12+

Роман Лизякин

ЭРА ОРКЕСТРАТОРА

КАК ВЫЖИТЬ СИСТЕМНОМУ АНАЛИТИКУ В МИРЕ ПОБЕДИВШЕГО ИИ

ИИ ДЛЯ СИСТЕМНОГО АНАЛИТИКА:
ОТ ТРЕБОВАНИЙ ДО ГОСТОВ

Роман Лизякин

Эра оркестратора. Как выжить системному аналитику в мире победившего ИИ. ИИ для системного аналитика: от требований до ГОСТов

https://litres.ru/74429284
ISBN 9785007118200

Аннотация

Хватит перекладывать карточки в Jira и писать тонны бесполезного текста в Confluence, который никто не читает. В 2026 году старый системный анализ мертв. Если вся твоя ценность сводится к рисованию банальных BPMN-схем и ручной копипасте требований — ИИ заменит тебя уже к следующему релизу.

Содержание

GraphQL) по текстовому описанию

Конец ознакомительного фрагмента.

Эра оркестратора. Как выжить системному аналитику в мире победившего ИИ
ИИ для системного аналитика: от требований до ГОСТов

Роман Сергеевич Лизякин

Редактор Елена Некрасова
Корректор Валентина Корионова

ISBN 978-5-0071-1820-0
Создано в интеллектуальной издательской системе Ridero

Эра оркестратора. Как выжить системному аналитику в мире победившего ИИ

Нейросети вот-вот заменят всех ИТ-специалистов, а системных аналитиков уволят первыми? Опытный ментор и Senior-аналитик разрушает этот миф и предлагает жёсткий, честный и хардкорный план выживания в новой реальности.

Эта книга — не очередной сборник банальных советов вроде «как поболтать с ChatGPT», а руководство, в котором детально разобран процесс сквозной интеграции больших языковых моделей. Что позволит перейти от роли «писателя текстов по шаблону» к роли **оркестратора интеллектуальных систем**. Автор буквально на пальцах объясняет, как запрячь большие языковые модели (LLM) на всех этапах проектирования: от расшифровки хаотичных созвонов с бизнесом до автоматической генерации Swagger-контрактов, отрисовки BPMN-схем через код и создания ТЗ по ГОСТ 34 за несколько минут.

Вы узнаете о том, как выявлять скрытые архитектурные уязвимости и галлюцинации ИИ, настраивать автоматический нормоконтроль требований и разворачивать локальных ИИ-агентов (On-Premise RAG) на базе Python без интернета, сохраняя коммерческую тайну компании.

Главный манифест книги: ИИ заменит не системных аналитиков, а тех, кто отказывается его использовать. Включайте хард-скиллы, забирайте готовые паттерны и промпты высокой точности, чтобы стать специалистом, которого невозможно уволить.

Введение

Давай разберёмся, куда катится наша профессия

Привет! Раз ты открыл эту книгу, значит, уже работаешь системным аналитиком либо очень хочешь им стать, но в ужасе смотришь на новости о том, как «нейросети вот-вот заменят всех айтишников». Предлагаю снизить этот градус паники. Запомни простую вещь (эту напоминалку можно даже повесить над рабочим столом!): **ИИ заменит не системных аналитиков, а только тех, кто отказывается его использовать.**

За последние 12 лет в ИТ-индустрии я прошёл путь от джуниора, перерисовывающего стрелочки в Enterprise Architect, до Senior-аналитика и ментора. Я видел, как выгорают команды из-за бессмысленного копипаста требований и как умирают проекты из-за одной неверно выбранной переменной в ТЗ. Когда появились большие языковые модели, я понял: правила игры изменились навсегда.

То, как проектировали системы десять или даже пять лет назад, сегодня превращается в легаси. Но чтобы понять, куда мы движемся, предлагаю всё-таки заглянуть в прошлое.

Нам нужно понять, как менялась наша роль, чтобы осознать: Prompt Engineering — это не модная фишка, а логичный шаг эволюции.

Эволюция роли аналитика: от потрёпанного блокнота до промпт-инжиниринга

Наша профессия никогда не останавливалась в развитии. Мы постоянно адаптировались под технологии. Давай пройдёмся по эпохам, и ты увидишь, как менялся наш главный фокус и парадигма в ИТ-анализе: от стенографии к дирижированию. Я бы выделил четыре периода.

1. Эпоха «Блокнота» (1970—1990-е): аналитик-стенографист

Представь себе аналитика тех времён. Его главным оружием были блокнот, диктофон и пишущая машинка (позже — первый MS Word). Бизнес-заказчик сидел напротив и часами рассказывал о том, как он видит работу своей фабрики или бухгалтерии. Аналитик работал словно оживший диктофон: он судорожно записывал всё подряд, а потом неделями пытался превратить этот хаос в монолитное техническое задание.

Знаешь, в чём была главная беда? Эти ТЗ напоминали исторические романы. Их объём измерялся сотнями страниц, их согласовывали месяцами, и к моменту передачи документа в команду разработки описанные в нём бизнес-требования безнадёжно устаревали. Ценность аналитика тогда за-

ключалась исключительно в усидчивости. Это была чистая стенография.

2. Эпоха CASE-средств (2000-е): аналитик-чертёжник

Потом индустрия поняла, что тонны текста не работают, и качнулась в другую крайность. Появился язык UML (Unified Modeling Language), методологии Rational Unified Process (RUP) и тяжеловесные программы вроде Rational Rose или Enterprise Architect.

Аналитики резко превратились в чертёжников. Мы верили: если мы нарисуем идеальную диаграмму классов, распишем все Use Case до мельчайших деталей и построим ER-модель базы данных, то код сгенерируется сам нажатием одной кнопки. Но не тут-то было! Нас накрыла бюрократия. Схемы становились настолько огромными и запутанными, что малейшее изменение (например, бизнес решил поменять логику расчёта скидки) приводило к тому, что аналитик на неделю выпадал из жизни, перерисовывая стрелочки между кубиками. Мы рисовали ради рисования, теряя контакт с реальностью.

3. Эпоха Agile и Wiki-систем (2010—2020-е): аналитик-коммуникатор

Пришёл Agile, распилил монолитные ТЗ на юзер-стори (User Stories), заставил всех работать спринтами и усадил за Jira и Confluence. Системы стали микросервисными, мы начали проектировать REST API, писать спецификации в

Swagger (OpenAPI) и рисовать быстрые наброски в Miro.

И вот тут мы с тобой попали в ловушку рутины. Роль аналитика сместилась в сторону фасилитатора: мы стали «клеем» команды, человеком, который бегает между разработчиками, тестировщиками и представителем заказчика. Но платой за это стал жуткий, выжигающий мозг копипаст. Вспомни свой рабочий день: открыть Confluence, скопировать шаблон, поменять три строчки, сопоставить 50 полей, в ручном режиме перекидывая их из одной системы в другую в Excel-таблице, написать JSON-запрос… До половины времени уходило на механическую работу! Мы превратились в высокооплачиваемых операторов Ctrl + C / Ctrl + V.

4. Эпоха Prompt Engineering (наше время): аналитик-оркестратор

И вот мы здесь. Появились большие языковые модели (LLM), которые забирают у нас именно эту механическую, рутинную часть работы. Они пишут JSON-схемы за секунды, они расставляют отступы в YAML-файлах без ошибок, они переводят текст в код диаграмм.

Кем становишься ты? Ты становишься **оркестратором**, дирижёром ИИ-агентов. Твоя задача теперь — не писать текст по шаблону, а концептуально проектировать систему на верхнем уровне. Ты формулируешь инварианты — жёсткие правила и ограничения, которые ИИ не имеет права нарушить. Prompt Engineering в нашей работе — это строгая алгоритмизация собственного мышления. Если ты не пони-

маешь, как устроена REST-архитектура, как работают очереди сообщений или транзакции в базах данных, твой промпт будет размытым, а ответ ИИ — мусором. Твои хард-скиллы теперь важны как никогда, а вот навык «красиво заполнять вордовские шаблоны» можно смело отправить на свалку истории.

Почему LLM идеально подходят для работы с требованиями

Давай снимем розовые очки и разберёмся, почему нейросети так круто справляются с требованиями. Дело не в магии, а в математике и архитектуре трансформеров.

• **Мы убираем «семантический разрыв».** Бизнес говорит на языке прагматики и эмоций: *«Сделайте так, чтобы клиенты не уходили с экрана оплаты, пусть всё летает!»* Разработка требует синтаксиса и строгой семантики: типы данных, коды ответов, индексы. LLM обучались и на человеческих языках, и на коде (Java, Python, SQL, YAML). Модель выступает как идеальный транслятор смыслов. Она берёт хаотичный поток мыслей заказчика и проецирует его в строгое математическое пространство технических контрактов.

• **У ИИ нет «когнитивной слепоты».** Человеческая рабочая память ограничена. Одномоментно мы способны удерживать в голове архитектуру из трёх — пяти микросервисов. Если проект огромный, неизбежно упустим, как изменение типа поля в одном модуле сломает интеграцию в другом смежном сервисе через три колена. У современных LLM контекстное окно измеряется сотнями тысяч токенов. Ты можешь загрузить в модель спецификации всей системы, и она методом Self-Attention (механизм внимания) мгновенно подсветит тебе скрытые зависимости, на поиск которых у

команды ушли бы недели.

• **ИИ — идеальный зануда.** Requirements Engineering (инженерия требований) требует методичности. Тебе нужно написать 40 однотипных Use Case для административной панели? Человек на десятом сценарии устанет, на двадцатом начнёт ошибаться, а на тридцатом скопирует прошлый текст с ошибкой. ИИ не устаёт. Дай ему один крутой шаблон (Few-Shot Prompting), и он сгенерирует все 40 сценариев с одинаковым, практически дотошным занудством.

Границы возможностей ИИ: где он бог, а где — генератор катастроф

Запомни раз и навсегда: **LLM ничего не знает о реальном мире.** Она не понимает физический смысл того, что пишет. Она просто вычисляет вероятность появления следующего слова (токена) на основе огромной математической матрицы. Если ты будешь слепо верить всему, что выдаёт чат-бокс, тебя уволят в первый же месяц. Давай проведём чёткую границу.

Зона твоей стопроцентной уверенности

ИИ работает как бог. Можешь делегировать эти задачи с закрытыми глазами (но проверяй синтаксис):

- **Трансформация форматов:** перегнать JSON-схему в XML-структуру или написать SQL DDL-скрипт создания таблиц по текстовому списку сущностей.
- **Синтаксический нормоконтроль:** проверить, валиден ли твой OpenAPI (Swagger) файл, не пропустил ли ты двоеточие или отступ в YAML.
- **Генерация схем из текста:** превратить пошаговый сценарий в код для Mermaid или PlantUML.

Зона системных катастроф (здесь ИИ безбожно врёт)

Модель включает режим «галлюцинации уверенности». Она никогда не скажет: «Я не знаю». Она выдаст тебе чушь с апломбом ведущего архитектора.

• **Выдумывание внешних интеграций:** попроси ИИ написать контракт интеграции с «Госуслугами» или СМЭВ. Он выдаст тебе шикарный, профессиональный YAML-файл. Вот только 40% эндпоинтов и параметров в нём будут полностью выдуманы. Модель просто склеит похожие паттерны из интернета.

• **Слепота к неявному легаси:** ИИ проектирует системы для идеального вакуума. Он не знает, что у вашей старой базы данных Oracle, развёрнутой в 2012 году, есть ограничение на количество символов в имени таблицы или особый формат хранения дат, о котором никто не написал в Confluence.

• **Финансовые формулы:** никогда не проси ИИ составить формулу сложного распределения долей или начисления процентов без подключения специальных плагинов (вроде Python-интерпретатора). Модель «угадывает» символы чисел по принципу вероятности, а не считает их логически.

Совет на будущее: введи жёсткое **«архитектурное вето»**. Всё, что касается безопасности, обработки персональных данных, критических транзакций и движения реальных денег, ты проектируешь сам, закладывая в промпт жёсткие

варианты, а генерацию ИИ перепроверяешь по три раза.

Глава 1. Как запрячь ИИ на этапе сбора требований и не захлебнуться в хаосе

Транскрибация и саммаризация: как выжать суть из созвона с заказчиком

Буду честен: классические интервью с бизнесом — это ад. Ты сидишь на созвоне в Zoom, заказчик увлечённо рассказывает про свои боли, перескакивает с темы на тему, вспоминает про конкурентов. Ты пытаешься слушать, задавать правильные вопросы и одновременно судорожно записываешь тезисы в блокнот или открытый документ. В итоге либо упускаешь важную техническую деталь, либо теряешь эмоциональный контакт с собеседником и не понимаешь, где у него реальная боль, а где — второстепенная хотелка.

Мы решим эту проблему раз и навсегда. Твой новый пайплайн работы на встречах выглядит так: ты включаешь запись созвона, расслабляешься и вовлекаешься в разговор на 100%. А после встречи за дело берётся связка Whisper + LLM.

Конвейер обработки аудио:

[Запись созвона MP4/WAV] ——► (Whisper: Temp = 0) ——► [Текст без потерь] ——► (ИИ-чистка) ——► [Чистые требования]

Когда будешь прогонять аудио через Whisper (неважно, через облачный API или локальную модель), всегда выставляй параметр temperature = 0. Зачем? Если на записи будут шумы, вздохи или долгие паузы, модель с высокой температурой начнёт генерировать галлюцинации (вплоть до того, что вставит фразу «Спасибо за просмотр, ставьте лайки» в процессе обсуждения архитектуры). Нулевая температура заставляет её быть максимально занудной и переводить в текст только то, что реально было сказано.

Практический кейс: разбор хаотичного монолога

Представь, что после созвона ты получил вот такой сырой транскрипт (бизнес-заказчик формулирует фичу):

«Ну, смотри… Нам, короче, нужна такая штука, типа личного кабинета для партнёров. Сейчас они нам акты выполненных работ шлют на почту, это вообще ад, менеджеры тонут, теряют файлы. Надо, чтобы партнёр зашёл, нажал кнопку там… „Загрузить акт“, прикрепил PDF-ку. И чтобы система сразу проверяла ИНН. Если ИНН нет в нашей базе 1С, то сразу от ворот поворот, типа ошибка на экране: „Партнёр не найден, обратитесь в поддержку“. А если всё ОК, то файл сохраняется, статус меняется на „На

проверке" и нашему бухгалтеру летит уведомление в „Телеграм", мол, иди проверяй. Да, и важно, чтобы файлик был не больше 10 мегабайт, а то они нам туда видео начнут пихать…»

Если ты отдашь этот текст разработчику, он вернёт его тебе со словами: «Напиши нормально». Давай применим промпт высокой точности, чтобы вытащить онтологическую суть процесса.

Промпт для извлечения сути встреч:

text

Роль: опытный бизнес-аналитик с системным мышлением.

Задача: проанализируй сырой транскрипт интервью с заказчиком. Очисти текст от междометий, повторов и словесного мусора.

Выведи результат строго по следующим блокам:

1. Основная бизнес-цель (какую проблему решаем?).
2. Участники процесса (акторы).
3. Главный успешный сценарий (Happy Path) по шагам.
4. Выявленные бизнес-ограничения и правила валидации.

Текст транскрипта для анализа:

[ВСТАВИТЬ ТЕКСТ ВЫШЕ]

Что вернёт тебе ИИ-ассистент?

• **Бизнес-цель:** автоматизация процесса приёма актов выполненных работ от партнёров через личный кабинет с целью снижения нагрузки на менеджеров и исключения потери документов в email-каналах.

• **Акторы:** партнёр, система (личный кабинет), интеграционный модуль 1С, бухгалтер.

• **Happy Path (успешный сценарий):**

1. Партнёр инициирует загрузку акта в личном кабинете.

2. Партнёр прикрепляет файл в формате PDF и отправляет запрос.

3. Система выполняет валидацию ИНН контрагента в базе данных 1С.

4. При успешном нахождении контрагента система сохраняет документ, присваивает ему статус «На проверке» и отправляет триггерное уведомление бухгалтеру в Telegram.

• **Ограничения и валидации:**

- Формат файла: строго PDF.
- Максимальный размер файла: 10 МБ.
- Проверка ИНН: обязательное присутствие контрагента в учётной системе 1С. При отсутствии — вывод ошибки «Партнёр не найден, обратитесь в поддержку».

Ловушка «Смена контекста», о которой ты должен знать. Бизнес-заказчики часто передумывают прямо на ходу. На десятой минуте созвона он говорит: «Мы работаем

только по предоплате», а на пятидесятой минуте вспоминает: «А, ну, для крупных клиентов у нас есть постоплата в течение 30 дней». Если ты просто скормишь весь лог базовому промпту саммаризации, ИИ может выдать оба требования как равнозначные и запутать команду.

Как это лечить? Используй двухэтапный промптинг. После получения первого саммари дай модели второй запрос: *«Проанализируй этот транскрипт ещё раз. Найди места, где заказчик сначала утверждал одно бизнес-правило, а затем скорректировал его или высказал противоречие. Выведи эти моменты отдельно»*. Это спасёт твою архитектуру от скрытых мин.

Разделение требований на функциональные (FR) и нефункциональные (NFR) по стандарту ISO 29148

Бизнес никогда не придёт к тебе со словами: *«Коллеги, нам необходим показатель доступности нашей распределённой системы на уровне трёх девяток и p99 latency не более 300 миллисекунд»*. Бизнес скажет проще: *«Приложение не должно падать в пятницу вечером, когда блогеры дадут рекламу, и экран оплаты не должен тупить, а то клиенты уйдут»*.

Твоя задача как системного аналитика — перевести этот разговор на строгий инженерный язык спецификаций. И здесь мы будем опираться на международный стандарт **ISO/IEC/IEEE 29148**. Давай заставим ИИ сделать первичную классификацию и декомпозицию.

Системный промпт комплаенса требований:

text

Роль: главный системный инженер и ИТ-архитектор, проверяющий требования по стандартам ISO/IEC/IEEE 29148 и ISO 25010.

Задача: проанализируй входящее текстовое

описание фичи. Выдели и классифицируй все требования на функциональные (FR) и нефункциональные (NFR).

Разнеси нефункциональные требования (NFR) строго по следующим архитектурным атрибутам качества:

1. Performance efficiency (производительность: время отклика, пропускная способность).
2. Reliability & Availability (надёжность и доступность: отказоустойчивость, резервирование).
3. Security (информационная безопасность: шифрование, доступ).

Переведи размытые бизнес-фразы в строгие, атомарные и верифицируемые инженерные формулировки.

Исходный текст-запрос от бизнеса (входные данные):

«Нам нужно запустить мобильное приложение для заказа еды. Главное, чтобы пользователь мог быстро закинуть бургер в корзину и оплатить через СБП. Карты тоже должны работать, но СБП в приоритете. Экран оплаты не должен тупить, максимум две секунды на загрузку. Мы планируем рекламу у блогеров, так что система должна держать наплыв до 5000 человек одновременно в пятницу вечером. Все данные карт надо шифровать, безопасность

на первом месте, никаких утечек! И ещё приложение должно быть доступно постоянно, даже если упадёт основной сервер, должна подниматься резервная копия в другом дата-центре».

Что выдаст тебе ИИ-линтер? Это готовый блок для ТЗ

1. Функциональные требования (Functional Requirements, FR)

• **FR-1 (управление корзиной):** система должна обеспечивать пользователю возможность добавления и хранения товарных позиций в корзине в интерфейсе мобильного приложения.

• **FR-2 (интеграция платёжных шлюзов):** система должна поддерживать проведение онлайн-платежей через Систему быстрых платежей (СБП) и эквайринг банковских карт. Интерфейс экрана оплаты должен приоритизировать выбор СБП.

2. Нефункциональные требования (Non-Functional Requirements, NFR)

Производительность (Performance Efficiency)

• **NFR-PERF-1 (время отклика):** время полной загрузки и отображения интерфейса экрана оплаты не должно превышать 2,0 секунды при нормативных параметрах сетевого соединения на стороне клиента.

Масштабируемость и нагрузка (Scalability/Capacity)

• **NFR-SCAL-1 (пиковая нагрузка):** архитектура системы должна обеспечивать обработку одновременной сессии не менее чем 5000 активных пользователей (Concurrent Users) без деградации времени отклика бизнес-компонентов.

Надёжность и доступность (Availability & Reliability)

• **NFR-AVAIL-1 (отказоустойчивость кластера):** система должна функционировать в режиме 24/7/365. Требуется реализовать схему георезервирования (Active-Passive или Active-Active) с автоматическим переключением (Failover) инфраструктуры на резервный дата-центр при отказе основного сервера.

Информационная безопасность (Security)

• **NFR-SEC-1 (защита платёжных данных):** все передаваемые и хранящиеся данные банковских карт пользователей должны подвергаться строгому шифрованию. Компоненты системы должны соответствовать базовым принципам стандарта безопасности PCI DSS.

Разбор от ментора: где ИИ недотянул и как это исправить. Посмотри внимательно на то, что сгенерировала модель. Текст выглядит очень солидно и профессионально, верно? Но для реального, боевого ТЗ здесь всё ещё не хватает инженерной глубины.

Требование NFR-SCAL-1 говорит про «5000 активных пользователей». Для разработчика и нагрузочного тестиров-

щика это пустая фраза. Что эти пользователи делают? Они просто смотрят меню или одновременно нажимают кнопку «Оплатить» в 21:00 в пятницу?

Твой шаг как Senior-специалиста — применить теорию массового обслуживания. Ты должен дописать промпт или скорректировать текст руками, переведя абстрактных пользователей в конкретную интенсивность входящего потока запросов (\ (\lambda \)): *«Система должна обеспечивать пропускную способность не менее 800 RPS (запросов в секунду) на эндпоинты создания заказа и не менее 1200 RPS на эндпоинты чтения каталога при сохранении p99 latency <= 200 мс»*. Вот это уже измеримое и тестируемое требование по ISO 29148.

Генерация User Stories и Acceptance Criteria по методологии BDD

Перевод требований в формат пользовательских историй и критериев приёмки — это классический пример механической Agile-рутины. Поскольку формат User Story имеет жёсткий шаблон (*«Как [Роль], я хочу [Действие], чтобы [Ценность]»*), а критерии приёмки подчиняются строгому синтаксису BDD (*Given-When-Then*), ИИ справляется с этой задачей идеально. Он экономит тебе кучу времени, избавляя от пропущенных логических веток и опечаток.

Давай разберём фичу, которую Product Owner набросал тебе в Telegram: «Надо сделать сброс пароля для покупателей через SMS. Пусть вводят свой телефон, мы им шлём код из четырёх цифр. Код должен жить минуты три, не больше, ради безопасности. Если код ввели правильно — разрешаем ввести новый пароль два раза, чтобы не ошиблись. Если код неверный — пишем ошибку. Больше трёх раз неверно ввели код — блокируем кнопку отправки на полчаса, чтобы роботы нам баланс SMS не скрутили».

Промпт для генерации Agile-артефактов:

text

Роль: Senior System Analyst в продуктовой Agile-команде.

Задача: преобразуй бизнес-требование к фиче сброса пароля через SMS в формат User Story и подробных Acceptance Criteria (критериев приёмки) в формате BDD (Given-When-Then).

Требования к оформлению:

1. User Story должна чётко подсвечивать бизнес-ценность фичи.
2. Acceptance Criteria должны покрывать: позитивный сценарий, негативный сценарий (истечение тайм-аута кода) и сценарий безопасности (блокировка при переборе).
3. Используй строгий синтаксис Given (Дано) -> When (Когда) -> Then (Тогда).

Итоговая карточка, готовая к импорту в Jira:

User Story

Как зарегистрированный покупатель,

Я хочу иметь возможность восстановить доступ к своему аккаунту с помощью одноразового SMS-кода,

Чтобы быстро войти в систему и завершить покупки, если я забыл свой текущий пароль.

Acceptance Criteria (критерии приёмки)

Сценарий 1: успешное восстановление пароля (Happy Path)

• **Given** (Дано)**:** пользователь находится на экране «Восстановление пароля» и ввёл свой валидный номер телефона, привязанный к профилю.

• **When** (Когда)**:** пользователь нажимает кнопку «Получить код» и вводит корректный 4-значный код из SMS в течение 180 секунд с момента отправки.

• **Then** (Тогда)**:** система перенаправляет пользователя на экран ввода нового пароля, содержащий поля «Новый пароль» и «Подтвердите пароль».

Сценарий 2: истечение срока действия SMS-кода (негативный)

• **Given** (Дано)**:** система сгенерировала и отправила пользователю SMS-код, но с момента отправки прошло более 180 секунд (три минуты).

• **When** (Когда)**:** пользователь вводит этот код в поле подтверждения и нажимает «Проверить».

• **Then** (Тогда)**:** система блокирует верификацию, отображает ошибку «Срок действия кода истёк» и активирует ссылку «Запросить код повторно».

Сценарий 3: превышение лимита попыток ввода (безопасность / защита от brute-force)

• **Given** (Дано)**:** пользователь ввёл неверный 4-значный код из SMS три раза подряд.

• **When** (Когда)**:** пользователь совершает четвёртую по-

пытку ввода неверного кода или пытается повторно нажать кнопку «Получить код».

• **Then** (Тогда)**:** система блокирует возможность отправки и верификации SMS-кодов для данного номера телефона на 30 минут, отображая на экране таймер обратного отсчёта и текст: «Доступ заблокирован из-за частых запросов. Попробуйте через 30 минут».

Архитектурный разбор уязвимости: где джуниор поймает факап. Посмотри на сценарий 3. ИИ написал прекрасную логику для интерфейса пользователя. Но если ты отдашь это в сыром виде, то неопытный джун-разработчик может реализовать счётчик попыток и таймер блокировки прямо в памяти мобильного приложения (на фронтенде).

К чему это приведёт? Злоумышленник запустит скрипт перебора кодов напрямую через API, обходя мобильный интерфейс, или обычный пользователь просто закроет приложение, откроет его снова, память обнулится, и он сможет подбирать код дальше. Баланс компании на SMS-шлюзе улетит в трубу за час.

Твоя задача как системного аналитика — вмешаться, заблокировать эту генерацию и жёстко дописать техническое требование к бэкенду: «На уровне бэкенда в кеше Redis необходимо реализовать два разных ключа с разным временем жизни (TTL) для разграничения бизнес-логики и безопасности:

1. Ключ самого SMS-кода (например, sms_code: phone_number): содержит сгенерированные четыре цифры. Время жизни ключа TTL = 3 минуты (180 секунд) по требованию бизнеса. По истечении этого времени ключ автоматически удаляется, делая старый код невалидным.

2. Ключ счётчика блокировки (например, sms_block: phone_number): инкрементируется при каждой неудачной попытке ввода. При достижении трёх попыток флаг блокировки фиксируется в Redis со временем жизни TTL = 30 минут. Бэкенд должен мгновенно отдавать фронтенду ошибку 423 Locked на любые запросы авторизации по этому номеру, пока ключ блокировки не удалится по тайм-ауту».

Совет на будущее: фронтенд в этой схеме должен лишь отображать состояние и включать визуальный таймер обратного отсчёта, присланный от API. Теперь система надёжно защищена от обхода интерфейса.

Глава 2. Проектирование интеграций и API с помощью ИИ

Генерация контрактов API (REST, gRPC, GraphQL) по текстовому описанию

Когда-то аналитики гордились тем, что могут без единой ошибки в отступах написать руками 500 строчек YAML-файла для Swagger. Сегодня этот навык не стоит ничего. Расстановка двоеточий, скобок и дефисов — это чистая механика, которую нужно со свистом делегировать ИИ.

Твоя ценность как аналитика теперь в другом — ты проектируешь **семантику и поведение ресурса**. Ты решаешь, какой HTTP-метод использовать (PATCH для частичного изменения или PUT для полной замены), как обеспечить идемпотентность и какие бизнес-коды ошибок возвращать команде фронтенда.

Давай посмотрим, как ИИ справляется с генерацией контрактов под три разных архитектурных стиля на основе обычного текстового ТЗ.

1. REST API (формат OpenAPI 3.0 / Swagger)

Представь задачу: курьер в мобильном приложении нажимает кнопку «Доставил заказ». Тебе нужно спроектировать метод частичного обновления статуса.

Промпт высокой точности:

text

Роль: API-архитектор и системный аналитик.

Задача: напиши валидную спецификацию OpenAPI 3.0 в формате YAML на основе текстового описания метода.

Контекст метода:

— Действие: обновление статуса заказа курьером.

— Путь: /api/v1/orders/ {orderId} /status

Конец ознакомительного фрагмента.