Embedder (если используется RAG)
:param use_documents: режим "по документам" по умолчанию
```

```
"""
self.model = model
self.store = store
self.embedder = embedder
self.use_documents = use_documents
print("Инициализация распознавания речи...")
self.recognizer = SpeechRecognizer(model_size=whisper_size)
print("Инициализация синтеза речи...")
self.tts = TextToSpeech(speaker=tts_speaker)
print("Голосовой ассистент готов.\n")
def listen(self) -> str:
"""Слушает микрофон и возвращает распознанный текст."""
text = self.recognizer.recognize_speech()
if text:
print(f"Вы: {text}")
return text
def respond(self, text: str):
"""Генерирует ответ и озвучивает его."""
print("Думаю...", end="", flush=True)
# Определяем, какой метод модели использовать
if self.use_documents and self.store and self.embedder:
response = self.model.ask_documents(
question=text,
store=self.store,
embedder=self.embedder
)
else:
response = self.model.chat(
messages=[{"role": "user", "content": text}],
temperature=0.7
)
print(f"\rАссистент: {response}")
# Озвучиваем ответ
self.tts.speak(response)
def run(self, wake_word: str = None):
"""
Основной цикл. Если wake_word задан, ассистент ждёт ключевое слово.
Иначе — слушает сразу после нажатия Enter.
"""
print("=" * 50)
print("Голосовой ассистент запущен.")
print("Нажмите Ctrl+C для выхода.\n")
if wake_word:
print(f"Ассистент ждёт ключевое слово: \"{wake_word}\"...")
else:
print("Нажмите Enter, чтобы начать говорить...")
try:
while True:
if wake_word:
```

```
# Ждём ключевое слово в цикле
text = self.listen()
if text and wake_word.lower() in text.lower():
print(f"Ключевое слово обнаружено!")
# Убираем ключевое слово из запроса
query = text.lower().replace(wake_word.lower(), "").strip()
if query:
self.respond(query)
else:
# Если только ключевое слово — спросить, что нужно
self.tts.speak("Я слушаю.")
else:
# Ручной режим: ждём Enter
input()
text = self.listen()
if text:
# Простые голосовые команды управления
if text.lower() in ["выход", "пока", "завершить"]:
self.tts.speak("До свидания!")
break
elif text.lower() in ["режим документы"]:
self.use_documents = not self.use_documents
status = "включён" if self.use_documents else "выключен"
self.tts.speak(f"Режим документы {status}")
continue
else:
self.respond(text)
except KeyboardInterrupt:
print("\nЗавершение работы...")
self.tts.speak("Работа завершена.")
```

Пояснения к коду:

· **Конструктор** принимает все необходимые компоненты. Параметры store и embedder необязательны: если вы не планируете использовать RAG, просто не передавайте их. Флаг use_documents позволяет переключать режим на лету.

· **Метод** listen() — обёртка над SpeechRecognizer.recognize_speech(), добавляющая вывод распознанного текста.

· **Метод** respond() — в зависимости от флага выбирает либо обычный чат, либо ask_documents(). Затем выводит ответ на экран и озвучивает его. Обратите внимание на трюк с \r и end="": сначала печатается «Думаю...», а когда ответ готов, курсор возвращается в начало строки и заменяет это слово на «Ассистент:».

· **Метод** run() — основной цикл. Поддерживает два режима:

о **Ручной (по Enter):** для отладки и использования за компьютером, когда вы готовы нажать клавишу перед тем, как говорить.

о **С ключевым словом:** ассистент постоянно слушает микрофон и реагирует только когда услышит заданное слово (например, «ассистент»). Это приближает нас к поведению «умной колонки». Однако постоянное распознавание речи требует больше ресурсов, поэтому по умолчанию режим ручной.

Внутри цикла реализованы простые голосовые команды:

· *«выход»*, *«пока»*, *«завершить»* — завершают работу.

· *«режим документы»* — переключает между обычным чатом и режимом RAG.

Тестирование голосового ассистента

Создайте скрипт test_voice_assistant.py:

python

```
# test_voice_assistant.py
from model_loader import download_model
from local_model import LocalModel
from voice_assistant import VoiceAssistant
from vector_store import VectorStore
from embedder import Embedder
# Загружаем языковую модель
print("Загрузка языковой модели...")
model_path = download_model()
llm = LocalModel(model_path=model_path, n_threads=4)
# Если у вас проиндексированы документы, подключим их
store = VectorStore(collection_name="my_docs")
emb = Embedder()
use_docs = True # поставьте False, если документов нет
# Создаём ассистента
assistant = VoiceAssistant(
model=llm,
whisper_size="tiny",
tts_speaker="xenia",
store=store,
embedder=emb,
use_documents=use_docs
)
# Запускаем
assistant.run() # ручной режим (по Enter)
```

Запустите:

bash

```
python test_voice_assistant.py
```

Пример сеанса (ручной режим):

text

```
Инициализация распознавания речи...
Модель готова.
Инициализация синтеза речи...
Модель готова.
Голосовой ассистент готов.
==================================================
Голосовой ассистент запущен.
Нажмите Ctrl+C для выхода.
Нажмите Enter, чтобы начать говорить...
[Enter]
Слушаю... (говорите)
Тишина, завершаю запись.
Вы: какой сегодня день недели
Думаю...
Ассистент: К сожалению, я не имею доступа к текущей дате.
```

(озвучивается голосом)

[Enter]

Слушаю... (говорите)

Вы: расскажи анекдот про программистов

Ассистент: Почему программисты путают Хэллоуин и Рождество? Потому что 31 OCT = 25 DEC!

(озвучивается)

Режим с ключевым словом

Если вы хотите, чтобы ассистент слушал постоянно и откликался на ключевое слово, замените вызов:

python

assistant.run(wake_word="ассистент")

Теперь ассистент будет непрерывно слушать микрофон и реагировать, когда вы скажете «ассистент, какой сегодня день?». Учтите, что постоянный мониторинг увеличивает нагрузку на процессор. Для реального использования можно добавить «засыпание» после периода бездействия, но для нашего проекта ручной режим более практичен.

Потоковое озвучивание (опционально)

Сейчас ассистент ждёт полного ответа модели, прежде чем начать озвучивание. Это может создавать заметную паузу, особенно если модель генерирует длинный ответ. Чтобы ответ начал озвучиваться по мере генерации, можно объединить потоковый вывод модели с озвучиванием. Для этого потребуется:

· Разбивать ответ на предложения (по точке, вопросительному или восклицательному знаку).

· Озвучивать каждое предложение, как только оно сформировано.

Это более сложная техника, и мы оставим её для самостоятельного изучения. В рамках книги синхронный respond() полностью выполняет задачу.

Возможные проблемы с голосовым ассистентом

Микрофон не реагирует или слышен постоянный шум. Порог чувствительности, заданный в SILENCE_THRESHOLD, не подходит для вашего помещения. Если микрофон вообще не реагирует на голос — уменьшите порог до 0.005 или даже 0.003. Если, наоборот, запись никогда не останавливается из-за фонового шума — увеличьте порог до 0.02. Универсальное решение — использовать гарнитуру: она даёт чистый сигнал, и настройки по умолчанию работают хорошо.

Whisper не распознаёт тихую речь. Проверьте уровень громкости микрофона в системных настройках — возможно, он убавлен до минимума. Поднимите усиление микрофона (Mic Boost), если такая опция есть. Если речь всё равно распознаётся плохо, замените модель tiny на base — она чувствительнее к тихим звукам и лучше понимает нечёткую дикцию, хотя работает чуть медленнее.

Конец ознакомительного фрагмента.