Мира С. Девлин

Построение LLM-агентов



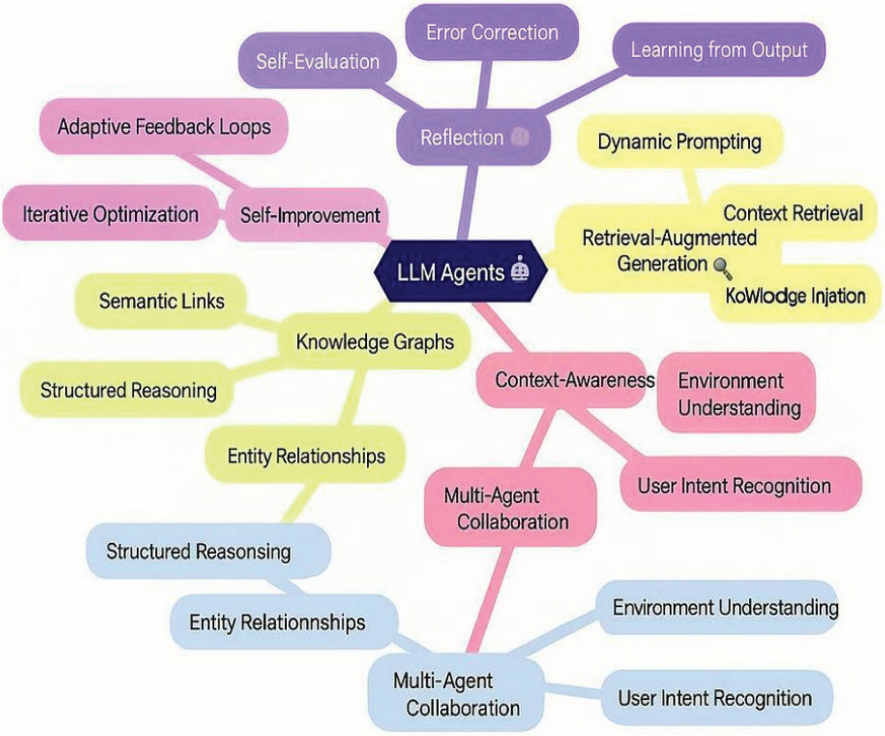
с помощью RAG, графов знаний и осмысления

ДМК ПРЕСС
издательство

Мира С. Девлин

Построение LLM-агентов с помощью RAG, графов знаний и осмысления

BUILDING LLM AGENTS WITH RAG, KNOWLEDGE GRAPHS & REFLECTION



A Practical Guide to Building Intelligent, Context-Aware, and Self-Improving AI Agents

Mira S. Devlin

Построение LLM-агентов с помощью RAG, графов знаний и осмысления

Мира С. Девлин

ДМК Пресс
издательство
Москва, 2026

УДК 004.8
ББК 16.6
Д25

Мира С. Девлин

Д25 Построение LLM-агентов с помощью RAG, графов знаний и осмысления / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2026. – 198 с.: ил.

ISBN 978-5-93700-481-9

Искусственный интеллект уже не сводится к генерации текстов – он начинает рассуждать, искать информацию и действовать. Эта трансформация знаменует переход от диалоговых языковых моделей к автономным системам, которые планируют, обучаются и совершенствуются. В данной книге предлагается схема разработки когнитивного поведения как метод создания ИИ-агентов, которые опираются на факты, контекст и самокорректируются. Основное внимание уделяется проектированию и построению индивидуальных агентов и небольших команд агентов.

Издание адресовано всем, кто хочет понять, как устроены и работают современные интеллектуальные системы и – что еще важнее – как строить их исходя из теоретических принципов.

УДК 004.8
ББК 16.6

First Edition, © 2025 Mira S. Devlin. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (рус.) 978-5-93700-481-9

© 2025 Mira S. Devlin. All rights reserved.
© Оформление, издание, перевод, ДМК Пресс, 2026

Говоря об интеллекте,
мы имеем в виду не только знания,
но и умение связывать, рассуждать
и отражать внутреннюю структуру.

– Мира С. Девлин

***

Написание книги об искусственном интеллекте не подвиг одиночки.

Эта работа основана на блестящих достижениях бесчисленных инженеров, исследователей и мыслителей, благодаря которым стала возможна нынешняя эра ИИ, – от разработчиков трансформера до сообществ, создающих фреймворки с открытым исходным кодом для генерации с дополненной выборкой и оркестрацией агентов.

Я искренне благодарна всем участникам проектов с открытым исходным кодом, которые писали код, положенный в основу демонстраций в этой книге, а также преподавателям и энтузиастам ИИ, продолжающим бесплатно делиться своими знаниями.

Я обращаюсь к глобальному сообществу разработчиков – ваша любознательность и стремление ответственно подходить к созданию интеллектуальных систем каждодневно вдохновляют меня.

Наконец, я сердечно признательна читателям – тем, для которых ИИ является не просто инструментом, а новой формой рассуждения. Вы истинные исследователи нашего времени.

Искусственный интеллект уже не сводится к генерации текстов – он начинает рассуждать, искать информацию и действовать.

Эта трансформация знаменует переход от языковых моделей как движков диалоговых систем к автономным системам, которые планируют, обучаются и совершенствуются.

Книга, которую вы держите в руках, была написана, чтобы сделать эту эволюцию доступной каждому. Она адресована всем, кто хочет понять, как устроены и работают современные интеллектуальные системы и – что еще важнее – как строить их исходя из теоретических принципов.

Книга начинается с изложения основ: как «думают» трансформеры и почему большие языковые модели (LLM) ограничены статическими обучающими данными. Далее рассказывается о генерации с дополненной выборкой (retrieval-augmented generation – RAG), графах знаний и рассуждениях, основанных на самоанализе и осмыслении, а завершается все описанием архитектуры многоагентного сотрудничества.

В каждой главе теория перемежается с приложениями, что позволит вам освоить идеи и паттерны кода, которые напрямую транслируются в работающие системы. Каждый раздел «Агент в действии» иллюстрирует превращение идеи в реализацию.

Эта книга не задумана как очередное собрание пособий. Вместо этого в ней предлагается схема разработки когнитивного поведения – метод создания ИИ-агентов, которые опираются на факты, контекст и самокорректируются. Настоящая работа является первой частью двухтомника.

В этом томе основное внимание уделяется проектированию и построению индивидуальных агентов и небольших команд агентов, а во втором будет рассмотрено, как эти интеллектуальные системы масштабируются в распределенные коллаборативные сети, способные решать задачи на уровне предприятия и экосистемы.

Оглавление

Предисловие от издательства 9
Об авторе 10
Как организована эта книга 11
Обучение посредством действия 13
Для кого предназначена эта книга 13
Как пользоваться книгой на практике 13
Что дальше? 13

Глава 1. Новая эра ИИ-агентов 16

1.1. От чат-ботов к когнитивным системам 16
1.2. Из чего состоит ИИ-агент: поиск, рассуждение, смысление и действие 19
1.3. Переход от статичных LLM к адаптивным системам 22
1.4. Ключевая идея – LLM генерируют текст, агенты генерируют результаты 25
1.5. Практический пример: самообновляющийся исследовательский помощник 28
1.6. Агент в действии: построение простого вопросно-ответного агента с применением GPT и внешнего API 30

Глава 2. Как «мыслят» LLM: трансформер и не только 42

2.1. О нейросетевом движке, лежащем в основе современного ИИ 42
2.2. Предобучение, базовое дообучение и инструктивное дообучение 46
2.3. Ограничения LLM: память, галлюцинации и пробелы в рассуждениях 50
2.4. Введение в вызовы функций, погружение и контекстную осведомленность 54
2.5. От ChatGPT к Gemini и Claude: архитектурные различия 58
2.6. Агент в действии: построение модели с рассуждениями, оптимизированными под запрос с помощью вызовов функций 61

Глава 3. RAG: основа правдивых агентов 66

3.1. Генерация с дополненной выборкой как средство обоснования фактами 66
3.2. Архитектура RAG-систем: поисковик, ранжировщик, генератор 70
3.3. Векторные базы данных: FAISS, Pinecone, Chroma и Weaviate 74
3.4. Разбиение документов на фрагменты, погружения и логика поиска 79
3.5. Метрики оценки RAG, отладка и повышение правдивости систем генерации с дополненной выборкой 83

3.6. Агент в действии: построение мини-бота знаний на основе RAG, который умеет подгружать факты, разбивать на фрагменты, искать и генерировать обоснованные ответы 87
3.7. На практике: как RAG доводит концепцию корпоративных чат-ботов до эксплуатации: построение крупномасштабного обоснованного интеллекта 94

Глава 4. Графы знаний: придание структуры хаосу 100
4.1. Что такое граф знаний 100
4.2. Как LLM могут использовать и обновлять структурированные данные 103
4.3. Интеграция Neo4j, ArangoDB и GraphRAG 107
4.4. Создание сущностей, связей и контекстуальных путей – семантического каркаса интеллектуальных систем 113
4.5. Запросы на языках Cypher и SPARQL 117
4.6. Агент в действии: соединение LLM с основанным на графах конвейером рассуждений 122
4.7. На практике: графы знаний в фармацевтике, финтехе и академических учреждениях 127

Глава 5. Когнитивные циклы: разум агента 133
5.1. Как агенты планируют, рассуждают и осмысляют 133
5.2. Системы памяти: кратковременная и долговременная 137
5.3. Роль контекстных окон и погружений 141
5.4. Модели действий и циклы самооценки 146
5.5. Рефлексивные и реактивные агенты 151
5.6. Агент в действии: построение самокорректирующегося ИИ-ассистента с циклами осмысления 156

Глава 6. Многоагентные системы: сотрудничество и координация 162
6.1. От индивидуальных ботов к командному интеллекту 162
6.2. Роли, коммуникация и делегирование работы другим агентам 167
6.3. Фреймворки: CrewAI, LangGraph и AutoGPT 173
6.4. Архитектура многоагентного рабочего пространства: планировщик, исполнитель, оцениватель 179
6.5. Разрешение конфликтов и кооперативная память 186
6.6. Агент в действии: создаем многоагентную «команду ИИ-стартапа» (Исследователь, Разработчик, Рецензент) 192

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Мира С. Девлин (Mira S. Devlin) – исследователь и проектировщик систем искусственного интеллекта, специализирующаяся на архитектуре больших языковых моделей, генерации с дополненным поиском (RAG) и агентном интеллекте. В ее работах соединяются когнитивный дизайн, рассуждающие системы и прикладное машинное обучение. Преподавательская и писательская деятельность Миры направлена на то, чтобы сделать передовые концепции искусственного интеллекта практичными, объяснимыми и доступными для всех, кто увлечен интеллектуальными системами.

Как организована эта книга

Эта книга состоит из двух частей, призванных помочь читателю пройти путь от понимания основ до практического проектирования и реализации интеллектуальных ИИ-агентов, осведомленных о контексте.

Сначала описывается теоретическая структура больших языковых моделей, а затем – построение коллаборативных агентных систем, основанных на рассуждении и поиске информации.

Каждая глава зависит от предыдущих, но при этом остается относительно замкнутой и допускающей независимые ссылки.

В тексте концептуальная ясность сочетается с практическими демонстрациями.

Часть I. Что такое интеллектуальность

В первой части рассматриваются когнитивные и архитектурные основы современных интеллектуальных систем. Она начинается с эволюции языковых моделей от пассивных генераторов текста к адаптивным рассуждающим агентам. Читатель получает представление об анатомии ИИ-агентов: поиск информации, рассуждение, осмысление и действие, а затем знакомится с внутренним устройством трансформеров, логикой генерации с дополненной выборкой (RAG) и сутью перехода от реактивного интеллекта к обоснованному.

После ознакомления с этой частью читатель будет знать, как проектировать агенты, способные не только к поддержанию диалога, но и к структурированным рассуждениям, основанным на фактах.

Глава 1. Новая эра ИИ-агентов

Прослеживается историческое и концептуальное развитие от чат-ботов до когнитивных систем. Рассказывается о четырех основополагающих способностях агентов и демонстрируется простой агент, умеющий искать информацию и отвечать на запросы о фактах.

Глава 2. Как «мыслят» LLM: трансформер и не только

Объясняются механизмы работы больших языковых моделей, включая внимание, токенизацию и обработку контекста. Рассматриваются методы обучения, отмечаются ограничения моделей и исследуются архитектурные различия между наиболее распространенными системами.

Глава 3. RAG: основа правдивых агентов

Генерация представляется как средство создания языковых моделей на основе аутентичных данных. Читатель узнает о структуре RAG-систем, функциях векторных баз данных и о том, как оценивать качество поиска. Глава завершается практическим примером бота, подключающегося к базе знаний.

Часть II. Построение фундамента интеллектуальной системы

Во второй части мы переходим от понимания когнитивных процессов к их инженерной разработке.

Читатель узнает, как встраивать структуру, память и сотрудничество в интеллектуальные системы. Объясняется, как графы знаний предоставляют реляционный контекст, как когнитивные циклы обеспечивают самосовершенствование и как несколько агентов координируют свои действия для решения сложных задач.

К концу этой части читатель сможет создавать интеллектуальные агенты, которые самостоятельно рассуждают, оценивают свое качество и взаимодействуют в многоагентном окружении.

Глава 4. Графы знаний: придание структуры хаосу

Вводится понятие графа знаний и их роль в контекстуальных рассуждениях. Описывается интеграция графовых баз данных с языковыми моделями и демонстрируется, как структурированные данные повышают надежность и объяснимость.

Глава 5. Когнитивные циклы: разум агента

Рассматривается циклический процесс планирования, действия, осмысления и пересмотра, который позволяет агентам самостоятельно совершенствоваться. Представлены методы управления кратковременной и долговременной памятью, контекстными окнами и механизмами самоанализа.

Глава 6. Многоагентные системы: сотрудничество и координация

Исследуются системы, в которых несколько агентов работают сообща для достижения общих целей. Читатель знакомится с протоколами связи, системами координации и стратегиями разрешения конфликтов и поддержания кооперативной памяти. В конце главы описывается проект автономной «команды ИИ-стартапа».

Обучение посредством действия

Каждая глава завершается практическим разделом «Агент в действии».

В этих упражнениях представлены короткие, ориентированные на реализацию проекты, которые иллюстрируют основные идеи, изложенные в главе.

Написаны они так, чтобы поощрить эксперименты и модификации, что позволит читателю превратить каждый прототип в более сложную или зависящую от конкретной предметной области систему.

Для наглядности основные концепции сопровождаются диаграммами и иллюстрациями архитектуры.

Читателям рекомендуется воспроизвести эти диаграммы в своих блокнотах или инструментах автоматизированного проектирования в процессе построения проектов.

Для кого предназначена эта книга

Эта книга написана для разработчиков, исследователей и специалистов, желающих перейти от проектирования запросов к проектированию интеллектуальных систем.

Знакомство с основами программирования на Python и концепциями машинного обучения будет полезно, хотя и не обязательно.

Материал предполагает интеллектуальное любопытство и готовность к концептуальному осмыслению способов проектирования и совершенствования искусственного интеллекта.

Как пользоваться книгой на практике

Есть два подхода к чтению этого текста.

- Последовательный – начать с главы 1 и читать подряд, пройдя весь путь от фундаментальной теории к реализации.
- Модульный – читать избранные главы, относящиеся к текущим проектам (например, сосредоточиться на RAG-системах или координации нескольких агентов).

Каждая глава построена как учебный модуль и справочный материал – с четкими определениями, подведением итогов и допускающими воспроизведение примерами.

Что дальше?

Этот том закладывает фундамент для построения интеллектуальных, способных к осмыслению и взаимодействующих агентов.

В следующем томе эти идеи получат развитие и будет описано построение распределенных и масштабируемых агентных систем – сетей координированных объектов ИИ, способных к сложным рассуждениям и автономным операциям в большой окружающей среде.

Читатели, освоившие материал этой книги, будут хорошо подготовлены к работе с этими более крупными и амбициозными архитектурами.

Часть I

Что такое интеллектуальность

Глава 1

Новая эра ИИ-агентов

1.1. От чат-ботов к когнитивным системам

Поворотная точка в искусственном интеллекте

Менее чем за десять лет искусственный интеллект совершил скачок от составления предложений к выполнению заданий. То, что начиналось как простые чат-боты, способные вести светскую беседу, превратилось в автономных агентов – системы, которые могут рассуждать, искать информацию и действовать в сложных окружающих средах почти без участия человека.

Подумайте о разнице между Siri, спрашивающей, какую песню вы хотели бы послушать, и исследовательским ИИ-агентом, который самостоятельно просматривает новые научные статьи, выявляет пробелы в исследованиях, обобщает результаты и выдвигает гипотезу. Это не просто чат-бот с увеличенной моделью, а когнитивная система.

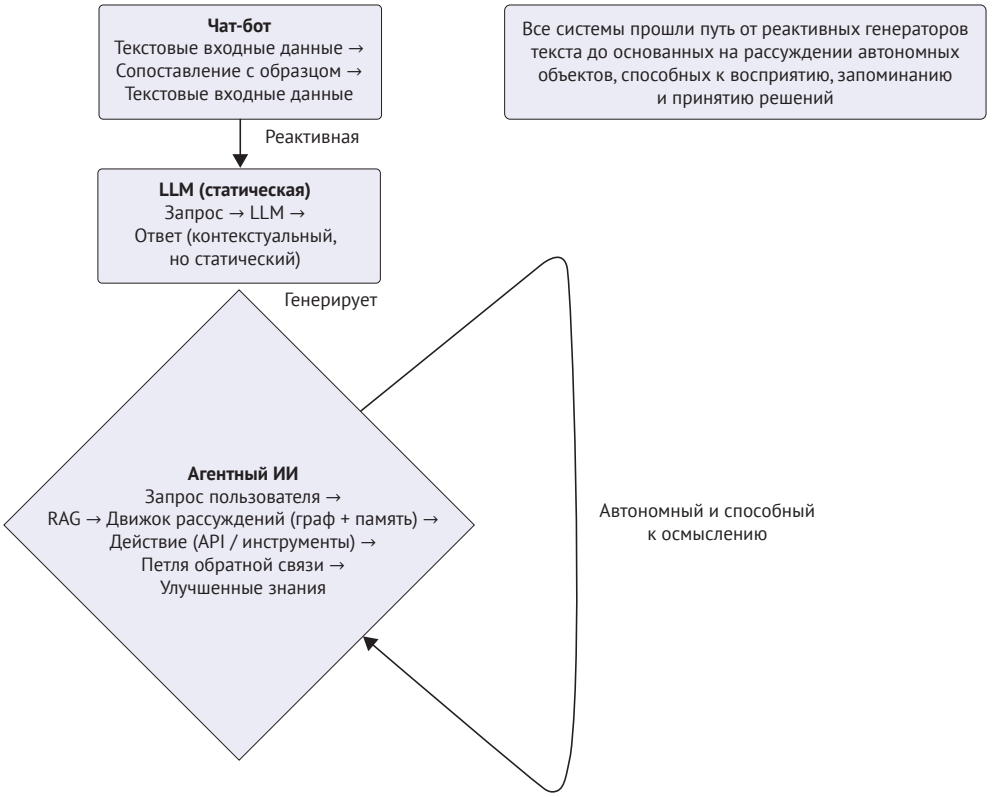
Этот переход знаменует агентную революцию – новый этап, когда LLM является не просто средством для обработки слов, а движком рассуждения и принятия решений.

От слов к мирам

Традиционная языковая модель типа GPT-3 или Claude-2 работает в закрытом ящике: она предсказывает следующее слово на основе паттернов в обучающих данных. Она превосходно генерирует связный текст, но не «знает» ни о чем находящемся за пределами своего контекстного окна.

С другой стороны, ИИ-агент обитает в открытом мире. Он не останавливается не генерации текста, а в режиме реального времени ищет информацию из внешних источников (с помощью RAG), рассуждает об этой информации, применяя структурированные знания (организованные в виде графов или памяти), и автономно действует, обращаясь к помощи API, инструментов или других агентов.

Эта триада – поиск, рассуждение, действие – образует основу агентного интеллекта.



Чего не хватило чат-ботам

Ранние чат-боты могли имитировать диалог, но не были способны на понимание контекста. Им недоставало обоснованности – они могли догадываться, что вы имеете в виду, но не могли проверить свою догадку.

Поэтому чат-бот мог бы уверенно сказать: «Высота Эйфелевой башни равна 1024 футам», даже если бы вы переформулировали вопрос в виде «Чему равна высота Эйфелевой башни в метрах?». Здесь изменен только текст вопроса, но не факты.

Эта «проблема галлюцинаций» обнажила важнейшую истину: понимание языка ≠ рассуждению на основе знаний.

Развитие агентных систем было призвано устранить этот разрыв, связав языковые модели с источниками данных, системами рассуждений и интерфейсами к действиям. По сути дела, они наделили LLM разумом, памятью и средствами действовать.

Определение современного ИИ-агента

ИИ-агент можно определить следующим образом.

Система, которая использует языковую модель в качестве ядра рассуждений, дополненная памятью, поиском внешних данных и интерфейсами с инструментальными средствами и предназначенная для автономного достижения цели.

Каждый из этих слоев превращает пассивную модель в активного «мыслителя» – не разумного в том смысле, в каком мы применяем это слово к людям, но движимого целью, итеративного и осведомленного о ситуации.

Агентный цикл: интеллект в действии

Отличительной чертой интеллектуального агента является цикл, в котором он воспринимает, планирует и выполняет действия.

Упрощенно цикл выглядит следующим образом:

- восприятие: прочитать входные данные или контекст (запрос пользователя, состояние данных);
- поиск: собрать дополнительную информацию (RAG, база данных или память);
- рассуждение: сгенерировать план действий (декомпозиция задачи, осмысление);
- действие: выполнить шаги с помощью доступных инструментов или API;
- осмысление: проанализировать результат, скорректировать подход и повторить.

Этот непрерывный цикл совершенствования отражает подход человека к решению задачи, и именно он отличает агента от модели.

Пример: от ответов на вопросы к настоящей автономности

Рассмотрим конкретный пример трансформации.

- Чат-бот: «Какова рыночная капитализация компании Tesla?» → статический поиск в старых данных.
- Агент: «Проанализировать последние квартальные отчеты Tesla, сопоставить рост от кода к году и визуализировать результаты».

Агент ищет финансовые данные с помощью API, выполняет запрограммированный на Python анализ, строит график и даже составляет пояснительную записку – все это без дальнейшего участия человека.

Речь идет не о более масштабной модели, а о связной итеративной системе, которая обучается благодаря структурированной обратной связи.

ИИ-агенты представляют собой следующий этап в развитии машинного интеллекта, когда системы больше не полагаются на полученную в ходе обучения информацию, а могут динамически подключаться, производить вычисления и делать выводы в реальных условиях.

В следующем разделе мы рассмотрим, почему эта трансформация произошла именно сейчас – технологические и когнитивные прорывы, которые сделали «агентную революцию» неизбежной.

1.2. Из чего состоит ИИ-агент: поиск, рассуждение, осмысление и действие

Анатомия машинной автономности

Говоря об ИИ, мы описываем не какую-то одну модель или алгоритм, а интеллектуальную экосистему.

Агент состоит из нескольких частей: он воспринимает окружающую среду, рассуждает о своих целях, находит внешние знания, выполняет действия и обучается на результатах.

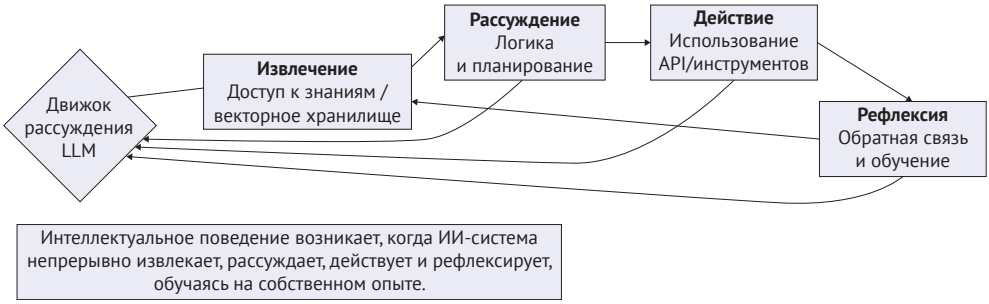
По сути дела, ИИ-агенты – это системы, построенные на основе обратной связи. Это не монолог предсказания, а диалог между восприятием, рассуждением и адаптацией. Если чат-бот – это одностороннее зеркало (ввод → вывод), то агент способен «думать».

Четыре столпа когнитивной системы

Интеллект ИИ-агента опирается на четыре фундаментальных процесса:

- поиск (Retieval) – доступ к нужной информации в нужное время;
- рассуждение (Reasoning) – интерпретация этой информации в контексте;
- осмысление (Reflection) – оценка того, что сработало, а что нет;
- действие (Action) – выполнение планов во внешнем мире.

Все вместе они образуют цикл R^3A (Retrieval, Reasoning, Reflection, Action) – структуру, в основе которой лежат когнитивистика и системы управления.



Поиск: погружение агента в реальный мир

LLM обучаются на больших корпусах текстов, но их внутренние знания статичны – заморожены во времени. Когда необоснованную, т. е. не имеющую связи с внешним миром LLM просят конспективно изложить последние сведения о программе НАСА «Артемида», она может лишь сфабриковать какую-то информацию, потому что ее память заканчивается на том моменте, до которого было доведено обучение.

Генерация с дополненной выборкой (RAG) решает эту проблему, связывая модель с внешними источниками знаний: документами, базами данных, API или даже «живым» вебом.

Благодаря поиску, основанному на погружении, агент преобразует запрос в вектор чисел, сравнивает его с векторной базой данных и извлекает семантически наиболее релевантные фрагменты. Затем эти фрагменты становятся контекстом для процесса рассуждения модели.

Поиск превращает модель в информированную систему, способную давать точные ответы, зависящие от контекста. Без поиска LLM обладает богатым воображением, но невежественна. С поиском она становится информированной и полезной.

Рассуждение: мозг определяет поведение

Рассуждение – это процесс, в ходе которого модель интерпретирует найденный контекст, планирует действия и выстраивает последовательность шагов к достижению цели.

Современные фреймворки, такие как LangChain, CrewAI и LangGraph, имитируют рассуждение, разбивая большие цели на более мелкие подзадачи – процесс, известный как декомпозиция задачи. Агент составляет «план» (цепочку рассуждений), выполняет подзадачи и объединяет результаты.

Существует несколько схем рассуждений:

- последовательное рассуждение (линейное пошаговое планирование);
- параллельное рассуждение (делегирование задач нескольким подагентам);
- древовидное рассуждение (исследование ветвей и оценка путей).

Следовательно, интеллект агента кроется не в весах его нейронов, а в структуре мышления – как он организует свои решения, оценивает контекст и выполняет планы.

«Модель предсказывает, агент планирует».

Осмысление: машинное метапознание

Осмысление привносит в картину метаосведомленность – способность анализировать собственные рассуждения.

У людей это проявляется в виде вопроса «Все ли я сделал правильно?». А в ИИ принимает форму самоаналитических запросов, обновления памяти и обратной связи с вознаграждением.

Например, после выполнения поиска агент мог бы проанализировать свой результат, воспользовавшись вспомогательной LLM: «Является ли мое конспективное резюме правильным ответом на исходный вопрос?» Если нет, то агент снова ищет информацию, изменяет план и корректирует свой подход.

Осмысление преобразует подход на основе проб и ошибок в самосовершенствование.

Это начало машинного метапознания – систем, которые обучаются не только на данных, но и на анализе качества своей работы.

Некоторые современные фреймворки реализуют циклы осмысления посредством:

- самокритичных запросов (фреймворки ReAct, Reflexion);
- обновлений памяти (сохранение уроков в качестве будущего контекста);
- сигналов подкрепления (назначения оценок успехам и неудачам).

Осмысление – движок совершенствования. Без него агент выходит на плато. С ним агенты эволюционируют.

Действие: интерфейс между мыслью и внешним миром

Любое рассуждение бессмысленно, если оно не ведет к действию.

Действие – это то место, где агенты сопрягаются с окружающей средой – выполняют код, вызывают API, управляют программным обеспечением или взаимодействуют с людьми.

Благодаря применению инструментов LLM не ограничиваются операциями с текстом, а манипулируют данными, совершают вызовы API или автоматизируют технологические процессы.

Например:

- исследовательский агент находит статьи → реферирует их → вызывает средство построения графиков для визуализации тенденций;
- торговый агент анализирует рыночные данные → выполняет имитацию торгов → осмысляет прибыли и убытки;
- медицинский помощник запрашивает базы данных о симптомах → ставит дифференциальный диагноз → предлагает дальнейшие шаги по диагностике.

Каждое действие подает новую информацию в петлю обратной связи, создавая непрерывный цикл: восприятие → познание → выполнение → обучение.

В современных архитектурах действия обычно опосредуются:

- инструментами LangChain и агентами;
- циклом REPL в Python или конечными точками API;
- внешними скриптами или интеграцией с внешними программами (Zapier, Notion, Google API).

Собираем все вместе

Представим этот процесс наглядно на реальном примере.

Пример: исследовательский агент-ассистент

Поиск: находит последние научные статьи по квантовым вычислениям.

Рассуждение: реферирует все статьи и выявляет главные тенденции в проведении экспериментов.

Осмысление: анализирует, какие открытия коррелируют с целями текущего проекта.

Действие: обновляет базу данных Notion и генерирует отчет для человека.

На каждой итерации точность и эффективность повышаются – мы имеем непрерывный цикл обучения, имитирующий человеческий процесс познания, но со скоростью машины.

Принцип R³A: новая парадигма интеллекта

Благодаря соединению поиска, рассуждения, осмысления и действия агенты знаменуют переход от пассивного машинного предсказания к активному интеллекту.

Они не просто отвечают на вопросы, а преследуют определенные цели.

ИИ-агент определяется не столько моделью, сколько своим циклом.

Поиск поставляет истинные факты, рассуждение привносит структуру, осмысление является видом обучения, а действие порождает результаты.

В совокупности они образуют бьющееся сердце интеллектуальной автономности.

1.3. Переход от статичных LLM к адаптивным системам

От неподвижного знания к живому интеллекту

Большие языковые модели (LLM) изменили способ, которым машины понимают и порождают языковые конструкции, но они никогда не были динамичными.

Это статичные монолиты: будучи один раз обучены, они застыли во времени и не способны обновить свое представление о мире после развертывания.

Подобная жесткость была вполне пригодна для решения таких задач, как обобщение документов или составление черновиков текста. Но если мир начинает эволюционировать быстрее, чем обучающие данные, – вследствие новых открытий, меняющихся правил и предписаний, динамичности информации, – статичные LLM быстро устаревают.

Статичная LLM может имитировать, но не может развиваться.

Им на смену приходят адаптивные системы – следующий этап эволюции искусственного интеллекта, – когда модели перестают быть изолированными, а становятся взаимосвязанными, осведомленными о контексте и самообновляющимися благодаря взаимодействию с реальными данными и непрерывному рассуждению.

Статичные LLM: блестящие, но слепые

Чтобы оценить этот тезис, еще раз посмотрим, как работают традиционные LLM.

Модель типа GPT-3 или LLaMA-2, обученная на триллионах токенов, усваивает статистические паттерны, существующие между словами. Она улавливает необычайно большой объем лингвистических и концептуальных связей, но после обучения ее мир оказывается застывшим.

Спросите ее, кто получил Нобелевскую премию по физике за 2025 год, и она не ответит – не потому, что глупа, а потому, что обитает в закупоренной временной капсуле.

У статичных LLM есть три основных ограничения:

- отсутствие осведомленности в реальном времени – они не могут ни получить, ни обработать новые данные;
- отсутствие памяти рассуждений – каждый запрос обрабатывается независимо;
- отсутствие автономности – они не умеют делать ничего, кроме дополнения текста.

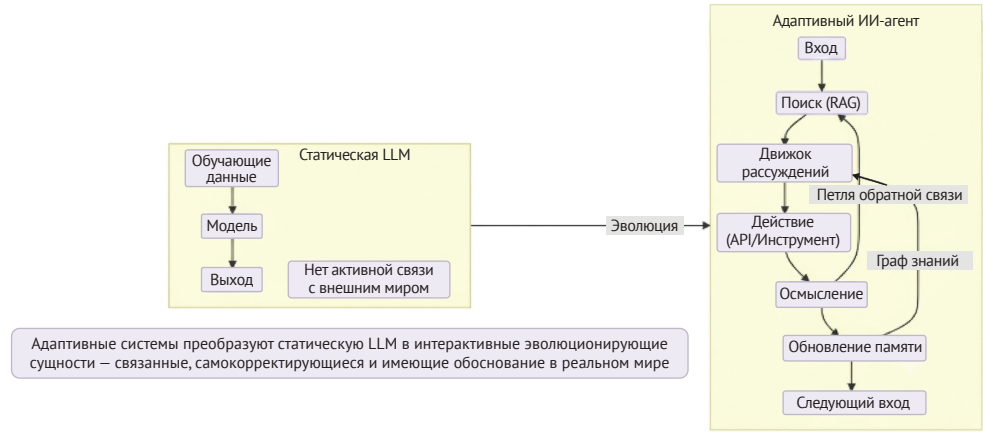
Эти ограничения позволили провести важную границу: языковое предсказание ≠ интеллект.

Адаптивные системы: связь моделей с окружающим миром

Адаптивные системы устраняют этот пробел, наделяя LLM памятью, обоснованностью и функциональностью агентов.

Они связывают языковую модель с внешними инструментами, базами данных и окружениями, создавая живую архитектуру, способную непрерывно развиваться.

В итоге адаптивная система не просто генерирует ответы, а вырабатывает понимание. Она наблюдает за результатами, уточняет свою внутреннюю логику и постепенно достигает совершенства в достижении целей.



Ключевые факторы, обеспечивающие адаптивность

Переход к адаптивному интеллекту стал возможен благодаря нескольким инновациям, усиливающим друг друга:

- генерация с дополненной выборкой (RAG):

  в реальном времени обеспечивает доступ к актуальной фактической информации, выходящей за пределы обучающих данных, → LLM получают возможность поиска знаний;
- интеграция с графами знаний:

  привносит структурированные связи между концептами, сущностями и фактами → LLM получают контекст для рассуждений;
- системы памяти (векторные + эпизодические):

  сохраняют историю диалогов, обеспечивают осмысление и обучение на предшествующих сеансах → LLM приобретают непрерывность и долгосрочную адаптацию;
- использование инструментов и оркестрация API:

  позволяет моделям выполнять такие действия, как выполнение кода и поиск в сети, → LLM получают функциональность агентов;

- циклы подкрепления и осмысления:

  наделяют агентов средствами самоанализа и уточнения стратегий → LLM получают возможности самосовершенствования.

В совокупности эти технологии превращают LLM из статичного генератора знаний в динамичный процессор знаний, способный к взаимодействию, адаптации и постоянно остающийся актуальным.

Аналогия с человеком: студент и исследователь

Статичную LLM можно представлять себе как студента, который выучил наизусть все учебники, но перестал учиться после получения диплома. Он может вспомнить факты и написать эссе, но не может рассказать о новшествах в своей области.

А адаптивного агента можно представлять себе как исследователя:

- он ежедневно читает новые статьи;
- связывает идеи, относящиеся к разным дисциплинам;
- проверяет гипотезы;
- обновляет свои знания на основе результатов.

Статичные системы вспоминают то, что знают; адаптивные системы – создают новые знания и развиваются. Адаптивная система не просто отвечает на вопросы, а учится отвечать лучше в следующий раз.

Рождение динамического интеллекта

Адаптивные системы представляют собой фундаментальный сдвиг в философии искусственного интеллекта – от припоминания знаний к их эволюции.

Им свойственны три определяющих признака:

- осведомленность о контексте – сохранение и применение памяти при выполнении задач;
- причинно-следственные рассуждения – планирование действий на основе целей, а не запросов;
- непрерывное обучение – использование обратной связи для уточнения будущих результатов.

Это знаменует зарождение «живых» ИИ-систем – не в биологическом смысле, а в информационном. Они воспринимают, обрабатывают, действуют и эволюционируют в своей цифровой среде.

Статические модели обучаются, адаптивные агенты непрерывно обучаются на основе опыта.

Адаптивность превращает интеллект из неподвижного продукта в живой процесс, соединяя LLM с окружающим миром, обеспечивая им обоснованность и позволяя действовать целеустремленно.

1.4. Ключевая идея – LLM генерируют текст, агенты генерируют результаты

Смена акцентов в определении интеллекта

В первые годы развития генеративного ИИ мы оценивали успех по беглости речи – насколько хорошо модель могла предсказать следующее слово, сохранить связность или имитировать интонацию.

Но беглость – еще не интеллект. Это поверхностная иллюзия мышления.

Большая языковая модель может генерировать звучащий разумно текст, потому что изучила структуру человеческого общения. Однако за этим красноречием не скрывается понимание целей, истины или последствий.

Именно этим и отличаются ИИ-агенты.

Ключевая идея: «LLM генерируют текст, агенты генерируют результаты».

Это простое утверждение отражает важнейшее различие – между дополнением языка и выполнением целей.

LLM: мастера выражения, но узники контекста

LLM типа GPT-4 или Claude 3 могут создавать удивительно детальные эссе, код или объяснения, но их работа фундаментально ограничена.

Они отвечают на запрос, генерируя статистически наиболее вероятный текст. Попросите LLM «написать маркетинговую стратегию для экологически чистого стартапа», и она составит связный, структурированный план. Но этот план является выводом, а не результатом. По сути дела:

- мерой успеха LLM является лингвистическое правдоподобие – «разумно ли это звучит?»;
- мерой успеха агента является функциональное достижение – «была ли достигнута поставленная цель?».

Агенты: от коммуникации к достижению цели

ИИ-агент наследует возможности LLM и встраивает их в замкнутую систему, которая взаимодействует с окружающим миром.

Вместо того чтобы остановиться на генерации текста, он:

- ищет факты и данные, которые можно было бы положить в основу рассуждения;
- планирует последовательность шагов для достижения цели;
- действует путем выполнения кода, обращения к API или взаимодействия с другими агентами;
- осмысляет успешность, корректирует план и повторяет.

Например:

- LLM: «Вот как можно проанализировать эмоциональное состояние клиента по его сообщениям в Twitter»;
- агент: фактически извлекает твиты, выполняет анализ, обобщает полученные данные и сохраняет результаты на информационной панели.

LLM порождает текст, агент производит воздействие.

ИИ-агент
Цель
Поиск
Движок рассуждений
Действие (инструмент/API)
Обратная связь
Результат
Целеориентированная; осведомленная о контексте, осмысляющая
Эволюция интеллекта → От слов к результату
Большая языковая модель
Запрос
LLM
Текстовый вывод
Предсказывающая, с ограниченным контекстом, реактивная
LLM – языковые эксперты. Агенты – решатели задач

Почему это различие важно

Непонимание возможностей ИИ возникает главным образом из-за того, что генерацию путают с интеллектом.

Бегло говорящая LLM может показаться разумной, но она не способна проверить истинность или преследовать какую-то цель. Она отвечает на вопрос «что?», но не «почему?» или «для чего?».

Напротив, ИИ-агент ориентирован на решение задач. Показателем его успеха является не связность речи, а достижение результата. Независимо от того, является ли задача обобщением результатов исследований, написанием кода или оптимизацией технологического процесса, агент оценивает себя не с точки зрения риторики.

LLM красиво говорит.

Агент выполняет осмысленные действия.

Практическая иллюстрация: от чата к действию

Давайте рассмотрим это различие на примере реального сценария.

Задача: «Составить отчет о финансовых результатах в третьем квартале».

Это и есть суть перехода – от разговоров к действию.

Когнитивный скачок: намерение и причинно-следственная связь

За этим поведенческим отличием кроется более глубокая когнитивная трансформация – осознание намерений.

LLM обрабатывают язык как корреляцию.

Агенты обрабатывают его как инструкцию. Это означает, что агенты могут интерпретировать не только то, о чем их спрашивают, но и почему, – и соответствующим образом выстраивать свои действия.

Они моделируют причинно-следственную связь, а не просто вероятность.

Когда вы говорите: «Выясните, почему активность пользователей снизилась в этом месяце», LLM выдвигает гипотезы, а агент исследует: он собирает аналитические данные, анализирует закономерности, сравнивает с историческими данными и предлагает проверенные идеи.

Это рассуждение о причинах и следствиях знаменует рождение машинной интенциональности (направленности на предмет).

Новый показатель успеха ИИ

На протяжении десятилетий ИИ оценивался по точности, сложности или метрике качества перевода BLEU (bilingual evaluation understudy).

Теперь, в эпоху агентных систем, мы оцениваем успех по достижению целей – по тому, может ли система делать что-то полезное надежно и автономно.

Эта эволюция говорит о зрелости ИИ, а именно о переходе от моделей, которые имитируют конечные результаты, полученные человеком, к системам, которые воспроизводят цели человека.

LLM общаются. Агенты вносят вклад в результаты.

LLM генерируют предложения. Агенты генерируют решения.

В будущем ИИ язык перестанет быть конечным продуктом – он станет интерфейсом между намерением и разумным действием.

1.5. Практический пример: самообновляющийся исследовательский помощник

Почему этот пример важен

Научный мир меняется ежеминутно. Публикуются новые статьи, обновляются наборы данных, развиваются теории – и все это быстрее, чем способен уследить человек.

Традиционные поисковые системы и статичные LLM с этим не справляются, потому что:

- поисковые системы находят исходные данные, но не предлагают никакой их обработки;
- LLM синтезируют тексты, но на основе уже устаревших обучающих данных.

Нужна система, способная читать, рассуждать и обновлять себя, – цифровой помощник исследователя, который не просто ответит на сегодняшний вопрос, но будет актуален и завтра.

Именно это и делает исследовательский ИИ-агент. Он не просто «знает», он продолжает пополнять свои знания.

От помощника к сотруднику

Представьте себя ученым, занимающимся исследованиями по теме «конструирование белков с применением ИИ». Вы просите агента: «Найди и обобщи опубликованные на этой неделе статьи об укладке белков на основе искусственного интеллекта». За считанные минуты агент производит следующие действия:

- опрашивает API баз данных arXiv, PubMed и Semantic Scholar;
- находит статьи, опубликованные за последние 7 дней;
- извлекает из них авторефераты, ключевые слова и авторов;
- пользуется генерацией с дополненной выборкой (RAG), чтобы обобщить каждую статью;
- обновляет свои знания, связывая модели, наборы данных и результаты;
- сохраняет новый контекст в своей векторной памяти;
- сообщает: «Найдено 3 новые статьи. Главная тенденция: диффузионные модели превосходят производные от AlphaFold при наличии небольшого количества данных».

В следующий раз, когда вы спросите об укладке белков, агент автоматически обратится к этим новым статьям.

Никакого переобучения. Никакой ручной загрузки на сервер.

Агент обучился в ходе взаимодействия.

Какие технические средства стоят за всем этим

Такую систему можно построить с помощью легкодоступных фреймворков и API. Ниже приведен пример архитектуры, сочетающей ясность и практичность.

Этот модульный подход гарантирует, что агент не просто «вызывает API», а обдумывает данные.

Внутри цикла агента

Давайте расшифруем цикл R^3A (Поиск → Рассуждение → Осмысление → Действие) применительно к этому помощнику.

1. Поиск.

   Извлекает последние статьи с помощью API и сохраняет их во временном наборе данных.

2. Рассуждение.

   Использует LLM, чтобы идентифицировать повторяющиеся темы, новые модели или упоминаемые наборы данных.

3. Осмысление.

   Сравнивает с существующим графом:

   - являются ли эти находки чем-то новым?
   - противоречат ли они чему-то?
   - связаны ли они с ранее выявленными сущностями (авторами, научными учреждениями, ключевыми словами)?

4. Действие.

- обновляет свой внутренний граф и векторную память;
- выводит синтезированный отчет;
- факультативно планирует повторную проверку раз в 7 дней (автономное выполнение).

Этот цикл превращает статичное средство обобщения в обучаемую систему – помощника, который «запоминает, рассуждает и пересматривает».

Магию вершит память

Здесь определяющей характеристикой является не поиск, а память.

Каждая новая статья становится частью персональной базы знаний агента, которая со временем пополняется и связывается с более старыми знаниями.

В течение нескольких недель или месяцев агент создает эволюционирующую «мысленную модель» вашей области исследований – понимание того, какие темы появляются вновь, а какие уходят на второй план, и какие исследователи или научные учреждения являются ведущими на данный момент.

Агент может отвечать на такие вопросы, как:

- что изменилось в методах укладки белков с июня?
- какие исследовательские группы работают над диффузионными моделями?
- покажите хронологию основанных на ИИ подходов в биоинформатике.

Ответы улучшаются, потому что агент помнит свои прошлые рассуждения.

Взгляд в будущее

В академическом или производственном контексте такого рода агент мог бы:

- автоматически поддерживать актуальность баз данных по литературе;
- еженедельно генерировать интеллектуальные отчеты;
- обнаруживать новые области исследований;
- служить узлом знаний в интранете организации.

Это не просто автоматизация чтения, а еще и усиление когнитивных способностей. Скоро агенты будут не просто обобщать, но и выдвигать гипотезы, связывая идеи из дисциплин, которые люди еще не связали.

Завтрашний исследователь не будет начинать каждое утро с чистого листа – его помощник уже прочитал все статьи по теме.

Самообновляющийся помощник исследователя представляет собой идеальное сочетание столпов, на которых стоит агент:

- поиск информации в режиме реального времени;
- рассуждение для понимания тенденций;
- осмысление для совершенствования;
- действие для интеграции с памятью и отчетности.

Он не просто описывает будущее исследования – он является его воплощением.

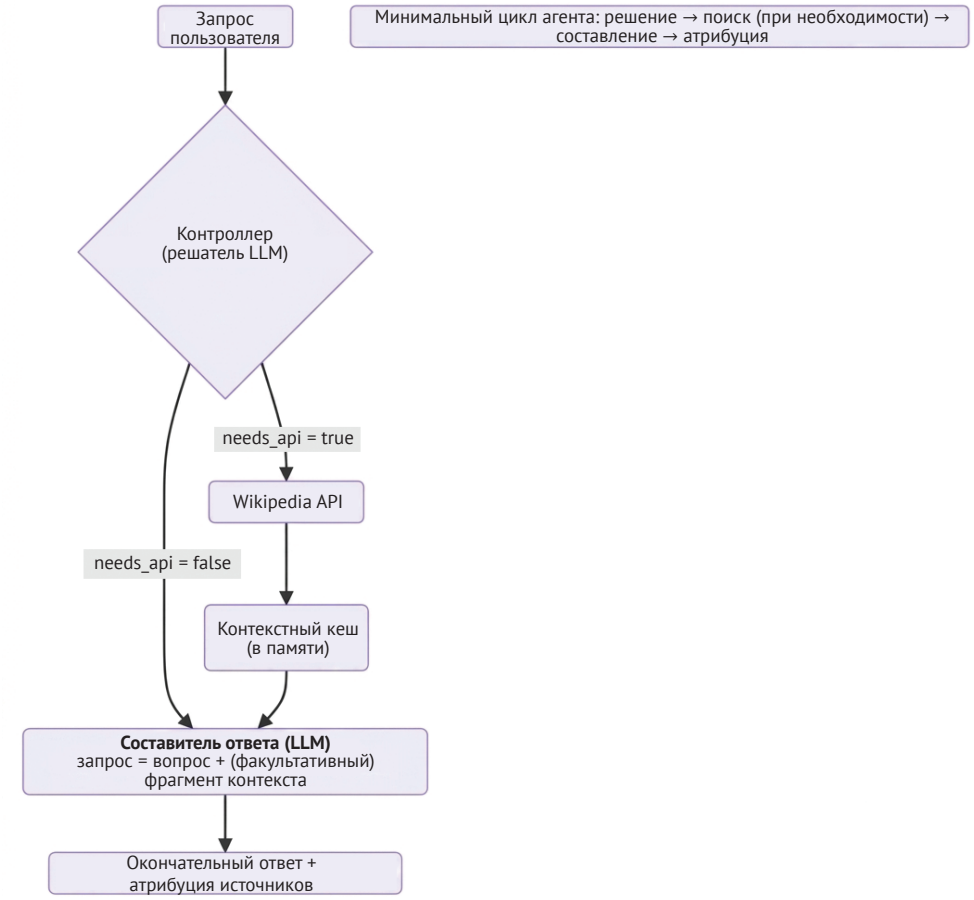
1.6. Агент в действии: построение простого вопросно-ответного агента с применением GPT и внешнего API

Что мы построим (за ~60–90 мин)

Компактный вопросно-ответный агент, который:

- понимает вопрос на естественном языке;
- решает, нужны ли ему свежие знания из интернета;
- при необходимости вызывает внешний API (Wikipedia REST);
- синтезирует обоснованный ответ со ссылками с помощью GPT;
- объясняет, какие источники он использовал.

Мы будем пользоваться как инструментом командной строки, так и конечной точкой FastAPI HTTP.



Необходимые условия

- Python 3.10+.
- Пакеты: pip install fastapi uvicorn httpx pydantic python-dotenv.
- Ключ поставщика LLM (например, set OPENAI_API_KEY или ANTHROPIC_API_KEY).
- В этой главе используются обобщенные обертки «llm_complete(...)», чтобы можно было менять поставщиков.

Замечание по безопасности: всегда загружайте ключи API из переменных окружения, никогда не зашивайте их в код.

Структура проекта

mini_qa_agent/
├─ .env
├─ agent.py # основная логика (решение → поиск → составление)
├─ tools.py # клиентский API, например Wikipedia
├─ llm.py # не зависящая от поставщика обертка LLM
├─ serve.py # сервер FastAPI
└─ cli.py # исполнитель, запускаемый из командной строки

Шаг 1 – не зависящая от поставщика обертка LLM (llm.py)

```
# llm.py
import os
import httpx

from typing import List, Dict, Any

class LLMError(Exception):
    pass
    def _openai_complete(messages: List[Dict[str, str]], model: str = "gpt-4o-mini") -> str:
        api_key = os.getenv("OPENAI_API_KEY")
        if not api_key:
            raise LLMError("Missing OPENAI_API_KEY")
        url = "https://api.openai.com/v1/chat/completions"
        payload = {"model": model, "messages": messages, "temperature": 0.2}
        headers = {"Authorization": f"Bearer {api_key}"}
        with httpx.Client(timeout=60) as client:
            r = client.post(url, json=payload, headers=headers)
            r.raise_for_status()
            return r.json()["choices"][0]["message"]["content"].strip()

    # Подменить здесь, если вы пользуетесь другим поставщиком (Anthropic и т. д.)
    def llm_complete(messages: List[Dict[str, str]], model: str = "gpt-4omini") -> str:
        return _openai_complete(messages, model=model)
```

Шаг 2 – внешний API: Wikipedia Summary (tools.py)

```
# tools.py
import httpx
from typing import Optional, Dict

WIKI_SUMMARY = "https://en.wikipedia.org/api/rest_v1/page/summary/{title}"

def wikipedia_summary(title: str) -> Optional[Dict]:
    """
    Возвращает {'title':..., 'extract':..., 'url':...} или None, если не найдено.
    """
    safe = title.replace(" ", "_")
    with httpx.Client(timeout=20) as client:
        r = client.get(WIKI_SUMMARY.format(title=safe), headers=
            {"accept": "application/json"})
```

```
        if r.status_code != 200:
            return None
        j = r.json()
        return {
            "title": j.get("title"),
            "extract": j.get("extract"),
            "url": j.get("content_urls", {}).get("desktop", {}).get("page"),
        }
```

Шаг 3 – ядро агента: решение → поиск → составление (agent.py)

Мы используем схему с двумя запросами:

- запрос к контроллеру: просит LLM решить, вызывать ли API и если да, то с каким запросом;
- запрос к составителю: просит LLM составить обоснованный ответ с цитатой.

```
# agent.py
import json
from typing import Dict, Any, Optional
from llm import llm_complete
from tools import wikipedia_summary

CONTROLLER_SYSTEM = """Ты контроллер для агента-помощника.
Реши, нужна ли актуальная внешняя информация. Выведи ответ в строгом формате JSON:
{"needs_api": true|false, "api": "wikipedia", "query": "search term>",
"reason": ""}
Используй только «Википедию», если вопрос касается фактов, людей, мест, событий или
определений, для которых важна информация из регулярно обновляемого или канонического
источника.
"""

COMPOSER_SYSTEM = """Ты добросовестный ИИ, который выдает обоснованнные ответы.
- Используй предоставленный CONTEXT буквально для поиска фактов.
- Если контекст пустой, скажи, что ответ дан на основе общих знаний.
- Включи краткий раздел 'Источники', содержащий URL-адреса, если они заданы.
- Будь краток, точен и беспристрастен.
"""

def decide_api(question: str) -> Dict[str, Any]:
    messages = [
        {"role": "system", "content": CONTROLLER_SYSTEM},
        {"role": "user", "content": f"Question: {question}\nReturn JSON only."},
    ]
    raw = llm_complete(messages)

    # Будь терпим к мелким проблемам форматирования
    start = raw.find("{")
    end = raw.rfind("}")
    parsed = {"needs_api": False, "api": "wikipedia", "query": "", "reason": "default"}
```

```
    if start != -1 and end != -1:
        try:
            parsed = json.loads(raw[start:end+1])
        except Exception:
            pass
            return parsed

def compose_answer(question: str, context: Optional[Dict[str, str]]) -> str:
    context_block = ""
    if context and context.get("extract"):
        context_block = f"TITLE: {context.get('title')}\nTEXT:
        {context.get('extract')}\nURL: {context.get('url')}"
    messages = [
        {"role": "system", "content": COMPOSER_SYSTEM},
        {"role": "user", "content":
            f"QUESTION:\n{question}\n\nCONTEXT:\n{context_block}"},
    ]
    return llm_complete(messages)

def answer(question: str) -> Dict[str, Any]:
    # 1) Решение
    decision = decide_api(question)
    context = None
    # 2) Поиск при необходимости
    if decision.get("needs_api") and decision.get("api") == "wikipedia":
        q = decision.get("query") or question
        context = wikipedia_summary(q)
    # 3) Составление обоснованного ответа
    final = compose_answer(question, context)
    return {
        "question": question,
        "decision": decision,
        "context_used": bool(context),
        "context_meta": {"title": context.get("title") if context else None,
        "url": context.get("url") if context else None},
        "answer": final
}
```

Шаг 4 – исполнитель с интерфейсом командной строки (cli.py)

```
# cli.py
import sys, json
from agent import answer

def main():
    if len(sys.argv) < 2:
        print("Usage: python cli.py \"Your question here\"")
        sys.exit(1)
    q = sys.argv[1]
    result = answer(q)
    print(json.dumps(result, indent=2, ensure_ascii=False))
```

```
    if __name__ == "__main__":
        main()
```

Тестируем:

```
export OPENAI_API_KEY=sk-...
python cli.py "Кто открыл пенициллин и в каком году?"
```

Шаг 5 – в виде HTTP-службы (serve.py)

```
# serve.py
from fastapi import FastAPI
from pydantic import BaseModel
from agent import answer

app = FastAPI(title="Mini QA Agent")

class Q(BaseModel):
    question: str
    @app.post("/qa")

    def qa(q: Q):
        return answer(q.question)
```

Запускаем сервер:

```
uvicorn serve:app—reload—port 8000
```

Тестируем с помощью curl:

```
curl -s -X POST http://localhost:8000/qa \
   -H "Content-Type: application/json" \
   -d '{"question":"What is CRISPR and who pioneered it?"}' | jq .
```

Шаг 6 – ограничения, надежность и анализ

Ограничение на уровне запроса:

- в переменной `COMPOSER_SYSTEM` запретить предположения: «Если контекст не содержит факта, сказать “недостаточный контекст”»;
- добавить стилистический контракт: «Вернуть ⩽ 150 слов. Включить источники, если таковые имеются».

Устойчивость решения

- Если при разборе JSON-кода, возвращенного контроллером, возникает ошибка, выбрать резервный путь: установить `needs_api=true` для ориентированных на сущности вопросов (эвристика: содержит имена собственные, годы или слова «кто/где/когда»).

Дисциплина атрибуции

- Всегда показывать использованные URL-адреса. Если таковых не было, сообщить «Внешние источники не использовались».

Анализ (облегченный)

- Создать небольшой набор данных из 20 пар вопросов и ответов и измерить:
  - правильность (оценивается человеком, принимает значения 0/1);
  - покрытие ссылками (% ответов с указанием источника, если необходимо);
  - частоту галлюцинаций (пометить случаи, когда ответ включает факты, отсутствующие в контексте).

Шаг 7 – полезные расширения (следующие итерации)

- Кеширование: для уменьшения задержки сохранять ответы «Википедии» (заголовок → извлеченный фрагмент), сопровождаемые временем хранения.
- Несколько источников: добавить инструменты баз данных arXiv/Semantic Scholar для предметных областей, связанных с наукой.
- Переранжирование: если используется API поисковой системы, извлекать первые *k* результатов, а перед составлением ответа переранжировать.
- Запоминание: сохранять для каждого пользователя вектор заданных вопросов и принятых ответов.
- Несколько агентов: разделить роли: Исследователь (поиск) → Писатель (составление) → Контролер (проверка на непротиворечивость).
- UI: упрощенный интерфейс, содержащий ответ + источники на разделенной надвое панели.

К чему мы пришли

- Минимальный цикл агента не нуждается в сложном фреймворке: решатель, инструмент и составитель уже дают обоснованные результаты.
- Решение → Поиск → Составление – наименьшая надежная схема вопросно-ответной системы производственного качества.
- Источники и прозрачность следует рассматривать как полноправные компоненты; они придают доверие к результатам и уменьшают галлюцинации.

От теории к практике

До сих пор мы обсуждали, что для ИИ-агента означают поиск, рассуждение, осмысление и действие.

Теперь давайте создадим небольшой, но полностью функциональный вопросно-ответный агент, который связывает языковую модель (GPT) с внешним источником данных (Wikipedia API) для генерации основанных на фактах ответов с указанием источников.

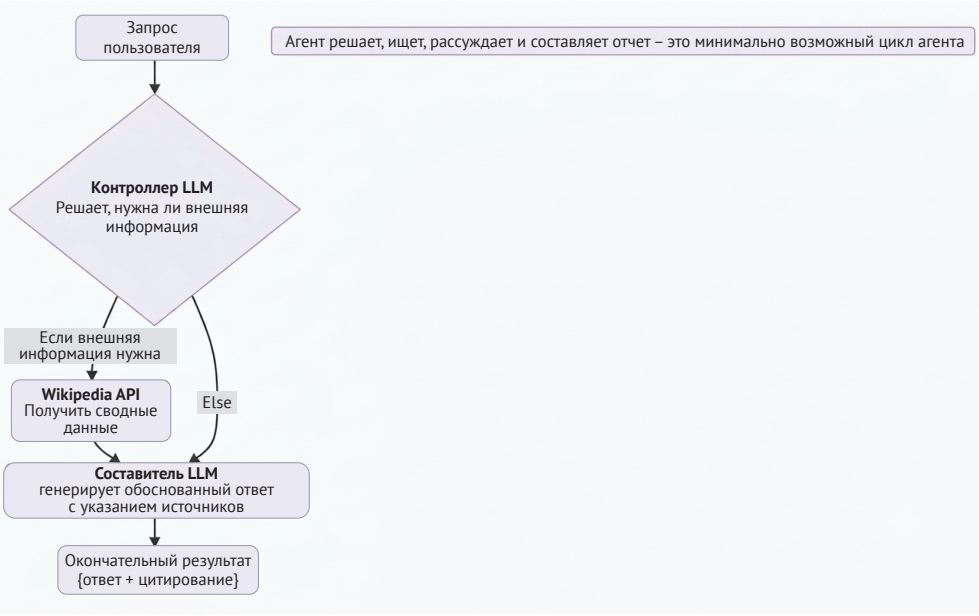
Этот пример демонстрирует цикл агента в действии – систему, которая не просто генерирует текст, но ищет информацию, рассуждает и обосновывает результаты реальными данными.

Цель: превратить LLM из генератора текста в информационного агента, занимающегося поиском истины.

Что делает этот мини-агент

- Принимает вопрос на естественном языке.
- Решает, нужны ли внешние данные.
- Находит релевантную информацию, пользуясь Wikipedia REST API.
- Генерирует краткий ответ с цитатами с помощью GPT.
- Возвращает сам ответ и использованные источники.

Это наименьшая рабочая форма RAG-агента – основной строительный блок для более сложных автономных систем.



Почему этот пример важен

Этот проект демонстрирует ключевой переход: LLM предсказывает – агент выполняет.

Вы увидите, как несколько простых компонентов – уровень принятия решений, внешний инструмент и уровень композиции – превращают модель в самоуправляемую основанную на фактах систему.

Тот же паттерн вертикально масштабируется на архитектуры масштаба предприятия для самостоятельных исследователей, ботов поддержки, информационных ассистентов и аналитиков знаний.

Подготовка: какие инструменты вам понадобятся

Для продолжения установите необходимые пакеты:

```
pip install fastapi uvicorn httpx pydantic python-dotenv
```

Вам также понадобится ключ, совместимый с OpenAI (для GPT). Сохраните его в виде переменной окружения в своем env-файле:

```
OPENAI_API_KEY=sk-xxxxxxxxxxxxxxxxxx
```

Шаг 1 – создание структуры простого проекта

```
mini_qa_agent/
├─ .env
├─ agent.py # основная логика (решение → поиск → составление)
├─ tools.py # клиентский API (Wikipedia)
├─ llm.py # обертка LLM
├─ serve.py # сервер FastAPI
└─ cli.py # исполнитель, запускаемый из командной строки
```

Шаг 2 – обертка LLM (llm.py)

Эта обертка связывает ваш локальный Python-код с GPT (или с любым API, совместимым с OpenAI). Она изолирует логику поставщика, чтобы впоследствии одну модель можно было заменить другой.

```
# llm.py
import os, httpx

def llm_complete(messages, model="gpt-4o-mini"):
    api_key = os.getenv("OPENAI_API_KEY")
    url = "https://api.openai.com/v1/chat/completions"
    headers = {"Authorization": f"Bearer {api_key}"}
    payload = {"model": model, "messages": messages, "temperature": 0.2}

    with httpx.Client(timeout=60) as client:
        r = client.post(url, headers=headers, json=payload)
        r.raise_for_status()
        return r.json()["choices"][0]["message"]["content"].strip()
```

Шаг 3 – Wikipedia API (tools.py)

Внешние «органы чувств» агента – выборка актуальных фактов.

```
# tools.py
import httpx

def wikipedia_summary(title: str):
    safe = title.replace(" ", "_")
    url = f"https://en.wikipedia.org/api/rest_v1/page/summary/{safe}"

    with httpx.Client(timeout=20) as client:
        r = client.get(url, headers={"accept": "application/json"})
```

```
        if r.status_code != 200:
            return None
        j = r.json()
        return {
            "title": j.get("title"),
            "extract": j.get("extract"),
            "url": j.get("content_urls", {}).get("desktop", {}).get("page"),
        }
```

Шаг 4 – логика агента (agent.py)

Это «разум» системы – сочетание контроллера и составителя.

```
# agent.py
import json
from llm import llm_complete
from tools import wikipedia_summary

CONTROLLER_SYSTEM = """Ты контроллер для агента-помощника.
Реши, нужна ли актуальная внешняя информация. Выведи ответ в строгом формате JSON:
{"needs_api": true|false, "query": "search term>"}
"""

COMPOSER_SYSTEM = """Ты точный ИИ-помощник.
- Используй CONTEXT для поиска фактов.
- Включи краткий раздел 'Источники', содержащий URL-адреса.
- Если контекста нет, четко сообщи об этом. Используй не более 150 слов.
"""
def decide_api(question: str):
    messages = [
        {"role": "system", "content": CONTROLLER_SYSTEM},
        {"role": "user", "content": f"Question: {question}"},
    ]
    raw = llm_complete(messages)
    try:
        start, end = raw.find("{"), raw.rfind("}")
        return json.loads(raw[start:end + 1])
    except Exception:
        return {"needs_api": False, "query": question}

def compose_answer(question: str, context: dict | None):
    context_text = (
        f"Title: {context['title']}\n
          Text: {context['extract']}\n
          URL: {context['url']}" if context else "No external context available."
    )
    messages = [
        {"role": "system", "content": COMPOSER_SYSTEM},
        {"role": "user", "content": f"Question:
        {question}\n\nContext:\n{context_text}"},
    ]
    return llm_complete(messages)
```

```
def answer(question: str):
    decision = decide_api(question)
    context = wikipedia_summary(decision["query"]) if
    decision["needs_api"] else None
    final = compose_answer(question, context)
    return {"question": question, "answer": final, "context_used": bool(context)}
```

Шаг 5 – запуск из командной строки

```
# cli.py
import sys, json
from agent import answer

if __name__ == "__main__":
    question = " ".join(sys.argv[1:]) or "Что такое квантовые вычисления?"
    result = answer(question)
    print(json.dumps(result, indent=2, ensure_ascii=False))
```

Попробуем запустить:

```
python cli.py "Кто изобрел телефон?"
```

Вы получите обоснованный ответ вида:

```
{
    "question": "Кто изобрел телефон?",
    "answer": "Александр Грэйам Белл изобрел телефон
    в 1876 году. [Источники:
    https://en.wikipedia.org/wiki/Alexander_Graham_Bell]",
    "context_used": true
}
```

Шаг 6 – факультативный: конечная точка API (FastAPI)

```
# serve.py
from fastapi import FastAPI
from pydantic import BaseModel
from agent import answer

app = FastAPI(title="Mini QA Agent")
class Q(BaseModel):
    question: str
    @app.post("/qa")

    def qa(q: Q):
        return answer(q.question)
```

Запустим сервер:

```
uvicorn serve:app—reload—port 8000
```

Отправим запрос с помощью cURL:

```
curl -X POST http://localhost:8000/qa \
     -H "Content-Type: application/json" \
     -d '{"question": "What is CRISPR?"}'
```

Иллюстрация цикла агента

Даже этот миниатюрный проект демонстрирует все четыре столпа ИИ-агента. Каждый шаг имитирует поведение более продвинутых ИИ-агентов типа LangChain, CrewAI или OpenDevin в части оркестрации сложного многошагового рассуждения, только здесь все просто, наглядно и прозрачно.

Ключевые выводы

Минимально возможная агентная система не сложна – она скоординирована.

Языковая модель сама по себе генерирует текст, но, получив возможность принимать решения, искать информацию и совершать действия, она генерирует достоверные результаты.

Это упражнение – ваш первый шаг к мышлению в терминах агентной архитектуры – проектированию ИИ не в виде единой модели, а в виде набора интеллектуальных взаимодействующих компонентов.

Глава 2

Как «мыслят» LLM: трансформер и не только

2.1. О нейросетевом движке, лежащем в основе современного ИИ

От нейронов к словам: новый вид интеллекта

Когда вы беседуете с такой LLM, как GPT или Claude, создается впечатление, что модель вас понимает, но на самом деле имеет место пиршество математики.

Место возбуждения нейронов в человеческом мозге занимают векторы, перебирающие миллиарды параметров, организованных в виде слоев, которые называются трансформерами.

Каждый слой преобразует текст в смысловое значение, это значение в контекст, а контекст обратно в текст.

LLM понимает язык не так, как мы, – она представляет его геометрически. Эта геометрия, погруженная в многомерное векторное пространство, и есть то, что наделяет современный ИИ удивительной способностью рассуждать, переводить, обобщать и даже воображать.

1. Революция трансформеров

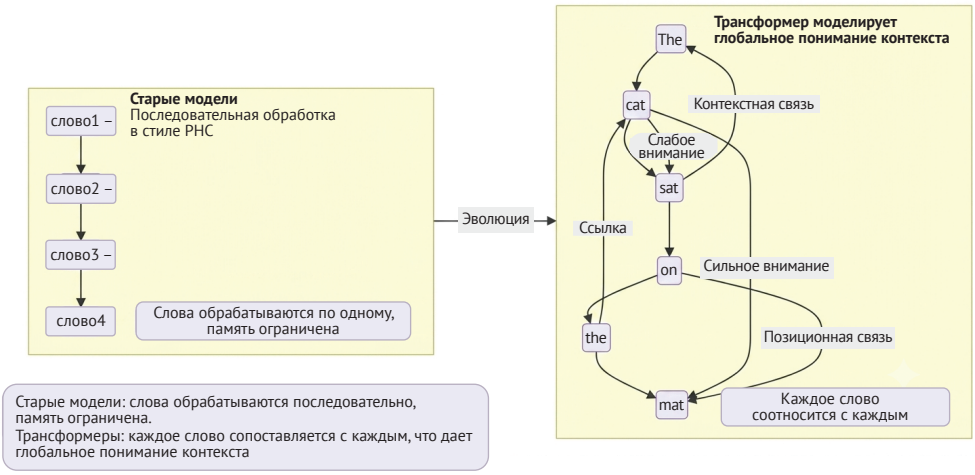
До 2017 года в основе большинства языковых моделей лежали рекуррентные нейронные сети (РНС) и долгая кратковременная память (LSTM). Они обрабатывали один токен за другим; это можно сравнить с чтением книги по одной букве без возможности вернуться назад.

А в 2017 году группа Google Research опубликовала статью под названием «Attention Is All You Need».

Эта фраза стала пророческой.

Трансформерная архитектура изменила все. Вместо того чтобы обрабатывать текст шаг за шагом, трансформер рассматривает все слова сразу и анализирует, как одни слова связаны с другими.

Этот механизм называется самовниманием, он позволяет модели сфокусироваться на тех частях предложения, которые дают наибольший вклад в предсказание продолжения.



2. Как работает механизм внимания (без математической зауми)

Изложим этот вопрос на интуитивном уровне.

Читая предложение, трансформер не обрабатывает его как совокупность букв или звуков, а разбивает на токены (элементарные смысловые единицы).

Каждый токен преобразуется в вектор – список чисел, представляющих смысл этого токена в многомерном пространстве.

Затем механизм внимания задает простой, но чрезвычайно важный вопрос: «Вот есть слово; на какие другие слова в предложении мне следует обратить внимание и сколько именно?» Например:

- в предложении «The cat sat on the mat»[1] слово «sat» обращает наибольшее внимание на слова «cat» и «mat»;
- в предложении «The bank approved the loan»[2] слово «bank» обращает внимание на «loan», а не на «river».

Каждый слой трансформера повторяет этот процесс, в результате чего уточняются связи, абстрагируются смысловые значения и строится когнитивная карта связи слов.

Аналогия: трансформер как фабрика идей

Представьте себе группу аналитиков (слоев модели).

Каждый аналитик читает один и тот же отчет (текст). Они работают не изолированно, а постоянно заглядывают в заметки друг друга, корректируя релевантность. В конечном итоге каждый аналитик сформирует собственное понимание, а кроме того, будет выработано также коллективное мнение.

Это и есть внимание: разделяемая осведомленность внутри модели.

---

[1] Кошка сидела на коврике. – *Прим. перев.*

[2] Банк одобрил заем. Слово «bank» означает также «берег», отсюда и упоминание «river» (река). – *Прим. перев.*

3. Магия токенизации

Прежде чем LLM сможет что-то сделать, она должна увидеть текст. Но компьютеры читают не слова, а числа. Токенизация – это процесс преобразования исходного текста в нечто, понятное модели. Предложение разбивается на токены, которые обычно являются частями слов, а не целыми словами. Например, фраза

Transformers are revolutionary.

могла бы принять вид:

["Transform", "ers", " are", " revolution", "ary", "."]

Каждому токену сопоставляется уникальный числовой идентификатор (что-то вроде индекса слова).

В процессе обучения модель учится предсказывать идентификатор следующего токена по идентификатору предшествующего ему.

Каждый токен является координатой, и предсказание следующей координаты – это и есть суть того, как модель «думает наперед».

Чем меньше и эффективнее токенизатор, тем лучше модель обобщается на разные языки, включая ошибки правописания и даже выдуманные слова.

4. Слои, веса и смысловые значения

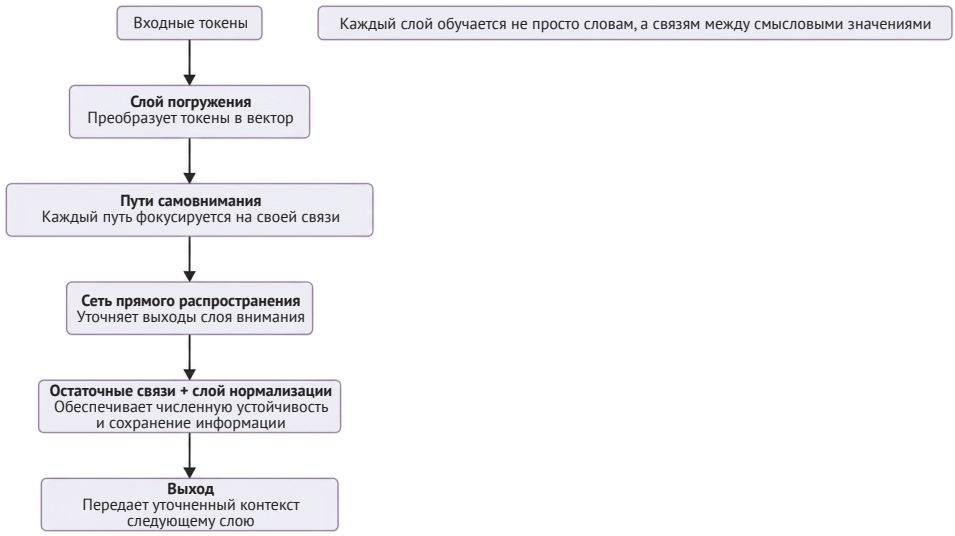
Любая LLM построена из повторяющихся трансформерных блоков, каждый из которых содержит:

- многопутевое самовнимание: ищет связи между токенами;
- слои прямого распространения: уплотняют и уточняют информацию;
- остаточные связи: сохраняют прошлый контекст, так что детали не теряются;
- слои нормализации: обеспечивают численную устойчивость значений.

Будучи составленными в стопку (иногда содержащую сотни слоев), эти слои образуют глубокую структуру: более ранние слои обучаются локальному синтаксису (слова «собака» и «лаять» связаны), а более поздние обучаются абстрактной семантике («радость», «страх», «прибыль»).

Последние слои на основе полученной ранее информации вычисляют вероятности следующих токенов.

Благодаря такой архитектуре мы получаем связные абзацы, неожиданные метафоры и даже целые обобщения исследований – все путем предсказания по одному токену за раз.



Обучение LLM можно описать одним обманчиво простым правилом: предсказать следующий токен, зная все предыдущие.

Собственно, это и все. Но когда это проделывается миллиарды раз с триллионами токенов, возникает нечто удивительное: семантический интеллект.

Модель начинает понимать, что между словами, фразами и концептами имеются связи, и эти связи можно обобщить на данные, которые не предъявлялись на этапе обучения.

Со временем такие паттерны, как:

- грамматика;
- контекст;
- причина и следствие;
- юмор и интонация

... оказываются закодированы в параметрическом пространстве модели.

Именно по этой причине трансформеры отвечают интуиции – они статистически моделируют паттерны рассуждения, свойственные человеку.

6. Почему трансформеры заменяет все, что существовало прежде

Трансформеры предлагают три суперспособности, которых недостает прежним архитектурам.

Благодаря эффективности и гибкости трансформеры стали универсальной архитектурой – они лежат в основе GPT, Claude, Gemini и большинства современных систем ИИ.

7. От трансформера к агенту

Трансформеры представляют не только академический интерес – их освоение критически важно для построения интеллектуальных агентов.

Любая цепочка рассуждений, поисковый запрос или обращение к памяти, которые выполняет агент, основаны на механизмах внимания той модели, на которой он базируется.

«Мыслительный процесс» агента – планирование, действие, осмысление – это всего лишь оркестрация вызовов трансформера в разные моменты времени и в разных контекстах.

Соединяя трансформеры с поиском, осмыслением и внешними инструментами, мы преобразуем статистическое предсказание в ситуационное познание – отличительный признак агентного ИИ.

LLM – мозг. Агенты – разум, который пользуется мозгом.

Трансформер – это нейросетевой движок, лежащий в основе современного искусственного интеллекта.

Он разбивает текст на токены, собирает их в векторы, обучается паттернам с помощью механизма внимания и реконструирует смысл, предсказывая следующее слово.

Понимать, как работает трансформер, не просто полезно – это первый шаг к пониманию того, как думают, планируют и рассуждают агенты.

2.2. Предобучение, базовое дообучение и инструктивное дообучение

Языковые модели обучаются, адаптируются и подстраиваются под цели человека.

Эволюция обучения в больших языковых моделях

Любая интеллектуальная система, будь то человек или машина, проходит три этапа обучения:

- обучение на опыте (полная открытость миру);
- обучение на примерах (ориентация на практические задачи);
- обучение путем обратной связи (подстраивание под ценности и намерения человека).

В случае LLM этим этапам соответствуют предобучение, базовое дообучение и инструктивное дообучение.

Каждый этап углубляет понимание модели – от паттернов исходного языка до целенаправленного поведения.

Рассмотрим их поочередно – на концептуальном, интуитивном и практическом уровне.

1. Предобучение: научение модели понимать мир

Цель: получить общие знания и познакомиться с лингвистической структурой.

Предобучение – это основа, этап, на котором LLM обучается понимать мир с помощью больших объемов неразмеченного текста. Представьте, что на вход модели подается весь интернет, «Википедия», научные статьи, код и книги. На этапе предобучения задача модели формулируется просто и элегантно: предсказать следующий токен, зная все предшествующие.

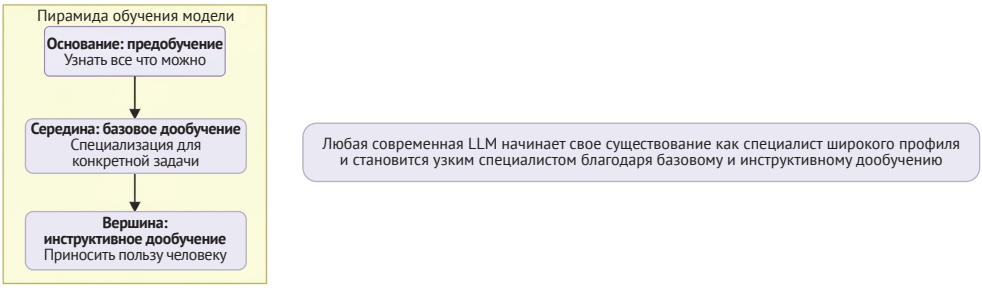
Это – и все: никаких меток, никакого учителя, никаких явных смысловых значений.

Но благодаря триллионам таких предсказаний модель усваивает:

- грамматику и синтаксис;
- факты и ассоциации;
- логические паттерны;
- культурные нормы;
- эвристические приемы решения задач.

Можете считать, что предобучение – это чтение без определенной цели, итогом которого тем не менее является обширная и структурированная база знаний.

Именно это наделяет GPT-4, Claude и Gemini поражающей воображение широтой понимания.



Пример: предобучение на практике

Предобученная модель уже способна:

- завершать предложения: «Столицей Франции является ___»;
- писать связные абзацы;
- генерировать успешно компилируемый код.

Но она также может:

- неправильно отвечать на вопросы типа «Как приготовить воду быстрее?» (галлюцинируя, придумывает метод);
- или отвечать как робот либо небезопасно.

Это объясняется тем, что предобучение дает знания, но не понимание.

Чтобы сделать модели полезными, мы должны научить их использовать свои знания безопасно и в соответствии с контекстом.

2. Базовое дообучение: специализация модели

Цель: адаптировать общую модель к конкретной предметной области или задаче.

После того как модель обрела общие знания, дообучение работает как аспирантура. Оно сфокусировано на знаниях о конкретной проблематике.
Примеры:

- медицинская LLM, дообученная на клинических наблюдениях, → более точные диагнозы;
- юридическая LLM, дообученная на правоприменительной практике, → точный анализ контрактов;
- модель кодирования, дообученная на репозиториях открытого исходного кода, → помощь при программировании.

Базовое дообучение изменяет веса модели – параметры, в которых закодированы обученные связи, – так что она работает лучше на специализированном наборе данных.

Аналогия: от библиотеки к лаборатории

Если в процессе предобучения модель получает знания в объеме библиотеки, то базовое дообучение можно уподобить лабораторной работе – применению теории к практическим примерам.

Предобученная модель знает, что такое клиническое наблюдение.

Дообученная модель понимает, как его безопасно интерпретировать.

Базовое дообучение в реальном мире

Дообучить LLM можно с использованием:

- дообучения с учителем (supervised fine-tuning – SFT): требуется предоставить правильные примеры входа-выхода;
- дообучения для предметной области: требуется предоставить документы или данные, относящиеся к конкретной области;
- дообучения с адаптерами (LoRA, PEFT): модифицировать не всю модель, а лишь небольшое число параметров для повышения эффективности.

Дообучение формирует область компетенций модели, но не ее характер. Она становится мастером в своем деле, но по-прежнему не распознает интонацию, ничего не знает об этических нормах и о намерениях пользователя.

Это задача следующего этапа.

3. Инструктивное дообучение: модель учится слушать

Цель: адаптировать модель под человеческое общение – чтобы она следовала инструкциям естественно и безопасно.

Инструктивное дообучение (иногда называемое подстраиванием) – это то, что делает из мощной LLM полезную.

Этот процесс дообучает модель не на данных предметной области, а на парах инструкция–реакция – примерах того, как люди задают вопросы и каких ответов ожидают.

Благодаря обучению на тысячах таких примеров модель обучается:

- точно следовать намерениям человека;
- использовать полезные и безопасные формулировки;
- соблюдать этические и культурные нормы.

На этом этапе модель обретает черты личности – причина, по которой GPT или Claude кажутся приятными собеседниками, отвечающими потребностям пользователя.

Не только инструкции: RLHF и DPO

Инструктивное дообучение часто сочетается с обучением с подкреплением на основе обратной связи с человеком (Reinforcement Learning from Human Feedback – RLHF) или с прямой оптимизацией предпочтений (Direct Preference Optimization – DPO).

В этих методах оценки, поставленные человеком, используются для вознаграждения за полезное, безопасное или точное поведение.

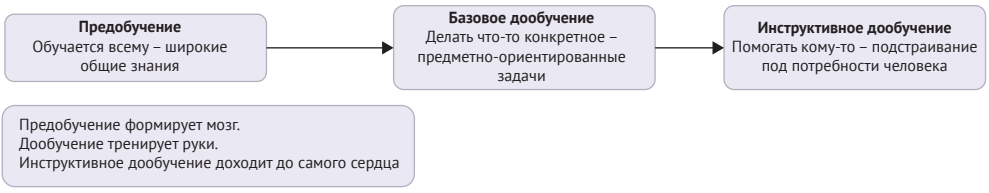
- RLHF: модель генерирует несколько ответов → человек ранжирует их → модель обучается на этих предпочтениях.
- DPO: описанный выше процесс автоматизируется с помощью заранее собранных данных о предпочтениях.

Инструктивное дообучение превращает «может сделать» в «должна сделать». Именно на этом этапе базовый интеллект подстраивается под намерения.

4. Соберем все вместе: от хаоса к упорядоченности

В совокупности эти этапы позволяют создать модель, которая:

- обладает обширными знаниями (благодаря предобучению);
- ориентирована на конкретные навыки (благодаря базовому дообучению);
- подстраивается под пользователя (благодаря инструктивному дообучению).



5. Почему это важно для агентного ИИ

В мире ИИ-агентов понимание этих этапов представляет не только академический интерес – оно определяет стратегию.

- Предобученные модели обладают широкими возможностями.
- Модели, прошедшие базовое дообучение, приобретают способность к специализированным рассуждениям.

- Модели, прошедшие инструктивное дообучение, обеспечивают надежную кооперацию с человеком.

При проектировании ИИ-агентов мы часто комбинируем модели, полученные на различных этапах:

- предобученную основную модель для свободных рассуждений;
- модель, прошедшую базовое дообучение, для достижения точности с учетом фактов;
- оркестратор, прошедший инструктивное дообучение, который умеет естественно взаимодействовать с человеком.

Именно это модульное использование по-разному дообученных моделей и наделяет многоагентные системы мощью и интерпретируемостью.

Предобучение привносит язык. Базовое дообучение привносит опыт. Инструктивное обучение привносит эмпатию. В совокупности они превращают исходный трансформер в помощника, который понимает, действует и подстраивается.

2.3. Ограничения LLM: память, галлюцинации и пробелы в рассуждениях

Даже самые умные модели нуждаются в структуре и средствах поиска информации.

Миф о всеведущей модели

Современные большие языковые модели (LLM) – GPT-4, Claude, Gemini и их родственники с открытым исходным кодом – умеют писать эссе, отлаживать код и даже имитировать диалог лучше, чем многие люди.

Но за этой беглостью скрывается суровая правда: LLM не «знают», они предсказывают.

Каждый ответ является статистически наилучшей гипотезой, а не настоящей гарантией.

Хотя трансформеры умеют моделировать связи между триллионами токенов, их способность к рассуждениям ограничена архитектурой, обучающими данными и отсутствием постоянной памяти.

Чтобы создавать по-настоящему интеллектуальные агенты, мы должны понимать, чего LLM не могут делать и почему эти пробелы имеют значение.

1. Проблема памяти: мышление, лишенное непрерывности

По умолчанию LLM не хранят состояние. Они обрабатывают каждый запрос независимо, будучи ограничены контекстным окном – количеством токенов (слов или частей слов), которые могут «помнить» в каждый момент времени. Например:

- GPT-4 может обрабатывать примерно 128K токенов, что приблизительно равно размеру короткой новеллы;
- Gemini 1.5 заявляет, что помнит 1 млн токенов, но все-таки это скользящее окно внимания, а не настоящая долговременная память.

Когда окно сдвигается, ранее хранившаяся в нем информация теряется. Модель не помнит прошлые диалоги или уроки.

LLM «думает» как склеротик – глубоко, но не помня, что было раньше.

Аналогия: идеальная амнезия

Представьте себе блестящего ученого, который способен мгновенно понять любую статью, но забывает ее в тот момент, как закрывает последнюю страницу.

Это и есть образ LLM без памяти.

Именно из-за этого ограничения ИИ-агенты включают векторную память, графы знаний или внешние базы данных, что наделяет модель «долговременным мозгом», способным хранить и извлекать полученные ранее знания.

2. Проблема галлюцинаций: бегло, но неправильно

LLM – мастера демонстрировать уверенность: они могут сгенерировать текст, который выглядит абсолютно правдоподобно, даже если он полностью сфабрикован. Это явление называется галлюцинацией.

Галлюцинации вызваны следующими причинами:

- LLM обучаются, так чтобы давать вероятные, а не истинные ответы;
- они оптимизируются с целью лингвистической, а не фактической правильности;
- они не имеют доступа к данным в режиме реального времени;
- после обучения их знания замораживаются;
- они склонны к чрезмерному обобщению;
- в случае недостатка информации они интерполируют похожие паттерны – и выдают убедительную бессмыслицу.

Пример: несуществующая цитата

Попросим базовую модель: «Дай мне три последних научных статьи по графеновым батареям не позднее 2024 года».

Она может уверенно выдать такой ответ:

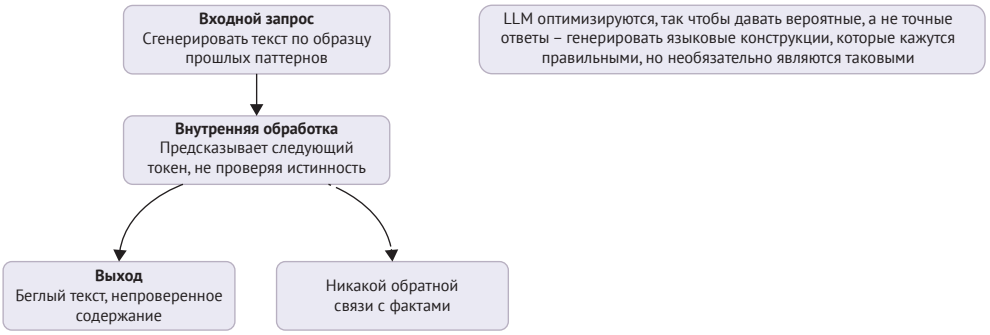
1. Zhang et al., Advanced Graphene Battery Nature Energy, 2024.
2. Li and Kumar, Graphene Ion Transport Science, 2024.
3. Wu et al., High-Density Graphene IEEE, 2024.

Ни одной из этих работ не существует.

Модель просто сконструировала правдоподобные имена, журналы и годы издания исходя из своего прошлого статистического опыта.

На неопытный взгляд, все выглядит правильно.

Но для архитектора системы это синтезированная истина – правдоподобная, но лишенная оснований.



3. Проблема рассуждения: корреляция без причинно-следственной связи

LLM преуспевают в распознавании образов, но испытывают трудности с причинно-следственными рассуждениями – пониманием того, почему некое явление происходит.

Их «мыслительный» процесс управляется корреляцией, а не логикой.

Например, зададим вопрос: «Если у Алисы было два яблока и одно из них она отдала Бобу, сколько яблок у нее осталось?» Модель без труда ответит «одно».

Но давайте включим цепочку рассуждений: «У Алисы было два яблока, одно она отдала Бобу, а Боб отдал одно Кэрол. У кого больше яблок?»

Некоторые сравнительно небольшие или слабые модели спотыкаются, неправильно считают или путают роли. И вот почему:

- трансформеры не способны к явным рассуждениям с символами;
- они не могут «запоминать» промежуточные логические состояния, используя кратковременную память, как человек;
- в основе их понимания лежат лингвистические ассоциации, а не пошаговая дедукция.

Пределы эмерджентного рассуждения

По мере масштабирования (увеличения числа параметров, повышения качества данных) кажется, что модель стала рассуждать лучше, но по большей части это рассуждение все равно остается распознаванием эмерджентных паттернов, а не осознанной логикой.

Этот пробел и является причиной, по которой мы сочетаем LLM с внешними фреймворками рассуждений, например агентами LangChain, запросами к ReAct и графовыми когнитивными архитектурами, – для имитации целенаправленного мышления.

Аналогия: знающий рассказчик

LLM можно уподобить рассказчику, который прочитал все когда-либо написанные книги.

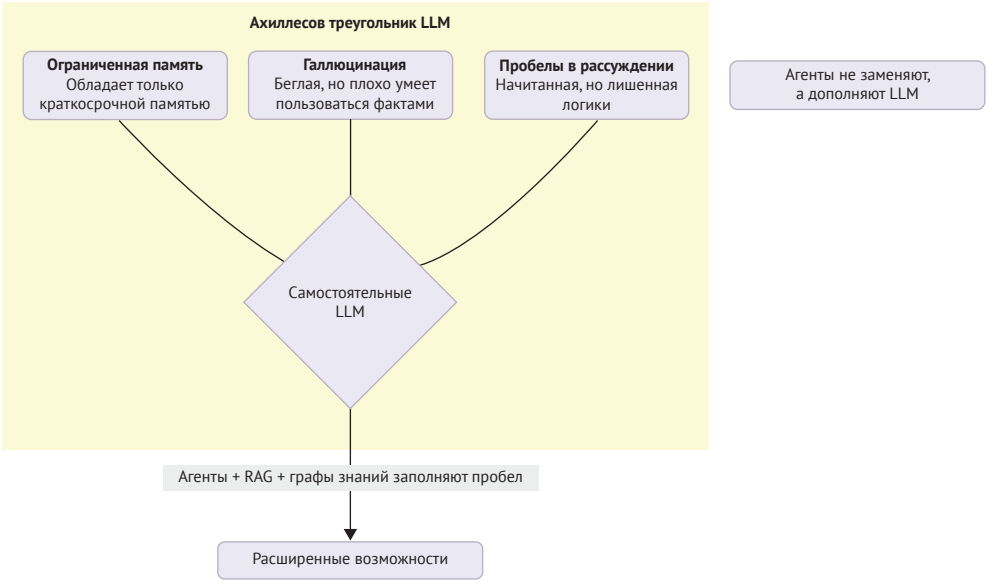
Они могут имитировать любого автора, процитировать любой текст и сочинять потрясающие истории. Но попросите их доказать теорему или построить логистический маршрут, и они уверенно выдадут гипотезу – зачастую оказывающуюся неверной.

Они блестяще справляются с ответами, которые кажутся правильными, но не обязательно являются таковыми.

4. Триада ограничений

Подводя итоги, перечислим три основные слабости LLM: память, галлюцинации и пробелы в рассуждениях. В совокупности они называются «ахиллесовым треугольником» больших моделей.

Эти проблемы не означают, что LLM «дефектны» – несовершенство делает их похожими на людей. Но если люди могут учиться на опыте и осмыслении, то LLM для этого необходима архитектура.



5. Почему агенты являются решением

LLM статичны. Агенты – это динамические системы, компенсирующие их недостатки путем введения структуры.

- Нехватка памяти → решается с помощью векторной памяти и графов знаний.
- Галлюцинации → уменьшаются путем дополнения выборки (RAG).
- Пробелы в рассуждениях → устраняются с помощью циклов осмысления и многоагентного сотрудничества.

Агентная архитектура не альтернатива LLM, а их эволюционное развитие.

Осознание этих ограничений должно не обескураживать, а настраивать на работу. Понимая, чего недостает модели, мы знаем, к чему приложить усилия.

LLM предсказывают, а не воспринимают.

Их ограничения – забывчивость, фальсификация и хрупкая логика – как раз и являются причинами существования агентных систем.

Память дает им контекст, поиск информации – истину, осмысление – мудрость.

2.4. Введение в вызовы функций, погружение и контекстную осведомленность

Современные LLM связывают, представляют и запоминают.

Следующий шаг эволюции: от языка к логике и действию

После того как мы поняли механизм работы трансформера – внимание и предсказание токена, – возникает следующий вопрос: «Как он это делает на практике?»

Ответ дают три столпа современных LLM-систем:

- вызов функций – мост между моделью и внешними инструментами;
- погружения – математический язык смысловых значений и сходства;
- контекстная осведомленность – искусство запоминать и рассуждать.

В совокупности эти столпы наделяют LLM возможностью адаптироваться, а это и составляет суть интеллектуальных ИИ-агентов.

1. Вызов функций: придание LLM рук и инструментов

Первоначально LLM были ограничены порождением обычного текста – вы могли запросить прогноз погоды, но модель могла лишь притворяться, что знает температуру.

Вызов функции меняет дело. Благодаря вызову функций LLM может обратиться к внешнему инструменту или API, когда понимает, что ей нужно получить дополнительную информацию или выполнить вычисление, на которое сама она не способна.

Как работает вызов функции (концептуально)

Обнаружение

Модель интерпретирует ваш запрос и решает, какой инструмент нужен.

Например, пользователь спрашивает: «Какая сейчас температура в Париже?» Модель понимает, что не может ответить и должна вызвать API погодного сервиса.

Генерация

Модель генерирует не текст, а структурированный вызов функции в формате JSON, например:

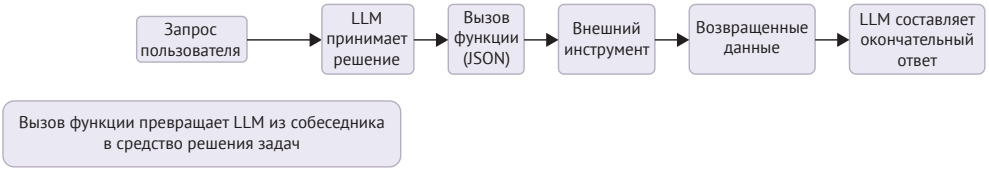
```
{
  "function": "get_weather",
  "arguments": {"city": "Paris"}
}
```

Выполнение

Ваше приложение выполняет эту функцию (вызывает API погодного сервиса).

Возвращенное значение и ответ

Результат (например, `{"temperature": 19, "condition": "Cloudy"}`) подается модели, которая составляет окончательный ответ на естественном языке: «Сейчас в Париже 19 °C и облачно».



Пример: средство прогноза погоды в действии (упрощенно)

```
def
    return {"city": city, "temperature": 19, "condition": "Cloudy"}
    # Внутри приложения:
    response = llm.complete("""
    Ты полезный помощник.
    Если пользователь спросит о погоде, вызови get_weather(city=...).
    """)
    # LLM генерирует структурированный вывод
    {
        "function": "get_weather",
        "arguments": {"city": "Paris"}
    }
```

Этот структурированный вывод является мостом между пониманием языка и реальным выполнением и может считаться одной из самых важных вех в эволюции LLM.

LLM генерируют текст на языке, вызовы функций генерируют его последствия.

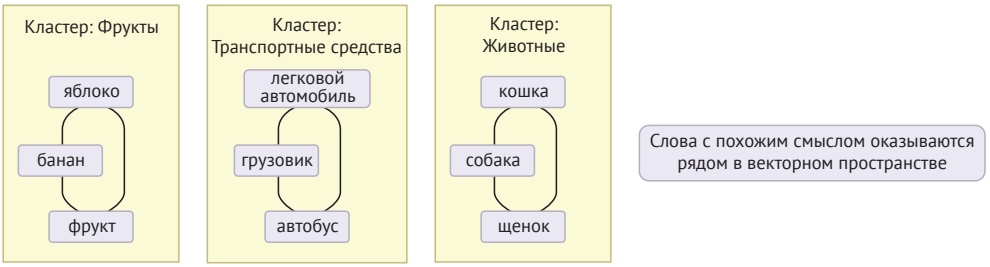
2. Погружения: геометрия смысловых значений

Если вызовы функций придают LLM «руки», то погружения вручают им «карту» – способ понять смысл математически.

Погружение – это вектор чисел, представляющий слово, предложение или документ в многомерном пространстве, где векторы, соответствующие похожим смыслам, близки. Например:

- расстояние между векторами, соответствующими словам «собака» и «щенок», может быть равно 0.12 единиц;
- расстояние между векторами, соответствующими словам «собака» и «кошка», может быть равно 0.85 единиц.

Эта простая идея лежит в основе семантического поиска, RAG (генерации с дополненной выборкой) и обращения к памяти в ИИ-агентах.



Почему погружения важны

Благодаря погружениям LLM становятся осведомлены о контексте, выходящем за пределы самого запроса.

Связав погружения с векторной базой данных (такой как Pinecone, Chroma или Weaviate), вы наделяете своего агента долгосрочной памятью – способностью «вспоминать» и «соотносить».

Пример: мини-демонстрация семантического поиска

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("all-MiniLM-L6-v2")
sentences = ["Кошка сидела на циновке.", "Собака лежала на подстилке.", "Сегодня рынок акций
вырос."]
embeddings = model.encode(sentences)
query = "животное спит на коврике"
query_emb = model.encode(query)

scores = util.cos_sim(query_emb, embeddings)
print(scores)
```

Результат

Максимальная оценка сходства → «Собака лежала на подстилке».

Хотя слова различаются, смысловые значения близки, потому что погружения улавливают семантическое расстояние, а не совпадение слов.

3. Контекстная осведомленность: искусство сохранять связность

Даже при наличии механизмов внимания и погружения интеллект LLM остается ограниченным без контекстной осведомленности – способности сохранять релевантную информацию между взаимодействиями.

Контекстная осведомленность означает, что модель:

- запоминает прошлые повороты в диалоге;
- понимает, что осталось неизменным;
- отдает приоритет недавним или важным деталям;
- избегает противоречий и повторов.

Контекстное окно

Контекстное окно LLM определяет, сколько текста она может «видеть» одновременно. Можете считать, что это кратковременная рабочая память.

В пределах этого окна модель использует механизм самовнимания, чтобы решить, какие токены (слова) релевантны текущему предсказанию.

Когда контекст превышает лимит:

- самые старые токены «вытесняются»;
- модель забывает более ранние детали;
- непрерывность нарушается.

Расширение контекста: роль систем памяти

Чтобы сделать модель контекстно устойчивой, разработчики расширяют контекст следующими способами.

- Векторная память:

  сохраняются прошлые погружения и для каждого запроса извлекается самый релевантный контекст. Применяется в LangChain, LlamaIndex, CrewAI.

- Трансформеры с длинным контекстом:

  применяются в архитектурах типа Gemini или Claude 3, способных эффективно обрабатывать 100K – 1M токенов.

- Иерархическое обобщение:

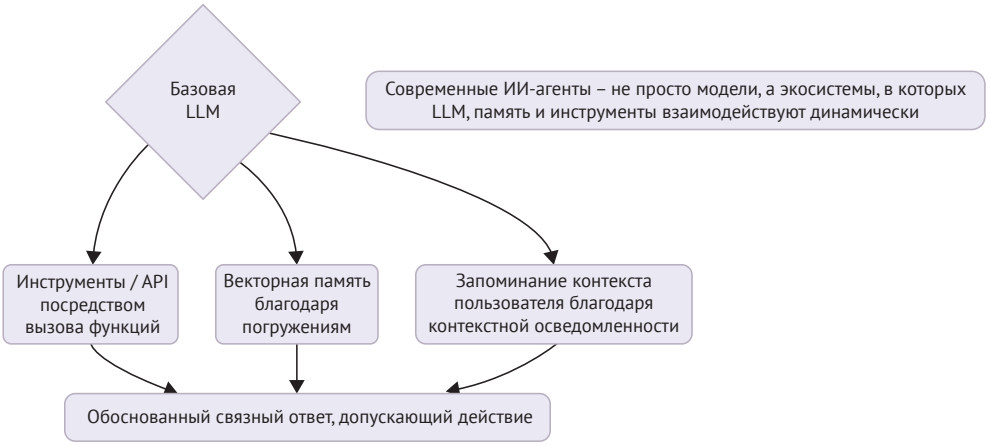
  прошлые контексты обобщаются, чтобы сохранить существенные детали, одновременно сэкономив место. Цель состоит не в том, чтобы помнить все, а в том, чтобы помнить важное.

Аналогия: внимательный собеседник

Представьте себе человека, который не запоминает каждое сказанное вами слово, но помнит суть вашей мысли. Это и есть контекстная осведомленность: баланс между точностью и эффективностью.

Как эти три столпа работают совместно

В сочетании это позволяет перейти от пассивных LLM к активным, контекстно обоснованным ИИ-агентам, которые способны воспринимать, запоминать и выполнять осмысленные действия.



Вызов функций наделяет LLM инструментами. Вложения обеспечивают их памятью. Контекстная осведомленность придает им непрерывность.

Три этих инновации превратили языковые модели в интерактивные рассуждающие системы, способные связывать факты, запоминать и интеллектуально действовать в режиме реального времени.

2.5. От ChatGPT к Gemini и Claude: архитектурные различия

Ведущие современные модели различаются по дизайну, обучению и назначению.

Подготовка сцены: три флагманские модели, одна архитектурная линейка

Хотя модели ChatGPT, Gemini и Claude имеют общее наследие – архитектуру на основе трансформеров, – они различаются по проектным решениям, режимам обучения, приоритетам подстраивания и специализированным возможностям. В этом разделе мы сравним их по трем измерениям: (1) архитектуре и модальности, (2) обучению и подстраиванию, (3) сценариям использования и компромиссам при развертывании. Понимание этих различий поможет вам выбрать правильную базовую модель для своей агентной системы и понять, почему архитектурные «отпечатки пальцев» важны.

1. Различия в архитектуре и модальности

Все три модели построены на базе трансформерных слоев (внимания, прямого распространения, остаточных), но различаются поддержкой модальности, размером контекстного окна, маршрутизацией к экспертам и аппаратными оптимизациями.

ChatGPT (OpenAI)

- Изначально построена на декодере в духе GPT.
- Основное назначение: текст и код. Мультимодальные возможности были добавлены позднее (например, входные изображения в GPT-4).

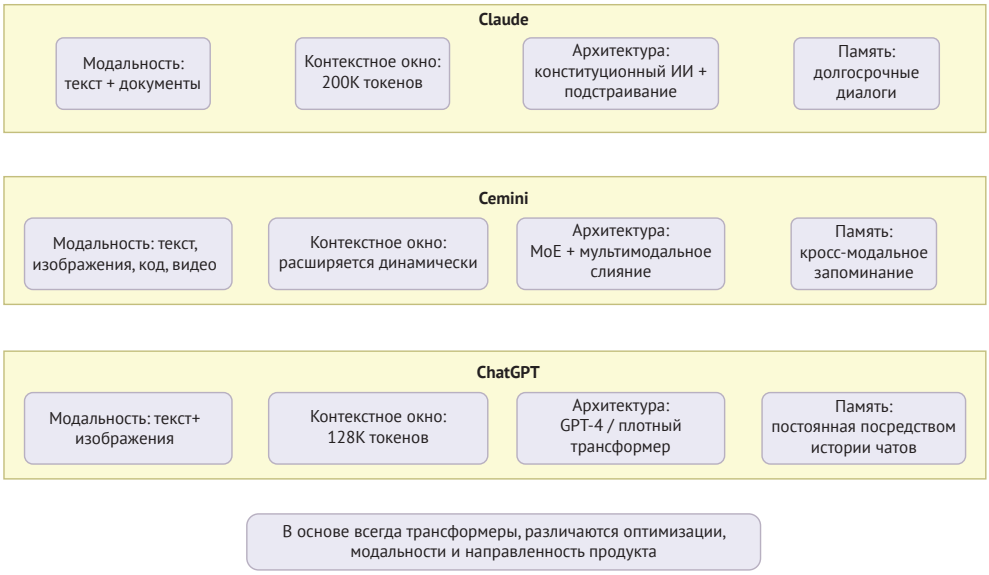
- Контекстные окна увеличены, но в основном ориентированы на текст.
- Упор на универсальность и широкую доступность.

Gemini (Google / DeepMind)

- Изначально создавалась как мультимодальная.
- Принимает и рассуждает о тексте, изображениях, коде, аудио.
- Увеличенные контекстные окна (например, на 1 млн токенов в некоторых вариантах) и аппаратно оптимизированные пути для сервера и клиента.
- Сообщается, что в некоторых версиях используется архитектура смеси экспертов (MoE) (Gemini 1.5) в целях эффективности.

Claude (Anthropic)

- Также основана на трансформерах с упором на безопасность, подстраивание и способность к рассуждениям.
- Много внимания уделено точности диалогов и аналитики, а не только мультимодальности.
- Постепенно интегрируются память, осмысление и более долгосрочный контекст (недавно анонсировано включение «памяти» о прошлом).



2. Обучение, подстраивание и оптимизация

Конвейер обучения и варианты подстраивания обусловливают основные различия в поведении, безопасности и способности к рассуждениям.

Предобучение и дообучение

- ChatGPT: предобучена на большом корпусе текстов и кода, а затем дообучена с учителем и RLHF.
- Gemini: предобучена как истинно мультимодальная модель (текст + изображения + аудио + видео) с намерением обобщения.
- Claude: во время предобучения и подстраивания во главу угла ставилась безопасность (Anthropic отдает предпочтение конституционному ИИ с безопасными умолчаниями).

Подстраивание и возможности

- ChatGPT: часто упоминается широкая доступность, быстрое внедрение инноваций, но отмечаются эпизодические пробелы в глубоких рассуждениях и предметной ориентированности.
- Gemini: сильная сторона – мультимодальные задачи и обширный контекст; новые версии продвигаются как «рассуждающие модели».
- Claude: отличительная особенность – осторожные ответы, сильная аналитика, более безопасное поведение в многопользовательской среде.

Контекстное окно и память

- Варианты Gemini поддерживают очень широкие контекстные окна (до 1M токенов).
- В Claude включена память, в которой сохраняется пользовательский контекст.
- ChatGPT продолжает улучшать контекстные окна и «память чата», но исторически уступает Gemini в плане мультимодальности.

Оптимизация сценариев использования

- ChatGPT: универсальный ассистент «мастер на все руки», умеющий все – от поддержания диалога до помощи при написании кода.
- Gemini: оптимизирована для понимания мультимодального контента, использования агентов и крупномасштабных приложений.
- Claude: оптимизирована для глубокой аналитики, безопасной совместной работы, рассуждений в масштабе предприятия.

3. Сценарии использования, сильные стороны и компромиссы

В одной статье, посвященной сравнению LLM, отмечено: «ChatGPT отлично справляется с разговорными задачами и творческим написанием текстов. Gemini блистает в мультимодальных взаимодействиях и анализе данных в режиме реального времени, а Claude фокусируется на тонких рассуждениях и детальной аналитике».

4. Какое все это имеет отношение к проектированию агентного ИИ

При построении агентов на базе LLM выбор конкретной модели – ChatGPT, Gemini или Claude – имеет значение и определяется не только ценой, но и возможностями.

- Если вам необходим мультимодальный вход и выход, очень широкие контекстные окна или оркестрация в масштабе агента, то обратите внимание на Gemini.
- Если нужны безопасные и подстраиваемые под пользователей рассуждения, управление в масштабе предприятия или проработанные нюансы совместной работы, то лучше предпочесть Claude.
- Если нужна широкая поддержка сообщества, возможность быстро ставить эксперименты или сбалансированный универсальный агент, то ChatGPT по-прежнему послужит крепким фундаментом.

Во многих агентных архитектурах допускается смешивать модели: одну использовать для поиска информации, другую для композиции, а третью для проверки. Понимание особенностей архитектуры и подстраивания позволит вам выстроить гибридные цепочки агентов, в которых каждая модель проявляет свои сильные стороны.

Хотя и ChatGPT, и Gemini, и Claude основаны на трансформерах, они различаются в таких аспектах, как поддержка модальностей, размер контекста, подстраивание обучения и общая направленность продукта. Из-за этого существенно различаются и способы их использования в агентных системах.

Вам подойдет не просто «новейшая» модель, а та, чья архитектура отвечает целям, модальностям и ограничениям на развертывание вашего агента.

2.6. Агент в действии: построение модели с рассуждениями, оптимизированными под запрос с помощью вызовов функций

От теории к практике: объединение рассуждения, поиска информации и действия

До сих пор мы говорили о том, как трансформеры обрабатывают язык, как LLM обучаются путем предобучения и дообучения, а также о назначении погружений, контекстной осведомленности и вызова функций.

Теперь настало время для построения – собрать из всех этих кусочков способный к рассуждениям мини-агент, умеющий думать и действовать.

Следующий далее пример демонстрирует, как:

- использовать подпорки в виде рассуждений на уровне запроса (паттерн «цепочка мыслей»);
- включать вызовы функций для выполнения действий в реальном мире;

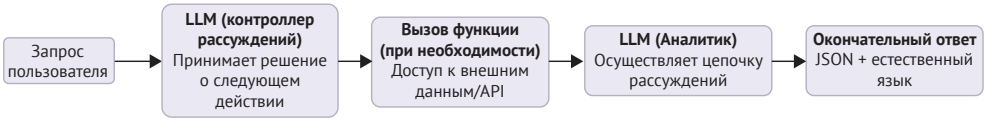
- использовать погружения для обращений к прошлому контексту;
- возвращать структурированный, допускающий проверку ответ, а не простой текст.

Здесь LLM перестает болтать и начинает вычислять.

1. Обзор архитектуры

Цель: построить простой «аналитически рассуждающий помощник», который умеет:

- интерпретировать запрос на естественном языке;
- решать, необходим ли поиск внешних данных;
- искать данные с помощью вызовов функций;
- рассуждать с применением оптимизации запроса;
- возвращать окончательный структурированный ответ.



2. Реализация

Шаг 1 – определяем свои функции

```
# tools.py
import httpx

def get_crypto_price(symbol="BTC"):
    """Получить текущую цену в криптовалюте (демонстрационный API)."""
    url = f"https://api.coindesk.com/v1/bpi/currentprice/{symbol}.json"
    with httpx.Client(timeout=15) as client:
        r = client.get(url)
        if r.status_code != 200:
            return {"error": "Failed to fetch price"}
        data = r.json()
        return {
            "currency": symbol,
            "price_usd": data["bpi"]["USD"]["rate"],
            "time": data["time"]["updated"]
        }
```

Шаг 2 – шаблоны запросов для рассуждений

```
# prompts.py
REASONING_SYSTEM = """Ты помощник, умеющий рассуждать аналитически.
Продумай шаги, прежде чем ответить. Пользуйся доступными функциями
при необходимости.
```

```
После рассуждения выдай окончательный JSON-ответ: {"reasoning": "...",
"result": "..."}.
Рассуждение должно быть кратким, но логичным.
"""
```

Шаг 3 – логика агента

```
# agent_reasoner.py
import json
from llm import llm_complete
from tools import get_crypto_price
from prompts import REASONING_SYSTEM

def reasoning_agent(question: str):
    # Шаг 1: спросить у модели, нужны ли внешние данные
    decision_prompt = f"""
    Question: {question}
    Если внешняя информация (например, текущая цена или данные) нужна,
    ответь в формате JSON:
    {{"needs_function": true|false, "function": "get_crypto_price", "arg":""}}.
    """
    decision = llm_complete([{"role": "user", "content": decision_prompt}])
    try:
        d = json.loads(decision)
    except:
        d = {"needs_function": False}

    # Шаг 2: если нужна, запросить ее
    context = None
    if d.get("needs_function"):
        func = d.get("function")
        arg = d.get("arg", "BTC")
    if func == "get_crypto_price":
        context = get_crypto_price(arg)
    # Шаг 3: выполнить рассуждение
    reasoning_prompt = f"""
        {REASONING_SYSTEM}
        QUESTION: {question}
        CONTEXT: {json.dumps(context, indent=2)}
    """
    answer = llm_complete([
        {"role": "system", "content": REASONING_SYSTEM},
        {"role": "user", "content": reasoning_prompt}
    ])
    return answer
```

Шаг 4 – выполнить и ознакомиться с результатом

```
python -i
    >>> from agent_reasoner import reasoning_agent
    >>> reasoning_agent("Следует ли мне сегодня инвестировать в биткойн?")
```

Пример вывода:

```
{
    "reasoning": "Сейчас стоимость биткойна равна приблизительно $68000.
    Цены быстро меняются; краткосрочные решения должны учитывать риск."
    "result": "Рынок остается высоким – проявляйте осторожность, не
    совершайте импульсивные покупки."
}
```

3. Оптимизация запроса: направления мыслительного процесса модели в нужную сторону

Под оптимизацией запроса понимается структурирование когнитивного потока.

Простая техника подпорок типа цепочки мыслей (Chain-of-Thought – CoT) повышает точность рассуждения на 20–40 %.

Пример шаблона:

- шаг 1: понять вопрос;
- шаг 2: найти необходимые факты (с помощью функции или в памяти);
- шаг 3: логически проанализировать следствия;
- шаг 4: дать четкое заключение и краткое описание рассуждения.

Встраивание такого пути рассуждения непосредственно в обработку запроса системой способствует детерминированному рассуждению и уменьшению галлюцинаций.

Оптимизация запроса требует проектирования когнитивной архитектуры, а не только техники подсказок.

5. Расширение модели

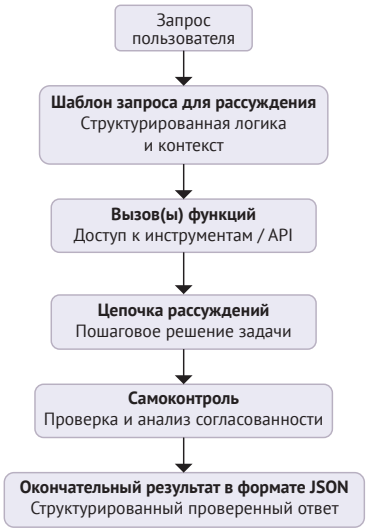
Этот мини-агент можно довести до уровня системы рассуждений производственного качества, если добавить:

- погружения + векторную память: для семантического запоминания истории взаимодействия с пользователем или предметных документов;
- цикл осмысления: попросить модель проанализировать собственные рассуждения и повторить в случае обнаружения дефектов;
- многофункциональный реестр: подключаться к нескольким API (финансы, новости, научные исследования);
- совместную работу моделей: использовать одну модель для рассуждений (Claude), а другую для написания текстов (ChatGPT).

Каждое добавление расширяет когнитивный диапазон агента – от реактивной вопросно-ответной системы до стратегического рассуждения с широким контекстом.

6. Тестирование и оценка

Рассуждающий агент с хорошо оптимизированными запросами должен порождать выход, допускающий трассировку и интроспекцию и основанный на фактах.



Главный урок

Рассуждения не магия, это архитектура.

Комбинируя шаблоны запросов, вызовы функций и контекстную обоснованность, мы превращаем языковую модель в рассуждающую систему, способную выдавать допускающие проверку ответы, ориентированные на конечный результат.

Секрет не просто в увеличении числа параметров, а в более качественном дизайне.

Глава 3

RAG: основа правдивых агентов

3.1. Генерация с дополненной выборкой как средство обоснования фактами

RAG имеет значение – война с галлюцинациями. От уверенной догадки к обоснованному знанию

В мире больших языковых моделей беглость обходится дешево, истина – нет.

Модель типа GPT-4 или Claude может объяснить квантовую физику, составить реферат научной статьи или сгенерировать юридические аргументы, но без доступа к допускающим проверку данным она также может изобретать факты, сохраняя при этом иллюзию полной уверенности.

Это расхождение между красноречием и доказательствами является важнейшей проблемой современного ИИ. LLM предсказывают текст на каком-то языке, RAG-системы извлекают знания.

Именно этот архитектурный сдвиг – от замкнутого предсказания к открытому поиску информации – превратил генеративный ИИ в надежные когнитивные системы, способные решать реальные задачи.

И снова о проблеме галлюцинаций

В главе 2 мы видели, что языковые модели научаются генерировать правдоподобный, а не правдивый текст.

Заложенная в природе LLM тяга к предсказаниям приводит к галлюцинациям – беглым, но сфальсифицированным ответам.

Примеры:

- ссылки на несуществующие работы («Smith et al., Nature 2024»);
- упоминание несуществующих API или Python-библиотек;
- противоречивые числовые результаты.

Для поддержания беседы на общие темы это, возможно, и безвредно. Но в таких областях, как медицина, финансы или юриспруденция, галлюцинации неприемлемы.

Поэтому разработчики начинают задаваться вопросом: «Что, если бы модель могла проверить факты, прежде чем выдавать ответ?»

Добро пожаловать в мир генерации с дополненной выборкой (RAG) – архитектурный ответ на галлюцинации.

1. Что такое RAG?

Генерация с дополненной выборкой состоит из двух движков:

- поисковика – ищет релевантную информацию во внешних источниках (документах, базах данных, API, векторных хранилищах);
- генератора – LLM, которая использует найденный контекст для составления обоснованного ответа в понятной человеку форме.

Считайте RAG моделью, в которую встроен Google.

Вместо того чтобы полагаться исключительно на то, что она «запомнила» в ходе предобучения, модель может извлекать актуальные фактические данные и рассуждать на их основе, что значительно повышает точность и прозрачность.

2. Базовая архитектура

Современный конвейер RAG состоит из четырех функциональных стадий:

- разбиения документа на фрагменты;
- преобразования фрагментов в погружения;
- индексации и хранения погружений;
- извлечения – поиска релевантных фрагментов.

Каждую стадию можно настроить, расширить или заменить, что делает RAG одной из самых гибких и модульных архитектур при проектировании ИИ-систем.

3. Почему RAG работает

LLM по природе своей вероятностные системы, они понимают связи, а не факты.

RAG является дополнением в том смысле, что предоставляет явную фактическую информацию на этапе генерации.

Преимущества:

- обоснованность фактами: ответы ссылаются на источники реальных данных;
- динамические знания: поиск гарантирует поддержание актуальности модели без переобучения;
- больше «мозгов» при меньшем размере модели: компактная LLM + хороший поисковик могут превзойти большие автономные модели по качеству;
- объяснимость: вы можете проследить, какие документы повлияли на ответ.

Пример: от галлюцинаций к обоснованности

Без RAG: «iPhone 17 был выпущен в апреле 2024» (галлюцинация: модели iPhone 17 не существует).

С RAG:

- поисковик находит пресс-релизы Apple за 2023–2025 годы;
- генератор читает их и формулирует ответ: «По состоянию на 2025 год Apple не объявляла о выпуске iPhone 17; последней была выпущена модель iPhone 15 Pro (в сентябре 2023 года). [Источник: Apple Newsroom]».

Результат: обоснованный, прослеживаемый и проверяемый.

4. Как работает поиск информации

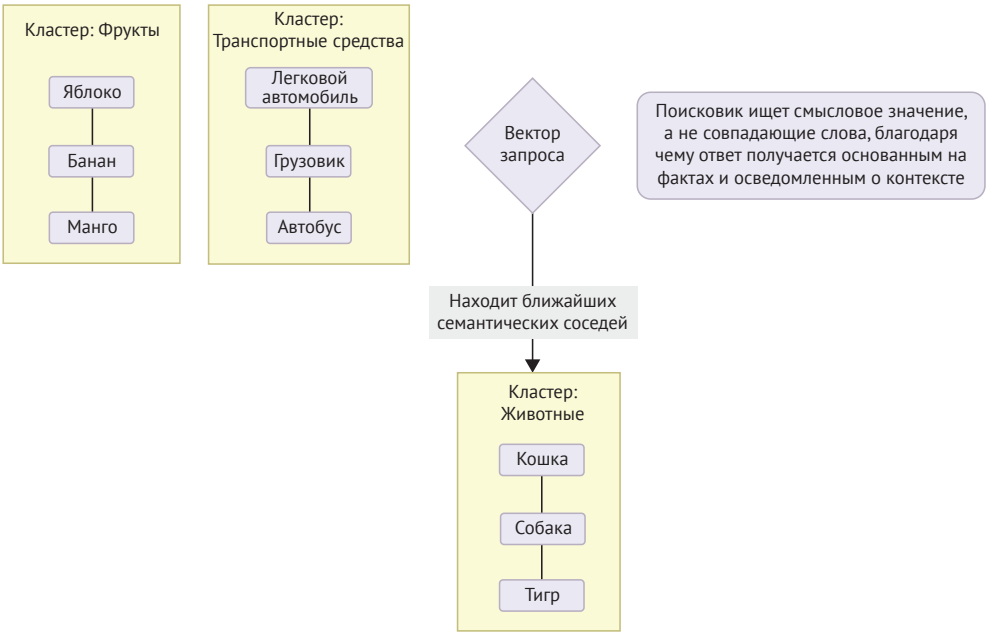
Поиск начинается с преобразования вашей базы знаний (документов, PDF-файлов, статей и т. д.) в погружения – числовые векторы, кодирующие смысловое значение.

Когда пользователь задает вопрос, система:

- преобразует вопрос в вектор погружений;
- ищет близкие векторы в базе данных;
- возвращает первые *k* наиболее похожих фрагментов.

Затем LLM читает эти фрагменты как контекст.

RAG не заменяет мозг LLM, а расширяет ее память.



5. Почему RAG важна для агентов

В агенте RAG выступает в качестве гаранта истинности. Благодаря ей любое рассуждение или принятие решения основывается на проверенных данных, а не просто на памяти модели.

Агент без RAG подобен адвокату без судебной практики – красноречив, но ненадежен.

6. Борьба с галлюцинациями

RAG не устраняет галлюцинации полностью, но если поиск информации произведен качественно, то их частота и серьезность существенно снижаются. Это достигается следующими способами:

- обоснование генерации внешними истинными фактами;
- ограничение контекста релевантной информацией;

- предоставление явных доказательств (ссылки, идентификаторы документов);
- ретроспективная валидация – автоматическая перекрестная проверка фактов.

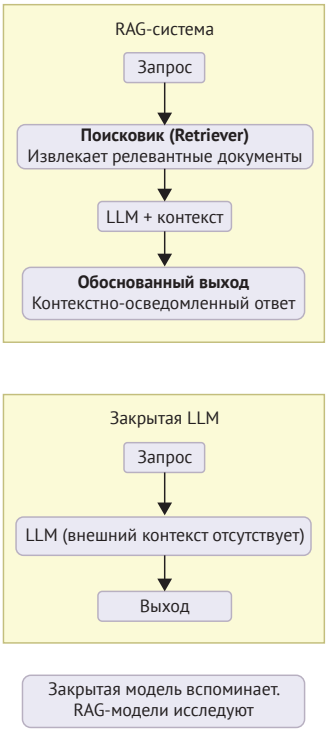
Именно вследствие такой встроенной в архитектуру обоснованности практически все ИИ-системы производственного уровня – от ChatGPT Browse компании OpenAI до Claude Pro Research Mode компании Anthropic – ныне включают какую-то форму RAG.

RAG относится к обоснованности фактами так же, как внимание относилось к контексту, – смена парадигмы, а не заплата.

7. Основные проблемы в RAG-системах

Но RAG несет с собой и новые инженерные проблемы.

Их решение как раз и отличает игрушечные чат-боты от ИИ-агентов производственного уровня.



RAG превращает предсказание в точный ответ. Она делает модели не умнее, а точнее.

В борьбе с галлюцинациями RAG является опорой истинности, так как обосновывает генерацию фактами, контекстами и рассуждением, не полагаясь на случайность.

3.2. Архитектура RAG-систем: поисковик, ранжировщик, генератор

Движок обоснованного интеллекта

Если рассматривать большую языковую модель (LLM) как мозг ИИ-системы, то архитектура RAG будет нервной цепью, связывающей мозг с внешним миром.

Она находит факты, фильтрует их по релевантности и использует для генерации обоснованных и допускающих проверку ответов:

RAG = Поиск + Ранжирование + Генерация.

Каждой части этого равенства соответствует конкретный архитектурный уровень.

- Поисковик → ищет релевантные данные.
- Ранжировщик → фильтрует данные и назначает им приоритеты.
- Генератор → синтезирует обоснованный фактами ответ в понятной человеку форме.

В совокупности они образуют петлю обратной связи, ведущую к истине.

1. Поисковик: ищет нужные знания

Поисковик – это поисковая система, входящая в состав конвейера RAG. Его задача – найти фрагменты текста (или данных), наиболее релевантные запросу пользователя, но не по ключевым словам, а по смыслу.

Как это работает

Вопрос пользователя преобразуется в векторное погружение (числовое представление смыслового значения).

Система выполняет поиск семантически схожих погружений в векторной базе данных.

В качестве потенциального контекста возвращаются k наиболее похожих фрагментов (например, 3–5).

Этот процесс заменяет технологию поиска на основе «мешка слов» (например, TF-IDF) семантическим поиском, тем самым наделяя его контекстуальным интеллектом.

Разбиение на фрагменты и создание погружений: основа хорошего поиска

Качество поисковика зависит от разбиения документов на фрагменты:

- слишком много фрагментов → нерелевантный шум;
- слишком мало фрагментов → непригодный контекст.

Рекомендация: включать в один фрагмент 400–800 токенов при 10–20-процентном перекрытии.

Это обеспечивает плавные переходы между абзацами и улучшает семантическую непрерывность.

2. Ранжировщик: выбор наиболее релевантных фактов

Отобрав первые *k* найденных фрагментов, вы должны решить, какие из них действительно важны. Тут-то и приходит на помощь ранжировщик.

Ранжировщик назначает приоритеты по сходству, но похожий ≠ полезный. Ранжировщик выполняет дополнительный проход, чтобы оценить релевантность и качество в контексте вопроса.

Почему ранжирование важно

Допустим, вы задаете вопрос: «Каковы основные причины инфляции в 2025 году?» Поисковик мог бы вернуть такие источники:

- новости о ценах на нефть;
- статью в «Википедии» об инфляции;
- пост в блоге о волатильности криптовалют.

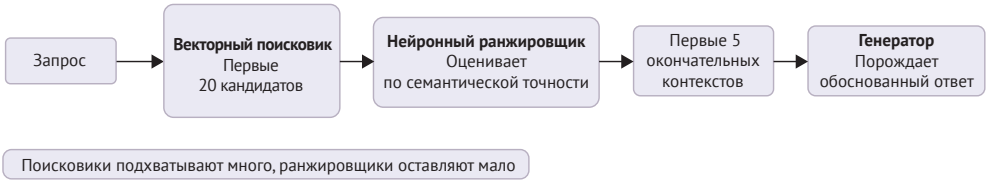
Все верны, но не все релевантны.

Ранжировщик отфильтровывает шум, гарантируя, что генератор увидит только наиболее контекстуально полезную информацию.

Подходы к ранжированию

В большинстве современных конвейеров RAG наилучшее соотношение между точностью и скоростью дает гибридное ранжирование.

Применить быстрый поиск → ограничиться 20 фрагментами → переранжировать первые 5 с помощью нейросетевой модели.



3. Генератор: преобразует контекст в связные ответы

Генератор – это креативная часть системы – LLM, которая принимает структурированный контекст от поисковика и преобразует его в связный, понятный человеку ответ.

Он не «знает», откуда поступила информация, – вы должны сказать ему об этом. Тут в игру вступает дизайн запроса.

Пример шаблона запроса

```
СИСТЕМА
Ты ассистент по фактам. Используй предоставленный КОНТЕКСТ для точного
ответа. Если ответ не найден в КОНТЕКСТЕ, скажи «Недостаточно информации».

ВОПРОС ПОЛЬЗОВАТЕЛЯ
{{ question }}
```

КОНТЕКСТ
{{ top_passages }}

ФОРМАТ
Дай ясный и краткий ответ, указав источники.

Этот шаблон достигает трех целей:

- ограничивает воображение модели – она не может галлюцинировать за пределами контекста;
- поощряет явное обоснование – «используй предоставленный КОНТЕКСТ»;
- поддерживает прослеживаемость – источники могут быть процитированы или присоединены.

Шаг генерации

На этом этапе LLM объединяет:

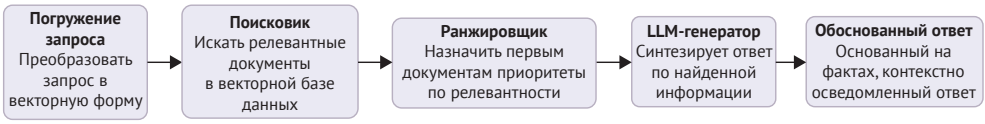
- намерение пользователя (вопрос);
- найденный контент (контекст);
- инструкции в запросе (ограничения).

На выходе получается контекстно осведомленный, допускающий проверку ответ, который можно подвергнуть аудиту и воспроизвести.

Оценка: обоснованная и необоснованная генерация

Если генератор обоснован качественным поиском, то он перестает гадать и начинает рассуждать.

4. Соберем все вместе: полный цикл RAG



Конвейер RAG превращает LLM из рассказчиков в исследователей

5. Дополнение конвейера RAG

После освоения трех базовых уровней возможности RAG можно расширить с помощью дополнительных компонентов.

- Переформулирование и декомпозиция запросов.
- Гибридный поиск (ключевые слова + векторное извлечение).
- Фильтрация по метаданным.
- Модели переранжирования.
- Многошаговое извлечение.
- Использование инструментов и вызов API.
- Сжатие длинного контекста.

- Системы памяти и история диалога.
- Графы знаний.
- Циклы рефлексии и самокоррекции.
- Фреймворки агентной оркестрации.
- Кеширование ответов.
- Генерация цитат.
- Контроль доступа и фильтрация безопасности.
- Непрерывные конвейеры оценки качества.

Эти методы превращают простую конфигурацию RAG в корпоративную архитектуру знаний – масштабируемую, объяснимую и эффективную.

6. Типичные точки отказа

Даже хорошо спроектированные системы RAG могут работать плохо, если их компоненты не согласованы между собой.

- Слабые эмбеддинги ухудшают качество семантического поиска.
- Некачественное разбиение документов нарушает целостность контекста.
- Нерелевантное извлечение загрязняет контекст промпта.
- Избыточный контекст перегружает модель и размывает внимание.
- Модели ранжирования могут отдавать приоритет популярности вместо релевантности.
- Многоступенчатые конвейеры увеличивают задержки.
- Инструкции промпта могут конфликтовать с извлеченными данными.
- Устаревшие базы знаний приводят к устаревшим ответам.
- Галлюцинации возможны даже при неполных доказательствах.
- Системы цитирования могут ссылаться на неверные фрагменты.
- Слишком агрессивная фильтрация скрывает полезную информацию.
- Метрики оценки не всегда отражают реальное удовлетворение пользователей.
- Пробелы в безопасности могут привести к утечке внутренних данных.
- Даже у больших моделей окно контекста остается ограниченным.
- Стоимость быстро растет при длинных промптах и больших объемах извлеченных данных.

RAG-система будет надежной, если полнота, точность и время задержки сбалансированы – извечный треугольник информационного поиска.

7. Итоги: троица RAG

Системы RAG строятся вокруг трех фундаментальных компонентов:

- поисковик находит релевантные знания;
- ранжировщик отбирает наиболее полезные данные;
- генератор преобразует факты в связный ответ.

Вместе они образуют конвейер интеллекта, в котором факты собираются, фильтруются и преобразуются в осмысленное изложение.

Поисковик находит. Ранжировщик уточняет. Генератор рассуждает.

RAG – это не просто информационный поиск, а создание структурированного интеллекта.

Поисковик снабжает модель фактами, ранжировщик заботится об их релевантности, а генератор преобразует знания в текст на естественном языке.

Эта троица определяет современный стандарт правдивых, допускающих проверку и интеллектуальных ИИ-агентов.

3.3. Векторные базы данных: FAISS, Pinecone, Chroma и Weaviate

Уровень памяти интеллектуальных агентов. Скрытое ядро RAG: хранение смысловых значений, а не слов

В глубине любой системы генерации с дополненной выборкой лежит один важный вопрос: «Где находятся знания?»

Отвечаем – не в самой языковой модели. Знания хранятся в векторной базе данных – системе, которая предназначена для хранения, организации и извлечения смысловых значений, а не просто слов.

Векторная база данных служит основой памяти ИИ-агента. В ней хранятся погружения – математические представления текста, кода и других данных, позволяющие системе эффективно выполнять поиск и извлечение семантически схожей информации.

Короче говоря:

- языковые модели понимают язык;
- векторные базы данных запоминают его.

1. Что такое векторная база данных?

В векторной базе данных хранятся многомерные векторы, каждый из которых представляет собой фрагмент информации (например, абзац, предложение или изображение).

Когда пользователь предъявляет запрос, система выполняет три шага:

- преобразует запрос в векторное погружение, применяя модель погружений;
- ищет в базе данных векторы, наиболее похожие на запрос;
- возвращает языковой модели фрагменты с наивысшим рангом для рассуждения и генерации ответа.

Этот процесс обеспечивает семантический поиск – поиск по смысловому значению, а не по ключевым словам.

2. Почему векторные базы данных важны

Традиционные базы данных или поисковые системы, основанные на ключевых словах (например, ElasticSearch), опираются на лексические соответ-

ствия. Они работают быстро, но часто не позволяют уловить концептуальное сходство.

С другой стороны, векторные базы данных позволяют выполнять поиск по семантической близости и таким образом находить релевантный контекст, даже когда слова различаются.

Векторная база данных преобразует статический текст в допускающее поиск смысловое значение – важнейшее свойство для современных конвейеров RAG.

3. Основные компоненты векторной базы данных

Любая векторная база данных обычно включает или использует следующие основные компоненты:

- модель эмбеддингов (embeddings model) – преобразует данные (текст, изображения, аудио) в векторы фиксированной размерности;
- векторный индекс (indexing structure) – специализированная структура данных для быстрого поиска ближайших векторов (например, HNSW, IVF, PQ, Annoy);
- метрику сходства (similarity metric) – способ вычисления близости между векторами (cosine similarity, dot product, Euclidean distance);
- механизм поиска ближайших соседей (ANN/KNN search engine) – алгоритм поиска наиболее похожих векторов.

В совокупности эти компоненты образуют уровень семантического поиска для RAG-системы.

4. Наиболее популярные векторные базы данных

Рассмотрим четыре наиболее распространенные векторные базы данных в экосистеме ИИ: FAISS, Pinecone, Chroma и Weaviate. У каждой из них своя философия дизайна, сильные стороны и идеальные сценарии использования.

A. FAISS (Facebook AI Similarity Search)

- Тип: библиотека с открытым исходным кодом (с интерфейсами к C++ и Python).
- Оптимальна для: исследований, экспериментов и развертывания на территории предприятия.
- Основные плюсы: высокая производительность, настраиваемая индексация, возможность ускорения с помощью GPU.

Особенности

- Разработана Meta AI Research.
- Поддерживает миллиарды векторов.
- Реализует эффективные алгоритмы приближенного поиска ближайшего соседа, такие как IVF и HNSW.
- Развертывается на собственной инфраструктуре, что обеспечивает контроль и прозрачность.

Пример сценария использования

Исследовательская группа создает движок семантического поиска по миллионам рефератов научных статей.

Пример кода

```
import faiss
import numpy as np
index = faiss.IndexFlatL2(1536)
index.add(np.array(embeddings)) # добавить векторы документов
D, I = index.search(np.array([query_vector]), k=5)
```

Резюме

FAISS предлагает скорость, гибкость и средства управления, делающие ее идеальной для экспериментов и сред, в которых конфиденциальности уделяется особое внимание.

B. Pinecone

- Тип: управляемая облачная служба.
- Оптимальна для: ИИ-приложений масштаба предприятия производственного уровня.
- Основные плюсы: высокая доступность, поиск векторов в режиме реального времени, автоматическое масштабирование.

Особенности

- Полностью управляемая инфраструктура со встроенной фильтрацией по метаданным.
- Органичная интеграция с LangChain, OpenAI и LlamaIndex.
- Поддерживает миллиарды векторов с малой задержкой.
- Поддерживает развертывание по регионам и управление версиями.

Пример сценария использования

RAG-система уровня предприятия, лежащая в основе финансового помощника-исследователя, способного опрашивать миллионы документов в режиме реального времени.

Пример кода

```
from pinecone import Pinecone
pc = Pinecone(api_key="your-key")
index = pc.Index("company-insights")
results = index.query(vector=query_emb, top_k=5, include_metadata=True)
```

Резюме

Pinecone – полностью управляемое, готовое к эксплуатации решение с упором на производительность и надежность.

C. Chroma

- Тип: облегченное локальное векторное хранилище с открытым исходным кодом.
- Оптимальна для: создания прототипов, экспериментов или небольших приложений.
- Основные плюсы: простота и отсутствие внешних зависимостей.

Особенности

- Хранилище по умолчанию для LangChain и LlamaIndex.
- Работает локально с факультативным постоянным запоминанием.
- Простой интерфейс к Python; требуется лишь минимальная настройка.

Пример сценария использования

Локальный RAG-агент, который индексирует научные статьи или внутреннюю документацию проекта для персонального использования.

Пример кода

```
from langchain_community.vectorstores import Chroma
db = Chroma(persist_directory="./chroma_store")
docs = db.similarity_search("как RAG повышает точность?", k=3)
```

Резюме

Chroma идеальна для разработчиков, которые создают прототипы или исполняют технологические процессы с применением RAG на локальной машине.

D. Weaviate

- Тип: управляемая гибридная платформа с открытым исходным кодом.
- Оптимальна для: гибридных сематических и символических систем знаний.
- Основные плюсы: сочетает поиск векторов со структурными графоподобными связями.

Особенности

- Предлагает API на основе REST и GraphQL.
- Встроенная генерация погружений и гибридный поиск по ключевым словам и смысловым значениям.
- Поддерживает текст, изображения и другие модальности.
- Расширяется с помощью плагинов и масштабируется в облаке.

Пример сценария использования

Корпоративный граф знаний, объединяющий в одной семантической системе документацию по продуктам, заявки в службу поддержки и FAQ.

Пример запроса (GraphQL):

```
{
  Get {
    Article(
      nearText: { concepts: ["retrieval augmented generation"], certainty: 0.8 }
      limit: 3
    ) {
      title
      summary
    }
  }
}
```

Резюме

Weaviate устраняет разрыв между семантическим поиском и реляционными структурами знаний и таким образом может быть положена в основу гибридных интеллектуальных систем.

5. Выбор подходящей векторной базы данных

На какой векторной базе данных остановиться, зависит от масштаба, инфраструктуры и целей проекта.

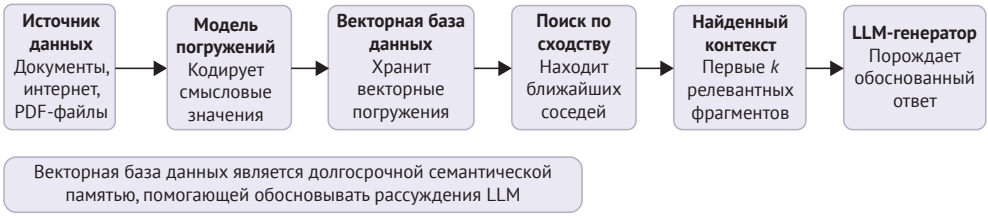
Эвристическое правило:

- используйте FAISS, когда нужна управляемость;
- используйте Pinecone, когда нужна масштабируемость;
- используйте Chroma, когда нужны скорость и простота;
- используйте Weaviate, когда нужна сложная гибридная рассуждающая система.

6. Рекомендации по работе с векторными базами данных

- Используйте единую модель погружений для всех данных.
- Храните метаданные (источник, дата, тип документа) вместе с каждым вектором.
- Сочетайте семантические фильтры с фильтрами по ключевым словам для более точного информационного поиска.
- Кешируйте частые запросы, чтобы уменьшить задержку.
- Переиндексируйте данные, когда модели погружений изменяются.
- Оценивайте качество поиска с помощью recall@k или precision@k.
- Относитесь к своей векторной базе данных как к базовому компоненту инфраструктуры, а не как к добавлению задним числом. От ее качества зависит, насколько правдивой и последовательной окажется ваша RAG- система.

Векторные базы данных – это архитектура памяти, лежащая в основе всех операций системы генерации с дополненной выборкой. В них хранятся смысловые значения, а не слова, что позволяет моделям находить информацию, рассуждать и давать ответы, основываясь на фактических, а не на запомненных когда-то данных.



Независимо от того, отдаете вы предпочтение FAISS ради точности, Pinecone ради масштабируемости, Chroma ради простоты или Weaviate ради гибридных рассуждений, выбор векторной базы данных определяет фактическую надежность вашего ИИ-агента.

3.4. Разбиение документов на фрагменты, погружения и логика поиска

От неструктурированных данных к допускающим поиск смысловым значениям. От документов к знаниям

Каждая RAG-система начинается с исходных материалов: PDF-файлов, статей, расшифровок устных выступлений или журналов чатов.

Однако эти исходные тексты слишком велики и не структурированы, поэтому языковая модель не может обработать их непосредственно. Задача состоит в том, чтобы преобразовать неструктурированный текст в семантические единицы – смысловые фрагменты, которые можно эффективно хранить, индексировать и извлекать.

Это преобразование выполняется в три этапа:

- разбиение на фрагменты: разбить документы на управляемые, сохраняющие контекст куски;
- погружение: преобразовать эти фрагменты в числовые векторы;
- логика поиска: определить, как искать и ранжировать эти фрагменты в ответ на запрос пользователя.

В совокупности эти шаги превращают статическую коллекцию документов в «живую» базу знаний, благодаря которой ИИ-агент может рассуждать.

1. Разбиение документа на фрагменты: создание осмысленных единиц

Почему разбиение на фрагменты важно

Языковые модели работают с ограниченным контекстным окном. Загружать документы целиком неэффективно и зачастую нерелевантно.

Разбиение на фрагменты решает эту проблему путем деления длинного текста на меньшие, семантически связные части, в которых гранулярность (достаточная детализация) и контекст (достаточное смысловое значение) уравновешены.

Хорошие фрагменты должны:

- содержать законченную мысль или тему (а не быть просто случайными текстами заданной длины);
- в минимально необходимой степени перекрываться с соседними фрагментами, чтобы обеспечить непрерывность контекста;
- обеспечить согласованность во всех документах для надежного поиска.

Пример: функция разбиения на Python

```
def chunk_size=500, overlap=50):
    chunks = []
    start = 0
    while start < len(text):
        end = min(len(text), start + chunk_size)
        chunk = text[start:end]
        chunks.append(chunk)
        start += chunk_size - overlap
    return chunks
```

Эта простая функция создает перекрывающиеся фрагменты текста, пригодные для погружения и поиска.

Рекомендации по разбиению на фрагменты

- Для RAG-систем на основе LLM ориентируйтесь на фрагменты длиной 400–800 токенов.
- Обеспечьте логическую связность – старайтесь не разрывать предложения.
- Оставляйте небольшие перекрытия (10–20 %) для сохранения непрерывности контекста.
- Включайте для каждого фрагмента метаданные (название документа, источник, временную метку).
- Для структурированных данных (например, PDF-файлов) старайтесь выполнять семантическую фрагментацию по границам разделов или заголовкам.

Разбиение на фрагменты призвано не только уменьшить размер текста, но и сохранить его смысл.

2. Погружение: преобразования смысла в векторы

Что такое погружение?

Погружение – это числовой вектор, который представляет смысловое значение фрагмента текста.

Семантически схожие тексты имеют схожие погружения, что позволяет системе измерять их близость по коэффициенту Отиаи или другой метрике.

Модели погружения отображают текст в многомерное пространство, где контекстуальное сходство кодируется геометрически.

Выбор модели погружения

Правило выбора: выбирайте модель погружения исходя из масштаба предметной области, потребностей языка и среды развертывания (в облаке или локально).

Пример: генерация погружений

```
from openai import OpenAI
client = OpenAI(api_key="your_key")

def get_embeddings(texts):
    response = client.embeddings.create(
        input=texts,
        model="text-embedding-3-small"
    )
    return [item.embedding for item in response.data]
```

Каждый фрагмент вашего набора данных становится вектором, готовым к сохранению в векторной базе данных, такой как FAISS, Pinecone, Chroma или Weaviate.

Рекомендации по погружению

- Всегда применяйте одну и ту же модель погружения к своему набору данных.
- Нормируйте векторы на единичную длину для получения согласованных метрик схожести.
- Храните метаданные (название, раздел, источник, временную метку) вместе с погружениями.
- Организуйте погружения в пакеты, чтобы уменьшить число вызовов API и задержку.
- Периодически заново вычисляйте погружения для всего корпуса текстов, когда модели улучшаются.

3. Логика поиска: нахождение подходящих фрагментов

Как работает поиск

После того как пользователь задал вопрос:

- вопрос преобразуется в погружение запроса;
- система ищет ближайшие векторы в базе данных;
- первые *k* результатов (например, 3–5) возвращаются в качестве наиболее релевантных контекстов;
- эти результаты подаются на вход LLM в качестве найденных фактов, чтобы та могла выполнить генерацию обоснованного ответа.

Этот процесс позволяет агенту справляться с фактами, а не полагаться на состояние памяти после предобучения.

Пример: технологический процесс поиска

```
def index, top_k=5):
    query_vector = embed_query(query)
    distances, indices = index.search(query_vector, top_k)
    results = [documents[i] for i in indices[0]]
    return results
```

Выбор подходящей метрики сходства

В большинстве RAG-систем в качестве стандартной метрики принимается коэффициент Отиаи (косинусное расстояние).

Параметры поиска

Сочетание поиска с облегченным ранжированием гарантирует, что только наиболее релевантные факты достигают модели.

4. Конвейер интеграции

Полностью конвейер RAG теперь выглядит следующим образом:

- получить документы: загрузить и предварительно обработать текст;
- разбить текст на фрагменты: разбить на логически связные куски;
- сгенерировать погружения: преобразовать каждый фрагмент в вектор;
- сохранить в векторной базе данных: вставить векторы вместе с метаданными в базу;
- обработать поисковый запрос: преобразовать запрос пользователя в вектор;
- ранжировать и отфильтровать: выбрать первые *k* релевантных фрагментов;
- сгенерировать ответ: передать найденный контекст LLM для генерации обоснованного ответа.

5. Оценка и тестирование

Измеряйте качество поиска, прежде чем подключаться к LLM:

- recall@k: насколько часто правильный ответ входит в число первых *k* результатов;
- precision@k: насколько релевантны найденные результаты;
- задержка: среднее время обработки запроса;
- покрытие: сколько документов участвовало в поиске по прошествии некоторого времени.

Оценка качества поиска дает независимое подтверждение того, что уровень обоснования работает, еще до того, как начинается генерация.

Разбиение документов на фрагменты, погружение и поиск – фундамент основанного на знаниях интеллекта в RAG-системах. Эти действия определяют, насколько хорошо ваша модель справляется с доступом к внешним фактам, их интерпретацией и использованием.

Хорошо спроектированный конвейер преобразует текст в структурированный смысл и тем самым устраняет разрыв между человеческими знаниями и машинным рассуждением.

Хороший информационный поиск на 80 % обеспечивает успех генерации. Без него предоставленная самой себе, даже самая продвинутая модель будет галлюцинировать.

3.5. Метрики оценки RAG, отладка и повышение правдивости систем генерации с дополненной выборкой

От работающей к надежной

Работающая RAG-система и хорошо работающая RAG-система – необязательно одно и то же.

Вы можете соединить между собой поисковик, векторное хранилище и генератор, но все равно получать расплывчатые, неполные или галлюцинирующие ответы.

Для обеспечения надежности необходимо количественно оценить каждый компонент и качественно отладить систему.

Оценка RAG-системы сфокусирована на трех уровнях качества:

- качестве поиска – отобрала ли система правильную информацию?
- качестве генерации – правильно ли LLM использовала эту информацию?
- точности обоснования – является ли окончательный ответ согласованным с найденными фактами?

На каждом уровне имеются свои собственные метрики и методы диагностики, совокупность которых является основой для создания правдивых, допускающих проверку и объяснимых ИИ-агентов.

1. Оценивание поисковика

Поисковик отвечает за поиск правильной информации. При его оценке основное внимание следует обращать на полноту (сколько релевантных данных он находит) и точность (сколько нерелевантных данных он исключает).

Интерпретация

- Высокая полнота: поисковик находит все возможные релевантные фрагменты.
- Высокая точность: поисковик эффективно отфильтровывает шум.
- Высокое MAP (Mean Average Precision – среднее значение средней точности) или nDCG (нормированный дисконтированный кумулятивный выигрыш): релевантные документы неизменно оказываются близко к началу.

Практический совет

В производственных RAG-системах отдавайте предпочтение полноте на этапе поиска, а затем улучшайте точность с помощью переранжирования. При таком подходе у LLM никогда не будет недостатка в релевантном материале.

2. Оценивание генератора

Генератор (LLM) принимает найденный контекст и составляет окончательный ответ.

В его работе есть тонкости: необходимо найти баланс между обоснованностью фактами, беглостью текста и верностью найденным свидетельствам.

Количественные подходы

- Автоматизированное оценивание фактов: сравнить сгенерированные варианты ответов с найденным текстом, пользуясь моделью логического следования (entailment) (например, DeBERTa или оценкой GPT-4).
- Оценивание человеком: воспользоваться услугами экспертов в предметной области для оценки соответствия фактам по пятибалльной шкале.
- Оценивание по сходству погружений: вычислить коэффициент Отиаи между выходом и релевантными фрагментами.

Генерация хороша лишь настолько, насколько хорош механизм обоснования. Если найденные факты не годятся, то даже самая лучшая LLM будет галлюцинировать.

3. Вычисление сквозного качества конвейера RAG

Для оценки качества всего конвейера следует объединить метрики поиска и генерации.

Эвристическое правило

RAG-система с высоким значением Recall@k, но плохими метриками AG (Answer Grounding) или CU (Context Utilization), скорее всего, находит слишком много нерелевантных данных, что перегружает контекстное окно модели.

4. Отладка RAG-системы

Когда оценка начинает уменьшаться, необходимо отладить конвейер RAG.

Большинство неприятностей происходят в одном из трех мест: поиске, запросе или управлении контекстом.

A. Отладка поиска

Типичные проблемы:

- возвращен контекст, не релевантный запросу;
- в поиске доминирует текст общего характера или повторяющийся;
- полезные разделы пропущены из-за плохих погружений.

Контрольный список для отладки

- Проверить первые *k* найденных фрагментов на релевантность.
- Протоколировать оценки сходства – для нерелевантных фрагментов коэффициент Отиаи обычно меньше (< 0.6).
- Визуализировать погружения (с помощью PCA или t-SNE), чтобы выявить ошибки кластеризации.

- Попробовать другую модель погружений или повторить разбиение документов на фрагменты.
- Реализовать предметно ориентированные фильтры (по метаданным, по временному промежутку, по категории).

B. Отладка запросов

Типичные проблемы:

- LLM игнорирует найденный контекст;
- галлюцинации, несмотря на точный поиск;
- слишком длинные или противоречивые инструкции.

Контрольный список для отладки

- Явно проинструктировать модель: «Используй для ответа только указанный ниже контекст».
- Применять структурированный вывод (JSON или цитаты).
- Использовать температуру ⩽ 0.3 для фактических задач.
- Добавить подпорки для рассуждения (например, «пошаговое оправдание»).
- Сравнить обоснованный результат с необоснованным.

C. Отладка управления контекстом

Типичные проблемы:

- модель перегружена слишком большим числом найденных фрагментов;
- избыточные или повторяющиеся факты в контексте;
- из-за отсечения отсутствуют критически важные факты.

Контрольный список для отладки

- Ограничить размер контекста (например, первые 3–5 фрагментов).
- Обобщить контекст, прежде чем подавать на вход модели.
- Отслеживать использование токенов и точки отсечения.
- Использовать ранжирование, взвешенное по релевантности (фрагменты с большими оценками раньше).
- Поэкспериментировать с моделями сжатия при поиске (например, с «контекстными фильтрами» в LlamaIndex).

5. Методы диагностики

A. Визуализация пространства погружений

Постройте погружения документа в двумерное пространство (с помощью PCA или t-SNE), чтобы увидеть, попадают ли похожие темы в один кластер.

Плохая кластеризация обычно свидетельствует о низком качестве погружения или неудачной нормировке.

B. Журналы поиска

Записывайте в журналы:

- текст запроса;
- идентификаторы найденных документов;
- оценки сходства;
- результаты, выданные моделью;
- был ли использован найденный контекст.

Анализ журналов часто дает информацию о том, нашел ли поисковик подходящие факты для запросов.

C. Ручное выполнение контрольного следа

Для особо ответственных систем (например, юридических или медицинских) выполняйте «контрольный след» – отображение каждого сгенерированного утверждения на служащий его подкреплением контекст.

Это позволит аналитику-человеку проверить каждое якобы основанное на фактах утверждение.

6. Автоматизация оценивания

Автоматическая оценка RAG может сильно ускорить итерации.

Ниже показан минимальный технологический процесс.

```
def evaluate_rag_system(queries, gold_answers, retriever, llm):
    results = []
    for query, gold in zip(queries, gold_answers):
        retrieved = retriever.retrieve(query)
        generated = llm.generate(query, retrieved)
        recall = compute_recall(retrieved, gold)
        precision = compute_precision(retrieved, gold)
        groundedness = check_grounding(generated, retrieved)
        results.append((recall, precision, groundedness))
    return results
```

Его можно дополнить более продвинутыми методами оценивания, такими как модели логического вывода (entailment), анализ разнообразия результатов или включение обратной связи с пользователем.

7. Непрерывный мониторинг в производственном режиме

Оценивание – это не разовая процедура. В развернутых системах дрейф параметров, обновления данных или изменения модели погружения могут со временем приводить к снижению качества.

Для реализации мониторинга используйте следующие приемы.

- Периодически предоставляйте выборочные запросы человеку для анализа.
- Применяйте автоматизированные инструментальные панели метрик (Recall@k, частоты галлюцинаций).
- Устанавливайте тревожные механизмы в случае внезапных провалов показателей или всплесков задержки.

- Применяйте версионирование поисковиков и погружений для обеспечения воспроизводимости.
- Метрики Recall@k и Precision@k вычисляют качество поиска; высокое значение первой гарантирует полноту, а второй – релевантность.
- Показатели правдивости (faithfulness), обоснованности (groundedness) и частоты галлюцинаций (hallucination rate) измеряют правильность генерации с точки зрения фактов.
- Проводите отладку систематически – сначала проверяйте поиск, затем структуру запроса и затем размер контекста.
- Применяйте протоколирование, визуализацию и автоматизированные конвейеры оценивания для текущего сопровождения.
- Непрерывность оценивания важна; без мониторинга RAG-системы постепенно деградируют.

RAG-системы отказывают не замолкая, а незаметно уходя в сторону.

Без систематического оценивания даже небольшой дрейф поиска или неподстроенный запрос может привести к серьезным фактическим ошибкам.

Успешный конвейер RAG не просто строится, а измеряется, мониторится и сопровождается так же дисциплинированно, как любая информационная система.

Истинность ИИ не функция, а метрика.

3.6. Агент в действии: построение мини-бота знаний на основе RAG, который умеет подгружать факты, разбивать на фрагменты, искать и генерировать обоснованные ответы

Что будет построено

Компактный бот знаний, который:

- заполняет папку документов (PDF/TXT/MD);
- разбивает их на фрагменты и подгружает;
- сохраняет векторы в локальном векторном хранилище;
- находит первые *k* фактов в ответ на запрос;
- генерирует обоснованный ответ с указанием источников.

Компоненты готовой системы

- Командный инструмент: kb ask «ваш вопрос».
- Простой HTTP API на основе FastAPI.
- Допускающие повторное использование функции для подгрузки и опроса документов.

Обзор архитектуры

Конвейер: подгрузка → разбиение на фрагменты → погружение → векторное хранилище → поиск → ранжирование → генерация (с цитатами).

Диаграмма (для дизайнера).

Название: «Минимальный RAG-бот знаний».

Поток: Файлы → Загрузчик текстов → Средство разбиения на фрагменты (500–800 токенов, перекрытие 10–20 %) → Погружения→ Векторная БД (Chroma/FAISS) → Запрос → Поиск по сходству (первые *k*) → Повторное ранжирование (факультативно) → Запрос к LLM (контекст + правила) → Ответ + Источники.

Примечание: бот обосновывает каждый ответ найденными фактами, а не памятью модели

Предварительные условия

- Python 3.10+

Рекомендуется:

```
pip install fastapi uvicorn pydantic python-dotenv httpx pypdf
chromadb sentence-transformers
```

Если для погружений или генерации используются продукты OpenAI, то:

```
pip install openai
```

- Переменные окружения (если используются продукты OpenAI):

```
OPENAI_API_KEY=...
```

- Структура проекта:

```
mini_rag_bot/
├─ .env
├─ data/ # исходные документы
├─ store/ # локальное векторное хранилище
├─ ingest.py # построение индекса
├─ rag.py # поиск + генерация
├─ prompts.py # обоснованные запросы генерации
├─ serve.py # конечная точка FastAPI
└─ cli.py # командная управляющая программа
```

Шаг 1 – подгрузка, разбиение на фрагменты и погружение

Используется простое, согласованное разбиение на фрагменты и надежная модель погружения. В случае полностью локальных операций используется SentenceTransformers, в случае генерации в облаке – погружения OpenAI.

```
# ingest.py
import os, glob
from pathlib import Path
from typing import List, Dict
from sentence_transformers import SentenceTransformer
```

```
import chromadb
from chromadb.utils import embedding_functions

CHROMA_DIR = "store"
DATA_DIR = "data"
COLLECTION = "kb"

def load_texts(path: str) -> List[Dict]:
    docs = []
    for fp in glob.glob(os.path.join(path, "**/*.*"), recursive=True):
        ext = Path(fp).suffix.lower()
        if ext in [".txt", ".md"]:
            docs.append({"id": fp, "text": Path(fp).read_text(encoding="utf-8"),
                         "source": fp})
        elif ext == ".pdf":
        # минимальное извлечение текста из PDF
        from pypdf import PdfReader
        text = []
        for page in PdfReader(fp).pages:
            text.append(page.extract_text() or "")
        docs.append({"id": fp, "text": "\n".join(text), "source": fp})
    return docs
def chunk(text: str, size: int = 800, overlap: int = 120) -> List[str]:
    out, start = [], 0
    while start < len(text):
        end = min(len(text), start + size)
        piece = text[start:end]
        # стараемся избежать разрыва посередине предложения
        if end < len(text):
            last_period = piece.rfind(". ")
            if last_period > int(size * 0.6):
                end = start + last_period + 2
                piece = text[start:end]
        out.append(piece.strip())
        start = max(end - overlap, end)
    return [c for c in out if c]

def build_index():
    client = chromadb.PersistentClient(path=CHROMA_DIR)
    # Локальные погружения (без внешних вызовов). При желании можете
    # переключиться на OpenAI.
    model = SentenceTransformer("all-MiniLM-L6-v2")
    ef = embedding_functions.SentenceTransformerEmbeddingFunction(
           model_name="all-MiniLM-L6-v2")
    col = client.get_or_create_collection(name=COLLECTION,
                                          embedding_function=ef)
    docs = load_texts(DATA_DIR)
    ids, texts, metas = [], [], []
    for d in docs:
        pieces = chunk(d["text"])
        for i, p in enumerate(pieces):
            ids.append(f"{d['id']}::chunk::{i}")
```

```
            texts.append(p)
            metas.append({"source": d["source"], "chunk": i})

    # Вставить или заменить пакетно
    B = 256
    for i in range(0, len(texts), B):
        col.upsert(ids=ids[i:i+B], documents=texts[i:i+B],
                   metadatas=metas[i:i+B])
        print(f"Indexed {len(texts)} chunks from {len(docs)} documents into
{COLLECTION}.")

if __name__ == "__main__":
    build_index()
```

Примечания

- Размер фрагмента 800 символов с перекрытием 120 символов дает хороший баланс между контекстом и полнотой.
- Если требуется разбиение с учетом токенов, переключитесь на токенизатор.
- Для обеспечения прослеживаемости храните источник и фрагмент в качестве метаданных.

Шаг 2 – поиск и обоснованная генерация

Строгий запрос предотвращает галлюцинации. Если контекста недостаточно, бот должен так и сказать.

```
# prompts.py
SYSTEM_GROUNDED = """Ты основанныый на фактах ассистент. Для ответа
используй ТОЛЬКО предоставленный контекст CONTEXT.
Если в CONTEXT нет ответа, скажи: "Недостаточно информации в предоставленных
документах."
Пиши понятно и кратко. Верни короткий список всех использованных путей к
источникам 'Sources'."""
USER_TEMPLATE = """QUESTION:
{question}
CONTEXT:
{context}
"""


# rag.py
from typing import List, Tuple
import chromadb
from prompts import SYSTEM_GROUNDED, USER_TEMPLATE

# выбрать одну из платформ генерации:
USE_OPENAI = False
if USE_OPENAI:
    from openai import OpenAI
    import os
    client_oa = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

```
    def generate(text: str) -> str:
        resp = client_oa.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role":"system","content":SYSTEM_GROUNDED},
                      {"role":"user","content":text}],
            temperature=0.2,
        )
        return resp.choices[0].message.content.strip()
else:
    # облегченная локальная генерация с использованием sentence-transformers для
    # обеспечения релевантности и минималистского основанного только на шаблоне
    # ответа (для иллюстрации).
    def generate(text: str) -> str:
        return "Сервер локальной генерации не сконфигурирован. Подключите поставщика LLM."

def retrieve(query: str, k: int = 5) -> List[Tuple[str, dict]]:
    client = chromadb.PersistentClient(path="store")
    col = client.get_collection("kb")
    res = col.query(query_texts=[query], n_results=k,
                    include= ["documents","metadatas","distances"])
    docs = res["documents"][0]
    metas = res["metadatas"][0]
    dists = res["distances"][0]
    out = []

    for doc, meta, dist in zip(docs, metas, dists):
        out.append((doc, meta | {"distance": dist}))
    return out

def format_context(snippets: List[Tuple[str, dict]]) -> str:
    lines = []
    for i, (doc, meta) in enumerate(snippets, 1):
        src = meta.get("source","unknown")
        lines.append(f"[{i}] Source: {src} (chunk
            {meta.get('chunk')})\n{doc}\n")
    return "\n".join(lines)

def answer(question: str, k: int = 5) -> dict:
    hits = retrieve(question, k=k)
    context = format_context(hits)
    prompt = USER_TEMPLATE.format(question=question, context=context)
    output = generate(prompt)
    sources = [h[1].get("source") for h in hits]
    return {"question": question, "answer": output, "sources": sources, "k": k}

if __name__ == "__main__":
    print(answer("Какие шаги установки описаны в документации?"))
```

Рекомендации

- Установите USE_OPENAI = True для реальных ответов.
- Если корпус документов велик, подумайте о добавлении средства переранжирования на основе кросс-кодировщика перед `format_context`.

Шаг 3 – командный и HTTP-интерфейс

```
# cli.py
import sys, json
from rag import answer

def main():
    q = " ".join(sys.argv[1:]) or "Что справочник говорит о PTO?"
    res = answer(q, k=5)
    print(json.dumps(res, indent=2, ensure_ascii=False))
    if __name__ == "__main__":
        main()

# serve.py
from fastapi import FastAPI
from pydantic import BaseModel
from rag import answer
app = FastAPI(title="Mini RAG Knowledge Bot")
class Q(BaseModel):
    question: str
    k: int = 5
    @app.post("/ask")

    def ask(q: Q):
        return answer(q.question, k=q.k)
```

Выполнение

```
python ingest.py
uvicorn serve:app—reload—port 8000
python cli.py "Summarize onboarding policy."
```

Укрепление цикла

- Переранжирование.
- Используйте кросс-кодировщик (например, cross-encoder/ms-marco-MiniLM-L-6-v2) для переранжирования первых 20 результатов поиска и оставьте лучшие 5.
- Сжатие контекста.
- Если текст объединенных фрагментов превосходит размер контекстного окна вашей LLM, обобщите каждый фрагмент перед сборкой, сохранив ссылки на источники.
- Строгая обоснованность.

  Форматируйте результат в соответствии с JSON-схемой:

  ```
  { "answer": "...", "citations": [ { "source": "...", "chunk": n } ] }
  ```

- Ограничения.

  Если оценки сходства малы или число релевантных фрагментов меньше 2, верните «Недостаточно информации в предоставленных документах» и предложите пользователю добавить источники.

Контрольные списки для отладки

Поиск.

- Проверяйте первые *k* фрагментов на тематическую релевантность.
- Подберите размер фрагмента (начните с 800 символов с перекрытием 10–20 %).
- Перейдите на другую модель погружений или дообучите текущую, если результаты не соответствуют теме.

Формулировка запроса

- Включите в запрос системе инструкцию «Используй ТОЛЬКО предоставленный КОНТЕКСТ».
- Для задач, обоснованных фактами, делайте температуру низкой (0.0–0.3).

Задержка

- Собирайте погружения в пакет на этапе подгрузки.
- Используйте подходящие индексы (HNSW/IVF), если количество векторов исчисляется миллионами и более.

Минимальная программа оценивания

Измеряйте качество поиска и обоснованности перед окончательным развертыванием.

```
# eval_retrieval.py
from rag import retrieve

def recall_at_k(queries, gold_sets, k=5):
    hits = 0; total = 0
    for q, gold_sources in zip(queries, gold_sets):
        res = retrieve(q, k=k)
        found = {m[1]["source"] for m in res}
        hits += len(found.intersection(set(gold_sources))) > 0
        total += 1
    return hits / total
    # gold_sets: список множеств ожидаемых файлов-источников на один запрос
```

Иногда подключайте человека: для случайно выбранного запроса удостоверьтесь, что все утверждения обоснованы найденным контекстом.

Безопасность и гигиена данных

- Никогда не включайте в данные секреты или персональную идентифицирующую информацию. Фильтруйте входные данные и исключайте содержащие секреты строки на этапе подгрузки.
- Сохраняйте минимальные данные, необходимые для обеспечения прослеживаемости (пусть к файлу, раздел, временную метку).
- Назначайте номера версий погружениям при изменении моделей; сохраняйте скрипты миграции для детерминированного перестроения индекса.

- Качество RAG определяется прежде всего хорошим разбиением на фрагменты и погружениями. Начинайте с простого и постепенно улучшайте.
- Запросы должны быть строгими: если ответ отсутствует в контексте, так и скажите.
- По мере роста корпуса добавляйте переранжирование и сжатие.
- Сначала оценивайте поиск, затем генерацию; обосновывайте все явными источниками.

3.7. На практике: как RAG доводит концепцию корпоративных чат-ботов до эксплуатации: построение крупномасштабного обоснованного интеллекта

Эволюция чат-ботов: от скрипта до системы знаний

Более десяти лет большинство чат-ботов следовало предсказуемой схеме – это основанные на правилах скрипты, которые производили поиск по ключевым словам и возвращали предопределенные ответы. Эта схема годилась для таких ограниченных задач, как поиск среди часто задаваемых вопросов (FAQ), но давала сбой при столкновении с нюансами, неоднозначностью или контекстом, который не был явно запрограммирован.

Появление генерации с дополненной выборкой (RAG) все изменило.

Благодаря связыванию больших языковых моделей (LLM) с источниками корпоративных данных чат-боты на основе RAG могут отвечать динамически, точно и с учетом контекста, не требуя переобучения модели после каждого поступления новой информации.

Сегодня RAG является базовой архитектурой, стоящей за генерацией в корпоративных диалоговых системах: помощниках в службе поддержки клиентов, внутренних ботах знаний, консультантах по соблюдению требований регуляторов (комплаенсу) и т. д.

RAG превращает чат-боты из разговорных интерфейсов в рассуждающие системы.

1. Почему корпорации переходят на RAG

Для бизнеса опора на факты и объяснимость играют такую же важную роль, как беглость разговорного языка.

Чат-бот, который придумывает ответ (галлюцинирует) не просто некорректен – он может нанести финансовый или юридический ущерб.

Отделяя знания от модели, RAG предоставляет организациям детальный контроль над тем, что чат-бот знает и как он рассуждает.

2. Обзор архитектуры корпоративной RAG

RAG-чат-бот производственного качества обычно включает пять основных уровней.

Уровень подгрузки данных

- Собирает структурированную (базы данных, API) и неструктурированную (документы, PDF-файлы, вики) информацию.
- Включает конвейеры ETL и нормализации контента.

Уровень векторизации

- Преобразует все данные в погружения, используя модель, специализированную для предметной области.
- Сохраняет результаты в масштабируемой векторной базе данных (Pinecone, Weaviate или кластере FAISS).

Уровень поиска

- Производит гибридный поиск по сочетанию ключевых слов, метаданных и сходству векторов.
- Возвращает первые *k* результатов, отвечающих запросу пользователя.

Уровень рассуждений (LLM)

- Языковая модель интерпретирует запрос, читает найденный контент и составляет обоснованный фактами ответ, следуя инструкциям в RAG-запросе.

Уровень оркестрации

- Управляет технологическим процессом: маршрутизирует запросы, контролирует доступ, мониторит использование и кеширует частые результаты.

3. Практические примеры

А. Автоматизация поддержки пользователей

Глобальный поставщик SaaS заменил свой основанный на правилах бот техподдержки RAG-системой, построенной на основе OpenAI GPT-4 и Pinecone.

До внедрения RAG:

- ответы устаревали за несколько недель;
- не было возможности управлять исключениями из политик;
- 40 % запросов передавалось операторам.

После внедрения RAG:

- данные извлекаются из актуальных политик и документации по API;
- 92 % запросов решаются в процессе общения с чат-ботом;
- генерируемые ответы обоснованы базой знаний компании и снабжены ссылками на источники.

Архитектурные особенности

- Векторное хранилище: Pinecone с фильтрацией метаданных по продукту и версии.
- Поиск: гибридный поиск по ключевым словам и смысловым значениям.
- Генератор: GPT-4 с температурой 0.2 для согласованности с фактами.

B. Внутренний ассистент знаний

Крупная фармацевтическая компания развернула внутреннего RAG-ассистента для научно-исследовательского персонала, который отвечает на запросы по клиническим испытаниям и нормативным документам.

Компоненты системы

- Подгрузка данных с помощью безопасных конвейеров документов (SharePoint, PDF-файлы, научные журналы).
- Погружения, генерируемые с помощью предметно-ориентированной модели (база биомедицинских погружений).
- Поиск, интегрированный с Neo4j, для связывания структурированных сущностей (например, препарат → испытания → результат).
- Выход включает фрагменты документов и ссылки.

Результаты

- Время поиска сократилось с часов до секунд.
- Рецензенты отмечали улучшение точности исследований на 65 %.
- Группы комплаенса получили полный провенанс информации.

C. Боты-консультанты по комплаенсу и юридическим вопросам

В регулируемых отраслях (финансы, страхование, здравоохранение) RAG-чат-боты проектируются так, что никогда не прибегают к догадкам.

Принцип проектирования

Если контекст не содержит ответа, бот должен явно ответить: «Этой информации нет в текущем наборе данных».

Этот принцип гарантирует возможность аудита и защиты в суде.

Организации часто реализуют автоматизированную логику в случае, если ответ не найден: такие запросы протоколируются и добавляются в корпус после проверки.

4. Вопросы проектирования для корпоративных RAG-систем

Безопасность данных

- Хранить погружения и документы в защищенной корпоративной среде.
- Использовать локальные модели погружения, если вызовы внешних API рискованны с точки зрения комплаенса.
- Реализовать контроль доступа на уровне строк в векторных базах данных.

Оптимизация задержки

- Кешировать частые запросы в Redis или с помощью локального уровня кеширования.
- Использовать индексы для поиска приближенных ближайших соседей (approximate nearest neighbor – ANN) в случае больших корпусов.
- Предварительно вычислять кластеры запросов для предсказания популярных тем.

Масштабируемость

- Горизонтальное масштабирование с помощью управляемых служб (Pinecone, Weaviate Cloud).
- Периодическая переиндексация для новых наборов документов.
- Микросервисная архитектура поиска, генерации и мониторинга.

Оценивание и мониторинг

- Постоянно следить за Recall@k, обоснованностью и частотой галлюцинаций.
- Включать обратную связь с человеком (например, значок одобрения или неодобрения в UI).
- Протоколировать все сгенерированные утверждения вместе с контекстом для возможности аудита.

5. Измеримые последствия внедрения RAG-чат-ботов

Предприятия, внедрившие RAG-чат-боты, отмечают следующее:

- улучшение обоснованности фактами до 70 % по сравнению с простыми LLM;
- сокращение затрат в связи с более редкими циклами дообучения и переобучения;
- ускорение обновления контента – часы вместо недель;
- повышение доверия пользователей благодаря прозрачности ссылок на источники.

Хорошо спроектированный RAG-чат-бот не просто отвечает на вопрос, а строит коллективную память, которая растет вместе с каждым взаимодействием.

6. Будущее: RAG соединяется с графами знаний и многоагентными системами

Вновь создаваемые корпоративные системы сочетают RAG с графами знаний, что позволяет реализовать структурированное рассуждение поверх найденного контекста.

Этот гибридный подход эволюционировал в «RAG 2.0», где:

- RAG отвечает за обоснованность фактами;
- графы знаний добавляют структурированный вывод;
- многоагентные системы координируют рассуждения и проверку.

Такие системы уже появляются в продвинутом корпоративном поиске, медицинской диагностике и юриспруденции.

7. Резюме: от ответов к пониманию

Генерация с дополненной выборкой стала стандартной схемой в корпоративных ИИ-системах, потому что позволяет языковым моделям сочетать контекстную беглость с основанной на фактах дисциплине.

Чат-бот, основанный на RAG, обладает следующими возможностями:

- обосновывает свои рассуждения проверенными данными;
- объясняет, откуда взят каждый факт;
- мгновенно адаптируется при поступлении новой информации.

Будущее корпоративных чатов не в генерации текстов, а в генерации заслуживающих доверия результатов. Это устраняет разрыв между креативностью LLM и корпоративной надежностью.

Обосновывая каждый ответ найденными фактами, допускающими аудит, организации могут наконец внедрить разговорный ИИ, который будет точным, прозрачным и постоянно обучающимся.

Часть II

Построение фундамента интеллектуальной системы

Глава 4

Графы знаний: придание структуры хаосу

4.1. Что такое граф знаний

От неструктурированных данных к рассуждениям с учетом контекста. Внесение порядка в информацию

Сегодня любое предприятие, исследовательская лаборатория или ИИ-система сталкиваются с одним и тем же парадоксом: чем больше данных она собирает, тем меньше знает.

Документы, журналы, электронные таблицы, PDF-файлы, сообщения электронной почты – все они содержат ценные факты, но в своей исходной форме остаются неструктурированными островками знаний. RAG-система может извлечь релевантные фрагменты, но ей недостает знания о связях – о том, как связаны между собой люди, события и концепты.

На помощь приходит граф знаний (ГЗ).

Граф знаний придает информации форму, выражая ее в виде связей между сущностями. Он хранит не предложения, а смысловые значения – кто что сделал, когда и как это соотносится со всем остальным.

По сути дела, граф знаний преобразует разрозненные факты в машинно-читаемую карту смыслов.

1. Определение

Граф знаний – это структурированное представление данных, которое организует сущности (предметы, люди, места, события, концепции) и их связи в виде графа, состоящего из вершин и ребер.

Формально: граф знаний G = (V, E) состоит из множества вершин (V), представляющих сущности, и множества ребер (E), представляющих связи между ними.

Каждая связь представляется тройкой, которую часто записывают в виде (субъект) – (предикат) – (объект). Например:

«Алан Тьюринг» – «изобрел» – «вычислительные машины и интеллект»;

«Алан Тьюринг» – «родился_в»» – «Лондон».

Эти тройки становятся фактами, которые система может запрашивать, о которых может рассуждать и включать в более крупные контексты.

2. Граф знаний и базы данных

На первый взгляд граф знаний напоминает базу данных – и там, и там хранится информация.

Но по структуре и назначению они принципиально различаются.

Реляционная база данных может сказать, какие данные существуют.

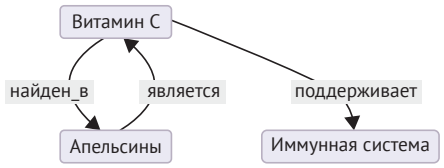
Граф знаний может сказать, как данные соотносятся, что открывает возможность для контекстуальных рассуждений.

3. Компоненты графа знаний

Типичный граф знаний состоит из пяти основных компонентов:

- сущности (концепции);
- связи (предикаты);
- атрибуты (свойства);
- онтологии;
- сообщества.

Такая структура позволяет человеку и машине осмысленно переходить от одной концепции к другой.



Граф знаний моделирует концепции и их связи, создавая сеть взаимосвязанных смысловых значений

4. Графы знаний в экосистеме ИИ

Графы знаний играют важную роль в обосновании рассуждений ИИ. Они служат структурированной памятью для людей и машин.

Например, в конвейере RAG:

- поисковик находит факты, выраженные в текстовом виде;
- граф знаний предоставляет семантическую структуру, что позволяет рассуждать о сущностях и связях;
- затем LLM объединяет структурированные и неструктурированные данные для генерации ответа с учетом контекста.

Эта гибридная модель дает системе возможность отвечать не только на вопрос «что?», но и на вопрос «почему?».

Пример интеграции

В корпоративном ассистенте RAG находит документ, в котором говорится, что «Алиса управляет проектом Phoenix». Граф знаний связывает:

- «Алиса» → управляет → «проект Phoenix»;
- «проект Phoenix» → относится_к → «отдел маркетинга»;
- «отдел маркетинга» → подчиняется → «вице-президент по оперативной деятельности».

Теперь ассистент может ответить на вопросы «Кто подчиняется вице-президенту по оперативной деятельности?» или «Кто управляет проектами в отделе маркетинга?», несмотря на то что точно в таком виде эти предложения не встречались в тексте источника.

5. Графы знаний и погружения

Графы знаний легко перепутать с погружениями, поскольку те и другие представляют связи. Но работают они на разных уровнях и на практике дополняют друг друга.

- Погружения находят факты, релевантные запросу.
- Графы знаний объясняют, как эти факты связаны.

6. Преимущества использования графов знаний

- Контекстуальное рассуждение – связывает сущности, встречающиеся в разных документах и наборах данных.
- Объяснимость – для каждого ответа можно проследить явные связи.
- Динамическое обновление – новые сущности и связи можно добавлять, не переделывая схему.
- Междоменные связи – включает структурированные и неструктурированные источники данных.
- Семантический поиск – допускает запросы вида «Показать все лекарственные препараты, которые ингибируют энзимы, связанные с путями развития рака».

Граф знаний работает как системы памяти и опора для рассуждений интеллектуальных агентов.

7. Реальные примеры графов знаний

- Google Knowledge Graph – стоит за результатами поиска и панелями сущностей.
- Microsoft Satori – связывает сущности, встречающиеся в Bing, LinkedIn и Xbox.
- Wikidata – открытая, создаваемая сообществом семантическая база данных, используемая в «Википедии».
- Amazon Product Graph – моделирует связи между товарами, отзывами и поведением пользователей.
- Медицинские графы – связывают заболевания, гены и методы лечения для исследований в области биомедицины.

Во всех примерах используется одна и та же фундаментальная идея: связывать знания, а не просто хранить их.

8. Как графы знаний дополняют RAG

RAG извлекает факты из неструктурированного текста, но ей не хватает памяти о том, как эти факты взаимосвязаны.

Граф знаний восполняет этот пробел, поскольку допускает:

- многошаговые рассуждения: объединение нескольких найденных фактов в связную цепочку;
- устранение неоднозначности: различает «Apple Inc.» и «фрукт яблоко (apple)»;
- верификация: проверяет, согласуются ли найденные факты с известными связями.

Пример

Юридический RAG-ассистент находит пункт, утверждающий, что «контракт X регулируется законодательством Великобритании».

В графе знаний закодировано:

«Contract X» → принадлежит → «Клиент Y» → базирующемуся_в → «Великобритания».

Эти знания допускают рассуждения вида «Какие контракты, регулируемые законодательством Великобритании, распространяются на клиентов, находящихся вне Великобритании?». Чистый поиск на такой вопрос ответить не может.

Граф знаний – это структурированный интеллект, постоянно развивающаяся сеть, связывающая факты в смысловое значение. Он позволяет системе перейти от поиска данных к их пониманию.

RAG предоставляет ИИ доступ к знаниям.

Графы знаний наделяют его контекстом, логикой и объяснимостью.

4.2. Как LLM могут использовать и обновлять структурированные данные

Интеллект на базе естественного языка и символических знаний. Разрыв между текстом и структурой

Большие языковые модели (LLM) великолепно понимают и генерируют язык, но по построению статически обучаются на текстах и не приспособлены к рассуждениям на основе структурированных данных.

Они принципиально не понимают, что «Париж – столица Франции» и «столицей Франции является Париж» – это один и тот же логический факт.

Структурированные системы данных, например базы данных или графы знаний, прекрасно справляются с этим, но им недостает гибкости и лингвистической интуиции.

Самые мощные из современных ИИ-систем соединяют эти два мира.

- LLM дают понимание языка и генеративное рассуждение.
- Графы знаний (и другие структурированные источники данных) дают точность, память и контекстуальную устойчивость.

В результате получается гибридная когнитивная архитектура, в которой LLM не просто читает данные, но также умеет запрашивать, интерпретировать и обновлять структурированные знания с помощью контролируемого взаимодействия.

1. Как LLM взаимодействуют со структурированными данными

LLM не могут «думать» непосредственно в терминах графов или таблиц, но их можно снабдить инструментарием для использования структурированных данных с помощью интерфейсов – API, вызовов функций или уровней трансляции запросов.

Процесс обычно состоит из следующих этапов:

- распознавание намерений – LLM интерпретирует вопрос пользователя и решает, нужны ли для ответа структурированные данные;
- построение запроса – LLM транслирует вопрос на естественном языке в формальный запрос (например, на языке SQL, SPARQL или Cypher);
- поиск данных – выполняет запрос с помощью коннектора или вызова функции;
- интерпретация и рассуждение – преобразует найденные структурированные результаты в удобочитаемые утверждения или синтезированные умозаключения;
- (факультативно) обновление или вставка – при необходимости может создать или модифицировать записи графа знаний с помощью структурированных обновлений.

Эта схема превращает LLM из предиктора текстов в оркестратор данных.

2. Пример: трансляция языка в структурированные запросы

Разберем пример графа знаний, моделирующего работников, отделы и проекты.

Вопрос пользователя: «Кто управляет проектом Apollo и в каком отделе он работает?»

LLM интерпретирует это как структурированный вопрос и конструирует эквивалентный запрос на языке Cypher (для Neo4j):

```
MATCH (p:Project {name: "Project Apollo"})<-[:MANAGES]-
(m:Employee)-[:WORKS_IN]->(d:Department)
RETURN m.name AS manager, d.name AS department
```

После извлечения данных из графовой базы данных LLM форматирует ответ на естественном языке: «Проектом Apollo управляет Алиса Смит из аэрокосмического отдела».

Интеллект находится не в LLM, запомнившей ответ, а в знании того, как задать вопрос структурированной системе.

3. Вызов функций и коннекторы API

Современные LLM (такие как GPT-4, Gemini и Claude) поддерживают вызовы функций – механизм обращения к внешним системам.

В контексте структурированных данных вызов функции служит мостом между текстом в свободной форме и символической логикой.

Пример

```
{
  "name": "query_knowledge_graph",
  "description": "Retrieve relationships between entities in the corporate graph",
  "parameters": {
    "type": "object",
    "properties": {
      "subject": {"type": "string"},
      "predicate": {"type": "string"},
      "object": {"type": "string"}
    },
    "required": ["subject"]
  }
}
```

Пример запроса

Пользователь: «Покажи мне все проекты, которыми управляет Алиса Смит».

LLM понимает намерение и генерирует запрос:

```
{
  "subject": "Alice Smith",
  "predicate": "MANAGES",
  "object": null
}
```

Затем система исполняет запрос на языке Cypher или SPARQL и возвращает результаты модели для генерации повествовательного ответа.

Этот цикл, включающий структурированный запрос и вызов функции, позволяет LLM оставаться посредниками в диалоге, не имеющими состояния, тогда как граф знаний предоставляет хранимый контекст.

4. Обновление структурированных данных

Если извлечение данных – вещь понятная, то обновление структурированных данных требует тщательного контроля, чтобы не нарушить целостность и избежать вставки в базу знаний фактов, порожденных галлюцинациями.

Типичный безопасный технологический процесс выглядит так.

Извлечение фактов

LLM идентифицирует тройки-кандидаты, или связи в новых документах.

Пример

Текст: «Д-р Джейн Фостер поступила на работу в Quantum Labs в 2023 году».

Тройка-кандидат:

(«Джейн Фостер») – («работает_в») – («Quantum Labs»)

с присоединенным свойством start_year = 2023.

Проверка и оценивание уверенности

LLM назначает оценку уверенности на основе лингвистической уверенности или сопоставления фактов.

Проверка с участием человека

При корпоративном или регулируемом использовании предложенные тройки помещаются в очередь для одобрения, прежде чем будут вставлены в базу знаний.

Обновление графа (посредством API)

После одобрения система выполняет формальное обновление командой на языке Cypher или SPARQL:

```
MERGE (p:Person {name: "Jane Foster"})
MERGE (o:Organization {name: "Quantum Labs"})
MERGE (p)-[:WORKS_AT {since: 2023}]->(o)
```

Протоколирование и версионирование

Все модификации протоколируются в интересах прослеживаемости, так чтобы эволюция графа знаний была прозрачна.

Этот процесс позволяет LLM предлагать обновления, но запрещает самостоятельное переписывание знаний.

5. Гибридный пример: самообновляющийся корпоративный ассистент

Представьте себе корпоративного ассистента знаний, интегрированного с внутренними данными и документам отдела кадров.

Работники загружают в RAG-систему отчеты и служебные записки.

LLM извлекает связи типа «Алиса управляет проектом Titan». Она опрашивает существующий граф:

```
MATCH (e:Employee {name: "Alice"})-[:MANAGES]->(p:Project {name: "Project Titan"})
RETURN e, p
```

Если связь отсутствует, то LLM предлагает тройку обновления для проверки. После проверки она обновляет граф.

Будущие запросы вида «Кто управляет проектами в отделе маркетинга?» разрешаются мгновенно, даже если данные поступили из тестового отчета, загруженного часом ранее.

Так создается мозг организации, пополняемый сведениями из структурированных и неструктурированных источников.

6. Почему обновлять структурированные данные важно

Без динамики графы знаний быстро деградируют – их релевантность падает по мере изменения фактов.

LLM продлевают время актуальности благодаря:

- автоматизации извлечения фактов из неструктурированного текста;
- поддержанию семантической согласованности между документами и структурированными сущностями;
- организации моста между внутренними хранилищами, в которых сосуществуют структурированные и неструктурированные данные (например, CRM-системы, вики, базы данных).

В сочетании с конвейерами RAG мы получаем актуальные, допускающие проверку и самоулучшающиеся системы знаний.

7. Проблемы и соображения

Надежная система характеризуется четким разделением обязанностей: LLM предлагают, а структурированные системы проверяют и сохраняют.

- LLM могут опрашивать, рассуждать и предлагать обновления структурированным системам с помощью контролируемых интерфейсов.
- Вызовы функций, трансляция в запросы и конвейеры проверки необходимы для сопряжения рассуждений на естественном и символическом языке.
- Структурированные данные обеспечивают точность, LLM привносят контекст и гибкость.
- Благодаря синергии создаются адаптивные системы знаний, которые обучаются, проверяют и развиваются в режиме реального времени.

RAG соединяет LLM с истиной. Графы знаний организуют истину, обеспечивая ее актуальность и объяснимость.

4.3. Интеграция Neo4j, ArangoDB и GraphRAG

Гибридный интеллект: там, где граф встречается с генеративными моделями. От автономных графов к связному интеллекту

В традиционных системах данных графы знаний существовали как изолированные репозитории – эффективное средство для структурированных рассуждений, но недоступное для разговорных интерфейсов или понимания неструктурированного текста.

RAG изменила это положение вещей, перекинув мост между рассуждениями на основе текстов и структурированным графовым интеллектом. Современная архитектура объединяет:

- графовые базы данных – для представления связей и структурированных фактов;
- LLM – для интерпретации естественного языка и синтеза ответов;
- конвейеры RAG – для соединения обоих миров путем поиска информации, рассуждения и генерации.

Эта интеграция лежит в основе контекстно осведомленных, объяснимых и динамически обновляемых ИИ-агентов, которые не только находят факты, но и понимают, как они связаны между собой.

1. Роль графовых баз данных в RAG

В графовой базе данные хранятся в виде взаимосвязанных вершин (сущностей) и ребер (связей), что делает ее идеальной для рассуждений, в которых связи важнее индивидуальных фактов.

В конвейерах RAG графовая база данных играет следующие роли:

- уровень структурного поиска – дополняет векторные результаты поиска явными связями;
- уровень рассуждения – позволяет модели выводить многоступенчатые связи;
- уровень хранения – сохраняет проверенные связи, выявленные LLM.

Этот структурированный компонент дополняет поиск в неструктурированных документах, реализуемых стандартной RAG, что выливается в GraphRAG – гибридную систему, которая обосновывает рассуждения фактами и связями.

2. Neo4j: индустриальный стандарт для рассуждений на графах

Общие сведения

Neo4j – одна из самых распространенных графовых баз данных, известная своей производительностью, зрелой экосистемой и выразительным языком запросов Cypher.

Основные преимущества

- Интуитивно понятная модель графа свойств: вершины, связи и атрибуты.
- Зрелый инструментарий (Neo4j Bloom, NeoDash) для визуализации.
- Глубокая поддержка экосистемы (Python, Java, REST API).
- Идеальна для графов знаний масштаба предприятия и интеграции с гибридным ИИ.

Схема интеграции: LLM + Neo4j

Обзор архитектуры:

- пользователь предъявляет запрос на естественном языке;
- LLM распознает намерение и генерирует запрос на языке Cypher;
- Neo4j выполняет запрос и находит структурные связи;
- LLM обобщает результаты и факультативно обосновывает их найденным текстом.

Пример потока

Пользователь: «Какие исследователи из МТИ публиковали статьи по квантовой оптимизации, начиная с 2020 года?»

Сгенерированный LLM запрос на языке Cypher:

```
MATCH (r:Researcher)-[:AFFILIATED_WITH]->(i:Institution {name:
"MIT"})
MATCH (r)-[:AUTHORED]->(p:Publication)
WHERE p.topic CONTAINS "quantum optimization" AND p.year >=
2020
RETURN r.name AS researcher, p.title AS paper
```

Обобщение, сгенерированное LLM: «Три исследователя из МТИ – д-р Чен, д-р Альварес и д-р Сингх – недавно публиковали статьи по квантовой оптимизации».

Пример интеграции с Neo4j (Python)

```
from neo4j import GraphDatabase
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))
def query_neo4j(cypher):
with driver.session() as session:
    result = session.run(cypher)
    return [record.data() for record in result]

# Пример использования
cypher = """
MATCH (a:Author)-[:AUTHORED]->(p:Paper)
WHERE p.year = 2024
RETURN a.name, p.title
"""
print(query_neo4j(cypher))
```

Простой API драйвера Neo4j позволяет непосредственно включить результаты поиска на графе в конвейер RAG для последующего рассуждения или обобщения средствами LLM.

3. ArangoDB: мультимодальная альтернатива

Общие сведения

ArangoDB – база данных, поддерживающая хранение документов, графов и пар ключ–значение в одной системе.

Это делает ее особенно гибкой для гибридных RAG-систем, где данные могут поступать в нескольких форматах.

Основные преимущества

- Унифицированный язык запросов (AQL) для данных, хранящихся в графах и документах.
- Поддерживает соединения между векторными хранилищами, документами и графами.
- Без дополнительных средств интегрируется с клиентами на Python и Node.js.
- Может служить слоем структурированных и неструктурированных данных в одной системе.

Схема интеграции: RAG + ArangoDB

Пример сценария использования

Ассистент знаний в системе розничной торговли извлекает как неструктурированные FAQ (хранящиеся в виде документов), так и связи между товарами (хранящиеся в виде графа).

Пример AQL

```
FOR product IN Products
FILTER product.category == "laptops"
FOR review IN Reviews
FILTER review.productId == product._key
RETURN { product: product.name, review: review.text }
```

Пример гибридного запроса (поиск в графе + текстовых векторах):

```
LET vector = @queryVector
FOR doc IN productVectors
LET score = COSINE_SIMILARITY(doc.vector, vector)
FILTER score > 0.8
SORT score DESC
LIMIT 5
RETURN MERGE(doc, {score})
```

Такая гибкость позволяет ArangoDB выступать в роли автономной серверной части гибридной RAG, соединяя поиск по сходству текста с основанным на связях рассуждении без помощи внешних служб.

4. GraphRAG: уровень конвергенции

Что такое GraphRAG?

GraphRAG – это архитектурная парадигма, объединяющая семантический поиск (RAG) с реляционным рассуждением (графы знаний).

Если RAG обосновывает результаты LLM найденными фактами, то GraphRAG позволяет модели:

- выполнять многошаговые рассуждения о сущностях;
- разрешать неоднозначности сущностей (например, различать компанию и реку Amazon (по-русски Амазонка));
- извлекать контекст связи, а не только ее содержание;
- строить постоянную структурированную память, эволюционирующую со временем.

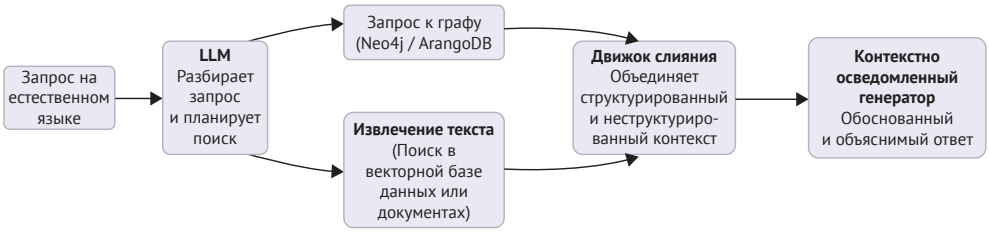
Короче говоря:

- RAG находит факты;
- GraphRAG объясняет, как эти факты согласуются.

Архитектура GraphRAG

Последовательность действий

- Запрос → LLM определяет, нужно ли рассуждение на графе.
- LLM генерирует запрос на языке Cypher или SPARQL.
- Графовая база данных выполняет запрос и возвращает структурированные тройки.
- Найденный контекст объединяется с тестовыми фрагментами с помощью поисковика RAG.
- Генератор порождает интегрированный, контекстно осведомленный ответ.



GraphRAG объединяет результаты структурированного и неструктурированного поиска и возвращает семантически обоснованный и объяснимый ответ

5. Пример: реализация гибридного запроса в GraphRAG

```
from neo4j import GraphDatabase
from openai import OpenAI
import chromadb

driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "password"))
client = OpenAI(api_key="your-key")
chroma = chromadb.PersistentClient(path="store")
collection = chroma.get_collection("kb")

def hybrid_query(question):
    # Шаг 1: попросить LLM сгенерировать запрос на Cypher
    cypher_prompt = f"Сгенерируй на Cypher запрос для следующего вопроса:
{question}"
    cypher_query = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role":"user","content":cypher_prompt}],
        temperature=0.2
    ).choices[0].message.content

    # Шаг 2: выполнить запрос к графу
    with driver.session() as session:
        graph_data = [r.data() for r in session.run(cypher_query)]

    # Шаг 3: найти текстовый контекст
    res = collection.query(query_texts=[question], n_results=5, include=
["documents"])
    docs = "\n".join(res["documents"][0])

    # Шаг 4: объединить результаты
    context = f"GRAPH DATA:\n{graph_data}\n\nTEXT DATA:\n{docs}"

    # Шаг 5: сгенерировать обоснованный ответ
    final_prompt = (
        f"Используй следующие структурированные и неструктурированные данные для ответа:\n"
        f"{context}\n"
        f"Question: {question}"
    answer = client.chat.completions.create(
```

```
        model="gpt-4o-mini",
        messages=[{"role":"system","content":"You are a factual assistant."},
                  {"role":"user","content":final_prompt}],
                 temperature=0.3
    ).choices[0].message.content
    return answer
```

Этот гибридный подход позволяет чат-боту использовать хранящиеся в графе связи для рассуждения, а основанный на векторах поиск для обоснования, сочетая фактическую глубину с лингвистической беглостью.

6. Когда использовать Neo4j, а когда ArangoDB

Рекомендация

- Используйте Neo4j, когда важны связи и глубина рассуждения (исследования, аналитика, корпоративные графы знаний).
- Используйте ArangoDB, когда в данных одновременно присутствуют текст, метаданные и связи, обрабатываемые в едином процессе.

7. Преимущества GraphRAG-систем

- Многошаговое рассуждение: позволяет LLM устанавливать логические цепочки фактов.
- Разрешение неоднозначности сущностей: устраняет семантическую неоднозначность данных из различных наборов.
- Объяснимый ИИ: обеспечивает прослеживаемость путей рассуждения.
- Самообновляющиеся знания: автоматически включают новые проверенные связи.
- Унифицированный контекст: сочетает точность поиска с реляционным интеллектом.

8. Проблемы и рекомендации

- Neo4j предлагает отработанные рассуждения на основе связей, что идеально для структурированного интеллекта.
- ArangoDB дает гибкость, необходимую для гибридных моделей данных.
- GraphRAG объединяет структурированный и неструктурированный поиск с целью достижения обоснованных фактами ответов с учетом контекста.
- Гибридная модель преобразует RAG из инструмента поиска во фреймворк рассуждений, понимающий не только что факты связаны, но и почему это так.

RAG ищет истину. Графы объясняют ее. Совместно они ведут к рассуждению, поддающемуся проверке.

4.4. Создание сущностей, связей и контекстуальных путей – семантического каркаса интеллектуальных систем

От единичных данных к соединениям знаний

В RAG-системе информационный поиск управляется семантической близостью – отысканием похожих фрагментов текста.

Но в графе знаний интеллект является порождением явных связей между сущностями.

Чтобы построить систему, способную к рассуждениям, а также пригодную для аудита и контекстуальной навигации, необходимо определить три основополагающих компонента:

- сущности – «вещи», существующие в предметной области;
- связи – как эти вещи связаны между собой;
- контекстуальные пути – маршруты рассуждений, которые приводят к осмысленным соединениям.

Если все сделано правильно, то эта сеть становится семантическим каркасом вашей системы – динамической структурой, которую люди и LLM могут использовать для рассуждений об информации.

1. Сущности: определение вершин графа знаний

Сущность представляет понятие реального мира: человека, предмет, событие, организацию или идею.

Каждая сущность должна иметь:

- уникальный идентификатор (например, ИД или имя);
- информацию о типе (например, Человек, Продукт, Бумага, Проект);
- атрибуты – описывающие ее свойства (например, «год_рождения», «цена», «состояние»).

Совет

Сущности – это имена существительные в вашем графе. Стремитесь к созданию стабильных, допускающих повторное использование вершин, которые со временем будут обрастать связями.

2. Связи: кодирование семантических отношений между сущностями

Связь выражает то, как сущности взаимодействуют или взаимосвязаны. Связи – это глаголы в графе знаний. Каждая связь имеет:

- направление (например, A → B);
- тип (например, «является»);
- факультативные свойства (например, дата, степень уверенности, роль).

Связи могут сцепляться, образуя путь в совокупности знаний, что позволяет проводить рассуждения, выходящие далеко за рамки поиска по ключевым словам.

3. Контекстуальные пути: маршруты рассуждений

Контекстуальный путь – это последовательность связей, соединяющих две или более сущностей осмысленными ассоциациями.

Именно пути определяют действенность графов знаний, наделяя их возможностью многошаговых рассуждений и объяснимостью.

Пример контекстуального пути

Какие современные институты изучают вопросы, относящиеся к открытиям Кюри?

(Кюри) (Радий)
(Радий) – [:ИСПОЛЬЗУЕТСЯ_В] → (Терапия рака)
(Терапия рака) – [:ИЗУЧАЕТСЯ_В] → (Медицинский факультет МТИ)

Эта трехзвенная цепочка образует контекстуальный путь от Марии Кюри к современному институту – простая поисковая система никогда не смогла бы его вывести.

На естественном языке: «На медицинском факультете МТИ изучает методы терапии рака, в которых используется радий, открытый Марией Кюри».

Такой вид объяснимого следа является сутью структурированного рассуждения.

4. Процесс построения графа

Создание сущностей, связей и контекстуальных путей может производиться вручную или автоматически с использованием LLM.

Шаг 1: определить онтологию

- Идентифицировать основные типы сущностей (Человек, Организация, Продукт, Концепт, Событие).
- Определить допустимые типы связей (работает_в, основан, зависит_от, принадлежит_автору).
- Определить свойства (дата, источник, степень_уверенности).

Шаг 2: извлечь факты

Использовать LLM для генерации троек по неструктурированному тексту:

Вход: «В 1898 году Мария и Пьер Кюри открыли радий».

Выходные тройки:

(«Мария Кюри») - [:DISCOVERED {year:1898}] -> («Радий»)
(«Пьер Кюри») - [:DISCOVERED {year:1898}] -> («Радий»).

Шаг 3: проверка и нормализация

- Объединить дубликаты («МТИ» = «Массачусетский технологический институт»).

- Убедиться, что связи согласованы с онтологией.
- Применить оценку степени уверенности к недостоверным фактам.

Шаг 4: сохранение в графовой базе данных

Загрузить данные в графовую базу, например Neo4j или ArangoDB:

```
MERGE (p:Person {name:"Мария Кюри"})
MERGE (e:Element {name:"Радий"})
MERGE (p)-[:DISCOVERED {year:1898}]->(e)
```

Шаг 5: генерация контекстуальных путей

Использовать Cypher для изучения цепочек рассуждений:

```
MATCH (a:Person {name:"Мария Кюри"})-[:DISCOVERED]->(x)-
[:USED_IN]->(y)
RETURN a.name, x.name, y.name
```

Выход:
Мария Кюри → Радий → Терапия рака.

Теперь ваша система может отвечать на вопросы, используя многошаговые объяснимые рассуждения.

6. Автоматизация построения графа с помощью LLM

Современные ИИ-системы могут оказать помощь в автоматическом построении и расширении графов знаний, взяв на себя извлечение сущностей и связей и неструктурированных источников.

Пример технологического процесса

- Разбить текстовые документы на фрагменты с соблюдением семантических границ (шаг RAG).
- Запросить у LLM извлечение троек:

  «Извлеки основанные на фактах связи в формате (субъект, предикат, объект)».
- Проверить извлеченные тройки на согласованность с существующей онтологией.
- Сохранить их в графовой базе данных, возможно, объединив с существующими.

Пример кода на Python

```
from openai import OpenAI
from neo4j import GraphDatabase

client = OpenAI(api_key="your_key")
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j","password"))

def extract_triples(text):
    prompt = f"Извлеки фактические триплеты из следующего
        текста:\n{text}\nВыведи в виде JSON-списка."
    res = client.chat.completions.create(model="gpt-4o-mini",
```

```
    messages=[{"role":"user","content":prompt}], temperature=0.2)
    return res.choices[0].message.content

def insert_triple(tx, s, p, o):
    tx.run("MERGE (a:Entity {name:$s}) MERGE (b:Entity {name:$o})
MERGE (a)-[r:REL {type:$p}]->(b)", s=s, p=p, o=o)
    text = "Tesla Motors была основана Илоном Маском в 2003 году."
    triples = extract_triples(text)
    with driver.session() as session:
        for t in triples:
            session.write_transaction(insert_triple, t['subject'], t['predicate'],
                                      t['object'])
```

Этот процесс позволяет автоматизировать заполнение графа знаний, т. е. создать структурированный интеллект из текста на естественном языке.

7. Контекстуальные пути в системах RAG + граф

Будучи включены в конвейер RAG, контекстуальные пути улучшают как поиск, так и рассуждение:

- поисковик выбирает документы, в которых упоминаются релевантные сущности;
- графовая база данных определяет, как эти сущности связаны;
- LLM использует эту связь, чтобы генерировать обоснованные, объяснимые ответы.

Пример сценария использования

Вопрос: «Какие основные технологии стали результатом исследований Алана Тьюринга?»

Путь в графе
(Алан Тьюринг) — [:INFLUENCED] → (Теория вычислимости) —
[:INSPIRED] → (Современный ИИ) — [:IMPLEMENTED_IN] → (Нейронные сети)

LLM объясняет

«Ранние работы Тьюринга по теории вычислимости заложили фундамент ИИ и послужили источником вдохновения для применяемых сегодня архитектур нейронных сетей».

8. Принципы проектирования, ведущие к эффективному моделированию графов

- Думайте семантически, а не структурно.

  Моделируйте смысл, а не просто формат данных. Улавливайте связи, которые объясняют, почему сущности важны.
- Определяйте связи, допускающие действие.

  Каждое ребро должно поддерживать вопрос, на который, по вашему мнению, система должна уметь отвечать.

- Используйте типизированные сущности.

  Помечайте вершины семантически значимыми типами (например, :Человек, :Компания, :Идея), чтобы поддержать предметно-ориентированное рассуждение.
- Избегайте чрезмерного количества связей.

  Не каждая пара сущностей нуждается в соединении. Обращайте внимание на релевантность и объяснимость.
- Не забывайте о провенансе.

  Всегда сохраняйте источник (документ или URL-адрес) каждого факта, чтобы рассуждения были прозрачны.

9. Пример: запрос контекстуального пути в Neo4j

Цель: найти все организации, связанные с исследованиями по «квантовым вычислениям», финансируемым «NSF» (Национальным научным фондом).

```
MATCH (f:Organization {name:"NSF"})-[:FUNDED]->(p:Project)-
[:FOCUSES_ON]->(r:ResearchArea {name:"Квантовые вычисления"})
MATCH (p)<-[:WORKS_ON]-(o:Organization)
RETURN DISTINCT o.name AS Organization
```

Результат:

IBM Research
MIT Lincoln Lab
Stanford Quantum Group

Этот простой запрос вскрывает глубокие семантические связи, охватывающие несколько документов и типов данных.

- Сущности – это ваши имена существительные, они определяют основные объекты знаний.
- Связи – это ваши глаголы, они улавливают, как эти объекты связаны.
- Контекстуальные пути – это ваша логика, способы рассуждений, практикуемые вашим ИИ.
- В сочетании они превращают статические факты в объяснимый интеллект.

RAG-система извлекает знания.
Граф знаний соединяет их.
Контекстуальные пути делают их доступными пониманию.

4.5. Запросы на языках Cypher и SPARQL

Язык знаний: как ИИ-агенты задают вопросы графам. Роль языков запросов в ИИ-системах

Граф знаний полезен лишь настолько, насколько полезны вопросы, которые ему можно задавать.

Если конвейер RAG находит релевантный текст с помощью погружений, то запрос к графу находит семантически значимые связи с помощью явной логики.

В современных графовых системах доминируют два языка запросов:

- Cypher – применяется в Neo4j и других графовых базах данных со свойствами;
- SPARQL – применяется в графах на основе RDF (типа Wikidata и систем, управляемых онтологиями).

Оба служат языками общения между уровнем рассуждений вашей LLM и необходимыми ей структурированными знаниями.

Посмотрим, как эти языки работают, как LLM могут динамически генерировать запросы на них и как они способствуют гибридным рассуждениям.

1. Почему языки запросов важны

В чистой RAG-системе ИИ-агент ищет фрагменты неструктурированного текста и применяет к ним неформальные рассуждения.

При наличии графа знаний агент может пойти дальше – задавать конкретные символические вопросы:

- «Кто автор этой статьи?»;
- «Какие проекты связаны общими источниками финансирования?»;
- «Покажи мне все молекулы, нацеленные на тот же белок, что и препарат X».

Такие языки запросов, как Cypher и SPARQL, делают эти вопросы машинно-выполнимыми. Они дают LLM возможность транслировать язык в логику – важный шаг на пути к рассуждениям, допускающим проверку.

2. Cypher: язык запросов Neo4j

Cypher – декларативный язык запросов, спроектированный для графов со свойствами, в которых вершины и ребра могут быть снабжены атрибутами.

Базовый синтаксис

Язык Cypher основан на образцах и является удобочитаемым для человека, что делает его пригодным равно для разработчиков и LLM.

Пример запроса по образцу

```
MATCH (a:Person)-[:AUTHORED]->(b:Paper)
WHERE b.year > 2020
RETURN a.name, b.title
```

Этот запрос находит людей, являющихся авторами статей, опубликованных после 2020 года.

Пример 1. Нахождение связей

Вопрос: «Какие исследователи сотрудничали с Алисой в любом из проектов по ИИ?»

Cypher:

```
MATCH (a:Person {name:"Alice"})-[:WORKED_ON]->(p:Project)<-
[:WORKED_ON]-(collaborator)
RETURN DISTINCT collaborator.name
```

Этот запрос обходит проекты в поисках сотрудников.

Пример 2. Многошаговое рассуждение

Вопрос: «Какие университеты ассоциированы с компаниями, работающими над квантовыми вычислениями?»

Cypher:

```
MATCH (u:University)<-[:AFFILIATED_WITH]-(r:Researcher)-
[:WORKS_AT]->(c:Company)
WHERE c.field = "Quantum Computing"
RETURN DISTINCT u.name, c.name
```

Это многошаговое рассуждение – одно из ключевых преимуществ запросов к графам. Традиционные RAG-системы на него не способны без структурированного соединения.

3. SPARQL: запросы к RDF-графам

SPARQL (произносится «спаркл») – стандарт W3C для формулирования запросов к RDF-графам, в которых данные хранятся как тройки субъект–предикат–объект.

Обычно он используется в семантической веб, связанных данных и системах, основанных на онтологиях типа Wikidata, DBpedia или графах биомедицинских знаний.

Структура запроса на SPARQL

```
PREFIX ex:
  SELECT ?person ?paper
  WHERE {
    ?person a ex:Researcher .
    ?person ex:authored ?paper .
    ?paper ex:year ?year .
    FILTER(?year > 2020)
  }
```

SPARQL рассматривает это как подлежащую заполнению логическую форму – какие сущности отвечают такому образцу связей?

Пример 1. Сцепленные запросы

Вопрос: «Перечислить лекарственные препараты, которые ингибируют белки, ассоциированные с болезнью Альцгеймера».

```
PREFIX ex:
SELECT ?drug ?protein
WHERE {
  ?disease a ex:Disease ;
  ex:name "Alzheimer's Disease" .
  ?protein ex:associated_with ?disease .
  ?drug ex:inhibits ?protein .
}
```

Этот запрос выражает трехшаговую цепочку рассуждений: соединение заболеваний, белков и препаратов с помощью явных связей.

Пример 2. Рассуждение с онтологиями

SPARQL может также использовать онтологические иерархии с помощью `rdfs:subClassOf` или `owl:sameAs`, что позволяет выполнять более глубокие рассуждения.

```
PREFIX rdfs:
SELECT ?substance
WHERE {
  ?substance rdfs:subClassOf ex:ChemicalSubstance .
}
```

В ответ возвращаются все категории, классифицированные как химические вещества или их подклассы, что поддерживает семантический вывод в иерархии.

4. Сравнение Cypher и SPARQL

Рекомендация

Используйте Cypher, когда граф прагматический и операционный (например, в корпоративных, исследовательских или связанных с конкретным продуктом системах).

Используйте SPARQL, когда граф семантический и ориентированный на знания (например, онтологии, научные данные или связанные открытые данные).

5. Как LLM могут автоматически генерировать запросы

Современные LLM умеют динамически транслировать вопросы на естественном языке на Cypher или SPARQL.

Пример запроса для LLM

Преобразовать следующий вопрос пользователя в запрос Neo4j на языке Cypher: «Какие исследователи из МТИ публиковали статьи по нейронным сетям после 2018 года?»

Выход LLM

```
MATCH (r:Researcher)-[:AFFILIATED_WITH]->(i:Institution
{name:"MIT"})
MATCH (r)-[:AUTHORED]->(p:Paper)
WHERE p.topic CONTAINS "neural networks" AND p.year > 2018
RETURN DISTINCT r.name, p.title
```

В этом случае понимание естественного языка трансформируется в инструкции для структурированного рассуждения, что является критическим компонентом интеграции LLM с графовой базой данных (GraphRAG).

6. Интеграция запросов в технологический процесс RAG

Выполнение запросов на языке Cypher или SPARQL можно включить в гибридный конвейер:

- LLM определяет намерение и генерирует запрос к графу;
- графовая база данных выполняет запрос и возвращает связи между фактами;

- поисковик находит релевантный неструктурированный контекст (документы, их фрагменты);
- LLM-генератор составляет окончательный обоснованный ответ на естественном языке.

Пример заготовки интеграции (Python):

```
def hybrid_graph_query(question):
    cypher_query = llm_generate_cypher(question)
    graph_results = run_neo4j(cypher_query)
    text_context = retriever.retrieve(question)
    final_prompt = f"Объедини графовые данные с текстовыми фактами ниже и
дай ответ.\nGRAPH: {graph_results}\nTEXT: {text_context}"
    return llm_generate(final_prompt)
```

Эта структура обеспечивает как символическую точность, так и семантическую глубину – отличительный знак рассуждения современного ИИ.

7. Пример в действии: гибридная генерация ответа

Пользователь: «Какие исследователи в области ИИ сотрудничали со Стэнфордом и имеют статьи по обучению с подкреплением?»

Шаги системы

LLM генерирует запрос на языке Cypher:

```
MATCH (r:Researcher)-[:AFFILIATED_WITH]->(i:Institution
{name:"Стэнфорд"})
MATCH (r)-[:AUTHORED]->(p:Paper)
WHERE p.topic CONTAINS "обучение с подкреплением"
RETURN DISTINCT r.name, p.title
```

- В графе найдены результаты.
- Поисковик RAG выбирает рефераты последних публикаций.
- LLM-генератор объединяет то и другое в ответ: «Исследователи из Стэнфорда, такие как д-р Ли, д-р Ю и д-р Эрнандес, недавно публиковали статьи по обучению с подкреплением, обращая особое внимание на многоагентные среды и моделирование вознаграждения».

Каждый элемент ответа можно проследить до структурированных данных в графе и найденных документов.

- Cypher используется в графах со свойствами (Neo4j, Memgraph).
- SPARQL используется в графах RDF и семантических графах (Wikidata, GraphDB).
- Тот и другой позволяют LLM задавать точные, логически обоснованные вопросы.
- При интеграции в конвейер RAG они перекидывают мост между языком и логикой.
- Всегда проверяйте запросы на точность, объяснимость и соответствие требованиям регулятора и протоколируйте результаты.

Поиск находит информацию.

Запрос к графу выявляет связи.

В сочетании они являются основой ИИ-агентов, которые «думают» в терминах связей, а не ключевых слов.

4.6. Агент в действии: соединение LLM с основанным на графах конвейером рассуждений

До сих пор мы изучали, как графы знаний организуют факты и как их можно опрашивать с помощью языков Cypher или SPARQL.

Настало время практики – построить ИИ-агент, который использует как естественно-языковой интеллект LLM, так и структурированные рассуждения на графах.

В этом разделе мы построим подкрепленный графом рассуждающий агент, умеющий:

- понимать вопросы на естественном языке;
- транслировать их в запросы к структурированному графу;
- выполнять эти запросы в Neo4j (или другой графовой базе данных);
- объединять структурированные результаты с неструктурированным контентом с помощью RAG;
- порождать объяснимый обоснованный ответ с прослеживаемым рассуждением.

Этот гибридный дизайн лежит в основе корпоративных ассистентов следующего поколения – инструментов, которые ищут информацию, рассуждают и проверяют ответ перед выдачей пользователю.

1. Обзор архитектуры

Подкрепленный графом рассуждающий агент состоит из пяти уровней:

- уровень интерфейса принимает вопросы от пользователя (в диалоге, с помощью API или в командной строке);
- интерпретатор намерений (LLM) определяет, какого рода информация запрашивается;
- генератор запросов преобразует вопрос на естественном языке в запрос на языке Cypher или SPARQL;
- движок выполнения запросов к графу находит связи и сущности;
- уровень синтеза снова прибегает к помощи LLM, чтобы объединить структурированный и неструктурированный контекст в связный ответ, основанный на фактах.

2. Предварительные условия и подготовка

Мы будем использовать:

- Neo4j для хранения графа и рассуждений о нем;
- OpenAI GPT-4o-mini для рассуждений на уровне LLM;

- LangChain или свой собственный код на Python для оркестрации;
- ChromaDB для поиска в неструктурированных документах (факультативно).

Зависимости

```
pip install neo4j openai langchain chromadb python-dotenv
```

Переменные среды

```
OPENAI_API_KEY=your-key
NEO4J_URI=bolt://localhost:7687
NEO4J_USER=neo4j
NEO4J_PASSWORD=your-password
```

3. Построение базовых компонентов

Разобьем последовательность действий на шаги.

Шаг 1 – коннектор с графовой базой данных

Сначала напишем простой коннектор с нашей графовой базой данных.

```
# graph_connector.py
from neo4j import GraphDatabase
import os

class GraphConnector:
    def __init__(self):
    self.driver = GraphDatabase.driver(
        os.getenv("NEO4J_URI"),
        auth=(os.getenv("NEO4J_USER"),
        os.getenv("NEO4J_PASSWORD"))
    )

    def run_query(self, query):
        with self.driver.session() as session:
            result = session.run(query)
            return [record.data() for record in result]

    def close(self):
        self.driver.close()
```

Эта абстракция позволяет LLM взаимодействовать с графом не напрямую – безопасно и единообразно.

Шаг 2 – LLM-генератор запросов

Роль этой LLM состоит в том, чтобы интерпретировать вопрос и транслировать его в запрос на языке Cypher.

```
# llm_query_gen.py
from openai import OpenAI

client = OpenAI()
```

```
def generate_cypher(question, schema_hint=None):
    system_prompt = """
    Ты генератор Cypher-запросов для графа знаний в Neo4j.
    Транслируй вопрос пользователя в корректный Cypher-запрос.
    Верни только код на Cypher, без пояснений.
    Если не уверена, попроси уточнить.
    Указание по схеме:
    - Вершины: Person, Organization, Project, ResearchField
    - Связи: WORKS_AT, AUTHORED, FUNDED, COLLABORATES_WITH, STUDIES
"""
    user_prompt = f"User Question: {question}"
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.2
    )
    return response.choices[0].message.content.strip()
```

Пример входа и выхода

Вход: «Какие исследователи из МТИ работали в области квантовых вычислений?»

Выход:

```
MATCH (r:Researcher)-[:AFFILIATED_WITH]->(i:Institution
{name:"MIT"})
MATCH (r)-[:WORKED_ON]->(p:Project {field:"Quantum
Computing"})
RETURN r.name, p.name
```

Шаг 3 – выполнение запроса к графу

```
# graph_agent.py
from graph_connector import GraphConnector
from llm_query_gen import generate_cypher
from openai import OpenAI

client = OpenAI()

def answer_question(question):
    graph = GraphConnector()
    cypher_query = generate_cypher(question)
    graph_results = graph.run_query(cypher_query)
    graph.close()
    formatted_data = "\n".join(str(r) for r in graph_results)
    synthesis_prompt = f"""
Ты ассистент по работе с фактами. Используй структурированные результаты,
полученные от графа, чтобы ответить на вопрос пользователя кратко и ясно.
Если результатов не получено, скажи 'В графе не найдено подходящих данных.'
Результаты, полученные от графа:
{formatted_data}
"""
```

```
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a reasoning assistant."},
            {"role": "user", "content": synthesis_prompt}
        ],
    temperature=0.3
    )
    return response.choices[0].message.content.strip()
```

Выполнение:

```
print(answer_question("Перечисли проекты в области ИИ, финансируемые NSF."))
```

Ожидаемый результат

Национальный научный фонд (NSF) финансировал такие проекты в области ИИ, как VisionNet, NeuroGrid и Learning Systems 3.0.

4. Факультативно: добавить RAG для поиска контекста

Графы улавливают структуру, но иногда вам необходим описательный контекст (статьи, рефераты, отчеты).

Ниже показано, как объединить то и другое, воспользовавшись гибридным поиском.

```
from chromadb import PersistentClient

def retrieve_context(question, k=3):
    chroma = PersistentClient(path="store")
    collection = chroma.get_collection("kb")
    res = collection.query(query_texts=[question], n_results=k, include=
["documents"])
    return "\n".join(res["documents"][0])

def hybrid_answer(question):
    graph_data = answer_question(question)
    text_context = retrieve_context(question)
    final_prompt = f"""
Объедини следующие структурированные данные из графа с текстовым контекстом,
чтобы дать обоснованный фактами ответ:
GRAPH DATA:
{graph_data}
TEXT CONTEXT:
{text_context}
"""
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a research assistant."},
            {"role": "user", "content": final_prompt}
        ],
        temperature=0.3
    )
    return response.choices[0].message.content.strip()
```

Теперь ваш агент может дать обоснованный фактами ответ, подтвержденный источниками.

5. Пример взаимодействия

Пользователь: «Кто в Стэнфорде занимается исследованиями в области обучения с подкреплением и сотрудничает с DeepMind?»

Последовательность работы агента

- LLM генерирует Cypher-запрос:

```
MATCH (r:Researcher)-[:AFFILIATED_WITH]->(i:Institution
{name:"Stanford"})
MATCH (r)-[:COLLABORATES_WITH]->(c:Organization
{name:"DeepMind"})
MATCH (r)-[:STUDIES]->(f:Field {name:"Reinforcement Learning"})
RETURN DISTINCT r.name, f.name
```

- Neo4j исполняет этот запрос.
- Найденные вершины: д-р Лин, д-р Эррера, д-р О'Нил.
- RAG находит рефераты последних статей по сотрудничеству в области обучения с подкреплением.
- LLM синтезирует окончательный ответ.

  Исследователи д-р Лин, д-р Эррера и д-р О'Нил из Стэнфорда возглавляют проекты в области обучения с подкреплением в сотрудничестве с DeepMind, уделяя особое внимание координации нескольких агентов и оптимизации вознаграждения.

Этот ответ объясним, допускает аудит и основан на структурированных данных и найденных фактах.

6. Развитие агента

Вы можете дополнить конвейер, включив в него:

- выбор инструментария – динамически выбирать между RAG и запросами к графу в зависимости от намерений пользователя;
- обновление графа – извлекать новые тройки из найденного текста и автоматически включать их в базу данных Neo4j (если соблюдены пороговые значения уверенности);
- граф в памяти – сохранять взаимодействия в виде новых связей ((Пользователь) -> [:СПРАШИВАЛ_О] -> (Тема)), что позволило бы агенту вспоминать контекст диалога;
- визуализацию – использовать Neo4j Bloom или NetworkX для интерактивного показа путей рассуждения.

8. Заключительные замечания об архитектуре

- Соединение LLM с графовыми базами данных порождает рассуждения, а не простые поисковые системы.

- Cypher или SPARQL служат логическим уровнем, который открывает путь к объяснимому и допускающему аудит интеллекту.
- Объединение графов с RAG наделяет агента семантической глубиной и фактической обоснованностью.
- Результатом является гибридный ИИ, способный проследить свою логику: вопрос → связи → факты → ответ.

LLM наделяют машины языком.
Графы знаний наделяют их логикой.
Вместе они порождают понимание.

4.7. На практике: графы знаний в фармацевтике, финтехе и академических учреждениях

Структурированный интеллект обеспечивает открытия, соответствие требованиям регуляторов и проникновение в суть вещей.

От теории к результатам

До сих пор мы строили графы знаний, соединяли их с LLM и исследовали их роль в конвейерах рассуждения. Но как все это выглядит на практике?

Организации из разных отраслей обращаются к графам знаний, чтобы унифицировать разрозненные данные, уменьшить неоднозначность и внедрить автоматизацию на основе рассуждений.

Будь то фармацевтическая компания, которая связывает гены с заболеваниями, фирма в области финтеха, стремящаяся обнаружить мошенничество, или университет, являющийся частью глобальной сети сотрудничества, – графы начинают образовывать становой хребет интеллектуальных систем.

В этом последнем разделе мы посмотрим, как три крупные отрасли – фармацевтика, финтех и академические учреждения – используют подкрепленный графами ИИ для решения ответственных задач, требующих точности и контекста.

1. Графы знаний в фармацевтике и науках о жизни

Проблема

Фармацевтическим компаниям приходится иметь дело с океанами данных: клинические испытания, библиотеки соединений, генно-белковые взаимодействия, реестры побочных эффектов, нормативно-правовые требования и научные публикации.

Традиционные реляционные системы не могут представить сложные связи между этими сущностями – взаимодействие лекарственного препарата с белком является частью метаболического пути, связанного с заболеванием, которое имеет генетические разновидности в разных популяциях.

Решение: графы биомедицинских знаний

Граф биомедицинских знаний связывает сущности, например:

- лекарственные препараты → нацелены_на → белки;
- белки → участвуют_в → метаболические пути;
- метаболические пути → связаны_с → заболевания;
- заболевания → лечатся → лекарственные препараты.

Эта реляционная модель обеспечивает многошаговое рассуждения: «Какие из существующих лекарственных препаратов нацелены на белки, ассоциированные с ранней стадией болезни Альцгеймера?»

Архитектура в действии

1. Источники данных: PubMed, DrugBank, Gene Ontology, ClinicalTrials.gov.
2. Графовая база данных: Neo4j или AWS Neptune.
3. Конвейер:
   - подгрузить данные в вершины: Лекарственный препарат, Белок, Заболевание, Метаболический путь;
   - определить связи (TREATS, TARGETS, ASSOCIATED_WITH);
   - соединить LLM с запросом к графу на языке Cypher или SPARQL.

Пример запроса

```
MATCH (d:Drug)-[:TARGETS]->(p:Protein)-[:INVOLVED_IN]->
(pw:Pathway)-[:LINKED_TO]->(ds:Disease {name:"Alzheimer's
Disease"})
RETURN DISTINCT d.name, p.name, pw.name
```

Результаты

- Позволяет перепрофилировать препараты благодаря нахождению скрытых связей между биологическими данными.
- Сокращает циклы исследований и разработок благодаря выявлению связей между существующими соединениями и непроверенными метаболическими путями.
- Способствует внедрению объяснимого ИИ в нормативно-правовую документацию, предоставляя прослеживаемые пути рассуждений.

Реальный пример

Фармацевтическая компания, входящая в десятку крупнейших, использовала ИИ-конвейер на основе графов для предложения новых гипотез о мишенях для лекарств от нейродегенеративных заболеваний. Это позволило сократить время на ранних этапах исследований на 45 % и повысить воспроизводимость результатов за счет прозрачных запросов к графам.

2. Графы знаний в финтехе и банковской индустрии

Проблема

Современные финансовые экосистемы состоят из сложных взаимосвязанных объектов: клиентов, продавцов, счетов, транзакций, IP-адресов и устройств.

Обнаружение мошенничества, соблюдение нормативно-правовых требований и моделирование кредитных рисков зависят не только от того, что происходит, но и от того, кто с кем связан и каким образом.

Традиционные табличные системы могут выявлять аномалии в транзакциях, но не способны обнаружить реляционное мошенничество, когда несколько действий с низким уровнем риска объединяются в сеть с высоким уровнем риска.

Решение: графы финансовых связей.

- Клиент → владеет → Счет.
- Счет → переводит_на → Счет.
- Счет → связан_с → Устройство/IP-адрес.
- Клиент → ассоциирован_с → Юридическое лицо.

Этот граф позволяет проследить движение денежных средств и выявить связи.

Пример запроса

Найти все счета, косвенно связанные с подсанкционной организацией более чем через двух посредников.

```
MATCH path = (c1:Customer {status:"sanctioned"})-
[:OWNS|:TRANSFERRED_TO*1..3]-(c2:Customer)
RETURN DISTINCT c2.name, length(path) AS connection_depth
ORDER BY connection_depth ASC
```

Применения

- Обнаружение мошеннических действий.

  Выявление групп злоумышленников или мошеннических схем, которые ускользают от внимания традиционных систем, основанных на правилах.
- Противодействие отмыванию доходов.

  Построение карты многозвенных связей между счетами и организациями для генерации уведомлений на основе шаблонов.
- Кредитный скоринг.

  Анализ связей между физическими и юридическими лицами (например, общие работодатели, поручители) для повышения уровня доверия.
- Соблюдение нормативных требований.

  Обеспечение возможности аудита с объяснением для каждого помеченного пути выполнения транзакции.

Интеграция с LLM

Благодаря подключению LLM к графу финансовых знаний:

- специалисты по комплаенсу могут задавать вопросы на естественном языке: «Покажи мне все счета, с которых средства были переведены в регион с высоким уровнем риска через подставные компании»;
- LLM преобразует вопрос в запрос на языке Cypher и находит допускающие проверку пути в графе;
- результаты обобщаются с указанием полного провенанса, что обеспечивает объяснимость человеку, включенному в процесс.

Реальный пример

Европейская компания в области финтеха внедрила основанный на Neo4j граф знаний, интегрированный с GPT, для автоматизации антиотмывочных расследований.

Аналитики сократили время рассмотрения дел на 60 %, сохранив при этом прослеживаемые, допускающие аудит логические пути.

3. Графы знаний в академических и научно-исследовательских учреждениях

Проблема

Научные знания разбросаны по разным статьям, цитатам, институтам, проектам и дисциплинам.

Поисковые системы находят документы, но не понимают связей между исследователями, методологиями и новыми тематиками.

Решение: графы научных знаний

Граф научных знаний определяет следующие связи:

- Автор → аффилирован_с → Учреждение;
- Автор → опубликовал → Статья;
- Статье → ссылается_на → Статья;
- Статья → относится_к → Область/Тема.

Это открывает возможность для многошагового рассуждения: «Какие учреждения являются наиболее авторитетными в области обучения с подкреплением, начиная с 2020 года?»

Пример запроса на Cypher

```
MATCH (a:Author)-[:AUTHORED]->(p:Paper)-[:BELONGS_TO]->
(f:Field {name:"Reinforcement Learning"})
MATCH (a)-[:AFFILIATED_WITH]->(i:Institution)
WHERE p.year >= 2020
RETURN i.name AS institution, COUNT(DISTINCT p) AS papers
ORDER BY papers DESC
LIMIT 10
```

Преимущества

- Позволяет анализировать тенденции в разных дисциплинах и за разные годы.
- Строит карту сетей сотрудничества и выявляет междисциплинарные исследования.
- Облегчает автоматизированный анализ литературы для ИИ-агентов с помощью RAG в сочетании с рассуждениями на графах.
- Выявляет зарождающиеся исследовательские кластеры еще до того, как они будут отражены в традиционных метриках.

Реальный пример

Академический консорциум создал исследовательский граф, объединяющий данные Scopus, ORCID и arXiv, а затем надстроил над ним RAG-интерфейс на основе LLM.

Исследователи получили возможность задавать вопросы вида ««Кто является ведущими авторами в области квантовых вычислений и криптографии?» и получать список имен и путь рассуждений в обоснование их связи.

4. Зарождающиеся тенденции и перспективы на будущее

- Графовые нейронные сети (ГНС).

  Сочетание символических графов с нейросетевым обучением для прогнозирования взаимосвязей и встраивания данных непосредственно из структуры.
- Масштабирование GraphRAG.

  Конвейеры RAG нового поколения объединяют векторные хранилища и графовые запросы в одном унифицированном поиске.
- Автоматическое извлечение знаний.

  LLM непрерывно анализируют текстовые потоки (научные статьи, документы, отчеты) и самостоятельно предлагают обновления графов.
- Стандартизация и интероперабельность.

  Отрасль переходит на открытые схемы (такие как BioKG, FinGraph и OpenAlex) для обмена междисциплинарными графами.
- Объяснимый корпоративный ИИ.

  Соблюдение нормативных требований (особенно в сфере здравоохранения и финансов) все чаще требует наличия основанного на графах провенанса для каждого автоматизированного решения.

5. Общее направление

Будь то выявление случаев мошенничества с лекарствами или научные инновации, графы знаний служат одной и той же цели:

- они преобразуют «сырую» информацию в структурированный вид, более пригодный для понимания;
- RAG находит релевантные данные;
- графы соединяют релевантные данные;
- LLM синтезируют ответ с учетом контекста.

Будущее ИИ – не за большими моделями, а за лучшей проработкой связей между данными.

6. Основные выводы

- Фармацевтика: используйте графы, чтобы связать биологические, химические и клинические данные для ускорения научных открытий.

- Финтех: используйте графы для моделирования связей с целью выявления мошенничества и обеспечения прозрачного комплаенса.
- Академические учреждения: используйте графы для связывания исследований, авторов и дисциплин с целью выявить скрытые тенденции.
- Во всех областях Graph + RAG + LLM = объяснимый, управляемый рассуждениями ИИ.

7. Заключительные мысли

Графы знаний не базы данных, а карты смысловых значений.

Они позволяют машинам мыслить реляционно, а не только статистически.

В сочетании с языковыми моделями они образуют основу новой парадигмы искусственного интеллекта, в которой понимание заменяет запоминание, а рассуждение приходит на смену предсказанию.

Данные говорят, что произошло.

Графы знаний говорят, почему это произошло.

Глава 5

Когнитивные циклы: разум агента

5.1. Как агенты планируют, рассуждают и осмысляют

Концепция когнитивных циклов – Планирование → Действие → Осмысление → Пересмотр. От реактивных моделей к рефлексивным агентам

Большие языковые модели (LLM) отлично справляются с генерацией ответов, но они не умеют осмыслять свои решения. Как только ответ получен, рассуждение завершено. Система не анализирует, что пошло не так, что можно улучшить и как спланировать следующий шаг.

Однако человеческое мышление устроено иначе.

Мы планируем, прежде чем действовать, наблюдаем результаты, осмысляем их качество и уточняем будущие стратегии.

Чтобы перейти от реактивных результатов к интеллектуальному, адаптивному поведению, ИИ-системы должны использовать такую же схему – когнитивный цикл.

Именно этот цикл – Планирование → Действие → Осмысление → Пересмотр – превращает систему на основе LLM в настоящего агента.

1. Что такое когнитивный цикл?

Когнитивный цикл – это структурированный процесс, который позволяет агенту непрерывно рассуждать о своем окружении, оценивать свои действия и адаптироваться с течением времени.

Можете считать, что это мыслительный пульс автономной системы.

Каждое биение пульса позволяет агенту:

- сформировать намерение (Планирование);
- выполнить действие (Действие);
- проанализировать результат (Осмысление);
- скорректировать свою стратегию (Пересмотр).

Тем самым создается самоулучшающийся цикл с обратной связью, что сильно напоминает то, как люди учатся на опыте.

2. Четыре этапа когнитивного цикла

A. Планирование – определение намерений и стратегии

Агент начинает с анализа цели и составления плана ее достижения. Он может включать:

- разбиение целей на более мелкие подзадачи;
- выбор подходящих инструментов или API;
- прогнозирование возможных результатов или трудностей.

Пример

ИИ-ассистент по исследованиям получает задание: «Обобщи все последние статьи на тему обучения с подкреплением с сайта arXiv».

План

- Найти релевантные статьи.
- Извлечь из них рефераты.
- Обобщить основные методы и результаты.
- Поместить результаты обобщения в базу знаний.

Планирование превращает расплывчатые цели в исполнимые последовательности действий, что является основой рассуждения.

B. Действие – выполнение с учетом контекста и контроль

Агент выполняет одно или несколько действий из своего плана:

- запрос API;
- выполнение фрагмента кода;
- извлечение документов;
- генерирование отчета или сводки.

Каждое действие должно быть запротоколировано и наблюдаемо, чтобы система могла понять, что она сделала и что изменилось.

Пример

Исследовательский агент использует API для получения данных из набора arXiv, фильтрует их по запросу «обучение с подкреплением» и обобщает рефераты с помощью своей LLM.

C. Осмысление – анализ результатов

После выполнения действия агент должен приостановиться и приступить к анализу:

- отвечает ли результат поставленной цели?
- имели ли место ошибки, галлюцинации или нерелевантные результаты?
- насколько система уверена в своем ответе?

Осмысление – то место, где большинство современных систем терпят неудачу, потому что требуется рассуждение о своем собственном рассуждении.

Агенты на основе LLM реализуют осмысление, выполняя вспомогательный проход рассуждения о полученном результате.

Пример

Агент анализирует составленные им же обобщения и спрашивает себя: «Уловил ли я основные тенденции или упустил из виду такие новые области исследований, как офлайновое обучение с подкреплением (ОП) или ОП на базе модели?»

Этот шаг осмысления значительно повышает согласованность с фактами и качество проникновения в суть дела.

D. Пересмотр – адаптация стратегии

После того как в процессе осмысления были выявлены слабые места, агент переходит к пересмотру:

- корректирует свой план;
- изменяет параметры (например, температуру или глубину поиска);
- повторяет неудачные действия, применив новую логику;
- обновляет память или хранилища обратной связи.

Этап пересмотра завершает цикл, гарантируя непрерывное улучшение.

Пример

Ассистент по исследованиям осознает, что пропустил некоторые статьи 2025 года из-за фильтрации по датам.

Он изменяет запрос, перезапускает поиск и уточняет обобщение, получая тем самым более полный ответ.



Когнитивный цикл преобразует статические ответы в адаптивный интеллект с помощью непрерывного рассуждения

3. Почему когнитивные циклы важны?

Без когнитивного цикла LLM можно уподобить калькулятору: она обрабатывает вход, генерирует выход и останавливается.

С когнитивным циклом она становится учащимся – способным к повторениям, анализу и совершенствованию.

Когнитивные циклы – то, что отличает выполнение задачи от ее понимания.

4. Когнитивные циклы в современных агентных фреймворках

Каждый фреймворк реализует когнитивный цикл по-разному, но принцип остается неизменным: Планирование → Действие → Осмысление → Пересмотр, пока не будут удовлетворены критерии успеха.

5. Пример: когнитивный цикл на псевдокоде

```
def
    plan = llm.generate_plan(goal)
    while not goal_achieved(plan):
        result = execute(plan)
        reflection = llm.reflect_on(result)
        plan = llm.revise_plan(plan, reflection)
    return summarize(result)
```

Здесь LLM выступает в двух ролях: составителя плана и критика, так что образовывается рекурсивный цикл самосовершенствования.

Каждая итерация делает агента ближе к точности, полноте и связности изложения.

6. Пример в действии – исследовательский обобщающий агент

Цель: «Составь отчет, обобщающий тенденции в области исследований по графовым нейронным сетям (ГНС)».

Выполнение цикла

План: найти 10 лучших статей за 2024–2025 годы.

Действие: извлечь рефераты и обобщить их.

Осмысление: выявить пропущенные темы (например, методы обеспечения интерпретируемости).

Пересмотр: изменить запрос, уделить внимание интерпретируемости и заново обобщить.

Результат: полный отчет с учетом контекста, обоснованный структурированными данными и уточненный благодаря самоанализу.

7. Эволюция интеллекта

Люди становятся умными не потому, что никогда не совершают ошибок, а потому что замечают, анализируют и исправляют их.

Когнитивные циклы привносят эту суть в работу ИИ-агентов.

Интеллект заключается не в том, чтобы знать все, а в том, чтобы знать, как улучшить то, что ты знаешь.

Включая циклы планирования, действия, осмысления и пересмотра, мы наделяем машины не только способностью действовать, но и потенциалом для развития.

- Когнитивные циклы позволяют агентам быть осведомленными о собственной эффективности.
- Каждый этап – планирование, действие, осмысление, пересмотр – добавляет еще один уровень адаптивности.
- Цикл превращает статические LLM в динамические рассуждающие системы.
- Осмысление обеспечивает согласованность с фактами; пересмотр способствует прогрессу.
- Это основа автономного познания в ИИ.

5.2. Системы памяти: кратковременная и долговременная

С течением времени агенты запоминают, забывают и учатся.

От однократных запросов к анализу состояния

Ранее мы рассматривали когнитивные циклы – непрерывный цикл планирования, действий, осмысления и пересмотра, который позволяет агенту динамично адаптироваться.

Но никакие рассуждения не приведут к успеху, если агент забывает обо всем между этапами.

Настоящий интеллект требует памяти – способности сохранять, извлекать и повторно использовать контекст в течение длительного времени.

В этом разделе мы рассмотрим, как ИИ-агенты моделируют познание с помощью систем кратковременной и долговременной памяти, как эти виды памяти структурированы и как они работают совместно для создания последовательных, эволюционирующих рассуждений.

1. Почему память важна

Статическая LLM начинает с чистого листа при каждом запросе, она не помнит, что было раньше.

Для простых вопросно-ответных задач такая стратегия годится, но терпит неудачу в любом сценарии, требующем непрерывности, например:

- многошаговое рассуждение;
- отслеживание проекта;
- контекстно-зависимые диалоги;
- обучение на предыдущих результатах.

Наличие памяти превращает LLM из чат-бота в когнитивную систему. Память позволяет агенту:

- сохранять релевантные факты при переходе от одной итерации к другой;
- вспоминать прошлые осмысления и ошибки;
- накапливать знания;
- сохранять индивидуальность и помнить о цели.

Баз памяти нет никакой самосогласованности, а каждый цикл – это полная перезагрузка.

2. Когнитивная архитектура памяти

Память ИИ-агентов можно отнести к двум большим категориям: долговременной и кратковременной.

Эти две системы образуют иерархическую когнитивную модель, очень похожую на способ познания у человека. В кратковременной памяти хранится «сейчас», а в долговременной – то, что «известно».

3. Кратковременная память (рабочий контекст)

Кратковременная память относится к непосредственному контексту, доступному LLM во время взаимодействия – обычно в пределах его окна токенов (например, 8K, 32K или 128K токенов).

Основные особенности

- Существует только в течение активного рассуждения (один цикл или цепочка запросов).
- Автоматически забывается, когда контекстное окно заполняется.
- Хранит текущие цели, недавние осмысления и временные данные.
- Обеспечивает связность и целенаправленность рассуждения.

Пример

Когда ИИ-агент планирует многошаговую задачу извлечения данных:

- он хранит текущую подзадачу (например, «извлечь данные с сайта X»);
- сохраняет промежуточные результаты (частичные обобщения);
- отслеживает результаты (успешно или неудачно) последних шагов.

Когда цикл заканчивается или сеанс инициализируется заново, эта память уничтожается, если только явно не сохранена в долговременном хранилище.

Ограничения

- Хрупкая и недолговечная.
- Теряется непрерывность по истечении времени жизни токена или завершения сеанса.
- Нет возможности вспомнить прошлые проекты, пользователей или осмысления.

Аналогия

Похожа на человека, который помнит только что прочитанный абзац, но забывает после короткого сна.

4. Долговременная память (постоянные знания)

Долговременная память обеспечивает непрерывность, обучение и рост.

Она не стирается по завершении сеанса, что позволяет агентам «запоминать» предыдущий опыт, идеи или взаимодействия.

Технически это реализовано как слой памяти, основанный на поиске, когда ключевые идеи, события и результаты хранятся в виде погружений в векторной базе данных или графе знаний.

Ключевые функции

- Хранит осмысления, обобщения и результаты прошлых циклов.
- Извлекает релевантные факты для будущих рассуждений.

- Поддерживает восстановление контекста в новых диалогах или проектах.
- Закладывает фундамент «личности» агента и накопления знаний.

Пример реализации

По завершении задачи агент сохраняет: [Резюме: извлеченные из данных PubMed пары препарат–мишень; частота успеха 92 %; осмысление: улучшить фильтр в виде регулярного выражения].

Столкнувшись впоследствии с похожей задачей:

- агент извлекает свою память с помощью поиска в векторной базе данных;
- включает прошлые уроки в новый план.

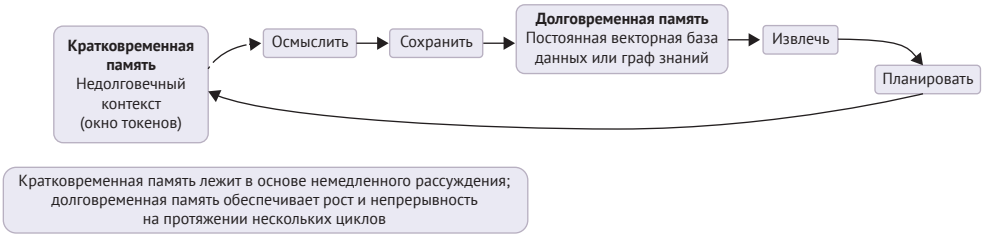
В виде кода:

```
memory.save_embedding("drug_target_extraction_reflection", embedding_vector)
```

При запросе:

```
context = memory.retrieve("extract relationships from biomedical text")
```

Затем LLM использует эту извлеченную из базы память в качестве контекстуального обоснования перед новым планированием.



5. Как память интегрируется с когнитивными циклами

Каждый этап цикла «Планирование → Действие → Осмысление → Пересмотр» взаимодействует с памятью по-своему.

- **Планирование** использует долговременную память для извлечения целей, правил, прошлых решений и релевантного контекста.
- **Действие** опирается на рабочую память, в которой хранятся текущие задачи, промежуточные результаты и активное состояние среды.
- **Осмысление** анализирует последствия выполненных действий, сравнивает ожидаемый и фактический результат и выявляет ошибки или новые закономерности.
- **Пересмотр** обновляет память: полезные стратегии закрепляются, неудачные корректируются, а новые знания добавляются в долговременное хранилище.

Таким образом, память выступает не просто архивом данных, а активным механизмом обучения и адаптации. Каждый когнитивный цикл изменяет внутреннее состояние системы, улучшая ее последующие решения.

Этот циклический поток гарантирует, что каждый цикл вносит вклад в накопленный интеллект.

6. Кодирование памяти и логика поиска

В момент сохранения информации агенты конвертируют текст (например, обобщения, осмысления, взаимодействия с пользователем) в векторы погружения.

Пример

```
vector = The last attempt failed due to incorrect query logic.")
memory_db.add(id="reflection_102", vector=vector,
              metadata={"task":"query_building"})
```

Чтобы вспомнить релевантный опыт:

```
results = memory_db.query("optimize query for RAG pipeline", top_k=3)
```

В результате извлекается прошлый опыт, относящийся к оптимизации RAG, что позволяет агенту «вспомнить, как он обучался».

7. Угасание памяти и забывание

Не все воспоминания следует хранить вечно.

Чтобы избежать перегрузки информацией и дрейфа, когнитивные агенты реализуют избирательное забывание посредством:

- угасания со временем: постепенно понижают важность старых воспоминаний;
- отсечения по релевантности: удалением воспоминаний, которые больше не актуальны для предметной области агента;
- сжатия: обобщением связанного опыта в понимание более высокого уровня.

Пример

Если в журнале агента имеется 100 записей «ошибка при разборе JSON», то он может обобщить их в следующее метанаблюдение: «При разборе JSON, полученного от API, следить за правильной кодировкой и повторять попытку в случае ошибки».

Этот подход отражает человеческий способ познания – сохранять выводы, а не разноречивые факты.

8. Практический пример: осмысляющий исследовательский агент с памятью

Сценарий

Исследовательский агент ежедневно обобщает научные статьи и обновляет базу знаний.

Динамика памяти

- Кратковременная память: хранит название и извлеченные тематики обрабатываемых в текущем сеансе статей.
- Долговременная память: хранит обобщенные результаты, повторяющиеся темы и прошлые осмысления.

Технологический процесс

- План: извлечь из памяти обобщения за последнюю неделю.
- Действия: обработать новые статьи.
- Осмысление: выявить повторяющиеся тенденции (например, «трансформеры в молекулярной биологии»).
- Пересмотр: сохранить выводы для извлечения в будущем.

После нескольких таких циклов агент формирует хранимое понимание развития темы – чат-боту такое в принципе недоступно.

9. Когнитивный принцип памяти

В природе память не просто хранилище. Это опыт, сжатый в удобную для использования форму.

Для ИИ-агентов память играет ту же роль:

- она обеспечивает непрерывность;
- без нее невозможно осмысление;
- она дает контекст для будущих действий.

Без памяти агенты – рабочие. С памятью они становятся учащимися.

Интеллект = Осмысление × Память × Время.

Каждый цикл, в котором сохраняется и вспоминается опыт, завтра сделает агента умнее, чем сегодня.

- Кратковременная память: сиюминутный изменчивый контекст для рассуждений.
- Долговременная память: постоянные знания, допускающие извлечение, для поддержания непрерывности.
- Память питает каждый этап когнитивного цикла, обеспечивая возможность планирования, осмысления и совершенствования.
- Векторные хранилища и графы знаний образуют техническую основу постоянной памяти.
- Интеллектуальное забывание так же важно, как вспоминание.

5.3. Роль контекстных окон и погружений

Агенты помнят не только токены, но и смысловые значения

От токенов к мысли: почему контекст важен

Каждый интеллектуальный диалог – между людьми или между человеком и машиной – зависит от контекста.

Без контекста смысл ускользает.

В случае людей контекст хранится в памяти и формируется в результате опыта.

В случае ИИ, например GPT, контекст опирается на два технических столпа:

- контекстное окно, которое определяет, какой объем контекста модель может «видеть» одновременно;
- погружения, которые определяют, как модель понимает и вспоминает смысл.

В совокупности эти системы позволяют LLM и агентам имитировать непрерывность, сохранять релевантность и связывать идеи во времени, несмотря на то что их архитектура сама по себе не обладает внутренне присущей долговременной памятью.

Давайте рассмотрим, как эти два механизма работают совместно, обеспечивая когнитивные функции современных ИИ-агентов.

1. Контекстное окно: рабочая память LLM

Контекстное окно – это количество токенов (грубо говоря, слов или фрагментов слов), которые LLM может рассматривать одновременно.

Это кратковременная память модели – место, в котором производится рассуждение.

Определение

Контекстным окном называется буфер фиксированной длины, в котором хранятся:

- текущие данные, полученные от пользователя;
- предыдущие ответы модели;
- запросы или инструкции системе;
- извлеченные документы или память.

Пример

- У GPT-4o-mini размер окна равен приблизительно 128 000 токенов.
- Claude 3 Opus предлагает окно размером более 200 000 токенов.

Это означает, что агент может за один присест обработать целую главу книги, набор данных или длинный диалог, при условии, что их размер не выходит за пределы этого ограничения.

2. Как работают контекстные окна

Можете считать, что контекстное окно LLM – это скользящий кадр мысли.

По мере поступления новых данных старые токены стираются – как забываются детали долгого разговора.

Последовательность выполнения процесса

- Пользователь отправляет сообщение.
- LLM кодирует все видимые токены (запрос + история диалога).

- Уровни внимания обрабатывают эти токены, чтобы предсказать следующее слово.
- Когда лимит токенов достигнут, самый старый контекст вытесняется.

Аналогия

Точно так же как вы способны держать в голове лишь несколько мыслей во время рассуждения, LLM может «думать» ровно столько, сколько влезает в ее окно.

3. Проблема: ограничения контекстного окна

Несмотря на то что современные модели имеют большие контекстные окна, они все равно конечны.

Как только буфер переполняется:

- информация о начале диалога исчезает;
- агент теряет непрерывность процесса;
- длительные начинания (например, многошаговые рассуждения) страдают от «затухания контекста».

Вот почему системы внешней памяти (векторные базы данных, обобщения или графы) имеют решающее значение – они расширяют контекст за пределы ограниченного числа токенов.

Ключевая идея

- Контекстные окна привносят осведомленность.
- Системы памяти привносят постоянство.

4. Погружения: преобразование смысла в математику

Если контекстное окно – это память системы, то погружения – язык понимания.

Определение

Погружение – это числовое представление текста, изображения или данных, улавливающее смысловое значение, а не поверхностную форму.

Каждое предложение, фраза или концепт преобразуются в вектор – список чисел с плавающей точкой (например, 1536-мерный в модели OpenAI text-embedding-3-large).

Если два текста семантически схожи, то их векторы будут близки, что позволяет моделям измерять семантическое сходство с помощью простой математики (коэффициента Отиаи).

5. Как работают погружения

Чем ближе два вектора в пространстве, тем сильнее связаны соответствующие им смысловые значения.

Формула

$$\text{similarity}(A,B) = (A \cdot B) / \|A\| \, \|B\|$$

Это косинусное расстояние лежит в основе поиска в RAG-системах, что позволяет агентам вспоминать релевантную информацию, даже когда она изложена другими словами.

6. Расширение контекста за счет погружений

Когда контекстное окно агента заполняется, погружения позволяют выгрузить память во внешнее хранилище.

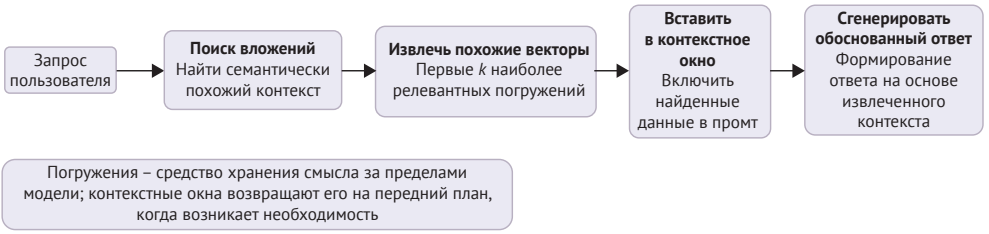
Вместо запоминания точных слов запоминается смысл.

Пример сценария

Пользователь спрашивает: «Какие побочные эффекты имеет ибупрофен?»

- Агент производит поиск в векторной базе данных погружений, созданной на основе медицинских документов
- Извлекаются наиболее семантически похожие фрагменты (например, «нестероидные противовоспалительные препараты» и «раздражение желудка»).
- Эти извлеченные фрагменты загружаются в контекстное окно LLM.
- Теперь модель «помнит» релевантные факты, оставаясь в пределах ограничений на число токенов.

Этот процесс эффективно расширяет память LLM за пределы ее внутреннего контекстного окна.



7. Баланс между контекстом и погружениями

В сочетании контекст и погружения образуют гибридный интеллект, когда «рабочая память» модели (контекст) органично взаимодействует с ее «семантической памятью» (погружениями).

8. Использование погружений в когнитивных циклах

Погружения стоят за этапами осмысления и пересмотра когнитивного цикла.

1. Осмысление

- Обобщить то, что было выучено.
- Закодировать результаты осмысления в виде вектора погружения.
- Сохранить его в векторной памяти.

2. Пересмотр

- Столкнувшись с похожей проблемой позже, опросить погружения.
- Извлечь прошлые осмысления с высокой степенью сходства.
- Использовать их для уточнения плана и процесса принятия решений.

Пример

```
# Хранилище осмыслений
vector = embed("Теперь знаю, что для запроса к большим графам в Neo4j требуется явно
 построить индекс.")
memory.add("graph_query_reflection", vector)

# Извлечение
query_vector = embed("оптимизировать Cypher-запросы к большим наборам данных")
similar_reflections = memory.query(query_vector, top_k=3)
```

Этот цикл извлечения позволяет агентам повторно использовать опыт – так же работает человеческая интуиция.

9. Сжатие контекста: хранение того, что действительно важно

Даже при использовании погружений контекстные окна могут переполняться.

Чтобы сохранить главное и при этом сэкономить место, агенты применяют сжатие контекста.

- Обобщения: заменить полную запись беседы кратким рефератом.
- Фильтрация по релевантности: оставлять только первые k самых релевантных результатов.
- Разбиение на фрагменты: разбивать длинные документы на пригодные для работы семантические куски.
- Иерархический контекст: хранить обобщения на нескольких уровнях (абзац → раздел → тема).

Это гарантирует, что в ограниченном рабочем пространстве модели всегда будет храниться максимальный сигнал с минимальным шумом.

10. Практический пример: GraphRAG-агент с контекстом и погружениями

Биомедицинский агент использует:

- контекстное окно для текущего рассуждения (не более 128K токенов);
- погружения для долговременной памяти (хранящейся в FAISS).
- запросы к графу для поиска связей между белками, генами и лекарственными препаратами.

Когда пользователь просит: «Найди соединения, аналогичные тем, что воздействуют на белки, вызывающие болезнь Альцгеймера», агент выполняет следующие действия:

- ищет векторные погружения для похожих соединений;
- запрашивает у графа знаний биологические связи;
- вставляет лучшие результаты в контекстное окно LLM;
- генерирует ответ с объяснением химического и реляционного сходства.

Результат

Гибридный процесс рассуждения, для которого имеют место обоснованность, контекстная осведомленность и объяснимость.

11. Принцип познания

В случае человека:

- рабочая память содержит то, о чем мы размышляем сейчас;
- в семантической памяти хранится то, чему мы научились со временем.

В случае познания ИИ:

- рабочей памятью являются контекстные окна;
- семантической памятью являются погружения.

Это не противоположности, а взаимодополняющие полушария искусственного интеллекта.

Контекст дает концентрированность.

Погружения дают память.

Вместе они дают понимание.

- Контекстные окна определяют непосредственную область рассуждений LLM.
- Погружения кодируют смысл для долговременного запоминания и семантического поиска.
- Вместе они расширяют когнитивные возможности агента далеко за пределы его базовой архитектуры.
- Погружения служат «семантическими короткими путями» – они извлекают релевантный смысл, а не просто текст.
- Эффективное управление контекстом – ключ к стабильному и объяснимому поведению агента.

5.4. Модели действий и циклы самооценки

Агенты учатся действовать, наблюдать и самосовершенствоваться.

От генерации ответа к осознанному действию

Различие между LLM и интеллектуальными агентами – в действии, т. е. способности взаимодействовать с инструментами, API, окружающей средой и источниками данных для автономного достижения целей.

Но действие без осмысления быстро превращается в хаос.

Чтобы действовать разумно, агент должен оценивать качество своей работы, обнаруживать, когда он сбился с курса, и корректировать свою стратегию.

В этом и состоит сущность модели действий в сочетании с циклами самооценки – рабочего механизма автономного рассуждения.

1. Что такое модель действий?

Модель действий определяет, как агент выполняет свои намерения в реальном мире.

Это внутренний механизм, который соединяет планирование (что делать) с исполнением (как это сделать).

Компоненты модели действий

Модель действий дает агенту то, что делает его агентом, – способность выбирать, действовать и учиться на результатах действий.

2. Как работают модели действий

Модель действий обычно представляет собой структурированный цикл принятия решений, аналогичный человеческому мышлению:

Цель → План → Выполнение → Наблюдение → Анализ → Корректировка.

Каждый шаг служит основой для следующего благодаря циклам самооценки. Разберем один за другим.

3. Шаг 1: планирование действия

Прежде чем что-то делать, агент составляет план:

- идентифицировать цели и ограничения;
- выбрать подходящие инструменты или методы;
- предсказать результаты.

Пример

Агент финансовой аналитики получает запрос: «Сравни показатели Tesla за второй квартал 2025 года с показателями Ford».

План

- Получить квартальные данные Tesla и Ford, пользуясь финансовым API.
- Вычислить темпы роста и рентабельность.
- Обобщить результаты.

4. Шаг 2: действие (этап выполнения)

Составив план, агент выполняет необходимые шаги с помощью вызова инструментов или внутренней логики.

Пример

```
def fetch_financials(company, quarter):
    return call_api(f"https://api.marketdata.io/{company}/{quarter}")
```

Здесь особенно важны журналы выполнения, так как в них сохраняется трасса памяти, которую агент позже будет анализировать на этапе осмысления.

5. Шаг 3: наблюдение за средой

Наблюдение замыкает цикл обратной связи между ожиданиями модели и реальностью.

Агент анализирует:

- отвечал ли API корректно?
- были ли полны извлеченные данные?
- попадают ли значения в ожидаемые диапазоны?

Наблюдения составляют основу для самооценки – так же как сенсорная обратная связь у биологических организмов.

6. Шаг 4: самооценка (осмысление в действии)

После того как действие выполнено и наблюдения стали доступны, агент входит в цикл самооценки.

Процесс оценивания

- Сравнить полученный результат с ожидаемым.
- Оценить точность и полноту.
- Выявить ошибки, несоответствия или пропущенные шаги.
- Обобщить результаты осмысления.
- При необходимости изменить параметры и составить новый план.

Пример запроса на этапе осмысления

- Осмысли свое последнее действие. Была ли достигнута цель?
- Если нет, опиши проблему и предложи корректировку.

Ответ агента

- API вернул для Tesla неполные данные за второй квартал из-за ошибки аутентификации. Попробовать еще раз с измененными учетными данными.

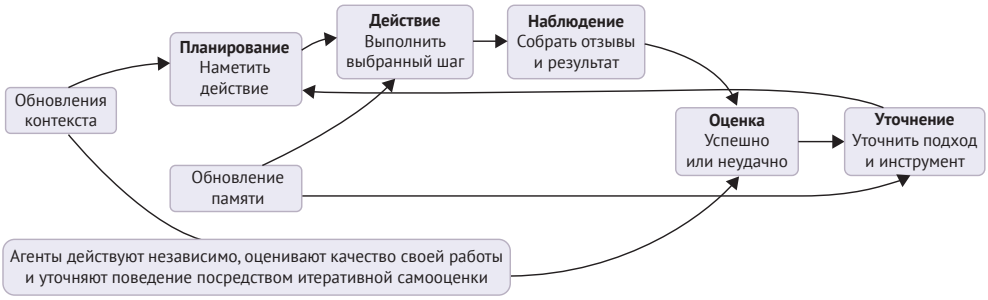
  Самооценка превращает выполнение в обучение.

7. Шаг 5: пересмотр и повтор

Результаты оценки подаются на вход шага планирования.

- Если результаты неудовлетворительны → составить новый план и повторить.
- Если к качеству претензий нет → сохранить осмысление в долговременной памяти.

Так создается когнитивный цикл «Планирование → Действие → Осмысление → Пересмотр».



8. Модели действий на практике

A. Агенты, пользующиеся инструментами

Такие фреймворки, как AutoGPT и CrewAI, реализуют модели действий с помощью инструментов.

Пример последовательности

- LLM решает, какой инструмент использовать.
- Исполнитель действия: выполняет операцию.
- Анализатор: проверяет результат.
- Осмыслитель: генерирует предложения по улучшению.

Упрощенный пример журнала

- [План] Получить данные о компании.
- [Действие] GET /api/finance/tesla
- [Наблюдение] 401 Unauthorized.
- [Осмысление] Истек срок действия токена. Обновить и повторить.
- [Пересмотр] Новый план – повторить аутентификацию.

Каждая итерация цикла повышает надежность – частота ошибок со временем уменьшается.

B. Агенты, обучающиеся с подкреплением

При обучении с подкреплением модель действий определяет множество действий, доступных в окружающей среде («политику»).

Самооценка происходит с помощью сигналов: вознаграждения за успех, штрафа за неудачу или уведомления об ошибке.

Ключевая параллель

- Сигнал обучения с подкреплением = количественная обратная связь.
- Осмысление в когнитивном цикле = качественная обратная связь.

То и другое служит одной цели: оптимизации поведения.

C. Многоагентные системы

В многоагентных фреймворках (например, MetaGPT, OpenDevin) у каждого агента может быть собственная модель действий и процесс оценки.

- Агент-планировщик: создает список задач.
- Агент-исполнитель: выполняет задачи.
- Агент-критик: оценивает результаты и предоставляет отзыв.

Это разделение ролей зеркально отражает человеческие организации – сотрудничество посредством межагентного осмысления.

9. Пример: отладка по итогам осмысления

Сценарий

Агент по классификации документов раз за разом некорректно помечает «финансовые отчеты» как «пресс-релизы».

Выполнение когнитивного цикла

- Действие: классифицировать документы → обнаружена неправильная классификация.
- Осмысление: «Недоразумение вызвано совпадением ключевых слов (earnings release)».
- Пересмотр: изменить правило классификации, включив проверку на ключевые слова «10-K» или «SEC filing».
- Повторное действие: точность улучшилась.

Со временем такое уточнение по итогам самооценки приводит к формированию адаптированного к предметной области интеллекта.

10. Включение метрик оценки

Самооценка может быть не только качественной. Агенты также могут отслеживать количественные показатели эффективности.

Записывая эти метрики в журнал, агент может мониторить динамику своего обучения – примерно так же, как люди отслеживают прогресс с помощью обратной связи.

11. Запросы самооценки на практике

Агенты могут использовать рефлексивные запросы для оценки своего качества.

Пример метазапроса: «Оцени свой последний шаг рассуждения».

1. Какого результата ты ожидал?
2. Удалось ли его достичь?
3. Если нет, то почему?
4. Что можно сделать по-другому в следующий раз?

Эти метазапросы можно автоматизировать после каждого крупного действия или решения.

В длительных технологических процессах они образуют основу самоанализа и самокоррекции.

12. Связывание с памятью и контекстом

Осмысление бесполезно, если оно не запоминается. И тут в игру вступают системы памяти и погружения контекста (см. предыдущие разделы).

Интеграция с технологическим процессом

Обобщенные осмысления кодируются в виде погружений. Они сохраняются в долговременной памяти и извлекаются в ходе будущего планирования, если оказываются семантически релевантны.

Пример

Сохраненное осмысление: «Всегда проверять диапазоны дат в финансовых запросах».

Впоследствии при анализе очередного квартального отчета это осмысление извлекается автоматически.

Результат: накопление опыта со временем.

13. Когнитивный принцип: размышления о мышлении

У людей метапознание – способность размышлять о собственных мыслях – именно то, что отличает рутинное поведение от обучения. У ИИ-агентов ту же роль играет самооценка.

- Она повышает осведомленность о качестве рассуждений.
- Она преобразует данные в полезную информацию.
- Она позволяет вносить коррективы целенаправленно, а не методом проб и ошибок.

Разумное поведение заключается не в том, чтобы избегать ошибок, а в том, чтобы учиться на них быстрее, чем они происходят.

- Модели действий определяют, как агенты преобразуют цели в поведение.
- Циклы самооценки позволяют агентам измерять и уточнять свои действия.
- Вместе они образуют операционную основу автономности.
- Журналы осмысления становятся частью долговременной памяти – растущего хранилища опыта.
- Интеллектуальные агенты несовершенны, но способны к самокоррекции.

5.5. Рефлексивные и реактивные агенты

Эволюция от реакции на запросы до самосознающих мыслителей. От реакции к осмыслению

Большинство современных систем искусственного интеллекта реактивны – они реагируют на входные данные, выполняют команды и выдают результаты.

Они не приостанавливаются, чтобы спросить:

«Был ли этот подход лучшим?», «Ничего ли я не упустила?», «Может, стоит попробовать по-другому?»

А рефлексивные агенты все это делают.

Они занимаются метапознанием – рассуждениями о собственных рассуждениях. Они оценивают результаты, выявляют ошибки, уточняют методы и учатся на собственном опыте.

Это различие знаменует переход от механической автоматизации к адаптивному интеллекту – от систем, реагирующих на запросы, к системам, способным к самообучению.

1. Что такое реактивный агент?

Реактивным называется агент, который реагирует непосредственно на раздражители, выполняя заранее заданные или немедленные действия без обдумывания и использования памяти.

Он предназначен для быстрых, но не обязательно разумных действий.

Характеристики реактивных агентов

Реактивные агенты эффективны для предсказуемых коротких заданий, но не справляются, когда:

- контекст охватывает несколько шагов;
- цели сформулированы расплывчато;
- требуется обучение или адаптация.

Пример

Реактивный бот технической поддержки может отвечать «Перезапустите устройство» на любой вопрос – даже когда пользователь говорит, что уже сделал это. Он не рассуждает, а следует правилам.

2. Что такое рефлексивный агент?

С другой стороны, рефлексивный агент включает самооценку, обратную связь, обоснованную рассуждением, и циклы обучения. Он не просто действует, а еще и размышляет о своих действиях.

Рефлексивные агенты сочетают рассуждение, осмысление и пересмотр, преобразуя опыт в стратегию.

Пример

Исследовательский ассистент, которому поручено составить обзор литературы, мог бы:

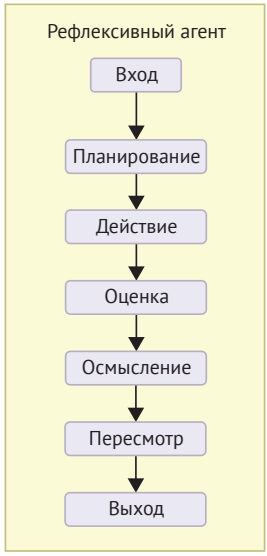
- обобщить первые найденные работы;
- в порядке осмысления задаться вопросом: «Не пропустил ли я статьи за 2024 год?»;
- найти отсутствующие работы;
- повторить обобщение с учетом новых работ.

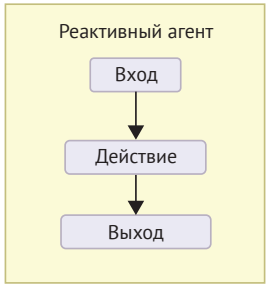
Это не автоматизация – это познание.

3. В чем разница

Говоря по-простому:

- реактивные агенты делают;
- рефлексивные агенты понимают, что делают и почему.





Реактивные агенты отвечают однократно; рефлексивные агенты выполняют циклическую самооценку, чтобы улучшить результаты и обучиться

5. Как осмысление улучшает познание

Осмысление привносит три суперспособности, которых недостает реактивным системам.

A. Осведомленность о течении времени

Рефлексивные агенты помнят прошлое. Они поддерживают непрерывность, т. е. помнят цели, прошлые результаты и обратную связь от пользователей.

B. Самодиагностика

Они анализируют, согласуются ли их действия с намерениями.

Вместо вопроса «Что дальше?» они задают вопрос «Что пошло не так и как я могу это исправить?».

C. Непрерывное улучшение

Они эволюционируют. Каждое осмысление улучшает следующий цикл, снижая количество ошибок и повышая уверенность в себе.

Этот процесс зеркально отражает человеческий опыт: осознанную практику, самооценку и постоянное совершенствование.

Результат: рефлексивный агент обеспечивает точность, полноту и объяснимость – три характеристики, которых недостает реактивным системам.

6. Реализация осмысления в агентных архитектурах

Рефлексивные агенты обычно используют запросы с целью анализа или субагентов, выполняющих роль критика.

Пример цикла осмысления (псевдокод):

```
def reflective_agent(input):
    plan = llm.plan(input)
    result = execute(plan)
    evaluation = llm.evaluate(result)
    if "improvement" in evaluation:
        plan = llm.revise(plan, evaluation)
        result = execute(plan)
    return summarize(result)
```

Этот двухпроходный процесс (выполнение → самокритика → пересмотр) добавляет когнитивный уровень в технологические процессы, в остальном линейные.

7. Роль памяти в осмыслении

Осмысление важно, только если оно сохраняется. В противном случае агент забывает, чему научился, и повторяет одни и те же ошибки.

Поэтому рефлексивные системы интегрированы с долговременной памятью, в которой хранятся:

- извлеченные уроки;
- типичные ошибки;
- принесшие успех эвристики;
- моментальные снимки контекста.

Затем на этапе планирования агент извлекает эти осмысления, чтобы сформировать новые стратегии.

Пример

Во время прошлых прогонов запросы к API заканчивались неудачно из-за ограничений на частоту. Добавить повтор с экспоненциальной задержкой.

Тем самым осмысление преобразуется в прикладной опыт – основу развивающегося интеллекта.

8. Сильные стороны и компромиссы

За глубину рассуждений рефлексивные агенты расплачиваются производительностью, но взамен приобретают долговременную надежность и постоянное совершенствование.

9. Пример журнала осмыслений (реалистичный)

Найти недавние статьи по этике ИИ.

[Действие] Выбрано 50 рефератов из arXiv.

[Наблюдение] Многие результаты датированы 2022 годом.

[Осмысление] Запрос не отобрал более свежие статьи из-за отсутствующего фильтра «2023/2024».

[Пересмотр] В запрос добавлен фильтр по году.

[Повтор действия] Найдены новые статьи, обобщение отражает текущее положение дел.

Этот журнал обеспечивает возможность прослеживания, а именно показывает, как и почему улучшился агент.

10. Когда какой тип агента использовать

В особо ответственных областях – медицине, юриспруденции, финансах, научных исследованиях – осмысление не является факультативным.

Здесь проходит граница между автоматизацией и подотчетностью.

11. Когнитивный принцип

В терминах, применимых к людям:

- реактивное поведение инстинктивно – оно быстрое, автоматическое и краткосрочное;
- рефлексивное поведение осознанно – оно медленное, вдумчивое и самокорректирующееся.

В терминах, применимых к ММ:

- реактивные агенты предсказывают следующее слово;
- рефлексивные агенты предсказывают следующее улучшение.

Осмысление – это реакция, дополненная памятью и намерением. Это когнитивный скачок к автономному интеллекту.

- Реактивные агенты реагируют мгновенно, но неспособны к самоанализу.
- Рефлексивные агенты анализируют свою работу и совершенствуются благодаря обратной связи.
- Осмысление привносит метапознание – размышление о мышлении.
- Память позволяет накапливать осмысления, формируя долговременное обучение.
- Компромисс: замедленная реакция, но экспоненциально растущий интеллект.

5.6. Агент в действии: построение самокорректирующегося ИИ-ассистента с циклами осмысления

Сквозная реализация цикла «Планирование → Действие → Осмысление → Пересмотр»

Что мы построим

Компактный помощник, который отвечает на вопросы, ссылаясь на источники, и автоматически проверяет свои ответы. Если критик отмечает пробелы или ошибки, то помощник пересматривает свой ответ с помощью контролируемого цикла осмысления. Запустить его можно из командной строки или с помощью API.

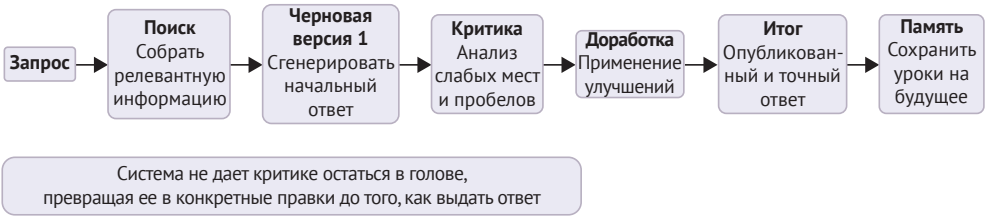
Основные возможности

- Извлечение контекста (точка подключения, готовая к использованию RAG, работает с векторным хранилищем или без него).
- Генерация начального ответа со ссылками на источники.
- Критика ответа на предмет обоснованности фактами, покрытия и следования инструкциям.
- Пересмотр ответа, если критик находит проблемы.
- Сохранение осмыслений в долговременной памяти для будущих запросов.

Архитектура

Шаги конвейера:

- планирование: разобрать полученное от пользователя задание и определить стратегию поиска (с RAG или без);
- действие: сгенерировать черновую версию v1, используя найденный контекст;
- осмысление: провести структурированный самоанализ версии v1;
- пересмотр: сгенерировать черновую версию v2, учитывающую критику; при необходимости повторить;
- отчет: вернуть окончательный ответ с указанием источников и краткое обобщение осмыслений;
- запоминание: сохранить погружение осмысления и основные уроки.



Использовать строгие, разделенные по ролям запросы, чтобы свести дрейф к минимуму.

SYSTEM_ANSWER

Ты базирующийся на фактах ассистент. Используй для ответа ТОЛЬКО предоставленный КОНТЕКСТ.

Если ответ не найден, сообщи: «В предоставленном контексте недостаточно информации».

Используй ссылки вида [S1], [S2], ... с указанием предоставленных идентификаторов источников. Будь точен и краток.

SYSTEM_CRITIC

Ты строгий рецензент. Оцени черновую версию DRAFT на соответствие ВОПРОСУ и КОНТЕКСТу.

- Верни JSON со следующими полями:
  - issues: список строк (отсутствующие факты, галлюцинации, нарушения инструкций);
  - required_fixes: список конкретных изменений, которые должен внести автор;
  - verdict: одно из значений ["approve", "revise"].
- Будь строг. Если для какого-то утверждения нет опоры в КОНТЕКСТе, отметь это.

SYSTEM_REVISER

Ты опытный редактор. Примени REQUIRED_FIXES к черновому варианту, используя КОНТЕКСТ.

Сохраняй корректное содержимое. Сохраняй ссылки на источники. Если контекста недостаточно, явно сообщи об этом.

Возвращай только отредактированный текст ответа.

Минимальный уровень поиска (готов к подключению плагинов)

Подключить векторное хранилище можно позже; эта заглушка демонстрирует детерминированный интерфейс.

```
# retrieval.py
from typing import List, Dict

def retrieve_context(question: str, k: int = 5) -> List[Dict]:
    """
    Вернуть список {id: 'S1', text: '...', meta: {...}}.
    Заменить эту заглушку вызовом FAISS/Chroma/Pinecone/Weaviate.
    """

    # Минимальный остов: никаких внешних зависимостей
    corpus = [
        {"id": "S1", "text": "RAG снижает галлюцинации, вставляя найденные
фрагменты в запрос."},
        {"id": "S2", "text": "Графы знаний моделируют сущности и связи;
Cypher-запросы обходят их."},
        {"id": "S3", "text": "Коэффициент Отиаи часто используется для поиска
погружений в векторной базе данных."},
    ]
```

```
    return corpus[:min(k, len(corpus))]

def format_context(snippets: List[Dict]) -> str:
    blocks = []
    for s in snippets:
        blocks.append(f"[{s['id']}] {s['text']}")
    return "\n".join(blocks)
```

Цикл самокоррекции (один исполняемый файл)

```
import json, os
from typing import List, Dict
from retrieval import retrieve_context, format_context

USE_OPENAI = bool(os.getenv("OPENAI_API_KEY"))
if USE_OPENAI:
    from openai import OpenAI
    client = OpenAI()

def chat(messages, model="gpt-4o-mini", temperature=0.2) -> str:
    if not USE_OPENAI:
        # Заглушка для локального агента – только для иллюстрации
        return "Локальный сервер не сконфигурирован."

    resp = client.chat.completions.create(model=model,
        messages=messages, temperature=temperature)
    return resp.choices[0].message.content.strip()

Ты базирующийся на фактах ассистент. Используй для ответа ТОЛЬКО предоставленный КОНТЕКСТ.
Если ответ не найден, сообщи: «В предоставленном контексте недостаточно информации».
Используй ссылки вида [S1], [S2], ... с указанием предоставленных идентификаторов источников.
Будь точен и краток.

SYSTEM_ANSWER = """Ты базирующийся на фактах ассистент. Используй для ответа ТОЛЬКО
предоставленный КОНТЕКСТ.
Если ответ не найден, сообщи: "В предоставленном контексте недостаточно информации".
Используй ссылки вида [S1], [S2], ... с указанием предоставленных идентификаторов источников.
Будь точен и краток."""

SYSTEM_CRITIC = """Ты строгий рецензент. Оцени черновую версию DRAFT
на соответствие ВОПРОСУ и КОНТЕКСТу.
Верни JSON со следующими полями:
- issues: список строк (отсутствующие факты, галлюцинации, нарушения инструкций);
- required_fixes: список конкретных изменений, которые должен произвести автор;
- verdict: одно из значений ["approve", "revise"].
Будь строг. Если для какого-то утверждения нет опоры в КОНТЕКСТе, отметь это."""

SYSTEM_REVISER = """Ты опытный редактор. Примени REQUIRED_FIXES
к черновому варианту, используя КОНТЕКСТ.
Сохраняй корректное содержимое. Сохраняй ссылки на источники.
Если контекста недостаточно, явно сообщи об этом.
Возвращай только отредактированный текст ответа."""
```

```
def answer_with_reflection(question: str, max_revisions: int = 1) -> Dict:
    # 1) Поиск
    ctx_snippets = retrieve_context(question, k=5)
    context_text = format_context(ctx_snippets)
    # 2) Черновая версия v1
    draft = chat([
        {"role":"system","content":SYSTEM_ANSWER},
        {"role":"user","content":f"QUESTION:\n{question}\n\nCONTEXT:\n
        {context_text}"}
    ])
    reflections: List[Dict] = []
    final = draft
    # 3) Критик + факультативный цикл редактирования
    for _ in range(max_revisions + 1):
        critic_raw = chat([
            {"role":"system","content":SYSTEM_CRITIC},
            {"role":"user","content":f"QUESTION:\n{question}\n\nCONTEXT:\n
            {context_text}\n\nDRAFT:\n{final}"}
        ], temperature=0.0)
        try:
            critique = json.loads(critic_raw)
        except Exception:
        critique = {"issues":["critic returned non-JSON"],
                    "required_fixes": ["Return JSON."],
                    "verdict":"revise"}
        reflections.append(critique)
        if critique.get("verdict","revise") == "approve":
            break
        required_fixes = critique.get("required_fixes", [])
        fixes_text = "\n".join(f"- {fx}" for fx in required_fixes) or
            "-Strengthen citations."
        revised = chat([
            {"role":"system","content":SYSTEM_REVISER},
            {"role":"user","content":f"QUESTION:\n{question}\n\nCONTEXT:\n
            {context_text}\n\nREQUIRED_FIXES:\n{fixes_text}\n\nDRAFT:\n{final}"}
        ])
    final = revised
    return {
        "question": question,
        "answer": final,
        "citations": [s["id"] for s in ctx_snippets],
        "reflections": reflections,
    }

if __name__ == "__main__":
    print(json.dumps(answer_with_reflection("Как RAG уменьшает галлюцинации?"),
          indent=2))
```

Обертки – командная и с помощью API

```
# cli.py
import sys, json
from self_correcting_agent import answer_with_reflection
```

```
if __name__ == "__main__":
    q = " ".join(sys.argv[1:]) or "Explain Cypher vs. SPARQL."
    res = answer_with_reflection(q, max_revisions=1)
    print(json.dumps(res, indent=2, ensure_ascii=False))

# serve.py
from fastapi import FastAPI
from pydantic import BaseModel
from self_correcting_agent import answer_with_reflection

app = FastAPI(title="Self-Correcting Assistant")

class Q(BaseModel):
    question: str
    max_revisions: int = 1
    @app.post("/ask")

    def ask(q: Q):
        return answer_with_reflection(q.question, q.max_revisions)
```

Ограничения и ошибки

Строгая обоснованность: запрос к системе запрещает ничем не подкрепленные утверждения; за этим следит критик.

Критик возвращает ответ не в формате JSON: резервный вариант перехватывает ошибку и принудительно выполняет одно редактирование.

Недостаточный контекст: редактор должен сообщить «Недостаточно информации...» и ничего не изобретать.

Управление циклом: max_revisions предотвращает зацикливание (начините со значения 1 или 2).

Детерминированность для аудита: поддерживать низкую температуру (0.0–0.3) в критике/редакторе.

Постоянная память для осмыслений (факультативно)

Сохраняйте краткие уроки, чтобы будущие ответы были лучше.

```
# memory.py (sketch)
from typing import Dict, List

# Замените своей векторной БД; здесь она моделируется списком
MEM = []
def store_reflection(critique: Dict, question: str):
    summary = "; ".join(critique.get("issues", [])) or "no-issues"
    MEM.append({"q": question, "summary": summary})

def recall_reflections(query: str) -> List[str]:
    return [m["summary"] for m in MEM][-3:]
```

Вызывайте `store_reflection` для последнего критика и вставляйте `recall_reflections(question)` в начало запроса при последующих запусках.

Контрольный список

Перед развертыванием проверьте следующие моменты.

- Поиск: Recall@k ⩾ 0.8 на небольшом «золотом» наборе данных.
- Обоснованность: произведите случайную выборку ответов; каждому утверждению должен соответствовать идентификатор цитированного источника.
- Частота галлюцинаций: цель – приблизиться к нулю при строгих запросах.
- Отдача редактирования: доля прогонов, в которых черновая версия v2 существенно улучшает черновую версию v1.
- Расплата задержками: критика и пересмотр обходятся в дополнительные вызовы: кеширование результатов поиска и сжатие контекста.

Развертывание в режиме эксплуатации

- Добавить векторную базу данных и средство переранжирования (например, NSW + cross-encoder).
- Добавить вызов функций инструментов (веб, SQL, запросы к графам).
- Протоколировать каждый шаг цикла, включая идентификаторы для аудита.
- Экспортировать компактный «отчет об осмыслении» вместе с окончательным ответом.
- Самокорректирующийся ассистент приводит в действие последовательность «Планирование → Действие → Осмысление → Пересмотр».
- Критик должен быть строгим и структурированным; редактор должен работать над конкретными исправлениями.
- Обоснованность плюс осмысление уменьшают количество галлюцинаций и повышают полноту.
- Сохранение осмыслений превращает отдельные успехи в повторяющееся поведение.

Глава 6

Многоагентные системы: сотрудничество и координация

6.1. От индивидуальных ботов к командному интеллекту

Многоагентные системы важны. От единичного разума к коллективному интеллекту

До сих пор мы изучали внутреннюю механику одного интеллектуального агента – как он планирует, действует, осмысляет и обучается с помощью когнитивных циклов.

Но в реальном мире ни один агент не может все делать хорошо. Подобно тому, как люди разделяются по специальностям, – ученые проводят исследования, инженеры строят, менеджеры координируют, – ИИ-агенты преуспевают, когда сотрудничают.

Это эволюция от индивидуального интеллекта к коллективному познанию.

И этот переход от рассуждений одного агента к сотрудничеству нескольких агентов меняет представление о будущем искусственного интеллекта.

1. Ограничения индивидуального агента

Даже самая продвинутая система с одним агентом, оснащенная RAG, памятью и осмыслением, сталкивается с фундаментальными ограничениями при решении сложных междисциплинарных задач.

Ограничения одного агента

Человек в процессе познания решал такие проблемы с помощью специализации и кооперации – и ИИ идет по тому же пути.

2. Что такое многоагентные системы (MAC)?

Многоагентная система (MAC) – это скоординированная сеть автономных агентов, которые взаимодействуют, сотрудничают и преследуют общие цели для коллективного решения сложных задач.

Каждый агент:

- имеет свою роль, цели и возможности;
- действует независимо, но в рамках общей среды;
- обменивается информацией с другими агентами посредством структурированных протоколов связи.

Вместе они образуют распределенный интеллект, способный решать задачи, которые не под силу ни одной отдельной модели.

Определение

Многоагентная система – это совокупность интеллектуальных объектов, которые взаимодействуют для достижения индивидуальных или общих целей посредством коммуникации, координации и кооперации.

3. Аналогия: от сольного исполнения к оркестру

Можете считать, что один агент – это сольный исполнитель: умелый, экспрессивный, но ограниченный единственным инструментом.

Многоагентная система – это целый оркестр, где каждый исполнитель играет свою роль, следуя общей партитуре и подстраиваясь в режиме реального времени под других оркестрантов.

- Агент-планировщик выступает в роли дирижера.
- Агент-исследователь начинает исполнять мелодию.
- Агент-критик задает ритм посредством оценивания.
- Агент-исполнитель закрепляет все достигнутое благодаря точной реализации.

Когда эти части синхронизируются, система демонстрирует эмерджентный интеллект – способность к рассуждению, которая превосходит сумму составляющих ее частей.

4. Почему многоагентные системы важны

Появление многоагентных фреймворков сигнализирует о поворотном моменте в развитии искусственного интеллекта.

Давайте рассмотрим ключевые мотивы и преимущества.

A. Разделение труда в процессе познания

Специализированные агенты позволяют распределять экспертные знания, как это делает команда профессионалов.

- Агент-исследователь занимается поиском и синтезом.
- Агент-разработчик отвечает за генерацию и выполнение кода.
- Агент-критик оценивает качество и согласованность.
- Агент-менеджер координирует задачи и разрешает конфликты.

Такая модульность обеспечивает более глубокое рассуждение и высокую пропускную способность.

Пример

В случае разработки продукта:

- агент-исследователь → собирает требования пользователей;
- агент-проектировщик → создает прототипы;
- агент-инженер → готовит признаки;

- агент по контролю качества → тестирует в поисках ошибок;
- агент по документированию → составляет руководства.

Все они работают автономно, но в единой иерархии целей.

B. Распараллеливание и масштабируемость

Одноагентные системы рассуждают последовательно, а многоагентные – параллельно.

Разные агенты могут исследовать различные гипотезы, наборы данных или пути решения одновременно, а затем объединять свои результаты.

Благодаря такому распараллеливанию:

- увеличивается глубина рассуждения;
- уменьшается общая задержка;
- наличие разных точек зрения повышает надежность.

На практике многоагентный конвейер мог бы:

- запустить 3 специализированных агента для решения разных подзадач;
- дать каждому возможность независимо генерировать частичные результаты;
- объединить результаты с помощью агента-супервизора, который устраняет пересечения или конфликты.

Итог: более быстрая сходимость, более глубокое понимание.

C. Избыточность и исправление ошибок

Одиночная LLM может легко впасть в галлюцинации или слишком тесно подстроиться к своей собственной цепочке рассуждений.

Многагентные системы решают эту проблему, позволяя агентам проводить взаимную проверку.

- Один агент генерирует ответ.
- Другой его критикует (осмысление).
- Третий согласовывает или пересматривает.

Так же устроен и научный метод познания – высказывание гипотезы, ее проверка и пересмотр в процессе диалога.

Такие фреймворки, как MetaGPT или ReflectionChain, пользуются этим, чтобы уменьшить частоту галлюцинаций на 30–40 %.

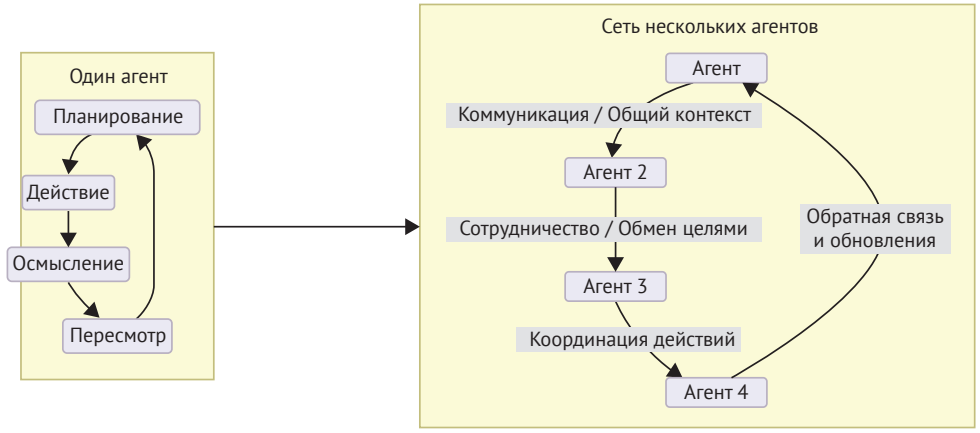
D. Эмерджентное сотрудничество и креативность

При наличии нескольких агентов появляются новые формы интеллекта – коллективное рассуждение, переговоры и творческий синтез.

Пример

- Агент по данным интерпретирует исходные данные.
- Агент-мыслитель ищет закономерности.
- Агент-визуализатор преобразует их в логические выводы.
- Вместе они порождают результат, который никакая модель не могла бы выдать в одиночку.

Такого рода эмерджентная креативность – скоординированная, но не запрограммированная – является одним из самых многообещающих направлений в области агентного ИИ.



В многоагентной системе каждый агент исполняет собственный когнитивный цикл, но все они синхронизируются путем обобществления контекста и обмена целями

5. Пример сценария: коллаборативные исследовательские агенты

Проиллюстрируем, как это работает на примере реального технологического процесса.

Задание: «Подготовь официальный документ о безопасности искусственного интеллекта в крупномасштабных автономных системах».

Агенты

- Агент-планировщик: определяет основные этапы и распределяет задачи.
- Агент-исследователь: подбирает и обобщает источники.
- Агент-писатель: создает структурированный контент.
- Агент-критик: проверяет связность изложения и обоснованность фактами.
- Агент-редактор: объединяет разделы и применяет правки.

Технологический процесс

- Планировщик раздает цели и контекст каждому агенту.
- Исследователь находит научные статьи.
- Писатель создает начальные версии разделов.
- Критик выявляет отсутствующие подтверждения или противоречия.
- Редактор сводит все материалы в окончательный документ и обновляет разделяемую память.

Этот процесс порождает исчерпывающие и достоверные результаты – быстрее, чем могли бы коллективы людей, и без централизованного микроменеджмента.

7. Почему именно сейчас? Технологические факторы

Сегодня многоагентные системы стали жизнеспособными благодаря трем основным достижениям.

1. **Большие языковые модели (LLM).** Современные LLM обеспечивают высокий уровень языкового понимания, логического вывода и генерации кода, позволяя агентам выполнять сложные интеллектуальные задачи автономно.
2. **Доступность API и инструментальных экосистем.** Развитие API, фреймворков оркестрации и облачных сервисов позволило агентам взаимодействовать с внешними инструментами, базами данных, вебсервисами и друг с другом в реальном времени.
3. **Рост вычислительных мощностей и снижение стоимости инфраструктуры.** GPU-кластеры, облачные платформы и масштабируемые распределенные системы сделали возможным одновременный запуск множества взаимодействующих агентов без чрезмерных затрат.

Вместе эти факторы превращают статические LLM в кооперативные объекты, способные к распределенному планированию и динамической координации.

8. Когнитивный принцип: распределенное мышление

Многоагентная система не просто умножает возможности, она расширяет перспективы.

Каждый агент имеет частичное представление о мире, но вместе они приближаются к целостному пониманию.

Ни одна модель не содержит всей истины, но в диалоге они к ней приближаются.

Такой подход, основанный на распределенном рассуждении, отражает масштабированное человеческое познание – коллективный разум, возникающий из множества отдельных интеллектов.

- Многоагентные системы обеспечивают взаимодействие специализированных ИИ-агентов.
- Они решают задачи, которые выходят за рамки возможностей отдельных моделей в плане рассуждения или объема контекста.
- Преимущества таких систем включают в себя распараллеливание, исправление ошибок, масштабируемость и эмерджентную креативность.
- Каждый агент может поддерживать собственный когнитивный цикл, обмениваясь информацией через общую память или протоколы.
- Многоагентная координация – не будущее ИИ, а современный передний край прикладного интеллекта.

6.2. Роли, коммуникация и делегирование работы другим агентам

Интеллектуальные агенты общаются, сотрудничают и разделяют работу в процессе познания. От одиночного интеллекта к организованному сотрудничеству

У людей структура и коммуникации делают возможной командную работу – роли определяют, кто что делает, а четкая коммуникация гарантирует, что никакие усилия не пропадают втуне.

Те же принципы применимы к многоагентным системам (MAC).

В отсутствие структуры несколько агентов легко могут впасть в замешательство:

- они дублируют работу друг друга;
- они неверно интерпретируют цели;
- они застревают в петлях обратной связи.

Таким образом, успешная многоагентная экосистема зиждется на трех столпах:

- определенные роли – четкое распределение обязанностей;
- эффективная коммуникация – структурированный обмен сообщениями и общий язык;
- протоколы делегирования – правила раздачи и совместного выполнения задач.

Вместе они превращают группу изолированных LLM в слаженный, координированный коллектив, способный рассуждать, вести переговоры и добиваться сложных результатов.

1. Определение ролей: основа многоагентного сотрудничества

В многоагентных системах роли – это не просто названия, они определяют цели, возможности и поведенческие границы.

От роли агента зависит:

- какие задачи он выполняет;
- какую информацию он потребляет или производит;
- с кем и когда он взаимодействует.

Общие ролевые архетипы

Каждая роль выступает в качестве когнитивного модуля – подобно тому, как в человеческом мозге есть специализированные области для восприятия, рассуждения и принятия решений.

2. Модели координации ролей

Разные системы организуют агентов в различные иерархии в зависимости от сложности и цели.

A. Иерархическая модель

Один агент-менеджер контролирует нескольких специализированных исполнителей.

- Идеально подходит для декомпозиции задач и последовательных технологических процессов.
- Простая структура взаимодействия (сверху вниз).

Пример

Планировщик → [Исследователь, Писатель, Критик] → Планировщик агрегирует результаты.

B. Одноранговая модель

Агенты – равноправные участники, координирующие свои действия посредством переговоров или голосования.

- Подходит для исследований, мозговых штурмов и принятия решений на основе консенсуса.
- Повышает разнообразие рассуждений (несколько точек зрения).

Пример

Три независимых агента дебатируют и приходят к оптимальному решению (как в архитектуре искусственного интеллекта «Сообщество разума» или в сократическом ИИ).

C. Гибридная модель

Сочетает иерархию и сотрудничество – кластеры специалистов, координируемых менеджерами.

Это основная структура в таких продвинутых фреймворках, как MetaGPT и CrewAI, где поддерживается баланс между контролем и креативностью.

3. Коммуникация: язык координации

Коммуникация – это нейронная сеть многоагентной системы.

Без нее роли остаются изолированными, и образуются информационные «очаги» (silo).

Агенты взаимодействуют посредством структурированной передачи сообщений, часто на естественном языке (чат между LLM), но в соответствии с определенными протоколами для поддержания порядка.

Ключевые компоненты коммуникации между агентами

Надежный коммуникационный уровень обеспечивает синхронизацию, прослеживаемость и выполнение без избыточности.

4. Коммуникационные протоколы в действии

Рассмотрим типичную беседу между несколькими агентами во время совместного рассуждения.

- Планировщик → Исследователь:

  «Найдите рецензируемые статьи об интерпретируемости LLM, опубликованные после 2023 года».
- Исследователь → Планировщик:

  «Найдено 12 статей. Резюме готовы».
- Планировщик → Критик:

  «Оцени резюме на предмет фактической точности и полноты».
- Критик → Планировщик:

  «Две статьи классифицированы неправильно, исправленные резюме прилагаются».
- Планировщик → Синтезатор:

  «Сведи окончательные резюме в один структурированный раздел».
- Синтезатор → Планировщик:

  «Сделано. Готово к публикации».

Каждое сообщение – шаг в диалоге по координации действий.

Протоколы этих шагов образуют контрольный журнал – уровень объяснимости многоагентного ИИ.

5. Делегирование: как агенты распределяют и назначают задачи

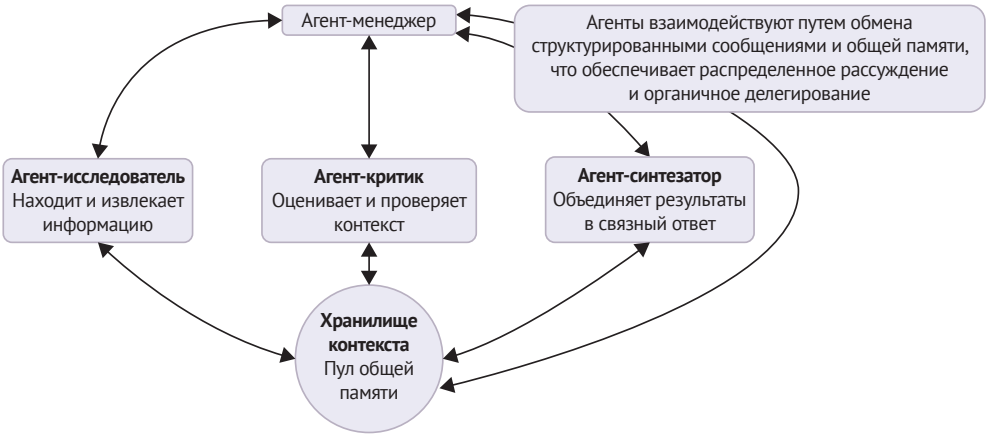
Делегирование – это процесс, при котором один агент назначает другому какую-то подцель или обязанность.

В продвинутых фреймворках это происходит динамически – агенты договариваются о том, кто какую задачу должен выполнять, исходя из своих возможностей и загруженности.

Цикл делегирования

- Определение намерений – агент-менеджер определяет подцели.
- Решение о назначении – он выбирает подходящего агента (исходя из его опыта или доступности).
- Выполнение – получивший назначение агент действует независимо.
- Отчетность – он возвращает результаты или осмысление.
- Оценка – менеджер проверяет частичный результат и включает его в общий.

Этот цикл может повторяться рекурсивно – агент сам может делегировать подзадачи другим агентам, так что образуются вложенные деревья координации.



6. Практический пример: многоагентная писательская команда

Цель: создать научную статью по этическим рискам генеративного ИИ.

Агенты и роли

- Планировщик: определяет общую структуру и основные этапы.
- Исследователь: ищет научные источники.
- Писатель: готовит черновики разделов с использованием найденного контекста.
- Критик: оценивает обоснованность фактами и тональность.
- Редактор: приводит черновик к окончательной форме.

Технологический процесс

- Планировщик раздает исследователям подтемы (конфиденциальность, предвзятость, неправомерное использование).
- Исследователь подбирает статьи для каждой подтемы.
- Писатель составляет черновик раздела.
- Критик оценивает качество и точность.
- Редактор сводит разделы воедино и выдает окончательный вариант.
- Планировщик просматривает и одобряет публикацию.

Каждый агент вносит свой предметно-ориентированный вклад, и в итоге складывается «коллектив авторов».

Этот технологический процесс не прописан в сценарии – он основан на обмене сообщениями и рефлексивной обратной связи.

7. Коммуникационные паттерны в современных фреймворках

В каждом фреймворке своя грамматика диалога, но суть остается неизменной: агенты взаимодействуют, делегируют задачи и осмысляют сообща.

8. Стратегии делегирования: статическая и динамическая

Динамическое делегирование больше похоже на реальную командную работу – оно адаптивное, автономное и учитывает контекст.

Пример диалога (динамическое делегирование)

- Критик → Планировщик:

  «В текущем наборе данных нет записей за 2025 год. Следует ли мне запросить новый поиск?»
- Планировщик → Исследователь:

  «Да, расширь область поиска, включив публикации за 2025 год».

Здесь агенты демонстрирую инициативу, а не слепое послушание.

9. Общая память: становой хребет диалога

В то время как сообщения обеспечивают коммуникацию, общая память обеспечивает непрерывность.

Агенты могут обращаться к центральному хранилищу контекста, которое обычно реализуется следующим образом:

- векторная база данных (семантический поиск);
- граф знаний (структурированные связи);
- журнал сообщений (необработанная история диалога).

Это позволяет:

- сохранять состояние между задачами;
- координировать осмысление (агенты могут критиковать предыдущие результаты);
- повторно использовать контекст (агенты учатся на результатах друг друга).

Пример

Агент-критик обращается к индексу памяти исследователя, чтобы перепроверить факты, избегая избыточного поиска.

10. Возможные причины ошибок при взаимодействии нескольких агентов

Когда коммуникация нарушается, возникает хаос.

Наиболее распространенные проблемы включают:

- потерю сообщений;
- противоречивые инструкции;

- неоднозначность интерпретации;
- рассинхронизацию состояния между агентами;
- циклическую передачу задач и «зацикливание» диалогов;
- информационные перегрузки;
- задержки и тайм-ауты;
- конкурирующие приоритеты;
- отсутствие подтверждения выполнения;
- поврежденный или неполный контекст;
- ошибки маршрутизации сообщений;
- каскадное распространение ошибочных решений;
- чрезмерную зависимость от одного агента;
- потерю памяти о предыдущих шагах;
- конфликтующие цели между агентами;
- неконтролируемую эскалацию задач;
- дублирование работы;
- несогласованность форматов данных;
- отсутствие механизмов разрешения конфликтов;
- накопление устаревшей информации.

Хорошо спроектированные протоколы коммуникации предотвращают подобные сбои и поддерживают координацию системы даже при высокой нагрузке.

11. Когнитивный принцип: интеллект через взаимодействие

Истинный интеллект возникает при взаимодействии агентов, а не внутри них.

Когда несколько рассуждающих систем обмениваются результатами осмысления, оспаривают друг друга и сотрудничают, они генерируют идеи, которые не смогла бы породить ни одна отдельно взятая модель.

Один агент знает ответы.

Много агентов приходят к пониманию.

Такое распределенное осмысление – масштабированное метапознание – является основой коллективного ИИ.

- Роли привносят порядок и специализацию в сотрудничество агентов.
- Коммуникация превращает изолированный интеллект в коллективное рассуждение.
- Делегирование эффективно и динамично распределяет когнитивную нагрузку.
- Общая память обеспечивает непрерывность и предотвращает дублирование.
- Координация между несколькими агентами напоминает командную работу людей – структурированную, но адаптивную.

6.3. Фреймворки: CrewAI, LangGraph и AutoGPT

Блоки современного многоагентного интеллекта

1. Почему фреймворки важны

Проектирование интеллектуальной многоагентной системы с нуля может оказаться неподъемной задачей.

Вам понадобятся компоненты для организации памяти, передачи сообщений, планирования задач, распределения ролей, осмысления и восстановления после сбоев – и при этом система должна быть масштабируемой и объяснимой.

Чтобы решить эту проблему, несколько фреймворков с открытым исходным кодом предлагают готовые уровни оркестрации для многоагентного сотрудничества.

Они предоставляют готовые инструменты для определения ролей, коммуникационных протоколов и когнитивных циклов, чтобы разработчики могли сосредоточиться на проектировании собственно интеллекта, а не на инфраструктуре.

Среди этих фреймворков три занимают особое место:

- CrewAI – ролевая оркестрация командной работы и технологического процесса;
- LangGraph – основанная на графах координация для больших сетей агентов с сохраняемым состоянием;
- AutoGPT – автономия с самозапросами и декомпозицией задач на основе целей.

Посмотрим, чем они отличаются и когда какой использовать.

2. CrewAI – ролевой командный интеллект

CrewAI – это написанный на Python фреймворк с открытым исходным кодом, который позволяет организовывать команды из специализированных ИИ-агентов, называемые «бригадами».

Каждая «бригада» определяется ролями, целями, инструментами и правилами сотрудничества.

Основная идея

CrewAI имитирует работу небольшой компании:

- каждый агент выступает в роли сотрудника (исследователя, писателя, рецензента);
- система определяет проект (цель);
- «бригада» сотрудничает, чтобы автономно выполнить задачу;

Ключевые особенности

- Шаблоны ролей: определение конкретных целей и навыков для каждого агента.

- Упорядочение задач: задачи выполняются в логическом порядке с учетом зависимостей.
- Делегирование: одни агенты могут передавать подзадачи другим.
- Общая память: контекст сохраняется с помощью обобщений или векторных хранилищ.
- Поддержка осмысления: агенты могут критиковать свои или чужие результаты.

Когда использовать CrewAI

- Вам нужно структурированное сотрудничество (например, исследование → написание → редактирование → публикация).
- Ваш технологический процесс последовательный или ролевой.
- Вам нужна ясность и объяснимость ролей агентов.

Пример сценария использования

Формирование команды

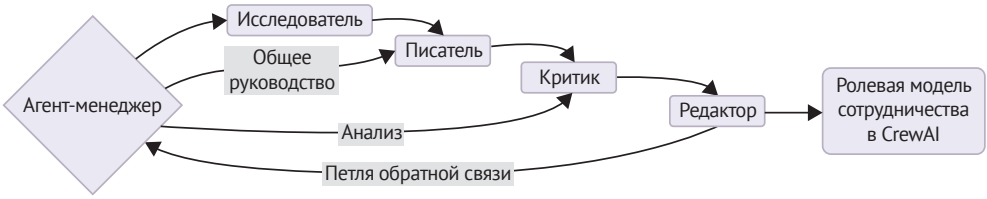
- Агент-исследователь собирает факты.
- Агент-писатель составляет черновик текста.
- Агент-критик оценивает стиль и тональность.
- Агент-редактор дорабатывает окончательную версию.

Преимущества

- Простота реализации.
- Понятные человеку определения ролей на языке YAML.
- Идеально подходит для деловых и креативных конвейеров.

Ограничения

- В меньшей степени подходит для динамичного, неструктурированного сотрудничества.
- Отсутствует оркестрация на уровне графа (агенты работают преимущественно в линейном порядке).



3. LangGraph – координация агентов на основе графа

LangGraph – это продвинутый фреймворк, разработанный группой LangChain.

Вместо того чтобы определять технологические процессы как линейные цепочки, LangGraph моделирует их в виде графов – сети вершин (агентов), соединенных ребрами (коммуникационными путями).

Это делает LangGraph идеальным решением для сложных систем с хранимым состоянием и множеством акторов, где задачи могут разветвляться, образовывать циклы или выполняться параллельно.

Основная идея

Каждый агент является вершиной динамического графа.

Ребра представляют взаимодействие или поток данных между агентами.

Графовый движок управляет передачей сообщений, сохранением состояния и маршрутизацией задач.

Ключевые возможности

- Технологические процессы представлены графами: взаимодействия агентов определяются визуально или программно.
- Хранимое состояние: агенты сохраняют долговременный контекст между запусками.
- Параллельное выполнение: несколько агентов работают одновременно.
- Восстановление после ошибок: циклы и условные предложения позволяют выполнять повторные попытки и ветвление.
- Поддержка участия человека: супервизоры или аналитики могут вмешиваться в процесс в любой момент.

Когда использовать LangGraph

- Вам нужно, чтобы много агентов сотрудничало динамически.
- Вам нужен детальный контроль над маршрутизацией сообщений и состоянием системы.
- Вы создаете ИИ-систему корпоративного уровня с возможностью аудита и управления версиями.

Пример сценария использования

Сеть помощников для научных исследований:

- один агент ищет статьи;
- другой извлекает ключевые результаты;
- третий анализирует тенденции;
- четвертый генерирует визуальные обобщения.

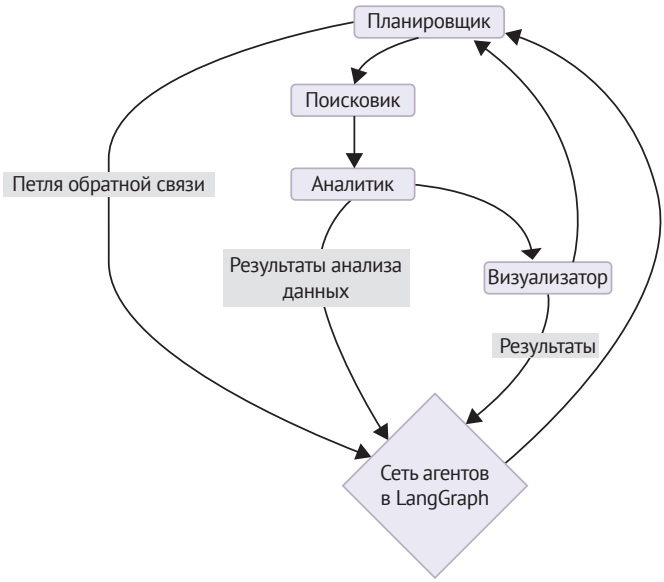
Все агенты общаются с помощью топологии, управляемой LangGraph, что обеспечивает обратную связь и итеративность.

Преимущества

- Высочайшая гибкость и масштабируемость.
- Идеально подходит для длительных технологических процессов с большим объемом данных.
- Интегрируется с экосистемой LangChain (память, инструменты, RAG).

Ограничения

- Требует больше времени на настройку; необходимо вручную определять вершины и ребра.
- Новичкам будет сложнее разобраться.



4. AutoGPT – автономная декомпозиция задач

AutoGPT – одна из первых платформ с открытым исходным кодом, популяризировавших полностью автономных ИИ-агентов.

Вместо того чтобы оркестрировать несколько предопределенных агентов, AutoGPT фокусируется на одном самостоятельном агенте, способном осуществлять декомпозицию задач, выполнять действия и самостоятельно формулировать запросы до тех пор, пока цель не будет достигнута.

Основная идея

Поставьте перед агентом какую-нибудь общую цель (например, «Изучить новые технологии в области электромобилей»), и он:

- разделит цель на подзадачи;
- выполнит их последовательно, используя инструменты;
- будет самостоятельно осмыслять и перепланировать действия до тех пор, пока цель не будет достигнута.

Ключевые особенности

- Механизм самозапросов: агент самостоятельно генерирует и контролирует свои дальнейшие действия.
- Интеграция с инструментами: просмотр веб-страниц, файловый ввод-вывод, API и внешние функции.

- Циклы осмысления: встроенные самооценка и повторение попыток.
- Постоянная память: локальное хранение данных о прогрессе и осмыслении.

Когда использовать AutoGPT

- Вам нужны автономные агенты, способные работать в течение длительного времени и требующие минимального контроля.
- Вы экспериментируете с целями, требующими нестандартного подхода, или с научно-исследовательскими задачами.
- Вы ставите автономность выше предсказуемости.

Пример сценария использования

Бот для маркетинговых исследований, который:

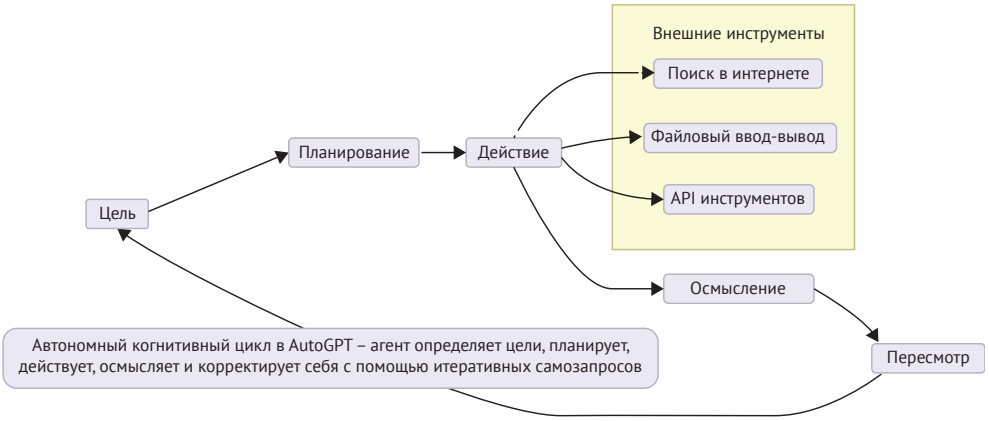
- определяет стратегию исследований;
- собирает данные из интернета;
- анализирует тенденции;
- обобщает результаты и сохраняет их в памяти – и все это без дополнительных инструкций.

Сильные стороны

- Высокая автономность и адаптивность.
- Отлично подходит для исследовательских и творческих задач.
- Минимальные требования к настройке – достаточно указать цель и предоставить инструменты.

Ограничения

- Сложнее контролировать и отлаживать.
- Риск возникновения «неконтролируемых циклов» или нерелевантных действий.
- Не подходит для особо ответственных или находящихся в сфере внимания регуляторов задач, выполняемых без контроля.



5. Выбор подходящего фреймворка

При выборе фреймворка учитывайте структуру своего технологического процесса, возможности команды и допустимый уровень риска.

Зачастую наилучшее решение предполагает сочетание всех трех вариантов.

- Используйте CrewAI для определения ролей в команде.
- Используйте LangGraph для оркестрации коммуникации и обмена данными.
- Встраивайте автономность, как в AutoGPT, для субагентов, которые нуждаются в самостоятельном рассуждении.

Этот гибридный подход обеспечивает одновременно структурированность и гибкость – отличительные черты современных интеллектуальных систем.

6. Когнитивный принцип: структура обеспечивает масштабируемость

Одна интеллектуальная модель – хорошо, но структурированная система обеспечивает более стабильный интеллект.

Определяя, как сотрудничают, общаются и обучаются агенты, фреймворки превращают абстрактное познание в повторяемую и надежную деятельность.

Автономность без структуры – хаос.

Структура без автономности – застой.

Фреймворки обеспечивают баланс между тем и другим, позволяя масштабировать интеллект.

- CrewAI: самый простой в освоении – понятные роли и предсказуемые процессы.
- LangGraph: готов к использованию в корпоративной среде – сложное сотрудничество с хранимым состоянием.
- AutoGPT: экспериментальный – самостоятельное исследование и творческий подход.

- Каждый из этих фреймворков представляет собой отличный от других баланс между контролем и автономностью.
- Диаграммы и модели коммуникации делают эти фреймворки обучаемыми, объяснимыми и воспроизводимыми.

6.4. Архитектура многоагентного рабочего пространства: планировщик, исполнитель, оцениватель

Когнитивная триада интеллектуального сотрудничества

1. Схема координации

В любой высокоэффективной организации, где работают люди, существует определенный ритм планирования, исполнения и оценки.

Руководитель определяет план, команда выполняет, а рецензент оценивает результаты.

Многоагентные системы работают по тому же принципу.

Наиболее стабильной и масштабируемой конструкцией для автономного сотрудничества является архитектура «Планировщик – Исполнитель – Оцениватель» (ПИО) – простая, но эффективная триада, которая обеспечивает структуру, направление и подотчетность агентов.

В этом разделе мы рассмотрим, как спроектировать такое рабочее пространство.

- Агент-планировщик разбивает сложные цели на более простые.
- Агент-исполнитель выполняет задачи.
- Агент-оцениватель обеспечивает качество, согласованность и соответствие требованиям.

Этот цикл с участием трех агентов образует основу интеллектуальной координации – баланса между автономностью и контролем, а также между скоростью и точностью.

2. Обзор модели «Планировщик – Исполнитель – Оцениватель»

По сути своей модель ПИО представляет замкнутый когнитивный цикл, который обеспечивает непрерывное совершенствование и исправление ошибок.

Основной процесс

- Планировщик интерпретирует общую цель пользователя → разбивает ее на выполнимые задачи.
- Исполнитель выполняет каждую задачу с помощью инструментов, API или цепочек рассуждений.
- Оцениватель анализирует результаты → выявляет проблемы или пробелы → отправляет отзыв Планировщику.

Этот цикл с обратной связью продолжается до тех пор, пока конечный результат не будет соответствовать заданным критериям.

Почему это работает

- Предотвращает каскады галлюцинаций (когда плохие результаты приводят к еще худшим).
- Позволяет разделить роли – каждый агент фокусируется на одной когнитивной функции.
- Поддерживает итеративное улучшение и самокоррекцию в режиме реального времени.



3. Агент-планировщик

Агент-планировщик выступает в роли руководителя проекта – он преобразует абстрактные цели в четкие и выполнимые планы.

Обязанности

- Интерпретация намерений пользователя или запроса верхнего уровня.
- Разбивка задачи на структурированные подзадачи.
- Назначение подзадачи подходящим агентам.
- Определение критериев оценки и ожидаемых результатов.
- Корректировка плана на основе отзывов Оценивателя.

Входы

- Запрос пользователя или цель всей миссии.
- Текущий контекст или состояние памяти.

Выходы

- Список или последовательность задач.
- Инструкции по делегированию задач Исполнителям.

Пример

Цель: обобщить последние изменения в климатической политике.

План

- Найти последние публикации правительства (Исполнитель 1).
- Выявить различия в политике, произошедшие с 2023 по 2025 год (Исполнитель 2).
- Проверить источники данных (Оцениватель).
- Сгенерировать структурированное резюме.

Рекомендации

- Делайте план модульным – задачи должны быть атомарными и допускающими измерение.
- Сохраняйте метаданные задач в общей памяти для обеспечения прослеживаемости.
- Всегда указывайте ожидаемый формат вывода (текст, JSON, таблица и т. д.).



4. Агент-исполнитель

Исполнитель – это механизм осуществления действий. Он выполняет назначенную ему задачу, используя рассуждение, API или внешние инструменты.

Обязанности

- Автономно выполнять назначенные подзадачи.
- Извлекать или обрабатывать данные (с помощью веб-интерфейса, API, кода или RAG).
- Генерировать результаты в требуемом формате.
- Сообщать Оценивателю о завершении и результатах.

Входы

- Инструкции по выполнению задачи от Планировщика.
- Контекстные данные или состояние памяти.

Выходы

- Результаты выполнения задачи.
- Журналы или промежуточные артефакты.

Пример

Задача: найти 10 лучших новостных статей о квантовых вычислениях, опубликованных в этом месяце.

Результат: список заголовков, URL-адресов и кратких резюме.

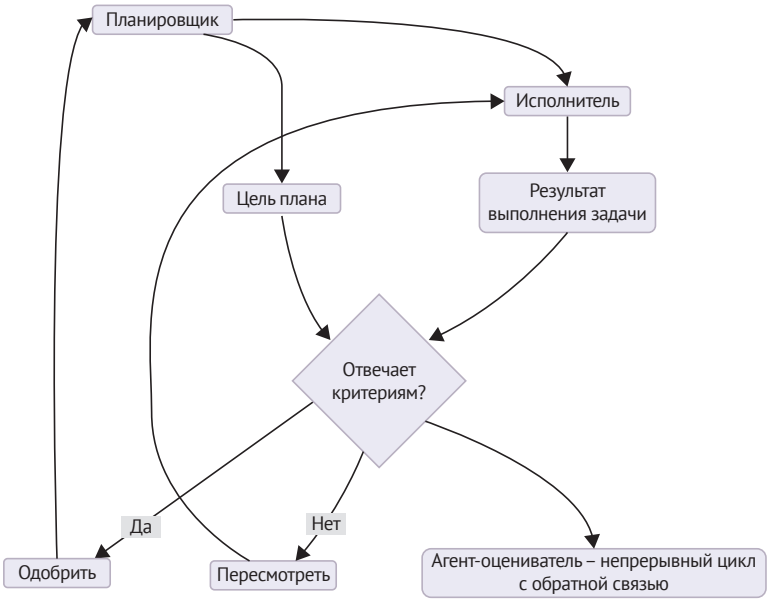
Ключевое преимущество

Исполнители составляют план.

Они являются мостом между рассуждениями и действиями, превращая структурированные цели в осязаемые результаты.

Рекомендации

- Оснастите Исполнителей специализированными инструментами (поисковыми API, парсерами данных, исполнителями кода).
- Следите за строгой обоснованностью, чтобы избежать галлюцинаций.
- Реализуйте тайм-ауты и логику повтора для долго работающих задач.



5. Агент-оцениватель

Оцениватель следит за тем, чтобы система не доверяла слепо своей собственной работе.

Он действует как уровень контроля качества, который проверяет точность, связность изложения и соответствие первоначальному намерению.

Обязанности

- Сравнивать результат Исполнителя с намерением Планировщика.
- Обнаруживать фактические ошибки, отсутствующий контекст или логические противоречия.
- Предоставлять структурированный отзыв для пересмотра.
- Факультативно проставлять оценку степени уверенности или качества.

Входы

- Результат выполнения задачи Исполнителем.
- Спецификация задачи от Планировщика.

Выходы

- Отчет о результатах проверки (проблемы, предложения по улучшению).
- Факультативный запрос на пересмотр или флаг одобрения.

Пример

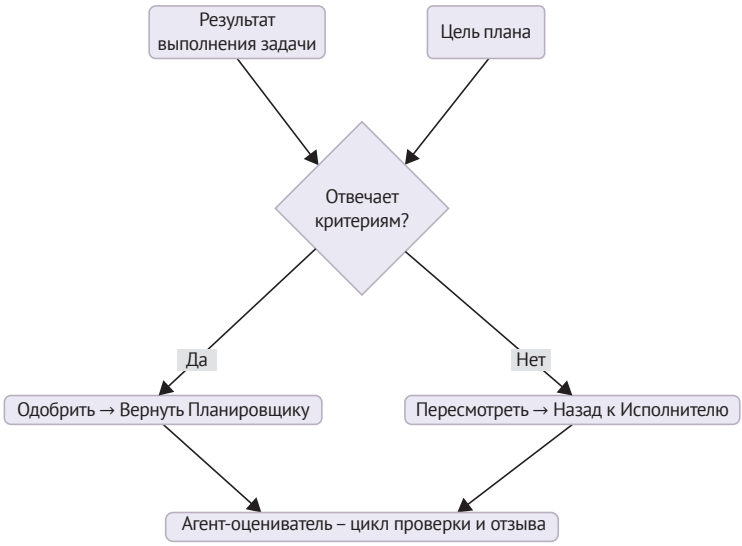
Резюме не противоречит фактам, но не хватает ссылок в разделах 3 и 4. Рекомендуется добавить еще 2 ссылки.

Методы оценивания

- Оценка на основе запроса: LLM выступает в роли критика самой себя, используя структурированные шаблоны.
- Эвристические проверки: проверка по регулярным выражениям или на основе API (диапазонов дат, ссылок).
- Межагентное сравнение: несколько Оценивателей голосуют или стремятся достичь консенсуса.

Рекомендации

- Будьте беспристрастны – Оцениватель не должен напрямую переписывать контент.
- Сосредоточьтесь на допускающих измерения критериях: обоснованность фактами, структура, полнота.
- Сохраняйте метаданные оценки для будущего обучения осмыслению.



6. Соберем все вместе: когнитивная триада

Планировщик и Оцениватель работают в непрерывном когнитивном цикле с обратной связью, имитируя целенаправленную работу команды людей.

Пример технологического процесса

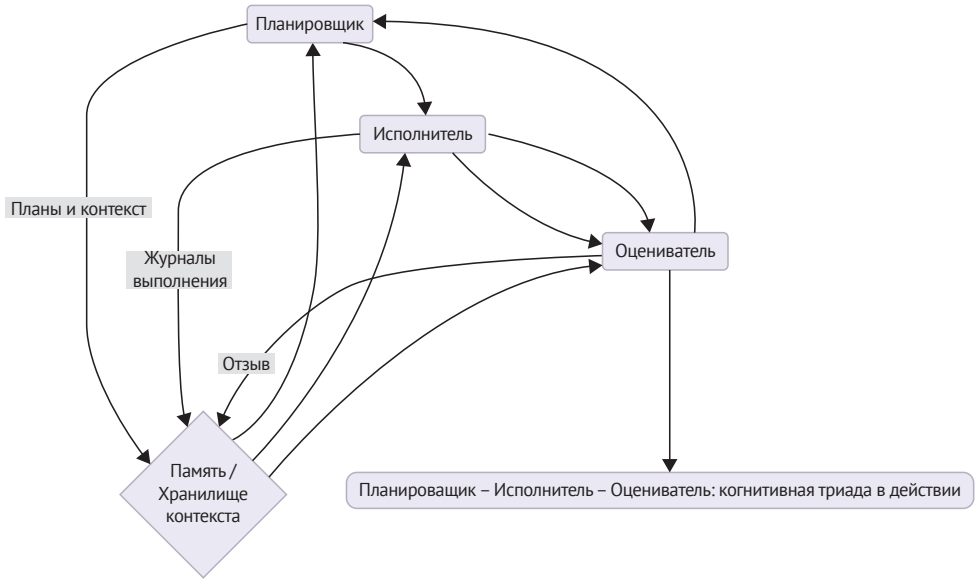
Планировщик: «Цель: подготовить отчет по анализу рынка электромобилей».
Исполнитель: собирает данные о продажах, проводит анализ, составляет резюме.

Оцениватель: проверяет точность, отсутствующие показатели или устаревшие источники.

Планировщик: пересматривает цели или запрашивает дополнительные данные.

Цикл продолжается до тех пор, пока не будет достигнут заданный уровень качества.

Эта модель обеспечивает как глубину, так и надежность – каждая итерация повышает точность и понимание контекста.



7. Пример реализации (упрощенный псевдокод)

```
def
    plan = planner_agent.create_plan(goal)
    while True:
        result = executor_agent.execute(plan)
        feedback = evaluator_agent.evaluate(result, plan)
        if feedback["verdict"] == "approve":
            return result
        else:
            plan = planner_agent.revise_plan(feedback)
```

Этот короткий цикл заключает в себе суть агентного интеллекта – планирование, действия и осмысление до тех пор, пока процесс не сойдется.

8. Когнитивный принцип: триадный интеллект

Эта триада отражает фундаментальное утверждение о природе познания: интеллект – не знание, а планирование, выполнение и совершенствование.

Встраивая планирование, действия и осмысление в индивидуальные, но связанные между собой агенты, мы достигаем распределенного познания – системы, способной мыслить, проверять и развиваться как коллектив.

- Модель «Планировщик – Исполнитель – Оцениватель» обеспечивает устойчивый фундамент многоагентного проектирования.
- Каждый агент выполняет одну когнитивную функцию: формулирование стратегии, действие или критику.

- Замкнутый цикл с обратной связью гарантирует непрерывное совершенствование.
- Общая память обеспечивает передачу знаний между итерациями.
- Эта триадная структура обеспечивает баланс между автономностью и контролем, что важно для достойного доверия ИИ.

6.5. Разрешение конфликтов и кооперативная память

Интеллектуальные агенты расходятся во мнениях, ведут переговоры и учатся вместе

1. Когда сотрудничество превращается в соперничество

В многоагентной среде разногласия – не ошибка, а признак интеллекта.

Когда автономные агенты анализируют данные, выбирают разные пути рассуждения или критикуют результаты работы друг друга, конфликты неизбежны.

Эти конфликты могут быть логическими («Какой вывод правильный?»), процедурными («Кто должен действовать следующим?») или эпистемическими («Какой источник надежнее?»).

В отсутствие контроля конфликты ведут к хаосу – дублированию работы, несогласованным состояниям или бесконечным циклам с обратной связью.

Но при продуманном проектировании конфликт становится катализатором совершенствования.

Именно здесь на помощь приходят разрешение конфликтов и кооперативная память – два основных механизма проектирования, которые позволяют многоагентным системам развиваться гармонично, не порождая разрушений.

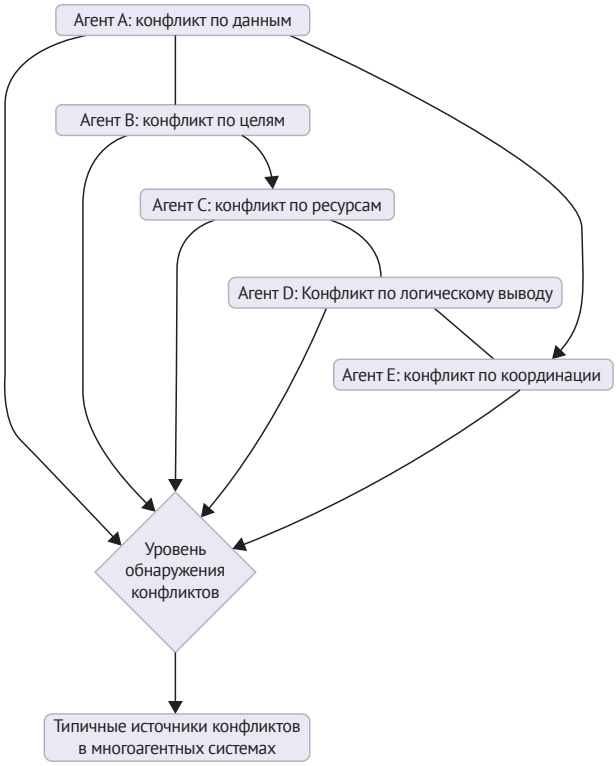
2. Природа конфликтов в многоагентных системах

Не все конфликты одинаковы. В многоагентных средах конфликты могут возникать на нескольких уровнях:

- конфликт целей – агенты стремятся к несовместимым результатам;
- конфликт ресурсов – несколько агентов претендуют на одни и те же вычислительные или информационные ресурсы;
- конфликт приоритетов – различие в оценке важности задач;
- конфликт знаний – агенты обладают противоречивыми данными или выводами;
- конфликт интерпретаций – одинаковая информация трактуется по-разному;
- конфликт ролей – пересечение обязанностей и зон ответственности;
- временной конфликт – несовпадение последовательности или момента выполнения действий;
- конфликт стратегий – различие методов достижения цели;
- конфликт полномочий – неопределенность в принятии окончательного решения;

- конфликт памяти и контекста – агенты используют разные версии состояния системы;
- конфликт коммуникационных протоколов – несовместимость форматов или правил обмена сообщениями;
- конфликт доверия – сомнения в корректности выводов других агентов;
- конфликт планирования – несовместимые планы действий;
- конфликт распределения задач – дублирование или перекрытие работы;
- конфликт ограничений – локальные ограничения агента противоречат глобальным требованиям системы;
- конфликт оценки рисков – агенты по-разному воспринимают допустимый уровень риска;
- конфликт оптимизации – улучшение локальной метрики ухудшает общую эффективность системы;
- конфликт координации – действия агентов теряют согласованность;
- конфликт ответственности – неясно, какой агент должен исправлять ошибку;
- конфликт эскалации – агенты взаимно усиливают ошибочные решения вместо их коррекции.

Конфликты, если они разрешаются системно и последовательно, усиливают коллективное мышление, превращая противоречия в согласованный консенсус.



3. Роль посредника или агента-координатора

Для предотвращения конфликтов в многоагентных системах часто используется агент-посредник – беспристрастный координатор, который обнаруживает, анализирует и разрешает конфликты между агентами.

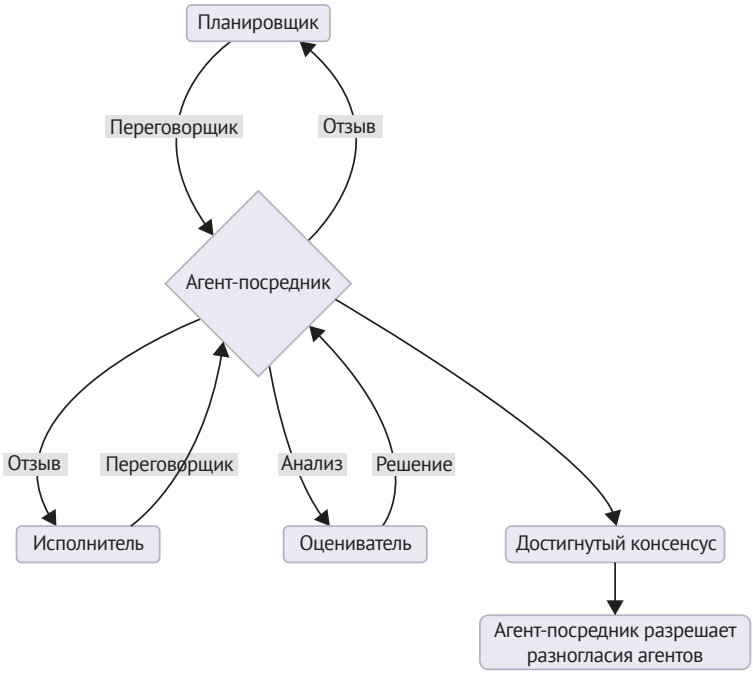
Обязанности

- Отслеживать сообщения между агентами на предмет противоречий.
- Выявлять расхождения в рассуждениях или результатах.
- Инициировать протоколы переговоров или арбитража.
- Регистрировать детали конфликтов для последующего анализа.

Пример

Если два агента порождают противоречивые результата анализа, то Посредник просит каждого обосновать свои рассуждения, сравнивает их доводы и решает, какой результат соответствует глобальной цели системы.

Типичные стратегии разрешения конфликтов



4. Кооперативная память: основа коллективного обучения

Если разрешение конфликтов способствует достижению мира, то кооперативная память способствует прогрессу.

Этот механизм позволяет агентам делиться тем, чему они научились, предотвращая тем самым повторяющиеся ошибки и закрепляя успешные стратегии.

Определение

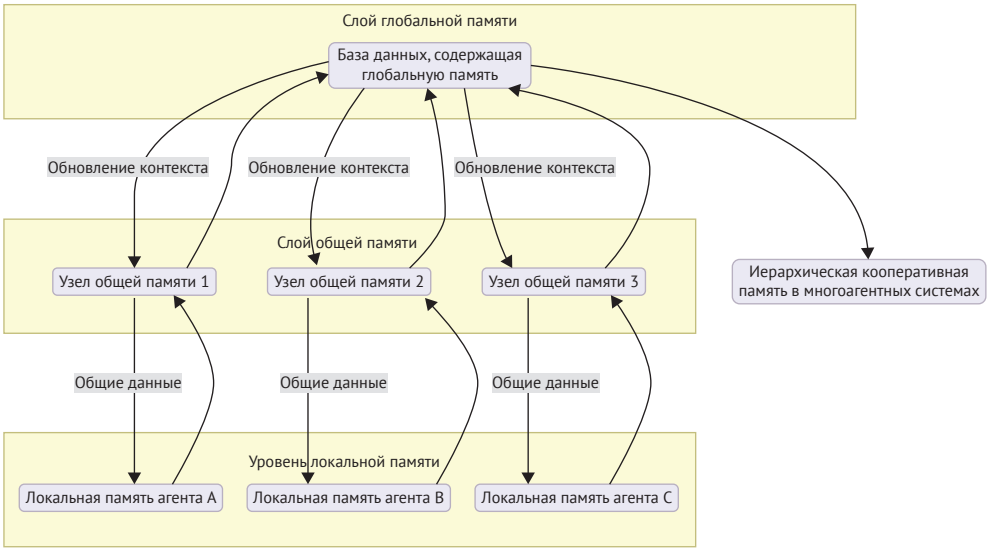
Кооперативная память – это общее структурированное хранилище, в котором агенты хранят коллективный опыт. Оно объединяет факты, осмысления и прошлые разрешения конфликтов.

Она работает как коллективная память сообщества агентов, гарантируя, что система не будет многократно учиться на одних и тех же ошибках.

5. Слои памяти при сотрудничестве нескольких агентов

Рекомендации по проектированию

- Используйте векторные хранилища для семантического поиска (например, Chroma, FAISS).
- Используйте графы знаний для структурированного, реляционного понимания.
- Реализуйте управление доступом (не все агенты должны видеть все).



6. Как совместная память улучшает координацию

Предотвращение конфликтов

- Прежде чем предпринимать какие-то действия, агенты справляются с общей историей, что сокращает количество избыточных или противоречивых действий.

Обоснованность фактами

Все агенты обосновывают свои рассуждения одной и той же базой найденных знаний.

Более быстрая сходимость

Оцениватели и планировщики могут обратиться к предыдущим отзывам, чтобы быстро уточнить планы.

Накопительный интеллект

Память преобразует изолированные действия агентов в развивающуюся группу.

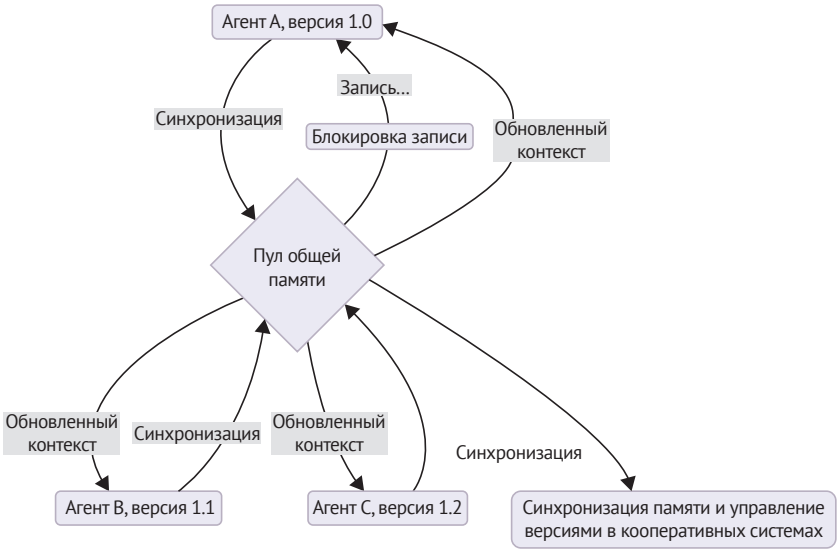
Пример

После нескольких итераций агент-исследователь запоминает, какие источники данных ранее были помечены Оценивателем как «ненадежные», и автоматически отфильтровывает их в будущем.

7. Синхронизация памяти и управление версиями

Для обслуживания памяти имеются протоколы синхронизации, которые гарантируют, что агенты не будут перезаписывать обновления друг друга или работать с устаревшими данными.

Надежная система синхронизации памяти гарантирует согласованность коллективных рассуждений даже в распределенных средах.



8. Разрешение конфликтов и кооперативная память

Эти две концепции подкрепляют друг друга:

- разрешение конфликтов привносит краткосрочную стабильность;
- кооперативная совместная память привносит долгосрочную гармонию.

Когда возникает конфликт:

- Посредник документирует разногласие;
- Оцениватель оценивает достоверность и рассуждение;

- решение и обоснование сохраняются в общей памяти;
- будущие агенты ссылаются на эту запись, чтобы предотвратить повторение конфликта.

Со временем система накапливает память о разрешенных конфликтах – саморазвивающуюся структуру, которая постулирует, что означает «истина» в рамках данной экосистемы.



9. Схема реализации (концептуальный псевдокод)

```
def
    plan = planner.create_plan(goal)
    result = executor.execute(plan)
    feedback = evaluator.evaluate(result)
    if feedback["verdict"] == "disagree":
        resolution = mediator.resolve_conflict(result, feedback)
        memory.store_resolution(resolution)
        return planner.replan_with_context(resolution)
    else:
        memory.update_with_success(goal, result)
        return result
```

Такой дизайн гарантирует, что любое разногласие либо улучшает план, либо обогащает общий интеллект системы.

11. Когнитивный принцип: память дарит гармонию

Сотрудничество между людьми процветает, потому что мы помним прошлые ошибки, общие уроки и решения, которые сформировали нашу коллективную мудрость.

Многоагентные системы достигают такой же стабильности благодаря кооперативной памяти.

Память превращает соперничество в сотрудничество.

Она наставляет агентов не просто действовать, а учиться.

Конфликт не провал, а способ познания: в сочетании со структурированной памятью он становится двигателем коллективного разума.

- Конфликты между агентами естественны и необходимы – они выявляют когнитивное разнообразие.
- Агенты-посредники и структурированные протоколы переговоров предотвращают тупиковые ситуации.
- Кооперативная память преобразует изолированные действия в коллективные знания.

- Вместе они создают самокорректирующиеся, самообучающиеся экосистемы агентов.
- Будущее сотрудничества в рамках ИИ заключается не в том, чтобы избегать разногласий, а в том, чтобы помнить, как их разрешать.

6.6. Агент в действии: создаем многоагентную «команду ИИ-стартапа» (Исследователь, Разработчик, Рецензент)

Модель ИИ-организации, которая совместно думает, кодирует и совершенствуется

1. От программистов-одиночек до автономных команд

Представьте, что вы могли бы создать виртуальный стартап, полностью укомплектованный интеллектуальными агентами, – каждый со своим опытом, стилем общения и подотчетностью.

В этой системе один агент изучает тенденции рынка, другой пишет код производственного уровня, а третий анализирует и совершенствует работу перед развертыванием.

Это не научная фантастика, а практически применимый паттерн проектирования, который стал возможен благодаря современным многоагентным фреймворкам.

В этом разделе мы создадим упрощенную, но реалистичную команду ИИ-стартапа, состоящую из трех сотрудничающих агентов:

- агент-исследователь: собирает знания, идеи и требования;
- агент-разработчик: реализует план в виде кода или функциональных результатов;
- агент-рецензент: тестирует, критикует и проверяет результаты.

Эта система с тремя агентами отражает работу реального стартапа – постановка задачи → выполнение → анализ – в непрерывном цикле совершенствования.

2. Цель

Мы создадим мини-компанию в коде – автономную ИИ-команду, способную:

- выявлять проблему пользователя;
- исследовать потенциальное решение с использованием ИИ;
- разработать прототип (например, демонстрационный код);
- проанализировать его и доработать с целью повысить точность и улучшить стиль;
- документировать процесс в общей памяти для будущих запусков.

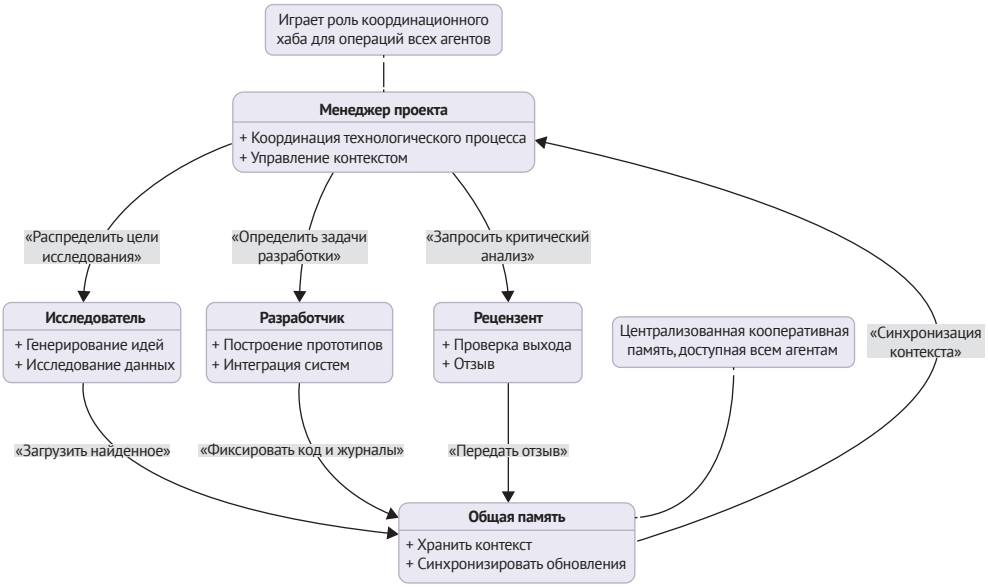
Этот паттерн может обеспечить автономные исследования и разработки, создание контента, быстрое прототипирование и даже моделирование разработки продукта.

3. Обзор архитектуры

Команда ИИ-стартапа придерживается тех же принципов когнитивного проектирования, что и команды людей, – четкие роли, протоколы общения и общая память.

Агенты и роли

Каждый агент взаимодействует с другими посредством цикла координации задач, который инициируется центральным оркестратором или «Агентом – менеджером проекта», отвечающим за контекст.



4. Коммуникация и процесс выполнения задачи

Агенты выполняют структурированный цикл взаимодействия под управлением менеджера проекта.

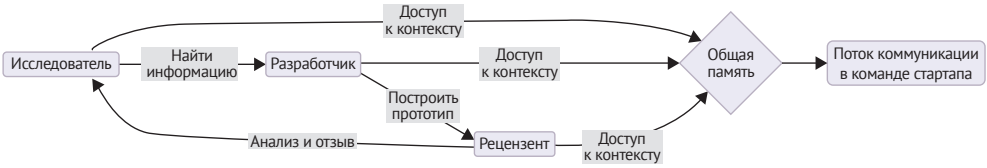
Технологический процесс

- Пользователь/менеджер: определяет цель проекта.
- Исследователь: собирает информацию, ссылки или наборы данных.
- Разработчик: создает прототип или реализует логику на основе отчета Исследователя.
- Рецензент: оценивает результаты, сообщает о проблемах и запрашивает улучшения.
- Менеджер: обновляет общую память и координирует следующую итерацию.

Каждое сообщение имеет следующее информационное наполнение:

- отправитель, получатель;
- намерение (задача, отзыв, вопрос);
- контент (данные, результаты или осмысление)

Это создает прозрачный след общения между агентами, пригодный для аудита.



5. Пример проекта: создаем анализатор эмоциональной окраски

Рассмотрим реалистичный проект – инструмент для анализа эмоциональной окраски.

Цель

Разработай инструмент на Python, который анализирует отзывы клиентов и классифицирует их эмоциональную окраску как положительную, нейтральную или отрицательную.

Шаг 1. Агент-исследователь

- Ищет подходы к анализу эмоциональной окраски.
- Обобщает плюсы и минусы различных моделей (например, BERT, VADER).
- Рекомендует подход, основанный на доступных инструментах.
- Порождает краткий отчет для Разработчика.

Пример вывода

Рекомендуемый подход: использовать VADER для быстрой оценки эмоциональной окраски (графический процессор не требуется). Вход: текст. Выход: оценка эмоциональной окраски. Библиотеки: nltk, vaderSentiment.

Шаг 2. Агент-разработчик

- Читает отчет Исследователя.
- Генерирует код на Python, реализующий решение.
- Использует память, чтобы найти паттерны кода, оказавшиеся успешными ранее.

Пример вывода

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()

def analyze_sentiment(text):
    score = analyzer.polarity_scores(text)
    if score['compound'] >= 0.05:
        return "Positive"
```

```
elif score['compound'] <= -0.05:
    return "Negative"
else:
    return "Neutral"
```

Шаг 3. Агент-рецензент

- Тестирует функцию на примерах входных данных.
- Проверяет точность, производительность и стиль написания кода.
- Формирует отзыв.

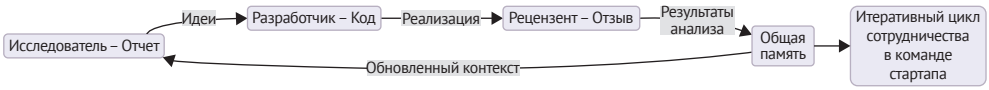
Пример отзыва

Код работает корректно. Предлагаю добавить обработку исключений для нестроковых входных данных и включить документацию для удобства чтения.

Шаг 4. Итеративное улучшение

- Отзывы рецензента хранятся в общей памяти.
- Разработчик вносит изменения в код.
- Исследователь может добавлять новые методы (например, интегрировать модель на основе трансформера).

Каждая итерация повышает качество результатов и сплоченность команды, создавая автономный жизненный цикл разработки.



6. Общая память и обучение

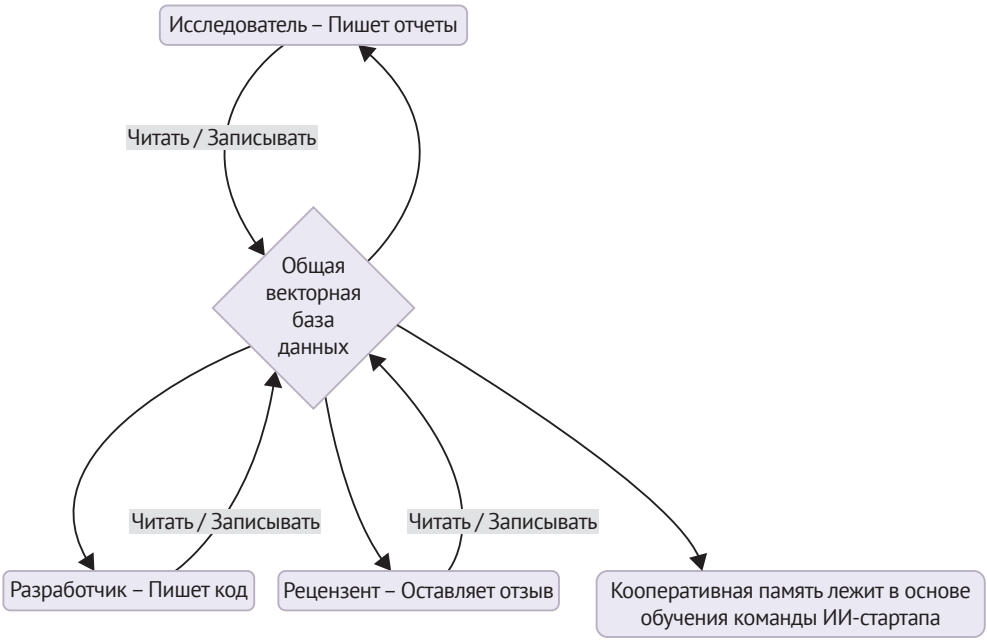
Команда поддерживает общую векторную память для хранения:

- заметок об исследованиях;
- фрагментов кода и предыдущих версий;
- комментариев рецензентов и показателей производительности;
- разрешенных конфликтов (например, споров о точности модели).

Эта память становится коллективной базой знаний ИИ-стартапа, благодаря чему будущие проекты можно будет начинать с более высокого уровня знаний.

Ключевые советы по дизайну

- Используйте облегченное векторное хранилище типа Chroma или FAISS, чтобы поиск осуществлялся быстрее.
- Помечайте записи идентификаторами задачи, агента и итерации.
- Предоставляйте Оценивателю/Рецензенту доступ к полным историческим журналам для обнаружения предвзятости.

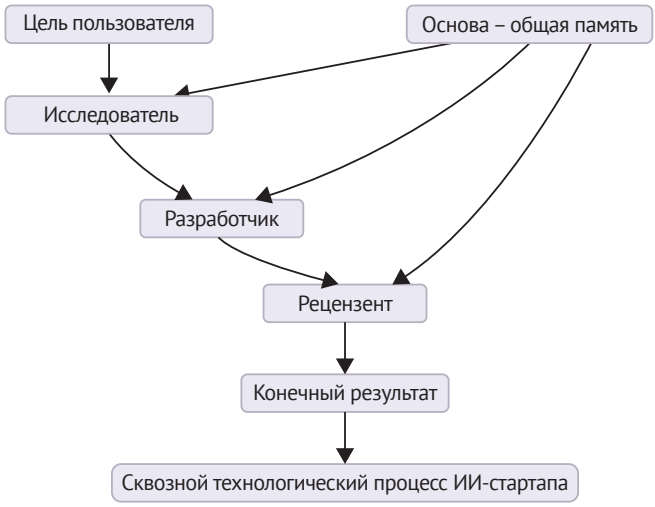


7. Набросок реализации (упрощенной)

```
def
    memory = VectorMemory()
    researcher = ResearchAgent(memory)
    developer = DeveloperAgent(memory)
    reviewer = ReviewerAgent(memory)
    research = researcher.investigate(goal)
    code = developer.build(research)

    feedback = reviewer.review(code)
    memory.store({
        "goal": goal,
        "research": research,
        "code": code,
        "feedback": feedback
    })
    if "improve" in feedback.lower():
        revised_code = developer.revise(code, feedback)
        reviewer.review(revised_code)
    return code
```

Эта структура отражает полный цикл совместного сотрудничества нескольких агентов – модульный, расширяемый и допускающий проверку.



8. Когнитивный принцип: организация – это интеллект

Стартапы, созданные людьми, процветают благодаря разделению труда, обратной связи и обучению.

Та же когнитивная архитектура, реализованная с помощью ИИ, порождает эмерджентный интеллект – коллектив более творческий, надежный и адаптивный, чем любая отдельно взятая модель.

Агент-одиночка может думать.

Команда агентов может создавать.

Этот принцип определяет будущее ИИ-систем: кооперативный интеллект – когда рассуждения распределяются, память используется совместно, а творчество становится коллективным.

Ключевые выводы

- Многоагентные команды стартапов моделируют поведение реальной организации.
- Роли (Исследователь, Разработчик, Рецензент) аналогичны встречающимся в технологических процессах с участием людей.
- Общая память позволяет осуществлять обучение на нескольких проектах.
- Циклы взаимодействия обеспечивают подотчетность и самосовершенствование.
- Модель естественным образом масштабируется, становясь образцом для автономных предприятий.

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7 (499) 322–19–38,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Мира С. Девлин

Построение LLM-агентов с помощью RAG, графов знаний и осмысления

Главный редактор *Яценков В. С.*
editor@dmkpress.com

Перевод *Слинкин А. А.*
Корректор *Абросимова Л. А.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 16,09. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com



Создавайте проактивные ИИ-системы, способные сотрудничать и развиваться

В книге детально рассматривается принципиально новая архитектура искусственного интеллекта, когда сети специализированных LLM-агентов координируют действия, рассуждают, осмысляют и оптимизируют себя, достигая результата, недоступного одиночной модели.
Вы научитесь создавать кластеры агентов, системы с общей памятью, уровни маршрутизации, средства оценки и инструментальные центры, которые динамически адаптируются и постоянно совершенствуются.

Основные темы книги:

- **объединение агентов LLM в кластеры с четкими ролями, возможностями и протоколами связи;**
- **децентрализованные системы маршрутизации для быстрой масштабируемой передачи сообщений;**
- **системы общей векторной памяти для постоянного хранения, обоснования и объединения контекста;**
- **циклы осмысления и оптимизации, которые помогают агентам корректировать свои действия и обучаться на результатах;**
- **инструментальная оркестрация с использованием API, вызовов функций и внешних технологических процессов;**
- **уровни мониторинга, оценки и телеметрии для обеспечения безопасности, надежности и прозрачности вашей многоагентной системы;**
- **масштабируемые топологии системы для производственных задач.**

Издание предназначено инженерам по ИИ, разработчикам приложений с интегрированным ИИ, исследователям и всем желающим выйти за рамки простых запросов и освоить настоящую инженерию LLM.

Мира С. Девлин – исследователь и проектировщик систем искусственного интеллекта, специализирующаяся на архитектуре больших языковых моделей, генерации с дополненным поиском (RAG) и агентном интеллекте. В ее работах соединяются когнитивный дизайн, рассуждающие системы и прикладное машинное обучение.

ISBN 978-5-93700-481-9

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф