1 Концепция и архитектура LLM-двойника

Мы твердо убеждены в том, что лучший способ понять работу *больших языковых моделей* (large language model, LLM) и принципы промышленного *машинного обучения* (machine learning, ML) — это не ограничиваться теорией, а создавать реальные системы. В книге мы покажем, как построить LLM-двойника — интеллектуального агента, способного писать в стиле конкретного человека, воспроизводя его манеру, голос и индивидуальность. На этом примере мы последовательно рассмотрим все стадии жизненного цикла ML-системы — от сбора данных до развертывания и мониторинга. Большинство концепций, которые вы освоите при реализации LLM-двойника, будут применимы и в других проектах, основанных на LLM или машинном обучении.

С инженерной точки зрения перед реализацией нового продукта необходимо пройти три этапа планирования. В первую очередь важно понять, какую задачу мы решаем и что именно собираемся создавать. В нашем случае следует определиться с тем, что представляет собой LLM-двойник и зачем он нужен. На этом шаге мы сосредотачиваемся на вопросе «Зачем?».

На втором этапе мы должны спроектировать первую версию продукта с минимальной функциональностью, чтобы приблизить процесс разработки к реальным условиям. Здесь нам нужно четко определить ключевые функции, необходимые для создания работоспособного и полезного продукта. Выбор зависит от сроков, доступных ресурсов и уровня подготовки команды. Таким образом, мы должны ответить на вопрос «Что именно мы собираемся построить?».

На третьем этапе мы проектируем систему, определяя ее архитектуру и принимая ключевые инженерные решения, лежащие в основе приложения на базе LLM. Обратите внимание: первые две стадии ориентированы на сам продукт, а третья имеет техническую направленность и отвечает на вопрос «Как?».

Эти три этапа естественным образом встраиваются в процесс создания реального продукта. Хотя первый и второй не имеют непосредственного отношения

к области машинного обучения, их необходимо пройти, чтобы разобраться в том, *как* строить продукт.

К концу главы у вас сложится ясное представление о том, что именно вы будете создавать при работе с книгой.

Понимание концепции LLM-двойника

Первым делом необходимо сформировать четкое видение того, что именно мы хотим создать и почему это ценно. Концепция LLM-двойника является новой, и, прежде чем переходить к техническим деталям, важно понять, что представляет собой этот двойник, чего от него следует ожидать и как он должен работать. Осознание конечной цели существенно облегчает усвоение теории, кода и инфраструктурных решений, представленных в книге.

Что такое LLM-двойник

Если в двух словах, то LLM-двойник — это интеллектуальный агент, встраивающий ваш стиль письма, голос и индивидуальность в большую языковую модель, то есть в сложную ИИ-модель. Можно сказать, что это цифровая *проекция* вашей личности. В отличие от универсальной модели, обученной на содержимом Интернета, LLM-двойник тонко настраивается на данных о вас самих. Поскольку любая ML-модель отражает те данные, на которых она обучалась, LLM-двойник будет воспроизводить ваш стиль письма, голос и особенности личности. Мы сознательно используем здесь слово «проекция». Любая проекция сопряжена с потерей значительной части информации, поэтому созданная модель не станет *вами*, но будет воспроизводить тот аспект вашей личности, который представлен в обучающих данных.

Важно понимать, что LLM отражает обучающие данные. Если вы «скормите» ей произведения Шекспира, она начнет писать в его манере. Если обучите модель на песнях Билли Айлиш, она будет создавать тексты, похожие на ее творчество. Эта концепция, известная как перенос стиля, широко используется и при генерации изображений, например, когда вы хотите изобразить кошку в стиле Ван Гога. Мы тоже применим эту стратегию, только стиль будет не заимствованным, а нашим собственным.

Для адаптации LLM к определенному стилю мы не только выполним тонкую настройку, но и задействуем различные продвинутые методы *генерации, дополненной поиском* (retrieval-augmented generation, RAG), используя эмбеддинги, чтобы направлять процесс авторегрессии сквозь данные о нас самих.

Методы тонкой настройки (доообучения) мы подробно рассмотрим в главе 5, а подходы RAG — в главах 4 и 9, а пока приведем несколько примеров, проясняющих изложенные выше идеи.

Вот на чем можно дообучить LLM, чтобы сделать из нее своего двойника:

- *публикации в социальных сетях* — настройка LLM на создание контента для социальных сетей;
- *переписка с друзьями и членами семьи* — адаптация LLM к вашему неформальному стилю общения;
- *научные статьи и публикации* — калибровка LLM для написания формального и образовательного контента;
- *программный код* — настройка LLM на генерацию кода в вашей манере.

Все эти сценарии сводятся к одной и той же базовой стратегии: собрать ваши цифровые данные и «скормить» их LLM с помощью соответствующих алгоритмов. В итоге модель будет отражать стиль, характерный для представленных данных. На первый взгляд все просто.

Но, к сожалению, здесь возникают как технические, так и этические вопросы. Как получить доступ к нужным данным? Достаточно ли у нас цифровых следов для создания LLM-двойника? Какие типы данных действительно полезны? Правильно ли вообще создавать такую копию себя? Хотим ли мы, чтобы кто-то — или что-то — подражал нам? Будет ли модель действительно использовать наш стиль или просто имитировать его?

Помните: цель этого раздела — ответить на вопрос «Зачем?», а не «Что?» и «Как?». Поэтому давайте обсудим, в чем ценность LLM-двойника и при каких условиях его создание может быть этически оправданно.

Причины для создания LLM-двойника

Для инженера (как и для любого другого специалиста) создание личного бренда зачастую оказывается более ценным, чем создание обычного резюме. Однако основная трудность заключается в том, что написание материалов для профильных площадок отнимает много времени. Даже если вам нравится писать и делиться идеями, рано или поздно вы столкнетесь с нехваткой времени или вдохновения и почувствуете, что вам требуется помощник. Мы не стремимся превратить этот раздел в рекламу, а лишь проясняем масштаб и задачи проекта, который нам предстоит реализовать.

Мы хотим создать LLM-двойника, который сможет писать персонализированный контент для социальных сетей, используя ваш уникальный стиль. Этот инструмент не предполагается использовать в неэтичных целях — он будет выполнять роль соавтора и помощника в рамках письменной коммуникации.

На основе описанных в книге техник вы сможете самостоятельно адаптировать модель для решения других задач, однако в рамках рассматриваемого примера мы сосредоточимся на генерации контента для социальных сетей и блогов. Благодаря этому вместо написания текста с нуля вы сможете передать LLM-двойнику основную идею и позволить ему сделать за вас всю рутинную работу.

В конечном счете нам все равно придется проверять, все ли работает корректно, и преобразовывать результаты в желаемый формат (подробнее об этом — в разделе «Планирование MVP для LLM-двойника» далее). Таким образом, мы проецируем себя в LLM-двойника — помощника, который будет автоматизировать наш процесс написания текстов. Эта конкретная LLM, скорее всего, окажется неэффективной в других сценариях, потому что мы специально адаптируем ее под одну задачу с помощью методов тонкой настройки, промпт-инжиниринга и RAG.

Итак, создание LLM-двойника поможет вам:

- выстраивать личный бренд;
- автоматизировать процесс написания текстов;
- генерировать новые творческие идеи.

Чем ассистент отличается от LLM-двойника

Цифровой ассистент и цифровой двойник — это разные понятия, сочетание которых образует по-настоящему мощное решение.

- Ассистент — это ИИ-инструмент, поддерживающий пользователя при выполнении задач, связанных с программированием, написанием текстов или созданием контента.
- Двойник — это цифровое представление реального объекта, часто использующее методы ИИ для преодоления разрыва между физическим и цифровым мирами. Например, LLM-двойник — это языковая модель, способная воспроизводить ваш голос, стиль письма и особенности личности.

С учетом этих определений можно сказать, что LLM-двойник выполняет роль ассистента, способного генерировать тексты в вашем стиле.

Важно подчеркнуть, что создание LLM-двойника вполне этично. Мы тонко настраиваем модель исключительно на собственных цифровых данных, не используя чужую информацию и не пытаясь имитировать чью-либо личность. Наша цель — создать персонализированную модель, подражающую нашему стилю письма. Таким образом, у каждого человека будет свой LLM-двойник с ограниченным доступом.

Конечно, существует множество вопросов, связанных с безопасностью, но мы не будем углубляться в них, так как эта тема заслуживает отдельной книги.

Почему не стоит использовать ChatGPT (и подобные чат-боты)

В этом разделе речь пойдет об использовании ChatGPT (или другого подобного чат-бота) исключительно в контексте генерации персонализированного контента.

Ответ на этот вопрос уже был дан: ChatGPT не *адаптирован* под ваш стиль письма и манеру изложения. Он генерирует текст, который кажется слишком общим, многословным и лишенным выразительности. Сохранение индивидуального стиля имеет решающее значение для долгосрочного успеха при формировании собственного бренда. Поэтому наличие ChatGPT или Gemini в вашем проекте не поспособствует достижению оптимального результата. Даже если вы готовы публиковать неперсонализированный контент, бездумное использование ChatGPT может привести к возникновению следующих проблем.

- *Искажение фактов из-за галлюцинаций.* Ручная проверка вывода или верификация текста с помощью сторонних инструментов утомительна и малоэффективна.
- *Монотонная ручная генерация промптов.* Самостоятельно формулировать запросы и внедрять в них внешнюю информацию тоже довольно утомительно, а получаемые ответы невозможно воспроизводить в точности, так как вы не контролируете содержимое промпта и структуру подаваемых данных. Частично эту проблему можно решить с помощью API и инструментов вроде LangChain[1], но это требует навыков программирования.

Наш опыт подсказывает, что если вы хотите создавать действительно качественный контент с реальной ценностью, то на проверку и правку сгенерированного текста вы потратите больше времени, чем на его написание с нуля.

Суть LLM-двойника определяется тем:

- какие данные мы собираем;
- как мы их обрабатываем;
- как подаем их на вход LLM;
- как выстраиваем цепочки промптов для получения нужного результата;
- как оцениваем сгенерированный контент.

---

[1] LangChain — это фреймворк для разработки приложений на базе LLM, предоставляющий инструменты для интеграции моделей с внешними источниками данных, API и последовательными пайплайнами обработки. — *Примеч. науч. ред.*

Разумеется, сама LLM тоже имеет значение. Однако стоит подчеркнуть, что использование веб-интерфейса ChatGPT крайне неудобно в плане управления данными, интеграции различных источников и оценки результатов. Решением этой проблемы является создание системы на базе LLM, которая инкапсулирует и автоматизирует все перечисленные ниже этапы работы (каждый раз выполнять их вручную — неэффективно и нецелесообразно в долгосрочной перспективе):

- сбор данных;
- предварительную обработку данных;
- хранение, версионирование и извлечение данных;
- тонкую настройку (дообучение) модели;
- генерацию, дополненную поиском (RAG);
- оценку сгенерированного контента.

При этом мы не утверждаем, что не стоит пользоваться API от OpenAI. Мы просто хотим подчеркнуть, что представленная в книге архитектура не будет зависеть от конкретной модели. Это означает, что, если модель можно использовать программно и она предоставляет интерфейс для тонкой настройки, значит, ее можно интегрировать в систему LLM-двойника. Ключ к успеху при создании ML-продуктов — быть ориентированными на данные и делать архитектуру независимой от конкретной модели. Тогда вы сможете легко экспериментировать с различными моделями на собственных данных.

Планирование MVP для LLM-двойника

Теперь, когда мы разобрались, что такое LLM-двойник и зачем его создавать, необходимо определиться с его функционалом. В книге мы сосредоточимся на первом этапе жизненного цикла, то есть на создании *минимально жизнеспособного продукта* (Minimum Viable Product, MVP). Основная задача здесь состоит в том, чтобы согласовать наше видение с реалистичными и достижимыми бизнес-целями, учитывая имеющиеся ресурсы. По мере роста инженеру необходимо проходить через этот этап, чтобы устранить разрыв между потребностями бизнеса и его техническими возможностями.

Что такое MVP

MVP — это версия продукта с минимальным набором функций, достаточным для привлечения первых пользователей и проверки жизнеспособности идеи на раннем этапе разработки. Обычно цель создания MVP — получить обратную связь от потенциальных потребителей с минимальными затратами.

Преимуществами стратегии разработки MVP являются:

- *быстрый выход на рынок* — максимально быстрый запуск продукта и привлечение первых пользователей;
- *проверка идеи* — тестирование продукта с участием реальных пользователей до начала его полномасштабной разработки;
- *исследование рынка* — получение данных о потребностях целевой аудитории;
- *снижение рисков* — минимизация затрат времени и ресурсов на разработку потенциально неуспешного продукта.

На этом этапе крайне важно создать именно *жизнеспособный* продукт, который предоставляет полноценный пользовательский опыт даже при реализации ограниченной функциональности. Он должен быть настолько работоспособным и удобным, чтобы людям захотелось продолжать им пользоваться и наблюдать за его развитием.

Определение MVP для LLM-двойника

Давайте проведем мысленный эксперимент и представим, что мы создаем не учебный проект из книги, а реальный продукт. Какие ресурсы есть в нашем распоряжении? Увы, их совсем немного:

- команда из трех человек — двух ML-инженеров и одного исследователя;
- наши ноутбуки;
- личные средства для оплаты вычислительных ресурсов (например, для обучения моделей);
- энтузиазм.

Как видите, наши ресурсы весьма ограниченны. И хотя это гипотетическая ситуация, она отражает реальность, с которой сталкивается большинство стартапов на начальном этапе своей деятельности. Поэтому мы должны стратегически подойти к определению MVP для LLM-двойника и к выбору функций, которые нам следует реализовать. Наша цель — максимально увеличить ценность продукта относительно вложенных усилий и ресурсов.

Чтобы упростить задачу, мы сосредоточимся на реализации таких функций LLM-двойника, как:

- сбор данных из пользовательских профилей в различных социальных сетях и на платформах вроде Substack и GitHub;
- тонкая настройка LLM с открытым исходным кодом на основе этих данных;

- заполнение векторной базы данных (БД) цифровыми материалами для их последующего использования в рамках RAG;
- создание публикаций в соцсетях с использованием:
  - пользовательских промптов;
  - механизма RAG (для повторного использования старого контента);
  - новых публикаций, статей и документов (в качестве дополнительного контекста для генерации ответов);
- наличие простого веб-интерфейса, позволяющего пользователю:
  - указать ссылки на свои соцсети и запустить процесс сбора данных;
  - отправлять промпты или ссылки на внешние ресурсы.

Это и будет MVP для нашего LLM-двойника. И пусть его функциональность ограничена, главное — чтобы система была экономически эффективной, масштабируемой и модульной.

Даже если мы сосредоточимся лишь на базовых функциях LLM-двойника, определенных в этом разделе, продукт будет разрабатываться с учетом современных исследований в области LLM, а также лучших практик программной инженерии и MLOps, поскольку наша цель заключается в демонстрации процесса проектирования экономически эффективного и масштабируемого приложения на базе LLM.

До этого момента мы рассматривали LLM-двойника с позиции пользователя и бизнеса. Теперь пришло время взглянуть на него с инженерной точки зрения и наметить план его разработки. Далее в книге мы сосредоточимся на технических аспектах реализации LLM-двойника.

Построение ML-систем с использованием пайплайнов признаков, обучения и инференса

Прежде чем переходить к обсуждению архитектуры LLM-двойника, необходимо рассмотреть лежащий в ее основе паттерн, называемый *FTI-архитектурой* (feature/training/inference — «признаки/обучение/инференс»). В этом разделе представлен общий обзор FTI-пайплайнов и методов их использования при построении ML-приложений.

Давайте посмотрим, как можно использовать FTI-пайплайны при разработке архитектуры LLM-двойника.

Проблема построения ML-систем

Создание ML-систем, готовых к развертыванию в продакшене, не ограничивается обучением модели. С технической точки зрения само обучение является одним из простейших этапов в большинстве случаев. Главная сложность при этом заключается в выборе правильной архитектуры и гиперпараметров[1] — но это уже исследовательская, а не инженерная проблема.

Мы же сосредоточимся на проектировании архитектуры, готовой к реальному использованию. Создание высокоточной модели, безусловно, важно, но ее обучения на статическом наборе данных недостаточно. Перед развертыванием системы нам необходимо подумать о том, как:

- загружать, очищать и осуществлять валидацию новых данных;
- разграничивать процессы обучения и инференса;
- вычислять признаки[2] и сохранять их в соответствующей среде;
- развертывать и обслуживать модель с минимальными затратами;
- версионировать, отслеживать и распространять наборы данных и модели;
- осуществлять мониторинг инфраструктуры и поведения моделей;
- развертывать модель в масштабируемой инфраструктуре;
- автоматизировать процессы обучения и развертывания.

Решение таких вопросов обычно возлагается на ML- или MLOps-инженера, в то время как за обучение модели чаще всего отвечает команда исследователей или специалистов по анализу данных.

На рис. 1.1 показаны основные компоненты, которые, по мнению команды Google Cloud, необходимы для создания полноценной ML- и MLOps-системы.

Помимо ML-кода (программного кода системы машинного обучения), данная схема должна включать в себя ряд других взаимосвязанных этапов, таких как конфигурации, автоматизация, сбор и проверка данных, тестирование и отладка, управление ресурсами, процессами и метаданными, анализ и мониторинг модели, а также инфраструктура ее обслуживания. Таким образом, при развертывании ML-модели в продакшене необходимо учитывать множество факторов.

---

[1] Гиперпараметры — это внешние настройки алгоритма машинного обучения, которые задаются до начала обучения и не изменяются самой моделью в процессе. Они управляют тем, как именно модель обучается, определяя ее структуру, скорость и качество работы. — *Примеч. науч. ред.*

[2] Признаки — это единицы информации, используемые для поиска подходящего контекста и генерации на его основе ответа от LLM. Например, для текста это может быть его абзац. — *Примеч. науч. ред.*

Рис. 1.1. Типовые компоненты ML-системы

В связи с этим возникает важный вопрос: как объединить все эти компоненты в согласованную систему? Чтобы на него ответить, нужно создать четкий шаблон для проектирования архитектуры ML-систем.

Аналогичный подход уже давно применяется в сфере разработки программного обеспечения. Большинство приложений можно разложить на три уровня: базу данных, бизнес-логику и пользовательский интерфейс. Каждый из этих уровней может быть сколь угодно сложным, но в обобщенном виде архитектура ПО сводится к этим трем компонентам.

Существует ли нечто подобное для ML-приложений? Прежде чем ответить на данный вопрос, рассмотрим существующие решения и выясним, почему они не подходят для построения масштабируемых ML-систем.

Проблема традиционных решений

На рис. 1.2 представлена типовая архитектура, используемая во многих ML-приложениях. Она монолитна, поскольку объединяет этапы создания признаков, обучения модели и инференса в один компонент. Такой подход позволяет быстро устранить одну из главных проблем машинного обучения — расхождение

между обучением и инференсом. Эта проблема возникает в тех случаях, когда признаки, подаваемые модели на стадии инференса, вычисляются иначе, чем на этапе ее обучения.

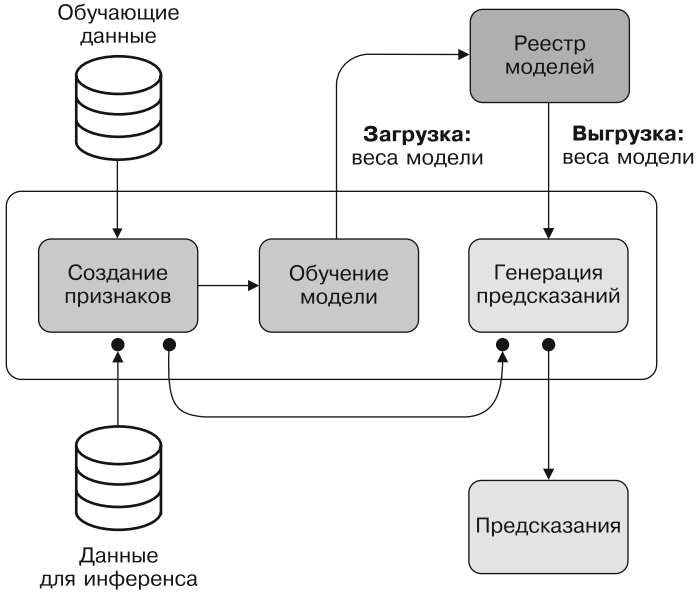

Рис. 1.2. Архитектура монолитного пакетного пайплайна

В рамках данной архитектуры признаки вычисляются одним и тем же кодом, что автоматически устраняет упомянутое расхождение. Такая схема вполне подходит для работы с небольшими объемами данных. Пайплайн запускается по расписанию в пакетном режиме, а предсказания используются сторонним приложением, например дашбордом.

Однако построение монолитной пакетной системы порождает ряд других проблем:

- признаки нельзя повторно использовать (ни внутри системы, ни в других проектах)[1];

[1] Например, для одной задачи определения вероятности покупки нужно через запрос рассчитать количество заходов на сайт покупателя. Через какое-то время появляется задача предсказать вероятность мошеннической покупки, на которую это значение тоже влияет, и в таком случае его придется заново рассчитывать из исходных данных. — *Примеч. науч. ред.*

- при росте объема данных приходится полностью перерабатывать код, чтобы обеспечить поддержку фреймворков PySpark или Ray;
- модуль предсказаний сложно переписать на более эффективном языке, например C++, Java или Rust;
- трудно разделить работу между командами, отвечающими за признаки, обучение и предсказания;
- невозможно переключиться на потоковую обработку для реализации обучения в реальном времени.

На рис. 1.3 показан аналогичный сценарий, но уже для системы с инференсом в режиме реального времени. Здесь к перечисленным проблемам добавляется еще одна: для генерации предсказаний все состояние приходится передавать в клиентском запросе — только так можно вычислить признаки и передать их модели.

Рис. 1.3. Архитектура системы с инференсом в режиме реального времени без сохранения состояния

Рассмотрим пример с рекомендацией фильмов для пользователей. Вместо того чтобы просто передать идентификатор пользователя, нам приходится отправлять полное его состояние, включая имя, возраст, пол, историю просмотров и т. п. Такой подход чреват ошибками, так как клиент должен знать, как получить данные сведения, и при этом он оказывается жестко связан с модельным сервисом.

Аналогичная ситуация возникает при реализации LLM с поддержкой RAG. Документы, добавляемые к запросу в качестве контекста, представляют собой внешнее состояние. Если бы мы не сохраняли эти записи в векторной базе данных, их пришлось бы передавать вместе с пользовательским запросом. Это потребовало бы от клиента умения запрашивать и извлекать документы, что весьма непрактично. Когда клиентское приложение вынуждено само вычислять или извлекать признаки, это считается антипаттерном. Если вы пока не знакомы с принципами работы механизма RAG, не беспокойтесь, мы подробно разберем их в главах 8 и 9.

Таким образом, наша главная задача состоит в том, чтобы понять, как получить доступ к признакам, необходимым для генерации предсказаний, не передавая их с клиентским запросом. Вернемся к примеру с рекомендациями фильмов: как предсказать предпочтения пользователя, опираясь исключительно на его идентификатор? Запомните эти вопросы, мы еще к ним вернемся.

На другом конце спектра находится архитектура от Google Cloud — готовое к использованию решение, представленное на рис. 1.4. Однако, несмотря на свою работоспособность, оно является довольно сложным и неинтуитивным. Без опыта в развертывании и сопровождении ML-моделей в продакшене разобраться в этой архитектуре будет непросто. Кроме того, не вполне ясно, как запустить такую систему в упрощенном виде и постепенно масштабировать ее.

Диаграмма на рис. 1.4 взята из документа, опубликованного компанией Google и распространяемого на условиях лицензии Creative Commons Attribution 4.0.

И здесь на сцену выходят архитектуры ML-систем на основе FTI-пайплайнов, которые позволяют решить указанные выше фундаментальные проблемы.

Решение: ML-пайплайны для построения ML-систем

Предлагаемое решение основано на построении четкой и простой ментальной модели, которой может следовать любая команда или инженер при создании признаков, обучении модели и генерации предсказаний. Эта структура, охватывающая три ключевых этапа работы любой ML-системы, известна как FTI-пайплайн. В чем же его отличие от ранее представленных подходов?

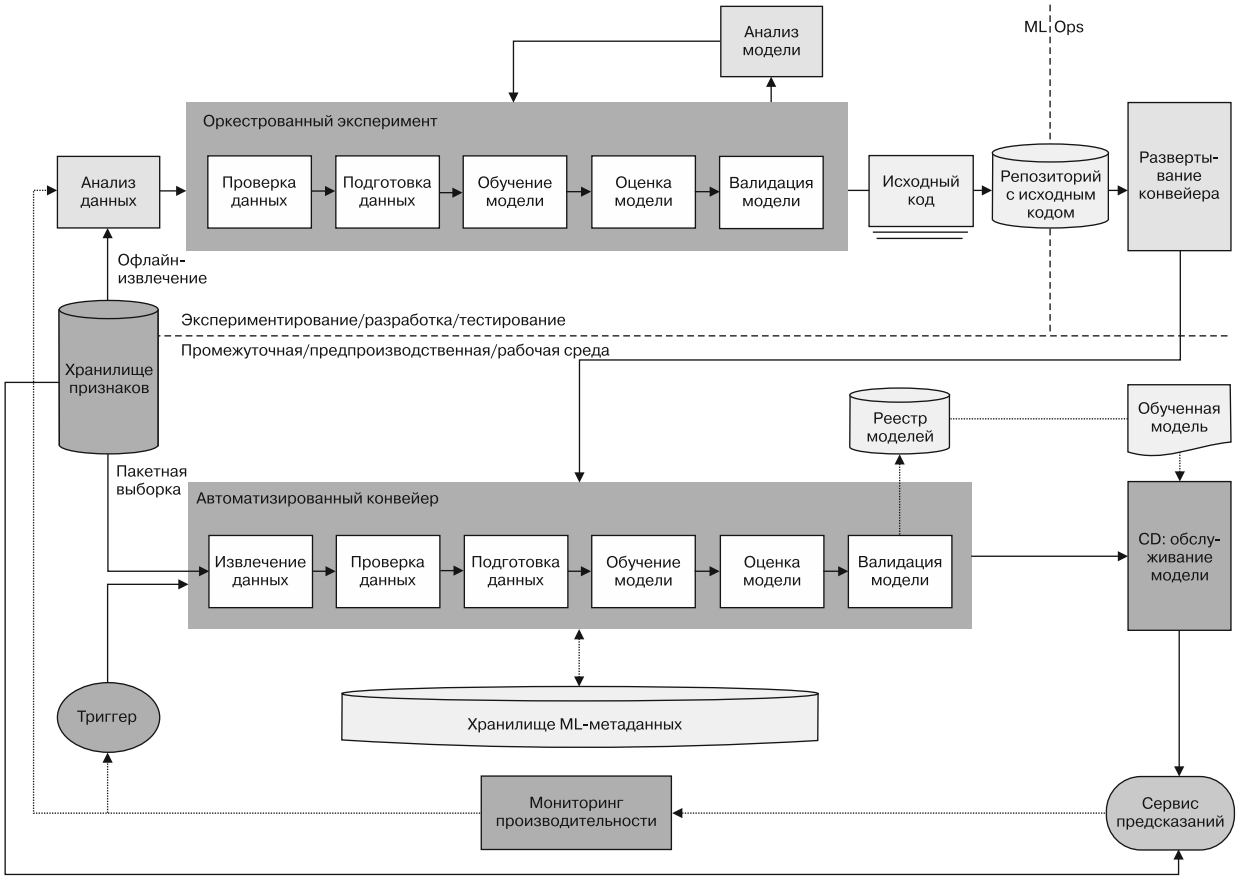

Рис. 1.4. Автоматизация ML-пайплайна для непрерывного обучения (источник: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

Данный паттерн предполагает, что любую ML-систему можно свести к трем базовым пайплайнам: признаков, обучения и инференса (по аналогии с такими уровнями традиционных систем, как база данных, бизнес-логика и пользовательский интерфейс). Этот подход позволяет четко определить границы и интерфейсы каждого пайплайна, а также понять, как они взаимодействуют друг с другом. В итоге вместо двадцати взаимосвязанных компонентов, представленных на рис. 1.4, у нас остается всего три, что существенно упрощает работу.

На рис. 1.5 изображена архитектура на основе FTI-пайплайнов. В следующих разделах мы подробно рассмотрим задачи и интерфейсы каждого из них.

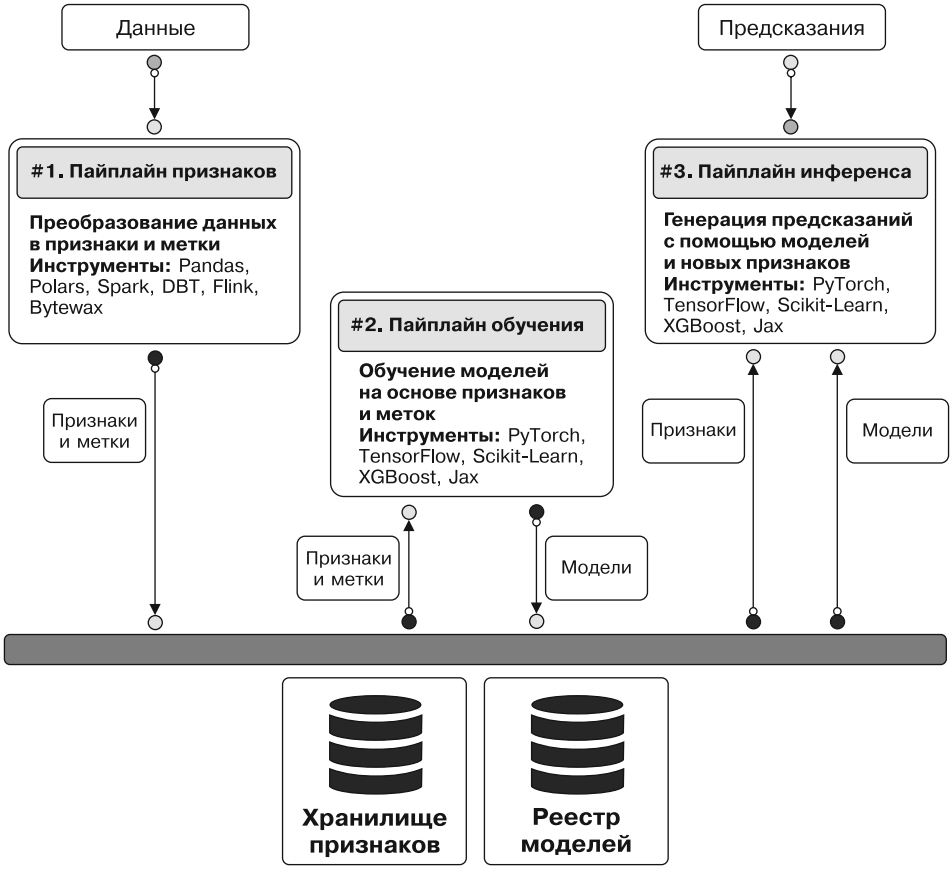

Рис. 1.5. Архитектура на основе FTI-пайплайнов

Прежде чем перейти к деталям, важно отметить, что каждый из этих пайплайнов представляет собой отдельный компонент, который может выполняться в отдельном процессе или даже на другом аппаратном обеспечении. Поэтому

любой такой пайплайн может быть реализован с использованием разных технологий, поддерживаться отдельной командой и масштабироваться независимо от остальных. Ключевая идея состоит в том, что архитектура остается максимально гибкой, что позволяет адаптировать ее под нужды вашей команды. Таким образом, данный паттерн служит мысленной моделью для структурирования ML-системы.

Пайплайн признаков

Пайплайн признаков принимает на вход сырые данные, обрабатывает их и формирует признаки и метки, необходимые модели для обучения или инференса. Однако эти признаки и метки не передаются непосредственно в модель, а отправляются в хранилище признаков, задача которого заключается в их хранении, версионировании, отслеживании и предоставлении. Благодаря этому у нас всегда есть сохраненное состояние признаков, и мы можем легко передавать их в пайплайны обучения и инференса.

Версионирование данных обеспечивает совпадение признаков на всех этапах работы модели и тем самым устраняет проблему расхождения между обучением и инференсом.

Пайплайн обучения

Пайплайн обучения получает на вход признаки и метки из хранилища признаков и дообучает одну или несколько моделей, которые сохраняются в реестре моделей (model registry). Этот реестр выполняет ту же функцию, что и хранилище признаков, но применительно к моделям. Таким образом, реестр моделей отвечает за их хранение, версионирование, отслеживание и передачу в пайплайн инференса.

Большинство современных реестров моделей поддерживают хранилище метаданных, где фиксируются основные сведения об обучении модели, в первую очередь использованные в ходе этого процесса признаки, метки и их версии, благодаря чему мы всегда знаем, на каких данных была обучена каждая конкретная модель.

Пайплайн инференса

Пайплайн инференса принимает на вход признаки и метки из хранилища признаков и обученную модель из реестра моделей. Этих двух компонентов достаточно для того, чтобы генерировать ответы — как в пакетном режиме, так и в режиме реального времени.

Поскольку данный паттерн достаточно универсален, вы сами определяете, как обрабатывать результаты предсказаний. В пакетной системе они, как правило, записываются в базу данных. В системе, работающей в режиме реального

времени, они возвращаются клиенту, инициировавшему запрос. Кроме того, все признаки, метки и модели версионируются, что упрощает обновление и откат развернутых решений. Например, мы знаем, что модель v1 использует признаки F1, F2 и F3, а модель v2 — F2, F3 и F4, благодаря чему мы можем быстро переключаться между разными конфигурациями модели и признаков.

Преимущества FTI-архитектуры

В завершение еще раз подчеркнем ключевые особенности интерфейсов FTI-пайплайнов:

- пайплайн признаков принимает на вход данные и сохраняет признаки и метки в хранилище признаков;
- пайплайн обучения извлекает признаки и метки из хранилища признаков и сохраняет обученную модель в реестре моделей;
- пайплайн инференса использует признаки из хранилища и модель из реестра для генерации предсказаний.

Какой бы сложной ни была ваша ML-система, эти интерфейсы всегда останутся неизменными.

Теперь, когда структура FTI-паттерна стала более ясной, сформулируем его основные преимущества:

- простота и наглядность — всего три ключевых компонента, которые легко понять и реализовать;
- гибкость в выборе технологий — каждый компонент может быть реализован на своей технологической платформе, что позволяет адаптировать систему под конкретные задачи, например, для обработки больших объемов данных или их потоковой обработки;
- разделение ответственности — благодаря четким границам интерфейсов каждый компонент может разрабатываться отдельной командой, что упрощает масштабирование и управление проектом;
- независимое развертывание — каждый компонент можно развертывать, масштабировать и мониторить независимо от остальных.

Важно понимать, что реальная система иногда включает в себя больше трех компонентов. Например, в состав пайплайна признаков могут входить отдельные сервисы для вычисления признаков и валидации данных, а в пайплайн обучения — модули обучения и оценки модели.

FTI-пайплайны следует воспринимать как логические уровни. Каждый из них может быть достаточно сложным и включать множество сервисов. Однако принципиально важно придерживаться единого интерфейса взаимодействия между

пайплайнами через хранилище признаков и реестр моделей. Это позволяет каждому элементу FTI-архитектуры развиваться независимо, без необходимости знать внутреннее устройство других компонентов и без риска нарушить работу системы при внесении изменений.

Если вы хотите глубже разобраться в FTI-архитектуре, рекомендуем ознакомиться со статьей From MLOps to ML Systems with Feature/Training/Inference Pipelines Джима Даулинга, главного исполнительного директора и сооснователя компании Hopsworks. Именно она вдохновила авторов книги на написание данного раздела (https://www.hopsworks.ai/post/mlops-to-ml-systems-with-fti-pipelines).

Теперь, когда мы разобрались с архитектурой FTI-пайплайнов, посмотрим, как ее можно применить для создания LLM-двойника.

Проектирование архитектуры системы LLM-двойника

В этом разделе мы перечислим конкретные технические требования к LLM-двойнику и разберем, как их можно реализовать с помощью FTI-архитектуры. Однако, прежде чем переходить к описанию пайплайнов, важно подчеркнуть, что на данном этапе мы не будем фокусироваться на инструментах или технологическом стеке. Сейчас наша цель — определить архитектуру системы на концептуальном уровне, не привязываясь к языкам программирования, фреймворкам, платформам и инфраструктуре. Здесь мы сосредоточимся на задачах каждого компонента, их интерфейсах и взаимосвязях, а в последующих главах подробно рассмотрим вопросы реализации и используемые технологии.

Перечень технических требований к архитектуре LLM-двойника

До сих пор мы говорили о том, какие функции должен поддерживать LLM-двойник с точки зрения пользователя. Теперь сформулируем требования к системе машинного обучения с сугубо технической позиции.

В рамках работы с данными мы должны иметь возможность делать следующее:

- автоматически и по расписанию собирать данные из соцсетей;
- стандартизировать собранные данные и сохранять их в хранилище;
- очищать исходные данные;